



**Hochschule**  
**Augsburg** University of  
Applied Sciences

## Masterarbeit

zur Erlangung des Grades  
Master of Science (M. Sc.) in Informatik

Fakultät für  
Informatik

# Co-Location Toolkit - Ein Game Engine Plugin zur Erstellung von Ad-hoc Co-Location VR-Anwendungen für Geräte mit räumlichem Gedächtnis

Westerbuchberg 79  
83236 Übersee  
+49 8642-969  
fabien.zwick@hs-augsburg.de

Matrikelnummer:  
2082143

erstellt von  
Fabien J. E. Zwick

Hochschule für angewandte  
Wissenschaften Augsburg

An der Hochschule 1  
D-86161 Augsburg

Telefon: +49 821 5586-0  
Fax: +49 821 5586-3222  
[www.hs-augsburg.de](http://www.hs-augsburg.de)  
[info@hs-augsburg.de](mailto:info@hs-augsburg.de)

---

Erstprüfer	Prof. Dr.-Ing. Christian Märtin
Zweitprüfer	Prof. Dr.-Ing. Alexandra Teynor
Abgabedatum	20. Juli 2022
Geheimhaltungsvereinbarung	Nein

---

## **Kurzfassung**

Aktuelle Lösungen zur Erstellung von co-located VR-Anwendungen basieren typischerweise auf der Synchronisierung von plattformspezifischen Raumkern oder geteilten SLAM-Daten, welche meist an Cloud-Services gekoppelt sind. Diese Lösungen benötigen entweder Zugriff auf die Bilder der Kamera oder auf die gesammelten SLAM-Daten, was einige VR-Headsets untersagen. Aus diesem Grund werden in dieser Arbeit zwei bekannte Kalibrationsmethoden engineunabhängig hergeleitet, erweitert und auch eine völlig neue Methode zur Lösung des Co-Location-Problems vorgestellt, die für alle XR-Geräte mit räumlichem Gedächtnis anwendbar sind. Die Arbeit beschreibt, wie diese Methoden in einem zum größten Teil nativ implementierten Toolkit umgesetzt und Entwicklern über ein Unity-Package nutzbar gemacht wurden. Außerdem werden diverse Aspekte des Toolkit, wie etwa die Usability der Methoden, die verwendeten Algorithmen und der auftretende Positionsfehler evaluiert.

# Inhaltsverzeichnis

1.	Einleitung	2
1.1.	Ziele der Arbeit . . . . .	3
1.2.	Aufbau der Arbeit . . . . .	3
2.	Grundlagen	5
2.1.	Virtual Reality . . . . .	5
2.1.1.	Tracking Methoden . . . . .	6
2.2.	Co-Located Anwendungen . . . . .	8
2.2.1.	Co-Location Lösungen für Geräte mit räumlichem Gedächtnis . . . . .	9
2.3.	Mathematische Grundlagen . . . . .	11
2.3.1.	Position, Orientierung und Frame . . . . .	12
2.3.2.	Der Trackingspace . . . . .	16
2.4.	Verwandte Arbeiten . . . . .	18
3.	Transformationsbestimmung	22
3.1.	Problemstellung . . . . .	22
3.2.	Transformationsbestimmung durch eine Referenzpose . . . . .	23
3.2.1.	Ausrichtung des Trackingspace . . . . .	24
3.2.2.	Reduzierung des Rotationsfehlers . . . . .	27
3.2.3.	Kalibrationsfehler . . . . .	28
3.3.	Transformationsbestimmung durch zwei Referenzpunkte . . . . .	30
3.3.1.	Frame Ausrichtung und Rotationsreduzierung . . . . .	31
3.3.2.	Kalibrationsfehler . . . . .	32
3.4.	Transformationsbestimmung durch Punktwolkenregistrierung . . . . .	33
3.4.1.	Frameausrichtung . . . . .	35
3.4.2.	Punktwolkenregistrierung . . . . .	35
4.	Anforderungen	42
4.1.	Projektziel Überblick . . . . .	42
4.2.	Kalibration . . . . .	43
4.2.1.	Berechnung der Transformation zwischen lokalen und remote Koordinatensystem . . . . .	43
4.2.2.	Abbildung von (Remote-)Posen in das lokale Koordinatensystem	43

4.2.3. Rekalibrierung . . . . .	43
4.2.4. Ausrichtung der Frames ( <i>Opt.</i> ) . . . . .	43
4.3. Netzwerk . . . . .	44
4.3.1. Austausch von Kalibrationsdaten . . . . .	44
4.3.2. Anwendungs- und Raumidentifikation . . . . .	44
4.3.3. Automatische Teilnehmererkennung . . . . .	44
4.3.4. Automatische Sitzungsregistration . . . . .	44
4.3.5. Sitzungsverwaltung . . . . .	44
4.3.6. Plattformunabhängigkeit . . . . .	45
4.3.7. Wiederverwendbares Framework (opt.) . . . . .	45
4.4. Render-Engine Integration (Unity) . . . . .	45
4.4.1. Unity Package . . . . .	45
4.4.2. C# API . . . . .	45
4.4.3. GameObject Synchronisierung . . . . .	46
4.4.4. Netzwerk Events . . . . .	46
5. Toolkit Architektur und Umsetzung	47
5.1. Architektur Überblick . . . . .	47
5.2. Die <i>Shared</i> -Bibliothek . . . . .	48
5.3. Die <i>Calibration</i> -Bibliothek . . . . .	49
5.3.1. ICP-Algorithmus . . . . .	52
5.4. Die <i>Networking</i> -Bibliothek . . . . .	61
5.4.1. Socket-Wrapper . . . . .	61
5.4.2. Server-Client Grundgerüst . . . . .	63
5.4.3. TCP Client-Server-Framework . . . . .	67
5.5. Die <i>Core</i> -Bibliothek . . . . .	71
5.6. Unity Package . . . . .	76
5.6.1. C# Plugin API . . . . .	76
5.6.2. Der COLTKManager und Subsysteme . . . . .	78
5.6.3. Calibration & Alignment Komponenten . . . . .	80
5.6.4. Synchronisierung der Nodes . . . . .	81
6. Evaluation	83
6.1. Vergleich der ICP-Algorithmen . . . . .	83
6.2. Evaluation durch "How-To" Demonstration . . . . .	88
6.2.1. Kalibrierung und Ausrichtung durch die Punkte-Methode . . . . .	89
6.2.2. Ausrichtung durch die Guardian-Methode . . . . .	92
6.3. Vergleich der Positionsfehler . . . . .	95
6.3.1. Ablauf und Testaufbau . . . . .	96
6.3.2. Ergebnisse . . . . .	96

6.4.	Usability der Kalibrationsmethoden . . . . .	101
6.4.1.	Testablauf und Aufbau . . . . .	102
6.4.2.	Ergebnisse . . . . .	102
6.5.	Evaluation der Anforderungen . . . . .	106
7.	Zusammenfassung und Ausblick	109
7.1.	Zusammenfassung . . . . .	109
7.2.	Ausblick . . . . .	111
A.	Anhang: Usability Evaluation der Methoden	114
A.1.	Testleitfaden . . . . .	114
A.2.	Antworten der Tester bei der Befragung . . . . .	123
A.3.	Ausgefüllte SUS-Fragebögen . . . . .	126
Literatur		146

# 1. Einleitung

Extended Reality (XR)-Geräte sind in unserem täglichen Leben mittlerweile allgegenwärtig. Von iOS- und Android-Smartphones bis hinzu VR- und AR-Headsets, die sowohl von den Verbrauchern als auch von der Industrie immer stärker genutzt werden. In Hinblick auf gemeinsame und soziale Erfahrungen lag der Fokus bisher meist auf räumlich getrennter Nutzung wie z. B. Social VR(z. B. Facebook Horizon [1], Mozilla Hubs[2]) und VR-Mulitplayer-Gaming (z.B. [3]). XR-Geräte haben jedoch auch das Potenzial, höchst ansprechende Erlebnisse in einer geteilten Umgebung zu schaffen [4], bei denen die Realität der Menschen im Raum ergänzt oder sogar vollständig ersetzt werden kann. In jüngster Zeit haben die Betreiber der XR-Plattformen verstärkt daran gearbeitet, co-location Anwendungen zu ermöglichen. Dies wurde durch das inzwischen weit verbreitete 6DoF-Inside-Out Tracking vorangetrieben, welches interne Sensoren zur Bestimmung der Echtzeitposition und -ausrichtung eines Geräts in Bezug auf die Umgebung verwendet. Die hierbei verwendeten Verfahren sind zunehmend robuster und zuverlässiger geworden, wodurch sie beispielsweise das charakteristische Merkmal des *Meta Quest* VR-Headsets wurden. Bei Verwendung eines inside-out 6Dof-Tracking muss zur Erstellung von Co-Location-Erlebnissen die Transformation zwischen den Koordinatensystemen jedes XR-Geräts oder zwischen jedem Gerät und einem gemeinsamen System bestimmt werden können [5].

Aktuelle Ansätze von Google[6], Microsoft[7] und Apple[8] setzen auf synchronisierte (oft cloudbasierte) Raumanker, welche gemeinsame Bezugspunkte für alle Teilnehmer ermöglichen oder auf synchronisierte SLAM-Daten[9], um nur ein Koordinatensystem für alle Geräte zu verwenden. Bei diesen Lösungen handelt es sich jedoch häufig um „Black Boxes“, die eine enge Kopplung an spezifische SDKs vorsehen und sich häufig auf internetabhängige, cloudbasierte APIs stützen. Außerdem untersagen manche Geräte wie die *Meta Quest* den Zugriff auf die Kamera und auf die SLAM-Daten, sodass die zur Verfügung stehenden APIs und SDKs nicht verwendet werden können.

Anderseits wurden in der Forschung schon co-located Erfahrungen untersucht, bei denen solche Abhängigkeiten nicht bestanden [5, 10, 11, 12, 13]. Die Methoden, mit denen co-located Erlebnisse ermöglicht wurden, sind jedoch entweder nicht ausreichend dokumentiert oder haben zu starke Game-Engine Abhängigkeiten, um eine Replikation zu ermöglichen. Außerdem wird keiner der bisher vorgestellten Methoden, die bei 6DoF-VR-Headsets oft zur Verfügung gestellte Spielfeldbegrenzung verwendet.

## 1.1. Ziele der Arbeit

Das Ziel dieser Arbeit ist die Erstellung eines Toolkits (inkl. Plugin), das Entwicklern die Erstellung von co-located VR-Anwendungen ermöglicht. Hierfür werden bereits aus anderer Literatur bekannte Lösungen, Engine unabhängig neu hergeleitet und auch eine völlig neue Methode entwickelt, welche die Spielfeldbegrenzung in die Berechnungen einbezieht. Bei den bereits existierenden Lösungen werden die Koordinatensysteme aller Teilnehmer an dieselbe Position in der echten Welt verschoben (ausgerichtet). Diese Arbeit soll die notwendigen Berechnungen zeigen, die nötig sind, um auch ohne Ausrichtung der Koordinatensysteme die Tracking-Nodes in das jeweilige Koordinatensystem eines Teilnehmers abzubilden.

Alle vorgestellten Methoden sollen keinen Zugriff auf die Kamera oder SLAM-Daten der Geräte benötigen.

Diese Methoden sollen dann in einer nativen Bibliothek umgesetzt werden, damit keinerlei Abhängigkeiten an eine Game-Engine bestehen. Dennoch soll für die Unity-Engine eine Package bereitgestellt werden, welches den Umgang mit der nativen Bibliothek erleichtert. In der existierenden Lösungen, die bereits ein Skelett oder Framework zur Erstellung von co-location Anwendungen bereitstellt, wird die Synchronisation zwischen Teilnehmern auf Engine-Ebene umgesetzt und sind somit nicht von der Game-Engine zu lösen. Um das Plugin engineunabhängig zu halten, soll die Synchronisation ebenfalls auf nativer Ebene geschehen.

## 1.2. Aufbau der Arbeit

In Kapitel 2 werden zunächst alle Grundlagen behandelt, die für ein gutes Verständnis der Arbeit benötigt werden. Zum Beispiel wird erklärt, worum es sich bei „co-location“ handelt. Des Weiteren wird auf die minimalen mathematischen Grundlagen eingegangen. Zuletzt werden einige wissenschaftliche Arbeiten behandelt, welche sich mit einer ähnlichen Problemstellung befassen.

In Kapitel 3 wird zu Beginn das grundlegende mathematische Problem beschrieben, welches durch die daraufhin vorgestellten drei Methoden gelöst wird. Bei jeder Methode wird auf die Berechnung der Transformation, die Verwendung zur Ausrichtung und ggf. auf den zu erwartenden Fehler eingegangen. In diesem Kapitel werden neben den Neuinterpretationen der bereits bekannten Methoden als Transformationsabbildungen auch eine völlig neue Methode vorgestellt, die eine Punktwolkeregistrierung verwendet, um die benötigte Transformation zu berechnen.

In Kapitel 4 werden alle Anforderungen an das im Laufe dieser Arbeit entwickelte Toolkit und das darin enthaltene Plugin beschrieben. Die Anforderungen wurden hierbei

aufgeteilt in die Kategorien Kalibration, Netzwerk und Render-Engine.

In Kapitel 5 wird der Softwareentwurf und dessen Umsetzung beschrieben, welcher die in Kapitel 2 beschriebenen Methoden beinhaltet. Es wird die Architektur und die vier Bibliotheken beschrieben, aus denen sich der native Teil des Toolkits zusammensetzt. Außerdem werden die Unity-Komponenten beschrieben, die die Verwendung des Toolkits auf Engine-Ebene ermöglicht.

In Kapitel 6 werden mehrere Aspekte des Toolkits evaluiert. Es werden zunächst die für die neu vorgestellte Methode benötigten Algorithmen miteinander verglichen. Danach wird die Funktionalität des Toolkits evaluiert, indem die Anwendung demonstriert wird. Außerdem wird ein Verfahren vorgestellt, mit dem die Anwendbarkeit der Methoden in einem Raum bewertet werden kann. Durch die konkrete Umsetzung der vorgestellten Methoden wird dann noch deren Gebrauchstauglichkeit bewertet. Zuletzt wird evaluiert, welche Anforderungen aus Kapitel 4 erfüllt wurden.

Im letzten Kapitel wird die Arbeit und ihre Ergebnisse noch einmal zusammengefasst.

## 2. Grundlagen

In diesem Kapitel werden zunächst die wichtigsten Grundlagen und Begriffe erläutert, die für ein gutes Verständnis des Hauptteils der Arbeit benötigt werden. Als Erstes wird eine kurze Einführung in die VR- bzw. XR-Technologie und deren Tracking-Methoden gegeben. Danach wird erklärt, was man unter Co-Location-Anwendungen versteht und wie diese aktuell umgesetzt werden. Als Nächstes wird auf die wichtigsten mathematischen Grundlagen eingegangen, die für ein gutes Verständnis der vorgestellten Lösungen nötig sind. Zuletzt werden verwandte wissenschaftliche Arbeiten gezeigt, die sich mit ähnlichen Themen wie diese Arbeit befassen.

### 2.1. Virtual Reality

Virtual Reality (kurz VR) ermöglicht es Benutzern, eine virtuelle Welt immersiv zu erleben. Der Benutzer erlebt eine komplett computergenerierte Umgebung, taucht vollkommen in die virtuelle Welt ein und interagieren mit dieser, während die reale Welt dabei ausgeblendet ist. Hierfür wird meist ein Head-Mounted-Display (HMD) verwendet. In diesem Headset befindet sich jeweils ein hochauflöster Bildschirm für jedes Auge, durch die der Nutzer die Welt und deren Tiefe wahrnehmen kann. Diese HMDs besitzen meist interne und oft auch externe Sensoren, mit denen die Bewegungen des Benutzers erfasst werden können, um diese entsprechend in der virtuellen Welt mit drei (Rotation) oder sechs Freiheitsgraden (Rotation und Translation) darstellen zu können. Um die Immersion noch zu erhöhen, bieten viele VR-Headset-Hersteller auch Controller an, um die Hände in VR darstellen zu können. Manche Hersteller bieten auch universelle Tracker an, mit denen beliebige Objekte getrackt oder sogar ein Ganz-Körper-Tracking umgesetzt werden kann. Zu den populärsten VR-Headsets gehören beispielsweise die *HTC Vive Pro*[14], *Meta Rift*[15] und die *Meta Quest 2*[16]. Die zuletzt genannte VR-Brille ist die Zielplattform dieser Arbeit und ist in Abbildung 2.1 zu sehen.

Zu den mit VR verwandten Technologien zählen *Augmented Reality (AR)* und *Mixed Reality (MR)*, welche sich zusammen unter dem Begriff *Extended Reality (XR)* zusammenfassen lassen. Bei Augmented Reality wird die reale Welt erweitert. Hierfür werden virtuelle Objekte und Informationen in die reale Welt eingefügt. Dies können z. B.



Abbildung 2.1.: Das Meta Quest 2 VR-Headset [16]

3D-Modelle, Bilder, Text oder Animationen sein. Bei AR bleibt aber die reale Umgebung Mittelpunkt der Wahrnehmung und die Interaktion mit den virtuellen Objekten ist meist nur sehr eingeschränkt möglich. Um AR erleben zu können ist kein HMD nötig, sondern kann mit einem herkömmlichen Smartphone oder Tablett erlebt werden.

Mixed Reality verbindet Elemente aus VR und AR. Aus der virtuellen und realen Umgebung entsteht eine neue Umwelt, in der der Benutzer mit der realen und virtuellen Umgebung interagiert. Physikalische Objekte haben hierbei Einfluss auf digitale Elemente. Ein populäres Beispiel eines solchen MX-Headsets ist die Microsoft Hololens 2[17].

Laut dem Titel dieser Arbeit soll das Toolkit die Erstellung von co-located VR-Anwendungen ermöglichen, da das *Meta Quest*-Headset die Zielplattform darstellt. Die in dieser Arbeit vorgestellten Kalibrierungs- und Ausrichtungsmethoden können jedoch für alle XR fähigen Geräte verwendet werden, sofern sie über ein räumliches Gedächtnis verfügen oder ein externes Trackingverfahren verwenden. Mit Ausnahme der Punktwolke-Kalibrierung. Hierfür muss das Gerät das Konzept eines Guardians verwenden.

Alle oben genannten Geräte verwenden entweder Outside-in oder Inside-out Tracking, um die Bewegungen des Headsets zu erfassen. Der Unterschied dieser Trackingverfahren wird im nächsten Unterkapitel erläutert.

### 2.1.1. Tracking Methoden

Wie bereits erwähnt, ist das Zielgerät des in dieser Arbeit erstellten Plugins die *Meta Quest 2*. Diese verwendet Inside-Out Tracking zur Positionsbestimmung. Was dies ist und welche Unterschiede es zu anderen Verfahren gibt, wird in Folgendem erläutert. VR bzw. MX Head-Mounted-Displays verwenden entweder *Outside-In* oder *Inside-Out* Trackingverfahren, um die Position und Orientierung des Geräts und ggf. der Controller im Raum festzustellen.

## Outside-In Tracking

Bei Outside-In Trackingsystemen werden externe Kameras verwendet, um die Pose des Headsets und der Controller zu berechnen. Hierfür werden zwei oder mehr Kameras in einem Raum platziert und so konfiguriert, dass das System die Position der Sensoren relativ zueinander und relativ zu einem Welt-Referenz-System kennt. Wenn das System konfiguriert ist, tracken die Kameras Referenzpunkte auf dem Headset und den Controllern. Da die Positionen der Punkte auf dem Gerät bekannt sind und die Positionen der Kameras nach der Konfiguration ebenfalls, ist es möglich, die Position des Headsets und Controller durch Berechnungen zu rekonstruieren. Dieses Trackingverfahren ist äußerst präzise. Geräte, die dieses Verfahren verwenden, sind zum Beispiel die *Meta Rift 1*, das *Optitrack*-System[18] und das *PSVR*-Headset[19].

## Inside-Out Tracking

Bei Inside-Out Tracking kann noch einmal zwischen markerbasiertem und markerlosem Inside-Out Tracking unterschieden werden. Beide Verfahren haben gemeinsam, dass sie ausschließlich interne Sensoren verwenden. Bei markerbasiertem Tracking müssen jedoch im Raum (oft Infrarot oder QR ähnliche) Marker platziert werden. Ein Beispiel hierfür ist das SteamVR-Trackingsystem[20], welches vom HTC-Vive-HMD verwendet wird. Dieses wird in Artikeln oft fälschlicherweise als Outside-In Tracking bezeichnet[21, 22], da externe Basisstationen aufgestellt werden müssen. Diese sind jedoch keine Sensoren, sondern senden Laserstrahlen aus, die von den Sensoren auf dem Headset und den Controllern aufgefangen werden. Die Basisstationen sind somit komplett passiv und müssen nicht mit einem Computer verbunden werden.

Für das markerlose Inside-Out Tracking wird (bei Stand-alone-Geräten) keinerlei externe Peripherie benötigt. Geräte mit diesem Trackingverfahren verwenden mehrere integrierte Kameras, um die eigene Position im Raum zu bestimmen. Da die Kameras ihre relative Position zueinander kennen, ist keine Kalibration notwendig. Die Bilder der Kameras werden auf Feature-Punkte untersucht. Für Feature-Punkte, die in von mehreren Kameras erkannt werden, können die 3D-Positionen berechnet werden und daraus ein grobes 3D-Modell des umgebenden Raumes erstellt werden. Durch die Beobachtung der Bewegung dieser Punkte über einen Zeitraum ist es durch SLAM-Algorithmen möglich, die Bewegung des Headsets zu berechnen und somit Positionstracking zu ermöglichen. Bei Outeside-In und markerbasiertem Tracking, kann ein Koordinatensystem benutzt werden, das auch für mehrere Geräte konsistent ist. Da es bei markerlosen Inside-Out Tracking keine gemeinsamen Punkte zwischen mehreren Headsets gibt, erstellt jedes Headset sein eigenes lokales Koordinatensystem. Außerdem ist die Präzision dieses Verfahrens

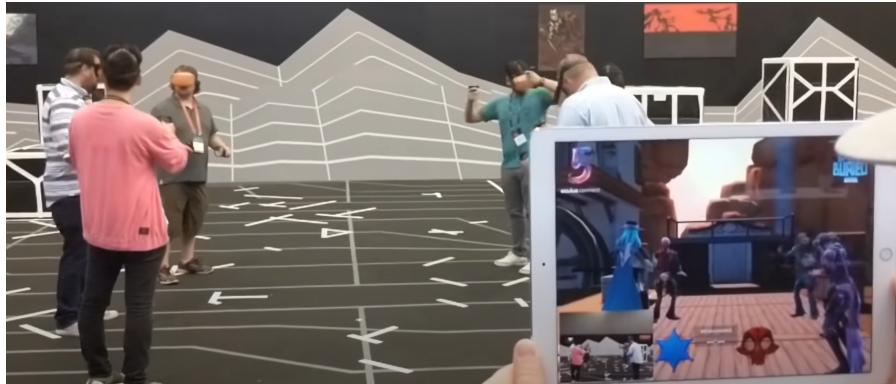


Abbildung 2.2.: Beispiel für die Überlagerung der virtuellen und realen Welt bei co-located Anwendungen[9]

sehr abhängig von der Umgebung, in der sie angewandt wird. Beispielsweise kann es zu Problemen bei zu wenig Licht, Spiegelungen oder monotonen Einrichtung kommen.

