



Peter Kopylov

Enhancing Task Performance through Predictive Motion Planning and Human-Machine Interaction in Collaborative Environments

Bachelorarbeit
zur Erlangung des Universitätsabschlusses
Bachelor of Science
Informatik

eingereicht bei
Technische Universität Berlin

Betreuer: M.Sc. Teham Bhuiyan
Erstprüfer: Prof. Dr.-Ing. Jens Lambrecht
Zweitprüfer: Prof. Dr.-Ing. Sebastian Möller

Industry Grade Networks and Clouds

14. Februar 2024

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 14. Februar 2024

Kurzfassung

Kürzliche Entwicklungen im Bereich des Deep Reinforcement Learnings (DRL) und der Robotik zeigen vielversprechende Ansätze für die Ausführung komplexer Bewegungsaufgaben in dynamischen Umgebungen durch Verbindung der beiden Bereiche. Die Robotik ist bereits jetzt von besonderem Nutzen für die Industrie. Gepaart mit DRL könnte insbesondere die Produktionsleistung massiv steigen, wenn dadurch Industrieroboter ihre Umgebung wahrnehmen und folglich mit menschlichen Arbeitskräften direkt und ohne Sicherheitsbedenken zusammenarbeiten könnten.

Ein Schlüsselement für den erfolgreichen Einsatz von DRL in der Robotik ist die Umgebungswahrnehmung. Diese war meist kein Hauptaugenmerk existierender Arbeiten.

In dieser Arbeit wird eine Perception-Pipeline vorgestellt, die diese Lücke schließen soll. Diese soll in der Forschung, sowie für experimentelles Training und Deployment von Robotern verwendet werden können. Dabei nimmt die vorgestellte Perception-Pipeline ausschließlich menschliche Akteure wahr.

Über die Integration von Tiefenkameras mit OpenPose, einem neuronalen Netzwerk, das menschliche Posen erkennt, rekonstruiert die Pipeline einen Menschen drei-dimensional in Echtzeit als ein aus simplen geometrischen Körpern bestehendes Männchen in PyBullet. Entwickelte konventionelle Algorithmen versuchen, fehlerhafte Posen, die beispielsweise durch Verdeckung mit einem anderen Objekt entstehen können, zu erkennen und zu beheben. Die menschliche Pose kann aufgenommen werden, um auf realistischen Daten zu trainieren, ohne die rechenaufwändige Wahrnehmung simultan laufen lassen zu müssen. Die Pipeline ist in andere PyBullet-Umgebungen ohne viel Aufwand integrierbar.

Zusammengefasst schließt diese Arbeit die Wahrnehmungslücke in der DRL-Robotik und stellt eine Perception-Pipeline vor, die die menschliche Pose in Echtzeit drei-dimensional rekonstruiert.

Abstract

Recent developments in the field of deep reinforcement learning (DRL) and robotics show promising approaches for the execution of complex motion tasks in dynamic environments by combining the two areas. Robotics is already of particular benefit to industry. Paired with DRL, production output in particular could increase massively if industrial robots were able to perceive their surroundings and consequently collaborate with human workers directly and without safety concerns.

A key element for the successful use of DRL in robotics is environmental awareness. This has not usually been a key focus of existing work.

This thesis presents a perception pipeline that aims to close this gap. It is intended to be used in research as well as for experimental training and the deployment of robots. The perception pipeline presented here only perceives human actors.

By integrating depth cameras with OpenPose, a neural network that recognizes human poses, the pipeline reconstructs a human three-dimensionally in real time as a figure consisting of simple geometric bodies in PyBullet. Developed conventional algorithms attempt to recognize and correct incorrect poses, which can occur, for example, due to occlusion with another object. The human pose can be recorded to train on realistic data without having to run the computationally expensive perception simultaneously. The pipeline can be easily integrated into other PyBullet environments.

In summary, this work closes the perception gap in DRL robotics and presents a perception pipeline that reconstructs the human pose in real time in three dimensions.

Inhaltsverzeichnis

Abbildungsverzeichnis	viii
Tabellenverzeichnis	x
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	2
2 Verwandte Arbeiten	3
3 Grundlagen	6
3.1 Deep Reinforcement Learning	6
3.2 Roboterkontrolle	8
3.3 Wahrnehmung	10
4 Konzeptentwurf	13
4.1 Grundlegene Funktionsweise	13
5 Implementierung	17
5.1 Module	17
5.1.1 main.py	17
5.1.2 config.py	17
5.1.3 perceptor.py	17
5.1.4 simulator.py	20
5.1.5 openpose_handler.py	23
5.1.6 custom_3d_joint_transforms.py	23
5.1.7 debug.py	24
5.2 Validierung	24
5.2.1 Algorithmus	24
5.2.2 Kalibrierung	31
5.3 Konfiguration	32
5.4 Integration in IR-DRL	35
6 Evaluation	36
6.1 Theoretische Limitierungen	36
6.1.1 OpenPose	36
6.1.2 Deprojektion	36

Inhaltsverzeichnis

6.1.3	Verschiebung und Rotation	36
6.1.4	Validierung	37
6.2	Korrektheit	48
6.3	Robustheit	55
6.4	Performance	58
7	Fazit	61
8	Weiterführende Arbeiten	63
8.1	Wahrnehmung	63
8.2	Training und Deployment	64
Literatur		65

Abbildungsverzeichnis

3.1	Die Elemente des Reinforcement Learnings	7
3.2	Relation direkte und inverse Kinematik	10
3.3	Kernmodule typische Perception-Pipeline	10
4.1	Visualisiertes Konzept der vorgestellten Pipeline	14
4.2	Erkennung der Pose im Color-Stream	15
4.3	Visualisierte Tiefenwerte	16
4.4	Resultat der Simulation	16
5.1	Idee der Längenvalidierung: Ein Joint ist verdeckt	25
5.2	Idee der Längenvalidierung: Zwei Joints sind verdeckt	25
5.3	Alle Verbindungen werden erkannt	29
5.4	Verdeckung einiger Joints	29
5.5	Suchbereich	30
5.6	Farbmaske	30
6.1	Visualisierung des Falls „Ein Joint verdeckt“	37
6.2	Funktion f für die Verbindung Schulter-Ellbogen	43
6.3	Funktion f für die Verbindung Nacken-Hüfte	44
6.4	Numerische Statistiken für die Gesamtlänge, auf der die Validierung bei einem verdeckten Joint fehlschlagen wird, pro Verbindung	44
6.5	Visualisierung des Falls „Zwei Joints verdeckt“	46
6.6	Numerische Statistiken für die Gesamtlänge, auf der die Validierung bei zwei verdeckten Joints fehlschlagen wird, pro Verbindung	48
6.7	Funktion f_2 für die Verbindung Schulter-Ellbogen	49
6.8	Funktion f_2 für die Verbindung Nacken-Hüfte	49
6.9	Joint-Geschwindigkeiten aus einer Aufnahme	52
6.10	Der Anteil invalider Joint-Verbindungen einer Aufnahme	52
6.11	Die absoluten Differenzen der zu langen invaliden Verbindungen zu den Maximallängen	53
6.12	Die absoluten Differenzen der zu kurzen invaliden Verbindungen zu den Minimallängen einer Aufnahme	54
6.13	Die absoluten Differenzen der Verbindungen, die eine zu hohe Tiefenabweichung haben, zu den maximalen Tiefendifferenz einer Aufnahme	54

Abbildungsverzeichnis

6.14 Der Anteil valider Joints an der Anzahl der Frames einer Aufnahme.	56
6.15 Histogramm der Strähnen für den Hüft-Joint.	56
6.16 Histogramm der Strähnen für den Joint des linken Handgelenks.	57
6.17 Reale Performance im Test.	59
6.18 Performance einer abgespielten Bag-Datei im Test.	60

Tabellenverzeichnis

5.1 Konfiguration Perceptor	32
5.2 Konfiguration Simulator	34
6.1 Hardware	58
6.2 Software	58

1 Einleitung

1.1 Motivation

Roboterarme, auch robotische Manipulatoren genannt, spielen in der heutigen Industrie eine zentrale Rolle. Ob in Fertigungslien der Automobilindustrie, der Elektrobranche oder der Logistik, Manipulatoren sind aufgrund ihrer Effizienz und Präzision weltweit im Einsatz. Jedoch sind diese Systeme durch ihre feste Codierung in ihrer Flexibilität beschränkt. So werden diese häufig von menschlichen Arbeitskräften aus Sicherheitsgründen räumlich getrennt, da traditionelle Manipulatoren sich ihrer Umgebung nicht bewusst sind und sonst ein potenzielles Risiko für Menschen darstellen würden. Diese Einschränkung geht auf die Kosten der Effizienz, denn würden Menschen mit Manipulatoren direkt zusammenarbeiten, könnte sich die Gesamtleistung stark erhöhen.

Zur Überwindung dieser Einschränkung verlagert sich der Schwerpunkt der Forschung zunehmend auf den Einsatz von DRL (Deep Reinforcement Learning) in der Robotik. DRL ermöglicht Robotern, sich durch Lernen aus Interaktionen mit ihrer Umgebung neuen Situationen anzupassen. So ist es vorstellbar, dass ein Manipulator seine Aufgabenstellung in einer Produktionslinie erfüllt, während er zu keinem Zeitpunkt einen direkt nebenan arbeitenden Menschen gefährdet und diesem gegebenenfalls ausweicht.

Typischerweise findet das Training in simulierten Umgebungen statt. Dies beschleunigt das Training, da mehrere Simulationen gleichzeitig laufen können. Zusätzlich vergeht simulierte Zeit häufig schneller als die reale - wodurch Agenten schneller lernen. Weiterhin kann der Agent in der Simulation ausgiebig getestet werden, bevor er in der Realität eingesetzt wird, was Sicherheitsrisiken minimiert.

Ein Schlüsselement für den erfolgreichen Einsatz von DRL ist die Umgebungswahrnehmung des Manipulators. Diese muss präzise genug und möglichst verzögerungsfrei sein, um effektive und sichere Interaktionen in Echtzeit zu ermöglichen. Darüber hinaus könnte der Agent durch realistische Trainingsdaten besser lernen.

Angesichts dieser Herausforderungen stellt die vorliegende Arbeit eine

Perception-Pipeline vor, die nur den wesentlichsten Aspekt der Umgebung erfasst und verarbeitet: den Menschen. Diese Pipeline soll sowohl in Echtzeit im Deployment, als auch zur Aufnahme von realistischen Trainingsdaten eingesetzt werden können.

1.2 Zielsetzung

Der Schwerpunkt dieser Arbeit liegt auf der Entwicklung einer Perception-Pipeline, die einen Menschen in Echtzeit wahrnehmen kann. Zusätzlich werden klassische Algorithmen implementiert, die diese Pipeline präziser und robuster machen, um fehlerhaft wahrgenommene Posen, die hauptsächlich durch Verdeckung mit einem anderen Objekt entstehen, zu erkennen und zu korrigieren.

Weiterhin wird eine umfassende Konfiguration der Pipeline ermöglicht, um verschiedenen Bedürfnissen und Umgebungen gerecht zu werden. Die Erhöhung der Präzision und Robustheit bedarf einer relativ genauen Konfiguration. Daher werden Debug-Tools bereitgestellt, die diese erleichtern sollen.

Neben der Wahrnehmung wird ein Simulator entwickelt, der den wahrgenommenen Menschen in PyBullet mit simplen geometrischen Formen in Echtzeit darstellt. Die Kollisionserkennung mit potenziellen anderen Objekten der Simulation (z.B. bei Integration in andere Programme, siehe nächster Punkt) wird ermöglicht.

Zusätzlich soll die Pipeline in andere PyBullet-Programme ohne viel Aufwand integriert werden können. Beispielhaft wird das an einer exemplarischen Integration in die IR-DRL-Suite [9] demonstriert.

Auch wird eine Aufnahme- und Abspielfunktion entwickelt. Aufnahmen können z.B. zum Trainieren von Robotern benutzt werden. Die gesamte Pipeline kann der Wahrnehmung im experimentellen Deployment dienen. Schließlich wird die Präzision, Robustheit und Performance evaluiert.

2 Verwandte Arbeiten

In den letzten Jahren wurden bedeutende Fortschritte in der Industrierobotik unter Einsatz von Deep Reinforcement Learning (DRL) erzielt. Insbesondere lag der Schwerpunkt auf der Bewegungsplanung in Echtzeit unter Vermeidung von (menschlichen) Hindernissen.

Anfangs lag der Fokus auf traditionellen Methoden der Bewegungsplanung:

Chen et al. [1] stellten eine wissensgesteuerte Planung unter der Verwendung von „Relative State Trees“ vor, reduzierten benötigten Speicherplatz und verbesserten die Echtzeit-Performance von mobilen Robotern.

Um Problemen wie lokalen Minima und ineffizienten Pfaden bei Einsatz von traditionellem APF (Artificial Potential Field) entgegenzuwirken, entwickelten Lee et al. [15] eine auf den Lidar-Sensor spezialisierte NP-APF (New-Point-APF) Methode.

Die Limitierungen von RRT (Rapidly Exploring Random Tree) für den Einsatz in der Bewegungsplanung in dynamischen und unstrukturierten Umgebung wurde von Wei et al. [22] erfolgreich angegangen, indem ein verbesserter Smoothly-RRT (S-RRT) Algorithmus verwendet wurde, der die Pfadoptimierungsstrategie auf der maximalen Krümmung basiert.

Xinyu et al. [24] verbesserten ebenfalls den klassischen RRT, und optimierten diesen für den Einsatz in engen Kanälen unter Nutzung von weniger Speicherplatz.

Dennoch werden traditionelle Algorithmen der Bewegungsplanung von Problemen geprägt. Insbesondere das mangelnde Anpassungsvermögen macht den Einsatz dieser Ansätze in komplexen, dynamischen Umgebung und für breitere Tätigkeitsbereiche zur Herausforderung.

Kästner et al. [11] entwickelten eine Suite zum Trainieren, Testen und Evaluieren von Navigationsplanern. Erfolgreich konnten sie mehrere Planer in dieser Suite trainieren.

Einen interessanten Ansatz verfolgten Dugas et al. [4], indem sie unüberwachte Repräsentationen für Reinforcement Learning in dynamischen, menschlichen Umgebungen verwendeten, und entdeckten, dass diese eine konsistenter Performance in generellen Aufgaben liefern.

Ein weiteres neuronales Netz von Everett et al. [5] war in der Lage, einen Roboter im Schritttempo in einer Umgebung mit Menschen und anderen

Robotern erfolgreich zu navigieren.

DRL-Ansätze zeigen häufig überlegene Performance gegenüber klassischen Methoden in hochdynamischen Umgebungen. Jedoch behalten klassische Methoden häufig die Überhand in der Navigation über längere Strecken, da DRL-Ansätze u.a. anfällig für das Problem der lokalen Minima sind. Daher entwickelten Kästner et al. in [12] und [16] Methoden, um klassische und DRL-Methoden sinnvoll miteinander zu kombinieren.

Auffällig ist, dass die Agenten dieser Studien sich zwar in 3D-Umgebungen befinden, jedoch nur in einer flachen 2D-Ebene operieren. Industrielle Roboter operieren jedoch im deutlich komplexeren 3D-Raum. Wo bei der Navigation in der Ebene ein Lidar-Sensor für einfache Abstandsmessungen genügt, sind die Anforderungen an die Wahrnehmung für einen im 3D-Raum operierenden Roboter deutlich höher.

Erste Ansätze waren auch hier konventioneller Natur:

Mohammed et al. [3] verwendeten Tiefenkameras, um, abhängig vom Abstand zwischen Mensch und Manipulator, den Roboter zu stoppen, seinen Pfad zu manipulieren oder einen menschlichen Remote-Operator zu alarmieren.

Safeea et al. [19] verfolgten einen ähnlichen Ansatz. Ein vorgenerierter Pfad für einen Manipulator wird in Echtzeit durch die Verwendung Abstoßungs- und Anziehungsvektoren modifiziert, wenn das System einen Menschen im Arbeitsraum des Manipulators, die beide intern durch geometrische Primitive dargestellt werden, identifiziert.

Doch auch hier untersuchten Forscher bereits die mögliche Verwendung von DRL:

Lucchi et al. [17] demonstrierten in einem Experiment die Fähigkeit von DRL, mit dem Endeffektor eines Roboterarms einen Punkt in einer Halbkugel anzusteuern - jedoch ohne Hindernisse.

Einen weiteren, rein simulierten Ansatz präsentierte Sangiovanni et al. [20]. Erfolgreich trainierten sie mit DRL einen Agenten, der mit dem Endeffektor ein Ziel erreichte, während der Arbeitsraum von Hindernissen randomisiert durchquert wurde. Jedoch befand sich das Ziel stets auf einer definierten Ebene. In einer weiteren Arbeit [21] kombinierten sie klassische Algorithmen für eine Greifbewegung mit einem DRL-Ansatz zur Kollisionsvermeidung. Das System schaltet auf DRL-Kontrolle um, sobald definierte Bedingungen bezüglich der Distanz eines Hindernisses erfüllt sind.

Einen rein auf DRL basierenden Ansatz präsentierte Zhu et al. [25], den sie erfolgreich im Deployment testeten. Mit DRL lernte der Manipulator, Kollisionen zu vermeiden. Als Beobachtung wurden rohe Tiefenbilder ver-

2 Verwandte Arbeiten

wendet, die zur Dimensionsreduktion durch ein Auto-Encoder-Netzwerk geschickt wurden.

Die meisten dieser Ansätze legen starken Fokus auf den DRL-Aspekt. Die Wahrnehmung ist häufig nicht der Fokus der Arbeiten. Daher werden oft nur zufällig generierte oder von Hand definierte Simulationen von Hindernissen verwendet. Viele Ansätze verbleiben auch ausschließlich in der Simulation. Dies zeigt die Notwendigkeit einer Perception-Pipeline, die sowohl im Deployment als auch zur Aufnahme von realistischen Trainingsdaten verwendet werden kann, was Schwerpunkt dieser Arbeit ist.

3 Grundlagen

3.1 Deep Reinforcement Learning

In Anlehnung an [14] werden im Folgenden die Grundlagen zu Deep Reinforcement Learning erklärt.

Reinforcement Learning ist eine Technik des maschinellen Lernens, in der ein Agent Aktionen in einer Umwelt durchführt, die ihm bestimmte Beobachtungen ermöglicht. Der Agent erlernt selbstständig eine Strategie zur Erreichung eines Ziels, indem seine Aktionen belohnt oder bestraft werden. „Deep“ bezieht sich auf die Nutzung von Deep Learning. Das ist eine Methode des maschinellen Lernens, die vielschichtige neuronale Netzwerke nutzt. Diese sind auf der Funktionsweise des menschlichen Gehirns basiert und sollen durch das Trainieren auf großen Datenmengen komplexe Zusammenhänge als Funktion approximieren können.

Dem Deep Reinforcement Learning liegt somit ein neuronales Netz zugrunde, das durch Methoden des Reinforcement Learnings trainiert wird. Die einzelnen Elemente, die in Abb. 3.1 skizziert sind, werden im Folgenden erläutert:

Belohnung

Die Belohnung ist ein Skalar, der angibt, wie gut sich der Agent benommen hat. Diesen erhält der Agent von der Umgebung - meist periodisch jeweils alle n Zeitschritte. Wichtig ist, dass die Belohnung *lokal* ist, sodass sie nur den kürzlichen Erfolg - und nicht einen über die Zeit akkumulierten Erfolg - repräsentiert. Der Agent versucht, die akkumulierte Belohnung zu maximieren.

Im Kontext dieser Arbeit wäre eine mögliche Belohnung eine Funktion, in die die Nähe des Endeffektors des Roboters zum Endziel positiv einfließt - und die Nähe des Roboters zum menschlichen Hindernis negativ. Bei Kollision gäbe es eine hohe negative Belohnung - und bei Erreichen des Ziels eine hohe positive.

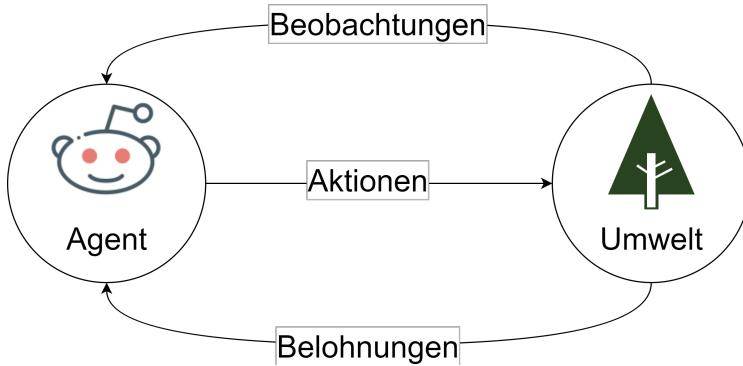


Abbildung 3.1: Die Elemente des Reinforcement Learnings. In Anlehnung an eine Figur in [14].

Agent

Der Agent ist eine Instanz, die mit der Umwelt durch definierte Aktionen interagiert und Beobachtungen und Belohnungen erhält.
Im Kontext dieser Arbeit ist der Agent der Roboter.

Umwelt

Die Umwelt ist alles außerhalb des Agenten. Der Austausch mit der Umwelt findet über Beobachtungen, Belohnungen und Aktionen statt. Häufig ist nicht die gesamte Umwelt beobachtbar.

Im Kontext dieser Arbeit ist die Umwelt die Simulationsumgebung, in der sich der Roboter befindet. Diese beinhaltet unter anderem den simulierten Menschen.

Aktionen

Aktionen sind beliebig komplexe Handlungen, die der Agent ausführen kann. Meist wirken sich Aktionen auf die Umwelt aus.

Im Kontext dieser Arbeit sind die Aktionen von kontinuierlicher Form und werden in Abschnitt 3.2 näher erläutert. Mögliche Aktionen sind die Geschwindigkeiten der Robotergelenke.

Beobachtungen

Beobachtungen sind Informationen, die der Agent von der Umwelt erhält.
Im Kontext dieser Arbeit ist die Beobachtung die Positionen der Gliedmaßen des Menschen im Raum.

3.2 Roboterkontrolle

In Anlehnung an [2] werden im Folgenden die Grundlagen der Roboterkontrolle erklärt.

