

Hand Simulation for Virtual Climbing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Roman Voglhuber, BSc

Matrikelnummer 01127128

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Wien, 21. Mai 2019

Roman Voglhuber

Horst Eidenberger

Hand Simulation for Virtual Climbing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Roman Voglhuber, BSc

Registration Number 01127128

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Vienna, 21st May, 2019

Roman Voglhuber

Horst Eidenberger

Erklärung zur Verfassung der Arbeit

Roman Voglhuber, BSc
Linke Wienzeile 126/34
1060 Wien

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

Wien, 21. Mai 2019

Roman Voglhuber

Acknowledgements

First, I would like to thank Professor Eidenberger for the opportunity to work on such an interesting topic and for his excellent supervision throughout this thesis. Furthermore, I want to express my deepest gratitude to my parents Brigitte und Kurt for their patience and support whenever I needed it. I also want to thank my girlfriend Ines for her encouragement and motivation over the last few years.

Kurzfassung

Virtual Reality (VR) Anwendungen ermöglichen es den Usern, in eine virtuelle Welt einzutauchen. Bei aktuellen VR-Systemen tragen die Benutzer eine VR-Brille und können anhand von Controllern mit der virtuellen Welt interagieren. Die Controller eignen sich für eine Vielzahl von Anwendungen, allerdings können sie etwa nicht verwendet werden, wenn die Benutzer mit realen Objekten interagieren sollen. Ein Beispiel für eine Anwendung dieser Art ist das VreeClimber-Projekt. Es kombiniert eine bewegliche Kletterwand mit VR Klettern. Nachdem die Hände der User während des Kletterns auch in der virtuellen Welt sichtbar sein sollten, wird ein optisches Tracking-System zum Erfassen der aktuellen Positionen der Hände verwendet.

Im Rahmen dieser Diplomarbeit wurden zwei Software-Komponenten für das VreeClimber Projekt erstellt. Zuerst wurde die Software von einem bereits entwickelten Tracking-System namens VreeTracker mit Hilfe der Computer-Vision-Bibliothek OpenCV neu umgesetzt. Während dem Erstellen dieser Software wurde mit dem Vive Tracker eine erschwingliche Hardwarelösung vorgestellt, welche ebenfalls zum Tracken der Gliedmaßen eingesetzt werden kann. In der Evaluierung wurden die Genauigkeit der beiden Tracking-Systeme durch verschiedene Tests miteinander verglichen.

Da die User während des Kletterns ihre eigenen Hände nicht sehen können, wurde im zweiten Teil dieser Arbeit ein Algorithmus entwickelt, welcher mit Hilfe der erfassten Positionen die Handbewegungen der Kletterer in der virtuellen Welt nachstellt. Die entwickelte Handsimulation zeigt vielversprechende Ergebnisse bei typischen Griffbewegungen während des Kletterns, speziell aber bei schnellen Bewegungen oder bei kleinen Klettergriffen kann es zu Abweichungen gegenüber der echten Hand kommen. Eine wichtige Voraussetzung ist die sorgfältige Kalibrierung vor dem Starten des Kletterns, da andernfalls die Positionen der echten und der virtuellen Klettergriffe voneinander abweichen können, wodurch das virtuelle Klettererlebnis gegebenenfalls verschlechtert wird.

Die Evaluierung hat gezeigt, dass der Vive Tracker bessere Ergebnisse erzielt als das vorher entwickelte VreeTracker System. Aufgrund der höheren Präzision und der einfachen Integration in das bereits verwendete VR System, macht es Sinn, in Zukunft Vive Tracker bei dem VreeClimber Projekt einzusetzen. Generell schnitt die virtuelle Handsimulation bei der Evaluierung gut ab, es wurden aber auch kleinere Schwachstellen in Bezug auf Griffbewegungen einzelner Finger festgestellt. Nach einer kurzen Analyse werden in dieser Diplomarbeit entsprechende Verbesserungsvorschläge präsentiert.

Abstract

VR applications enable users to immerse into a virtual world. In current VR systems, users wear a Head-Mounted Display (HMD) and can use handheld controllers to interact with the virtual world. The controllers are suitable for a variety of applications, but they cannot be used if users are to interact with real objects. An example for such an application is the VreeClimber project. It combines a moveable climbing wall with VR climbing. Since the hands of the users should also be visible in the virtual world during climbing, an optical tracking system is used to capture the hand positions in real time.

In the course of this thesis, two software components were created for the VreeClimber project. At first, the software of an already developed tracking system called VreeTracker was rewritten with the computer vision library OpenCV. During the development of this software, the affordable Vive Tracker was released, which can also be used to track extremities. The evaluation compares the accuracy of the two tracking systems by different tests.

Since the users are not able to see their own hands during climbing, an algorithm was created in the second part of this thesis, which uses the detected hand positions to simulate the hand movements of the climbers. The developed hand simulation shows promising results for typical grasp movements during climbing, however especially fast movements or small climbing holds can result in deviations from the real hand pose. An important requirement is an accurate calibration before climbing, otherwise the positions of the real and virtual climbing holds may differ, which reduces the climbing experience significantly.

The evaluation shows that the commercial Vive Tracker achieves better results than the previously developed VreeTracker system. Due to better precision and an easy integration into the already used VR system, it makes sense to use the Vive Tracker for the VreeClimber project in the future. In general, the virtual hand simulation performed well in the evaluation, however minor flaws in the grasp movements of individual fingers have been revealed. After a short analysis, appropriate suggestions for improvement have been presented in this thesis.

Contents

Kurzfassung	ix
Abstract	xi
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Structure of the Work	3
2 State of the Art	5
2.1 Hand Tracking for Virtual Reality	5
2.1.1 Marker Tracking	5
2.1.2 Finger Tracking	6
2.1.3 Available Tracking Solutions	8
2.2 Object Grasping	9
2.2.1 Simple Object Grasping with VR Controllers	9
2.2.2 Object Grasping with Finger Tracking	9
2.2.3 Algorithms and Methods from Robotics	11
3 Theoretical Background	13
3.1 Computer Vision	13
3.1.1 Infrared Light	13
3.1.2 Pinhole Camera	14
3.1.3 Camera Calibration and Triangulation	16
3.1.4 Image Manipulation and Blob Detection	18
3.1.5 Position Prediction with Kalman Filter	22
3.2 The Human Hand	24
3.2.1 Anatomy of the Human Hand	25
3.2.2 Joints and Range of Motion	25
3.2.3 Different Grasp Poses	26
4 System Design	29
4.1 Requirements	29
4.1.1 Climbing Scenario	29

4.1.2	VreeTracker Software	30
4.1.3	Virtual Hand Simulation	30
4.1.4	General Software Requirements	30
4.2	Marker Tracking	31
4.2.1	Tracking Setup	31
4.2.2	Hardware Architecture	33
4.2.3	Software Architecture	35
4.3	Virtual Hand Simulation	38
4.3.1	Physics Colliders	38
4.3.2	Climbing Grips	40
4.3.3	Closing of the Hand Model	41
4.3.4	Different Grasp Poses	42
5	Implementation	45
5.1	Marker Tracker	45
5.1.1	OpenCV	45
5.1.2	Class Overview	46
5.1.3	GUI	50
5.1.4	Calibration	53
5.1.5	Tracking	54
5.2	Virtual Hand Simulation	55
5.2.1	HTC Vive Setup	56
5.2.2	Unity Project Setup	56
5.2.3	Virtual Climbing Scene	57
5.2.4	Hand Model	58
5.2.5	VreeTracker Integration	60
5.2.6	HTC Vive Tracker Integration	61
5.2.7	Grasp Algorithm	62
6	Evaluation	67
6.1	Setup	67
6.1.1	Vive Tracker Evaluation	67
6.1.2	Virtual Hand Simulation Evaluation	67
6.2	VreeTracker and Vive Tracker Comparison	69
6.3	Grasp Performance	71
6.4	Discussion	73
7	Conclusion and Future Work	75
	List of Figures	77
	List of Tables	79
	Bibliography	81

Introduction

1.1 Motivation

The VR market has great potential, but it is still waiting for the big breakthrough. Hardware and software improves every year and gets more affordable. To experience VR, a cheap case for a smartphone like the Google Cardboard [13] is enough, but there are also more advanced products such as the HTC Vive [20] or Oculus Rift [45]. These products' screen resolution is better and the position of the user can be tracked with base stations. These more expensive products also support handheld controllers that allow the user to interact with the virtual world.

An important part of VR is the immersion of the user, which is the feeling of the user being physically present in the virtual environment. Two user studies by Insko [23] showed that passive haptics can increase the presence in a VR world. Therefore, the user experience can be improved if real objects are visible in the virtual world and the user can also touch them. The VreeClimber project to which this thesis makes a contribution tries to utilize this and combines real climbing with VR.

The VreeClimber is a climbing wall which can move vertically like a treadmill and uses VR to let the climber experience different virtual worlds. As Figure 1.1 shows, VreeClimber consists of many movable boards with climbing holds. During climbing the boards will move slowly in the opposite direction to allow a theoretically endless climb. Also the injury risk is very low because the climber will never be more than one meter above the ground. The climber is equipped with a HMD and can therefore explore different virtual worlds such as on planet Mars or on a famous building. All climbing holds were 3D scanned and are also added into the virtual world at the correct positions. This allows the climber to really immerse themselves into the VR world.

When climbing on the wall, it is important that the climber can see their own hands to safely grasp the climbing holds. In a previous thesis, a tracking system prototype



Figure 1.1: VreeClimber with climbing holds from both sides

was developed, which uses multiple cameras to track infrared (IR) markers. It is called VreeTracker, and it tracks the hands and feet of the climber [56]. It was also planned to use cameras to track each finger, but the performance was sufficient for this application. Therefore, the outcome of this thesis should be a software to track and simulate the hands of the climber based on the tracking data from VreeTracker.

1.2 Problem Statement

VreeTracker uses four high-resolution webcams to detect up to four IR trackers in real time. The 3D printed trackers are placed at the wrists and ankles of the climber. With a computer vision code for MATLAB [35], the position of the trackers is calculated and sent to a VR application. For further use, the software should be reimplemented with an open source computer vision library.

Another part of the VreeTracker project was a software module to detect the fingers of each hand with additional RGB cameras. The finger tracking worked, but the performance was not good enough to use in a VR application. There are commercial solutions with gloves available for finger tracking, but they are very expensive and therefore not suitable for the VreeClimber project [42, 68]. This is why the fingers of the climbers should be simulated by a software.

The aim of this thesis is to reimplement the tracking code of the VreeTracker with an open source computer vision library and create resources for Unity [61] to simulate the hand movements of the climber in VR. It is important that the virtual hands represent the real hands as closely as possible to give the climber the feeling that they are seeing their own hands [37]. The basic idea is to show a 3D hand model at the position from the VreeTracker and then calculate the distance between the hand model and the nearest

climbing hold. If the hand is close enough, it will start grasping for the climbing hold.

The climber will use different grips based on the climbing holds and the angle from where they try to reach them. The software should consider this and choose an appropriate grasp based on these factors. For example, if a climber tries to hold onto a very small climbing hold right above them, they will probably only use four fingers, while they will also use their thumb for a bigger hold on their side. Graphical glitches, where parts of the virtual hand move into the wall or climbing holds, should be avoided if possible.

The methodological approach for this thesis consists of three parts. The first part is a literature review to collect information about existing solutions for object grasping and hand animation. Furthermore, different open source computer vision libraries should be researched and the most suitable one should be selected.

After the literature review, the existing MATLAB code for the marker tracking should be reimplemented with the chosen computer vision library. The already developed prototype hardware should be reused if possible.

In the third part, the resources and algorithm for the hand simulation should be developed for Unity. During the development, a simple virtual climbing scene should be created to test the progress with the real climbing wall. After the implementation, the results should be tested and evaluated.

1.3 Structure of the Work

Chapter 2 gives an overview over relevant solutions for VR tracking and briefly describes how the different technologies work. It also includes findings from relevant research fields for the simulation of a virtual hand, such as how to grab objects with human-like hands from robotics [26]. In Chapter 3, the computer vision techniques for the marker tracking are explained. Additionally, the anatomy and joints of the human hand are described and the different grasp poses which are usually used by inexperienced climbers.

The general design of the marker tracker and the virtual hand simulation is explained in Chapter 4, which also contains images of the VreeTracker hardware and describes how a hand model can be animated in Unity. Chapter 5 contains detailed information about how the different parts of this project were implemented and how problems were solved. This chapter is split into two main sections for the marker tracker and the virtual hand simulation. Then in Chapter 6, the results of the VreeTracker are compared to a recently released commercial tracking solution for the HTC Vive. Also, the precision of the virtual hand grasps are evaluated. The conclusion of the project and possible future work is discussed in Chapter 7.

State of the Art

2.1 Hand Tracking for Virtual Reality

2.1.1 Marker Tracking

To create an immersive VR experience, multiple movements of the user should be tracked. The head motions are the most important because they determine the field of view. Hand tracking is the next big improvement for interactive applications. The HTC Vive [20] and Oculus Rift [45] both track their handheld controllers and allow the users to perform gestures or use virtual buttons. This works for many cases, but for some applications the user needs free hands to grab different objects or, as in this project, to climb. In such cases, a marker can be placed at the wrist to allow free hand movements without interference.

Active and Passive Markers

The markers can either be passive or active [36]. Figure 2.1 shows a camera with IR LEDs and a passive marker, which reflects the light back. Passive markers can only have one retro-reflective sphere, but multiple spheres can be used to uniquely identify the marker and also calculate its orientation. Passive markers are very cheap and can be attached at various spots due to their low weight and small size. An active marker is usually equipped with an IR LED, which means it needs a power source and some kind of circuit. This adds complexity and some limitations for the placement, but additional illumination is not needed for active markers. The LEDs of each active marker can blink in a different frequency to make it uniquely identifiable.

Position Tracking

An optical marker tracking system uses two or more cameras to monitor a defined space. Most of them use the IR light spectrum because it is not visible for humans and the

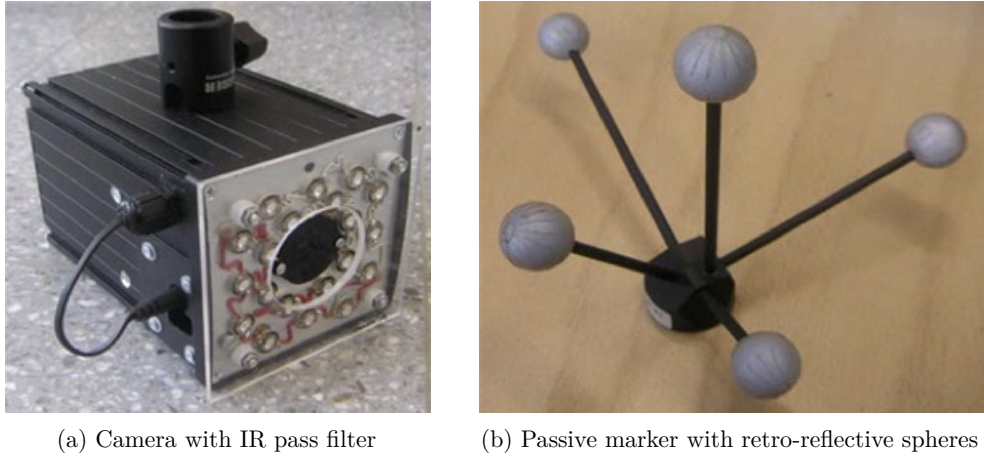


Figure 2.1: Camera with IR LEDs and a passive marker [17]

grayscale images do not need as much processing power to be analysed. Therefore, the cameras have an IR pass filter in front of the lens which blocks all other frequencies of light [36]. The calculation of the position works the same for both marker types. The cameras have to be calibrated before the tracking to know their location in the scene. The position of the marker in both camera frames is detected with computer vision. Triangulation is then used to calculate the 3D coordinates of the marker in the scene [16].

Orientation Tracking

Another important metric is the orientation of the marker. It can be determined with an optical tracking system or an inertial measurement unit (IMU), which measures changes of the orientation. The process for optical tracking consists of two steps. First, if multiple markers are used, each one has to be uniquely identified. As seen in Figure 2.1b, each marker has multiple retro-reflective spheres in different constellations. Model fitting [48] is used to identify each marker. In the second step, the visible spheres are used to calculate the orientation and translation of the marker. Alternatively, IMUs are placed on the marker to keep track of orientation changes. Then the data is analysed by a microprocessor and the current orientation is sent over a wireless connection. This solution makes sense especially for active markers because they mostly already use a microprocessor and can easily be extended with an IMU.

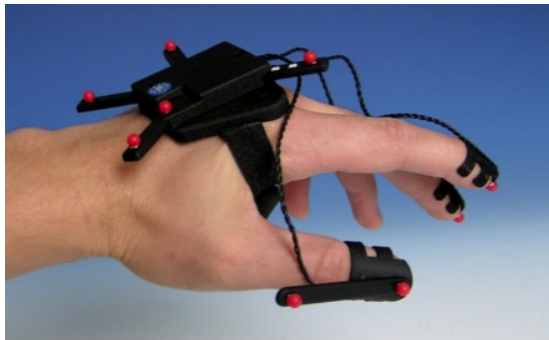
2.1.2 Finger Tracking

Finger Tracking for virtual reality is a big challenge, because of possible occlusion and the need of real-time tracking data [52]. Most of the time the hands are very close to the user's point of view and even small tracking errors or latency of finger movements can be noticed by the user. In the following section, optical and glove-based finger tracking are explained.

Optical Tracking

There are different approaches for optical finger tracking. One of them is very similar to the marker tracking described in Section 2.1.1. As seen in Figure 2.2a, active IR light sources are placed on the fingertips and on the back of the hand to track different points [17]. Then, the position and orientation of each phalanx is calculated with inverse kinematics. The active markers use time division multiplexed addressing to be uniquely identifiable. This helps to detect crossed fingers, which is usually a problem for optical finger tracking systems.

A popular commercial finger tracking product is the Leap Motion [39], which also uses IR light and consists of two cameras and three LEDs [5]. The main difference is that the Leap Motion does not use markers, which limits its detection range to about 60 cm. This is why it is usually placed on the HMD for VR applications. The light of the LEDs is reflected by the user's hands, and grayscale images are recorded by the cameras. Then, software calculates a 3D representation of the hands and extracts the position and orientation of the different fingers. It also tries to infer the position of occluded objects and uses filtering techniques to smoothen movements between frames.



(a) Finger tracking with optical IR LEDs [17]



(b) Manus VR Glove [67]

Figure 2.2: Optical and glove-based finger tracking

Another possibility is to use RGB cameras and depth data. For example, the Microsoft Kinect uses an IR projector and camera to generate a depth map [72]. Then this data is combined with the RGB images to detect poses and gestures. The Kinect itself does not support finger tracking, but the Kinect software development kit (SDK) is used by third-party libraries for finger tracking [31]. At least two Kinects are needed to avoid occlusion issues for finger tracking in a climbing scenario. Because of possible interference between multiple Kinects, Vive and VreeTracker, the Kinect is not suitable for this project [30, 54].

Glove-Based Tracking

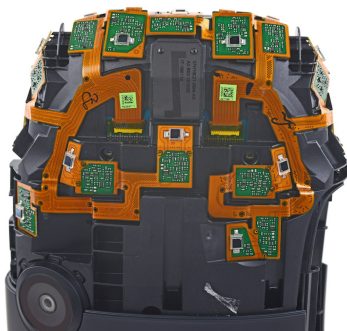
Gloves are another way to track finger movements. Figure 2.2b shows one of the currently available tracking gloves. The difficult part is to add sufficiently precise sensors on all parts of the glove and still keep it flexible enough to not bother the user. In earlier research magnetic, induction coils were used as sensors [9]. Because of the technological progress, more advanced hardware sensors became available, which were also small enough to be used on gloves [29, 27]. The main goal of those prototypes was to recognize gestures and use them as input device for computers.

The commercial gloves that are currently available use multiple IMUs with accelerometers, gyroscopes and magnetometers to detect the smallest movements of each finger [69]. Then, software uses this data to calculate the bending of each finger joint. The gloves themselves cannot track their position, which is why additional tracking systems such as VreeTracker are needed. [69, 41] The calculated data is transferred wirelessly with a latency < 5 ms and can be used in all major gaming engines. Another interesting feature is haptic feedback, which allows developers to use vibrations to indicate the user collisions with objects.

2.1.3 Available Tracking Solutions

HTC Vive

The Vive [20] is a HMD that is developed by HTC and Valve. It comes with two lighthouse base stations and two handheld controllers. The lighthouses emit defined IR pulses, which are picked up by sensors in the headset as seen in Figure 2.3a [7, 34]. The position and orientation can then be calculated by the time difference between emitting the pulse and hitting the sensor. The lighthouses should be placed over the user's head in opposing corners of the room to avoid tracking issues. The current price of the HTC Vive is 599€. During the writing of this thesis, HTC also released the Vive Tracker [19], which can be used as an alternative to the VreeTracker.