## 2.2. Co-Located Anwendungen

Da das Ziel dieser Arbeit ein Toolkit zur Erstellung von Co-Location-Anwendung ist, wird in diesem Unterkapitel erklärt, was mit Co-Location im VR-Kontext gemeint ist.

Co-Located oder auch Shared-Room VR-Applikationen sind Anwendungen, in denen sich die Nutzer in einem Raum befinden und sich in der virtuellen Welt dort wahrnehmen können, wo sie sich in auch in der realen Welt befinden. Die virtuelle Welt ist hierbei für alle Teilnehmer konsistent. Ist ein Objekt beispielsweise für einen Nutzer in der Mitte des Raumes, ist es dies auch für alle anderen Benutzer. Man kann sich die virtuelle Welt quasi als Ebene vorstellen, die die echte Welt überlagert. In Abbildung 2.2 ist ein Auszug des Spiels *Dead and Buried*[9] zu sehen, in dem das soeben beschriebene gut visuell darstellt wird. Ihre Spielcharaktere befinden sich in der virtuellen Welt an der gleichen Position wie in der echten Welt. Bekannte Beispiele für Co-Located Anwendungen sind die Arena-Scale Mehrspieler Erlebnisse, die von TrueVR[23] und Hologate[24] entwickelt werden. Bei diesen Spielen treffen sich mehrere Spieler in einer Arena und können gemeinsam, kooperativ oder gegeneinander verschiedene VR-Spiele erleben.

Die meisten Shared-Room VR-Anwendungen benutzen derzeit Outerside-In-, markerbasiertes Inside-Out Tracking oder eine Kombination dieser beiden. Da bei diesen Verfahren die relativen Positionen der Markierungen oder Sensoren allen Geräten bekannt sind, kann die Position und Orientierung aller Geräte trivial in einem gemeinsamen Koordinatensystem erfasst werden. Bei markerlosem Inside-Out Tracking gibt es keine statischen

Punkte, die die Geräte erkennen, um somit ihre Pose bestimmen zu können. Aus diesem Grund wurden für solche Geräte andere Lösungen entwickelt, um auf dasselbe Ergebnis zu kommen. Welche dies sind und warum sie sich nicht für die Verwendung mit der *Meta Quest* eignen, wird in Folgendem erläutert.

### 2.2.1. Co-Location Lösungen für Geräte mit räumlichem Gedächtnis

Wie oben erwähnt, verwenden inside-out Geräte meist SLAM[25] zur Bestimmung der eigenen Position und Orientierung im Raum. SLAM-Verfahren generiert Wissen über die reale Umgebung mithilfe optischer Daten, welche mit den Daten der IMU (Internal Measurement Unit) kombiniert werden und somit Positionsverfolgung zu ermöglichen. SLAM-Verfahren können durch relative Positionsverfolgung umgesetzt sein (z. B. auf der Grundlage von Optical-Flow[26] und IMU-Daten), aber verfügen meist auch über ein räumliches Gedächtnis der Umgebung (z. B. durch gespeicherte Punktwolken der Umgebungsmerkmale). Das bedeutet, dass die Positionierung absolut sein kann, wenn genügend Wissen der Umgebung vorhanden ist. Hierbei ist zu erwähnen, dass das Tracking nur in Bezug auf jedes einzelne Gerät absolut und intern konsistent ist. Die Koordinatenräume zwischen den Geräten sind jedoch unterschiedlich (Abhängig davon, wo und mit welcher Orientierung das VR-Gerät gestartet wurde). Um trotz der unterschiedlichen Koordinatenräume co-located Anwendungen erstellen zu können, gibt es bereits Lösungen, die sich in die Kategorien *Ausrichtung durch Features* und *Synchronisiertes SLAM* einordnen lassen. *Ausrichtung* bedeutet in diesem Kontext die Translation und Rotation der Koordinatensysteme der einzelnen Geräte, sodass sie sich am Ende überlagern bzw. in der realen Welt die gleiche Position und Orientierung haben. Genauer wird dies in Kapitel 3 beschrieben.

#### Ausrichtung durch Features

Eine Möglichkeit, eine Ausrichtung durch Features zu erreichen, ist die Nutzung von externer Zusatzhardware. Hier wird sozusagen ein Gerät mit markerlosem Inside-Out Tracking zu einem markerbasiertem inside-out Gerät. Ein Beispiel hierfür ist *SynchronizeAR*[27]. In diesem Papier benutzen Huo et al. ein Ultra-Wide Bandwidth (UWB) Modul, das an jedes XR-Gerät befestigt wurde. Durch diese Module konnten die Entfernung zwischen allen Geräten einer Sitzung gemessen werden, wodurch die Positionen (relativ zueinander) berechnet werden konnten. Sobald die Synchronisation beendet war, wurde auf das interne SLAM-Tracking des Geräts gesetzt, um die nachfolgenden Positionsaktualisierungen bereitzustellen, während sich die Benutzer im Raum bewegten. Dieser Ansatz war sehr effektiv, erforderte jedoch zusätzliche und maßgeschneiderte Hardware, die an jedem XR-Gerät angebracht werden musste.

Bei Geräten, die den Zugriff auf die Kamera erlauben, können auch gemeinsam getrackte Markierungen für die Ausrichtung verwendet werden. Zum Beispiel reichen ArUco-Markierungen[28] oder LightAnchors[29] mit parallelen Positionen in der virtuellen Szene für ein Gerät aus, um sein Koordinatensystem ausrichten. Jede größere AR-Plattform hat einen Mechanismus, um dies implizit ohne physikalische Markierungen, typischerweise durch Raumanker (eng. spatial anchors) zu ermöglichen. Raumanker definieren einen Punkt im Raum, der von jedem Gerät getrackt werden kann, indem Features im physikalischen Raum identifiziert werden. Das Microsoft Mixed Reality SDK unterstützt beispielsweise solche Anker [30], welche mit mehreren Geräten geteilt werden können. Hierbei unterstützt das SDK die größten AR-Toolkits (ARKit, ARCore und Microsoft) durch die Nutzung der *Azure Spatial Anchors*[31]. ARCore (Android) und ARKit (iOS) stellen mit *Cloud Anchors*[32] und *ARWorldMap*[8] auch ihre eigenen Services für Raumanker zur Verfügung.

Durch einen Raumanker können die individuellen Koordinatensysteme jedes XR-Geräts auf den Anker ausgerichtet werden. Gibt es beim Identifizieren und Tracking einer einzelnen Markierung oder Feature Ungenauigkeiten, verursacht dies jedoch einen starken Positionsfehler der Teilnehmer.

Wie genau Koordinatensysteme an Markierungen ausgerichtet werden können, wird in Kapitel 3 beschrieben. Außerdem wird dort auch genauer auf den möglichen Fehler eingegangen.

### Synchronisiertes SLAM

Eine Alternative zur Ausrichtung durch Features ist die Synchronisation der SLAM-Daten. Anstelle eine oder mehrere Markierungen zu tracken und das System an diesen auszurichten, können Geräte stattdessen gemeinsam [6] und kollaborativ die Umgebung abbilden[33], sodass sie alle denselben Koordinatenraum und dieselbe absolute Positionierung innerhalb dieser Umgebung teilen. In 2018 demonstrierte *Meta* bereits ein Beispiel für den synchronisierten SLAM Ansatz[9]. Dabei zeigten sie ein arena-scale co-located VR-Spiel, in dem die Synchronisation durch die gemeinsame Nutzung einer vorab erfassten Karte der Umgebung erreicht wurde, auf die sich alle Headsets beziehen konnten. Dadurch konnte das Inside-Out Tracking die absolute Position eines Geräts relativ zur Umgebung bestimmen und somit war keine Ausrichtung des Koordinatensystems nötig.

Synchronisiertes SLAM und geteilte Features sind die gebräuchlichsten Verfahren für co-located XR-Anwendungen. Allerdings gibt es einige Aspekte, die man beachten sollte. Bei synchronisiertem SLAM gibt es insbesondere Probleme hinsichtlich der Ende-zu-Ende-Latenz bei der Erkennung und Ausrichtung, der Bandbreitennutzung (vor allem bei der anfänglichen Synchronisation der Karte) und Rechenanforderungen (vor allem, wenn die Verarbeitung der Karte in die Cloud ausgelagert wird).

Beide Lösungen haben die folgenden Aspekte gemeinsam[34]:

- **Datenschutz:** Sie erfordern die Übertragung von Daten wie Geräteorientierung, Punktwolken von Kameradaten. Die gemeinsame Nutzung dieser Daten wirft Bedenken hinsichtlich des Datenschutzes auf, da die reale Umgebungen kartiert werden und auf anderen Geräten zugänglich gemacht werden kann.
- **Internet Zugang:** Sie erfordern oft eine aktive Internetverbindung zu einer Cloud-Infrastruktur, was für öffentliche Einsetze einen kritischen Fehlerpunkt darstellen kann.
- **Plattformspezifische Abhängigkeiten:** Es gibt kein SDK, das alle Geräte unterstützt.

Der letzte Punkt ist für diese Arbeit am relevantesten. *Meta* veröffentlichte seit 2018 keine API oder Ähnliches, um Entwicklern die co-location Funktionalität zur Verfügung zu stellen[35]. Stattdessen wurde der Nutzerbedingung noch folgende Einschränkung hinzugefügt: „Unless separately approved in writing by Oculus, you will not... modify the tracking functionality (including the implementation of any custom co-location functionality) on your Software“[35].

Falls *Meta* in Zukunft doch eine Co-Location SDK veröffentlicht, bleibt die Frage offen, wie Geräte anderer Hersteller für gemeinsame Erlebnisse integriert werden können. Die Kalibrationsmethoden, welche im Laufe dieser Arbeit besprochen werden (siehe Kapitel 3), sind universell für alle Plattformen anwendbar (sofern gewisse Anforderungen erfüllt sind) und durch die Implementierung in C++ kann das Toolkit für die meisten Architekturen nativ kompiliert werden. Außerdem benötigt das Plugin keinen Internetzugang. Ist die Darstellung anderer Teilnehmer gewünscht, ist die Verbindung zu einem lokalen Netzwerk ausreichend.

## 2.3. Mathematische Grundlagen

In dieser Arbeit werden mehrere Kalibrationsansätze vorgestellt, die zum Verständnis alle ein gewisses Grundwissen der linearen Algebra erfordern. Um also die Lösungsansätze dieser Methoden gut nachvollziehen zu können, ist es wichtig, die relevanten mathematischen Grundlagen zu kennen. In Folgendem werden diese erläutert. Hierbei wird jedoch nicht auf Herleitungen etc. eingegangen, da dies den Rahmen dieser Arbeit sprengen würde und bereits durch eine Vielzahl anderer Literatur [36, 37, 38] abgedeckt ist.

### 2.3.1. Position, Orientierung und Frame

Eine der Hauptaufgaben des Plugins dieser Arbeit ist das Abbilden von Positionen und Orientierungen von externen Koordinatensystemen in das lokale System. Deshalb wird in Folgendem kurz erläutert, wie diese definiert sind und welche Notation in dieser Arbeit verwendet wird.

#### Position

In einem kartesischen Koordinatensystem kann jeder Punkt mithilfe eines  $3 \times 1$  Positionsvektors dargestellt werden. Da Positionen neben dem Ursprungssystem auch relativ zu anderen Koordinatensystemen beschrieben werden können, müssen Vektoren mit einer Zusatzinformation ergänzt werden, die angibt, in welchem Koordinatensystem dieser definiert ist. In dieser Arbeit werden Vektoren mit einem vorangestellten Hochzeichen geschrieben, welches das Koordinatensystem angibt, auf das er sich bezieht. Der Punkt  ${}^A P$  beispielsweise gibt an, dass seine Komponenten sich auf die Entferungen entlang der Achsen des Koordinatensystems  $\{A\}$  beziehen. Ein Punkt im Koordinatensystem von  $\{A\}$  ist also mittels eines Positionsvektors definiert als:

$${}^A P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \quad (2.1)$$

#### Orientierung

Oft ist es nötig, nicht nur einen Punkt im Raum zu beschreiben, sondern auch die Orientierung eines Körpers an diesem Punkt. Bei einem VR-Controller zum Beispiel reicht es nicht aus, zu wissen, wo sich dieser im Koordinatensystem befinden. Man muss auch seine Orientierung kennen, um ihn in der Render-Engine korrekt anzuzeigen oder Interaktionen (z. B. per Laserstrahl ausgehen vom Controller) durchführen zu können. Um die Orientierung eines Körpers zu beschreiben, kann an dessen Position ein neues Koordinatensystem angefügt werden, welches die Orientierung relativ zu einem Referenzsystem beschreibt. Positionen von Punkten werden also durch Vektoren beschrieben und die Orientierung von Körpern durch angefügte Koordinatensysteme. Eine Möglichkeit ein angefügtes Koordinatensystem  $\{B\}$  zu beschreiben, besteht darin, die Einheitsvektoren der drei Hauptachsen in Bezug auf ein Referenzkoordinatensystem  $\{A\}$  zu schreiben. Die drei Einheitsvektoren, die das Koordinatensystem  $\{B\}$  aufziehen, werden als  $\hat{X}_B$ ,  $\hat{Y}_B$  und  $\hat{Z}_B$  bezeichnet. Mit  $\{A\}$  als Referenzsystem:  ${}^A \hat{X}_B$ ,  ${}^A \hat{Y}_B$ ,  ${}^A \hat{Z}_B$ . Wie in Gleichung 2.2 zu sehen, können diese drei Einheitsvektoren spaltenweise in eine  $3 \times 3$  Matrix geschrieben werden, um die Rotationsmatrix  ${}^A_B R$  zu erhalten.

$${}^A_B R = \begin{bmatrix} {}^A \hat{X}_B & {}^A \hat{Y}_B & {}^A \hat{Z}_B \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad (2.2)$$

Diese beschreibt die Rotation von Koordinatensystem  $\{B\}$  relative zu  $\{A\}$ . Die Reihen dieser Matrix beschreiben außerdem die Einheitsvektoren von  $\{A\}$  ausgedrückt in  $\{B\}$ .

$${}^A_B R = \begin{bmatrix} {}^A \hat{X}_B & {}^A \hat{Y}_B & {}^A \hat{Z}_B \end{bmatrix} = \begin{bmatrix} {}^B \hat{X}_A^T \\ {}^B \hat{Y}_A^T \\ {}^B \hat{Z}_A^T \end{bmatrix} \quad (2.3)$$

Die Rotationsmatrix  ${}^B_A R$  (Rotation von  $\{A\}$  mit  $\{B\}$  als Basis) erhält man also mit:

$${}^B_A R = {}^A_B R^T. \quad (2.4)$$

### Frames

Die Kombination aus Positionsvektor und Orientierung wird in Gebieten wie der Robotik so häufig verwendet, dass in der Literatur hierfür oft die Bezeichnung *Frame* (deu. Bezugssystem) verwendet wird[38]. Ein Frame ist also ein Koordinatensystem, welches zusätzlich zur Orientierung einen Positionsvektor enthält, welcher den Ursprung des Koordinatensystems relativ zu einem anderen Frame angibt. Zum Beispiel kann Frame  $\{B\}$  beschrieben werden durch die Rotation  ${}^A_B R$  und den Vektor  ${}^A P_{BURSP}$ , der die Position des Ursprungs angibt .

$$\{B\} = \{{}^A_B R, {}^A P_{BURSP}\} \quad (2.5)$$

### Abbildungen zwischen Frames (Basiswechsel)

Eine der Kernaufgaben des in dieser Arbeit entwickelten Plugins, ist die Umrechnung von Positionen und Orientierungen von einem Koordinatensystem in andere Systeme. Deshalb ist es essenziell zu verstehen, wie Punkte und Vektoren auf verschiedene Koordinatensystemen abgebildet werden können.

Die Ausgangslage ist folgende Situation. Es existiert ein Koordinatensystem  $\{A\}$  und ein Koordinatensystem  $\{B\}$ . Die Position und Orientierung von  $\{B\}$  relativ zu  $\{A\}$  ist bekannt. Ein Punkt  ${}^B P$  ist im Koordinatensystem  $\{B\}$  definiert. Die Position dieses Punktes soll in Koordinatensystem  $\{A\}$  angegeben werden. Dies kann mit der folgenden Gleichung erreicht werden:

$${}^A P = {}^A_B R {}^B P + {}^A P_{BURSP} \quad (2.6)$$

Die obenstehende Gleichung beschreibt eine generelle Transformationsabbildung eines Vektors von einem Frame in einen anderen. Die Gleichung 2.6 kann, wie in 2.7 zu sehen, als Matrixmultiplikation dargestellt werden.

$${}^A P = {}_B^A T \cdot {}^B P \quad (2.7)$$

Wobei  ${}_B^A T$  eine homogene Transformationsmatrix ist, die wie folgt aufgebaut ist.

$$\begin{bmatrix} {}^A P \\ 1 \end{bmatrix} = \left[ \begin{array}{c|c} {}_B^A R & {}^A P_{BURSP} \\ \hline 0 & 1 \end{array} \right] \begin{bmatrix} {}^B P \\ 1 \end{bmatrix} \quad (2.8)$$

Transformationen wurden hier zwar im Kontext von Abbildungen beschrieben, sie können aber auch zur Beschreibung von Frames genutzt werden. Die Beschreibung von Frame {B} relative zu {A} ist  ${}_B^A T$ .

Homogene Transformationen können im Kontext dieser Arbeit drei unterschiedliche Interpretationen besitzen.

1. Als *Beschreibung eines Frames*.  ${}_B^A T$  beschreibt Frame {B} relative zum Frame {A}. Genauer gesagt, sind die Spalten von  ${}_B^A R$  die Einheitsvektoren, die die Basisvektoren von {B} definieren und  ${}^A P_{BURSP}$  gibt die Position des Ursprungs von {B} an.
2. Als *Transformationsabbildung*. Mit  ${}_B^A T$  wird  ${}^B P \rightarrow {}^A P$  abgebildet.
3. Als *Transformationsoperator*.  $T$  operiert auf  ${}^A P_1$  um  ${}^A P_2$  zu erhalten.

### Verkettete Frames und Transformationsgleichungen

Das Abbilden von Punkten und Vektoren zwischen zwei Frames reicht zur Lösung der meisten in dieser Arbeit auftretenden Problemstellungen nicht aus. Oft müssen Abbildungen zwischen mehr als zwei verketteten Frames oder auch nicht bekannte Transformationen zwischen Frames berechnet werden. Um diese Berechnungen anschaulich darstellen zu können, ist die grafische Darstellung der Transformationsberechnungen wie folgt definiert. Ein Frame bzw. Koordinatensystem wird durch drei Pfeile repräsentiert, welche dessen Basisachsen darstellen. Sind nur zwei Pfeile zu abgebildet, zeigt der fehlende Basisvektor in die Arbeit hinein. Pfeile mit gestrichelten Linien zwischen Frames stellen den Vektor vom Ursprung eines Frames zum Ursprung eines anderen Frames dar. Der Frame, auf den der Pfeil zeigt, ist hierbei relativ zum Frame bekannt, in dessen Ursprung der Pfeil startet. In Abbildung 2.3 ist ein Beispiel dieser grafischen Darstellung zu sehen. In der Abbildung ist die Transformation von {C} relativ zu {W} und außerdem {D} relativ zu {C} bekannt. Um mathematische Gleichungen visuell darstellen zu können,

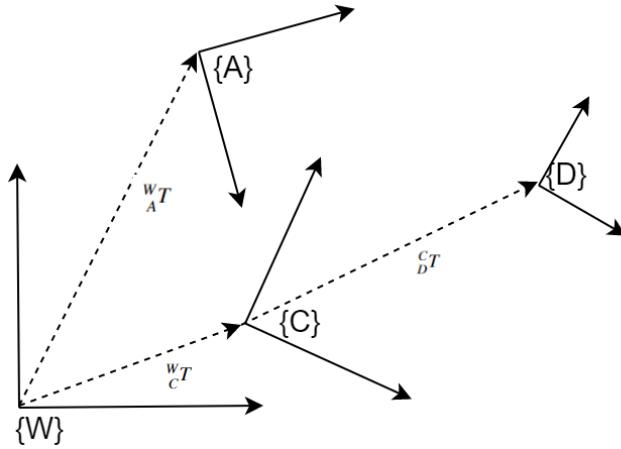


Abbildung 2.3.: Grafische Darstellung der Beziehungen zwischen mehreren Frames

werden diese Pfeile oft mit Transformationen beschriftet. Die Transformation bzw. die Abbildung zwischen Frame  $\{W\}$  und  $\{C\}$  würde zum Beispiel mit  ${}^W_C T$  beschriftet werden. Wie oben erwähnt ist diese Transformation die Beschreibung von Frame  $\{C\}$  relativ zu  $\{W\}$ , aber auch gleichzeitig die Transformation, mit der Punkte und Vektoren von  $\{C\}$  nach  $\{W\}$  abgebildet werden können. Die Richtung eines Pfeils kann durch das Invertieren einer Transformation umgekehrt werden:

$${}^C_W T = {}^W_C T^{-1} \quad (2.9)$$

Die Abbildung 2.3 zeigt den Frame  $\{C\}$  der relativ zu  $W$  bekannt ist (definiert durch  ${}^W_C T$ ) und  $\{D\}$  der relativ zu  $\{C\}$  bekannt (definiert durch  ${}^C_D T$ ) ist. Um Vektoren bzw. Punkte von Frame  $\{D\}$  direkt in Frame  $\{W\}$  abbilden zu können, wird die Transformation  ${}^W_D T$  benötigt. Diese lässt sich durch die Matrixmultiplikation der bekannten Transformationen berechnen. Bildlich kann die Reihenfolge dieser Multiplikationen einfach abgelesen werden. Ausgehend vom Frame, in das abgebildet werden soll, muss den Pfeilen bis zum Zielframe gefolgt werden. Angewandt auf dieses Beispiel erhält man folgende Formel:

$${}^W_D T = {}^W_C T \cdot {}^C_D T. \quad (2.10)$$

Ein relativ zu  $\{D\}$  bekannter Punkt  ${}^D P$  kann dann mit

$${}^A P = {}^W_D T \cdot {}^D P \quad (2.11)$$

nach  $\{W\}$  abgebildet werden.

Neben den Abbildungen zwischen mehreren verketteten Frames ist es für die in dieser Arbeit vorgestellten Lösungen auch oft notwendig, unbekannte Transformationen zwischen Frames zu ermitteln. In Abbildung 2.3 ist zu sehen, dass Frame {D} relativ zu {C} bekannt ist, aber nicht relativ zu {A}. Diese Transformation  ${}^A_D T$  (grafisch ein Pfeil von {A} nach {D}) zu berechnen, kann mittels Transformationsgleichungen erreicht werden. Um  ${}^A_D T$  zu ermitteln, muss zunächst die folgende Gleichung aufgestellt werden:

$${}^W_A T \ {}^A_D T = {}^W_C T \ {}^C_D T \quad (2.12)$$

Grafisch betrachtet wird diese Gleichung aufgestellt, indem man bei Frame {W} startet, sich einen Pfad entlang der Pfeile wählt, bis der Zielframe erreicht ist und alle zugehörigen Transformationen auf eine Seite der Gleichung schreibt. Die andere Seite der Gleichung wird aufgestellt, indem ebenfalls ein Pfad ausgehend von {W} gewählt wird, der als letzte Transformation jedoch die gesuchte Transformation  ${}^A_D T$  enthält.