Die im Kontext dieser Arbeit referenzierten Roboterarme sind eine Art von mechanischer Manipulatoren. Manipulatoren sind die beweglichen Teile eines Roboters und bestehen aus starren Gliedern, die durch Gelenke miteinander verbunden sind und so gewisse Bewegungen ermöglichen. Im Kontext dieser Bachelorarbeit handelt es sich dabei um Drehgelenke, sodass die Positionierung zweier verbundener Glieder zueinander in *Winkeln* angegeben werden kann.

Der *Freiheitsgrad* gibt an, wie viele Positionsvariablen angegeben werden müssen, um alle Teile des Mechanismus zu bewegen. Im Falle von typischen Roboterarmen entspricht der Freiheitsgrad häufig der Gelenkanzahl, da eine Gelenkposition meist mit einer einzigen Variablen (e.g. dem Winkel) angegeben wird.

Am freien Ende des Roboters befindet sich der Endeffektor, an den verschiedene Geräte montiert werden können. In unserem Fall soll der Endeffektor eine bestimmte Position erreichen, während der gesamte Roboterarm nicht kollidiert.

Gelenakraum und kartesischer Raum

Die Positionierung eines Manipulators wird typischerweise in zwei verschiedenen Räumen angegeben:

- **Gelenakraum:** Die Position des Manipulators wird hier durch die Parameter seiner Gelenke beschrieben. Im Falle eines Roboterarms mit Drehgelenken ist der Parameter des Gelenks i der Winkel θ_i zwischen den beiden Gliedern des Gelenks. Hat dieser Roboterarm n Gelenke und n Freiheitsgrade, so lässt sich seine Position im Gelenakraum durch einen n -dimensionalen Vektor angeben, mit je einem Eintrag pro Gelenk:

$$\boldsymbol{\theta} = (\theta_1, \theta_2, \dots, \theta_n) \quad (3.1)$$

- **Kartesischer Raum:** Im kartesischen Raum wird typischerweise die Position und Orientierung des Endeffektors angegeben. Die Position wird meist mit den Koordinaten $[x, y, z]$ und die Orientierung mit Roll-Nick-Gier-Winkeln $[\alpha, \beta, \gamma]$ (spezielle Eulerwinkel zur Beschreibung einer Ausrichtung) angegeben.

Damit ergibt sich ein sechs-dimensionaler Vektor zur Angabe der

Position im kartesischen Raum:

$$\mathbf{p} = [x, y, z, \alpha, \beta, \gamma] \quad (3.2)$$

Zu beachten ist, dass eine Position im kartesischen Raum durch mehrere Positionen im Gelenkraum erreicht werden kann.

Kinematik

Kinematik ist die Lehre der Bewegung, ohne die Kräfte zu beachten, die diese erzeugen. Die Abbildungen vom Gelenkraum Θ in den kartesischen Raum P und umgekehrt sind die Grundlage der robotischen Bewegung. Diese sind schematisch in Abb. 3.2 skizziert.

- **Direkte Kinematik:** Die direkte Kinematik beschäftigt sich mit der Frage, was die Position und Orientierung des Endeffektors sind, wenn die Winkel der Gelenke θ gegeben sind. Da ein Element des Gelenkraums $\theta \in \Theta$ auf genau ein Element des kartesischen Raums $p \in P$ abbildet, betrachten wir mathematisch gesehen eine Funktion:

$$f: \Theta \rightarrow P \quad (3.3)$$

Die direkte Kinematik lässt sich algorithmisch relativ einfach lösen.

- **Inverser Kinematik:** Die inverse Kinematik beschäftigt sich mit der Frage, was die Winkel der Gelenke sein können, wenn die Position und Orientierung des Endeffektors $p \in P$ gegeben sind.

Im Gegensatz zur direkten Kinematik könnte eine Position und Orientierung des Endeffektors durch unterschiedliche Winkel erreicht werden. Dies macht diese Berechnung deutlich komplexer.

Mathematisch gesehen, suchen wir alle Winkel $\theta \in \Theta$, die auf ein festes Element des kartesischen Raums $p \in P$ abbilden:

$$\{\theta \in \Theta \mid f(\theta) = p\} \quad (3.4)$$

Kontrollmodi

Ist die direkte und inverse Kinematik implementiert, so lässt sich der Manipulator in mehreren Modi kontrollieren:

- Gelenkwinkelmodus: Der Manipulator erhält einen Gelenkwinkelvektor und richtet sich danach aus.
- Gelenkwinkelgeschwindigkeitsmodus: Zum Zeitpunkt t erhält der Manipulator einen Gelenkwinkelgeschwindigkeitsvektor.
- Kartesischer Positionsmodus: Der Manipulator erhält Position und Orientierung des Endeffektors und richtet sich mithilfe inverser Kinematik danach aus.
- Kartesischer Geschwindigkeitsmodus: Der Manipulator erhält eine lineare und eine Winkelgeschwindigkeit für den Endeffektor.

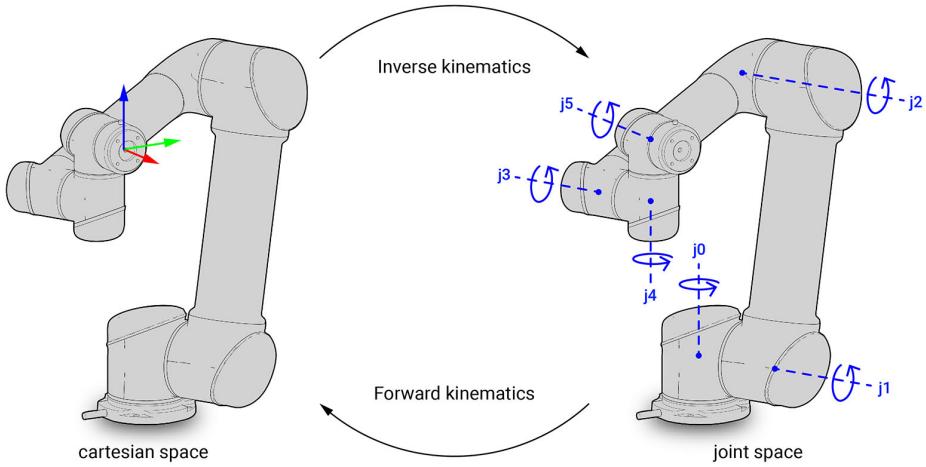


Abbildung 3.2: Die Relation zwischen direkter und inverser Kinematik. Auf der linken Seite steht das eingezeichnete Koordinatensystem stellvertretend für die Angabe der Position und Orientierung des Endeffektors im kartesischen Raum, siehe Gleichung (3.2). Eine Abb. entnommen aus [6].

3.3 Wahrnehmung

Präzise Wahrnehmung ist besonders wichtig für robotische Applikationen, da diese dem Roboter erlaubt, in einer realen Welt zu operieren. Grobe Ungenauigkeiten, geringe Bildraten und hohe Verzögerungen in der Perception-Pipeline könnten zu Planungsfehlern führen.

Wie in Abb. 3.3 illustriert, besteht eine typische Perception-Pipeline für Roboter aus drei Kernmodulen, die im Folgenden erklärt werden [18]:

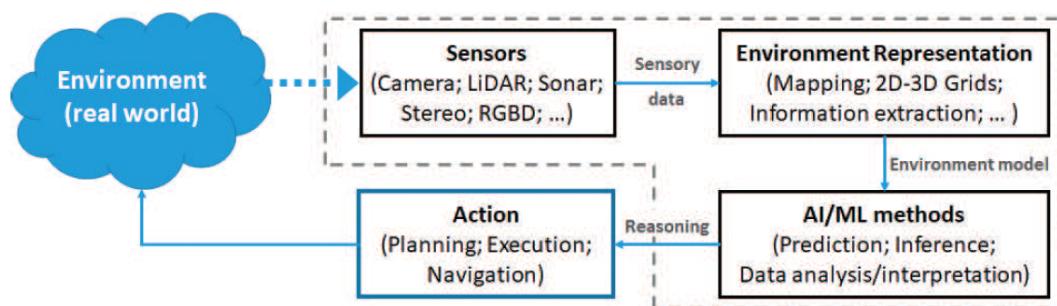


Abbildung 3.3: Kernmodule einer typischen robotischen Perception-Pipeline. Figure 1 in [18].

Sensorik

Unabhängig vom weiteren Vorgehen braucht es Sensoren, um Daten der Umgebung zu erhalten. Es können (Kombinationen von) Sensoren unterschiedlichster Art verwendet werden. Da robotische Applikationen in Abhängigkeit ihrer Umgebung operieren sollen, liegt es nahe, Sensoren zu verwenden, die ihre Umgebung drei-dimensional wahrnehmen können. Dazu gehören u.a.:

- **Lidar-Systeme:** Lidar nutzt Laserstrahlen, um Entferungen zu Objekten zu messen. Zu den Vorteilen gehören hohe Genauigkeit und gute Leistung bei allen Lichtverhältnissen. Nachteilig sind hohe Kosten und begrenzte Farbinformationen.
- **Ultraschallkameras:** Ultraschallkameras verwenden Schallwellen zur Entfernungsmessung. Sie kosten wenig und sind einfach zu handhaben. Im Gegenzug bieten sie eine relativ geringe Auflösung und Genauigkeit.
- **RGBD-Kameras:** RGBD-Kameras kombinieren Farbbilder mit Tiefeninformationen. Daraus ergibt sich eine detaillierte Farb- und Tiefenwahrnehmung, die sich für Objekterkennung und -interaktion eignet. Jedoch ist die Reichweite relativ gering und die Qualität von guten Lichtverhältnissen abhängig.

Im Kontext dieser Arbeit spielt die Wahrnehmung von Menschen bei guten Lichtverhältnissen in Kameranähe eine Rolle, daher wird eine RGBD-Kamera als Sensor verwendet.

Umgebungsrepräsentation

Der Schritt der Umgebungsrepräsentation ist essenziell für die Vorverarbeitung der rohen Daten des Sensors. Verschiedene Ansätze der Umgebungsrepräsentation reichen dabei von rein metrischen Modellen der Umgebung bis hin zu höheren semantischen Darstellungen. Dies ist wichtig, um dem Roboter für den weiteren Verlauf sinnvolle Informationen bereitzustellen.

Als Beispiel für eine eher metrische und weniger semantische Repräsentation eignen sich Punktewolken, die von RGBD-Kameras erzeugt werden können. Dies sind entsprechend gefärbte Tiefenkarten der Umgebung. Jeder Punkt $[x, y, z]$ der Punktewolke mit Farbe $[r, g, b]$ wurde aus einem Pixel $[p_1, p_2]$ mit Farbe $[r, g, b]$ und einer assoziierten Tiefe z' berechnet.

Im Rahmen dieser Arbeit wird direkt ein drei-dimensionales Modell eines Menschen aus einer RGBD-Aufnahme erzeugt. Dies stellt eine semantisch höhere Repräsentation dar.

Interpretation

Die Daten aus dem letzten Schritt werden nun vom Roboter analysiert und interpretiert. Häufig kommen an diesem Punkt Methoden des maschinellen Lernens zum Einsatz. Im Anschluss dient die Interpretation der Umgebungsrepräsentation der Planung und Ausführung von Aktionen durch den Roboter. Interpretation und Aktion müssen nicht unbedingt voneinander getrennt werden, es kann sinnvoll sein, diese beispielsweise innerhalb eines neuronalen Netzes aneinander zu koppeln.

4 Konzeptentwurf

Das Ziel dieses Projekts ist es, eine Perception-Pipeline speziell für Mensch-Maschine-Interaktionen zu entwickeln, um Kollisionen von Robotern mit Menschen zu vermeiden. Im Vergleich zur Voxelisierung soll vor allem die Performance verbessert werden. Die Pipeline soll sowohl für das Training als auch für das Deployment eingesetzt werden können, daher muss die gesamte Pipeline mit geringer Verzögerung und hoher Framerate laufen.

Die Perception-Pipeline nimmt einen Menschen mithilfe einer Tiefenkamera wahr und bildet diesen in einer 3D-Simulation als Männchen ab. Dabei soll das Programm vielseitig und einfach konfigurierbar sein, um unterschiedlichen Bedürfnissen, Umgebungen und Set-Ups gerecht zu werden. Zusätzlich soll es ohne viel Aufwand in die existierende IR-DRL [9] integrierbar sein.

4.1 Grundlegene Funktionsweise

Die grundlegende Funktionsweise der Pipeline ist in Abb. 4.1 skizziert und sieht dabei folgendermaßen aus:

- Rohe RGBD-Daten werden von der Kamera wahrgenommen und in die Pipeline eingespeist. RGBD steht dabei für "Red-Green-Blue-Depth". Somit beinhaltet dieser RGBD-Stream nicht nur die übliche Farbkomponente eines Bildes, sondern auch die Tiefe des jeweiligen Pixels.
- Die RGB-Komponente (Color-Stream) des derzeitigen Frames wird an OpenPose übergeben. *OpenPose* ist ein neuronales Netz, das die Joint-Positionen eines Menschen auf einem Bild erkennt. Die Joints repräsentieren das Skelett des Menschen. Dazu gehören unter anderem: Kopf, Nacken, Schultern, Ellbogen, Handgelenk, Hüfte usw. OpenPose gibt die Pixel dieser Joints zurück. Zu sehen ist dies in Abb. 4.2.
- Ein Pixel kann mithilfe der Tiefeninformation im Depth-Stream (dargestellt in Abb. 4.3) in eine 3D-Koordinate deprojiziert werden. Somit werden im nächsten Schritt die Joint-Pixel zusammen mit der Tiefe zu 3D-Joint-Positionen kombiniert.

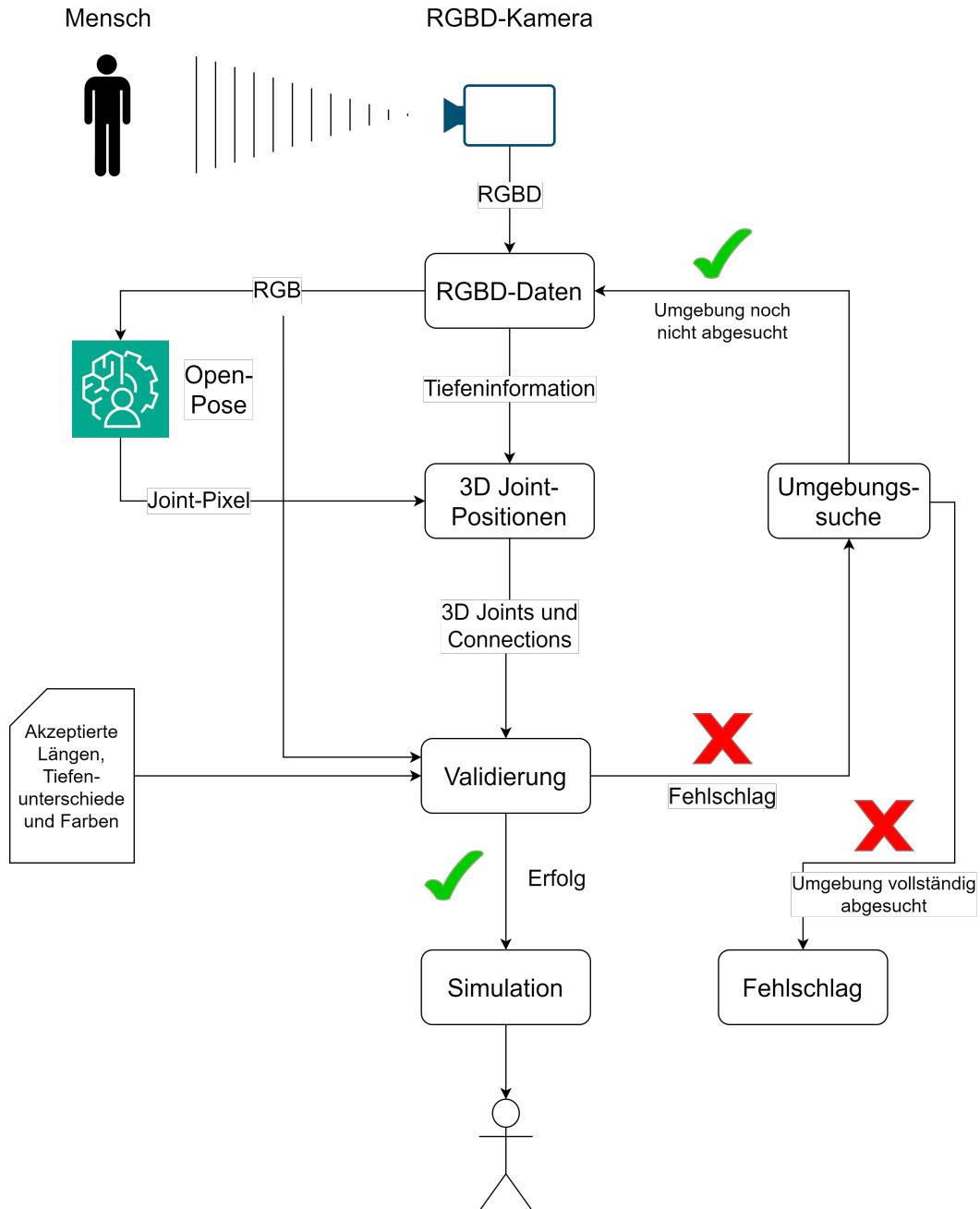


Abbildung 4.1: Visualisiertes Konzept der vorgestellten Pipeline.

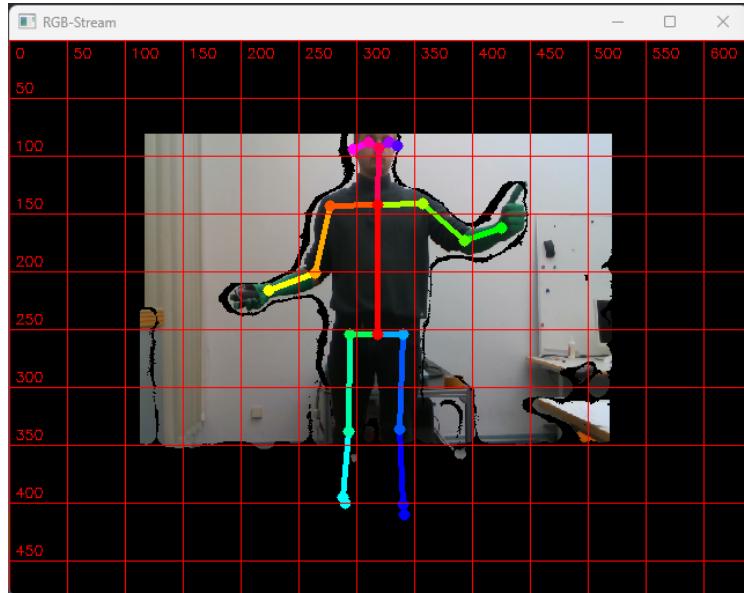


Abbildung 4.2: Erkennung der Pose im Color-Stream. Die Joints (farbige Punkte) werden durch OpenPose erkannt. Die schwarzen Schatten und Balken sind Resultat der Ausrichtung des Color-Streams auf den Depth-Stream. Die grünen Handschuhe tragen zur besseren Validierung der Hände bei.

- Diese 3D-Joint-Positionen können unpräzise oder gar falsch sein, wenn sie beispielsweise durch ein Objekt verdeckt werden. Daher werden sie im nächsten Schritt validiert. Dabei werden die Länge und die Tiefendifferenzen zwischen benachbarten Joints mit definierten Akzeptanzintervallen abgeglichen.
Zusätzlich lassen sich Joints über ihre Farbe validieren: Die Farbe des Pixels des Joints muss innerhalb eines bestimmten Farbbereiches liegen, um den Joint zu validieren. So kann der Benutzer beispielsweise grüne Handschuhe tragen, um der Pipeline die Validierung der Hände zu erleichtern.
Dabei wird die Validierung über die Akzeptanzbereiche für Längen und Tiefendifferenzen umgangen.
- Wurde ein Joint erfolgreich validiert, kann dieser simuliert werden.
- Um das Programm robuster zu gestalten, wird in der Umgebung des Joints nach Tiefenwerten bzw. Farben gesucht, die den Joint validieren.
- Falls in der Umgebung eines invaliden Joints kein solcher Wert gefunden wurde, so wird dieser nicht simuliert.

Das Ergebnis ist in Abb. 4.4 zu sehen.

4 Konzeptentwurf

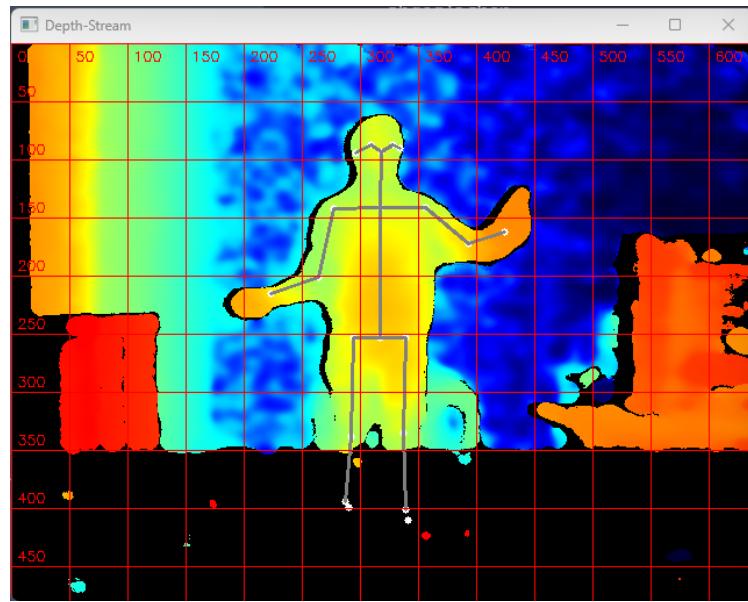


Abbildung 4.3: Visualisierte Tiefenwerte im Depth-Stream. Je näher ein Objekt, desto röter ist es eingefärbt. Je weiter weg, desto blauer. Für schwarze Stellen liegt keine Information vor.

