Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Lars Harmsen

Bewegungslatenzkompensation für HMD von Telepräsenzrobotersystemen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lars Harmsen

Bewegungslatenzkompensation für HMD von Telepräsenzrobotersystemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Dipl.-Ing. Thomas Lehmann
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 8. August 2014

Lars Harmsen

Thema der Arbeit
Bewegungslatenzkompensation für HMD von Telepräsenzrobotersystemen

Stichworte
Latenzkompensation, Oculus Rift, Head-Mounted-Display, Virtual-Reality Brille, Telepräsenzroboter, schwenkbarer Kamerakopf, virtuelles Schwenken

Kurzzusammenfassung
Thema dieser Arbeit ist die Kompensation von Bewegungslatenzen, welche bei der Kopplung von HMDs mit Head-Tracking-System und Telepräsenzrobotern auftreten. Bei Übertragung und Verarbeitung von Bild- und Sensordaten treten Latenzen auf, welche den Eindruck der Telepräsenz beim Umsehen trüben. Es wird versucht, diese Latenzen durch Hinzunahme von virtuellem Schwenken weitestgehend zu kompensieren. Das dazu nötige Kamerasystem wird diskutiert. Des Weiteren werden Gedanken für einblendbare Informationen gesammelt. Für die Umsetzung wurde mit dem Development Kit 1 der Oculus Rift und dem Pioneer P3-DX inklusive schwenkbarem Kamerakopf gearbeitet.

Lars Harmsen

Title of the paper
Movement latency compensation for HMD of telepresence robot systems

Keywords
latencycompensation, Oculus Rift, Head-Mounted-Display, Virtual-Reality Headset, telepresencerobot, tiltable camerahead, virtual tilting

Abstract
The subject of this paper is the compensation of movementlatencies that occur by combining a telepresencerobot and a HMD with a head-tracking-system. With the transfer and processing of image- and sensordata latencies occur which reduce the effect of the telepresence while observing the surrounding. The attempt is to compensate this latency by using virtual paning and tilting. The necessary camerasystem is discussed. This paper also includes some thoughts about additional information that can be displayed to the user. For the implementation the development kit 1 of the Oculus Rift and a Pinoeer P3-DX with a tiltable camera mount was used.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Ein Telepräsenzroboter vertritt einen Menschen, wenn dieser selbst nicht an dem gewünschten Ort präsent sein kann. Etwa weil die Entfernung und damit der Zeitaufwand zu groß wäre, weil der Ort schwer zugänglich ist oder einfach zu gefährlich. Um dem Benutzer trotzdem das Gefühl vermitteln zu können, dass er vor Ort ist, wird der Roboter mit einem Kamerasystem bestückt und dem Benutzer liegen ein Head-Mounted-Display (HMD) inklusive einem Head-Tracking-System vor, sodass sich der Nutzer umgucken kann, als wäre er vor Ort.

1.1. Motivation

Die Basisplattform für einen Telepräsenzroboter aus der vorhergehenden Arbeit (siehe Wiese (2014)) bietet bereits die nötigen Grundlagen für die Kopplung eines Telepräsenzroboters mit einem HMD und einem Head-Tracking-System. Der Eindruck dabei wird jedoch deutlich durch hohe Latenzen (siehe Kapitel 6 Latenz), die durch die großen Entfernungen und das genutzte Übertragungsmedium entstehen sowie durch mechanisch bedingte Latenzen bei den Kamerabewegungen von mechanischen Kamerasystemen, beeinträchtigt.

Bei gegebenen Daten und gegebenem Übertragungskanal lassen sich Latenzen nicht umgehen, jedoch ist es möglich, sie durch Hinzunahme von Software zu kompensieren. Da hierbei bereits eine Bildverarbeitung stattfindet, können ebenfalls weitere Gedanken zu Erweiterungen hierzu eingehen.

1.2. Anforderung und Zielsetzung

Ziel ist es, ein Programm zu entwickeln, welches bewegungslatenzkompensierende Maßnahmen auf der zur Verfügung stehenden Hardware umsetzt. Der bestehende Kamerakopf soll dabei nicht ersetzt, sondern durch Software erweitert werden. Die vorgenommenen Maßnahmen sollen durch Latenzmessungen ihren Einfluss aufzeigen.

Das implementierte Programm soll mit wenig Aufwand für weitere HMD-, Kamera- und Head-Tracking-Systeme abänderbar sein.

Zusätzlich werden Erweiterungen der Anzeige diskutiert und exemplarisch implementiert werden. Abschließend werden hierfür, und für weitergehende Arbeiten, nötige Hardwareveränderungen oder Installationen genannt.

2. Head-Mounted-Display

Das wichtigste Gerät der Basisstation ist das Head-Mounted-Display (HMD). Es lässt den Benutzer durch die Augen des Roboters sehen. Ein HMD ist, vereinfacht gesagt, nur ein Monitor, wodurch es alleine nicht lauffähig ist, sondern einen Computer benötigt, der die Bilder darauf anzeigt. Eine Unterkategorie des HMDs stellt die VR-Brille dar. Dabei wird das Display in eine tragbare Brille integriert, die die Sicht auf die Umwelt so abschirmt, dass nur noch der dargestellte Inhalt des Displays für den Benutzer zu sehen ist. Hierdurch kann eine Immersion entstehen, sodass dem Benutzer ein realer Eindruck vermittelt wird. Es bietet eine Anzeige für beide Augen, sodass eine dreidimensionale Sicht möglich ist. Im Verlauf dieser Arbeit wird unter dem Begriff HMD meist eine solche VR-Brille verstanden. Wichtige Spezifikationen bilden das Gewicht (Tragekomfort), die Displayart und -auflösung sowie die Größe des Sichtfeldes.

Die Entwicklung nimmt seit 2012, mit der Oculus Rift als Innovator, auch im Konsumerbereich zu. Aktuell sind bereits einige Prototypen sowie fertige Produkte erhältlich. HMDs die als VR-Brille genutzt werden, haben für gewöhnlich schon ein Head-Tracking-System integriert. Zu diesem Thema siehe Kapitel 3 Head-Tracking-System. Das folgende Kapitel hat nicht den Anspruch auf Vollständigkeit, sondern soll lediglich einen Überblick über verschiedene Produkte bieten.

2.1. Oculus Rift

Die Oculus Rift gehört zu einer der ersten VR-Brillen, die es bis in den Konsumerbereich geschafft haben und damit eine neue Ära in Sachen Virtuelle Realität einleitete. Seit die Oculus Rift 2012 durch die Cowdfunding Plattform Kickstarter die nötige Finanzierung für die Entwicklung gesammelt hat, wurden bereits zwei Developer Kits ausgeliefert (Developer Kit 1 hier in Verwendung (siehe Abbildung 2.1)). Das Endprodukt wird 2015 erwartet (siehe Edwards (2014)).

Die Oculus Rift bietet ein 90° breites und 110° hohes Sichtfeld (siehe Lang (2012)). Dieses weite Sichtfeld wird trotz dichtem Display erreicht, indem Linsen die Sicht verzeichnen (siehe

Abbildung 2.2). Um eine klare Sicht zu erzeugen, müssen die Bilder auf dem Display entgegen der Linsen tonnenverzeichnet werden (siehe Abschnitt 8.1).

In der VR-Brille befindet sich ein sieben Zoll Display, das sich auf beide Augen aufteilt. Die Auflösung und Display-Technik änderte sich mit den erschienen Dev Kits und es sind noch keine Spezifikationen für das Endprodukt verfügbar. Das eingesetzte Dev Kit 1 nutzt ein LCD mit einer Auflösung von 1280x800[1] Pixeln.

Abbildung 2.1.: Foto vom Oculus Rift Dev Kit 1. (Froehlich (2014))

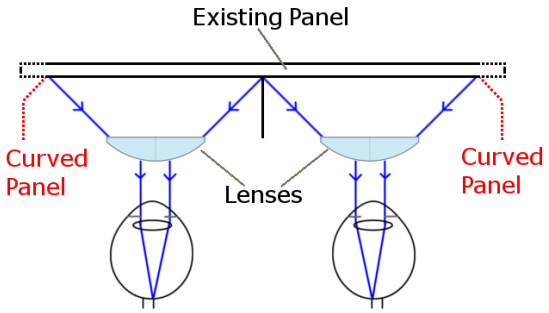

Abbildung 2.2.: Skizzierung der Linsenfunktionalität in der Oculus Rift. (Calanar (2014))

[1] 640x800 pro Auge

2.2. Google Cardboard

Am 26.06.2014 wurde auf der Entwicklerkonferenz *Google I/O* das Projekt Cardboard (siehe Abbildung 2.3) vorgestellt, mit dem Ziel Virtual Reality günstig für die Masse anzubieten. Beim Projekt Cardboard handelt es sich um eine VR-Brille, die aus Pappe gefaltet wird. Zum Zusammenbau benötigt man zusätzlich noch zwei Linsen, Magneten, Klettband und Gummibänder. Als Display und für die Sensorik dient ein Android Smartphone, welches man in die Papphalterung einsetzt.

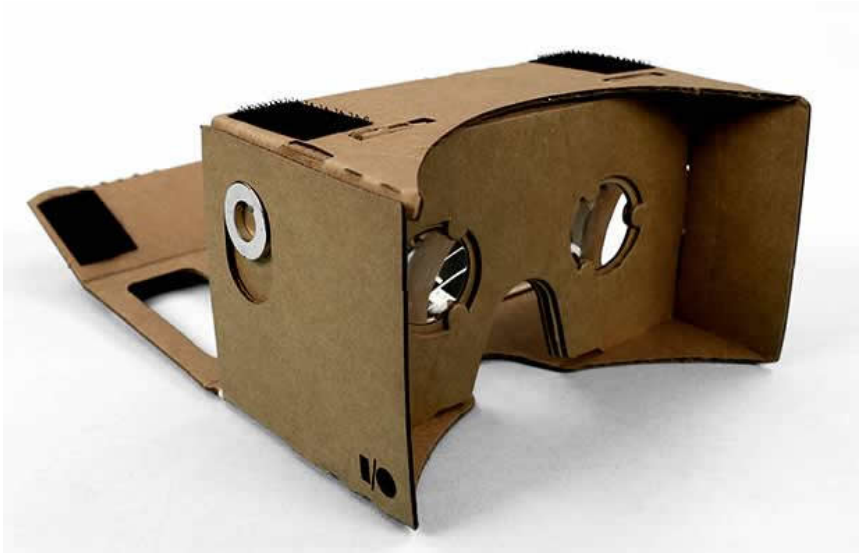

Abbildung 2.3.: Google Cardboard (Google Inc. (2014))

Es wird ein komplett kabelloser Virtual Reality (VR)-Eindruck geliefert, ohne dass es einen weiteren Computer bedarf. An der Performance des Smartphones und der Genauigkeit der Sensoren kann gezweifelt werden. Da das angedachte Szenario dieser Arbeit von einem mobilen Roboter und einer stationären Basis ausgeht, wird eine weniger kompakte Bauweise mit zusätzlichem Computer, zu Gunsten der Performance, in Kauf genommen, wodurch sich dieses HMD weniger eignet. Dadurch, dass die genutzte Kommunikationsplattform Roboter Operating System (ROS) inoffiziell und die Bildverarbeitungsbibliothek OpenCV offiziell für Android entwickelt werden, wäre eine Kompatibilität durchaus möglich.

2.3. Weitere Systeme

Ein HMD, welches der Oculus Rift sehr ähnlich sein soll, wird aktuell von Sony entwickelt. Das *Project Morpheus* genannte System soll für die Playstation umgesetzt werden, sodass eine Kompatibilität mit einem ROS-System zur Zeit nicht bekannt ist.

Ähnlich dem Google Cardboard baut auch Samsung ein System, das mit einem Smartphone als Display dient. Über *Samsung Gear VR* ist allerdings zum Zeitpunkt der Erstellung dieser Arbeit noch nicht viel bekannt.

Des Weiteren sind noch ältere Modelle von anderen Herstellern verfügbar, welche in Bezug auf Auflösung und Sichtfeld jedoch nicht mit aktuellen Produkten vergleichbar sind.

3. Head-Tracking-System

Eine zentrale Rolle bei der Telepräsenz spielt das Erfassen der Kopfbewegungen, um diese an den Kamerakopf des Roboters und/oder an die Bildverarbeitung weiterzuleiten.

In dieser Arbeit wird sich hauptsächlich mit der Orientierung des Kopfes befasst. Bewegungen, die keine Rotation mit sich führen, könnten zur Steuerung des Roboters dienen, womit sich bereits eine weitere Arbeit befasst. Die Kombination von beidem bezeichnet man als *Motion Capture*.

Zur Aufzeichnung dieser Bewegungen gibt es verschiedene Techniken. Die zwei wesentlichen Techniken setzen voraus, dass Sensoren direkt am Kopf befestigt sind oder dass die Bewegungen von einem externen Gerät aufgenommen werden.

3.1. Oculus Rift

Das Oculus Rift Dev Kit 1 benutzt ein 3-Achsen-Gyrometer, um die Orientierung des Kopfes zu messen. Zusätzlich wird unterstützend ein Beschleunigungssensor genutzt, um *Driften* bei Rotationen herauszurechnen. Die Oculus Rift erreicht dabei eine Frequenz von einem kHz (siehe LaValle (2014)).

Ein Gyroskop ist ein Kreiselinstrument, das die Drehimpulserhaltung nutzt, um seine Orientierung beizubehalten. In den meisten Geräten wird jedoch kein mechanisch rotierender Kreisel verbaut, stattdessen wird ein Mikrosystem[1] verwendet, welches in Schwingung versetzt wird und sich bei Rotationen verändern.

Bei Beschleunigungssensoren gibt es eine gefederte Masse, welche bei Beschleunigung durch ihre Trägheit zurückhängt. Die Auslenkung der Masse kann durch die Spannung, die mittels dem piezoelektrischen Effekt entsteht, gemessen werden oder es wird die Änderung der Kapazität zwischen dem gefederten und dem festen Teil des Systems gemessen (siehe wikipedia.de (2014)).

[1] Im englischen als *MEMS* bekannt

3.2. Optisches System

Beim optischen Tracking System wird das zu trackende Objekt mit kleinen reflektierenden Kugeln bestückt. Um das Objekt werden Kameras positioniert, welche Infrarotlicht ausstrahlen. Dieses wird von den Kugeln reflektiert und von den Kameras aufgenommen. Durch die verschiedenen Perspektiven lässt sich damit die Position der Kugeln bestimmen. Bei Verwendung von mehreren Kugeln kann die Positionen zu einander genutzt werden, um zusätzlich die Orientierung zu errechnen. Die Frequenz hängt von den eingesetzten Kameras ab. 60 Hz wird als übliche Einstiegsbildwiederholungsrate angesehen.

Solch ein System ist in Kombination mit der Wellenfeldsynthese an der HAW verfügbar. Der Nachteil an einem optischen System ist der große Platzbedarf und die vielen zusätzlichen Geräte.

Seit dem Dev Kit 2 der Oculus Rift wird auch die Position im Raum erfasst. Dies geschieht durch eine Infrarotkamera, die vor dem Anwender angebracht ist (siehe Oculus VR Inc (2014)). Damit lassen sich seitliche Bewegungen in einem kleinen Bereich erfassen.

Ein weiteres Produkt dieser Kategorie ist die *Kinect* von Microsoft, welche eine Kamera beinhaltet, die mit einer Infrarotkamera für das *Tiefeempfinden* unterstützt wird. Die Kameras arbeiten mit einer Frequenz von 30 Hz (siehe Microsoft MSDN (2014)).

4. Recherche

Während die Kombination vom mechanischen und virtuellen Schwenken noch nicht in auffindbaren Projekten vertreten ist, gibt es viele Projekte, die sich auf eines von beiden beschränken. Auch die Hinzunahme einer VR-Brille zeigt sich in letzter Zeit immer häufiger.

4.1. Mechanisches Schwenken

Das mechanische Schwenken war lange Zeit die einzige Möglichkeit, den Fokus auf einen bestimmten Bereich zu richten. Es lässt sich mit zwei bis drei Servomotoren umsetzen.

Hierbei wird der zusätzliche Verschleiß, welcher weitere Wartungen nach sich zieht sowie die geringe Bewegungsgeschwindigkeit als problematisch angesehen. Des Weiteren ist die nötige Verkabelung, inklusive Stromversorgung, neben der Kamera als Nachteil zu nennen.

4.1.1. Überwachungskamera

Das bekannteste Beispiel für eine mechanisch schwenkbare Kamera ist sicherlich die Überwachungskamera, wie sie an vielen öffentlichen und privaten Orten genutzt wird. Festmontiert kann ein bestimmtes Gebiet eingesehen werden. Durch das Schwenken kann die Sicht variieren. Meist ist auch eine Zoomfunktionalität mit inbegriffen.

4.1.2. Schwenkbare Kamerahalterung für Drohnen

In einem Projekt der Norwegischen Universität für Wissenschaft und Technologie wurde eine hölzerne Kameraplattform mit zwei Servos und zwei angebrachten Kameras entwickelt (siehe Erik Hals (2014)). Die Halterung kann horizontal und vertikal schwenken, sich jedoch nicht um die Rollachse bewegen. Durch die zwei Kameras wird ein dreidimensionales Bild erzeugt. Angebracht an einer Drohne, soll die Halterung dem Piloten ermöglichen, ein sonst unerreichbares Gebiet zu erkunden. Als Übertragungskanal für die Bilddaten wird ein Empfänger und ein Sender vom Typ „*5.8GHz FPV AV 600mW RC832*“ angegeben.

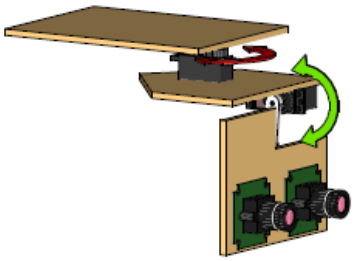

Abbildung 4.1.: Bild einer schwenkbaren Kamerahaltung. (Erik Hals (2014))

4.1.3. Oculus Crane

Bei dem Projekt *Oculus Crane* (siehe Thomas (2014)) wurde eine schwenkbare Kamerahalterung an einen Miniaturkran (siehe Abbildung 4.2) gebaut, um den Haken, und die daran hängenden Lasten, besser positionieren zu können. Auch diese vorliegende Funktionalität kommt dem in dieser Arbeit verwendeten Kamerakopf sehr nahe. Jedoch handelt es sich beim Gesamtaufbau nicht um ein verteiltes System, sodass viele Latenzen vermieden werden können.

Des Weiteren wurde ein Heads-up-Display (HUD) entworfen, das anzeigt, wie weit die *Laufkatze*[1] ausgefahren ist, einen Kompass für die Orientierung des Krans bietet, Neigungswinkel der Kamera anzeigt und darstellt, ob sich die Last auf oder ab bewegt (siehe Abbildung 12.5). Weitere Information zum HUD folgen im Abschnitt 12.2.

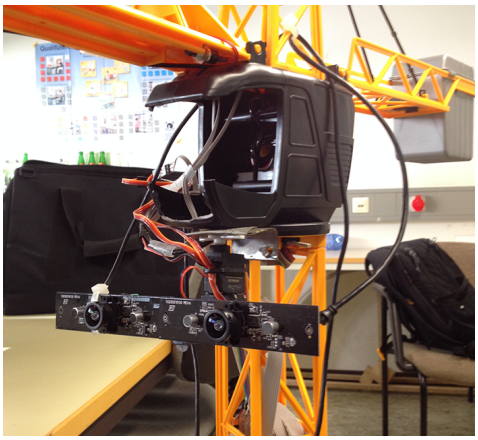

Abbildung 4.2.: Kamerahalterung vom *Oculus Crane* Projekt. (Thomas (2014))

[1] Bewegliches Bauteil am Kranarm, das den Haken führt.

4.1.4. Robot Ribbon Cutting

Bei der Eröffnung eines neuen Standortes des Roboterherstellers *Clearpath* wurde traditionell ein Band zerschnitten (siehe Baranov (2014)). Für diesen Akt wurde ein Telepräsenzroboter benutzt, sodass die Ehre des Zerschneidens einer entfernten Person zuteil werden konnte (siehe Abbildung 4.3). Für die Telepräsenz wurde die Oculus Rift genutzt, welche mit ihrem Head-Tracking-System den Kopf des Roboters steuerte. Um die Arme zu bewegen, wurde die steuernde Person mit der Microsoft Kinect aufgenommen, sodass Armbewegungen auf die des Roboters umgesetzt werden konnten. Als System wurde ROS eingesetzt. Für eine Kompatibilität zu der Kinect musste hierbei die Windows Portierung genutzt werden.

Gerade beim Interagieren mit der Umwelt und dem Bewegen von Armen ist eine hohe Latenz sehr hinderlich. Schwierigkeiten beim Zerschneiden des Bandes sind im veröffentlichten Video deutlich zu sehen.

Abbildung 4.3.: Zeremonielles Zerschneiden von einem Band zur Eröffnung eines neuen Standortes durch einen Roboter. (Baranov (2014))

4.2. Virtuelles Schwenken

Beim virtuellen Schwenken wird auf mechanisch bewegliche Teile verzichtet. Von dem, meist hohen, Aufnahmebereich wird dem Nutzer nur ein Ausschnitt gezeigt. Geschwenkt wird durch das Umsetzen des Bildausschnittes. Dies wird durch digitale Bildverarbeitung erreicht, was Rechenaufwand und damit Latenzen nach sich zieht. Gerade in Bereichen, welche keine stereoskopische Sicht benötigen, ist das virtuelle Schwenken häufig vertreten.

4.2.1. Parrot Bebop

Die Bebop Drohne von Parrot, welche im vierten Quartal 2014 erscheinen soll, besitzt eine einzelne Kamera mit Fischaugenoptik, wodurch ein sehr hoher Sichtwinkel erreicht wird. Die Verbindung wird mittels Wi-Fi, mit Unterstützung des 802.11ac Standards, hergestellt (siehe PARROT SA (2014)). Auf Grund der einzelnen Kamera, liegt das Bild nur in 2D vor. Die Drohne lässt sich mit der Oculus Rift koppeln, wobei die Kamera nicht mechanisch schwenkbar ist, sodass Bewegungen des HMDs ausschließlich virtuell umgesetzt werden können.

4.2.2. Rundumsicht für norwegische Panzerfahrer

In der norwegischen Stadt Rena hat das Militär testweise an einem Panzer eine Weitwinkelkamera pro Seite angebracht und den Fahrer mit einer Oculus Rift ausgestattet (siehe Urke (2014)). Die Panzerfahrer sind so in der Lage sich schnell umzusehen, während sie im Panzer sitzen. Des Weiteren werden dem Fahrer auf dem HMD Informationen wie eine Karte, die Orientierung sowie die aktuelle Geschwindigkeit angezeigt.

„Computer games have indeed helped inspire the new systems [..] with our software you can add the information and views you are used to from games“ Urke (2014)

4.2.3. Überwachungskamerasystem bestehend aus mehreren Kameras

In der Masterthesis *Virtual pan-tilt-zoom for a wide-area-video surveillance system* (siehe Sinn (2008)) wird die Überlegenheit vom virtuellen Schwenken und Zoomen gegenüber dem mechanischen Systemen diskutiert. Jedoch ist das Thema ein Überwachungssystem, wodurch einige Vorteile in Bezug auf den Telepräsenzroboter unwichtig erscheinen. So zum Beispiel die Möglichkeit mehrere Ausschnitte gleichzeitig anzeigen zu können. Auch wird die Implementierung auf einem FPGA umgesetzt, was aufzeigt, wie hoch der benötigte Rechenaufwand ist.