Diese resultierende Gleichung kann dann durch das Anmultiplizieren von Inversen der bekannten Transformationen so umgestellt werden, dass nach der unbekannten  ${}^A_D T$  aufgelöst wird.

$$\begin{aligned} {}^W_A T \ {}^A_D T &= {}^W_C T \ {}^C_D T \\ {}^W_A T^{-1} \ {}^W_A T \ {}^A_D T &= {}^W_A T^{-1} \ {}^W_C T \ {}^C_D T \\ I \ {}^A_D T &= {}^W_A T^{-1} \ {}^W_C T \ {}^C_D T \\ {}^A_D T &= {}^W_A T^{-1} \ {}^W_C T \ {}^C_D T \end{aligned}$$

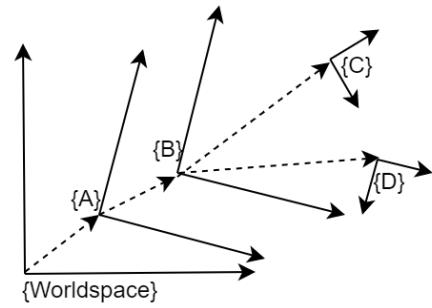
Die meisten Problemstellungen dieser Arbeit können durch das aufstellen von Transformationsgleichungen gelöst werden. Mehr dazu in Kapitel 3.

### 2.3.2. Der Trackingspace

Da im Laufe dieser Arbeit der Begriff *Trackingspace* häufig verwendet wird, wird dieser an diesem Punkt einmal erläutert. Der Trackingspace bezeichnet im XR-Kontext ein spezielles Koordinatensystem bzw. einen speziellen Frame. Je nach genutztem Toolkit/SDK kann dieses Koordinatensystem auch andere Bezeichnungen haben (z.B. *ARSessionOrigin* in ARFoundation[39]), die aber grundlegend dieselbe Aufgabe erfüllen. Jedes XR-Gerät definiert sein eigenes Koordinatensystem. Also seinen eigenen Ursprung und Orientierung, die z. B. beim Starten des Geräts oder beim Öffnen einer App definiert wird. Die Pose des Headsets und (falls vorhanden) der Controller wird immer relativ zu diesem Startpunkt angegeben. Posen sind wie Frames eine Kombination aus Position und



(a) Beispiel einer Hierarchie in der Unity-Engine.



(b) Hierarchien dargestellt durch Transformationen.

Abbildung 2.4.: Eine Hierarchie in Unity (links) dargestellt mithilfe der grafischen Notation als Transformationen (rechts).

Orientierung. Mathematisch besteht zwischen Posen und Frames kein Unterschied (wenn die Skalierung keine Rolle spielt) und werden daher im Laufe der Arbeit als dasselbe behandelt.

Wie eben erwähnt definieren XR-Geräte ihr eigenes Koordinatensystem, in dem die aktuelle Position und Orientierung ihrer getrackten Geräte angegeben wird. Jede Game-Engine definiert ebenfalls ein Koordinatensystem, den sogenannte *Worldspace*. Game-Engines verwenden meist auch einen Szenegrafen[40], mit dem es möglich ist, Hierarchien von Objekt-Transformationen zu definieren. In der Unity-Engine können beispielsweise wie in Abbildung 2.4a zu sehen Hierarchien von Unity-GameObjects erstellt werden. Die in Abbildung 2.4a abgebildete Hierarchie kann in der oben definierten bildlichen Darstellung wie in Abbildung 2.4b visualisiert werden.

Wenn ein Knoten in dieser Hierarchie bewegt oder rotiert wird, bleibt die Position und Rotation aller Kindknoten relativ zum bewegten Knoten gleich. In Abbildung 2.5 ist dies bildlich veranschaulicht. Hier wurde  $\{T\}$  bewegt und rotiert, doch die Frames  $\{N_1\}$  und  $\{N_2\}$  sind relative zu  $\{T\}$  gleich geblieben.

Der Trackingspace ist das Koordinatensystem eines XR-Geräts, dargestellt in der Game-Engine. Die Posen der XR-Tracking-Nodes kommen nicht im Worldspace in der Engine an, sondern relativ zu diesem Trackingspace. Der Trackingspace ist also ein Frame, der relativ zum Worldspace bekannt ist und der die Posen-Updates relativ zu diesem Frame erhält. Wenn noch einmal Abbildung 2.5 betrachtet wird, stellt  $\{W\}$  den Worldspace,  $\{T\}$  den Trackingspace und  $\{N_{1/2}\}$  die Tracking-Nodes dar, so wie sie in einer Game-Engine existieren. Wird also der Trackingspace verschoben, bewegen sie alle getrackten Geräte wie Headset und Controller mit diesem mit. Dies wird in späteren Kapiteln ausgenutzt, um Szenen auszurichten.

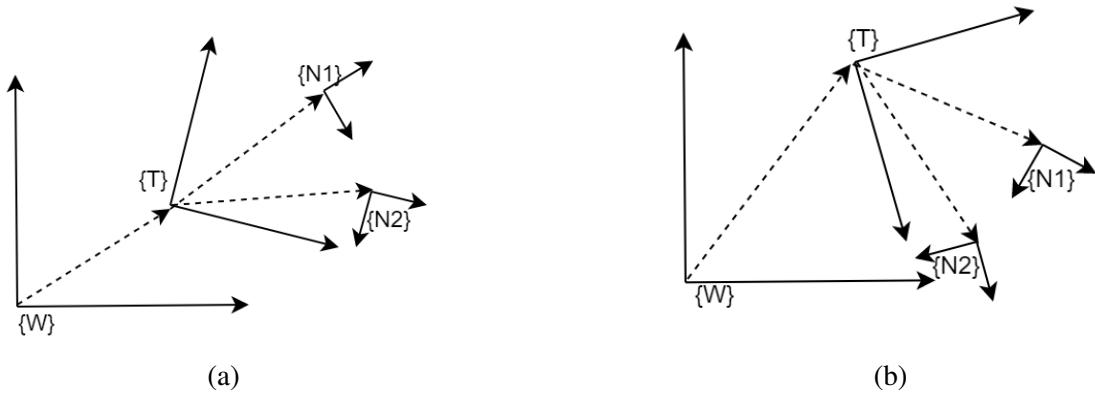


Abbildung 2.5.: Vor und nach der tranlation und rotation des Frames  $\{T\}$ . Die Frames  $\{N1\}$  und  $\{N2\}$  habe relativ zu  $\{T\}$  noch die gleiche Transformation.

## 2.4. Verwandte Arbeiten

In diesem Unterkapitel wird Literatur vorgestellt, die sich mit derselben oder einer ähnlichen Problemstellung beschäftigen wie diese Arbeit.

DeFanti beschreibt in seiner Dissertation[5] seine Erfahrungen mit co-located VR und AR, die mittels mehrerer Experimente im Laufe von fünf Jahren gesammelt wurden. Bereits 2015 entwickelte DeFanti mit seinem Team *Holojam*. Holojam ermöglichte es vier Teilnehmer gleichzeitig an einer co-located Erfahrung teilzunehmen. Zu diesem Zeitpunkt waren fast alle Geräte, die 6DoF (Degrees-of-Freedom) Tracking ermöglichen, PC gebunden. Es bestand also oft die Gefahr, über Kabel zu stolpern, oder man musste wie bei *VRcade*[41] oder *The Void*[42] einen schweren Rucksack mit sich tragen, in dem sich ein Laptop befand. DeFanti löste diese Probleme, indem er bei Holojam auf 3DoF-Headsets (mobiles Endgerät mit Samsung-Gear) im Zusammenspiel mit dem motioncapture Trackingsystem von OptiTrack[18] setzte. Für die korrekte Positionierung und Orientierung der Teilnehmer wurde also das interne 3DoF der Smartphones mit den externen getrackten Markern kombiniert. DeFanti beschreibt in seiner Dissertation, wie diese Daten kombiniert werden können, um die Position der Benutzer zu ermitteln, sowie dessen Orientierung zu korrigieren (3-DOF Geräte leiden unter *drift*[43]), ohne dabei Motionsickness bei den Nutzern auszulösen.

Die zwei darauf folgenden Experimente *I AM ROBOT* und *HOLOJAM in Wonderland* ähneln Holojam technisch sehr. Anstelle von dem ca. 50-tausend Euro teurem OptiTrack-System setzte DeFanti hierbei jedoch auf das deutlich günstigere Trackingsystem der HTC-Vive. Das Grundkonzept blieb jedoch gleich.

Bei *CAVRN* setzte DeFanti erstmals auf 6DoF-Geräte (Google Lenovo Mirage Solo[44]). Da hier nicht mehr auf ein zentralisiertes Trackingsystem gesetzt wird, sondern jedes Gerät

sein eigenes internes Tracking umsetzt, beschreibt DeFanti Lösungen, um die internen Frames der einzelnen Geräte aufeinander auszurichten. Bei einer der beschriebenen Lösungen wird die Differenz (der Rotation und Translation) der Frames berechnet, indem die Headsets sich gegenseitig (mithilfe von ArcUno Markierungen und CV Algorithmen) tracken. Da die *Meta Quest* den Zugriff auf die Kamera untersagt, ist diese Lösung für diese Arbeit unbrauchbar. Die zweite vorgestellte Lösung ist jedoch universell für alle SLAM-basierten XR-Geräte einsetzbar. Bei dieser Lösung werden zwei physikalische Markierungen im Raum benötigt, deren Position bei allen teilnehmenden Geräten in ihrem internen Koordinatensystem ermittelt werden müssen. DeFanti erreichte dies, indem er die Headsets auf die Markierung legt und jeweils die Position durch Knopfdruck bestätigte. Sind diese zwei Punkte bekannt, kann mit Defantis Gleichungen die Translation und Rotation berechnet werden, die auf ein Koordinatensystem angewandt werden muss, damit sich diese überlagern. In ähnlicher Form wird diese Methode in Kapitel 3.3.1 beschrieben und auch im Plugin umgesetzt. Es wird jedoch eine eigene Herleitung und Berechnungen verwenden, die ausschließlich auf Multiplikation von homogenen Transformationen setzt. Außerdem muss bei der Lösung, die in dieser Arbeit vorgestellt wird, der Frame nicht zwingend ausgerichtet werden. Jeder Benutzer kann sein ursprüngliches Koordinatensystem unverändert lassen, hat aber, falls gewünscht, die Möglichkeit eine Ausrichtung durchzuführen.

McGill, Gugenheimer und Freeman setzten in ihrem Papier[34] die eben beschriebene Methode von DeFanti zur Frameausrichtung auf Basis der Unity-Engine um. Neben dieser Methode beschreiben sie eine sehr ähnliche Methode, bei der nur einer anstatt zwei Punkte zur Ausrichtung des Frames verwendet wird. Im Grunde wird hierbei genauso vorgegangen wie bei der verbreiteten Feature-basierten Ausrichtung (siehe 3.2.1). Außerdem formalisierten McGill et al. die theoretische Genauigkeit und Limitationen beider Herangehensweisen. Die Methoden wurden so implementiert, dass sie neben der *Meta Quest* auch jedes andere SLAM-basierte Gerät unterstützt. McGill et al. stellen des Weiteren eine Methode vor, um die Anwendbarkeit dieser Ausrichtungsmethoden in jeder Umgebung bewerten zu können, ohne ein teures optisches Trackingsystem zu benötigen. Stattdessen genügt für ihre Bewertungsmethode ein 30-Dollar Distanz-Laser-Messer. Auch die in der McGill's Arbeit vorgestellte *1-Punkt-Ausrichtung* wird im Plugin dieser Arbeit implementiert. Aber auch hier wird eine Render-Engine unabhängige Lösung hergeleitet und vorgestellt, da die Beschreibung und Implementation von McGill's Team stark von der Unity-Engine-API abhängig ist. Des Weiteren wird in Kapitel 6.3 ein eigenes Verfahren zur Bewertung der Anwendbarkeit einer Methode in beliebigen Umgebungen vorgestellt, welches neben den zwei *Meta Quests* keinerlei zusätzliche Geräte benötigt. Zur Synchronisierung der Teilnehmer benutzt McGill et al. das Open-Source Unity-Package Mirror[45]. Dies ist eine weitere Unity Abhängigkeit, die nicht übertragbar auf andere Render-Engines ist. Das in dieser Arbeit entwickelte Plugin setzt die Synchronisation auf nativer Ebene um und ist deshalb (optional) mit jeder Engine nutzbar, die die definierte

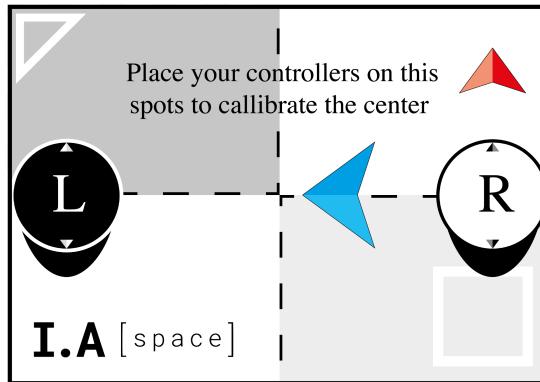


Abbildung 2.6.: Für den Kalibrations-Flow bereitgestellter Druck von Leisi et. al[46]

Schnittstelle implementiert.

Leisi und Sahli beschreiben in [46], wie sie ein Multiuser-Virtual-Reality-Framework für die Unity Engine implementierten. Die verwendete Kalibrationsmethode ist hierbei wiederum das Ausrichten der Frames an einem oder zwei Punkten, wie bereits von DeFanti und McGill beschrieben. Das Framework von McGill und Leisi sind vom Funktionsumfang sehr ähnlich. Beide nutzen dieselben Kalibrationsmethoden und auch dasselbe Networking Package (Mirror). In der Publikation von McGill et al. lag der Fokus jedoch mehr auf den generellen Methoden und deren Fehlertoleranz bzw. Robustheit. Leisi et al. beschreiben in ihrem Papier mehr eine konkrete Implementation dieser Methoden. Ihr Framework beinhalten einen kompletten Kalibrationsflow, welcher für *Meta Quest*, sowie Mobile AR-Geräte nutzbar ist. Dies wird durch die Bereitstellung der in Abbildung 2.6 gezeigten druckbaren Markierung ermöglicht. Auf dem Druck finden sich zwei kreisartige Markierungen, die für XR-Headsets mit Kontroller zur Ausrichtung an zwei Punkten verwendet werden können. AR-Geräte können die Pose des Drucks durch Bilderkennung ermitteln und deren Frame an diesem ausrichten. Dieser Vordruck macht die Kalibration bei verschiedenen Raumgrößen aber sehr unflexibel. Wie in Kapitel 3.3.2 zu lesen, wird der Positionsfehler kleiner, je weiter die zwei Kalibrationspunkte auseinander liegen. Durch den vordefinierten Abstand kann es bei sehr großen Räumen am Rand des Trackingbereichs zu starken Fehlern kommen. Bis auf den Kalibrationsworkflow bietet das Framework von Leisi et al. im Vergleich zu McGill et al. nur ein einziges neues Feature: Synchronisation der Hände. Für die Synchronisierung der Hände und der Teilnehmer wird hier ebenfalls auf das Mirror Unity-Package gesetzt.

Neben der bis hierher gezeigten Literatur gibt es noch weitere Publikationen[47, 48], die im Kern aber dieselben Methoden zur Ausrichtung von Frames verwenden. Diese oben gezeigten Kalibrationsmethoden sind also zurzeit der Standard, wenn es um die Kalibrierung/Frameausrichtung von SLAM-basierten XR-Geräten ohne direkten Kamerazugriff

geht und unterscheiden sich im Wesentlichen nur in der letztendlichen Umsetzung. In [48] werden z.B. anstatt Controller oder Headsets die Hände als Feature-Punkte zum Kalibrieren verwendet. Eine weitere Gemeinsamkeit aller gezeigten Frameworks ist die Abhängigkeit zur Render-Engine Unity. Mit dieser Arbeit und dem dazugehörigen Plugin wird ein Beitrag zur Forschung geleistet, indem die bisherigen Methoden in einer Render-Engine unabhängigen Weise hergeleitet und plattformunabhängig in einem nativen Plugin implementiert werden. Des Weiteren wird ein völlig neues Kalibrationsverfahren vorgestellt, welches die Spielbereichsmarkierungen von Inside-Out HMDs wie der *Meta Quest 2* verwendet, um Transformationen zwischen Koordinatensystem zu berechnen und somit eine Ausrichtung der Frames und die Abbildung der Posen ermöglicht. Eine weitere Abhängigkeit, die die meisten bisher gezeigten Arbeiten gemeinsam haben, ist die Synchronisierung der Teilnehmer auf Engine-Ebene. Um unabhängig von Engine-APIs zu sein, wird im Plugin dieser Arbeit eine native Netzwerklösung implementiert, die ad-hoc Server-Client Verbindungen mit Synchronisierung getrackter VR-Nodes ermöglicht.

# 3. Transformationsbestimmung

In diesem Kapitel wird zunächst die mathematische Problemstellung erläutert, welche zu lösen ist, um das Toolkit implementieren zu können. Danach werden drei Lösungsansätze für dieses Problem beschrieben.

## 3.1. Problemstellung

Jedes XR-Gerät, das Inside-Out Tracking verwendet, definiert sein eigenes internes Koordinatensystem (Trackingspace). Damit synchronisierte Tracking-Nodes (z. B. Headset und Controller) oder statische Objekte (modellierte Szene) für verschiedene Geräte an derselben Stelle mit derselben Orientierung dargestellt werden, müssten sich diese lokalen Koordinatensysteme exakt überlagern. Um dies zu erreichen, wird üblicherweise eine Frameausrichtung durchgeführt. Eine andere Möglichkeit, Tracking-Nodes für die Nutzer konsistent darzustellen, ist die Abbildung der Node-Posen von Remote-Benutzern in das lokale Koordinatensystem des jeweiligen XR-Geräts. In Abbildung 3.1 werden die Koordinatensysteme zweier XR-Geräte mit  $Q_1$  und  $Q_2$  dargestellt.

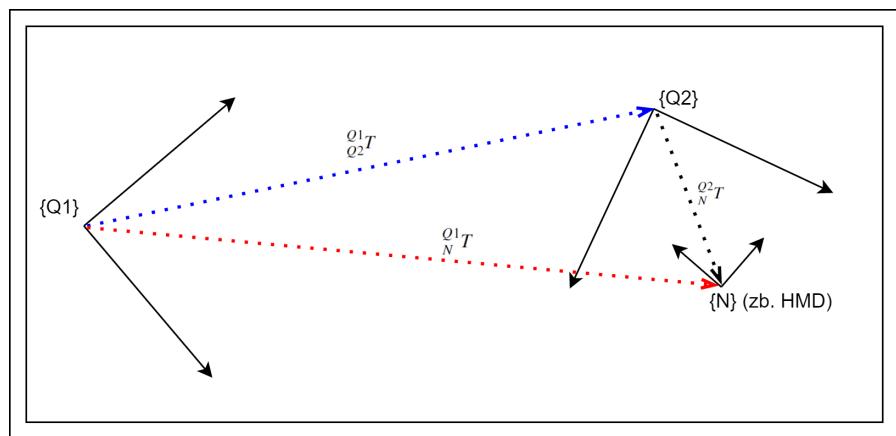


Abbildung 3.1.: Die internen Koordinatensysteme zweier XR-Geräte ( $\{Q_1\}$  und  $\{Q_2\}$ ) und die benötigten Transformationen (Pfeile) um  $\{N\}$  nach  $\{Q_1\}$  abzubilden:  ${}^{Q_1}T_{Q_2}$   ${}^{Q_2}T_N$

$\{N\}$  ist eine Pose, die im Frame von  $\{Q2\}$  bekannt ist. Dies kann z. B. das Headset oder ein Controller des Geräts sein, welches  $\{Q2\}$  definiert. Um die Pose  $\{N\}$  und alle anderen Posen, Punkte und Vektoren von  $\{Q2\}$  nach  $\{Q1\}$  abbilden zu können, muss eine Möglichkeit gefunden werden, die Transformation  ${}_{Q2}^{Q1}T$  (blauer Pfeil in Abbildung 3.1) zu bestimmen. Wenn diese Transformation bekannt ist, kann die Pose  $N$  wie folgt abgebildet werden:

$${}_{N}^{Q1}T = {}_{Q2}^{Q1}T \cdot {}_N^{Q2}T \quad (3.1)$$

Diese Berechnung liefert eine weitere homogene Transformationsmatrix. Aus dieser Matrix kann, wie in Gleichung 3.2 zu sehen, direkt die Translation und Rotation der Pose  $\{N\}$  abgelesen werden. Hierbei sind die Spalten der 3x3 Rotationsmatrix  ${}_{N}^{Q1}R$  die Einheitsvektoren von  $\{N\}$  und  ${}^{Q1}P_{NURSP}$  der Vektor zum Ursprung relativ zum Koordinatensystem  $\{Q1\}$ . Anders gesagt beschreibt  ${}^{Q1}P_{NURSP}$  die Translation und  ${}_{N}^{Q1}R$  die Orientierung von Tracking-Node  $\{N\}$  im Zielkoordinatensystem  $\{Q1\}$ .

$${}_{N}^{Q1}T = \left[ \begin{array}{ccc|c} & {}_{N}^{Q1}R & & {}^{Q1}P_{NURSP} \\ & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (3.2)$$

Durch die Bestimmung einer Transformation  ${}_{Q2}^{Q1}T$  können also alle Tracking-Nodes von einem Koordinatensystem  $\{Q2\}$  in System  $\{Q1\}$  abgebildet werden, ohne dafür eine Ausrichtung des Frames vornehmen zu müssen.

Im Folgenden werden drei Methoden vorgestellt, mithilfe der genau diese Transformation zwischen zwei Koordinatensystemen bestimmt werden kann.

