Insitut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Allmandring 19
70569 Stuttgart

Bachelorarbeit

Avatar-Posen-Schätzung aus Kopf- und Handpositionen für kollaborative VR-Umgebungen

Patrick Scheurenbrand

Studiengang:	Softwaretechnik
Prüfer:	Jun. Prof. Dr. Michael Sedlmair
Betreuer:	Dr. Guido Reina, Dr. Sebastian Grottel, Sergej Geringer, M.Sc.
Beginn am:	12. November 2018
Beendet am:	13. Mai 2019

Kurzfassung

Durch die in den letzten Jahren immer besser werdenden Virtual-Reality-Headsets steigt deren Verbreitung, wodurch sie in immer mehr Feldern eingesetzt werden. Darunter gibt es auch kooperative Anwendungsfälle, in welchen sich mehrere Personen gemeinsam in einer virtuellen Umgebung befinden. Sind mehrere Personen in derselben virtuellen Umgebung unterwegs, ist es dabei oft wünschenswert, die anderen Personen auch zu sehen. Da sich bei den meisten aktuellen VR-Headsets die erhobenen Sensorwerte auf die Kopf- und Hand-Positionen sowie deren Orientierung beschränken, ist es nicht direkt möglich die Pose eines Avatars daraus zu generieren. In dieser Bachelorarbeit geht es deshalb darum, aus den erhobenen Sensorwerten eines VR-Headsets eine möglichst realistische Avatar-Pose zu generieren. Dafür wurden in dieser Arbeit bestehende Ansätze untersucht, erweitert und eigene dafür entwickelt. Unter den für die Arbeit entwickelten Ansätzen gehört eine Erweiterung des inverse Kinematik Algorithmus FABRIK, welche es ermöglicht, diesen in der inversen Kinematik für Menschen einzusetzen. Zusätzlich wurde auch noch eine eigene Gangsynthese entwickelt, die speziell für den Einsatz in VR vorgesehen ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Diese Arbeit beschäftigt sich mit dem Problem, aus der Position und der Orientierung des Kopfes und der beiden Hände, eine realistische Avatar-Pose abzuschätzen. Dieses Problem stellt sich im Moment in Virtual Reality Anwendungen, welche die Benutzer mittels Avatars visualisieren wollen. Dies ist besonders in kollaborativen VR-Anwendungen wünschenswert, in denen man mit anderen Benutzern in Kontakt kommt. Das Abschätzen der Pose ist nötig, da sich die erhobenen Sensorwerte in aktuellen VR-Anwendungen meistens auf die Positionen und Orientierungen des VR-Headsets und deren Controllern, beschränken.

Während dieser Arbeit wurde zusammen mit der Firma FARO Scanner Production GmbH gearbeitet, ein weltweit führender Anbieter von industrieller 3D-Messtechnik. Einsatzszenarien finden sich auf allen Längenskalen, von der Prüfung einzelner Bauteile bis zur räumlichen Erfassung ganzer Gebäude in Millimetergenauigkeit. FARO nutzt VR zur immersiven Inspektion und zur Präsentation durch 3D-Scans gewonnener Daten. In einem Prototyp werden hierbei auch kollaborative Szenarien durchgeführt. Die Darstellung der Benutzer beschränkt sich im Moment noch auf simple, starre 3D Modelle des Kopfes und der beiden Hände. Diese Darstellung soll durch komplett animierte 3D Avatare ersetzt werden. Um die erarbeitete Lösung möglichst praxistauglich zu gestalten wurde speziell auf externe Sensoren verzichtet, welche den portablen Einsatz der VR-Anwendung stark einschränken würde. Die Abbildung 1.1 zeigt zwei Beispiele einer VR-Kollaboration mit drei Teilnehmern. Jeder Teilnehmer hat einen „Zeigestrahl“ an seinem rechten Controller der für alle Teilnehmer sichtbar ist. So kann man anderen Teilnehmern Details direkt zeigen. Im rechten Bild zeigt ein virtuelles Tablet Informationen zu einer Messung, die von einem anderen Teilnehmer dieser VR-Sitzung erzeugt wurde.

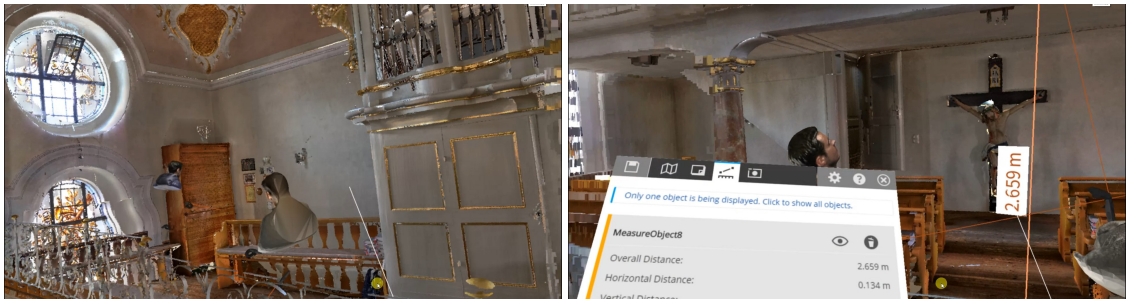

Abbildung 1.1: VR-Kollaboration

Diese Arbeit fängt mit dem Kapitel 2 an, welches Grundlagen erklärt, die zum besseren Verständnis beitragen sollen. Danach wird in Kapitel 3 genauer auf das verwendete Avatar Rig eingegangen und beschrieben, wie der Benutzer sich in der Umgebung bewegt. In Kapitel 4 wird die, in dieser Arbeit verwendete, inverse Kinematik beschrieben, welche für die beiden Arme und die Beine verwendet wird. Kapitel 5 beschäftigt sich mit dem Oberkörper des Avatars und wie dessen Pose aus der Position und der Orientierung des VR-Headsets bestimmt werden kann. Das Kapitel 6 beschreibt

die in dieser Arbeit entwickelte Gangsynthese, die speziell für den Einsatz in VR geeignet ist. Schließlich fasst Kapitel 7 die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit den Grundlagen, welche zum besseren Verständnis dieser Bachelorarbeit beitragen sollen.

2.1 Unity

Für die Implementierung dieser Arbeit wurde die Unity Engine (Version 2018.2.18f1) verwendet. Die Unity Engine ist eine weit verbreitete 3D-Engine, die es ermöglicht, neben Spielen auch beliebige 2D und 3D Anwendungen zu erstellen. Unity unterstützt viele unterschiedliche Plattformen, darunter Windows, Linux und MacOS. Die Anwendungen werden dabei in C# geschrieben. In dieser Arbeit wurde dafür Visual Studio 2017 verwendet. Durch das Verwenden von Unity war es möglich, sich auf den wesentlichen Inhalt dieser Arbeit zu konzentrieren und sich nicht damit beschäftigen zu müssen, wie man bspw. Szenen und Modelle in VR rendert. In Abbildung 2.1 ist der Editor der Unity Engine zu sehen. Das von Unity verwendete Koordinatensystem wird in Abbildung 2.2 dargestellt.

Abbildung 2.1: Die Unity Engine

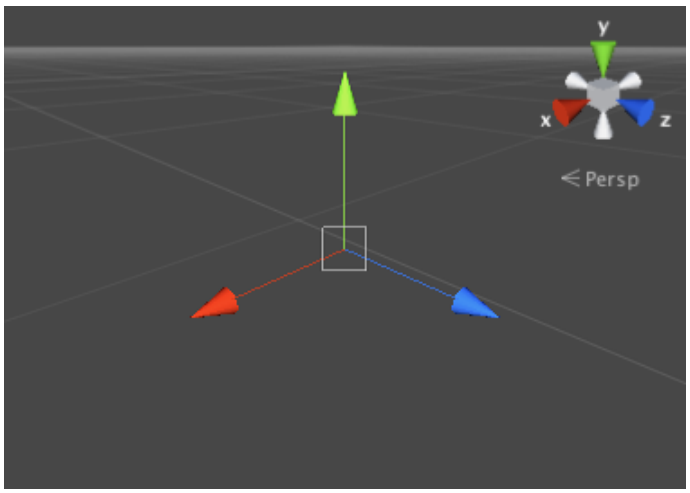

Abbildung 2.2: Das von Unity verwendete Koordinatensystem
Quelle: https://docs.unity3d.com/560/Documentation/Manual/Transforms.html

2.2 AnimationCurve

In der Gangsynthese wurde, der in der Unity Engine vorhandene Curve Editor verwendet. Der Curve Editor erlaubt das Erstellen sogenannter *AnimationCurves*. Eine Kurve liegt in einem zweidimensionalen Koordinatensystem und besteht aus definierbaren Punkten. Jeder Punkt besteht aus einem *time*- und einem *value*-Wert, welcher jeweils für die X- und die Y-Achse steht. Jeder Punkt ist mit dem nächsten Wert, ausgehend von dessen *time*-Wert, in positiver und negativer Richtung verbunden. Zusätzlich besitzt jeder Punkt auch noch die Möglichkeit, die Kurve zum vorherigen und zum nächsten Punkt anzupassen. Mittels der $Evaluate(float\ time)$-Funktion der *AnimationCurve* lässt sich der Y-Wert bzw. der *value*-Wert, zu einem übergebenen X-Wert, bzw. *time*-Wert, finden. Ist der übergebene Wert für *time* kleiner als alle Punkte auf der Kurve, wird der *value*-Wert des Punktes mit dem kleinsten *time*-Wert zurückgegeben. Liegt der übergebene Wert für *time* über allen *time*-Werten der Punkte auf der Kurve, wird der *value*-Wert des Punktes mit dem größten *time*-Wert zurückgeben. Ansonsten liefert die $Evaluate(float\ time)$-Funktion den Wert zurück, welcher im Curve Editor Fenster auf der Kurve angezeigt wird. In Abbildung 2.3 sieht man die für die Gangsynthese verwendete *StepLiftBackwardCurve*. Sie besteht aus sechs Punkten, die jeweils zwischen $time = 0$ und $time = 1$ liegen. Jeder Punkt lässt sich mit der Maus verschieben. Man kann aber auch die beiden Werte für *time* und *value* genau setzen, wie dies für den vierten Punkt zu sehen ist.

2.3 Quaternionen

Unity verwendet für die Darstellung von Rotationen standardmäßig Quaternionen. Deren genaue Funktionsweise muss für diese Arbeit aber nicht verstanden werden. Wichtig ist nur zu wissen, dass es möglich ist, eine Rotation mittels zweier Vektoren darzustellen. In Unity gibt es daher die Funktion $Quaternion.LookRotation(Vector3\ forward, Vector3\ upwards)$. Diese Funktion

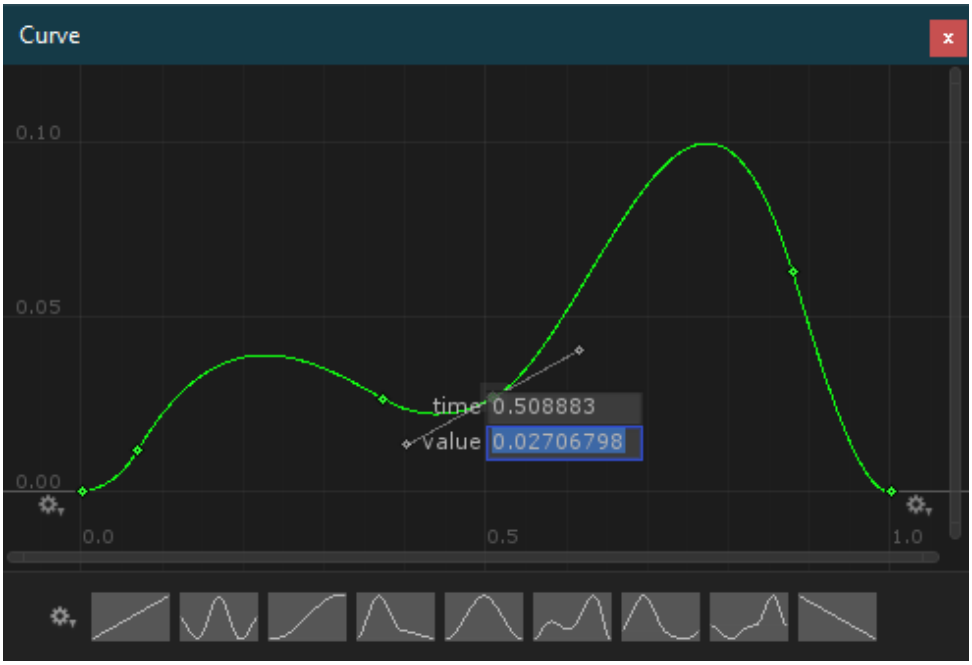

Abbildung 2.3: Unity Cuve Editor

erzeugt dabei aus den zwei 3D Vektoren eine Quaternion. Sie wird bspw. für die Berechnung der Rotation der Knochen verwendet. Dabei bildet der Vektor, welcher von der Start- bis zur Endposition des Knochens verläuft, den $forward$-Vektor. Wie der $upward$-Vektor für die einzelne Knochen berechnet werden kann, wird in späteren Kapiteln erklärt.

2.4 VR-Hardware

Für die Entwicklung dieser Arbeit wurde eine Oculus Rift mit zwei Touch-Controllern verwendet, siehe 2.4. Die Positionen des VR-Headsets und der beiden Controller wird dabei mittels Kameras im 3D-Raum getrackt. Für die Integration in Unity wurde das offizielle Unity Plugin von Oculus verwendet. Dadurch ist es möglich in Unity für VR-Headsets von Oculus, aber auch für SteamVR kompatible VR-Headsets zu entwickeln. Mittels des Plugins werden die Positionen, sowie Orientierungen der beiden Controller, sowie des VR-Headsets in Unity zugänglich gemacht. Die Positionen werden durch einen 3D Vektor mit jeweils einem Wert für X, Y und Z dargestellt. Die Orientierung wird in Unity standardmäßig mittels eines Quaternion zurückgegeben, lässt sich aber auch in Euler Darstellung umwandeln. Das verwendete Unity Plugin hat sich auch um das Rendern der Szene in dem VR-Headset gekümmert, ohne dass dafür speziell etwas angepasst werden musste.

Abbildung 2.4: links: Oculus Rift, **rechts:** Oculus Touch Controller
Quelle: https://en.oculusbrand.com/assets/hardware

2.5 Inverse Kinematik

Inverse Kinematik ist ein aus der Robotik stammender Begriff. Dabei geht es darum, eine Kette an Gelenken so auszurichten, dass das letzte Gelenk die Position und Orientierung eines vorgegebenen Ziels einnimmt. Die inverse Kinematik wird in verschiedenen Feldern eingesetzt, wobei jedes Feld auch unterschiedliche Anforderungen an sie stellt. Dabei muss bspw. ein inverse-Kinematik-Algorithmus, welcher in der Robotik verwendet wird, die genauen Einschränkungen des Roboterarms unterstützen. Die Zeit, die der Algorithmus zum Lösen braucht, ist dabei eventuell aber nicht so wichtig. Bei einem Anwendungsfall, wie in dieser Arbeit, sind andere Anforderungen wichtiger. Da die inverse Kinematik pro Avatar um die 90-mal in der Sekunde (abhängig von verwendetem VR-Headset) ausgeführt wird, muss der Algorithmus schnell sein. Zusätzlich ist es auch wichtig, dass die Lösungen, die der Algorithmus liefert, sich nicht sprunghaft ändern, da dies zu einem für Menschen unrealistischen Verhalten führen würde. Auch ist die hundertprozentige Korrektheit der Lösung von geringerer Relevanz. Bewegt der Benutzer bspw. schnell seinen Arm, ist es unerheblich, wenn die gelieferte Lösung in dieser Zeit wenige Millimeter daneben liegt.

2.6 Funktionen

In dieser Arbeit werden unterschiedliche Funktionen verwendet, diese werden hier einmal kurz erklärt. Alle Funktionen, bis auf $lineIntersection$, sind in der Unity Engine standardmäßig enthalten.

$$clamp(value, min, max) \tag{2.1}$$

Die $clamp$-Funktion schränkt den $value$-Wert ein, damit dieser immer zwischen dem min- und dem max-Wert liegt. Ist $value$ also kleiner als min liefert die Funktion den Wert min zurück. Liegt der Wert von $value$ über dem max-Wert liefert die Funktion den Wert von max zurück. Ansonsten wird der $value$-Wert zurückgegeben.

$$lerp(start,\ end,\ intrpolationValue) \tag{2.2}$$

Die *lerp*-Funktion interpoliert zwei Werte linear. Der Wert für $interpolationValue$ liegt zwischen 0 und 1 und gibt an, wo genau der zurückgegebene Wert zwischen *start* und *end* liegt. Diese Funktion ist nützlich, wenn man einer Variablen einen neuen Wert zuweisen will, dieser Wert aber nicht plötzlich gesetzt werden soll, sondern langsam angenähert werden muss.

$$magnitued(vector) \tag{2.3}$$

Die *magnitude*-Funktion gibt die Länge eines Vektors zurück.

$$normalize(vector) \tag{2.4}$$

Die *normalize*-Funktion normalisiert einen Eingabevektor ($vector$) und gibt diesen zurück. Ein normalisierter Vektor besitzt dieselbe Richtung, wobei die Länge aber immer 1 ist.

$$sign(value) \tag{2.5}$$

Die *sign*-Funktion liefert das Vorzeichen der Eingabe in Form von -1 oder 1 zurück.

$$min(first,\ second) \tag{2.6}$$

Die *min*-Funktion liefert die kleinere der beiden Eingangsvariablen ($first$, $second$) zurück.

$$lineIntersection(position_{first},\ direction_{first},\ position_{second},\ direction_{second}) \tag{2.7}$$

Die $lineIntersection$-Funktion liefert einen 2D Punkt zurück, welcher den Schnittpunkt der beiden Eingabelinien enthält. Die Linien werden jeweils mit einem Punkt ($position$) und einer Richtung ($direction$) dargestellt.

$$moveTowards(current,\ target,\ maxDistanceDelta) \tag{2.8}$$

Die $moveTowards$-Funktion liefert einen 3D Vektor, welcher von dem $current$-Vektor ausgeht und sich in Richtung von $target$ bewegt. Dabei ist die Distanz zwischen $current$ und dem Rückgabewert der Funktion maximal so groß wie $maxDistanceDelta$.

$$abs(value) \tag{2.9}$$

Die *abs*-Funktion liefert den absoluten Betrag der Eingabe zurück.

2.7 Verwandte Arbeiten

Für die Avatar-Posen-Schätzung gibt es bereits einige unterschiedliche Ansätze. Ein Teil dieser Ansätze wird in dieser Sektion vorgestellt.

Für die Unity Engine existiert das Final IK Plugin [7]. Dieses besitzt eine Komponente (VRIK), welche eine Lösung der Avatar-Posen-Schätzung für VR bietet. Das Produkt ist kommerziell im Unity Store erwerblich und eine genaue Beschreibung der Funktionsweise ist daher nicht verfügbar. Neben den beiden Controllern und dem VR-Headset besteht dort auch noch die Möglichkeit, weitere Trackingpunkte in die Posen-Schätzung des Avatars zu integrieren. Somit ist es möglich, die Posen des Unterkörpers mittels Trackingsensoren wie den Vive Trackern [1] zu verbessern.

Das Paper [5] beschreibt eine Möglichkeit der Posen-Schätzung ausgehend von den Positionen und Orientierungen eines VR-Headsets und den zwei Controllern. Dabei wird der Körper in zwei Teile aufgeteilt, für welche die Posen-Schätzung separat gelöst wird. Für die obere Hälfte kann die Pose des Avatars recht genau abgeschätzt werden, da für den Oberkörper die Positionen der Controller und des VR-Headsets gegeben sind. Für den Unterkörper wird die Pose mittels Animationsblending erzeugt.

Das Paper [9] beschreibt eine Methode der Posen-Schätzung der Arme, ausgehend von den Positionen und Orientierungen des VR-Headsets und den zwei Controllern. Dabei wird die vorgestellte Methode auch noch in einer Studie, mit Motion Capturing und dem alleinigen Darstellen der Hände, verglichen.

Im Paper [11] wird die Pose des Avatars mittels der Noitom perception neuron gloves getrackt. Dies macht es möglich, die genauen Positionen der Finger und der Arme zu tracken. Diese Methode des Trackings bietet aktuell aber noch einige Probleme. Einerseits muss das Tracking Equipment erst aufwendig vom Benutzer angelegt werden. Zusätzlich muss es während der Benutzung, regelmäßig kalibriert werden, um weiterhin genaue Positionen zu liefern. In dem Paper wird auch noch auf die Posen-Schätzung der Arme mittels inverser Kinematik eingegangen. Wie genau deren Implementierung funktioniert, wird jedoch nicht beschrieben.

Im Paper [4] wird eine Methode des Posen-Trackings mittels zwei Kinect-Sensoren beschrieben. Dies bietet den Vorteil, dass der Benutzer für die Verwendung des Systems nicht umständlich Tracking-Sensoren anlegen muss. Zusätzlich sind die zwei verwendeten Kinect-Sensoren, im Vergleich zu anderen nicht für Normalverbraucher ausgelegten Tracking-Systeme, deutlich preiswerter.

[1] https://www.vive.com/eu/vive-tracker/

3 Avatar

Der in dieser Arbeit verwendete Avatar wurde aus dem Unity Store kostenlos entnommen, ist aber zum aktuellen Zeitpunkt dort nicht mehr verfügbar. Der Avatar besitzt ein 3D Model, ist texturiert, gerigged und auch schon geskinnt. Für die Arbeit war es daher einfach möglich, die Pose des Avatars anzupassen, indem die Position und Orientierung einzelner Knochen gesetzt wurde.

3.1 Rig

In dieser Arbeit wird für den Avatar ein recht simples Rig verwendet, wie es in Spieleengines üblich ist. Die Wirbelsäule, welche im menschlichen Körper aus 24 beweglichen Wirbeln besteht, wird hier bspw. nur durch vier Knochen angenähert. Die in dieser Arbeit verwendeten Knochen aus dem Rig sind unten aufgeführt und jeweils mit ihren Nachbarn verbunden:

- Hip - Spine1 - Spine2 - Spine3 - Neck - Head
- Hip - LeftUpLeg - LeftLeg - LeftFoot
- Hip - RightUpLeg - RightLeg - RightFoot
- Spine3 - LeftShoulder - LeftUpperArm - LeftLowerArm - LeftHand
- Spine3 - RightShoulder - RightUpperArm - RightLowerArm - RightHand

Der *Hip*-Knochen ist in dem Rig die Basis, mit welcher alle anderen Knochen direkt oder indirekt verbunden sind. Wird der *Hip*-Knochen bewegt, bewegt sich daher auch gleich der komplette Avatar mit. In der Abbildung 3.1 ist der Avatar mit seinen Knochen von vorne und von der Seite zu sehen. Die Fingerknochen wurden in der Arbeit nicht verwendet.

3.2 Knochen

Jeder Knochen besteht aus einem Positionswert und einem Orientierungswert. Der Positionswert wird dabei mittels eines 3D Vektors dargestellt und der Orientierungswert mittels eines Quaternion. Ändert sich die Position bzw. die Orientierung eines Knochens, so bewegen sich seine Kinderknochen dabei automatisch mit. Dies muss beachtet werden, da bspw. beim Setzten der *Spine*-Knochen unterschiedliche Reihenfolgen zu unterschiedlichen Posen führen können. Deshalb wird die *Spine* Rotation immer ausgehend von dem *Hip*-Knochen gesetzt, um unerwünschte Rotationsänderungen der Kinderknochen zu verhindern.