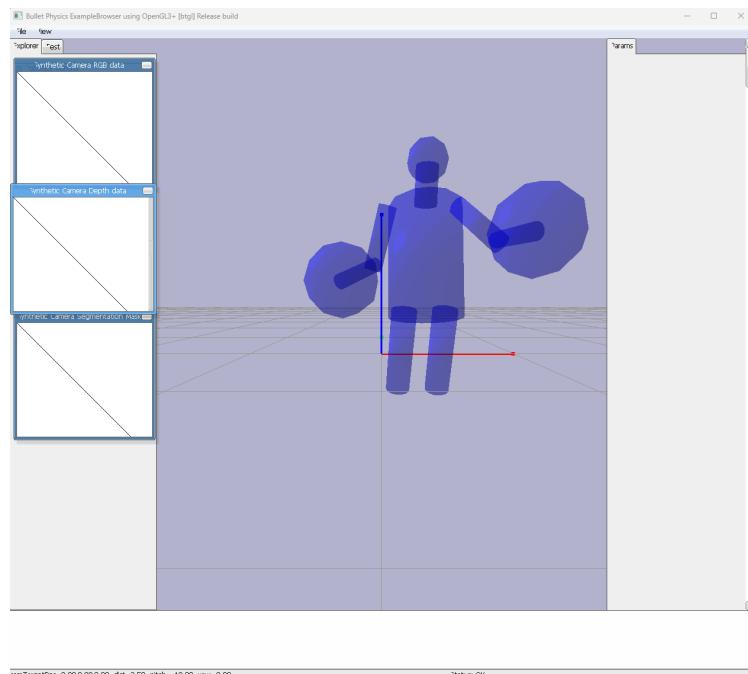


Abbildung 4.4: Resultat der Simulation. Kugeln und Zylinder repräsentieren kombiniert das Männchen.

5 Implementierung

Die eigens implementierte Pipeline wird in GitHub zur Verfügung gestellt [13]. Im Folgenden wird auf die Struktur, Implementierung und Benutzung eingegangen.

5.1 Module

Die Pipeline gliedert sich in mehrere Module und Dateien, die nachfolgend vorgestellt werden.

5.1.1 main.py

Dieses Modul führt den Code wie konfiguriert aus. Es startet entweder das Perceptor-, Simulator- oder Debug-Modul (siehe folgende Abschnitte). Unabhängig davon, ob man den Perceptor oder den Simulator startet, ist mit entsprechender Konfiguration die gesamte Pipeline verfügbar. Da Perceptor und Simulator in unterschiedlichen Prozessen laufen, kann man sich bei der Einbettung in ein anderes Programm durch die Wahl aussuchen, auf welches Objekt man zur Laufzeit einfachen Zugriff haben möchte.

5.1.2 config.py

Dieses Modul lädt die Konfigurationsdatei. Die zu verwendende Konfiguration ist hier anzugeben. Die wichtigsten Einstellungen der Konfiguration werden in Abschnitt 5.3 vorgestellt.

5.1.3 perceptor.py

Hier ist die Hauptlogik implementiert. Die Pipeline erhält einen RGBD-Stream von einer Intel RealSense-Kamera oder aus einer vorher aufgenommenen rohen Playback-Datei. Ein RGBD-Frame besteht aus einem Farbbild und einem Tiefenbild. Es sendet das Farbbild (RGB) an OpenPose und erhält die Pixel der Joint-Positionen. Mithilfe des Tiefenbildes werden die 3D-Koordinaten der Joints extrahiert und wie konfiguriert verschoben und rotiert. Die Joints werden wie in Abschnitt 5.2.1 beschrieben validiert. Invaliden Joints werden auf (0,0,0) gesetzt. Anschließend werden diese der

Simulation übergeben.

Sowohl der rohe RGBD-Stream als auch die mit Zeitstempeln versehenen, extrahierten 3D-Koordinaten der Joints können gespeichert werden. Letzteres dient der Aufnahme der Trainingsdaten.

Vorbereitung

Bevor die Pipeline starten kann, müssen mehrere Dinge vorbereitet werden:

- Falls dies der Hauptprozess ist, wird die Simulation in einem separaten Prozess gestartet. Objekte zum Austausch mit dem Simulatorprozess werden erzeugt:
 - Ein Event, das signalisiert, dass der Prozess die Vorbereitungen abgeschlossen hat und bereit ist.
 - Ein Array zum Austausch der Joint-Positionen.
 - Ein Event zum Signalisieren, dass neue Joint-Positionen vorliegen.
 - Ein Event, das zur Beendigung auffordert.

Anschließend wird gewartet, bis der Simulator bereit ist.

- Der OpenPose-Handler wird initialisiert. Siehe Abschnitt 5.1.5.
- Die Intel RealSense-Kamera wird gestartet. Die intrinsischen Kameraparameter, die wichtig zur Deprojektion eines Pixels und einem Tiefenwert in eine 3D-Koordinate sind, werden gespeichert.
- Der Tiefensor der Kamera wird konfiguriert, siehe Abschnitt 5.3.
- Falls der Perceptor ein Subprozess des Simulators ist, wird anschließend kommuniziert, dass der Perceptor bereit ist.

Aufnahme eines Frames

In der Standartkonfiguration gibt es 30 RGBD-Frames pro Sekunde. Direkt nachdem ein Frame erfasst wurde, wird die aktuelle Zeit gespeichert, die als Zeitstempel für die berechneten 3D-Joint-Positionen dient, falls diese gespeichert werden sollen.

Anschließend werden mehrere Filter (nach Konfiguration) angewendet, siehe Abschnitt 5.3.

Dann wird das Farbbild je nach Kameraausrichtung um 90° , 180° oder 270° gedreht, da OpenPose am besten mit aufrechten Bildern funktioniert.

Folgend wird das Farbbild an OpenPose übergeben, siehe Abschnitt 5.1.5. Der Perceptor bekommt die Jointpixel zurück.

Letztlich werden sowohl die Jointpixel als auch das Farbbild in die Ausgangslage zurückgedreht, um mit der Perspektive des Tiefenbildes übereinzustimmen. Nun kann mit den Tiefen und den intrinsischen Kameraparametern die 3D-Koordinaten der Joints über die pyrealsense2-Funktion `rs2_deproject_pixel_to_point` berechnet werden. Dadurch ist die Länge zwischen zwei Joints messbar, was zur Validierung genutzt wird.

Validierung

Die 3D-Koordinaten der Joints werden, wie in Abschnitt 5.2.1 beschrieben, validiert. Dabei versuchen klassische Algorithmen, fehlerhafte Joint-Koordinaten, die beispielsweise durch Verdeckung entstehen, zu erkennen und zu beheben.

Benutzerdefinierte Algorithmen und Transformation

Nachdem die Joints validiert wurden, werden die in Abschnitt 5.1.6 definierten Algorithmen auf die Joints angewendet. Als Standard ist ein simpler Algorithmus implementiert, der die Tiefe des potenziell unvalidierten Nasen-Joints über die Tiefen der potenziell validierten Augen- und Ohren-Joints abschätzt. Danach werden alle invaliden Joints (mit $z = 0$) auf $(0, 0, 0)$ gesetzt. Anschließend werden die 3D-Koordinaten der Joints je nach Kameraausrichtung aufrecht gedreht.

Abschließend wird die vom Benutzer angegebene Verschiebung (x, y, z) in Metern und Rotation $(\alpha_x, \alpha_y, \alpha_z)$ in Radianen entlang bzw. um die Achsen mit folgender Funktion $f(\mathbf{j})$, wobei $\mathbf{j} \in \mathbb{R}^3$ die Koordinate eines Joints ist, angewendet:

$$f(\mathbf{j}) = \mathbf{R} \cdot \begin{bmatrix} j_x + x \\ j_y + y \\ j_z + z \end{bmatrix}, \text{ mit} \quad (5.1)$$

$$\mathbf{R} = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \quad (5.2)$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha_x) & -\sin(\alpha_x) \\ 0 & \sin(\alpha_x) & \cos(\alpha_x) \end{bmatrix} \begin{bmatrix} \cos(\alpha_y) & 0 & \sin(\alpha_y) \\ 0 & 1 & 0 \\ -\sin(\alpha_y) & 0 & \cos(\alpha_y) \end{bmatrix} \begin{bmatrix} \cos(\alpha_z) & -\sin(\alpha_z) & 0 \\ \sin(\alpha_z) & \cos(\alpha_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.3)$$

Somit wird erst um (x, y, z) verschoben und anschließend um \mathbf{R} rotiert. Die Rotationsmatrix \mathbf{R} wird durch Matrixmultiplikation aus den Rotationsmatrizen um die drei Achsen berechnet. Typischerweise wird so verschoben und rotiert, dass der Roboterfuß im Ursprung ist und eine bestimmte Orientierung hat.

Zu guter Letzt werden die Joints in das mit dem Simulator geteilte Array geschrieben und es wird die Flag gesetzt, dass neue Joints vorliegen. Gegebenenfalls werden die Joint-Positionen mit gesetzten Zeitstempeln gespeichert.

5.1.4 simulator.py

Im Simulator werden einfache geometrische Formen verwendet, um die Position und Pose des Menschen im Raum zu approximieren. In PyBullet wird ein simples, aus Zylindern und Kugeln bestehendes, drei-dimensionales Männchen aus drei-dimensionalen Joint-Koordinaten simuliert. Diese werden entweder vom Perceptor in Echtzeit bereitgestellt, oder aus einer Datei gelesen. Im letzten Fall werden im Echtzeitmodus Frames erst zur mitgespeicherten Zeit simuliert – oder übersprungen, falls die Simulation zu langsam ist. Dieser Ansatz ermöglicht eine flüssige Simulation und sorgt dafür, dass das System nicht auf Verzögerungen wartet. Aus Performancegründen wird die gesamte benötigte Geometrie vorher generiert und im Anschluss nur verschoben und rotiert.

Kugeln repräsentieren in der Simulation Gliedmaßen, die aus einem Joint bestehen, beispielsweise Kopf und Hände. Der Radius ist konfigurierbar und ändert sich während der Ausführung nicht. Die Kugel wird jeweils so platziert, dass der Joint im Mittelpunkt ist.

Zylinder repräsentieren in der Simulation Gliedmaßen, die aus zwei Joints bestehen, beispielsweise der Torso oder die Arme. Auch hier ändert sich die Länge und der Radius während der Ausführung nicht. Der Zylinder wird jeweils so platziert, dass er längs mittig zwischen den beiden Joints liegt.

Kollisionen mit dem Männchen können durch PyBullet automatisch erkannt werden. Den Objekten des Männchens wird eine Kollisionsgruppe (frei wählbar) zugewiesen. Kollisionen zwischen Objekten der gleichen Gruppen werden von PyBullet nicht als solche registriert, nur Kollisionen zwischen Objekten verschiedener Gruppen oder gruppenlosen Objekten werden registriert. Wird diese Pipeline in ein Robotersimulationsprogramm integriert, können Kollisionen zwischen Roboter und Männchen erkannt werden.

In jedem Simulationsschritt werden nur diejenigen Gliedmaßen simuliert, deren Joints *alle* valide sind. Dabei werden diese Gliedmaßen neu positioniert, sichtbar gemacht und die Kollision wird für diese Gliedmaße aktiviert. Alle Gliedmaßen, die *mindestens einen* invaliden Joint haben, werden unsichtbar gemacht und die Kollision wird für sie deaktiviert.

Der Simulator kann in andere Programme integriert werden. Aktuelle (zur Zeit simulierte) 3D-Koordinaten der Joints des Männchens sind in `simulator_object.joints` gespeichert. Aktuelle valide (Kollisionserkennung aktiviert und sichtbar) bzw. invalide (Kollisionserkennung deaktiviert und unsichtbar) Gliedmaßen sind in `simulator_object.limb_list_valid`

respektive `simulator_object.limb_list_invalid` zu finden.

Die Kugeln und Zylinder müssen mit jedem neuen Frame neu im Raum positioniert werden. Dafür werden sie verschoben und gedreht. Man kann entweder innerhalb der Physiksimulation verschieben und rotieren, oder außerhalb. Im ersten Fall erhält ein Objekt eine lineare und eine Winkelgeschwindigkeit, die so gesetzt sind, dass das Objekt innerhalb eines Simulationsschritts an die gewünschte Position „fliegt“. Dabei können Kollisionen mit anderen Objekten festgestellt werden. Würden Objekte über größere Distanzen fliegen, so könnte die Simulationspräzision leiden, da Kollisionen, die so in Wirklichkeit nicht unbedingt stattfanden, registriert werden könnten. Das ist insbesondere dann der Fall, wenn eine Gliedmaße über eine längere Zeit nicht erkannt und damit nicht simuliert wurde. Dann verbleibt sie (unsichtbar) an derselben Position. Wird diese auf einmal erkannt, und der Mensch hat sich bereits über eine größere Distanz bewegt, fliegt diese Gliedmaße durch den gesamten Raum und kollidiert ggf. fälschlicherweise. Daher werden Objekte über größere Distanzen (konfigurierbar) außerhalb der Physiksimulation bewegt. Ein Objekt wird dabei einfach an eine neue Position mit neuer Orientierung teleportiert, Kollisionen werden nicht erkannt.

Positionierung außerhalb der Physiksimulation

Die Positionierung außerhalb der Physiksimulation bezieht sich auf den PyBullet-Befehl `resetBasePositionAndOrientation`. Dabei wird das Objekt an die gewünschte Position „teleportiert“. Es vergeht keine simulierte Zeit, allerdings ist keine Kollisionserkennung möglich. Daher eignet sich dies für größere Distanzen.

Der Befehl `resetBasePositionAndOrientation` erwartet einen Mittelpunkt und eine Orientierungsquaternion. Letzteres lässt sich mit der PyBullet-Funktion `getQuaternionFromAxisAngle` aus der Rotationsachse und dem Rotationswinkel berechnen.

Für eine Kugel haben wir einen Joint gegeben $j := [x, y, z]$ gegeben. Offensichtlich ist, dass der Mittelpunkt $m = j$ und die Orientierung bei einer Kugel irrelevant ist, sodass diese auf 0 gesetzt werden kann.

Für einen Zylinder haben wir zwei Joints $j_1 := [x_1, y_1, z_1]$ und $j_2 := [x_2, y_2, z_2]$ gegeben. Der Mittelpunkt ist:

$$m = \frac{j_1 + j_2}{2} \quad (5.4)$$

Um die Rotationsachse und den Rotationswinkel zu bestimmen, müssen wir verstehen, dass PyBullet bei der gewählten Methode eine Orientierungsqua-

ternion erwartet, die die Rotation von der Standardorientierung $d_0 := [0, 0, 1]$ zur gewünschten Ausrichtung beschreibt.

Die Rotationsachse wird daher als Kreuzprodukt zwischen der Standardorientierung d_0 und der gewünschten Orientierung

$$d = j_2 - j_1 \quad (5.5)$$

berechnet. Das Kreuzprodukt erzeugt einen zu beiden Vektoren senkrechten Vektor. Eine Rotation um diese Achse kippt den Zylinder von der einen Richtung in die andere:

$$r = d_0 \times d \quad (5.6)$$

Aus der Definition des Skalarprodukts $a \cdot b = ||a|| \cdot ||b|| \cdot \cos(\phi)$, wobei a, b Vektoren sind und ϕ der Winkel zwischen a und b ist, lässt sich der Rotationswinkel berechnen:

$$\theta = \arccos\left(\frac{d_0 \cdot d}{||d_0|| \cdot ||d||}\right) \quad (5.7)$$

Aus der Rotationsachse r und dem Rotationswinkel θ berechnet PyBullet die Orientierungsquaternion q , woraus es wiederum zusammen mit dem Mittelpunkt m die neue Position des Zylinders errechnet.

Positionierung innerhalb der Physiksimulation

Für die Kollisionserkennung ist es erforderlich, dass Objekte innerhalb der Physiksimulation bewegt werden. Dabei vergeht Simulationszeit, den Objekte werden aktiv im Raum bewegt und nicht nur „teleportiert“, daher eignet sich dies für kürzere Distanzen.

Die Positionierung innerhalb der Physiksimulation funktioniert über den PyBullet-Befehl `resetBaseVelocity`, der eine lineare Geschwindigkeit und eine Winkelgeschwindigkeit erwartet. Neben dem bereits berechneten Mittelpunkt m und der Orientierungsquaternion q benötigen wir ein vorher festgelegtes Zeitdelta Δt . Diese Zeitdifferenz entspricht der Dauer eines Simulationsschritts und bestimmt, wie schnell sich das Objekt in der realen Zeit bewegen wird. Wir wollen das Objekt innerhalb eines Simulationsschritts an die gewünschte Position befördern.

Um die lineare Geschwindigkeit zu berechnen, berechnen wir erst die Differenz zwischen der gewünschten Position m und der aktuellen Position m_0 . Dies gibt die Richtung an, in die sich das Objekt bewegen muss. Die benötigte lineare Geschwindigkeit, um das Objekt innerhalb eines Simulationsschritts ans Ziel zu bringen, erhalten wir, indem wir mit Δt dividieren:

$$v = \frac{m - m_0}{\Delta t} \quad (5.8)$$

Um die Winkelgeschwindigkeit zu berechnen, berechnen wir erst die Differenz zwischen der gewünschten Orientierungsquaternion q und der derzeitigen Orientierungsquaternion q_0 . Dies geht mithilfe der PyBullet-Funktion `getDifferenceQuaternion`.

Anschließend konvertiert die PyBullet-Funktion `getAxisAngleFromQuaternion` den Quaternion-Unterschied in eine Rotationsachse r' und einen Rotationswinkel θ' . Diese beiden Werte beschreiben, um welche Achse und um welchen Winkel sich das Objekt drehen muss, um von seiner aktuellen Orientierung zur Zielorientierung zu gelangen. Dabei ist r' bereits *normalisiert*, d.h. es gilt $\|r'\| = 1$. Dies ist wichtig, um ausschließlich die Rotation, nicht aber die Skalierung zu beeinflussen.

Die Winkelgeschwindigkeit, die das Objekt innerhalb eines Simulations schritts korrekt rotiert, erhalten wir, indem wir die Rotationsachse mit dem Rotationswinkel multiplizieren und mit Δt dividieren:

$$\omega = \frac{\theta' \cdot r'}{\Delta t} \quad (5.9)$$

Nach einem Simulationsschritt wird die lineare und die Winkelgeschwindigkeit auf 0 gesetzt, damit etwaige zusätzliche Simulationsschritte die Objekte nicht weiter verschieben.

5.1.5 openpose_handler.py

OpenPose ist ein neuronales Netz, dass definierte Joint-Position an einem Menschen erkennen kann.

Der OpenPose-Handler startet bei Initialisierung den Python-Wrapper für OpenPose und konfiguriert diesen so, dass dieser auf Bildern (.png, .jpg) arbeitet und maximal eine Person erkennt. Die Anwendung dieses neuronalen Netzes ist für bereits niedrig-aufgelöste Bilder sehr rechenintensiv für die Grafikkarte.

Das erhaltene Bild muss erst im PNG-Format codiert werden, da OpenPose nicht mit unkomprimierten Bildern arbeiten kann. Anschließend verarbeitet OpenPose das Bild und gibt die Pixel der Joints zurück.

5.1.6 custom_3d_joint_transforms.py

Hier können Funktionen definiert werden, die nach dem Validierungsschritt auf alle Joints angewendet werden. Ein simpler Algorithmus ist als Standard vorgegeben, dieser wird in Abschnitt 5.1.3 erklärt.

5.1.7 debug.py

Mehrere Debug-Tools sind implementiert, die vor allem die Kalibrierung für die Validierung der Joints erleichtern, siehe Abschnitt 5.2.2.

5.2 Validierung

5.2.1 Algorithmus

Eine der größten Herausforderungen stellt die Validierung von Joints dar, den die 3D-Joint-Positionen können unpräzise oder falsch sein. Das kann mehrere Gründe haben:

- Zum einen kann die Kamera an sich die Tiefe falsch oder gar nicht messen.
- Zum anderen könnten manche Joints durch andere Objekte (bspw. durch den Roboter) verdeckt sein. Ist die Verdeckung nicht groß genug, so liefert OpenPose eine Schätzung des Pixels des jeweiligen Joints zurück. Das hat zur Folge, dass die berechnete 3D-Position die des verdeckenden Objekts ist und nicht die des Joints.

Vor allem die Verdeckung sorgt ohne Nachbesserung für sehr unpräzise und stark variierende Joint-Positionen. Daher müssen Methoden implementiert werden, die vermeintlich falsche oder unrealistische Joint-Positionen filtern. Gleichzeitig wäre eine robuste Pipeline wünschenswert, die trotz initial falscher Joint-Positionen vernünftige Werte liefert. Daher werden die Joint-Positionen über definierte Verbindungen, die je aus zwei Joints bestehen und deren Länge bekannt ist, validiert. Eine Verbindung ist beispielsweise Ellbogen-Handgelenk oder Nacken-Hüfte. Dazu wird, wie im Folgenden beschrieben, die Länge zwischen zwei Joints mit der typischen Länge dieses Körperabschnitts abgeglichen.

Die Idee:

Wenn ein Objekt eine Verbindung verdeckt, gibt es zwei Möglichkeiten: Entweder, das Objekt verdeckt nur einen Joint der Verbindung, e.g. den Ellbogen. Dann wird die Hand korrekt deprojiziert, der Ellbogen jedoch nicht: Die Deprojektion des Ellbogens befindet sich nun näher an der Kamera, dort, wo die Linie Kamera-Ellbogen das Objekt schneidet. Jetzt ist es wahrscheinlich, dass die Länge zwischen den Deprojektionen der Hand und des Ellbogens höher als üblich ist. Siehe Abb. 5.1.

Oder, das Objekt verdeckt beide Joints der Verbindung. Dann werden beide falsch deprojiziert und beide befinden sich näher an der Kamera. Beide Deprojektionen befinden sich dort, wo die Linien zwischen der Kamera und den respektiven Joints das Objekt schneiden. Das heißt wiederum, dass die

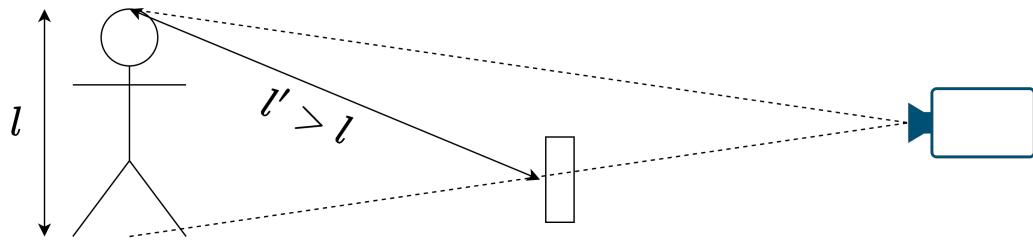


Abbildung 5.1: Idee der Längenvalidierung: Ein Joint ist verdeckt. Dann ist es wahrscheinlich, dass die gemessene Länge l' größer ist, als die wahre Länge l . Das Männchen steht repräsentativ für eine Gliedmaße.