(a) Sensors on the HTC Vive [21]



(b) IR LED pattern on Oculus Rift [22]

Figure 2.3: Headset differences because of tracking method

Oculus Rift

The second big consumer VR platform is the Oculus Rift [45]. The features of the Rift and Vive are very similar; the only big difference is the tracking method. On the Rift headset and controllers are IR LEDs in a specific pattern as seen in Figure 2.3b [2, 22]. Around the scene, two or more Oculus Sensors [44] with cameras are placed, which can calculate the position and orientation of the devices based on the visible LEDs. The Rift costs 449€ and also includes two Oculus Sensors and two controllers.

Gloves

Many companies are trying to build Gloves for finger tracking, but only a few are available for purchase and are suitable for climbing. The two most promising gloves are the Manus VR [67] and Hi5 VR Glove [43]. Both offer 9 Degrees of Freedom (9DoF) and use multiple IMUs for accurate finger tracking. For positional tracking, a third-party tracker has to be used, such as VreeTracker or Vive Tracker. The developer version of the Manus VR costs 1.990€ and the Hi5 VR Glove costs 999\$. Both prices are too high for the VreeClimber project.

2.2 Object Grasping

2.2.1 Simple Object Grasping with VR Controllers

Since the release of handheld controllers for VR platforms, many games offer the possibility of interacting with the virtual world to the player. Some games use this technology for simple actions such as opening a door or pressing a button, but there are also more interactive games in which a player can pick up objects and throw them away. Usually, this object grasping is very simple. The player just has to point the controller at the object and then hold a button to grab it. Upon the release of the button, the object is also released.

While this object grasping does not represent physical grasping with hands, it can still feel intuitive after a few uses. Many games only show the controllers and not visual representations of the hands. This simplifies many things, because otherwise an appropriate pose and realistic contact points have to be calculated for each object. For the VreeClimber project, a better solution is needed, since it is of paramount importance that the user feels safe when climbing.

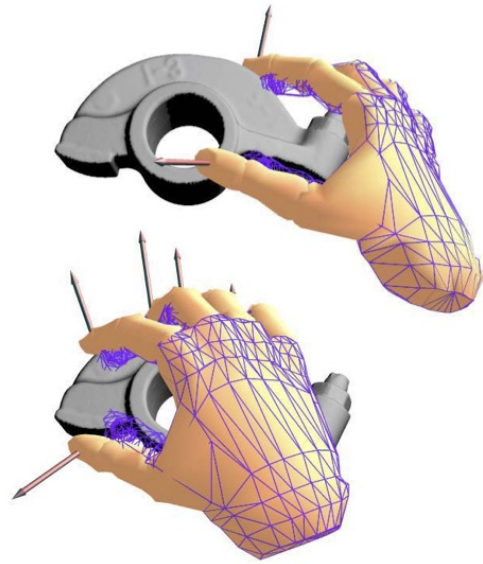
2.2.2 Object Grasping with Finger Tracking

Finger Tracking allows for the implementation of more precious and complex object grasping. An advantage is that the pose of the hand and the contact points do not need to be calculated. Other problems, such as overlapping of the hand and virtual objects are hard to solve. Figure 2.4 shows special gloves with force-feedback, which allow the user to "feel" when they touch an object [3]. In this case, the software shows a virtual hand

which does not exactly represent the tracked hand to give the user the impression of a more realistic grasp. This approach can also be implemented with any kind of collision detection, such as the physics engine of Unity. If a virtual finger collides with an object, it stops moving until the tracked hand moves away from the object. The balance between an accurate representation of the tracked hand and a realistic virtual grasp is one of the difficulties of this approach.



(a) Setup with glove and monitor



(b) Virtual hand with force feedback vectors and representation of the tracked hand (mesh)

Figure 2.4: Grasping a virtual object with a finger tracking glove [3]

Grasping Real Objects

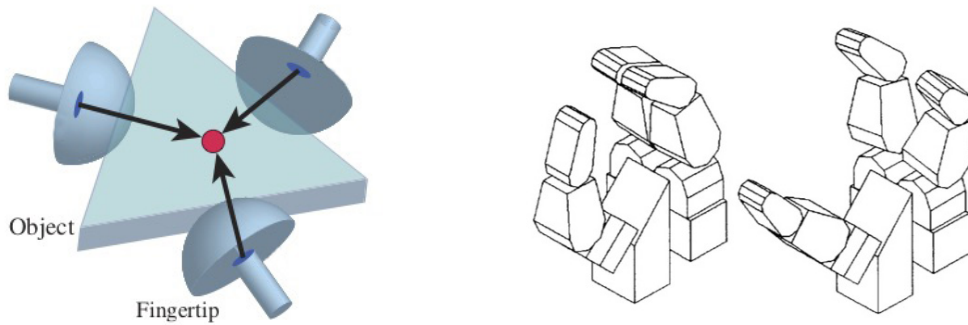
The best solution for a great immersive VR experience would be to grasp real objects which are also shown in the virtual world. That way, the user could interact with the real objects, but there are also certain limitations for this approach. For example, if an optical tracking system is used to track the objects, occlusion can be a problem during grasps. Furthermore, the tracking of the fingers and objects has to be very accurate to avoid unrealistic behaviour. If the objects are static, such as walls or tables, the problem becomes easier because the position can be determined during a calibration at the start. Then, only an accurate virtual representation of the room has to be created and the finger tracking system has to be integrated. No further calculations such as collision detection have to be made. The VreeClimber project is somewhere in-between. The climbing wall can move, but the current position of the climbing holds will be known and the climber cannot manipulate them. This means that the climbing holds are more like static objects which can be grasped.

2.2.3 Algorithms and Methods from Robotics

The goal of robotics is to use machines to automate tasks. For example, many tasks in production need to handle and manipulate objects with different forms. Because of the versatility of the human hand, a lot of research has been done to replicate some of its capabilities [26, 49, 50, 53]. For this project, we will focus on the grasping problem, because 3D models of all climbing grips will be available. It describes the difficulty to find an appropriate point where and how an object should be grasped.

Calculate Proper Contact Points

The contact points of a grasp are very important. The grasp can only be considered stable if the forces of different fingers come from different directions. A stable grip is especially important for climbing because the climber has to put a lot of weight on it. One method is to choose a focus point first and then try to place the fingers around this point [26, 49]. The lines of force from the fingers will then intersect at the focus point, as shown in Figure 2.5. The position of the point depends on the form of the object. Because of the fixed location of the climbing grips, the contact points on top of the grip should be the main priority, but a stable grip is still important to keep the balance on the climbing wall.



(a) The line of forces intersect at the focus point [26]

(b) Ideal grasps for objects with different sizes [49]

Figure 2.5: Focus point and example grasps of a robotic hand

Find a Feasible Grasp

The next step is to find a feasible grasp, which uses the calculated contact points. Each robotic hand has some limitations based on the used components and their Degrees of Freedom (DoF). If a set of contact points gets discarded, a different set has to be calculated. In a first step, all kinematically infeasible contact points are discarded [49]. For example, when the distance between two contact points is bigger than the physical limit of a robotic hand, it must be discarded. In the next step, a starting configuration for the grasp is selected. The basis of this configuration is an ideal grasp, which means

that the angle of each joint should be similar. Two ideal grasps for a small and a big object are shown in Figure 2.5. This helps to avoid limitations of single joints and also looks more natural. Finally, the configuration is checked for collisions between the components of the robot and other objects within reach of the hand. If a problem is found, adjustments of the pose or contact points are made to avoid them.

This chapter gave an overview of solutions for VR tracking and how object grasping is implemented in different fields. The next chapter describes theoretical principles of computer vision, which are the basis for hand tracking. Furthermore, it contains information about the anatomy of the human hand.



Theoretical Background

3.1 Computer Vision

This section explains how computer vision can be used to determine the location of an IR marker. After some basic information about IR light, the pinhole camera model and the camera parameters are introduced. Then, the camera calibration process and triangulation are described. Finally, some computer vision operations and the Kalman filter are explained.

3.1.1 Infrared Light

VreeTracker is an optical tracking system that uses multiple cameras to capture the IR light from the markers. IR light is just a small part of the electromagnetic spectrum [18]. As seen in Figure 3.1, the visible light for human eyes has a wavelength of 380 nm to 780 nm. The IR spectrum is from 780 nm to 1 mm, which makes it invisible to the human eye. For optical tracking systems, the so-called Near Infrared (NIR) spectrum is especially interesting because commonly used CCD and CMOS camera sensors are sensitive to wavelengths from 350 nm to about 1000 nm [18].

Light Filters

Typical consumer webcams also use CCD or CMOS sensors. Therefore, they are sensitive to visible and IR light. To increase the quality of the images, manufacturers usually add a light filter to block IR light, which means that IR light is not visible on the recorded images. A so-called bandpass filter only transmits light within a certain wavelength band [18]. Hence, if the IR filter of a webcam is replaced with one that only transmits NIR light, the camera can be used for an optical tracking system.

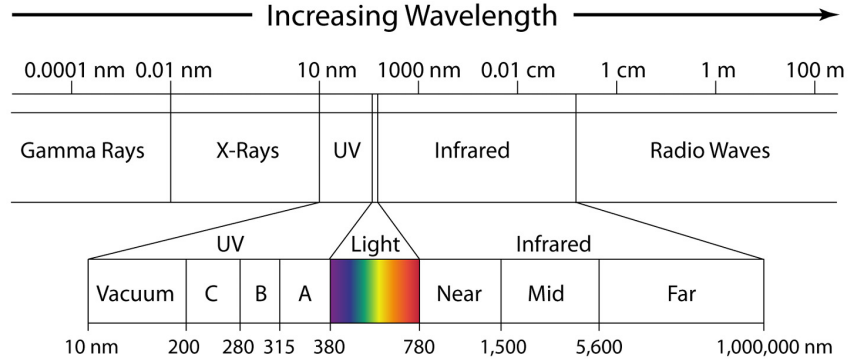


Figure 3.1: Electromagnetic spectrum [11]

3.1.2 Pinhole Camera

The basic camera model for computer vision is the pinhole camera model, which describes the geometric relation of 3D and 2D points on the image plane. As shown in Figure 3.2a, a pinhole camera is a closed box with a tiny hole on one side and a photosensitive surface on the opposite side. Light enters through the hole and projects an inverse image onto the photosensitive surface. The pinhole is called the center of projection, and the projection expressed by the pinhole model is a perspective projection [16].

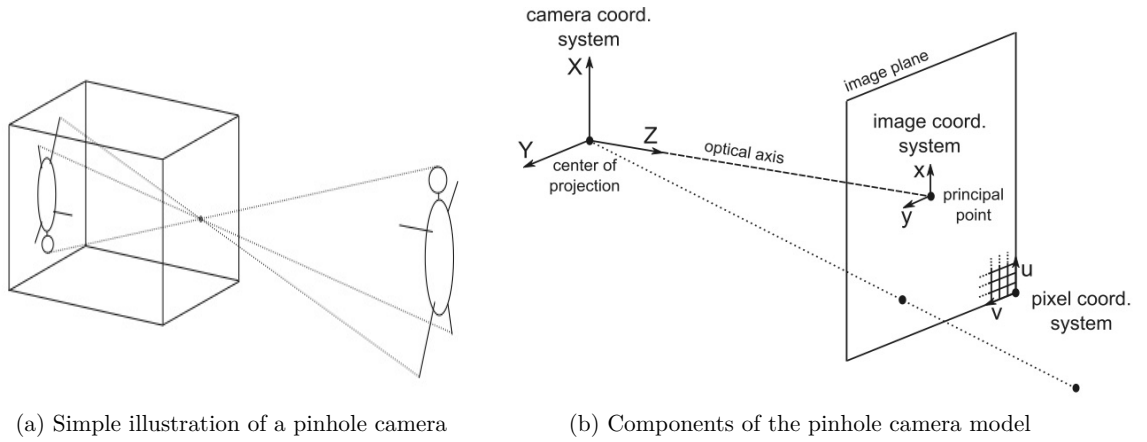


Figure 3.2: The pinhole camera model [57]

The distance between the center of projection and the image plane is the focal length f and the intersection point of the optical axis and the image plane is called the principal point. This point is also the center of the image coordinate system. The digital image is represented by a pixel coordinate system, which usually originates in one of the corners of the image. The offsets between the image and pixel coordinate system are $(-x_0, -y_0)$. Let (X, Y, Z) be the coordinates in 3D space. Then, we can use Equation 3.1 to calculate (x, y) in the image coordinate system [57].

$$x = f \frac{X}{Z} \quad y = f \frac{Y}{Z} \quad (3.1)$$

In the next step, the coordinates should be transferred to the pixel coordinate system. The values (x_0, y_0) are added as offset to the point of origin of the image coordinate system. Also, the aspect ratio of the camera, which represents the density of pixels horizontally and vertically, has to be considered. Therefore, we introduce the variables k_u and k_v (measured as number of pixels per millimeter) [57].

$$u = k_u f \frac{X}{Z} + k_u x_0 \quad v = k_v f \frac{Y}{Z} + k_v y_0 \quad (3.2)$$

Projection equations can also be expressed using homogenous coordinates, which results in Equation 3.3.

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} k_u f & 0 & k_u x_0 & 0 \\ 0 & k_v f & k_v y_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (3.3)$$

The \sim in Equation 3.3 signifies equality up to scale of vectors or matrices [57].

Intrinsic Camera Parameters

The intrinsic camera parameters describe what happens "inside" the camera. Specifically, these are the focal length in number of pixels ($k_u f, k_v f$) and the offset of the pixel coordinate system ($k_u x_0, k_v y_0$). There can also be a so-called skew parameter, but it can be neglected with modern cameras [57]. By convention the parameters are usually represented as an upper triangular matrix and this representation is called calibration matrix [58]. This matrix is also part of Equation 3.3.

$$K = \begin{pmatrix} k_u f & 0 & k_u x_0 \\ 0 & k_v f & k_v y_0 \\ 0 & 0 & 1 \end{pmatrix} \quad (3.4)$$

Extrinsic Camera Parameters

In addition to the intrinsic camera parameters, there are also two extrinsic camera parameters. Let R represent the camera's orientation and t the center of projection's coordinates in the world coordinate system. R is a 3x3 rotation matrix and t a 3D translation vector. As a result, we get Equation 3.5 to translate a 3D point from the world to the camera coordinate system [57].

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} R & -Rt \\ 0^T & 1 \end{pmatrix} \begin{pmatrix} X^w \\ Y^w \\ Z^w \\ 1 \end{pmatrix} \quad (3.5)$$

Parts of Equation 3.3 can be replaced and simplified with the calibration matrix from Equation 3.4 and the extrinsic camera parameters from Equation 3.5. Equation 3.6 is the result, where Id_3 is the 3x3 identity matrix and P is the so-called camera matrix [57].

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \underbrace{KR(Id_3 - t)}_P \begin{pmatrix} X^w \\ Y^w \\ Z^w \\ 1 \end{pmatrix} \quad (3.6)$$

3.1.3 Camera Calibration and Triangulation

The process to calculate the camera parameters mentioned in Section 3.1.2 is called camera calibration. The simple pinhole camera model does not take distortions into account, which occur due to the used camera lens. The distortions become more significant with a shorter focal length [16]. During the camera calibration process, these distortions are detected and can then be corrected with different techniques.

Distortion

The two major distortions which should be considered during camera calibration are radial and tangential distortion [58]. Radial distortion causes straight lines in a scene to appear curved on the recorded image. Figure 3.3 shows the two most common types of radial distortion. If the curved lines bend outward, it is called a barrel distortion. The opposite is the pincushion distortion, where the lines are bent inwards. The effects increase with distance from the center of the image.

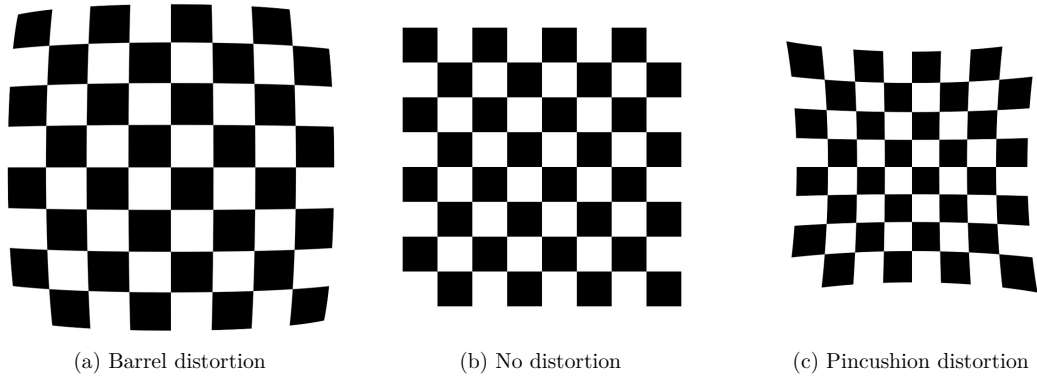


Figure 3.3: Different types of distortion compared to no distortion

Tangential distortion occurs if the camera lens is not aligned perfectly parallel to the image plane [12]. This leads to the effect that some parts of the image appear closer to the camera than others.

Camera Calibration

For this project, the camera calibration algorithm by Zhang was used [71]. The algorithm is already implemented in the OpenCV library. There are no specific requirements during the calibration, such as a predefined angle or distance to the camera. The proposed technique consists of the following six steps:

1. **Pattern selection:** First, one of the supported patterns, such as a checkerboard or a circle grid, has to be chosen. Then, the selected pattern has to be printed and attached to a planar surface (e.g. hard book cover).
2. **Take images:** In this step multiple images of the pattern should be taken from different orientations. Either the camera or the pattern can be moved. To get good results, at least ten images should be taken from different distances.
3. **Pattern recognition:** Then the pattern has to be recognized by the algorithm in each picture. For example, the 8×5 checkerboard from Figure 3.4 has 28 corner points between the squares, which are detected with a corner detection algorithm. The number of rows and columns of the printed pattern should not be the same to avoid orientation ambiguity.
4. **Estimate camera parameters:** The five intrinsic and all extrinsic camera parameters are estimated with a closed-form solution [71].
5. **Estimate distortion:** The first two coefficients of the radial distortion are estimated with the linear least-squares method.
6. **Refine all parameters:** All parameters are refined to consider the estimated distortion from the previous step. This is done with the Levenberg-Marquardt algorithm [38].

The full calibration with multiple images of the pattern has to be done only once per camera, because the intrinsic parameters of the camera will not change. After moving or rotating the camera, only the extrinsic camera parameters need to be calculated again, which can be done with only one image of the pattern. For this project the checkerboard pattern was chosen and, as seen in Figure 3.4, the feature points get detected by the camera calibration algorithm. The top left feature point of the checkerboard (the first red dot in Figure 3.4) is the point with the coordinates $(0,0)$.

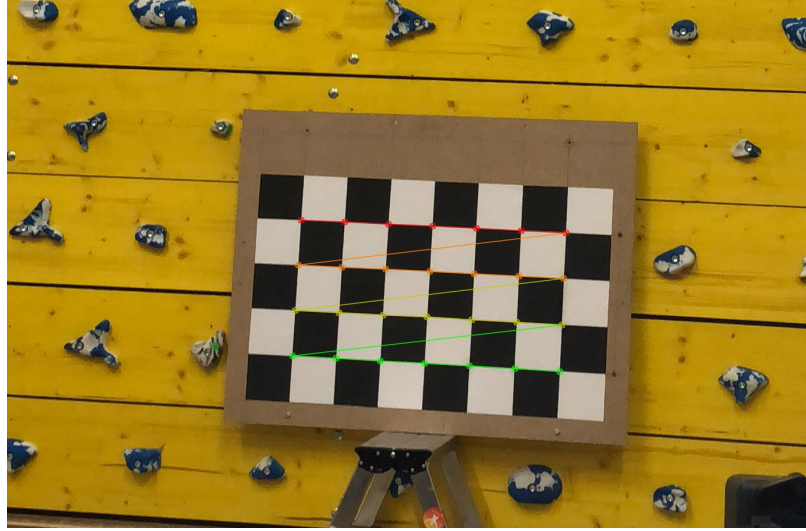


Figure 3.4: Checkerboard pattern with feature points overlay

Triangulation

The camera images are a projection from 3D to 2D space, so we are not able to determine the location of a point in 3D space from a single image. Therefore, two images from different angles are needed to calculate the location in 3D space. This calculation is called triangulation, and the cameras have to be calibrated first.

If we assume there are no measurement errors, we can just project rays back from the camera through the image points and would get the point in 3D space at the intersection of the rays. However, as seen in Figure 3.5a, the rays will never intersect because of minor errors during calibration and calculation of the image points x and x' [16].