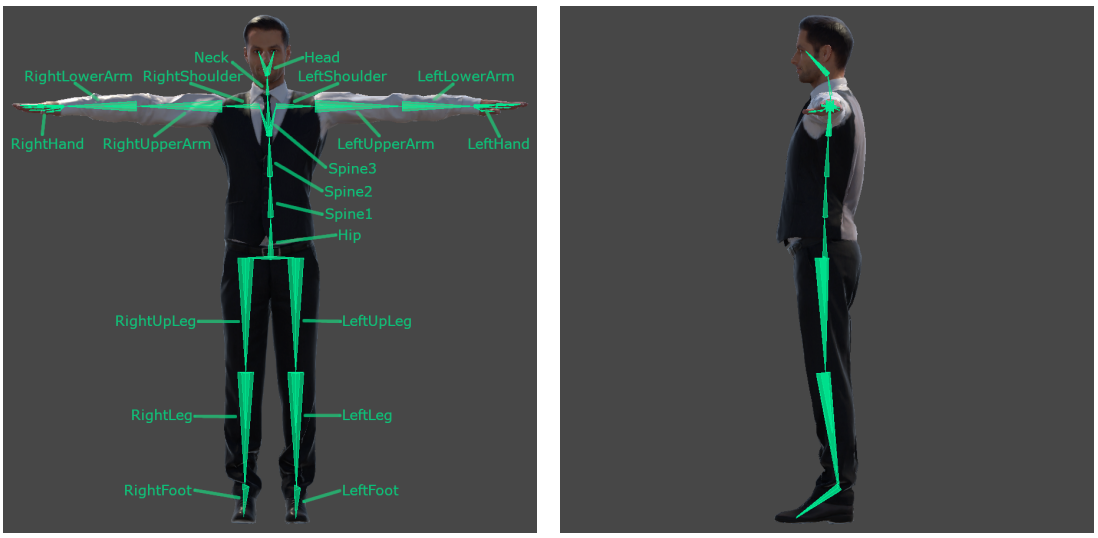

Abbildung 3.1: links: Avatar von vorne; **rechts:** Avatar von der Seite

3.3 Player Controller

Der Benutzer kann sich in der virtuellen Welt bewegen, indem er sich in der echten Welt bewegt. Alternativ kann er sich auch mittels eines Controllers bewegen. Dabei befindet sich um den Avatar ein sphärischer Collider, welcher sich mit dem Avatar bewegt und diesen daran hindert durch Objekte hindurch zu laufen. Bewegt sich der Avatar auf ein Objekt zu, wird der Avatar mittels des Colliders über das Objekt geschoben, falls das Objekt nicht zu hoch ist. Dies ist in Abbildung 3.2 zu sehen. Im linken Bild befindet sich der Collider mit dem Avatar auf dem Boden. Im rechten Bild hat der Avatar sich in Richtung der orangenen Box bewegt, wobei der Collider dafür gesorgt hat, dass der Avatar nicht durch die Box durchläuft, sondern darauf positioniert wird. Auf den Collider wirkt zusätzlich noch eine Schwerkraft, welche dafür sorgt, dass der Avatar immer wieder auf den Boden zurück bewegt wird. Der unterste Y-Punkt der Sphäre wird als $floor_{pos}.y$ angesehen und für die Krümmung des Oberkörpers in 5.4 verwendet.

Die $TargetOffset$-Position befindet sich auch unterhalb der Sphäre und besitzt die Positionen des VR-Headsets und der Controller als Kinder. Wird der $TargetOffset$ bewegt, bewegen sich die Positionen des VR-Headsets und der Controller mit. Dies wird in Sektion 6.10 verwendet, um die Position des VR-Headsets so zu korrigieren, dass beide Beine in der Lage sind, den Boden zu berühren.

Der $forwardDirection$-Vektor ist der Vektor, der in die Richtung zeigt in welche der Avatar steht. Dieser wird in Kapitel 6 für die Gangsynthese verwendet.

Der Avatar besitzt zusätzlich noch einen $Center\ of\ Mass$-Punkt (CoM) welcher in der Gangsynthese, in Kapitel 6 Verwendung findet. In Abbildung 3.3 ist der Punkt in der Brust des Avatars zu sehen. Der CoM bewegt sich mit der Brust des Avatars mit und ist daher nur eine sehr simple Abschätzung des reellen CoM.

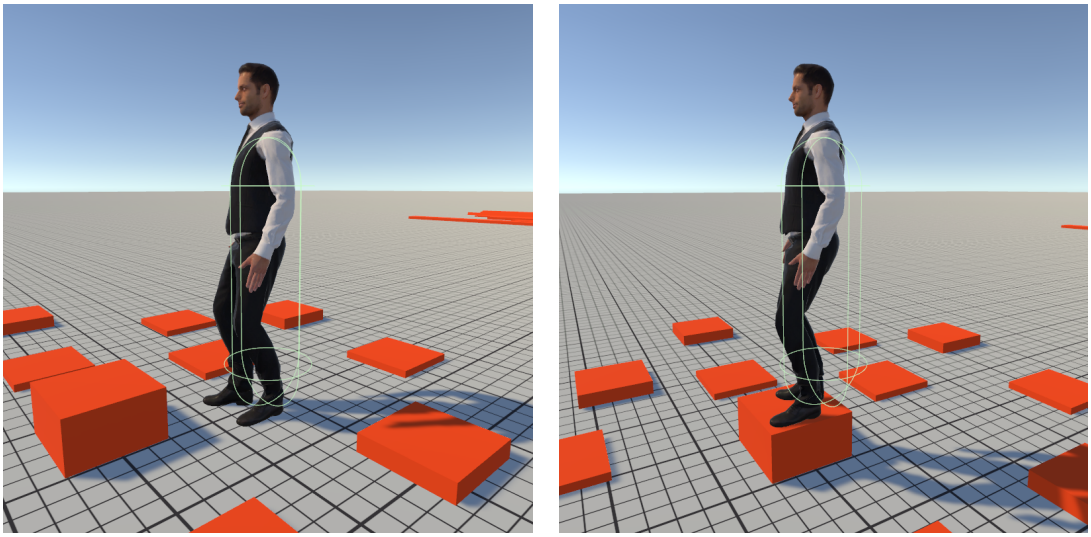

Abbildung 3.2: Visualisierung der Kollisionssphäre des Avatars

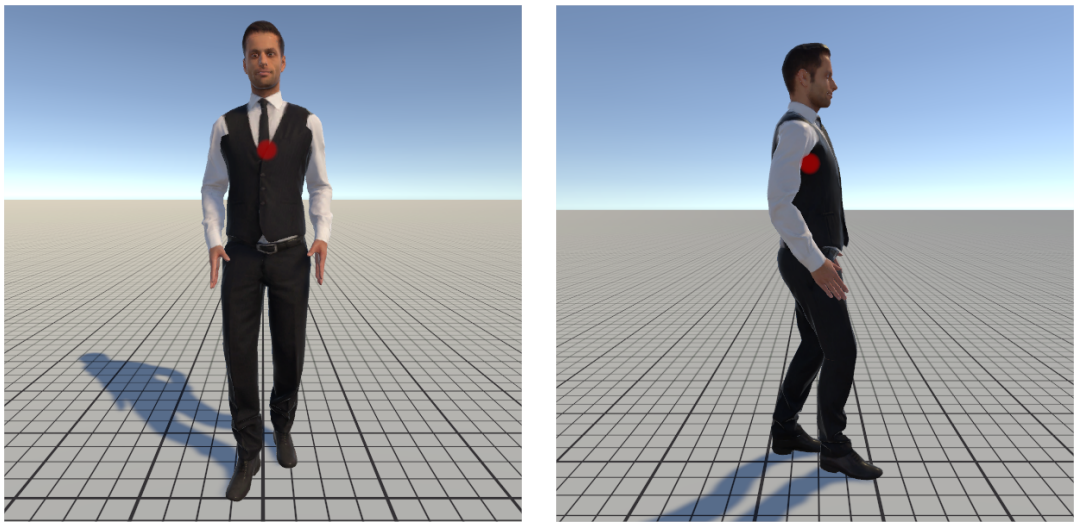

Abbildung 3.3: Center of Mass des Avatars

4 Inverse Kinematik

In dieser Arbeit wird die inverse Kinematik einerseits für die beiden Arme verwendet, aber auch in Zusammenarbeit mit der Gangsynthese für die beiden Beine. Zur Lösung der inversen Kinematik wurde der FABRIK Algorithmus ausgewählt. Dieser Algorithmus ist schnell, produziert natürliche Posen und ist einfach zu implementieren. Die genaue Funktionsweise des Algorithmus, wie sie aus [2] übernommen wurde, wird in Sektion 4.1 beschrieben. Für die Arme werden Constraints verwendet, welche in den Sektionen 4.2 und 4.3 genauer beschrieben werden. Zusätzlich dazu wurde der Algorithmus in dieser Arbeit noch um Magnetfelder erweitert, welche es einem erlauben, die durch den Algorithmus berechneten Posen zu beeinflussen. Wie dies genau geschieht, wird in Sektion 4.4 beschrieben. Für die Schulter wurde in dieser Arbeit noch eine Erweiterung des FABRIK Algorithmus entwickelt, welche in Sektion 4.5 beschrieben wird. Da der FABRIK Algorithmus das IK Problem auf Punkte und Linien herunterbricht und dadurch die Orientierungen der einzelnen Glieder verloren geht, werden diese mittels dem in 4.6 beschriebenen Verfahren ermittelt. Wie genau die inverse Kinematik für die Beine funktioniert, wird in der Sektion 4.7 dieses Kapitels beschrieben.

4.1 FABRIK

Der FABRIK Algorithmus verfolgt einen iterativen Ansatz, in welchem nacheinander die Gelenkpositionen aktualisiert werden, um so in jeder Iteration näher an die Zielposition zu gelangen. Im ersten Schritt wird zunächst geschaut, ob der Abstand von der Startposition der Gelenkkette bis zur Zielposition kleiner als die Summe der einzelnen Knochenlängen ist und das Ziel somit erreichbar. Falls dies nicht der Fall ist, wird die Gelenkkette in Richtung des Ziels voll ausgestreckt, um so möglichst nah an der Zielposition zu liegen. Falls der Abstand zwischen der Start- und der Zielposition kleiner ist als die Summe der Knochenlängen, ist das Ziel erreichbar und der iterative Algorithmus wird ausgeführt. Der Algorithmus besteht aus zwei Phasen, der Forward Reaching und der Backward Reaching Phase. In der Forward Reaching Phase werden die Gelenkpositionen ausgehend von dem Ende der Gelenkkette aktualisiert und zur Zielposition gebracht. Nach der Forward Reaching Phase liegt die Endposition der Gelenkkette auf der Zielposition. Die Startposition der Gelenkkette liegt aber möglicherweise nicht mehr auf ihrer vorgegebenen Startposition. Da sich die Startposition nicht ändern darf, gibt es die Backward Reaching Phase, welche ausgehend von der Startposition, die Gelenkpositionen so anpasst, dass die Startposition wieder an der richtigen Stelle liegt. Diese zwei Phasen werden so lange nacheinander ausgeführt, bis der Abstand der Position des letzten Gelenks der Gelenkkette und die Zielposition kleiner als eine vorgegebene Toleranz sind oder ein vorgeschriebenes Iterationslimit überschritten wurde.

In Abbildung 4.1 ist eine volle Iteration des FABRIK Algorithmus zu sehen. Die Gelenkkette besteht aus drei Knochen mit der Länge d_i für $i = 1, 2, 3$ und vier Gelenken mit der Position p_i für $i = 1, ..., 4$. In Abbildung **(a)** ist die initiale Position der Gelenkkette zu sehen und die Position des

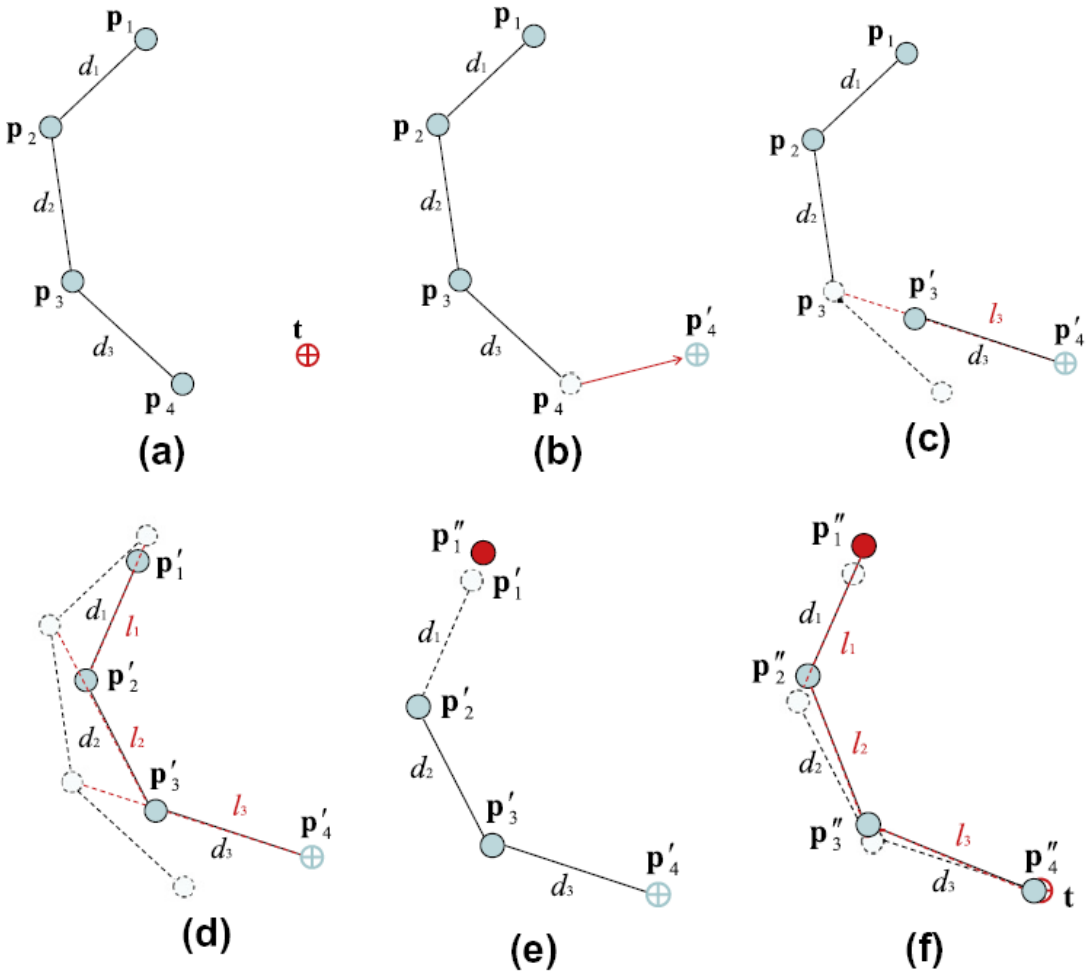

Abbildung 4.1: Beispiel einer ganzen Iteration des FABRIK Algorithmus
Quelle: [1]

Ziels **t**. In den Abbildungen **(b)** bis **(d)** ist die Forward Reaching Phase zu sehen. Angefangen mit dem Setzen von p'_4 auf die Zielposition **t** in **(b)**. Die Position p'_3 wird anschließend in **(c)** aus der Position p'_4, der Position p_3 und der Länge des Gelenks d_3 berechnet. p'_3 liegt mit dem Abstand d_3 von p'_4 auf der Linie l_3 welche von p'_4 in Richtung p_3 verläuft. Mit demselben Verfahren werden auch die Positionen der Gelenke p_1 und p_2 angepasst. In der Abbildung **(d)** ist das Resultat der Forward Reaching Phase zu sehen. p'_4 liegt auf der Zielposition und die Gelenkpositionen p'_i für $i = 1, 2, 3$ wurden so angepasst, dass sich die Knochenlängen l_i für $i = 1, 2, 3$ nicht ändern.

In Abbildung **(e)** ist der erste Schritt der Backward Reaching Phase zu sehen. In ihr wird die Position p''_1 wieder zurück auf die Startposition p_1 gesetzt. Anschließend werden die Positionen p''_i für $i = 2, 3, 4$ berechnet, welche jeweils von p''_{i-1} ausgehen und durch p'_i verlaufen. Das Resultat der Backward Reaching Phase ist in Abbildung **(f)** zu sehen. p''_1 liegt auf der Startposition und das letzte Gelenk der Gelenkkette p''_4 liegt deutlich näher am Ziel als noch vor dem Ausführen des FABRIK Algorithmus in Abbildung **(a)**. Im Pseudocode 4.1 wird der genaue Ablauf des Algorithmus beschrieben.

Algorithmus 4.1 Eine Iteration des FABRIK Algorithmus

Eingabe: Die Gelenkpositionen p_i für $i = 1, ..., n$, die Zielposition t und den Abstand zwischen den Gelenken $d_i = |p_{i+1} - p_i|$ für $i = 1, ..., n-1$
Ausgabe: Die neuen Gelenkpositionen p_i für $i = 1, ..., n$

% Der Abstand zwischen der Start- und Zielposition
$dist \leftarrow |p_1 - t|$
% schaut ob die Zielposition in Reichweite ist
if $dist > d_1 + d_2 + ... + d_{n-1}$ **then**
    % Das Ziel ist nicht in Reichweite
    for $i \leftarrow 1, n-1$ **do**
        % berechne den Abstand r_i zwischen der Zielposition t und der Gelenkposition p_i
        $r_i \leftarrow |t - p_i|$
        $\lambda_i \leftarrow d_i / r_i$
        % berechne die neue Gelenkposition p_i
        $p_{i+1} \leftarrow (1 - \lambda_i) p_i + \lambda_i t$
    end for
else
    % Das Ziel ist in Reichweite; setze b als die initiale Gelenkposition p_1
    $b \leftarrow p_1$
    % überprüfe ob der Abstand zwischen dem letzten Gelenk p_n und der Zielposition t größer ist als die gesetzte Toleranz
    $dif_A \leftarrow |p_n - t|$
    while $dif_A > tol$ **do**
        % Phase 1: FORWARD REACHING
        % setzte die Position des letzten Gelenks p_n auf die Position t
        $p_n \leftarrow t$
        for $i \leftarrow n-1, 1$ **do**
            % berechne den Abstand r_i zwischen der neuen Gelenkposition p_{i+1} und p_i
            $r_i \leftarrow |p_{i+1} - p_i|$
            $\lambda_i \leftarrow d_i / r_i$
            % berechne die neue Gelenkposition p_i
            $p_i \leftarrow (1 - \lambda_i) p_{i+1} + \lambda_i p_i$
        end for
        % Phase 2: BACKWARD REACHING
        % setze die Startposition b auf die Position des ersten Gelenks p_1
        $p_1 \leftarrow b$
        for $i \leftarrow 1, n-1$ **do**
            % berechne den Abstand r_i zwischen der neuen Gelenkposition p_i und p_{i+1}
            $r_i \leftarrow |p_{i+1} - p_i|$
            $\lambda_i \leftarrow d_i / r_i$
            % berechne die neue Gelenkposition p_i
            $p_{i+1} \leftarrow (1 - \lambda_i) p_i + \lambda_i p_{i+1}$
        end for
        $dif_A \leftarrow |p_n - t|$
    end while
end if

4.2 Constraints

Im oben beschriebenen FABRIK Algorithmus sind noch keine Constraints enthalten. Da Gelenke am menschlichen Körper keine unendliche Bewegungsfreiheit besitzen, ist es sinnvoll, Constraints in den FABRIK Algorithmus einzubringen. Dies ist möglich, indem man in jedem Schritt des Algorithmus schaut, ob die neu berechnete Position die vorgegebenen Constraints erfüllt. Falls dies nicht der Fall ist, wird die berechnete Position so angepasst, dass diese die vorgegebene Constraints erfüllt. Eine Möglichkeit wie dies geschehen kann wird in [1] beschrieben. Dafür wird das 3D Problem auf ein 2D Problem heruntergebrochen. Die dort beschriebene Methode projiziert die neu berechnete Position eines Gelenkes auf eine 2D Ebene. Auf dieser Ebene wird eine Fläche mittels vier Ellipsen beschrieben. Diese vier Ellipsen beschreiben die mögliche $Range\ of\ Motion$ des Gelenkes. Falls sich der projizierte Punkt t innerhalb der durch die vier Ellipsen beschriebenen Fläche befindet, liegt der Punkt innerhalb der erlaubten $Range\ of\ Motion$ und der FABRIK Algorithmus kann weiter ausgeführt werden. Falls der Punkt außerhalb der erlaubten Fläche liegt, wird ein neuer Punkt berechnet, welcher möglichst nah am Punkt t liegt, aber sich noch innerhalb der erlaubten Fläche befindet. Dieser Punkt wird nun wieder von der 2D Projektion zurück in die 3D Ebene gebracht und anschließend im FABRIK Algorithmus verwendet.

Mittels des Abstands S und dem Winkeln θ_i für $i = 1, ..., 4$ können die Werte q_i für $i = 1, ..., 4$ berechnet werden, welche die Ellipsen auf der 2D Ebene beschreiben. Dies ist in Abbildung 4.2 zu sehen. Dabei beschreiben jeweils zwei nebeneinander liegende q_i-Werte die Fläche einer Ellipse. Je nachdem in welchem Bereich sich t befindet wird jeweils die passende Ellipse verwendet um zu prüfen, ob t sich innerhalb oder außerhalb dieser befindet, siehe 4.3. Überprüft wird dies mittels der Formel, in welcher q_w und q_h die jeweiligen Maße der Ellipse sind in welcher sich t befindet:

$$d = \frac{t_x^2}{q_w^2} + \frac{t_y^2}{q_h^2} \tag{4.1}$$

Ist d kleiner oder gleich 1, liegt der Punkt t innerhalb der Ellipse und somit innerhalb der $Range\ of\ Motion$ des Gelenks. Ist d größer als 1, liegt t somit außerhalb der Ellipse und damit außerhalb der gegebenen $Range\ of\ Motion$. Falls der Wert außerhalb der $Range\ of\ Motion$ liegt, muss t' berechnet werden, welcher den neuen korrigierten Wert von t darstellt, welcher innerhalb der Ellipse und somit auch innerhalb der $Range\ of\ Motion$ liegt. Um t' zu berechnen wird zunächst wie in Formel 4.2 der Winkel α_t berechnet, welcher den Winkel zu t darstellt. Solange die Breite q_w und die Höhe q_h gleich sind, und die Ellipse somit einem Kreis entspricht, sind α_t und ϕ gleich. Falls dies nicht der Fall ist, wird ϕ aus mit den Werten q_w, q_h und α_t wie in Formel 4.3 berechnet[1]. Anschließend kann aus dem Winkel ϕ die Position $t^{'}$, bestehend aus den zwei Werten für t'_x und t'_y, berechnet werden. Dies ist in den Formeln 4.4 und 4.5 zu sehen. Die zwei Werte t'_x und t'_y beschreiben die eingeschränkte Position des Gelenks auf der 2D Ebene. In Abbildung 4.4 ist die Berechnung von t' visualisiert. Um nun die Position im 3D Raum zu finden, müssen die Transformationen rückgängig gemacht werden, welche zuvor t auf die 2D Ebene projiziert haben. Der genaue Ablauf des Verfahrens ist in Algorithmus 4.2 zu sehen.