## 3.2. Transformationsbestimmung durch eine Referenzpose

Eine Möglichkeit, die Abbildungs transformation zwischen zwei Frames zu bestimmen, ist mithilfe einer geteilten Pose in der realen Welt. Diese Methode wird in dieser Arbeit als *Posen-Kalibrierung* (*Eng. Pose Calibration*) oder auch *Posen-Methode* bezeichnet. Bei der Posen-Kalibrierung wird eine Pose bzw. ein Frame (Position und Orientierung) relativ zu zwei internen Koordinatensystem benötigt. Diese Referenzpose kann z. B. bestimmt werden, indem die zwei XR-Geräte nacheinander an dieselbe Position (mit selber Rotation) im Raum gelegt werden und dabei ihre lokalen Posen intern aufgezeichnet werden. In Abbildung 3.2 sind wieder die internen Koordinatensysteme  $\{Q1\}$  und  $\{Q2\}$  zweier XR-Geräte zu sehen. Der ebenfalls dargestellte Frame  $\{R\}$  ist hierbei die Referenzpose, die relativ zu beiden internen Frames bekannt ist. Das Gerät mit Frame  $\{Q1\}$  kennt also

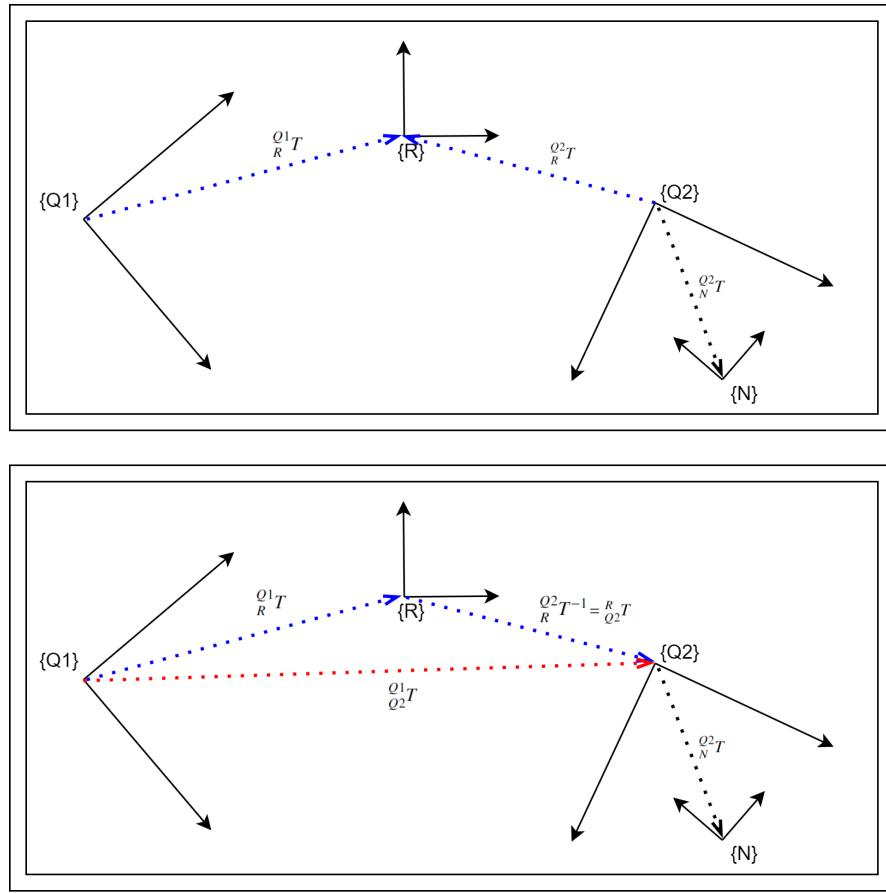


Abbildung 3.2.: Visualisierung der Gleichung 3.3

${}^Q_R T$  und das andere Gerät kennt  ${}^Q_2 T$ . Ist es möglich, diese Daten auszutauschen, kann mit folgender Gleichung somit die Transformation  ${}^Q_1 T$  und  ${}^Q_2 T$  berechnet werden:

$${}^Q_1 T = {}^Q_R T \cdot {}^R_R T^{-1} = {}^Q_R T \cdot {}^R_Q T \quad (3.3)$$

Dies ist in Abbildung 3.2 anschaulich dargestellt. Die Transformation  ${}^Q_2 T$  ist dann gegeben durch  ${}^Q_1 T^{-1}$ .

### 3.2.1. Ausrichtung des Trackingspace

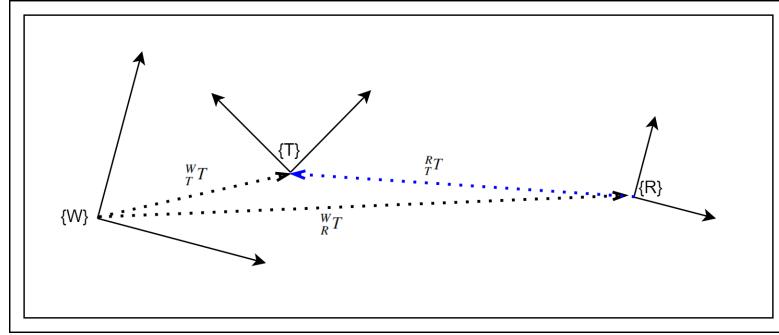
Die oben gezeigte Methode zur Abbildung von Posen im Remote-System in das lokale Koordinatensystem eignet sich gut für die Verwendung mit dynamischen Objekten, die

regelmäßig aktualisiert werden müssen. Für statische Gegenstände hingegen, wie z. B. eine in der Render-Engine modellierte Szene ist diese Methode jedoch nicht optimal, da hier die Pose jedes Unity-GameObjects in der Szene mindestens einmal an alle Teilnehmer übertragen werden müsste. Für diesen Anwendungsfall ist eine Frameausrichtung besser geeignet. Wie in 2.4 zu lesen, wurde die Frameausrichtung bereits in anderer Literatur beschrieben. In diesen Lösungen wird durch einen oder mehreren Referenzpunkten die Rotations- und Translationsdifferenz zweier Frames bestimmt und auf einen der Frames angewandt, um diese auszurichten. Im Folgenden wird beschrieben, wie man zum selben Ergebnis kommt, jedoch hergeleitet durch anschauliche Transformationsgleichungen.

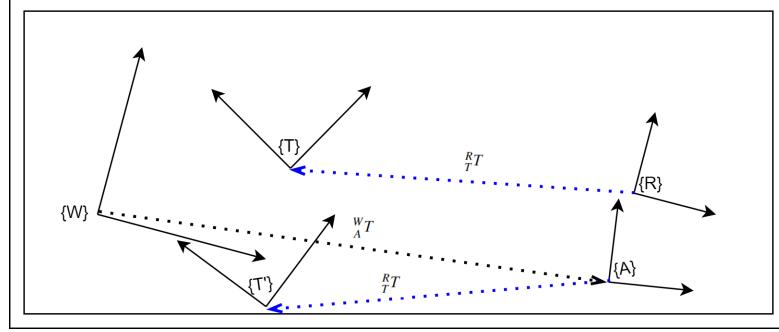
In 2.3.2 wurde bereits das Konzept des Trackingspaces beschrieben. Dieser ist ein spezielles Koordinatensystem, welches das Koordinatensystem des XR-Geräts in der Render-Engine darstellt und dessen Position und Orientierung im Worldspace der Engine bekannt ist. Um eine statische Szene in einer XR-Anwendung zu verschieben, können entweder alle Objekte in eine Richtung verschoben werden oder der Trackingspace in die entgegengesetzte Richtung. Da die Kamera-Pose des XR-Geräts relativ zum Trackingspace gleich bleibt und zusammen mit diesem verschoben wird, hat dies für den Nutzer denselben Effekt wie das Verschieben aller Objekte. Das Neupositionieren aller Objekte in der Szene ist oft nicht gewünscht, da z. B. die Lightmaps invalidiert werden würden. Daher ist die bessere Alternative das Verschieben des Trackingspace.

Zur Ausrichtung des Frames wird bei dieser Lösung neben einer Referenzpose eine sogenannte Ausrichtungspose benötigt. Diese Ausrichtungspose wird durch einen Entwickler auf Render-Engine ebene definiert. Der Trackingspace wird bei der Ausrichtung so verschoben, dass dort, wo sich die Referenzpose zur Zeit der Ausrichtung befindet, nach der Ausrichtung auch die in der Szene definierte Ausrichtungspose befindet. Der Ausrichtungspunkt kann also verwendet werden, um in der Render-Engine eine Szene relativ zu dieser Pose zu modellieren. Die Ausrichtungspose kann beispielsweise durch einen Leveldesigner im Zentrum einer Szene auf Bodenhöhe definiert werden. Ein Nutzer kann dann seinen Referenzpunkt in die Mitte des realen Raumes auf den Boden platzieren. Nach der Ausrichtung wird sich dort die Mitte der Szene befinden. Für alle Nutzer, die diese Ausrichtung mit derselben Referenzpose durchführen, werden sich die statischen Objekte konsistent an derselben Stelle befinden.

Die Kernaufgabe der Frameausrichtung ist es, die Pose im Worldspace zu finden, an die der Trackingspace bewegt werden muss, damit sich die Ausrichtungspose und die Referenzpose überlagern. Dies kann wie folgt ermittelt werden. Gegeben sind zunächst wie in Abbildung 5.3a zu sehen die drei Frames  $\{W\}$  (Worldspace),  $\{T\}$  (Trackingspace) und  $\{R\}$  (Referenzpose). Zunächst ist zu ermitteln, wo sich der Trackingspace relativ zu  $\{R\}$  befindet. Gesucht wird also  ${}^R T$ . Diese Transformation kann z. B. durch folgende Gleichung ermittelt werden. Zur Veranschaulichung siehe Abbildung 5.3a.



(a) Darstellung der gesuchten Transformation (blauer Pfeil)



(b) Verkettung der zuvor berechneten Transformation zur Bestimmung der neuen Trackingspace-Pose

Abbildung 3.3.: Visualisierung der Frameausrichtungs Berechnungen

$${}^W_T T = {}^W_R T {}^R_T T \quad (3.4)$$

$${}^W_R T^{-1} {}^W_T T = {}^R_T T \quad (3.5)$$

Die ermittelte Transformation  ${}^R_T T$  beschreibt also, wo sich der Trackingspace  $\{T\}$  aus Sicht des Frames  $\{R\}$  befindet. Da sich die Referenzpose und die Ausrichtungspose überlagern sollen, muss der Trackingspace am Ende dargestellt in Frame  $\{A\}$  (Ausrichtungspose) und Frame  $\{R\}$  jeweils dieselbe Pose besitzen. Relativ zu  $\{R\}$  ist die Pose bereits bekannt, durch Verketten mit  $\{A\}$  kann somit die neue Pose des Trackingspace  $\{T'\}$  im Worldspace wie folgt ermittelt werden. Dies ist anschaulich in Abbildung 5.3d dargestellt.

$${}^W_T T = {}^W_A T {}^R_T T \quad (3.6)$$

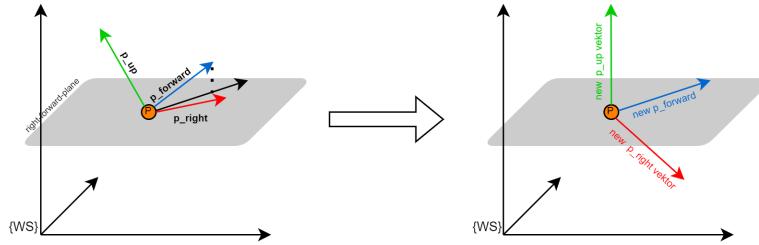


Abbildung 3.4.: Reduzierung der Rotationen auf eine Achse

${}^W_T T$  beschreibt die Pose im Worldspace der Render-Engine an die der Trackingspace transformiert werden muss. Da der Referenzpunkt sich mit dem Trackingspace mitbewegt, überlagern sich somit die Referenz- und Ausrichtungsposen am Ende.

### 3.2.2. Reduzierung des Rotationsfehlers

Wie im nächsten Unterkapitel (3.2.3) gezeigt wird, können bereits kleine Unterschiede bei der Erfassung der Referenzpose zweier Benutzer starke Positions- und Rotationsabweichungen verursachen. Um diese Fehler so gering wie möglich zu halten, wird in diesem Unterkapitel gezeigt, wie man den Rotationsfehler auf eine Achse beschränken kann.

Bei der Ausrichtung von Frames sowie die Transformationsbestimmung zwischen Trackingspaces ist nur die Rotationsdifferenz um den Basisvektor relevant, welcher die Richtung nach oben definiert. Dieser Vektor wird im Folgenden als *Up-Vektor* bezeichnet. Die Rotationen um den *Right- und Forward-Vektor* können also ignoriert werden, um somit den Rotationsfehler bei der Kalibration auf den Up-Vektor zu begrenzen. Ist eine Transformation  ${}^W_R T$  gegeben, die die Referenzpose  $\{R\}$  im Worldspace  $\{WS\}$  beschreibt, kann die Rotation wie folgt auf den Up-Vektor reduziert werden. Da Render-Engines nicht immer das gleiche Koordinatensystem verwenden, muss zunächst bekannt sein, welcher Basisvektor den Up-Vektor definiert. In Unity ist dieser definiert durch  $\hat{V}_{up} = (0, 0, 1)^T$ . Wie bereits mehrfach erwähnt, können in einer homogenen Transformation die Spalten der Rotationsmatrix als Basisvektoren eines Koordinatensystems interpretiert werden. Im Kontext der Referenzpose geben diese Vektoren also die Orientierung aus Sicht des Worldspaces an. Das Ziel ist nun, den Up-Vektor der Referenz-Pose an den des Basisvektors des Worldspace anzulegen, aber dabei die Richtung des Forward-Vektors beizubehalten, um somit alle Rotationen um die Achsen des Forward- und Right-Vektor zu entfernen. Dieser Ansatz ist in Abbildung 3.4 veranschaulicht. Da die Richtung des Up-Vektors der Referenzpose, dieselbe sein soll wie die des Worldspace, ist der neue Up-Vektor der Referenzpose mit  $\hat{P}_{new\_up} = (0, 0, 1)^T$  bereits bekannt. Bleiben also noch  $\hat{P}_{new\_forward}$  und  $\hat{P}_{new\_right}$  zu ermitteln. Als Nächstes muss ein Einheitsvektor gewählt

werden, welcher die Vorwärtsrichtung der Referenzpose darstellt. Im Folgenden wird hierfür die Y-Achse (2. Spalte in der Rotationsmatrix) gewählt, da diese auch in Unity als Vorwärtsrichtung verwendet wird. Dieser Vektor kann nun normalisiert auf die Ebene projiziert werden, die durch den Right- und Forward-Vektor des Worldspace definiert ist. Das Ergebnis dieser Projektion ist der neue Vorwärtsvektor  $\hat{P}_{\text{new\_forward}}$ . Durch das Kreuzprodukt der nun bekannten zwei Einheitsvektoren kann der letzte Vektor  $\hat{P}_{\text{new\_right}}$  berechnet werden. Zuletzt können diese drei ermittelten Vektoren in die drei Spalten der Rotationsmatrix, die Teil von  ${}^R_{\text{WST}}$  ist, als neue Basisvektoren geschrieben werden. Die resultierende Referenzpose kann dann genau wie oben beschrieben zur Frameausrichtung und Transformationsbestimmung verwendet werden. Sobald ein Nutzer die Rotationsreduktion verwendet, muss diese auch von allen anderen Benutzern ausgeführt werden.

### 3.2.3. Kalibrationsfehler

McGill et al. haben die theoretische Genauigkeit und Limitationen, die bei Kalibration durch einen Referenzpunkt bereits in [34] mathematisch formalisiert. Da der Fehler für eine Wahl der geeigneten Kalibrationsmethode wichtig ist, wird auf diese im Folgenden noch einmal eingegangen.

Der Kalibrationsfehler der Posen-Methode hängt im hohen Maße davon ab, wie genau ein Benutzer die Referenzpose in der physikalischen Welt bestimmen kann und wie präzise die XR-Geräte die Pose messen können. Beispielsweise kann eine Ausrichtung oder Transformationsbestimmung erfolgen, indem sich zunächst ein Benutzer an eine bestimmte Stelle des Raums stellt und während der Kalibration in eine vorgegebene Richtung blickt. Der nächste Benutzer stellt sich dann auf dieselbe Position und blickt in dieselbe Richtung. Theoretisch ist das genug, um eine Co-Located Erfahrung zu ermöglichen, aber in der Praxis wird es zu großen Unterschieden zwischen der virtuellen und realen Position der Teilnehmer geben. Der theoretische Fehler kann, wie in [34] beschrieben, wie folgt berechnet werden.

Angenommen, es existiert ein Spielareal mit dem Radius  $R$  und dem Mittelpunkt (Referenzpose)  $Q_i$  in der realen Welt. Ein Benutzer verwendet die Posen-Methode für die Kalibration. Die Referenzpose wurde jedoch an der Position  $P_i$  mit einer Winkelabweichung  $\theta$  erfasst. Bewegt sich der Nutzer entlang der x-Achse zum Rand (Entfernung  $R$ ) des Spielareals, sollte der Nutzer sich eigentlich bei Punkt  $Q_e$  befinden. Durch die Abweichungen bei der Kalibration befindet sich der Spieler aber an Punkt  $P_a$ . Die Position von  $P_a$  lässt sich berechnen durch

$$P_a = P_i + (R \cdot \cos(\theta), R \cdot \sin(\theta))^T \quad (3.7)$$

Es ergibt also ein Positionsfehler  $E_p$  ( $E_p = d(Q_e, P_a)$ ). Visuell ist dies in Abbildung 3.5 dargestellt.

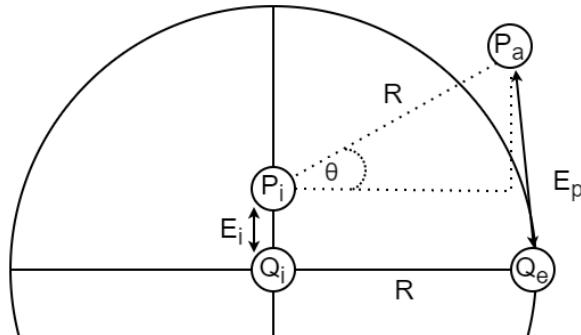


Abbildung 3.5.: Beispiel für die Durchführung einer Kalibration mittels der Posen-Methode. Durch die Positions- und Winkelabweichung gibt es am Rand des Spielfelds zwischen der Soll- und Ist-Position den Fehler  $E_p$

Angenommen, es werden verschiedene Radien von Spielarealen betrachtet, deren Kalibration im Zentrum erfolgt, erhöht sich der Fehler am Rande der Area linear mit dem Winkelfehler. Siehe folgende Abbildung:

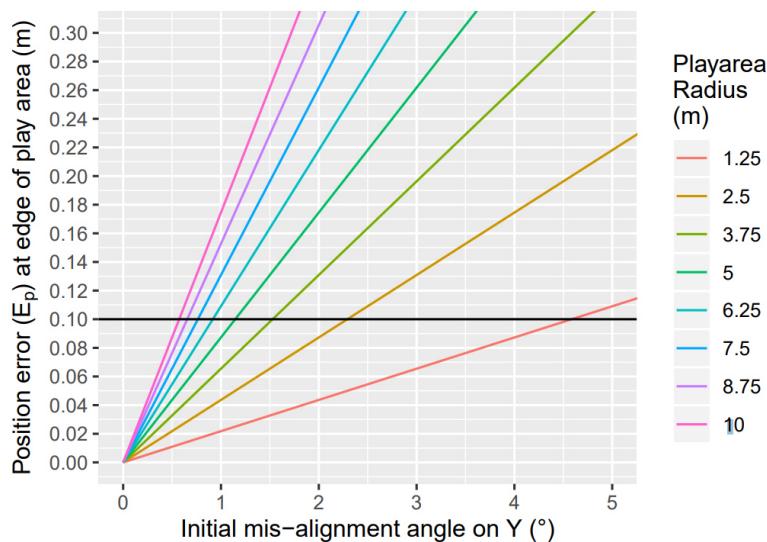


Abbildung 3.6.: Positionsfehler am Rand des Spielareals laut McGill et al. aus [34].

In dieser Abbildung ist zu erkennen, wie der Positionsfehler mit dem Abweichungswinkel und der Raumgröße zusammenhängt. Beispielsweise wird gezeigt, dass bei einem sehr

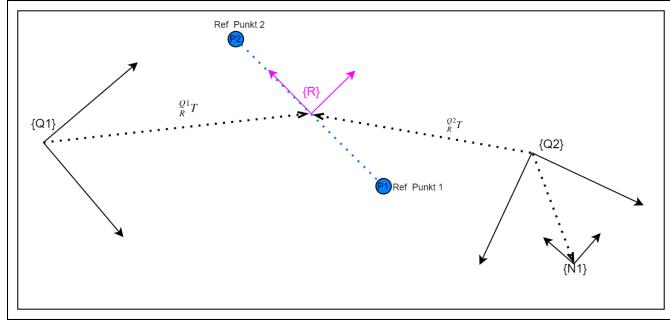


Abbildung 3.7.: Visualisierung der Erstellung einer der virtuellen Referenzpose. Durch  $P_1$  und  $P_2$  wird diese in beiden Koordinatensystemen definiert.

kleinen Raum mit einem Radius von 1,25 m und ein Abweichungswinkel von ca.  $4.5^\circ$ , an dessen Rand nur ein Positionsfehler von  $\pm 10\text{cm}$  besteht. Ob die Posen-Methode einsetzbar ist, kann für jeden Anwendungsfall unter Einbezug von Raumgröße und Fehlertoleranz also individuell eingeschätzt werden.

### 3.3. Transformationsbestimmung durch zwei Referenzpunkte

Eine weitere Methode, die Abbildungstransformation zwischen zwei Frames zu bestimmen, ist mithilfe von zwei geteilten Punkten in der realen Welt. Diese Methode wird in dieser Arbeit als *Punkte-Kalibrierung* (eng. *Points-Calibration*) oder *Punkte-Methode* bezeichnet. Die Punkte-Kalibrierung ähnelt sehr der Posen-Kalibrierung. Jedoch wird bei der Punkte-Kalibrierung der Rotationsfehler verringert, indem die Richtung zwischen zwei Referenzpunkten als Orientierung verwendet wird anstelle der Orientierung einer einzelnen Pose. Wie sich dies auf die Fehler auswirkt, wird in 3.3.2 beschrieben.

Die zwei Punkte können zum Beispiel ermittelt werden, indem die Controller auf markierte Positionen im Raum gelegt und deren Position aufgezeichnet wird. Andere Teilnehmer würden dann ihre Controller auf dieselben Markierungen platzieren, um die Positionen der Markierungen im jeweiligen internen Koordinatensystem zu erhalten. Dies ist in Abbildung 3.7 veranschaulicht. Hierbei sind  $\{Q1\}$  und  $\{Q2\}$  die internen Frames der zwei XR-Geräte und  $P1$  und  $P2$  die zwei aufgezeichneten Referenzpunkte.