Länge zwischen den Deprojektionen wahrscheinlich kürzer ist, als sie sein sollte. Siehe Abb. 5.2.

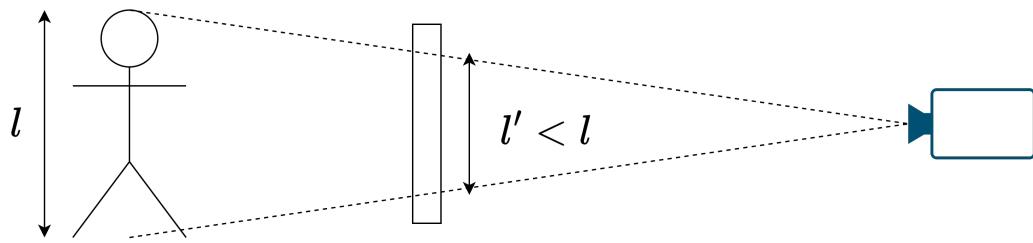


Abbildung 5.2: Idee der Längenvalidierung: Zwei Joints sind verdeckt. Dann ist es wahrscheinlich, dass die gemessene Länge l' kürzer ist, als die wahre Länge l . Das Männchen steht repräsentativ für eine Gliedmaße.

Zusätzlich ist es bei einigen Verbindungen sinnvoll, die alleinige Differenz der Tiefe (z -Komponente) ebenfalls zu berücksichtigen. Beispielsweise können wir davon ausgehen, dass der Torso mehr oder weniger senkrecht bleibt. Wird ein Joint des Torsos verdeckt, so erhöht sich vor allem die Differenz der Tiefenkomponenten der jeweiligen Deprojektionen.

In den Abb. 5.3 und 5.4 sieht man diese Idee in Aktion. Wird nichts verdeckt (Abb. 5.3), so sind die Längen und Tiefen der Verbindungen im Akzeptanzbereich. Wird ein Joint verdeckt (Abb. 5.4), so wird dies erkannt, da Länge und/oder Tiefe der anliegenden Verbindungen nicht mehr im Akzeptanzbereich liegen.

Letztlich, gibt es Joints, die kritisch oder fehleranfällig bei den vorher genannten Validierungen sind. So sind die Hände sowohl besonders signifikant, als auch fehleranfällig: Die einzige sinnvolle Verbindung zur Validierung der Hand über eine Länge ist der Ellbogen. Wird der Ellbogen nicht erkannt, kann die Hand nicht validiert werden.

Daher kann alternativ die Farbe der Pixel der jeweiligen Joints mit einem Farbbereich abgeglichen werden. So könnte der Benutzer grüne Handschuhe tragen, um die Validierung der Hände zu erleichtern, wenn im Sichtfeld der Kamera sonst nichts Grünes ist. Ist der Pixel des Joints des Handgelenks grün, so wird angenommen, dass es sich dabei um die Hand handelt, siehe Abb. 5.6.

Der Pseudo-Algorithmus ist auf einer der nächsten Seiten dargestellt. Zusammengefasst:

- Eine Verbindung besteht aus zwei Joints: $c := (j_1, j_2)$
- Jede Verbindung hat einen definierten, akzeptierten Längenbereich $[l_{min}, l_{max}]$.
Die Länge einer Verbindung ist *valide*, wenn die Länge im Längenbereich liegt:

$$l_{min} \leq \|j_2 - j_1\| \leq l_{max} \quad (5.10)$$

- Jede Verbindung hat eine definierte, akzeptierte maximale Tiefendifferenz z_{max} .

Die Tiefenabweichung einer Verbindung ist *valide*, wenn die Differenz der Tiefe unter dem angegebenen Wert liegt:

$$|j_2[z] - j_1[z]| \leq z_{max} \quad (5.11)$$

- Die Verbindung ist *valide*, wenn sowohl die Länge als auch die Tiefendifferenz *valide* ist.
- Die Farbe f des Pixels für in der Konfiguration spezifizierte Joints kann mit einem Farbbereich $[f_{min}, f_{max}]$ abgeglichen werden.

Die Farbe ist *valide*, wenn die Farbe im Farbbereich liegt:

$$\bigwedge_{i \in \{r,g,b\}} f_{min_i} \leq f_i \leq f_{max_i} \quad (5.12)$$

- Ein Joint ist *valide*, wenn *mindestens eine* seiner Verbindungen valide ist oder wenn seine Farbe valide ist.

Für invalide Joints wird in der näheren Umgebung des Joints nach Pixeln gesucht, deren Tiefen oder Farben den Joint validieren können, um das Programm robuster zu machen. Dabei wird die 3D-Koordinate aus den echten (x, y) -Koordinaten des Pixels des Joints und der neuen Tiefe z' berechnet. Falls kein z' aus der Umgebung zur Validierung führt, so wird die z -Koordinate des Joints 0 gesetzt und der Joint wird nicht simuliert.

Betrachten wir zum Verständnis Abb. 5.5. Wir sehen, dass der Hüft-Joint (in der Mitte des Vierecks) durch ein Objekt verdeckt ist, aber von OpenPose erkannt wird. Nun wird dieser über anliegende Verbindungen validiert, was jedoch scheitert, da die Abstände zu kurz oder zu lang sind, weil durch die Verdeckung der Hüft-Joint weiter vorne erscheint. Die roten Pixel stellen den Suchbereich dar. Für jedes Pixel des Suchbereichs wird die Tiefe z' ausgelesen. Jetzt wird angenommen, dass sich der Hüft-Joint an der 3D-Koordinate befindet, die sich aus Kombination der originalen 2D-Pixel-Koordinate des Joints (da wo dieser wirklich ist) und der neuen Tiefe z' ergibt. Dieser Joint wird nun mit seiner neuen Koordinate über die anliegenden Verbindungen validiert. Im Beispiel wäre dies für eine Tiefe der oberen roten Pixel erfolgreich, da dort die Objektverdeckung endet. Dadurch würde die Länge der Verbindung Nacken-Hüfte stimmen.

Ähnlich wird eine passende Farbe gesucht, falls diese Option für diesen Joint aktiviert ist (üblicherweise für die Hände). Hat einer der rot eingefärbten Pixel (im Original) die korrekte Farbe, so bekommt der Joint die 3D-Koordinate aus der originalen Pixel-Koordinate und der entsprechenden neuen Tiefe z' .

Falls ein Joint in diesem Prozess durch die Länge und Tiefendifferenz validiert wurde, so änderte sich seine Position im Raum. Das könnte die Validierung von anderen Joints ermöglichen, sodass dieser Prozess im Anschluss für alle invaliden Joints wiederholt wird, bis kein Joint innerhalb einer Iteration validiert wurde.

Das könnte theoretisch zu Verzögerungen führen, wenn pro Iteration nur wenige Joints validiert werden, diese aber die Validierung weniger anderer Joints in der nächsten Iteration ermöglichen. In der Praxis begrenzte sich die Anzahl der Iterationen auf 3.

Algorithm 1 Validation Algorithm

```

Require: joints_2d           ▷ 2D joint coordinates from OpenPose
Require: depths             ▷ depth information for each pixel

joints_3d ← INITIALIZE(0)
for all i ∈ joints do
    joints_3d[i] ← GET_3D_COORD(joints_2d[i], depths[joints_2d[i]])
        ▷ save the unvalidated 3D coordinates for each joint

val ← INITIALIZE(false)          ▷ saves which joints are validated
for all (i, j) ∈ connections do
    if ISVALID(joints_3d[i], joints_3d[j]) then
        val[i] ← val[j] ← True
            ▷ validate each joint on each of its connections

for all i ∈ color_joints do
    if ¬val[i] then
        x, y ← joints_2d[i]
        for all xs, ys ∈ SEARCHAREA(x, y) do
            if ISCOLORVALID(color[xs, ys]) then
                joints_3d[i] ← GET_3D_COORD(x, y, depths[xs, ys])) ▷ use old
x, y but depth at xs, ys
        val[i] ← True
    ▷ for the joints which can be validated by color, search in their search
area to find the right color

change ← True
while change do
    change ← False
    for all i ∈ joints do
        if ¬val[i] then
            x, y ← joints_2d[i]
            for all xs, ys ∈ SEARCHAREA(x, y) do
                joints_3d[i] ← GET_3D_COORD(x, y, depths[xs, ys]))
                for all j which i has a connection with do
                    if ISVALID(joints_3d[i], joints_3d[j]) then
                        val[i] ← val[j] ← True
                        change ← True
    ▷ search around the joints to find a depth to validate them, perform on
unvalidated joints until no change

for all i ∈ joints do
    if ¬val[i] then
        joints_3d[i] ← 0
            ▷ set all unvalidated joints to 0

```

5 Implementierung

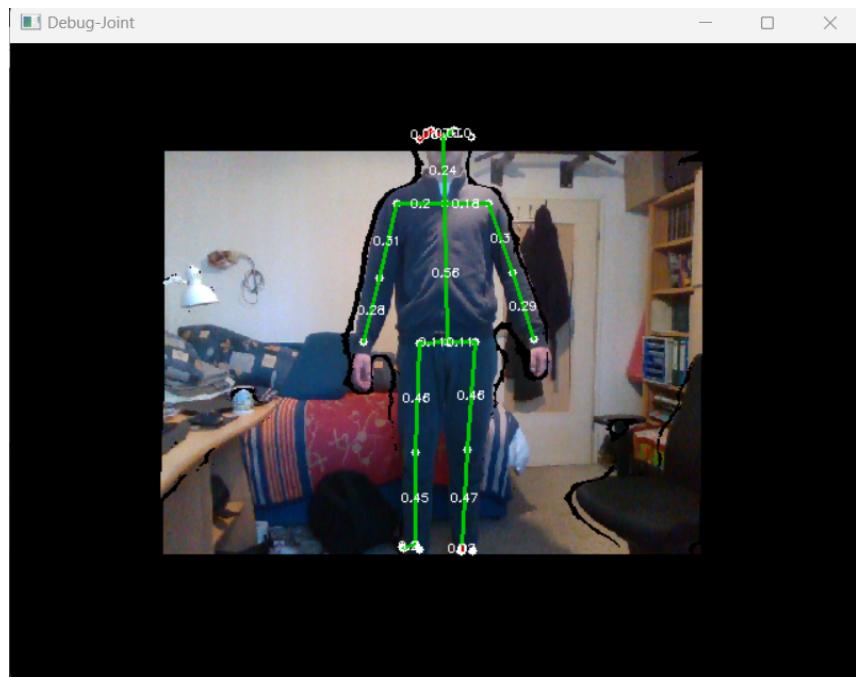


Abbildung 5.3: Verbindungen werden optimalerweise unter normalen Verhältnissen alle validiert (grün).

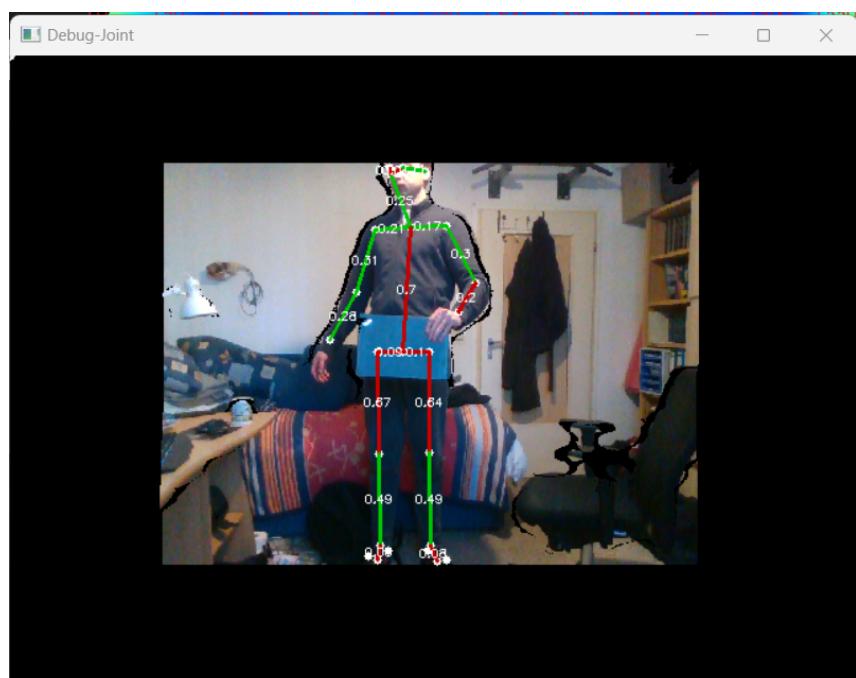


Abbildung 5.4: Die Hüfte wird durch ein Objekt verdeckt. Ohne Suchbereich werden anliegende Verbindungen nicht validiert (rot). Der Unterarm wird nicht validiert, da dieser unter diesem Winkel als zu kurz erscheint.

5 Implementierung

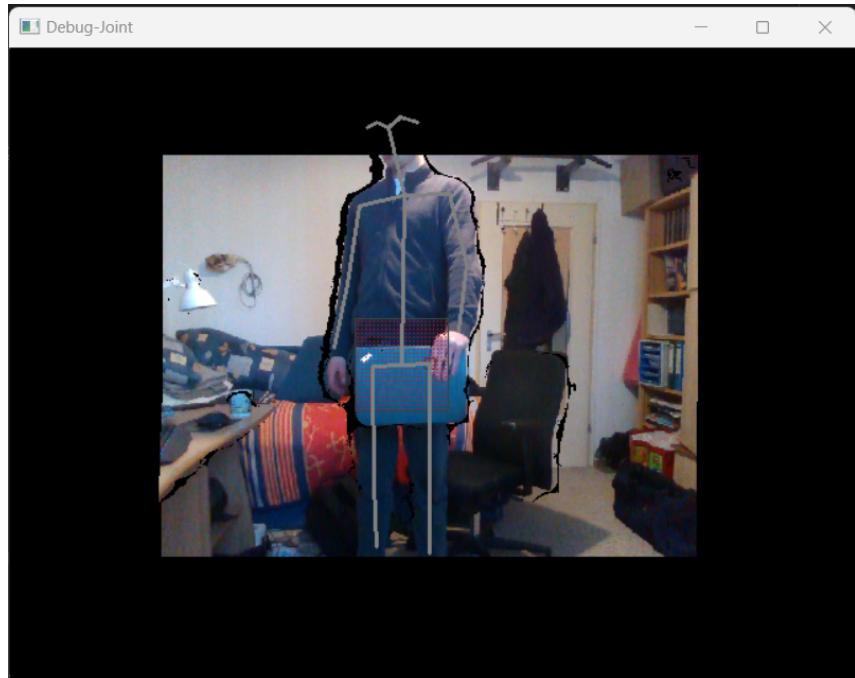


Abbildung 5.5: Die roten Pixel stellen den Suchbereich für den Hüftjoint dar, die genutzt werden, falls der Hüftjoint nicht ohne Weiteres validiert werden kann.

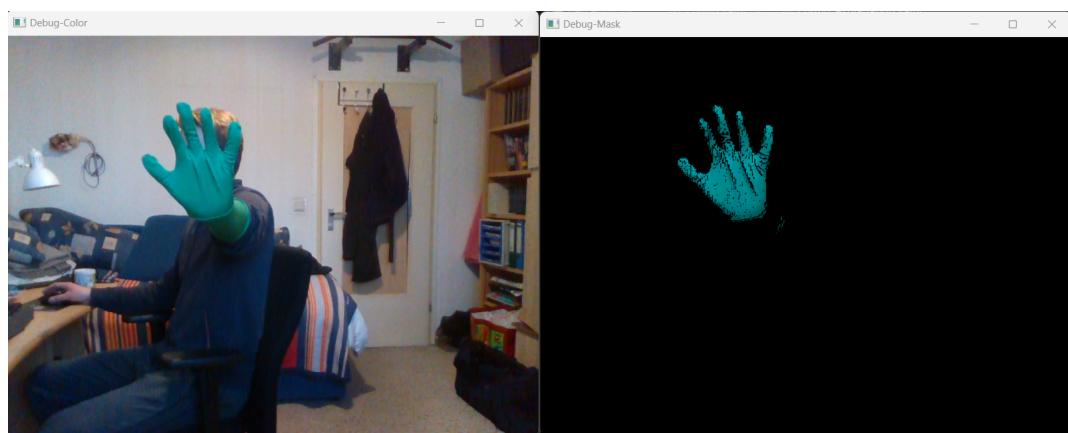


Abbildung 5.6: Da die Hände nur über die Verbindung Hand-Ellbogen validiert werden können, sind sie besonders anfällig. Daher kann der Nutzer z.B. grüne Handschuhe tragen, sodass klar ist, wo die Hand im Bild ist.

5.2.2 Kalibrierung

Damit die Validierung möglichst gut funktioniert, müssen verschiedene Parameter möglichst präzise kalibriert werden. Dazu werden im Debug-Modus Hilfen bereitgestellt. Im Folgenden wird die Herangehensweise an die Kalibrierung erläutert.

Farbbereich

Ein Modus hilft bei der Setzung des richtigen Farbbereiches. Dargestellt wird ein Stream, bei dem nur Pixel innerhalb des Farbbereiches dargestellt werden und der Rest schwarz ist. Über die Regler lässt sich der Farbbereich einstellen. Er sollte so eingestellt werden, dass die gewünschte Farbe klar zu erkennen ist, und alles andere schwarz ist. Im Zweifel sollte man lieber zu viel schwärzen als zu wenig. Man sollte ebenfalls beachten, dass die Lichtverhältnisse (e.g. Tageszeit, Bewölkung, Position und Ausrichtung der Lampen) und der Winkel des gefärbten Objekts (e.g. Hand im grünen Handschuh) starke Einflüsse auf die Farbe des Objekts für einen Computer haben.

Suchbereich

Ein anderer Modus hilft bei der Kalibrierung der Suchbereiche für den in Abschnitt 5.2.1 beschriebenen Validierungsprozess. Dabei wird die Größe und Dichte des Bereiches für jeden einzelnen Joint über Regler eingestellt. Dargestellt wird der Color-Stream mit den entsprechenden Suchbereichen um die Joints herum. Die Pixel des Suchbereiches werden rot eingefärbt. Die Suchbereiche sollten ungefähr die Größe der Joints haben, beispielsweise sollte dieser für den Torso größer sein als für den Ellbogen. Je wichtiger der Joint (z.B. Hand), desto dichter sollte der Suchbereich sein. Viele große, dichte Suchbereiche könnten zu Verzögerungen führen, daher ist es wichtig, eine Balance zu finden.

Längen- und Tiefenbereich

Ein weiterer Modus hilft bei der Bestimmung der akzeptierten Längen- bzw. Tiefenbereiche, die sich bei verschiedenen Personen unterscheiden können. Im Folgenden wird der Prozess für die Länge erklärt, für die Tiefe ist es analog.

Dargestellt werden die Verbindungen mit Längenangaben. Folgendes ist eine vernünftige Vorgehensweise zur Bestimmung eines sinnvollen Akzeptanzbereiches. Das Ziel ist es, diesen so zu bestimmen, dass Längen bei normalen Bewegungen akzeptiert werden, aber bei Verdeckung durch Objekte abgelehnt werden.

- Im ersten Schritt sollte man normale Bewegungen ausführen, ohne dass man von Objekten verdeckt wird. Im Anschluss werden statistische Werte, wie Durchschnitt, Median, Varianz und Dezile ausgegeben, die eine Hilfe sind.
- Jede Verbindung sollte einzeln betrachtet werden. Dabei sind normale Bewegung in verschiedener Distanz zur Kamera auszuführen und die Werte zu beobachten. Dabei sollten die Joints für die Kamera sichtbar und nicht zu stark angewinkelt sein. Man kann nun den Längenbereich so setzen, dass sie knapp akzeptiert werden.
- Im Anschluss verdeckt man erst einen, dann beide Joints mit einem Objekt und bewegt dieses näher an die Kamera. Den niedrigeren/höheren Wert setzt man so, dass die Ungenauigkeit akzeptabel ist.

Man sollte beachten, dass dies nur eine generelle Vorgehensweise ist. Die „richtige“ Vorgehensweise kann sich von Situation zu Situation unterscheiden.

5.3 Konfiguration

Hier werden die wichtigsten Einstellungen aufgelistet und erklärt. Für eine vollständige Liste siehe `config_explanation.yaml` und `README` in [13].

Der Parameter `run_from` entscheidet, ob entweder Perceptor oder Simulator als Hauptprozess gestartet wird. Je nach weiterer Konfiguration ist die gesamte Pipeline verfügbar. Wenn die Pipeline in ein anderes Programm integriert ist und aus diesem heraus gestartet wird, steht nur der Hauptprozess als Objekt zur Verfügung, das andere ist ein Subprozess, dessen Variablen und Funktionen nicht ohne weiteres abgerufen werden können.

Tabelle 5.1: Konfiguration Perceptor

Parameter	Erklärung
<code>resolution & fps</code>	Mögliche Werte sind entweder in der Intel RealSense-Dokumentation oder dem Intel RealSense Viewer einsehbar. Eine niedrige Auflösung sollte gewählt werden, da sie ausreicht und OpenPose die Pipeline sonst zu stark verlangsamt.
<code>flip</code>	Bestimmt, wie der Stream intern in 90°-Schritten gedreht wird. Sollte so gewählt werden, dass die angezeigten Streams aufrecht sind, da sonst OpenPose nicht optimal funktioniert.