To still get a point in 3D space despite the errors, we estimate a point \hat{X} , as seen in Figure 3.5b. \hat{X} projects to the two images at \hat{x} and \hat{x}' , which satisfy the epipolar constraint [18]. The point \hat{X} is chosen so that the reprojection error $d^2 + d'^2$ is minimized [16].

3.1.4 Image Manipulation and Blob Detection

VreeTracker uses different computer vision operations to extract the location of a marker from the images. This real-time location data is then used in a VR application, so the performance is very important. Simple operations such as cropping are used to only analyze the regions of the images which we are interested in. Because the cameras only detect IR light, the recorded images will be mostly black with some white spots representing the IR markers. To further improve efficiency, the color images are converted to grayscale images. A so-called blob detector then calculates the center of each white spot.

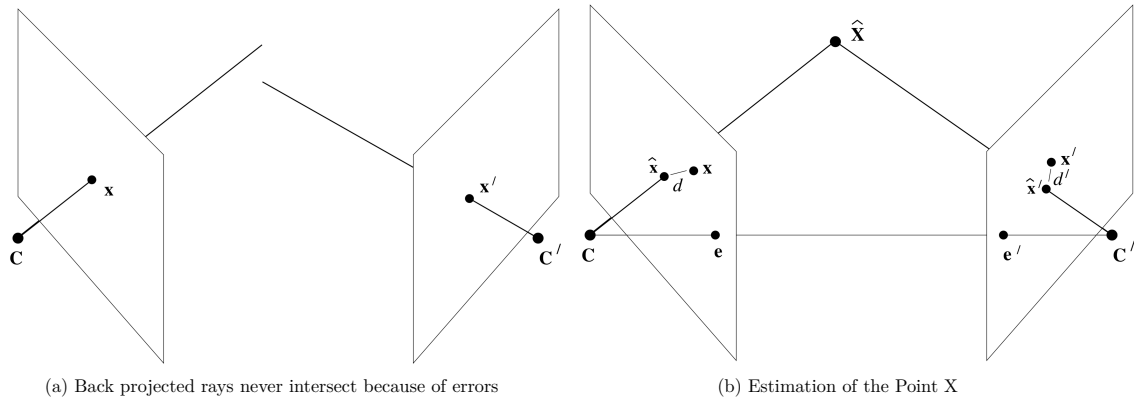


Figure 3.5: Triangulation in epipolar geometry [16]

Color Models

Color models describe colors in a 3D space. A color model has three channels which represent the axis of the coordinate system. Each point in this space describes a different color [18]. The following two examples show that the characteristics of color models can be quite different.

RGB

One of the most popular color models for digital images is RGB [10]. It describes a color by the mixture of the three primary colors red, green and blue [58]. A digital image is the representation of this information for each pixel. Figure 3.6 shows an image of the VreeTracker on the left and then a grayscale image of each color channel (red, green and blue). The red and green color channels from Figure 3.6 are brighter because yellow is a mixture of red and green. One advantage of the RGB model is that it is easy to understand and for example, thresholds can be found quickly with just looking at the data.



Figure 3.6: RGB image and each color channel (red, green and blue)

YCbCr

An alternative to the RGB color model is $Y'C_bC_r$. It was created for digital video encoding but is also used for image compression. A popular example is the JPEG compression for images produced by digital photography. The Y' channel represents the luminance of the color. The C_b and C_r channels represent the blue and red difference chroma. As shown in Figure 3.7, the Y' channel is more or less just a grayscale version of the original image. Only the C_b and C_r channels hold the information about the color. The C_b and C_r channels can be compressed without losing much image quality, because the human eye is more sensitive to the luminance. Equation 3.7 shows how an $R'G'B'$ image (gamma-compressed version of an RGB image) can be transformed to the $Y'C_bC_r$ model [58].

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad (3.7)$$

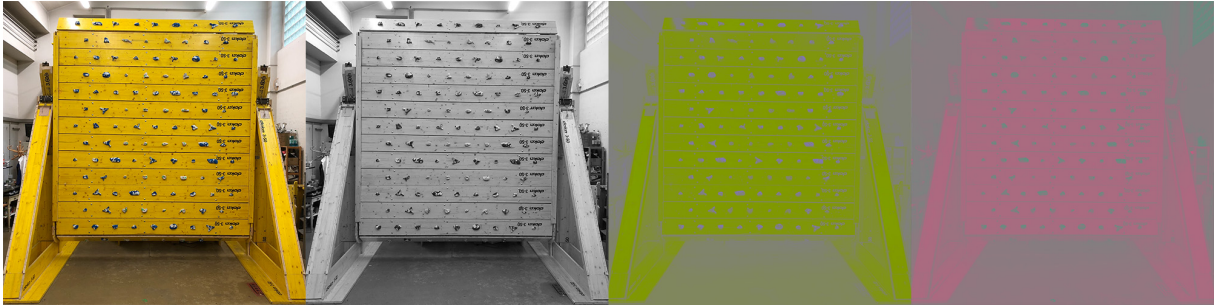


Figure 3.7: RGB image and each color channel (Y' , C_b , C_r)

Color Model Selection

Image processing is used in many different scenarios and it often is important to choose an appropriate color model. The RGB model is easy to understand and can be used to quickly create a prototype or select a threshold. A more practical example is an algorithm that uses the RGB model to automatically detect the state of an Rubik's Cube [15]. If the images of the Rubik's Cube are recorded in different lighting conditions (e.g. outdoor and indoor), the previously working algorithm fails. This problem is hard to fix because in the RGB space the change of illumination means that the variation of the color channel values is high. So if the thresholds are adjusted, similar colors get wrongfully detected.

One possible solution is to use the $Y'C_bC_r$ color model. Since it separates the image into luminance and chroma, mostly the value of the Y' channel changes with the different illumination. The variation of the chroma channel values is small enough to use thresholds and avoid false positives [15]. Another advantage of $Y'C_bC_r$ is the possible compression, which can be useful if many images should be saved or transferred over a network.

Blob Detection

In computer vision, a blob is a region in an image that has different properties such as color or intensity. A blob detection algorithm tries to find blobs which match a certain pattern [25]. There are many different blob detection methods, but because we only need to detect white blobs on a black background, a simple method based on edge detection is sufficient. Nonetheless, this operation is computationally expensive, and the parameters should be optimized to improve the performance.

For the VreeTracker software the SimpleBlobDetector of the OpenCV library is used [47, 51]. All relevant parameters for our scenario are listed in Table 3.1. The algorithm executes the following steps:

1. **Thresholding:** The source image is then converted into multiple binary images based on the threshold steps. For example, if we use the values (minThreshold = 120, maxThreshold = 150, thresholdStep = 10) three binary images with a threshold of 120, 130 and 140 are created.
2. **Find blobs:** In each binary image connected white pixels are detected as blobs. Furthermore, their centers are calculated.
3. **Grouping:** The calculated centers from the previous step are grouped together from all binary images based on the distance between them (minDistBetweenBlobs). If a group contains at least as many blobs as the minimum repeatability parameter, it is a valid blob.
4. **Center and radius:** In the last step the center and radius of the grouped blobs are calculated.

Parameter	Description
minThreshold	Minimum threshold (inclusive)
maxThreshold	Maximum threshold (exclusive)
thresholdStep	Step size between minimum and maximum threshold
minDistBetweenBlobs	Minimum distance between blobs
minRepeatability	Blob must be detected in at least as many threshold steps

Table 3.1: Relevant parameters for the SimpleBlobDetector [47]

Figure 3.8 shows an image with different circular blobs from white to almost black. The SimpleBlobDetector was used to detect blobs with a threshold of 60 to 255 and all found blobs are marked with a red circle. The blobs in the last line are under the threshold and are therefore ignored. For the VreeTracker only the center will be used as the current location of a marker in the image. The connected blobs in the middle and on the right side of the image are detected as one blob and the center is adjusted accordingly. This can potentially be one of the problems of occlusion and should be considered by the VreeTracker software.

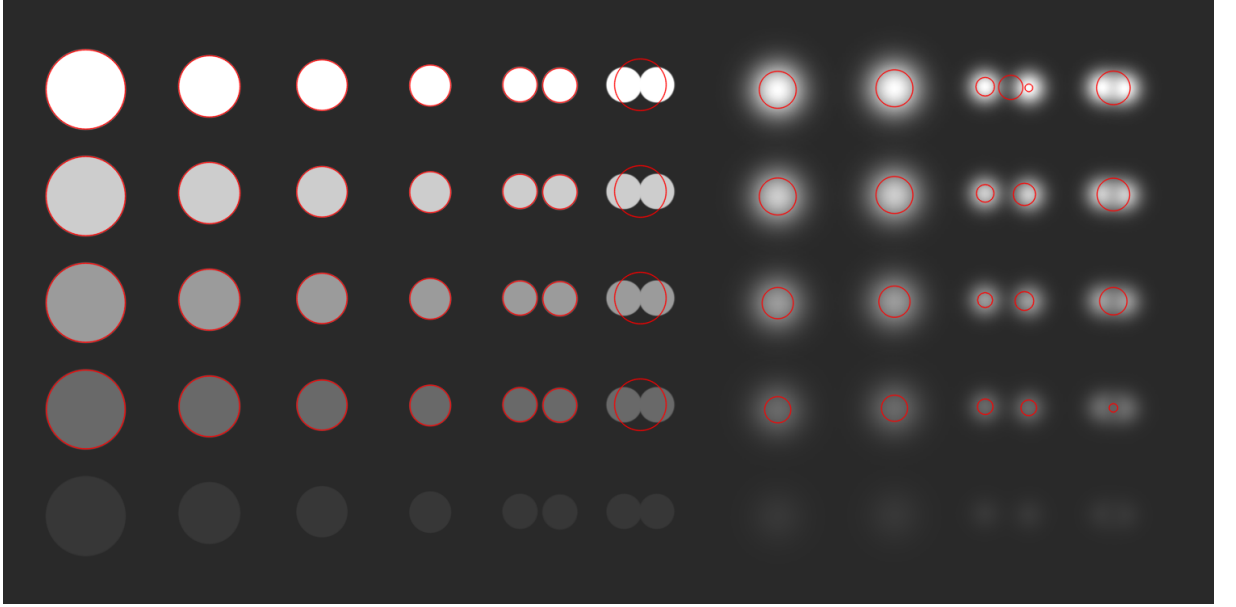


Figure 3.8: Example image with circular blobs in white and different shades of gray

3.1.5 Position Prediction with Kalman Filter

The Kalman filter was introduced in 1960 and is used to predict future states of a process based on previous measurements [24]. Some popular use cases are the Apollo Project and navigation systems [14]. The Kalman filter helps to reduce the noise of inaccurate GPS measurements and provides estimates if the GPS signal is lost (e.g., in a tunnel). For a vision-based marker tracking system, the usage of a Kalman filter has multiple advantages. Because of estimates during the camera calibration and triangulation, the calculated positions always have small errors. The Kalman filter reduces those error noises, which is especially useful for VR experiences, because even small unexpected motions can be perceived negatively [70]. Other problems of vision-based systems are occlusion and performance. With a Kalman filter, the marker's position can be predicted if the precise calculation takes too long or the marker is currently not visible.

Requirements

The Kalman filter cannot be used on all problems where some values need to be estimated. The process needs to be convertible to following two equations [70].

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \quad (3.8)$$

$$z_k = Hx_k + v_k \quad (3.9)$$

Equation 3.8 estimates the current state $x \in \mathbb{R}^n$ of a discrete-time controlled process. k represents the current time step and A is a $n \times n$ matrix that relates state of the previous time step to the current state. The $n \times l$ matrix B relates the optional control input $u \in \mathbb{R}^l$ to the state x . w_{k-1} is the process noise of the previous time step.

The measurement Equation 3.9 describes a measurement $z \in \mathbb{R}^m$. H is a $m \times n$ that relates the state to the measurement z_k and v_k is the measurement noise. In practice H can change with each time step, but often we can assume it is constant [70].

Kalman Filter Cycle

The two repeating steps of the discrete Kalman filter are time update and measurement update as seen in Figure 3.9. The time update returns a prediction for the future based on previous measurements. The measurement update adjusts the estimate from the time update with an actual measurement. The two steps are explained in more detail later in this section. After each step the next step uses the calculated values from the previous step. This recursive nature is one of the advantages of the Kalman filter. It is not necessary to save all previous measurements and estimates, but they are still considered for the current step.

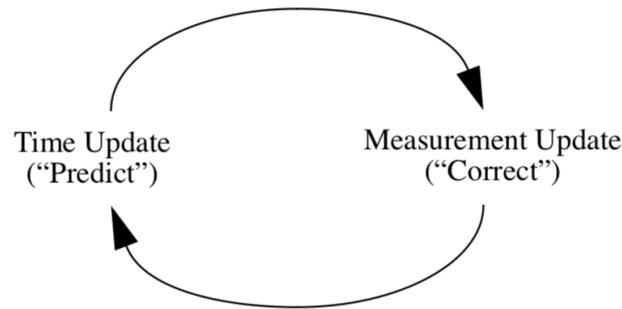


Figure 3.9: Cycle of a discrete Kalman filter [70]

Time Update

The time update provides a prediction of the state and the error covariance for the next time step. Table 3.2 contains the specific equations for this step. Equation 3.10 calculates $\hat{x}_k^- \in \mathbb{R}^n$ (note the "super minus"), which is defined as the a priori state estimate at step k with consideration of all knowledge collected before step k [70]. $\hat{x}_k \in \mathbb{R}^n$ is a posteriori state estimate at step k given measurement z_k . A , B and u are from Equation 3.8. Equation 3.11 calculates the a priori estimate error covariance P_k^- at step k with the process noise covariance Q .

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \quad (3.10)$$

$$P_k^- = AP_{k-1}A^T + Q \quad (3.11)$$

Table 3.2: Discrete Kalman filter time update equations [70]

Measurement Update

The measurement update consists of the three equations from Table 3.3. First, the Kalman gain K from Equation 3.12 is calculated with H from Equation 3.9 and the measurement noise covariance R . The Kalman gain is a $n \times m$ matrix and defines if the prediction or the measurement should be trusted more. Then the a posteriori state \hat{x}_k is calculated with the measurement value z_k and the Kalman gain from the previous calculation. In Equation 3.14 the a posteriori error covariance is estimated, which is needed for the next time update.

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} \quad (3.12)$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \quad (3.13)$$

$$P_k = (I - K_k H) P_k^- \quad (3.14)$$

Table 3.3: Discrete Kalman filter measurement update equations [70]

3.2 The Human Hand

To create a realistic virtual hand, it is not enough to create a 3D model that looks like a human hand. It is also important that its movements look realistic. In modern game engines such as Unity, it is possible to create a skeleton of bones and joints for models. Then, the meshes of the model are linked with the closest bone. If the skeleton or parts of it are moved, all connected meshes will also move accordingly and make the movements look more real. In this section the anatomy of the human hand is described to better understand how the virtual hand should move. Furthermore, grasp poses typically used during climbing are explained.

3.2.1 Anatomy of the Human Hand

As seen in Figure 3.10a, the 27 bones of the human hand are segmented into five groups. The carpals make up the wrist and the root of the hand. The anatomical design of each finger is very similar, with some exceptions for the thumb. While the metacarpal bone of the thumb can be moved independently, the other four metacarpals are closely linked and only allow very limited independent movements. The human hand has 14 phalanges: three in each finger and only two in the thumb, because it is missing the middle phalange [59].

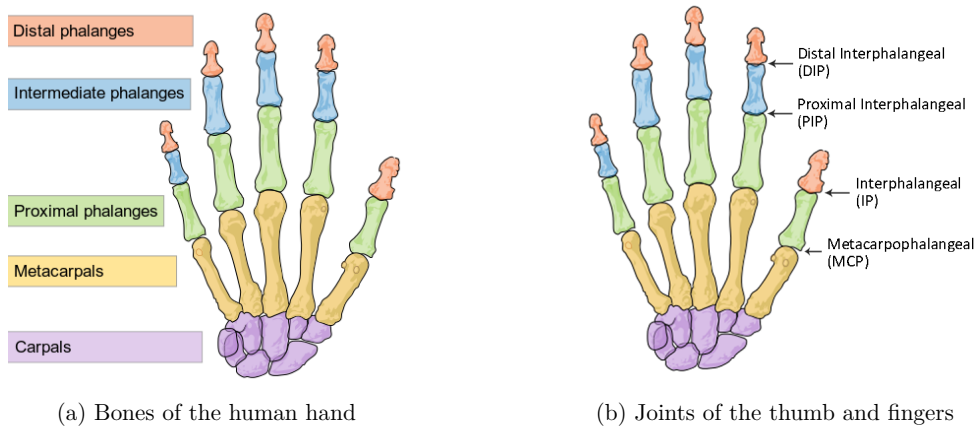


Figure 3.10: Bones and joints of the human hand [66]

3.2.2 Joints and Range of Motion

Most muscles that move the hand and fingers lie in the forearm. Tendons connect the muscles with different parts of the hand [59]. They can easily be seen on the back of the hand when the fingers are moved. Figure 3.10b shows the joints of the fingers between the bones. For our climbing scenario, the most interesting joints are the metacarpophalangeal (MCP), proximal interphalangeal (PIP), distal interphalangeal (DIP), and interphalangeal (IP). The difficulty is to choose appropriate angles for each joint without any additional data. We only know the position and orientation of the hand.

Table 3.4 shows the range of motion of each joint and the average position during the grasp of different objects [4]. During a grasp, the PIP is the most angled joint. Furthermore, research has shown that there is a relation between the positions of different joints [8]. This relationship is especially noticeable on the PIP and DIP joints of one finger, but generally the positions of the four fingers are similar during a grasp. Of course, this also depends on the form of the object, but this information can be useful to animate more realistic virtual grasps on climbing holds.

	Joint	Range of Motion	Average Position during Grasp
Thumb	MCP	0 - 56°	10° (\pm 9°)
	IP	-5 - 73°	28° (\pm 11°)
Fingers	MCP	0 - 100°	33° (\pm 6°)
	PIP	0 - 105°	39° (\pm 7°)
	DIP	0 - 85°	26° (\pm 5°)

Table 3.4: Thumb and fingers range of motion and average position during a grasp of an object [4]

3.2.3 Different Grasp Poses

The human hand can adapt to the form of many differently shaped objects. Figure 3.11 shows the six basic types of prehension defined by Schlesinger [1, 59]. For climbing the most relevant grasp poses are the cylindrical and spherical grasp, but some smaller climbing holds also require a hooklike pose. The climbing holds of the VreeClimber all have similar forms, so one algorithm for the grasp pose should be enough. But the algorithm should consider the orientation of the hand during the grasp. For example, if the climber reaches for a climbing hold right above his head the pose should be different to a grasp on his side where the hand is in an angle of 45 degrees. The target group of the VreeClimber are unexperienced climbers. Therefore, complicated climbing grasps such as from underneath are not specifically considered.

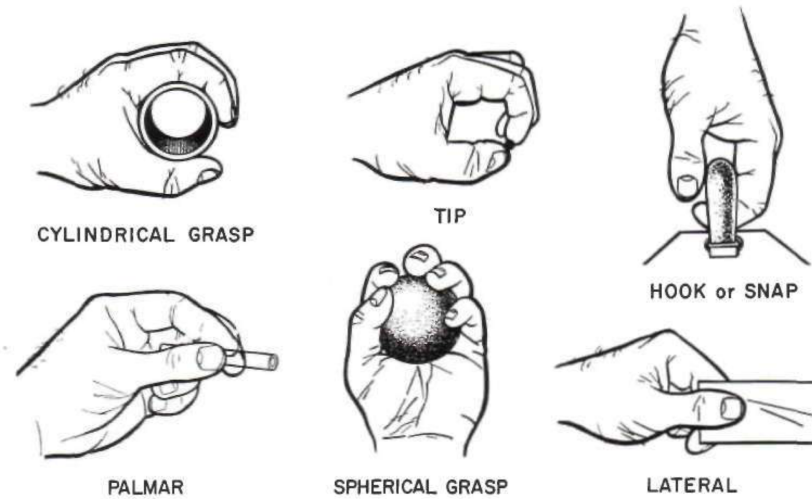


Figure 3.11: Six basic types of prehension [59]

Cylindrical Grasp

This pose is used when a cylindrical object is grabbed. All fingers are side by side and flexed around the object. The thumb is wrapped around the other side, and if the object is small enough, the thumb can overlap the fingers. During climbing this grasp gives the climber stability because the climbing hold is grasped from two sides.

Spherical Grasp

The fingers and thumb are placed around the object. There are not many spherical climbing holds, but grasps similar to this one are still frequently used during climbing. This pose also provides good stability because the fingers are on opposite sides of the object.

Hook

In this pose, usually only the four fingers are used. They are flexed like a hook and wrapped around the object. Climbers use it on small climbing holds, and only a fraction of the force can be applied. Most of the time the thumb is not used, because the surface of the hold is too small.

