



# I believe I can fly 3.0 - Integration von Kopfbewegungen und Realanwendung

## STUDIENARBEIT

für die Prüfung zum  
Bachelor of Science  
des Studienganges Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Aaron Mann**

Abgabedatum 17. Mai 2021

Bearbeitungszeitraum	25 Wochen
Matrikelnummer	9989768
Kurs	TINF18B5
Betreuer der DHBW	Prof. Dr. Marcus Strand

## **Erklärung**

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: „I believe I can fly 3.0 - Integration von Kopfbewegungen und Realanwendung“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

---

Ort      Datum

---

Unterschrift

## **Zusammenfassung**

Menschen wollen schon immer, was sie nicht haben oder nicht können. Zum Beispiel wollen Menschen schon immer Fliegen. Die Welt von oben betrachten und ein Gefühl der Leichtigkeit spüren.

Dies zu ermöglichen ist das Ziel einer Reihe von Studienarbeiten unter dem Titel „I believe I can fly“. Bisherige Studienarbeiten ermöglichen das Steuern einer Drohne mit Körpergesten. Das Kamerabild der Drohne wird in einer Videobrille gezeigt, sodass der Pilot die Umgebung aus der nach vorne gerichteten Perspektive der Drohne wahrnimmt. Allerdings kann der Drohnenführer dabei nicht umherschauen und zudem die Drohne nur in einer simulierten Umgebung steuern, sodass das Gefühl tatsächlich selbst zu fliegen, noch nicht erreicht wird.

Diese Studienarbeit soll beide Probleme lösen. Der Pilot soll den Blickwinkel aus der Drohnenperspektive durch die Neigung des Kopfes ändern können. Zudem soll er in der Lage sein reale Drohnen steuern zu können.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>11</b>
1.1 Problemstellung . . . . .	12
1.2 Zielsetzung . . . . .	12
1.3 Aufbau . . . . .	12
<b>2 Stand der Technik</b>	<b>13</b>
2.1 Keypunkteabgleich zwischen rekonstruierten 3D Gesichtsmodellen und 2D Bildern . . . . .	13
2.2 Neuronales Netz zur Bestimmung von 2D Gelenken und das anschließende Generierung eines 3D Modells . . . . .	14
2.3 Von OpenPose detektierte Keypunkte für die Bestimmung der Kopforientierung verwenden . . . . .	15
2.4 Verwendung von verschiedenen Keypunkten im Gesicht . . . . .	16
2.5 Verwendung einer IMU . . . . .	16
<b>3 Bestimmung der Kopfausrichtung</b>	<b>17</b>
<b>4 Steuerung der realen Drohne</b>	<b>23</b>
4.1 Parrot AR.Drone 2.0 . . . . .	23
4.1.1 SDK 2.0 . . . . .	23
4.1.2 Olympe . . . . .	24
4.1.3 ardrone_autonomy . . . . .	24
4.1.4 Eigene Implementierung . . . . .	24
4.2 Parrot Bebop Drone . . . . .	25
4.2.1 SDK 3.0 . . . . .	26
4.2.2 Olympe . . . . .	26
4.2.3 bebop_autonomy . . . . .	26

## *Inhaltsverzeichnis*

<b>5 Grundlagen</b>	<b>27</b>
5.1 Eulersche Winkel . . . . .	27
5.1.1 Eigentliche Eulerwinkel und Kardan-Winkel . . . . .	27
5.1.2 Gimbal Lock . . . . .	28
5.2 Quaternionen . . . . .	29
<b>6 Implementierung</b>	<b>31</b>
6.1 Werkzeuge . . . . .	31
6.2 Kameraausrichtung entsprechend der Kopforientierung des Piloten . . . . .	32
6.2.1 Verbindung und Konfiguration der IMU . . . . .	32
6.2.2 Datenabfrage von der IMU . . . . .	32
6.2.3 Kalibrierungswerte speichern . . . . .	33
6.2.4 Berechnung der Kopforientierung . . . . .	34
6.2.5 Messaging der Kopforientierung . . . . .	35
6.2.6 Änderung der Kameraausrichtung in der AirSim Simulation . . . . .	35
6.3 Steuerung einer realen Drohne . . . . .	36
<b>7 Evaluation</b>	<b>37</b>
7.1 Laufzeitevaluation . . . . .	37
7.2 Benutzerausgabe . . . . .	38
7.2.1 Vor der Kalibrierung . . . . .	39
7.2.2 Nach der Kalibrierung . . . . .	39
<b>8 Fazit</b>	<b>45</b>
<b>Anhang</b>	<b>46</b>
<b>Index</b>	<b>46</b>
<b>Literaturverzeichnis</b>	<b>46</b>

# Abbildungsverzeichnis

2.1	Ablauf des Keypunkteabgleichs zwischen dem rekonstruierten 3D Gesichtsmodell und dem 2D Bild [4] . . . . .	14
2.2	Bestimmen der 2D Gelenke und erstellen eines passenden 3D Modells [6]	14
2.3	Beispiele von Detektionen verschiedener Kopfausrichtungen mit OpenPose [9]	15
3.1	Detektion der Gesichts-Keypunkte mit OpenPose ohne Videobrille – Kopfausrichtung von rechts nach Links . . . . .	18
3.2	Detektion der Gesichts-Keypunkte mit OpenPose ohne Videobrille – Kopfausrichtung von unten nach oben . . . . .	18
3.3	Detektion der Gesichts-Keypunkte mit OpenPose ohne Videobrille – Kopfausrichtung schräg nach unten und oben . . . . .	18
3.4	Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille – Kopfausrichtung von rechts nach Links . . . . .	19
3.5	Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille – Kopfausrichtung von unten nach oben . . . . .	19
3.6	Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille – Kopfausrichtung schräg nach unten und oben . . . . .	19
3.7	Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille und aufgeklebtem Gesicht – Kopfausrichtung von rechts nach Links . . . . .	20
3.8	Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille und aufgeklebtem Gesicht – Kopfausrichtung von unten nach oben . . . . .	20
3.9	Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille und aufgeklebtem Gesicht – Kopfausrichtung schräg nach unten und oben . . . . .	21
3.10	Netzwerkarchitektur OpenPose [14] . . . . .	22
5.1	Parrot AR.Drone 2.0 mit eingezeichneten roll, pitch und yaw Achsen [22]	28
5.2	Gimbal Lock – Ausführungsreihenfolge: roll, pitch, yaw [23] . . . . .	29

## *Abbildungsverzeichnis*

6.1	ROS Node Architektur [3]	36
7.1	Beispielhafte Benutzerausgabe	38
7.2	IMU ist oben an der Videobrille befestigt	38
7.3	Kein Einfluss auf die Kameraausrichtung vor der Kalibrierung	39
7.4	Einfluss auf die Kameraausrichtung nach der Kalibrierung	40
7.5	Steuerung der Kameraausrichtung während dem Flug	41
7.6	Unstimmigkeiten bei der Steuerung der Kameraausrichtung nach einer bestimmten Flugzeit	42
7.7	Kameraausrichtung entspricht nicht der Kopforientierung	43

# **Tabellenverzeichnis**

2.1	OpenPose Keypunkte zur Klassifikation der Kopforientierung . . . . .	15
7.1	Laufzeiten der einzelnen Funktionen . . . . .	37
7.2	Fehlerhafte IMU Daten . . . . .	43
7.3	Resultierende Kopfausrichtung durch die fehlerhaften IMU Daten . . . . .	43

# **Formelverzeichnis**

(5.1) Aufbau eines Quaternion . . . . .	29
(5.2) Regeln des imaginären Teils eines Quaternion . . . . .	30
(5.3) Einheitsquaternion . . . . .	30
(5.4) Einheitsquaternion zur Darstellung einer Rotation um eine beliebige Achse $n$ um einen Winkel $\alpha$ . . . . .	30
(6.1) Konvertierung von Quaternion zu Eulerwinkeln . . . . .	33
(6.2) Differenz von zwei Eulerwinkeln . . . . .	34
(6.3) Konvertierung von Eulerwinkeln zu Quaternion . . . . .	35

# Abkürzungsverzeichnis

<b>NUI</b>	Natural User Interface . . . . .	12
<b>IMU</b>	Inertiale Messeinheit (engl. Intertial Measurement Unit) . . . . .	13
<b>CNN</b>	Convolutional Neuronal Network . . . . .	13
<b>DLT</b>	Direkte Lineare Transformation . . . . .	16
<b>GPUs</b>	Grafikprozessoren (engl. Graphics Processing Units) . . . . .	21
<b>SDKs</b>	Software Development Kits . . . . .	23
<b>ROS</b>	Robot Operating System . . . . .	23
<b>APIs</b>	Programmierschnittstellen (engl. Application Programming Interface) . .	23
<b>PIP</b>	Bild in Bild (engl. Picture In Picture) . . . . .	24
<b>FOV</b>	Field Of View . . . . .	26
<b>CPU</b>	Prozessor (engl. Central Processing Unit) . . . . .	31
<b>RAM</b>	Random-Access Memory . . . . .	31
<b>IDE</b>	integrierte Entwicklungsumgebung (engl. Integrated Development Environment) . . . . .	31
<b>OS</b>	Betriebssystem (engl. Operating System) . . . . .	31
<b>MAC-Adresse</b>	Media-Access-Control-Adresse . . . . .	32
<b>dps</b>	Grad pro Sekunde (engl. Degree Per Second) . . . . .	32
<b>FPS</b>	Bilder pro Sekunde (engl. Frames Per Second) . . . . .	37

# 1 Einleitung

Ziel einer Reihe von Studienarbeiten unter dem Titel „I believe I can fly“, zu welcher auch diese Studienarbeit zählt, ist es, dem Menschen durch Technik das Gefühl zu vermitteln, selbst zu fliegen. Dabei wird eine Drohne mit verschiedenen Körpergesten gesteuert und das Livebild der Drohnenkamera dem Piloten der Drohne in einer Videobrille angezeigt. Der Einsatzzweck kann dabei verschieden sein. Von Unterhaltung bis zu Arbeitseinsätzen ist dabei keine Grenze gesetzt. Wichtig ist dabei nur, dass die Drohne zuverlässig und präzise gesteuert werden kann und der Pilot ein realistisches Gefühl bekommt zu Fliegen. Die erste Studienarbeit dieser Reihe von 2015 nutzte einen Sensor der Xbox Kinect, um die Bewegungen des Benutzers mit einem 3D-fähigen Infrarot-Sensors zu erfassen [1]. Diese Bewegungen wurden dann in Steuerungsbefehle der Drohne umgerechnet und ermöglichten die Steuerung einer realen Drohne.

Bei der darauf folgenden Studienarbeit ging es um die Entwicklung eines intelligenten Assistenzsystems, zur Navigation durch Engstellen [2]. Der Pilot sollte dabei beim Fliegen durch Engstellen unterstützt werden. Allerdings kam diese Arbeit zu keinem verwendbaren Ergebnis.

Bei der Studienarbeit vom Jahr 2019 mit dem Titel „I believe I can fly 2.0“ wurde, im Vergleich zur ersten Studienarbeit dieser Reihe, der Fokus auf einen anderen Ansatz gelegt [3]. Da die Produktion des verwendeten Kinect-Sensors eingestellt wurde und somit nicht für jedermann zur Verfügung steht, entschied man sich die Gestenerkennung in zweidimensionalen Farbbildern zu realisieren. Dies ermöglichte den mobilen Einsatz von „I believe I can fly“, da die meisten Laptops eine Kamera besitzen. Um auch bei schlechten Wetterbedingungen das Fliegen zu ermöglichen wurde ein Simulator erstellt, in welchem eine simulierte Drohne gesteuert wird. Geplant war auch die Steuerung einer realen Drohne, allerdings konnte dies aus Zeitgründen nicht realisiert werden.

## 1.1 Problemstellung

Die Studienarbeit „I believe I can fly 2.0“ ermöglicht bereits das Steuern einer Drohne in einer simulierten Umgebung mit Hilfe eines Natural User Interface (NUI). Dabei bekommt der Pilot der Drohne in einer Videobrille das Livebild der Drohnenkamera, welche, von der Position der Drohne aus gesehen, gerade aus nach vorne gerichtet ist, zu sehen. Da der Pilot allerdings nicht durch die Neigung des Kopfes die Umgebung betrachten kann und die Drohne nur in einer simulierten Umgebung gesteuert wird, wird das Gefühl selbst zu fliegen noch nicht komplett erreicht. Dies soll in dieser Studienarbeit gelöst werden.