[1] http://www.petercollingridge.co.uk/tutorials/computational-geometry/finding-angle-around-ellipse/

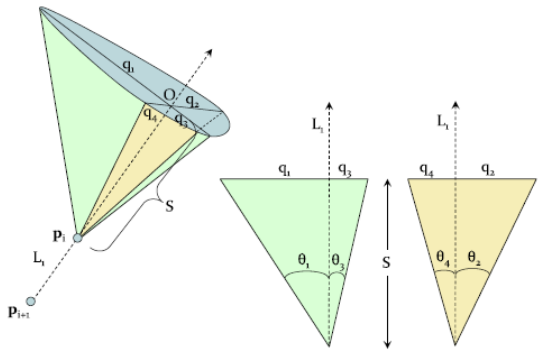

Abbildung 4.2: Ellipse Contraints
Quelle: [1]

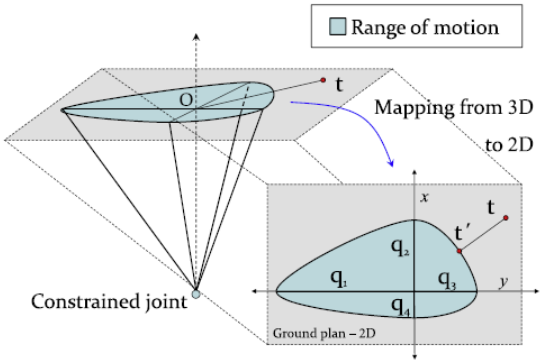

Abbildung 4.3: Constraint enforcement
Quelle: [1]

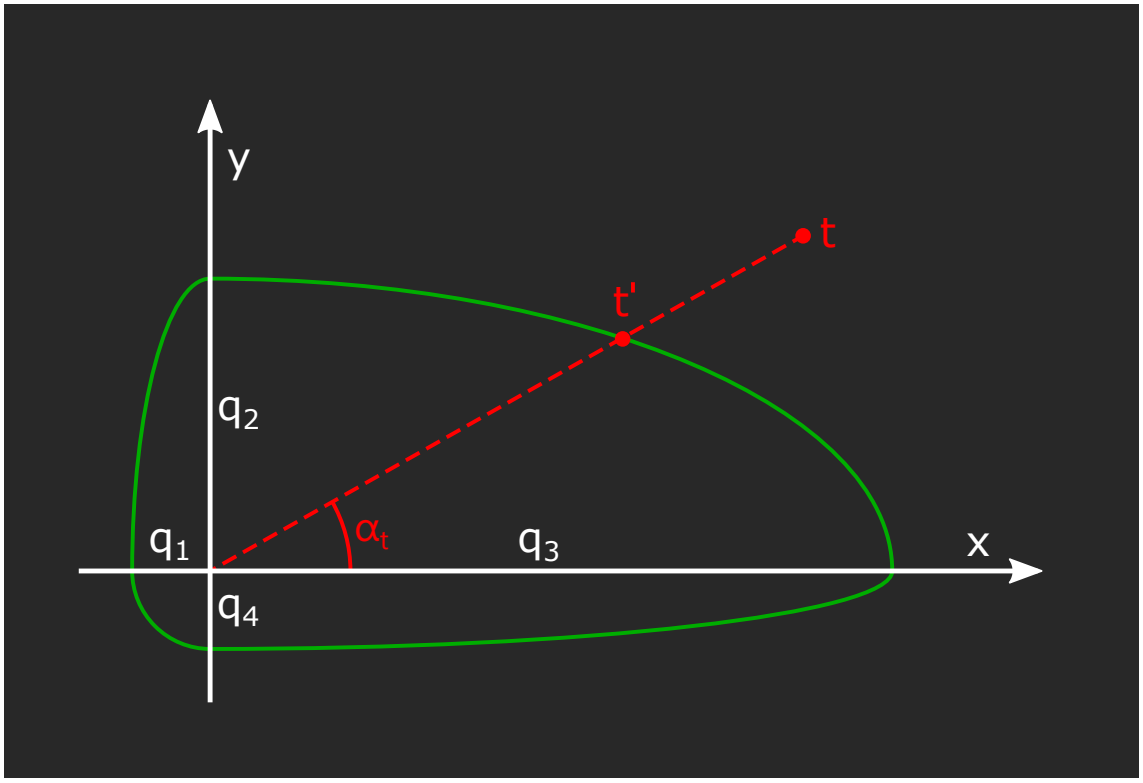

Abbildung 4.4: Berechnung der neuen Position von t

$$\alpha_t = \text{atan2}(t_y, t_x) \tag{4.2}$$

$$\phi = \text{atan}(\frac{q_w}{q_h} \cdot tan(\alpha)) \tag{4.3}$$

$$t'_x = q_w \cdot cos(\phi) \tag{4.4}$$

$$t'_y = q_h \cdot sin(\phi) \tag{4.5}$$

Algorithmus 4.2 Ellipse Constraints Algorithmus

% $Eingabe: \ forwardDirection, \ boneLength, \theta_1, \theta_2, \theta_3, \theta_4$
% $Ausgabe: \ constraintDirection$
$t \leftarrow project(forwardDirection, \ Vector3.forward)$
% Abstand der forwardDirection zur Ebene mit der Normalen Vector3.forward
$s \leftarrow distance(forwardDirection, \ Plane(Vector3.forward))$
$q_1 \leftarrow |s| * tan(\theta_1)$
$q_2 \leftarrow |s| * tan(\theta_2)$
$q_3 \leftarrow |s| * tan(\theta_3)$
$q_4 \leftarrow |s| * tan(\theta_4)$
q_w, q_h
% Schauen welche der vier Ellipsen verwendet wird
if $t_x > 0$ **then** $q_w \leftarrow q_2$
    if $t_y > 0$ **then**
        $q_h \leftarrow q_3$
    else
        $q_h \leftarrow q_1$
    end if
else
    $q_w \leftarrow q_4$
    if $t_y > 0$ **then**
        $q_h \leftarrow q_3$
    else
        $q_h \leftarrow q_1$
    end if
end if
$d \leftarrow t_x^2/q_w^2 + t_y^2/q_h^2$
% liegt t außerhalb von der Ellipse oder auf der falschen Seite?
if $d > 1$ or $s < 0$ **then**
    % Der forwardDirection Vektor liegt außerhalb des Constraints
    $angle \leftarrow Atan2(t_y, t_x)$
    $angle_ellipse \leftarrow Atan(q_w/q_h * tan(angle))$
    $constraintDirection \leftarrow Vector3(q_w * Cos(angle_ellipse), q_h * Sin(angle_ellipse), |s|).normalized * boneLength$
else
    % Der forwardDirection Vektor liegt innerhalb des Constraints
    $return \ forwardDirection$
end if

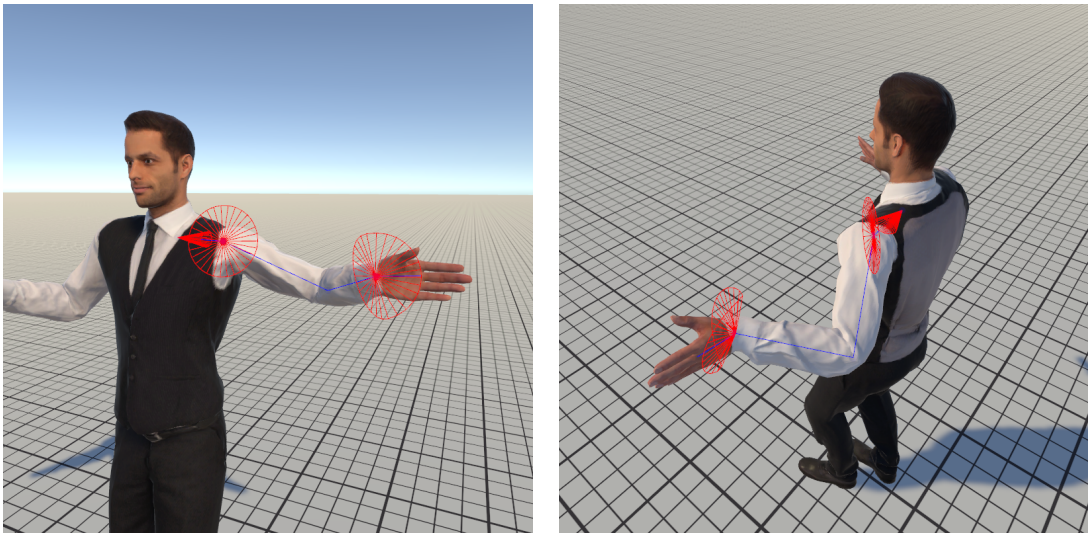

Abbildung 4.5: Visualisierung der für den linken Arm verwendeten Constraints

4.3 Arm Constraints

Die in der vorherigen Sektion beschriebenen Constraints können in jeder Iteration des FABRIK Algorithmus angewandt werden. In dieser Arbeit wurden Constraints aber nur begrenzt für die Arme verwendet, da der FABRIK Algorithmus besser funktioniert, je weniger Constraints verwendet werden. In Abbildung 4.5 sind die drei verwendeten Constraints in Form der roten kegelförmigen Linien zu sehen. Das Constraint im Schultergelenk erlaubt nur einen kleinen Bewegungsfreiraum und wird nur in der Backward Reaching Phase des FABRIK Algorithmus erzwungen. Die sonst sehr komplexe Schulter wird damit recht simpel angenähert, um Posen wie das nach vorne Ausstrecken des Arms zu unterstützen. Das Constraint des Oberarmgelenks wird genau wie das Schultergelenk in der Backward Reaching Phase angewandt. Das Constraint des Handgelenkes hingegen wird in der Forward Reaching Phase des Algorithmus angewandt. Constraints die in der Backward Phase erzwungenen werden, werden auch am Ende des FABRIK Algorithmus eingehalten. Die Constraints, die in der Forward Reaching Phase erzwungenen werden, wirken sich zwar auf das Endergebnis aus, werden aber nicht unbedingt eingehalten, falls dies nicht möglich ist. Dies ist für das Handgelenk hilfreich, um zu verhindern, dass wenn der Arm in eine Position kommt, in welcher der Algorithmus keine Lösung findet und sich deshalb die Glieder sprunghaft bewegen. Aufgrund der Tatsache, dass man die Handpositionen aus den Controllerpositionen bekommt und der Benutzer die Controller möglicherweise anders halten kann als erwartet, ist es für die Handgelenke möglich, sich in einer Pose zu befinden, welche durch die Constraints unerreichbar wären. Für die beiden Beine werden keine Constraints verwendet, da die Positionen der Füße mittels der Gangsynthese gesetzt werden, welche selbst schaut, dass die Füße sich nicht in einer unmöglichen Pose befinden.

4.4 Magnetfelder

In dieser Arbeit wurde eine Erweiterung des FABRIK Algorithmus entwickelt, welche es erlaubt, die durch den FABRIK Algorithmus generierten Posen, zu beeinflussen. Dies ist besonders hilfreich für die inverse Kinematik bei Menschen, da es oft mehrere mögliche Lösungen gibt, aber nicht unbedingt alle davon realistisch sind. Die Erweiterung des FABRIK Algorithmus um sogenannte Magnetfelder erlaubt das Beeinflussen der generierten Posen und somit das Generieren von realistischeren Posen. Zusätzlich kann der Algorithmus mögliche Posen verhindern, welche nicht erwünscht sind. Dies ist nötig, da durch die oben beschriebenen Constraints für Menschen unmögliche Posen zustande kommen könnten. Durch das Verwenden der Magnetfelder ist es möglich, weniger Constraints zu verwenden, um dadurch robustere inverse Kinematik zu erhalten und dabei zusätzlich noch in der Lage zu sein, die durch die inverse Kinematik generierte Posen zu beeinflussen. Magnetfelder werden in der Arbeit für beide Arme verwendet.

Der Magnetfeld-Algorithmus wird vor jedem Durchlauf des FABRIK Algorithmus ausgeführt und passt dabei die Gelenkpositionen an. Die Magnetfelder sind damit nur in der Lage, die Ausrichtung der Knochen zu beeinflussen, ändern aber nichts an der Funktionalität des FABRIK Algorithmus. Jedes Magnetfeld ist in der Lage, Gelenke entweder anzuziehen oder diesen abzustoßen. Ein Magnetfeld besitzt eine Position $m_{position}$, einen Radius m_{radius} und eine anziehende bzw. abstoßende Kraft $m_{strength}$. Diese drei Werte geben an, in welchem Raum und mit welcher Stärke ein bestimmtes Magnetfeld auf ein Gelenk einwirken kann. Befindet sich ein Gelenk im Radius eines Magnetfeldes, wirken auf dieses anziehende bzw. abstoßende Kräfte. Die Richtung dieser Kraft ergibt sich aus der Position des Gelenks und der Position $m_{position}$ des Magnetfeldes. Die Kraft, die auf ein Gelenk wirkt, hängt von der Distanz zwischen ihm und dem des Magnetfeldes sowie dessen Stärke $m_{strength}$ und dessen Radius m_{radius} ab. Zur Bestimmung der Kraft wird die Distanz, sowie die Stärke des Magnetfeldes und dessen Radius verwendet. Die genaue Formel ist in 4.7 zu sehen.

$$d = p_i - m_{position} \tag{4.6}$$

$$s = cos(\frac{magnitude(d)}{m_{range}} \cdot \frac{\pi}{2}) \cdot m_{strength} \tag{4.7}$$

$$p_{offset} = normalize(d) \cdot s \tag{4.8}$$

Diese Formel sorgt für ein Magnetfeld, welches vom Rand aus schnell an Stärke gewinnt. Der Offset eines Gelenks ist die Summe der gerichteten Kraft jedes Magnetfeldes, in dem das Gelenk sich befindet. Die gerichtete Kraft ist das Produkt des normalisierten Richtungsvektor zwischen der Position des Gelenks, des Magnetfeldes und der berechneten Kraft, siehe Formel 4.8.

In Abbildung 4.6 ist die Funktionsweise eines Magnetfeldes auf die Gelenkposition p zu sehen. Der rote Punkt zeigt die Position, aus welchem das Magnetfeld ausgeht und der umgebende rote Kreis zeigt das Feld indem das Magnetfeld wirkt. In diesem Fall übt das Magnetfeld eine abstoßende Kraft aus. Da sich der Punkt p innerhalb des Magnetfeldes befindet wird dieser abgestoßen. Der p_{offset} Vektor zeigt die Richtung und Länge an, in welche der Punkt verschoben wird. Nach dem Einwirken des Magnetfeldes befindet sich die Gelenkposition p an der neuen Position p' und somit außerhalb des Magnetfeldes. Die genaue Funktionsweise des Algorithmus wird nochmal im Algorithmus 4.3 in Form von Pseudocode beschrieben. In Abbildung 4.7 ist die Platzierung der Magnetfelder für

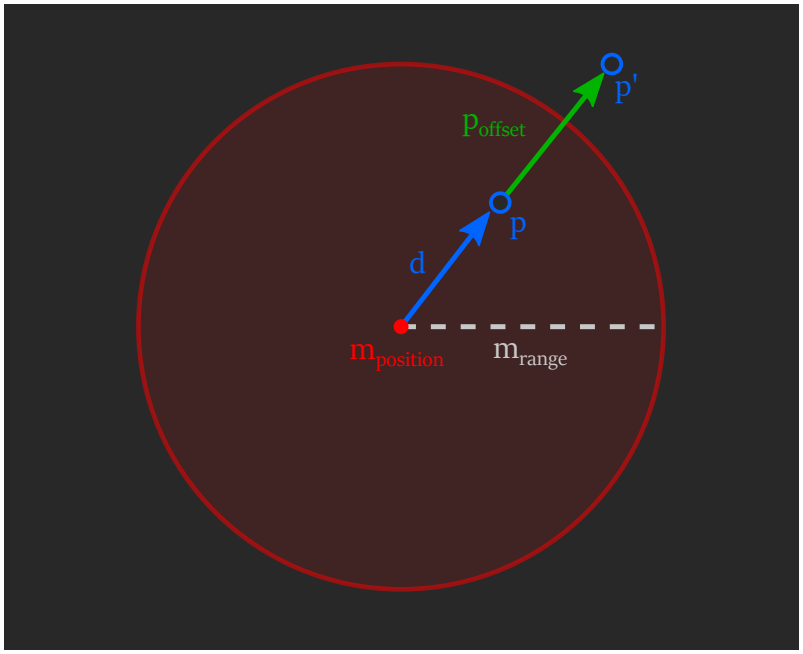

Abbildung 4.6: Das Einwirken eines Magnetfeldes auf die Gelenkposition *p*

den linken Arm zu sehen. Die Magnetfelder wirken jeweils auf das Ellenbogengelenk. Die kleinen Kugeln innerhalb der grünen und der roten Kugeln zeigen den Mittelpunkt der Magnetfelder und die umgebende größere Kugel zeigt den Raum, in dem das Magnetfeld wirkt. Im Oberkörper des Avatars wurden vier Magnetfelder platziert, welche den Ellenboden abstoßen und verhindern, dass dieser den Körper schneidet. Hinter dem Avatar wird ein großes Magnetfeld platziert, welches den Ellenbogen anzieht und versucht, diesen in eine natürliche Position zu bringen.

Das Resultat der Magnetfelder ist in Abbildung 4.8 zu sehen. In der Abbildung durchläuft der Avatar jeweils die exakt gleichen Posen, nur einmal unter Verwendung des Magnetfeld-Algorithmus und einmal ohne dessen Verwendung. In der oberen Reihe sind die Posen zu sehen, welche ohne den Magnetfeld-Algorithmus entstanden sind. In der unteren Reihe sind hingegen die Posen zu sehen, die unter der Verwendung des Magnetfeld-Algorithmus entstanden sind. Gut zu erkennen ist, wie die unteren Posen deutlich natürlicher wirken. Der Algorithmus sorgt dafür, dass die Ellenbogen nicht in unmögliche Richtungen verdreht sind und er sorgt auch dafür, dass die entstehenden Posen bei einer gleichen Eingabe (Positionen der Hände) auch immer dieselbe Ausgabe liefern. Ohne den Algorithmus hängen die Posen, die durch den FABRIK Algorithmus entstehen, stark von dem vorherigen Zustand ab. Mit dem Algorithmus ist dies nicht mehr der Fall, insofern sich das Gelenk innerhalb von Magnetfeldern befindet. Die entstehenden Posen werden dadurch berechenbarer und mittels unterschiedlicher Positionierungen der Magnetfelder, deutlich anpassungsfähiger.

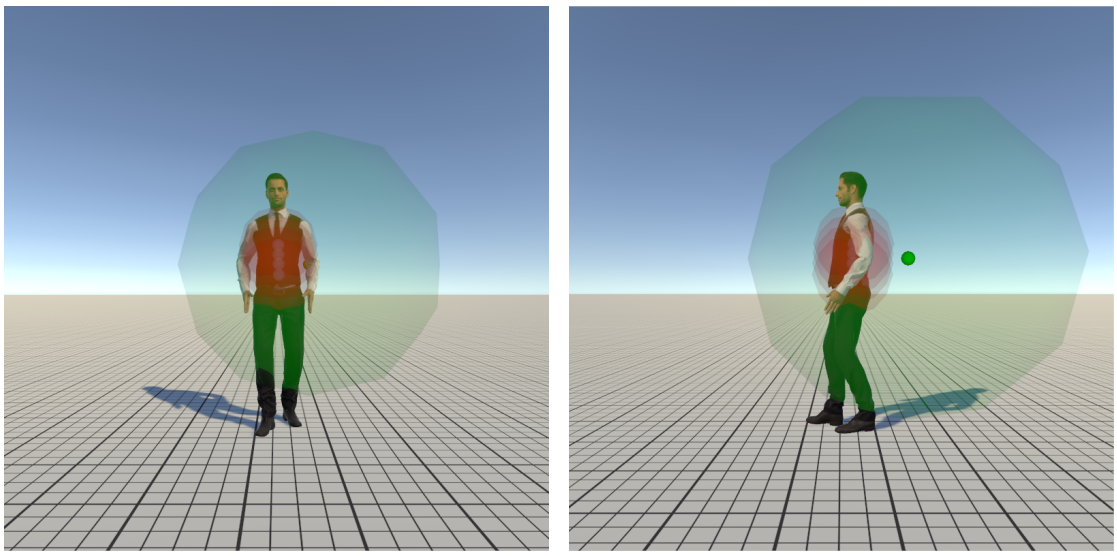

Abbildung 4.7: Magnetfelder für den linken Arm; die roten Felder wirken abstoßend; das grüne Feld wirkt anziehend

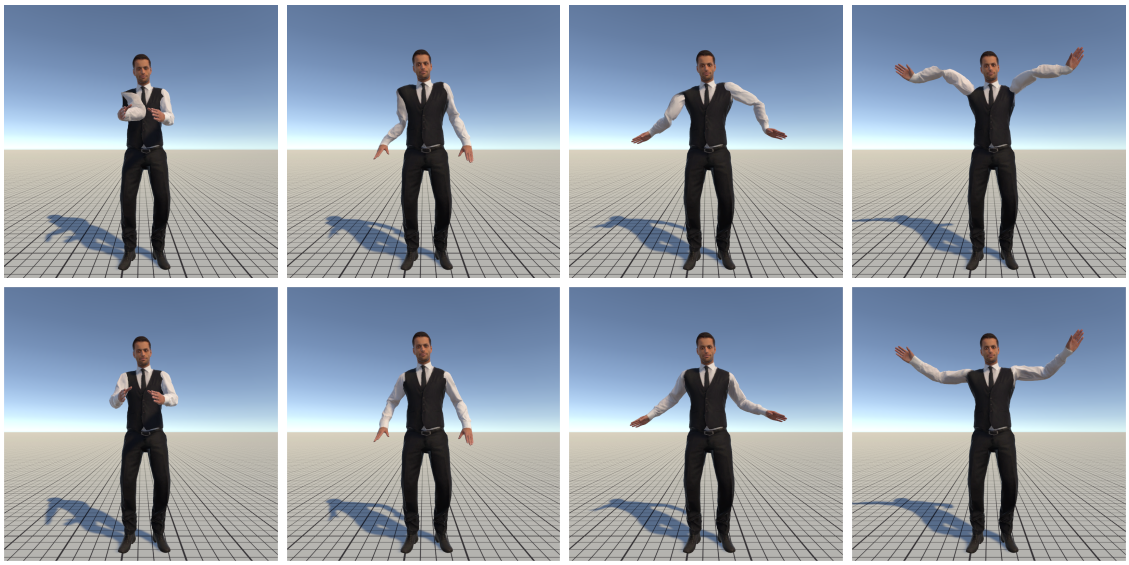

Abbildung 4.8: Obere Reihe: Ohne Magnetfelder; **Untere Reihe:** Mit Magnetfeldern

Algorithmus 4.3 Der Magnetfeld-Algorithmus

```
1: newElbowPosition ← currentElbowPosition
2:
3: for all Magnetfelder do
4:     magnetElbowVector ← magnetfield.position − elbowPosition
5:     magnetElbowNormal ← Normalize(magnetElbowVector)
6:     magnetElbowDistance ← magnitude(magnetElbowVector)
7:
8:     if magnetElbowDistance <= magnetfield.radius then
9:         pullVector ← magnetElbowNormal∗cos((magnetElbowDistance/magnetfield.range)∗
   π/2) ∗ magnetfield.strength
10:
11:        % Das Magnetfeld kann abstoßend oder auch anziehend wirken
12:        if magnetfield.repulsive then
13:            pullVector ← −pullVector
14:        end if
15:
16:        newElbowPosition ← newElbowPosition + pullVector
17:    end if
18: end for
19:
20: elbowPosition ← newElbowPosition
```

4.5 Schulter

Um das Schultergelenk des Avatars besser zu unterstützen, wurde in dieser Arbeit eine Erweiterung für den FABRIK Algorithmus entwickelt. Ohne die Erweiterung würde sich die Schulterposition frei in ihren Constraints (siehe Sektion 4.3) bewegen können, würde sich aber nicht unbedingt zurückbewegen. Die Erweiterung sorgt dafür, dass sich das Schultergelenk immer wieder zurück an seine ursprüngliche Position bewegt, um zu verhindern, dass das Schultergelenk an Posen beteiligt ist, in welchen dieses nicht unbedingt teilhaben müsste. Der erste Schritt des Algorithmus wird vor dem FABRIK Algorithmus ausgeführt und bewegt die aktuelle Schulterpostion wieder in Richtung der neutralen Position zurück, siehe Formel 4.9. Der zweite Schritt findet in der Backward Reaching Phase des FABRIK Algorithmus statt und begrenzt die maximale Positionsänderung der Schulterpositionen. Dafür wird die Differenz zwischen der neu berechneten Schulterposition und der Schulterposition aus dem, im ersten Schritt berechneten Wert, errechnet. Falls diese größer ist als eine bestimmter Wert (hier $5.01cm$), wird diese eingeschränkt. Dies ist in Formel 4.10 zu sehen. Der Algorithmus erlaubt es der Schulter, sich mehr zu bewegen als die Schulter in jedem Schritt wieder zurückbewegt wird. Dies sorgt dafür, dass die Schulter nur in Posen miteinbezogen wird, in welchen die Schulter auch wirklich benötigt wird.

$$upperArm'_{pos} = Vector3.MoveTowards(upperArm_{pos},\ upperArmNeutral_{pos},\ 0.05) \quad (4.9)$$

$$magnitude(upperArm_{pos} - upperArm'_{pos}) < 0.0501 \quad (4.10)$$

Abbildung 4.9: Position der Schulter in unterschiedlichen Posen

Das Resultat des Algorithmus ist in Abbildung 4.9 zu sehen. Die zwei roten Linien zeigen die neutrale Schulterposition und die blauen Linien die aktuellen Positionen der Armknochen. Im ersten Bild liegt die blaue Linie auf der roten Linie, die Schulterpositionen befinden sich also im neutralen Zustand. Das ist der Fall, da die Arme auch ohne die Schulter gut in der Lage sind, ihre Zielpositionen zu erreichen. In den zwei darauffolgenden Bildern streckt der Avatar seine beiden Arme immer weiter aus. Je weiter sich die Hände vom Körper bewegen, desto mehr werden auch die Schultergelenke in die Bewegung mit einbezogen. Im letzten Bild sind die beiden Arme voll ausgestreckt und die beiden Schultergelenke sind an ihrem, durch die Constraints definieren, Maximum. Ohne den Algorithmus würde sich beim Zurückbewegen der beiden Arme die Positionen der Schulter möglicherweise nicht ändern und an den gleichen Positionen bleiben. Dies würde dazu führen, dass die Schultern in den folgenden Posen unrealistisch positioniert wären. Durch das Verwenden des Algorithmus bewegen sich die Schultern immer wieder zurück und man erhält eine Pose, wie sie im ersten Bild zu sehen ist.