The first section of this chapter gave an overview of the technologies and methods necessary for hand tracking. The second part described the anatomy of the human hand and grasp poses used during climbing. This information is the basis for the next chapter, which specifies the architecture of the VreeTracker and the virtual hand simulation.

System Design

The last chapter gave a theoretical overview of different topics, which are the basis for this project. This chapter is the beginning of the more practical part of this thesis. First, the climbing scenario is briefly described and then the software requirements are defined. Then, the hardware and software architecture of the VreeTracker is explained. The system design of the virtual hand simulation is presented in the last part of this chapter.

4.1 Requirements

Before the implementation of the tracking software and virtual hand simulation, different requirements should be defined in order to become able to continuously verify that the original goals of the project are met by the software. The requirements should be clear and easy to check, but especially for a VR application this is not always possible since every user perceives the virtual world differently. The requirements focus on the VreeTracker software and the virtual hand simulation software, because the hardware was developed in a previous project [56]. Table 4.1 gives an overview of the requirements at the end of this section.

4.1.1 Climbing Scenario

As mentioned in the introduction, the VreeClimber consists of multiple boards that slowly move downward when the climber reaches a certain height on the climbing wall. The climber wears an HMD of a VR system and sees a virtual world where they can climb. The combination of the VreeClimber functionality and the virtual world theoretically allows an endless virtual climbing scenario. In front of the VreeClimber, two stereo cameras are placed to track four IR markers, which the climber wears at their wrists and ankles. Also, two base stations of the HTC Vive are placed in front of the climbing wall to guarantee the tracking of the HMD and the controllers. Before the system can be

used, both stereo cameras, the VR system, and the location of the climbing wall in the VR application need to be calibrated.

4.1.2 VreeTracker Software

For the marker tracking, accuracy and performance are the most important requirements since the VR application needs accurate position data to show the user's hands at the correct location from their point of view. Performance is important, because the data is useless if the software needs multiple seconds to calculate it. Therefore, the tracking software should be optimized to calculate the position nearly at the frame rate of the VR application, which is 60 frames per second. Even if some precision is lost, it is crucial that the tracked movements are in real time and look smooth in the virtual world. Another important requirement is, that each marker is associated with the correct extremity. This problem can always occur when a visual tracking solution is used. So temporary occlusions should be considered by the software. Other metrics to consider are a good tracking coverage of the whole climbing wall and minimization of possible occlusions.

4.1.3 Virtual Hand Simulation

The virtual hand simulation has to be developed as an algorithm for Unity and should include an attractive hand model. The most important requirement is that the movements of the hand should feel natural for the climber and should help them to safely grasp for the climbing holds. To achieve this, the grasp animation has to be adapted for each climbing hold. One factor is the size of the hold. For example, a big hold can be grabbed with the complete hand, whereas only the fingertips are used on small ones. The solution should be useable for many different climbing hold forms, hence it makes no sense to categorize them manually. The algorithm should try to get all necessary information, such as size and depth, from the 3D model automatically.

Another problem can be graphical glitches where parts of the hand move into the climbing wall. These can be especially difficult to solve because the virtual hand should grasp the climbing holds. Therefore, a procedure should be developed which allows the virtual hand to grasp an object as closely as possible, but also prevents overlapping between them. As mentioned earlier, the climbing holds have various forms, therefore the animation should be individual for each finger. For example, if the index finger already touches the hold, the animation of the index finger is stopped. However, the other fingers should still continue to move until they also touch the hold or reach their final position. The animation algorithm should be efficient enough to run with 60 frames per second on consumer hardware, which is the normal frame rate of VR applications.

4.1.4 General Software Requirements

Besides the specific requirements mentioned above, all parts of the software should meet some general requirements. The code should be well-structured and separated into smaller modules. This allows the replacement or extension of specific modules without the need

VreeTracker	
Accuracy	Accuracy of the marker tracking should be $< 0.5\text{cm}$
Performance	60 frames per second with calculation and prediction
Tracking Coverage	Good coverage of the climbing wall to avoid occlusions
Occlusion Handling	Occlusion handling to correctly associate markers with extremities
Distributed System	Possibility to run the software on two PCs
Virtual Hand Simulation	
Natural Movements	Hand movements should look natural and smooth
Different Grasp Poses	The software should support different grasp poses
Individual Animations	Each finger should be animated individually
Performance	60 frames per second while animating two hands
General Requirements	
Modular Code	Split software into smaller modules
Configurability	GUI with settings or config file
Documentation	Code should be well documented

Table 4.1: Requirements for the VreeTracker and virtual hand simulation software

to rewrite the entire software. Furthermore, the software should be configurable with a graphical user interface (GUI) or a config file to quickly change some parameters. Due to possible hardware limitations, it should be possible to run the VreeTracker software for each stereo camera on a separate PC. As a consequence, the results of the calculation should be transferred over a network connection with the user datagram protocol (UDP). Additionally, the code should be well-documented, especially the more complex parts.

4.2 Marker Tracking

In this section, the hardware and software architecture of the VreeTracker are explained. First, the hardware components, which were built by Ludwig Steindl, are described to better understand their functionality. Then, the architecture of the marker tracking software is described.

4.2.1 Tracking Setup

The VreeTracker uses a stereo camera to track the location of the marker. The 3D location of the marker is calculated with triangulation, which means that both cameras need an unobstructed view of the marker. Theoretically, this can be done for all four markers with one stereo camera, but as soon as the climber moves their hands in-front

of his body, the tracking is not possible anymore. This is why a setup with two stereo cameras was chosen.



Figure 4.1: Setup of the stereo cameras in front of the VreeClimber

Figure 4.1 shows the two stereo cameras in front of the climbing wall. Each stereo camera tracks a hand and a foot of one side of the climber's body. The cameras are placed at the edge of the climbing wall so that they have a good field of view and the view is obstructed only if the markers are right in front of the climber. Because of the camera calibration method, which was explained in Section 3.1.3, there are no restrictions on the exact location of the cameras or the distance to the climbing wall. During the development, a setup such as the one shown in Figure 4.1 provided good results.

4.2.2 Hardware Architecture

The VreeTracker system consists of two different types of hardware components. The first type are the IR markers, which are equipped with active IR light LEDs and inertial sensors. The second type are the modified webcams that only record IR light. For the tracking of all extremities of a climber, four markers and four cameras are required.

IR Marker

The VreeTracker uses active IR markers for the visual tracking of the extremities. The hardware prototype was built by Ludwig Steindl and consists of a microcontroller, circuit board, IR LED, battery, and a 3D printed case with one or more straps. Figure 4.2 shows the last version of the VreeTracker prototypes. On the left is a marker for a hand with a wristband, and on the right is a marker with three Velcro straps to attach it to the climber's ankle and shoe.



Figure 4.2: Two VreeTracker marker prototypes with straps for the wrist and ankle

All markers are equipped with a light diffusing sphere to emit the light in all directions. This component makes them more visible for the cameras in different orientations. They are only invisible for the cameras when the diffusing sphere is facing the wall, but this should not be the case during the climbing. The small red switch on the front is used to turn the tracker on and off. By removing two screws, the 3D printed case can be opened quickly to replace the Li-Ion battery.

The hardware of a basic active IR marker for visual position tracking could be simpler, but with the present components the VreeTracker marker can also track its orientation. An inertial sensor array is used to calculate the absolute orientation of the marker. The microcontroller connects to a Wi-Fi network and transmit the marker's orientation in real time. More details about the marker prototype and its development can be found in Steindl's thesis [56].

Cameras

The used webcams are Logitech HD Pro Webcam C920 [32]. These webcams can record 30 frames per second with a resolution of up to 1920x1080 pixels. By default, they are equipped with a bypass filter, which only lets visible light through. Since we want to record IR light, the filter has to be removed and replaced with an IR pass filter. The original filter is placed right in front of the camera sensor, which means each camera had to be opened to remove the filter. A simple IR pass filter for SLR cameras is attached at the front of each webcam as seen in Figure 4.3 [40].

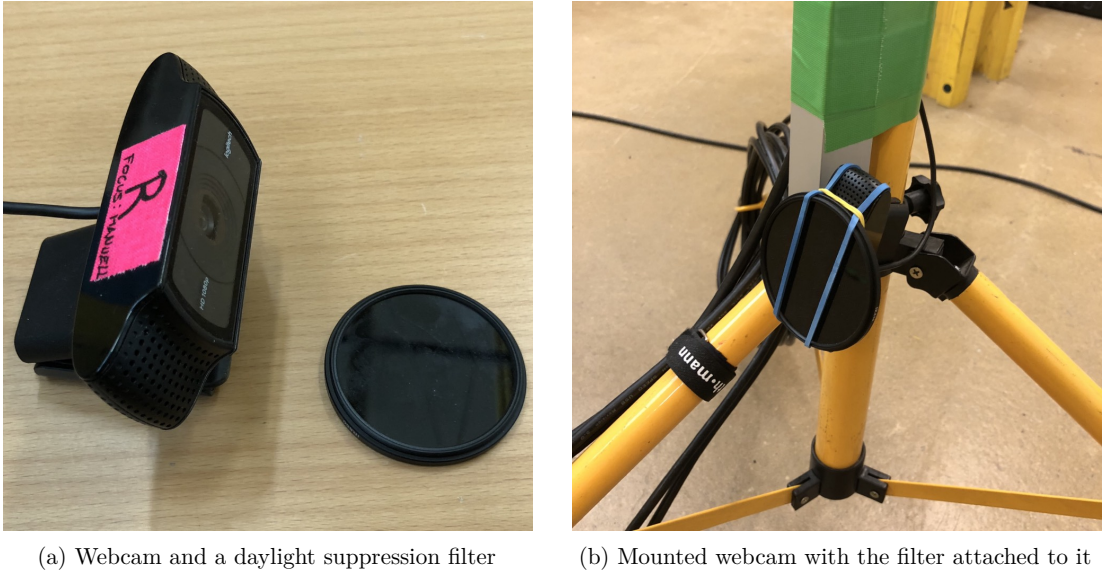


Figure 4.3: Logitech HD Pro Webcam C920 with a suppression filter

For each side of the climber's body, two webcams are used. As seen in Figure 4.1, a stand is used for this stereo camera setup, where one camera is placed at the bottom and the other one on the top to get more precise tracking results. The cameras are rotated inwards to better cover the whole climbing wall with their 16:9 aspect ratio. This helps to avoid problems with significant occlusions and still gives a good view of the whole VreeClimber wall. The two webcams are connected to a USB hub, which is then connected to a PC that runs the image processing software. In the remaining parts of this thesis, such a stereo camera setup is referred to as a *camera rig*.

Wi-Fi Router

To simplify the Wi-Fi setup for the IR markers, a pre-configured Wi-Fi router is used. The router's SSID and password are known by the markers so that they can automatically connect to it. If multiple PCs are used, they can be connected to the router with LAN cables to avoid Wi-Fi delays.

4.2.3 Software Architecture

For the marker tracking, two different software components are necessary. The component which is developed as part of this thesis analyzes the images from the stereo cameras and calculates the location of the markers in front of the climbing wall. The second component runs on each marker's microcontroller and uses different sensors to calculate the current rotation of the marker. This software was written by Ludwig Steindl and it was possible to use it for this project without any changes.

Tracking Software

The main purpose of the tracking software is to use computer vision to calculate the location of the markers. But as stated in Section 4.1, it also has to consider different aspects such as occlusions to avoid calculation errors. Each functionality is split into a separate module, and code is reused where possible. The inputs of the software are the frames from the camera rig, and for each frame pair the same steps are necessary to calculate the marker positions in the frames. Figure 4.4 shows the basic architecture of the tracking software. Each module represents a specific task, and its output is the input for the next module.

Stereo Camera

This module uses the OpenCV library to configure and control the two webcams of a camera rig. Before the actual tracking, this module also handles the camera calibration. If a full intrinsic and extrinsic calibration is necessary (e.g. if new webcams are used), 15 images of the calibration pattern in different orientations and positions have to be recorded. For the extrinsic calibration, one image of the calibration pattern is enough to determine the camera's position relative to the pattern. After the calibration, this module reads the frames from the two webcams at a specified frame rate. The frame rate is limited by the webcams, which is 30 frames for the Logitech C920 [32].

IR Interference Filter

The modified webcams record any IR light. This means that if there are any other IR light sources or reflections near the climbing wall, the software could mistake it for markers. The most common interferences are sunlight or reflections of it. This module analyzes the first few frames after the calibration and creates a mask with all visible IR spots for each webcam. This mask is then subtracted from all subsequently recorded frame pairs in order to eliminate these static interferences.

Region of Interest Filter

The efficiency of the software is important to get a high frame rate. A simple method to improve efficiency is to reduce the frame size. The user of the software can select a region of interest (e.g. only the area which contains parts of the climbing wall). Each frame is

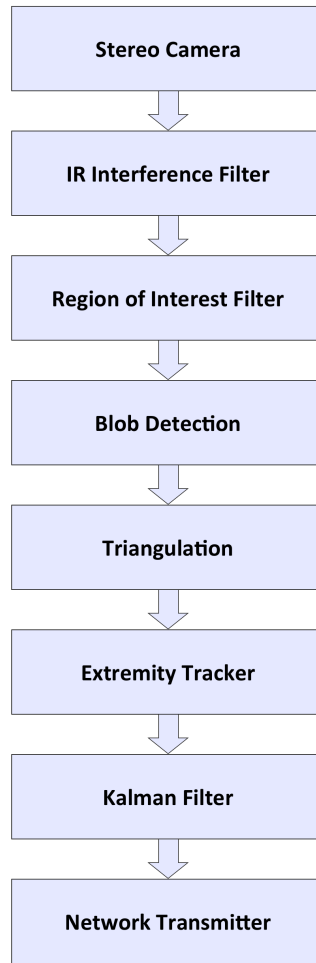


Figure 4.4: Software modules of the tracking software

then cropped to this region of interest by this module. A positive side effect of this is that other IR lights outside of the region of interest are as well ignored by the software.

Blob Detection

The principles of a blob detector were already explained in Section 3.1.4. The blob detection module uses the SimpleBlobDetector from the OpenCV library to find all blobs with a specific minimum size and distance between them [47]. The result of the SimpleBlobDetector is a list of KeyPoints. A KeyPoint is a data structure from OpenCV that describes blobs. The most useful values for our use case are the coordinates of the center and the size of the blob. The final output of the blob detection module is a list of KeyPoints for each frame.

Triangulation

The triangulation module tries to find all reasonable points in the 3D space by combining the blobs previously found in the frame pair. This module only uses the two blobs, which are closest to the edge of the tracked side of each frame. For example, if the left hand and foot should be tracked, the blobs closest to the left edge are used. In the next step, all combinations of the selected blobs are used to triangulate a point in 3D space. This means that we get four points in 3D space if at least two blobs are found in both frames. Points which do not meet certain criteria, such as being inside an area of one meter in front of the climbing wall, are filtered out. Most of the time, this is enough to filter out unreasonable points.

The combination of all points is necessary because two or more markers could be close to each other or a marker could be temporarily not visible to a camera. With this additional information the software can try to find the most reasonable combinations. The module returns a list of possible 3D points where a marker could be located.

Extremity Tracker

At this point, the triangulation module calculated all possible points and the extremity tracker module tries to match these possible points with the last position of the extremities. The basic idea is to keep track of what the last positions of the extremities were and only consider points which are near that position. There are also configurable tolerances of how far the distance between the last and the new position may be. This helps to avoid misinterpretations if a marker is temporarily occluded. The position is updated only if the calculated location is inside the tolerances of the last position of an extremity. The extremity tracking only starts when all markers are visible to get an initial position for all extremities.

Kalman Filter

A detailed explanation of the Kalman filter was already given in Section 3.1.5. In the VreeTracker software, the Kalman filter has two purposes. One is to smoothen the movements of the extremities, which improves the VR experience. The other is to predict a location between two calculations of the algorithm. The prediction is needed because the webcams can only record with 30 frames per second, but the VR application runs with 60 frames per second. Therefore, between each calculation from the webcam images, a predicted position is added. It is also possible to get a relatively good position prediction if a marker becomes briefly invisible.

Network Transmitter

The network transmitter module sends the calculated and predicted positions of the extremities over a network to a configurable IP address. The module uses UDP and the port is different for each extremity. For example, the position of the left hand is sent to

port 8001 and the left foot's position is sent to port 8002. The VR application only has to listen to these ports to get the newest position data. This approach allows the use of multiple computers and other software can easily use the position data if needed.

Marker Software

As mentioned above, the software for the marker was written by Ludwig Steindl and runs on a microcontroller [56]. It combines data from an accelerometer, gyroscope, and magnetometer to calculate the absolute orientation. The microcontroller automatically connects to a pre-configured wireless network and sends the current orientation in real time to a specified port. It also uses UDP for this transmission.

4.3 Virtual Hand Simulation

The VR application was created with Unity and uses an HTC Vive to give the user a virtual climbing experience. The real time position and orientation data for the climber's limbs come from the VreeTracker. The main purpose of the application is to simulate the hand movements of the climber. Figure 4.5 shows the different components which are used by the application. At the top are two instances of the VreeTracker software, which send the real time positions of the climber's body parts to the Unity application. Additionally, the four markers themselves send their current orientation to the application. To receive and attribute the data to the right body parts the software listens on multiple ports for the UDP packets.

On the bottom of Figure 4.5 are SteamVR and the HTC Vive. SteamVR is the software which is needed to use the HTC Vive on a PC. To integrate the Vive in a Unity project the SteamVR SDK has to be used [63]. The SDK handles all the communication with the Vive and automatically includes the HMD at the correct position in the scene. The main view of the scene is linked with the HMD, so if the user moves his head around, the visible area in the scene changes accordingly. The Vive controllers are also supported out of the box and are used in this project to calibrate the exact location of the climbing wall. The 3D hand model is another component which is added to the Unity project. This universally usable hand model can be purchased from the Unity Asset Store and is the base for the virtual hand used in the simulation [62]. The next few sections explain the different components in greater detail.

4.3.1 Physics Colliders

The physics engine of Unity can be activated for any object in the scene. For example, it is possible to add a ball to the scene, and with a few configuration changes this ball can behave like a metal or rubber ball. To get a physically realistic behavior in a scene, all physics objects need to be moved with forces. For instance, the ball is not moved one meter from its current position, but a force is applied to it, which moves the ball. This approach is not suitable for our project because we get the location of the climber's

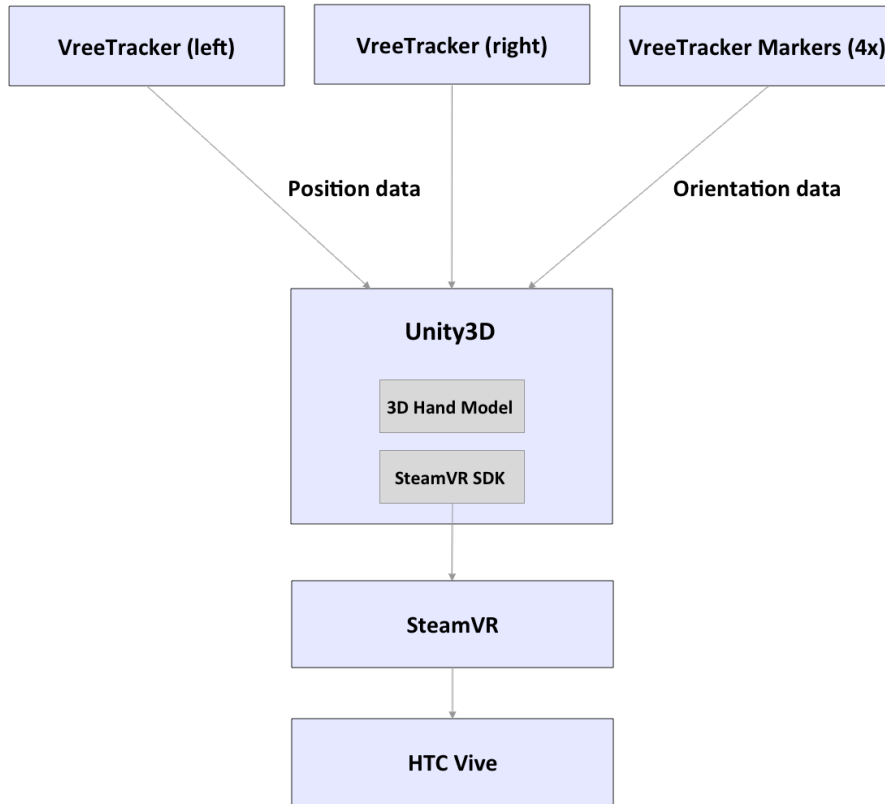


Figure 4.5: Overview of the components used by the VR application

body parts from an external source. This is why physics colliders are only used to detect physical collisions, but the objects are not controlled by the physics engine.