## 1.2 Zielsetzung

Ziel dieser Arbeit ist es die Kamera der Drohne im Simulator entsprechend der Kopfbewegung des Piloten auszurichten. Zudem soll diese Arbeit das Steuern einer echten Drohne (Parrot AR Drone 2.0 und Parrot Bebop Drone) mit Hilfe der Gesten des Piloten ermöglichen. Bei der Steuerung der Parrot Bebop Drohne soll der, in der Videobrille gezeigte, Kamera-Bildausschnitt der Fisheye-Kamera so ausgewählt werden, dass er zu der Kopfbewegung des Piloten passt.

## 1.3 Aufbau

Kapitel 2 erläutert verschiedene Ansätze, welche zur Bestimmung der Kopforientierung verwendet werden können. In Kapitel 3 werden die zuvor genannten Ansätze hinsichtlich ihrer Eignung für dieses Projekt bewertet und anschließend ein Ansatz ausgewählt, welcher implementiert werden soll. Möglichkeiten zur Steuerung einer realen Drohne werden in Kapitel 4 beschrieben. Dabei werden verschiedene Bibliotheken und Schnittstellen für das Steuern der Parrot AR.Drone 2.0 und der Parrot Bebop Drohne vorgestellt. Kapitel 5 deckt verschiedene Grundlagen im Bereich der dreidimensionalen Rotation ab. Dabei wird auf Eulersche Winkel und Quaternion eingegangen. Die Implementierung dieser Arbeit wird in Kapitel 6 ausführlich dargelegt. Es werden verwendete Werkzeuge und die unterschiedlichen implementierten Methoden erläutert. In Kapitel 7 wird diese Arbeit evaluiert. Es wird sowohl die Laufzeit analysiert, als auch die Benutzerausgabe qualitativ bewertet. Eine kurze Zusammenfassung und einen Ausblick auf mögliche weitere Tätigkeiten dieser Reihe von Studienarbeiten folgen in Kapitel 8.

## 2 Stand der Technik

In diesem Kapitel werden verschiedene Ansätze zur Bestimmung der Kopfausrichtung beschrieben. Dazu zählt ein Keypunkteabgleich zwischen rekonstruierten 3D Gesichtsmoellen und 2D Bildern, die Verwendung eines Neuronalen Netzes zur Detektion von 2D Gelenken und das anschließende Bestimmen eines 3D Models, die Bestimmung der Kopforientierung mit OpenPose-detektierten Keypunkten, das Verwenden von verschiedenen Keypunkten im Gesicht zur Berechnung der Kopforientierung oder auch die Verwendung von Sensoren, wie beispielsweise einer Inertiale Messeinheit (engl. Intertial Measurement Unit) (IMU), zum Bestimmen der Kopforientierung.

### 2.1 Keypunkteabgleich zwischen rekonstruierten 3D Gesichtsmoellen und 2D Bildern

Bei diesem Ansatz werden Keypunkte zwischen einem rekonstruierten 3D Gesichtsmoell und den 2D Eingangsbildern abgeglichen, um die Kopforientierung zu bestimmen [4]. Dabei gibt es 2 Phasen: In der ersten Phase wird ein personalisiertes dreidimensionales Gesichtsmoell anhand des 2D Bildes mit einem Convolutional Neuronal Network (CNN) erstellt. In der zweiten Phase wird ein iterativer Optimierungsalgorithmus verwendet, um die Keypunkte zwischen dem rekonstruierten 3D Gesichtsmoell und dem 2D Bild effektiv unter der Einschränkung der perspektivischen Transformation zu vergleichen (siehe Abbildung 2.1).

## 2 Stand der Technik

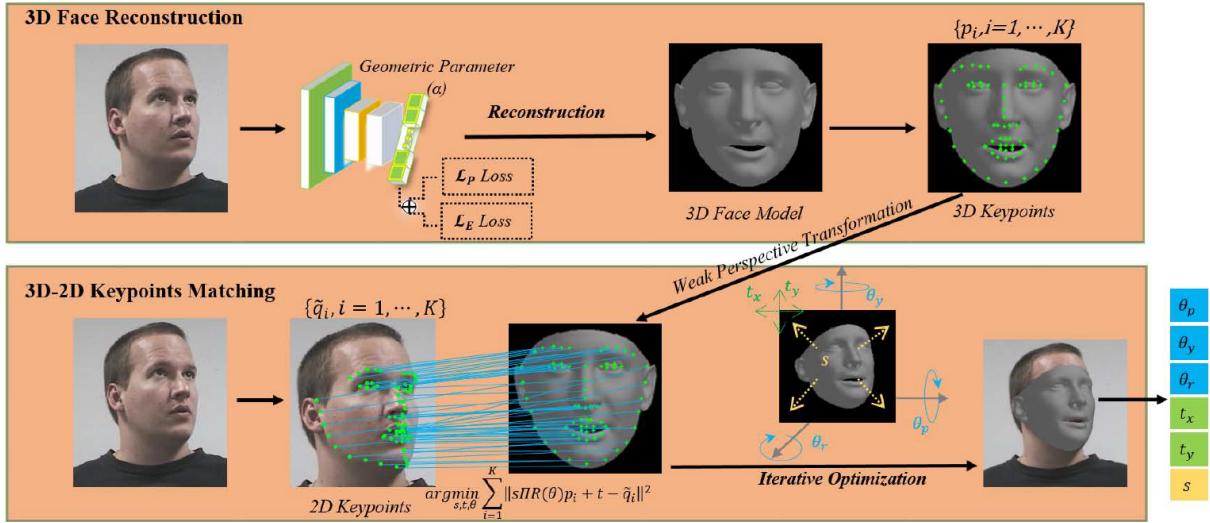


Abbildung 2.1: Ablauf des Keypunkteabgleichs zwischen dem rekonstruierten 3D Gesichtsmodell und dem 2D Bild [4]

## 2.2 Neuronales Netz zur Bestimmung von 2D Gelenken und das anschließende Generierung eines 3D Modells

Dieser Ansatz ist dem vorher beschriebenen Ansatz ähnlich. Das CNN DeepCut wird genutzt, um 2D Gelenke zu schätzen [5] [6]. Anschließend wird in einem zweiten Schritt ein 3D generatives Modell namens SMPL genutzt, um ein 3D Modell eines Menschen passend zu den 2D Gelenken zu generieren [7]. SMPL wurde mit tausenden 3D Scans von Menschen trainiert, sodass es so viele Informationen über die Form des menschlichen Körpers enthält, wodurch nur wenig Daten nötig sind, um so ein Modell entsprechend der Daten anzupassen. In Abbildung 2.2 ist ein Beispielergebnis dieses Ansatzes zu sehen.



Abbildung 2.2: Bestimmen der 2D Gelenke und erstellen eines passenden 3D Modells [6]

## 2.3 Von OpenPose detektierte Keypunkte für die Bestimmung der Kopforientierung verwenden

OpenPose ist eine Bibliothek für die Detektion von Körper, Gesicht, Händen und Füßen mehrerer Personen in Echtzeit [8]. Dabei werden verschiedene Keypunkte, beispielsweise an den Gelenken von Armen und Beinen, oder auch die Augen oder Ohren, detektiert. Die Keypunkte können verwendet werden, um die Kopforientierung der detektierten Personen in acht verschiedene Richtungen zu klassifizieren (vgl. Abbildung 2.3) [9].

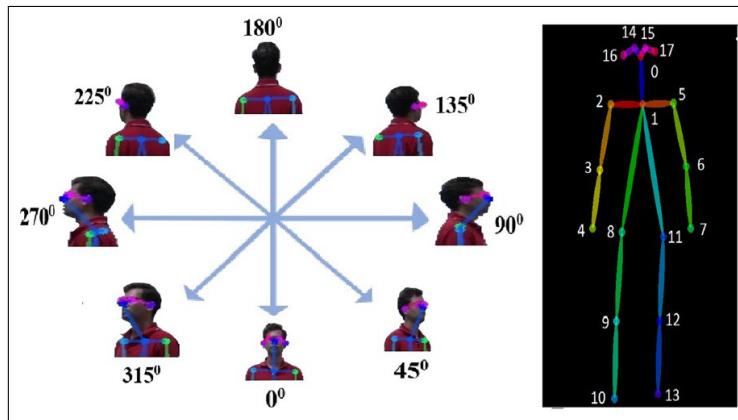


Abbildung 2.3: Beispiele von Detektionen verschiedener Kopfausrichtungen mit OpenPose [9]

Für die Klassifizierung in die acht verschiedenen Orientierung werden die Keypunkte aus Tabelle 2.1 verwendet.

Kopforientierung	Detektierte Gesichts-Keypunkte
0°	[0, 14, 15, 16, 17]
45°	[0, 14, 15, 16]
90°	[0, 14, 16]
135°	[14, 16]
180°	[Keine Keypunkte detektiert]
225°	[15, 17]
270°	[0, 15, 17]
315°	[0, 14, 15, 17]

Tabelle 2.1: OpenPose Keypunkte zur Klassifikation der Kopforientierung

## **2.4 Verwendung von verschiedenen Keypunkten im Gesicht**

Zur Berechnung der Kopforientierung können auch verschiedene Keypunkte im Gesicht verwendet werden [10]. Dabei eignen sich die äußereren Ecken der Augen, die Nasenspitze, die Ecken des Mundes, oder das Kinn. Diese Punkte können mit Dlib, einem C++ Toolkit mit Algorithmen aus dem Maschinellen Lernen, oder auch OpenPose gewonnen werden [11][8]. Mit diesen Punkten, deren 3D Lokationen, und den intrinsischen Parametern der Kamera, wie Brennweite, das optische Zentrum im Bild und die radialen Verzerrungsparameter kann die Kopforientierung berechnet werden. Die 3D Lokationen der Keypunkte im Gesicht können mit einem 3D Modell, idealerweise, aber nicht notwendiger weise, von der Person auf dem Bild, bestimmt, oder definiert werden. Für die eigentliche Posenschätzung wird die Direkte Lineare Transformation (DLT) verwendet mit einer anschließenden Levenberg-Marquardt Optimierung.

## **2.5 Verwendung einer IMU**

Eine Internale Messeinheit (engl. Intertial Measurement Unit) besteht aus verschiedenen Trägheitssensoren, wie Beschleunigungssensoren und Geschwindigkeitsgyroskope, um die lineare Beschleunigung und die Winkelgeschwindigkeit zu messen [12]. Damit ist es möglich die Position und Orientierung der IMU zu bestimmen. Für diese Arbeit kann eine IMU an der Videobrille befestigt werden, um die Kopforientierung zu tracken.

# 3 Bestimmung der Kopfausrichtung

Die in Kapitel 2 aufgezeigten Ansätze eignen sich nicht alle für dieses Projekt.

Der Keypunkteabgleich zwischen einem rekonstruierten 3D Gesichtsmodell und dem 2D Bild, welcher in Abschnitt 2.1 beschrieben wurde, ist für diese Arbeit nicht geeignet, da der Nutzer eine Videobrille aufgezogen hat, wodurch die Rekonstruktion des 3D Gesichtsmodells nicht möglich ist.

Auch die in Abschnitt 2.2 erläuterte Generierung eines 3D Modells entsprechend detektierten 2D Gelenken kommt für dieses Projekt nicht in Frage, da die Kopfausrichtung bei diesem Ansatz nicht immer gut erkannt wurde. Zudem könnte das CNN bei der Posenbestimmung an seine Grenzen stoßen, da der Nutzer eine Videobrille trägt.

Der in Abschnitt 2.3 dargelegte Ansatz kann die Kopforientierung in acht verschiedene horizontale Ausrichtungen klassifizieren. Da der Nutzer allerdings nicht nur in diese acht Richtungen schaut, den Kopf auch nicht nur horizontal bewegt, sondern auch vertikal, und die Implementierung der Kameraausrichtung entsprechend der Kopforientierung das Ziel hat, dem Nutzer noch mehr das Gefühl zu geben tatsächlich selbst zu fliegen, ist auch dieser Ansatz für diese Arbeit ungeeignet.

Ein Ansatz, welcher für dieses Projekt verwendet werden könnte, nutzt verschiedene Keypunkte im Gesicht zur Berechnung der Kopfausrichtung und wird in Abschnitt 2.4 aufgezeigt. Da der Nutzer allerdings eine Videobrille trägt, können Keypunkte, wie die äußeren Ecken der Augen oder die Nasenspitze, nicht detektiert werden. Um zu überprüfen, ob OpenPose genug andere Keypunkte im Gesicht bei allen möglichen Kopfausrichtungen erkennt, wurden verschiedene Tests durchgeführt.

Zu Beginn wurde getestet, ob OpenPose genug Keypunkte im Gesicht detektiert, wenn der Nutzer keine Videobrille trägt. Wie in Abbildung 3.1 zu sehen ist, werden sowohl bei einer Kopfausrichtung nach rechts, als auch nach links genug Keypunkte erkannt. Auch beim nach unten und oben Schauen (vgl. Abbildung 3.2), sowie beim schräg nach unten und oben Schauen (vgl. Abbildung 3.3), werden genug Keypunkte ermittelt, wobei beim

### 3 Bestimmung der Kopfausrichtung

nach oben Schauen der Keypunkt am Kinn nach unten versetzt ist (siehe Abbildung 3.2c).



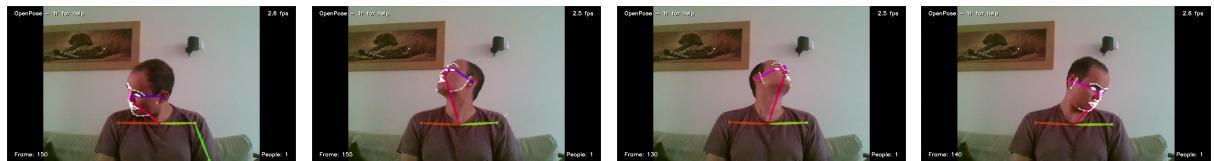
(a) Kopfausrichtung nach rechts    (b) Kopfausrichtung gerade aus    (c) Kopfausrichtung nach links

Abbildung 3.1: Detektion der Gesichts-Keypunkte mit OpenPose ohne Videobrille – Kopfausrichtung von rechts nach Links



(a) Kopfausrichtung nach unten    (b) Kopfausrichtung gerade aus    (c) Kopfausrichtung nach oben

Abbildung 3.2: Detektion der Gesichts-Keypunkte mit OpenPose ohne Videobrille – Kopfausrichtung von unten nach oben



(a) Kopfausrichtung nach rechts unten    (b) Kopfausrichtung nach rechts oben    (c) Kopfausrichtung nach links oben    (d) Kopfausrichtung nach links unten

Abbildung 3.3: Detektion der Gesichts-Keypunkte mit OpenPose ohne Videobrille – Kopfausrichtung schräg nach unten und oben

Mit der Videobrille sind die Ergebnisse leider nicht so gut. Beim gerade nach vorne schauen werden noch genug Keypunkte im Gesicht detektiert (vgl. Abbildung 3.4b). Allerdings werden, wie in Abbildung 3.4 zu sehen ist, schon bei leichtem nach rechts und links Schauen keine oder zu wenig Keypunkte erkannt. Des Weiteren werden auch bei leichtem nach unten und oben Schauen keine Keypunkte mehr ermittelt (vlg. Abbildung 3.5).

### 3 Bestimmung der Kopfausrichtung

Abbildung 3.6 zeigt, dass dementsprechend auch keine Keypunkte erkannt werden, wenn der Kopf schräg nach unten oder oben ausgerichtet ist.



Abbildung 3.4: Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille – Kopfausrichtung von rechts nach Links



Abbildung 3.5: Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille – Kopfausrichtung von unten nach oben



Abbildung 3.6: Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille – Kopfausrichtung schräg nach unten und oben

Mit Hilfe eines an die Videobrille geklebten Gesichtes wurde versucht, das Problem bei der Detektion der Keypunkte, wenn der Nutzer eine Videobrille trägt, zu umgehen. Wie in Abbildung 3.7 zu sehen ist, wird beim gerade aus Schauen (vgl. Abbildung 3.7b) nicht das Linke Ohr des Bildes, sondern das des Nutzers, als Ohr erkannt, wodurch in diesem Fall

### 3 Bestimmung der Kopfausrichtung

eine Berechnung der Kopfausrichtung mit Hilfe der Augen, Ohren und Nase nicht in Frage kommt. Beim nach links Schauen (vgl. Abbildung 3.7c) wird die Nase im aufgeklebten Gesichtsfoto viel zu hoch detektiert, wodurch auch hier eine Berechnung mit den Augen und der Nase nicht funktioniert. Abbildung 3.8 zeigt, dass beim runter und hoch Schauen meistens die „Gelenke“ verwendet werden können. Allerdings werden diese unpräzise erkannt und manchmal werden dabei auch Gelenke des Piloten und nicht Gelenke des Fotos detektiert (siehe Abbildung 3.8d), weshalb auch diese für die Berechnung nicht geeignet sind. Ebenso sieht das Ganze bei der Detektion der Bilder aus, auf welchen der Nutzer schräg nach oben und unten schaut (vgl. Abbildung 3.9). Auch hier werden „Gelenke“ des Nutzers unpräzise erkannt (vgl. Abbildung 3.9a 3.9b 3.9c), oder die Nase zu hoch detektiert (vgl. Abbildung 3.9d).



Abbildung 3.7: Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille und aufgeklebtem Gesicht – Kopfausrichtung von rechts nach Links

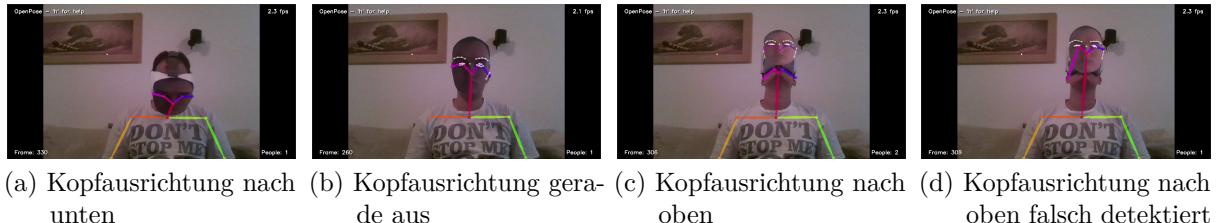


Abbildung 3.8: Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille und aufgeklebtem Gesicht – Kopfausrichtung von unten nach oben

### 3 Bestimmung der Kopfausrichtung



(a) Kopfausrichtung nach rechts unten (b) Kopfausrichtung nach rechts oben (c) Kopfausrichtung nach links oben (d) Kopfausrichtung nach links unten

Abbildung 3.9: Detektion der Gesichts-Keypunkte mit OpenPose mit Videobrille und aufgeklebtem Gesicht – Kopfausrichtung schräg nach unten und oben

Auch dieser Ansatz ist deshalb nicht für dieses Projekt geeignet.

Damit die Kameraausrichtung entsprechend der Kopfbewegung des Piloten gesetzt werden kann, bleiben zwei Optionen übrig. Zum Einen könnte OpenPose nachtrainiert werden, sodass es auch Personen mit einer Videobrille zuverlässig detektiert. Zum Anderen könnte eine IMU verwendet werden, um die Kopforientierung zu bestimmen.

Um OpenPose nachzutrainieren liefert OpenPose selbst einige Scripts und eine Anleitung [13]. Dabei wird das Training in zwei Sektionen geteilt: das „Body-Training“, welches genutzt wird, um das COCO Body-Modell zu trainieren und das „Whole-Body-Training“, um das whole-body-Modell zu trainieren. Wenn die Scripte beider Sektionen kombiniert werden, sind alle Arten von Training möglich (bspw. Körper und Hände, nur das Gesicht, etc.). OpenPose liefert dabei allerdings nur Beispiele für das Training der Körperfunktion mit dem COCO Datensatz, für das Training des Körpers und der Füße, sowie für das Training des gesamten Körpers. Es wird empfohlen das Training auf einer Maschine mit 4 Grafikprozessoren (engl. Graphics Processing Units) (GPUs) und einer Satzgröße („Batch Size“) von 10 mit 800000 Iterationen durchzuführen.

Ein Mitentwickler von OpenPose stellt eine weitere Anleitung zum Nachtrainieren von OpenPose, weitere Scripts, die Netzwerkarchitektur von OpenPose, sowie unterschiedliche Re-Implementierungen in verschiedenen Frameworks zur Verfügung [14]. Die Netzwerkarchitektur von OpenPose wird in Abbildung 3.10 gezeigt.

### 3 Bestimmung der Kopfausrichtung

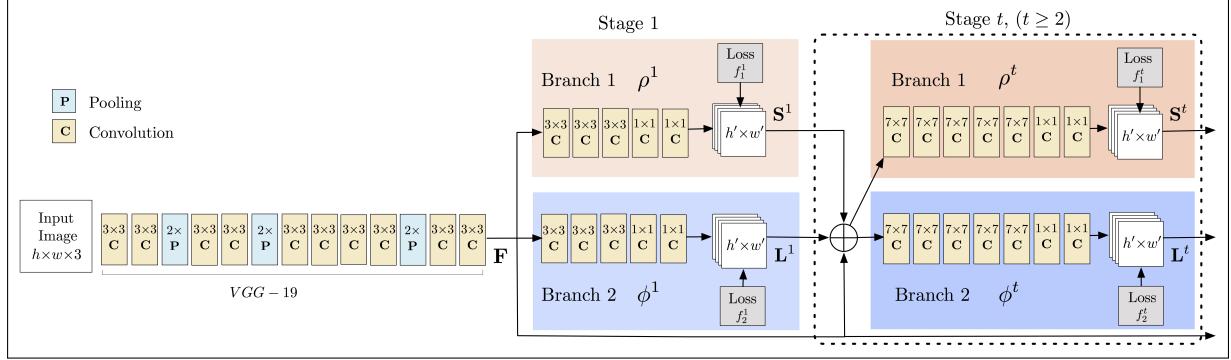


Abbildung 3.10: Netzwerkarchitektur OpenPose [14]

Für das Training werden die verschiedenen Datensätze benötigt, welche auch beim Training von OpenPose benutzt wurden. Des Weiteren werden viele annotierte Bilddaten benötigt, auf denen Personen zu sehen sind, welche eine Videobrille tragen.

Da die Generierung eines solchen Datensatzes viel Zeit in Anspruch nimmt, wurde sich dagegen entschieden OpenPose nachzutrainieren und stattdessen beschlossen eine IMU für die Bestimmung der Kopfausrichtung zu verwenden.

# 4 Steuerung der realen Drohne

Dieses Kapitel stellt verschiedene Möglichkeiten zur Steuerung der Parrot AR.Drone 2.0 und der Parrot Bebop Drone vor. Es werden von Parrot bereitgestellte Software Development Kits (SDKs) und Schnittstellen, aber auch von anderen entwickelte Lösungen erläutert.

## 4.1 Parrot AR.Drone 2.0

Die Parrot AR.Drone 2.0 kann mit der von Parrot gelieferten SDK 2.0 gesteuert werden. Zudem wird eine Python Controller Schnittstelle namens „Olympe“ bereitgestellt. Des Weiteren gibt es eine weit verbreitete Schnittstelle für Robot Operating System (ROS) mit dem Namen „ardrone\_autonomy“, um AR.Drone 1.0 und 2.0 zu steuern. Es ist auch möglich die Implementierung komplett selbst zu schreiben.

### 4.1.1 SDK 2.0

Das SDK 2.0 liefert drei Bibliotheken, um mit der Parrot AR.Drone 2.0 kommunizieren und sie steuern zu können [15]. Zudem beinhaltet es Codebeispiele für die Kontrolle der Drohne von einem Linux-Computer.

Als Bibliothek wird die AR.Drone 2.0 Library (ARDroneLIB) bereitgestellt, welche Programmierschnittstellen (engl. Application Programming Interface) (APIs) zur Verfügung stellt, um einfach mit einem AR.Drone 2.0 Produkt zu kommunizieren und es zu konfigurieren.

Außerdem behinhaltet das SDK die AR.Drone 2.0 Tool (ARDroneTool) Bibliothek. Diese liefert einen vollfunktionalen Drohnen-Client, bei dem Entwickler nur noch ihren eigenen anwendungsspezifischen Code einfügen müssen.

Die dritte Bibliothek, welche von der SDK 2.0 zur Verfügung gestellt wird, ist die AR.Drone 2.0 Control Engine Bibliothek. Diese liefert ein intuitives Kontrollinterface, um AR.Drone

## *4 Steuerung der realen Drohne*

2.0 Produkte von iOS Geräten zu kontrollieren.

Die Bibliotheken sind alle in C programmiert.

### **4.1.2 Olympe**

Olympe liefert eine Python Controller Programmierschnittstelle für Parrot Drohnen. Dadurch kann Olympe genutzt werden, um einen Computer mit der Drohne zu verbinden und sie zu kontrollieren. Primär ist Olympe dazu gedacht simulierte Drohnen zu steuern, indem Sphinx (der Parrot Drohnen Simulator) genutzt wird. Allerdings kann Olympe auch dazu verwendet werden reale Drohnen zu steuern.

Um Olympe Scripte zu starten, muss eine entsprechende Python-Umgebung aufgesetzt werden. Dies erschwert die Nutzung von Olympe in einem ROS Programm.

### **4.1.3 ardrone\_autonomy**

Diese Schnittstelle ist ein ROS Treiber für die Parrot AR.Drone 1.0 und 2.0 und basiert auf der offiziellen AR.Drone SDK 2.0.1 von Parrot [16]. ardrone\_autonomy wurde von Mani Monajjemi entwickelt und ist gut dokumentiert. Mit diesem Treiber kann die Parrot AR.Drone 2.0 konfiguriert und gesteuert werden. Es können Updatefrequenzen eingestellt werden, die Navigationsdaten gelesen werden, die Drohnensensordaten abgefragt werden oder auch das Kamerabild gelesen werden. Die AR.Drone 2.0 besitzt eine Frontkamera und eine Bodenkamera. Da sie im Gegensatz zur AR.Drone 1.0 kein Bild in Bild (engl. Picture In Picture) (PIP) Feature unterstützt kann nur das Bild einer der beiden Kameras gleichzeitig abgefragt werden. Zudem können Kommandos an die AR.Drone geschickt werden, wie beispielsweise starten, landen, reset (Notfall-Stop) oder allgemeine Fluganweisungen.

### **4.1.4 Eigene Implementierung**

Die Steuerung und Konfigurierung der AR.Drone 2.0 kann auch komplett selbst implementiert werden. Notwendige Schritte dafür finden sich im Parrot Developer Guide [15]. Zu beachten ist hierbei vor allem, dass die AT-Kommandos (Steuerungsbefehle) alle 30ms gesendet werden, damit sich die Drohne gleichmäßig bewegt. Es muss mindestens ein Kommando in weniger als zwei Sekunden geschickt werden, sodass die Drohne weiterhin Steuerungsbefehle ausführt und nicht damit rechnet, dass die WIFI Verbindung getrennt wurde.

## *4 Steuerung der realen Drohne*

### **WIFI Netzwerk und Verbindung**

Die AR.Drone 2.0 erstellt ein WIFI Netzwerk, normalerweise mit dem Namen ardrone2\_xxx, und weißt sich selbst eine freie, ungerade IP-Adresse zu (standardmäßig 192.168.1.1). Der Nutzer muss seinen PC mit diesem Netzwerk verbinden, um die Drohne steuern zu können. Dazu fordert das Client-Gerät eine IP Adresse von dem Drohnen-DHCP-Server an. Der AR.Drone DHCP Server bewilligt dem Client eine IP Adresse, welche der Drohnen-IP-Adresse addiert um eine Zahl zwischen 1 und 4 entspricht. Anschließend kann der Nutzer mit seinem PC Anfragen an die AR.Drone IP-Adresse und ihren Service-Ports senden.

### **Kommunikationsservices**

Die AR.Drone kann mit drei Hauptkommunikationsservices kontrolliert werden.

Für die Kontrollierung und Konfiguration der Drohne werden AT Kommandos auf Port 5556 (UDP) gesendet. Normalerweise werden davon 30 Befehle pro Sekunde gesendet. Informationen über die Drohne wie Status, Position, Geschwindigkeit und Motorrotationsgeschwindigkeit, die sogenannten Navigationsdaten, werden von der Drohne zum Client auf Port 5554 (UDP) gesendet. Im Demo-Modus werden diese Daten etwa 15 mal pro Sekunde gesendet. Im Debug-Modus 200 mal pro Sekunde.

Der Videostream wird von der AR.Drone zum Client auf Port 5555 (TCP) übertragen. Ein vierter Kommunikationskanal, der sogenannte „control port“ kann auf Port 5559 (TCP) eingerichtet werden, um kritische Daten zu transferieren. Er wird genutzt um Konfigurationsdaten zu erhalten und um wichtige Informationen, wie beispielsweise das Senden von Konfigurationsinformationen, zu bestätigen.

## **4.2 Parrot Bebop Drone**

Für die Steuerung der Parrot Bebop Drohne liefert Parrot das SDK 3.0. Des Weiteren kann auch für diese Drohne die Python-Programmierschnittstelle Olympe eingesetzt werden. Darüber hinaus hat Mani Monajjemi auch für diese Parrot Drohne eine ROS Schnittstelle mit dem Namen „bebop\_autonomy“ entwickelt.

## *4 Steuerung der realen Drohne*

### **4.2.1 SDK 3.0**

Die SDK 3.0 von Parrot ermöglicht das Verbinden, Steuern, Erreichen des Streams, Speichern und Verbinden der Drohnenmedien, das Senden und Starten von Autopilot-Flugplänen sowie das Updaten der Drohne [17]. Dies funktioniert bei verschiedenen Parrot Drohnen, wie der Rolling Spider, Jumping Sumo oder auch der Bebop Drone 1.0 und 2.0. Das SDK ist hauptsächlich in C geschrieben und liefert Bibliotheken für Unix-Systeme, Android Geräte und iOS Geräte.

Um das SDK zu verwenden müssen die Bibliotheken dem Projekt hinzugefügt werden. Mehrere Codebeispiele für Unix erleichtern den Einstieg in die Anwendung des SDK.

### **4.2.2 Olympe**

Auch für die Steuerung der Parrot Bebop Drone kann Olympe verwendet werden. Mehr dazu in Unterabschnitt 4.1.2

### **4.2.3 bebop\_autonomy**

Dieser ROS Treiber für die Parrot Bebop 1.0 und 2.0 Drohnen basiert auf der offiziellen ARDroneSDK3 [18]. Er wurde von Mani Monajjemi entwickelt und wird von Sepehr MohaimenianPour, Thomas Bamford und Tobias Naegeli auf dem Laufenden gehalten. Der Treiber kann für verschiedene ROS eingesetzt werden, wie beispielsweise dem ROS Kinetic. Die Dokumentation ist sehr gut, wodurch ein Einstieg in diese Schnittstelle gut möglich ist. bebop\_autonomy kann verwendet werden um Steuerungsbefehle, wie starten, landen, Notfall-Stop oder allgemeine Steuerungsbefehle an die Drohne zu senden. Da die Bebop Drohne über eine Kamera mit Fischauge verfügt kann sogar die virtuelle Kamera innerhalb des Field Of View (FOV) bewegt werden. Dies ist horizontal bis zu ca. 80° und vertikal bis zu ca. 50° möglich. Zudem ist es möglich autonome Flugpläne an die Drohne zu schicken oder sie automatisch zum Startpunkt zurück kehren zu lassen. Des Weiteren können natürlich verschiedenen Daten von der Bebop Drohne gelesen werden. Dazu zählt das Kamerabild mit einer Auflösung von 640x368 Pixel bei einer Abfragefrequenz von 30 Hz, GPS Daten, Navigationsdaten und Daten von anderen Sensoren.

# 5 Grundlagen

In diesem Kapitel werden verschiedene Grundlagen im Bereich der dreidimensionalen Rotation abgedeckt. Dazu zählen die Eulerschen Winkel und die Quaternionen, welche benutzt werden um Orientierungen im 3D Raum zu beschreiben.

## 5.1 Eulersche Winkel

Mit den drei sogenannten Eulerwinkeln, lässt sich eine beliebige Drehung im 3D Raum beschreiben [19]. Diese drei Winkel bzw. Achsen ( $x, y, z$ ) stehen alle orthogonal zueinander. Um die Ausrichtung eines Objekt darzustellen, werden drei Drehungen benötigt.

### 5.1.1 Eigentliche Eulerwinkel und Kardan-Winkel

Bei den eigentlichen Eulerwinkeln wird ein Objekt bei der ersten und dritten Drehung um die gleiche Achse gedreht. Die zweite Drehung findet an einer anderen Achse statt. So ergeben sie die Möglichkeiten  $xyx, xzx, yxy, yzy, zxz$  und  $zyz$ .

Die Kardan-Winkel sind eine spezielle Form der Eulerwinkel [20]. Bei diesen erfolgen die drei Drehungen um die drei verschiedenen Achsen. Auch hier ergeben sich sechs verschiedene Kombinationen:  $xyz, xzy, yxz, yzx, zxy$  und  $zyx$ . Die  $x - \text{Achse}$  wird dabei als „roll“ (rollen) bezeichnet und wird eingesetzt, um das Objekt zu rollen. Die  $y - \text{Achse}$  wird genutzt, um das Objekt zu neigen, weshalb sie „pitch“ (nicken) genannt wird. Zum Schwenken des Objektes wird die  $z - \text{Achse}$ , welche „yaw“ (gieren, schwenken) genannt wird, verwendet. Die drei Winkel entsprechen dabei dem körperfesten Koordinatensystem [21]. Das bedeutet, dass die Achsen entsprechend dem Objekt oder Körper ausgerichtet sind. Die positive  $x - \text{Achse}$  (roll) zeigt dabei nach vorne, die positive  $y - \text{Achse}$  (pitch) nach links und die positive  $z - \text{Achse}$  (yaw) nach oben (vgl. Abbildung 5.1). Dadurch, dass die Achsen entsprechend des Objektes ausgerichtet sind, spielt die Reihenfolge der

## 5 Grundlagen

Achsen, um welche die Drehung stattfindet, eine Rolle für die endgültige Ausrichtung des Objektes.

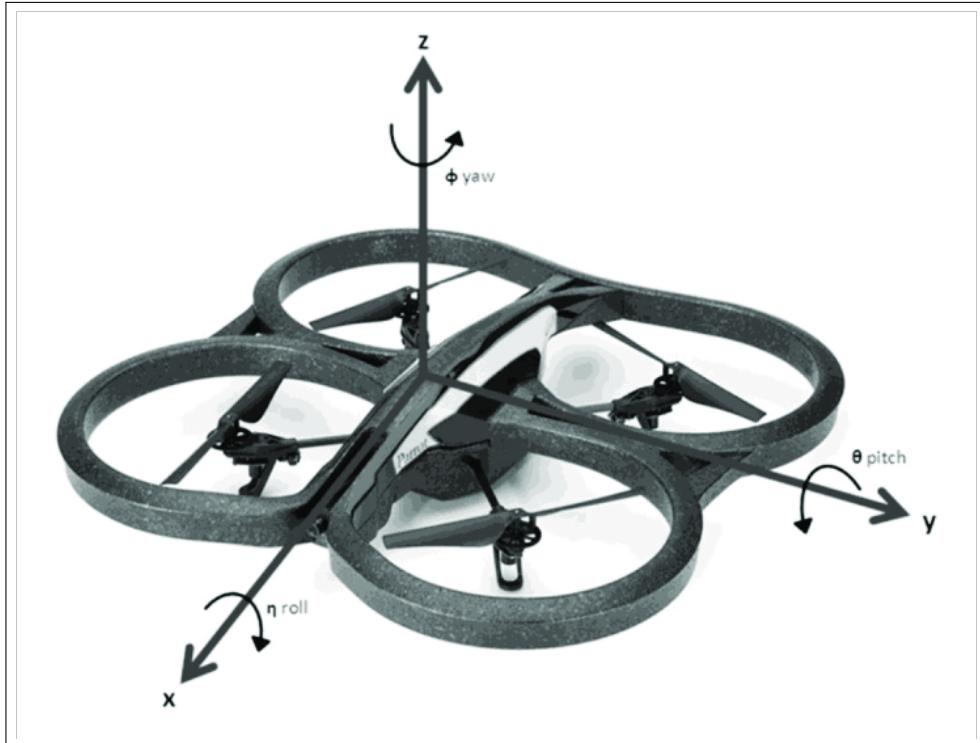


Abbildung 5.1: Parrot AR.Drone 2.0 mit eingezeichneten roll, pitch und yaw Achsen [22]

### 5.1.2 Gimbal Lock

Ein Problem der Eulerwinkel ist der sogenannte „Gimbal Lock“. Dieser tritt auf, wenn ein körperfestes Koordinatensystem verwendet wird und eine Drehung von 90 Grad um die  $y$  – Achse (pitch) durchgeführt wird. Denn in diesem Fall geht eine Drehrichtung verloren, wenn zuvor um die  $x$  – Achse (roll) gedreht wird und nach der Drehung um die  $y$  – Achse auch noch um die  $z$  – Achse gedreht wird. Werden diese Drehungen in einem raumfesten Koordinatensystem betrachtet, so drehen sich die Drehungen um die körperfeste  $x$  – Achse und die körperfeste  $z$  – Achse um die gleiche raumfeste Achse. Ein Beispiel davon wird in Abbildung 5.2 gezeigt. Die roll und yaw Drehung des Flugzeugs finden dabei beide um die raumfeste rote Achse statt.

Um dieses Problem zu umgehen, können Drehungen mit Quaternionen angegeben werden.

## 5 Grundlagen

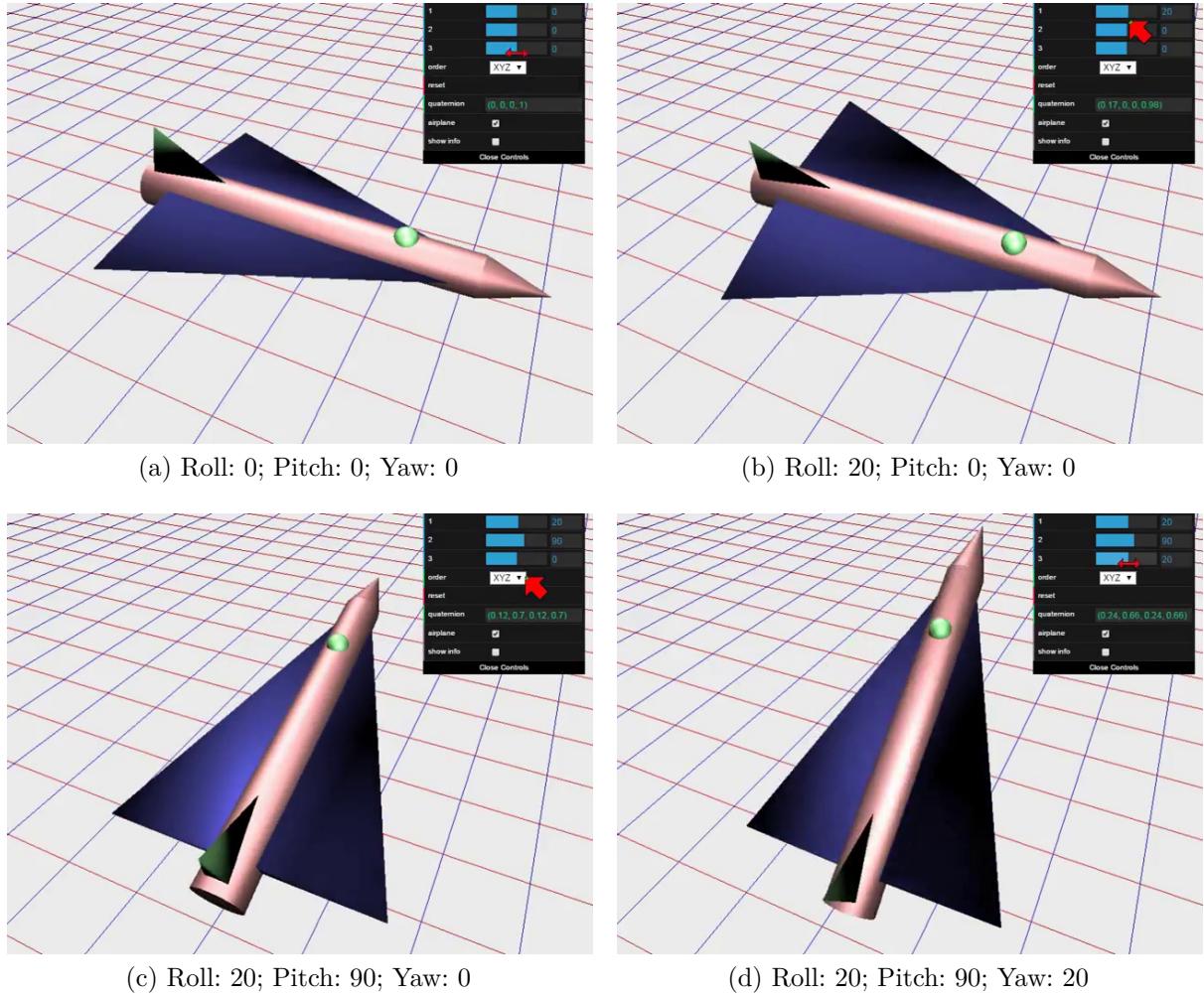


Abbildung 5.2: Gimbal Lock – Ausführungsreihenfolge: roll, pitch, yaw [23]

## 5.2 Quaternionen

Bei den Eulerwinkeln wurde ein Objekt um drei Achsen gedreht. Es ist jedoch möglich eine beliebige dreidimensionale Rotation als eine Drehung um genau eine Achse, der sogenannten Drehachse, zu beschreiben. Für diese Beschreibung können Quaternionen verwendet werden. Diese bestehen wie Komplexe Zahlen aus einem realen und einem imaginären Teil und sind, wie in Gleichung 5.1 zu sehen, aufgebaut [19].

$$q = w + i \cdot v_x + j \cdot v_y + k \cdot v_z \quad (5.1)$$

## 5 Grundlagen

Dabei sind  $w$ ,  $v_x$ ,  $v_y$  und  $v_z$  reelle Zahlen. Der imaginäre Teil besteht aus  $i$ ,  $j$  und  $k$  für welchen Gleichung 5.2 gilt.

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1 \quad (5.2)$$

Ist die Bedingung von Gleichung 5.3 erfüllt, so wird das Quaternion als Einheitsquaternion bezeichnet.

$$w^2 + v_x^2 + v_y^2 + v_z^2 = 1 \quad (5.3)$$

Zur Darstellung einer Rotation um eine beliebige Achse  $n$  (mit  $|n| = 1$ ) um einen Winkel  $\alpha$  kann ein Einheitsquaternion verwendet werden. Dieser wird dabei mit Gleichung 5.4 definiert [24].

$$q = \begin{pmatrix} w \\ v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} \cos\left(\frac{\alpha}{2}\right) \\ n_x \sin\left(\frac{\alpha}{2}\right) \\ n_y \sin\left(\frac{\alpha}{2}\right) \\ n_z \sin\left(\frac{\alpha}{2}\right) \end{pmatrix} = \left( \cos\left(\frac{\alpha}{2}\right), \left( \sin\left(\frac{\alpha}{2}\right) \right) n^T \right)^T \quad (5.4)$$

# 6 Implementierung

In diesem Kapitel wird die Implementierung dieses Projektes beschrieben. Zu Beginn werden die verwendetet Werkzeuge genannt. Anschließend wird die Implementierung der Steuerung der Kameraausrichtung entsprechend der Kopforientierung erläutert. Dabei werden alle Schritte von der Verbindung und Konfiguration der IMU bis hin zur Änderung der Kameraausrichtung in der AirSim Simulation aufgezeigt. Am Ende wird kurz auf die Implementierung der Steuerung einer realen Drohne eingegangen.

## 6.1 Werkzeuge

Verwendet wird ein Laptop der Firma MSI. Dieser hat eine Intel® Core™ i7-6700HQ Prozessor (engl. Central Processing Unit) (CPU) mit 4 Kernen, einer Grundtaktfrequenz von 2,60 GHz und einer maximalen Turbo-Taktfrequenz von 3,50 GHz [25]. Zudem sind 16GB Random-Access Memory (RAM) und eine GeForce GTX 1060 Grafikkarte verbaut. Als Betriebssystem (engl. Operating System) (OS) ist Ubuntu 16.04 installiert. Zudem wird ROS Kinetic verwendet. Für die Programmierung wird als integrierte Entwicklungsumgebung (engl. Integrated Development Environment) (IDE) PyCharm 2021.1 (Community Edition) genutzt. Programmiert wird in Python 3.8, allerdings verwendet ROS Kinetic Python 2.7. Das MetaMotion C Kit von mbientlab wird als IMU eingesetzt. Die von mbientlab bereitgestellte Bibliothek metawear wird für die Kommunikation mit der IMU verwendet.

## 6.2 Kameraausrichtung entsprechend der Kopforientierung des Piloten

In diesem Unterkapitel wird die Implementierung zur Steuerung der Kameraausrichtung entsprechend der Kopforientierung des Piloten erläutert. Dabei wird auf die Verbindung und Konfiguration der IMU (MetaMotionC Kit), die Datenabfrage von der IMU, das Speichern der Kalibrierungswerte, die Berechnung der Kopfausrichtung, das anschließende Messaging der Kopforientierung und auf das Ändern der Kameraausrichtung in der AirSim Simulation eingegangen.

### 6.2.1 Verbindung und Konfiguration der IMU

Das MetaMotionC Kit hat eine bestimmte Media-Access-Control-Adresse (MAC-Adresse) zu welcher eine Verbindung via Bluetooth aufgebaut wird. Nachdem die Verbindung hergestellt ist, wird die IMU konfiguriert. Dabei wird als Verbindungsintervall 7,5 ms gesetzt, sodass die Daten alle 7,5 ms aktualisiert werden, als Latenz 0 ms gesetzt und als Timeout 6000 ms. Als Modus wird der NDoF Modus eingestellt, bei welchem die absolute Orientierung vom Beschleunigungssensor, Gyroskop und Magnetfeldstärkemessgerät kalkuliert wird. Der Beschleunigungsmessbereich wird auf +/-8g und der Gyroskopbereich auf 2000 bGrad pro Sekunde (engl. Degree Per Second) (dps) gesetzt. Danach wird als Signal des Sensordatenstreams das Quaternion-Signal eingestellt und abonniert. Als Rückruffunktion (Callback) wird eine selbst implementierte Methode gesetzt, mehr dazu in Unterabschnitt 6.2.2. Anschließend wird der Quaternion-Stream aktiviert und gestartet.

### 6.2.2 Datenabfrage von der IMU

Um die von der IMU empfangenen Daten korrekt zu interpretieren wurde ein Daten-Handler geschrieben. Dieser nutzt die *parse\_value*-Funktion der metawear Bibliothek, um die Daten in einen String zu parsen. Der String ist dabei folgendermaßen aufgebaut:

```
1 quaternion_string =  
2     "W: [FLOAT VALUE], X: [FLOAT VALUE], Y: [FLOAT VALUE],  
3         Z: [FLOAT VALUE]"
```

Um die Float-Werte aus dem String zu extrahieren wird folgender regulärer Ausdruck verwendet:

## 6 Implementierung

```

1 quaternion =
2     re.findall(r"[+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)
3        (?:[eE][+-]?\d+)?", quaternion_string)

```

Dieser füllt die Liste „quaternion“ mit den Strings aller Float-Werte. Diese Strings werden anschließend in Float-Werte konvertiert und der Klassenvariable „self.quaternion“, welcher eine Liste ist, hinzugefügt. Die eigene Klasse „get\_quaternion\_data“ gibt die Klassenvariable „self.quaternion“ zurück.

### 6.2.3 Kalibrierungswerte speichern

Damit der Pilot die Drohne steuern kann, muss zunächst eine Kalibrierung durchgeführt werden, bei der bestimmte Skelett-Daten des Nutzers, wie Armlänge oder die Distanz der Schultern gemessen und gespeichert werden. Dazu stellt sich der Pilot mittig ins Bild und streckt die Arme im 90° Winkel vom Körper weg und schaut nach vorne zur Kamera.

Um die IMU zu kalibrieren, werden während der Kalibrierungsphase des Skelettes die Quaternion Daten in einer Liste gespeichert. Sobald die Kalibrierung erfolgreich abgeschlossen wurde, wird unter anderem die Methode „detect\_head\_orientation“ aufgerufen. Bei diesem ersten Aufruf der Methode ist die Kalibrierung der Quaternionendaten noch nicht abgeschlossen. Deshalb werden dann die Daten des, bei der Kalibrierung zuletzt gespeicherten, Quaternion als Ausgangsquaternion genommen, bei welchem der Nutzer nach vorne schaut.

Um die Kopforientierung zu berechnen wird die Differenz zwischen diesen gespeicherten Quaternion und den aktuellen Quaternionendaten kalkuliert. Da es sich allerdings bei den von der IMU gelieferten Quaternion um sogenannte Einheitsquaternion bezeichnet (vgl. Gleichung 5.3) kann es sein, dass eine Differenz zwischen den zwei Quaternion die Bedingung von Gleichung 5.3 nicht erfüllt, wodurch die Rotation um eine beliebige Achse um einen Winkel  $\alpha$  nicht korrekt dargestellt wird.

Deshalb werden die gespeicherten Ausgangsquaternion in Eulerwinkel umgerechnet und als Ausgangseigenwinkel gespeichert. Um ein Quaternion in Eulerwinkel umzurechnen wird Gleichung 6.1 genutzt [26].

$$\begin{bmatrix} \Phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \arctan\left(\frac{2(w \cdot x + y \cdot z)}{1 - 2 \cdot (x^2 + y^2)}\right) \\ \arcsin(2 \cdot (w \cdot y - z \cdot x)) \\ \arctan\left(\frac{2(w \cdot z + x \cdot y)}{1 - 2 \cdot (y^2 + z^2)}\right) \end{bmatrix} \quad (6.1)$$

## 6 Implementierung

$\Phi$  entspricht dabei der  $x - Achse$  (roll),  $\theta$  der  $y - Achse$  (pitch) und  $\psi$  der  $z - Achse$  (yaw). Zu beachten ist hierbei, dass arctan und arcsin Funktionen in Programmiersprachen nur Ergebnisse zwischen  $-\pi/2$  und  $\pi/2$  liefern und dadurch nicht alle möglichen Orientierungen dargestellt werden können. Deshalb müssen in diesem Fall die arctan Funktionen im Computercode durch atan2 ersetzt werden. Der entsprechende Code für die Konvertierung sieht dann folgendermaßen aus:

```

1 t0 = +2.0 * (w * x + y * z)
2 t1 = +1.0 - 2.0 * (x * x + y * y)
3 roll = math.atan2(t0, t1)
4
5 t2 = +2.0 * (w * y - z * x)
6 t2 = +1.0 if t2 > +1.0 else t2
7 t2 = -1.0 if t2 < -1.0 else t2
8 pitch = math.asin(t2)
9
10 t3 = +2.0 * (w * z + x * y)
11 t4 = +1.0 - 2.0 * (y * y + z * z)
12 yaw = math.atan2(t3, t4)
```

### 6.2.4 Berechnung der Kopforientierung

Für die Berechnung der Kopfausrichtung werden die aktuellen Quaternion-Daten ebenfalls mit Gleichung 6.1 in Eulerwinkel konvertiert. Anschließend wird die Differenz dieses Eulerwinkels zu dem Ausgangseulerwinkel berechnet. Dazu wird Gleichung 6.2 verwendet.

$$\begin{bmatrix} \Phi_d \\ \theta_d \\ \psi_d \end{bmatrix} = \begin{bmatrix} \Phi_n - \Phi_a \\ \theta_a - \theta_n \\ \psi_a - \psi_n \end{bmatrix} \quad (6.2)$$

Dabei entsprechen  $\Phi_d$ ,  $\theta_d$  und  $\psi_d$  der Differenz des Ausgangseulerwinkels ( $\Phi_a$ ,  $\theta_a$ ,  $\psi_a$ ) und des aktuellen Eulerwinkels ( $\Phi_n$ ,  $\theta_n$ ,  $\psi_n$ ). Zu beachten ist, dass bei der Berechnung der  $x - Achse$  (roll) der entsprechende Wert des Ausgangseulerwinkels von dem entsprechenden Wert des aktuellen Eulerwinkels abgezogen wird und nicht umgekehrt. Dies liegt an den Daten die die IMU liefert.

Nachdem die Differenz der beiden Eulerwinkel berechnet wurde, wird das Ergebnis wieder in Quaternion umgerechnet und zurück gegeben. Dazu wird Gleichung 6.3 verwendet [26].

## 6 Implementierung

$$\begin{aligned}
 q &= \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\psi_d/2) \\ 0 \\ 0 \\ \sin(\psi_d/2) \end{bmatrix} \begin{bmatrix} \cos(\theta_d/2) \\ 0 \\ \sin(\theta_d/2) \\ 0 \end{bmatrix} \begin{bmatrix} \cos(\Phi_d/2) \\ \sin(\Phi_d/2) \\ 0 \\ 0 \end{bmatrix} \\
 &= \begin{bmatrix} \cos(\Phi_d/2) \cdot \cos(\theta_d/2) \cdot \cos(\psi_d/2) + \sin(\Phi_d/2) \cdot \sin(\theta_d/2) \cdot \sin(\psi_d/2) \\ \sin(\Phi_d/2) \cdot \cos(\theta_d/2) \cdot \cos(\psi_d/2) - \cos(\Phi_d/2) \cdot \sin(\theta_d/2) \cdot \sin(\psi_d/2) \\ \cos(\Phi_d/2) \cdot \sin(\theta_d/2) \cdot \cos(\psi_d/2) + \sin(\Phi_d/2) \cdot \cos(\theta_d/2) \cdot \sin(\psi_d/2) \\ \cos(\Phi_d/2) \cdot \cos(\theta_d/2) \cdot \sin(\psi_d/2) - \sin(\Phi_d/2) \cdot \sin(\theta_d/2) \cdot \cos(\psi_d/2) \end{bmatrix}
 \end{aligned} \tag{6.3}$$

### 6.2.5 Messaging der Kopforientierung

Um die Kopforientierung an den Drohnencontroller zu senden, werden die Messages von ROS Kinetic verwendet. Die „FOVInstructions.msg“ enthält vier float32 Werte:  $w$ ,  $x$ ,  $y$  und  $z$ . Es können beliebige Komponenten des Programms diese Daten erhalten, indem sie diese Message mit dem ROS-Subscriber abonnieren und eine Callback-Funktion zuweisen. Der „communication\_manager“ des Drohnencontrollers abonniert die „FOVInstructions.msg“ und nimmt in der Callback-Funktion die Quaternion-Daten entgegen und gibt sie an die Methode zum Setzen der Kameraausrichtung weiter.

### 6.2.6 Änderung der Kameraausrichtung in der AirSim Simulation

Um den Winkel der Drohnen-Kamera in der AirSim Simulation zu ändern werden zunächst die von der „FOVInstructions.msg“ erhaltenen Quaternion-Daten in die Datenstruktur eines AirSim-Quaternion konvertiert. Um die Kameraorientierung zu steuern, liefert AirSim bei ihrem „MultirotorClient“ die Methode „simSetCameraOrientation“, welcher die ID der Kamera und die neue Ausrichtung übergeben werden. In unserem Fall wollen wir die Frontkamera der Drohne mit der ID „front\_center“ anders ausrichten. Der Code dazu sieht folgendermaßen aus:

```

1 AirSim_quaternion = AirSim.Quaternionr(w_val = quaternion[0],
2     x_val = quaternion[1], y_val = quaternion[2],
3     z_val = quaternion[3])
4 # set camera orientation according to quaternion
5 self.fov_client.simSetCameraOrientation("front_center",
6     AirSim_quaternion)

```

## 6 Implementierung

Die Liste „quaternion[]“ entspricht den, durch die „FOVInstructions.msg“ erhaltenen, Quaternion-Daten. Der „fov\_client“ ist ein MultirotorClient von AirSim.

### 6.3 Steuerung einer realen Drohne

Aufgrund von technischen Schwierigkeiten und dem daraus resultierenden Zeitmangel konnte die Steuerung einer Parrot AR.Drone 2.0 oder Parrot Bebop Drone nicht in dieser Arbeit implementiert werden. Verschiedene Ansätze zu einer möglichen Implementierung zur Steuerung der beiden Parrot Drohnen wurden in Kapitel 4 aufgezeigt. Die Implementierung der Steuerung einer realen Drohne wurde bei der Entwicklung der Softwarearchitektur während der Studienarbeit „I believe I can fly 2.0“ berücksichtigt, sodass einer tatsächlichen Umsetzung von Seiten der Softwarearchitektur nichts im Wege steht. Die Architektur der ROS Nodes wird in Abbildung 6.1 gezeigt.

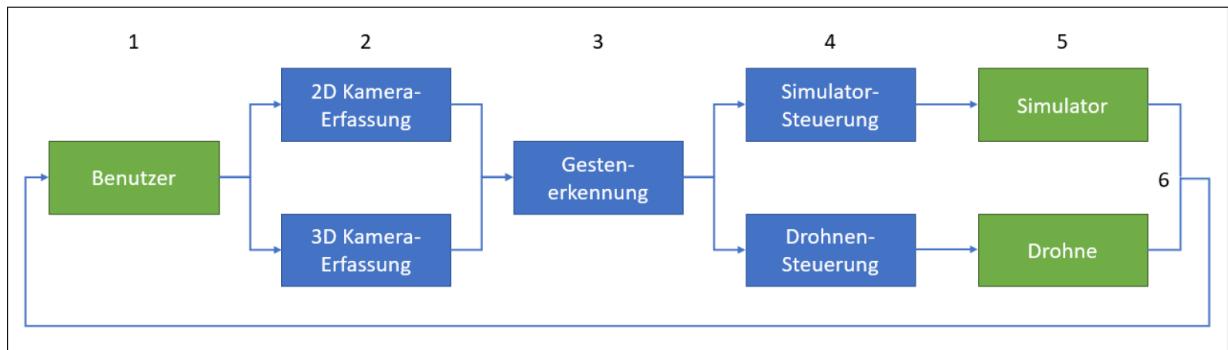


Abbildung 6.1: ROS Node Architektur [3]

Die grünen Ressourcen werden für die Ausführung benötigt, sind allerdings kein direkter Teil der Software. Die blauen Ressourcen entsprechen jeweils einer ROS Node. Der in dieser Arbeit entwickelte Code befindet sich in der Node für die Gestenerkennung und in der Node für die Simulator-Steuerung.

# 7 Evaluation

In diesem Kapitel wird das Ergebnis des Projektes evaluiert. Dazu wird die Laufzeit gemessen und die Benutzerausgabe qualitativ bewertet.

## 7.1 Laufzeitevaluation

Für die Laufzeitevaluation wurden über 340 Laufzeiten der einzelnen Funktionen gemessen. Zu diesen Funktionen zählt die Berechnung der Kopforientierung, das Veröffentlichen der Quaternion-Daten mit einer Message, die Entgegennahme der Quaternion-Daten und das Ändern der Kameraausrichtung in der AirSim Simulation. Dabei wurden im Durchschnitt die Zeiten aus Tabelle 7.1 gemessen.

Funktion	Laufzeit in ms
Berechnung der Kopfausrichtung	0.067163957
Veröffentlichen der Quaternion-Daten	0.117163837
Entgegennahme der Quaternion-Daten	0.065310153
Änderung der Kameraausrichtung	4.146136989
<b>Insgesamt</b>	<b>4.395774936</b>

Tabelle 7.1: Laufzeiten der einzelnen Funktionen

Die Berechnung der Kopfausrichtung schließt die Abfrage der aktuellen Quaternion-Daten nicht mit ein. Stattdessen wird der zuletzt von der IMU erhaltene Wert abgefragt. Diese streamt ihre Daten mit einer Frequenz von 100 Hz.

Mit einer insgesamten Laufzeit von 4.395774936 ms wäre für diese Aufgabe theoretisch eine Bildfrequenz von ca. 227 Bilder pro Sekunde (engl. Frames Per Second) (FPS) möglich. Da jedoch noch andere Berechnungen, wie die des Skelettes durch OpenPose oder die Berechnung der Flugbefehle, ausgeführt werden, ist die tatsächliche Bildfrequenz deutlich geringer.

## 7 Evaluation

### 7.2 Benutzerausgabe

Der Benutzer bekommt den Bildschirm des Laptops auf einer Videobrille angezeigt. Eine beispielhafte Ausgabe wird in Abbildung 7.1 angezeigt. Oben an dieser Videobrille ist die IMU befestigt (siehe Abbildung 7.2).

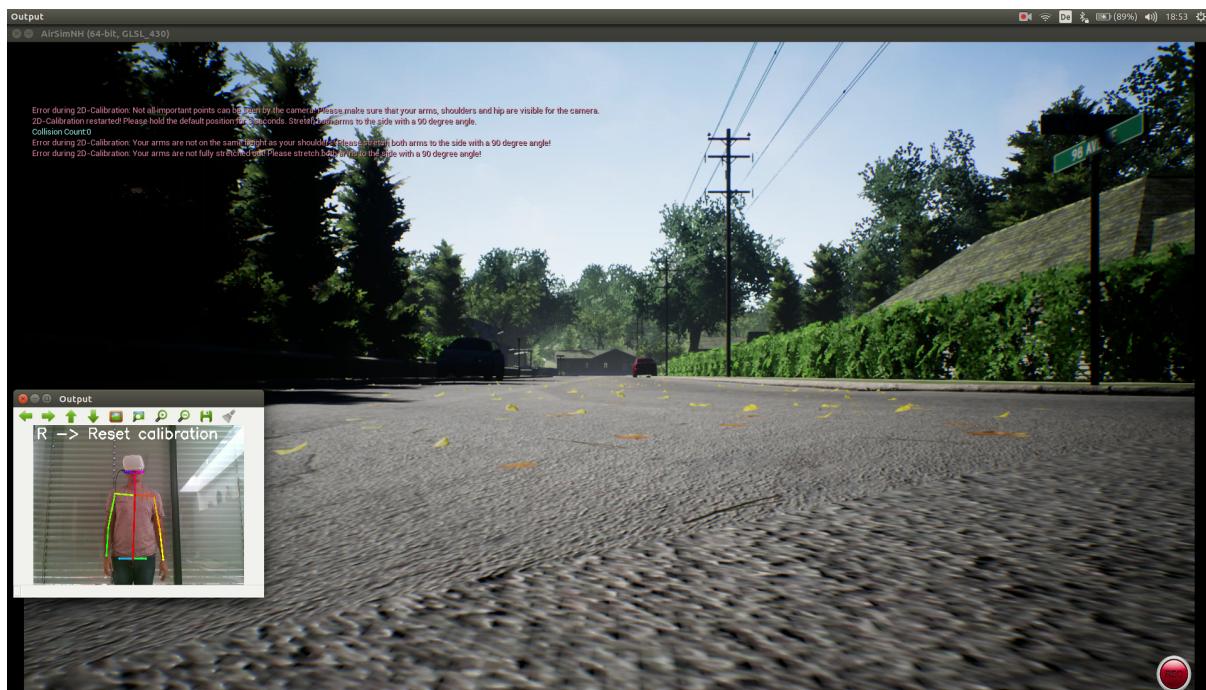


Abbildung 7.1: Beispielhafte Benutzerausgabe



Abbildung 7.2: IMU ist oben an der Videobrille befestigt

## 7 Evaluation

### 7.2.1 Vor der Kalibrierung

Vor der Kalibrierung kann der Pilot, wie geplant, keinen Einfluss auf die Kameraausrichtung nehmen. Dies ist in Abbildung 7.3 zu sehen.

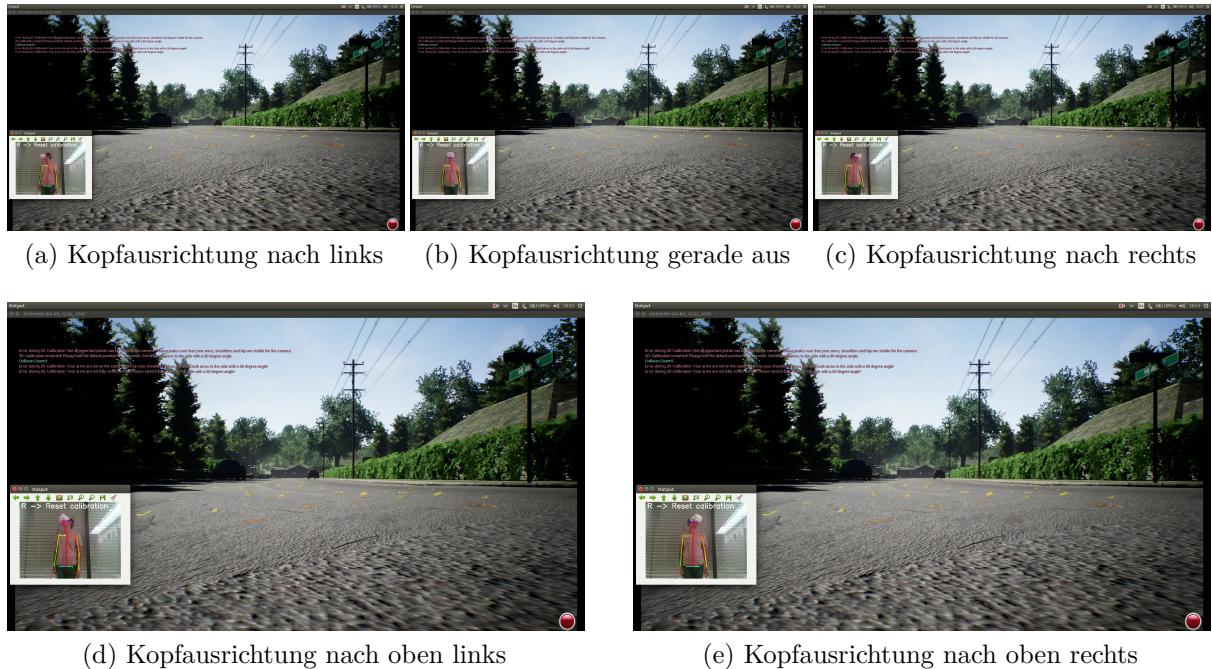


Abbildung 7.3: Kein Einfluss auf die Kameraausrichtung vor der Kalibrierung

Für die Kalibrierung des Nutzers muss dieser seine Arme von seinem Körper weg strecken und nach vorne schauen, sodass seine Körperform einem „T“ ähnelt. Diese Position muss für 3 Sekunden beibehalten werden.

### 7.2.2 Nach der Kalibrierung

Direkt nach der erfolgreichen Kalibrierung wird die Kamera entsprechend der Kopforientierung des Piloten ausgerichtet, sodass der Nutzer die Umgebung der Drohne beobachten kann.

#### Umherschauen

Wie in Abbildung 7.4 zu sehen ist, funktioniert die Kameraausrichtung in der AirSim Simulation kurz nach der Kalibrierung sehr gut. Verschiedene Nutzer konnten keine

## 7 Evaluation

Unstimmigkeiten bei der Kameraausrichtung feststellen. Sie bekamen alle das Gefühl den Blickwinkel direkt durch die Bewegung des Kopfes bestimmen zu können. Dabei handelt es sich um Testpersonen im Alter zwischen 21 und 51 Jahren, unterschiedlichen Geschlechtes. Aufgrund der aktuellen Covid-19-Pandemie ist die Anzahl an Testpersonen auf Familienmitglieder und enge Freunde beschränkt.

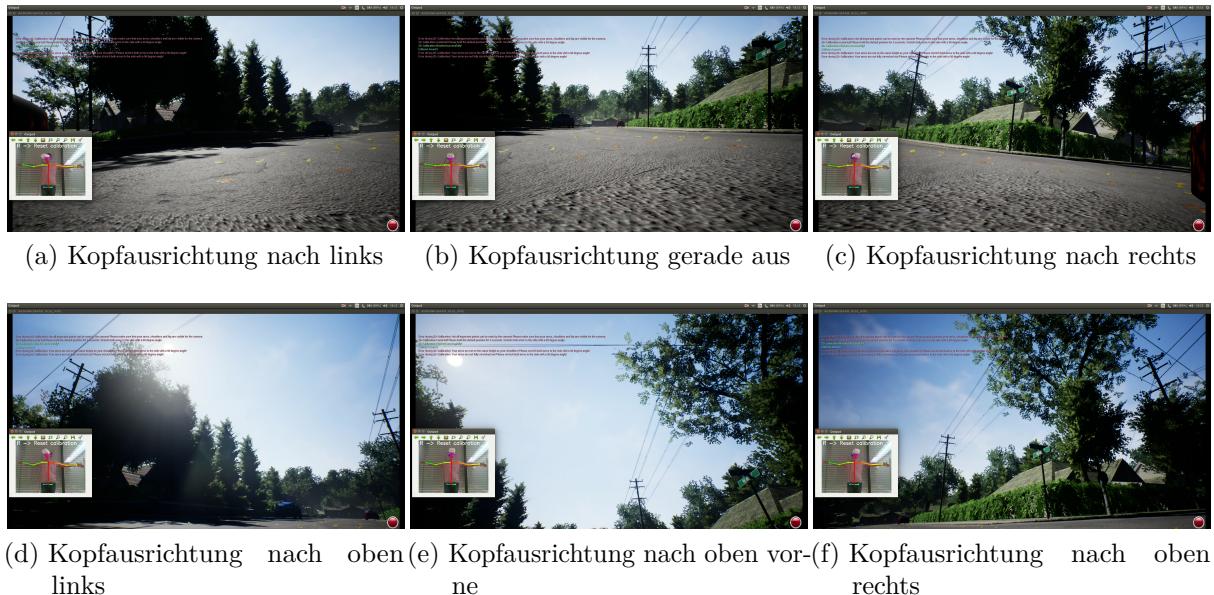


Abbildung 7.4: Einfluss auf die Kameraausrichtung nach der Kalibrierung

### Fliegen und umherschauen

Während dem Fliegen ist es ebenfalls möglich in der simulierten Umgebung umherzuschauen, wie in Abbildung 7.5 gezeigt. Auch dabei konnten verschiedene Testpersonen keine Unstimmigkeiten bei der Kameraausrichtung kurz nach der Kalibrierung feststellen.

## 7 Evaluation



Abbildung 7.5: Steuerung der Kameraausrichtung während dem Flug

Bei diesen Aufnahmen wurde entlang der Straße nach vorne geflogen. Die Bilder sind zeitlich von links nach rechts geordnet.

### **Unstimmigkeiten bei der Kameraausrichtung**

Nach einer gewissen Flugzeit fällt den Testpersonen jedoch auf, dass die Kameraausrichtung nicht mehr der Kopforientierung entspricht. In Abbildung 7.6 schaut der Pilot nach links und nicht gerade aus, während er der Drohne den Befehl erteilt nach vorne zu fliegen. Dennoch schaut der Nutzer in die Flugrichtung.

## 7 Evaluation



(a) Kopfausrichtung nach links während Vorwärtsflug (b) Kopfausrichtung nach links während Vorwärtsflug



(c) Kopfausrichtung nach links während Vorwärtsflug (d) Kopfausrichtung nach links während Vorwärtsflug

Abbildung 7.6: Unstimmigkeiten bei der Steuerung der Kameraausrichtung nach einer bestimmten Flugzeit

Die einzelnen Bilder sind zeitlich von a) nach d) geordnet.

Diese Unstimmigkeiten können zwei verschiedene Ursachen haben. Zum Einen, dass die Drohne in die falsche Richtung fliegt. Zum Anderen könnte dies daran liegen, dass die Drohnen-Kamera nicht mehr in die richtige Richtung zeigt.

Abbildung 7.7, welche kurze Zeit später aufgenommen wurde, widerlegt die erste mögliche Ursache. Der Pilot schaut nach vorne, doch es ist deutlich ein Teil des vorderen rechten Rotors der Drohne zu erkennen. Die Kamera ist also nach rechts ausgerichtet. Gründe können Fehler beim Setzen der Kameraausrichtung, beim Umrechnen der Quaternion-Daten in Eulerwinkel und wieder zurück, oder fehlerhafte IMU-Daten sein.

## 7 Evaluation



Abbildung 7.7: Kameraausrichtung entspricht nicht der Kopforientierung

Die gespeicherten IMU-Daten (vgl. Tabelle 7.2) und die daraus resultierenden Kopfausrichtungen (vgl. Tabelle 7.3) bestätigen, dass die IMU nach einer Weile falsche Quaternion-Daten liefert. Die genaue Ursache für diese fehlerhaften Daten konnte nicht ermittelt werden. Der Fehler kann allerdings behoben werden, indem die Kalibrierung des Piloten zurückgesetzt und neu durchgeführt wird. Denn bei der Kalibrierung werden am Ende die aktuellen Quaternion-Daten als Ausgangsquaternion gespeichert und als Ausrichtung genutzt, bei welcher der Pilot nach vorne schaut.

Zeitpunkt	<b>Q (W, X, Y, Z) - IMU</b>	<b>E (Roll, Pitch, Yaw) - IMU</b>
Ausgangsdaten	0.053, -0.527, 0.848, 0.025	-3.128, 0.117, -2.030
Direkt nach Kalibrierung	0.052, -0.526, 0.848, 0.026	-3.131, 0.116, -2.0298
Nach 2.5 Minuten	0.058, -0.061, 0.996, -0.015	-3.104, 0.114, -3.017

Tabelle 7.2: Fehlerhafte IMU Daten

Zeitpunkt	<b>Q (W, X, Y, Z) - KA</b>	<b>E (Roll, Pitch, Yaw) - KA</b>
Direkt nach Kalibrierung	0.999, -0.001, 0.000, 0.000	-0.003, 0.001, 0.001
Nach 2.5 Minuten	0.880, 0.010, 0.007, 0.474	0.023, 0.003, 0.988

Tabelle 7.3: Resultierende Kopfausrichtung durch die fehlerhaften IMU Daten

Q steht dabei für Quaternion, E für Eulerwinkel und KA für Kopfausrichtung. Zu den entsprechenden Zeitpunkten wurde immer nach vorne geschaut. Die Eulerwinkel wurden aus den Quaternion-Daten mit Gleichung 6.1 berechnet und auf drei Nachkommastellen

## *7 Evaluation*

gerundet. Die Quaternion-Daten von der Kopforientierung wurden ebenfalls auf drei Nachkommastellen gerundet. Zu beachten ist, dass die Eulerwinkel nicht in Grad, sondern in Radian angegeben sind.

Aus den Daten geht hervor, dass 2.5 Minuten nach der Kalibrierung bei den Eulerwinkeln vor allem die Drehung um die Yaw-Achse einen deutlich anderen Wert hat, als kurz nach der Kalibrierung. Die kleinen Veränderungen bei der Roll- und Pitch-Achse sind vernachlässigbar, da sie durch leichte Unterschiede bei der Kopforientierung zu den verschiedenen Zeitpunkten auftreten können, wie man auch schon an den ersten beiden Spalten in Tabelle 7.2 erkennt. Dort unterscheiden sich die 3 Werte bei den Eulerwinkeln auch leicht, obwohl zwischen der Aufnahme der Daten nur wenige Millisekunden liegen. Die Quaternion-Daten 2.5 Minuten nach der Kalibrierung unterscheiden sich natürlich bei allen Variablen deutlich von den Quaternion-Daten bei, und kurz nach der Kalibrierung, da sich die kleine Änderung der Yaw-Achse (bei den Eulerwinkeln) auf die einzelne Drehachse der Quaternion stark auswirkt (vgl. das Prinzip der Quaternion aus Abschnitt 5.2).

## 8 Fazit

In dieser Studienarbeit wurden verschiedene Methoden zur Bestimmung der Kopforientierung aufgezeigt und geprüft, ob sie für das Projekt „I believe I can fly“ und diese Arbeit in Frage kommen. Es wurden Methoden implementiert, die Kamera der Drohne im AirSim Simulator entsprechend der Kopfbewegung des Piloten auszurichten. Dabei wird eine IMU für die Bestimmung der Kopfausrichtung verwendet.

Möglichkeiten zum Ansteuern von realen Drohnen, wie die Parrot AR.Drone 2.0 oder die Parrot Bebop Drohne, wurden ebenfalls dargelegt. Die eigentlich geplante Implementierung der Steuerung von realen Drohnen konnte leider zeitlich nicht umgesetzt werden.

Die Implementierung der Kameraausrichtung wurde anschließend auf die Laufzeit und Qualität evaluiert. Dabei wurde festgestellt, dass die Implementierung kaum Laufzeit benötigt, aber die IMU nach einer gewissen Zeit fehlerhafte Daten liefert. Die Ursache dafür konnte nicht ermittelt werden. Eine kurzzeitige Lösung ist den Flug zu unterbrechen und die Kalibrierung des Piloten neu zu starten.

Das Projekt „I believe I can fly“ kann auf verschiedene Arten fortgesetzt werden. Zum Einen könnte OpenPose nachtrainiert werden, um über OpenPose auch die Kopforientierung bestimmen zu können. Darunter würde vermutlich die Performance leiden, aber die Qualität bei der langfristigen Bestimmung der Kopfausrichtung wahrscheinlich steigen.

Zum Anderen kann die Ansteuerung von realen Drohnen implementiert werden, um das Erlebnis des Nutzers durch reale Videoaufnahmen zu steigern.

Eine weitere Entwicklungsmöglichkeit wäre das Einbeziehen der Kopfausrichtung in die Steuerung der Drohne. Damit könnte beispielsweise bei einem Vorwärtsflug die Flugrichtung durch die Kopfausrichtung bestimmt werden.

Um das Problem der fehlerhaften IMU Daten kurzfristig zu lösen, könnte zudem eine Methode zum Reset der IMU-Kalibrierung implementiert werden. So könnte der Nutzer beim nach vorne Schauen mit einem Knopfdruck die Methode ausführen, sodass die Ausgangsdaten der IMU neu kalibriert werden, ohne dass die Kalibrierung des Piloten zurückgesetzt wird.

# Literatur

- [1] Cicolai BENZ, Marcus von BERGEN und Denis VONSCHEIDT. „Gestensteuerung eines Flugroboters im AR-Kontext - I believe I can fly“. In: 2015 [siehe S. 11].
- [2] Christoph MEISE und Max LENK. „Entwicklung eines Indoor-Assistenzsystems für Multicopter“. In: 2017 [siehe S. 11].
- [3] Henri KOHLBERG. „I believe I can fly V2.0 - Detektion und Klassifikation von NUI-Flugbefehlen anhand einzelner Farbkamera-Bilder“. In: 2019 [siehe S. 11, 36].
- [4] Leyuan LIU u. a. „Head Pose Estimation through Keypoints Matching between Reconstructed 3D Face Model and 2D Image“. In: *Sensors* 21 [März 2021], S. 1841. DOI: 10.3390/s21051841 [siehe S. 13, 14].
- [5] Leonid PISHCHULIN u. a. *DeepCut: Joint Subset Partition and Labeling for Multi Person Pose Estimation*. 2016. arXiv: 1511.06645 [cs.CV] [siehe S. 14].
- [6] Federica BOGO u. a. „Keep It SMPL: Automatic Estimation of 3D Human Pose and Shape from a Single Image“. In: Bd. 9909. Okt. 2016, S. 561–578. ISBN: 978-3-319-46453-4. DOI: 10.1007/978-3-319-46454-1\_34 [siehe S. 14].
- [7] Matthew LOPER u. a. „SMPL: A Skinned Multi-Person Linear Model“. In: *ACM Trans. Graphics (Proc. SIGGRAPH Asia)* 34.6 [Okt. 2015], 248:1–248:16 [siehe S. 14].
- [8] Tomas Simon Shih-En Wei Yaadhav Raaj Hanbyul Joo und Yaser Sheikh GINÉS HIDALGO Zje Cao. *OpenPose*. <https://github.com/CMU-Perceptual-Computing-Lab/openpose>. Mai 2021 [siehe S. 15, 16].
- [9] Md ISLAM u. a. „An intelligent shopping support robot: understanding shopping behavior from 2D skeleton data using GRU network“. In: *ROBOMECH Journal* 6 [Dez. 2019]. DOI: 10.1186/s40648-019-0150-1 [siehe S. 15].

## Literatur

- [10] Satya MALLICK. *Head Pose Estimation Using OpenCV and Dlib*. <https://learnopencv.com/head-pose-estimation-using-opencv-and-dlib/>. Mai 2021 [siehe S. 16].
- [11] DLIB. *Dlib C++ Library*. <http://dlib.net/>. Mai 2021 [siehe S. 16].
- [12] Oliver J. WOODMAN. *An introduction to inertial navigation*. Techn. Ber. UCAM-CL-TR-696. University of Cambridge, Computer Laboratory, Aug. 2007. DOI: 10.48456/tr-696 [siehe S. 16].
- [13] OPENPOSE. *OpenPose Training*. [https://github.com/CMU-Perceptual-Computing-Lab/openpose\\_train/tree/master/training](https://github.com/CMU-Perceptual-Computing-Lab/openpose_train/tree/master/training). Mai 2021 [siehe S. 21].
- [14] ZHE CAO. *Realtime Multi-Person Pose Estimation*. [https://github.com/ZheC/Realtime\\_Multi-Person\\_Pose\\_Estimation](https://github.com/ZheC/Realtime_Multi-Person_Pose_Estimation). Mai 2021 [siehe S. 21, 22].
- [15] PARROT. *AR.Drone Developer Guide*. <https://jpchanson.github.io/ARdrone/ParrotDevGuide.pdf>. Mai 2021 [siehe S. 23, 24].
- [16] MANI MONAJJEMI. *ardrone\_autonomy Documentation*. <https://ardrone-autonomy.readthedocs.io/en/latest/index.html>. Mai 2021 [siehe S. 24].
- [17] PARROT. *Parrot Bebop Drone for Developers - ARDroneSDK3 API Reference*. <https://developer.parrot.com/docs/SDK3/>. Mai 2021 [siehe S. 26].
- [18] MANI MONAJJEMI. *bebop\_autonomy Documentation*. <https://bebop-autonomy.readthedocs.io/en/latest/index.html>. Mai 2021 [siehe S. 26].
- [19] PROF. DR. O. BITTEL. *Position und Orientierung*. [http://www-home.hwg-konstanz.de/~bittel/ain\\_robo/Vorlesung/02\\_PositionUndOrientierung.pdf](http://www-home.hwg-konstanz.de/~bittel/ain_robo/Vorlesung/02_PositionUndOrientierung.pdf). Mai 2021 [siehe S. 27, 29].
- [20] S. MÜLLER. *Orientierung - Vorlesung Animation und Simulation*. [https://userpages.uni-koblenz.de/~cg/ss13/ansim/04\\_orientierung.pdf](https://userpages.uni-koblenz.de/~cg/ss13/ansim/04_orientierung.pdf). Mai 2021 [siehe S. 27].
- [21] WIKIPEDIA. *Eulersche Winkel*. [https://de.wikipedia.org/wiki/Eulersche\\_Winkel](https://de.wikipedia.org/wiki/Eulersche_Winkel). Mai 2021 [siehe S. 27].
- [22] Darío MARAVALL, Javier LOPE und Juan Pablo FUENTES BREA. „Navigation and Self-Semantic Location of Drones in Indoor Environments by Combining the Visual Bug Algorithm and Entropy-Based Vision“. In: *Frontiers in Neurorobotics* 11 [Aug. 2017], S. 46. DOI: 10.3389/fnbot.2017.00046 [siehe S. 28].

## Literatur

- [23] Edmund WEITZ. *Quaternionen und Gimbal Lock (kardanische Blockade)*. <https://www.youtube.com/watch?v=kHFwysRM2ps>. Video file. Mai 2017 [siehe S. 29].
- [24] THORSTEN THORMÄHLEN. *3D Transformationen - Grafikprogrammierung*. [https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/graphics\\_5\\_2\\_ger\\_web.html](https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/graphics_5_2_ger_web.html). Mai 2021 [siehe S. 30].
- [25] INTEL. *Intel® Core™ i7-6700HQ Prozessor*. <https://ark.intel.com/content/www/de/de/ark/products/88967/intel-core-i7-6700hq-processor-6m-cache-up-to-3-50-ghz.html>. Mai 2021 [siehe S. 31].
- [26] WIKIPEDIA. *Conversion between quaternions and Euler angles*. [https://en.wikipedia.org/wiki/Conversion\\_between\\_quaternions\\_and\\_Euler\\_angles](https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles). Mai 2021 [siehe S. 33, 34].