4.6 Arm Orientierung

Die inverse Kinematik der Schulter besteht aus drei Knochen. Dem *Shoulder*-, dem *UpperArm*- und dem *LowerArm*-Knochen. Die Orientierung des Schultergelenkes wird, wie im vorherigen Kapitel beschrieben, bestimmt. Die Rotation um die eigene Achse bleibt dabei immer gleich der Startrotation der Schulter.

Zur Bestimmung der Orientierung der beiden Oberarme wird auf die Position des *UpperArm*-, *LowerArm*- und dem *Hand*-Knochens zurückgegriffen. Diese drei Positionen definieren eine Ebene, von welcher die Normale dem Up-Vektor der Rotation des Oberarms entspricht. In Abbildung 4.10 ist dies zu sehen. Das blaue Dreieck deutet die Ebene an, welche durch die Positionen der drei Gelenke definiert wird. Die rote Linie zeigt die Normale, die als Up-Vektor der *UpperArm*-Rotation verwendet wird. Die Rotation des Oberarms ergibt sich also aus dem Forward-Vektor, welcher von *UpperArm* in Richtung des *LowerArm* verläuft und dem Up-Vektor, der wie oben beschrieben, berechnet wird. Die Normale dieser Ebene lässt sich einfach mit dem Kreuzprodukt, wie in Formel 4.11 zu sehen ist, berechnen.

$$normal = Vector3.Cross(LowerArm_{pos} - UpperArm_{pos},\ Hand_{pos} - UpperArm_{pos}) \quad (4.11)$$

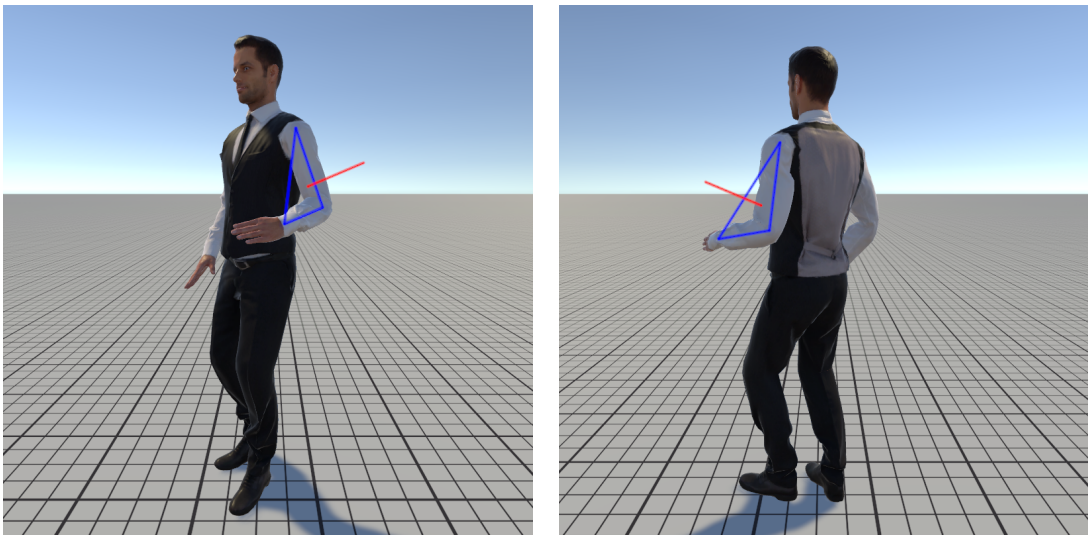

Abbildung 4.10: Rotation des Oberarms

Die Orientierung der Unterarme hängt hingegen einfach von der Orientierung der Hände ab. Dies ist in Abbildung 4.11 zu sehen. Da sich das Handgelenk immer zusammen mit dem Gelenk des Unterarmes zusammen dreht, besitzt das Ellenbogengelenk immer dieselbe Rotation wie das Handgelenk. Die Orientierung des Handgelenkes wird dabei aus der Orientierung der beiden Controller genommen.

4.7 Beine

Für die beiden Beine wird, genau wie für die beiden Arme, der FABRIK Algorithmus verwendet. Da der Benutzer die Beinpositionen nicht frei wählen kann, sondern diese von der in Kapitel 6 beschriebenen Gangsynthese gesetzt werden, werden für die Beine keine Constraints oder Ma-

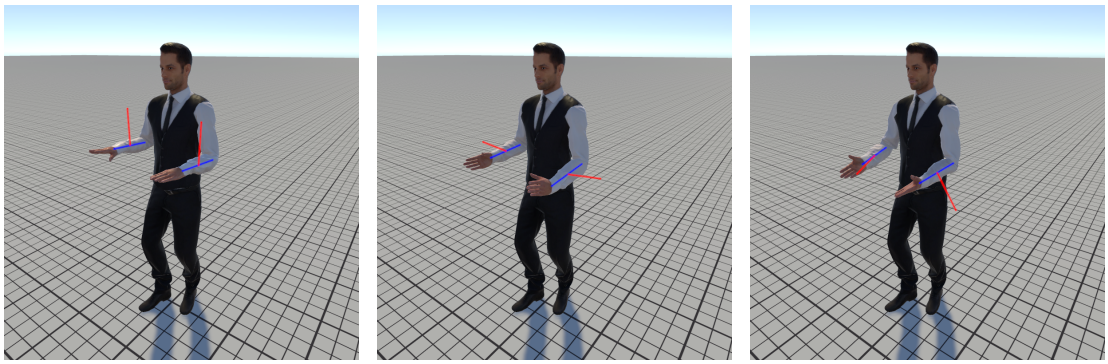

Abbildung 4.11: Rotation der Unterarme

gnetfelder verwendet. Um das Kniegelenk zu realisieren, wird ein abgeänderter Ansatz verwendet, wie er in dem Paper [1] beschrieben wird. Zusätzlich dazu wird auch noch dafür gesorgt, dass das Kniegelenk immer in die richtige Richtung eingeknickt ist.

Das Kniegelenk wird so realisiert, dass dieses sich nur auf einer Ebene bewegen kann. Dies wird erreicht, indem vor dem Ausführen des FABRIK Algorithmus die Kniegelenkposition so gesetzt wird, dass diese selbst nach dem Ausführen des FABRIK Algorithmus auf der festgelegten Ebene bleibt. Dies ist möglich, da keine Constraints verwendet werden und durch die Funktionsweise des FABRIK Algorithmus das Kniegelenk auf der zuvor gesetzten Ebene bleibt. Die Ebene, auf der sich das Kniegelenk bewegt, kann mittels der $UpperLeg$-Position und der Inversen-Kinematik-Zielposition des Fußes ($footTarget$) und dessen Orientierung bestimmt werden. Dabei sind die $UpperLeg$- und die $footTarget$-Positionen zwei Punkte, welche auf dieser Ebene liegen. Der Up-Vektor des $footTarget$ Gelenkes liegt ebenfalls auf dieser Ebene. Um sicherzustellen, dass das Kniegelenk (p_2) auch auf dieser Ebene liegt, wird dessen Position, von der vorherigen Position auf die neue Ebene projiziert. Dieser Vorgang ist in Abbildung 4.12 zu sehen. Dabei wird die Position des Kniegelenkes p_2 genommen und von der vorherigen Ebene (Φ_1) auf die neue Ebene (Φ_2) projiziert. Dass sich dabei der Abstand der Punkte zueinander ändert und nicht mehr den Knochenlängen entspricht, ist dabei nicht wichtig, da diese beim Anwenden des FABRIK Algorithmus wieder richtig gesetzt werden. Nach dem Durchlaufen des FABRIK Algorithmus befinden sich alle drei Punkte weiterhin auf der Ebene, dann aber mit dem richtigen Abstand zueinander.

Um sicherzustellen, dass das Kniegelenk in die richtige Richtung eingeknickt ist, wird dies zuerst überprüft und falls die Knieposition auf der falschen Seite liegt, korrigiert. Dabei wird dieselbe Ebene wie oben verwendet, nur 90° um die Achse, die von $UpperLeg$ nach $footTarget$ verläuft, gedreht. Nach dem Berechnen dieser Ebene, wird der Abstand zwischen der Ebene und der Kniegelenkposition p_2, mit Vorzeichen, berechnet. Ist der Abstand kleiner als 0, liegt die Knieposition auf der falschen Seite und das Kniegelenk ist in die falsche Richtung eingeknickt. Um dies zu beheben, wird der Punkt p_2 auf die entgegen liegende Seite von der Ebene projiziert. Dies ist in Abbildung 4.13 zu sehen, wo der Punkt p_2 an der Ebene Φ_1 gespiegelt wird. Ist die berechnete Distanz größer als 0, befindet sich die Knieposition auf der richtigen Seite und es muss nichts getan werden.

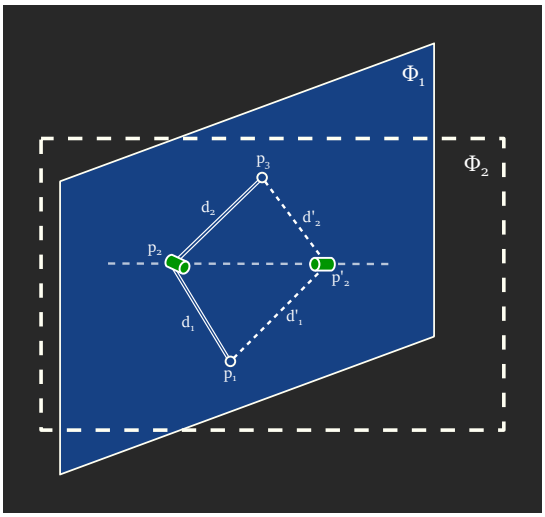

Abbildung 4.12: Kniegelenk Projektion

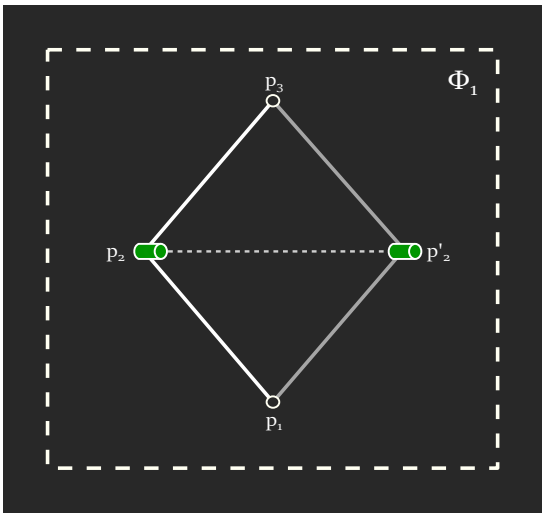

Abbildung 4.13: Kniegelenkknick

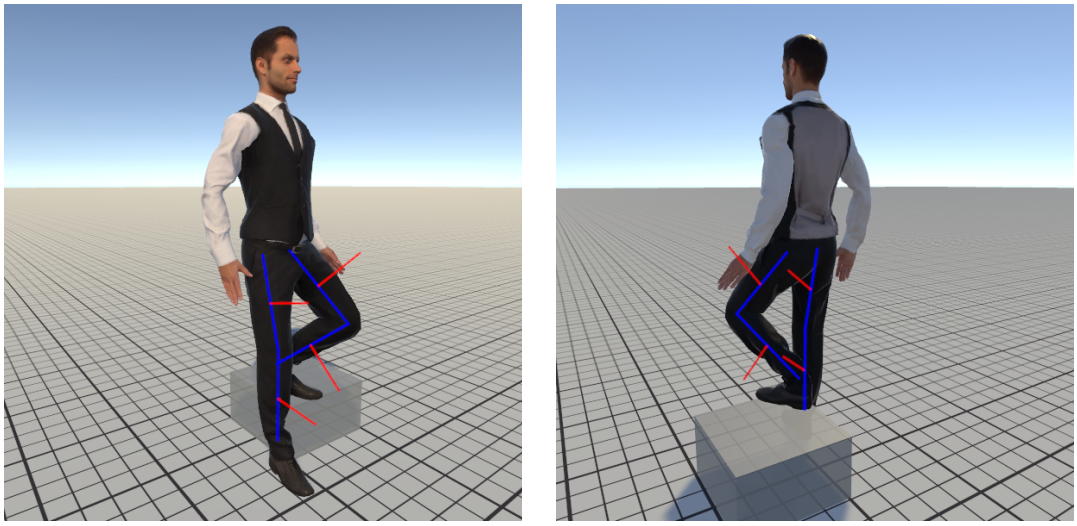

Abbildung 4.14: Rotation der Beine

Die Rotation um die eigene Achse der Beine wird genau wie die Orientierung der Arme in Sektion 4.6 berechnet. Nur werden die $UpperLeg$-Gelenke dabei 90° um ihre eigene Achse gedreht. Die Rotation der $LowerLeg$ Gelenke kann direkt aus den $footTarget$ Rotationen genommen werden, diese werden in Kapitel 6 berechnet. In Abbildung 4.14 ist der Avatar zu sehen, wie er auf einer Box steht. Die Knochen wurden dabei durch die blaue Linie visualisiert und die roten Linien zeigen jeweils die Normalen der einzelnen Knochen an.

5 Oberkörper

Zur Berechnung der Pose des Oberkörpers wurde in dieser Arbeit ein eigener Ansatz entwickelt. Der Oberkörper des Avatars besteht aus sechs Gelenken: Hip, $Spine1$, $Spine2$, $Spine3$, $Neck$ und dem $Head$. Die Positionen und Orientierungen all dieser Gelenke werden aus der Position bzw. der Orientierung des VR-Headsets geschlossen. Die Posen-Berechnung des Oberkörpers besteht aus vier Phasen. Diese sind in dem Algorithmus 5.1 zu sehen. Als erstes werden die Position und die Orientierung des VR-Headsets eingeschränkt. Wie dies genau geschieht wird in Sektion 5.1 beschrieben. Danach wird die Richtung, in welche der Avatar steht, berechnet, siehe Sektion 5.2. Schlussendlich werden dann die Positionen und Orientierungen der oben genannten Gelenke berechnet und gesetzt. Dies wird in den Sektionen 5.3 und 5.4 beschrieben.

Algorithmus 5.1 Berechnung der Oberkörperpose des Avatars

```
1: ConstrainHeadsetRotation()
2: ConstrainHeadsetPosition()
3: UpdateAvatarRotation()
4: UpdateUpperBodyRotation()
```

5.1 Constraints

Um unlogische Posen zu verhindern, während das VR-Headset beispielsweise auf dem Boden liegt oder dieses sich in einer Position/Orientierung befindet, welche nicht von dem beschriebenen Algorithmus unterstützt wird, wird die Position bzw. die Orientierung eingeschränkt. Das Einschränken der Position geschieht durch das Einschränken der Y-Position des VR-Headsets, dies ist in Formel 5.1 zu sehen. Dabei wird die Y-Position immer mindestens $80cm$ vom und maximal $2.25m$ über dem Boden gehalten.

$$head_{pos}.y = clamp(headset_{pos}.y,\ floor_{pos}.y + 0.8,\ floor_{pos}.y + 2.25) \tag{5.1}$$

Zur Einschränkung der Orientierung wird das in Kapitel 4.2 beschriebene Verfahren verwendet. Die Kopfrotation wird dabei so eingeschränkt, dass sich der Kopf in alle Richtungen um 85° neigen kann, ausgehend von der neutralen Kopfrotation.

5.2 Rotation des Körpers

In welche Richtung der Körper des Avatares rotiert ist, wird wie folgt berechnet. Als erstes wird die Differenz ($fromToAngle$) zwischen der aktuellen Y-Rotation des Avatars und der Y-Rotation des VR-Headsets berechnet, siehe Formel 5.2. Es wird der Wert $fromToAngle_{clamped}$ berechnet, dieser schränkt den $fromToAngle$-Wert zwischen r_{min} und r_{max} ein. Die Werte r_{min} und r_{max} beschreiben den Bereich, in welchem der Avatar seinen Kopf um die Y-Achse rotieren kann, ohne dass er dabei seinen Körper mitbewegt. Der Bereich wurde in dieser Arbeit auf -20° bis 20° festgelegt, wodurch der Avatar immer ungefähr in Kopfrichtung steht, aber nicht bei jeder kleinen Bewegung seinen ganzen Körper mitbewegt. Danach wird der $fromToAngle$ Wert auf die aktuelle Y-Rotation des Körpers addiert und der eingeschränkte $fromToAngle_{clamped}$-Wert wird wieder abgezogen, siehe Formel 5.4. Liegt die Rotation also im erlaubten Bereich und $fromToAngle$ entspricht dem eingeschränkten $fromToAngle_{clamped}$ Wert, entspricht $rotation$ dem aktuellen Wert der Rotation des Körpers $body_{euler}.y$. Liegt die Rotation nicht im erlaubten Bereich, sind die Werte von $fromToAngle$ und $fromToAngle_{clamped}$ unterschiedlich und der $rotation$ Wert entspricht dem des Körpers, welcher die VR-Headset-Rotation im beschränkten Bereich enthält. Um den Körper nicht allzu mechanisch mit dem Kopf verbunden erscheinen zu lassen, wird der $rotation$-Wert nicht einfach dem $body_{euler}.y$-Wert zugewiesen. Es wird, wie in Formel 5.5 zu sehen ist, zwischen dem $body_{euler}.y$ und dem $rotation$-Werten interpoliert. In Abbildung 5.1 ist die Funktionsweise des beschriebenen Ansatzes zu sehen. Der Raum, in welchem sich der Kopf bewegen kann, ohne dass der Körper mitzieht, wird mittels der grünen, und die aktuelle Blickrichtung des VR-Headsets wird mittels der gelben Linie, dargestellt. In der oberen Reihe bewegt der Avatar seinen Kopf innerhalb der Einschränkungen, wodurch sich die Rotation des Körpers nicht ändert. In der unteren Reihe hingegen bewegt der Avatar einen Kopf über die Einschränkung hinaus, wodurch sich der Körper mitbewegt, um die Kopfrichtung innerhalb dessen Einschränkungen zu halten.

$$fromToAngle = body_{euler}.y - headset_{euler}.y \tag{5.2}$$

$$fromToAngle_{clamped} = clamp(fromToAngle,\ r_{min},\ r_{max}) \tag{5.3}$$

$$rotation = body_{euler}.y + fromToAngle - fromToAngle_{clamped} \tag{5.4}$$

$$Hip_{euler}.y' = lerp(Hip_{euler}.y,\ rotation, 0.5) \tag{5.5}$$

5.3 Y-Achsen Rotation

Lehnt der Avatar seinen Kopf zur Seite, werden mit dem Kopf auch noch die Gelenke von Hip bis $Neck$ auf der Z-Achse rotiert. Das Ausmaß der Rotation der einzelnen Gelenke, hängt von der Z-Achsen-Rotation des VR-Headsets ab, welche jeweils mit einem konstanten Wert multipliziert wird. Dieser konstante Wert kann für jedes Gelenk individuell angepasst werden. Die Z-Achse besitzt eine $deadzone_Z$, welche angibt, in welchem Bereich der Avatar seinen Kopf neigen kann, ohne dass dies Auswirkungen auf die Rotation der Hip- bis $Spine3$-Gelenke hat. Das $Neck$-Gelenk wird hingegen selbst im festgelegten Bereich mitrotiert. Die Z-Achsen-Rotation des $Neck$-Gelenks wird, wie in Formel 5.6 zu sehen ist, berechnet. Zur Berechnung der Rotation der Hip- bis $Spine3$-Gelenke wird zuerst der $normalizedZ$-Wert genommen, welcher wie in Formel 5.7 berechnet wird. Für den

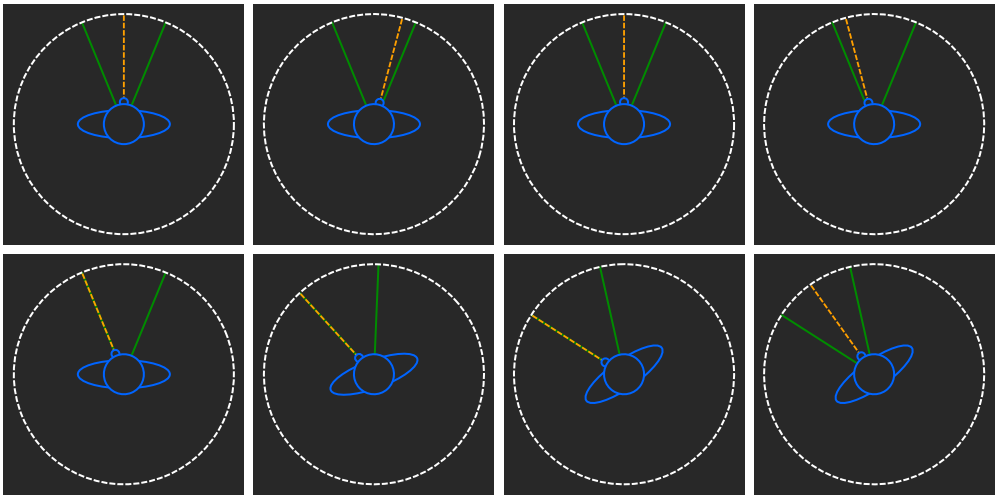

Abbildung 5.1: Rotation des Körpers

$deadzone_Z$-Wert wurde in dieser Arbeit 25° genommen. Der Avatar kann also seinen Kopf ±25° um die Z-Achse rotieren ohne dass sich die *Hip*- bis *Spine*3-Gelenke dabei mitbewegen. In Formel 5.8 ist zu sehen, wie die Z-Achsen-Rotation für die *Spine*1, ausgehend vom *normalizedZ*-Wert, berechnet wird. Dieselbe Formel wird für die restlichen Gelenke verwendet, jeweils aber mit einer individuellen Multiplikator-Konstanten. Die in dieser Arbeit verwendeten Multiplikatoren sind in der Tabelle 5.1 in der Spalte *Z Multiplikator*, aufgelistet. Je weiter man sich vom *Neck*-Gelenk in Richtung des *Hip*-Gelenkes bewegt, desto kleiner werden die jeweiligen Multiplikatoren. Dadurch hat die Z-Achsen-Rotation des VR-Headsets eine immer geringer werdende Auswirkung auf ein Gelenk, je weiter dieses vom *Neck*-Gelenk entfernt ist.

$$neck_{euler}.z = headset_{euler}.z \cdot NeckMultiply \tag{5.6}$$

$$normalizedZ = sign(headset_{euler}.z) \cdot clamp(abs(head_{euler}.z) - deadzone_Z,\ 0,\ max) \tag{5.7}$$

$$Spine1_{euler}.z = normalizedZ \cdot Spine1Multiply \tag{5.8}$$

In Abbildung 5.2 ist die Rotation des VR-Headsets und deren Auswirkung auf die Pose des Avatars zu sehen. Im ersten Bild ist das VR-Headset kaum um die Z-Achse rotiert, wodurch alle Gelenken auch ungefähr die gleiche Z-Achsen-Rotation besitzen. Im zweiten Bild lehnt der Benutzer seinen Kopf leicht nach rechts. Die Rotation befindet sich aber immer noch im Bereich der $deadzone_Z$, wodurch sich nur der Kopf und der Nacken des Avatars bewegen. Im dritten Bild ist gut zu erkennen, wie der Benutzer seinen Kopf deutlich stärker geneigt hat und die Rotation somit nicht mehr in der $deadzone_Z$ liegt. Dies hat zur Folge, dass alle Gelenken von *Spine*1 bis *Head* auf ihrer Z-Achse rotiert werden. Je weiter das Gelenk von dem *Head*-Gelenk entfernt ist, desto weniger wird die Z-Achsen-Rotation davon beeinflusst.