The colliders are invisible and do not need to have exactly the same shape as the visible objects they are attached to. It is recommended to use simple shapes such as boxes or spheres to roughly approximate the shape to make the calculations more efficient. If two physics colliders intersect each other, a collision is detected by the physics engine. A custom script is used to handle the collision accordingly and stop the animation of the involved finger, for example.

The basic idea is to define the virtual hands, the climbing wall and the climbing holds as physics colliders. That way, it is possible to detect when the virtual hand models touch the climbing wall or a climbing hold. To create a more realistic animation of the grasp movement, three colliders are added to each finger. With this setup it is possible to detect which part of a finger collided with the wall. With this information the animation script can then stop the animation for certain parts of the finger that collided. To make



Figure 4.6: Hand model with physics colliders

the grasp more realistic, the other parts of the finger keep moving until they also collide with the wall or the climbing hold. Another positive side effect is that no graphical glitches occur, where parts of the virtual hand move into the climbing wall, because the animation is stopped when the objects touch each other.

One aspect that still needs to be considered is the hand movements of the climber. For example, even after all virtual fingers collided with the wall, the hand could be moved in any direction and now the position of the fingers do not match the climbing hold anymore. This means that the finger positions have to be adjusted after each movement of the virtual hand.

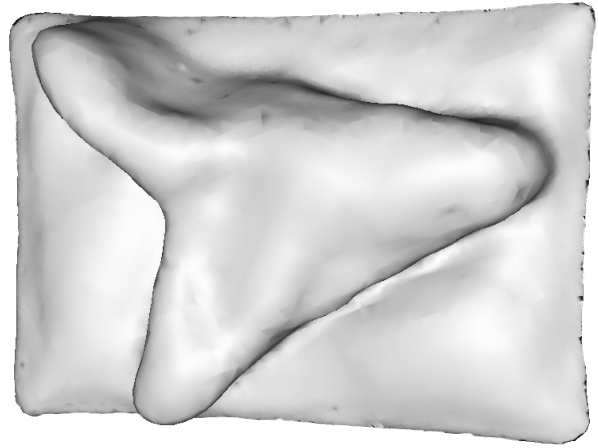
4.3.2 Climbing Grips

The climbers wear the HMD of the VR system, so they cannot see the real climbing wall or the climbing holds. The climbing wall has to be rebuilt in the virtual world and the climbing holds have to be of the same size and at the same location as on the real climbing wall. Especially for a virtual climbing scenario, it is very important that the

climbers can see the climbing holds to safely grab them. Besides the size and location, the form of the virtual climbing holds should also be at least very similar so the climber can trust what they see and do not have to feel the form of every hold. One possibility to achieve this behavior is to 3D scan all climbing holds. Figure 4.7 shows a real mounted climbing hold on the left and its 3D scan on the right. The 3D scan is not perfect, but the quality is generally good enough for our virtual climbing scenario. Even with better scans, there would be some tracking and calibration errors. In the current approach, the 3D scans are Wavefront 3D object files (.obj file extension) and can be directly imported in Unity.



(a) Climbing hold on climbing wall



(b) 3D scan of climbing hold

Figure 4.7: Image and 3D scan of the same climbing hold

For the VreeClimber, many different climbing holds are used, so each one should be marked (e.g. with a number) and matched with the correct 3D scans. This helps to quickly add them to the unity project at the right location. The recreation of the climbing wall in Unity3D can still be a time-consuming task. As mentioned earlier, the alignment of the virtual climbing holds does not have to be perfect, but the position, orientation and size should be roughly the same to ensure a good VR immersion. As seen in Figure 4.7b, the plane on which the climbing holds were placed during the 3D scanning is also part of the 3D model. To avoid that this area is shown in the virtual world, the models need to be placed by a small distance inside the climbing wall.

4.3.3 Closing of the Hand Model

The VR application only gets position data for the whole hand and not for each finger. Therefore, an algorithm is necessary to calculate when the virtual hand should start closing to grasp a climbing hold. During a usual grasp movement the hand is already closing while it approaches the climbing hold. Hence, we need to calculate how far the

hand should be closed during the approach. The algorithm calculates a value between 0 and 1 for each joint individually to describe how far it should be closed. For example, a value of 0 means it has the same angle as in the open hand pose, while a value of 1 means it is bent to represent the fully closed hand pose. After each hand movement the values are recalculated and consequently the hand movements look smooth, even if the hand is moved back and forth.

The calculations of the algorithm are based on two factors. The first factor is the distance of the finger to the climbing hold. This factor can be configured with two settings in Unity. The first setting is the distance where the closing animation of the virtual finger starts. The second setting is the distance where the finger is closed as far as possible, if it did not already collide with the climbing hold. The calculation of the distance between two complex 3D Objects can be a computational expensive task, hence an invisible bounding box is created automatically for each climbing hold. This makes the calculation of the distance between the fingertip and the climbing hold faster and the result is still sufficiently accurate.

The second factor is the average joint angle value from the other fingers of the hand. This is the result of findings from Section 3.2, where it was described that the angles of the different finger joints are related to each other. This relation is especially important if the hand approaches a climbing hold sideways. Without this factor the fingers closer to the climbing hold would start closing, while the fingers on the other side of the hand would still be extended. In most cases this animation would look unnatural. Therefore, the average joint angle value is added to the calculation, if the difference is greater than a certain threshold.

4.3.4 Different Grasp Poses

The used climbing holds on the VreeClimber vary significantly. Some are big enough to be grabbed with the whole hand, while others are very small and only the fingertips can be used. This means that the climber's hand automatically adapts to the different climbing hold sizes. Figure 4.8 shows some grasp poses for different climbing holds. In the first image the whole hand is used to get a good grip, while in the other two images mostly the fingers are used to grab the smaller climbing holds. Therefore, the grasp algorithm distinguishes between big and small climbing holds. This distinction is based on the depth of the 3D model. For each size a different animation is used to better match the real grasp pose of the climber.

Besides the size of the climbing hold, the orientation of the climber's arm also affects the grasp type. As seen in Figure 4.8b, the climber will try to place his thumb and one or two fingers on top of the climbing hold during a horizontal grasp to get a stable grip. Compared to a vertical grasp the joints are not bent as much. Figure 4.8c shows a vertical grasp of a small climbing hold, where the thumb does not even touch the climbing hold. As a result of this observation, the algorithm also differentiates between a horizontal and vertical grasp. This distinction is made additionally, after determining if a climbing hold

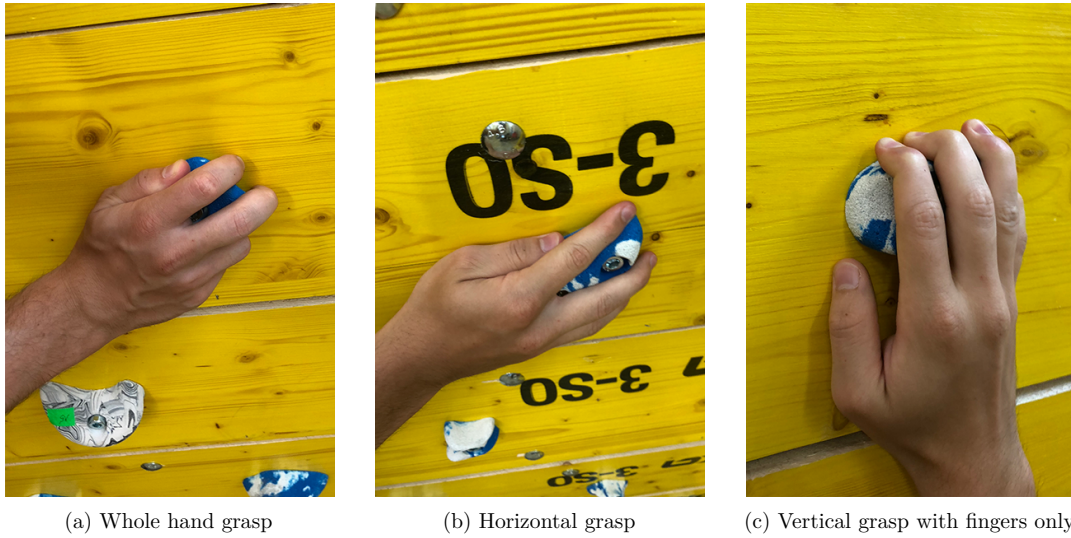
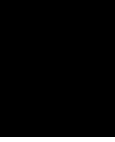


Figure 4.8: Examples of different grasp poses

is big or small. Therefore, the algorithm supports four different grasp poses to provide a good simulation of the grasp movements. This part of the software was designed to be easily extendable by further distinctions.

In the first part of this chapter, the requirements for the VreeTracker and virtual hand simulation software were defined. Then, the architecture of the VreeTracker hardware and software was presented. In the last part, the system design of the virtual hand simulation was explained. These information given, the next chapter describes the technologies and methods that were used to implement the VreeTracker and the virtual hand simulation software.



Implementation

This chapter describes how the marker tracker and the virtual hand simulation software were implemented. The first part contains information about the OpenCV library, the classes of the marker tracker, and its GUI. In the second part of the chapter, the virtual climbing scene and the grasp algorithm are explained in detail. It also contains some information about the Unity project setup and how the VreeTracker or the HTC Vive Tracker can be integrated.

5.1 Marker Tracker

5.1.1 OpenCV

The OpenCV library is an open source library that can be used for various computer vision tasks [60]. The main focus of the library is real-time applications. It is written in C/C++, but there are also interfaces and wrappers for other programming languages available. For the VreeTracker software, the Java version of OpenCV 3.3.1 was used. The advantages of this library are that it includes implementations of many different computer vision algorithms and has a large community which continuously improves it. For example, the camera calibration algorithm by Zhang from Section 3.1.3 is implemented and can be used with a few method calls.

To use the Java wrapper of the OpenCV library, the .jar and .dll files need to be added. These files can either be downloaded from the project's website or can be compiled from the source code [60]. After loading the native library (.dll), all methods of OpenCV can be used. Most of the documentation and code examples for OpenCV are for C++ and Python, but there are also many posts in forums and blogs which describe the Java version of methods. The syntax and naming of the Java wrapper are a little different than the C++ implementation, but since the same base library is used, the code can be translated with some research.

One of the most important OpenCV modules for this project is the camera module. This module works with many different cameras and supports most of their functionality because of the widespread use of OpenCV. For example, the automatic focus can be deactivated, or the recording resolution can be chosen. One of the problems of the camera module is that the cameras are not identified by a unique ID or a serial number. A camera index (integer value) from the operating system is used to choose one of the available cameras. The problem with multiple cameras is that the camera index can be different every time the cameras are reconnected. Therefore, the GUI needs to include a camera selector and preview to allow the user to quickly configure the camera rigs. After the successful initialization of a camera module, the current frame can be retrieved with a simple method call.

In OpenCV, the data structure to save an image is called *Mat* and it represents a frame as a multidimensional array. For example, a simple grayscale image with a pixel resolution of 100×50 is a two-dimensional array with 100 columns and 50 rows. For each pixel, a grayscale value between 0 and 255 is stored. The same image in the BGR color model has three channels in which each one represents one color. Therefore, the *Mat* object consists of $100 \times 50 \times 3$ integer values from 0 to 255. This means that a conversion to a grayscale image makes the *Mat* object two thirds smaller. The optimization is not so important regarding the memory space, but because different computer vision algorithms have to check each pixel multiple times, considerable computational effort can be saved if a grayscale image can be used to get the same results.

5.1.2 Class Overview

The class diagram in Figure 5.1 contains the most important properties and methods of each class to give a good overview of the functionality and interaction between the classes. The diagram clearly shows that the *Controller* handles all communication between the classes. The *GUIController* handles all of the user inputs and gives the user the possibility to change different settings and to start or stop the tracking. The *update* method of the *Controller* handles the processing of one frame pair by calling all of the different modules one after the other. Next, each class is described briefly.

GUIController

The *GUIController* is the entry point of the software. Before the tracking starts, the user can configure the process in the GUI. The images of the connected cameras can be previewed to select the proper cameras for each camera rig. The camera-specific settings, such as contrast or brightness, can also be configured in the GUI. During the calibration, the user can see the recorded images and if the calibration pattern was detected by the software.

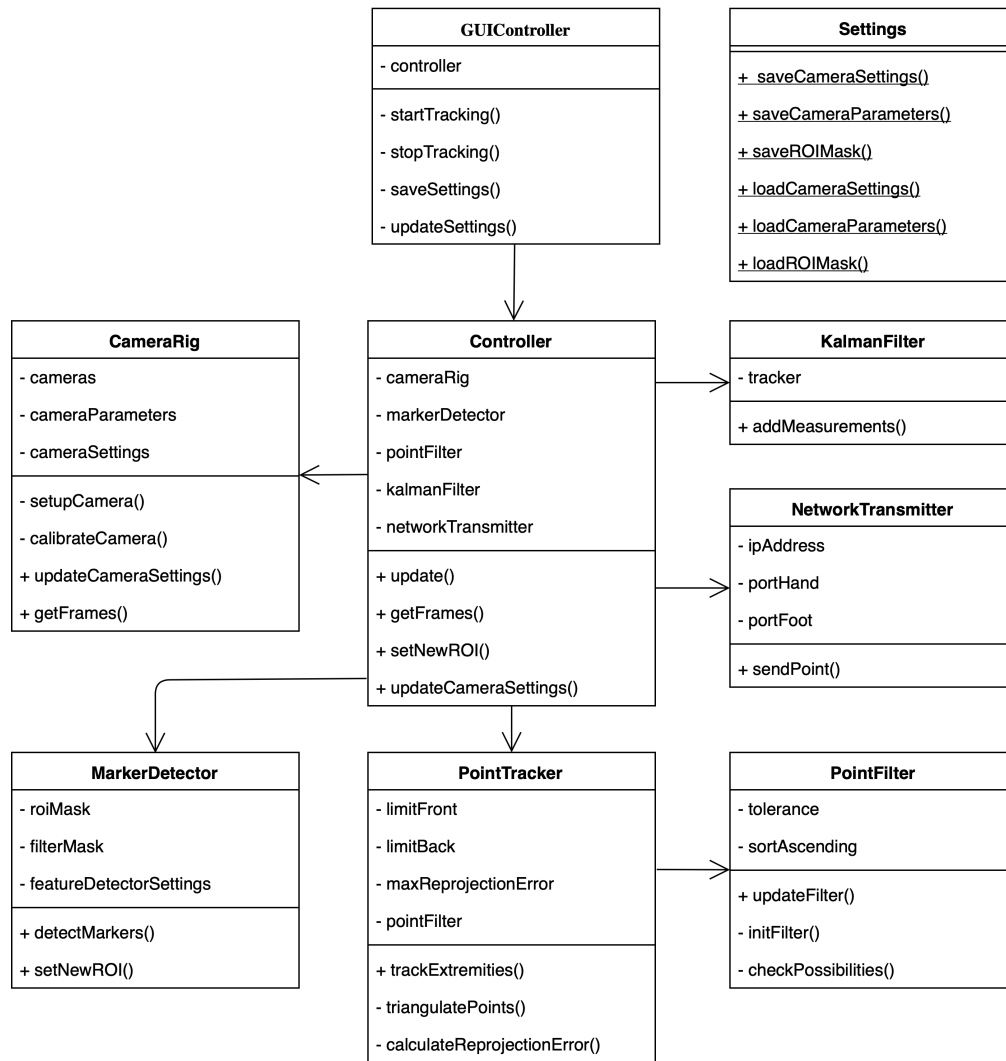


Figure 5.1: Class diagram of the tracking software

Controller

The *Controller* is the component that connects all classes. It also contains the code that checks the results of the different modules and calls the next module until the final 3D coordinates are sent by the *NetworkTransmitter*. This process is executed each time the *update* method is called. The other methods handle communication between the GUI and other classes.

Settings

The *Settings* class handles the saving and loading of the general configuration file, the camera parameters, and the region of interest mask. All settings are saved as JSON files.

The user can edit these files with any text editor, but most of the values can also be changed in the GUI. The last used configuration is automatically loaded and saved after each start of a tracking session. All methods are static and can be used by any other class.

CameraRig

The *CameraRig* handles the communication with the two USB-connected webcams. It uses OpenCV to change settings and receive frames from the cameras. Another big part of the class is the calibration logic. The users can select in the GUI if they only want to calibrate the extrinsic camera parameters or if they want to do a full calibration including the intrinsic camera parameters. For the full calibration, 15 images with the calibration pattern are needed, while the extrinsic calibration only needs one image where the calibration pattern is visible. After the start of the calibration, the software checks the camera images repeatedly for a few seconds for the calibration pattern. If the pattern was detected, the frames and the locations of the significant points of the pattern are saved. After collecting the necessary number of images, the calibration algorithm of OpenCV is used to do the actual calibration. The results of the calculations are the camera parameters, which are saved in a JSON file, so the calibration does not have to be done before each tracking session. After a successful calibration or if already saved camera parameters can be used, the *getFrames* method returns the last recorded camera frames.

MarkerDetector

The main task of the *MarkerDetector* is to find all white blobs in the frame pairs. Before the frames are analyzed, the interference filter mask is initialized. This is done by checking the first few frames after the start of the tracking. Only white blobs that are visible at the same location in all analyzed frames are added to the filter mask because they are probably from IR light sources that could disturb the marker tracking. This means that a visible marker which is moved (e.g. because it is worn by the climber) will be ignored and not added to the filter mask. The interference filter mask is then subtracted from each frame pair before the SimpleBlobDetector algorithm of OpenCV is used to find the blobs. Besides the filter mask, the frames are also cropped to the region of interest before the blob detection. The parameters for the SimpleBlobDetector are loaded from an XML file, which can be adjusted if needed. The result of the *detectMarkers* method are two lists of *KeyPoints*, which describe the detected blobs by their center and radius.

PointTracker

This class is responsible for the calculation of the markers' locations in 3D space. As mentioned in Section 4.2.3, the *PointTracker* only uses the two points closer to the edge because usually all four markers are visible on each camera. After picking these two points for each frame, a point in 3D space is calculated for each combination. Therefore,

four 3D points are calculated. Most of the time, two points will be completely wrong and meters away from the climbing wall because the coordinates for the triangulation are from two different markers. These calculations are still necessary because we cannot know which two blobs represent the same marker. The obviously wrong results are filtered out by checking if their position is within a configurable area in front of the climbing wall. Besides this check, the reprojection error for each point needs to be below a certain limit. The reprojection error is calculated by reprojecting the 3D point to the 2D space of each camera. The Euclidean distance between the reprojected point and the detected blob's point is the reprojection error. Therefore, we get a reprojection error value for each camera. If one of them is over a configurable limit, the point is discarded. All remaining points are then passed on to the *PointFilter*. The final return value of the *PointTracker* is the return value of the *PointFilter*.

PointFilter

The purpose of the *PointFilter* is to filter out position calculations that seem unlikely. This is done by comparing the just-calculated points and the last positions of the markers. First, the matching probability of each point with each marker is checked by calculating the distance between the points. The distance of each coordinate (e.g. x, y and z) needs to be inside a configurable tolerance. For all points inside this tolerance, the absolute distance to the last position is calculated. Then, the position of each marker is set to the closest point, as long as this point is not the closest for both markers. In that case, the position of the marker that is further away is updated to the next possible point, if there is any inside the tolerance. Before the updated positions are returned, the *PointFilter* checks if the climber's hand position is lower than the foot position. If that is the case, the two positions are swapped because the user target group are unexperienced climbers and, hence, this scenario is very unlikely. The new positions are saved and returned to the *PointTracker*.

KalmanFilter

The general functionality of a Kalman filter was already explained in Section 3.1.5. The basis for the *KalmanFilter* class is an extension of the OpenCV Kalman filter by Kim Son [55]. The code and configuration of the Kalman filter was adapted for this project. All valid positions from the *PointFilter* are added as measurements to the Kalman filter. Then, the results from the Kalman filter are used as the current positions. These results can be different from the measurements because of the functionality of the Kalman filter. The effect of this is that the tracked movements become smoother, which improves the VR experience. Between the visual tracking of two frame pairs, the *KalmanFilter* is used to predict a position. This is necessary because the maximum frame rate of the cameras are 30 frames per second, but the VR application runs at 60 frames per second. Additionally, if a position of an extremity could not be calculated (e.g. because of an occlusion), a prediction from the Kalman filter is used.

NetworkTransmitter

The NetworkTransmitter is a simple class that sends the position data of the extremities to a configured IP address. The ports are also configurable and are different for the hand and the foot. The UDP packets are strings which start with "hand" or "foot" and contain the x, y, and z coordinates separated with a colon.

5.1.3 GUI

In this section, the different parts of the GUI are explained. The top part of the GUI shows the currently recorded images of the webcams. Underneath the camera previews, the user can change different settings and start the marker tracking. Figure 5.2 shows a screenshot of the GUI with camera settings which are usually only used for the calibration.