Fortsetzung der Tabelle 5.1

Parameter	Erklärung
translation & rotation	Verschiebt und rotiert die berechneten 3D-Joints am Ende, bevor sie an den Simulator weitergereicht werden, sodass die Joints in der gewünschten Perspektive sind, siehe Abschnitt 5.1.3.
visual_preset	Gibt an, ob das Tiefenbild weniger, aber dafür genauere Daten, oder mehr, aber dafür weniger genaue Daten haben soll [8]. Die Präzision der ungenauen Variante ist prinzipiell ausreichend. Jedoch kommt es dann am Rand von verdeckenden Objekten zu unpräzisen Tiefenwerten. Wird mit Farbe validiert (siehe Abschnitt 5.2.1), so könnte das größere Fehler begünstigen.
exposure	Die Belichtungszeit bestimmt, wie lange der Kamerasensor Licht sammelt, bevor ein Bild aufgenommen wird. Längere Belichtungszeit führt zu klareren Aufnahmen in dunkleren Umgebungen, erhöht aber die Bewegungsunschärfe. Dies ist eine sehr <i>kritische</i> Einstellung für den Tiefensensor und sollte bei neuen Lichtverhältnissen neu kalibriert werden [8].
gain	Die Verstärkung ist ein Maß für die Verstärkung des Signals und erhöht Helligkeit und Bildrauschen. Intel empfiehlt diesen zugunsten der Bildqualität bei der niedrigsten Einstellung zu belassen [8].
laser_power	Eine höhere Laserleistung erhöht meist die Qualität [8].
depth_units	Die Tiefeneinheiten definieren die kleinste messbare Distanzänderung. Je kleiner die Tiefeneinheit, desto kleiner die messbare Maximaldistanz [8].
alignment	Da Color- und Depth-Streams von verschiedenen Sensoren aufgenommen werden, sind sie nicht aufeinander abgestimmt. Das heißt, dass die Tiefe für Pixel (x, y) des Color-Streams nicht an der Position (x, y) des Depth-Streams sein muss. Daher müssen beide Streams aufeinander ausgerichtet werden. Wenn der Depth-Stream auf den Color-Stream ausgerichtet wird, kann es bei manchen Kameras zu deutlich falschen Tiefenwerten näher zum Bildrand hin führen. Alternativ lässt sich der Color-Stream auf den Depth-Stream ausrichten, was nur zu unschönen Visualisierung führt (siehe Abb. 4.2).
depth2disparity	Wandelt Tiefenwerte in Disparitätswerte um und verbessert damit die nächsten zwei Filter. Sollte nur zusammen mit disparity2depth (de)aktiviert werden [8].

Fortsetzung der Tabelle 5.1

Parameter	Erklärung
spatial_filter	Räumlicher Rauschfilter, der Bildrauschen reduziert und gleichzeitig Objektkanten erhält [8].
temporal_filter	Verbessert Beständigkeit der Tiefendaten, in dem Werte basierend auf vergangenen Werten manipuliert werden. Weniger geeignet für sehr dynamische Szenen [8].
disparity2depth	Wandelt Disparitätswerte zurück in Tiefenwerte um. Sollte nur zusammen mit depth2disparity (de)aktiviert werden, damit die Pipeline Tiefenwerte erhält [8].
hole_filling-filter	Die Hole-Filling-Filter versuchen mit verschiedenen Methoden, Lücken im Depth-Frame zu füllen. Diese sollten insbesondere dann deaktiviert werden, wenn mit Farbe validiert wird (siehe Abschnitt 5.2.1), da sonst Ungenauigkeiten entstehen könnten [8].
connections_hr	Verbindungen, die zur Validierung mit Längen und Tiefendifferenzen verwendet werden.
color_validation_joints_hr	Joints, die mit Farbe validiert werden.
color_range	Der in der Validierung genutzte Farbbereich.
lengths_hr	Die in der Validierung genutzten Längsbereiche.
depth_deviations_hr	Die in der Validierung genutzten maximalen Tiefendifferenzen.
search_areas_hr	Die in der Validierung genutzten Suchbereiche.
joint_3d_transforms	Namen der Funktionen, die nach der Validierung auf die Joints angewendet werden sollen, siehe Abschnitt 5.1.6.

Tabelle 5.2: Konfiguration Simulator

Parameter	Erklärung
start_pybullet	Ob der Simulator PyBullet starten soll. Sollte deaktiviert werden, wenn es in ein anderes Programm eingebettet wird, das PyBullet startet.
build_geometry	Ob der Simulator die Geometrie (Gliedmaßen) generieren soll. Falls deaktiviert, sollte der einbettende Prozess diese generieren, wie in der README [13] beschrieben.
playback & playback_mode	Ob der Simulator Joints aus einer Datei lesen soll, anstelle diese vom Perceptor zu erhalten. Die Modi „Normal“ (Frame-by-Frame), „Real-Time“ und „Step-by-Step“ stehen zur Verfügung.
move_in_physic_sim	Aktiviert die Kollisionserkennung, wie in Abschnitt 5.1.4 erklärt.

Fortsetzung der Tabelle 5.2

Parameter	Erklärung
distance_to_move_outside_physic_sim	Minimale Distanz, um Objekte ohne Kollisionserkennung zu bewegen. Die Simulationspräzision und -geschwindigkeit verschlechtern sich bei zu hohen Werten. Die Kollisionserkennung verschlechtert sich bei zu niedrigen Werten. Siehe Abschnitt 5.1.4. Guter Wert in der Praxis: 0.05.
time_delta_move_in_physic_sim	Beeinflusst die Geschwindigkeit bei Bewegung mit Kollisionserkennung, siehe Abschnitt 5.1.4. Kleinere Werte bewegen Objekte schneller und verbessern die Simulationspräzision, aber verschlechtern die Kollisionserkennung. Guter Wert in der Praxis: 0.01.
limbs	Zu simulierende Gliedmaßen. Gliedmaßen, die aus einem Joint bestehen, werden durch eine Kugel simuliert. Gliedmaßen, die aus zwei Joints bestehen, werden durch einen Zylinder simuliert.
radii	Die Radien für die Gliedmaßen (Kugel und Zylinder).
lengths	Die Längen für die Gliedmaßen (nur Zylinder).

5.4 Integration in IR-DRL

Die Integration in die existierende IR-DRL-Suite [9] wird beispielhaft in [10] implementiert. Dabei gestaltet sich diese recht einfach:

- Die Geometrie kann manuell mit den Tools von IR-DRL generiert werden oder automatisch durch den Simulator. Beide bedürfen kleiner Anpassungen der Dictionaries im Simulator oder IR-DRL, die PyBullet-Objektverweise für das jeweilige Programm speichern.
- Falls der Simulator in Echtzeit mit dem Perceptor läuft, so lassen sich die Positionen des Männchens mit `simulator.process_frame_sync()` mit den jüngsten Informationen updaten.
- Falls der Simulator die Joint-Positionen aus einer Datei liest, so lässt sich die Position des Männchens mit `simulator.process_frame_at_time()` updaten. Die Funktion erfordert die Übergabe eines Zeitstempels, was hier der simulierten Zeit entsprechen wird. Die Positionen mit dem nächsthöheren Zeitstempel werden geladen.
- Die aktuellen Joint-Positionen sind stets unter `simulator.joints` verfügbar.

Wie man sieht, gestaltet sich die Integration der Pipeline in die IR-DRL-Suite ohne viel Aufwand.

6 Evaluation

6.1 Theoretische Limitierungen

In folgenden Abschnitten betrachten wir mögliche Limitierungen der Pipeline bezüglich der Korrektheit und/oder Robustheit.

6.1.1 OpenPose

OpenPose funktioniert größtenteils sehr zuverlässig, jedoch werden teilweise andere Objekte, z.B. Rucksäcke oder Jacken, fälschlicherweise als Menschen erkannt. Diese lassen sich jedoch relativ schnell identifizieren und aus dem Sichtfeld der Kamera entfernen.

In gewissen Konstellationen kann OpenPose fälschlicherweise den Manipulator als menschlichen Arm erkennen. Diese Fehler wurden in Tests durch die Validierung erkannt und entfernt.

6.1.2 Deprojektion

Theoretisch könnte ein Pixel an die falsche Koordinate des Raumes deprojiziert werden. Bei vernünftiger Kalibrierung war die Deprojektion, soweit ich es mit meinen begrenzten Mitteln beurteilen kann, relativ genau und fällt in Anbetracht anderer Fehlerquellen nicht ins Gewicht. Die häufigste Ursache für fehlerhafte Deprojektionen ist eine schlechte Kamerakalibrierung:

- Belichtungszeit, Verstärkung, Laserleistung sollten vernünftig eingestellt sein. Vor allem ersteres kann je nach Lichtsituation eine Anpassung erfordern.
- Bei manchen Kameras muss der Color-Stream auf den Depth-Stream ausgerichtet werden (alignment), da es sonst zu groben Ungenauigkeiten näher zum Sichtfeldrand führen kann.

6.1.3 Verschiebung und Rotation

Die Transformation der Joint-Koordinaten aus dem Koordinatensystem der Kamera in das gewünschte Koordinatensystem ist ein wichtiger Schritt, der besondere Präzision erfordert.

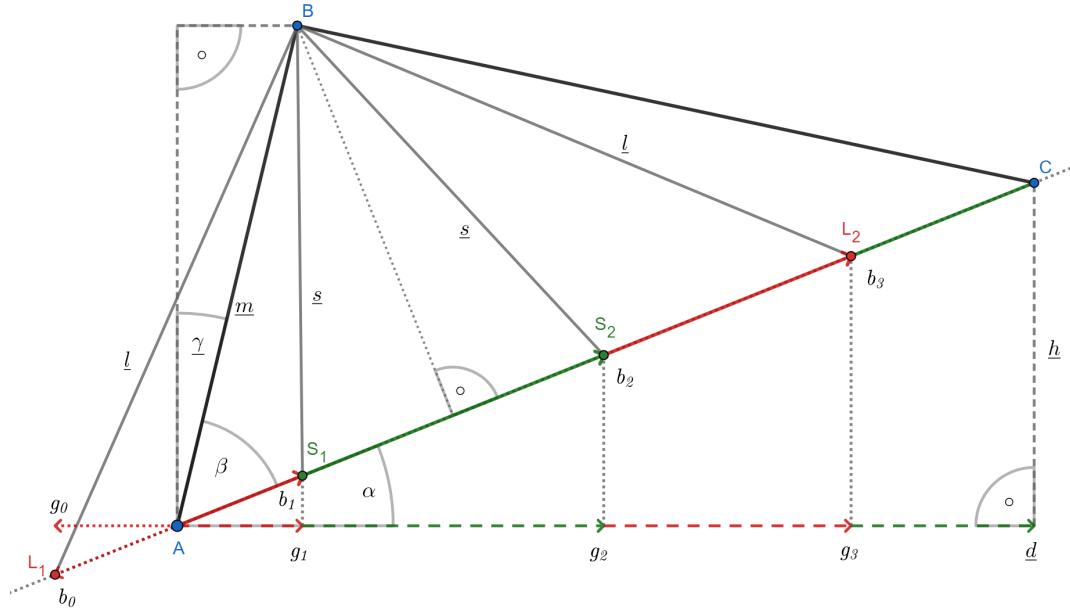


Abbildung 6.1: Visualisierung des Falls „Ein Joint verdeckt“. Unterstrichene Variablen sind gegeben. Das Ziel ist es, g_0, g_1, g_2, g_3 herauszufinden. Die Validierung funktioniert auf den grün markierten Bereichen, während sie auf den roten scheitert.

Meist ist die Koordinate des Manipulatorfußes bekannt. Ist die Kamera nicht rotiert, so lässt sich diese aus der Kameraperspektive mithilfe des Intel RealSense Viewers ablesen und die benötigte Verschiebung relativ genau einstellen.

Ohne genauere Messmethoden ist dazu zu raten, nach Möglichkeit die Kamera so auszurichten, dass diese in Richtung einer der Achsen des gewünschten Koordinatensystems zeigt, sodass die Rotation aus Gleichung (5.1) in 90° -Schritten angegeben werden kann. Ansonsten könnten kleine Ungenauigkeiten in groben Fehlern der Joint-Positionen im gewünschten Koordinatensystem resultieren.

6.1.4 Validierung

Fall: Ein Joint verdeckt

Die Validierung basiert auf der Annahme, dass bei der Verdeckung eines Joints die gemessene Länge einer Verbindung außerhalb eines definierten Akzeptanzintervalls liegt. Bei genau einem verdeckten Joint ist es wahrscheinlich, dass die Länge zwischen dem unverdeckten Joint und dem verdeckenden Objekt größer (und mit einer geringen Wahrscheinlichkeit kleiner) ist als der erwartete Längenbereich. Betrachten wir zunächst Abb. 6.1, um den Sachverhalt zu verstehen. Aus Einfachheit wird mit einer

2D-Präsentation gearbeitet, dies ist analog auf das drei-dimensionale übertragbar.

Die Punkte A und B repräsentieren zwei Joints, die den Abstand m zueinander haben. Punkt A ist fest und B ist um γ um A im Uhrzeigersinn gedreht, da Joints selbstverständlich nicht immer direkt übereinander liegen. Punkt C ist die Kamera und bildet zusammen mit den Joints das Dreieck $\triangle ABC$. C hat den waagerechten Abstand d und den senkrechten Abstand h von A .

Nun kann es passieren, dass Joint A verdeckt wird. Dazu können wir uns einen Punkt Q auf \overline{AC} vorstellen. Ausschlaggebend zur erfolgreichen Detektion der Verdeckung ist der Abstand $|\overline{QB}|$. Sei $[s, l]$ der Akzeptanzbereich, dann muss $|\overline{QB}| \notin [s, l]$ sein, damit eine Verdeckung erkannt wird. Dies können wir modellieren: Auf der Strecke $\overline{S_1 S_2}$ des gleichschenkligen Dreiecks $\triangle S_1 B S_2$ mit Schenkellänge s ist der Abstand zu B kleiner als s , sodass dort eine Verdeckung erkannt würde. Auf der Strecke $\overline{L_1 L_2}$ des gleichschenkligen Dreiecks $\triangle L_1 B L_2$ mit Schenkellänge l ist der Abstand zu B kleiner als l . Auf den Strecken $\overline{AS_1}$ (eig. $\overline{L_1 S_1}$, allerdings ist $\overline{L_1 A}$ für diesen Fall irrelevant, da dort ein Objekt A nicht verdecken würde) und $\overline{S_2 L_2}$ ist der Abstand zu B zwischen s und l , sodass eine Verdeckung in diesem Bereich nicht erkannt würde. Da durch eine variierende Kamerahöhe \overline{AC} variiert, interessieren wir uns für die Summe der waagerechten Längen der Strecken, wo eine Verdeckung nicht erkannt werden würde: $g_3 - g_2 + g_1$. Diese lässt sich geometrisch berechnen:

Gegeben sind die maximale Akzeptanzlänge $l \in \mathbb{R}^+$, die kürzeste Akzeptanzlänge $s \in [0, l]$, die tatsächliche Länge der Verbindung $m \in [s, l]$ (für den nächsten Fall wird die Funktion so konstruiert, dass diese auch auf $m \in \mathbb{R}^+$ funktioniert), der Neigungswinkel der Joints $\gamma \in [-\pi, \pi]$, der waagerechte Kameraabstand $d \in [\max(0, m \cdot \sin(\gamma)), \infty]$ (weiter rechts als A und B) und der vertikale Kameraabstand $h \in \mathbb{R}$.

Der Winkel α lässt sich aus den Regeln für rechtwinklige Dreiecke mit d, h berechnen:

$$\alpha = \arctan\left(\frac{h}{d}\right) \quad (6.1)$$

α, β und γ ergeben einen rechten Winkel, woraus sich β errechnen lässt:

$$\beta = \frac{\pi}{2} - \alpha - \gamma \quad (6.2)$$

Die Strecken b_0, b_1, b_2, b_3 lassen sich aus dem Kosinussatz berechnen:

$$s^2 = m^2 + b_{12}^2 - 2mb_{12} \cdot \cos(\beta) \quad (6.3)$$

$$\Leftrightarrow b_{12} = m \cdot \cos(\beta) \pm \sqrt{m^2 \cdot \cos^2(\beta) + s^2 - m^2} \quad (6.4)$$

Damit suchen wir nach der Länge einer Seite, der der Winkel β anliegt, in einem Dreieck, in dem die beiden anderen Seiten m und l lang sind. Man kann sich simpel visualisieren, dass das für zwei Seitenlängen erfüllt ist: b_1 mit „–“ und b_2 mit „+“. Zusätzlich sehen wir, dass, wenn der Wurzelterm 0 wird, $b_{12} = m \cdot \cos(\beta)$ gilt, also genau der Abstand zur Senkrechten von B auf \overline{AC} . Ist s zu kurz, so wird der Wurzelterm imaginär, was uns hier nichts nützt. Um die Funktion auf der gesamten Definitionsmenge sinnvoll zu definieren, definieren wir daher:

$$\overset{\circ}{\sqrt{x}} = \begin{cases} \sqrt{x} & \text{wenn } x \in \mathbb{R}^+, \\ 0 & \text{sonst.} \end{cases} \quad (6.5)$$

Damit erhalten wir:

$$b_1 = m \cdot \cos(\beta) - \overset{\circ}{\sqrt{m^2 \cdot \cos^2(\beta) + s^2 - m^2}} \quad (6.6)$$

$$b_2 = m \cdot \cos(\beta) + \overset{\circ}{\sqrt{m^2 \cdot \cos^2(\beta) + s^2 - m^2}} \quad (6.7)$$

Analog berechnen wir b_3 :

$$l^2 = m^2 + b_{03}^2 - 2mb_{03} \cdot \cos(\beta) \quad (6.8)$$

$$\Leftrightarrow b_{03} = m \cdot \cos(\beta) \pm \sqrt{m^2 \cdot \cos^2(\beta) + l^2 - m^2} \quad (6.9)$$

Wir sehen, dass $\sqrt{m^2 \cdot \cos^2(\beta) + l^2 - m^2} \geq m \cdot \cos(\beta)$ gilt, da $l \geq m$. Somit ist die „+“-Lösung immer positiv, und die „–“-Lösung immer negativ. Erste ist damit das gesuchte b_3 und letztere ist b_0 ($\overline{AL_1}$) und liegt im Fall $l \geq m$ stets links von A . Auch wenn dies hier garantiert ist (da die Verbindung nicht länger sein sollte, als das angegebene Maximum), schließen wir den Fall ein, dass b_0 rechts von A liegen könnte, mit ein, da dies für später nützlich sein wird:

$$b_0 = m \cdot \cos(\beta) - \overset{\circ}{\sqrt{m^2 \cdot \cos^2(\beta) + l^2 - m^2}} \quad (6.10)$$

$$b_3 = m \cdot \cos(\beta) + \overset{\circ}{\sqrt{m^2 \cdot \cos^2(\beta) + l^2 - m^2}} \quad (6.11)$$

Mithilfe der Regeln für rechtwinklige Dreiecke berechnen wir g_0, g_1, g_2, g_3 aus b_0, b_1, b_2, b_3 und α . Obwohl $b_3 \geq 0$ gilt, gilt dies nicht für b_1, b_2 . Da negative Abstände links von A und Abstände $> d$ rechts von der Kamera liegen und somit nicht zur Verdeckung beitragen, begrenzen wir die g_s im Folgenden auf den Bereich $[0, d]$. Dafür definieren wir:

$$\text{betw}(a, x, b) := \begin{cases} a & \text{wenn } x < a, \\ b & \text{wenn } x > b, \\ x & \text{sonst.} \end{cases} \quad (6.12)$$

$$g_0(m, l, s, d, h, \gamma) = \text{betw}[0, b_0 \cdot \cos(\alpha), d] = \dots \quad (6.13)$$

$$= \text{betw} \left[0, \frac{m \cdot \sin(\gamma + \arctan(\frac{h}{d})) - \sqrt{m^2 \cdot \sin^2(\gamma + \arctan(\frac{h}{d})) + l^2 - m^2}}{\sqrt{1 + \frac{h^2}{d^2}}}, d \right] \quad (6.14)$$

$$g_1(m, l, s, d, h, \gamma) = \text{betw}[0, b_1 \cdot \cos(\alpha), d] = \dots \quad (6.15)$$

$$= \text{betw} \left[0, \frac{m \cdot \sin(\gamma + \arctan(\frac{h}{d})) - \sqrt{m^2 \cdot \sin^2(\gamma + \arctan(\frac{h}{d})) + s^2 - m^2}}{\sqrt{1 + \frac{h^2}{d^2}}}, d \right] \quad (6.16)$$

$$g_2(m, l, s, d, h, \gamma) = \text{betw}[0, b_2 \cdot \cos(\alpha), d] = \dots \quad (6.17)$$

$$= \text{betw} \left[0, \frac{m \cdot \sin(\gamma + \arctan(\frac{h}{d})) + \sqrt{m^2 \cdot \sin^2(\gamma + \arctan(\frac{h}{d})) + s^2 - m^2}}{\sqrt{1 + \frac{h^2}{d^2}}}, d \right] \quad (6.18)$$

$$g_3(m, l, s, d, h, \gamma) = \text{betw}[0, b_3 \cdot \cos(\alpha), d] = \dots \quad (6.19)$$

$$= \text{betw} \left[0, \frac{m \cdot \sin(\gamma + \arctan(\frac{h}{d})) + \sqrt{m^2 \cdot \sin^2(\gamma + \arctan(\frac{h}{d})) + l^2 - m^2}}{\sqrt{1 + \frac{h^2}{d^2}}}, d \right] \quad (6.20)$$

Angemerkt werden sollte, dass für den hiesigen Fall $m \leq l$ $g_0 = 0$ gilt.

Damit können wir \tilde{f} definieren, das die waagerechte Gesamtstrecke angibt, auf der eine Verdeckung nicht erkannt wird:

$$\tilde{f}(m, l, s, d, h, \gamma) = g_3 - g_2 + g_1 - g_0 \quad (6.21)$$

Die Funktionsparameter der rechten Seite wurden der Übersichtlichkeit halber weggelassen.

Hinzu kommt die Möglichkeit, über eine maximale Tiefendifferenz $t \in \mathbb{R}^+$ zu validieren. In diesem Fall interessiert uns die Tiefendifferenz (waagerechte Komponente) zu B . Darstellen lässt sich das über zwei Punkte t_- und t_+ :

$$t_-(m, l, s, d, h, \gamma, t) = \text{betw}(g_0, m \cdot \sin(\gamma) - t, g_3) \quad (6.22)$$

$$t_+(m, l, s, d, h, \gamma, t) = \text{betw}(g_0, m \cdot \sin(\gamma) + t, g_3) \quad (6.23)$$

Die Funktionsparameter der rechten Seite wurden der Übersichtlichkeit halber weggelassen.