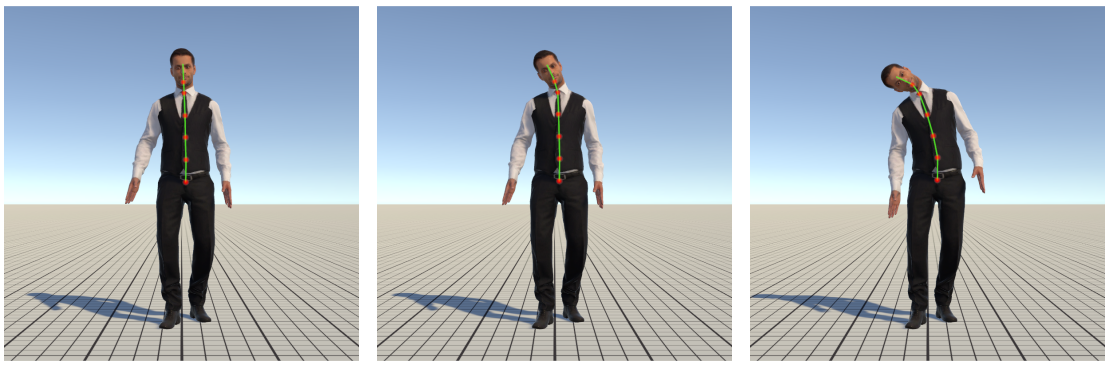

Abbildung 5.2: Kopfrotation auf der Z-Achse

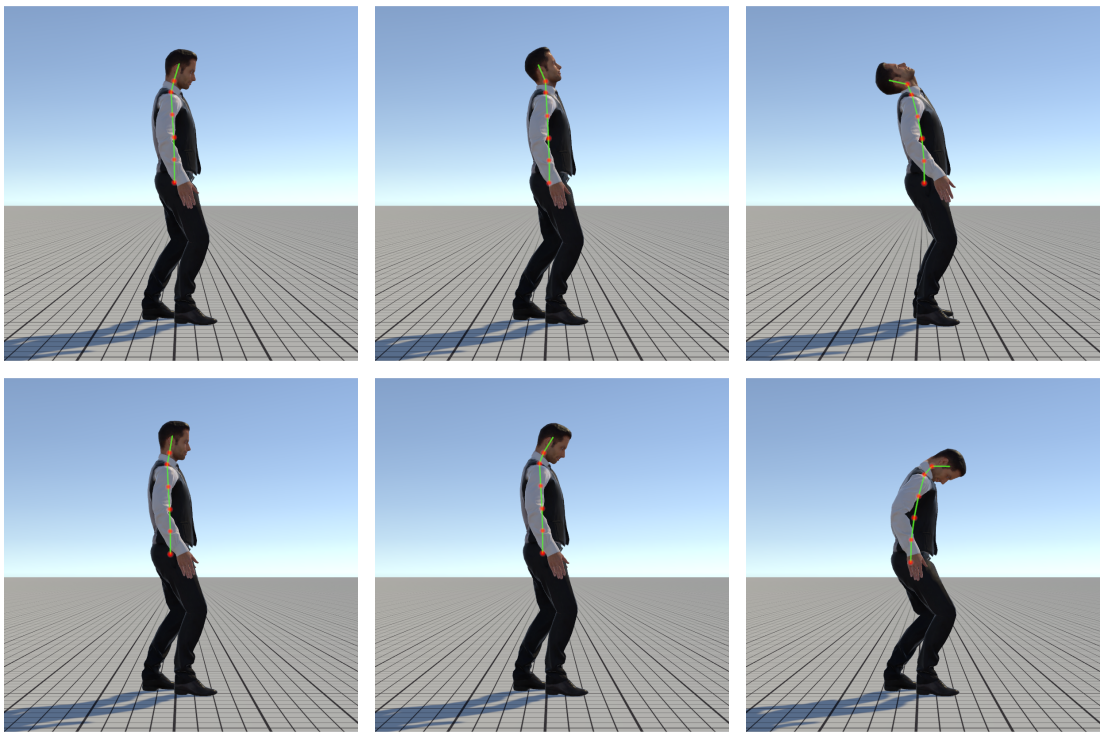

Abbildung 5.3: Kopfrotation auf der X-Achse

5.4 X-Achsen Rotation

Für die Rotation des Kopfes auf der X-Achse wird das selbe Prinzip wie für die Rotation auf der Z-Achse in Sektion 5.3, verwendet. Der $deadzone_X$-Wert ist dabei etwas größer als bei der Z-Achse und liegt im Bereich von ±35°. Das Resultat davon ist in der Abbildung 5.3 zu sehen. In der oberen Reihe lehnt der Benutzer seinen Kopf nach hinten. Im ersten und zweiten Bild befindet sich die X-Achsen-Rotation noch in der $deadzone_X$, wodurch nur das $Head$- und das $Neck$-Gelenk bewegt werden. Im dritten Bild liegt die X-Achsen-Rotation außerhalb der $deadzone_X$, was dazu führt, dass die restlichen Gelenken mit rotiert werden. In der unteren Reihe ist genau derselbe Ablauf zu sehen, nur neigt der Benutzer seinen Kopf in diesem Fall nach vorne und nicht nach hinten.

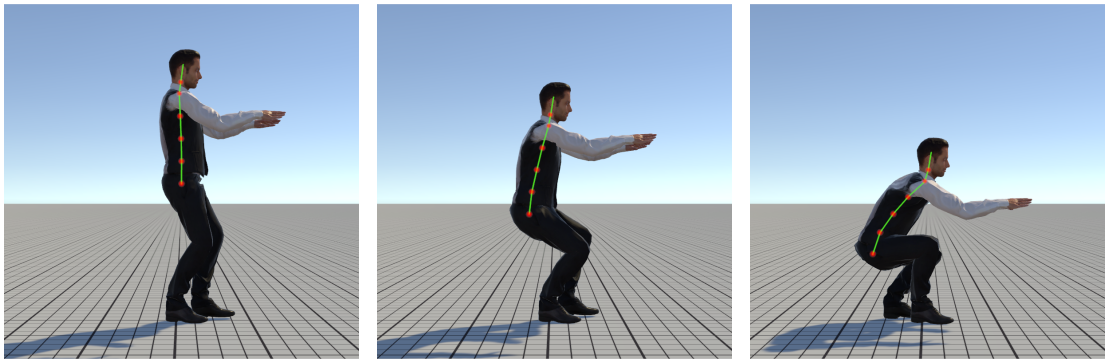

Abbildung 5.4: Duckend

Bei der Berechnung der X-Achsen-Rotation der Gelenke wird, anders als bei der Z-Achse, nicht nur die Rotation des VR-Headsets berücksichtigt, sondern auch noch die Y-Position des VR-Headsets, relativ zur Y-Position des Bodens ($floor_{pos}.y$). Je näher sich das VR-Headsets dem Boden nähert, desto größer wird auch der $heightRotation$-Wert, welcher zusätzlich noch die Z-Achsen-Rotation der Gelenke beeinflusst. Der $heightRotation$-Wert wird wie in Formel 5.9, berechnet. Dafür wird die Differenz zwischen der Y-Position des VR-Headsets und der Y-Position des Bodens genommen und dieses von der Größe des Benutzers ($userHeight$) abgezogen. Dadurch besitzt $heightRotation$ den Wert 0, wenn der Benutzer aufrecht steht. Duckt sich der Benutzer hingegen, wird der $heightRotation$-Wert größer. Der berechnete Wert wird zusätzlich begrenzt, um nicht kleiner als 0 zu sein und so zu verhindern dass sich der Oberkörper unrealistisch verdreht, wenn der $userHeight$-Wert falsch eingestellt wurde, der Benutzer auf seinen Zehenspitzen steht oder sogar springt. Zur Berechnung der X-Achsen-Rotationswerte der Gelenke, wird $heightRotation$ mit einem entsprechenden Multiplikator multipliziert. Anschließend werden die neuen Rotationen der Gelenke wie in Formel 5.11 berechnet. Um sicher zu stellen, dass die Rotation nicht unrealistisch groß oder klein ist, wird dieser Wert noch, wie in Formel 5.12 zu sehen ist, zwischen -45° und 90° eingeschränkt. In der Tabelle 5.1 werden die jeweiligen Multiplikatoren der Gelenken in der $X\ Multiplikator$- und $Hoehen\ Multiplikator$-Spalte aufgeführt. In Abbildung 5.4 ist das Ergebnis zu sehen. Zu erkennen ist, wie sich die Kopfposition dem Boden immer weiter nähert und daher die Krümmung der Wirbelsäule immer größer wird.

$$heightRotation = clamp(userHeight - (headset_{pos}.y - floor_{pos}.y),\ 0,\ max) \cdot 100 \tag{5.9}$$

$$normalizedX = sign(headset_{euler}.x) \cdot clamp(abs(head_{euler}.x) - deadzone_X,\ 0,\ max) \tag{5.10}$$

$$newSpineX = heightRotation \cdot spine1HeightMultiply + normalizedX \cdot spine1XMultiply \tag{5.11}$$

$$Spine1_{euler}.z = clamp(newSpineX,\ -45,\ 90) \tag{5.12}$$

	X Multiplikator	Z Multiplikator	Höhen Multiplikator
Hip	0	0	0.3
Spine1	0.1	0.25	0.6
Spine2	0.2	0.35	0.7
Spine3	0.3	0.45	0.8
Neck	0.4	0.55	0.15

Tabelle 5.1: Die für den Oberkörper verwendeten Multiplikatoren

6 Gangsynthese

Um einen realistischen Avatar zu visualisieren, ist es wichtig, dass eine realistische Gangsynthese existiert. Im Vergleich zu traditionellen nicht-VR-Ansätzen für das Animieren von Avatars bieten Laufanimationen in VR neue Probleme. In traditionellen 3D Anwendungen besitzt ein Avatar oft nur wenige unterschiedliche Gangarten. Er kann bspw. nur stehend und gebückt gehen. In VR kann sich der Benutzer aber in jeder möglichen Position zwischen stehend und gebückt befinden. In traditionellen 3D Anwendungen ist es möglich, dem Benutzer nicht die volle Kontrolle über den Avatar zu geben, um Animationen möglichst gut ineinander fließen zu lassen. Dies ist in VR nicht möglich, da der Benutzer sich in der reellen Welt frei bewegen kann und dies möglichst genau von seinem Avatar widergespiegelt werden soll. Zusätzlich dazu wird nicht der ganze Avatar animiert, sondern nur der Teil von ihm, welcher nicht schon durch die inverse Kinematik bestimmt wurde. Dies sind einige der Gründe, weshalb man nicht aus 3D Anwendungen bekannte Ansätze verwenden kann. Aufgrund dessen wurde für die Laufanimationen des Avatars in dieser Arbeit eine eigene Gangsynthese entwickelt, welche auf der in Kapitel 4 beschriebene inverse Kinematik aufbaut. Die Gangsynthese legt dabei lediglich die Positionen und Orientierungen der Füße fest und überlässt das genaue Platzieren der Gelenke der oben beschriebenen inversen Kinematik. Zusätzlich dazu beeinflusst die Gangsynthese auch noch die Positionen des Kopfes, des Oberkörpers und der Hüfte, um eine möglichst realistischen Laufanimation zu generieren.

6.1 Aktuelle Ansätze

Das Problem der Gangsynthese in VR kann über unterschiedliche Ansätze angegangen werden. Die einfachste Methode ist, den Unterkörper des Benutzers nicht darzustellen und somit das komplette Problem zu umgehen. Dies ist in vielen Situationen eine sehr unkomplizierte und effektive Lösung. Dieser Ansatz wird aktuell von vielen VR-Anwendungen verfolgt, selbst große Firmen wie Oculus beschränken sich bei der Darstellung von Avatars auf deren Kopf und dessen Hände[1].

Ein weitere Möglichkeit wird in dem Paper [10] beschrieben, wobei das Problem der Gangsynthese umgangen wird und die exakte Positionen der einzelnen Glieder des Benutzers mittels externen Sensoren getrackt werden. Dies hat einerseits einen großen Vorteil, da die Pose des Benutzers eins zu eins übernommen werden kann und man dadurch zu jedem Zeitpunkt eine realistische Pose des Avatars erhält. Der Ansatz besitzt aber auch Negativpunkte, weshalb er sich in vielen Situationen nicht eignet. Einer dieser Negativpunkte ist die teure Hardware, welche zusätzliche noch aufwendig vom Benutzer angelegt werden muss. Ein weiter Negativpunkt ergibt sich aber auch aus der Tatsache, dass die Pose des Benutzers genau getrackt wird. Steht der Benutzer bspw. in der virtuellen Welt mit einem Bein auf einer Stufe, in der echten Welt aber auf einer flachen Ebene, befindet sich der Avatar

[1] https://developer.oculus.com/documentation/avatarsdk/latest/concepts/avatars-sdk-intro/

in einer Pose welche nicht zu der virtuellen Welt passt. Soll der Benutzer zusätzlich auch noch in der Lage sein, mittels eines Controllers durch die virtuelle Welt zu navigieren, muss zusätzlich zu dem Tracking auch noch eine separate Gangsynthese existieren.

In dem Paper [5] wird die Gangsynthese mittels Animation Blending gelöst. Dabei besitzt der Avatar acht unterschiedliche Laufanimationen, die in unterschiedlichen Richtungen unterteilt sind zwischen denen geblendet wird. Die Animation ist dabei von der Laufrichtung abhängig. Dies ist eine recht einfache Methode, welche aber auch Probleme mit sich bringt. Einer dieser Probleme ist die durch VR gegebene Bewegungsfreiheit des Benutzers, der sich auf viele unterschiedliche Arten bewegen kann, welche unmöglich alle in dem Blend Space dargestellt werden können. Bspw. können kurze, kleine Bewegungen des Benutzers zu Animationen führen, die kurz nach dem Start immer wieder abgebrochen werden. Dabei ist das Gleiten der Füße auf dem Boden auch nur mit sehr viel Aufwand zu umgehen.

In dem Artikel [8] wird eine weitere Gangsynthese beschrieben, die in dem Spiel „Dead and Buried“ verwendet wird. Diese Gangsynthese basiert auf inverser Kinematik und versucht dabei die Füße so zu positionieren, dass der Center of Mass des Avatars immer zwischen den beiden Füßen liegt. Liegt der Center of Mass nicht zwischen den beiden Füßen oder hat sich der Benutzer auf der Stelle gedreht, werden die Füße neu positioniert. Diese Methode hat den Vorteil, dass das Gleiten der Beine auf dem Boden vermieden werden kann und auch kleine, kurze Bewegungen, wie sie in VR oft auftreten, gut unterstützt werden. Probleme besitzt diese Methode aber, wenn der Avatar läuft, wobei es so aussehen kann, als würde der Avatar seine Beine hinterher ziehen. Die in dieser Arbeit entwickelte Gangsynthese versucht deshalb, auf diesem Ansatz aufzubauen und das Laufen des Avatars mittels *AnimationCurve* besser zu unterstützen.

6.2 Eigener Algorithmus

Die in dieser Arbeit entwickelte Gangsynthese besteht aus unterschiedlichen Schritten, welche jeweils nacheinander ausgeführt werden. Die einzelnen Schritte sind in dem Algorithmus 6.1 zu sehen. Als Erstes wird in jedem Durchlauf des Algorithmus die Durchschnittsgeschwindigkeit des Avatars berechnet. Diese wird benötigt um bspw. die Schrittlänge zu bestimmen. Wie dies genau geschieht, wird in Sektion 6.4 beschrieben. Als Nächstes wird die Schrittgröße, wie in Sektion 6.5 beschrieben, berechnet. Danach werden die aktuell ablaufenden Steps aktualisiert und die Positionen der Füße für die inverse Kinematik gesetzt. Dies wird in Sektion 6.3 genauer beschrieben. Anschließend werden die aktuell ablaufenden Steps korrigiert, um möglichst genau mit der aktuellen Bewegung des Avatars überein zu stimmen. Dies ist nötig, da der Avatar bspw. seine Geschwindigkeit oder seine Richtung ändern kann, während dieser sich in einem Step befindet. Für die Korrektur gibt es zwei unterschiedliche Schritte. In einem Schritt wird die Zielposition der Steps korrigiert, um an einer sinnvollen Position zu landen, dies wird in Sektion 6.8 beschrieben. Zusätzlich dazu wird die Schrittgeschwindigkeit angepasst, um den jeweiligen Step auch im richtigen Moment zu Ende zu bringen. Dies wird in Sektion 6.9 beschrieben. Nach den Korrekturschritten wird die Position des Kopfes, die Orientierung der Hüfte und des Oberkörpers aktualisiert. Dies soll zu einer möglichst realistischen Ganganimation führen. Wie dies genau geschieht, wird in den Sektionen 6.11 und 6.12 beschrieben. Danach wird die Position des Avatars nach oben oder unten korrigiert, sodass es ihm immer möglich ist, mit beiden Beinen den Boden zu berühren, auch wenn er bspw. auf einer Stufe steht. Dies wird in Sektion 6.10 beschrieben. Zu guter Letzt wird noch geprüft, ob der Avatar sich

aktuell in einer Situation befindet, in welcher er einen neuen Step starten sollte und falls dies der Fall ist, wird ein neuer Step gestartet. Dies wird in Sektion 6.6 beschrieben. Wie genau ein Step abläuft und wie dabei die Positionen und die Orientierungen der Füße angepasst werden, wird in Sektion 6.3 beschrieben.

Algorithmus 6.1 Eine Iteration der Gangsynthese

```
1: CalculateAverage()
2: CalculateStepSize()
3: UpdateSteps()
4: CorrectStepPlacement()
5: CorrectStepTiming()
6: MoveHips()
7: MoveHead()
8: GroundFeets()
9: CheckForNewSteps()
```

6.3 Step

Ein Step ist die Bewegung eines Fußes vom abgesetzten Zustand angehoben zu werden, zu einer neuen Position bewegt zu werden und am Ende wieder auf dem Boden aufzusetzen. Jeder Step besteht aus einer Startposition $start$ und einer Endposition $goal$, zwischen denen sich der Fuß bewegt. Die aktuelle Position eines Fußes, welche für die inverse Kinematik verwendet wird, ist die $currentPosition$. Wobei $footTarget$ die Position und die Orientierung enthält, welche für die inverse Kinematik verwendet wird. Der aktuelle Fortschritt wird mit der $stepCount$-Variablen repräsentiert, welche zwischen 0 und 1 liegt. Wobei 0 für den Start des Steps steht und 1 für das Ende des Steps. Der $stepCount$-Wert wird in jeder Iteration des Algorithmus im Verhältnis zu $stepDuration$ erhöht. Der $stepDuration$-Wert gibt an, wie lange ein Step in Sekunden vom Start bis zum Ende benötigt. Je kleiner die $stepDuration$, desto schneller verläuft der Step. Zwischen der Start- und der Endposition eines Steps wird mittels der $stepCount$-Variablen und der $stepInterpolation$-Kurve (siehe linkes Bild in Abbildung 6.2) interpoliert. Die verwendete Kurve verläuft anfangs langsam, gewinnt dann schnell an Steigung und flacht gegen Ende wieder ab. Dies sorgt dafür, dass der Fuß langsam in Bewegung kommt und vor dem Stehenbleiben wieder abbremst. Das Verwenden einer linearen *AnimationCurve* würde zu einem Step führen, der weder beschleunigt noch abbremst und den Fuß vom $start$- bis zum $goal$-Punkt mit gleichbleibender Geschwindigkeit bewegt.

Die Positionen $start$ und $goal$ liegen beide auf der X-Z-Ebene und besitzen keinen Y-Wert. Für die Berechnung des Y-Wertes wird daher zwischen dem Y-Wert der Startposition des Fußes und dem Y-Wert der Endposition des Fußes interpoliert. Die Startposition ist in diesem Fall die $currentPosition$ des Fußes am Start des Steps. Die Endposition wird in jeder Iteration neu berechnet. Dafür wird für den Punkt $goal$ auf der Höhe des Player Controllers heruntergeschaut und ein Schnittpunkt mit dem Boden gesucht. Liegt $goal$ bspw. an der Position einer Stufe, wird deren Höhe verwendet. In Abbildung 6.1 ist die Interpolation der Fußposition zu sehen. Der grüne Punkt steht für die Endposition des Steps und der rote Punkt für die Startposition. Die Endposition liegt auf der orangenen Box. Die grüne Linie zeigt den aktuellen $lift$-Wert an, der wie unten beschrieben, aus einer *AnimationCurve* und der aktuellen Geschwindigkeit berechnet wird und die Fußposition anhebt. Der untere Punkt

der grünen Linie liegt an dem Interpolationspunkt der Start- und Endposition. Der obere Punkt der grünen Linie liegt an der *currentPosition*, welche die Position ist, die für die inverse Kinematik verwendet wird.