Camera Preview

The camera preview always shows the frames with the camera settings of the current mode. As mentioned above, Figure 5.2 was created during the calibration phase, while Figure 5.3 shows a climber in a similar position in the tracking mode. The best-case scenario during tracking are two black images with some white spots, which are the IR markers of the climber. Usually, the camera preview is only used during the calibration and to check if the camera settings are correct. Then, the camera preview can be deactivated to avoid unnecessary CPU usage. As mentioned earlier, the cameras are rotated inwards, hence the frames in the GUI are also rotated 90 degrees to avoid confusion. The two white spots at the climber's hands in Figure 5.2 are the IR markers. The used camera settings (e.g. exposure, contrast and gain) are necessary to make the calibration pattern visible during the calibration phase.

The blue rectangle in the two images represents the region of interest that was set by the user. Only the parts inside the rectangle are considered during the tracking. If the "Set new ROI" option is selected, a new rectangle can be chosen by marking the four corners with clicks on the camera preview. Under the camera images, the camera index can be chosen. The camera index is automatically assigned by the operating system and can be different every time multiple cameras are connected to the PC. Therefore, the user can try different indices to find the two correct cameras for the camera rig. The selected camera indices are saved after each start, but as stated before, this procedure does not guarantee the right mapping after the cameras are reconnected.

Tracking Settings

The settings in the bottom left of the GUI are the tracking settings. With the first dropdown selection, the left or right camera rig can be chosen. In consequence, the different options are loaded from the configuration file. The selection also determines in which direction the preview image is rotated, because the cameras of the left camera rig are rotated 90 degrees clockwise while the right camera rig's cameras are rotated



Figure 5.2: GUI with typical calibration camera settings

counterclockwise. If the recalibration option is checked, the type of calibration can be chosen with the dropdown selection next to it. As mentioned in the previous section, a new region of interest can be selected if the next checkbox is ticked. The last option helps to check if the software correctly tracks the IR markers and how accurate the position tracking was compared to the white spots from the markers. The currently calculated location of the hand is shown as a green circle while the foot is represented by a blue circle.

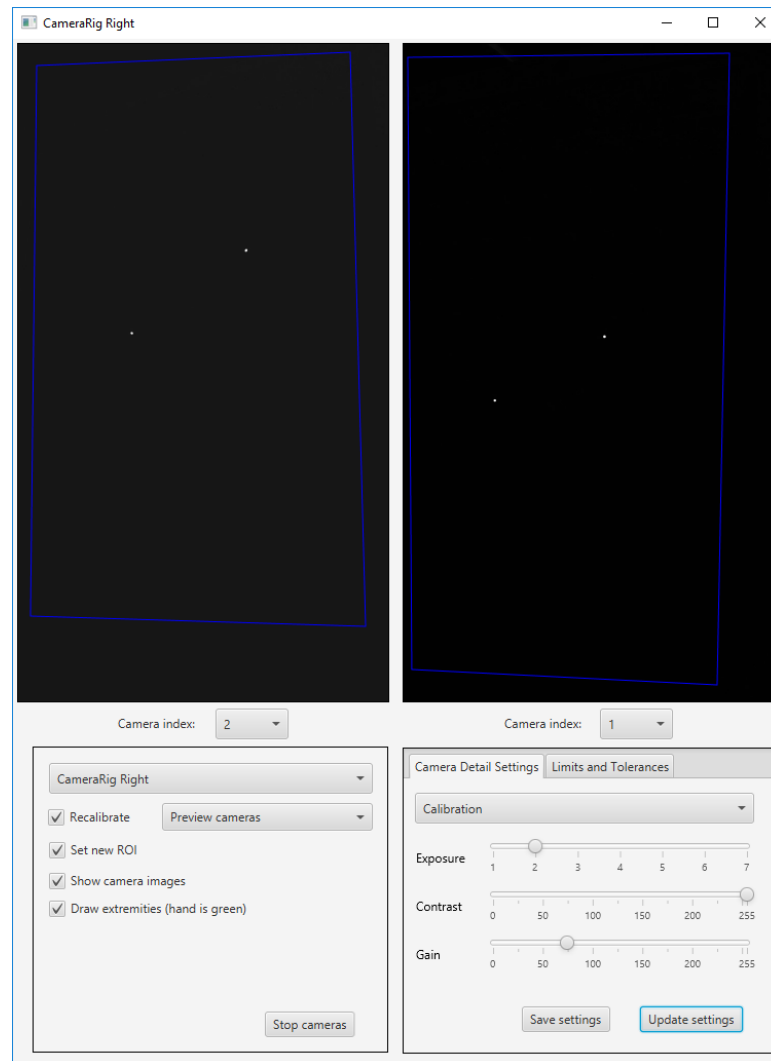


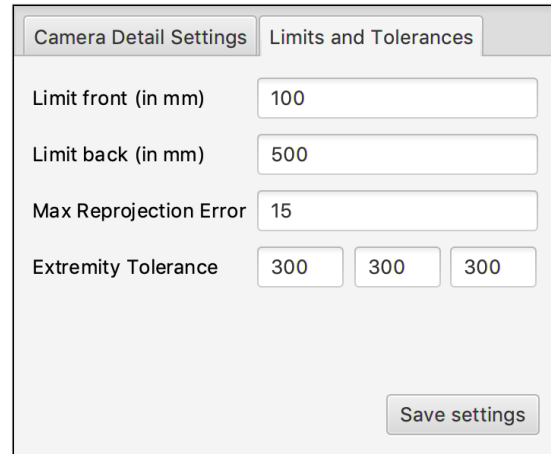
Figure 5.3: GUI with two visible IR markers

Camera Settings

The first tab of the settings in the bottom right is the camera settings. With the three sliders, the exposure, contrast, and gain of the cameras can be set. The settings can and should be different for calibration and tracking. With the dropdown selection, the different settings for each mode can be chosen. Some example settings for the calibration and tracking mode can be seen in Figure 5.2 and Figure 5.3. The sliders allow the user to quickly adapt the camera settings to the current lightning conditions. With the "update settings" button, the settings can be changed during the tracking. With the other button, the settings are saved in the configuration file and are loaded again at the next start.

Limits and Tolerances

On the second tab, the limits and tolerances of the tracking algorithm can be configured. Figure 5.4 shows the tab with some example values. The first two values define the area in which every valid 3D point has to lie. The origin of this area is the top left corner of the calibration pattern. Therefore, a valid point can be in front of or behind this point. The maximum reprojection error is another condition for valid points. The extremity tolerance is the maximum distance between the new and last position of a marker.



Setting	Value
Limit front (in mm)	100
Limit back (in mm)	500
Max Reprojection Error	15
Extremity Tolerance	300, 300, 300

Figure 5.4: Settings of limits and tolerances in the GUI

5.1.4 Calibration

The VreeTracker software can do a full calibration of all camera parameters or only a partial calibration to recalibrate the camera rig's location. For the full calibration, the algorithm from Zhang is used [71]. Details of this algorithm were already explained in Section 3.1.3. The OpenCV implementation of the algorithm is used for this project. A 8×5 chessboard was chosen as the calibration pattern. Each square of the pattern must have a side length of 10 cm. The pattern itself does not reflect IR light very well, so the camera settings need to be adjusted to make it visible during the calibration mode. The results are grayscale images such as in Figure 5.5. While the software is in calibration mode, it checks the camera frames every few seconds if it can find the calibration pattern. After finding 15 images with the pattern for each camera, the calibration algorithm is started. To get the best results, the calibration pattern should be moved and rotated in front of the VreeClimber as seen in Figure 5.5.

The calibration of the intrinsic camera parameters is independent for each camera, so it is not necessary that the images are taken at the same time for both cameras. For the extrinsic calibration one image of the calibration pattern is enough, but the pattern cannot be moved until both camera rigs are calibrated. This is necessary to guarantee that the point of origin of the 3D coordination system is the same for both camera

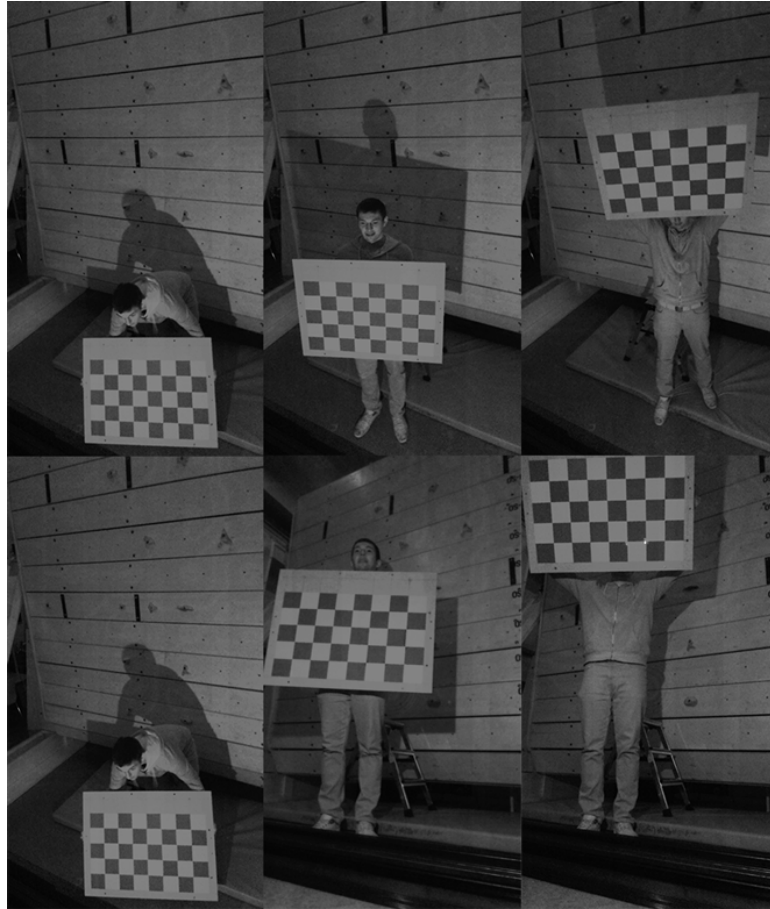


Figure 5.5: Images recorded during full camera calibration

rigs. Therefore, the pattern should be placed in front of the VreeClimber as shown in Figure 5.6. For the extrinsic calibration, the OpenCV method *solvePnP* is used [46]. This method estimates the camera's pose based on the intrinsic camera parameters from a previous full calibration and the identified image points of the calibration pattern. After a successful calibration, the software automatically changes the camera settings according to the tracking mode and the tracking of the IR markers is started.

5.1.5 Tracking

The theoretical background of the tracking and the different software modules were already explained in detail. In this section, the full process and the relations between the software modules are described briefly. The Controller receives 30 frame pairs per second from the camera rig. It passes the frames to the blob detector, which uses the first few frames to calculate the IR interference filter. After the initialization, the filter is applied on all following frames. The frames are also cropped to the selected region of



Figure 5.6: Calibration pattern in front of VreeClimber wall

interest. Then, the blobs are detected and the results are returned to the controller. In the PointTracker module, the 3D points are calculated with triangulation, and all invalid points are filtered out. The remaining points are then passed on to the PointFilter, which updates the extremities' positions if the new points are close enough to their last position. The updated positions for the hand and foot are then returned to the Controller. In the next step, the Kalman filter is updated with the new positions and the returned results are sent to the VR application with the NetworkTransmitter class. If there was any problem, for example, that the points are not within the configured tolerances, the Kalman filter is used to estimate the current positions. These estimations are also used between two camera frame pairs to increase the position output to 60 updates per second, which makes the movements appear smoother in the VR application.

5.2 Virtual Hand Simulation

In this part of the chapter, the different parts of the VR application and the grasp algorithm are described. It starts with the setup of the Vive and the Unity project. Then, the virtual climbing scene, the hand model and the marker tracker integration are explained. Finally, the implementation of the grasp algorithm and its challenges are described.

5.2.1 HTC Vive Setup

An HTC Vive with two controllers and two lighthouses was used for the virtual reality application. Some information about the Vive was already introduced in Section 2.1.3. After setting up the hardware as described by the user manual, the SteamVR software has to be downloaded [65]. The install wizard guides the user through multiple steps including the room setup. During the room setup, the size of the available space and the general direction of the virtual room are configured.



Figure 5.7: Interactive room setup in SteamVR [33]

For this project, it is important to set up the room in the direction of the climbing wall as seen in Figure 5.7. The boundary in front of the climbing wall should be parallel and as close as possible to the VreeClimber. This makes the calibration of the climbing wall's position in the VR application easier. The result of the room setup is called the play area. It defines the area in which the user can move freely. If the user comes close to a border of the play area, a virtual wall is shown to prevent the user from stumbling over objects or walking into walls.

5.2.2 Unity Project Setup

As mentioned earlier, Unity is used to create and run the VR application. Unity provides a very good documentation and also many video tutorials to quickly learn the basics [64]. The code is written in C# and can be run on any Windows PC with up-to-date hardware. The *SteamVR SDK* for Unity was added to the project to integrate the HTC Vive [63]. Requirements to use the SteamVR SDK are a Unity account and the setup of SteamVR on the PC. The SteamVR SDK can be downloaded from the Unity Asset Store [63]. After importing all the necessary assets, the prefabs "[CameraRig]" and "[SteamVR]" have to

be added to the scene to automatically integrate the HTC Vive HMD and controllers when they are connected to the PC.

5.2.3 Virtual Climbing Scene

A simplified prototype of the VreeClimber climbing wall was created in Unity to test the hand simulation software. Figure 5.8 shows the prototype of the virtual climbing scene. This virtual climbing wall is placed at the edge of the play area and the different 3D-scanned climbing holds are placed at the same locations as on the real VreeClimber. The climbing holds are scaled and rotated accordingly. Besides the visual adjustments, a physics collider and the same tag are added to each climbing hold object in Unity. These are necessary to detect collisions with the virtual hands and for the grasp animation script. As mentioned earlier, the origin of the coordinate system of the tracked positions of extremities from VreeTracker is always the top left corner of the calibration pattern. This means that this point can be different between two calibrations. Therefore, the exact location of the climbing wall needs to be calibrated to ensure that the virtual climbing holds match the location of their real counterparts. For this calibration, an adapted algorithm from a similar virtual climbing application was used [28].

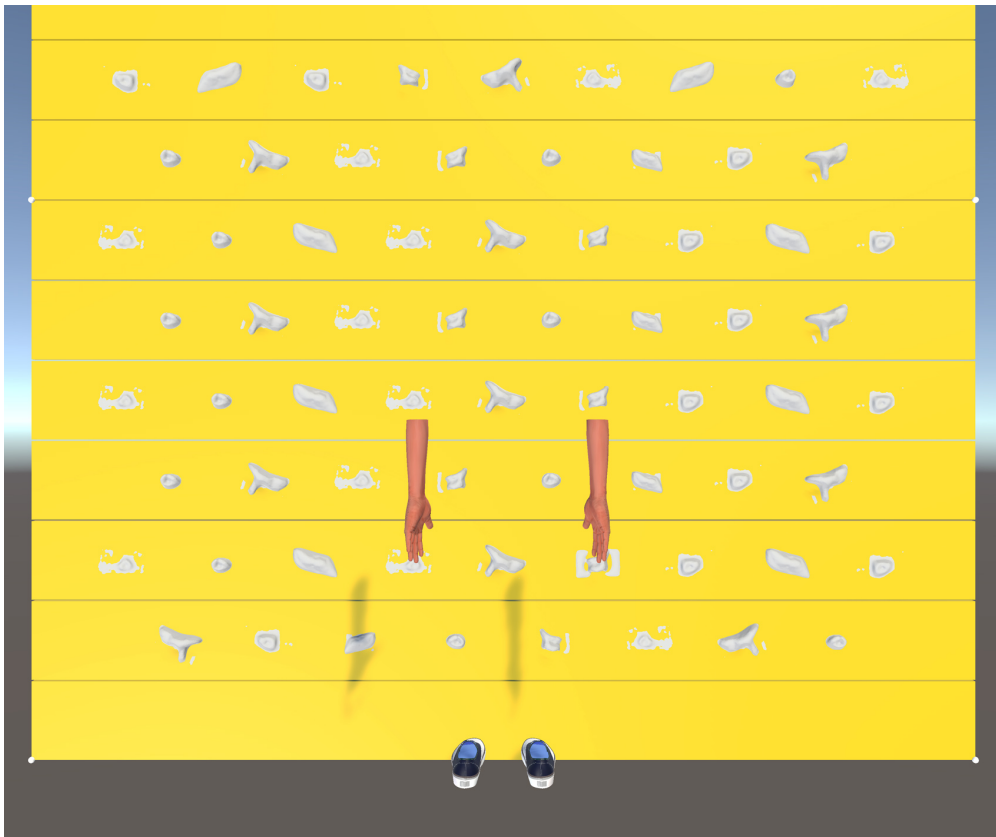


Figure 5.8: The virtual climbing scene with shoe and hand models

The basic idea of the algorithm is to choose four significant points of the VreeClimber, such as unique climbing holds. These points are then located on the VreeClimber with a Vive controller. After acquiring the coordinates of these points in the virtual world, the algorithm tries to match them with the predefined points on the virtual climbing wall as precisely as possible. The virtual climbing wall is then moved and rotated accordingly, so that the locations of the virtual climbing holds match their real counterpart. The new location of the climbing wall is saved in a file and is automatically loaded when the application is restarted.

After the wall calibration, the actual virtual climbing app can be run with a climber who wears the Vive HMD and four VreeTracker markers at wrists and ankles. To initialize the two VreeTracker camera rigs, all four markers need to be visible for all cameras. After a successful initiation, the climber should be able to see and move the virtual hands and feet. Figure 5.9 shows a climber in front of the climbing wall and the point of view in the virtual world. The virtual hand grasps the climbing hold because the small distance between hand and object triggers the grasp animation. If the climber moves the hand away from the climbing hold, the virtual hand will follow the climber's movement and open up.



(a) Climber with VR headset on the climbing wall



(b) The climber's view in the virtual climbing scene

Figure 5.9: Picture of a climber in front of the VreeClimber and the point of view rendering

5.2.4 Hand Model

Current VR systems such as the HTC Vive or Oculus Rift do not track any body parts of the user without additional hardware. They can only track the HMD and the controllers. Most applications use the position and orientation of the controllers to show virtual hands. For a virtual climbing application, this is not possible because the users needs their hands to climb. Therefore, the VreeTracker or another tracking system is required

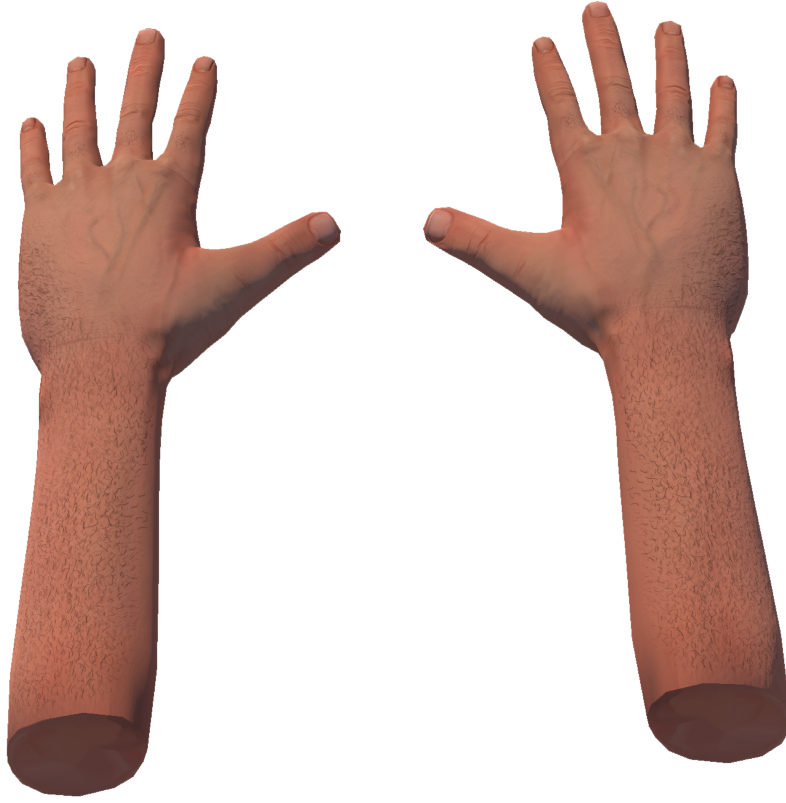


Figure 5.10: Used hand model from Unity asset store [62]

to get the real-time position and orientation of the markers, which are placed at the wrists and ankles of the climber. The challenge now is to add virtual hands to the application which look realistic enough and predict how the user will most likely grab the different climbing holds.