Mit diesen zwei Punkten  $P1$  und  $P2$  kann eine virtuelle Referenzpose  $\{R\}$  erstellt werden. Als Position von  $\{R\}$  wird der Schwerpunkt von  $P1$  und  $P2$  verwendet. Dieser kann im Frame  $\{Q1\}$  wie folgt berechnet werden:

$$\mathcal{Q}^1 \hat{R}_{URSP} = \frac{(\mathcal{Q}^1 \hat{P}1 + \mathcal{Q}^1 \hat{P}2)}{2} \quad (3.8)$$

Für die Orientierung von  $\{R\}$  wird die Richtung des Vektors verwendet, der  $P1$  mit  $P2$  verbindet. Berechnet wird dieser durch

$${}^Q_1 \hat{R}_{\text{forward}} = \text{normalize}({}^Q_1 \hat{P}_2 - {}^Q_1 \hat{P}_1). \quad (3.9)$$

Dieser Vektor beschreibt den Vorwärts-Basisvektor von  $\{R\}$  ausgedrückt in Frame  $\{Q1\}$ . Um die Orientierung vollständig beschreiben zu können, müssen noch die zwei Basisvektoren, die die Richtung nach rechts und oben angeben, ermittelt werden. Da das Kreuzprodukt zweier Vektoren einen Vektor ergibt, der senkrecht auf der von den beiden Vektoren aufgespannten Ebene steht, kann wie folgt der Right-Vektor  ${}^Q_1 \hat{R}_{\text{right}}$  durch das Kreuzprodukt des Forward-Vektors  ${}^Q_1 \hat{R}_{\text{forward}}$  und dem Up-Vektor der Render-Engine ( $\hat{V}_{\text{up}} = (0, 0, 1)^T$  in Unity) ermittelt werden.

$${}^Q_1 \hat{R}_{\text{right}} = {}^Q_1 \hat{V}_{\text{up}} \times {}^Q_1 \hat{R}_{\text{forward}} \quad (3.10)$$

Der korrekte Up-Vektor wird berechnet, indem wiederum durch ein Kreuzprodukt der Vektor berechnet wird, der senkrecht auf der von  ${}^Q_1 \hat{R}_{\text{forward}}$  und  ${}^Q_1 \hat{R}_{\text{right}}$  aufgespannten Ebene von steht:

$${}^Q_1 \hat{R}_{\text{up}} = {}^Q_1 \hat{R}_{\text{forward}} \times {}^Q_1 \hat{R}_{\text{right}} \quad (3.11)$$

Durch die ermittelte Position und die Basisvektoren wird die virtuelle Pose am Ende durch die folgende Transformationsmatrix  ${}^Q_R T$  beschrieben:

$${}^Q_R T = \left[ \begin{array}{ccc|c} {}^Q_1 \hat{R}_{\text{right}} & {}^Q_1 \hat{R}_{\text{forward}} & {}^Q_1 \hat{R}_{\text{up}} & {}^Q_1 \hat{R}_{\text{URSP}} \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (3.12)$$

Sobald  ${}^Q_R T$  definiert ist, kann diese Pose zur Kalibration bzw. zur Transformationsbestimmung  ${}^Q_2 T$  genutzt werden, wie in zuvor bei der Posen-Kalibration (siehe 3.2) beschrieben.

### 3.3.1. Frame Ausrichtung und Rotationsreduzierung

Frameausrichtung und Rotationsreduzierung werden hier in einem Unterkapitel zusammengefasst, da sich diese kaum von der für die in Posen-Kalibrierung beschriebenen unterscheiden. Sobald die virtuelle Pose durch die zwei Referenzpunkte erstellt wurde, kann mit dieser eine Frameausrichtung und Rotationsreduzierung wie in 3.2.1 und 3.2.2

beschrieben durchgeführt werden. Zu ergänzen ist jedoch, dass eine Rotationsreduzierung nur notwendig ist, falls die Punkte  $P_1$  und  $P_2$  sich nicht auf einer Ebene befinden. Ist die Höhe der beiden Punkte dieselbe, ergibt sich automatisch lediglich eine Rotation um den Up-Vektor.

### 3.3.2. Kalibrationsfehler

In 3.2.3 wurde gezeigt, dass es schon bei kleinen Winkelabweichungen bei Erfassung der Referenzpose am Rand des Spielbereichs zu großen Abweichungen der Positionen von virtuellen und realen Objekten kommt. Durch die Verwendung von zwei Referenzpunkten anstatt einer Referenzpose können diese Winkelabweichungen reduziert werden.

Man betrachte ein kreisförmiges Spielareal mit dem Radius  $R$  und die zwei physikalische Referenzpunkte  $Q_1$  und  $Q_2$  die zur Kalibration verwendet werden sollen. Bei der Durchführung der Kalibration werden die Punkte  $P_1$  und  $P_2$  erfasst, welche von  $Q_1$  und  $Q_2$  etwas abweichen. Wenn sich diese zwei Punkte jeweils am gegenüberliegenden Rand des Spielareals befinden, ist innerhalb des Spielareals der maximal mögliche Positionsfehler gleich dem Fehler zwischen einem physikalischen Referenzpunkt und dem zugehörigen tatsächlich aufgezeichnetem Punkt[34]. Wenn es einem Nutzer also möglich ist, die Referenzpunkte in 1 cm Nähe der physikalischen Punkte aufzuzeichnen, ist der maximale Positionsfehler im Spielbereich auch maximal 1 cm.

Durch die von McGill et al. in [34] vorgestellte Gleichung, kann ausgerechnet werden, wie weit die zwei Referenzpunkte voneinander entfernt sein müssen, um in einem Spielbereich an den Rändern einen vorgegebenen Maximalfehler nicht zu überschreiten.

$$d(Q1, C) = \frac{E_p}{E_b}(R + d(C, O)) \quad (3.13)$$

$E_p$  ist hierbei die erwartete Messgenauigkeit,  $E_b$  der maximale Fehler am Rand des Spielbereichs,  $C$  der Schwerpunkt der physikalischen Referenzpunkten  $Q_1$  und  $Q_2$  und  $O$  das Zentrum des Spielbereichs. Die Funktion  $d()$  gibt die Distanz zweier Punkte. Will man also z. B. die benötigte Entfernung der Referenzpunkte für einen Spielbereich mit 5 Metern Radius, einer erwarteten Messgenauigkeit von  $\pm 0,01$  m und einer Toleranz

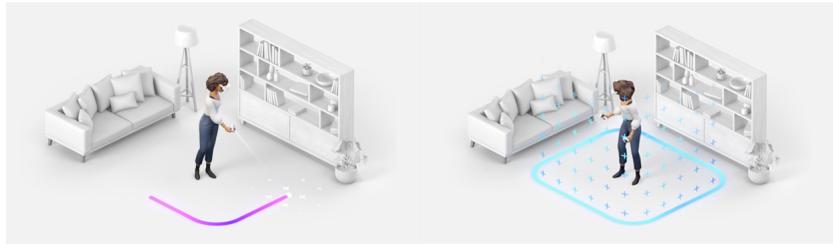


Abbildung 3.8.: Erstellung einer Guardian-Begrenzung und das Ergebnis. Die blaue Linie stellt den benutzerdefinierten Rand des Spielbereichs dar. Dieser wird intern als Punktfolge gespeichert.[16]

von  $\pm 0,1$  Metern am Rand des Spielareals, kann man dies wie folgt berechnen:

$$\begin{aligned} d(Q1, C) &= \frac{E_p}{E_b}(R + d(C, O)) \\ d(Q1, C) &= \frac{0,01}{0,1}(5 + 0) \\ d(Q1, C) &= 0,5 \end{aligned}$$

Für einen Spielbereich von 10 Metern Durchmesser, sollten die Referenzpunkte also mindestens 1 Meter ( $2 * 0.5m = d(Q1, Q2)$ ) voneinander entfernt sein, um am Spielarealrand einen maximalen Fehler von 10 cm zu erreichen. Diese Methode ist also durchaus auch für größere Räume geeignet.

### 3.4. Transformationsbestimmung durch Punktfolgenregistrierung

Die letzte in dieser Arbeit vorgestellte Methode zur Transformationsbestimmung zwischen den Koordinatensystemen zweier XR-Geräte kann verwendet werden, wenn die Geräte einen Spielbereich durch eine Punktfolge definieren. Die *Meta Quest 2* beispielsweise erlaubt den Benutzern den Spielfeldrand selbst zu definieren, indem sie diesen in VR mit einer Art Laser auf den Boden zeichnen (siehe Abbildung 3.8). Meta bezeichnet diese Abgrenzung als *Guardian*, weshalb die Methode im Folgenden als *Guardian-Kalibrierung* (eng. *Guardian-Calibration*) oder auch *Guardian-Methode* bezeichnet wird. Durch eine API kann die aktuelle Spielfeldbegrenzung als Punktfolge relativ zum Trackingspace abgefragt werden.

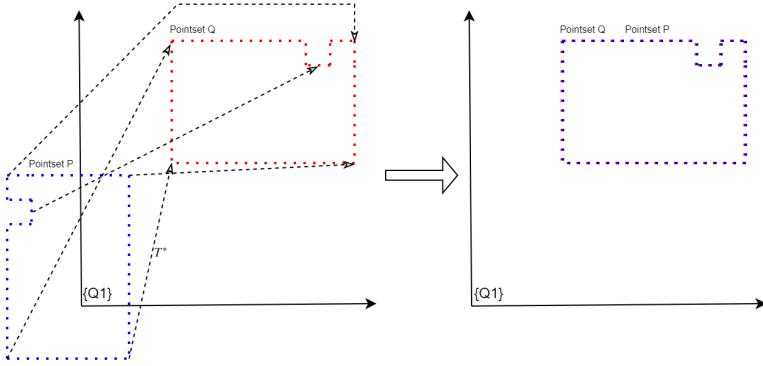


Abbildung 3.9.: Punktwolkenregistrierung im Kontext zweier Guardians

Die Guardian-Kalibration setzt voraus, dass die Punktwolken des lokalen und eines remote Geräts bekannt sind. Im Folgenden wird die lokale Wolke als  $P$  und die remote Wolke mit  $Q$  bezeichnet. In 3.4.2 wird näher auf die Methoden zur Punktwolkenregistrierung eingegangen. An dieser Stelle wird davon ausgegangen, dass ein Verfahren zur Bestimmung einer Transformation  $T^*$  existiert, sodass die Punkte der Wolke  $P$  transformiert mit  $T^*$  möglichst genau die Punkte der Wolke  $Q$  überlagern. Dies wird in Abbildung 3.9 veranschaulicht. In dieser Abbildung sind zwei Guardians dargestellt. Die Wolke  $P$  stellt einen lokalen Guardian im Frame  $\{Q1\}$  dar. Die Punktwolke  $Q$  ist dieselbe Spielfeldbegrenzung, aber aufgezeichnet in einem remote XR-Gerät. Eine Punktwolkenregistrierung liefert als Ergebnis die Transformation  $T^*$ . Multipliziert man diese Transformation mit den Punkten der Wolke  $P$  würden diese die Punkte der Wolke  $Q$  überlagern. Dies ist ebenfalls in Abbildung 3.9 dargestellt. Da man weiß, dass sich der Guardian in der physikalischen Welt für beide Geräte an derselben Stelle befindet, kann man durch  $T^*$  quasi ermitteln, welche Koordinaten Punkte im lokalen Koordinatensystem im remote System haben würden.

Da Koordinaten vom lokalen Frame  $\{Q1\}$  in ein remote Frame ( $\{Q2\}$ ) transformiert werden, kann  $T^*$  also als Abbildungs transformation, wie in Abbildung 3.10 veranschaulicht, als  $\frac{Q2}{Q1}T$  interpretiert werden. Die gesuchte Transformation  $\frac{Q1}{Q2}T$  kann also durch das Inverse ermittelt werden:

$$\frac{Q1}{Q2}T = \frac{Q2}{Q1}T^{-1} = T^{*-1} \quad (3.14)$$

Alternativ kann bei der Registrierung auch die remote und lokale Wolke vertauscht werden, um direkt  $\frac{Q1}{Q2}T$  zu erhalten.

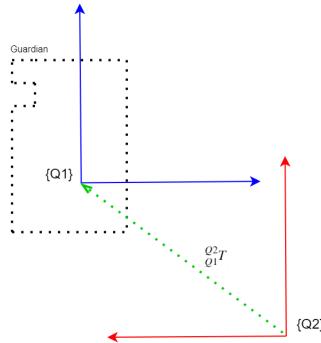


Abbildung 3.10.: Interpretation von  $T^*$  als Abbildungstransformation

### 3.4.1. Frameausrichtung

Auch bei der Guardian-Kalibration ist es möglich, den Trackingspace aller Teilnehmer so auszurichten, dass die statische modellierte Szene und dementsprechend auch die Trackingspaces sich in der realen Welt überlagert. Für den folgenden Lösungsansatz wird ein statischer Guardian benötigt, der den Raum darstellt, in dem später die Anwendung ausgeführt werden soll. Hierfür kann beispielsweise eine kleine App geschrieben werden, die nach Konfiguration des Guardians des Zielraums diesen auf dem Gerät abspeichert, welcher dann im Engine-Editor importiert werden kann. Im Editor kann anhand dieser Punktwolke eine Szene kreiert werden. Wenn der Guardian in der Engine visualisiert wird, können z. B. entlang des Spielfeldrands Wände gesetzt werden. Nach der erfolgreichen Frameausrichtung befinden diese sich dann in der realen Welt auch am Spielfeldrand. Um dies zu erreichen, muss wieder der Trackingspace  $\{T\}$  an die neue Pose  $\{T'\}$  verschoben werden. Diese neue Pose kann bestimmt werden, indem mittels Punktwolkneregistrierung die Transformation  $T^*$  berechnet wird, die den aktuellen Guardian  $P$  auf den statischen Guardian  $Q$  verschiebt. Vor der Registrierung muss hierbei darauf geachtet werden, die Punktwolke  $P$  in den Worldspace abzubilden, da die Punkte durch die (Oculus)API nur relativ zum Trackingspace abgerufen werden können.

Ist  $T^*$  bekannt, kann die neue Pose  $\{T'\}$  wie folgt berechnet werden. Dies ist in Abbildung 3.11 visualisiert.

$${}_{T'}^W T = {}_T^W T \cdot T^* \quad (3.15)$$

### 3.4.2. Punktwolkenregistrierung

In diesem Unterkapitel wird genauer darauf eingegangen, wie die oben beschriebene Registrierung umgesetzt werden kann.

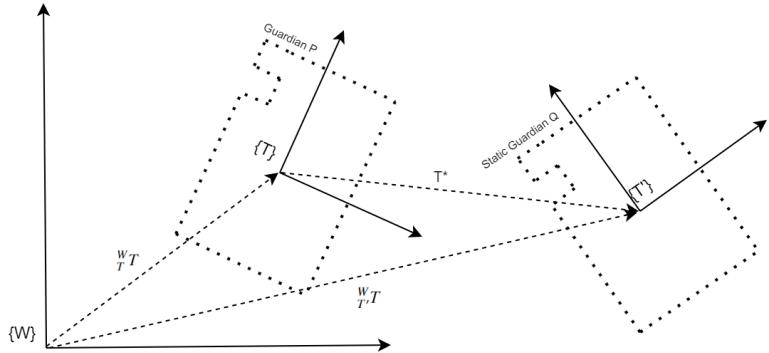


Abbildung 3.11.: Mithilfe der durch Punktwolkenregistrierung ermittelten Transformation  $T^*$  kann die neue Pose für den Trackingspace  $T'$  berechnet werden

Bei der Punktwolkenregistrierung kann zwischen einer starren (eng. rigid) und einer nicht-starren (eng. non-rigid) Registrierung unterschieden werden. Bei einer starren Registrierung wird davon ausgegangen, dass die Punkte der Wolke durch eine Transformation, die nur aus einer Translation und Rotation besteht, ausgerichtet werden können. Eine nicht-starre Registrierung erlaubt eine Registrierung von Punktwolken, die sich neben Rotation und Translation auch in Skalierung und Scherung unterscheiden können. Da die Guardian-Punktwolken zweier XR-Geräte sich nur in Rotation und Translation unterscheiden, genügt eine Lösung zur starren Registrierung.

Wie gerade erwähnt, setzt die starre Registrierung eine Umgebung voraus, in der die Transformation zwischen zwei Punktwolken in eine Translation und in eine Rotation zerlegt werden kann, sodass eine Punktwolke auf die andere Punktwolke abgebildet wird und dabei die gleiche Form und Größe behält. Mathematisch werden zwei Punktwolken  $P = \{p_1, p_2, \dots, p_n | p \in \mathbb{R}^3\}$  und  $Q = \{q_1, q_2, \dots, q_m | q \in \mathbb{R}^3\}$  betrachtet. Die Aufgabe der starren Registrierung besteht darin, die Transformation zu finden, welche  $P$  an  $Q$  angleicht. Der mathematische Prozess kann verkürzt wie folgt dargestellt werden:

Algorithmus : $R, T = rReg(P, Q)$ ,

Ergebnis : $P' = \{p'_i\} = \{R \cdot p_i + T, i = 1, 2, \dots, m\}$ ,

wobei  $rReg$  eine Funktion darstellt, welche die optimale Rotation  $R$  und Translation  $T$  einer starren Registrierung auf der Basis einer bestimmten Beziehung zwischen zwei Punktwolken bestimmt. Wie diese Funktion praktisch umgesetzt werden kann, ist der Schwerpunkt dieses Unterkapitels.

In den letzten Jahrzehnten wurden immer mehr und vielfältigere Methoden zu Registrierung von Punktwolken vorgestellt. Neben dem klassischen ICP-Algorithmus[49] gibt es zum Beispiel Feature-Basierende Methoden, Methoden, die Deep Learning verwendend und Methoden, die mit Wahrscheinlichkeiten arbeiten[13]. Punktwolkenregistrierung wird oft in der Computervision und Robotik verwendet. Hierbei müssen oft zur Kartierung der Umgebung Millionen von Punkten in Echtzeit registriert werden, weshalb sehr effiziente Algorithmen benötigt werden. Im Kontext dieser Arbeit bestehen die Punktwolken nur aus sehr wenigen Punkten (ca. 300-1000 pro Wolke). Für die Registrierung von Guardians ist also eine angepasste Version des klassischen ICP-Algorithmus ausreichend. Die generelle Funktionsweise eines klassischen ICP-Algorithmus wird im Folgenden vorgestellt.

### Iterative-Closest-Point Algorithmus

Der ICP-Algorithmus ist ein iterativer Algorithmus, der in unter idealen Umständen Genauigkeit, Konvergenzgeschwindigkeit und Stabilität gewährleisten kann. Der Algorithmus kann dabei als Erwartungs-Maximierungs-Problem (eng. expectation-maximization, Abk. EM)[50, 51] betrachtet werden, wobei er eine neue Transformation auf der Grundlage von Korrespondenzen berechnet, aktualisiert und dann auf diese dann auf eine Punktwolke anwendet, bis eine Fehlermetrik konvergiert. Hierbei gibt es jedoch keine Garantie, dass der ICP ein globales Optimum erreicht. Damit der Algorithmus nicht in ein lokales Minimum fällt, wird er meist mit einem anderen Algorithmen kombiniert, die dem ICP-Algorithmus bereits eine grobe Transformation (*Initial Guess*) als Input bereitstellt. Dies geschieht auch in der Implementation dieser Arbeit (siehe 5.3.1). Im Weiterem wird die statische Punktwolke, die bei der Registrierung nicht transformiert wird, mit  $Q$  und die Wolke, die für die Registrierung verschoben wird, mit  $P$  bezeichnet. Der ICP-Algorithmus lässt sich grob in vier Schritte unterteilen[52, 53]:

1. Punkteauswahl (eng. Point Selection)
2. Punktezuordnung (eng. Point Mapping)
3. Punkteverwerfung (eng. Point Rejection)
4. Fehler-Metrik Minimierung (eng. Error-Metric Minimization)

Auf diese Schritte wird im Folgenden näher eingegangen. Der grundlegende Ablauf des ICP-Algorithmus ist in Abbildung 3.12 veranschaulicht.

#### 1. Punkteauswahl

In diesem Schritt wird eine Teilmenge der Punktwolke ausgewählt, die die aktuelle Wolke repräsentieren soll. Bei Anwendungsgebieten wie der Kartierung von Umgebungen sind die unverarbeiteten Punktwolken meist sehr groß und besitzt daher eine hohe Rechenkomplexität, das zu einer untragbaren Rechenzeit führt. Deshalb wird die Punkteauswahl als Vorverarbeitungsschritt zur Reduzierung der Anzahl der Punkte verwendet. Hierfür

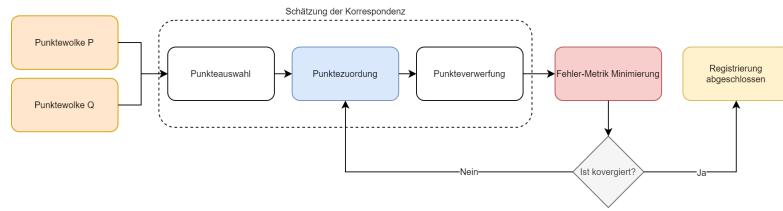


Abbildung 3.12.: Caption

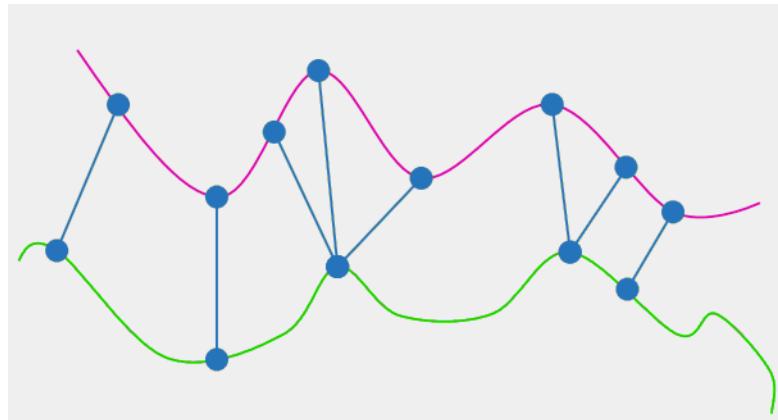


Abbildung 3.13.: Das Ergebnis der Punktezuordnung. Für jeden Punkt aus einer Punktwolke (Lila) wurde durch die euklidische Distanz der nächstgelegene Punkt der zweiten Wolke (Grün) zugewiesen und somit eine Korrespondenz erstellt.

wurden viele Downsampling Methoden wie z. B. *Random*, *Distance Limit* und *Uniform DownSampling* entwickelt [54, 55]. Da bei der Registrierung von XR-Guardians die Punktwolken vergleichsweise sehr klein sind, ist dieser Optimierungsschritt nicht notwendig.

## 2. Punktezuordnung

In diesem Schritt werden die Zusammengehörigkeiten von den Punkten aus den zwei Punktwolken ermittelt. Anders ausgedrückt wird ein Punkt aus der zu verschiebenden Wolke betrachtet und in der anderen Wolke ein Punkt gesucht, welcher diesen abbildet. Es wird also für jeden Punkt in Wolke  $P$  eine Korrespondenz zu einem Punkt aus Wolke  $Q$  erstellt. Die korrekte Translation und Rotation zwischen den beiden Punktwolken kann berechnet werden, wenn die entsprechenden Korrespondenzen bekannt sind. Korrespondenzen bzw. Punktpaare werden in der Regel erzeugt, indem für jeden Punkt in  $P$  der nächstgelegene Punkt (euklidischer Abstand) in  $Q$  ermittelt wird. Dies ist in Abbildung 3.13 dargestellt.

Allerdings ist die Suchoperation zum Bestimmen des nächstgelegenen Nachbarn eine *greedy* Annäherung und hat erhebliche Rechenkomplexität mit  $O(N * M)$ , wobei  $N$  und  $M$  die Anzahl der Punkte in der jeweiligen Punktewolke bezeichnet. Um diese aufwendige durchlaufen aller Punkte von  $Q$  zu vermeiden, wurden einige Datenstrukturen vorgestellt, mit denen die Laufzeit deutlich reduziert wird. Beispiele solcher Datenstrukturen sind *kd-trees*[56, 57], *octtrees*[58] und *multi-z-buffer*[59]. Kd-Trees werden auch im Toolkit dieser Arbeit verwendet (siehe 5.3.1).

Neben der Punktezuordnung durch die Distanz zum nächstgelegenen Nachbarn gibt es noch weitere Strategien. Zum Beispiel beschrieben Yong et al. in [60] die sogenannte *point-to-projection* Technik und Pulli [61] die *point-to-plane* Methode. Für das Toolkit wurde die Nächstgelegener-Nachbar-Methode gewählt.

### 3. Punkteverwerfung

Das Ziel dieses Schrittes ist invalide Korrespondenzen, die im vorherigen Schritt ermittelt wurden, zu verwerfen, da fehlerhafte Punktpaare die Transformationsberechnung zwischen Punktewolken beeinträchtigen können [62]. Außerdem kann die Verwerfung von invaliden Korrespondenzen die Robustheit gegenüber Rauschen und Ausreißern erhöhen [63].

Die meistverbreiteten Methoden, um fehlerhafte Punktpaare abzulehnen, basieren auf der euklidischen Distanz. Es kann beispielsweise eine fixe maximale Entfernung definiert werden und alle Punktpaare, deren Entfernung zueinander größer ist, werden verworfen[62]. Dies funktioniert gut, um Ausreißer außerhalb des Überlappungsbereichs zu filtern. Auf der Grundlage der eben beschriebenen fixen Entfernung zwischen einem Paar von Punkten können Korrespondenzen auch mit den schlechtesten  $n\%$  der Paare[61] oder mit einem Vielfachen der Standardabweichung der Abstände [64] verworfen werden. Da Verwerfung eines festen Prozenteils oder die Nutzung eines festen Entfernungsenschwellwerts häufig zu unflexibel ist, wurden einige andere Strategien vorgestellt. *Median distance*[65], *dynamic threshold*[61], *RANSAC*[66], *normal compatibility* und *duplicate matching*[67] sind einige Beispiele hierfür.

Wie später in der Arbeit gezeigt wird, ist der Schritt der Punkteverwerfung für den Anwendungsfall dieser Arbeit nicht notwendig, da 1) die Guardian-Punktewolke per Hand gezeichnet wird und somit keine Ausreißer existieren und 2) die initiale Schätzung der Transformation bereits genau genug ist und diesen Schritt überflüssig machen.

### 4. Fehlermetrik Minimierung

Ziel dieses Schrittes ist es, eine geeignete Kostenfunktion und Metrik einzusetzen, um festzustellen, ob ein gutes Transformationsergebnis erzielt wurde und dann durch eine weitere Iteration diese Fehlermetrik zu minimieren. Die Herangehensweisen [68, 69] *point-to-point*, *point-to-plane* und *point-to-projection* sind für die Registrierung

von starren Punktwolken weit verbreitet. Die *point-to-projection* Methode ist für den Anwendungsfall dieser Arbeit ungeeignet, da sich alle Punkte des Guardians bereits auf einer Ebene befinden und wird daher nicht näher behandelt.

Arun et al. [70] und Besl et al. [49] stellten eine **point-to-point** Strategie vor, in der jeder Punkt der Punktwolke  $P$  mit einem Punkt statischen Punktwolke  $Q$  gepaart wird, der ihm am nächsten ist. Wie in Gleichung 3.16 dargestellt, ist die Fehlermetrik die Summe der quadrierten Differenzen zwischen den Punktpaaren. Ziel ist es, für die gesamten Punktwolkendaten die optimale Rotation  $R$  und Translation  $T$  zu finden, die einen minimalen Wert der Gleichung 3.17 erreicht[13].

$$L(P, Q) = \sum_{i=1}^n \|q_i - p_i\|^2, \quad (3.16)$$

$$\hat{L}(R, T) = \arg \min_{R, T} \sum_{i=1}^n \|q_i - (Rp_i + T)\|^2 \quad (3.17)$$

wobei  $\|\cdot\|^2$  die euklidische Distanz zwischen  $q_i$  und  $p_i$  für das  $i$ -te Punktpaar der  $N$  Korrespondenzen von der Punktwolke  $Q$  zur Wolke  $P$  angibt.  $R$  und  $T$  repräsentieren hierbei der Rotations- und Translationsteil der Transformation. Die Punktwolke  $P$  wird mit dieser neuen Transformation aktualisiert. Falls  $\hat{L}$  nicht den geforderten Schwellwert entspricht und noch konvergiert, wird mit der aktuellen Punktwolke die nächste Berechnung inklusive Punktezuordnung (Schritt 2) und Punkteverwerfung (Schritt 3) durchgeführt. Die Fehlermetrik des ICP-Algorithmus konvergiert und endet, sobald die Differenz des aktuellen  $\hat{L}$  und des nächsten  $\hat{L}_{\text{next}}$  einen Schwellwert  $\delta$  unterschreitet:  $|\hat{L} - \hat{L}_{\text{next}}| < \delta$

Bergevin et al. [71] und Chen et al. [72] schlugen eine *point-to-plane* ICP-Variante vor. Dieser Algorithmus nimmt an, dass die Punktwolken lokal linear sind, sodass die lokale Nachbarschaft eines Punktes koplanar ist. Diese lokale Fläche kann durch ihren Normalenvektor beschrieben werden. Anstatt direkt den euklidischen Abstand zwischen den Punktpaaren zu minimieren, wird die Skalarprojektion dieses Abstands auf die durch den Normalenvektor definierte ebene Fläche minimiert. Dies ist in Abbildung 3.14 veranschaulicht. Die Fehlermetrik des *point-to-plane*-ICP ist also wie folgt definiert:

$$\hat{L}(R, T) = \arg \min_{R, T} \sum_{i=1}^n ((Rp_i + T - q_i) \cdot n_{q_i})^2, \quad (3.18)$$

wobei  $p_i$  von  $R$  und  $T$  transformiert wird und  $n_{q_i}$  der Normalenvektor des Punktes  $q_i$  ist. Alle anderen Variablen sind dieselben wie in Gleichung 3.17. Verglichen mit der

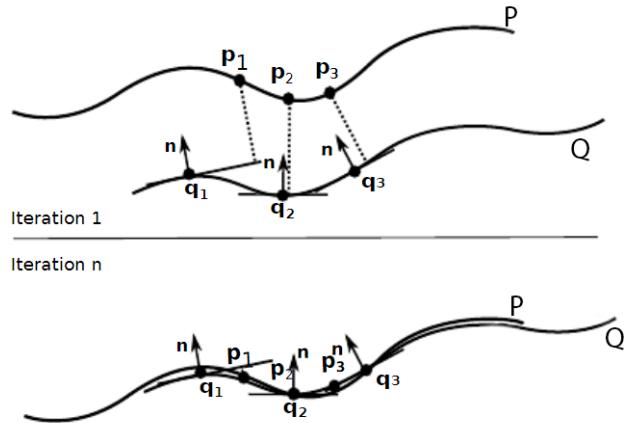


Abbildung 3.14.: Bei der Point-To-Plane ICP-Variante wird der Abstand zwischen Punkten aus  $P$  und deren Skalarprojektion auf den Normalenvektor des Korrespondenz-Punkts aus  $Q$  minimiert.[73]

*point-to-point*-basierte Methode ist die *point-to-plane*-basierte Methode robuster gegen Ausreißer und hat eine höhere Konvergenzgeschwindigkeit[13].

Um die optimale Rotation und Translation zu erhalten, kann das lineare Kleinsten-Quadrat-Problem so formuliert werden, dass es mit einer linearen Optimierung gelöst werden kann. Zum Beispiel mit einer Singulärwertzerlegung (eng. Singular Value Decomposition - SVD) [70] durch Berechnung von orthonormalen Matrizen [74], mithilfe von Einheitsquaterniionen[75] oder auch durch ein nicht lineares Lösungsverfahren, wie das Levenberg-Marquardt-Verfahren[76].

In 5.3.1 wird beschrieben, wie die ICP-Algorithmen des Plugins umgesetzt wurden.

# 4. Anforderungen

In diesem Kapitel werden die Anforderungen an das Toolkit bzw. an das Plugin beschrieben. Dieses Toolkit entsteht in Zusammenarbeit mit der MaibornWolff GmbH, daher wurden die Anforderungen durch den Firmenbetreuer gemeinsam mit dem Autor ermittelt. Hierbei setzen sich die ermittelten Anforderungen aus obligatorischen und optionalen Anforderungen zusammen. Optionale Anforderungen werden durch (*Opt.*) gekennzeichnet. Zunächst wird ein Überblick über das Projektziel gegeben. Danach wird im Detail auf die einzelnen Anforderungen eingegangen.

## 4.1. Projektziel Überblick

Im Rahmen dieses Projekts soll eine Software entwickelt werden, die es erlaubt, Co-Location VR-Anwendungen für Inside-Out VR-Headsets zu entwickeln und das Co-Location-Problem zu lösen.

Die Software soll in Form eines Toolkits für die Render-Engine Unity und perspektivisch auch für die Unreal Engine entwickelt werden. Sie soll die Netzwerk-Kommunikation zwischen mehreren Teilnehmern einer VR-Anwendung realisieren und über diese Netzwerkverbindung die Posen der typischen VR-Eingabegeräte (Headset, Controller, etc.) aller Teilnehmer an alle anderen Teilnehmer übermitteln. Die Software soll dabei alle Koordinatensystem-Transformationen durchführen, die notwendig sind, um die virtuelle Welt mit der realen Welt zur Deckung zu bringen.

Es soll so viel Logik wie möglich in plattformunabhängigem C++ implementiert werden, sodass der Kern des Toolkits sowohl für Unity als auch für Unreal genutzt werden kann. Im Rahmen dieses Projektes soll ein Binding-Layer implementiert werden, der den Aufruf des Toolkit Funktionalität aus C# Code für Unity ermöglicht und die Funktionalität in Unity integriert. Eine Integration in Unreal ist nicht Teil dieses Projektes, alle Entwurfsentscheidungen sollen die perspektivische Integration des Toolkits in Unreal jedoch sicherstellen.

## 4.2. Kalibration

In diesem Unterkapitel werden alle Anforderungen beschrieben, die mit der Kalibration der XR-Geräte zusammenhängen.

### 4.2.1. Berechnung der Transformation zwischen lokalen und remote Koordinatensystem

Eine der Kernaufgabe des Toolkits ist die Berechnung von Transformationen zwischen den lokalen Koordinatensystemen zweier Teilnehmer. Das Toolkit soll also die Möglichkeit bieten, mithilfe aller in Kapitel 3 vorgestellten Methoden diese Transformation zu bestimmen.

### 4.2.2. Abbildung von (Remote-)Posen in das lokale Koordinatensystem

Sobald die Transformation zwischen den Anwendungsinstanzen bestimmt wurde, soll jede Instanz in der Lage sein, die empfangenen Posen (Headset, Controller, etc.) aller anderen Instanzen entsprechend der realen Welt in das eigene Koordinatensystem der Render-Engine zu transformieren bzw. abzubilden.

### 4.2.3. Rekalibrierung

Das Kalibrationsverfahren soll während der Laufzeit beliebig oft ausführbar sein, sodass bei Trackingverlust des Headsets, bei verändertem internen Koordinatensystem oder bei unsauberer Ausführung der Kalibrationsmethoden die Transformation erneut ermittelt werden kann.

### 4.2.4. Ausrichtung der Frames (*Opt.*)

Das Toolkit soll die Möglichkeit bieten, statische, nicht durch das Netzwerk synchronisierte Objekte für alle Teilnehmer konsistent an derselben physikalischen Position anzuzeigen. Hierfür soll das Toolkit, wie in Kapitel 3 beschrieben, eine Ausrichtung des Trackingspaces ermöglichen.

## 4.3. Netzwerk

In diesem Unterkapitel werden die Anforderungen an die Netzwerkschicht beschrieben.

### 4.3.1. Austausch von Kalibrationsdaten

Die Netzwerkschicht soll nach erfolgter Kalibration die Kalibrationsinformation an alle verbundenen Instanzen ausliefern. Außerdem sollen auch alle Instanzen, die sich in Zukunft verbinden, diese Daten erhalten.

### 4.3.2. Anwendungs- und Raumidentifikation

Die Netzwerkschicht soll mit einer Anwendungs-ID und einer Raum-ID konfigurierbar sein, damit mehrere Anwendungen verschiedenen Typs oder gleicher Technologie im selben Wifi-Netzwerk genutzt werden können, ohne sich gegenseitig zu stören.

### 4.3.3. Automatische Teilnehmererkennung

Damit ad-hoc Instanzen ohne vorherige Netzwerkkonfiguration möglich sind, soll die Netzwerkschicht in der Lage sein, neue Teilnehmer innerhalb des lokalen Wifi-Netzwerks zu entdecken.

### 4.3.4. Automatische Sitzungsregistration

Sobald die eigene Instanz andere (kompatible) Teilnehmer im lokalen Netzwerk entdeckt, soll er sich mithilfe einer Netzwerknachricht registrieren, sodass er Teil des Gesamtnetzwerks wird.

### 4.3.5. Sitzungsverwaltung

Die Netzwerkschicht soll über ein internes Sitzungsmanagement verfügen. Innerhalb der Sitzung sollen alle Teilnehmer verwaltet werden, die der Sitzung beitreten oder sie wieder verlassen

#### 4.3.6. Plattformunabhängigkeit

Der native Teil des Plugins soll auf Geräten mit Android und Windows lauffähig sein, um das Toolkit auch im Engine-Editor verwenden zu können oder ggf. einen dedizierten Server erstellen zu können. Die Netzwerkschicht soll außerdem so implementiert werden, dass die Kommunikation auch zwischen verschiedenen CPU-Architekturen funktioniert.

#### 4.3.7. Wiederverwendbares Framework (opt.)

Die Erstellung von ad-hoc Netzwerken und der Austausch von Posen-Daten ist ein immer öfter auftretender Anwendungsfall für MaibornWolff. Deshalb soll die Netzwerkschicht so implementiert werden, dass möglichst viel Quellcode für andere Projekte wiederverwendet werden kann. Es soll also ein vom Umfang her angemessenes Networking-Framework erstellt werden.

### 4.4. Render-Engine Integration (Unity)

Dieses Unterkapitel beschreibt die Anforderungen an das Toolkit auf Render-Engine Ebene.

#### 4.4.1. Unity Package

Für die Anwendungsentwickler soll ein oder mehrere Unity-Packages bereitgestellt werden, das die Funktionalität in ein Unity Projekt integriert. Der native Teil des Plugins soll hierbei für Windows und Android als kompilierte geteilte Bibliothek inkludiert sein.

#### 4.4.2. C# API

Es soll eine einfach zu benutzende C# API zur Verfügung gestellt werden, um auf Engine-Ebene auf die Co-Location-Funktionalität zugreifen zu können.

#### 4.4.3. GameObject Synchronisierung

Es soll eine Funktionalität implementiert werden, um die Transforms von Unity Game-Objects mit den empfangenen Posen von anderen Teilnehmern zu synchronisieren. Zum Beispiel, wenn sich der Kopf eines Avatars bewegen soll, wie der Kopf eines anderen Teilnehmers, soll dem GameObject welches den Kopf darstellt, eine Unity-Komponente (Monobehaviour) hinzugefügt werden können, die das Objekt automatisch entsprechend der über das Netzwerk empfangenen Posen synchronisiert.

#### 4.4.4. Netzwerk Events

Es sollen Events implementiert werden, die es Unity Entwicklern ermöglichen, auf den Beitritt und Austritt von Teilnehmern der Session zu reagieren. Dies kann z. B. genutzt werden, um beim Eintritt einen Avatar zu instanziieren und diesen beim Austritt wieder zu entfernen.

# 5. Toolkit Architektur und Umsetzung

In diesem Kapitel wird die Architektur des Toolkits und dessen Komponenten beschrieben. Hierbei wird auf die wichtigsten Designentscheidungen und Algorithmen eingegangen. Zuerst wird ein Gesamtüberblick gegeben, der zeigt, wie alle Komponenten zusammenarbeiten. Im Anschluss wird im Detail auf die einzelnen Komponenten eingegangen, die den nativen Teil des Plugins darstellen. Zuletzt wird noch der Entwurf der Render-Engine-Ebene beschrieben.

## 5.1. Architektur Überblick

Das Plugin hat im Wesentlichen zwei Hauptaufgaben: 1) Die Berechnung von Transformationen zwischen Koordinatensystemen verschiedener VR-Geräte und 2) Die Synchronisation von VR-Nodes zwischen Teilnehmern im lokalen Netzwerk. Da diese Aufgaben unabhängig voneinander gelöst werden können und Entwickler die Möglichkeit haben sollen, die Kalibrationslogik ohne die Netzwerklogik verwenden zu können und vice versa, ist der native Teil wie folgt in mehrere Bibliotheken aufgeteilt: Das native Toolkit besteht aus insgesamt vier Bibliotheken, die im Laufe dieses Kapitels noch detailliert beschrieben werden.

In Abbildung 5.1 ist die Gesamtarchitektur auf höchster Ebene zu sehen. Die Pfeile stellen hierbei Abhängigkeiten dar. Ganz unten in der Abbildung ist die *Shared*-Bibliothek dargestellt. In ihr befinden sich Klassen, die in allen anderen Bibliotheken gebraucht werden, wie z. B. ein Timer, um die Geschwindigkeit von Funktionen zu erfassen. Über dieser Bibliothek sind in der Abbildung die zwei Bibliotheken *Networking* und *Calibration* zu sehen. In der *Networking*-Bibliothek befindet sich sämtliche Logik bzw. ein Framework, das zum Erstellen einer ad-hoc Server-Client Sitzungen und zum Austausch von Nachrichten im lokalen Netzwerk nötig ist. Die *Calibration*-Bibliothek beinhaltet die Logik, die zur Berechnung der Transformationen zwischen verschiedenen Koordinatensystemen von Geräten nötig ist. Zwischen diesen beiden Bibliotheken herrschen keinerlei Abhängigkeiten. Sie können daher unabhängig voneinander auch in anderen Projekten eingesetzt werden. Die letzte Bibliothek im nativen Teil des Toolkits ist der Kern (eng. Core). Diese Bibliothek beinhaltet eine Fassade[77], die das Zusammenspiel

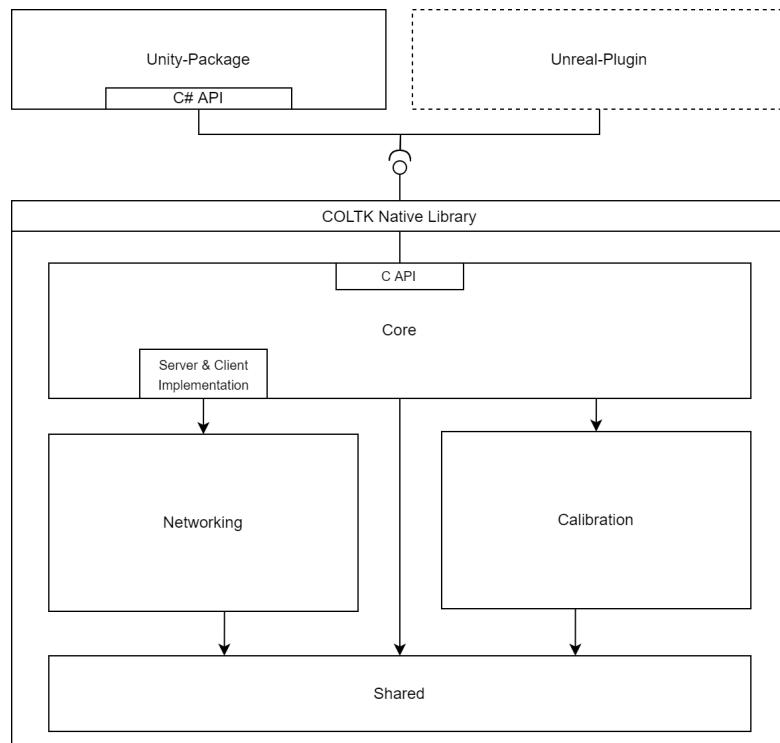


Abbildung 5.1.: Architektur des Co-Location Toolkits (COLTK)

aller anderen Bibliotheken und Klassen steuert und eine einfach zu verwendete API für Entwickler bereitstellt. Außerdem befinden sich im Kern die konkreten Server-Client-Implementierungen des im *Networking*-Bibliothek bereitgestellten Frameworks. In der Abbildung links oben ist das *Unity-Package* zu sehen. Dieses kann über eine C#-API, welche die C-API des native Plugins widerspiegelt, auf alle durch den Kern bereitgestellten Funktionen zugreifen. Im Folgenden wird im Detail auf die einzelnen Bibliotheken eingegangen.

## 5.2. Die *Shared*-Bibliothek

In dieser sehr kleinen Bibliothek befinden sich Klassen, die in allen anderen Bibliotheken benötigt werden. Genauer gesagt besteht die Bibliothek nur aus den folgenden zwei Klassen, die nur für Debug-Zwecke und zur Evaluation verwendet werden.

Die erste Klasse ist die **Timer**-Klasse. Diese bietet die Möglichkeit, die Geschwindigkeit von Algorithmen zu messen und wurde vor allem zur Beurteilung der ICP-Algorithmen

(siehe 6.1) verwendet. Die verwendete `std::clock` und die Zeiteinheit kann durch Template Argumente festgelegt werden. Standardmäßig werden Millisekunden und die `std::chrono::steady_clock` verwendet. Die Laufzeit eines Algorithmus kann dann z. B. wie folgt gemessen werden.

```
1 Timer t;
2 Algorithm();
3 float elapsed_time = t.Elapsed();
```

Die zweite Klasse in dieser Bibliothek ist die `Logger`-Klasse. Diese Klasse ermöglicht die Ausgabe von Text in der Konsole und auch im Render-Engine-Editor. Um die Debug-Nachrichten in der Render-Engine auszugeben, kann dem Logger eine *Callback*-Funktion übergeben werden, welche dann die Debug-Funktion auf Engine-Level mit den übergebenen Argumenten aufruft. Der `Logger` wurde als *Singleton*[78] implementiert und kann daher im Code der anderen Bibliotheken wie folgt verwendet werden:

```
1 Shared::Logger::sInstance().Log(ELogType::iWarning, "Debug Text")
2 //oder
3 DEBUG_LOG(ELogType::iWarning, "Debug Text") //makro
```

Der `Logger` akzeptiert hierbei einen Log-Level (Log, Warning und Error), mit dem angegeben werden kann, wie der Text in der Engine dargestellt werden soll. In Zeile 3 ist außerdem gezeigt, wie der direkte Aufruf durch ein Makro ersetzt werden kann. Dies bietet den Vorteil, dass durch ein Präprozessor-Flag alle Debug-Statements entfernt werden und somit in der Release-Version kein Overhead mehr vorhanden ist.