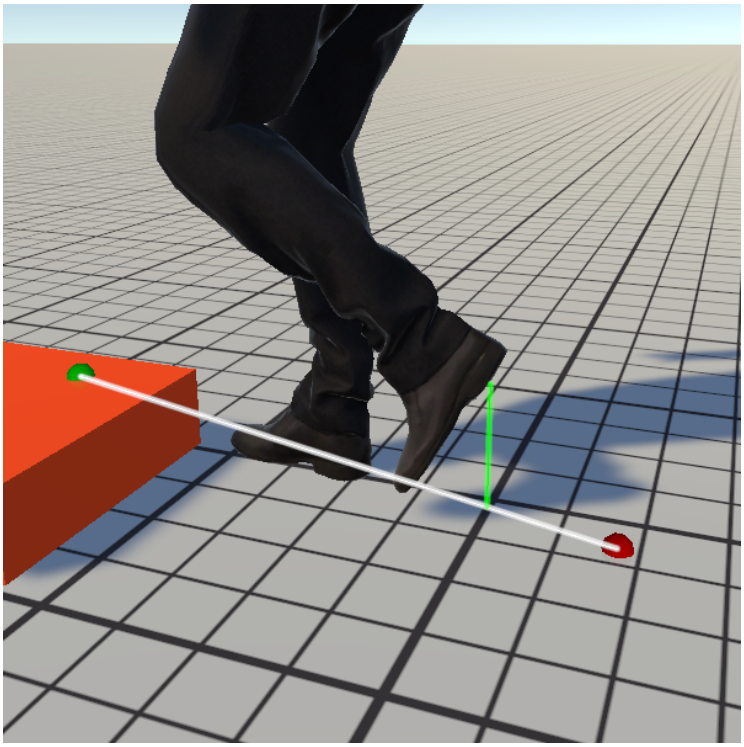

Abbildung 6.1: Interpolation der Fußposition

Für die Steps werden zusätzlich noch weitere Kurven verwendet. Die *StepOffsetDistance*-Kurve (Abbildung 6.2) gibt die Entfernung des Fußes zum Avatar, in dem Moment, in dem der Step beendet wird an. Die X-Achse steht dabei für die Geschwindigkeit des Avatars. Je schneller sich der Avatar bewegt, desto weiter sind auch seine Füße, in dem Moment in welchem der Step beendet wird, vom Körper entfernt. In Sektion 6.8 wird genauer auf den Wert eingegangen.

Die *FootDistanceDirectional*-Kurve (Abbildung 6.2) gibt an, wie weit die beiden Füße voneinander entfernt sind, wobei die X-Achse für die Laufrichtung des Avatars im Verhältnis zu seiner Orientierung steht. Die genaue Funktionsweise wird in den Sektionen 6.7 und 6.8 beschrieben.

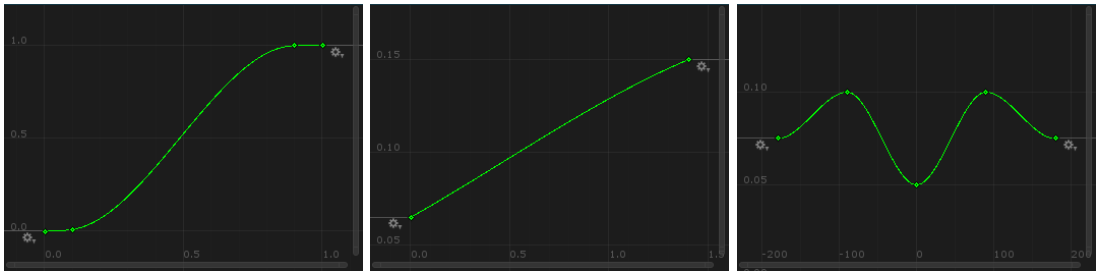

Abbildung 6.2: StepInterpolation, StepOffsetDistance, FootDistanceDirectional

Die *StepOverlapCurve*-Kurve (Abbildung 6.3) gibt an, wie weit ein Step vorangeschritten sein muss, damit mit dem anderen Fuß ein neuer Step begonnen werden kann. Läuft der Avatar langsam, werden die Steps nacheinander ausgeführt. Je schneller sich der Avatar aber bewegt, desto häufiger können sich die Steps überschneiden.

Die beiden Kurven *FootRotationForwardCurve* und die *FootRotationBackwardCurve* (Abbildung 6.3) beschreiben die Rotation des Fußes während eines Steps, jeweils zu dem Zeitpunkt, indem der Avatar sich nach vorne oder nach hinten bewegt. Die X-Achse geht dabei von 0 bis 1 und beschreibt den Fortschritt eines Steps. Die Rotation beim Vorwärtsgehen ist in Abbildung 6.5 zu sehen. Dabei rotiert der Fuß zuerst nach vorne und im Laufe des Steps rotiert er nach hinten, um schlussendlich auf dem Boden eben aufzuliegen. Die Rotation des Fußes beim Rückwärtsgehen ist in Abbildung 6.6 zu sehen. Dort rotiert der Fuß erst nach hinten und dann wieder nach vorne, um am Ende auch wieder eben aufzuliegen. Die Werte der Kurven werden jeweils mit der Geschwindigkeit des Avatars multipliziert. Dadurch rotieren die Füße schwächer, wenn der Avatar sich langsam bewegt. Je mehr sich der Avatar seitwärts bewegt, desto weniger rotieren dabei seine Füße.

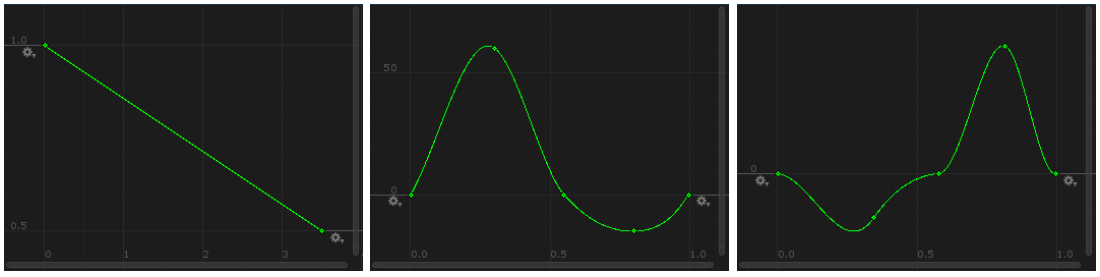

Abbildung 6.3: StepOverlapCurve, FootRotationForwardCurve, FootRotationBackwardCurve

Die drei Kurven *FootLiftForwardCurve*, *FootLiftSidewayCurve* und *FootLiftBackwardCurve* (Abbildung 6.4) geben an, wie hoch der Fuß im Verlauf eines Steps angehoben wird. Je nachdem, in welche Richtung sich der Avatar bewegt werden die Werte aus den einzelnen Kurven verhältnismäßig miteinander gemischt. In den drei Abbildungen 6.5, 6.7 und 6.6 ist jeweils der Ablauf eines Steps in unterschiedlichen Richtungen zu sehen. Dabei ist gut erkennbar, wie der Fuß in unterschiedlichen Richtungen auch unterschiedlich hoch angehoben wird. Der durch die Kurven berechnete Wert wird auch abhängig von der Geschwindigkeit des Avatars vergrößert oder verkleinert, wobei der Avatar beim langsamen Gehen seine Füße kaum anhebt.

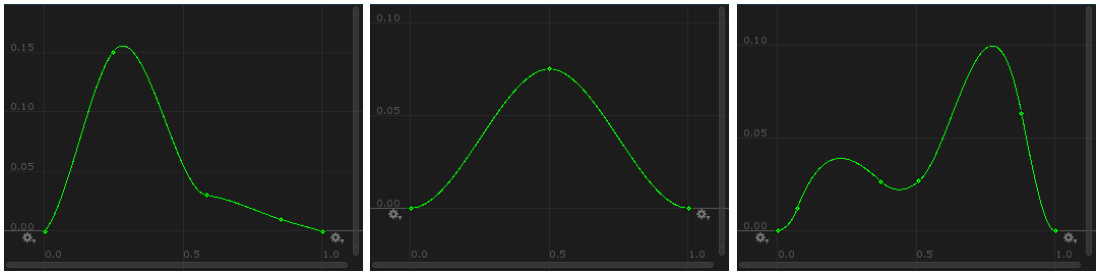

Abbildung 6.4: FootLiftForwardCurve, FootLiftSidewayCurve, FootLiftBackwardCurve

Abbildung 6.5: Der Ablauf eines Steps nach vorne

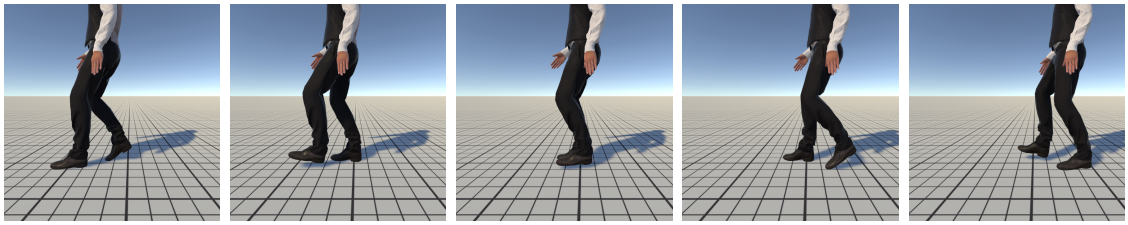

Abbildung 6.6: Der Ablauf eins Steps nach hinten

6.4 Durchschnittsgeschwindigkeit

Um die Positionen der Füße voraussagen zu können, muss die Geschwindigkeit bekannt sein, mit welcher sich die Person in der echten oder der virtuellen Welt bewegt. Dafür wird die Durchschnittsgeschwindigkeit der $Center\ of\ Mass$-Position des Avatars berechnet. Diese Geschwindigkeit wird als 3D Vektor v_{CoM}, der die Durchschnittsgeschwindigkeit der letzten 100ms im 3D Raum enthält, gespeichert. Dazu wird auch $vmag_{CoM}$ bestimmt, welcher die Länge des Geschwindigkeitsvektors v_{CoM} auf der X-Z-Ebene projiziert, darstellt. Ausgehend von der v_{CoM} wird die $direction$ berechnet, welche die Richtung in der sich der Avatar auf der X-Z-Ebene aktuell bewegt angibt. Diese wird wie in der Formel 6.1 zu sehen ist berechnet. Falls $vmag_{CoM}$ kleiner ist als ein vorgegebener Schwellwert (hier $0.125m/s$), wird $direction$ wie in 6.2 berechnet.

$$direction = normalize(vec3(v_{CoM}.x,\ 0,\ v_{CoM}.z)) \tag{6.1}$$

$$direction = normalize(vec3(forwardDirection.x,\ 0,\ forwardDirection.z)) \tag{6.2}$$

6.5 Schrittlänge

Zur Berechnung der Schrittlänge (sl) wird die Formel, welche aus dem Paper [3] stammt, verwendet. Diese wurde aus echten Messdaten hergeleitet und zeigt eine lineare Relation zwischen der Schrittlänge und der Laufgeschwindigkeit (v) bis zu einer Geschwindigkeit von ca. $6.66m/s$. Sie ist in Formel 6.3 zu sehen. Liegt die Laufgeschwindigkeit über $6.66m/s$, erhöht sich die Schrittlänge nicht mehr und bleibt konstant. Für die Formel wurde eine $AnimationCurve$ verwendet und diese wie in Formel 6.4 zu sehen ist verwendet. Die Kurve ist im linken Bild, in Abbildung 6.9, zu sehen.

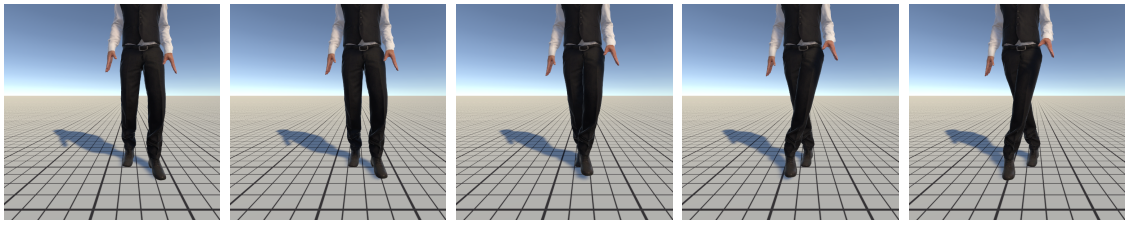

Abbildung 6.7: Der Ablauf eines Steps zu Seite

$$sl = 0.1394 + 0.00465v \tag{6.3}$$

$$sl = StepSizeDirectionCurve.Evaluate(v) \tag{6.4}$$

Je nachdem, wie der Oberkörper im Verhältnis zur Bewegungsrichtung steht, muss zusätzlich die Schrittgröße angepasst, werden um je nach Bewegungsrichtung unterschiedlich große Schritte zu verwenden. Um dies zu realisieren wurde eine *AnimationCurve* definiert, die von -180° bis 180° reicht und einen Wert zurück gibt, welcher mit der, in Formel 6.4 berechneten, Schrittgröße multipliziert wird. Der *time*-Wert der *AnimationCurve* stellt dabei die Bewegungsrichtung im Verhältnis zur $forwardDirection$ des Avatars dar. Läuft der Avatar geradeaus, liegt dieser Wert bei 0. Läuft er zur Seite, liegt dieser bei -90 bzw. +90. Im mittleren Bild in Abbildung 6.9 ist die dafür verwendete *AnimationCurve* zu sehen. Dabei wird beim Vorwärtsgehen die Schrittlänge mit dem Wert 1 multipliziert und bleibt somit unverändert. Beim Seitwärtsgehen wird die Schrittlänge mit dem Wert 0.6 multipliziert und beim Rückwärtsgehen mit dem Wert 0.8, siehe Formel 6.5. Dies hat zur Folge, dass der Avatar vorwärts mit größeren Schritten läuft als zur Seite oder nach hinten.

$$sl' = sl \cdot StepSizeDirectionCurve.Evaluate(direction) \tag{6.5}$$

Zusätzlich zur Bewegungsgeschwindigkeit und der Bewegungsrichtung trägt auch noch die Steigung der Umgebung zur Schrittgröße bei. Läuft der Avatar bspw. einen Berg oder eine Treppe hinauf, ist seine Schrittgröße kleiner, als wenn er sich auf einer flachen Ebene bewegt. Dafür wird, ausgehend von der CoM Position des Avatars, in Laufrichtung wiederholt die Höhe gemessen und dabei die Differenz zwischen dem kleinsten und dem größten gemessenen Wert berechnet. Dieser Wert wird nun durch sl geteilt um einen Steigerungswert zu berechnen. Siehe Formel 6.6. Um das Verhältnis von Steigung und Schrittgröße abzubilden, wird eine *AnimationCurve* verwendet, siehe rechtes Bild in Abbildung 6.9. Diese Kurve gibt einen Multiplikator zurück, welcher mit der in Formel 6.5 berechneten Schrittgröße multipliziert wird, um diese gegebenenfalls anzupassen.

In der Arbeit wurden fünf Messewerte in Laufrichtung gleichmäßig verteilt. In Abbildung 6.8 sind die gemessenen Werte durch die blauen Linien visualisiert. Auf dem linken Bild läuft der Avatar auf einer ebenen Fläche. Die gemessenen Höhen sind deshalb alle gleich und die Differenz zwischen dem größten und dem kleinsten Wert ist 0. Wird dieser Wert nun durch sl dividiert und in die $StepSizeSlopeCurve$ eingesetzt, erhält man den Wert 1 zurück. Dieser Wert wird wie in Formel 6.7 mit dem Wert sl' multipliziert, um die schlussendliche Schrittgröße zu erhalten. Im linken Bild der Abbildung 6.8 liegt die Differenz bei 0, wobei die Schrittgröße unverändert bleibt. Im zweiten

Bild liegt die Differenz über 0, was in der Formel 6.6 ein Wert >= 0 und in Formel 6.7 ein Wert < 1 ergibt. Dadurch erhält man in der Abbildung 6.8 im rechten Bild eine kleinere Schrittgröße als im linken Bild. Der Avatar läuft daher mit kleineren Schritten eine Steigung hoch bzw. herunter als es auf einer ebenen Fläche der Fall wäre.

$$a = \frac{y_{diff}}{sl} \tag{6.6}$$

$$sl'' = sl' \cdot StepSizeSlopeCurve.Evaluate(a) \tag{6.7}$$

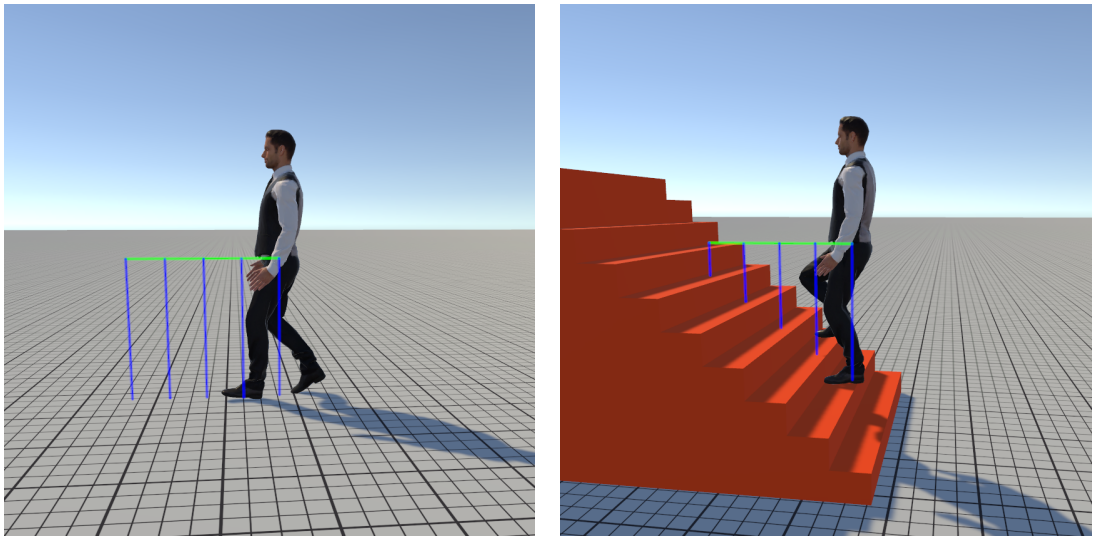

Abbildung 6.8: StepSizeAdjustmentPlane, StepSizeAdjustmentSteps

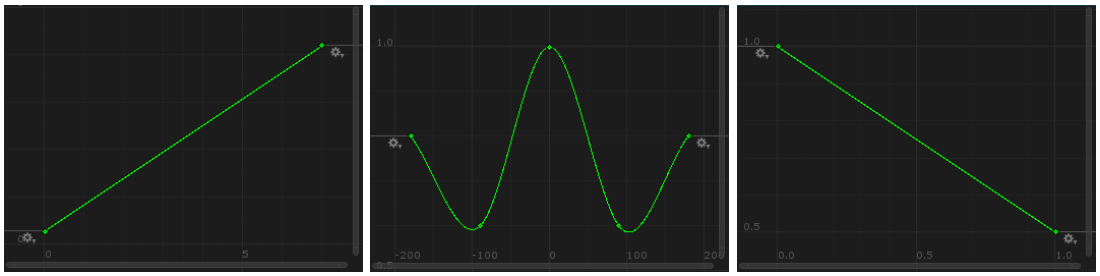

Abbildung 6.9: StepSizeCurve, StepSizeDirectionMultipierCurve, StepSizeSlopeCurve

6.6 Schritttrigger

Um zu entscheiden, ob der Avatar einen Step starten soll werden unterschiedliche Ansätze verfolgt. Einer dieser Ansätze wurde aus dem Artikel [8] genommen. Dafür wird der CoM-Punkt verwendet und abgeglichen, wie dieser im Verhältnis zum $Center\ of\ Pressure$ (CoP) steht. Der $Center\ of\ Pressure$-Punkt befindet sich in der Mitte der beiden Beine des Avatars. Liegt der

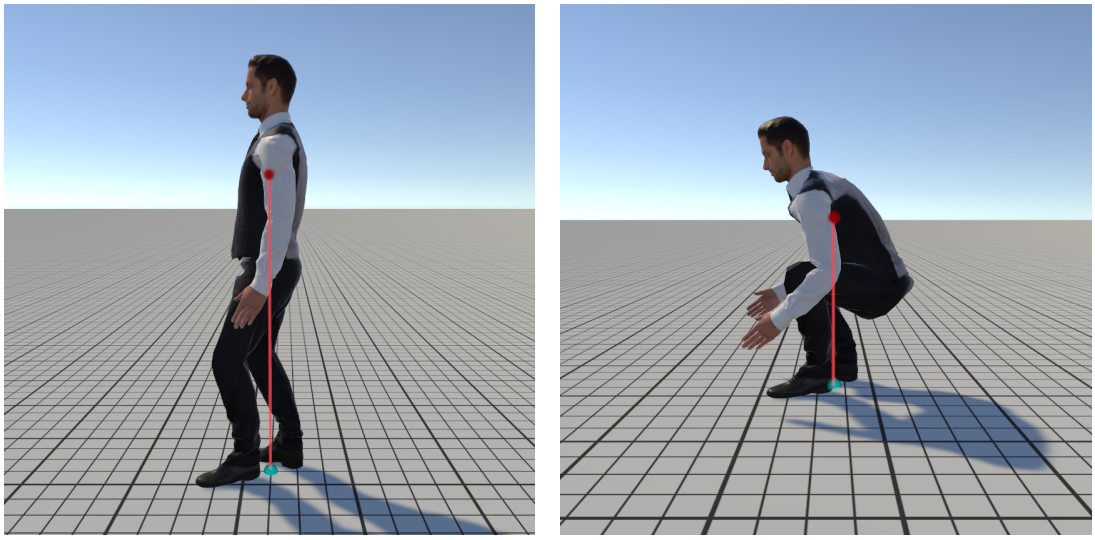

Abbildung 6.10: Der Center of Mass und der Center of Pressure

Winkel zwischen dem Vektor, der durch CoM und CoP verläuft und dem Up-Vektor über einem bestimmten Schwellwert, wird davon ausgegangen, dass der Avatar nicht mehr das Gleichgewicht halten kann und es wird ein neuer Step gestartet. In Abbildung 6.10 ist die CoM-Position in der Brust des Avatars in Form des roten Punktes zu sehen. Der CoP-Punkt liegt zwischen beiden Beinen und ist in Form des cyan-farbigen Punktes zu sehen.

Der zweite Auslöser für einen Step ist der Abstand zwischen den beiden Fußpositionen. Liegt der Abstand unter dem festgelegten $distance_{min}$ oder über dem $distance_{max}$ wird ein neuer Step ausgelöst. Dies soll verhindern, dass der Avatar zu breitbeinig steht oder er seine Beine unnatürlich nahe aneinander hält.

Der dritte Auslöser für einen Step ist das Verhältnis der Position der Füße des Avatars zu seiner $forwardDirection$. Liegt ein Fuß auf der falschen Seite, wird ein neuer Step gestartet. Dies verhindert, dass der Avatar seine Beine überkreuzt.

Ein weiterer Auslöser für einen Step ist der Winkel zwischen dem Fuß und der $forwardDirection$ des Avatars. Liegt dieser Winkel nicht zwischen $minFootRotation$ und $maxFootRotation$, wird für den jeweiligen Fuß ein Step gestartet. In dieser Arbeit wurde dafür ein Bereich von -5° bis 45° verwendet. Dreht sich der Avatar, werden dadurch neue Steps gestartet, um beide Fußrotationen in dem erlaubten Bereich zu behalten und die Beine nicht unrealistisch zu verdrehen.

Wenn einer dieser beschriebenen Fälle eintritt, wird ein neuer Step gestartet. Um zu bestimmten, welcher Fuß sich dafür bewegen soll, wird für beide Füße eine neue Position vorhergesagt (siehe Sektion 6.7). Daraufhin wir der Step für denjenigen Fuß gestartet, welcher sich am weitesten bewegen muss. Falls sich einer der beiden Füße bereits in einem Step befindet, wird automatisch mit dem Fuß, welcher sich aktuell nicht in einem Step befindet ein Step gestartet. Der Ablauf sieht dabei wie im Algorithmus 6.2 aus. Nach dem Starten eines Steps wird die Zeit berechnet die dieser bracht. Diese wird, wie in Sektion 6.9 beschrieben, berechnet.

Algorithmus 6.2 Start eines neuen Steps

```
1: leftPrediction ←PredictFootPosition(left)
2: leftDistance ←leftPrediction - currentFootPositionLeft
3: rightPrediction ←PredictionFootPosition(right)
4: rightDistance ←rightPrediction - currentFootPositionRight
5: if !isSteppingLeft||leftDistance > rightDistance then
6:     StartStep(left)
7: else
8:     StartStep(right)
9: end if
```

6.7 Placement Strategies

Der Lauf-Algorithmus verfolgt zur Platzierung der Füße unterschiedliche Ansätze. Der erste Ansatz wird verwendet, solange die Durchschnittsgeschwindigkeit des Avatars kleiner ist als ein bestimmter Schwellwert und davon ausgegangen wird, dass der Avatar steht. Befindet der Avatar sich im laufenden Zustand, gibt es zwei Strategien zur Platzierung der Füße. Die eine Strategie wird beim Starten eines Steps verwendet, die zweite, wenn der Step schon gestartet wurde. Diese unterschiedlichen Strategien sind nötig, um den stehenden und den laufenden Zustand bestmöglich zu unterstützen.