To save time, a professionally modeled hand model from the Unity Asset Store was employed [62]. Figure 5.10 shows the model of the left hand on the left and a mirrored copy on the right. The hand model supports some animations out of the box, but they are not useful for our project because there is no way to stop the animation for individual fingers which touch the climbing wall or a climbing hold. Such a feature is necessary to improve the user experience of the virtual climbing application. Therefore, new animation system was developed that can control each finger independently. Additionally, the animation can be stopped if it touches a climbing hold.

In Unity, it is possible to add a bone-like structure to a 3D model. All visible areas of the model are connected with one or more bones. If such a bone is moved, the connected



Figure 5.11: Bone structure of the virtual hand

parts of the model move accordingly. The bone structure of the hand model is shown in Figure 5.11 as a gray overlay. Each finger has three bones and therefore, each part can be moved individually. In Figure 5.11, the bones of the index finger point downwards and so does the index finger of the 3D model. Each bone can be moved with a simple command from Unity. Therefore, it is possible to write an animation script which can close and open a hand. To detect collisions between the virtual hand and the climbing wall, physics colliders were used (as mentioned in Section 4.3.1).

5.2.5 VreeTracker Integration

To read the UDP packets from the VreeTracker software, a script was added to the extremity objects in Unity. Each instance of the script is listening on a different port to receive the position and orientation data. The packets are UTF8 encoded strings with multiple float values separated by semicolons. Strings with 3 values are location coordinates and strings with 4 values are orientation data from the IR markers. The script parses the values and applies them to the corresponding object. These updates are performed in the background and applied whenever new data is received.

5.2.6 HTC Vive Tracker Integration

During the development of the virtual hand simulation software, HTC released their own trackers for the Vive system. The so-called *Vive Tracker* is fully integrated in the existing Vive system and can be attached to various objects or body parts [19]. This feature makes it an alternative for the VreeTracker, because then the additional hardware and calibration are not needed anymore. The Vive VR system is already a requirement for the virtual climbing application and the Vive trackers are automatically tracked if they are added to SteamVR and are inside the play area. Hence, the integration of the Vive Tracker to the project was simple and did not take much time. The Vive Trackers are represented as objects in Unity, which makes it possible to add the virtual hand models as children to these objects. In Unity, the position of children is related to the parent object's position. Therefore, when the parent object is moved, the children are also moved automatically. No other modifications had to be made to integrate the Vive Trackers.



Figure 5.12: Vive tracker attached to the back of the hand

After testing the movement tracking with wrist bands, such as used for the VreeTracker markers, an improvised strap for the Vive Tracker showed better results for the virtual hand simulation. Figure 5.12 shows the improvised strap that places the tracker at the back of the hand. During climbing and grasping, the angle and rotation of the wrist can be vary significantly. Therefore, the precision of the virtual hand simulation is immediately improved if we do not have to make assumptions about the current wrist orientation. The position change of the tracker is also easy to adjust in Unity, because only the hand's position relative to the tracker object needs to be changed.

5.2.7 Grasp Algorithm

The grasp algorithm controls the animation of the virtual hand when it comes close to a climbing hold. The goal is to make the grasp movements look as real as possible. Because we do not have any exact location data for the different fingers, we need to make certain assumptions. The first assumption is that when the climber's hand comes close to a climbing hold, it most likely will grasp it. Secondly, the target group for the VreeClimber app are beginners. Therefore, we assume that the climbers only use basic grasps. The third assumption is that it is not necessary that the virtual grasp looks exactly like the real grasp of the climber. The climber does not see the real hands; hence it is also harder for them to notice small differences. Additionally, the software can be added to other virtual climbing scenarios, such as climbing on Mars, which will also distract the climber. Eventually, the correct location of the virtual climbing holds is more important than the exact look of the grasp animation, therefore the virtual hands should just help the climber with the coordination of their hands.

The grasp algorithm is a script, which is attached to the hand model objects in Unity. At the start of the application, the setup method of the script is called. During the setup the rotation values of all bones of the hand model are saved as reference for the open hand pose. These values are also saved for the final pose of the closed hand models. With this information, it is possible to calculate the differences between the open and closed poses for each bone. After the setup, the grasp algorithm is called at each frame for both virtual hands. The algorithm can be separated into the following steps, which are called one after the other.

1. **Calculate Closing Value:** First, the nearest climbing hold is detected. With that information, the algorithm calculates how far each finger should be closed. This part was already explained in Section 4.3.3.
2. **Determine the Grasp Pose:** The pose is determined based on the climbing hold and orientation of the climber's arm. The details can be found in Section 4.3.4.
3. **Hand Movement Detection:** When the hand was moved, all fingers are closing/opening again until they reach the final pose or collide with the climbing hold.
4. **Smooth Animation:** Move the different parts of the hand in small steps until the calculated position is reached or the animation is stopped because of a collision.

In addition to these four steps, there are also two methods that are automatically called by Unity when a collision with a climbing hold or the climbing wall occurs. In the following sections the central parts of the algorithm are described in greater detail.

Closed Hand Models

The general process of how the grasp pose is selected was already explained in Section 4.3.4. The different grasp poses are represented as modified copies of the virtual hand model. Figure 5.13 shows the four currently supported poses. The poses 1, 2 are for small and big climbing holds when the climber's hand is horizontal, while the models 3, 4 are used when the hand is vertical. The difference of the poses for small and big climbing holds can be seen in model 3 and 4. For smaller climbing holds, only the finger tips are used. For bigger climbing holds, all joints of the fingers are bent to grab them with the whole hand. During the setup of the grasp algorithm, the angle value of each finger joint is saved for each grasp pose. With this information, it is possible to calculate the angle values when the hand should be halfway closed, for example. To identify the different grasp poses, the objects are tagged in Unity. The four currently supported poses can be easily augmented by additional poses. For example, an existing closed hand model can be copied, and the pose can be adjusted within Unity. Then, the new object needs to be tagged accordingly and an additional distinction has to be added to the grasp algorithm to define when the new pose should be used.



Figure 5.13: Closed hand models for different climbing hold sizes and hand orientations

Hand Movement Detection

The grasp animation algorithm always needs to consider that the climber could move the hands during an animation. Therefore, a grasp animation could look well, but if the hand is then moved closer to the climbing hold and the pose is not adapted, graphical glitches could occur. Generally, we have to assume that the tracking of the hand is precise enough so that the virtual hand is never moved into the virtual climbing wall. Movements of the hand are detected by comparing the current and last position of the virtual hand. If a movement was detected, previous collisions are ignored and the animation of each finger is started again from their current position. If a finger collides again or is still colliding, the finger is automatically opened until it does not collide anymore. This guaranties that there are no graphical glitches when the hand is moved closer to the climbing hold. The rest of the hand is closing until the fingers collide or the final position of the grasp pose is reached. The algorithm also recalculates how far the hand should be opened in case the hand is moved away from the climbing hold.

Smooth Animation

After all the calculations of the algorithm are done, we know the current and the target rotation of each bone. Since we want a realistically looking animation, we cannot just rotation the bones to the target value immediately. To smoothen the animation, only small changes are applied at each frame until the final rotation is reached. To make the algorithm more efficient, the target rotation is only recalculated if the hand was moved.

Collision Handling

As mentioned earlier, the physics engine of Unity is used to detect collisions between the virtual hand and the climbing wall. When a collision occurs, a delegate method in the script is called. The simplest implementation would be to just stop the animation for all bones of the finger, if any bone collided. But in many cases, this would not really look like the hand is completely grasping the climbing hold. Therefore, only the animation of the collided bone and the bones closer to the hand center are stopped. For example, if the fingertip collides, the animation of each bone is stopped, but if the middle part collides, the fingertip is still animated until it collides, too. If the bones closer to the center would still be rotated after the fingertip collided, the tip of the finger would be moved further into the climbing hold, which would result in graphical glitches. Unity makes it also possible to detect if two objects are still colliding at each frame. The algorithm uses this feature to open the finger until it is no longer touching the climbing hold or the climbing wall, thus helping to avoid graphical glitches.

Known Problems of the Virtual Hand Simulation

If the different calibrations were done carefully and the errors are small, the virtual hand simulation works well and is a good visual help for the climber, but of course the simulation can be improved. For example, when the hand is moved quickly back and

forth near a climbing hold, the virtual hand opens and closes fast. This behavior does not look fully natural, because usually a climber would open the hand about half way, but generally this can be seen as an edge case. Most amateur climbers will probably hold on their current position and then try to quickly grasp the next suitable climbing hold.

Another problem can be early collisions when the hand closes too fast, for example, when the fingertip collides while the hand moves over a climbing hold from underneath it. Then, the animation is stopped for the finger and it can happen that the finger moves into the climbing hold. Then, the finger opens as long as it is still colliding with the climbing hold and then starts closing again. This behavior can appear strange to the climber.

The biggest problem during testing was inaccurate tracking or a bad calibration of the climbing wall. Then the best algorithm cannot improve the user experience, because the position data of the virtual hand or the locations of the climbing holds are generally wrong. To avoid this situation, the calibration process should be repeated until the results are good enough to guarantee a good virtual climbing experience.

In the first part of the chapter, the implementation of the VreeTracker software was explained, while the second part discussed the virtual hand simulation and the VR application. In the next chapter, the precision and characteristics of the VreeTracker and the HTC Vive are compared. The performance of the grasp algorithm in some test scenarios is also evaluated.

Evaluation

In this chapter, some parts of this project are evaluated. Since the performance of the VreeTracker was already evaluated by Ludwig Steindl, his results are compared to the HTC Vive Tracker, which was not available at the time of that thesis [56]. The performance of the developed grasp algorithm is evaluated by a test scenario in which a virtual hand automatically grasps different climbing holds. Before the results are presented, the evaluation methods are explained.

6.1 Setup

6.1.1 Vive Tracker Evaluation

To compare the Vive Tracker and the VreeTracker, the tests of Ludwig Steindl were repeated with the Vive System [56]. The positional precision was evaluated by tracking a resting Vive Tracker for one minute. The calculated positions of the marker should not differ by much, but all visual tracking system produce few deviations. The maximum difference between individual positions is the so-called *jitter*. In a second test, a moving marker is tracked. Figure 6.1 shows the Vive Tracker on the test platform, which can be rotated at a consistent speed to get comparable results. The recorded data should represent a smooth circle without many outliers.

6.1.2 Virtual Hand Simulation Evaluation

The evaluation of the virtual hand simulation is difficult because there are many different factors and the most important aspect is the visual impression of the climber, which cannot be measured. The key component of the virtual hand simulation is the grasp algorithm. Therefore, the evaluation tries to find out how good the virtual hand can adapt to different climbing holds. A script is used to test the algorithm with ten small and ten large climbing holds. The virtual hand automatically grasps the grips, and

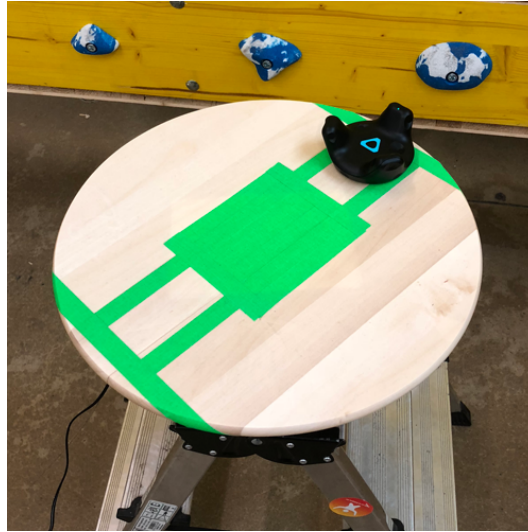


Figure 6.1: Rotating platform to evaluate precision during movement

when the final hand pose is reached, the distance between each fingertip and the grip is measured. Since Unity does not provide a method to measure the distance between a 3D point and a concave mesh, rays are sent out from each fingertip to find the minimal distance between them. The rays are sent in all directions and can only collide with the climbing hold. In Figure 6.2 the rays that first collided with the climbing hold are visualized in red. The algorithm will use two different grasp poses based on the size of the climbing hold. With this test, the general adaptability of the virtual hand and the grasp algorithm should be evaluated.



Figure 6.2: Shortest distances visualized as red rays

6.2 VreeTracker and Vive Tracker Comparison

The maximum jitter of the Vive Tracker along the horizontal, vertical and depth axes was $\pm 0.52mm$, $\pm 0.79mm$, and $\pm 0.56mm$. The measurements were made under typical conditions without an especially careful calibration or multiple runs. As mentioned in the setup section, the VreeTracker was already evaluated by Ludwig Steindl [56]. The results of the same test with the VreeTracker were $\pm 0.4mm$, $\pm 0.26mm$, and $\pm 0.75mm$, but the raw data did not seem to be evenly distributed. After some investigation Ludwig Steindl found out, that the VreeTracker returned only a few different values, which was probably caused by a rounding error [56]. Figure 6.3 shows the distribution of the measurements from the VreeTracker. The collected data contained some positions for a few hundred times. The values from the Vive Tracker are shown in Figure 6.4 and are more evenly distributed.

To compare the outcome of the second test, the results from both tracking systems, were normalized and added to the diagram shown in Figure 6.5. The blue points represent the Vive Tracker, while the VreeTracker's points are red. The circular movement is clearly visible for both tracking systems but the detail view in Figure 6.6 shows that the calculated coordinates from the Vive system are more accurate and do not contain as much jitter.

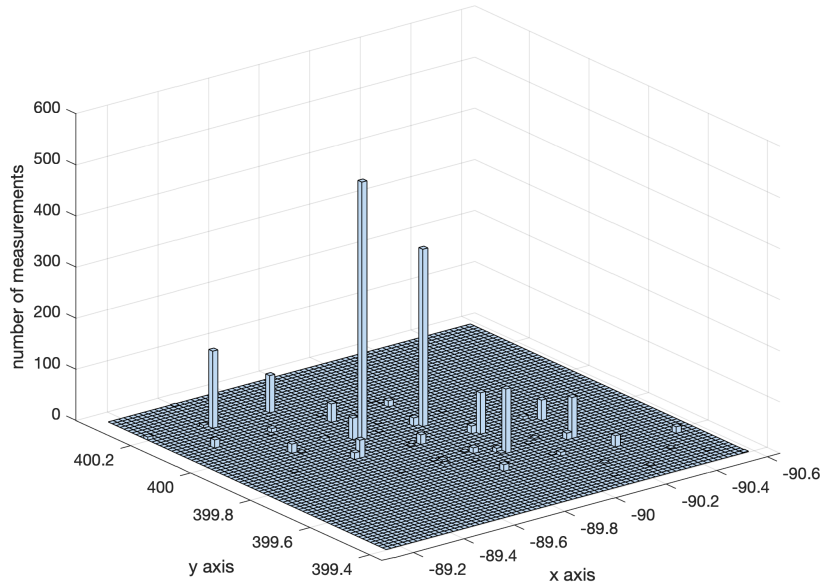


Figure 6.3: 3D histogram of VreeTracker's positional data

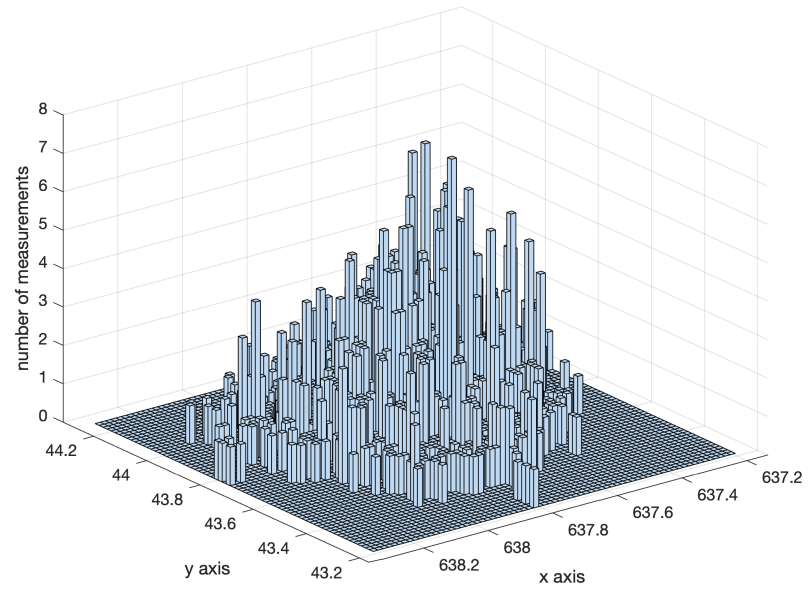


Figure 6.4: 3D histogram of Vive Tracker's positional data

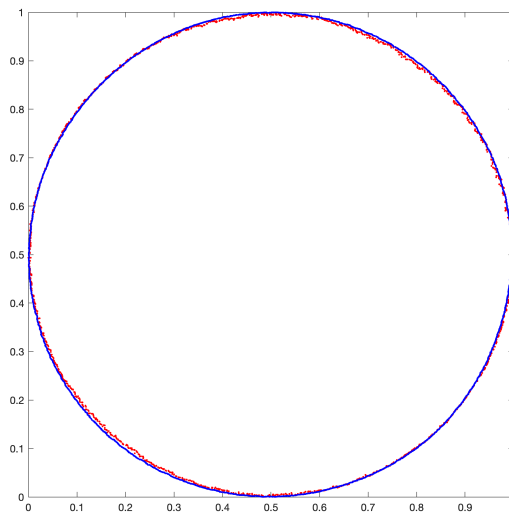


Figure 6.5: Positions of the markers during the circular movement

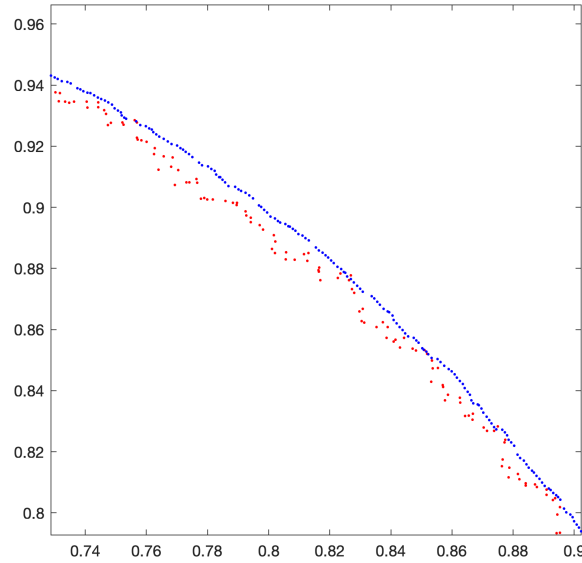


Figure 6.6: Detail view of the position data

6.3 Grasp Performance

The result of the grasp algorithm evaluation is an average distance of $13mm$ between each fingertip and the climbing hold. Figure 6.7 shows a boxplot of the data with the distance for each finger separately. As expected, the thumb and little finger did not perform as good as the other fingers with an average of $17mm$ and $19mm$ distance. The main reason is that most of the climbing holds of the VreeTracker are only as wide as the hand or smaller. Therefore, some of the climbing holds are too small to be grasped with all fingers. In the test this effect always concerns the thumb and little finger, because the centers of the hand and the climbing hold are aligned. Additionally, the virtual hand is always vertical during the test, hence the thumb sometimes could not even reach the climbing holds.

Interestingly the average distance of $13mm$ is still the same after the data is normalized by the climbing hold size. The differences can be seen in Figure 6.8. The middle and ring finger perform better for smaller grips because the different grasp pose allows their fingertips to often collide with the top side of the climbing hold. The average distance of the little finger is $23mm$ for small and $18mm$ for big climbing holds. This can also be explained with the width of the climbing hold. Figure 6.9a shows such a grasp where the thumb and the little finger do not reach the small climbing hold. Another typical error can occur when a finger collides with the climbing wall during the grasp movement. The animation of the finger is stopped and as seen in Figure 6.9b, the finger does not really grasp the climbing hold.

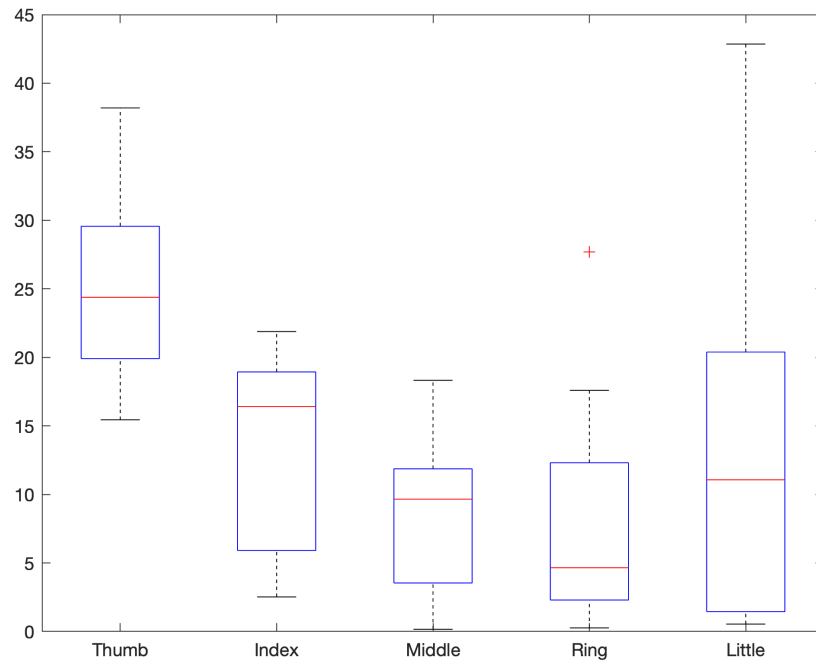


Figure 6.7: Boxplot of distance (mm) between each finger and the climbing hold

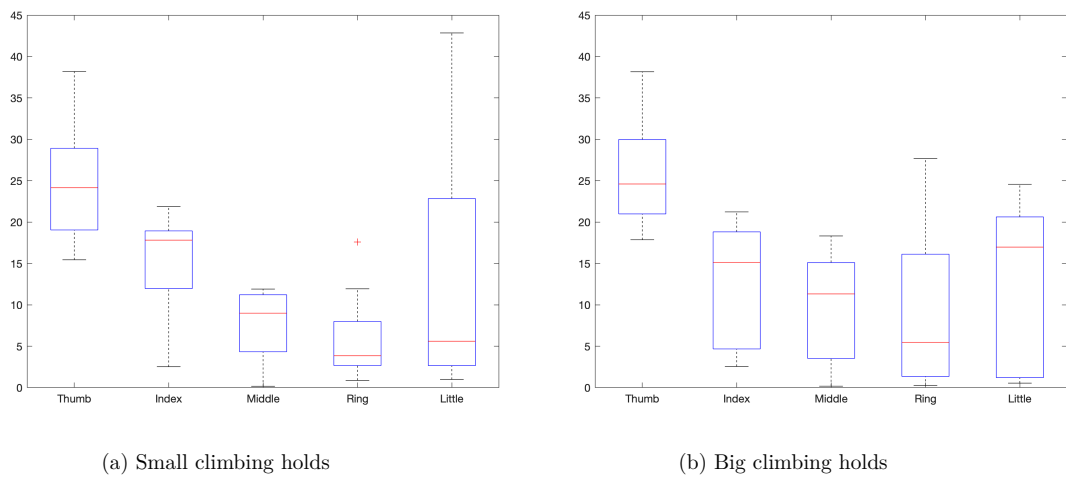


Figure 6.8: Boxplots of distances (mm) split by climbing hold size

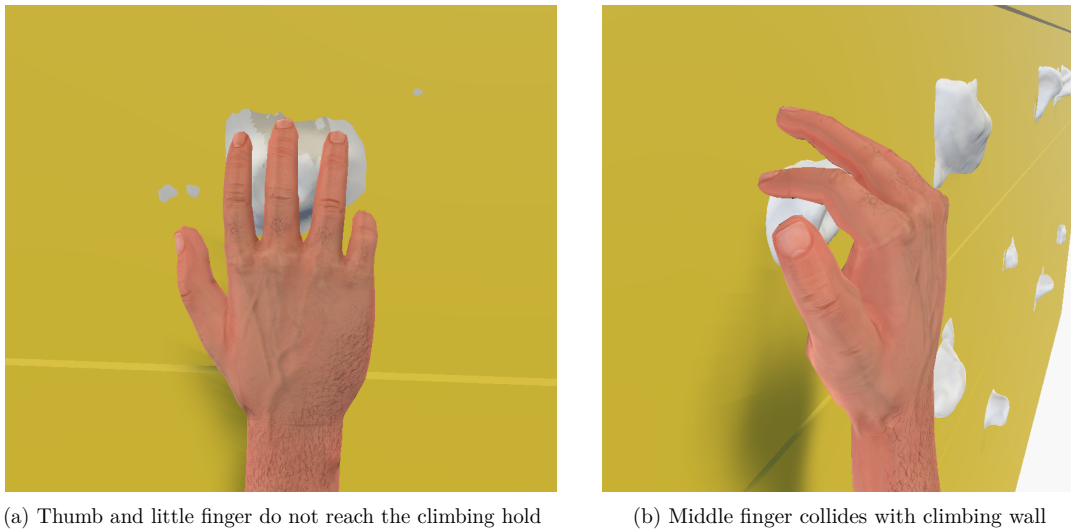
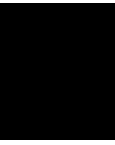


Figure 6.9: Typical errors during virtual hand simulation

6.4 Discussion

The comparison of the VreeTracker and Vive Tracker showed, that the calculated positions of the Vive Tracker are more evenly distributed and its moving object tracking performance is also superior. Beside the better precision, the easy integration in the already used Vive system is another good reason to use the Vive Tracker for the VreeClimber project. The results from the grasp algorithm evaluation showed, that the virtual hand is able to adapt to climbing holds of various sizes and forms. However, the data also showed, that the performance of the thumb and the little finger could be improved. One possibility would be to consider the width of the climbing hold to introduce a new grasp pose, in which the fingers are more closed towards the climbing hold's center.



Conclusion and Future Work

In the course of this thesis, the software of the VreeTracker was rewritten and an algorithm to simulate the hand movements in the VR world was created [56]. The original software of the VreeTracker was written for Matlab and is now replaced with a Java implementation that uses the OpenCV library. Therefore, Matlab is no longer required and the software can be run on any PC. The virtual hand simulation software was created for the game engine Unity and can be easily integrated in other Unity projects. The main component of the software is the grasp algorithm. It automatically opens and closes the virtual hand depending on the distance to the climbing hold. The algorithm also supports different grasp poses, which are selected based on the size of the climbing hold and the orientation of the hand.

During the creation of this thesis, HTC released the Vive Tracker, which has very similar features as the VreeTracker. The integration of the Vive Tracker in the already-used Vive system was straightforward, and with a few modifications, the tracking also worked with the virtual hand simulation. A substantial benefit of this change is that the camera rigs and their calibration are no longer necessary. The evaluation showed that the overall performance of the Vive Tracker is better than the VreeTracker's performance, and therefore it makes sense to only use the Vive system in future projects of the VreeClimber.

The virtual hand simulation works well for simple use cases, such as when the hand approaches the climbing hold with a constant movement. However, to grasp smaller climbing holds the tracking system and virtual climbing wall need to be calibrated accurately to ensure good results during climbing. If the calibration was not performed carefully, graphical glitches can occur, and the user immediately notices inaccuracies.

During the development of the grasp algorithm, some experimenting with the placement of the hand tracker was done. An improvised strap that placed it on the back of the

hand showed the most promising results. Hence, the purchase or development of a more suitable strap could further improve the user experience [6].

The evaluation of the grasp algorithm showed that the performance of the thumb and little finger can be improved. One possibility would be to add additional grasp poses to better adapt to the different sizes and forms of the climbing holds. Another area for potential improvements is the virtual hand model. Currently, the size of the climber's hand is not considered, which could be a problem for users with very small or big hands. Additionally, the hand's skin could be customized for different climbing scenarios.

List of Figures

1.1	VreeClimber with climbing holds from both sides	2
2.1	Camera with IR LEDs and a passive marker [17]	6
2.2	Optical and glove-based finger tracking	7
2.3	Headset differences because of tracking method	8
2.4	Grasping a virtual object with a finger tracking glove [3]	10
2.5	Focus point and example grasps of a robotic hand	11
3.1	Electromagnetic spectrum [11]	14
3.2	The pinhole camera model [57]	14
3.3	Different types of distortion compared to no distortion	16
3.4	Checkerboard pattern with feature points overlay	18
3.5	Triangulation in epipolar geometry [16]	19
3.6	RGB image and each color channel (red, green and blue)	19
3.7	RGB image and each color channel (Y', Cb, Cr)	20
3.8	Example image with circular blobs in white and different shades of gray .	22
3.9	Cycle of a discrete Kalman filter [70]	23
3.10	Bones and joints of the human hand [66]	25
3.11	Six basic types of prehension [59]	26
4.1	Setup of the stereo cameras in front of the VreeClimber	32
4.2	Two VreeTracker marker prototypes with straps for the wrist and ankle .	33
4.3	Logitech HD Pro Webcam C920 with a suppression filter	34
4.4	Software modules of the tracking software	36
4.5	Overview of the components used by the VR application	39
4.6	Hand model with physics colliders	40
4.7	Image and 3D scan of the same climbing hold	41
4.8	Examples of different grasp poses	43
5.1	Class diagram of the tracking software	47
5.2	GUI with typical calibration camera settings	51
5.3	GUI with two visible IR markers	52
5.4	Settings of limits and tolerances in the GUI	53
5.5	Images recorded during full camera calibration	54

5.6	Calibration pattern in front of VreeClimber wall	55
5.7	Interactive room setup in SteamVR [33]	56
5.8	The virtual climbing scene with shoe and hand models	57
5.9	Picture of a climber in front of the VreeClimber and the point of view rendering	58
5.10	Used hand model from Unity asset store [62]	59
5.11	Bone structure of the virtual hand	60
5.12	Vive tracker attached to the back of the hand	61
5.13	Closed hand models for different climbing hold sizes and hand orientations	63
6.1	Rotating platform to evaluate precision during movement	68
6.2	Shortest distances visualized as red rays	68
6.3	3D histogram of VreeTracker's positional data	69
6.4	3D histogram of Vive Tracker's positional data	70
6.5	Positions of the markers during the circular movement	70
6.6	Detail view of the position data	71
6.7	Boxplot of distance (mm) between each finger and the climbing hold . .	72
6.8	Boxplots of distances (mm) split by climbing hold size	72
6.9	Typical errors during virtual hand simulation	73

List of Tables

3.1	Relevant parameters for the SimpleBlobDetector [47]	21
3.2	Discrete Kalman filter time update equations [70]	24
3.3	Discrete Kalman filter measurement update equations [70]	24
3.4	Thumb and fingers range of motion and average position during a grasp of an object [4]	26
4.1	Requirements for the VreeTracker and virtual hand simulation software . .	31

Bibliography

- [1] M. Borchardt, K. Hartmann, R. Leymann, and S. Schlesinger. *Ersatzglieder und Arbeitshilfen: Für Kriegsbeschädigte und Unfallverletzte*. Springer Berlin Heidelberg, 1919.
- [2] A. Borrego, J. Latorre, M. Alcañiz, and R. Lloréns. Comparison of oculus rift and htc vive: Feasibility for virtual reality-based exploration, navigation, exergaming, and rehabilitation. *Games for health journal*, 7 3:151–156, 2018.
- [3] C. W. Borst and A. P. Indugula. Realistic virtual grasping. pages 91–98, 320, 2005.
- [4] M. C. Hume, H. Gellman, H. Mckellop, and R. H. Brumfield. Functional range of motion of the joints of the hand. *The Journal of Hand Surgery*, 15:240–243, 04 1990.
- [5] A. Cologan. How does the leap motion controller work? <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>. Accessed: 2018-08-02.
- [6] R. Corp. Trackstrap hand for htc vive tracker in vr and motion capture. <https://rebuffreality.com/products/trackstrap-hands>. Accessed: 2019-04-24.
- [7] P. Dempsey. The teardown: Htc vive vr headset. *Engineering Technology*, 11(7-8):80–81, Aug 2016.
- [8] G. ElKoura and K. Singh. Handrix: Animating the human hand. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 110–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.
- [9] C.-S. Fahn and H. Sun. Development of a data glove with reducing sensors based on magnetic induction. *IEEE Transactions on Industrial Electronics*, 52(2):585–594, April 2005.
- [10] J. Farley. A short guide to color models - sitepoint. <https://www.sitepoint.com/a-short-guide-to-color-models/>. Accessed: 2018-12-07.

- [11] A. I. for Artificial Intelligence. the electromagnetic spectrum (lesson 0753) - tqa explorer. http://data.allenai.org/tqa/the_electromagnetic_spectrum_L_0753/. Accessed: 2018-09-04.
- [12] B. Gabor. Opencv: Camera calibration with opencv. https://docs.opencv.org/3.4.3/d4/d94/tutorial_camera_calibration.html. Accessed: 2018-10-07.
- [13] Google. Google cardboard - google vr. <https://vr.google.com/cardboard/>. Accessed: 2018-07-18.
- [14] M. S. Grewal and A. P. Andrews. Applications of kalman filtering in aerospace 1960 to the present [historical perspectives]. *IEEE Control Systems Magazine*, 30(3):69–78, June 2010.
- [15] V. GUPTA. Color spaces in opencv (c++ / python) | learn opencv. <https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/>. Accessed: 2018-12-28.
- [16] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.
- [17] G. Hillebrand, M. Bauer, K. Achatz, and G. Klinker. Inverse kinematic infrared optical finger tracking. 2006.
- [18] A. Hornberg. *Handbook of Machine Vision*. Wiley-VCH, 2006.
- [19] HTC. Vive | vive tracker. <https://www.vive.com/us/vive-tracker/>. Accessed: 2018-08-02.
- [20] HTC. Vive | vive virtual reality system. <https://www.vive.com/us/product/vive-virtual-reality-system/>. Accessed: 2018-07-18.
- [21] iFixit. Htc vive teardown - ifixit. <https://www.ifixit.com/Teardown/HTC+Vive+Teardown/62213>. Accessed: 2018-08-14.
- [22] iFixit. Oculus rift cv1 teardown - ifixit. <https://de.ifixit.com/Teardown/Oculus+Rift+CV1+Teardown/60612>. Accessed: 2018-08-14.
- [23] B. E. Insko. *Passive Haptics Significantly Enhances Virtual Environments*. PhD thesis, University of North Carolina, 2001.
- [24] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME-Journal of Basic Engineering*, 82(Series D):35–45, 1960.
- [25] A. Kaspers. Blob detection. Master’s thesis, UMC Utrecht, 2011.
- [26] A. Kawamura, K. Tahara, R. Kurazume, and T. Hasegawa. Dynamic grasping of an arbitrary polyhedral object. *Robotica*, 31(4):511–523, 2013.

- [27] J. Kim, N. D. Thang, and T. Kim. 3-d hand motion tracking and gesture recognition using a data glove. In *2009 IEEE International Symposium on Industrial Electronics*, pages 1013–1018, July 2009.
- [28] F. Kosmalla. Virtual reality to reality calibration unity example. <https://github.com/felixkosmalla/unity-vive-reality-mapper>. Accessed: 2019-04-05.
- [29] P. Kumar, J. Verma, and S. Prasad. Hand data glove: A wearable real-time device for human-computer interaction. 43:15–26, 01 2012.
- [30] A. Kunz, L. Brogli, and A. Alavi. Interference measurement of kinect for xbox one. pages 345–346, 11 2016.
- [31] LightBuzz. Github - lightbuzz/kinect-finger-tracking: The most accurate way to track fingers using kinect v2. <https://github.com/LightBuzz/Kinect-Finger-Tracking>. Accessed: 2018-07-27.
- [32] logitech. Logitech hd pro webcam c920 für windows, mac und chrome os. <https://www.logitech.com/de-at/product/hd-pro-webcam-c920>. Accessed: 2019-02-12.
- [33] logitech. steamvr-setup-room-scale-9 - road to vr. <https://www.roadtovr.com/steamvr-setup-room-scale-9/>. Accessed: 2019-02-24.
- [34] E. Luckett. A quantitative evaluation of the htc vive for virtual reality research, 2018.
- [35] Mathworks. Matlab - mathworks - matlab & simulink. <https://www.mathworks.com/products/matlab.html>. Accessed: 2018-07-17.
- [36] M. Mehling. Implementation of a low cost marker based infrared light optical tracking system. Master’s thesis, Institute for Software Technology and Interactive Systems, 2006.
- [37] E. Mikael. Reaching out to grasp in virtual reality: A qualitative usability evaluation of interaction techniques for selection and manipulation in a vr game. Master’s thesis, KTH Royal Institute of Technology, 2016.
- [38] J. More. Levenberg–marquardt algorithm: implementation and theory. 1 1977.
- [39] L. Motion. Leap motion. <https://www.leapmotion.com/>. Accessed: 2018-08-02.
- [40] Neewer. Neewer 52mm 52 mm ir 850 nm 850nm infrared infra-red filter | neewer | photographic equipment and accessories for professionals, musicians, and amateur photographers. <https://neewer.com/product/10000319/>. Accessed: 2019-03-19.

- [41] Noitom. Can i use hi5 vr glove without adding any positional tracking devices? | hi5 vr glove. <https://support.hi5vrglove.com/hc/en-us/articles/360003162914>. Accessed: 2018-09-27.
- [42] Noitom. Hi5 vr glove business edition | hi5 vr glove. <https://hi5vrglove.com/store/hi5glove>. Accessed: 2018-07-18.
- [43] Noitom. Home | hi5 vr glove. <https://hi5vrglove.com/store/hi5glove>. Accessed: 2018-07-18.
- [44] Oculus. Accessories | oculus. <https://www.oculus.com/rift/accessories/>. Accessed: 2018-08-02.
- [45] Oculus. Oculus rift | oculus. <http://oculus.com/rift>. Accessed: 2018-07-18.
- [46] OpenCV. Opencv: Camera calibration and 3d reconstruction. https://docs.opencv.org/3.3.1/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d. Accessed: 2019-03-11.
- [47] OpenCV. Opencv: cv::simpleblobdetector class reference. https://docs.opencv.org/3.3.1/d0/d7a/classcv_1_1SimpleBlobDetector.html. Accessed: 2019-01-03.
- [48] T. Pintaric and H. Kaufmann. A rigid-body target design methodology for optical pose-tracking systems. In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology, VRST '08*, pages 73–76, New York, NY, USA, 2008. ACM.
- [49] N. Pollard. The grasping problem: Toward task-level programming for an articulated hand. *Technical Report 1214*, 1990.
- [50] J. Redmon and A. Angelova. Real-time grasp detection using convolutional neural networks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1316–1322, May 2015.
- [51] satya Mallick. Blob detection using opencv (c++ / python) | learn opencv. <https://www.learnopencv.com/blob-detection-using-opencv-python-c/>. Accessed: 2019-01-03.
- [52] N. Shaker and M. Abou Zliekha. Real-time finger tracking for interaction. pages 141 – 145, 10 2007.
- [53] K. B. Shimoga. Robot grasp synthesis algorithms: A survey. *Int. J. Rob. Res.*, 15(3):230–266, June 1996.
- [54] Skarredghost. How to use kinect with htc vive. <https://skarredghost.com/2016/12/09/how-to-use-kinect-with-htc-vive/>. Accessed: 2018-09-27.

- [55] K. D. Son. son-oh-yeah/moving-target-tracking-with-opencv. <https://github.com/son-oh-yeah/Moving-Target-Tracking-with-OpenCV>. Accessed: 2019-03-05.
- [56] L. Steindl. Hybrid tracking technology for virtual rock climbing. Master's thesis, Vienna University of Technology, 2018.
- [57] P. Sturm. *Pinhole Camera Model*, pages 610–613. Springer US, Boston, MA, 2014.
- [58] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.
- [59] C. J. Taylor and R. J. Schwarz. The anatomy and mechanics of the human hand. *Artificial limbs*, 2 2:22–35, 1955.
- [60] O. team. Opencv library. <https://opencv.org>. Accessed: 2019-04-05.
- [61] U. Technologies. Unity. <https://unity3d.com/de>. Accessed: 2018-07-17.
- [62] Unity. Animated hands with gloves - asset store. <https://assetstore.unity.com/packages/3d/characters/animated-hands-with-gloves-48520>. Accessed: 2019-02-23.
- [63] Unity. Steamvr plugin - asset store. <https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647>. Accessed: 2019-02-24.
- [64] Unity. Unity - manual: Unity user manual (2018.3). <https://docs.unity3d.com/Manual/index.html>. Accessed: 2019-02-24.
- [65] Valve. Steamvr. <https://steamcommunity.com/steamvr>. Accessed: 2019-02-24.
- [66] M. R. Villarreal. Scheme human hand bones-en - hand - wikipedia. https://en.wikipedia.org/wiki/Hand#/media/File:Scheme_human_hand_bones-en.svg. Accessed: 2018-11-26.
- [67] M. VR. Learn more | manus vr. <https://manus-vr.com/learn-more>. Accessed: 2018-08-2.
- [68] M. VR. Manus vr | order the manus vr development kit. <https://manus-vr.com/order.php>. Accessed: 2018-07-18.
- [69] M. VR. Tech specs | manus vr. <https://manus-vr.com/tech-specs>. Accessed: 2018-09-27.
- [70] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report 95-041, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

- [71] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 1, pages 666–673 vol.1, Sept 1999.
- [72] Z. Zhang. Microsoft kinect sensor and its effect. 19:4–10, 02 2012.