### 5.3. Die *Calibration*-Bibliothek

In der *Calibration*-Bibliothek befindet sich jegliche Logik, die nötig ist, um Transformationen zwischen Koordinatensystem zu berechnen und auch um Frameausrichtungen durchzuführen. Das Herz der Bibliothek ist die `Calibrator`-Klasse. Beim Entwurf dieser Klasse wurde sehr darauf geachtet, dass die API möglichst einfach nutzbar ist und dabei keinerlei mathematisches Hintergrundwissen notwendig ist. In 5.2 ist die Klasse als Klassendiagramm dargestellt. Dies ist eine gekürzte Repräsentation, bei der nicht relevante Variablen und Funktionen wie z. B. *Getter*-Methoden weggelassen wurden und größtenteils nur die Public-Funktionen (API) gezeigt werden, die zur Verwendung der Klasse nötig sind. Im Laufe dieser Arbeit werden die UML-Diagramme auch weiterhin meist nicht vollständig gezeigt. Wie in der Abbildung zu sehen, wurden die Public-Funktionen in zwei Kategorien unterteilt: *Calibration* und *Alignment*. Im *Calibration*-Bereich befinden sich alle Funktionen, die nötig sind, um die Transformation zwischen den Koordinatensystemen

aller Teilnehmern berechnen zu können. Die Transformation zwischen zwei Teilnehmern kann durch diese API mit nur zwei Funktionsaufrufen berechnet werden. Zuerst muss eine der überladenen `SetCalibrationData()`-Methoden aufgerufen werden, um die lokalen Informationen in der Klasseninstanz zu setzen. Jede dieser überladenen Methoden stellt eine der drei in Kapitel 3 beschriebenen Kalibrationsmethoden dar. Danach müssen mit `AddRemoteCalibrationData()` die Daten eines externen Teilnehmers gesetzt werden. Hierbei muss eine ID mit übergeben werden, da der *Calibrator* die berechneten Transformationen in einem Container speichert und diese mit der ID wieder abrufbar macht. Beide Funktionsaufrufe haben gewollte Seiteneffekte. Beim Aufruf einer der `SetCalibrationData()`-Funktionen werden alle bereits berechneten und im Container gespeicherten Transformationen zwischen lokalem und Remote-Koordinatensystemen neu berechnet. Außerdem wird, falls die überladende Funktion für Punkte-Kalibration aufgerufen wird, die zwei Punkte direkt, wie in 3.3 beschrieben, in eine virtuelle Pose umgewandelt. Wie in der UML-Abbildung zu sehen, gibt es daher für die lokalen Daten nur die zwei Felder `m_localPose` und `m_localGuardian`, um lokale Kalibrationsdaten zu speichern. Beim Aufruf von `AddRemoteCalibrationData()` wird nur die Transformation für diesen einen remote Teilnehmer berechnet und in den jeweiligen Containern gespeichert. Wie eben angedeutet, besitzt der *Calibrator* zwei Map-Container, in die er für jedes remote System die Kalibrationsdaten (Pose, Punkte und Guardian) und die berechnete Transformation speichert. Das Speichern der *CalibrationData* zusätzlich zur Transformation, ist notwendig, um die Transformationen zwischen allen Teilnehmern neu berechnen zu können, falls sich Referenzpunkte oder der Guardian im eigenen System ändern sollten.

Sobald die lokalen Daten und die eines remote Geräts gesetzt wurden, können mit der `RemoteToLocalSpaceInplace()`-Methode alle Positionen und Rotation unter Angabe der ID vom remote System in das lokale System abgebildet werden. Im Quellcode könnte das als Pseudocode wie folgt aussehen.

```

1 RemoteUser remote; //data of a remote user (id, calibration data, pose)
2 Calibrator c(m_settings);
3 //calculate transform between local user and the user
4 c.SetCalibrationData(calib_pose);
5 c.AddRemoteCalibrationData(remote.id, remote.data);
6 //use transform to map remote position and rotation to local frame
7 c.RemoteToLocalSpaceInplace(&remote.pos, &remote.rot, remote.id); //inplace

```

Die tatsächlichen Berechnungen werden in der geschützten Methode `CalculateTransformation()` durchgeführt. Welche Berechnungen in dieser Methode durchgeführt werden, hängt von dem im Konstruktor des *Calibrator* übergebenen Einstellungen und von den remote Daten ab. In den Einstellungen wird festgelegt, welche Kalibrationsmethode der lokale Nutzer verwendet, welche Engine benutzt wird und ob die Rotation, wie in 3.2.2 beschrieben, auf den Up-Vektor beschränkt werden soll. In

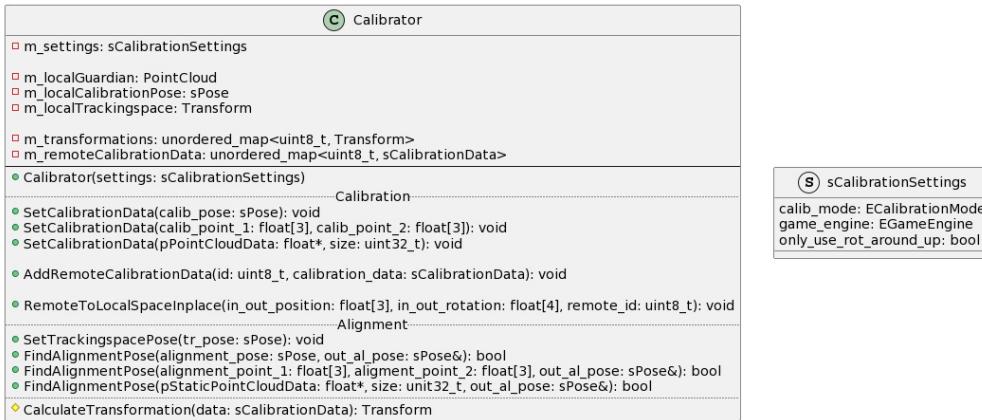


Abbildung 5.2.: Klassendiagramm der Calibrator-Klasse. Gekürzt auf Public-API und relevante Felder

der remote `CalibrationData` befinden neben den Daten die zur Kalibration nötig sind (Referenzpose etc. als Byte-Buffer) auch Information, welche Kalibrationsmethode ein remote Nutzer verwendet hat. Nur wenn die lokale Methode und die remote Methode kompatibel sind, wird eine Transformation zwischen den beiden Geräten berechnet und gespeichert. Pose- und Punkt-Kalibration sind hierbei miteinander kompatibel, sodass z. B. ein XR-Headset, das eine Punkte-Kalibration durchführt, zusammen mit einem Mobilgerät verwendet werden kann, welches einen visuellen Marker verwendet. Kurz gesagt wird in `CalculateTransformation()` die Transformation wie in 3.2 beschrieben berechnet, falls der Remote-User die Pose- oder Punkt-Kalibration verwendet hat und wie in 3.4 beschrieben, falls die Guardian-Kalibration durchgeführt wurde.

Der andere im Klassendiagramm 5.2 abgegrenzte Bereich wird als *Alignment* bezeichnet. Wie der Name vermuten lässt, sind hier alle Methoden gruppiert, welche zur Ausrichtung des Trackingspaces notwendig sind. Auch hier wurde wieder darauf geachtet, die API so simpel wie möglich zu gestalten. Die zur Ausrichtung benötigte Pose des Trackingspace kann deshalb wieder mit nur zwei Funktionsaufrufe in Erfahrung gebracht werden. Da der Trackingspace im Worldspace der Render-Engine verschoben wird, muss zunächst bekannt sein, wo sich der Trackingspaces relativ zum Worldspace befindet. Die Position und Rotation des Trackingspace muss daher mit einem Aufruf von `SetTrackingspacePose()` jedes Mal gesetzt werden, wenn er im Worldspace verschoben wird. Sobald der Trackingspace gesetzt wurde, kann mit einer der überladenen `FindAlignment()`-Methoden die benötigte Pose abgefragt werden. In dieser Methode wird je nach Wahl der Kalibrationsmethode die in 3.2.1, 3.3.1 oder 3.4.1 beschriebene Berechnung durchgeführt. Im Quellcode kann dies z. B. für die Punkte-Methode wie folgt aussehen.

```

1 sPose new_ts;
2 Calibrator c(m_settings);
3 c.SetTrackingspacePose(current_ts); //current_ts received from engine
4 c.FindAlignment(point1, point2, new_ts) //result will be in new_ts

```

### 5.3.1. ICP-Algorithmus

In 3.4.2 wurde beschrieben, wie ein ICP-Algorithmus theoretisch funktioniert. In diesem Unterkapitel wird gezeigt, wie der ICP-Algorithmus für das Toolkit konkret umgesetzt wurde und im **Calibrator** zum Einsatz kommt. Da der Anwendungsfall, welcher in dieser Arbeit behandelt wird, sehr speziell ist und das Plugin möglichst wenig Overhead haben sollte, wurde ein eigener ICP-Algorithmus entwickelt, anstatt einen aus bekannten Bibliotheken wie der *Point Cloud Library*[79] zu verwenden. Genauer gesagt wurden sogar vier unterschiedliche Algorithmen implementiert und verglichen. Im Folgenden wird die Funktionsweise dieser vier Algorithmen erläutert.

#### IcpBf

Der erste ICP-Algorithmus trägt die Bezeichnung **IcpBf**. Das „Bf“ steht hierbei für Brute-Force. Die MaibornWolff GmbH arbeitete zur Zeit der Erstellung dieser Arbeit bereits an einem sehr ähnlichen Problem. Die Firma entwickelte auf Basis der Unreal-Engine bereits einen ICP-Algorithmus, mit der die Transformation zwischen zwei Guardians ermittelt werden konnte. Da diese Arbeit in Zusammenarbeit mit der Maibornwolff GmbH entstand, wurde der Algorithmus als Ausgangspunkt zur Verfügung gestellt. Dieser Algorithmus unterscheidet sich in einigen Punkten stark von den herkömmlichen Algorithmen, die in 3.3 beschrieben wurden. Die Fehlermetrik wird beim *IcpBf* nicht durch Berechnungen minimiert, sondern durch Ausprobieren. Es wird also auf eine Art Brute-Force Ansatz gesetzt, bei dem wie folgt vorgegangen wird.

Zunächst werden die Schwerpunkte beider Punktwolken berechnet und die Punktwolke, die bewegt werden soll ( $P$ ), so verschoben, dass ihr Schwerpunkt sich im Ursprung des Koordinatensystems befindet (siehe Abbildung 5.3a). Diese Transformation zur Zentrierung von  $P$  sei  $T^{center\_P}$ . Die Verschiebung ist notwendig, damit sich alle Punkte von  $P$  bei einer Rotation um den Schwerpunkt drehen.

Ausgehend von dieser zentrierten Punktwolke  $P'$ , deren Schwerpunkt nun im Ursprung sitzt, können nun die Iterationen des ICP's durchgeführt werden. Das Ziel des Algorithmus ist es, eine Transformation  $T^{result}$  zu ermittelt, die die Punktwolke  $P'$  auf die Punktwolke  $Q$  verschiebt. Beim Start des Algorithmus wird  $T^{result}$  mit der Translation initialisiert, die nötig ist, um den Schwerpunkt von  $P'$  auf den Schwerpunkt von  $Q$  zu verschieben (Siehe Abbildung 5.3b). Diese grobe Transformation wird dann in einer festen Anzahl

von Schritten durch den Algorithmus immer weiter verfeinert. Die Verfeinerung findet statt, indem die Rotations- und Translationskomponente von  $T^{result}$  abwechselnd um ein immer kleiner werdendes  $\Delta$  verändert wird (siehe Abbildung 5.3c). Der Wert, bei dem die Fehlermetrik (quadrierte euklidische Distanz) am geringsten ist, wird als neuer Wert in  $T^{result}$  eingetragen. Als erster Verfeinerungsschritt wird beispielsweise die Rotationskomponente in  $T^{result}$  so angepasst, dass die Punktwolke  $P'$  in  $10^\circ$  Schritten einmal im Schwerpunkt von  $Q$  um  $360^\circ$  gedreht wird. Nach jedem dieser  $10^\circ$  Schritte wird die Fehlermetrik berechnet und der Schritt bei der diese am niedrigsten war, als neue Rotation nach  $T^{result}$  geschrieben. Angenommen, der Fehler war bei  $45^\circ$  am niedrigsten, würde im nächsten Schritt der Fehler von  $15^\circ$  bis  $75^\circ$  ( $\Delta$  von  $\pm 30^\circ$ ) in  $1^\circ$  Schritten betrachtet werden. Dies geschieht abwechselnd für die Rotation und Translation (X und Y Richtung separat betrachtet) für eine feste Anzahl Schritten. Am Ende wird  $T^{result}$  noch mit der initialen Zentrierung  $T^{center\_p}$  verrechnet, um das endgültige Ergebnis  $T^*$  zu erhalten (siehe Abbildung 5.3d).

$$T^* = T^{result} T^{center\_p} \quad (5.1)$$

Kurz zusammengefasst kann man sagen, dass die Wolke  $P$  auf Wolke  $Q$  verschoben wird und dann in immer kleiner werdenden Schritten rotiert und translatiert, wird bis die Fehlermetrik (quadratische euklidische Distanz) am kleinsten ist. Dieses Vorgehen wirft folgende Probleme auf:

1. Es wird immer eine feste Anzahl von Fehlermetriken ausgerechnet, auch wenn die initiale Schätzung der Transformation bereits sehr gut ist. Auch wenn die Punktwolken bereits fast aufeinanderliegen, würden alle 560 Verfeinerungsschritte durchgeführt werden.
2. Jedes Mal, wenn eine Rotation oder Translation um  $\Delta$  angepasst wurde, muss die Fehlermetrik neu berechnet werden. Das heißt, jedes Mal muss für jeden Punkt aus  $P$  der nächstgelegende Nachbar in  $Q$  ermittelt werden. Die Rechenkomplexität für eine Fehlermetrik ist also  $O(N * M)$  ( $N$  und  $M$  ist die Anzahl der Punkte in der jeweiligen Wolke), was dazu führt, dass der Algorithmus bei größeren Punktwolken nicht mehr in echtzeitfähig ist.
3. Der Algorithmus ist sehr unflexibel, da die Größe und die Anzahl der Verfeinerungsschritte fest vorgegeben ist. Dies kann bei unterschiedlichen Render-Engines zu Problemen führen. Beispielweise entspricht eine Einheit in Unity-Engine 100 Einheiten in Unreal-Engine. Beim Ermitteln der Translation, müsste hier also für jede Render-Engine eine Fallunterscheidung gemacht werden, da ein  $\Delta$  von 0,1 in Unity angemessen sein kann, aber in Unreal viel zu klein wäre.

Um diese Probleme zu lösen, wurde dieser Algorithmus zunächst erweitert und danach neue Algorithmen entwickelt, die sich mehr an dem klassischen ICP-Verfahren

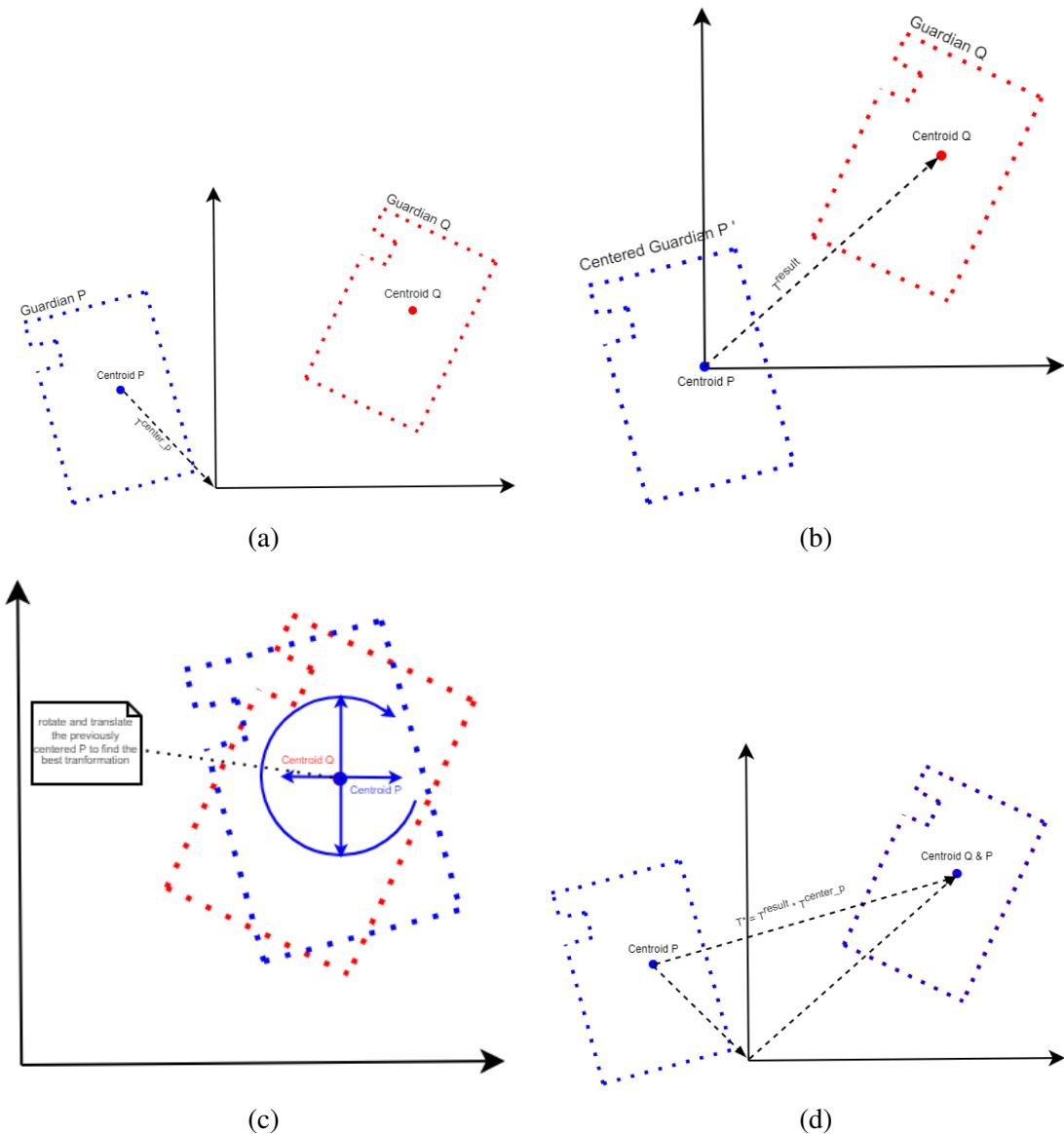


Abbildung 5.3.: Der Ablauf des IcpBf-Algorithmus bildlich dargestellt

orientieren.

### IcpBfKdTree

Der nächste im Toolkit enthaltene ICP-Algorithmus ist der **IcpBfKdTree**-Algorithmus. Dieser ist im Grunde derselbe wie der gerade beschriebene *IcpBf*, mit dem Unterschied, dass die Fehlerberechnung beschleunigt wurde. Hierfür kommt ein k-dimensionaler Baum[80] (abk. kd-Baum) zum Einsatz. In dieser Datenstruktur wird die statische Punktfolge  $Q$ , mit der einmaligen Rechenkomplexität von  $O(n \cdot (k + \log(n)))$  als k-dimensionaler (3-dimensional in diesem Fall) Baum dargestellt. Die Zugriffskomplexität, um den nächstgelegenen Nachbarn eines Punktes zu finden, wird dadurch auf  $O(n^{1-\frac{1}{k}} + 1)$  reduziert[80]. Allein diese Datenstruktur führt dazu, dass der *IcpBfKdTree*-Algorithmus, für den Anwendungsfall dieser Arbeit in der Regel ca. 4-mal schneller ein Ergebnis liefert als der vorher beschriebene *IcpBf*.

### IcpSvd

Der gerade beschriebene Algorithmus löst zwar bereits das Problem mit der hohen Rechenkomplexität zur *Nearest-Neighbor*-Bestimmung, hat aber immer noch die Probleme mit der konstanten Anzahl von Schritten und der Unflexibilität. Um diesen Problemen zu begegnen, wurde der **IcpSVD**-Algorithmus entworfen, der zur Lösung des in 3.3 beschriebenen Kleinsten-Quadrat-Problems auf Singulärwertzerlegung (SVD) setzt.

Angenommen, die Korrespondenzen zwischen den Punkten der Wolke  $P$  und  $Q$  seien bekannt. Das Minimieren der euklidischen Abstände zwischen diesen Punkten entspricht einem linearen Problem der kleinsten Quadrate, das mithilfe der SVD-Methode wie folgt, robust gelöst werden kann.

Basierend auf den Korrespondenzen kann die Kreuzkvarianzmatrix  $K$  der beiden zentrierten Punktfolgen ermittelt werden. Zerlegt man diese Matrix durch eine Singulärwertzerlegung, erhält man die Matrizen  $U$ ,  $S$  und  $V^T$ :

$$\text{SVD}(K) = USV^T \quad (5.2)$$

Die optimale Rotation  $R$  kann dann mit

$$R = UV^T \quad (5.3)$$

und die Translation  $t$  mit

$$t = c_q - R \cdot c_p \quad (5.4)$$

berechnet werden. Wobei  $c_q$  der Schwerpunkt von  $Q$  und  $c_p$  der Schwerpunkt der Wolke  $P$  ist.

Da die Korrespondenzen zwischen den Punktwolken jedoch nicht bekannt sind, müssen diese iterativ ermittelt werden. In jeder Iteration wird die Punktwolke  $P$  etwas näher an  $Q$  angepasst. Dadurch ergeben sich nach jeder Iteration neue Korrespondenzen, mit dem Ziel, dass am Ende alle Punktpaare korrekt ermittelt wurden. Eine Iteration des IcpSvd sieht in diesem Plugin wie folgt aus:

1. Zentriere Punktwolken: Beide Punktwolken werden im Koordinatensystem so verschoben, dass ihre Schwerpunkte im Ursprung liegen.
2. Ermittle Korrespondenzen: Unter Verwendung eines k-d-Baums werden die Korrespondenzen zwischen den zwei Punktwolken ermittelt. Hierbei wird nach der *Nearest-Neighbor*-Methode vorgegangen.
3. Kreuzkovarianzmatrix berechnen: Für die Singulärwertzerlegung wird die Keruzkovarianzmatrix der beiden Punktwolken benötigt. Diese wird mithilfe der im vorherigen Schritt ermittelten Korrespondenzen erstellt.
4. Rotation und Translation berechnen: Nutze die Gleichung 5.3 und 5.4, um mittels SVD die Rotation  $R$  und Translation  $t$  zu erhalten.
5. Aktualisierung der Wolke  $P$ : Die Punktwolke  $P$  wird mit den soeben berechneten  $R$  rotiert und mit  $t$  translatiert. Diese verschobene Wolke ist dann der Ausgangspunkt für die nächste Iteration.
6. Nächste Iteration: Solange neue Korrespondenzen ermittelt wurden, eine gegebene Anzahl von Iteration noch nicht überschritten wurde oder ein definiert Fehler schwellwert noch nicht erreicht wurde, gehe wieder zu Schritt 1.

Nachdem diese Schleife verlassen wurde, sind alle Korrespondenzen zwischen den beiden Punktwolken bekannt. Mit einer letzten Singulärwertzerlegung kann nun also die Transformation  $T^*$  zwischen der Ausgangswolke  $P$  und  $Q$  berechnet werden.