Bei $t = 0$ starten diese beiden Punkte auf der Waagerechten auf der Höhe von B und laufen gleichmäßig nach links und rechts. Innerhalb des Intervalls $[t_-, t_+]$ scheitert die Tiefenvalidierung, da dort die waagerechte (Tiefen-)Differenz zu B zu gering ist. Außerhalb des Intervalls funktioniert diese. Dabei sind sie durch g_0 nach links und durch g_3 nach rechts begrenzt, da ab den Grenzen dieses Bereiches die Validierung immer funktionieren würde (bzw. bei $g_0 = 0 \vee g_3 = d$ nicht sinnvoll ist).

Da wir bereits auf einigen Intervallen erfolgreich validieren können, interessiert uns für ein vollständiges Bild die Schnittmenge zwischen $[t_-, t_+]$ und den beiden rot markierten Intervallen $[g_0, g_1]$ und $[g_2, g_3]$, in denen die Validierung nicht funktioniert (in diesem Fall würde $[0, g_1]$ und $[g_2, g_3]$ reichen, aber wir halten es allgemein). Das ergibt dann die vollständige Funktion f , die den gesamten Abstand berechnet, auf dem die Validierung scheitert. Per Konstruktion wissen wir, dass $0 \leq g_0 \leq g_1 \leq g_2 \leq g_3 \leq d \wedge g_0 \leq t_- \leq t_+ \leq g_3$:

$$f(m, l, s, d, h, \gamma, t) = |([g_0, g_1] \cup [g_2, g_3]) \cap [t_-, t_+]| \quad (6.24)$$

$$= \max(0, \min(g_1, t_+) - \max(g_0, t_-)) + \max(0, \min(g_3, t_+) - \max(g_2, t_-)) \quad (6.25)$$

Nun können wir diese Funktion numerisch für verschiedene Kamerapositionen und Körperparameter analysieren und das Maximum finden. Zunächst

nehmen wir die Verbindung Schulter-Ellbogen. Diese hat bei mir eine maximale Akzeptanzlänge $l = 0.34$, eine minimale Akzeptanzlänge $s = 0.27$ und eine maximale Tiefenabweichung $t \rightarrow \infty$, da hier Tiefenvalidierung keinen Sinn ergibt. Jetzt suchen wir numerisch in den Bereichen der tatsächlichen Verbindungsgröße $m \in [s, l]$, des Kameraabstands $d \in [1, 5]$, der Kamerahöhe $h \in [-1.5, 1.5]$ und des Neigungswinkels der Verbindung $\gamma \in [-\pi/2, \pi/2]$. Dabei wird jeder Bereich in 100 Schritte geteilt, sodass insgesamt auf einer 10^8 Punkten umfassenden Menge gesucht wird. Der selbstgeschriebene Code ist unter [13] unter `theory.py` verfügbar; essenziell wird für jeden Punkt des Definitionsbereiches der Wert berechnet und der Maximale zusammen mit den dazugehörigen Parametern zurückgegeben. Zusätzlich wird der Durchschnitt, der Median und die Standardabweichung aller Bildwerte berechnet. Mithilfe von GPU-Parallelisierung ist dies in unter 20 Sekunden erledigt.

Das Maximum beträgt $f(m = 0.34, l = 0.34, s = 0.27, d = 1.566, h = -0.045, \gamma = 0.682, t \rightarrow \infty) = 0.413$ m. Das ist natürlich nicht besonders gut, jedoch handelt es sich um das numerische Maximum. Der Suchbereich war sehr fein, daher wird das tatsächliche Maximum im „vernünftigen“ Bereich kaum höher sein. Der Durchschnitt liegt bei 0.106 m, der Median bei 0.09 m und die Standardabweichung bei 0.076 m. Somit ist es wahrscheinlich, dass für die Verbindung Schulter-Ellbogen der Abstand, auf dem die Validierung fehlschlägt, bei unter 20 cm liegt. Das ist ein guter Wert. In Abb. 6.2 ist f für diese Verbindung geplottet. Dazu wurde ein Kameraabstand von $d = 3$ m und eine tatsächliche Verbindungsgröße von $m = 0.31$ m angenommen. Wie man sieht, ist f um $\gamma = 0$ und $\gamma = \pi$ herum maximal. Dazwischen fällt es auf 0.1 m und außerhalb auf beinahe 0 m ab. Zusätzlich sieht man einen leichten Anstieg zu $h = 0$ m bis $h = 0.5$ m. Dies würde bei einem größeren (unrealistischeren) Parameterbereich für h stärker ins Gewicht fallen.

Betrachten wir nun die Verbindung Nacken-Hüfte. Diese hat bei mir $l = 0.63, s = 0.5, t = 0.2$. Die Suchbereiche sind wie davor definiert. Das Maximum beträgt $f(m = 0.601, l = 0.63, s = 0.5, d = 1.0, h = -0.136, \gamma = 0.524, t = 0.2) = 0.4$ m. Im relativen Vergleich zur Größe der Verbindung ist dies besser als für die letztere. Der Durchschnitt liegt bei 0.059 m, der Median bei 0 m und die Standardabweichung bei 0.105 m. Da der Median deutlich kleiner als der Durchschnitt ist, deutet es darauf hin, dass es viele 0-Werte gibt (wo eine Verdeckung immer erkannt wird) und weniger, dafür umso höhere Werte, wo die Validierung wahrscheinlich scheitert. Somit ist es wahrscheinlich, dass für die Verbindung Nacken-Hüfte der Abstand, auf dem die Validierung fehlschlägt, bei unter 17 cm liegt. In Abb. 6.3 ist f für diese Verbindung geplottet. Dazu wurde ein Kameraabstand von $d = 3$ m

6 Evaluation

$f(m=0.31, l=0.34, s=0.27, d=3.0, h, \gamma, t=\infty)$

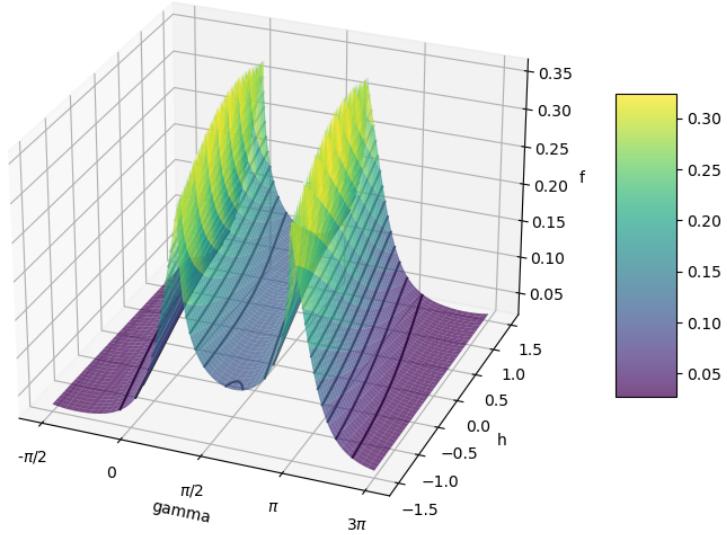


Abbildung 6.2: Funktion f für die Verbindung Schulter-Ellbogen: $f(m = 0.31, l = 0.34, s = 0.27, d = 3, h, \gamma, t \rightarrow \infty)$. Gibt bei einem verdeckten Joint die Gesamtlänge an, auf der die Validierung scheitert.

und eine tatsächliche Verbindungsänge von $m = 0.58$ m angenommen. Wie davor ist f um $\gamma = 0$ und $\gamma = \pi$ herum maximal, jedoch ist der Anstieg steiler. Dazwischen und außerhalb fällt es auf 0 m ab. Das bestätigt vorige Vermutungen. Bei einem höheren (unrealistischen) Bereich für h würde man sehen, wie f stark abfällt und sich ein Kamm in zwei spaltet.

In Abb. 6.4 ist diese Analyse für typische Körperverbindungen in komprimierter Form dargestellt. Für jede Verbindung ist das Akzeptanzintervall $[s, l]$ und maximale Tiefenabweichung t (durch meine Körperabmessungen) gegeben. Es wird pro Verbindung für die Bereiche $m \in [s, l], d \in [1, 5], h \in [-1.5, 1.5], \gamma \in [-\pi/2, \pi/2]$ mit je 100 Schritten pro Bereich der Wert von f berechnet und anschließend numerisches Maximum, Median, Mittel und Standardabweichung ermittelt. Mit einer Ausnahme liegen die Maxima für die Gesamtlängen, auf denen die Validierung fehlschlägt, bei unter ca. 40 cm. Mit einer Ausnahme ist die Wahrscheinlichkeit hoch, dass diese unter 20 cm bleibt.

6 Evaluation

$$f(m=0.58, l=0.63, s=0.5, d=3.0, h, \gamma, t=0.2)$$

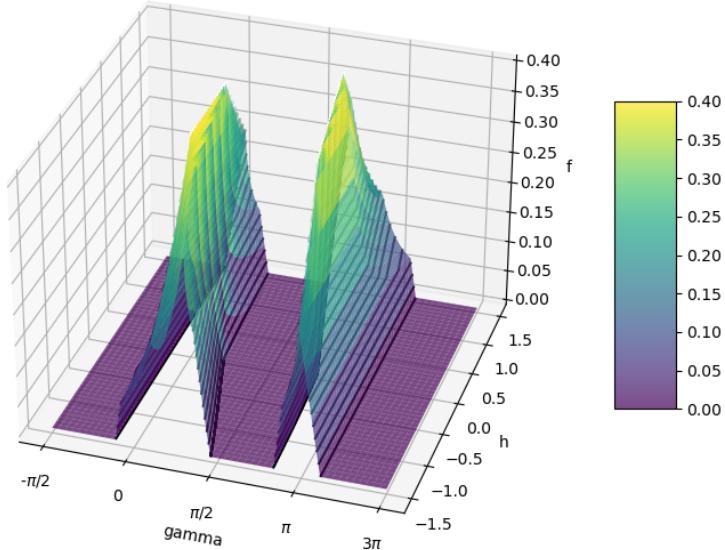


Abbildung 6.3: Funktion f für die Verbindung Nacken-Hüfte: $f(m = 0.58, l = 0.63, s = 0.5, d = 3, h, \gamma, t = 0.2)$. Gibt bei einem verdeckten Joint die Gesamtlänge an, auf der die Validierung scheitert.

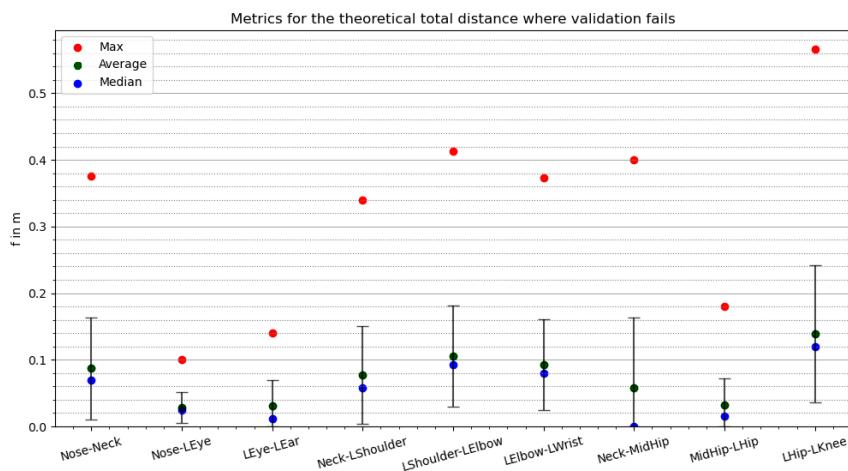


Abbildung 6.4: Numerische Statistiken für die Gesamtlänge, auf der die Validierung bei einem verdeckten Joint fehlschlagen wird, pro Verbindung. Die Parameter s, l, t sind für jede Verbindung (x-Achse) durch meine Kalibrierung basierend auf meinen Körperabmessungen gegeben und legen die angezeigte Körperverbindung fest. Basierend auf der Änderung restlicher Parameter wird Maximum, Mittel und Standardabweichung ermittelt.

Fall: Zwei Joints verdeckt

Die Validierung basiert auf der Annahme, dass bei der Verdeckung eines Joints die gemessene Länge einer Verbindung außerhalb eines definierten Akzeptanzintervalls liegt. Bei zwei verdeckten Joints ist es wahrscheinlich, dass die Länge zwischen den beiden verdeckenden Objekten kürzer (und mit einer geringeren Wahrscheinlichkeit länger) ist als der erwartete Längenbereich. Betrachten wir zunächst Abb. 6.5, um den Sachverhalt zu verstehen.

Gegeben ist das bekannte Dreieck aus Abb. 6.1, jedoch mit folgender Änderung: Dieses Mal wird zusätzlich B durch B' verdeckt. Wir beobachten das Folgende: Die Linien mit der Länge l und s können wir nun mit dem Punkt B' als Anker ziehen. Wenn wir jetzt m' und γ' berechnen, haben wir ein Problem, das zu vorigem äquivalent ist, sodass wir es mit der Funktion f berechnen können. Die Parameter l, s, d, h, t bleiben selbstverständlich gleich.

Wir haben den neuen Parameter $k \in [0, d - m \cdot \sin(\gamma)]$ gegeben, der beschreibt, wie weit der Punkt B' von B auf der Waagerechten ist. Da $B' B$ verdecken soll, liegt es zwischen B und C , wie mit dem Definitionsbereich von k beschrieben. Zusätzlich legen wir jetzt fest, dass $A = (0, 0)$, was keine Einschränkung darstellt.

Zunächst berechnen wir B' . Dazu stellen wir zwei Geradengleichungen auf und lösen diese. Die erste beschreibt \overline{BC} , und die zweite die Gerade, die durch den Punkt $(B'[x], B[y])$ senkrecht verläuft. Im Schnittpunkt der beiden Geraden liegt B .

$$\begin{pmatrix} m \cdot \sin(\gamma) \\ m \cdot \cos(\gamma) \end{pmatrix} + x_1 \cdot \begin{pmatrix} d - m \cdot \sin(\gamma) \\ h - m \cdot \cos(\gamma) \end{pmatrix} = \begin{pmatrix} m \cdot \sin(\gamma) + k \\ m \cdot \cos(\gamma) \end{pmatrix} + x_2 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad (6.26)$$

$$\Leftrightarrow x_2 = k \cdot \frac{h - m \cdot \cos(\gamma)}{d - m \cdot \sin(\gamma)} \quad (6.27)$$

$$\Rightarrow B' = \begin{pmatrix} m \cdot \sin(\gamma) + k \\ m \cdot \cos(\gamma) + k \cdot \frac{h - m \cdot \cos(\gamma)}{d - m \cdot \sin(\gamma)} \end{pmatrix} \quad (6.28)$$

Um m' zu berechnen, berechnen wir den Abstand zwischen A (dem Nullpunkt) und B' :

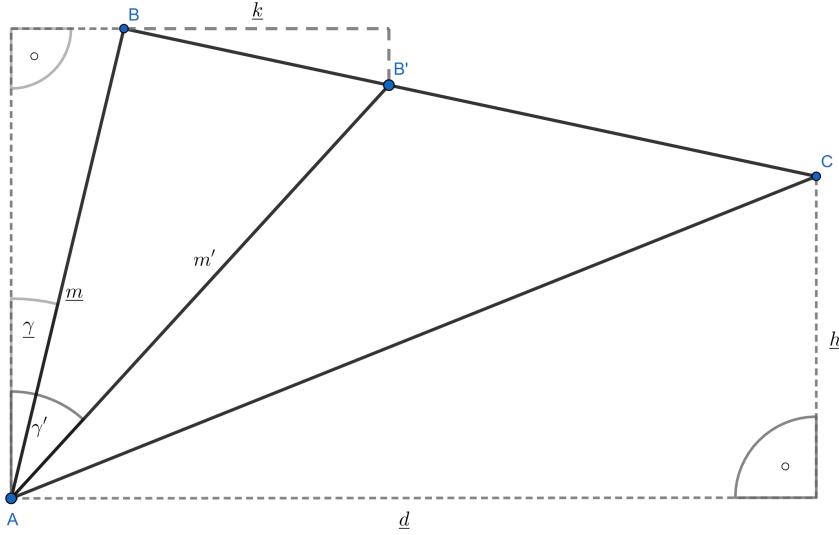


Abbildung 6.5: Visualisierung des Falls „Zwei Joints verdeckt“. Unterstrichene Variablen sind gegeben. Das Ziel ist es, m' , γ' herauszufinden, damit das Problem durch die Funktion f ausgedrückt werden kann.

$$m' = \|B' - A\| = \|B'\| \quad (6.29)$$

$$= \sqrt{(k + m \cdot \sin(\gamma))^2 + \left(m \cdot \cos(\gamma) + k \cdot \frac{h - m \cdot \cos(\gamma)}{d - m \cdot \sin(\gamma)} \right)^2} \quad (6.30)$$

Nun können wir γ' berechnen. Da auf einer Sinus-Periode ein Bild zweier Urbilder hat, müssen wir hier eine Fallunterscheidung einfügen, um γ' auf γ anzupassen. Falls B' unter der x -Achse liegt, nehmen wir die „andere“ Lösung der Periode:

$$\gamma' = \begin{cases} \arcsin\left(\frac{m \cdot \sin(\gamma) + k}{m'}\right) & \text{wenn } B'[y] \geq 0, \\ \pi - \arcsin\left(\frac{m \cdot \sin(\gamma) + k}{m'}\right) & \text{sonst.} \end{cases} \quad (6.31)$$

Damit können wir f_2 definieren, dass uns die Gesamtstrecke, auf der die Validierung bei zwei verdeckten Joints scheitert, berechnet:

$$f_2(m, l, s, d, h, \gamma, t, k) = f(m', l, s, d, h, \gamma', t) \quad (6.32)$$

Zu beachten ist, dass die Annahme $s \leq m' \leq l$ aus dem ersten Fall nicht mehr gilt. Damit könnte der Punkt L_1 aus Abb. 6.1 rechts von A liegen. Daher war es bei der Definition von f wichtig, g_0 mitzuberücksichtigen. Außerdem war es wichtig, g_0, g_1, g_2, g_3 auf den Bereich $[0, d]$ zu begrenzen.

Nun können wir auch f_2 numerisch untersuchen. Die Suchbereiche sind wie vorher definiert, zusätzlich haben wir jetzt den Suchbereich $k \in [0, d + l]$. $d + l$ ist das Maximum, das ein sinnvoll gewähltes k über alle anderen Parameter annimmt. Daher kann es vorkommen, dass entgegen der Annahme in manchen Fällen $k \notin [0, d - m \cdot \sin(\gamma)]$ gilt (B' vor B oder hinter C). Diese Werte werden auf NaN gesetzt und ignoriert. Weiterhin wird jeder Bereich aus Speichergründen diesmal in 40 zu untersuchende Stellen unterteilt. So mit werden 1.024×10^8 Punkte untersucht.

Abb. 6.6 stellt in komprimierter Form für typische Körperverbindungen bei zwei verdeckten Joints die Gesamtlänge dar, auf denen die Validierung fehlschlägt. Das Maximum bleibt gleich, und zwar für den Fall $k = 0$, also für den Fall, dass nur ein Joint verdeckt ist. Die Standardabweichung wird bei allen Verbindungen zum Teil deutlich kleiner. Bei einigen Gliedmaßen sinken Durchschnitt und Median, bei anderen steigen sie. So betragen diese bei der Verbindung Schulter-Ellbogen 14 cm bzw. 13.5 cm und somit in etwa 3.5 cm mehr. Jedoch bleibt die Summe aus Durchschnitt und einer Standardabweichung stabil bei 18 cm. In allen Fällen sinkt dieser Wert oder bleibt zumindest gleich. Der Durchschnitt der Verbindung Nacken-Hüfte fällt um 4 cm auf 2 cm. Die Summe aus Standardabweichung und Mittel beträgt mit 8 cm die Hälfte im Vergleich zu davor. Somit vergrößern sich im Fall, dass zwei Joints verdeckt sind, die Gesamtlängen, auf denen die Validierung scheitert, nur in wenigen Fällen um einen niedrigeren Wert als sie in anderen Fällen sinken.

In Abb. 6.7 und 6.8 sieht man beispielhaft Graphen in Abhängigkeit von γ und h mit Werten aus vorheriger Sektion und $k = 0.5$. Bei diesen Werten sinken in beiden Fällen im Vergleich zu davor die Maximalwerte. Für die Verbindung Schulter-Ellbogen flachen die beiden Piks ab, die Verbindung Nacken-Hüfte bekommt zwei „Dellen“.

6 Evaluation

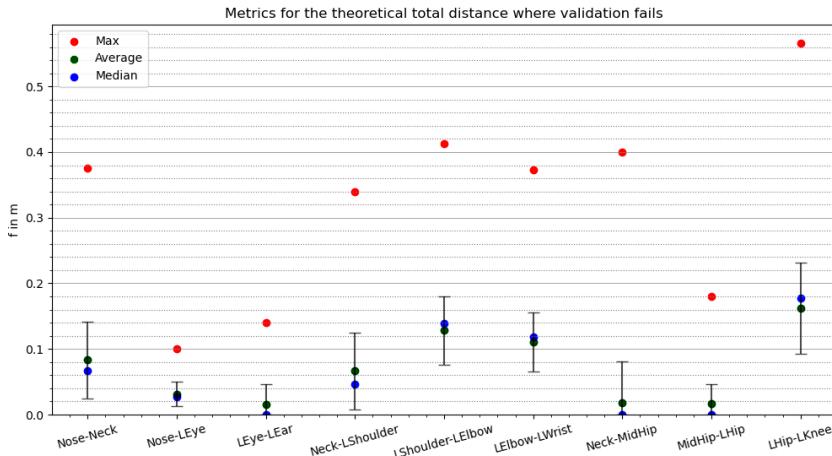


Abbildung 6.6: Numerische Statistiken für die Gesamtlänge, auf der die Validierung bei zwei verdeckten Joints fehlschlagen wird, pro Verbindung. Die Parameter s, l, t sind für jede Verbindung (x-Achse) durch meine Kalibrierung basierend auf meinen Körperabmessungen gegeben. Basierend auf der Änderung restlicher Parameter wird Maximum, Mittel und Standardabweichung ermittelt.