Für den ersten Ansatz wird zuerst $CoM_{predicion}$ berechnet. Dies ist die geschätzte Position des CoM, in dem Zeitpunkt, in dem der Fuß wieder aufkommen wird. Um diesen Punkt zu berechnen, wird der aktuelle CoM, die verbleibende $stepTime$ und die Durchschnittsgeschwindigkeit verwendet. In dem Codeausschnitt 6.1 ist das genaue Verfahren zu sehen.

```
var comPrediction = new Vector3(CenterOfMass.position.x, 0, CenterOfMass.position.z) +
new Vector3(v_{CoM}.x, 0, v_{CoM}.z) * timeLeft;
```

Listing 6.1: comPrediction

Die neue Position des Fußes, während der Avatar sich im stehenden Zustand befindet, wird wie in Abbildung 6.11 berechnet. Dafür wird zuerst p berechnet (Formel 6.8), welcher entgegen der Position des anderen Fußes p_{other}, ausgehend von $CoM_{predicion}$, liegt. Danach wird geprüft, ob sich der Punkt p auch auf der richtigen Seite des Avatars befindet. Falls das nicht der Fall ist, wird der Punkt p an der Linie $direction$ gespiegelt. Danach wird errechnet, wie weit der Punkt p von der $CoM_{predicion}$-Position entfernt ist. Liegt die Entfernung nicht zwischen dem $distance_{min}$- und dem $distance_{max}$-Wert wird p so positioniert, dass diese zwischen den beiden Werten liegt.

$$p = CoM_{predicion} + (CoM_{predicion} - p_{other}) \tag{6.8}$$

Wird ein neuer Step angefangen, während der Avatar sich im laufenden Zustand befindet, wird die neue Fußposition wie in Abbildung 6.12 berechnet. Die neue Zielposition wird dabei eine halbe $stepSize$ vor den anderen Fuß, in Richtung von $direction$, gelegt. Zuerst wird der Schnittpunkt int_{other} des entgegen liegenden Fußes berechnet. Die eine Linie geht von CoM aus und verläuft in Richtung von $direction$. Die zweit Linie geht von p_{other} aus und verläuft orthogonal zu $direction$. Ausgehend von dem Schnittpunkt wird nun die neue Zielposition p berechnet. Dafür wird von

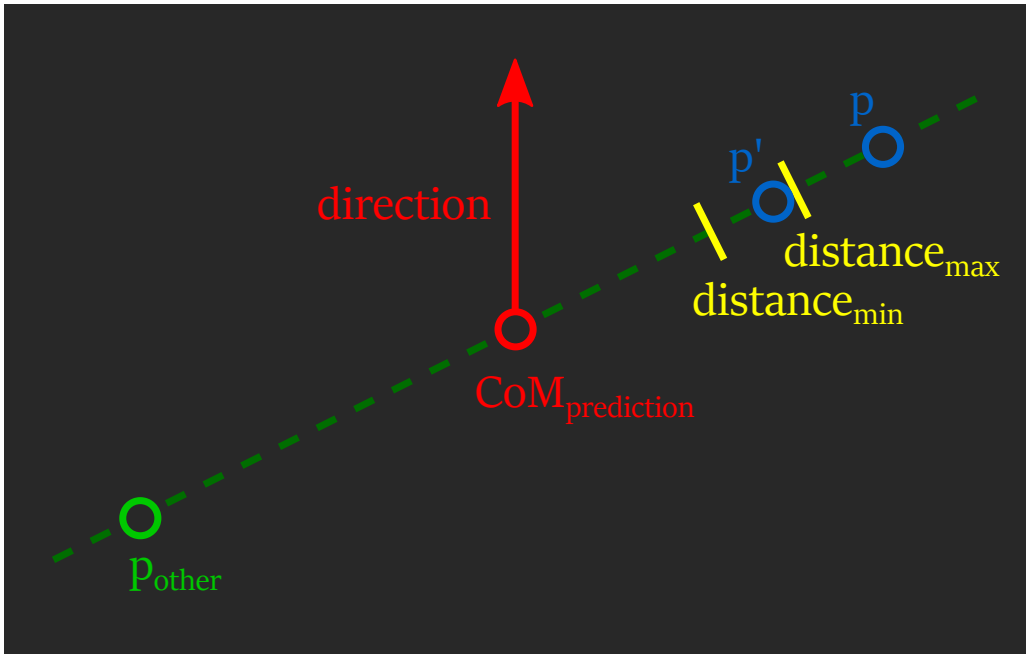

Abbildung 6.11: Positionsberechnung während des Stehens

int_{other} in Richtung von $direction$ mit der Länge $stepSize/2$ gegangen. Anschließend wird von diesem Punkt orthogonal zur $direction$ Linie mit der Länge $footDistance$ gegangen, um dadurch zur neuen Zielposition zu gelangen. Je nachdem, in welche Richtung sich der Avatar im Verhältnis zu seiner $forwardDirection$ bewegt, wird von p entweder nach links oder nach rechts gegangen. Dies wird wie im Codeausschnitt 6.2 berechnet.

```
var fromToAngle = Vector3.SignedAngle(direction, forwardDirection, Vector3.up);
var offsetDirection = (isLeftFoot ^ (-90 < fromToAngle && fromToAngle < 90)) ? 1 : -1;
```

Listing 6.2: comPrediction

Läuft der Avatar, wird die neue Zielposition ausgehend von der $CoM_{prediction}$, wie in Abbildung 6.13 zu sehen ist, berechnet. Dafür wird von $CoM_{prediction}$ in Richtung des $direction$-Vektors mit der Länge $offset$ gegangen und anschließend orthogonal zu $direction$ in der Länge von $footDist$ gegangen. Die Richtung, in welche gegangen wird, wird wie im Codeausschnitt 6.2 berechnet. Die Funktionsweise des $offset$-Wertes wird in der Sektion 6.9 erklärt.

6.8 Schrittplatzierungskorrektur

Ändert der Avatar während eines Steps die Geschwindigkeit oder die Laufrichtung, ändert sich auch die Position, auf der der Fuß aufkommen wird. Deshalb werden in jeder Iteration die Zielpositionen der beiden Füße aktualisiert, um der aktuellen Laufrichtung und Laufgeschwindigkeit zu entsprechen. Wie dies geschieht, ist in Abbildung 6.14 zu sehen. Dafür wird als erstes die neue Zielposition $goal'$, wie in Sektion 6.7 beschrieben, berechnet und der aktuellen Zielposition $goal$ zugewiesen. Danach wird die $start$ Position so angepasst, dass die aktuelle Fußposition bei der Interpolation zwischen

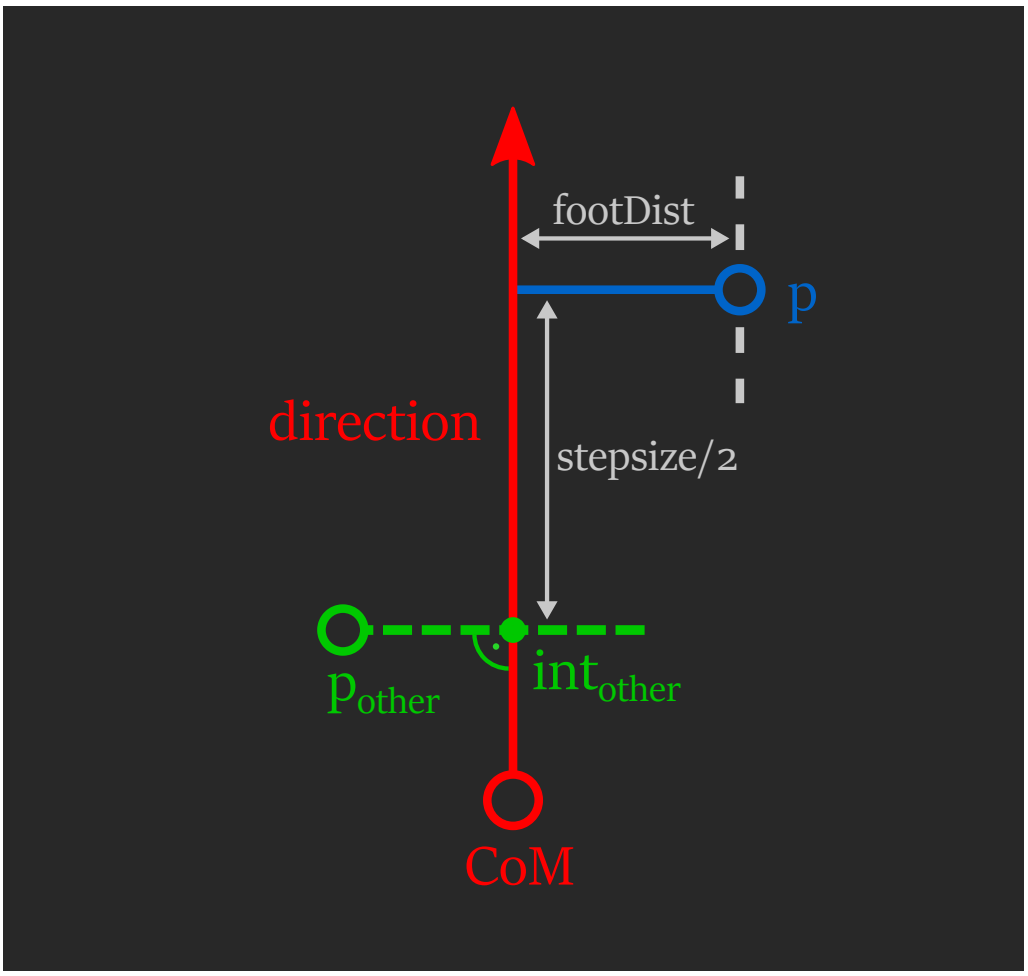

Abbildung 6.12: Positionsberechnung während des Gehens

$start'$ und $goal'$ unverändert bleibt. Dies geschieht, indem man als Erstes die aktuelle Position der Step-Interpolation berechnet, siehe Formel 6.9. Anschließend geht man, mit dem gleichen Längenverhältnis wie zuvor für d_f und d_b, von der aktuellen Fußposition in entgegengesetzte Richtung der neuen Zielposition, siehe Formel 6.10. Entspricht das Verhältnis von d_f und d_b dem von d_f' und d_b', bleibt die interpolierte Fußposition gleich. Dies ist wichtig, damit sich während eines Steps die Fußposition nicht sprunghaft ändert.

$$interpolation = StepInterpolationCurve.Evaluate(stepCount) \tag{6.9}$$

$$start' = position + \frac{position - goal}{1 - interpolation} \cdot interpolation \tag{6.10}$$

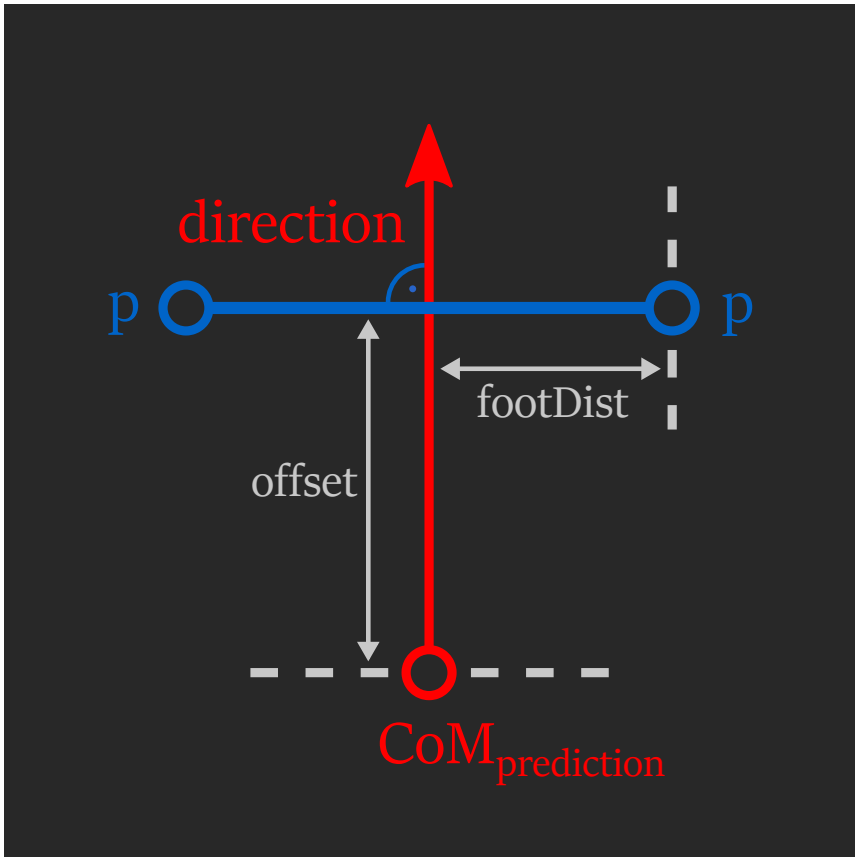

Abbildung 6.13: Fußplatzierung beim Laufen

6.9 Schrittgeschwindigkeitskorrektur

Mit der Schrittgeschwindigkeitskorrektur wird versucht, das Ende eines Steps auf den richtigen Zeitpunkt zu legen. Dafür wird vorerst kontrolliert, ob sich der Avatar langsamer als der Schwellwert bewegt und deshalb davon ausgegangen werden kann, dass der Avatar steht. Ist dies der Fall, wird die Schrittgeschwindigkeit ($stepDuration$) immer auf einen festen Wert gesetzt (hier 300ms). Dies hat zur Folge, dass Steps während des Stehens immer 300ms lang sind.

Kann man jedoch davon ausgehen, dass der Avatar sich bewegt, wird die neue $stepDuration'$ wie folgt berechnet. Damit wird erreicht, dass am Ende eines Steps der Fuß immer in einem bestimmten Abstand ($offset$) von CoM liegt. In Abbildung 6.15 ist diese Berechnung visualisiert. Als Erstes wird hierfür der Punkt p_{offset} berechnet, der ausgehend von der CoM-Position, einen Abstand von $offset$ in Laufrichtung ($direction$) besitzt. Als Nächstes wird der Schnittpunkt $p_{intersection}$ zwischen den beiden Linien $direction$ und der Linie, welche orthogonal zu $direction$ verläuft und durch $goal$ verläuft, berechnet. Nun wird der Vektor $distance$ zwischen dem Punkt p_{offset} und dem $p_{intersection}$ Punkt berechnet. Liegt dieser Vektor entgegen dem $direction$-Vektor, hat der Avatar den Zeitpunkt schon überschritten, in dem er den Fuß hätte absetzten sollen. Ist dies der Fall, wird die neue $stepDuration'$ so klein wie möglich ($minStepTime$) gesetzt, um den Step schnellstmöglich zu beenden. Andernfalls wird die $stepDuration'$, wie in Formel 6.11 und 6.12

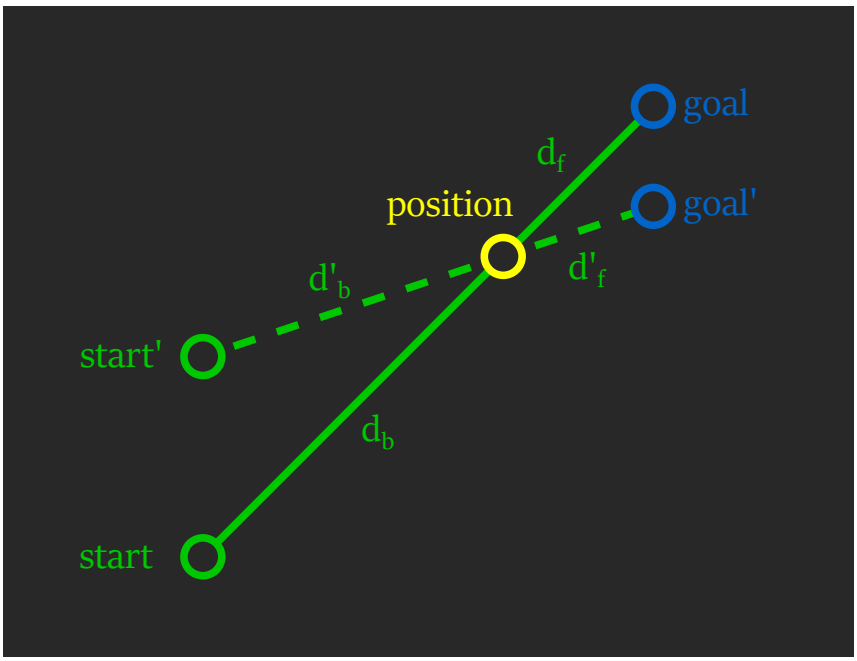

Abbildung 6.14: Berechnung der neuen Zielposition

berechnet. Anschließend wird dieser Wert eingeschränkt, um zu verhindern, dass der Avatar Steps durchführt, welche unrealistisch lange brauchen oder viel zu kurz sind, siehe Formel 6.13. Für die beiden Werte *minStepTime* und *maxStepTime* wurden in dieser Arbeit die Werte 0.1 und 1.5 verwendet. Dies sorgt dafür, dass ein Step minimal 100ms und maximal 1500ms lang sein darf.

$$diff = \frac{abs(distance)}{vmag_{CoM}} \tag{6.11}$$

$$stepDuration' = \frac{diff}{1 - stepCount} \tag{6.12}$$

$$stepDuration'' = clamp(stepDuration', minStepTime, maxStepTime) \tag{6.13}$$

Der Wert für $offset$ wird mittels der $stepOffsetDistance$-Kurve berechnet und hängt von der Bewegungsgeschwindigkeit ab. Die Kurve ist im mittleren Bild in Abbildung 6.2 zu sehen. Je schneller sich der Avatar bewegt, desto größer wird auch der Abstand zwischen ihm und dem aufgekommenen Fuß. Im linken Bild in Abbildung 6.16 ist der $offset$, in Form der grünen Linie, beim langsamen Gehen zu sehen. Im rechten Bild ist die Distanz beim schnelleren Gehen zu sehen. Mittels der *AnimationCurve* kann diese Distanz flexibel den unterschiedlichen Gangstielen angepasst werden.

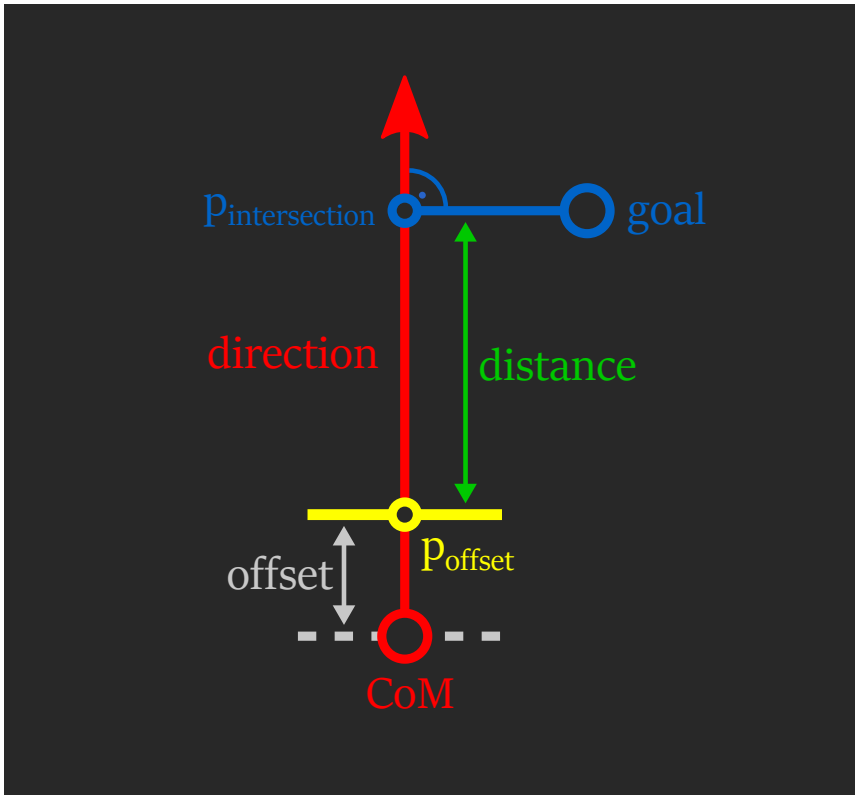

Abbildung 6.15: Berechnung der neuen *stepDuration*

6.10 Erdung

Um zu verhindern, dass der Avatar, wie im linken Bild der Abbildung 6.17 mit einem Bein in der Luft schwebt, muss die $TargetOffset$-Position so angepasst werden, dass beide Beine in der Lage sind, den Boden zu berühren. Dies geschieht, indem für beide Füße die Höhe des Terrains gemessen wird, um dann die $TargetOffset$-Position so anzupassen, dass beide Beine den Boden berühren können.

Befindet sich ein Bein im stehenden Zustand, wird dafür einfach die Y-Position von dessen $currentPosition$ verwendet. Dies ist in der Formel 6.14 für den linken Fuß zu sehen. Falls sich das Bein aber aktuell in einem Step befindet, wird zwischen der Y-Position der Startposition des Steps und der Höhe der Zielposition des Steps ($goalHeight$) interpoliert. Dies ist für den linken Fuß in der Formel 6.15 zu sehen. Anschließend wird, um den neuen $TargetOffset.y$-Wert zu erhalten, zwischen dem letzten $TargetOffset.y$-Wert und dem kleinsten Höhenwert der beiden Füße (Formel 6.16) interpoliert, siehe Formel 6.17. Dafür wird die $lerp$-Funktion mit dem Wert 0.5 verwendet. Dies sorgt dafür, dass der Avatar nicht plötzlich vom einen zum anderen Moment die Höhe ändert, sondern flüssig von der einen in die andere Position übergeht.

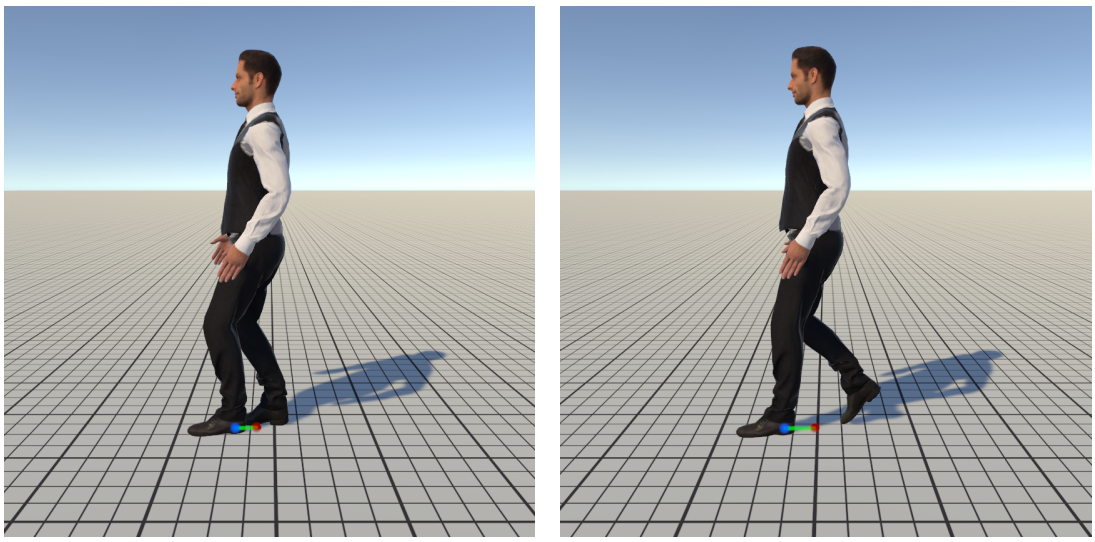

Abbildung 6.16: links: Offset bei langsamen Gehen; **rechts:** Offset beim schnelleren Gehen

$$target_{left} = currentPosition_{left}.y \tag{6.14}$$
$$target_{left} = lerp(currentPositionStart_{left},\ goalHeight_{left},\ stepCount_{left}) \tag{6.15}$$
$$target = min(target_{left},\ target_{right}) \tag{6.16}$$
$$TargetOffset.y = lerp(TargetOffset.y,\ target,\ 0.5) \tag{6.17}$$

Das Ergebnis davon ist in Abbildung 6.17 im rechten Bild zu sehen. Die Positionen des VR-Headsets und der Controller werden mittels dem $TargetOffset$ Wert so nach unten korrigiert, dass der Avatar in der Lage ist, mit beiden Beinen den Boden zu berühren.