Als Kamerasystem fungierte hierbei das *MASIV* System, welches aus vier *Kameras* besteht, welche jeweils mehrere CMOS Bildsensoren einsetzen, um möglichst viele Bilddaten aufnehmen zu können. Das Interessante dabei ist, dass eine Kamera kein vollständiges Bild darstellt, sondern Lücken aufweist und man erst, wenn man alle vier Kameras zusammen legt, die gesamte Sicht erhält (siehe Abbildung 4.4). So wird ein 880 Megapixel Bild erzeugt, welches große Areale darstellen kann, wenn die Aufnahmen aus der Luft gemacht werden.

Ein Mosaik aus mehreren Kamerabildern wird auch von der *DARPA* (Defense Advanced Research Projects Agency) genutzt. Das sogenannte *ARGUS-IS* (Autonome Real-Time Ground Ubiquitous Surveillance Imaging System) ist ein Kamerasystem, welches aus 368 Smartpho-

nekameras besteht und ein 1.800 Megapixel Bild erzeugt. *ARGUS-IS* soll, an einer Drohne angebracht, Gebiete aufnehmen, welche sich dann von einer Bodenstation aus überwachen lassen. In einer Höhe von etwa 6 km lässt sich damit eine Fläche von 25 km^2 einsehen (siehe Anthony (2013)). Bei der Bodenstation wird der überwachte Bereich mit 12 Bildern pro Sekunde angezeigt. Genauere technische Infos sind nicht freigegeben, es wird aber angegeben, dass 600 Gigabits an Daten pro Sekunde erzeugt werden[2], wodurch ein Bild mehr als 6 Gigabyte groß ist.

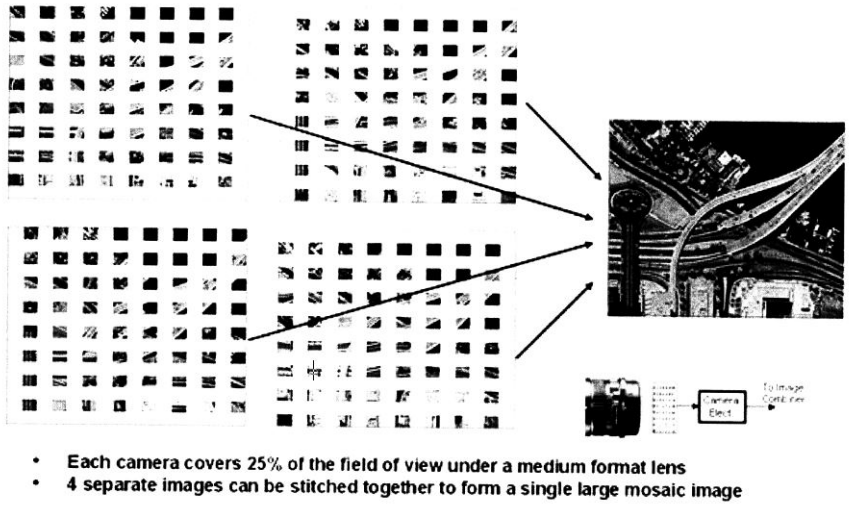

Abbildung 4.4.: Darstellung des *MASIV Kamerasystem*s, welches ähnlich einem Mosaik aus mehreren kleinen ein großes Bild erstellt. (Sinn (2008))

[2]Es ist unklar, ob sich diese enorme Datenmenge auf die aus der Luft aufgenommenen Bilder bezieht oder auf die Daten, welche auf der Bodenstation verarbeitet werden und unter Umständen weitere Informationen enthalten.

5. Systemaufbau

Das System besteht aus zwei Hauptkomponenten: dem Telepräsenzroboter (mobile Einheit) sowie dem Hostcomputer (Bedienerstation) mitsamt dem Benutzer (siehe Abbildung 5.1).

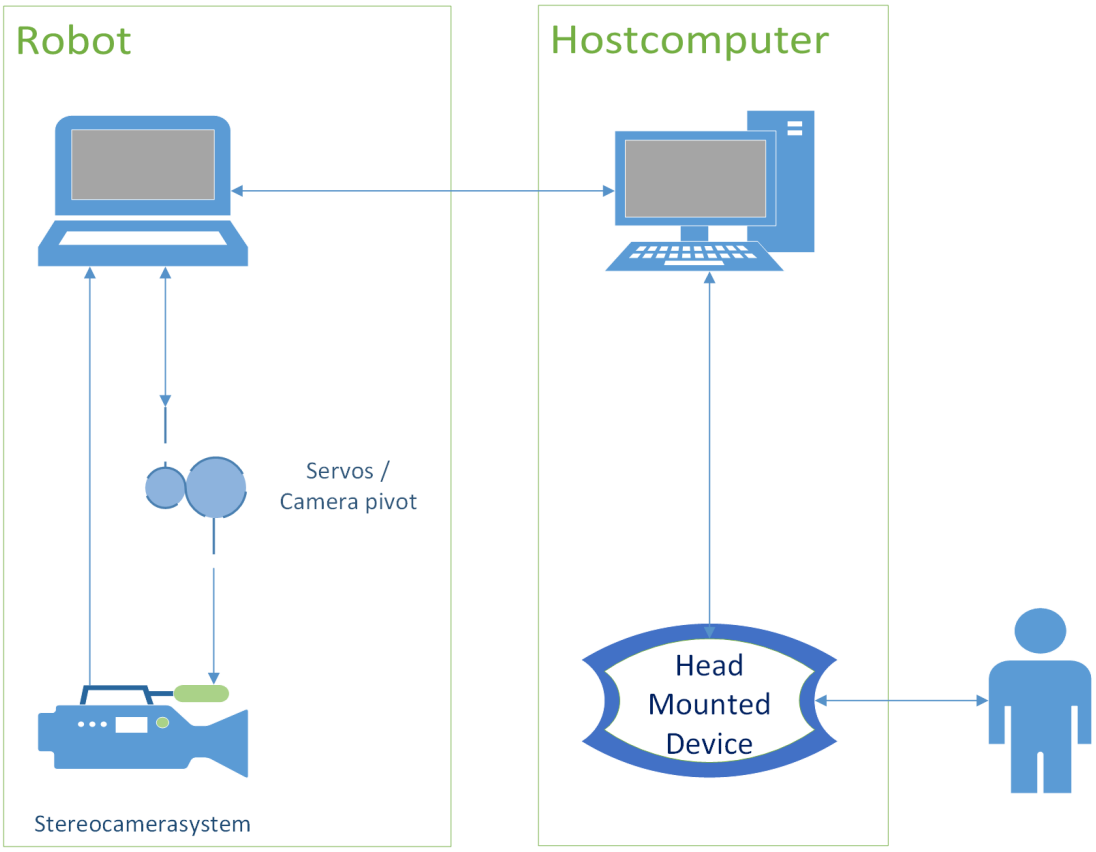

Abbildung 5.1.: Grober Systemaufbau

5.1. Genutzte Hardware

Die folgende Hardware stand an der HAW zur Verfügung.

- Roboterplattform: Pioneer P3-DX
- mechanisch schwenkbarer Kamerakopf (siehe Wiese (2014))
- Kamera: 2x Logitech HD Webcam C310
- Laserscanner: Hokuyo URG-04LX-UG01
- mobiler Computer: Zotac ZBox EI750 (i7-4770R)
- HMD + Head-Tracking-System: Oculus Rift Developer Kit 1
- stationärer Arbeitsplatzrechner: Intel Core i5-750, AMD RV730, 4 GB RAM
- Wireless Local Area Network (WLAN) Router: Asus RT-N66U

5.2. Telepräsenzroboter

Bei dem Telepräsenzroboter (siehe Abbildung 5.2) handelt es sich um einen fahrbaren Roboter auf Basis des Pioneer P3-DX. Der Pioneer P3-DX besitzt drei Räder, von denen zwei je einen Antriebsmotor haben und das letzte Rad frei drehbar ist. Somit kann der Roboter vorwärts, rückwärts und Kurven fahren sowie sich auf der Stelle drehen. Die maximale Drehgeschwindigkeit beträgt 300°/s (siehe adept mobilerobots (2011)). Die Stromversorgung wird mittels ein bis drei Bleiakkus sichergestellt, welche eine Spannung von 12 V bieten.

Auf dem Roboter befindet sich ein mobiler Computer, welcher über das lokale Netzwerk mittels ROS-Topics mit der Bedienerstation kommuniziert (siehe Abbildung 8.1). Hierbei handelt es sich um die ZBox EI750 Zotac. Dieser Mini-PC findet genügend Platz auf der Roboterplattform und ist mit dem WiFi Standard 802.11ac ausgestattet, welcher sich gut für die Verbindung zur Bedienerstation eignen sollte. Des Weiteren ist der Computer mit einem Intel Core i7-4770R ausgestattet, der genug Leistung für weiterführende Projekte liefert. Die ZBox EI750 wird mit 19 V Gleichspannung betrieben. Die Akkus des Pioneer P3-DX liefern eine Gleichspannung von 12 V. Dadurch wird der Computer aktuell mit einer Steckdose verbunden, wodurch der Operationsradius eingeschränkt ist. Ein Spannungswandler[1] soll dies lösen.

[1] Der Spannungswandler muss 12 V DC am Eingang akzeptierten und 19 V DC am Ausgang liefern. Das mitgelieferte Netzteil bietet einen Ausgangsstrom von bis zu 6,15 A. Der Spannungswandler sollte deshalb in der Lage sein, einen ähnlich hohen Strom zu liefern. Ein Computernetzteil 12 V KFZ Stecker ist als Lösung vorgesehen.

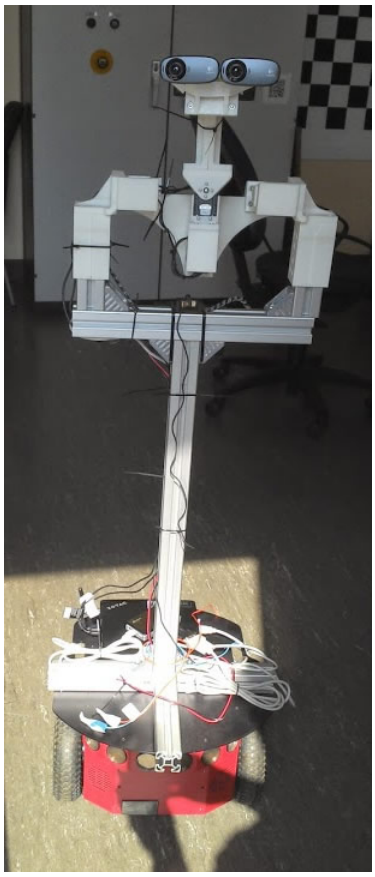

Abbildung 5.2.: Foto des genutzten Robotersystems

Hervorgehend aus der Arbeit von Herrn Wiese (siehe Wiese (2014)), wurde auf dem Roboter ein Kamerakopf verbaut (siehe Abbildung 5.3), welcher um alle drei Achsen im Raum schwenkbar ist. Dieser Kamerakopf ist mit einem stereoskopischen Kamerasystem, bestehend aus zwei *Logitech HD Webcam C310*, bestückt (siehe auch Kapitel 12.1 Kamerasystem).

Optional wird der Roboter zur Kartografie mit einem Laserscanner ausgestattet. Hierzu kann der vorhandene *Hokuyo URG-04LX-UG01* genutzt werden. Es handelt sich hierbei um einen Laserscanner mit einer Reichweite von 4 m und einer Abdeckung von 240° in der Horizontalen.

5.3. Bedienerstation

Die Bedienerstation ist die (stationäre) Basis, an der sich der Benutzer befindet. Sie besteht aus einem Computer, einem HMD und einem Head-Tracking-System.

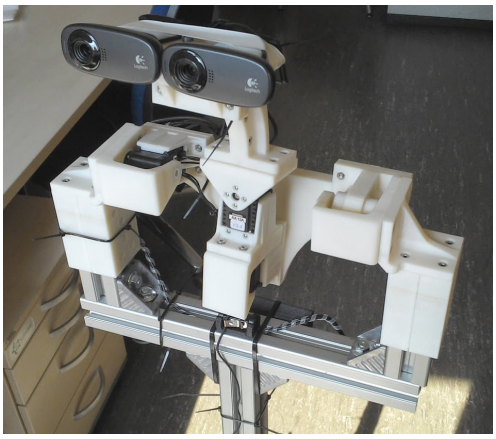

Abbildung 5.3.: Foto des genutzten Kamerasystems

Der Computer ist für die Verarbeitung und Anzeige von den Kamerabildern verantwortlich, wodurch das Ergebnis von hoher zur Verfügung stehender CPU-Leistung profitiert. Hierfür dient ein vorhandener Arbeitsplatzrechner, welcher mit einem Intel Core i5-750 ausgestattet ist.

Das HMD inklusive Head-Tracking-System ist mit dem Oculus Rift Dev Kit 1 gegeben und lässt sich über USB und HDMI mit dem Computer verbinden.

5.4. Übertragungsmedium

Zur Kommunikation war ein WLAN Netz angedacht, welches dem Roboter erlaubt, sich in einem bestimmten Bereich, zum Beispiel der Hochschule, frei zu bewegen.

Auf Grund fehlender Infrastruktur, beziehungsweise fehlender Erlaubnis, wurde vorerst nur eine kabelgebundene Ethernetverbindung genutzt. Für Messungen wurde der Roboter in ein WLAN Netz[2] eingebunden, welches ihm erlaubte, sich in einem Raum kabellos zu bewegen.

5.5. Systemkonfiguration

Auf beteiligten Computern wird ROS vorausgesetzt. Empfohlenes Betriebssystem dafür ist Ubuntu 12.04. Für die Installation der aktuellen ROS Version *Hydro* (Stand 2014-07-10) kann nach der Anleitung aus der ROS Wiki vorgegangen werden (siehe Open Source Robotics Foundation (2014)).

[2] Ausgehend von einem Asus RT-N66U

6. Latenz

Die Latenz (eng. delay) oder Verzögerung bezeichnet die Zeit, die vergeht, bis eine Reaktion auf eine Aktion folgt. Da der Roboter vollständig mobil, und somit kabellos, einsetzbar sein soll, ist mit spürbaren Latenzen bei der Telemetrie zu rechnen. Es gibt allerdings auch weitere Aspekte, welche Latenzen erzeugen. Die wesentlichen Latenzen entstehen bei der Datenübertragung zwischen den beiden Computersystemen, durch die Bearbeitungszeit der Bilder und mechanisch bedingt durch die träge Bewegung der Servomotoren im Kameragelenk.

Im folgenden Abschnitt werden die Latenzen näher beleuchtet und es wird dargelegt, was beim Systemaufbau bedacht wurde, sodass Latenzen zum Wohle der Immersion möglichst weitgehend reduziert bzw. kompensiert werden.

6.1. Mechanische Latenz

Die für die Kamerabewegung zuständigen Servomotoren haben eine maximale Rotationsgeschwindigkeit. Ein Mensch kann seinen Kopf sehr schnell drehen, sodass schnelle Bewegungen, auf Grund der limitierten Servogeschwindigkeit und der verwendeten Frequenz für Stellwinkel, nicht synchron umgesetzt werden können. Der Träger der Datenbrille würde nach der Kopfbewegung somit immer noch das gleiche Bild aus der alten Sicht sehen, welches nicht zur neuen Orientierung des Kopfes passt. Dies wirkt unnatürlich, trübt das Erlebnis und kann Übelkeit nach sich ziehen (siehe Orsini (2014)).

Die mechanische Latenz ist abhängig von den Geschwindigkeiten der verwendeten Servos sowie der Frequenz, mit der das Head-Tracking-System ausgelesen und als Stellwinkel an die Servomotoren vermittelt wird, und nicht zuletzt von den Kopfbewegungen des Benutzers.

Im benutzten Kameragelenk wurden 3 Servos der Bauart Dynamixel AX-12 von Robotis eingesetzt (siehe Wiese (2014)). Bei einer angelegten Spannung von 10 V benötigt dieser Servotyp 196 ms für eine Drehung von 60° (siehe ROBOTIS (2006)). Bei einer ruckartigen Kopfbewegung von 60° um eine Achse tritt dabei schon eine Mindestlatenz von fast 200 ms ein, bevor die Kamera ein Bild aus der neuen Orientierung aufnehmen kann. Um diesem entgegenzuwirken, wird ein virtuelles Schwenken und Rotieren benutzt (Siehe Kapitel 7 Virtuelles Rollen, Nicken und Gieren).

6.2. Verarbeitungszeit der Bilder

Um ein Bild, für die in diesem Projekt verwendete Oculus Rift, anzeigen zu können, muss das Bild, welches von der Kamera kommt, zunächst passend skaliert und für die Linsen tonnenförmig verzeichnet werden. Für das virtuelle Schwenken und Rotieren kommen noch einige weitere Bearbeitungen hinzu (Siehe Abbildung 9.2).

Die Aufgaben der Bildverarbeitung wurden mittels OpenCV umgesetzt. Um eine schnellere Verarbeitung zu erreichen, wurde das Graphics Processing Unit (GPU) Modul von OpenCV getestet, welches es erlaubt, Berechnungen auf die GPU auszulagern, sofern es sich dabei um eine GPU von NVidia mit Cuda Unterstützung handelt. Es stellte sich schnell heraus, dass die GPU in diesem Fall langsamer ist als die CPU. Der Grund dafür liegt darin, dass die Bilder bei der benutzten Developer Version der Oculus Rift nur sehr klein sein müssen, wodurch die Berechnungsdauer geringer ist als der Transport zum Speicher der GPU.

Die Latenz skaliert mit der Auflösung der Bilder, dem Verwenden vom virtuellen Schwenken und der Anzahl bzw. der Komplexität der zusätzlich angezeigten Elemente (siehe auch Abschnitt 12.2 Einblendbare Informationen) und hängt von der zur Verfügung stehenden CPU Leistung ab.

6.3. Latenz durch die Datenübertragung

Das verwendete System basiert auf ROS und die Kommunikation der einzelnen Knoten läuft über ROS-Topics. Diese benutzen standardmäßig *TCPROS*, welches auf TCP/IP aufbaut. Die Nachrichten werden innerhalb eines Local Area Network (LAN) verschickt.

In Bezug auf die verschickten Bilder ist dies ein essentieller Flaschenhals des Systems. Bei farbigen Bildern werden die Pixel in drei Farbkanälen gespeichert. Die übliche Farbtiefe von 24 Bit (8 Bit pro Kanal) bedeutet, dass ein Bild in VGA-Auflösung[1] schon ca. 0,9 MB groß ist[2]. Der Bildtransport von ROS erlaubt die Komprimierung von Bildern (siehe David Gossow (2012)), wodurch die Größe sehr stark verringert werden kann. Jedoch ist dadurch beim Sender zusätzlicher Aufwand für das Kodieren und beim Empfänger für das Dekodieren notwendig. Auch Daten abseits von Bildern müssen für den Transport aufbereitet und beim Empfänger *entpackt* werden. Dies erzeugt eine zusätzliche, wenn auch minimale, Latenz.

Damit die Latenzen des Netzwerkes das Bewegungsempfinden nicht stören, wird die Bildbearbeitung mit dem virtuellen Schwenken und Rotieren auf demselben Computer betrieben, auf dem auch die VR-Brille mit dem Head-Tracking-System angeschlossen ist. Somit werden die

[1] VGA-Auflösung entspricht 640x480 Pixel
[2] 640*480*3 Byte = 921.600 Byte

Kopfbewegungen praktisch verzögerungsfrei an den ROS-Node übertragen, welcher für die Darstellung verantwortlich ist.

Die Latenz bei der Datenübertragung ist abhängig von dem eingesetzten Netzwerk, der Größe bzw. Komprimierung der Bilder und der benutzen Frequenz für die Veröffentlichung der Daten.

6.4. Weitere Latenzen

Des Weiteren haben Displays eine Latenz bestehend aus der Zeit, die benötigt wird, um das Eingangssignal zu verarbeiten und es auf dem Bildschirm auszugeben (siehe Thiemann (2014)) sowie der Zeit die Pixel benötigen, um ihre Farbe zu ändern. Letzteres liegt bei dem Oculus Rift Dev Kit 2 bei etwa 2-3 ms (siehe Oculus VR Inc (2014)). Diese Latenzen können durch Anwendersoftware und Hardware nicht verändert werden und werden deswegen im weiteren Verlauf nicht weiter einkalkuliert, sind also bei nachfolgend genannten Zeiten aufzuschlagen.

Auch ist vorgesehen, dass Linsenverzeichnungen der Kameraaufnahmen korrigiert werden. Dies geschieht durch einen weiteren ROS-Node, der zwischen Kameratreiber und weitergehender Software sitzt. Da die Korrektur auf demselben Computer wie die Aufnahme stattfindet, ist mit keiner erheblichen Latenz bei der Datenübertragung zu rechnen. Die Latenz ist dadurch weitestgehend auf die CPU-Zeit der Bildkorrektur zurückzuführen.

Latenzen entstehen immer bei dem Betrieb von Sensorik. Die Daten eines Sensors werden meist zyklisch mit einer festen Frequenz abgefragt. In der Zeit zwischen zwei Datensätzen ist der Zustand ungewiss. Wird beispielsweise die Kamera mit einem Hz betrieben (ein Bild pro Sekunde), so kann bis zu eine Sekunde vergehen, bevor ein Objekt vor der Kamera auch auf dem Bild auftaucht. Eine Erhöhung der Frequenz verringert die Latenz, jedoch sind Sensoren hierin meist begrenzt, außerdem entsteht durch die Verarbeitung der höheren Datenmenge eine erhöhte CPU-Last.

6.5. Maßnahmen in der Implementierung

Die Priorität wurde in die Bewegung gelegt. Das essentielle Ziel ist es, dass der Benutzer seine Kopfbewegungen möglichst verzögerungsfrei auf dem HMD visualisiert bekommt. Deshalb werden die Sensordaten, die die Orientierung des Kopfes beinhalten, mit einer deutlich höheren Frequenz als andere Daten veröffentlicht.

$\text{Frequenz}_{HMDOrientierung}$ = 120 Hz

$\text{Frequenz}_{KameraBilder}$ = $\text{Frequenz}_{KameraOrientierung}$ = 30 Hz

Der Bildverarbeitungsknoten wird dauerhaft aus den gegebenen Daten ein Bild bereitstellen. Abhängig von der benutzten CPU wird hierbei eine Rate von ca. 20 Bildern pro Sekunde erreicht. Dies bedeutet, dass nicht alle Bilder und Sensordaten verwendbar sind.

Die Verzögerung, bis eine Bewegung des Kopfes sichtbar wird, beinhaltet durch virtuelles Schwenken nur noch folgende Punkte (siehe Abbildung 6.1):

1. Zeit bis die Sensordaten vom Head-Tracking-System ausgelesen werden.
2. Zeit bis die Daten über den ROS-Topic versendet und empfangen wurden (dies passiert auf einem Computer).
3. Zeit bis aus den vorliegenden Daten ein Bild generiert wird und es angezeigt wird (Bildverarbeitung).

Die Hauptlatenz wird bei der Bildbearbeitung vermutet.

6.6. Messmöglichkeiten

Die Gesamtlatenz, die bestimmt werden soll, ist die Zeit, die vergeht, bis eine Bewegung des Kopfes auf dem HMD zu sehen ist. Um festzustellen, wie groß die Latenz letztendlich ist und wie sie sich im Vergleich zum rein mechanischen Schwenken verändert hat, werden folgende Methoden in Betracht gezogen.

6.6.1. Zeitstempel (Timestamp)

Wenn Nachrichten in einem ROS Netz veröffentlicht werden, können diesen Nachrichten Zeitstempel hinzugefügt werden. Dadurch lässt sich berechnen, wie alt Sensordaten in etwa sind. Auch lässt sich im Programmverlauf jederzeit die aktuelle Systemzeit auslesen.

Die Systemzeit auf einem Linuxbetriebssystem hat für gewöhnlich eine Auflösung von wenigen Millisekunden. Die Frequenz lässt sich im Kernelheader einsehen. Listing 6.1 zeigt, dass die Bedienerstation eine Timerfrequenz von 250 Hz hat, wodurch die Uhr alle 4 ms gestellt wird. Typische Plattformen bieten jedoch auch eine *Real Time Clock*, wodurch Zeiten genauer als eine Mikrosekunde dargestellt werden können, sodass Abweichungen, durch die benötigte Auflösung, vernachlässigbar sind.

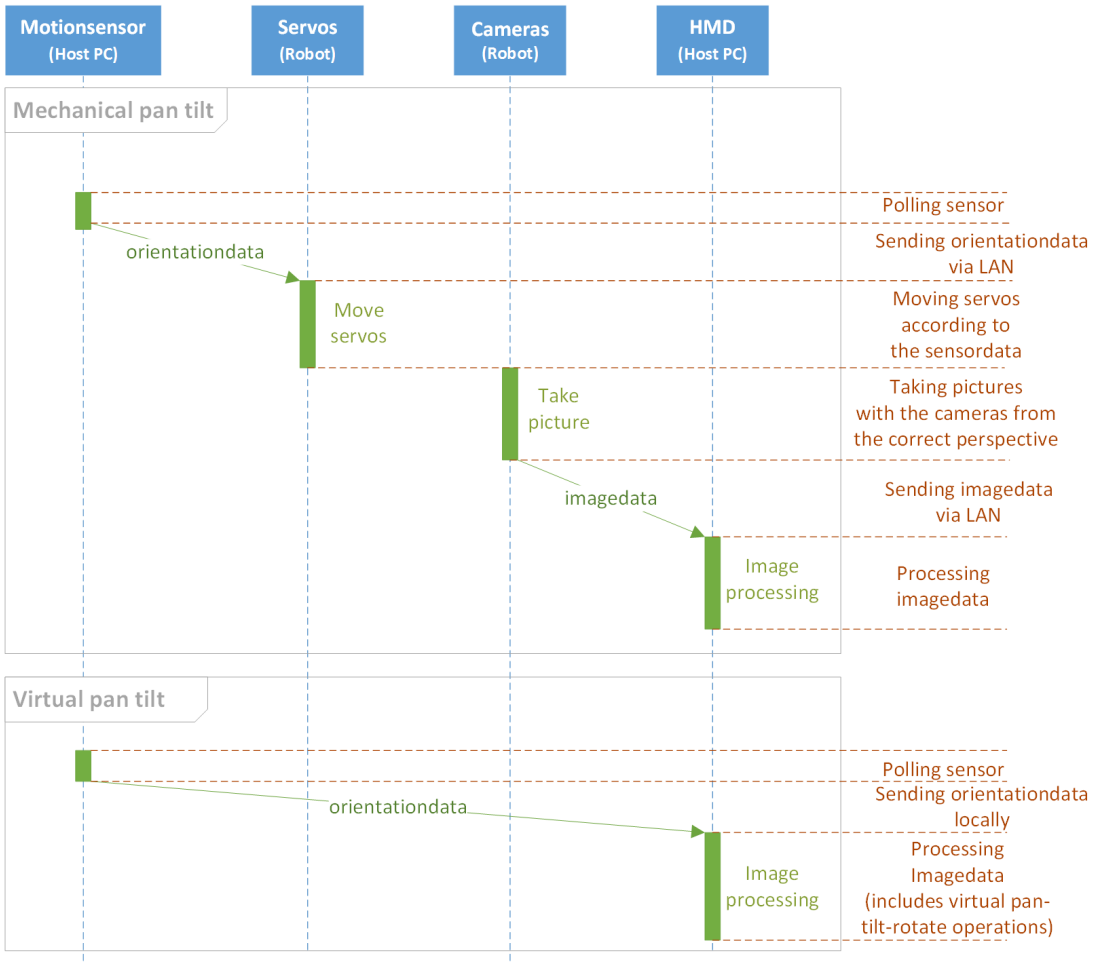

Abbildung 6.1.: Latenzen von und zwischen ROS-Nodes beim mechanischen und virtuellen Schwenken

```
$ cat /boot/config-`uname -r` | grep HZ
CONFIG_NO_HZ=y
CONFIG_RCU_FAST_NO_HZ=y
CONFIG_HZ_250=y
CONFIG_HZ=250
CONFIG_MACHZ_WDT=m
```

Listing 6.1: Frequenz der Systemzeit auf der Bedienerstation

Sollten mehrere Computer beteiligt sein, ist eine Zeitsynchronisation der Geräte Voraussetzung. Dies kann mittels dem Programm *chrony* (siehe Curnow (2014)) sichergestellt werden. Da die Sensordaten vom Head-Tracking-System jedoch auf dem Computer der Bedienerstation erstellt werden, auf dem letztendlich auch die Bildverarbeitung stattfindet, ist dies in diesem Fall nicht nötig.

Für den zeitlichen Verlauf der latenzwirksamen Nachrichten siehe auch Abbildung 6.1.

Virtuelles Schwenken

Die Sensordaten für die Orientierung des HMDs werden mit konstantem Takt veröffentlicht. Diese Frequenz ist höher gewählt als die Verarbeitungsfrequenz. Dadurch werden nicht alle Sensordaten empfangen und bearbeitet, jedoch ist für jedes neue, für das HMD bereitgestellte, Bild eine möglichst aktuelle Orientierung verfügbar (siehe Abbildung 6.2).

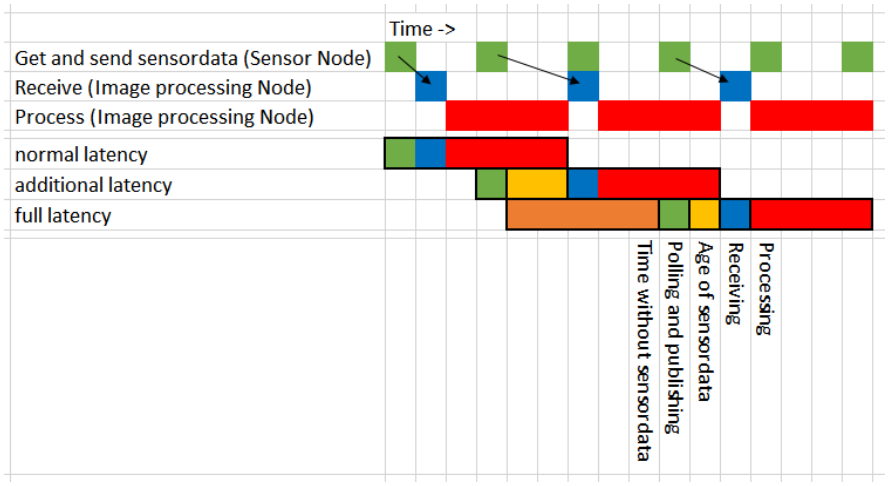

Abbildung 6.2.: Zeitlicher Verlauf der Sensordaten beim virtuellen Schwenken mit aufgezeigten Latenzen

In Abbildung 6.2 sind folgende drei Latenzen aufgezeigt:

- *normal latency*
  In dieser Latenz ist die Zeit enthalten, die benötigt wird, um die Sensordaten von der Oculus Rift auszulesen, diese zum Bildverarbeitungsknoten zu übertragen, auf ein Bild anzuwenden und es anzuzeigen.

- *additional latency*
  Hierbei wird zusätzlich noch bedacht, dass genutzte Sensorwerte nicht hochaktuell sind, sondern schon eine bestimmte Zeit[3] im Buffer bereit liegen können.

- *full latency*
  Bei der vollen Latenz wird auch noch die Zeit hinzugezählt, in der keine Sensordaten vorlagen, also die Lücke zwischen zwei genutzten Datensätzen.

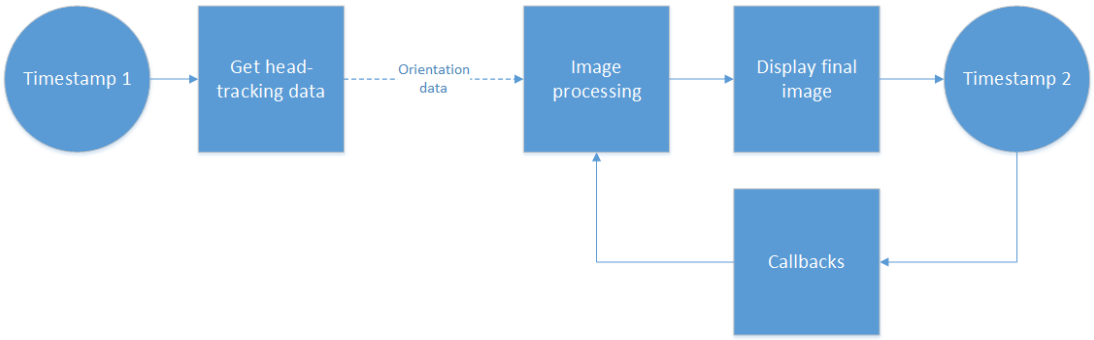

Abbildung 6.3.: In der Zeitstempelmessung enthaltene Prozesse beim virtuellen Schwenken. Durchgezogene Pfeile zeigen unmittelbare Übergänge; gestrichelte Pfeile zeigen Datentransporte.

Um die „additional latency“ zu erhalten, wird vor dem Entnehmen eines Sensorwertes (*Timestamp 1* in Abbildung 6.3) ein Zeitstempel erstellt. Dieser wird mit der Zeit verglichen, die vorliegt, nach dem der Sensorwert zum Bilderverarbeitungsknoten übertragen und daraus ein neues Bild erstellt und angezeigt wurde (*Timestamp 2* in Abbildung 6.3). Für die „full latency“ wird die Zeit vom Sensorwert mit der Zeit verglichen, zu der das übernächste Bild angezeigt wurde.

Nochmals anzumerken ist, dass dies ausschließlich die zeitliche Verzögerung der Bewegung darstellt, das zu sehende Kamerabild kann dabei wesentlich älter sein.

[3] maximal 1/Frequenz

Mechanisches Schwenken

Da man hierbei ein verteiltes System vorliegen hat, muss auf die Synchronisation der verschiedenen Systemzeiten geachtet werden. Da die Latenz der Kamerabewegungen jedoch von den Kopfbewegungen abhängt, wurde beschlossen, alle auftretenden Latenzen (siehe Abbildung 6.1) einzeln zu messen.

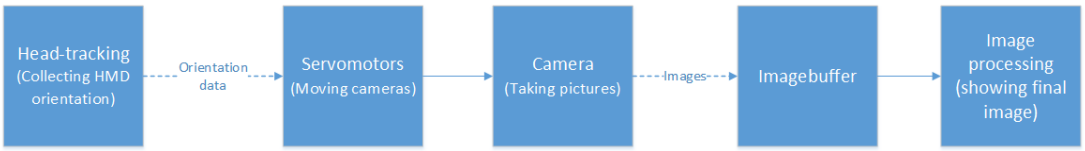

Abbildung 6.4.: Ablauf beim mechanischen Schwenken. Durchgezogene Pfeile zeigen unmittelbare Übergänge; gestrichelte Pfeile zeigen Datentransporte.

Abbildung 6.4 zeigt den Ablauf beim mechanischen Schwenken. Messungen sollten ergeben, wie lange in den einzelnen Prozessen verweilt wird und wie viel Zeit der Datentransport zwischen den Prozessen benötigt. Um die Datenübertragung in einem verteilten System zu messen, können die Daten vom Empfänger zurück zum Absender geschickt werden. Dadurch müssen die Zeitstempel nur noch auf einem System gemacht werden. Die Zeit, die vergeht bis der ursprüngliche Sender die Daten zurückbekommt, entspricht etwa der doppelten Übertragungszeit.

6.6.2. Hochgeschwindigkeitskamera

Für die Zeitmessung per Kamera benötigt man ein Modell, das in der Lage ist sehr schnell Bilder aufzunehmen. Für eine Millisekunden genaue Messung müssen mindestens 1000 Bilder pro Sekunden aufgenommen werden. Mit einer üblichen Videokamera, die 60 Bilder pro Sekunde macht, ist eine Genauigkeit von etwa 17 Millisekunden[4] möglich.

Für diese Messtechnik werden ein Monitor, der die Bilder für das HMD anzeigt, sowie das Head-Tracking-System so positioniert, dass sie beide auf der Kameraaufnahme zu sehen sind. Nach Start der Aufnahme wird das getrackte HMD etwas gedreht, wodurch sich das Bild für das HMD nach kurzer Zeit auf dem Monitor anpassen sollte. Auf dem aufgenommenen Video wird anschließend ausgezählt, wie viele Bilder zwischen der Bewegung vom Head-Tracking-System und der Bewegung auf dem Monitor liegen. Anhand der Bildanzahl und der Bildrate der Aufnahme lässt sich die Latenz errechnen.

#frames * 1000 / FPS = latency in milliseconds

[4] 1 Bild / 60 Bilder/s = 16,667 ms

Diese Messmethode kann gleichermaßen für virtuelles und mechanisches Schwenken genutzt werden, jedoch ist es bei der Versuchsdurchführung schwierig, das HMD mit gleicher Geschwindigkeit und im gleichen Winkel zu bewegen.

6.6.3. Oculus Latency Tester

Oculus VR bietet selbst ein Gerät an, um Latenzen zu messen (siehe Oculus VR Inc (2013)), welches man mit dem Dev Kit 1 nutzen kann und welches im Dev Kit 2 direkt verbaut ist (siehe Oculus VR Inc (2014)). Das Gerät wird statt einer Linse in die Oculus Rift eingebaut und scannt kontinuierlich den Bildschirm. Ein Knopfdruck schickt ein Signal über USB zum Computer. Die Software soll beim Empfangen dieses Signals ein bestimmtes Muster auf dem Bildschirm darstellen. Der Latency Tester erkennt das Muster und zeigt die Zeit inklusive Latenzen des Displays an, die vergangen ist, seit dem Drücken des Knopfes bis zum Erscheinen des Musters. Im übertragenen Sinne bedeutet dies, dass sich die Zeit messen lässt, die benötigt wird, um auf Signale der Oculus Rift, wie sie das Head-Tracking-System der Oculus Rift liefert, zu reagieren, bzw. bis sie für den Benutzer ersichtlich sind. Damit eignet sich die Messmethode für das virtuelle Schwenken, nicht jedoch für das Gesamtsystem mit mechanischem Schwenken. Auch muss bedacht werden, dass die Reaktion auf das Messsignal implementiert werden muss und dass das Zeichnen des Musters zusätzlichen Rechenaufwand mit sich zieht.

7. Virtuelles Rollen, Nicken und Gieren

Das virtuelle Rollen (eng. roll), Nicken (eng. pitch) und Gieren (eng. yaw) (siehe Abbildung 7.1) soll die mechanischen Bewegungen virtualisieren, um trotz langsamer Mechanik eine möglichst latenzfreie Reaktion auf Bewegungen zu bieten. Hierzu werden Kopfbewegungen virtuell durch Bildverarbeitung vorgenommen, bevor die Kamera sich tatsächlich bewegt. Um dabei weiterhin möglichst viel vom Bild darstellen zu können, sollte die Kameraaufnahme einen hohen Sichtwinkel haben (Siehe auch Abschnitt 12.1 für die Anforderungen an das Kamerasystem).

Der mechanische Kamerakopf arbeitet dabei weiter wie zuvor. Vorausgesetzt wird dabei, dass die Stellungen der Servomotoren übermittelt werden. Das virtuelle Schwenken wird bei dieser Umsetzung nur als Unterstützung genutzt und ersetzt das mechanische Schwenken im aktuellen System nicht[1].

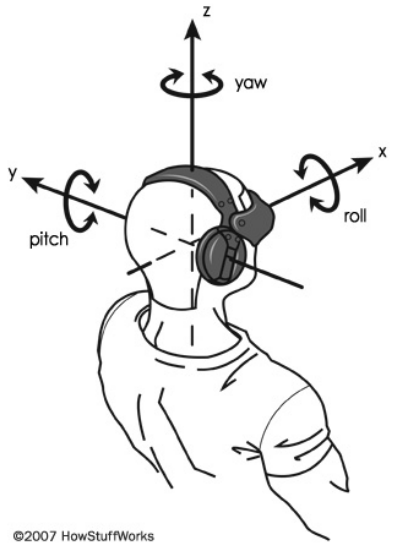

Abbildung 7.1.: Position der drei Rotationsachsen. (Strickland (2014))

[1] Die Implementierung würde dies problemlos unterstützen.

7.1. Rollen

Das Rollen bezogen auf den menschlichen Kopf bezeichnet das Neigen vom Kopf in Richtung der Schultern.

Um diese Bewegung bei einem starr aufgenommenem Bild nachzuvollziehen, wird das Bild um dessen Mittelpunkt rotiert (siehe Abbildung 7.2). Da es sich um ein Stereobild, bestehend aus zwei nebeneinander aufgenommenen Bildern, handelt, wird beim Rotieren diese stereoskopische Sicht zerstört. Da das Rollen, im Gegensatz zu den anderen beiden Bewegungsarten, für eine Umsicht im Raum weniger wichtig ist, und meist nur in einem sehr geringen Gradbereich passiert, wird dieser Mangel toleriert.

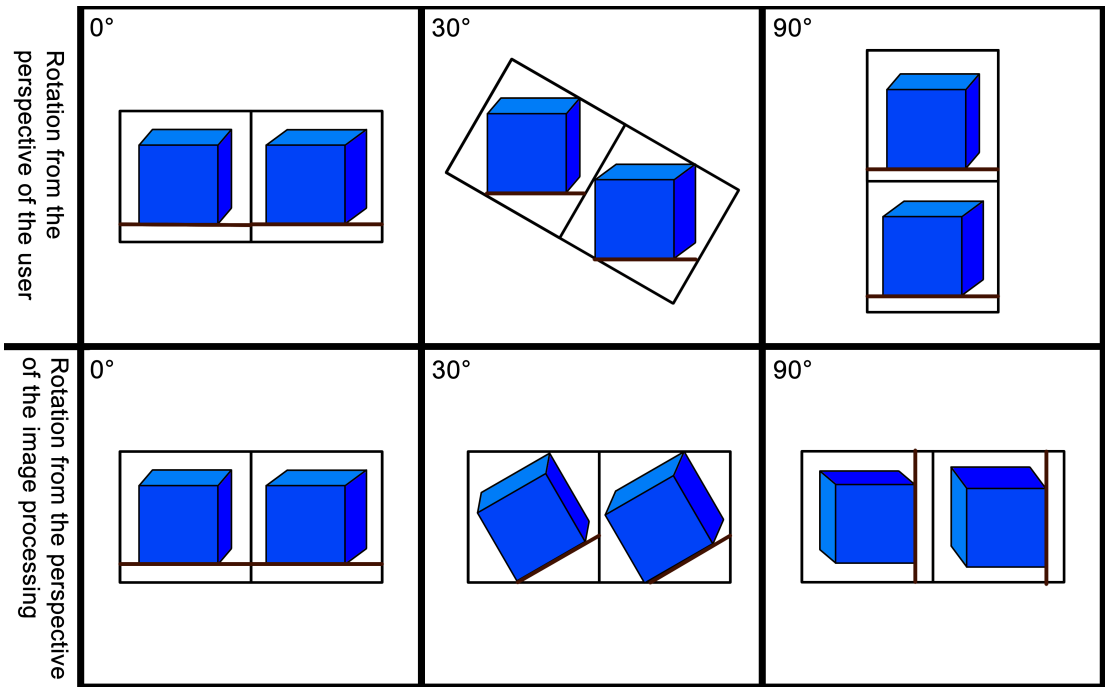

Abbildung 7.2.: Funktionsweise des virtuellen Rollens durch Rotation

7.2. Nicken und Gieren

Beim Nicken wird das Bild vertikal und beim Gieren horizontal verschoben (siehe Abbildung 7.3). Je nach Objektiv, und damit Sichtwinkel der Kamera, wird nur der Bildausschnitt (Region of Interest (ROI)) verschoben. Bei einem kleinen Sichtwinkel der Kamera beinhaltet dies auch, den Ausschnitt aus dem Aufnahmebereich herauszuschieben, wodurch der Benutzer schwarze

Bereiche sieht. Um die Bewegung des Benutzers besser nachempfinden zu können, wird dieser Kompromiss eingegangen.

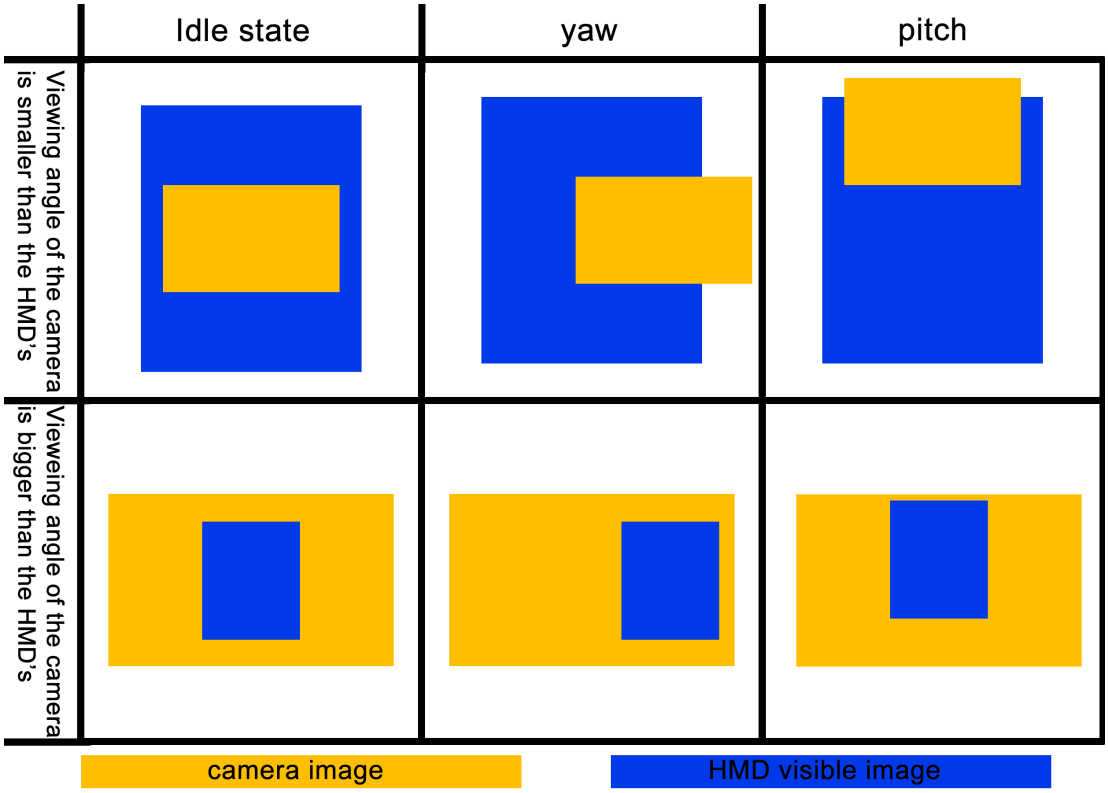

Abbildung 7.3.: Funktionsweise des virtuellen Schwenkens in horizontaler und vertikaler Richtung

7.3. Latenzbegriff

Wichtig anzumerken ist, dass die kompensierte Latenz beim virtuellen Schwenken ganz allein die Latenz der Bewegungen betrifft.

Es kann in keinem Fall die Latenz von Bildaufnahmen reduziert werden, eher das Gegenteil trifft, durch erhöhte CPU Last, zu. Latenzen der Bilder, verursacht durch die Übertragung, bleiben erhalten. So kann beispielsweise einem Hindernis nicht ausgewichen werden, weil das angezeigte Bild nicht die aktuelle Situation am Roboter darstellt, sondern die Situation von vor wenigen Millisekunden bis Sekunden[2], sodass schnelles Reagieren erschwert wird.

[2] Abhängig von der Übertragungsgeschwindigkeit der Bilder, dem Aufnahmetakt und der Zeit, welche für die Bildverarbeitung benötigt wird.

Ziel vom virtuellen Schwenken ist die Reduzierung der Latenz bei Bewegungen. Ein Schwenken des Bildes kann schnell durch das Umgehen von Intersystem-Kommunikation umgesetzt werden. Dies wirkt sich zum einen auf das Erlebnis des Benutzers aus, indem es *Motion Sickness* verhindert und zum anderen kann sich der Benutzer, passende Kameraoptik vorausgesetzt, unmittelbar vor Ort umsehen.

8. Programmkonzept

In diesem Kapitel werden angedachte Ideen und das Konzept für die Implementierung genannt. Das entwickelte Programm soll aus mehreren ROS-Nodes bestehen, die miteinander kommunizieren und auf den stationären und den mobilen Computer verteilt sind (siehe Abbildung 8.1). Der Vorteil eines solchen modularen Systems ist, dass Schnittstellen klar definiert sind und dadurch einzelne Komponenten leicht ausgetauscht werden können. So lässt sich zum Beispiel das Head-Tracking-System problemlos austauschen, solange die Signatur der Schnittstelle eingehalten wird.

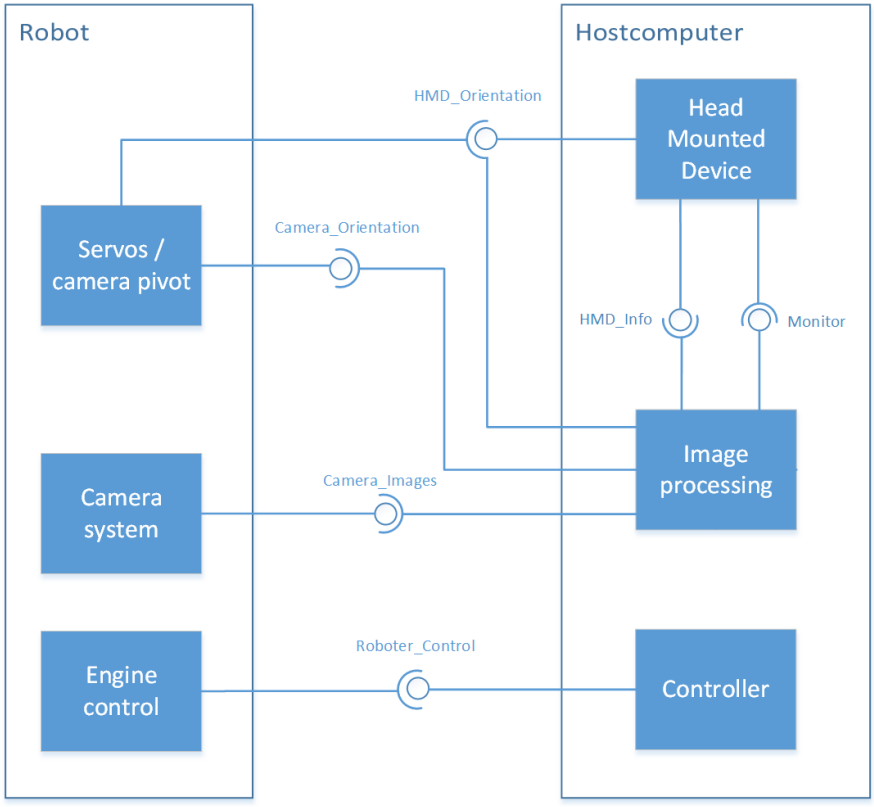

Abbildung 8.1.: Überblick der relevanten ROS-Nodes und deren ROS-Topics (Namen entsprechen nicht denen der Implementierung)

8.1. Bildverarbeitung

Das Programm zur Bildverarbeitung muss folgende Funktionen liefern:

- Bilder passend skalieren und positionieren
- virtuelles Schwenken anwenden
- Elemente des HUDs zeichnen
- Bild für die HMD Linsen verzeichnen

Bei der genutzten Oculus Rift, und auch anderen HMDs, wird nur ein Display eingesetzt. Um trotzdem jedem Auge eine separate Anzeige zu bieten, werden den Augen je eine Hälfte des Displays zugeteilt. Daher müssen das linke und das rechte Bild zum Anzeigen nebeneinander in ein großes Bild kopiert werden, welches sich über das gesamte HMD streckt.

8.2. HUD

Exemplarisch sollen Informationen über die Ansicht gelegt werden. Dies geschieht als sogenanntes Heads-up-Display (HUD). Hierzu muss eine Berechnung gefunden werden, die die Position für das linke und das rechte Auge ausgibt, um ein stereoskopisches Bild darzustellen.

Der Einfachheit soll vorerst nur ein Text eingeblendet werden. Zur Orientierung und Navigation soll eine Karte das Kernstück des HUDs bilden. Die Karte soll später mittels Laserscanner aufgezeichnet werden, vorerst wird sie aus *Dummy-Daten* erzeugt.

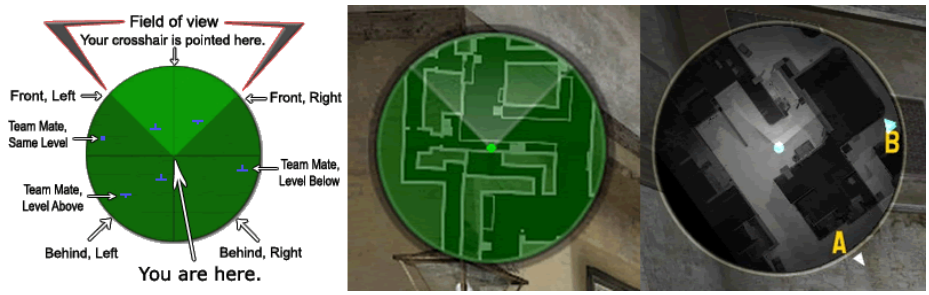

Abbildung 8.2.: Darstellung einer Karte in einem Egoshooter. Links: Erklärung des Radars aus dem Spiel Counter Strike (whisper.ausgamers.com (2006)). Mitte: Radar aus dem Spiel Counterstrike ProMod (http://media.moddb.com/ (2014)). Rechts: Radar aus dem Spiel Counter Strike Global Offensive (eigene Darstellung per Screenshot).

In Videospielen werden derartige Umsetzungen meist als Minimap dargestellt. Diese befinden sich überwiegend in einer der oberen Ecken des Sichtfeldes. Im gezeigten Beispiel (siehe Abbildung 8.2) passiert dies als Radar. Die Position des Spielers befindet sich im Zentrum mit Sicht immer nach oben.

Eine weitere in Spielen häufig vertretene Art der Karte kann man in Abbildung 8.3 sehen. Hierbei wird die Karte transparent über das komplette Bild gelegt und der Spieler steht praktisch in oder auf der Karte. Auch hierbei lässt sich die Karte mit fester Orientierung oder mit der Sichtrichtung immer nach oben zeichnen. Diese Darstellung eignet sich jedoch besser für Spiele aus der dritten Person als für Spiele aus der Egoperspektive.

Abbildung 8.3.: Karte aus dem Spiel Diablo II LOD von Blizzard (eigene Darstellung per Screenshot)

Beide Karten haben gemein, dass der Spieler das Zentrum bildet und dass die Karten durch weitere Infos wie Points of Interest (POIs) erweitert werden können. Des Weiteren gibt es die Möglichkeit einen fertigen Datensatz zu benutzen oder die Karte erst mit Erkundung der Umgebung aufzudecken.

Abseits von Videospielen sind HUDs auch vom Militär (beispielsweise Pilotenhelme) und in modernen Autos, mit der Windschutzscheibe als Displayersatz, bekannt. Im Allgemeinen lässt sich sagen, dass Informationen größtenteils an den Rändern des Sichtfeldes platziert werden, um die Sicht möglichst wenig einzuschränken. Ausnahme ist hierbei Augmented Reality (siehe

Abschnitt 12.2.2), welches auch direkt im Zentrum der Sicht Verwendung findet, da es mit den angezeigten Bildern agiert.

8.3. Odometrie

Mit Hilfe der Odometrie ist es möglich die Drehung und Position des Roboters anzugeben. Hierzu werden die Antriebsräder verwendet. Durch Bekanntheit des Radumfangs kann anhand der Steuersignale des Motors die zurückgelegte Strecke geschätzt werden. Durch getrennte Betrachtung der beiden Antriebe werden Rotationen erkannt, wodurch die Fahrtrichtung bestimmt werden kann, was die Angabe von x und y Koordinaten möglich macht. Es ist nur ein Schätzen, da die Werte ungenau sind und leicht verfälscht werden können. Auf ungeeigneter Oberfläche können die Räder beispielsweise leicht durchdrehen oder rutschen, sodass die gelieferten Werte mit der Zeit immer weiter vom eigentlichen Wert abweichen (Driften).

Die Informationen der Odometrie wurden für folgende Funktionen angedacht:

Zum Einen wird die Orientierung, genauer die Rotation um die Yaw-Achse, genutzt, um die absolute Ausrichtung des Kamerakopfes vom Roboter zu entkoppeln. Dies ist vergleichbar mit einem Menschen, der seinen Körper dreht, seinen Kopf, und damit seinen Blick, aber weiterhin auf ein Motiv fixiert hält. Wenn sich der Roboter also links herum dreht, wird die entgegengesetzte Bewegung (rechts herum) vom Kamerakopf umgesetzt, sodass die Sicht sich nicht verändert.

Zweitens wird die Odometrie zur Kartografie genutzt. Mit Hilfe eines Laserscanners kann, mittels Simultaneous Localization and Mapping (SLAM) und der Position und Orientierung des Roboters, eine Karte der Umgebung erstellt werden, in welcher, dank der Koordinaten der Odometrie, die Position des Roboters eingetragen werden kann.

9. Programmumsetzung

Basierend auf dem zuvor erläuterten Konzept, wurde ein lauffähiges Programm implementiert. Im folgenden Kapitel werden Schritte und Erkenntnisse der Entwicklung beschrieben.

9.1. Genutzte Software

- Betriebssystem: Ubuntu 12.04.3 LTS
- Kommunikation: ROS Hydro mit diversen ROS-Nodes (siehe Abschnitt 9.2)
- Bildverarbeitung: OpenCV 2.4.8
- Oculus Rift SDK Version 0.2.5

9.2. Verwendete ROS-Nodes

Folgende ROS-Nodes wurden für das System verwendet. Für eine genauere Spezifikation der Schnittstellen siehe Anhang B. Für die Verbindungen der Knoten siehe auch Abbildung D.6.

- oculus_cam_viewer: Bildverarbeitung und Anzeige für das HMD.
- oculus_driver: Treiber des Head-Tracking-Systems der Oculus Rift.
- headunit: Steuerung der Kopfeinheit.
- map2image: Wandelt eine Karte vom SLAM Node in ein Bild um und passt dabei den Ausschnitt und die Größe an.

9.2.1. Standard-Nodes

Des Weiteren wurden auch noch einige ROS-Nodes benutzt, welche zum Standardrepertoire von ROS gehören.

- usb_cam: Veröffentlicht die Bilder der USB-Kamera.

- image_proc: Rechnet Linsenverzeichnungen der Kamerabilder raus.
- joy_node: Bietet Controllerunterstützung (zum Steuern des Roboters).
- urg_node: Treiber für den vorhandenen Laserscanner.
- gmapper: Erstellt eine Karte aus den Daten eines Laserscanners mit Zunahme von Positions- und Orientierungsdaten (SLAM).
- RosAria: ROS Umsetzung der ARIA Bibliothek, welche Unterstützung für verschiedene Roboterplattformen, wie dem Pioneer 3DX, bietet.

9.2.2. Mess-Nodes

Um einzelne Kommunikationswege im System zu messen, wurden weitere ROS-Nodes geschrieben. Diese Programme werden im Normalbetrieb nicht benötigt, sondern dienen nur zu Testzwecken.

- dummy_map_publisher: Erzeugt Kartendaten aus einem Bild. Ersetzt den Laserscanner und gmapper bevor der Laserscanner in Betrieb genommen wird.
- measure_image_proc_delay: Bestimmt den zusätzlichen, zeitlichen Aufwand der Bildkalibrierung.
- measure_image_transfer: Misst die Übertragungszeit eines Kamerabildes im Netzwerk.
- measure_orientation_transfer: Misst die Übertragungszeit von Orientierungsdaten im Netzwerk.

9.3. Programmverlauf

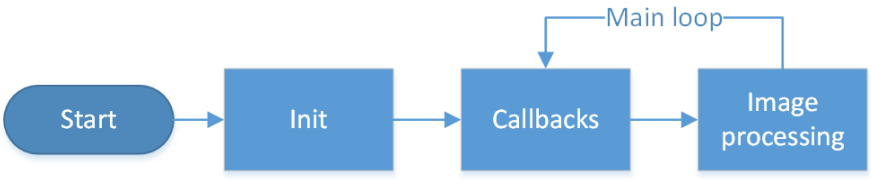

Abbildung 9.1.: Ablauf des Hauptprogramms auf der Bedienerstation

Abbildung 9.1 zeigt die Hauptschleife für das Programm zur Bildanzeige. Da der ROS-Node, der für das Verarbeiten und Anzeigen der Bilder zuständig ist, der Kernpunkt der Arbeit ist, wird sich im folgenden Text hauptsächlich darauf bezogen.

Die in Abbildung 9.1 zu sehenden *Callbacks* sind Funktionen, die einem Empfangskanal zugewiesen wurden. Wird der ROS-Node dazu aufgefordert, werden alle Nachrichten in der Empfangs-*Queue* abgearbeitet, indem die zuvor definierten Funktionen mit den Nachrichten als Parameter aufgerufen werden.

Die interne ROS Message *Queue*-Größe für Nachrichten der Orientierung der Oculus Rift wird beim Empfänger des Bildverarbeitungsknotens auf eins gestellt, sodass nur der neueste Datensatz verfügbar ist und die alten Datensätze, die nicht schnell genug verarbeitet wurden, verfallen. Für die Bilder und die Orientierung der Kamera wird die *Queue*-Größe auf zwei gestellt, dadurch können die zwei neuesten Datensätze mit ihrem Zeitstempel gepuffert werden, sodass für die Bildbearbeitung immer zwei Bilder und eine Orientierung gefunden werden können, die zeitlich zueinanderpassen (siehe Abbildung D.1). Dabei wird weniger Wert darauf gelegt, wie aktuell die Daten der Kameraeinheit sind, es wird bloß dafür gesorgt, dass Bilder und die dazu gehörige Orientierung eine stimmige Einheit bilden (zeitlich gesehen)[1]. Zu Beginn der Bildverarbeitung kann ein Bild dadurch maximal zwei Zyklen[2] alt sein, zuzüglich der Übertragungszeit.

Für den Fall, dass eine stereoskopische Kamera oder zwei industrielle Kameras benutzt werden, die sicherstellen, dass beide Bilder gleichzeitig aufgenommen werden, ist die Synchronisation der beiden Bilder in der Software nicht mehr nötig. Dies kann einen Performancevorteil nach sich ziehen. Jedoch muss bedacht werden, dass eine gleichzeitige Aufnahme nicht bedeutet, dass die Bilder auch sicher synchron übertragen und empfangen werden, sodass nach wie vor eine Art Software Synchronisation stattfinden sollte (siehe Abschnitt 12.1).

[1] Wenn dem Nutzer zwei Bilder gezeigt werden, die sich von der Aufnahmezeit stark unterscheiden, stört dies die Immersion und erzeugt insbesondere bei Bewegungen Nebeneffekte.

[2] 2 Bilder / 30 Bilder pro Sekunde = 66,667 ms

9.4. Bildverarbeitung

Die Bilder der Kamera werden beim stationären Computer gemäß dem virtuellen Schwenken transformiert, für die Linsen in der Occulus Rift verzeichnet und anschließend nebeneinander in einem Fenster dargestellt. Diese Schritte werden sequentiell am Bild vorgenommen und wurden mit OpenCV umgesetzt. OpenCV ist eine freie Bibliothek mit Funktionen für Bildverarbeitungen mit dem Schwerpunkt auf *Computer Vision*, wodurch es als geeignet erschien.

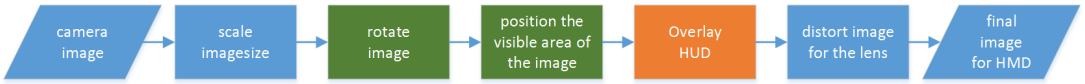

Abbildung 9.2.: Schritte der Bildverarbeitung. Grüne Prozesse werden für das virtuelle Schwenken und Drehen benötigt und sind somit optional; orange Prozesse sind für das HUD und ebenfalls optional.

Abbildung 9.2 zeigt die Verarbeitungskette. Da diese Kette zweimal (je einmal für das linke sowie das rechte Auge) vorgenommen werden muss, wird ein weiterer Thread gestartet, sodass beide Bilder parallel verarbeitet werden können. In der vorliegenden Implementierung wird bei jedem Durchlauf ein Thread mittels der *pthread* Bibliothek erstellt. Für eine Codeoptimierung, wäre ein Recyclen des Threads denkbar.

Im ersten Schritt wird die Auflösung des Kamerabildes an das Display vom HMD angepasst. Dies passiert durch eine Skalierung des Bildes. Der Skalierungsfaktor (siehe Listing 9.1) ist neben der Auflösung von der Kamera und dem HMD auch von deren Sichtwinkel abhängig. Wenn Kamera und HMD denselben Sichtwinkel besitzen, soll das Bild dieselbe Breite[3] wie das HMD annehmen. Bei einer Kamera mit hohem Sichtwinkel wird das Bild breiter als das HMD skaliert und bei niedrigem Sichtwinkel schmaler.

```
float scale = (displayWidth/2 / imageWidth)
            * (cameraViewingAngle / hmdViewingAngle);
```

Listing 9.1: Berechnung des Skalierungsfaktors für das Kamerabild in c++.

Optional wird anschließend das Bild rotiert. Dies passiert um den Mittelpunkt des Bildes. Der Winkel entspricht dem Roll-Winkel vom Head-Tracking-System abzüglich des Winkels, der beim Kamerakopf bereits um die Roll-Achse anliegt.

[3] Es wird davon ausgegangen, dass das Display horizontal geteilt wird, sodass jedem Auge eine Hälfte zur Verfügung steht. Wenn von der Breite des HMDs gesprochen wird, bezieht sich das unter Umständen auf eine Hälfte.

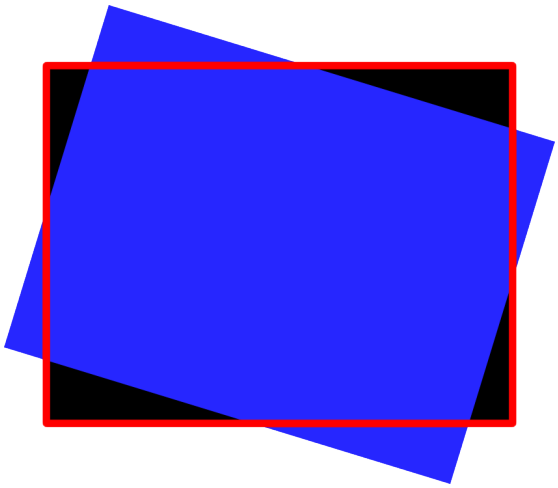

Abbildung 9.3.: Informationsverlust beim Rotieren. Das rote Rechteck markiert den zur Verfügung stehenden Bildbereich; das blaue Rechteck zeigt das rotierte Bild.

Da beim Rotieren die Bildgröße nicht verändert wird, gehen Informationen verloren (siehe Abbildung 9.3), weil die Ecken nicht mehr in das Bild passen. Aus diesem Grund wird vor dem rotieren der Bildbereich vergrößert.

Das Rotieren ist eine Matrixoperation mit relativ hohem Aufwand, aus diesem Grund wurde eine Schwelle eingeführt, die festlegt, ab welchem Winkel eine Rotation stattfindet. Der Wert dafür kann freigesetzt werden (siehe Anhang B Schnittstellen der ROS-Nodes) und liegt aktuell bei drei Grad. Dadurch wird dieser Schritt in der Bildverarbeitungskette übersprungen, wenn nur kleine oder langsame Bewegungen, bei denen die Mechanik schnell genug nachkommt, stattfinden.

Abseits vom virtuellen Schwenken müssen die Bilder mittig[4] für die Augen platziert werden. Die Errechnung der Augenposition kann durch die vorhandenen Informationen des HMDs, welche der Treiber der Oculus Rift (siehe Anhang B Schnittstellen der ROS-Nodes) liefert, vorgenommen werden (siehe Listing 9.2).

[4]Dies entspricht in der Horizontalen nicht zwangsläufig der Mitte des halben Displays.

```
lensSeparationDistancePixel = lensSeparationDistance
                            / horizontalScreenSize * displaySize.width;

eyeOffsetXRight = lensSeparationDistancePixel >> 1;
eyeOffsetXLeft = (displaySize.width>>1) - eyeOffsetXRight;

// relative to the HMD width
eyeOffsetXRight /= (displaySize.width>>1);
eyeOffsetXLeft  /= (displaySize.width>>1);
```

Listing 9.2: Berechnung des Bildmittelpunkts bzw. der Augenposition in c++

Da das Bild größer sein kann als der vorgesehene Bereich vom HMD und durch Hinzunahme des virtuellen Schwenkens das Bild teilweise oder komplett außerhalb des Sichtbereiches liegen kann, muss das Bild vor dem Platzieren noch passend zugeschnitten werden. Dies passiert, indem eine ROI definiert und auf das Bild angewendet wird.

Der Winkel für das virtuelle Schwenken ist die Differenz von der Drehung um die *Pitch* bzw. *Yaw* Achse des Head-Tracking-Systems mit der des Kamerakopfes. Um von den Winkeln in Grad auf eine Verschiebung in Pixeln zu schließen, wird der Quotient von der Breite bzw. Höhe des HMDs in Pixeln und dem horizontalen bzw. vertikalen Sichtwinkel genutzt (siehe Listing 9.3).

```
    // input: img = scaled camera image
    int posX = (displaySize.width*eyeOffsetX - img->cols) / 2;
    posX += (int)((HMDOrientation.yaw - cameraOrientation.yaw)
            * ((displaySize.width>>1)/hmdViewingAngleHorizontal));

    int posY = (displaySize.height - img->rows) >> 1;
    posY += (int)((HMDOrientation.pitch - cameraOrientation.pitch)
            * (displaySize.height/hmdViewingAngleVertical));

    cv::Mat dst(displaySize.height, displaySize.width>>1,
                img->type(), cv::Scalar(0, 0, 0));
    drawImageOnImage(img, &dst, posX, posY);
```

Listing 9.3: Berechnung der Bildposition beim virtuellen Schwenken in c++. Für die *drawImageOnImage* Funktion siehe Anhang C.

Letzter Verarbeitungsschritt, bevor das komplette Bild, welches das linke und das rechte Bild vereint, angezeigt werden kann, ist die Verzeichnung. Da die Linsen in der Oculus Rift die Sicht beim Hindurchsehen kissenförmig verzeichnen, muss der gegenteilige Effekt durch eine

Tonnenverzeichnung (eng. *barrel distortion*) auf die Bilder angewandt werden (siehe Abbildung 9.4).

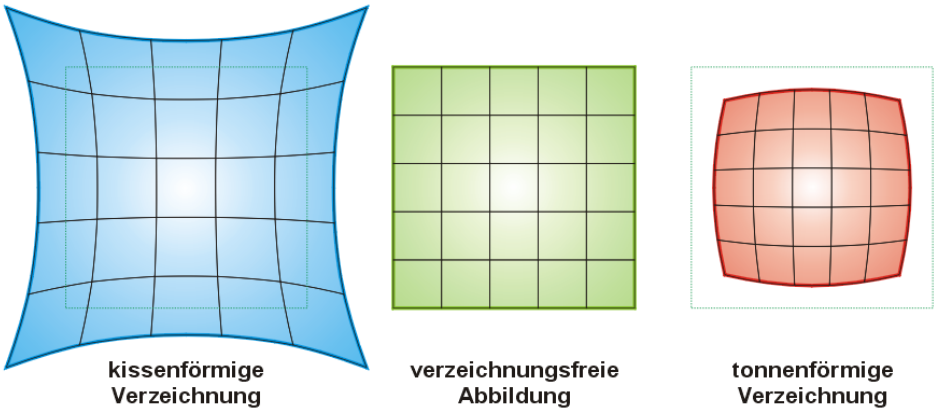

Abbildung 9.4.: Darstellung der Kissenverzeichnung (links) und der Tonnenverzeichnung (rechts). (Fantagu (2009))

Für die Tonnenverzeichnung sind eine Reihe von Koeffizienten nötig, die den Grad der Verzeichnung angeben. Diese werden der *HMD-Info* entnommen, welche der Oculus Driver liefert (siehe Anhang B Schnittstellen der ROS-Nodes). Des Weiteren muss das Zentrum der Verzeichnung angegeben werden. Dies ist die Position der Augen bzw. die Position des Linsenmittelpunktes und ist bereits aus dem vorhergehenden Schritt bekannt (siehe Listing 9.2).

Eine radiale Verzeichnung entspricht dem Umsetzen von Bildpunkten, abhängig von der Entfernung zum Zentrum der Verzeichnung. Bei der Tonnenverzeichnung wird das Bild nahe des Zentrums vergrößert. Diese Vergrößerung nimmt mit steigender Entfernung, gemäß der Koeffizienten, ab. Vereinfacht durch diese Formel[5]: $r_{dst} = K_0 + K_1 * r_{src}^2 + K_2 * r_{src}^4$

Für die Umsetzung (siehe Listing 9.4) wurde die *remap* Funktion von OpenCV benutzt. Dabei werden zwei Matrizen mit der gleichen Größe wie das Bild angelegt. In einer Matrix werden die X-Koordinaten und in der anderen die Y-Koordinaten von dem entsprechenden Bildpunkt geschrieben. Bei dem *Remapping* wird ein neues Bild angelegt, bei dem die Pixel sich wie folgt zusammensetzen:

$Pixel_{x,y} = Bild_{org}(map_x(x,y), map_y(x,y))$

Wobei:

[5] Mit r_{src} als Vektor/Entfernung vom Mittelpunkt zum Pixel im Quellbild; r_{dst} als Vektor neuer Länge in dieselbe Richtung, der die Position des Pixels im neuen Bild anzeigt und K_0, K_1, K_2 als Koeffizienten.

$Pixel_{x,y}$ = Pixel des neuen Bildes an der Position x,y
$Bild_{org}(x, y)$ = Pixel des original (unverzeichneten) Bildes an der Position x,y
$map_x(x, y)$ = Wert der X-Matrix an der Position x,y
$map_y(x, y)$ = Wert der Y-Matrix an der Position x,y

In Listing 9.4 ist die Implementierung zu sehen. Diese Funktion läuft unabhängig davon, ob virtuelles Schwenken benutzt wird und ist eine weitere Stelle, auf die bei Geschwindigkeitsoptimierungen ein besonderes Augenmerk gelegt werden sollte, da dieses die größte Zeit bei der Bildverarbeitung beansprucht (siehe auch Abbildung 10.2 von der Latenzmessung). Dies liegt daran, dass die Funktion mittels zwei geschachtelter Schleifen über alle Pixel rüberläuft, was bei dem Oculus Rift Dev Kit 1, das linke und das rechte Bild zusammen gezählt, einer Million Pixeln[6] entspricht. Bei jedem Pixel werden einige *floating point* Operationen vorgenommen. Da die Berechnungen für jeden Pixel erst einmal unabhängig sind, eignet sich diese Aufgabe hervorragend für eine Parallelisierung.

```
    cv::Mat* mapx = new cv::Mat(img->rows, img->cols, CV_32FC1);
    cv::Mat* mapy = new cv::Mat(img->rows, img->cols, CV_32FC1);

    float* pbufx = (float*)mapx->data;
    float* pbufy = (float*)mapy->data;

    const float unit_xr2 = (img->cols/2) * (img->cols/2);
    const float unit_yr2 = (img->rows/2) * (img->rows/2);

    for (int y = 0; y < img->rows; y++) {
        for (int x = 0; x < img->cols; x++) {
            const float r2 = (x-Cx)*(x-Cx)+(y-Cy)*(y-Cy);
            const float r2x = r2 / unit_xr2;
            const float r4x = r2x * r2x;
            const float r2y = r2 / unit_yr2;
            const float r4y = r2y * r2y;

            *pbufx = Cx + (x - Cx) * (k0 + k1 * r2x + k2 * r4x) * scale;
            *pbufy = Cy + (y - Cy) * (k0 + k1 * r2y + k2 * r4y) * scale;

            ++pbufx;
            ++pbufy;
        }
    }

    cv::Mat dst;
    cv::remap(*img, dst, *mapx, *mapy, CV_INTER_LINEAR);
```

Listing 9.4: Tonnenverzeichnung in c++

[6] 1280 Pixel * 800 Pixel = 1.024.000 Pixel

9.5. Odometrie

Durch *Aria* wird Odometrie für den Pioneer P3-DX direkt unterstützt und lässt sich mittels des ROS-Nodes *RosAria* problemlos einbinden. Die dadurch zur Verfügung stehenden Daten können von zukünftigen SLAM Programmen verwendet werden.

Die X- und Y-Koordinaten sowie die Orientierung um die *yaw*-Achse werden genutzt, um in der zur Verfügung gestellten Karte den passenden Bereich für das HUD auszuwählen. Auch der angedachte ROS-Node *gmapping*, welcher aus Daten des Laserscanners eine Karte erstellt, benötigt diese Informationen zur Orientierung des Roboters. Jedoch wird hierbei ein anderer Datentyp erwartet, sodass ein weiterer ROS-Node zur Konvertierung dazwischen geschaltet werden muss. Dies ist im Verlauf dieser Arbeit noch nicht geschehen, sodass *Dummy-Daten* für die Karte verwendet wurden.

Der *fixierte Blick*[7] ließ sich teilweise umsetzen. Hierfür wurde dem ROS-Nodes für die Kopfeinheit und für das virtuelle Schwenken neben der Orientierung vom Head-Tracking-System auch die Orientierung des Roboters mitgeteilt, sodass nur um den Wert der Differenz geschwenkt wird. Probleme gab es, wenn der Roboter sich weiter als 90° drehte. Wenn eine Drehung des Roboters erfolgt, wird jedoch vom Anwender auch eine Veränderung der Sicht erwartet. Dieses Feature wurde deshalb ausgeschaltet und lässt sich optional einschalten (siehe Anhang B Schnittstellen der ROS-Nodes).

9.6. HUD

Genau so, wie es schon bei der vorhergehenden Bildverarbeitung passiert ist, können mittels OpenCV weitere Bilder und auch Texte problemlos auf dem schon vorhandenen Bild platziert werden.

9.6.1. Positionierung

Für eine korrekte Sicht müssen die Objekte für beide Augen auf der Horizontalen korrekt positioniert werden. Als Grundsatz kann gesagt werden, dass Objekte, die auf beiden Augen dieselbe Position haben, weit weg sind. Um so weiter sie auseinander gehen, desto höher ist der 3D Effekt[8] und desto näher sind sie. Dabei muss das Objekt auf dem rechten Auge weiter links positioniert werden.

[7] Wenn sich der Roboter dreht, dreht sich der Kamerakopf in die entgegengesetzte Richtung, sodass sich die Sicht der Kameras nicht verändert.

[8] Vergleiche die verschobene Position von Objekten aus einem 3D-Film, die das Gefühl vermitteln, direkt vor der Nase zu schweben, bei Abnahme der 3D-Brille.

Auch muss darauf geachtet werden, dass die Informationen nicht zu weit entfernt dargestellt werden. Denn es ist damit zu rechnen, dass sich Objekte vor der Kamera bis auf wenige Dezimeter nähern. Wenn man beispielsweise vor einer Wand steht, die 50 cm entfernt ist und eine Information einen Meter entfernt anzeigt, wird die Sicht nicht mehr eindeutig.

Abbildung 9.5 zeigt die für die Berechnung nötigen Größen, um ein Objekt so in der Horizontalen zu positionieren, dass es von beiden Augen korrekt erfasst wird. Hierbei wird das Sichtfeld als gerade Wand dargestellt und nicht als gekrümmter Schirm, wodurch Objekte weiter weg vom Sichtzentrum erscheinen, als es möglicherweise der Fall ist.

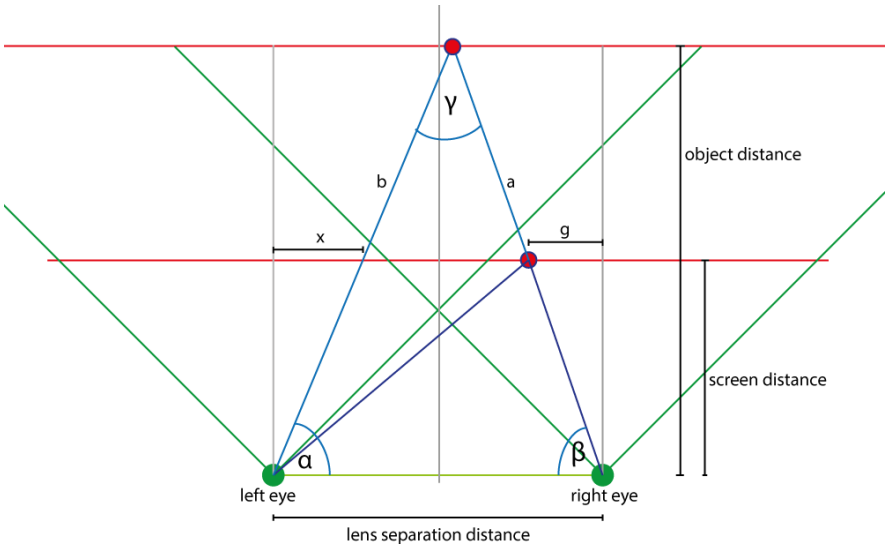

Abbildung 9.5.: Geometrische Darstellung der Sicht eines HMDs mit notwendigen Größen zum Berechnen der Position für beide Augen (Skizze in Eigenarbeit)

An Hand von Abbildung 9.5 wurde eine Funktion geschrieben, die aus der gegebenen Position für das eine Auge, die Position für das zweite Auge errechnet, abhängig von der gewünschten Objektentfernung. Dies ist in Listing 9.5 zu sehen.

```
float g = rightEyeCenter - posRight;    // can be negative
float beta = atan(abs(g) / screenDistancePixel);
if(g >= 0) {
    beta = (90.0f/180.0f*M_PI) - beta;
} else {
    beta = (90.0f/180.0f*M_PI) + beta;
}

float a = screenDistancePixel / sin(beta) * distanceFactor;

float b = sqrt  (a*a + c*c -2*a*c*cos(beta));

float alpha = acos((a*a - b*b - c*c) / (-2*b*c));

float tmp = (90.0f/180.0f*M_PI) - alpha;
if(alpha > 90) {
    tmp = alpha - (90.0f/180.0f*M_PI);
}
float x = screenDistancePixel * tan(tmp);

float posLeft = leftEyeCenter + x;
if(alpha > (90.0f/180.0f*M_PI)) {
    posLeft = leftEyeCenter - x;
}
```

Listing 9.5: Implementierung der Positionierung in c++. Code zeigt den Ablauf, um aus der Position des rechten Bildes auf die Position des linken zu schließen.

Abbildung 9.6.: Implementierte Ansicht des HUDs.

9.6.2. Text

Mittels der von OpenCV zur Verfügung gestellten Funktion *putText*, ist es möglich einen String auf eine gewünschte Position auf einem Bild zu zeichnen.

Um gleichzeitig einen Einblick der Performance zu erhalten, wurde die Bildrate als Textinhalt gewählt. Diese wurde in der linken oberen Ecke platziert (siehe Abbildung 9.6).

9.6.3. Karte

Weil der Roboter zu Anfang der Entwicklungszeit noch kabelgebunden war, und somit das Herumfahren und das Aufzeichnen der Umgebung mittels Laserscanner in der Priorität herabgesetzt wurde, wurde ein ROS-Node geschrieben, der die Funktionalität des Laserscanner und SLAM Nodes *gmapping* ersetzt, indem er ein Bild einer Karte einliest und im Format einer Karte veröffentlicht. Ein Austauschen der ROS-Nodes ist somit ohne Änderung der Schnittstelle möglich.

Die Karte wird als Matrix dargestellt, welche intern als eindimensionales Array gespeichert wird. Das Array ist mit acht Bit Werten gefüllt, wobei der Wert null bedeutet, dass es sich um Luft bzw. laserdurchlässiges Material handelt. Der Wert 100 bedeutet, dass der Laserstrahl auf ein Hindernis getroffen ist; alle Bereiche über die keine Informationen vorliegen werden mit einer 255 markiert. Das wären beispielsweise Bereiche außerhalb der Reichweite oder die hinter einer Wand liegen. Dieses Datenformat entspricht einem schwarzweiß Bild[9] mit acht Bit Farbtiefe, wodurch ein einfaches Kopieren des Speicherbereiches für die Konvertierung reichte. Die Koordinaten der Odometrie entsprechen dabei dem Mittelpunkt des Ausschnittes, während die Größe des Kartenausschnittes frei gewählt werden kann. Aus der Karte den passenden Bereich zu wählen, ist vergleichbar mit dem virtuellen Schwenken bei Aufnahmen mit hohem Sichtwinkel und passiert deshalb auf dieselbe Weise. Da der ROS-Node für die Bildverarbeitung die Karte in der gewünschten Auflösung erwartet, wird dem Bild der Karte ein schwarzer Rand hinzugefügt, sodass selbst am äußeren Rand der Karte noch ein Ausschnitt in der gewünschtes Größe bereit gestellt werden kann. Das Veröffentlichen dieses Bildes geschieht mit einer Frequenz von einem Hz und passiert auf der mobilen Einheit.

In dem letzten ROS-Node (Bildverarbeitung), der den Kartenausschnitt verwendet und anzeigt, werden noch weitere Modifikationen vorgenommen. Zum einen wird das Bild der Karte entsprechend des Head-Tracking-Systems rotiert, anschließend wird mit zwei Linien das Sichtfeld angezeigt, um zu verdeutlichen, welcher Bereich der Karte aktuell sichtbar ist. Dies wird nicht schon vom vorherigen ROS-Node gemacht, um die Latenzen bei der Rotation

[9] Beziehungsweise einem Bild mit einem beliebigen Farbkanal.

gering zu halten. Zum anderen wird ein Pfeil (in Form eines Dreiecks) in das Zentrum der Karte gezeichnet, welcher so gedreht wird, dass er die Orientierung um die Yaw-Achse des Roboters relativ zum Kamerakopf (und damit der Sicht) angibt.

Beim Einfügen der Karte in das Gesamtbild werden zwei Masken[10] kombiniert, um Transparenz zu erzeugen. Eine Maske entspricht der Karte selbst. Dadurch werden freie Bereiche nicht gezeichnet, sodass nur noch Hindernisse und unbekannte Bereiche zu sehen sind. Die zweite Maske wird einmalig erzeugt und entspricht einem Kreis. Hierdurch wird die Karte rund gezeichnet (siehe Abbildung 9.6).

[10] Eine Maske gibt an, welche Pixel von einem Bild verwendet werden sollen. Alle Pixel auf einer Position, die bei der Maske einen Wert ungleich null hat, werden gezeichnet.

10. Messergebnisse

Auf Grund der Einfachheit, der Genauigkeit und keiner Notwendigkeit für zusätzliche Hardware, wurden die Messungen mit Zeitstempeln durchgeführt (siehe Abschnitt 6.6.1).

10.1. Messaufbau

Abbildung 10.1 zeigt den Netzwerkaufbau, der für alle Tests benutzt wurde. Für weitere Informationen zum Systemaufbau siehe auch Kapitel 5.

Der Roboter war ca. drei Meter von dem Router entfernt, ohne störende Hindernisse dazwischen, was einen weitgehend optimalen Empfang mit sich zieht und in realistischen Szenarien sicherlich nicht immer gegeben sein wird. Dass die Bedienerstation via Kabel und die mobile Einheit kabellos angeschlossen ist, entspricht dem angedachten Anwendungsfall. Jedoch sind die benutzten Netzwerkstandards nicht auf dem neuesten Stand. Zum einen würde man typischer Weise Gigabit statt 100 Mbit Ethernet nutzen und zum anderen ist der neuere WiFi Standard *IEEE 802.11ac* verfügbar, welcher auch schon von dem eingesetzten Computer auf dem Roboter unterstützt wird und einen weiteren Geschwindigkeitsvorteil mit sich bringt.

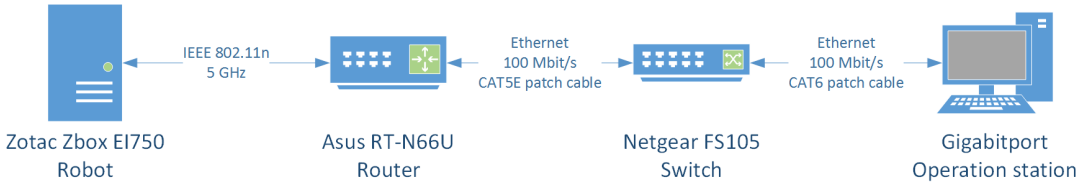

Abbildung 10.1.: Testaufbau des Netzwerkes

Zeitstempel wurden durch die *Time* Klasse aus dem ROS *Namespace*[1] erstellt. Hierbei wird, wenn möglich, die Funktion *clock_gettime* unter Linux mit dem Parameter *CLOCK_REALTIME* genutzt, wodurch eine Zeitauflösung im Nanosekundenbereich möglich ist. Für die Aufzeichnungen wurden Mikrosekunden als Einheit gewählt, was immer noch eine sehr hohe Genauigkeit bietet, da sich die zu messenden Zeiten im Millisekundenbereich bewegen.

[1] *ROS::Time*

10.2. Einzellatenzen

Zunächst folgen die Latenzen von einzelnen Abläufen, aus denen sich die Gesamtlatenz berechnet. Wenn nicht weiter erläutert, wurden die Werte der Messungen über etwa eine Minute aufgezeichnet, um anschließend daraus einen Durchschnittswert zu errechnen. Die Zeiten wurden auf der Konsole des Programms ausgegeben, welche in eine Textdatei geleitet wurden. Die Messungen stammen vom 28. Juli 2014 und basieren deshalb auf der Programmversion diesen Datums.

10.2.1. Bildverarbeitung

Nach der Hinzunahme des virtuellen Schwenkens, und damit dem Wegfallen von Übertragungslatenzen bei Bewegungen, ist die Bildverarbeitung der Hauptverursacher für Latenzen. Um die benötigte Zeit zu messen, wurden die zwei Zeitstempel aus Abbildung 6.3 genutzt. Dadurch wurde die Zeit zwischen dem Erstellen einer Orientierung der Oculus Rift und dem Anzeigen des Bildes, welches diese Orientierung nutzt, gemessen sowie die Zeit zwischen zwei angezeigten Bildern[2]. Beide Messungen wurden in einem Durchlauf in drei unterschiedlichen Betriebsmodi gemessen.

1. Virtuelles Schwenken wurde ausgestellt, sodass rein mechanisch bewegt wurde.
2. Virtuelles Schwenken wird verwendet, jedoch ohne Rotationen (bzw. hoher Schwelle (siehe Abschnitt 9.4)).
3. Virtuelles Schwenken inklusive Rotationen wurde verwendet (Schwelle wurde auf null gestellt).

Tabelle 10.1.: Diese Tabelle zeigt die Durchschnittszeiten bei der Bildverarbeitung mit der Standardabweichung, als Maß der Streuung, in Klammern dahinter.

	display to display	orientation to display
mechanical	53,3 ms (5,3 ms)	46,8 ms (5,4 ms)
no rotation	52,8 ms (5,4 ms)	46,9 ms (5,3 ms)
with rotation	57,1 ms (5,2 ms)	50,6 ms (5,4 ms)

In Tabelle 10.1 sind die Durchschnittswerte, errechnet aus tausend Werten, zu sehen. In Abbildung D.2 des Anhanges ist die Streuung der Zeiten erkennbar. Es ist zu erkennen, dass sich

[2]Entspricht dem Kehrwert der Frames per second (FPS).

die Zeiten für das mechanische und das virtuelle Schwenken nicht großartig unterscheiden[3], während die Rotation[4] die Latenz um etwa 4 ms erhöht. Das Benötigen einer Rotation wird durch eine gesetzte Schwelle minimiert und sollte in einem Nutzungsszenario sehr selten auftreten[5], sodass im folgenden die Messwerte ohne Rotation für das virtuelle Schwenken repräsentativ sind.

Ein Blick auf den Zeitbedarf der einzelnen Verarbeitungsschritte (siehe Abbildung 9.2) gibt Aufschluss über die Ähnlichkeit der Werte. Abbildung 10.2 zeigt den durchschnittlichen[6] Zeitaufwand bei der Verarbeitung von einem Bild[7]. Dabei wurde virtuelles Schwenken und Rotieren genutzt. Das HUD besteht aus einem Text und der Karte.

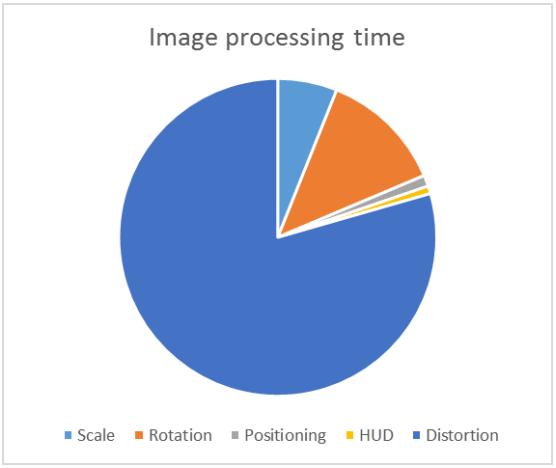

Abbildung 10.2.: Verteilung des Zeitaufwandes der einzelnen Verarbeitungsschritte bei der Bildverarbeitung.

10.2.2. Bildkorrektur

In der vorhergehenden Arbeit von Herrn Wiese (siehe Wiese (2014)) wurden für die zwei genutzten Kameras Kalibrierungsdaten erstellt. Der ROS-Node *image_proc* empfängt die Bilder der Kamera, wendet die Daten darauf an, um die Linsenverzeichnungen herauszurechnen und

[3] Dass das mechanische Schwenken in dieser Messreihe schneller ist, sollte sich als Zufall herausstellen und mit der geringen Anzahl an Messwerten erklärt werden.

[4] Die benötigte Zeit bei der Rotation ist unabhängig von der Größe des Winkels um den rotiert wird. Ausnahme ist dabei der Winkel 0° (bzw. ein Vielfaches von 360°). Da der Kamerakopf und das Head-Tracking-System aber nie komplett identische Orientierungen liefern, muss bei Messungen die Rotation nicht künstlich erzwungen werden.

[5] Es wird angenommen, dass Kopfbewegungen um die Roll-Achse wenig getätigt werden.

[6] Errechnet aus ca. 2.000 Messwerten.

[7] Die Zeiten wurden bei der Verarbeitung von einem Bild gemessen. Programmabläufe wie das Aufrufen von Callbacks oder das Zusammenführen und Anzeigen beider Bilder wurden in der Messung nicht betrachtet.

veröffentlicht das korrigierte Bild. Um die dabei entstehende Latenz zu messen, wurde wie folgt vorgegangen:

Auf der mobilen Einheit wurde der Standard Stack gestartet, der die Treiber der Kamera beinhaltet. Die Messung fand mit einem weiteren ROS-Node ebenfalls auf der mobilen Einheit statt, um so unabhängig von Netzwerklatenzen zu sein. Dieser ROS-Node empfängt sowohl die originalen als auch die korrigierten Bilder. Beim Empfangen eines Bildes wird die Empfangszeit und die Aufnahmezeit[8] vom Bild vermerkt. Sobald von einem Bild beide Exemplare vorliegen (identische Aufnahmezeit), wird die Differenz der Empfangszeiten ausgegeben. Weil damit zu rechnen ist, dass das korrigierte Bild später empfangen wird, wird die Differenz wie folgt gebildet:

$ReceivedTime_{corrected} - ReceivedTime_{untouched}$

Über 3.000 Messwerte ergaben eine durchschnittliche Differenz von ca. 5,8 ms mit einer Standardabweichung von ca. 2,4 ms. Die Verteilung kann in Abbildung D.5 eingesehen werden. Zu sehen sind auch negative Werte. Dies bedeutet, dass das korrigierte Bild vor dem Originalbild empfangen wurde[9]. Da ein korrigiertes Bild erst entstehen kann, wenn das Originalbild veröffentlicht wurde, ist dies mit Hängern vor der Empfangsroutine zu erklären[10].

10.2.3. Übertragungszeiten

Entscheidend für die Latenzen sind die Übertragungszeiten von den Bildern und den Orientierungen des HMD. Zur Messung wurden die entsprechenden Daten von der mobilen Einheit zur Bedienerstation geschickt, welche sie umgehend zurückgeschickt hat. Die Zeit vom Versenden auf der mobilen Einheit bis zum Empfangen auf selbiger wurde durch zwei geteilt, um die Übertragungszeit für eine Strecke zu schätzen. Die Messungen beinhalten damit auch die benötigte Zeit zum Versenden und zum Empfangen der Daten[11].

Bilddaten

Ein komprimiertes Bild der genutzten Kamera hatte im vorliegenden Netzwerk eine durchschnittliche Übertragungszeit von ca. 91,4 ms mit einer Standardabweichung von ca. 59 ms. Dies geht aus 419 Messungen hervor. Minimal wurden 82 ms gemessen. Nach oben gab es größere Schwankungen, so kann fast eine Sekunde benötigt werden. Abbildung D.3 visualisiert diese Verteilung. Große Schwankungen sind auf das WLAN zurückzuführen.

[8] Der Zeitstempel für die Aufnahmezeit wird bei der Korrektur nicht verändert.
[9] Etwa 1,5 % der Werte waren negativ.
[10] Es ist auch denkbar, dass sich das System mit einem Zeitserver synchronisiert hat. Die *RealClock* kann dabei auch zurückgedreht werden.
[11] Zum Beispiel durch Callbacks verursacht.

Orientierungsdaten

Da das Head-Tracking-System und der Kamerakopf ihre Orientierung im gleichen Format versenden, ist bei beiden mit derselben Übertragungszeit zu rechnen. Tausend Messungen ergaben einen Durchschnitt von ca. 26,4 ms bei einer Standardabweichung von ca. 2,7 ms. Ein Diagramm zur Verteilung ist im Anhang D.4 zu sehen.

10.3. mechanisches Schwenken

Anhand der gemessenen Einzellatenzen wird nun die Gesamtlatenz beim mechanischen Schwenken errechnet. Im Gegensatz zum virtuellen Schwenken kann die Latenz für die Bewegung nicht von der Latenz der Bilder getrennt werden, da Bewegungen immer erst sichtbar sind, wenn ein Bild aus der neuen Perspektive aufgenommen und anschließend verarbeitet wurde. Die Zeit, bis die Servomotoren die richtige Orientierung angenommen haben, ist davon abhängig, wie stark sich der Benutzer bewegt. Im Folgenden wurde eine plötzliche Drehung um 10° angenommen. Zeitwerte werden auf ganze Millisekunden gerundet. Für eine grafische Darstellung des Ablaufes siehe auch Abbildung 6.4.

1. Das Head-Tracking-System der Oculus Rift wird mit 120 Hz betrieben[12]. Dadurch wird etwa alle 8 ms ein Sensorwert entnommen. Kopfbewegungen können mit einer Verzögerung von bis zu 8,3 ms erkannt werden.

2. Zunächst muss die Orientierung vom Head-Tracking-System an der Bedienerstation zur mobilen Einheit transportiert werden. Hierbei wurde durch Messungen (siehe oben) der Wert 26 ms bestimmt.

3. Empfangene Orientierungsdaten werden praktisch sofort an die Servomotoren als Stellwert weitergegeben. Anhand der Spezifikationen aus dem Datenblatt der Servomotoren (siehe ROBOTIS (2006)) wird eine Stellzeit von 32 ms angenommen[13].

4. Sobald die Kamera in die richtige Stellung gebracht wurde, kann ein Bild aus der neuen Perspektive aufgenommen werden[14]. Da die Kamera mit 30 FPS aufnimmt, vergehen bis zu 33 ms, bevor ein neues Bild aufgenommen wird. Im besten Fall wird direkt nach dem Stellen ein Bild aufgenommen.

[12] Diese Frequenz wurde frei bestimmt. Die Oculus Rift unterstützt bis zu einem kHz.

[13] Angegeben wurden 196 ms für 60° bei 10 V Betriebsspannung. Im vorliegenden Aufbau wurden die Servomotoren mit 9 V betreiben (siehe Wiese (2014)).

[14] Selbstverständlich macht die Kamera bereits während des Stellens des Kamerakopfes Bilder, wodurch bereits vorher aufgenommene Bilder Bewegungen darstellen. In dieser Berechnung wird jedoch nur das Bild aus der finalen Position beachtet.

5. Das aufgenommene Bild muss von der mobilen Einheit zur Bedienerstation übertragen werden. Hierbei wurde ein Durchschnittswert von 91 ms bestimmt.

6. Im besten Fall, wird das empfangene Bild direkt verarbeitet und nach 53 ms angezeigt. Für die Synchronisation des linken und rechten Bildes werden die letzten beiden Bilder pro Seite gepuffert. So kann es passieren, dass ein empfangenes Bild erst in der übernächsten Anzeige Verwendung findet, da es nicht zum Bild der anderen Seite passt. Außerdem kann es passieren, dass die Bildverarbeitung kurz vor dem Empfangen des Bildes gestartet wurde, sodass es erst in der Anzeige danach verarbeitet werden kann. In beiden Fällen bedeutet dies, dass ein zusätzlicher Bildverarbeitungszyklus hinzukommt und somit 106 ms nötig sind.

Die genannten Zeiten zusammengerechnet, ergibt sich, bei einer Kopfbewegung von 10°, eine Bewegungslatenz im *Best Case* von etwa 202 ms und im *Worst Case* von ca. 296 ms.

10.4. Virtuelles Schwenken

Die Bewegungslatenz vom virtuellen Schwenken kann wie in Abschnitt 6.6.1 geschätzt werden. Die „additional latency“, die Zeit, die nach dem entnehmen eines Sensorwertes der Oculus Rift bis zum Anzeigen eines Bildes, welches diese Orientierung enthält, vergeht, wurde mit den Messungen der Bildverarbeitung bestimmt und beträgt im Durchschnitt knapp 47 ms. Da in dieser Zeit weitere Kopfbewegungen stattfinden, die erst im nächsten Bild angewendet werden, wurde die „full latency“ eingeführt, welche durch Hinzunahme der Zeit zwischen zwei angezeigten Bildern entsteht. Die Zeit, die vergeht bis das nächste Bild angezeigt wird, beträgt durchschnittlich knapp 53 ms. Dadurch ist eine „full latency“ von etwa 100 ms zu nennen. Dies ist die Zeit, die maximal vergeht, bis eine Kopfbewegung auf dem angezeigten Bild visualisiert wird und bildet somit den *Worst Case*.

10.5. Zusatzlatenzen

Der Genauigkeit halber sind noch folgende Zusatzlatenzen zu nennen:

Der Zeitstempel am Ende des Programmdurchlaufs markiert die Zeit, zu der das Bild im Programm zur Anzeige gebracht wird. Bis dieses tatsächlich auf dem Bildschirm zu sehen ist, vergeht noch einige Zeit (siehe Abschnitt 6.4). Dies geschieht durch den *inputlag* des Displays und der *pixelresponsetime* und betrifft sowohl das virtuelle als auch das mechanische Schwenken.

Es wurde nicht bestimmt, wie schnell die Servomotoren auf einen neuen Stellwert reagieren, und wie schnell Stellwerte an die Motoren gesendet werden können, weshalb mit einem sofortigen Start der Bewegung gerechnet wurde.

10.6. Erkenntnis

Es ist zu erkennen, dass die Bildverarbeitung durch das virtuelle Schwenken kaum zusätzliche Belastung zuungunsten der benötigten Zeit nach sich zieht. Das Rotieren kann durch eine gesetzte Schwelle weitestgehend vermieden werden und ist im Anwendungsfall selten in Gebrauch. Das Bild muss auch beim mechanischen Schwenken positioniert werden. Beim virtuellen Schwenken wird diese Position nur etwas angepasst, was durch simple Arithmetik verursacht wird, wodurch dies nicht ins Gewicht fällt.

Bei der Latenz von Bewegungsumsetzungen kann das virtuelle Schwenken überzeugen, sodass eine Halbierung der Latenz, im Vergleich zum mechanischen Schwenken, möglich ist. Auch ist das System weniger von Netzwerklatenzen betroffen, sodass Abweichungen seltener auftreten. Außerdem ist es problemlos möglich, die Latenz nochmals um einige Millisekunden zu senken, indem eine schnellere CPU verwendet wird[15], während es beim mechanischen Schwenken ein Zusammenspiel von vielen Aspekten ist, die alle optimal aufeinander eingestellt werden müssten.

[15] Auf der mobilen Einheit wurde beispielsweise eine Geschwindigkeitssteigerung von 50 % beobachtet. Beim virtuellen Schwenken (ohne Rotation) wurden beim genannten Messaufbau (Intel Core i5-750) etwa. 18,9 FPS erreicht, während der i7-4770R aus der mobilen Einheit ca. 29,7 FPS lieferte. Dabei wurden auf der mobilen Einheit beide Stacks gestartet, sodass das gesamte System auf einem Computer lief.

11. Fazit

Die Anforderungen (siehe Abschnitt 1.2) werden als erfüllt angesehen. Eine Reduzierung der Latenz, bezogen auf Bewegungen, wurde erfolgreich umgesetzt. Zusätzlich angezeigte Informationen wurden exemplarisch implementiert und können erweitert werden. Die Nutzung des vorhandenen Laserscanners wurde leider etwas vernachlässigt.

Die Notwendigkeit von mechanisch-schwenkbaren Kamerasystemen kann angezweifelt werden. Als problematisch stellten sich vor allem die unscharfen Bilder bei schnellen Bewegungen heraus, während beim rein virtuellen Schwenken schnelle Bewegungen mit scharfen Bildern umgesetzt wurden. Schuld daran trägt jedoch auch das verwendete Kamerasystem, welches nicht für dieses Szenario geschaffen ist.

Ein Verzicht des mechanischen Teils ist denkbar, jedoch muss daraus resultierend eine andere Hardware verwendet werden, denn auch wenn das virtuelle Schwenken bei den Tests als erfolgreich eingestuft werden kann, so ist die aktuelle Umsetzung nicht zukunftssicher, denn das, als Prototyp zu verstehende, verwendete HMD hat eine relativ geringe Auflösung, zusätzlich hat das Kamerasystem einen geringen Sichtwinkel. Bei Erhöhung würde die erforderte CPU-Leistung steigen und somit die Bildrate auf dem verwendeten System sinken, was ein stockendes Video hervorbringt, wodurch die Immersion deutlich gesenkt wird. Eine Verbesserung dieser Umstände sollte in weiterführenden Arbeiten berücksichtigt werden. Anregungen hierfür werden im Kapitel 12 Ausblick genannt.

Durch den Fortschritt in der Entwicklung von Prozessoren und die fortlaufend schneller werdenden Netzwerkverbindungen scheinen sich Probleme bei Latenzen von selbst zu lösen. Jedoch ist auch in der Entwicklung der VR-Brillen ein rasches Fortschreiten der Technik zu erkennen. So wird die fertige Brille von *Oculus VR* mit einem 4k Display[1] erwartet. Dies entsprecht einer Steigerung an Bildpunkten um den Faktor 8 gegenüber der hier verwendeten ersten Entwickler Version[2]. Durch die Notwendigkeit an größeren Bilder entstehen hierdurch höhere Anforderungen an Leistung und Übertragungsbandbreite. Somit werden auch in Zukunft latenzkompensierende Maßnahmen nicht ihre Daseinsberechtigung verlieren.

[1] 3840 x 2160 Pixel
[2] 1280 x 800 Pixel

12. Ausblick

Im Rahmen dieser Bachelorarbeit konnten nicht alle Aspekte dieses Themas betrachtet und umgesetzt werden. Im folgenden Kapitel werden Punkte genannt, die in weiterführenden Arbeiten aufgearbeitet werden können oder im Allgemeinen als nützlich erachtet werden.

Eine wichtige Verbesserung, die nötig ist, ist die Geschwindigkeit in Bezug auf die Bildverarbeitung. Um wirklich eine Immersion zu erzeugen, sollten dem Nutzer Bilder mit ca. 60 Hz gezeigt werden. Hierzu könnte eine bessere Hardware eingesetzt werden, wobei ein Field Programmable Gate Array (FPGA) oder eine GPU passend erscheinen. Es könnte auch versucht werden, die vorhandene CPU besser auszunutzen, da aktuell maximal 2 Kerne durch Bildverarbeitung ausgelastet werden. Ein anderer Ansatz ist das Vorausahnen von Bewegungen, sodass diese schon vorgefertigt werden können.

12.1. Kamerasystem

Auf dem Telepräsenzroboter befindet sich ein um alle drei Achsen im Raum schwenkbarer Kamerakopf. Auf dem Kamerakopf sollten zwei Kameras bzw. eine stereoskopische Kamera angebracht werden. Zur Zeit besteht das Kamerasystem aus zwei Webcams, welche nicht für dieses Projekt geeignet sind. Im Folgenden werden Anforderungen und Ideen für eine Neuanschaffung oder Erweiterung des Systems genannt.

12.1.1. Anforderungen

Für ein zufriedenstellendes Ergebnis sind folgende Anforderungen an das Kamerasystem zu stellen.

Gewicht

Um eine möglichst *weiche* Bewegung der Servobewegungen zu garantieren, muss auf das Gewicht des Kamerasystems geachtet werden. Die zunächst angedachte Bumblebee mit einem Gewicht von ca. 300 g führte zu äußerst schwerfälligen Bewegungen sowie zu Schwingungen

nach abruptem Abbremsen. Die zur Zeit genutzten *Logitech HD C310 Webcams* mit einem Gewicht von je 70 g funktionierten gewichtstechnisch problemlos. Ansonsten müsste die Halterung bzw. müssten die Servomotoren an ein Kamerasystem mit höherem Gewicht angepasst werden.

Sichtwinkel

Die genutzte Oculus Rift bietet einen horizontalen Sichtwinkel von 90° und 110° in der Vertikalen (siehe Lang (2012)). Ein geringerer Sichtwinkel der Kamera bedeutet, dass für eine naturgetreue Sicht das Kamerabild verkleinert werden muss und somit schwarze Ränder rund um das Bild entstehen. Ein höherer Sichtwinkel der Kamera bietet mehr Spielraum für das virtuelle Schwenken (siehe Kapitel 7 Virtuelles Rollen, Nicken und Gieren) und ist damit absolut wünschenswert.

Aufnahmesynchronisation

Für eine stereoskopische Sicht müssen den Augen des Benutzers zwei verschiedene Bilder gezeigt werden, welche in etwa mit dem gleichen Abstand aufgenommen wurden wie der zwischen den beiden Augen eines Menschen.

Bei einer Stereokamera muss nichts weiter zu diesem Thema bedacht werden. Sollte man jedoch zwei separate Kameramodule benutzen, so ist nicht sichergestellt, dass die Kameras die Bilder zeitgleich aufnehmen, was bei bewegten Bildern die Immersion mindert und eventuelle Reaktionen erschwert.

Industriekameras bieten unter Umständen einen Triggereingang, mit dem man den Aufnahmezeitpunkt erzwingen kann. Wenn man die Triggereingänge von beiden Kameras durch den selben Taktgeber betreibt, werden beide Bilder synchron aufgenommen.

Beachtet werden muss jedoch, dass somit zwar die Aufnahme synchron passiert, die Bilder aber nicht zwangsweise synchron verschickt bzw. empfangen werden. Da es sich um ein verteiltes System handelt, kann es also nach wie vor passieren, dass zum Zeitpunkt der Bildverarbeitung an der Bedienerstation zwei Bilder vorliegen, welche zu unterschiedlichen Zeiten aufgenommen wurden. Um diesem entgegenzuwirken, werden in der aktuellen Implementation die letzten zwei Bilder je Kamera inklusive des Zeitstempels, welcher die Zeit der Aufnahme markiert, auf der Bedienerstation gespeichert. Wenn beim Verarbeiten der Bilder die zeitliche Differenz zu hoch ist, wird auf ein älteres Bild zugegriffen, sodass beide Bilder möglichst ähnliche Aufnahmezeiten haben. Dadurch wird auch die Nutzung von Kameras kompensiert, die nicht synchron aufnehmen.

Bewegungsschärfe

Da sich sowohl die Kamera als auch die Objekte vor der Kamera bewegen können, sollte dafür gesorgt werden, dass die Motive auch bei schnellem Umgucken scharf abgebildet werden. Dies wird mit kurzen Belichtungszeiten bzw. Verschlusszeiten erreicht. Abbildung 12.1 verdeutlicht dies.

Abbildung 12.1.: Aufnahmen von fließendem Wasser mit Verschlusszeiten von einer Sekunde (links), 1/30 Sekunden (Mitte) und 1/800 Sekunden (rechts) (Maxwell (2006))

Des Weiteren ist die Art des Blendenverschlusses wichtig. Die vorhergehende Arbeit von Herrn Wiese (siehe Wiese (2014)) merkte an, dass bei dem Verzicht von *global-shutter*-Blenden *rolling-shutter*-Effekte auftreten können (siehe Abbildung 12.2), da hierbei das Bild zeilen-, oder spaltenweise statt simultan aufgenommen wird.

Abbildung 12.2.: *Rolling shutter* Effekt bei einem bewegenden Motiv. (Axel1963 (2010))

Linsenverzeichnung

Kameralinsen erzeugen ungewollt durch Ungenauigkeiten, teilweise auch gewollt durch spezielle Linsen, Verzeichnungen am Bild. Dabei entstehen Krümmungen, obwohl das Motiv gerade Linien aufweist (siehe Abbildung 12.3). Ein zur Verfügung stehender ROS-Node korrigiert diese Effekte, was eine weitere Latenz nach sich zieht. Wenn man bei der Kamerabeschaffenheit auf solche Verzeichnungen achtet, kann auf die Korrektur verzichtet werden, um die Latenz bei den Bildern zu verringern.

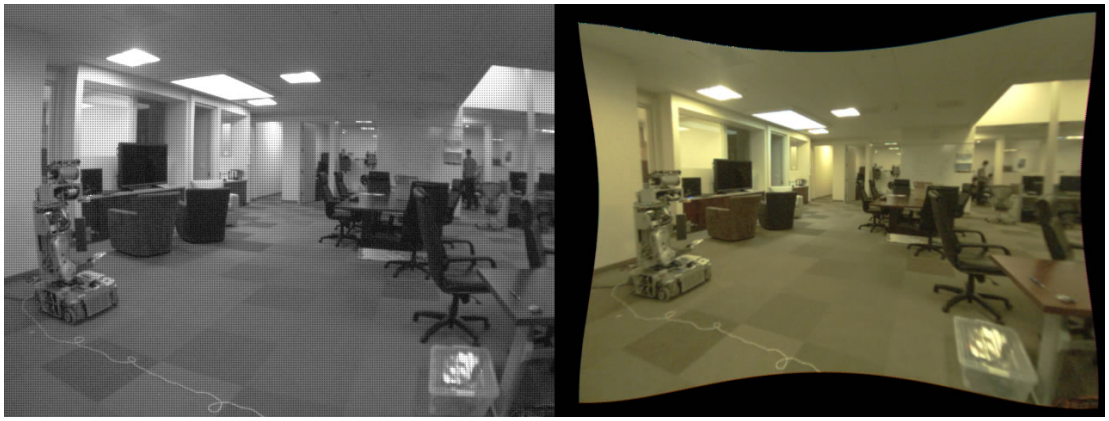

Abbildung 12.3.: Beispiel der Bildkorrektur. Links das aufgenommene Bild einer Kamera, rechts das korrigierte Bild. (Patrick Mihelich (2014))

12.1.2. Sichtfeld vergrößern

Wie bereits erwähnt, sollte für das virtuelle Schwenken der Sichtwinkel vergrößert werden, um mechanische Bewegungen zu vermeiden. Hierbei muss jedoch bedacht werden, dass ein größeres Bild eine längere Übertragung mit sich zieht. Um die Transportzeit trotzdem gering zu halten, könnte man den Bildausschnitt schon auf der mobilen Einheit wählen und so das Senden von Bilddaten außerhalb des Sichtfelds des HMDs vermeiden. Jedoch würde dies auch auf dem Roboter eine Bildverarbeitung nötig machen und zur Bestimmung des Bildausschnittes müsste vorerst mit dem Head-Tracking-System kommuniziert werden, was weitere Latenzen nach sich zieht.

Durch folgende Hardwarekomponenten ist eine Vergrößerung des Sichtfeldes möglich:

Objektiv

Das Sichtfeld einer Kamera ist von dem genutzten Objektiv abhängig. Bei einem Weitwinkelobjektiv mit geringer Brennweite lassen sich Sichtwinkel von über 70° realisieren (siehe Abbildung 12.4). Ein Superweitwinkelobjektiv mit einer Brennweite von 14 mm würde beispielsweise schon einen diagonalen Sichtwinkel von 114° haben (siehe de.wikipedia.org (2014)), was etwa 80° horizontal und vertikal entspricht[1]. Mit einem Fischaugenobjektiv lassen sich sogar Sichtwinkel von 180° umsetzen (siehe de.wikipedia.org (2014)), jedoch wird dabei das Bild tonnenverzeichnet. Diese Tonnenverzeichnung muss auf Grund der Linsen in der Oculus Rift auch für das HMD angewendet werden, dies geschieht in der Software durch Bildverarbeitung. Eine Einsparung der Verzeichnung durch die Bildverarbeitung ist dennoch nicht möglich, weil sich die Tonnenverzeichnung der Fischaugenoptik über das gesamte Bild erstreckt, während die Verzeichnung der Bildverarbeitung nur auf den tatsächlich angezeigten Bereich angewandt wird und somit eine andere Verzeichnung ergibt. Deshalb müssen die Kamerabilder auf jeden Fall vor der Weiterverarbeitung korrigiert werden.

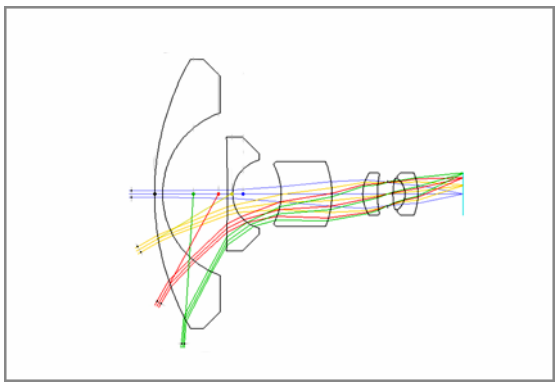

Abbildung 12.4.: Strahlengang eines Weitwinkelobjektivs (Schmitt, Jos. Schneider Optische Werke GmbH (2006))

Kameraarray

Man kann mehrere Kameras nebeneinander (eventuell auch übereinander) aufbauen und aus deren Bildern ein breites Panoramabild zusammenstellen. Hierdurch wird jedoch eine höhere Verarbeitungszeit der Bilder gefordert. Das Zusammensetzen der Bilder könnte auf dem Computer des Telepräsenzroboters durchgeführt werden, sodass der Hostcomputer weiterhin

[1] Ein quadratisches Bild mit 114° in der Diagonalen hat nach dem *Satz des Pythagoras* etwa 80° in der Horizontalen und Vertikalen.

nur ein Bild pro Auge bekommt, und sich somit nichts am Programm ändert. Der Prozess ließe sich auch auf ein FPGA auslagern, sodass es sich wie eine normale Kamera für den Computer des Roboters verhält.

360° Rundumsicht

Eine vollständige Rundumsicht hätte den Vorteil, dass horizontale Bewegungen komplett virtuell vollzogen werden können[2]. Es wäre denkbar, eine Rundumsicht parallel zum mechanisch schwenkbaren Kamerakopf zu benutzen. So könnten hochqualitative Kameras am Kopf die meiste Zeit genutzt werden und die Rundumsicht für schnelle Bewegungen, bis die Mechanik die gewünschte Position erreicht hat. Die 360° Rundumsicht wird keine stereoskopische Sicht zulassen, da man nicht zwei Stück von ihnen nebeneinander bauen kann, ohne dass sie sich gegenseitig aufnehmen. Außerdem könnte die Rundumsicht dem Nutzer als teil des HUDs eingeblendet werden.

12.2. Einblendbare Informationen

Ein Mensch hat wesentlich mehr Wahrnehmungen als den visuellen Eindruck alleine. Hinzu kommen beispielsweise akustische Signale, welche beim Telepräsenzroboter durch ein Stereomikrofon und beim Benutzer durch Stereolautsprecher umgesetzt werden können. Aber Sinneseindrücke wie Gerüche oder Temperaturen lassen sich schwer zum Benutzer transferieren.

Da Telepräsenzroboter auch für Missionen in Gefahrengebieten oder zu Erkundungen benutzt werden können, sind weitere Eindrücke denkbar, die ein Mensch vor Ort ohne entsprechendes Equipment gar nicht wahrnehmen könnte. Zum Beispiel könnte man mit Sensoren das Strahlungslevel oder den Sauerstoffgehalt der Luft messen.

Für alle diese Informationen, die nicht als Vor-Ort-Eindruck zum Benutzer übermittelt werden können, eignet sich eine visuelle Anzeige als sogenanntes HUD. Dabei werden die Informationen textuell oder grafisch im Sichtbereich, meist am Rand, positioniert, sodass sie dem Benutzer unmittelbar zur Verfügung stehen. In Abbildung 12.5 (Siehe dazu auch Kapitel 4 Recherche) wird ein Beispiel eines HUDs dargestellt. Oben ist der *Lastausleger* angezeigt mit der Position der *Laufkatze*. Am linken Rand wird die Ausrichtung der Kamera skizziert. Rechts zeigen die Pfeile die Bewegungsrichtung der Last an. Rechts unten sind die Orientierung des Krans und das Sichtfeld der Kamera dargestellt.

[2] Um die Bewegungen eines menschlichen Kopfes nach zu empfinden, wären schon etwa 300° ausreichend.

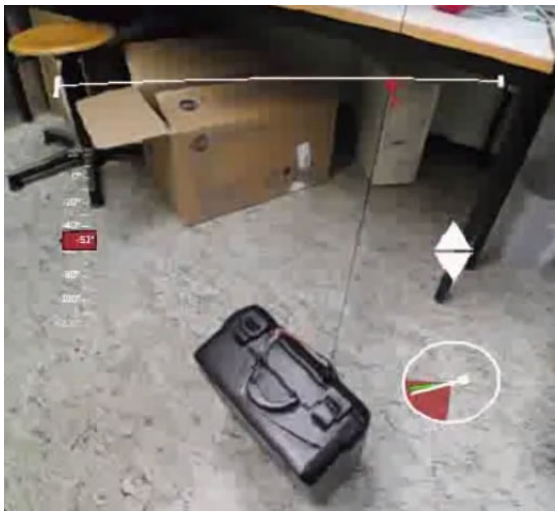

Abbildung 12.5.: HUD aus dem *Oculus Cran* Projekt (siehe Kapitel 4 Recherche). (Thomas (2014))

12.2.1. Informationen

Für den benutzten Telepräsenzroboter auf Basis des Pioneer P3-DX sind folgende Informationen denkbar:

Karte

Das Anzeigen einer Karte wurde bereits implementiert. Jedoch basiert diese zur Zeit noch auf *Dummy-Daten*. Eine Karte, die mittels SLAM auf tatsächlich aufgezeichneten Daten beruht, sollte folgen.

Die Karte kann durch POIs erweitert werden. Diese können direkt auf der Karte platziert werden oder als Pfeile am Rand der Karte, wenn sie außerhalb des Sichtausschnittes liegen. So ließe sich zum Beispiel die Richtung zur Bedienerstation um zurückzufinden oder Steckdosen für eine Lademöglichkeit anzeigen.

Ein weiterer Schritt wäre die autonome Navigation, sodass der Roboter autonom zu einem bestimmten Punkt fährt oder Funklöcher überwindet. Eine Arbeit zur Steuerung mittels optischem *Motion Capturing* ist bereits im Gange, sodass die Steuerung via Controller in bestimmten Räumlichkeiten obsolet wird.

Systemstatus

Neben Informationen über die Umwelt sind auch Daten über den Status des Systems denkbar. So ließe sich beispielsweise der Ladezustand der Akkus textuell oder als Batterie-Icon darstellen. Auch die Last und die Temperatur der Servomotoren lässt sich auslesen und anzeigen. An Hand der Odometrie kann zusätzlich die Geschwindigkeit errechnet und als Tachometer ersichtlich gemacht werden.

12.2.2. Augmented Reality

Bei Augmented Reality werden Informationen nicht unabhängig vom darunter liegenden Bild positioniert, sondern beziehen sich darauf. So können zum Beispiel beim Autorennen Informationen über andere Fahrer direkt am vorweg fahrenden Auto eingeblendet werden, statt sie am Rand aufzulisten (siehe Abbildung 12.6). Der Bremsspurenassistenten aus derselben Abbildung ist ein weiteres Beispiel dafür. Beim Telepräsenzroboter könnten so beispielsweise die Namen von Personen direkt über deren Köpfen angezeigt werden und auch andere Objekte könnten so mit Informationen bestückt werden.

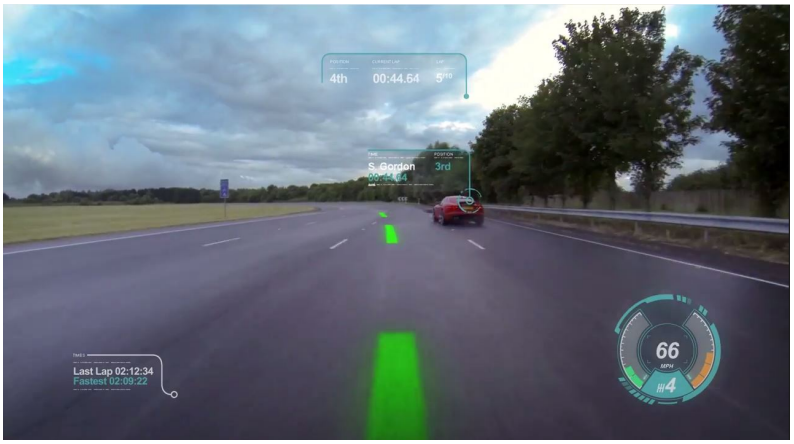

Abbildung 12.6.: Konzept von Jaguar für ein HUD auf der Windschutzscheibe eines PKWs. (Jaguar (2014))

Des Weiteren gehört zur Augmented Reality auch das verändern der Bilder. So könnten störende Objekte komplett entfernt oder farblich verändert werden. Es können auch Objekte hinzufügt werden. Zum Beispiel könnte der Roboter beim Fahren eine virtuelle Spur hinterlassen, sodass man beim Zurückblicken seinen Verlauf sieht.

Dies ist mit weiterem Rechenaufwand und mit erheblich komplexerer Bildverarbeitung verbunden. Es gibt bisher keine konkreten Konzepte zu einer solchen Umsetzung.

12.3. Sensorik

Neben den Kameras kann der Roboter mit weiteren Sensoren ausgestattet werden, um andere Aspekte der Umgebung aufzeichnen zu können. Im Nachfolgenden werden Sensorsysteme aufgezeigt, welche beim genutzten Telepräsenzroboter bereits vorhanden waren oder nachrüstbar sind[3].

12.3.1. Laserscanner

Zur Erfassung der Umgebung kann ein Laserscanner genutzt werden. Dabei wird ein Laserstrahl auf die umliegenden Objekte gelenkt und anhand der Zeit, die benötigt wird, bis der Strahl wieder bei der Quelle ankommt, die Entfernung bestimmt. Die Messungen werden für gewöhnlich auf einer horizontalen Ebene gemacht oder rasterförmig auf mehreren horizontalen Linien übereinander, wodurch komplette räumliche Strukturen aufgezeichnet werden können.

Bei einem Telepräsenzroboter ist der Hauptanwendungsbereich eines Laserscanners die Kartografie. Andere Anwendungen aus der Robotik sind das Identifizieren von Objekten und das Assistieren beim Interagieren mit solchen Objekten.

Bei der exemplarischen Umsetzung einer Kartendarstellung (siehe Abschnitt 12.2) wurde der Laserscanner *Hokuyo URG-04LX-UG01* in Betracht gezogen. Er bietet eine Reichweite von 4 m, eine horizontale Abdeckung von 240° und wird von ROS problemlos unterstützt. Dieses Modell kann genutzt werden, um zweidimensionale Karten von Gängen und kleinen Räumen zu machen. Für große Räumlichkeiten mit viel freier Fläche ist die Reichweite zu gering.

12.3.2. Mikrofon und Lautsprecher

Um die Geräuschkulisse des Roboters wahrnehmen zu können, ist ein Mikrofon am Roboter und ein Lautsprecher an der Basisstation Voraussetzung. Um dem menschlichen Gehör besser nachzuempfinden zu können, empfiehlt es sich, ein Stereomikrofon am Roboter und Stereolautsprecher, wie Kopfhörer, an der Bedienerstation zu benutzen. Um auch in die andere Richtung kommunizieren zu können, benötigt der Roboter einen Lautsprecher und die Basisstation ein Mikrofon, zum Beispiel in Form eines Headsets. Ein Stereomikrofon wurde provisorisch am Roboter befestigt, wurde jedoch nicht in Betrieb genommen.

An der HAW steht ein Raum für die Wellfeldsynthese zur Verfügung. Hierbei wird ein getaggtes Objekt (der Benutzer) durch Kameras im Raum lokalisiert und dann gemäß der zur

[3] Viele Themen für weiterführende Arbeiten sind bereits in der Arbeit von Herrn Wiese zu finden (siehe Wiese (2014)).

Verfügung stehenden Audiodaten beschallt. Bei Ausstattung des Telepräsenzroboters mit einem Raumklangmikrofon ist so eine sehr originalgetreue Audiowiedergabe möglich. Gedanken hierzu wurden bereits in der Arbeit von Herrn Wiese niedergeschrieben (siehe Wiese (2014)).

12.3.3. Ultraschall

Der verwendete Roboter Pioneer P3-DX besitzt an der Vorderseite acht Ultraschallsensoren, welche optional auch hinten nachgerüstet werden können, sodass es möglich ist alle Richtungen abzutasten (siehe Abbildung 12.7). Die Sensoren stehen in einem Winkel von 20° zueinander, mit der Ausnahme der zwei Äußeren auf jeder Seite, welche 40° abgewandt sind (siehe Cyberbotics (2014)).

Ähnlich wie mit einem Laserscanner kann so die Entfernung zu Hindernissen gemessen werden, jedoch mit geringer Auflösung[4]. Die Ultraschallsensoren werden zur Zeit nicht verwendet. In nachfolgenden Implementierungen könnten sie genutzt werden, um den Roboter vor Kollisionen zu schützen.

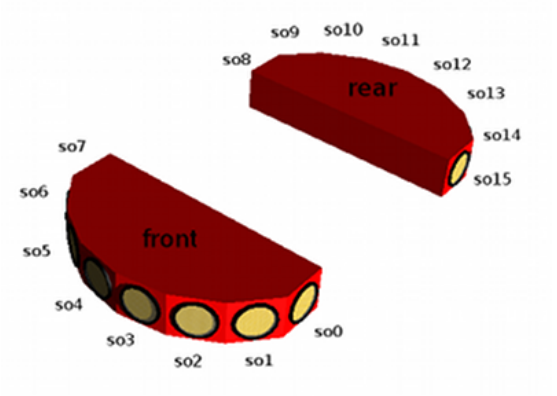

Abbildung 12.7.: Anordnung der Ultraschallsensoren beim Pioneer 3. (Cyberbotics (2014))

12.3.4. Externe Einsätze

Soll der Telepräsenzroboter auch außerhalb von Gebäuden eingesetzt werden, ist ein GPS-Empfänger für die Positionsbestimmung denkbar. Temperatur- und Feuchtigkeitssensoren wären ebenfalls für Einsätze im Freien hilfreich.

[4]Vergleiche 20° bis 40° beim Ultraschall mit 0,36° beim vorhandenen Laserscanner.

A. Workspace

Der Workspace ist die Ordnerstruktur, die den Source Code für die Programme beinhaltet und ist auf der beigelegten CD zu finden (siehe Anhang E Datenträger). Der Workspace wurde mittels GIT verwaltet und wird voraussichtlich unter folgender URL veröffentlicht:
`github.com/cubei/OculusCamViewer`.

A.1. Voraussetzungen

Um mit dem Workspace arbeiten zu können, werden neben ROS noch weitere Bibliotheken vorausgesetzt.

Um das benötigte RosAria zu installieren, können die Befehle, die im Listing A.1 zu sehen sind, in ein Terminal eingegeben werden.

```
$ sudo apt-get install libudev-dev
$ rosdep update
$ install rosaria
```

Listing A.1: Anweisungen für die Installation von RosAria

Für die Installation des verwendeten Treibers des Laserscanners und des Joysticks sind die Befehle aus Listing A.2 nötig.

```
$ sudo apt-get install ros-hydro-urg-node
$ sudo apt-get install ros-hydro-joy
```

Listing A.2: Anweisungen für die Installation des Laserscanner Treibers *urg_node* und den Treiber für Joysticks

OpenCV, als Bibliothek für sämtliche Bildverarbeitungen des Projektes, kann nach der Anleitung auf opencv.org installiert werden (siehe opencv dev team (2014)). In diesem Projekt wurde Version 2.4.8 verwendet.

Sollte es auf Grund von fehlenden Grafiktreibern zu Compilerfehlern kommen, kann die Bibliothek auch ohne GPU-Support compiliert werden. Hier zu wird *cmake* ein zusätzliches Argument übergeben (siehe Listing A.3).

```
-D BUILD_opencv_gpu=OFF
```

Listing A.3: cmake Argument, um openCV ohne GPU Unterstützung zu compilieren

A.2. Compilieren

Um den Source Code in ein ausführbares Programm zu übersetzen, wird der Befehl aus Listing A.4 in dem Hauptverzeichnis des Workspaces ausgeführt.

```
$ catkin_make
```

Listing A.4: Befehl zum Compilieren des Workspaces

A.3. Ausführung

Um nicht jeden Knoten einzeln händisch zu starten, wurden sogenannte *launch-files* geschrieben, die alle benötigten Knoten bündeln, konfigurieren und sie mit einem Befehl starten.

Um die Knoten mitsamt des ROS-Cores auf der mobilen Einheit zu starten, werden die Befehle aus Listing A.5 im Hauptverzeichnis des Workspaces aufgerufen.

```
$ source devel/setup.bash
$ roslaunch base zotac_me.launch
```

Listing A.5: Befehle zum Starten des Stacks auf der mobilen Einheit

Der stationären Einheit muss vor dem Start noch mitgeteilt werden, wo der ROS-Core zu finden ist. Auch hier werden die Befehle aus dem Hauptverzeichnis des Workspaces aus aufgerufen (siehe Listing A.6).

```
$ export ROS_MASTER_URI=http://zotac-tpr:11311
$ source devel/setup.bash
$ roslaunch oculus_cam_viewer oculus_viewer_driver.launch
```

Listing A.6: Befehle zum Starten des Stacks auf der Bedienerstation

B. Schnittstellen der ROS-Nodes

Im Folgenden werden die verwendeten ROS-Nodes mit ihren Schnittstellen aufgeführt.

B.1. oculus_cam_viewer

Dieser ROS-Node ist für die Anzeige des HMDs zuständig. Geschrieben in c++.

B.1.1. Published

- „/mapConfig“ (map2image::mapConfig): Informationen für den Kartengeber

B.1.2. Subscribed

- „/camera/left/image_rect_color“ (sensor_msgs::ImageConstPtr): Bilddaten links
- „/camera/right/image_rect_color“ (sensor_msgs::ImageConstPtr): Bilddaten rechts
- „/oculus/hmdinfo“ (oculus_msgs::HMDInfoPtr): Info über das verwendete HMD
- „/oculus/orientation_stamped“ (geometry_msgs::PoseStamped): Orientierung des HMDs
- „/headunit/orientation_stamped“ (geometry_msgs::PoseStamped): Orientierung des Kamerakopfes
- „/rosaria/pose“ (nav_msgs::Odometry): Orientierung und Position des Roboters (Odometrie)
- „/mapimage“ (sensor_msgs::ImageConstPtr): Bild der Karte

B.1.3. Parameter

- „showMap“ (bool): Karte wird angezeigt (Standardwert: true)
- „useVirtualPanTilt“ (bool): Virtuelles Schwenken und Rotieren wird angewandt (Standardwert: true)

- „measurement“ (bool): Messdaten werden gemacht und ausgegeben (Standardwert: false)
- „useRobotOrientation“ (bool): Rechnet die Orientierung der Odometrie für das virtuelle Schwenken raus (Standardwert: false)
- „rotateThreshold“ (double): Mindestwert für eine Rotation beim virtuellen Rollen (Standardwert: 3.0)

B.2. oculus_driver

Dieser ROS-Node ist für das Head-Tracking-System der Oculus Rift zuständig. Der Programmcode wurde von Takashi Ogura geschrieben (siehe Ogura (2014)) mit Erweiterung der Nachricht für Orientierungsdaten um einen Zeitstempel sowie Erhöhung der Standardfrequenz. Geschrieben in c++.

B.2.1. Published

- „/oculus/hmdinfo“ (oculus_msgs::HMDInfoPtr): Info über das verwendete HMD
- „/oculus/orientation_stamped“ (geometry_msgs::PoseStamped): Orientierung des HMDs

B.2.2. Parameter

- „frequency“ (double): Frequenz in Hz mit der Sensorwerte abgefragt und veröffentlicht werden (Standardwert: 120)

B.3. map2image

Dieser ROS-Node wandelt eine Karte vom gmapping Node in ein Bild um und passt dabei den Ausschnitt und die Größe an. Geschrieben in c++.

B.3.1. Published

- „/mapimage“ (sensor_msgs::ImageConstPtr): Bild der Karte

B.3.2. Subscribed

- „/map“ (nav_msgs::OccupancyGrid::ConstPtr): Karte

- „/rosaria/pose“ (nav_msgs::Odometry): Orientierung und Position des Roboters (Odometrie)
- „/mapConfig“ (map2image::mapConfig): Informationen für den Kartengeber

B.3.3. Parameter

- „drawArrow“ (bool): Zeichnet einen Pfeil in das Zentrum der Karte. Ausrichtung nach oben (Standardwert: false)
- „drawCircle“ (bool): Zeichnet einen Punkt in das Zentrum der Karte (Standardwert: false)
- „drawCone“ (bool): Zeichnet einen Kegel gemäß dem Sichtfeld (Standardwert: false)
- „biggerImageForRotation“ (bool): Der Kartenausschnitt wird größer geliefert als gefordert, sodass nach dem Rotieren keine leeren Ecken übrig bleiben (Standardwert: false)
- „useRotation“ (bool): Karte wird gemäß des Head-Tracking-Systems rotiert (Standardwert: false)
- „mapRange“ (double): Anzeigebereich der Karte in Metern (Standardwert: 10.0)

B.4. headunit

Der Programmcode wurde von Hendrik Wiese in python geschrieben (siehe Wiese (2014)). Erweitert wurde er um Zeitstempel und einer festen Frequenz für die Orientierungsdaten sowie um Odometriedaten für das Bewegen des Kamerakopfes.

B.4.1. Published

- „/headunit/orientation_stamped“ (geometry_msgs::PoseStamped): Orientierung des Kamerakopfes

B.4.2. Subscribed

- „/oculus/orientation_stamped“ (geometry_msgs::PoseStamped): Orientierung des HMDs
- „/rosaria/pose“ (nav_msgs::Odometry): Orientierung und Position des Roboters (Odometrie)

B.4.3. Parameter

- „pitch_min“, „pitch_max“, „yaw_min“, „yaw_max“, „roll_min“, „roll_max“ (int): Minimal- und Maximalwerte für die Servoauslenkung. Standardwerte wurden passend gesetzt.
- „use_robot_orientation“ (bool): Rechnet die Orientierung der Odometrie gegen die des HMD (Standardwert: false)

C. Funktion zum Kopieren von Bildern in Bilder

Listing C.1 zeigt die Funktion, die ein Bild in ein anderes kopiert. Dabei wird das zu kopierende Bild beschnitten, wenn es nicht vollständig in das Zielbild passt. Dadurch können negative Koordinaten und welche, die außerhalb des Zieles liegen, übergeben werden.

```
    if(posX > dst->cols || posY > dst->rows
            || posX + src->cols < 0
            || posY + src->rows < 0) {
        // src is outside of dst
        return;
    }

    int leftCrop = abs(min(0, posX));
    int rightCrop = max(posX + src->cols - dst->cols, 0);
    int topCrop = abs(min(0, posY));
    int bottomCrop = max(posY + src->rows - dst->rows, 0);

    cv::Rect croppedRect(   leftCrop, topCrop,
                            src->cols - leftCrop - rightCrop,
                            src->rows - topCrop - bottomCrop);
    cv::Mat modifiedImage = (*src)(croppedRect);

    cv::Rect copyRect(  std::max(posX, 0),
                        std::max(posY, 0),
                        modifiedImage.cols,
                        modifiedImage.rows);
    modifiedImage.copyTo((*dst)(copyRect));
```

Listing C.1: *drawImageOnImage* Funktion

D. Diagramme

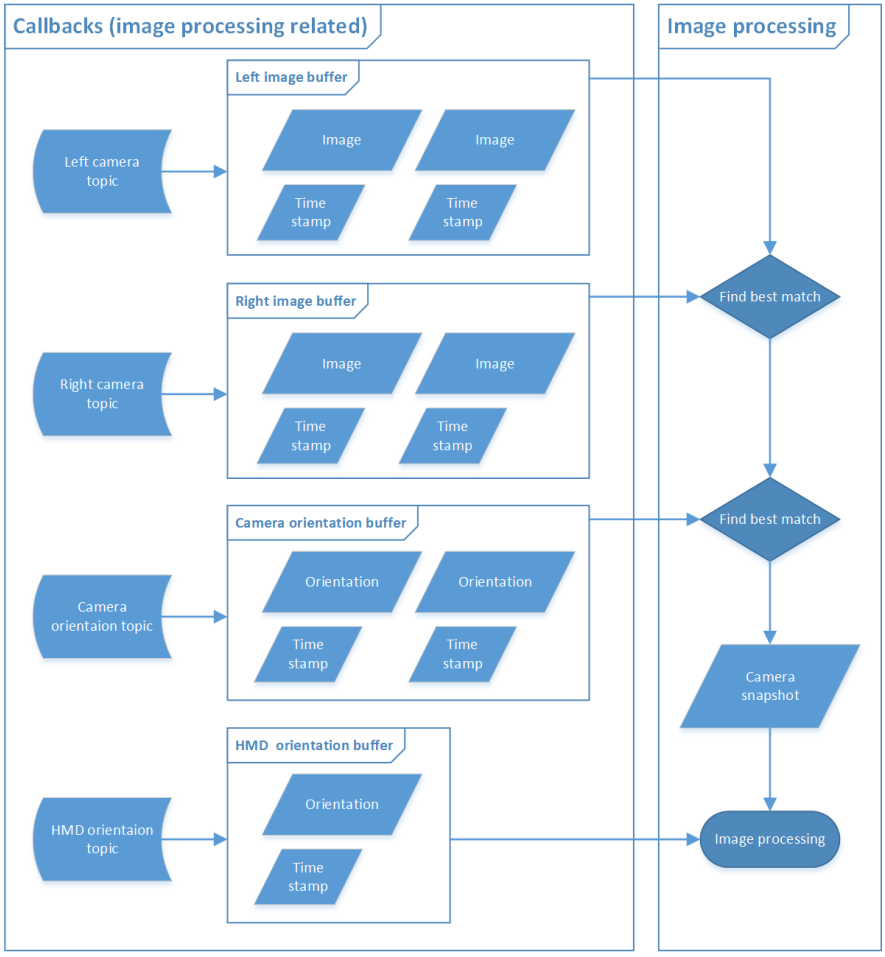

Abbildung D.1.: Zusammenstellung eines Kamera-Snapshots für die Bildverarbeitung, bestehend aus zwei Bildern und der dazu passenden Kameraorientierung

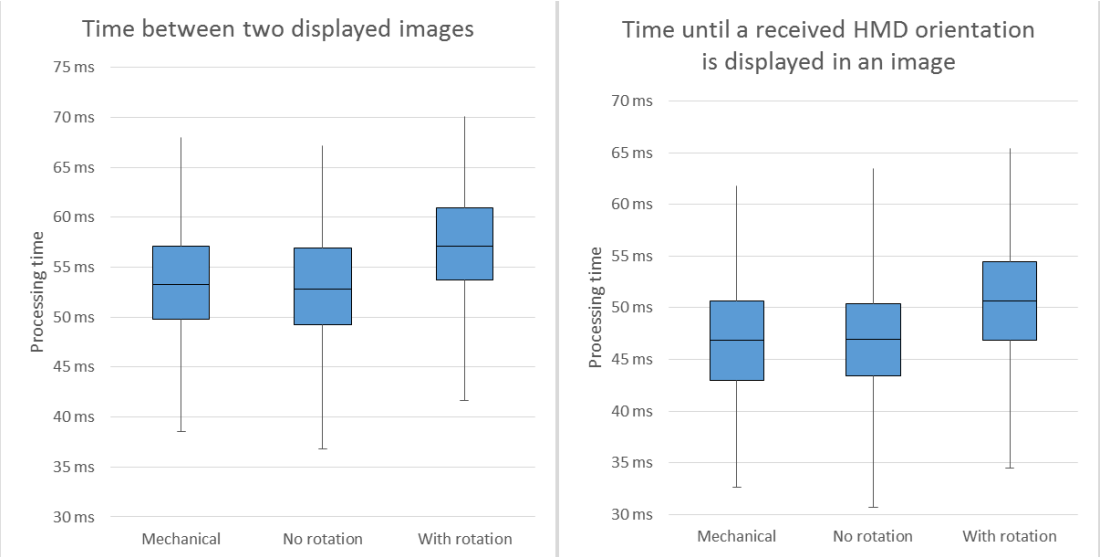

Abbildung D.2.: Benötigte Bildverarbeitungszeit, bis ein neues Bild angezeigt wird (links). Benötigte Bildverarbeitungszeit, bis eine HMD Orientierung in einem Bild angezeigt wird (rechts). Im markierten Feld befinden sich 50 % der Werte. Jeweils 25 % der Werte befinden sich darüber bzw. darunter entlang der eingezeichneten Linien. Der horizontale Strich markiert den Durchschnitt.

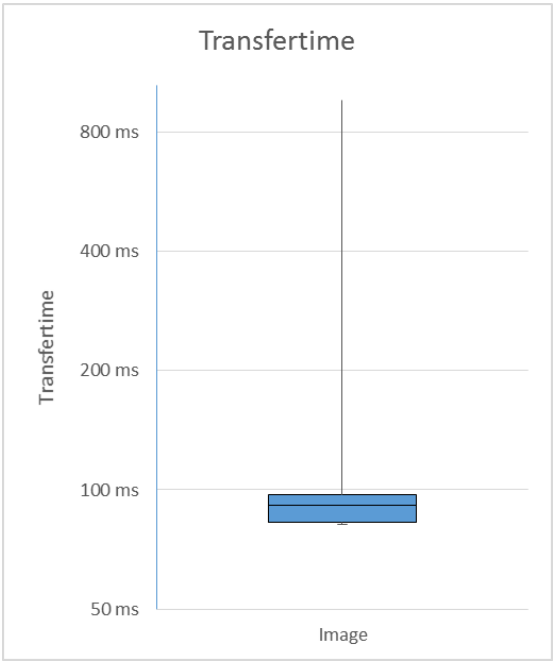

Abbildung D.3.: Benötigte Übertragungszeit von einem Bild. Im markierten Feld befinden sich 96 % der Werte. Jeweils 2 % der Werte befinden sich darüber bzw. darunter entlang der eingezeichneten Linien. Der horizontale Strich markiert den Durchschnitt. Die logarithmische Skalierung der vertikalen Achse zur Basis zwei ist zu beachten.

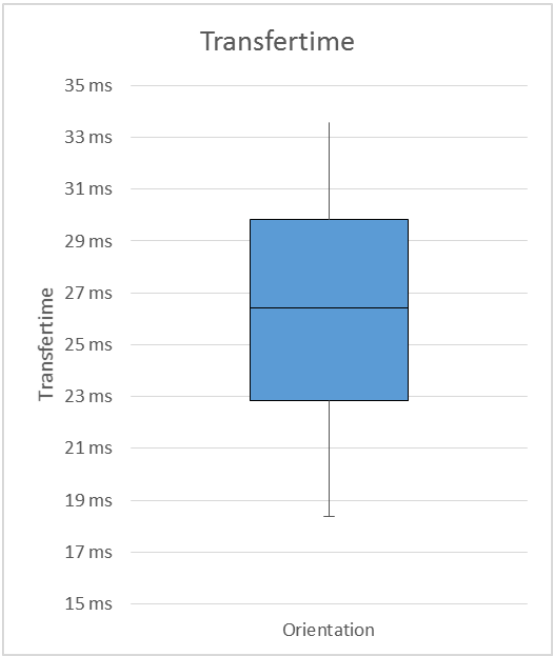

Abbildung D.4.: Benötigte Übertragungszeit von Orientierungsdaten. Im markierten Feld befinden sich 80 % der Werte. Jeweils 10 % der Werte befinden sich darüber bzw. darunter entlang der eingezeichneten Linien. Der horizontale Strich markiert den Durchschnitt.

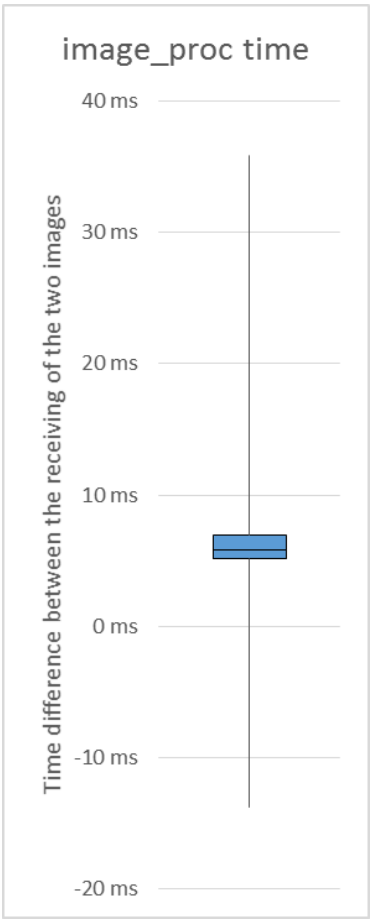

Abbildung D.5.: Verteilung der Bearbeitungszeit bei der Bildkorrektur. Im markierten Feld befinden sich 90 % der Werte. Jeweils 5 % der Werte befinden sich darüber bzw. darunter entlang der eingezeichneten Linien. Der horizontale Strich markiert den Durchschnitt.

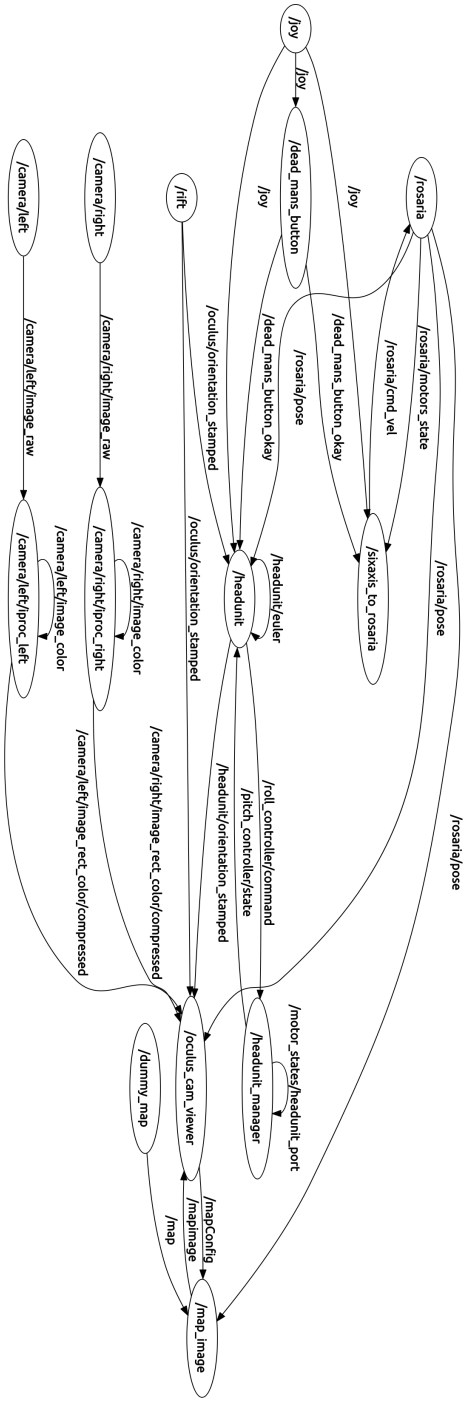

Abbildung D.6.: ROS Graph

E. Datenträger

Dieser Arbeit liegt eine CD mit folgendem Inhalt bei:

- LarsHarmsen_BA_2014.pdf
  PDF dieser Arbeit
- doc/
  Ordner mit den LaTeXQuelldateien für diese Arbeit
- workspace/
  Ordner mit dem Workspace (siehe Anhang A Workspace)

Abkürzungsverzeichnis

FPGA Field Programmable Gate Array. 56

FPS Frames per second. 25, 49, 52, 54

GPU Graphics Processing Unit. 19, 56, 67

HMD Head-Mounted-Display. VIII, 1, 3, 5, 6, 12, 15–17, 20, 21, 23, 25, 26, 32, 35, 38–41, 44, 51, 55, 59, 60, 69–72, 75

HUD Heads-up-Display. VIII, 10, 32, 33, 38, 43, 45, 50, 61–63

LAN Local Area Network. 19

POIs Points of Interest. 33, 62

ROI Region of Interest. 28, 40

ROS Roboter Operating System. IX, 5, 6, 11, 17, 19, 21, 35–37, 48, 64, 66, 79

SLAM Simultaneous Localization and Mapping. 34–36, 43, 46, 62

VR Virtual Reality. 5

WLAN Wireless Local Area Network. 15, 17, 51

Glossar

Augmented Reality Erweiterung von realen Objekten. Meist genutzt, um auf einem HUD, oder einer anderen Anzeige, Objekte zu verändern oder Informationen dazu einzublenden. 33, 63

Field Programmable Gate Array Programmierbare Hardware, bestehend aus Logikelementen. 56

Graphics Processing Unit Prozessor speziell für grafische Berechnungen. 19

Head-Mounted-Display Bezeichnet ein Display vor dem Sichtbereich. Bei Integration von einem Head-Tracking-System wird auch von einer VR-Brille gesprochen. VIII, 1, 3

Head-Tracking-System System zum Erfassen von Kopfbewegungen und Orientierung. 1, 3, 11, 15–19, 21, 23, 25, 26, 31, 35, 38, 40, 43, 46, 50, 52, 59, 70, 71

Heads-up-Display Bezeichnet ein Display im Sichtbereich. Wird gewöhnlich genutzt, um dem Nutzer Informationen anzuzeigen. VIII, 10, 32

Immersion Die Virtuelle Realität wird dem Nutzer so dargestellt, dass er in sie eintaucht und als weitgehend reales Erlebnis wahrnimmt. 3, 18, 37, 55, 56

Oculus Rift Eine VR-Brille. VII, 3, 4, 6–8, 11, 12, 15, 17, 19, 20, 24, 26, 32, 35, 37, 39, 40, 42, 49, 52, 53, 57, 60, 70

OpenCV Open Source Computer Vision Library. Bietet Bibliotheken an Funktionen für Bildverarbeitung, mit Fokus auf Computer Vision, für mehrere Plattformen. 5, 38, 41, 43, 46, 67

Roboter Operating System Bietet Bibliotheken und Werkzeuge für Robotersysteme. 5

ROS-Core Kernprogramm von ROS, welches zur Verfügung stehende ROS-Topics verwaltet. 67, 68

ROS-Node Programm das über ROS-Topics kommunizieren kann. VII, 20, 22, 31, 35–37, 43, 46, 50, 51, 59, 69, 70

ROS-Topic Kanal über den ROS-Nodes kommunizieren. VII, 15, 19, 21, 31

Sichtwinkel Winkel von dem kegelförmigen Bereich, der das Sichtfeld bildet. Bei einer Kamera auch Aufnahmewinkel oder Bildwinkel genannt. 12, 27, 28, 38, 40, 46, 55, 59, 60

Simultaneous Localization and Mapping Der Roboter erstellt eine Karte seiner Umgebung, indem es ihm möglich ist, sich zu lokalisieren. 34

Virtual Reality siehe Virtuelle Realität. 5

Virtuelle Realität Virtuelle Nachempfindung der Realität. 3

VR-Brille Ein HMD mit integriertem Head-Tracking-System. 3–5, 9, 19, 55

Literaturverzeichnis

[adept mobilerobots 2011] ADEPT MOBILEROBOTS: *Pioneer 3-DX [09366-P3DX Rev. A]*, 2011

[Anthony 2013] ANTHONY, Sebastian: *DARPA shows off 1.8-gigapixel surveillance drone, can spot a terrorist from 20,000 feet.* 01 2013. – http://www.extremetech.com/extreme/146909-darpa-shows-off-1-8-gigapixel-surveillance-drone-can-spot-a-terrorist-from-20000-feet

[Axel1963 2010] AXEL1963: *CMOS rolling shutter distortion.* 07 2010. – URL http://commons.wikimedia.org/wiki/File:CMOS_rolling_shutter_distortion.jpg. – Zugriffsdatum: 2014-07-10

[Baranov 2014] BARANOV, Ilia: *CLEARPATH?S ROBOT RIBBON CUTTING.* 2014. – URL http://www.clearpathrobotics.com/blog/grand-opening-pr2-ribbon-cutting/. – Zugriffsdatum: 2014-07-07

[Calanar 2014] CALANAR: *Oculus Prototype "Crystal Cove".* 01 2014. – URL https://developer.oculusvr.com/forums/viewtopic.php?f=26&t=5800&start=240. – Zugriffsdatum: 2014-06-29

[Curnow 2014] CURNOW, Richard: *Chrony.* 2014. – URL http://chrony.tuxfamily.org/. – Zugriffsdatum: 2014-07-17

[Cyberbotics 2014] CYBERBOTICS: *Using the Pioneer 3-AT and Pioneer 3-DX robots.* 2014. – URL http://www.cyberbotics.com/dvd/common/doc/webots/guide/section8.3.html. – Zugriffsdatum: 2014-07-05

[David Gossow 2012] DAVID GOSSOW, Julius K.: *compressed_image_transport.* 2012. – URL http://wiki.ros.org/compressed_image_transport. – Zugriffsdatum: 2012-12-19

[de.wikipedia.org 2014] DE.WIKIPEDIA.ORG: *Weitwinkelobjektiv.* 2014. – URL http://de.wikipedia.org/wiki/Weitwinkelobjektiv. – Zugriffsdatum: 2014-07-10

[Edwards 2014] Edwards, Jim: *Oculus Rift Will Finally Go On Sale To Consumers Next Year*. 04 2014. – URL http://www.businessinsider.com/oculus-riftdate-for-sale-to-consumers-2014-4. – Zugriffsdatum: 2014-06-29

[Erik Hals 2014] Erik Hals, Mats Krüger Svensson Mads Falmär W.: *Oculus FPV*. 04 2014. – URL https://github.com/Matsemann/oculus-fpv

[Fantagu 2009] Fantagu: *Verzeichnung3*. 2009. – URL http://de.wikipedia.org/wiki/Verzeichnung#mediaviewer/Datei:Verzeichnung3.png. – Zugriffsdatum: 2014-07-23

[Froehlich 2014] Froehlich, Tim: *Oculus Rift Dev Kit Foto CC BY-SA 3.0*. 2014. – URL http://upload.wikimedia.org/wikipedia/commons/d/dc/Oculus_Rift_Dev_Kit.jpg. – Zugriffsdatum: 2014-06-29

[Google Inc. 2014] Google Inc.: *Cardboard*. 06 2014. – URL https://developers.google.com/cardboard/. – Zugriffsdatum: 2014-06-26

[http://media.moddb.com/ 2014] http://media.moddb.com/: *CS ProMod Radar*. 2014. – URL http://media.moddb.com/images/mods/1/11/10613/cspromod_media_hud_001.jpg. – Zugriffsdatum: 2014-06-23

[Jaguar 2014] Jaguar: *'Jaguar reveals new 'virtual windscreen concept'*. 07 2014. – URL https://www.youtube.com/watch?v=FeK9IkSD_nI. – Zugriffsdatum: 2014-07-12

[Lang 2012] Lang, Ben: *Oculus Rift is an HMD with an Immersive 90-degree FoV, to Be Funded by Kickstater*. 05 2012. – URL http://www.roadtovr.com/oculus-rift-is-an-hmd-with-an-immersive-90-degree-fov-to-be-funded-by-kickstater-video/. – Zugriffsdatum: 2014-06-11

[LaValle 2014] LaValle, Steve: *Sensor Fusion: Keeping It Simple*. 2014. – URL http://www.oculusvr.com/blog/sensor-fusion-keeping-it-simple/. – Zugriffsdatum: 2014-06-09

[Maxwell 2006] Maxwell, Gregory F.: *Shutter speed waterfall*. 07 2006. – URL http://commons.wikimedia.org/wiki/File:Shutter_speed_waterfall.gif. – Zugriffsdatum: 2014-07-10

[Microsoft MSDN 2014] Microsoft MSDN: *Kinect for Windows Sensor Components and Specifications*. 2014. – URL http://msdn.microsoft.com/en-us/library/jj131033.aspx. – Zugriffsdatum: 2014-07-29

[Oculus VR Inc 2013] Oculus VR Inc: *Latency Tester Pre-Orders Now Open!* 09 2013. – URL http://www.oculusvr.com/blog/latency-tester-pre-orders-now-open/. – Zugriffsdatum: 2014-06-25

[Oculus VR Inc 2014] Oculus VR Inc: *DK2*. 2014. – URL http://www.oculusvr.com/dk2/. – Zugriffsdatum: 2014-06-09

[Ogura 2014] Ogura, Takashi: *ROS node for Oculus Rift*. 2014. – URL https://github.com/OTL/oculus. – Zugriffsdatum: 2014-07-05

[Open Source Robotics Foundation 2014] Open Source Robotics Foundation: *Ubuntu install of ROS Hydro*. 2014. – URL http://wiki.ros.org/hydro/Installation/Ubuntu. – Zugriffsdatum: 2014-07-10

[opencv dev team 2014] opencv dev team: *Installation in Linux*. 2014. – URL http://docs.opencv.org/master/doc/tutorials/introduction/linux_install/linux_install.html. – Zugriffsdatum: 2014-07-10

[Orsini 2014] Orsini, Lauren: *How Oculus Rift Intends To Solve SSimulator Sickness"*. 03 2014. – URL http://readwrite.com/2014/03/28/oculus-rift-motion-simulator-simulation-sickness. – Zugriffsdatum: 2014-06-11

[PARROT SA 2014] PARROT SA: *Parrot Bebop Drone*. 2014. – URL http://www.parrot.com/de/products/bebop-drone. – Zugriffsdatum: 2014-05-18

[Patrick Mihelich 2014] Patrick Mihelich, Jeremy L.: *image_proc*. 2014. – URL http://wiki.ros.org/image_proc. – Zugriffsdatum: 2014-07-10

[ROBOTIS 2006] ROBOTIS: *Dynamixel AX-12*, 2006

[Schmitt, Jos. Schneider Optische Werke GmbH 2006] Schmitt, Jos. Schneider Optische Werke GmbH: *Querschnitt Objektiv*. 03 2006. – URL http://de.wikipedia.org/w/index.php?title=Datei:Querschnitt_objektiv.png. – Zugriffsdatum: 2014-07-10

[Sinn 2008] Sinn, Richard: *Virtual pan-tilt-zoom for a wide-area-video surveillance system*. 11 2008

[Strickland 2014] Strickland, Jonathan: *How Virtual Reality Gear Works*. 2014. – URL http://electronics.howstuffworks.com/gadgets/other-gadgets/VR-gear6.htm. – Zugriffsdatum: 2014-06-11

[Thiemann 2014] Thiemann, Thomas: *Inputlag*. 2014. – URL http://www.prad.de/new/monitore/specials/inputlag/inputlag.html. – Zugriffsdatum: 2014-06-09

[Thomas 2014] Thomas: *Oculus Rift featured Crane control*. 06 2014. – URL http://hackaday.io/project/1689. – Zugriffsdatum: 2014-07-10

[Urke 2014] Urke, Eirik H.: *See the Norwegian Armed Forces driving with Oculus Rift*. 2014. – URL http://www.tujobs.com/news/238400-see-the-norwegian-armed-forces-driving-with-oculus-rift. – Zugriffsdatum: 2014-05-18

[whisper.ausgamers.com 2006] whisper.ausgamers.com: *Introduction to Counter-Strike*. 2006. – URL http://whisper.ausgamers.com/wiki/images/Radarcss_hd.png. – Zugriffsdatum: 2014-06-23

[Wiese 2014] Wiese, Hendrik: *Entwicklung einer Basisplattform für Telepräsenzsysteme*. 04 2014

[wikipedia.de 2014] wikipedia.de: *Beschleunigungssensor*. 2014. – URL http://de.wikipedia.org/wiki/Beschleunigungssensor. – Zugriffsdatum: 2014-07-27

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 8. August 2014 Lars Harmsen