Zusammengefasst:

Die angewandte Methode zur Validierung hat natürliche theoretische Limitierungen, jedoch halten sich diese für einen Großteil der Möglichkeiten in Grenzen. Da Kameraabstand und -höhe innerhalb vernünftiger Werte generell einen irrelevanten Beitrag zum Gesamtabstand, auf dem die Validierung scheitert, leisten, lassen sich auch keine generellen Kameraabstände und -höhen empfehlen. Die Kamera sollte so positioniert werden, dass der Mensch möglichst wenig durch andere Objekte verdeckt wird und gleichzeitig OpenPose gut funktioniert. Der Winkel der Gliedmaße hat einen großen Einfluss auf die theoretische Limitierung, dieser lässt sich jedoch nur schwer beeinflussen. Daher ist es besonders wichtig, die Akzeptanzbereiche für Längen und Tiefen gut einzugrenzen und zu konfigurieren, da ansonsten die Fehleranfälligkeit steigt.

6.2 Korrektheit

Abseits der theoretischen Überlegungen ist es schwer, die Korrektheit der Position und Pose des Menschen zu verifizieren, da mir keine Ground-Truth-Daten vorliegen.

Die Simulation sieht innerhalb der gegebenen Möglichkeiten vernünftig aus, obwohl mit dieser Methode unvermeidliche Fehler natürlich vorhanden sind. Dennoch gibt es einige Möglichkeiten, die Korrektheit mit statistischer Grundlage zu plausibilisieren. Dazu verwenden wir eine Aufnahme, in der

6 Evaluation

$f_2(m=0.31, l=0.34, s=0.27, d=3.0, h, \gamma, t=\infty, k=0.5)$

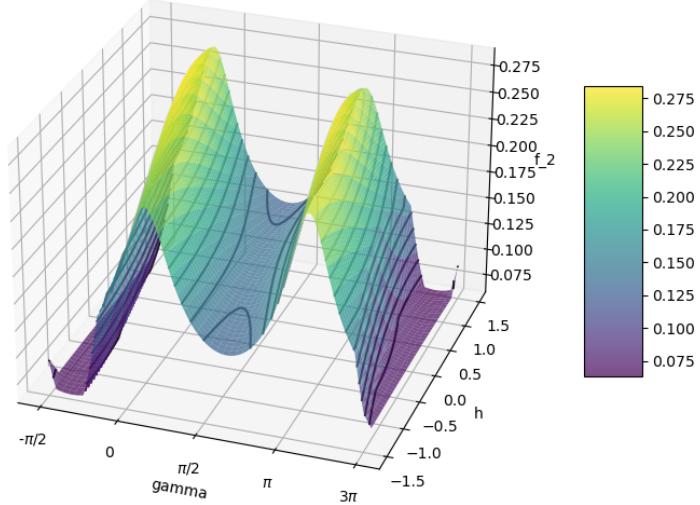


Abbildung 6.7: Funktion f_2 für die Verbindung Schulter-Ellbogen: $f_2(m = 0.31, l = 0.34, s = 0.27, d = 3, h, \gamma, t \rightarrow \infty, k = 0.5)$. Gibt bei zwei verdeckten Joints die Gesamtlänge an, auf der die Validierung scheitert.

$f_2(m=0.58, l=0.63, s=0.5, d=3.0, h, \gamma, t=0.2, k=0.5)$

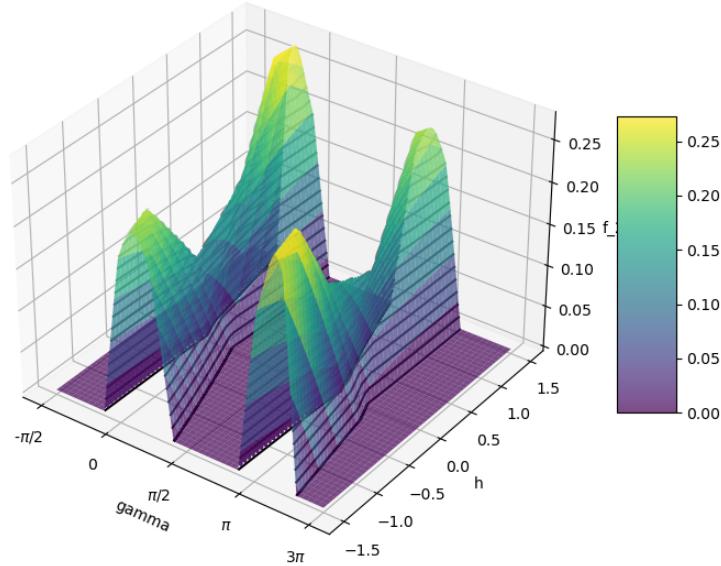


Abbildung 6.8: Funktion f_2 für die Verbindung Nacken-Hüfte: $f_2(m = 0.58, l = 0.63, s = 0.5, d = 3, h, \gamma, t = 0.2, k = 0.5)$. Gibt bei zwei verdeckten Joints die Gesamtlänge an, auf der die Validierung scheitert.

ich ca. 30 min. lang um den Roboter lief. Meine Joint-Positionen wurden regelmäßig in kleinen Zeitabständen aufgenommen und ausgewertet.

Joint-Geschwindigkeiten

Wie Abb. 6.9 zeigt, bewegen sich Joints, für die genügend Datenpunkte vorliegen, unter Berücksichtigung der Standardabweichung mit bis zu ca. 0.6 m/s. Ausnahmen stellen die Hände und Ellbogen dar, die sich unter Berücksichtigung der Standardabweichung mit bis zu 1.1 m/s bewegen. Da der Median stets unter dem Mittel liegt, deutet es darauf hin, dass die Joints häufiger langsam waren, aber dafür teilweise höhere Geschwindigkeiten erreichten. Diese Werte liegen im vernünftigen Bereich.

Invalide Verbindungen

Invalide Verbindungen zwischen Joints sind solche, die nicht im konfigurierten Längenbereich liegen, oder deren Tiefendifferenz zu hoch ist. Das kann auftreten, wenn beide Joints über andere Verbindungen validiert werden und daher als valide betrachtet werden. Wie man in Abb. 6.10 sieht, tritt dies auch nicht gerade selten auf. Vor allem die Verbindungen Nase-Nacken, Nacken-Rechte Schulter, Nacken-Linke Schulter, Hüfte-Linke Hüfte und Hüfte-Rechte Hüfte stechen besonders negativ heraus. Bei den beiden letzteren haben invalide Verbindungen einen Anteil an knapp über 50%.

Das deutet zwar natürlich darauf hin, dass die Methode keine perfekte Präzision hat. Jedoch wurden die Akzeptanzbereiche bewusst so gewählt, sodass natürliche Positionen in guten Winkeln (sodass der Joint nicht durch den eigenen Körper verdeckt wird) knapp validiert werden. Daher betrachten wir die Differenzen der invaliden Verbindungen zum Akzeptanzbereich:

In Abb. 6.11 sind solche Verbindungen aufgelistet, die zu lang sind. Alle Verbindungen, für die genügend Daten vorliegen, bleiben in der Standardabweichung unter 15 cm Differenz zum Maximalwert, die meisten unter 10 cm. Die beiden Verbindungen Ellbogen-Handgelenk sind neben der Verbindung Nase-Nacken besonders häufig zu lang. Eine vernünftige Erklärung hierfür ist, dass die Handgelenke zusätzlich über Farbe (e.g. grüne Handschuhe) validiert wurden, um die Robustheit zu erhöhen. Dies war insbesondere dann der Fall, wenn das Handgelenk nicht über die Verbindungen zum Ellbogen validiert werden konnte. Das erklärt den leichten Ausschlag für diese beiden Verbindungen. Da die Pipeline nie auf exakte Präzision ausgelegt war, sind diese Abweichungen akzeptabel.

In Abb. 6.12 werden solche Verbindungen aufgelistet, die zu kurz sind.

Alle Verbindungen, für die genügend Daten vorliegen, bleiben in der Standardabweichung deutlich unter 15 cm Differenz zum Minimalwerten, die meisten sind deutlich unter 10 cm. Den höchsten Ausschlag hat hier die Verbindungen Nacken-Hüfte. Der Minimalwert wurde hier so gewählt, dass dieser bei einem aufrechten Gang validiert. Beim Sitzen oder Krümmen wird die Distanz automatisch kleiner. Die beiden Verbindungen Nacken-Schulter und die zwei Hüftverbindungen sind besonders häufig zu kurz. Eine plausible Erklärung hierfür ist, dass diese Distanz aus Kameraperspektive deutlich kleiner wird, wenn der Mensch seitlich zur Kamera orientiert ist. Die 3D-Koordinaten beider Joints werden auf eine Schulter bzw. Seite projiziert, da Nacken bzw. mittlere Hüfte aus Kameraperspektive durch den eigenen Körper verdeckt werden. Auch hier sind die Abweichungen akzeptabel.

In Abb. 6.13 werden solche Verbindungen aufgelistet, deren Tiefenabweichung zu lang ist. Das hat natürlich eine gewisse Korrelation mit ersterem Fall. Alle Verbindungen, für die genügend Daten vorliegen, bleiben in der Standardabweichung deutlich unter 15 cm Differenz zu den Maximalwerten, mit Ausnahme der Verbindung Nacken-Hüfte, die mit über 20 cm Differenz ein Ausreißer ist. Dies hat womöglich damit zu tun, dass der Tisch in dem Set-up der Aufnahme teilweise den Hüft-Joint verdeckte. Das trifft sich auch mit der relativ geringen Anzahl von ca. 800 auftretenden Fällen. Mit Ausnahme dieser Verbindung sind die Differenzen akzeptabel. Als erster Fehlerbehebungsschritt sollten die fehlerträchtigen Verbindungen zusammen mit den anliegenden Verbindungen neu kalibriert und die Akzeptanzbereiche verkleinert werden.

Zusätzlich sollte nicht unerwähnt bleiben, dass dadurch, dass die meisten Simulationsobjekte aus zwei Joints berechnet werden, diese Ungenauigkeiten obere Schranken darstellen, da die Positionen solcher Objekte über zwei Joints gemittelt berechnet werden.

Zusammengefasst: Innerhalb der mir gegebenen Möglichkeiten liegen selbstverständlich Ungenauigkeiten vor, jedoch meist keine erheblichen.

6 Evaluation

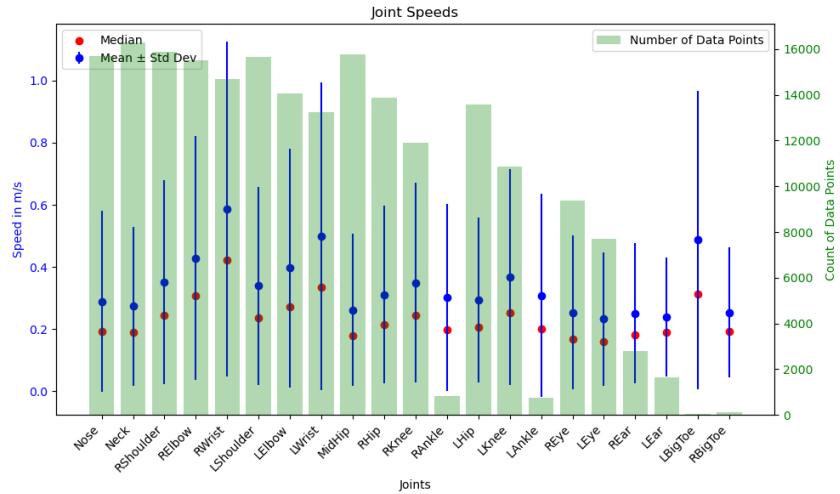


Abbildung 6.9: Joint-Geschwindigkeiten aus einer Aufnahme. Die Daten wurden erhoben, indem jeweils zwei valide, aufeinanderfolgende Joint-Positionen mit der Zeitdifferenz der jeweiligen Frames verrechnet wurden. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

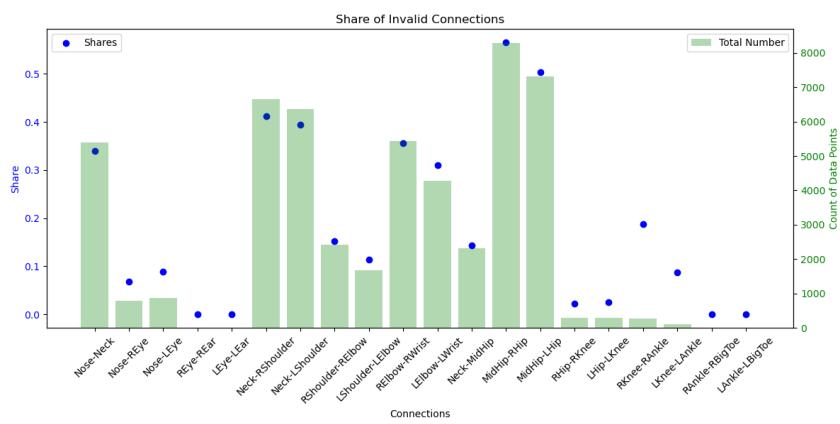


Abbildung 6.10: Der Anteil invalider Joint-Verbindungen einer Aufnahme. Die grünen Balken zeigen die Gesamtzahl dieser an. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

6 Evaluation

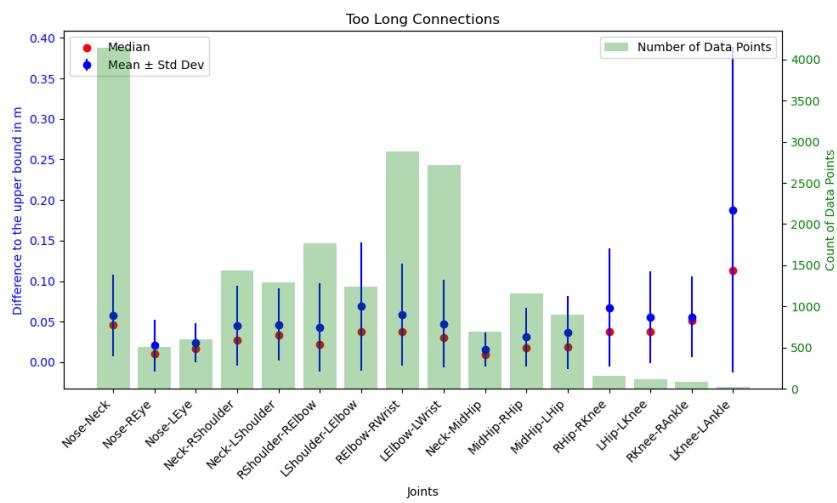


Abbildung 6.11: Die absoluten Differenzen der zu langen invaliden Verbindungen zu den Maximallängen einer Aufnahme. Die grünen Balken zeigen die Gesamtzahl der zu langen invaliden Verbindungen an. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

6 Evaluation

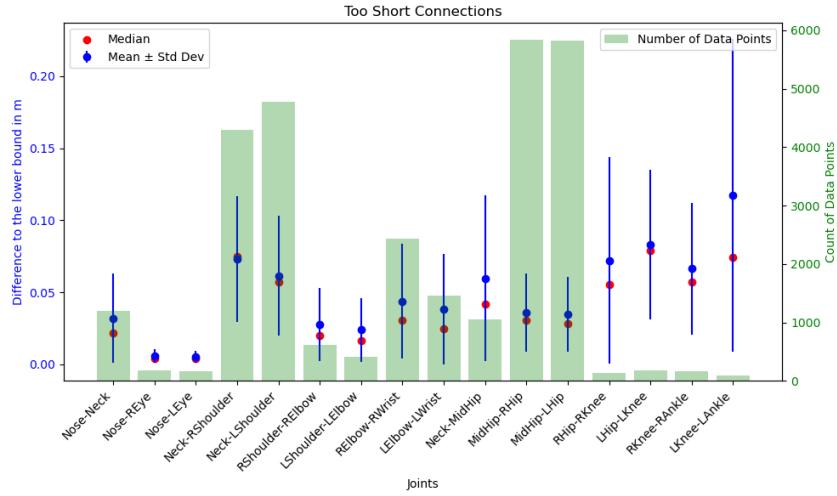


Abbildung 6.12: Die absoluten Differenzen der zu kurzen invaliden Verbindungen zu den Minimallängen einer Aufnahme. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

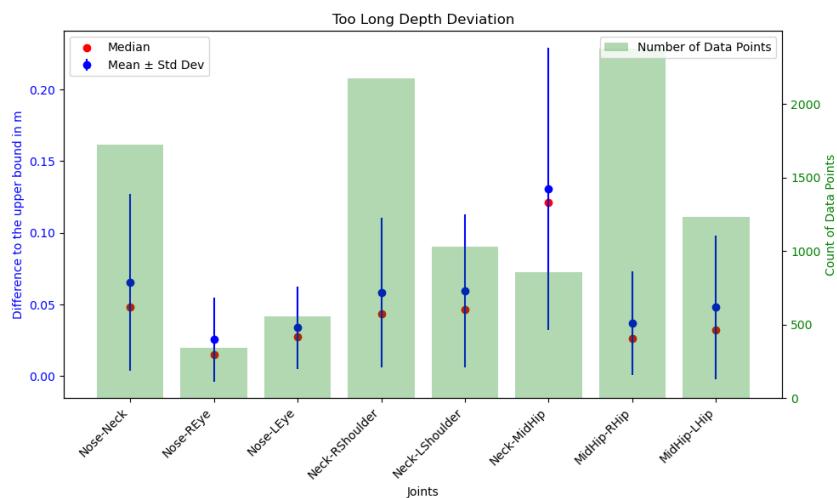


Abbildung 6.13: Die absoluten Differenzen der Verbindungen, die eine zu hohe Tiefenabweichung haben, zu den maximalen Tiefendifferenz einer Aufnahme. Die grünen Balken zeigen die Gesamtzahl dieser an. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

6.3 Robustheit

Die präziseste Pipeline bringt nichts, wenn Daten nur spärlich vorhanden sind. Mit dieser Idee wurde diese Pipeline designt und dafür werden (teils größere) Abschläge der Präzision in Kauf genommen. Ein Beispiel dafür ist der Validierungsprozess. Ein Joint wird über *mindestens* eine seiner Verbindungen validiert. Eine auf höhere Präzision ausgelegte Pipeline würde hier stärkere Anforderungen stellen.

Wie man in Abb. 6.14 sieht, ist die Robustheit sehr zufriedenstellend. Die wichtigsten Joints haben in über 80% der Fälle Daten vorliegen, die meisten in über 90%. Trotz aktiver Validierung über Farbe (e.g. grüne Handschuhe) liegt vor allem das linke Handgelenk abgeschlagen im unteren hohen Bereich. Das erklärt sich wahrscheinlich dadurch, dass die Hände besonders mobil sind und sich am schnellsten bewegen, wie man auch in Abb. 6.9 sieht. Ohne Farbvalidierung wäre dieser Wert deutlich niedriger, da nur über die Verbindung Handgelenk-Ellbogen verifiziert werden könnte.

Weiterhin können wir berücksichtigen, wie oft Joints für wie viele Sekunden durchgehend valide waren. Der relativ unbewegliche Hüft-Joint war mehrere Male bis zu 90 s lang durchgehend valide, wie man in Abb. 6.15 sieht. Sogar der relativ mobile Joint des linken Handgelenks Abb. 6.16 war einmal bis zu 90 s lang durchgehend valide, wobei seine Strähnen eher im Bereich von unter 40 s liegen.

Damit ist die Robustheit sehr zufriedenstellend.

6 Evaluation

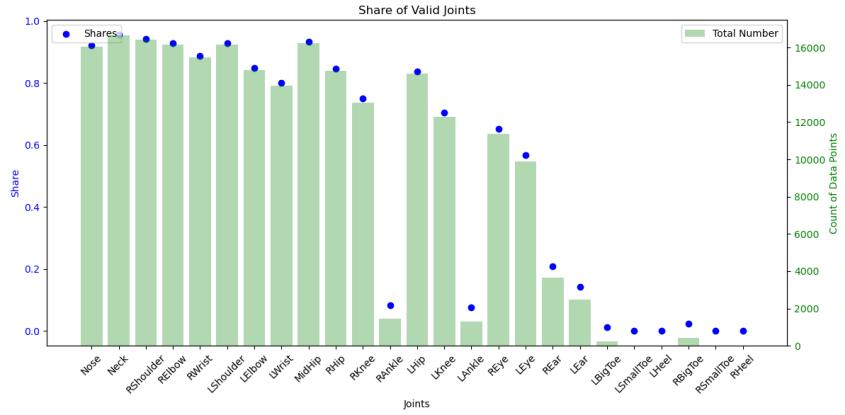


Abbildung 6.14: Der Anteil valider Joints an der Anzahl der Frames einer Aufnahme. Die grünen Balken zeigen die Gesamtzahl valider Joints an.

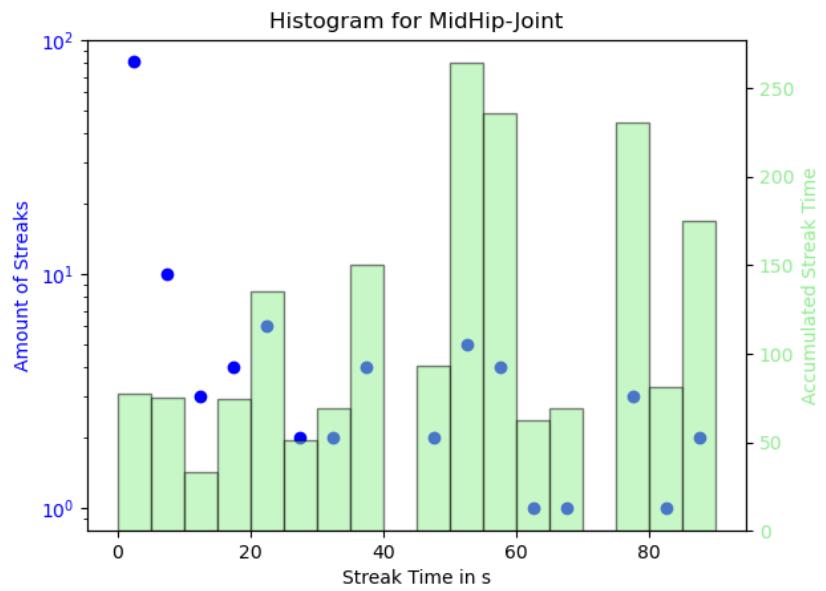


Abbildung 6.15: Histogramm der Strähnen (Streaks) für den Hüft-Joint. Es wird dargestellt, wie oft dieser bestimmte Zeiten lang durchgehend erkannt wurde.

6 Evaluation

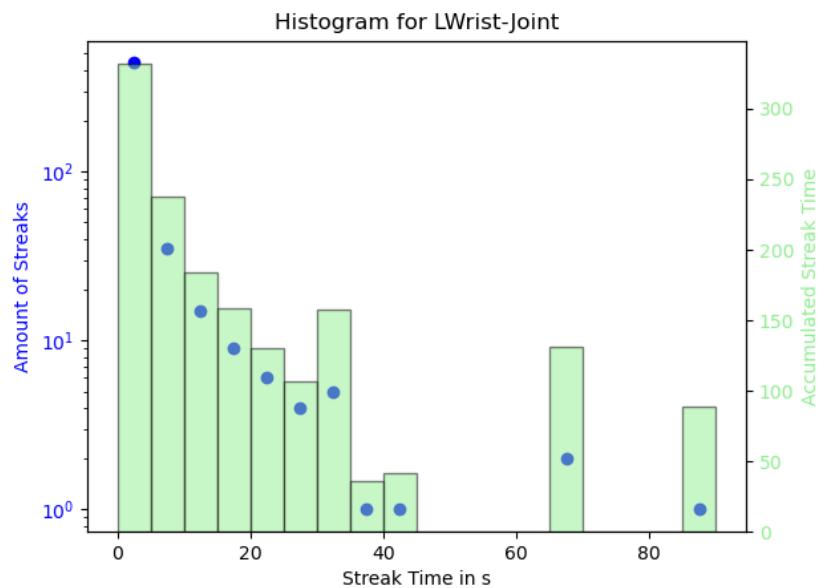


Abbildung 6.16: Histogramm der Strähnen (Streaks) für den Joint des linken Handgelenks.
Es wird dargestellt, wie oft dieser bestimmte Zeiten lang durchgehend erkannt wurde.

6.4 Performance

Zu guter Letzt wollen wir die Performance der Pipeline evaluieren. Die Tests wurden mit folgendem Set-up durchgeführt:

Tabelle 6.1: Hardware

Parameter	Spezifikation
Mainboard	MSI B450M PRO-M2 MAX
CPU	AMD Ryzen 9 3900X, 12C/24T, 3.8-4.6 GHz
RAM	Corsair Vengeance 32GB (2 × 16GB) DDR4 3200MHz
GPU	NVIDIA GeForce RTX 3070 8GB, Zotac Gaming Twin Edge OC
Kamera	Intel RealSense Depth Camera D435i

Tabelle 6.2: Software

Parameter	Spezifikation
Betriebssystem	Microsoft Windows 11 Pro
Python	v3.10.11
OpenPose	v1.7.0 mit Standardnetzauflösung
pyrealsense	v2.54.2.5684
pybullet	v3.2.6
numpy	v1.26.2
opencv-python	v4.8.1.78

Die Ergebnisse in der Realität sind in Abb. 6.17 dargestellt. Die Kameraauflösung betrug 640×480 . Die Kamera läuft mit 30 FPS, sodass die Wartezeiten des Perceptors auf den nächsten Frame beinahe immer 0 s betragen. Auch die Validierung hat eine gute Performance und ist größtenteils in unter 0.02 s fertig. Der Flaschenhals ist, wie erwartet, das Extrahieren der Joint-Positionen mit OpenPose. Dieser benötigt im Schnitt zwischen 0.9 s und 0.11 s. Dadurch benötigt der Perceptor im Schnitt zwischen 0.11 s und 0.135 s pro Frame. Der Simulator ist ebenfalls performant, denn ein Simulationsschritt benötigt im Durchschnitt bis zu 0.025 s. Die längste Zeit wartet dieser auf neue Daten vom Perceptor. Damit wird insgesamt ein Frame im Schnitt alle 0.11 s bis 0.14 s simuliert. Das sind 7.14 – 9.09 FPS.

Mir liegt noch eine Bag-Datei vor, die mit einer anderen Kamera aufgenommen wurde, die eine niedrigere Auflösung von 480×270 zuließ. Das genaue Modell ist mir nicht bekannt, es handelte sich um eine Intel RealSense der D400er Reihe. Die Bag-Datei umfasst rohe eine RGBD-Aufnahme, womit ich realistische Bedingungen beim Abspielen der Datei nachstellen kann. Der Performance-Test in Abb. 6.18 zeigt ähnliche Werte wie davor.

6 Evaluation

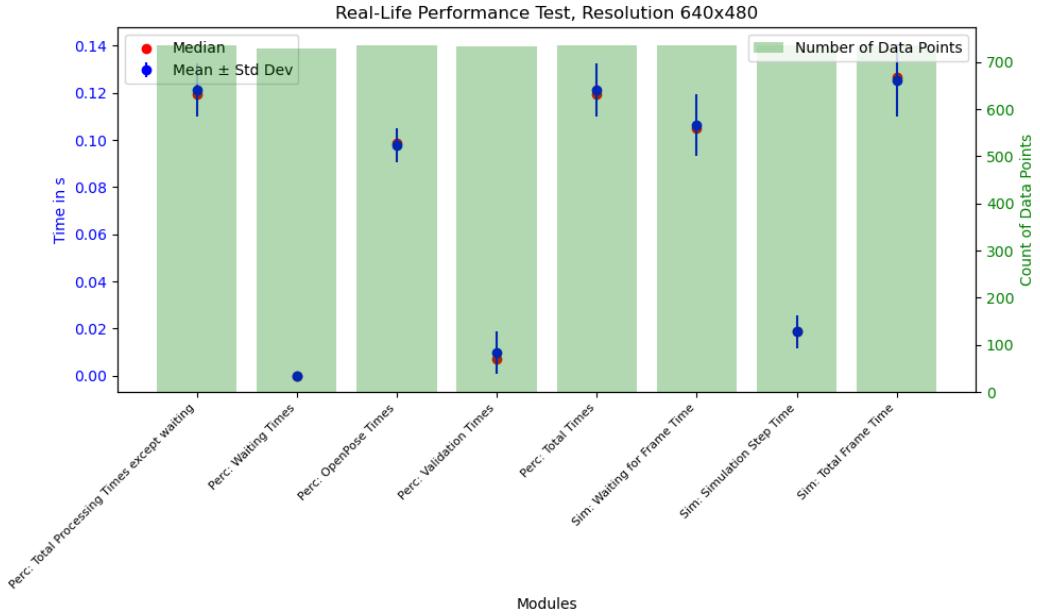


Abbildung 6.17: Reale Performance mit einer Auflösung von 640×480 im Test. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

Jedoch benötigt OpenPose durch die niedrigere Auflösung im Schnitt nur noch $0.0695 - 0.0825$ s pro Frame. Damit wird ein Frame im Schnitt alle $0.0875 - 0.117$ s simuliert. Das ergibt eine Verbesserung auf 8.55 – 11.43 FPS.

Somit ist klar, dass OpenPose in dieser Pipeline den Flaschenhals darstellt. Die Netzwerkauflösung von OpenPose wurde hier in der Standardeinstellung belassen. Durch eine niedrigere Auflösung lässt sich die Performance erhöhen, jedoch leidet die Qualität. Ich befand, dass die Standardeinstellung einen guten Kompromiss zwischen Qualität und Performance eingeht.

Damit liefert die Pipeline die Daten mit einer Verzögerung, die gering genug ist, und mit einem Durchsatz, der hoch genug ist, um sich für das experimentelle Deployment von Manipulatoren zu eignen.

Der große Vorteil ist, dass die Rechenanforderungen an die Simulation gering sind. Trainingsdaten werden als Jointpositionen aufgenommen und benötigen nur den Simulator, ohne Perceptor und OpenPose. Dadurch lässt sich im Trainingsprozess mit deutlich höherer Rate simulieren, da zumindest die Simulation des menschlichen Hindernisses keinen Flaschenhals mehr für das Training darstellt.

6 Evaluation

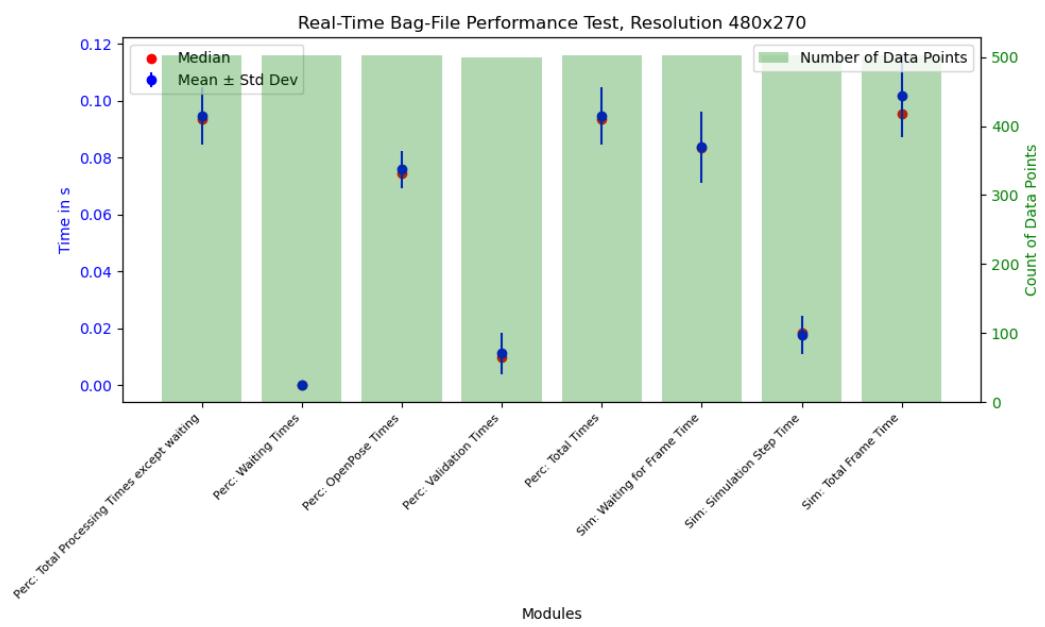


Abbildung 6.18: Performance einer abgespielten Bag-Datei mit einer Auflösung von 480×270 im Test. Die Daten wurden von Ausreißern durch das Entfernen von Datenpunkten bereinigt, die mehr als $z = 3$ Standardabweichungen vom Mittel entfernt liegen.

7 Fazit

In dieser Bachelorarbeit wurde eine Pipeline entwickelt, die einen Menschen in Echtzeit drei-dimensional wahrnimmt. Dabei wird ausschließlich der Mensch wahrgenommen und alle anderen Objekte ausgeblendet.

Über die Validierung und die Umgebungssuche wurden Techniken implementiert, die diese Pipeline präziser und robuster machen. So werden gröbere Fehler bei guter Konfiguration zuverlässig erkannt und über die Umgebungssuche wird versucht, diese Fehler zu beheben.

Es wurde gezeigt, wie die Pipeline über eine einzige Datei großflächig konfiguriert und an individuelle Bedürfnisse angepasst werden kann. Dabei ist die individuelle Zuschneidung auf einen Menschen durch das Setzen von Akzeptanzintervallen für Längen und Tiefen der einzelnen Verbindung zur Erhöhung der Präzision besonders kritisch, ebenso wie die Zuschneidung auf eine bestimmte Situation durch das Setzen von Suchbereichen und Farbintervallen. Dafür werden Debug-Tools bereitgestellt, die diese Konfiguration erleichtern.

Weiterhin wurde ein leichtgewichtiger Simulator in PyBullet entwickelt, der den Menschen mit simplen geometrischen Formen darstellt. Die Möglichkeit der Kollisionserkennung ist durch PyBullet automatisch gegeben.

Es wurde eine Aufnahme- und Abspiel-Funktion integriert. Dabei werden Zeitstempel mitgespeichert, was ein realistisches Abspielen ermöglicht. Dabei ist die Abspiel-Funktion von der eigentlichen Wahrnehmung entkoppelt und benutzt nur den leichtgewichtigen Simulator. Sollte auf aufgenommenen Daten trainiert werden, wird der Trainingsprozess beschleunigt, da die Simulation keinen Flaschenhals darstellt.

Die Pipeline wurde auch mit dem Ziel der einfachen Integrierbarkeit entwickelt. Diese wurde beispielhaft an der Integration in die IR-DRL-Suite demonstriert.

Die theoretische Evaluation ergab, dass der Validierungsprozess natürlichen Beschränkungen unterliegt. Es wurde ebenfalls gezeigt, dass sich diese Beschränkungen in Grenzen halten und für Proof of Concept und experi-

7 Fazit

mentelle Trainings und Deployments akzeptabel sind. Statistische Evaluationen der Ergebnisse im Rahmen meiner Möglichkeiten deuten ebenfalls auf eine akzeptable Korrektheit hin – und auf eine sehr zufriedenstellende Robustheit. Auch die Bildrate der Pipeline ist hoch genug, um sich für das experimentelle Deployment zu eignen. Die Simulation an sich ist sehr effizient, dadurch wird der Rechenaufwand für den Trainingsprozess stark entlastet.

Diese Arbeit stellt somit eine Perception-Pipeline vor, die sowohl im Training als auch im Deployment weiterführender Arbeiten im Bereich der Kollisionsvermeidungsalgorithmen als Grundlage der robotischen Wahrnehmung dienen kann.

8 Weiterführende Arbeiten

Es gibt mehrere Bereiche, in denen weiterführende Arbeiten an diese anschließen könnten:

8.1 Wahrnehmung

Zum einen kann die Präzision der Wahrnehmung weiter verbessert werden. Wie gezeigt wurde, unterliegt meine Methode natürlichen Beschränkungen. So könnten Relationen zwischen mehr als zwei Joints verwendet werden, um daraus ihre Validität abzuleiten. Beispielsweise könnten zusätzlich akzeptierte Winkel zwischen jeweils drei Joints definiert werden. Auch weitere, komplexere Abhängigkeiten sind denkbar. So könnte die berechnete Pose mit einem allgemeinen Posenmodell abgeglichen werden.

Ein Mensch könnte über Segmentierung von anderen Objekten unterschieden werden. Dieser Ansatz wurde zu Anfang dieses Projektes mit trainierten Mask R-CNN- und Detectron2-Modellen [7] [23] ausprobiert, jedoch für zu unpräzise befunden. Pixelgenaue Segmentierungen, die auch kleine und dünne verdeckende Objekte erkennen, könnten eine präzise Validierung ermöglichen.

Außerdem könnte man falsche Joint-Positionen über zu hohe Geschwindigkeiten detektieren. Weiterhin ist es denkbar, bestimmte Orte (Tisch, Roboter) zu definieren, wo ein Joint sich nicht befinden kann.

Zusätzlich lassen sich mehrere Tiefenkameras verwenden, um die Pose aus mehreren Perspektiven zu berechnen.

Weiterhin kann man Methoden implementieren, um Positionen invalider Joints über Extrapolation abzuschätzen. Simplere Ansätze könnten sich auf die Geschwindigkeit, die Beschleunigung und/oder den Ruck der Joints in aktuellen Frames stützen. Ein allgemeines Posenmodell würde präzisere Extrapolation erlauben.

Um die Korrektheit und Präzision der implementierten Algorithmen vernünftig verifizieren zu können, könnte man Joint-Ground-Truth-Daten mithilfe von Ganzkörperanzügen oder ähnlichen Technologien auslesen und zur statistischen Evaluation nutzen.

Letzteres bietet ebenfalls an, den Grad der klassischen Algorithmik zu verlassen und Machine Learning Ansätze auszuprobieren. So sollte eine Funktion gefunden werden, die die Joint-Ground-Truth abschätzt. Als Eingabe könnten sowohl rohe RGBD-Daten als auch mithilfe von OpenPose und der Tiefe berechnete Joint-Koordinaten dienen. Auch eine Kombination aus beiden ist denkbar. Aufgrund der Komplexität ist dies wahrscheinlich erst in größeren Forschungs- oder Industrieprojekten realisierbar.

8.2 Training und Deployment

Die Perception-Pipeline wurde im Hinblick auf das Training und Deployment entwickelt. Ein Mensch kann um den (inaktiven) Roboter laufen, seinen Arbeitsbereich schneiden und dies aufnehmen. Diese Daten könnten zum Training verwendet werden. Trainierten Roboter könnte die Pipeline als Wahrnehmung dienen, um ihre Fähigkeit in der Realität zu testen.

Literatur

- [1] Yang Chen u. a. »Knowledge-driven path planning for mobile robots: relative state tree«. In: *Soft Computing* 19 (2015), S. 763–773. DOI: 10.1007/s00500-014-1299-4 (siehe S. 3).
- [2] John Craig. *Introduction to Robotics Mechanics and Control, Global Edition*. Pearson Deutschland, 2021, S. 448. ISBN: 9781292164939. URL: <https://elibrary.pearson.de/book/99.150005/9781292164953> (siehe S. 8).
- [3] Yitao Ding und Ulrike Thomas. »Collision Avoidance with Proximity Servoing for Redundant Serial Robot Manipulators«. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, S. 10249–10255. DOI: 10.1109/ICRA40945.2020.9196759 (siehe S. 4).
- [4] Daniel Dugas u. a. »NavRep: Unsupervised Representations for Reinforcement Learning of Robot Navigation in Dynamic Human Environments«. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)* (2020), S. 7829–7835. DOI: 10.1109/ICRA48506.2021.9560951 (siehe S. 3).
- [5] Michael Everett, Yu Fan Chen und Jonathan P. How. »Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning«. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, S. 3052–3059. DOI: 10.1109/IROS.2018.8593871 (siehe S. 3).
- [6] *Forward and inverse kinematics*. 2017. URL: https://gramaziokohler.github.io/compas_fab/latest/examples/03_backends_ros/03_forward_and_inverse_kinematics.html (besucht am 27.12.2023) (siehe S. 10).
- [7] Kaiming He u. a. »Mask r-cnn«. In: *Proceedings of the IEEE international conference on computer vision*. 2017, S. 2961–2969 (siehe S. 63).
- [8] *Intel® RealSense™ Documentation*. URL: <https://dev.intelrealsense.com/docs/docs-get-started> (besucht am 27.12.2023) (siehe S. 33, 34).
- [9] *IR-DRL*. 2023. URL: <https://github.com/ignc-research/IR-DRL> (besucht am 02.01.2024) (siehe S. 2, 13, 35).
- [10] *IR-DRL-RGBD-to-3D-Pose*. 2024. URL: <https://github.com/ZalZarak/IR-DRL-RGBD-to-3D-Pose> (besucht am 19.01.2024) (siehe S. 35).

- [11] Linh Kästner u. a. »Arena-Bench: A Benchmarking Suite for Obstacle Avoidance Approaches in Highly Dynamic Environments«. In: *IEEE Robotics and Automation Letters* 7 (2022), S. 9477–9484. doi: 10.1109/LRA.2022.3190086 (siehe S. 3).
- [12] Linh Kästner u. a. »Interconnection between DRL-Based Local planners with Conventional Global Planners via waypoint generation«. In: () (siehe S. 4).
- [13] Peter Kopylov. *RGBD-to-3D-Pose*. 2023. URL: <https://github.com/ZalZarak/RGBD-to-3D-Pose> (besucht am 15.01.2024) (siehe S. 17, 32, 34, 42).
- [14] M. Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing, 2018 (siehe S. 6, 7).
- [15] Donghyun Lee u. a. »An improved artificial potential field method with a new point of attractive force for a mobile robot«. In: *2017 2nd International Conference on Robotics and Automation Engineering (ICRAE)* (2017), S. 63–67. doi: 10.1109/ICRAE.2017.8291354 (siehe S. 3).
- [16] KU Linh u. a. »All-in-one: A drl-based control switch combining state-of-the-art navigation planners«. In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, S. 2861–2867 (siehe S. 4).
- [17] M. Lucchi u. a. »robo-gym – An Open Source Toolkit for Distributed Deep Reinforcement Learning on Real and Simulated Robots«. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020), S. 5364–5371. doi: 10.1109/IROS45743.2020.9340956 (siehe S. 4).
- [18] Cristiano Premebida, Rares Ambrus und Zoltan-Csaba Marton. »Intelligent Robotic Perception Systems«. In: *Applications of Mobile Robots*. Hrsg. von Efren Gorrostieta Hurtado. Rijeka: IntechOpen, 2018. Kap. 6. doi: 10.5772/intechopen.79742. URL: <https://doi.org/10.5772/intechopen.79742> (siehe S. 10).
- [19] Mohammad Safeea, P. Neto und Richard Béarée. »On-line collision avoidance for collaborative robot manipulators by adjusting off-line generated paths: An industrial use case«. In: *ArXiv abs/1909.05159* (2019). doi: 10.1016/j.robot.2019.07.013 (siehe S. 4).
- [20] B. Sangiovanni u. a. »Deep Reinforcement Learning for Collision Avoidance of Robotic Manipulators«. In: *2018 European Control Conference (ECC)* (2018), S. 2063–2068. doi: 10.23919/ECC.2018.8550363 (siehe S. 4).

- [21] B. Sangiovanni u. a. »Self-Configuring Robot Path Planning With Obstacle Avoidance via Deep Reinforcement Learning«. In: *IEEE Control Systems Letters* 5 (2021), S. 397–402. DOI: 10.1109/LCSYS.2020.3002852 (siehe S. 4).
- [22] Kung-Ting Wei und Bingyin Ren. »A Method on Dynamic Path Planning for Robotic Manipulator Autonomous Obstacle Avoidance Based on an Improved RRT Algorithm«. In: *Sensors (Basel, Switzerland)* 18 (2018). doi: 10.3390/s18020571 (siehe S. 3).
- [23] Yuxin Wu u. a. *Detectron2*. <https://github.com/facebookresearch/detectron2>. 2019 (siehe S. 63).
- [24] Wang Xinyu u. a. »Bidirectional Potential Guided RRT* for Motion Planning«. In: *IEEE Access* 7 (2019), S. 95046–95057. DOI: 10.1109/ACCESS.2019.2928846 (siehe S. 3).
- [25] Xiaojun Zhu u. a. »Robot obstacle avoidance system using deep reinforcement learning«. In: *Ind. Robot* 49 (2021), S. 301–310. DOI: 10.1108/ir-06-2021-0127 (siehe S. 4).