Zusätzlich dazu wird auch noch dafür gesorgt, dass die Füße eben auf dem Boden liegen. Dabei wird für jeden Fuß eine senkrechte Linie nach unten gezogen. Von dieser Linie wird der erste Schnittpunkt berechnet. Ausgehend von dem Schnittpunkt und der Normalen an diesem Schnittpunkt (welche in Unity standardmäßig zurückgegeben wird), wird die Rotation des Fußes so angepasst, dass dieser eben auf dem Boden liegt. Das Resultat davon ist in Abbildung 6.18 zu sehen.

6.11 Kopfbewegung

Beim Gehen bewegt sich der Kopf einer Person auf und ab. In der Gangsynthese wird dies durch das Hoch- und Herunterbewegen des Kopfes in Relation zur aktuellen Fußposition simuliert. Dafür wird für beide Füße, wie in Abbildung 6.19 zu sehen ist, die Distanz d_l und d_r von der CoM zu den Fußpositionen auf der X-Z-Ebene berechnet. Die Positionen p_{left} und p_{right} sind dabei die aktuellen Positionen der Füße auf der X-Z-Ebene.

Die $target$-Variable gibt an, was der aktuelle Zielwert des $head_{offset}$-Wertes ist. Der $head_{offset}$-Wert wird auf den Y-Wert der aktuellen Kopfposition aufaddiert. Befinden sich beide Füße auf dem Boden, wird die Distanz mit dem kleineren Wert verwendet, siehe Formel 6.18. Befindet

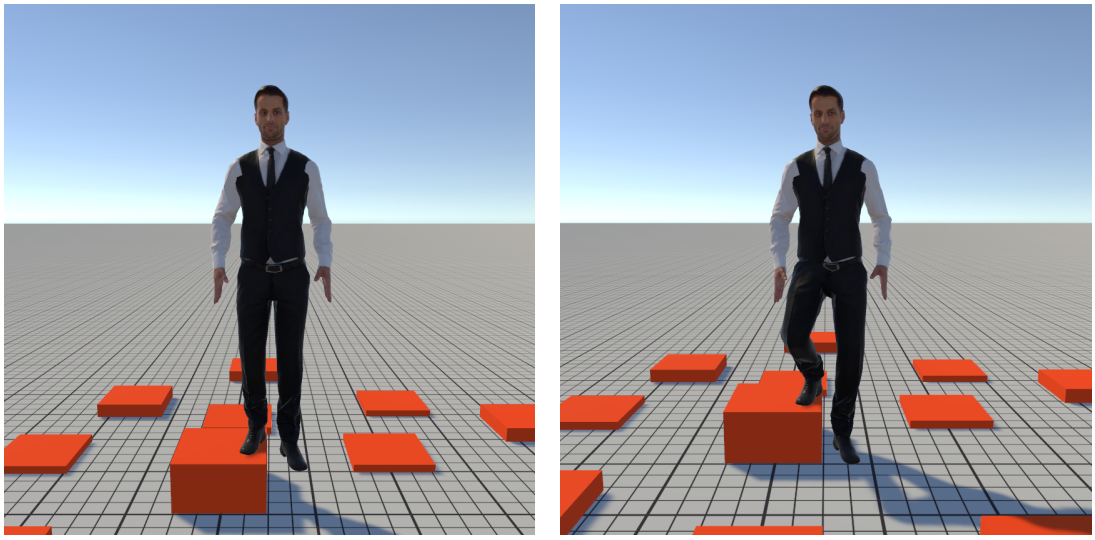

Abbildung 6.17: links: Ohne Erdung der Beine; **rechts:** Mit Erdung der Beine

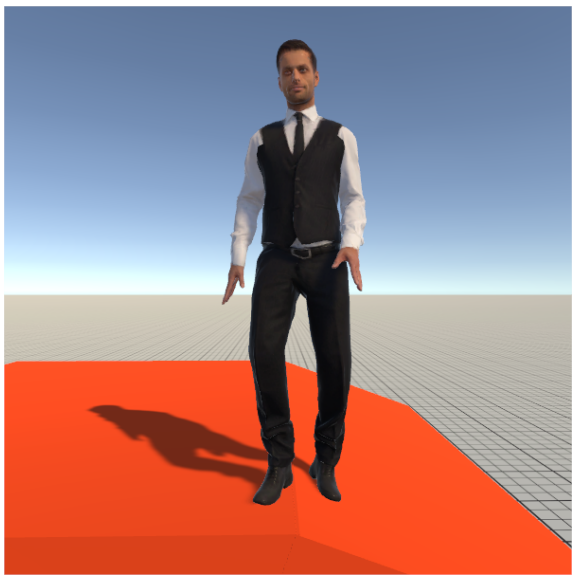

Abbildung 6.18: Anpassung der Füße an die Rotation des Bodens

sich nur ein Fuß auf dem Boden, wird dessen Distanz zum CoM verwendet. Die Formel für den linken Fuß ist in 6.19 und für den rechten Fuß in 6.20 zu sehen. Befinden sich beide Füße aktiv in einem Step, wird $target$ auf den Wert 0 gesetzt. Die berechneten Distanzwerte werden jeweils mit der $headOffsetMultiply$-Konstanten multipliziert, wodurch die Intensität der Kopfbewegung beeinflusst werden kann. Die $headOffsetLerpSpeed$-Variable gibt an, wie schnell sich der $head_{offset}$-Wert dem neu berechneten $target$-Wert nähert, siehe Formel 6.21. In dieser Arbeit wurde für $headOffsetMultiply$ der Wert 0.1 und für $headOffsetLerpSpeed$ der Wert 0.05 verwendet. Dadurch bewegt sich der Kopf beim Laufen leicht auf und ab. Der vergleichsweise kleine Wert für $headOffsetLerpSpeed$ verhindert hierbei das plötzliche Springen des $head_{offset}$.

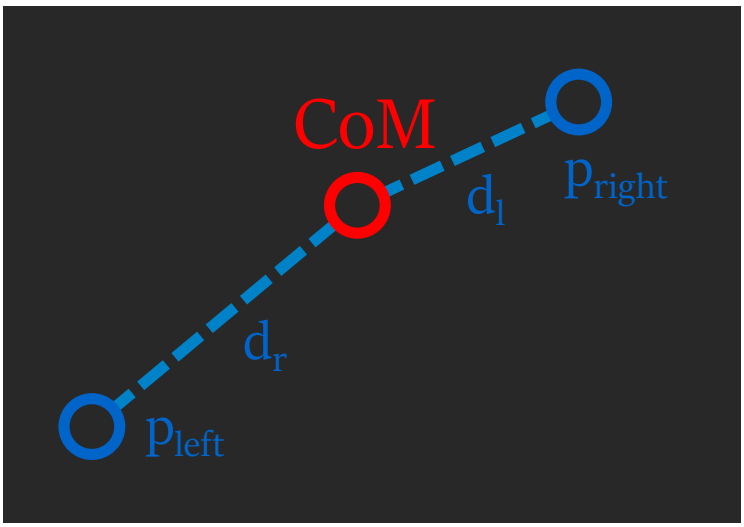

Abbildung 6.19: Berechnung der Kopfbewegung

$$target = min(d_l,\ d_r) \cdot headOffsetMultiply \tag{6.18}$$
$$target = d_l \cdot headOffsetMultiply \tag{6.19}$$
$$target = d_r \cdot headOffsetMultiply \tag{6.20}$$
$$head_{offset} = lerp(lastOffset,\ target,\ headOffsetLerpSpeed) \tag{6.21}$$

6.12 Schulter- und Hüftschwung

Während des Gehens bewegt der Avatar neben seinem Kopf auch noch seinen Oberkörper, sowie die Hüfte. Das Ausmaß der Bewegung wird, wie in Abbildung 6.20 zu sehen ist, berechnet. Als erstes wird eine Linie, ausgehend von der CoM, orthogonal zu dem $direction$-Vektor gezogen. Danach werden für beide Füße die Schnittpunkte dieser Linie mit der Linie, welche durch die aktuelle Fußposition in Richtung des $direktion$-Vektors verläuft, berechnet. Für beide dieser Schnittpunkte wird ein Vektor berechnet, der vom Schnittpunkt zur aktuellen Position des Fußes verläuft. Für diese Vektoren werden die Längen (mit Vorzeichen) in Richtung des $direction$-Vektors berechnet. Diese beiden Werte d_{left} und d_{right} werden von einander subtrahiert und man erhält das Ausmaß des Schulter- bzw. des Hüftschwunges, siehe Formel 6.22. Die jeweiligen Werte für den Schulter- bzw. den Hüftschwung werden mittels des $swing$-Wertes und einer Konstanten berechnet. Dies macht es möglich, Konstanten mit unterschiedlichen Vorzeichen und Größen zu verwenden, um die Schulter und die Hüfte in jeweils entgegengesetzte Richtungen mit unterschiedlicher Stärke zu bewegen.

$$swing = d_{left} - d_{right} \tag{6.22}$$

In Abbildung 6.21 sind die Vektoren d_{left} und d_{right} als grüne Linien zu sehen. Im ersten Bild bewegt sich der Avatar nach vorne und im zweiten Bild bewegt sich dieser zur Seite. Beim Vorwärtsgehen wechseln die beiden Werte ihre Vorzeichen und deren Längen, wodurch sich der *swing*-Wert auch immer vom positiven Bereich in den negativen und von dort wieder in den positiven bewegt. Beim Seitwärtsgehen bleiben beide Vektoren ungefähr gleich, wodurch sich der *swing*-Wert nur unwesentlich ändert. Der Oberkörper und die Hüfte bewegen sich also beim Vorwärts- und Rückwärtsgehen mit, beim Seitwärtsgehen aber nicht.

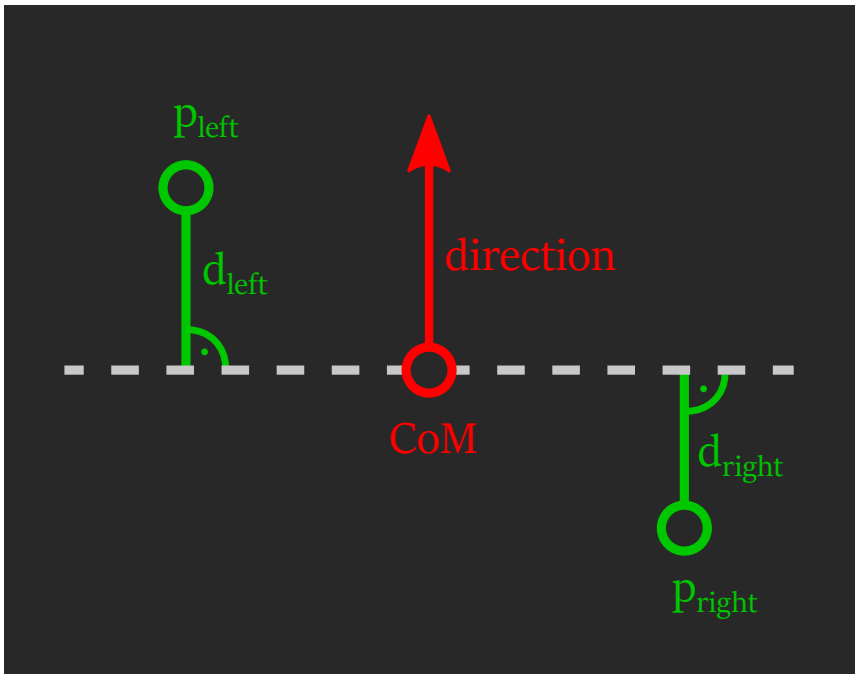

Abbildung 6.20: Schulter- und Hüftschwung

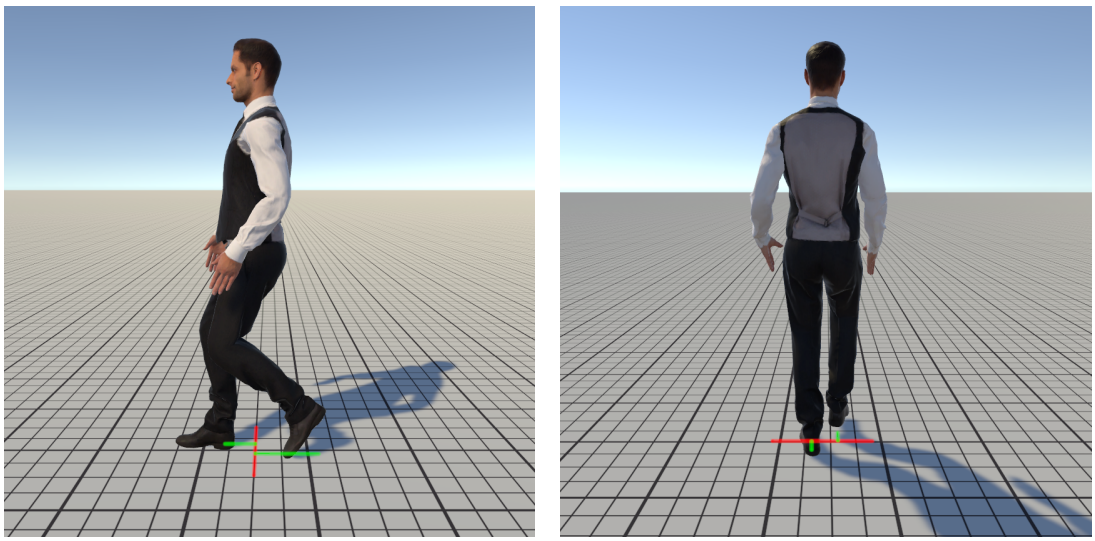

Abbildung 6.21: Berechnung des Schulter- und Hüftschwungs
links: Beim Vorwärtsgehen; **rechts:** Beim Seitwärtsgehen

7 Zusammenfassung und Ausblick

Bisher existieren nur wenige komplette Ansätze zur Avatar-Posen-Schätzung speziell für VR-Umgebungen. In dieser Arbeit wurden bestehende Ansätze untersucht und es wurde aus dem gesammelten Wissen eigene Ansätze zur Lösung des Problems entwickelt. Speziell der Magnetfeld-Algorithmus aus Sektion 4.4, welche in dieser Arbeit entwickelt wurde, bietet einen interessanten Lösungsansatz, um den FABRIK-Algorithmus für die inverse Kinematik in VR einzusetzen. Mit der, in dieser Arbeit entwickelten Gangsynthese wurde versucht, Probleme bestehender Ansätze zu umgehen und eine speziell für VR geeignete Gangsynthese zu entwickeln. Die Abschätzung der Pose eines Avatars in VR bietet viele unterschiedliche Herausforderungen, welche wahrscheinlich alle noch lange Verbesserungspotenzial bieten. Das Ergebnis dieser Arbeit ist in den Abbildungen 7.1 und 7.2 zu sehen.

Ausblick

Aktuell werden keine Rotations-Constraints ,um die eigene Achse eines Gelenks in der inversen Kinematik verwendet, da der FABRIK-Algorithmus diese nicht direkt unterstützt. Es wäre daher interessant, eine Lösung zu finden, wie man diese in den FABRIK-Algorithmus integrieren könnte, um somit Dinge, wie die Rotation der Hand in extremen Posen besser unterstützten zu können.

Die in dieser Arbeit erarbeitete Gangsynthese lässt sich durch die, von ihr verwendeten *AnimationCurves*, sehr flexibel anpassen. Durch die hohe Anzahl an Kurven ist es jedoch recht aufwendig, unterschiedliche Gangarten zu realisieren. Interessant wäre es daher, diese Kurven aus bestehenden Laufanimationen zu generieren. In der Masterarbeit [6] wird beschrieben, wie so etwas in Zukunft aussehen könnte. Für speziell diese Gangsynthese müssten dafür aber noch weitere Ansätze erarbeitet werden.

Die Synthese der Oberkörper-Position in dieser Arbeit ist mittels Trial-and-Error entstanden. Um die Posen-Schätzung des Oberkörpers genauer zu gestalten, wäre es interessant, die aktuelle Lösung mit echten Motion Capture Daten zu vergleichen, um diese dann ausgehend davon zu verbessern. Ein interessanter Ansatz könnte sein, mittels Machine Learning die verwendeten Parameter des Oberkörper-Models zu optimieren, um möglichst optimale Werte zu finden, welche den echten Motion Capture Daten möglichst nahekommen.

Um einen Avatar zu gestalten, der die Pose des Benutzers bestmöglich widerspiegelt, ist es wichtig, dass dieser auch dieselben Maße wie der Benutzer besitzt. In dieser Arbeit wurde dafür nur die Kopfhöhe des Benutzers gemessen und davon ausgehend die Skalierung des Avatars angepasst. Es wäre aber deutlich sinnvoller, die Maße des Avatars genauer anpassen zu können und bspw. die Armlänge in weiteren Kalibrierungsschritten zu erhalten. Ideal wäre ein System, das ohne Kalibrierung auskommt und die Maße des Benutzers während der Laufzeit automatisch anpasst. Ausgehend von den gemessenen Werten könnte dabei auch noch die Gangsynthese individuell

angepasst werden. In [3] wird bspw. die Abhängigkeit der Körperhöhe zur Schrittlänge beschrieben. Dies und andere Daten könnten dabei helfen, eine möglichst realistische Posen-Schätzung zu gestalten, welche für Personen mit unterschiedlichen Maßen gut funktioniert.

Abbildung 7.1: Unterschiedliche Avatar-Posen

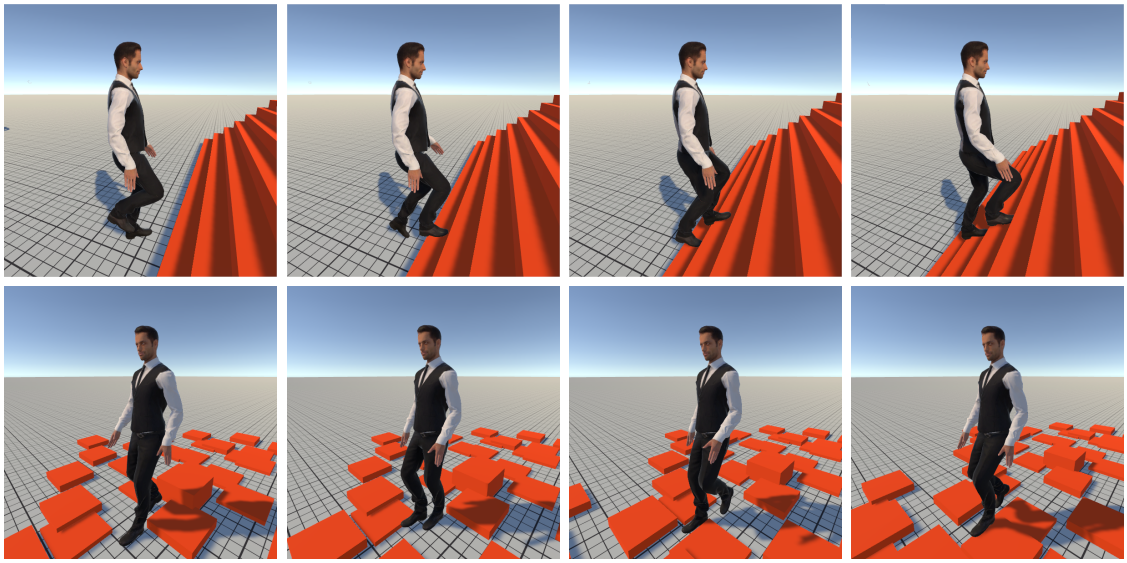

Abbildung 7.2: Laufanimationen in unterschiedlichen Umgebungen

Literaturverzeichnis

[1] Andreas Aristidou, Joan Lasenby. „FABRIK: A fast, iterative solver for the Inverse Kinematics problem“. In: *Graph. Models* 73.5 (Sep. 2011), S. 243–260. ISSN: 1524-0703. DOI: 10.1016/j.gmod.2011.05.003. URL: http://dx.doi.org/10.1016/j.gmod.2011.05.003.

[2] Andreas Aristidou, Joan Lasenby. *Inverse Kinematics: a review of existing techniques and introduction of a new fast iterative solver*. Sep. 2009.

[3] Armin Bruderlin, Tom Calvert. „Knowledge-driven, interactive animation of human running“. In: *Proceedings of the Graphics Interface 1996 Conference, May 22-24, 1996, Toronto, Ontario, Canada*. Canadian Human-Computer Communications Society, Mai 1996, S. 213–221. ISBN: 0-9695338-5-3. URL: http://graphicsinterface.org/wp-content/uploads/gi1996-25.pdf.

[4] Zhiquan Gao et al. „Leveraging Two Kinect Sensors for Accurate Full-Body Motion Capture“. In: *Sensors (Basel, Switzerland)* 15 (Sep. 2015), S. 24297–24317. DOI: 10.3390/s150924297.

[5] Fan Jiang, Xubo Yang, Lele Feng. „Real-time Full-body Motion Reconstruction and Recognition for Off-the-shelf VR Devices“. In: *Proceedings of the 15th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry - Volume 1*. VRCAI '16. Zhuhai, China: ACM, 2016, S. 309–318. ISBN: 978-1-4503-4692-4. DOI: 10.1145/3013971.3013987. URL: http://doi.acm.org/10.1145/3013971.3013987.

[6] R.S. Johansen. *Automated Semi-procedural Animation for Character Locomotion*. Aarhus Universitet, Institut for Informations- og Medievidenskab, 2009. URL: https://books.google.de/books?id=AIVCtwAACAAJ.

[7] Pärtel Lang. *Final IK Rootmotion*. http://www.root-motion.com/final-ik.html. [Online; accessed 03-May-2019]. 2019.

[8] Pärtel Lang. *Inverse Kinematics in Dead and Buried*. http://root-motion.com/2016/06/inverse-kinematics-in-dead-and-buried/. [Online; accessed 13-April-2019]. 2016.

[9] Mathias Parger et al. „Human Upper-body Inverse Kinematics for Increased Embodiment in Consumer-grade Virtual Reality“. In: *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*. VRST '18. Tokyo, Japan: ACM, 2018, 23:1–23:10. ISBN: 978-1-4503-6086-9. DOI: 10.1145/3281505.3281529. URL: http://doi.acm.org/10.1145/3281505.3281529.

[10] D. Roth et al. „A simplified inverse kinematic approach for embodied VR applications“. In: *2016 IEEE Virtual Reality (VR)*. März 2016, S. 275–276. DOI: 10.1109/VR.2016.7504760.

[11] Zhipeng Tan, Yuning Hu, Kun Xu. „Virtual Reality Based Immersive Telepresence System for Remote Conversation and Collaboration“. In: *Next Generation Computer Animation Techniques*. Hrsg. von Jian Chang et al. Cham: Springer International Publishing, 2017, S. 234–247. ISBN: 978-3-319-69487-0.

Alle URLs wurden zuletzt am 08. 05. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift