DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

VR Re-Embodiment: Establishing a Control Structure to enable Physic-based Movement of the Human Body in Unity 3D

Konstantin Karas

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

VR Re-Embodiment: Establishing a Control Structure to enable Physic-based Movement of the Human Body in Unity 3D

VR Re-Embodiment: Einführung einer Kontrollstruktur der Physikalischen Bewegungen Menschlicher Körperteile in Unity 3D

Author:	Konstantin Karas
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Sandro Weber, M.Sc.
Submission Date:	15.04.2020

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 07.04.2020 Konstantin Karas

Acknowledgements

I would like to thank the chair of Augmented Reality, especially Prof. Dr. Gudrun Klinker, for allowing me to write this bachelor's thesis. Further, I am grateful for the chair to generously offer me access to the hardware in the lab. Without this equipment the fine tuning and test of my system in a real environment would not have been possible.
The person who I would like to thank the most is my advisor M.Sc. Sandro Weber. His advice and guidance has helped me immeasurably and I deeply appreciate the time he has granted for our weekly meetings, answering my many questions.
Finally, I am grateful for my friends and family who during stressful times have kept my spirits up and my motivation going through their continuous support and encouragement.

Abstract

The goal of this bachelor thesis is to develop a control structure in Unity 3D that achieves physical movement of a humanoid remote avatar through local VR tracking information. The structure mainly utilizes Unity 3D's built-in components to connect and rotate the body parts. Estimated masses and angular motion limits of the different parts of the human body will be used to facilitate realistic motions. In order to avoid overshooting and dampened behaviour the system has been tuned using relay tuning. An editor has been designed and implemented to configure the control structure and to allow saving different settings. A user study has been designed that analyzes the impact of physical interactions between user and environment upon embodiment, in particular concerning the sense of agency. It has been conducted in the form of an expert study.

Contents

List of Abbreviations

- CoF: Center of Mass
- DoF: Degree of Freedom
- HMD: Head Mounted Display
- IK: Inverse Kinematics
- PD: Proportional-Derivative
- PID: Proportional-Integral-Derivative
- UI: User Interface
- VR: Virtual Reality

1. Introduction

Over the last century, robots have been more and more utilized in the industry. The automation of repetitive and mundane tasks has led to increased productivity in factories and manufacture leading to a decline of human workforce in that field [GM18]. Robots have performed exceptionally well in environments that ideally do not change. In this case, the robot can be configured to always perform the exact same task under identical conditions. However, when it comes to tasks in a dynamic environment, robots still require the guidance of a human operator. Creative or human thinking is often required in environments that are hazardous for the human body. To lower the risk for human workers a robot could be sent to the dangerous location instead, following the commands of the human. For example, a firefighter could remotely control a robot to rescue people trapped in an unstable building without directly risking his or her own health. Another use case is the underwater maintenance of pipelines on the seafloor.

Through game engines, simulations can be run in order to train the operator (e.g. firefighters [Cha+12]). This can be done even without a real existing robot by using a virtual representation instead which would be useful if access to the real robot is restricted due to the risk of damaging it or limited numbers of robots being available (e.g. due to costs). To achieve best control and familiarity with the robot, it would be useful if the operator could be re-embodied into the robot. The operator could then act and react to the given situation as if they were present on the scene themselves. Fortunately, with the advances made in virtual reality (VR) technologies that may become reality.

The effects of VR on the embodiment of the user have long been studied not only by computer scientists but also by professionals of many different fields. Embodiment is a highly complex subject so multiple different definitions have been established over the years. Kilteni's [KGS12] work has aimed to establish a proper definition of embodiment in the context of virtual reality. According to this, the "sense of embodiment" could be broken down into three parts. As long as one of them is at least somewhat present, the person experiences the sense of embodiment. When all of them are experienced to their maximum, the sense of embodiment is felt in its highest intensity.

The first component is the *sense of self-location*. It is the experience of being located inside of the virtual body and should not be confused with identifying with the body [KGS12]. The *sense of self-location* does not imply that one is in control of that body, an experience that is associated with the second component of embodiment, the *sense of agency*. Jeannerod [Jea03] describes this sense as the correlation between the actor's intent of their action and its observed outcome. The actor identifies themselves to be the ones that are responsible for performing that action.
The final component contributing to the sense of embodiment is the *sense of body ownership*. A person that experiences this sense feels as if an observed action has been caused by a body that belongs to themselves. This is different from the *sense of agency*: a person might feel that the body belongs to themselves but may not be in control of the actions done by it. The reverse also holds true, the person might feel as if they are controlling a body's action but rather as a puppeteer [I I00].

This thesis aims to strengthen the *sense of agency* of the user. The key is to have a framework that can be customized to model all properties of the real-world robot in order to steer it consciously, precisely and without input delay.

1.1. Related Work

Exemplary for a system that controls a human model by the means of forces while still being able to react to external influences is the work of Oshita and Makinouchi [OM01]. The humanoid model that has been utilized consists of 18 bones driven by 39 joints using proportional-derivative (PD) control (see Chapter 2.2.5). The controllers regulate the angle and angular velocity of each body part. The model supports retaining self-balance by a mix of active and passive muscle control. No VR input has been used but animation input instead.
Another publication that has utilized PD control is *"Motion capture-driven simulations that hit and react"* [ZH02]. As the title already suggests, motion capture has been used to record humanoid animations and is used to control the virtual body. The human model while simple is capable to modify the values of the motion capture data in order to react to impacts. Inverse kinematics (IK) are used to enhance and expand a motion capture library. The IK problem is best described as the task of computing the pose of a system that consists of a chain of jointed components (like an arm) based on the position and rotation of the final element in that chain [MC99]. For example, the rotation and position of the elbow and shoulder joint need to be acquired based on the position and rotation of the hand.
Upon impact, the PD controller is temporarily weakened to produce less stiff and more natural reactions. The model can remain balanced as well by trying to achieve a specified position of its center of mass. The system has been tested in the use case of table tennis (only upper body tracked, lower body controlled by balance force) and boxing. During

the boxing, the participants were able to stagger their opponents due to the physics acting on them, leading to counterbalancing measurements.

In 2008, the french company Aldebaran Robotics has made a novel humanoid robot available to the market, the NAO. The core feature of that robot is that it has been designed to be affordable for research teams. Being the size of a toddler, it is lightweight and has 25 degrees of freedom (DoFs) [Gou+09]. About 13000 NAO robots are currently used around the world, the latest version of the robot has been released back in 2018 [Rob]. Different approaches to steer the NAO with human movement have been established. The Microsoft Kinect [Mic] has been used frequently to gather information concerning the movement of the human operator. Research has been conducted in order to solve the problem of the difference in size among humans and compared to the NAO. The algorithms reconstruct the human body as vectors and obtain the IK from them [Wan+12][Zha+18]. This mismatch causes problems when navigating the NAO. By utilizing the starting position and rotation and applying changes based on them to the NAO, that problem can be further resolved. Due to the limitations of the Kinect (mainly camera resolution) it cannot be used for finger tracking [AM13].

It appears, that PD control and inverse kinematics have been utilized successfully in use cases similar to this thesis. Regarding applications for Unity, there are two relevant tool sets that can be purchased on the Unity Asset Store that focus on these topics. The first one named *Final IK* [Roo19a] deals with controlling inverse kinematics in the context of animations. It offers tools to specify when motion should be controlled by predefined animations and when the inverse kinematics will take over. This package works well in tandem with the other one: *PuppetMaster* [Roo19b]. *PuppetMaster* offers control over the setup of a character that is controlled by physics in Unity. As it will be done in this thesis, it utilizes the *ConfigurableJoint* component of Unity (see Chapter 2.2). Since the inverse kinematics has already been implemented in the current framework, *Final IK* has been deemed unnecessary for the work in this thesis. At the time of this thesis, both tool sets were available for €80.40 each. A goal of the thesis has been to understand the behaviour of some of the components of Unity in order to establish a control structure. Choosing a shortcut by utilizing *PuppetMaster* would have undermined that goal.

Re-embodiment, the experience that one is simultaneously both inside their own and another body is not only of interest for VR related studies but also for robotics. The difference is that in one case, the other body is virtual while being a real robot in the other one. In order to achieve re-embodiment it is mandatory for the user to be familiar with the input devices and visual interfaces so that their use becomes an unconscious task. Direct visual feedback and simultaneity of issuing commands and them being executed in the other body is vital to experience a sense of embodiment. However, re-embodiment is limited by the lack of tactile feedback [Bes15]. The extension of limbs, cognition and perception through re-embodiment depend on each other, making it impossible to merely focus on one of them. Each extension should be differentiated between prostheses and incorporation. Providing the user with an extension of one's arm control (e.g. a virtual arm) does not automatically integrate that virtual arm into the perception of one's body [Pre11]. Regarding latency in VR applications, it has been suggested that the complexity of the task in virtual reality directly impacts the perceived simultaneity and *sense of agency*. Simple tasks such as button presses are bound to perform better than complex tasks and as such the latency can be ignored more easily [Wal+16].

1.2. Original Framework

The work of this thesis is based on a framework developed by the Technical University of Munich (TUM) [WK19]. The purpose of the framework is to enable the re-embodiment of a VR user into a new virtual body, that can react and interact with a virtual world in a physical way. This means that the new body can collide with objects in the scene. It is driven by a physics engine instead of mere *set position* and *set rotation* commands. These commands would result in the instant, visual change of position and rotation of an object. The virtual body should rather apply forces and torques to its body parts in order to reach the desired target position and rotation. The base of the system is the game engine Unity 3D [Uni] (further called Unity) in the version 2018.2.21f. Unity handles all visual feedback for the user and is responsible for the user's input information to reach the virtual body. The framework (see Figure 1.1) essentially consists of three main components:

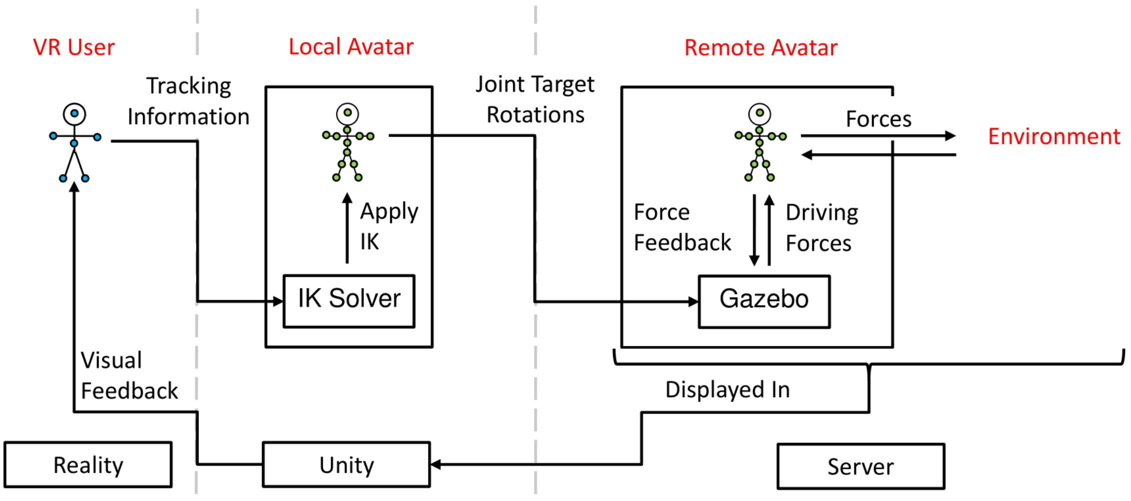


Figure 1.1.: An overview of the server dependent framework.

The most obvious and basic component is the tracker input of the VR device that the user is wearing. The system has been tailored to use the SteamVR Plugin [Ste19] for Unity that enables the use of SteamVR [Ste] for development in Unity. SteamVR is a program that enables playing VR games using a variety of popular VR devices e.g. developed by Oculus [Ocu] and Vive [Viv]. The user has to be equipped with a number of tracking devices. The Head Mounted Display (HMD) is required not only to receive tracking information concerning the movement of the head, but also serves as the visual feed back device of the system instead of an external monitor. There are multiple combinations of tracker positions currently supported. For the most complete tracking version, a tracker strapped around each shin (as closely to the foot as possible), one worn around the waist, one tracking controller held in each hand and the HMD are needed. The trackers around the shins provide information concerning the movement of the feet. Access to the body rotation and position inside of the tracked area is given by the waist tracker. The hand controllers obtain the movement of the hands in the wrist and the HMD of the user's head. The waist tracker is optional as its information can be (to a degree) inferred from the relative position of the head and feet. For testing purposes, the full body tracking using the HTC Vive has been used.

The second part of the system is the gathering of the inverse kinematics from the tracking information and applying them directly to a virtual body. Said virtual body is located inside of the Unity frontend environment and will from now on be referred to as *local avatar* to avoid confusion. This is however not the final virtual body of the user that is controlled by physics but rather the direct IK representation of the user as known through the tracking information. Unity's built-in IK solver automatically acquires the IK information from the tracking input. No further IK optimizations (e.g. for more anatomical correctness) have been implemented. As there is no tracking information provided for the fingers and toes, these parts of the *local avatar* cannot be controlled by the user.

For the backend component, a server-client communication has been established between the Unity application and a server that provides the physical calculations needed to move the final virtual body of the user, the *remote avatar*. Said server utilizes the physics engine *Gazebo* [OSR]. Additionally, the Unity application communicates with the *HBP Neurorobotics Platform* [HBP]. This platform allows for the online simulation of a robot and a virtual environment. This is the environment as seen by the user through the HMD inside of Unity. The robot is the *remote avatar* that is steered through the Gazebo engine by the inverse kinematics obtained by the *local avatar*.

Since information has to be sent to the server in order to steer the robot, the performance of the system is prone to suffer from network latency issues. Depending on the connection to the internet, server load or connectivity problems, the performance of the operator through the robot is impacted negatively. User input would be delayed and as a result, the user would have to wait for the *remote avatar* to follow their commands. The amount of time that the *remote avatar* would need to catch up to the user depends on the internet connectivity, making it unpredictable.

1.3. Contribution

This thesis contributes to the currently implemented frame work by replacing the server component of the system, namely the *Gazebo* physics simulation, by introducing a customisable physics simulation inside of Unity (see Figure 1.2). That physics simulation will from now on be referred to as *control structure*. For this purpose, the *remote avatar* has been transferred into the Unity application and equipped with components provided by Unity to enable physics based movement (see Figure 1.3). The inverse kinematics computed by the *local avatar* are now directly passed to the *remote avatar*. Since the *local avatar* and the *remote avatar* are now located in the same instance of Unity and information is no longer sent to the server, network latency issues have been avoided. The control structure has been implemented to support any humanoid model as its *remote avatar* and has been designed to be quick to learn and efficiently to use.

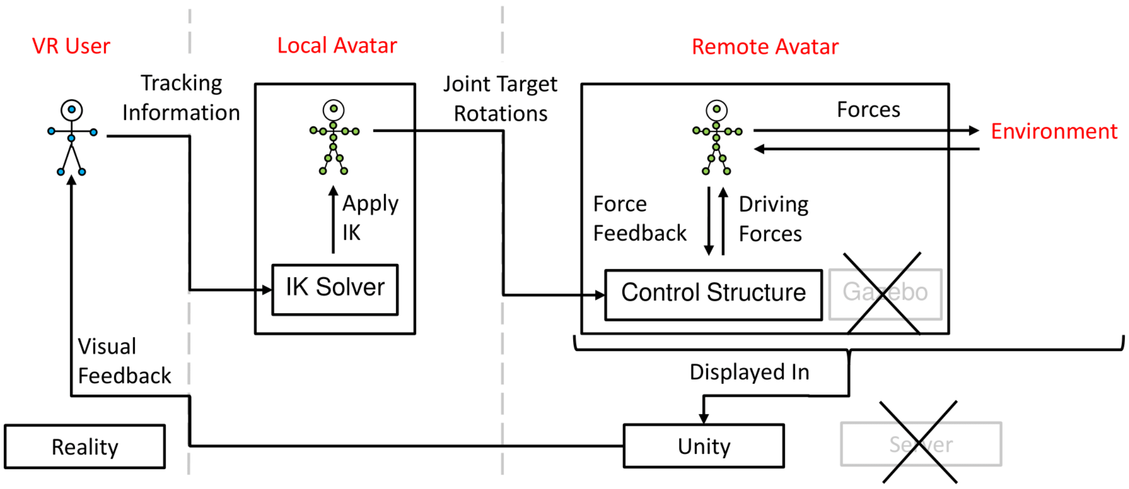


Figure 1.2.: The new server-independent framework. The *Gazebo* component was replaced by the control structure, the *remote avatar* and the environment were moved into Unity.

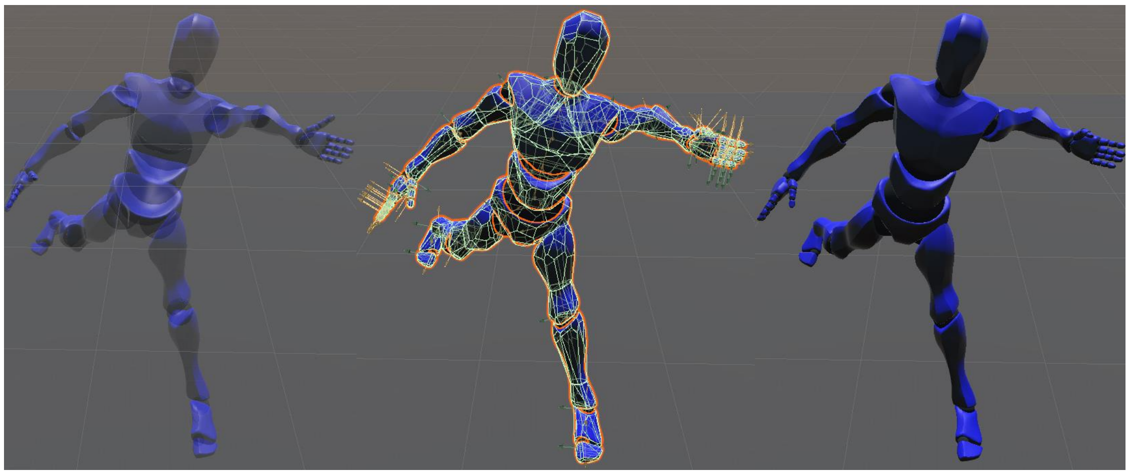


Figure 1.3.: The *local avatar* showing the IK result (left), the *remote avatar* with visible control structure imitating the *local avatar* (middle) and the *remote avatar* with hidden control structure (right). The *remote avatar* is driven by physics through the control structure and the *local avatar* follows the IK obtained from the tracking information.

To better understand the only vaguely documented functionality of the used components that are built into Unity, they have been explored and the findings are presented in Chapter 2. The simplifications made to human anatomy for the purposes of modelling the *remote avatar* as a humanoid are showcased in Chapter 3. The details of the components contributing to the control structure will be presented in Chapter 4. In order to keep the previously implemented tuning service operational, it has been made compatible with the control structure developed for this thesis. More information on this tuning service will be presented in Chapter 5. Since the structure consists of a great number of individually customisable components, an editor has been developed in order to handle the setup and offer the opportunity to load previously saved settings. Chapter 6 covers all functionalities and consequent changes made to the editor over the course of development. The thesis concludes with the description of an user study that evaluates the sense of embodiment provided by the established control structure, with a focus on the *sense of agency*. This user study has been conducted in the form of an expert study (see Chapter 7).

2. Unity – Physics Components

This chapter describes the most relevant components provided by Unity [Uni19a] used in this thesis that enable or utilize physics. The mentioned properties and their behaviours are based on version 2018.2.21f of Unity.

2.1. Rigidbody

In order for an object (also called *GameObject*) in Unity to be influenced by the physics engine, a *Rigidbody* [Uni19b] component has to be attached to it. In this section its most important properties for this thesis will be discussed.

2.1.1. Kinematic vs Dynamic mode

The *isKinematic* field of a *Rigidbody* denies or enables the *GameObject* that the *Rigidbody* is attached to to be moved and rotated by the means of the physics engine. Disabling this feature will cancel out all forces that have previously been applied to the *GameObject*. Normally, when a *GameObject* is a child of another *GameObject* in Unity, moving the parent will also move the child. *GameObjects* that are organised in a hierarchy, such as the skeleton of a humanoid model, will follow this principle. For example, if the left arm is rotated, not only will the upper arm be affected but also the entire arm, including forearm, hand and fingers. This concept does not hold true once a dynamic (*isKinematic* set to false) *Rigidbody* is assigned. Instead, the parenting has to be simulated by linking *GameObjects* containing *Rigidbodys* together by joints (see Chapter 2.2) through the physics engine.

2.1.2. Mass and Center of Mass

The *mass* property of the *Rigidbody* is self-explanatory: it defines the total weight of the *GameObject* that it is attached to. The *centre of mass* (CoM) however is not as straightforward. Díaz [Día18] describes it as the "weight average of the overall mass distribution over the body". Díaz further highlights, that this definition leads to the fact that each external influence on the individual mass points can be related to a single effect on the CoM. In Unity, forces are applied to the CoM unless a specific point has been specified. It should be noted that Unity has no individual mass points to compute the CoM, instead it is automatically calculated based on the combined shape of all *Colliders* attached to the *GameObject*. A *Collider* simply informs the physics engine about collisions occurring between *GameObjects*. Whenever the *Collider* changes (gets disabled, shape changes), the centre of mass is recalculated automatically. If the centre of mass should not be calculated automatically to avoid unintended behaviour, it should be explicitly set to a value once. After doing so no further recalculations are performed. By default, the CoM is set to the same position as the origin of the *GameObject*.

2.2. ConfigurableJoint

Unity provides a couple of different joint archetypes to set up a connection between two *Rigidbodies* quickly. Each of them has a parameter for the *connected body* of the joint. This parameter (a *Rigidbody*) refers to the predecessor of the *GameObject* to which the joint component is attached. For example, the left upper arm would be the predecessor of the left forearm and would be assigned to the *connected body* property of the joint attached to the forearm (see Figure 2.1). It is not possible to set successors. Multiple joints can connect the same body parts and influence each other. This could be used in order to assign a joint for each movement axis (see Chapter 4.2.3).

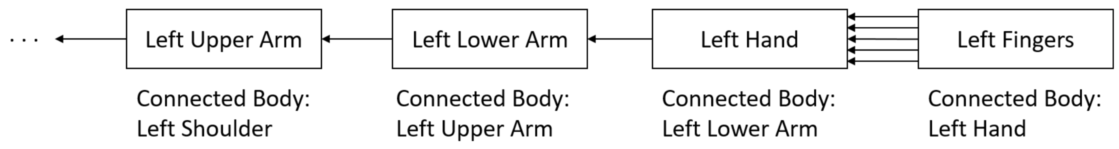

...
Left Upper Arm
Left Lower Arm
Left Hand
Left Fingers
Connected Body: Left Shoulder
Connected Body: Left Upper Arm
Connected Body: Left Lower Arm
Connected Body: Left Hand

Figure 2.1.: The connection between the parts of the left arm modeled by joints.

The most basic version of a joint is the *FixedJoint*. A body holding this component will directly follow the translations and angular motions of its predecessor. A *FixedJoint* can be used to simulate parenting for *GameObjects* using the physics engine of Unity. It acts as a joint that has zero degrees of freedom (DoF).
The *HingeJoint* on the other hand can be directly compared to the hinge of a door. It allows for exactly one single DoF namely to rotation about one axis. When modelling limbs like elbows or knees *HingeJoint*s could be considered. The *HingeJoint* introduces a spring force that allows the joint to maintain a specified rotation around its axis. Said force can be configured by adjusting its spring and damper values.
Unity provides the functionality to automatically set up ragdoll physics for a given character model. It assigns the last type of primitive joints to certain bones of the model, the *CharacterJoint*. While the *CharacterJoint* can be configured to act like a *HingeJoint*, it allows for further DoFs and can be used to model ball socket joints (e.g. for shoulders and thighs).

While each of the joint variants mentioned so far all have their respective uses, none provide steering functionalities to use for the control structure. The one that comes closest is the *HingeJoint* that can maintain a given angle. The drawback is that the *HingeJoint* only has one rotational DoF. Instead of modelling additional DoFs by adding multiple hinges per joint the *ConfigurableJoint* component is used. The *ConfigurableJoint* allows for far more detailed control and specifications of the joint. Not only can it be configured to act exactly like one of the other joint types, but it further adds to their functionality. The following section will provide a more detailed description of the *ConfigurableJoint*.

2.2.1. Joint Axis

Each joint specifies its individual coordinate system relative to the local coordinates of the *GameObject* (indicated by the *connected body* property) that the joint is attached to (see Figure 2.2). In the following this coordinate system will be referred to as *joint space*. The x axis of the joint is special since the *ConfigurableJoint* offers finer control over it when compared to the y or z axis. Settings for the latter are grouped together or not provided as is the case for low and high angular limit. Since the x axis provides the highest level of control, it should be chosen in a way that the movement about the x axis of the joint is the axis about which the primary motion of the body part occurs. The parameter *Axis* hereby refers to the x axis in joint space. To avoid confusion it will be referred to as *primary axis* from here on. For example, if this axis is the vector $(0, 0, 1)^T$, the *primary axis* of the joint motion is identical to the local z axis of the *GameObject* (not the *connected body*) that it is a component of.

A *secondary axis* is introduced in order to define the original rotation of the joint about its *primary axis*. All future rotations of the joint are relative to this initial rotation. The *secondary axis* defines the y axis of the joint space which means that *angular limits* (see Chapter 2.2.3) and *drives* (see Chapter 2.2.5) for the y dimension correspond to this axis. The third and final dimension (the z-axis) of the joint space is automatically calculated as the cross product between the *primary* and *secondary axis* and as such will always be perpendicular to both.

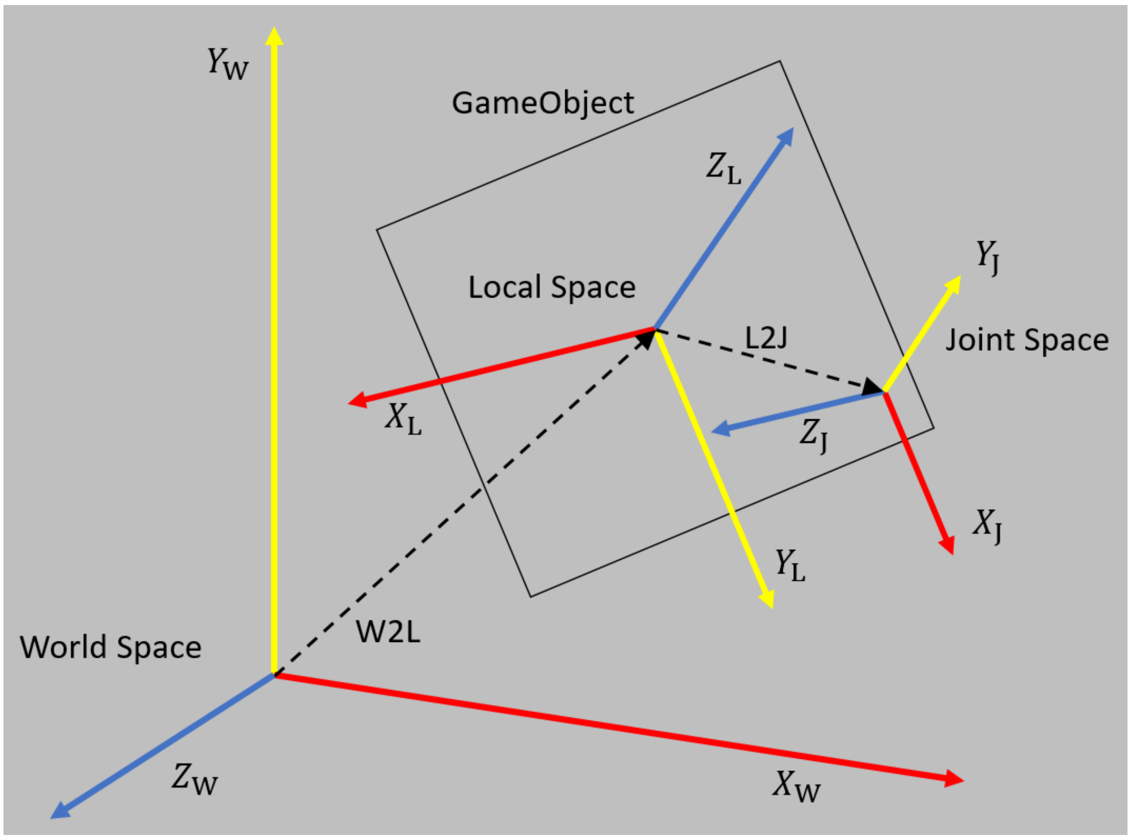
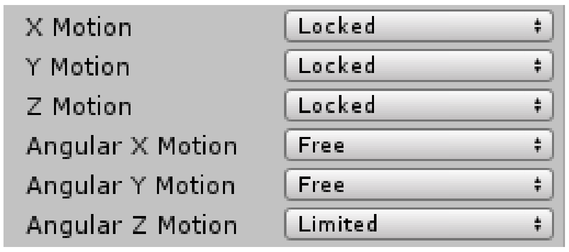


Figure 2.2.: An example for the axes definition of a *ConfigurableJoint*. The joint space is relative to the local space of a *GameObject* (L2J). The local space has been rotated and moved relative to the world space (W2L). The joint's *primary axis* (X_J) is identical to the local y axis of the *GameObject* (Y_L), the *secondary axis* (Y_J) matches the local z axis (Z_L).

2.2.2. Degrees of Freedom

It is possible to configure each mechanical degree of freedom (DoF) of the joint, one for each translation and rotation in each independent dimension [ZZ98]. For a *Configurable-Joint* a DoF can be set to either *Locked*, *Limited* or *Free* (see Figure 2.3). If it is *Locked*, no movement along the axis can occur reducing the DoFs of the joint by one. Essentially, when all dimensions are set to be *Locked* the *ConfigurableJoint* behaves like a *FixedJoint*. If it is *Limited*, movement about the axis has to adhere to the limits specified in the joint such as the maximum angle. If the limit is set to zero, it results in the same functionality as it has been the case in the *Locked* state.
Finally, the DoFs can be set to *Free*. In this case the movement along the axis is entirely unrestricted. No limits are considered when the joint movement is calculated by the physics engine.

The *x*-, *y*- and *z-Motion* parameters are referring to the translation along the respective axis in joint space. While this might be useful to construct e.g. sliding doors, it is not desired in the motion of the human body. Unless to simulate severe injuries, a bone should never separate from the body part that it is connected to by the joint. For this reason, all linear motions along the axes have been set to *Locked*. As a result, the joints configured for this thesis only allow for angular motions, effectively limiting the maximal possible DoF per joint to three instead of six. However, during testing it has been discovered to be possible for translations to occur despite being along a *Locked* axis leading in joints separating from each other. The causes for this instability have been tracked down to very light objects (~20g) of small volume. Hence said erroneous behaviour can occur in the joints of the fingers and toes when colliding with an immovable or relatively heavy object (~10 times as heavy).

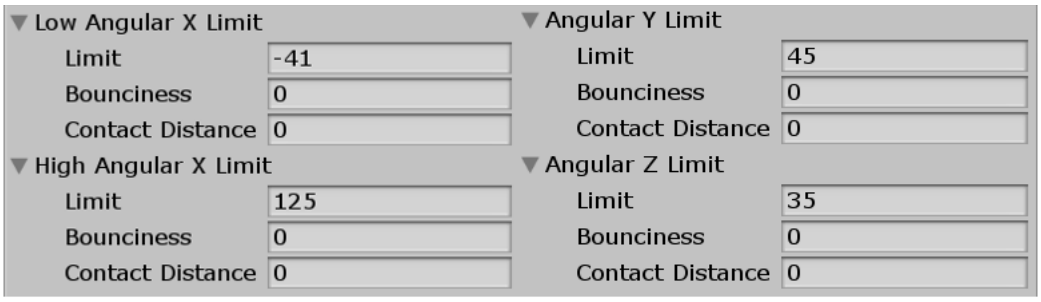


X Motion	Locked
Y Motion	Locked
Z Motion	Locked
Angular X Motion	Free
Angular Y Motion	Free
Angular Z Motion	Limited

Figure 2.3.: The DoFs parameters of a *ConfigurableJoint*.

2.2.3. Angular Limits

As mentioned above, setting the motion of an axis to *Limited* causes the physics to restrict the rotation of the joint around the specified axis. The components of the joint that handle this limit is the *Low Angular X Limit*, *High Angular X Limit* and *Angular YZ Limit*. Notice that the distinction between low and high angular limits for the y and z axis is missing (see Figure 2.4). These are grouped together and cannot be set individually as mentioned before.



▼ Low Angular X Limit

Limit	-41
Bounciness	0
Contact Distance	0

▼ High Angular X Limit

Limit	125
Bounciness	0
Contact Distance	0

▼ Angular Y Limit

Limit	45
Bounciness	0
Contact Distance	0

▼ Angular Z Limit

Limit	35
Bounciness	0
Contact Distance	0

Figure 2.4.: The exemplary properties of the angular limits of the *ConfigurableJoint* component as seen in Unity 2018.2.21f.

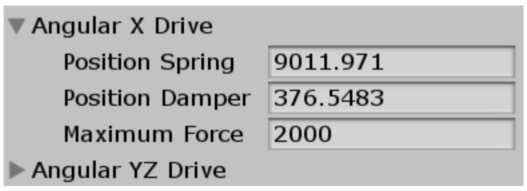
The limits for the x and combined yz axis can be further controlled by a separate spring that pulls the joint back once a limit has been breached. If the spring's spring value is set to zero, the limit cannot be violated resulting in a hard limit. For the purpose of this thesis it has been decided to set all limits as hard limits since having soft limits would result in unnatural, rubbery motions. For the actual limit, an angle can be specified that is represented in joint space. Confusingly, this angle is referenced as *limit* as well, but for the sake of clarity it will be further called *angle*. Since there is only one limit for the yz axis, it is not possible to specify a minimal and maximal angle. Instead the angle value is automatically treated as the unsigned value of both lower and upper limit of the joint's motion. Only symmetrical limits can be specified for the yz axis.
The bounciness of the limit can be set to a value greater than zero in order to make the joint bounce back when hitting a limit. While this would be useful to simulate very light or flexible objects such as a rope, this feature has been left disabled for controlling human joints. Otherwise there would be a jiggling motion of e.g. the arm when the elbow is stretched to its maximum.

2.2.4. Target Rotation and Angular Velocity

The most relevant aspect of the *ConfigurableJoint* for this thesis is that it is capable of turning into and maintaining a rotation relative to its joint space. The rotation is specified as a *Quaternion* and is by default a rotation of zero degrees so the joint will remain in its starting orientation. It should be noted that the *target rotation* is always relative to this original orientation of the object and not the current one. Further details on how to compute the *target rotation* will be provided later in Chapter 4.2.4.
Another interesting aspect of the *ConfigurableJoint* is the option to set a *target angular velocity*. This velocity is represented as a three-dimensional vector in joint space. Its direction dictates whether the rotation is clockwise or not. If a value has been set, the joint tries to maintain the specified angular velocity. The next section will provide further details.

2.2.5. Angular Drives

As it has been mentioned above, a *ConfigurableJoint* can create forces to steer into and maintain a *target rotation*. It can be visualized as a motor, like in a car, that is located inside of the joint. When needed, it applies a torque to the *GameObjects* connected by the joints, rotating them around the axis that the *angular drive* is responsible for. Similar to the *angular limits* there is a drive for the joint's x axis and a combined drive for the yz axis (see Figure 2.5). Each drive consists of the three parameters *position spring*, position damper and *maximum force*.



▼ Angular X Drive

Position Spring	9011.971
Position Damper	376.5483
Maximum Force	2000

▶ Angular YZ Drive

Figure 2.5.: The exemplary properties of an angular drive of the *ConfigurableJoint* component as seen in Unity 2018.2.21f.

According to the NVIDIA documentation [NVI17], the spring and damper value correspond to the proportional and derivative values of a proportional-derivative controller, a PD-Controller. To better understand these values the principle of PD-Control will be explained briefly. A PD-Controller is used in order to compute a correction ($y(t)$). Said correction can be then applied to an observed parameter in order to keep it stable at its target value. As the name already suggests the controller consists of a proportional and a derivative component. Central to the controller is an error quantity ($e(t)$) that expresses the divergence of the current and desired value of the input [RZ04]. In the case of the angular drives, that value would be the difference between the *target rotation* and the current rotation of the joint in joint space.

$$y_{\mathrm{R}}(t) = K_{\mathrm{PR}} e(t) + K_{\mathrm{D}} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2.1}$$

The proportional constant (K_{PR}) directly controls the impact of the proportional component of the controller on the correction. This leads to responsive changes but might overshoot and oscillate around the desired value. The derivative component utilizes the rate of change of the observed error in order to predict its future value and makes preventive corrections accordingly [KNM11]. Its derivative constant (K_{D}) is used to improve this behaviour but will delay the reaction of the controller [Föl+16] and might result in slow, overly dampened behaviour.
The *maximum force* limits the amount of force that the joint can exert at any time when trying to achieve a given *target rotation* or velocity. This value can be utilized to simulate the different strengths of the individual limbs. When a *target angular velocity* is given, the joint utilizes the damper value of the appropriate drive as a motor force in order to correct the difference in the current angular velocity and the desired one. As long as the damper remains at zero, no force will be applied and the joint will not move, as it has been observed during testing. If the intention of the user is to constantly drive the joint using its *maximum force*, it is recommended to set either the damper or the *target angular velocity* to a value that is several magnitudes higher than the *maximum force*. This provokes the joint into exerting full force with little to no wind-up time.

3. Modelling the Human Body

In the following an overview over the specified model that was used for the *remote avatar* is given. This includes the simplifications made to simulate a body part as well as the combined DoFs of the torso, arms and legs. These DoFs are in reference to the according parameters described in Chapter 2.2.2. All mentioned DoFs are angular ones. The problems that have been encountered and their solutions will be discussed briefly.

3.1. Simplifying the Human Body

In this bachelor thesis, all parts of the human body are individually treated as rigid. A rigid body is an idealized object that is defined by the constant distance between any pair of points located inside or on the surface of it [ZZ98]. This does implicate that the shape of the rigid body cannot change regardless of forces acting upon it. Hence a rigid body approximates the properties of a bone. In this thesis the whole volume corresponding to a body part is considered to be a bone and thus a rigid body. The soft tissue of human muscles, flesh and skin are not modelled and are assumed to be non-deformable for the sake of simplicity and computational cost. From here on a body part will be referenced as a bone.
The human body can be described as a kinematic skeleton. Multiple bones are chained together sequentially by different kinds of joints providing different functionalities [AA13]. Since no musculature is modeled the torque that is applied to the bone is generated by the joint that is connecting the respective bones.

3.2. Kinematic Skeleton

This section describes how the human skeleton has been approximated and will detail the properties of significant parts of the body. The simplification above is in accordance to Unity's handling of humanoid characters in the *Animator*[1].
Once a humanoid model is imported into Unity, Unity automatically estimates which body part of the model best resembles the internally used bone defined in the *Animator*. For example, if the model has an upper arm, Unity will recognize that body part as an upper arm independent from its name. This Enum called *HumanBodyBones* (see Figure 3.1) that Unity utilizes does not provide a value for each bone found in a real human body. Instead it is grouping bones into a single bone that cannot or are very

[1]a component that can control a human body using animations or IK

unlikely to move. An example for one of these groups would be the value *UpperChest* referencing the bones that together form the rib cage and its part of the spine. In total there are 56 bone values available that the *Animator* can distinguish between. If no bone in the model can be matched to a bone contained in *HumanBodyBones* that value will remain unassigned.
For both the *local* and *remote avatar* a humanoid, open source character model (*Y Bot*) [Mixb] has been chosen. It specifies 52 relevant body parts. These are parts of the body that can be articulated. Further parts like eyes and jaws are included but have not been previously been used and will not be used in this thesis. The following sections will describe the details of individual chains of the skeleton and their respective DoFs specified in the *ConfigurableJoints* that were attached to the *remote avatar*.

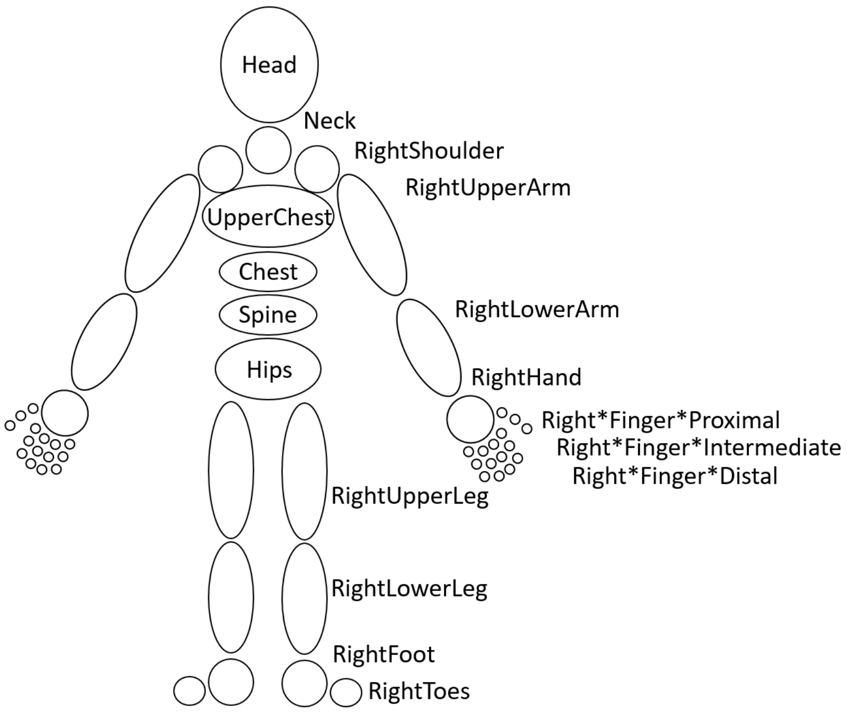


Figure 3.1.: The representation of a humanoid body in Unity expressed by the *HumanBodyBones* values utilized in this thesis without their left-side counterparts. Defined finger names are Thumb, Index, Middle, Ring and Little.

3.2.1. Torso

The Torso consists of the hip (translation: 3 DoFs, rotation: 3 DoFs), three bones along the spine (3 DoFs each), both shoulders (0 DoF since not used by IK of the arm) and includes the neck (3 DoFs) and head (3 DoFs) as well, totalling in 18 DoFs. In order to apply a force to the whole body, the hip bone is at the top level of the hierarchy. There exists one branch consisting of bones and joints from every bone of the body that ends at the hip bone. Since every bone is ultimately connected to the hip bone all forces applied to the hips will in return affect the joints of the other bones. In doing so the character can be moved through the environment by "dragging" its hips around instead of actively generating forces through the legs. The hip is bound to an anchor object by a *ConfigurableJoint* that is simulating a *FixedJoint*. It cannot be moved or rotated relative to that anchor. This abstraction addresses the issue of self-balancing and accurate locomotion which is not the scope of this thesis. Theoretically the movement of the hips could also be set to be limited to 0 and enforced by the appropriate joint drives, however the resulting performance has been deemed to be too unstable.

3.2.2. Arms

Each arm is greatly simplified regarding actual anatomy. The forearm and palm are each represented using a single bone. However, each individual finger segment is modelled with a bone. While the fingers are currently not driven by user input, it is possible that such input will be desired in future works. With that possibility in mind, the fingers have been modelled as well in the control structure. Since the arms are connected to the torso by *ConfigurableJoints*, each force applied by or to the components of the arms will travel along the shoulders and carry over to the arm on the other side. This can lead to slight jiggling motions in the arm on the opposite side when actively moving an arm. The DoFs of the arm are as follows: There are each 3 DoFs in the upper arm and hand as well as 2 DoFs (for spreading and grabbing) in each of the five Proximal finger bones, followed by 1 DoF in the respective Intermediate and Distal bone. The 3 DoFs in the elbow differ from human anatomy. This is due to compensate the behaviour of Unity's IK solver. It has been observed that the elbow joint behaves indeed like a 3 DoFs ball joint in certain situations.

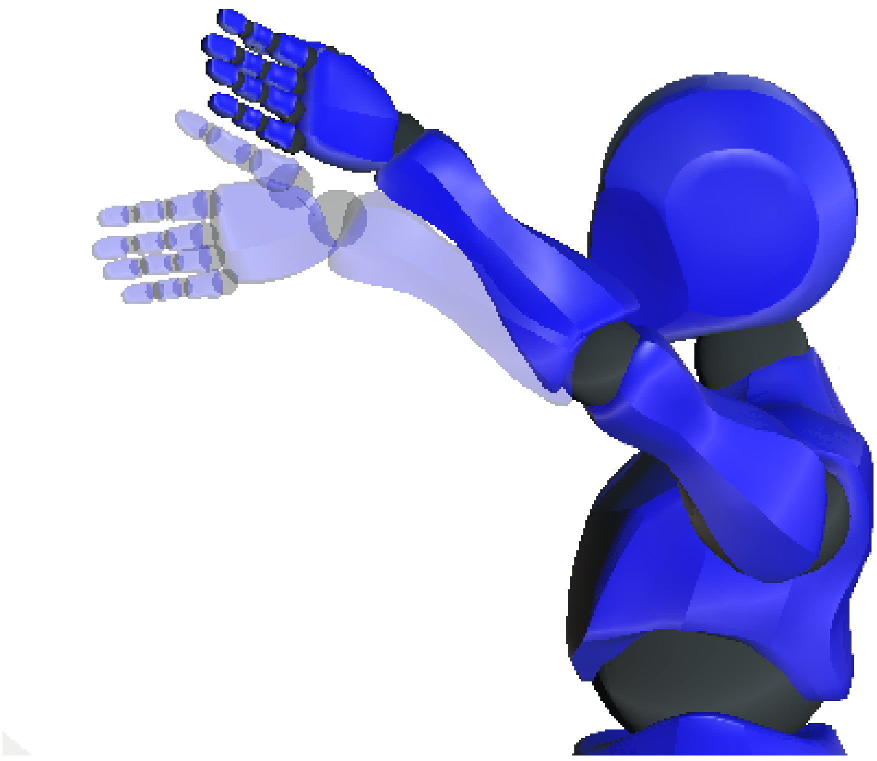


Figure 3.2.: Erroneous rotation of the IK solver. Transparent blue: *local avatar* with hand in correct pose, Opaque blue: *remote avatar* unable to get its hand into position because of 2 DoFs in the elbow.

Figure 3.2 depicts such erroneous behaviour. The *local avatar*'s hand is in the correct desired position, however this pose has been accomplished by bending the lower arm in an unnatural way instead of performing the correct rotation (rolling) in the upper arm in order to allow the lower arm to function like in reality. In order to satisfy the user input, it has been decided to allow for unnatural *angular limits* in the elbow. Since that movement is unnatural, there is no reference in reality for it. For time reasons, instead of manually figuring out the rotation range through trial and error, the associated DoF in the *ConfigurableJoint* has been set to *Free*. Note that this might result in unrealistic movement of the lower arm when colliding with objects, pushing it into the "wrong" direction. In the future one might narrow the *angular limit* down to a range that compensates the issue while not differing from anatomy too much. By modelling the arm with this correction, each arm accumulates a DoF of 29.

3.2.3. Legs

The legs are simplified similarily to the arms, with the exception that the toes are modelled as one single hinge connected to the foot. It is not possible to simulate each toe separately like the fingers. The thigh is connected to the hip bone which is constrained by a *FixedJoint*. This results in forces applied by the upper leg to the hips being neutralized and unable to travel through the system. While this is highly unrealistic, as the legs should be able to push and carry the rest of the body, it greatly improves the stability and balance of the system and is sufficient for the purpose of the system. Similar to the elbow, the knee is largely responsible for rolling the foot, very little additional (anatomically required) rotation is performed in the thigh. The 9 DoFs of the leg consist of 3 DoFs in the thigh, 2 DoFs in the knee, 3 in the foot (ankle) and 1 DoF for all five toes combined. No independent movement of the individual toes is supported since the *HumanBodyBones* do not provide values for the toes and thus cannot automatically associate the toes of a model with a representation in Unity's *Animator*.

3.3. Constraining Human Motion

When it comes to the motion of the human body, the three main constraints are categorized as *anatomic*, *actual* and *mechanical*. The control structure implemented for this thesis focuses on the *anatomic constraints* enforced by the geometry of the human body, including the range of motion that a joint is capable of. These are directly realized by making use of the *angular limits* provided by the *ConfigurableJoint* components. *Actual constraints* relate to external objects that hinder the movement of the body. The control structure has been designed to support collision detection with its environment and can thus react to these constraints. The last category has been discarded for the scope of this thesis. *Mechanical constraints* address the issue of keeping the body upright and in balance which as previously mentioned is resolved in this case by moving the avatar at its hips. [ZZ98]

4. Control Structure

So far, the former framework and the relevant components of Unity that enable physics control have been established. This chapter details the central part of this thesis, the control structure. Said control structure automatically assigns all of the components described in the previous chapter. It specifies the numerous values discussed before and achieves physics-based movement of the *remote avatar* in Unity. Furthermore, the structure should be efficient to set up and applicable to different virtual models.

4.1. Class: AvatarManager

This class exposes multiple settings of the control structure to the developer and is a component of the *remote avatar* root *GameObject*. Currently it is possible to switch between a PD and a *ConfigurableJoint* version.

4.1.1. PD Version

The first attempt at a control structure has utilized PD controllers. These controllers have been assigned to each bone of the *remote avatar* in order to apply a force and a torque that moves the body part exactly like the corresponding body part of the *local avatar*. The force and torque calculation yielded solid results with little overshoot and delay.

However, when a body part has been accelerated rapidly or the direction has inverted abruptly, the body parts have had the tendency to disconnect briefly from each other. This has been the case since only the rotation and position have been controlled, yet if a force has been applied to a body part, no information has been given to the other body parts connected to the affected one. A forearm would be displaced, but the hand did not experience a force and remained in its original place, causing a disconnect.
Furthermore, the PD controller has no angular restrictions to control the movement of body parts. Such a system could be written by hand and the disconnect could be solved by stringing the body parts together with *ConfigurableJoints*, but it has been decided to forego this approach in favour of utilizing solely the *ConfigurableJoint* component of Unity. Since joints in Unity are used to resemble the movement of parented objects by physics instead of the *Transform*[1] hierarchy, the issue mentioned above can be solved as well: Setting the respective connected objects relative position offset to zero can be

[1]position, rotation and scale of a *GameObject*, provides parenting information

achieved by locking the degrees of freedom for all translation axes. The *Configurable-Joint* as it has been discussed before also utilizes PD control (see Chapter 2.2.5), so the resulting behaviour should be similar.

4.1.2. Variables: Bone Dictionaries

The core variables of the class are two dictionaries: *gameObjectsPerBoneFromTarget* and *gameObjectsPerBoneFromRemoteAvatar*. These dictionaries contain key-value pairs with the key being of the type *HumanBodyBones* and the value a *GameObject*. The first dictionary contains information concerning the *local avatar's* body parts. The *Animator* component of said avatar provides access to the mapping between its *HumanBodyBones* and its corresponding *Transform* in the scene. The *GameObject* of that *Transform* is stored in the dictionary instead of the *Transform* itself. That way physical properties can be accessed easily via the *Rigidbody* component of the *GameObject*. The other dictionary resembles the first one with the exception that it is based on the *remote avatar* instead of the local one. This is the avatar that should be influenced by physics calculations and tries to mimic the motion of the *local avatar* (IK). Both dictionaries are initialized in the method *InitializeBodyStructures*.

4.1.3. Method: InitializeBodyStructures

Upon the respective user (developer) input, this method configures and calls the appropriate methods depending on the specified control structure. If the PD version has been chosen then the method will assign a *PDController* to each of the *GameObjects* referenced in *gameObjectsFromRemoteAvatar*. Otherwise, a *ConfigJointManager* is initialized that causes the initialization process of the system that operates on *ConfigurableJoints*.

4.2. Class: ConfigJointManager

In general, when the *ConfigurableJoint* variant has been chosen, the system copies the *ConfigurableJoint*, *Rigidbody* and if desired *Collider* components from a template in the scene for each body part of the *remote avatar* (see Figure 4.1). The template called *AvatarTemplate* resembles the structure of a humanoid character and thus provides its own set of *HumanBodyBones*. As it has been realized in the *AvatarManager*, the respective *GameObjects* that contain the transforms mapped to the bones are stored in a dictionary inside of the *ConfigJointManager*.
The *AvatarTemplate* serves as a blueprint for the physical properties of the *remote avatar*. The reason behind the disconnect between the control structure setup and the *remote avatar* is that in doing so the *remote avatar* model can be exchanged with minimal effort. The only condition that the *remote avatar* model has to meet is that it is of humanoid form.

The purpose of the class *ConfigJointManager* is twofold. It is primarily used to initialize a control structure that utilizes Unity's *ConfigurableJoint* component according to the specifications provided by the developer in the template. These specifications are then distributed to the class *JointSetup* where all components are being assigned to the body parts that can be found in both the template and the *remote avatar*. For example, if the *remote avatar* does not have a definition for individual finger segments the physical properties of fingers will not be copied into the *remote avatar*. The *ConfigJointManager* allows the developer to enable and disable features of the control structure (e.g. *Collider*) at runtime. These feature changes are then delegated to the *JointSetup* class where they are being executed. The secondary functionality of the *ConfigJointManager* is to provide the *ConfigurableJoints* of the body parts belonging to the *remote avatar* with the correct *target rotation* values to mimic the local rotation of their *local avatar* (driven by IK) counterparts.

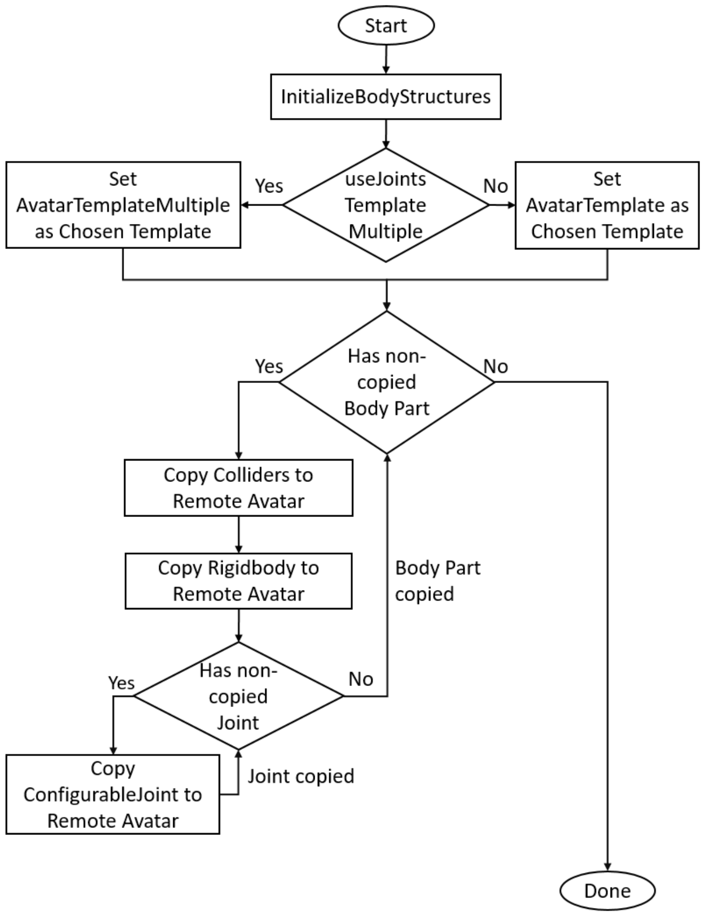

Start
InitializeBodyStructures
useJoints Template Multiple
Yes
No
Set AvatarTemplateMultiple as Chosen Template
Set AvatarTemplate as Chosen Template
Has non-copied Body Part
Yes
No
Copy Colliders to Remote Avatar
Copy Rigidbody to Remote Avatar
Body Part copied
Has non-copied Joint
Yes
No
Copy ConfigurableJoint to Remote Avatar
Joint copied
Done

Figure 4.1.: A simplified overview of the joint setup process. The components configured in the chosen template are copied to the *remote avatar*. Depending on the template chosen (see Chapter 4.2.3), a number of joints is copied to the *remote avatar*. Each body part (defined as a bone) is treated individually.

4.2.1. Simple Colliders

The first feature that the *ConfigJointManager* provides is to facilitate collision detection between the body parts and other objects in the scene. As described above *Collider* are needed to register collisions when dynamic rigid bodies are used. The developer can currently choose between simple *Colliders* and *Mesh Colliders* and can enable or disable them at runtime. When the simple option is selected the *JointSetup* class will copy all *Collider* components found at the body part of the template to the respective body part of the *remote avatar* (see Figure 4.2). These *Colliders* have been set up manually to best fit the shape of the template avatar by the using all basic 3D *Colliders* (box, sphere and capsule). In some cases, multiple *Colliders* have been attached to the same body part to achieve a better fit (see Figure 4.3). It should be noted that while these *Colliders* do match the shape of the avatar template, they will potentially be inadequate for the *remote avatar* when the *remote avatar*'s model has been changed.

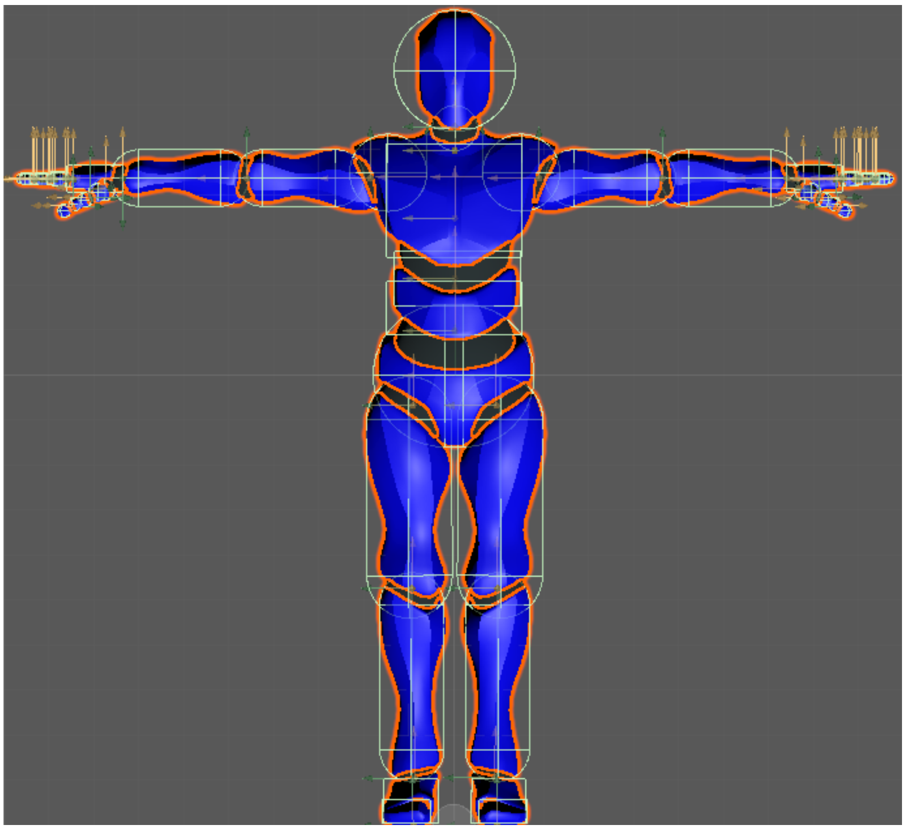


Figure 4.2.: The simple *Colliders* (yellow lines) attached to the *remote avatar*. Approximate the form of the model but no exact match (see arms, legs, head).

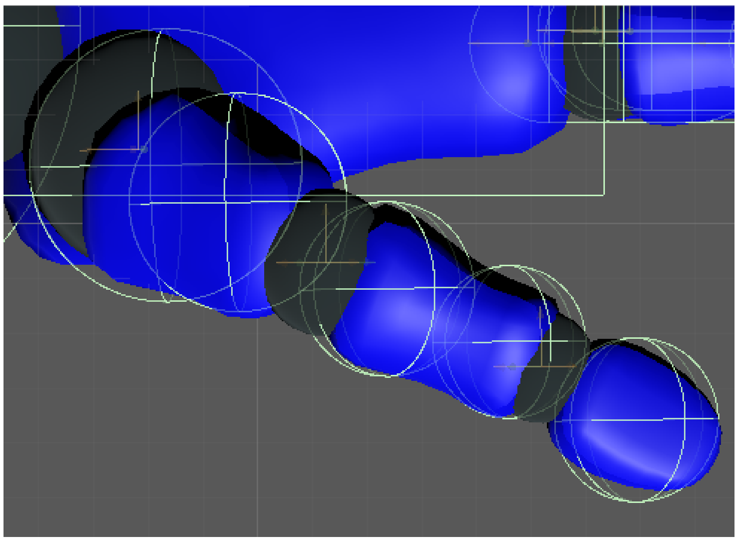
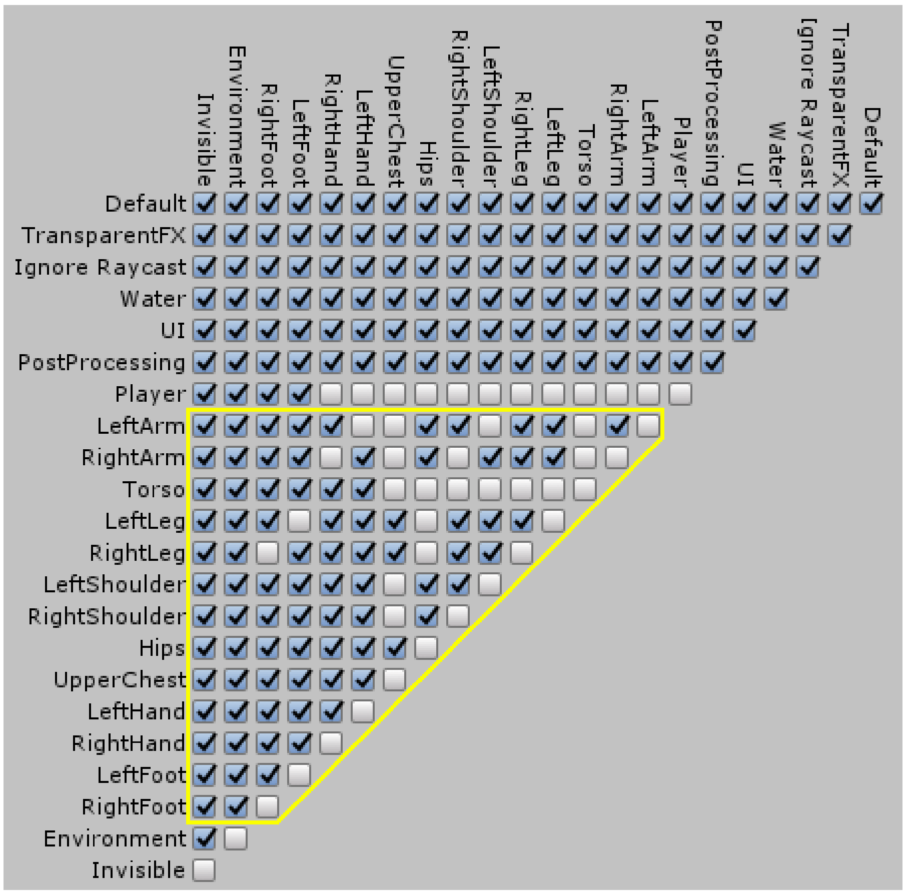


Figure 4.3.: Two spherical *Colliders* attached to the first and middle segment of the left thumb. Rotating a *Collider* is not possible in Unity so a single *Capsule Collider* cannot be used.

For the purpose of achieving a reasonable approximation some *Colliders* intersect with each other which has led to inadvertent collisions between the bones. These coillisons have resulted in an unstable, jittery behaviour.
To stabilize the *Colliders* and thus the body parts attached to them, individual layers[2] for the torso, left and right arm and left and right leg have been created. Additionally, layers for the hand, feet and shoulders have been introduced to further avoid self-collisions and provide greater mobility. For example, the left arm cannot collide with the shoulder or torso in order to be able to move the arm across the upper body. The hand however can still collide with both the arm and the upper body. The respective body parts of the template have been assigned to these layers. By utilizing the collision matrix (see Figure 4.4) in the physics settings of the project in Unity self-collisions have been disabled resulting in smooth and nearly unhindered movement.
Nearly, because due to the simple forms with which the *Colliders* imitate the shape of the body, some invisible collisions have occurred. This has been particularly obvious regarding the arm movement near the torso. The apparent V-shape of the trunk did not match the approximation with multiple cubes and capsules sufficiently enough to allow the arm to touch the visible surface of the torso when pressing against it (see Figure 4.5).

[2]customisable flag that can be set to manage collisions between *GameObjects* (see Figure 4.4)



	Invisible	Environment	RightFoot	LeftFoot	RightHand	LeftHand	UpperChest	Hips	RightShoulder	LeftShoulder	RightLeg	LeftLeg	Torso	RightArm	LeftArm	Player	PostProcessing	UI	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Water	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			
UI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				
PostProcessing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
Player	✓	✓	✓	✓	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐						
LeftArm	✓	✓	✓	✓	✓	☐	☐	✓	✓	☐	✓	✓	☐	✓	☐							
RightArm	✓	✓	✓	✓	☐	✓	☐	✓	☐	✓	✓	✓	☐	☐								
Torso	✓	✓	✓	✓	✓	✓	☐	☐	☐	☐	☐	☐	☐									
LeftLeg	✓	✓	✓	☐	✓	✓	✓	☐	✓	✓	✓	☐										
RightLeg	✓	✓	☐	✓	✓	✓	✓	☐	✓	✓	☐											
LeftShoulder	✓	✓	✓	✓	✓	✓	☐	✓	✓	☐												
RightShoulder	✓	✓	✓	✓	✓	✓	☐	✓	☐													
Hips	✓	✓	✓	✓	✓	✓	✓	☐														
UpperChest	✓	✓	✓	✓	✓	✓	☐															
LeftHand	✓	✓	✓	✓	✓	☐																
RightHand	✓	✓	✓	✓	☐																	
LeftFoot	✓	✓	✓	☐																		
RightFoot	✓	✓	☐																			
Environment	✓	☐																				
Invisible	☐																					

Figure 4.4.: The layer collision matrix configured for the control structure. Layers created to manage collisions for the *remote avatar* are marked in yellow. The checked boxes indicate that a collision can occur between *GameObjects* assigned to these layers (layer names at the top and left).

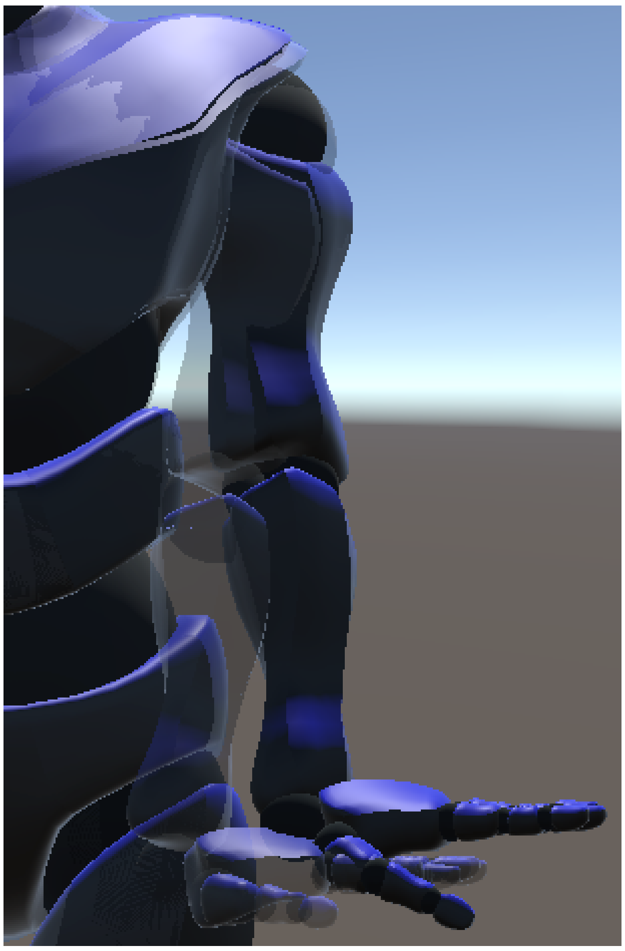


Figure 4.5.: Because of the simple form of the *Colliders*, the *remote avatar*'s hand (solid blue) gets stuck near the hips when moving the arm in front of the legs from an initial resting position.

4.2.2. Mesh Colliders

Alternatively, if the developer desires a highly accurate *Collider* shape, the *Add Mesh Colliders* option of the *ConfigJointManager* can be checked to achieve precise *Collider* shapes at the cost of performance. When the developer assigns the *AvatarManager* class to an *GameObject*, a *BoneMeshContainer* is required to be added as well. This class contains lists of meshes for most body parts and provides access to these lists given a respective *HumanBodyBone*. Currently these meshes need to be assigned by hand. If the option to use *Mesh Colliders* is chosen, the *JointSetup* will assign to each body part (see Figure 4.6) the *Mesh Colliders* that are specified in the respective list in the *BoneMeshCollider*.

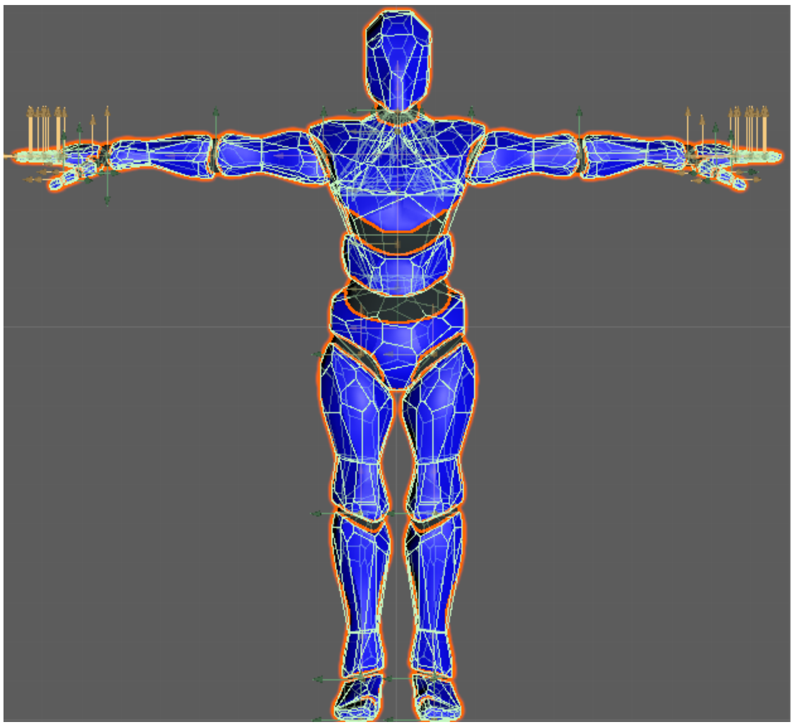


Figure 4.6.: The *Mesh Colliders* (yellow lines) attached to the *remote avatar*. *Colliders* closely match the form of the model.

In the case of the *Y Bot* character model from Mixamo [Mixb] only a single mesh for the character as a whole has been provided. Since *Mesh Colliders* in Unity are static as mentioned above, they cannot be deformed. Instead the developer has to split the mesh into the individual body parts. In this case it is required to import the model into a program of choice that is able to modify the vertices of a model. The following is a brief workflow overview to achieve matching *Mesh Colliders* for individual body parts in Unity by using the external software blender [Ble] (see Figure 4.7).

The first step is to split the mesh into individual objects by selecting the desired vertices of the mesh and pressing *p* in edit mode to separate them from the mesh. Some models already have the vertices of the individual body parts defined as groups for quicker selection.
Once the body parts have been separated there might be the need of further subdividing body parts into approximately convex shapes. This is likely the case for segments of the arms and legs to better cope with the shape of the musculature. For the model used in this paper these limbs have been cut in two at the approximate point where the shape changes from convex to concave.
The next step is to select the body parts and use the *Convex Hull* functionality to generate a completely convex (and closed) form of each body part. This also has the benefit of reducing the number of vertices in the mesh which will improve performance in Unity.

Since the convex hull is reducing the number of vertices and tries to reshape the form into a convex one, it is not recommended to perform this step before the body parts have been further split into subdivisions. The concave parts of the mesh will be spanned by the convex hull resulting in a significant loss of detail.
Finally, the origin of the objects has to be set to the exact same position as the bones in Unity in order to exactly match the shape of the untreated mesh in Unity. This can be accomplished by setting the 3D cursor in blender to the world coordinates of the *GameObject* of the respective bone in Unity. Then the origin of the body part in blender can be set to the 3D cursor. To convert from Unity to blender one should abide the following rules: the x coordinate is inverse in blender, the y axis becomes the z axis and the z axis the inverted y axis. The rotation and scale need to be paid attention to as well. Once all steps have been completed, the file can be exported back to Unity.

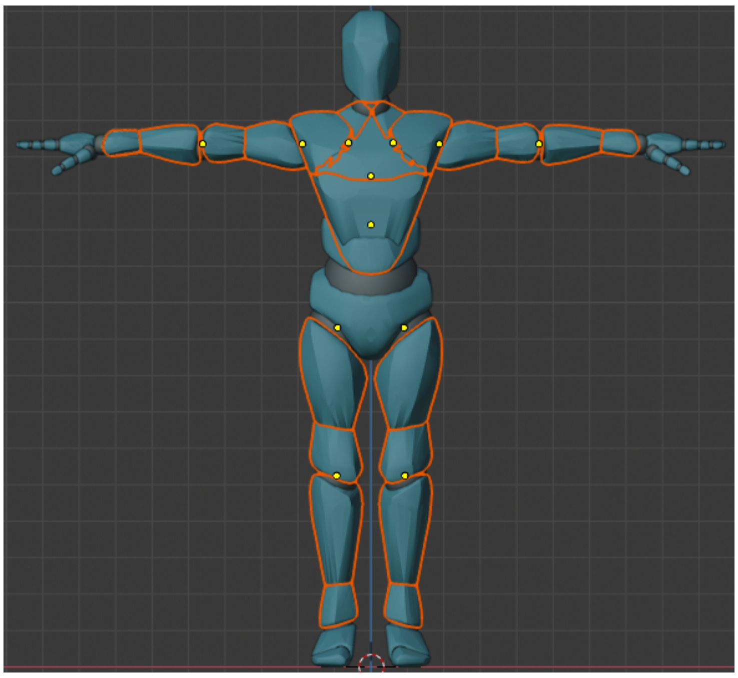


Figure 4.7.: The resulting meshes in blender. Highlighted body parts (orange) have been subdivided to approximate convex shapes. The point of origin of these parts is shown as a yellow dot at the root of the corresponding body part.

Back in Unity, the *ConfigJointManager* can enable or disable the different kinds of *Collider* in play mode. The *JointManager* handles the logic for this process by setting the *Collider* for each specified bone (all of them if called by the *ConfigJointManager*). This provides the possibility to enable and disable the *Collider* of specific body parts at runtime.

4.2.3. Multiple Joints per Body Part

The second main option provided by the *ConfigJointManager* is to enable the developer to choose between a system that assigns a single *ConfigurableJoint* (configured in the *AvatarTemplate*) and multiple joints (one for each axis). The following section will concentrate on the multiple joint variant, since the single joint one is mainly copying the *ConfigurableJoints* of each object in the template to the *remote avatar*'s objects. While the multiple joint version is copying the joints from a different template (called *AvatarTemplateMultiple*), the creation of this template calls for explanation.

The *ConfigJointManager* offers the option to split the *ConfigurableJoint* of the *AvatarTemplate* into three individual *ConfigurableJoints* that are each responsible for a unique axis. This is handled entirely by the *JointSetup* class. As a preparation step for the implementation all joints in the *AvatarTemplate* have been considered and the values of their *primary* and *secondary axis* noted. Then these values have been adjusted in order to make the x axis in joint space first match the y and then the z axis in joint space. This provides the necessary information to define the additional *ConfigurableJoints*. Depending on the original joint's axes, the remaining joints that each control their own axis can be configured. All values have been collected in a table and grouped by *primary* and *secondary axis* values as keys. The original joint's axes values have been restored afterwards.

The *JointSetup* receives the values of the joints' axes and assigns the identified values for the additional two joints accordingly. In order to behave exactly like the single joint from the *AvatarTemplate* the new *ConfigurableJoints* need to be modified in the following way:
The angular y and z DoFs are set to *Free* in order to not hinder the rest of the joints that control these dimensions, only the rotation about the x axis is set to *Limited*. The *Low* and *High Angular X Limits* are set to be the negative of the former given value and the value itself respectively. This is necessary since the *angular limit* for the joint's y and z axis is given as the absolute, symmetrical difference to zero in the original joint. For example, a formerly set *Angular Z Limit* of 20 degrees would be expressed in a separate joint attached to the same bones that has the local z axis set to be the *primary axis* with a *Low Angular X Limit* of -20 and 20 degrees respectively for the high one (see Figure 4.8)

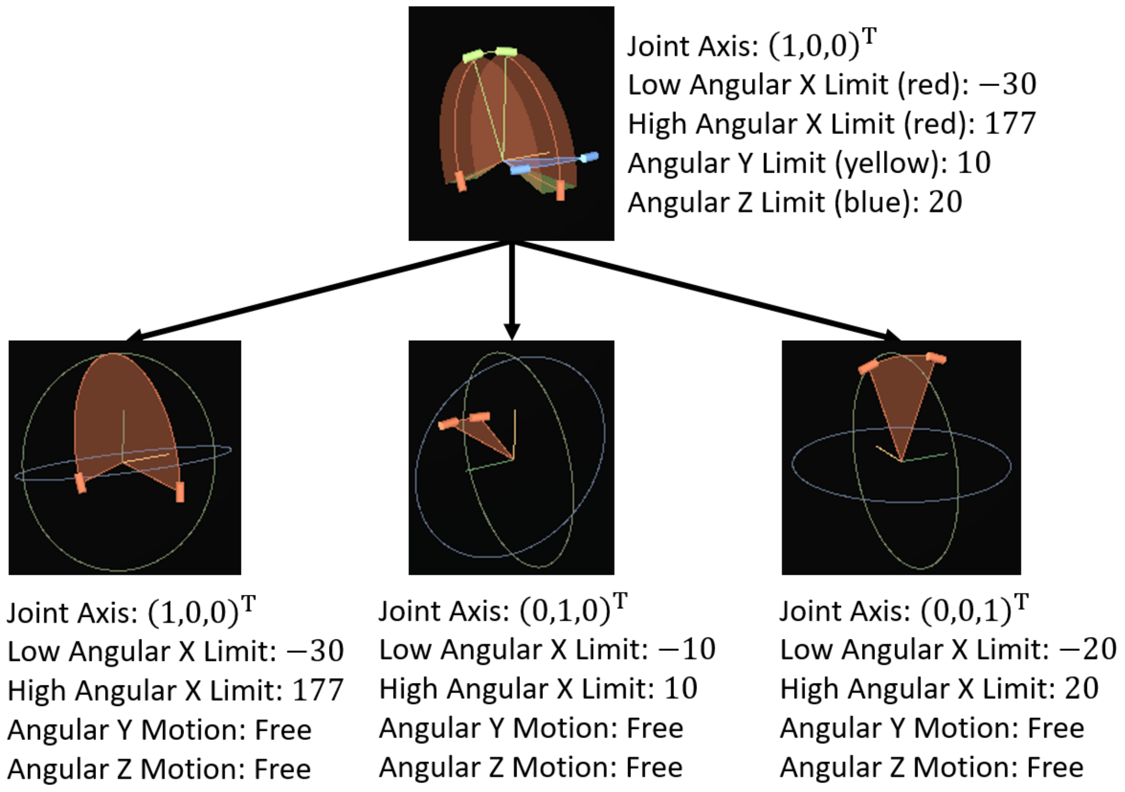


Figure 4.8.: An example for the splitting process: A single *ConfigurableJoint* (top) for all dimensions is divided into 3 individual *ConfigurableJoints* (bottom), each controlling its own axis while maintaining the same angular limits (written here as just the angles) as the original one. Note that while the process sets the y and z directions to *Free*, this figure depicts them set to *Locked* for the sake of visibility (circle outline instead of filled circle)

.

Before initializing the control structure, the developer is able to select either the single or multiple joint version, depending on whether finer control over the joints is required or not. The *AvatarTemplateMultiple* template has been created by copying the result of the joint split method and pasting it back into the scene once the play mode has been exited. Alternatively, the template could have been created by manually assigning three joints for each body part and setting each value individually. By wrapping this process in an editor script using the split joint functionality, changes in the *AvatarTemplate* can be integrated into the multiple joint template with low effort.
It is currently not supported to switch templates or to split the *AvatarTemplate* after initialization. This is due to the fact that switching caused the *remote avatar*'s body parts to rotate and twist. The reason for this behaviour is the angular offset between the body part in the template and in the *remote avatar*. The joint has originally been configured for the starting orientation of the template's body part which no longer matches the

orientation in the avatar since it has already moved. By removing the joint and applying new ones, gravity also affects the *remote avatar*, causing a slight disturbance which amplifies the offset.

Since most of the time finer control is desirable and in regard of the tuning of the joints (see Chapter 5), the *AvatarTemplateMultiple* has been used as the default case instead of the *AvatarTemplate*. Future adjustments to angular limits and tests have been conducted based on this template.

4.2.4. Joint Control

To recall, the second main purpose of the *ConfigJointManager* class is to control the *target rotation* of all joints assigned to the individual body parts. With each update of the physics engine, the current local rotation of each body part is converted to joint space and passed to the *ConfigurableJoints* of that body part as the desired *target rotation* that the joint then drives into. The computation of said *target rotation* is based on Stevenson's [Ste13] method and works as follows: for the first step, the joint's coordinate system is aligned with the one of the world. Then the intended *target rotation* is performed in world space, relative to the joint's starting orientation. This achievement is then converted back into joint space by reversing the previously made transformation into joint space.

5. Automatic Tuning of the Control Structure

This chapter details how the previous PID automatic tuning system implemented by Webel [Web19] in Unity for the former, *Gazebo*-based framework has been expanded. The new version should facilitate the local control structure that has been developed in this thesis while still supporting the former framework. The difference between a PID controller and previously described PD controller (see Chapter 2.2.5) is the addition of an integral (I) component. In [Gof04] the PID is defined as:

$$y_{\mathrm{R}}(t) = K_{\mathrm{PR}} e(t) + K_{\mathrm{I}} \int e(t) \mathrm{d}t + K_{\mathrm{D}} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{5.1}$$

Basically, said component enables the controller to consider all the previously observed error $e(t)$ values to compute the reaction [KNM11]. The term *tuning* in the context of PID or PD control refers to the process of selecting the correct values for the proportional (K_{PR}), derivative (K_{D}) and – in the case of PID control – the integral parameter (K_{I}). However "correct" depends on the behaviour that is intended for the controller which differs from case to case thus making manually tuning a demanding and prolonged task [Hor18]. The following chapter provides a brief overview of the previous tuning system and the changes made to make it compatible with the *ConfigurableJoint* control structure created in this thesis.

5.1. Gazebo tuning system

The implemented system belongs to the category of *Relay Autotuner*. Tuning systems of this type usually utilize a relay function to induce an oscillation into the observed value of the controller. Based on that oscillation behaviour the parameters of the controller can be calculated. Generally speaking an automatic tuning systems requires four steps: The initial step is an *experiment*. This is used to generate a *model* of the behaviour of the oscillation. From this *model* the correct *controller parameters* are obtained given a specified desired behaviour (heuristic). The last phase is the *evaluation*. It reviews the performance of the controller and either accepts the tuning results or restarts the process. [Ber17]

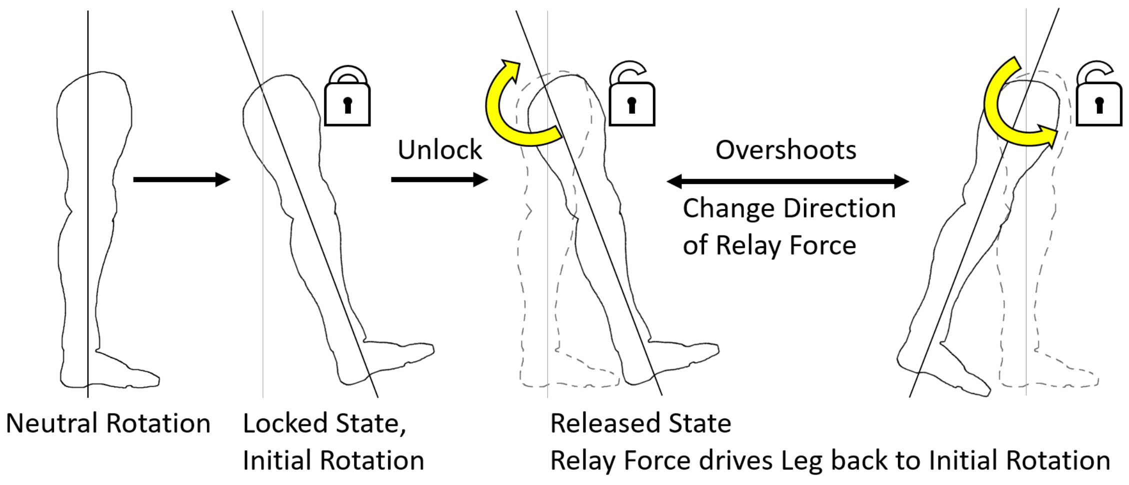

Unlock
Overshoots
Change Direction of Relay Force
Neutral Rotation
Locked State, Initial Rotation
Released State
Relay Force drives Leg back to Initial Rotation

Figure 5.1.: The tuning process applied to the upper leg. The Relay Force (yellow) is invoked by the joint itself and not an external force that affects the leg.

The current *Gazebo* tuning system (see Figure 5.1) tunes each joint individually, one after another. When a body part has an angular DoF greater than one, each axis is tuned separately. To start the process a joint is rotated in the desired direction and locked in that position. Then the joint is released and the maximum force that the joint is capable of exerting is used to drive the joint back towards its target rotation (usually zero). This force is called "relay force". Depending on the previously assigned parameters of the PID controller, the joint will overshoot its mark by a certain amount. Once that has happened, the joint will be driven in the opposite direction with the same relay force, creating an oscillation pattern. After a warm up period for the joint to reach an oscillation, the pattern is analysed and the parameters are retrieved based on a heuristic chosen by the user. The information regarding which joint is tuned and when the relay force is applied is send to the *Gazebo* server in order to follow these commands.
Aside from the automatic tuning, a test runner has been implemented that provides an evaluation on the performance of the PID controller after the tuning process. For a deeper description on both parts of the system – tuning and evaluation – it is recommended to look into Webel's [Web19] description of the system.

5.2. Adaptation of the Gazebo Automatic Tuning System

The main requirement for the changes made to the *Gazebo* tuning system has been that both versions, the *Gazebo* and the adapted local one, remain functional and can be effortlessly chosen by the user. In order to meet this demand, a simple boolean value has been introduced that can be toggled on and off by the user inside of Unity. At each step in the *Gazebo* tuning system, whenever information would be sent to the server an option for the local system has been introduced that will be chosen when the local system should be used instead of the *Gazebo* one. Information will then be send to the control structure instead of the server.

5.2.1. Challenges & Solutions

Since the *Gazebo* system did not utilize Unity's *HumanBodyBones* nor *ConfigurableJoint*, several mapping functions had to be created. The backbone of the tuning system is the *AvatarManager*, that provides the tuning service with the relevant constructs of the local control system. Each axis of a joint has been tuned individually in the *Gazebo* tuning system. To follow this principle, the local tuning system is utilizing the multiple joints template and the user is required to choose this option in the *AvatarManager*. The consequence of this choice is that a greater number of joints need to be tuned since there are more joints in the multiple joints template than supported by the *Gazebo* system. For example there are joints for each individual finger segment. Also due to how the generation of the multiple joints template has been implemented, each joint (corresponding to a *HumanBodyBone*) will always consist of three *ConfigurableJoints* (one for each axis). This is independent of whether movement in that direction is possible or not.
In order to address this issue, two optimizations have been made: first, each joint that has an *angular limit* of zero or is marked as *Locked* will not be tuned, in order to speed up the automatic tuning process by skipping immovable joints. The second one is the introduction of an optional mirror functionality. When this option is chosen by the user in advance of the automatic tuning, all joints that correspond to the right side of the avatar (including its shoulder) will not be considered in the tuning process. Instead their tuning results will be copied from their counterpart on the left side. This option should only be chosen under the assumption that the avatar is mostly symmetric in order to achieve satisfying results.

The following naming convention has been introduced to differentiate between the individual *ConfigurableJoints* of a body part: The name starts with the *HumanBodyBones* (converted to a String) and is followed by the capital letter of the axis. An example would be *LeftUpperLegX* for the first dimension of the joint that connects the left thigh to the hip bone. Based on this naming a function has been introduced that recovers the corresponding *ConfigurableJoint* inside of *AvatarTemplateMultiple*. It uses the *HumanBodyBones* component to acquire the correct body part and the axis's letter to find a *ConfigurableJoint* that has a matching *primary axis* vector, such as $(0, 0, 1)^T$ for the letter *Z*. This process directly enables the local control structure to be used for the tuning process since it does not interfere with previously established name based constructs for the *Gazebo* tunings. Instead of the name of a *Gazebo* joint, a name based on the convention described above is used and related to a *JointMapping*.

5.2.2. Relay Force

The aforementioned relay force (see Chapter 5.1) had to be replicated for the new tuning system as well. At first the solution of applying a torque in the direction that the joint can rotate has been explored. While it has yielded solid results, it has not technically followed the same concept as in the previous *Gazebo* tuning. This has been due to the fact that the torque has been applied to the *Rigidbody* component of the joint instead of the joint creating its own torque. A comparison would be to turn a wheel instead of running the motor in order to move the wheel.

Thus, the final solution utilizes the option of using the *ConfigurableJoint* as a motor in order to create the relay force. At the beginning of each individual tuning process the spring value will be set to zero so that no *target rotation* will hinder the movement of the joint and the dampening value will be set to a small value just to waken the motor. All body parts that are not driven by the joint have been set to being unaffected by the physics engine in order to stabilize the system. This has been done in order to eliminate forces propagating through the body and moving the currently tuned joint which would affect its angle and thus tamper with the tuning result. The *target angular velocity* of the joint has been set to an unreasonably high value in order to animate the joint to apply its *maximum force* to achieve that velocity. The relay force that has been chosen in the inspector is set as the value for that *maximum force* in order to keep that parameter impactful. Its sign is used in order to change the sign of the *target angular velocity* and with it the direction in which the body part will turn.

5.2.3. Saving the Tuning Results

Since there is no longer a server to which a PID configuration could be sent to, these configurations will now be sent to the respective *ConfigurableJoint* in the *remote avatar*. It is recommended to use the *PD tuning* heuristic in order to cope with the lack of an integral component of the *ConfigurableJoint*'s PD controllers. The drawback of this method is that the joint only exists during play mode and will be deleted by Unity upon exiting that mode. The option to send the configuration to any of the avatar templates' *Prefabs*[1] and updating the *Prefab* has been discarded in order to avoid unintended changes made to the *Prefab*. Each *Prefab* of the avatar templates should always remain as a backup for a valid or optimal configuration unless the developer explicitly confirms changes to the *Prefab*. Instead the JSON[2] saving mechanism of the previously created editor for the *JointSettings* of the control structure (see Chapter 6.1.3) has been reused to store all changes made to every joint during the tuning process. More information concerning the editor will be provided in Chapter 6.

5.3. Results

For most motions and tasks, the results form the automatic tuning yield a behaviour with little to no overshoot and both short settling and response time. However, quick and sudden movement especially when the user is spinning around rapidly in e.g. a dance the upper body tends to get dragged behind the target leading to a floppy appearance. This can be adjusted by manually configuring the joint specifications, mainly the *maximum force* that the joints can exert. It should be noted, that testing the performance with animations is not the most representative approach, since the animations have to be evaluated separately by the evaluation system. The lack of transition between them causes the *local avatar* to teleport between idle t-pose or the end pose of the previous animation and the pose of the first frame in the following animation. This instant change of position or rotation of body parts leads to errors in the physics calculation that will result in the temporary breaking of the *remote avatar*'s body.

[1]saved backup configuration of a *GameObject*, including all of its components (e.g. *ConfigurableJoints*)
[2]a standard format to save objects in human-readable text files. Objects can be recovered from text file.

6. Editor Window

During the setup of the control structure a user might be interested in changing the properties of some of the joints in order to create specific scenarios and to observe how different values affect the simulation. While it theoretically would be valid to simply select all joints of interest in the *AvatarTemplate* to change their values and after that performing the same changes in the multiple joint version of the template (*AvatarTemplateMultiple*) it would be practically unfeasible. In the worst case the user would have to unfold the entire hierarchy of the template in order to select the desired joints. Since there are currently three joints for each joint in the *AvatarTemplate*, applying the same change to the multiple joint version would add three times the number of fields to change. Furthermore, the user might be interested in revisiting previous settings of the *AvatarTemplate* e.g. in order to compare their performance to the current configuration. Yet again the user would have to tediously edit each and every field to replicate the settings. The problem is amplified if the user has not written down or memorized every value for each joint.

In order to address these issues an editor window has been created. An editor window is controlled by editor scripts in Unity. These scripts not only run in play mode but also inside of the editor. The script updates whenever the editor does, which is triggered by the user's input e.g when clicking or scrolling.

6.1. Editor Window: First Version

The first version of the editor window has utilized foldouts for every joint in the *AvatarTemplate*. Upon selection the user has been presented with the most crucial joint parameters for the purpose of controlling the *remote avatar*: the *angular limits* (actual degree values) of the joint and the *angular drives* (*X* and *YZ*). These parameters can be edited by the user and once the desired configuration has been established these settings can be applied to the *AvatarTemplate* by the press of a button. A checkbox provides the user with the possibility to change all joints at once which has been useful to adjust the *drives'* values uniformly for all joints. This option has been referred to as *global settings*.

6.1.1. Class: JointSettings

In order to store and present changes made in the editor to the joints' properties the class *JointSettings* has been introduced. Each *JointSettings* object corresponds to a joint that that is attached to a bone in Unity. The assigned bone is represented inside of the *JointSettings* as a variable that can be used to identify it. By providing the constructor with the *ConfigurableJoint* found at the bone in the *AvatarTemplate* all values can be initialized and updated. Additionally, information concerning the *Rigidbody* component of the joint can be obtained such as the mass and the centre of mass. The *HumanBodyBones* value corresponding to the body part that the joint in question is attached to is stored as well.
In order to set the *global settings* a separate constructor has been established. Since there is no *ConfigurableJoint* to be found in the *AvatarTemplate* that exists in all bones, the constructor uses the desired parameters individually. Likewise, since there is no bone corresponding to it, the *LastBone* value has been used in order to identify the *global settings*.

6.1.2. Applying Changes to Multiple Joints Template

To apply the changes made in the editor to the multiple joints template, the split joint functionality of the *JointSetup* class has been reused. Once the apply button in the editor window has been pressed a new *JointSetup* instance is created. Since there is no *ConfigJointManager* that could be passed over to the constructor the class had to be reworked in order to support being used outside of play mode. A distinction has been introduced in most of the class's methods in order to distinguish between called from the editor or from a *ConfigJointManager* during play mode. By using the *JointSetup*, changes made to the *Rigidbody*, Collider and the *ConfigurableJoint* component of a joint inside the *AvatarTemplate* are automatically carried over to the multiple joint template.

6.1.3. Save & Load

For saving and loading the settings of each joint it has been decided to utilize Unity's *JsonUtility* to store the values in a JSON formatted text file. The file is generated once the user presses the respective button and will be named after the current date and time of creation if no name has been provided. The user can drag and drop a previously generated text file into a field in the editor. Upon pressing the load button, the *JointSettings* stored in the text file will be recovered and will overwrite the current ones that the editor contains. Optionally the user can name the configuration and if the name of the file and the name the configuration match, the file will be overwritten instead of creating a new one.

Unfortunately, it is not possible to convert *ConfigurableJoints* to JSON since they are not serializable. Instead the *JointSettings* values stored inside of a dictionary (referenced by *HumanBodyBones* as keys) have to be utilized. The main challenge of the save and load functionality has been the lack of dictionary support of Unity's *JsonUtility*. A helper function has been introduced that parses the entries of a dictionary. It uses *HumanBodyBones* as a key and converts the *JointSettings* found for that key to JSON. Each JSON String has been added to the result String, followed by a line break to indicate where one *JointSettings* object ends and where the next one starts. When parsing the String returned by this function, the string is divided into substrings that represent the *JointSettings*. The *HumanBodyBones* value stored inside of them is used to recreate the dictionary. These values become the keys of the dictionary while the recovered *JointSettings* objects become the respective values of the dictionary.

6.1.4. Lessons Learned

While this version fulfils all functional requirements of the editor, it is however cumbersome and laborious to use. For instance, in order to set the *angular drives* of the fingers on the left, the user would have to unfold two elements per bone (first to access the bone and then a second to unfold the angular drive) with a total number of 15 bones for all fingers of the left hand. In that case it is reasonable to assume that the fingers should be of similar strength, so the values that the user will input are the same. This leads to the tedious repetitive task of typing in all desired values.

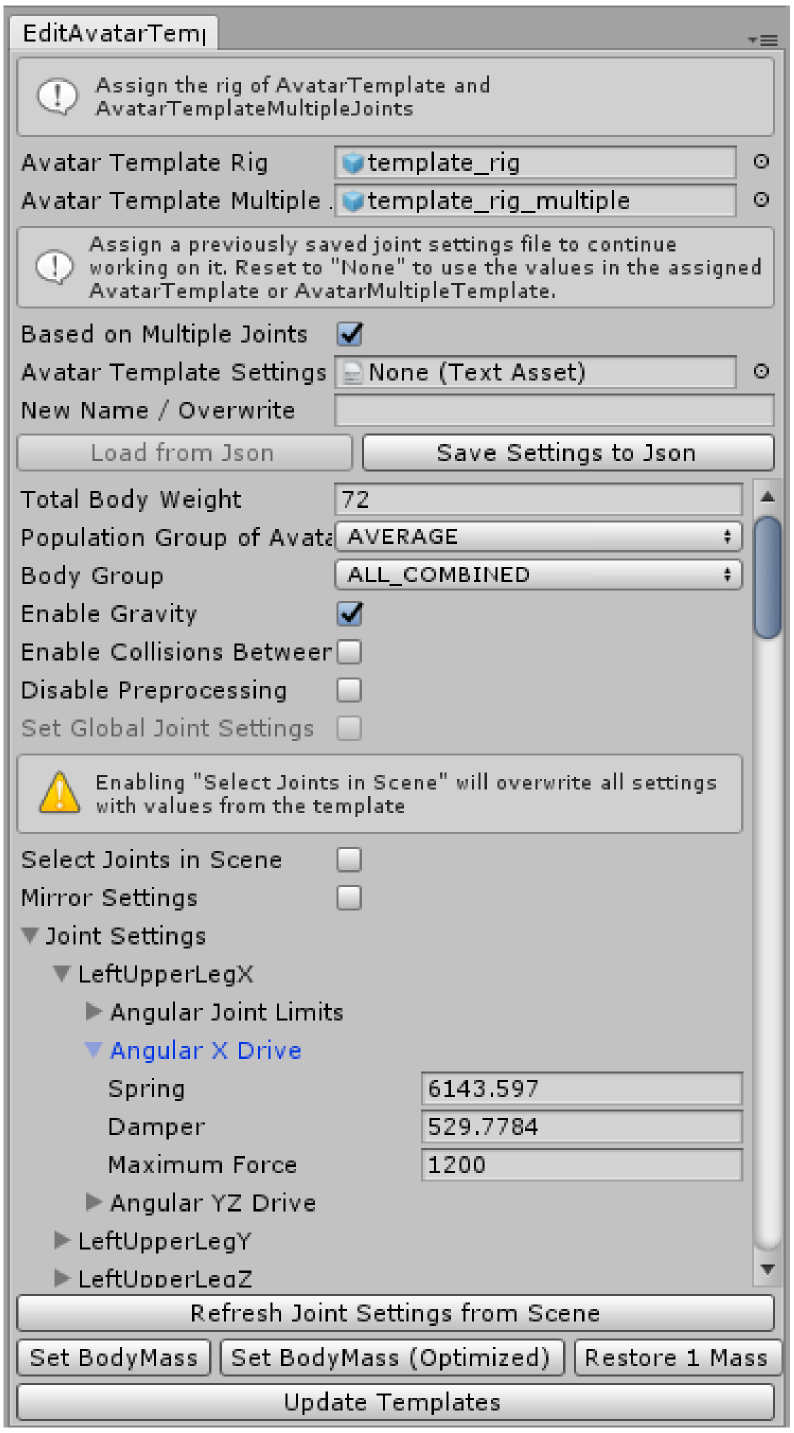
6.2. Editor Window: Selection-based Version

In order to address the issue of high fidelity, the first version has been expanded upon. A new feature has been added that automatically selects the currently filtered body parts (see Chapter 6.4.2) in the inspector. Since all selected objects will have a *ConfigurableJoint* component, changes made to the displayed *ConfigurableJoint* component displayed by Unity will affect each chosen object. That way changes made to the *AvatarTemplate* can be applied quickly. The developer has the option to load the values for the *JointSettings* from the *ConfigurableJoints* in the scene to save them as JSON (see Figure 6.1).

6.3. Tuning compatibility

In order to support results provided by the tuning system that has been added to the control structure later in development, new functionalities have been added to the editor. For example, the ability to use a multiple joints version to apply changes to the *AvatarTemplate* and the *AvatarTemplateMultiple* template. The capabilities of the editor that have been described in Chapter 6.1 and Chapter 6.2 remain unchanged.
Since the editor script is based on the avatar template that uses a single *ConfigurableJoint* for all rotational purposes, it had to be expanded upon to handle the information provided by the tuning process. The most significant change has been to enable the editor to be able to cope with the individual joints for each axis. The user is now able to choose which template (*AvatarTemplate* or *AvatarTemplateMultiple*) should be used as a base (see Figure 6.1). When loading results from the tuning the *AvatarTemplateMultiple* template should be used. When there is only information about a single joint per bone that option should be unchecked.

Since there is no option in the single joint case to control the minimum and maximum angles of each axis, it has been decided that changes made in *AvatarTemplateMultiple* will be applied to the *AvatarTemplate* as follows: Once the update button has been pressed, the maximum absolute value of the *Low* and *High Angular X Limit* of the joint controlling the future y axis (*secondary axis*) and z axis (perpendicular to *primary* and *secondary axis*) respectively will be calculated. These values will be assigned to the single joint's *Angular Y Limit* and *Angular Z Limit* value in the *AvatarTemplate*. This might cause that template to have a greater and more unnatural movement range but all-natural poses can still be achieved. When converting to the multiple joint template, the system remembers which absolute value of the *angular limits* has been greater (the *Low* or *High Angular X Limit*) and only applies changes made in the *AvatarTemplate* to that value. If the user wishes to change the other value that is only possible in *AvatarTemplateMultiple* since that information cannot be modeled otherwise in the template (without further storage mechanisms). It is thus recommended to mainly use the *AvatarTemplateMultiple* template regarding changes to the *angular limits*.


EditAvatarTem|

Assign the rig of AvatarTemplate and AvatarTemplateMultipleJoints

Avatar Template Rig: template_rig
Avatar Template Multiple: template_rig_multiple

Assign a previously saved joint settings file to continue working on it. Reset to "None" to use the values in the assigned AvatarTemplate or AvatarMultipleTemplate.

Based on Multiple Joints: ✓
Avatar Template Settings: None (Text Asset)
New Name / Overwrite:

Total Body Weight: 72
Population Group of Avata: AVERAGE
Body Group: ALL_COMBINED
Enable Gravity: ✓
Enable Collisions Betweer: ☐
Disable Preprocessing: ☐
Set Global Joint Settings: ☐

Enabling "Select Joints in Scene" will overwrite all settings with values from the template

Select Joints in Scene: ☐
Mirror Settings: ☐
Joint Settings
LeftUpperLegX
Angular Joint Limits
Angular X Drive
Spring: 6143.597
Damper: 529.7784
Maximum Force: 1200
Angular YZ Drive
LeftUpperLegY
LeftUpperLegZ

Figure 6.1.: The final version of the editor. The *JointSettings* for the *LeftUpperLegX* are unfolded to demonstrate the layout. These values correspond to the respective joint found in *AvatarTemplateMultiple* inside of the scene.

6.4. Supporting Constructs

For the purpose of the editor, two classes have been implemented in order to provide the developer with more tools to use when making changes to either of the templates. These classes could be expanded upon in the future if there is demand for further functionalities.

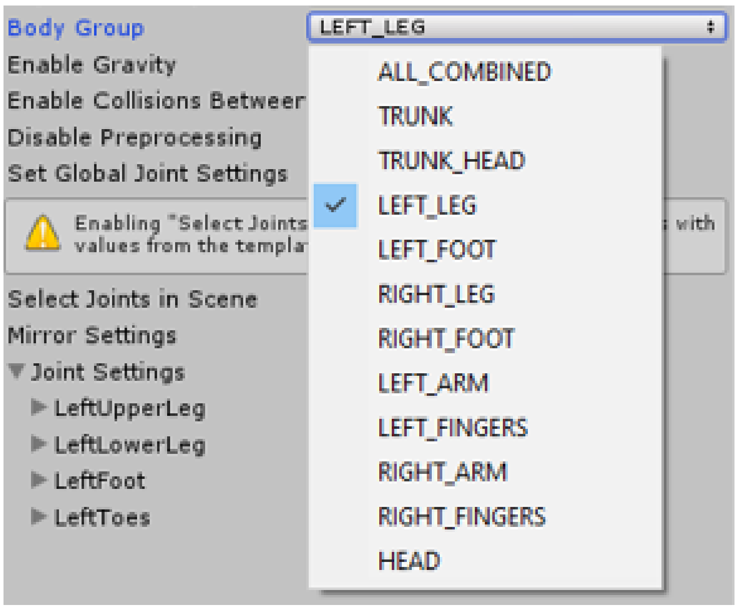
6.4.1. Body Mass

The intent of this class is to store the relative weight distribution of an average human male or female in percent and to apply these values when calculating the mass of a body part. Given a total body weight, the class assigns the individual masses by multiplying the total mass with the relative values. Based on the body parameter values adjusted by Leva [Lev96], those values that have not been documented or that have been included in other body parts have been estimated. For example, the weights of the shoulders and the upper chest have been assumed to be equal parts of the mass that has been measured for the *Upper Trunk*. The user can decide in the editor whether to utilize the *Male*, *Female* or *Average* (of both *Male* and *Female*) values for the avatar. Similarly, the user can set the anatomically correct centre of mass for the body parts.

6.4.2. Body Groups

In order to increase the level of control of the developer over the control system, the *BodyGroup* class has been introduced. The class contains dictionaries of specific parts of the human body, like a dictionary for the left arm or one for the right hand's fingers. The dictionaries can be easily merged together, for example to add the fingers to an arm's dictionary. Custom groupings of body parts could be added to the class in the future in order to meet the developers demands. The dictionaries are created at run time for the *local avatar* and could be used to control specific parts of the body separately. An application would be to completely lock the movement of the left fingers' joints in order to simulate a person wearing a cast.

For the editor, body groups are created as well in order to quickly filter through the list of displayed joint values (see Figure 6.2) and in the case of the selection-based version to select all joints in a group. Switching between the groups works smoothly, although if the developer decides to switch the template version, they have to press *Refresh Joint Settings from Scene* (see Figure 6.1) to reload the *JointSettings* from the new template. Only then the correct joints will be displayed in the editor.


Body Group LEFT_LEG

Enabling "Select Joints with
values from the templa

Figure 6.2.: The option to choose between body groups integrated into the editor. Here, the joints of the *AvatarTemplate* filtered by *LEFT_LEG* are displayed.

7. Expert Study

As mentioned earlier, the main purpose of the control structure established in this paper has been to provide a server independent system that is not impeded by network latency. A user study has been constructed in order to analyse how well the system has met that goal. For the purpose of the study, a buffer has been introduced that stores the desired *target rotation* for each body part and releases them with an offset (the latency) specified by the test operator. Of course, this has not been done when a latency of 0ms has been chosen to save computation time. Due to the fact that the hips are not controlled by the *target rotation* via script but rather by a *ConfigurableJoint* acting as a *FixedJoint* connecting it to the *local avatar*, the hips were not affected by the latency. Another possibility would have been to store the input of the user in a buffer instead, before sending it to the *local avatar*, but that would not be equivalent to the network latency in the former system. Extensive rework of the control structure would have to be conducted and thoroughly tested, to make sure that no other parts of the system are unstable. On the plus side, not introducing latency to the hips will reduce the risk of the participant losing their balance and lowers the risk of injury, which has been a concern when designing the test for the feet.

7.1. Experiment

This section describes the design and intent of the study. An overview over the test procedure and conditions will be provided.

7.1.1. Participants

Originally, it was intended to recruit the participants through a system of TUM that allows the informatics students of a course to gain a bonus to their mark (if they have passed the exam) for participating in a user study. Alternatively, students on the university campus could have been invited to join the study. Unfortunately, due to the spreading of the COVID-19 virus in Europe [WHO], the university had to take actions and close access to the VR equipment. Even without these measures, people were asked to avoid personal contact and because of the risk of infection, a user study that involves devices placed on the face like the HMD was out of the question. To compensate an expert study has been conducted, to assess the quality of the test. Nonetheless, the following sections will detail the structure of the originally designed user study. Future work can use this information to conduct the study once the situation has been resolved.

7.1.2. Test Environment

The virtual test environment (Figure 7.1) has been designed to be simplistic as to not distract the participant from the task at hand. Facing the participant's avatar is a full body mirror that displays the participant's movement as feedback to the participant at any time [Wal+16]. Furthermore, the mirror serves as a user interface (UI) that displays the remaining time and the amount of task completions. The avatar has been tuned by the newly integrated tuning system and is based on the multiple joint template. It utilizes *Mesh Colliders* for precise collision detection and the mass of its body parts have been based on an average (*BodyMass* uses *Average* values for body parts) person with a total weight of 72kg. The height of the *local* and *remote avatar* is set by the test operator to match the height provided by the participant. Positioned in front of the virtual body are empty boxes that are equipped with a *Collider* for each side wall as well as a kinematic *Rigidbody*. This has been done in order to enable physical interactions between the box and the virtual body while preventing the user from both moving and rotating the boxes. These boxes along with a green cube (the hand target) belong to the *hand phase* of the test. A similar setup is located on the ground for the user to step onto. That arrangement is called the *foot phase*.

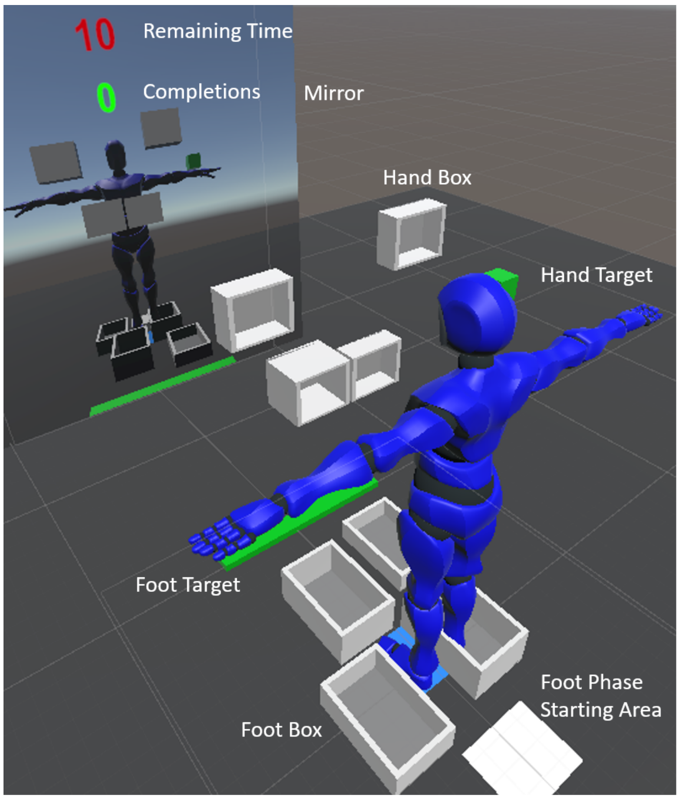


Figure 7.1.: Test setup at free-roam phase before test starts, both *hand* and *foot phase* visible.

7.1.3. Procedure

Once the demographic questionnaire (A.2) has been filled out and the instructions (A.1) have been read by the participant, the participant is equipped with the tracking devices. The participant initially starts with no virtual body, only the visible handheld controllers indicate their position. In the scene is the transparent *local avatar* standing in a t-pose. The participant is asked to move towards it and compare their heights. If needed the test operator can adjust the height of the *local avatar* until it matches. Once it does, the participant is asked to stand inside of the *local avatar* and perform a t-pose. Upon pressing the grip buttons the control structure is initialized and the *remote avatar* spawns. The *local avatar* becomes invisible to avoid distracting the participant. The participant is now given two minutes to free-roam the test environment to get accustomed to their new virtual body. If needed adjustments to the trackers are made.

Once the time is up, the operator presses a button correlating to a latency setting. Depending on the number of participants conducting the study, several latency settings can be tested for. Since the test has been designed to take place in the lecture free period after the exams of the TUM, a small number of participants has been expected. As a result the number of latencies to be analysed has been limited to three. The latencies 0ms, 125ms and 350ms have been chosen, requiring a total of 12 participants (3! permutations and one repetition to verify the results). According to the findings of Waltemate et al, the *sense of agency* should decline for latencies greater than 125ms and break apart when greater than 350ms [Wal+16]. At the start of the test the operator chooses a permutation of latencies and presses the correct button accordingly for the first latency setting. The core element of the study follows.

After a short preparation timer has run out, the *hand phase* starts. The participant has to use their writing hand to reach inside the boxes from right to left (or left to right if left handed) and touch the grey back panel of the box with their fingers. The panel turns green once they have touched it. They have to avoid hitting any other side of the box. Hitting another side causes the side to turn red. Once they no longer touch it it goes back to white. Meanwhile the time that the side has been red is measured in addition to the duration of each body part colliding with the side. That way it can be later evaluated how much e.g. the thumb has been involved in hitting a certain side. This tests the accuracy of the participants movement.
Once each back panel has been touched, the participant has to touch the green target in order to complete the task, they now complete the same task again in the reversed direction until the timer has reached zero. Once that happens, the participant is asked the embodiment questions as seen in Chapter 7.1.4 in random order. Most questions will be answered with a number between 1 (strongly disagree) and 7 (strongly agree).

The *foot phase* under the same latency conditions follows once all questions have been answered. The task is essentially the same, the participant has to step into the boxes while avoiding to hit the sides. Once the green target has been stepped onto, the test pauses and the participant is guided back to the starting position. Upon reaching it the test commences until the time has run out, the same embodiment questions follow. The process repeats two more times until all latency settings have been tested.

7.1.4. Perceptual Judgements

After each phase of a latency test, the participant is verbally asked the questions presented in the embodiment questionnaire that can be found in the appendix (A.3) in a random order. It has been adapted from a standardized questionnaire for the evaluation of embodiment [GP18] with the focus being the established components of embodiment (see Chapter 1: *sense of location*, *sense of agency* and *sense of body ownership*). Additional components (such as "response to external stimuli") included in said questionnaire have not been considered in the evaluation (Table 7.1) and not included in the questions of the embodiment questionnaire of this thesis. The questions remain the same, but are directly impacted by the task that has been completed prior to them. That way, the reaction to the individual phases, hand and foot, can be obtained.

	Body Ownership	Agency	Location
Questions	Q1, Q2, Q3, Q4, Q5	Q6, Q7, Q8, Q9	Q10, Q11, Q12
Evaluation	(Q1 – Q2) – Q3 + (Q4 – Q5)	Q6 + Q7 + Q8 – Q9	Q10 – Q11 + Q12
Sense of Embodiment	((Ownership/5) * 2 + (Agency/4) * 2 + (Location/3) * 2 + 0/4 + 0/4 + 0/5) / 9		

Table 7.1.: Sense of Embodiment evaluation scheme based on [GP18]. Not examined components have been set to their average value of 0. The answer $(1-7)$ to a question results in a value ranging from -3 (strongly disagree) to $+3$ (strongly agree) for the evaluation. Score ranges from left to right: $-15/15$, $-12/12$ and $-9/9$. Score range for sense of embodiment: $-2/2$.

The most relevant result of the study is the experienced *sense of agency*. By introducing artificial latency to the system, the temporal discrepancy between the participant moving their real body (intent) and the *remote avatar* responding to that input (result) increases which should directly impair that *sense of agency* of the participant.

7.1.5. Motor Performance

Accompanying the embodiment questionnaire is an evaluation of the motor performance of the user. Each joint is analysed regarding its precision (angular error) using the existing tuning evaluation system [Web19]. By comparing the participant's subjective answers from the embodiment questionnaire with the actual measured data of their performance, more insight into the quality of the system can be acquired than by merely basing the evaluation on the subjective view. The motor performance of the participant is further analysed by comparing it to the measured time spent hitting the sides of the boxes, leading to information concerning when a certain latency will impede the actual task. The information of the body parts involved in collisions can give an outlook on e.g. whether or not further finger tracking is required.

7.2. Expert Feedback

The feedback concerning the control through the physics engine as well as the conceptualized user study has been largely positive. Movement without artificial latency has been experienced to be simultaneous to the movement of the own body (see Table 7.2).

	Body Ownership	Agency	Location
Evaluation	((-2) – (-3)) – (-3) + (1 – (-3)) = 8	3 + 3 + (-2) – (-3) = 7	2 – (-3) + (-1) = 4
Sense of Embodiment	((8/5) * 2 + (7/4) * 2 + (4/3) * 2) / 9 ≈ 1.04		

Table 7.2.: The evaluation of the *hand phase* without any artificial latency from the expert study. The *sense of agency* received the highest score relative to its possible maximum. All scores are well above average (0).

The questions contained in the demographic questionnaire have been deemed useful in order to relate the embodiment results to attributes of the participant. Several suggestions and advice obtained from the expert feedback have been realized after the expert study:

- The order in which the boxes should be touched has not been made clear in the scene. To solve this issue numbers from $1 - 4$ have been introduced above (for the *hand phase*) or next to (for the *foot phase*) the boxes. In the case of the *hand phase*, the numbers will reverse once all targets have been touched to display to the participant, that they should now touch the boxes in the opposite order.
- A major point of critique for the *foot phase* has been that the boxes felt too small in the direction of the foot length. This issue has been said to originate from the lack of fine angular foot control (tracker is above ankles) which results in a feeling of wearing oversized shoes. As a result involuntary collisions with the boxes could not have been avoided or were too difficult to avoid. In order to obtain more precise results from the user study, these boxes have been lengthened by about 10%.
- The introduction text has been missing a statement at the end that encourages the participant to ask unanswered questions. That statement has been added.

As it has been expected, control for the fingers, especially the thumb have been requested in order to avoid collisions with the boxes. Moving the fingers together has been the instinctive approach which is currently not possible. The IK information obtained from the tracking data has the wrist at its end point. No information concerning the movement of the fingers is provided to the control structure and as a result the *target rotation* of these joints is not set. Another point of critique has been the aforementioned phenomenon that the finger segments do separate from each other upon collision. This looks unnatural and disturbing for the user. Solutions suggested in Chapter 2.2.2 have not been applied to the study in order to keep the controlled avatar as close to a real body as possible.

8. Conclusion

This work has introduced a control structure for Unity 2018.2.21f that has replaced the former *Gazebo*-based server physics simulation. The behaviour of the *ConfigurableJoints* provided by Unity has been analysed and used to implement a control structure that enables a humanoid avatar to be controlled through the physics engine of Unity based on the formerly implemented inverse kinematics system. The process is automatized and customisable. The introduction of the templates further increased the usability of the system, making it easily applicable to new models (see Figure 8.1). The editor has been reconfigured multiple times and serves as an overview of the control structures parameters. Finally, the tuning system has been successfully adapted to work together with the new control structure. It is recommended to predominately use the multiple joints template for the *remote avatar* as the tuning has been performed on that template. The *AvatarTemplate*'s tuned values on the other hand are a compromise that while working might lead to different behaviour in some joints. The designed user study and the *sense of agency* created through the implemented control structure have been rated positively by the expert study.

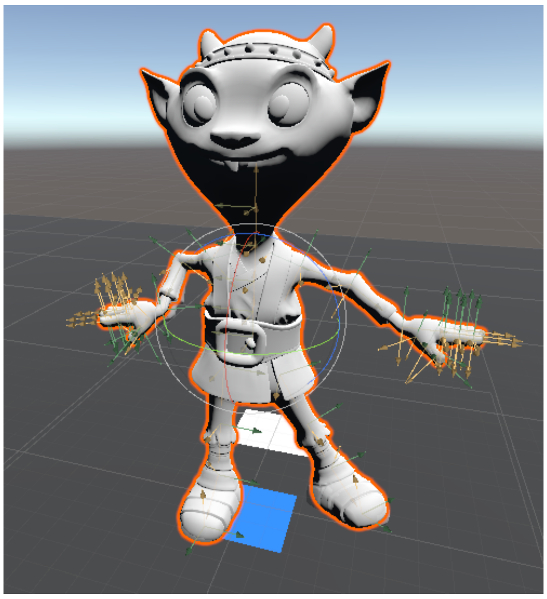


Figure 8.1.: An alternative model [Mixa] used for both the *local* and *remote avatar*. The control structure was automatically created (arrows) even though the model has not all *HumanBodyBones* that the templates define (4 fingers per hand) and the proportions are different (no *Colliders* added).

9. Future Work

This chapter summarizes some of the issues still remaining and if known provides suggestions how to resolve them:

The *ConfigurableJoints* tend to be prone to numerical errors of the physics engine and cannot deal with extremely rapid acceleration. For example, when the *local avatar* teleports (e.g. by spawning in a location different from where the *remote avatar* is) the *remote avatar* tends to *explode*. That means that the individual parts of the body get disconnected from each other and do not stabilize. This happens in spite of the maximum angular force being limited and the motion in all three axes set to *Locked*. The field *Angular X Limit Spring* has been used to stabilize the issue, it applies a force to the body part in order to enforce an *angular limit*. However, if the parameters of this force are set too low, the *angular limits* will be disabled. Setting them to a magnitude of around 400000 enforces the *limit*, but in some cases causes oscillations around the limit. These parameters would have to be tuned by the means of PD or PID control to find an optimal solution.

As mentioned before, the body parts tend to get disconnected when colliding with objects in the scene that are much heavier or immovable. The phenomenon can be avoided by scaling the avatar to a larger size or increasing the weight of the body parts in questions (fingers and toes). It has not become clear why this behaviour occurs since the joints have been configured to not allow relative motions between the body parts that they connect, only rotations.

The parts of the body tend to react very stiffly to collisions due to the method that has been used to drive them. The *target rotation* is based on the local rotation of the body parts of the *local avatar*. These have not changed since no physics have been applied to the *local avatar*. For example, when an object presses against the shin, the entire leg will be rotated and the lower legs remains unchanged relative to the thigh. By changing the computation of the *target rotation* to depend on the rotation of the body part in world space, that issue could be resolved. The *Mesh Colliders* have been constructed for the currently used model. An automation of the described *Mesh Collider* generation process (see Chapter 4.2.2) would save time when a range of different models would be used.

For the editor, a visual representation for the joints' *angular limits* would be useful to provide a more hands on visual feedback to the user who can then easily compare the movement range of the concerned body parts with his own as a reference.

The tuning of the system works, yet in some cases (mostly the wrist joints) it struggles to find an appropriate oscillation pattern and will be stuck until it has found one. This might lead to the system being stuck in an endless loop in the worst case. The automatic tuning sometimes delivers unsatisfying results as it has been previously been the case as well for the *Gazebo framework*. It is recommended to make manual adjustments to joints that sometimes move overly damped.

While the expert study has led to positive feedback, the actual user study has not been conducted. Future work may utilize the provided material and design to gain further information on the impact of the system on the user's *sense of agency*.

A. Appendix

A.1. Introduction Text

Before the experiment, you will be geared up with a belt tracker around your **waist**, 2 trackers strapped around your **shins** just above your shoes and the **HMD**. Please tell your tester if the equipment is uncomfortable so that it can be adjusted to suit you (especially important for the HMD). Whenever dealing with a VR experience there is a possibility to have **motion sickness**. If you are feeling sick and do not wish to continue or need a break please contact your tester immediately.

During the experiment, you will control a virtual body from a first-person point of view. At first you will not have a body. The Vive controllers floating in the scene indicate the position of your hands. You can see your body-to-be standing in the scene. Move behind it and tell your tester whether it is taller or shorter than yourself. He will then adjust the size of the body until it matches yours if needed. **Next step inside of the body** and perform a **T-pose** (standing upright facing forwards, legs together with feet pointing forward, arms stretched out at roughly 90 degrees to the torso) The Vive controllers should be inside of the hands of the virtual body. **Once the tester tells you, press the grip buttons** of the Vive controllers to materialize your virtual body. If you feel like your body is not in a natural position in relation to your head (e.g. you cannot see your feet when looking down, or a shoulder is at eye level) please inform your tester so that he can manually adjust until the offset is fixed.

You now have time to **explore the test setup** for a while before the actual test starts to get comfortable with your new body. In front of you is a mirror. You can see both phases of the test: **one for the hands and one for the feet.** The task is to t**ouch the back panel of each box (grey, turns green) without touching any edges.** If you touch an edge it will turn **red**. Each box has to be touched in the correct order. **Only move on to the next box once the back panel has been green.** One correct task completion is done when the **green cube (for the hand phase) and the green bar (for the foot phase) is touched with the correct limb** (hand/foot) respectively. For each phase you have **one minute**, try to complete as **many** tasks as you can while touching all back panels (turns green) and have red edges as **few** as possible. Do not try to cheat.

During the hand phase, only use your primary hand to touch the back panels. Leave the other hand at rest at your side. Touch the boxes then from right to left if right-handed and from left to right if left-handed. Upon hitting the green cube touch the boxes with the same hand in reversed order. If you cannot quite reach the boxes you can lean or make small side steps.

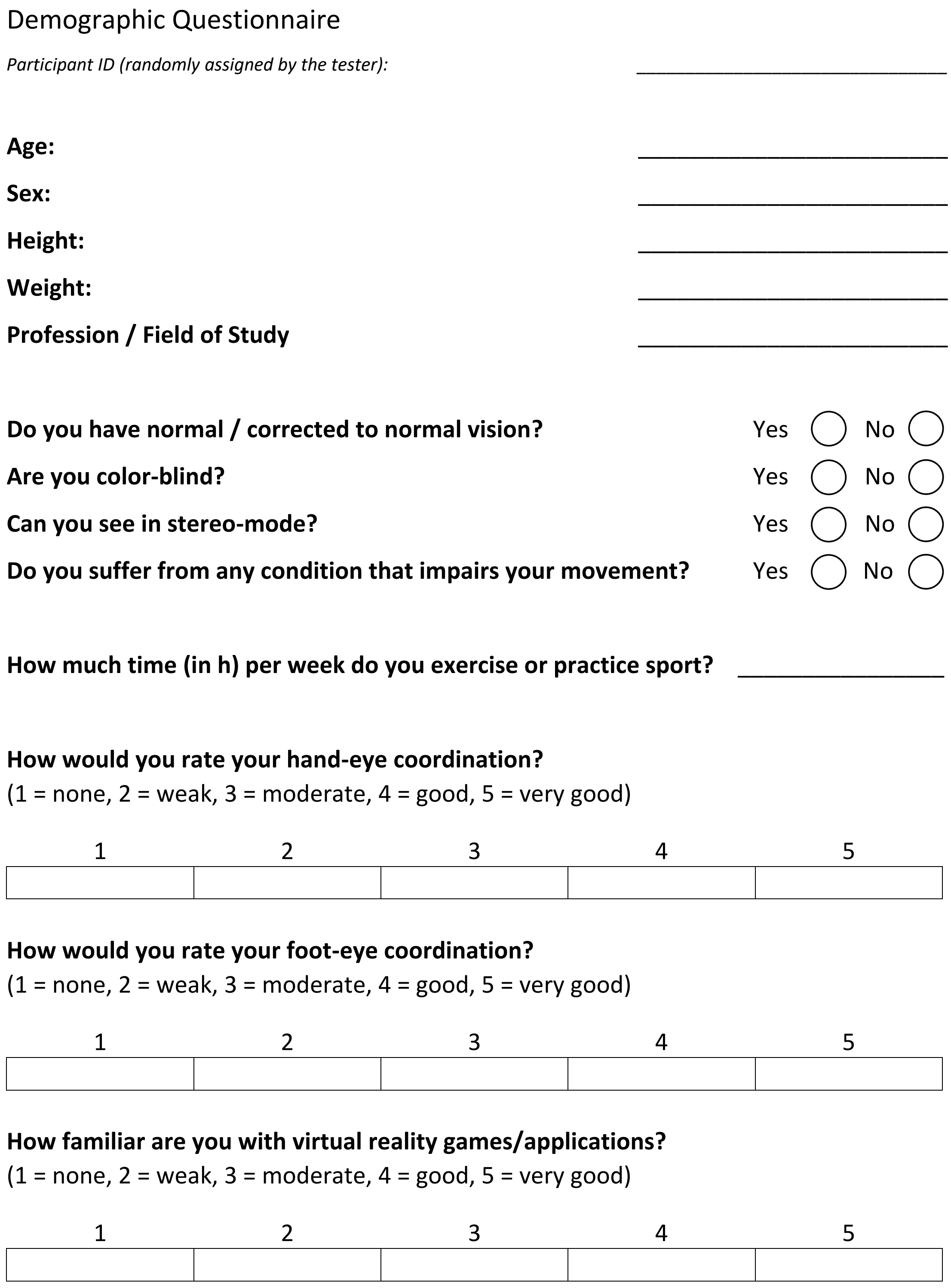
During the foot phase, once you have stepped on the green bar on the floor, the test will pause and your tester will guide you back to the start. Once you have reached the **white square** on the floor the test resumes. After each phase, the scene and your character will freeze for several seconds. Please move as little as possible during that time.

The boxes have physical properties and are immovable. You can touch them and they hinder your movement. It may happen that your finger joints do disconnect from each other when hitting an object.

After each phase you will be verbally asked questions from the embodiment questionnaire (16 questions). You will mostly answer with a number between 1 (strongly disagree) and 7 (strongly agree). Please be as honest and precise as possible! **The procedure will be repeated 3 times** (both hand and foot phases and questions).

If you have any questions right now, please feel free to ask your test operator.

A.2. Demographic Questionnaire



Demographic Questionnaire

Participant ID (randomly assigned by the tester): ______________________________

Age:	______________________________
Sex:	______________________________
Height:	______________________________
Weight:	______________________________
Profession / Field of Study	______________________________

Do you have normal / corrected to normal vision?	Yes ◯ No ◯
Are you color-blind?	Yes ◯ No ◯
Can you see in stereo-mode?	Yes ◯ No ◯
Do you suffer from any condition that impairs your movement?	Yes ◯ No ◯

How much time (in h) per week do you exercise or practice sport? ________________

How would you rate your hand-eye coordination?
(1 = none, 2 = weak, 3 = moderate, 4 = good, 5 = very good)

1	2	3	4	5

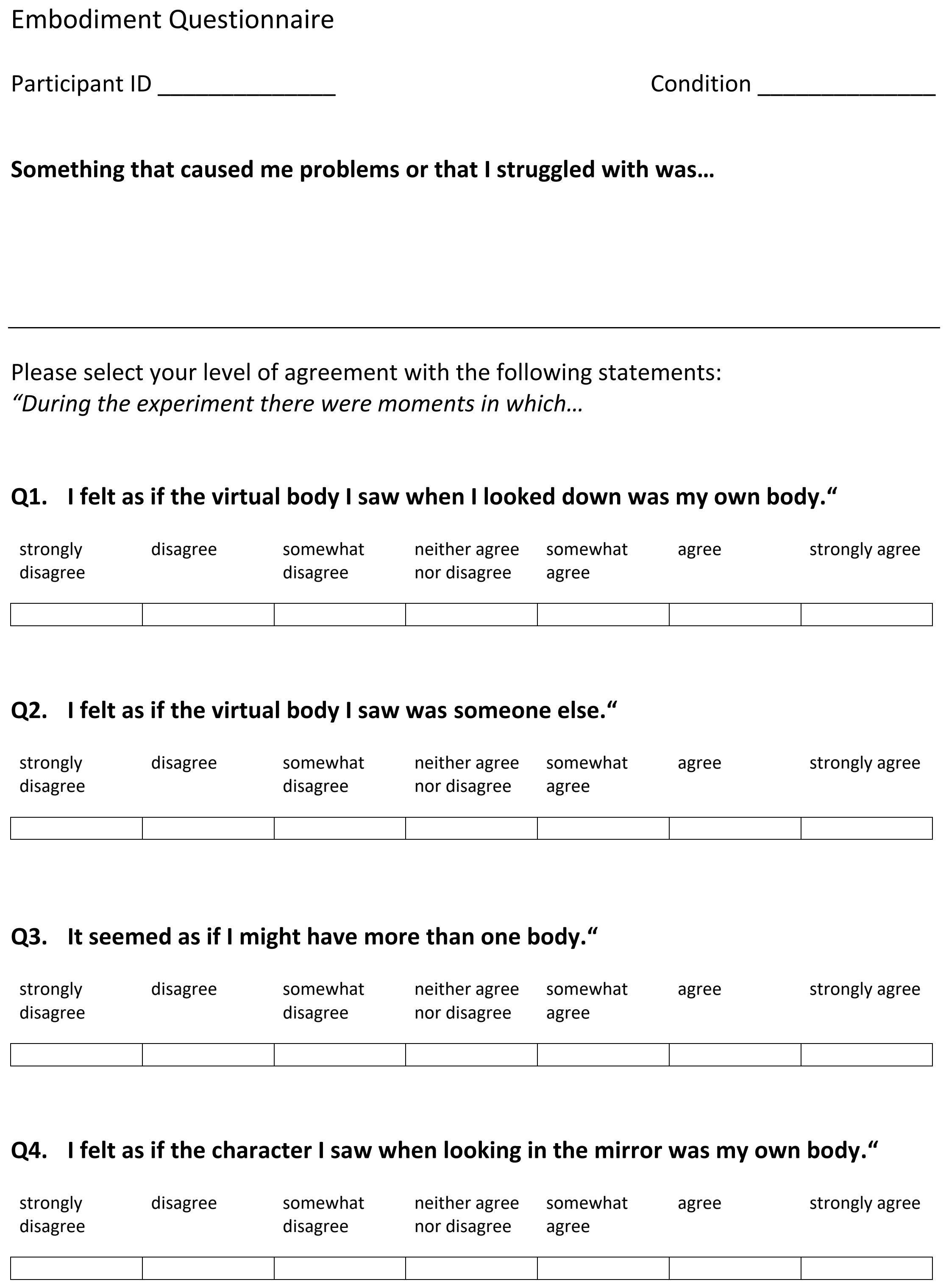
How would you rate your foot-eye coordination?
(1 = none, 2 = weak, 3 = moderate, 4 = good, 5 = very good)

1	2	3	4	5

How familiar are you with virtual reality games/applications?
(1 = none, 2 = weak, 3 = moderate, 4 = good, 5 = very good)

1	2	3	4	5

A.3. Embodiment Questionnaire



Embodiment Questionnaire

Participant ID ______________ Condition ______________

Something that caused me problems or that I struggled with was…

____

Please select your level of agreement with the following statements:
"During the experiment there were moments in which…

Q1. I felt as if the virtual body I saw when I looked down was my own body."

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q2. I felt as if the virtual body I saw was someone else."

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q3. It seemed as if I might have more than one body."

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q4. I felt as if the character I saw when looking in the mirror was my own body."

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q5. I felt as if the character I saw when looking in the mirror was another person.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q6. I felt like I could control the virtual body as if it was my own body.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q7. The movement of the virtual body were caused by my movements.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q8. I felt as if the movements of the virtual body were influencing my own movements.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q9. I felt as if the virtual body was moving by itself.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q10. I felt as if my body was located where I saw the virtual body.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q11. I felt out of my body.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q12. I felt as if my (real) body were drifting towards the virtual body or as if the virtual body were drifting towards my real body.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q13. I felt as if my real body were turning into an ‘avatar‘ body.“

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

Q14. I felt that the motion of the virtual body was simultaneous to my movement.”

strongly disagree	disagree	somewhat disagree	neither agree nor disagree	somewhat agree	agree	strongly agree

List of Figures

List of Tables

Bibliography

[AA13] K. A. Abdel-Malek and J. S. Arora. 'Chapter 2 - Human Modeling: Kinematics'. In: *Human Motion Simulation*. Ed. by K. Abdel-Malek and J. Arora. Burlington: Elsevier Science, 2013, pp. 7–40. ISBN: 978-0-12-405190-4. DOI: `10.1016/B978-0-12-405190-4.00002-7`. URL: `http://www.sciencedirect.com/science/article/pii/B9780124051904000027`.

[AM13] I. Almetwally and M. Mallem. 'Real-time tele-operation and tele-walking of humanoid Robot Nao using Kinect Depth Camera'. In: *10th IEEE International Conference on Networking, Sensing and Control (ICNSC), 2013*. Piscataway, NJ: IEEE, 2013. ISBN: 9781467352000. DOI: `10.1109/icnsc.2013.6548783`. URL: `http://dx.doi.org/10.1109/icnsc.2013.6548783`.

[Ber17] J. Berner. 'Automatic Controller Tuning using Relay-based Model Identification'. eng. PhD thesis. Lund University, Oct. 2017. ISBN: 978-91-7753-447-1. URL: `https://lup.lub.lu.se/search/ws/files/33100749/ThesisJosefinBerner.pdf`.

[Bes15] K. M. Besmer. 'What Robotic Re-embodiment Reveals about Virtual Re-embodiment'. In: *Postphenomenological Investigations* (2015), p. 55.

[Ble] Blender. *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. URL: `https://www.blender.org/` (visited on 06/04/2020).

[Cha+12] M. Cha, S. Han, J. Lee and B. Choi. 'A virtual reality based fire training simulator integrated with fire dynamics data'. In: *Fire Safety Journal* 50 (2012), pp. 12–24. ISSN: 0379-7112. DOI: `10.1016/j.firesaf.2012.01.004`. URL: `http://www.sciencedirect.com/science/article/pii/S0379711212000136`.

[Día18] E. O. Díaz. *3D Motion of Rigid Bodies: A Foundation for Robot Dynamics Analysis*. Springer, 2018. ISBN: 9783030042752.

[Föl+16] O. Föllinger, U. Konigorski, B. Lohmann, G. Roppenecker and A. Trächtler. *Regelungstechnik: Einführung in die Methoden und ihre Anwendung*. 12., überarbeitete Auflage. Lehrbuch Studium. VDE Verlag GMBH, 2016. ISBN: 3800742020.

[GM18] G. Graetz and G. Michaels. 'Robots at work'. In: *Review of Economics and Statistics* 100.5 (2018), pp. 753–768.

[Gof04] F. J. Goforth. 'On motion control design and tuning techniques'. In: (Boston, MA, USA). 2004, 716–721 vol.1. ISBN: 0-7803-8335-4. DOI: `10.23919/ACC.2004.1383689`.

[Gou+09] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre and B. Maisonnier. 'Mechatronic design of NAO humanoid'. In: *2009 IEEE International Conference on Robotics and Automation* (Kobe, Japan). IEEE. 2009, pp. 769–774.

[GP18] M. Gonzalez-Franco and T. C. Peck. 'Avatar Embodiment. Towards a Standardized Questionnaire'. In: *Frontiers in Robotics and AI* 5 (2018), p. 74. ISSN: 2296-9144. DOI: 10.3389/frobt.2018.00074. URL: https://www.frontiersin.org/articles/10.3389/frobt.2018.00074/pdf.

[HBP] HBP. *Neurorobotics Platform*. URL: https://neurorobotics.net/ (visited on 06/04/2020).

[Hor18] S. Hornsey. 'A Review of Relay Auto-tuning Methods for the Tuning of PID-type Controllers.' In: *Reinvention: an International Journal of Undergraduate Research* 11.1 (2018).

[I I00] I. I. Gallagher. 'Gallagher, S. 2000. Philosophical Conceptions of the Self: Implications for Cognitive Science'. In: *Trends in Cognitive Sciences* 4.1 (2000), pp. 14–21. ISSN: 1879-307X. DOI: 10.1016/S1364-6613(99)01417-5.

[Jea03] M. Jeannerod. 'The mechanism of self-recognition in humans'. In: *Behavioural Brain Research* 142.1 (2003), pp. 1–15. ISSN: 0166-4328. DOI: 10.1016/S0166-4328(02)00384-4. URL: http://www.sciencedirect.com/science/article/pii/S0166432802003844.

[KGS12] K. Kilteni, R. Groten and M. Slater. 'The Sense of Embodiment in Virtual Reality'. In: *Presence: Teleoperators and Virtual Environments* 21.4 (2012), pp. 373–387. ISSN: 1054-7460. DOI: 10.1162/PRES_a_00124.

[KNM11] V. Kumar, B. C. N. Nakra and A. Mittal. 'A Review of Classical and Fuzzy PID Controllers'. In: *International Journal of Intelligent Control and Systems* 16 (2011), pp. 170–181.

[Lev96] P. de Leva. 'Adjustments to Zatsiorsky-Seluyanov's segment inertia parameters'. In: *Journal of biomechanics* 29.9 (1996), pp. 1223–1230. ISSN: 0021-9290. DOI: 10.1016/0021-9290(95)00178-6. URL: http://www.sciencedirect.com/science/article/pii/0021929095001786.

[MC99] D. Manocha and J. F. Canny. 'Efficient Inverse Kinematics for General 6R Manipulators'. In: *IEEE Transactions on Robotics and Automation* (1999). ISSN: 1042-296X. URL: https://www.researchgate.net/publication/2237060_Efficient_Inverse_Kinematics_for_General_6R_Manipulators.

[Mic] Microsoft. *Kinect – Entwicklung von Windows-Apps*. URL: https://developer.microsoft.com/de-de/windows/kinect/ (visited on 06/04/2020).

[Mixa] Mixamo. *Mixamo - Doozy*. URL: https://www.mixamo.com/#/?page=1&query=doozy&type=Character (visited on 06/04/2020).

[Mixb] Mixamo. *Mixamo - Y Bot*. URL: `https://www.mixamo.com/#/?page=1&query=ybot&type=Character` (visited on 06/04/2020).

[NVI17] NVIDIA. *Joints — NVIDIA PhysX SDK 3.4.0 Documentation*. 2017. URL: `https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Joints.html#d6-joint` (visited on 06/04/2020).

[Ocu] Oculus. *Oculus | VR-Headsets & Geräte*. URL: `https://www.oculus.com/` (visited on 06/04/2020).

[OM01] M. Oshita and A. Makinouchi. 'A Dynamic Motion Control Technique for Human–like Articulated Figures'. In: *Computer Graphics Forum* 20.3 (2001), pp. 192–203. ISSN: 1467-8659. DOI: `10.1111/1467-8659.00512`. URL: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.00512`.

[OSR] OSRF. *Gazebo*. URL: `http://gazebosim.org/` (visited on 06/04/2020).

[Pre11] H. Preester. 'Technology and the Body: The (Im)Possibilities of Re-embodiment'. In: *Foundations of Science* 16 (2011), pp. 119–137. DOI: `10.1007/s10699-010-9188-5`.

[Rob] S. Robotics. *Softbank Robotics*. URL: `https://www.softbankrobotics.com/emea/en/company` (visited on 06/04/2020).

[Roo19a] RootMotion. *Final IK | Animation | Unity Asset Store*. 2019. URL: `https://assetstore.unity.com/packages/tools/animation/final-ik-14290` (visited on 06/04/2020).

[Roo19b] RootMotion. *PuppetMaster | Physics | Unity Asset Store*. 2019. URL: `https://assetstore.unity.com/packages/tools/physics/puppetmaster-48977` (visited on 06/04/2020).

[RZ04] M. Reuter and S. Zacher. *Regelungstechnik für Ingenieure: Analyse, Simulation und Entwurf von Regelkreisen ; mit 77 Beispielen und 34 Aufgaben*. 11., korrigierte Aufl. Viewegs Fachbücher der Technik. Vieweg, 2004. ISBN: 3528050047.

[Ste] Steam. *SteamVR*. URL: `https://store.steampowered.com/steamvr` (visited on 06/04/2020).

[Ste13] M. Stevenson. *ConfigurableJointExtensions: Unity extension methods for computing a ConfigurableJoint. TargetRotation value from a given local or world rotation*. 2013. URL: `https://gist.github.com/mstevenson/4958837` (visited on 06/04/2020).

[Ste19] Steam. *SteamVR Plugin - Asset Store*. 2019. URL: `https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647` (visited on 06/04/2020).

[Uni] Unity. *Unity Real-Time Development Platform | 3D, 2D VR & AR Visualizations*. URL: `https://unity.com/` (visited on 06/04/2020).

[Uni19a] Unity. *Unity - Manual: 3D Physics Reference*. 2019. URL: `https://docs.unity3d.com/2018.2/Documentation/Manual/Physics3DReference.html` (visited on 06/04/2020).

[Uni19b] Unity. *Unity - Scripting API: Rigidbody*. 2019. URL: `https://docs.unity3d.com/2018.2/Documentation/ScriptReference/Rigidbody.html` (visited on 06/04/2020).

[Viv] Vive. *VIVE | Discover Virtual Reality Beyond Imagination*. URL: `https://www.vive.com` (visited on 06/04/2020).

[Wal+16] T. Waltemate, I. Senna, F. Hülsmann, M. Rohde, S. Kopp, M. Ernst and M. Botsch. 'The impact of latency on perceptual judgments and motor performance in closed-loop interaction in virtual reality'. In: *Proceedings of the 22nd ACM conference on virtual reality software and technology* (Munich, Germany). 2016, pp. 27–35.

[Wan+12] F. Wang, C. Tang, Y. Ou and Y. Xu. 'A real-time human imitation system'. In: *10th World Congress on Intelligent Control and Automation (WCICA), 2012* (Beijing, China). Ed. by D. Cheng. Piscataway, NJ: IEEE, 2012. ISBN: 9781467313988. DOI: `10.1109/wcica.2012.6359088`. URL: `http://dx.doi.org/10.1109/wcica.2012.6359088`.

[Web19] M. Webel. *PID-Tuning Framework for Remotely Operated Humanoid Robots*. 2019. URL: `https://wiki.tum.de/display/infar/MA%3A+PID-Tuning+Framework+for+Remotely+Operated+Humanoid+Robots` (visited on 06/04/2020).

[WHO] WHO. *Novel Coronavirus (2019-nCoV) situation reports*. URL: `https://www.who.int/emergencies/diseases/novel-coronavirus-2019/situation-reports` (visited on 06/04/2020).

[WK19] S. Weber and G. Klinker. 'VR Re-Embodiment in the Neurorobotics Platform'. In: *Mensch und Computer 2019 - Workshopband* (Hamburg, Germany). Bonn: Gesellschaft für Informatik e.V., 2019. DOI: `10.18420/muc2019-ws-585`.

[ZH02] V. B. Zordan and J. K. Hodgins. 'Motion Capture-Driven Simulations That Hit and React'. In: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (San Antonio, Texas, USA). SCA '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 89–96. ISBN: 1581135734. DOI: `10.1145/545261.545276`. URL: `https://doi.org/10.1145/545261.545276`.

[Zha+18] Z. Zhang, Y. Niu, Z. Yan and S. Lin. 'Real-time whole-body imitation by humanoid robots and task-oriented teleoperation using an analytical mapping method and quantitative evaluation'. In: *Applied Sciences* 8.10 (2018), p. 2005.

[ZZ98] V. M. Zatsiorsky and V. M. Zaciorskij. *Kinematics of Human Motion*. Human Kinetics, 1998. ISBN: 9780880116763. URL: `https://books.google.de/books?id=mf4i7G5nXvkC`.