Der IcpSvd löst die zwei Probleme des vorherigen ICP-Alogithmus. Wenn die zwei Punktwolken sich bereits vor der Registrierung sehr ähneln, muss der IcpSvd keine feste Anzahl von Iterationen ausführen. Sollten die Wolken sich bereit ähneln, werden nur sehr wenige Iterationen notwendig sein. Außerdem ist dieser Algorithmus komplett unabhängig von den verwendeten Einheiten der Render-Engine. Des Weiteren wurde der Algorithmus so implementiert, dass er das Registierungsproblem von 3-Dimensionen auf 2-Dimensionen reduziert, da sich die Punkte der Guardian-Punktwolke immer auf einer Ebene befinden.

### IcpPoint2Point

Da bekannt ist, dass ICP-Algorithmen, die das Kleinste-Quadrat-Problem aus 3.3 durch lineare Verfahren lösen, schlechtere Ergebnisse liefern als nicht-lineare Lösungsverfahren [73], entstand der im folgenden beschriebenen IcpPoint2Point.

Jede Iteration des ICPs kann als Kleinstes-Quadrat-Problem behandelt werden. Die Funktion, die dabei minimiert werden soll, ist die quadrierte Summe der Abstände zwischen den Punkten der Punktwolken:

$$E = \sum_i [Rp_i + t - q_i]^2 \rightarrow \min \quad (5.5)$$

Um diese Funktion zu minimieren, wird, genau wie beim SVD-Algorithmus, die Rotation  $R$  und Translation  $t$  so angepasst, dass Punktwolke  $P$  die Punktwolke  $Q$  überlagert und somit die euklidische Distanz zwischen den Punktpaaren der Wolken minimiert. Die Kombination von  $R$  und  $t$  wird im Folgenden als Pose  $x$  bezeichnet, die durch den Vektor  $(x = (x, y, \theta)^T)$  beschrieben wird. Alternativ kann  $x$  auch durch die Rotationsmatrix  $R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$  und einen Translationsvektor  $t = (x, y)^T$  repräsentiert werden. Wie auch beim IcpSvd wird auch hier das Problem auf 2-Dimensionen reduziert.

Die Korrespondenzen für jede Iteration werden genau wie im IcpBfKdtree- und IcpSvd-Algorithmus mithilfe eines Kd-Baumes über die euklidische Distanz ermittelt und bilden die Menge  $C$ :

$$C = \{\{i, j\} : p_i \leftrightarrow q_j\}.$$

Um eine Lösung für das in Gleichung 5.5 gezeigte nicht-lineare Problem zu berechnen, wird das Gauss-Newton-Verfahren[81] verwendet. Die Lösung dieses ist äquivalent zur Lösung des folgenden Gleichungssystems:

$$H\Delta x = -E'(x), \quad (5.6)$$

wobei  $\Delta x$  das Inkrement der Argumente in  $x$  ( $[\Delta x, \Delta y, \Delta \theta]$ ),  $H$  die Hesse-Matrix von  $E$  und  $E'$  die Ableitung (Gradient) der zu minimierenden Funktion ist.  $H$  und  $E'$  können mithilfe einer Jacobi-Matrix (Ableitungsmatrix) berechnet werden, welche durch partielle Ableitung wie folgt definiert ist:

$$J = \begin{bmatrix} 1 & 0 & -\sin(\theta)p_i^x - \cos(\theta)p_i^y \\ 0 & 1 & \cos(\theta)p_i^x - \sin(\theta)p_i^y \end{bmatrix} \quad (5.7)$$

Details zur Herleitung dieser Matrix sind in [82, 83] zu finden.

Ein weiterer essenzieller Baustein, der zum Berechnen von  $H$  und  $E'$  nötig ist, ist die

oben erwähnte Fehlerfunktion. Diese ist wie folgt definiert:

$$e(x) = \sum_{\{i,j\} \in C} e_{i,j}(x), \quad (5.8)$$

$$e_{i,j} = R_\theta p_i + t - q_j. \quad (5.9)$$

Diese Funktion liefert also die Summe der Fehlervektoren zwischen der transformierten Punkte aus  $P$  und dem zugehörigen Punkt aus  $Q$ .

Mit der Fehlerfunktion und der Jakobi-Matrix kann nun das die Hesse-Matrix und der Gradient wie folgt berechnet werden. Die Hessematrix  $H$  und der Gradient  $g$  (a.k.a.  $E'$ ) werden zunächst mit Nullen initialisiert. Danach muss für jede Korrespondenz in  $C$  folgende Berechnung durchgeführt werden:

$$H \rightarrow H + J^T J \quad (5.10)$$

$$g \rightarrow g + J^T e \quad (5.11)$$

mit der resultierenden Matrix  $H$  und dem Gradientenvektor  $g$  kann nun durch Lösen des Gleichungssystem  $\Delta x$  berechnet werden:

$$H\Delta x = -g \Rightarrow \Delta x = -H^{-1}g \quad (5.12)$$

Dieses  $\Delta$  muss nun noch auf die aktuellen Argumente von  $x$  addiert werden, um die Rotation und Translation für die nächste Iteration zu erhalten.

Der soeben erläuterte Ablauf wird als Flussdiagramm in Abbildung 5.4a so gezeigt, wie er im Toolkit umgesetzt wurde. Hier fällt auf, dass es bereits bevor das Gleichungssystem das erste Mal gelöst wurde, eine Abfrage gibt, ob der Algorithmus bereits konvergiert ist. Dies geschieht für den Fall, dass sich die zwei Punktwolken zu Beginn bereits annähernd übereinander liegen. Die Fehlermetrik wird dabei bereits mit dem Gleichungssystem im *perpare-equation-system*-Schritt zurückgegeben, da diese ohnehin den Fehlervektor zur Aufstellung des Systems benötigt.

In Abbildung 5.4b ist außerdem ein Ablaufdiagramm zu sehen, in dem die einzelnen Schritte zur Gleichungssystemerstellung zu sehen sind.

### IcpPoint2Plane

Der letzte ICP-Algorithmus dieser Arbeit unterscheidet sich vom vorherigen Algorithmus nur an einer einzigen Stelle: Die genutzte Fehlermetrik. Der IcpPoint2Point-Algorithmus versucht die euklidische Distanz zwischen Punktpaaren zu minimieren. Der

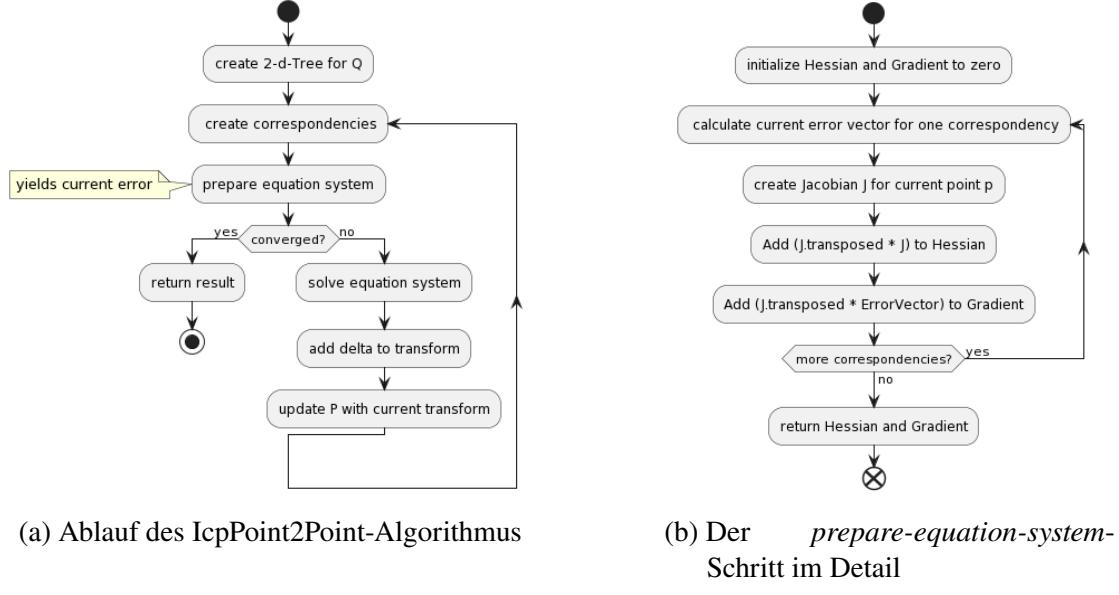


Abbildung 5.4.: Flussdiagramm des IcpPoint2Point-Algorithmus

IcpPoint2Plane hingegen minimiert, wie in 3.3 bereits erwähnt, nicht diese Distanz zwischen zwei Punkten, sondern die Skalarprojektion des Abstands auf die durch den Normalenvektor definierte ebene Fläche. Diese Vorgehensweise führt zu einer schnelleren Konvergenz[84] und besseren Robustheit, hat aber den Nachteil, dass das die Genauigkeit schlechter ist als bei der Point-to-Point-Metrik.

Die neue zu minimierende Fehlerfunktion ist also

$$E = \sum_i ((Rp_i + t - q_i) \cdot n_{qi})^2 \rightarrow \min, \quad (5.13)$$

wobei  $n_{qi}$  der Normalenvektor von Punkt  $q_i$  ist.

Durch diese Skalarprojektion wird die Jakobi-Matrix, von einer 2x3 zu einer 1x3 Matrix:

$$J = [n_x \ n_y \ n_x(-p_x \sin(\theta) - p_y \cos(\theta)) + n_y(p_x \cos(\theta) - p_y \sin(\theta))] \quad (5.14)$$

Alle anderen Berechnungen und der Ablauf des Algorithmus können unverändert bleiben.

Was für diese Methode natürlich noch benötigt wird, ist der Normalenvektor jedes Punktes aus  $Q$ . Für einen 2D-Vektor  $v = (x, y)^T$  kann die Normale trivial ermittelt werden durch  $n_v = (-y, x)^T$ . Für die Punktwolke  $Q$ , die im Spezialfall dieser Arbeit bereits geordnet

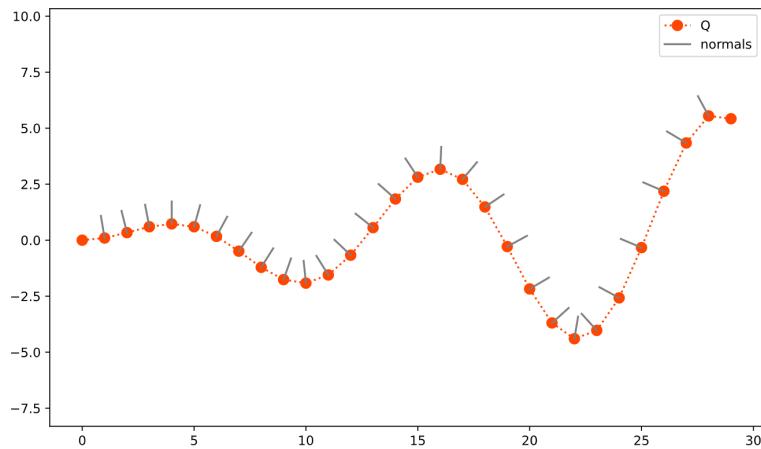


Abbildung 5.5.: Berechnung der Normalen für jeden Punkt einer sortierten Punktwolke.  
Aus [83]

von der Oculus-API abrufbar ist, kann für einen Punkt  $p_i$  der Normalenvektor bestimmt werden in dem zuerst der Vektor zwischen  $p_{i-1}$  und  $p_{i+1}$  berechnet wird. Die Normale dieses Vektors ist dann  $n_{pi}$ . In Abbildung 5.5 ist das Resultat des eben beschriebenen Verfahrens zur Normalenberechnung einer Punktwolke zu sehen.

### Wahl des ICPs

Alle beschriebenen Algorithmen befinden sich in der *Calibration*-Bibliothek und sind theoretisch nutzbar. Der **Calibrator**, der den Hauptzugriffspunkt zur Bibliothek darstellt, verwendet jedoch eine Kombination aus zwei der vorgestellten Algorithmen. Es wird eine initiale Transformationsschätzung mithilfe einer abgewandelten Form des **IcpBfKdtree** durchgeführt. Mit diesem sogenannten *Initial Guess*, wird die Punktwolke transformiert, damit sich die zwei Punktwolken schon grob annähern. Mit dieser neuen Punktwolke wird dann durch den **IcpPoint2Point**-Algorithmus die endgültige Transformation berechnet. Warum dies geschieht, ist in später im Detail in Kapitel 6.1 zu lesen, aber kurz zusammengefasst verhindert der *Initial Guess*, das Fallen in ein lokales Minimum und der **IcpPoint2Point** liefert das beste Registrierungsergebnis. In Pseudocode ausgedrückt, sieht dieser Ablauf im Quellcode des **Calibrator** in etwa wie folgt aus.

```

1 Transform T1 = IcpBfKdtree::InitialGuess(Q, P, 45);
2 PointCloud P_transformed = TransformPointCloud(P, T1);
3 Transform T2 = IcpPoint2Point::FindTransform(P_transformed, q);
4 Transform T_PQ = T2 * T1 //final transform P->Q

```

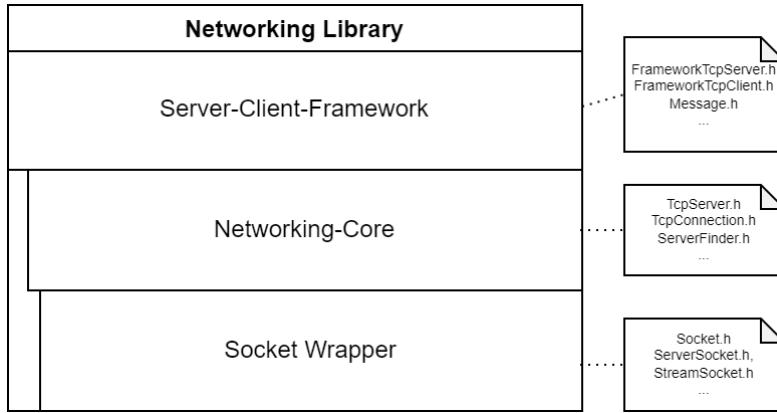


Abbildung 5.6.: Die Architektur der *Networking*-Bibliothek, aufgeteilt in Schichten.

Wie oben zu sehen, setzt sich die endgültige Transformation aus dem *Initial Guess* und dem Icp-Ergebniss zusammen:

$$T^* = T^{icp} * T^{init\_guess} \quad (5.15)$$

## 5.4. Die *Networking*-Bibliothek

In dieser Sektion wird die Architektur und Umsetzung der *Networking*-Bibliothek behandelt. Hierfür wurde die Sektion noch einmal in zwei Untersektionen aufgeteilt. Im ersten Teil werden alle Grundbausteine behandelt, die nötig sind, um Netzwerkkommunikation in einem lokalen Netzwerk umzusetzen. Im zweiten Teil wird beschrieben, wie aus diesen Grundbausteinen ein wiederverwendbares Client-Server-Framework erstellt wurde.

Die grobe Architektur der Netzwerk-Bibliothek ist in Abbildung 5.6 dargestellt. Wie dort erkennbar ist, wurde die Bibliothek in mehrere Ebenen aufgeteilt. Hierbei baut immer die obere Ebene auf der unteren Ebene auf. Diese Ebenen werden nun von unten nach oben näher betrachtet.

### 5.4.1. Socket-Wrapper

Auf der untersten Ebene der Netzwerk-Bibliothek befinden sich die *Socket-Wrapper*. Die Netzwerksocket-API wird vom Betriebssystem bereitgestellt und ermöglicht die Kommunikation zwischen Geräten auf niedrigster Ebene. Wie eben gesagt, wird die API vom Betriebssystem bereitgestellt, um also Sockets auf Windows Betriebssystem zu verwenden, müssen andere Bibliotheken und Funktionen verwendet werden als



Abbildung 5.7.: Klassendiagramm der Socket-Wrapper

auf Android/Unix-Systemen. Laut Anforderungen soll das Plugin auf diesen beiden Betriebssystem lauffähig sein. Um nicht bei jedem API-Aufruf eine Fallunterscheidung programmieren zu müssen, wurde für alle Socket Zugriffe Wrapper-Objekte entworfen. Durch diese Wrapper müssen die Fallunterscheidungen nur an einer einzigen Stelle erfolgen und machen den Code dadurch übersichtlicher und wartbarer.

Um Socketoperationen im Projekt über eine konsistente Schnittstelle nutzbar zu machen, wurden insgesamt vier Klassen entworfen. Diese sind im gekürzten UML-Diagramm in Abbildung 5.7 zu sehen. Wie zu erkennen ist, bildet die `Socket`-Klasse das Fundament für alle anderen Klassen. Hier ist der Großteil der Logik implementiert. Im Grunde wurden die meisten Socket-Funktionen der Betriebssysteme mit einem Objekt umhüllt. Beim Aufruf dieser Funktionen wird durch Präprozessor Anweisungen, die für das Betriebssystem benötigte Funktion verwendet. Beim Schließen eines Socket sieht das beispielsweise in etwa wie folgt aus.

```

1     bool Socket::Close() {
2         int result;
3 #ifdef _WIN32
4             result = closesocket(m_socketPtr);
5 #else

```

```

6         result = close(m_socketPtr);
7 #endif
8         return result == 0;
9     }

```

Die **Socket**-Klasse ist eine abstrakte Klasse, von der keine Instanzen existieren dürfen. Beim Öffnen eines Socket muss die Art des Sockets angegeben werden. Dies geschieht im Konstruktor der zwei Kindklassen **StreamSocket** und **DatagramSocket**. Der **DatagramSocket** repräsentiert einen Socket der mittels UDP-Protokoll, Nachrichten verbindungslos an gewünschte Adressen senden und auch empfängt. Hierbei wird vom Betriebssystem keine Verbindung verwaltet, sodass nicht garantiert ist, dass gesendete Nachrichten vollständig ankommen. Die **StreamSockets** hingegen repräsentiert einen Socket, welcher das TCP-Protocol verwendet. Solange eine Verbindung besteht, ist garantiert, dass alle Nachrichten, die in dieser Verbindung versendet wurden, auch vollständig empfangen werden. Bei StreamSockets kann noch einmal zwischen zwei Arten unterschieden werden. Es gibt aktive- und passive StreamSockets.

Die aktiven **StreamSocket** stellen die Verbindung zwischen zwei Geräten dar. Durch die Funktionen **Send()** und **Receive()** können Nachrichten zwischen den Geräten ausgetauscht werden. Passive **StreamSockets** (**ServerSockets**) warten an einem gewünschten Port auf Verbindungsanfragen und erstellen bei akzeptierter Anfrage, einen neuen aktiven **StreamSocket**, welcher diese neue Verbindung verwaltet. In der UML-Abbildung 5.7 ist zu sehen, dass die **ServerSocket**-Klasse privat von der **StreamSocket**-Klasse erbt. Dadurch wird verhindert, dass ein passiver Socket fälschlicherweise zum Senden oder Empfangen von Nachrichten verwendet wird.

Mithilfe dieser vier soeben beschriebenen Klassen ist also sämtliche Funktionalität gegeben, um die Kommunikation zwischen mehreren Geräten mit verschiedenen Betriebssystemen zu ermöglichen.

#### 5.4.2. Server-Client Grundgerüst

##### Acceptor-Connector

Durch die Socket-Wrapper-Ebene ist es also möglich, Plattform unabhängig auf die Socket-API zuzugreifen. Der nächste Schritt ist nun, diese API zu verwenden, um Verbindungen zwischen verteilten Systemen herzustellen. Das direkte Arbeiten mit Sockets ist sehr aufwendig und fehleranfällig. Deshalb werden in der *Networking-Core*-Ebene einige Klassen bereitgestellt, die den Verbindungsaufbau und das Verwalten der Verbindungen deutlich vereinfachen.

Um den Verbindungsaufbau und die Applikationslogik voneinander zu trennen, wurde eine abgeänderte Version des *Acceptor-Connector* Entwurfsmusters verwendet. Dieses Muster wurde 1997 von Douglas C. Schmidt[85] vorgestellt und hat das Ziel, den Verbindungsaufbau von der Service-Initialisierung bei verteilten Systemen zu entkoppeln. Diese Entkopplung wird mithilfe von drei Komponenten erzielt: Acceptor, Connector und Service-Handler. Ein Connector erstellt aktiv eine Verbindung mit einem remote Acceptor und initialisiert einen Service-Handler, um die ausgetauschten Daten dieser Verbindung zu verarbeiten. Auf der anderen Seite wartet der Acceptor passiv auf Verbindungsanfragen von remote Connectors und initialisiert bei Verbindungsaufbau auch einen Service-Handler. Die Service-Handler führen dann applikationsspezifischen Code aus und kommunizieren über die durch den Acceptor und Connector erstellte Verbindung. Da im Service-Handler und bei dessen Initialisierung beliebiger Applikationscode ausgeführt werden kann, eignet sich dieses Muster gut für ein Framework. Das *Acceptor-Connector-Pattern*, wie es in [85] beschrieben wurde, ist für einen asynchronen Prozessablauf konzipiert. Im Plugin dieser Arbeit wird beim Verarbeiten der Netzwerknachrichten jedoch auf Multithreading gesetzt, um die Rechenleistung der zusätzlichen Kerne auszunutzen. Deshalb wurde das Entwurfsmuster in manchen Punkten deutlich geändert, doch das Grundkonzept ist das Gleiche.

In Abbildung 5.8 ist ein Sequenzdiagramm zu sehen, welches den Ablauf zur Erstellung einer Verbindung aus Sicht des Servers zeigt. Ganz links ist der **TcpServer** zu sehen. Der **TcpServer** benutzt die **StreamAcceptor**-Klasse, um in einer Endlosschleife Verbindungsanfragen abzuarbeiten. Da diese Endlosschleife den Hauptthread der Render-Engine blockieren würde, wird die **Run()**-Funktion des Servers in einem separaten Thread ausgeführt. Um für Klassen eine einfache Möglichkeit zu bieten, Code in einem neuen Thread auszuführen, wurde eine **Thread**-Klasse implementiert. Diese kapselt, wie im Klassendiagramm 5.9 zu sehen, die **Thread**-API der Standard-Bibliothek in einer rein abstrakten Klasse. Wenn andere Klassen von dieser Erben müssen sie die **Run()**-Funktion überschreiben. Jeglicher Code in dieser überschriebenen Funktion wird beim Aufrufen von **Start()** in einem neuen Thread ausgeführt.

Anders als im original *Acceptor-Connector*-Muster, hat der **StreamAcceptor** wirklich nur die Aufgabe, Verbindungen zu akzeptieren. Im Originalmuster, hatte er außerdem die Aufgabe mithilfe einer Factory die Service-Handler zu erstellen. Diese Aufgabe übernimmt in der Umsetzung dieser Arbeit der **TcpServer**, damit der Acceptor wirklich nur eine Aufgabe zu erledigen hat. Der **TcpServer** benötigt also eine Factory, welche sich um die Erstellung und Initialisierung der Service-Handler kümmert. Für die Erstellung dieser Factory muss von der rein virtuellen Klasse **TcpConnectionFactory** geerbt und dessen einzige Funktion **CreateConnection()** implementiert werden. Eine Instanz dieser Klasse muss dem **TcpServer** dann im Konstruktor übergeben werden. Die Factory produziert also **TcpConnections**. Diese repräsentieren in dem Muster die

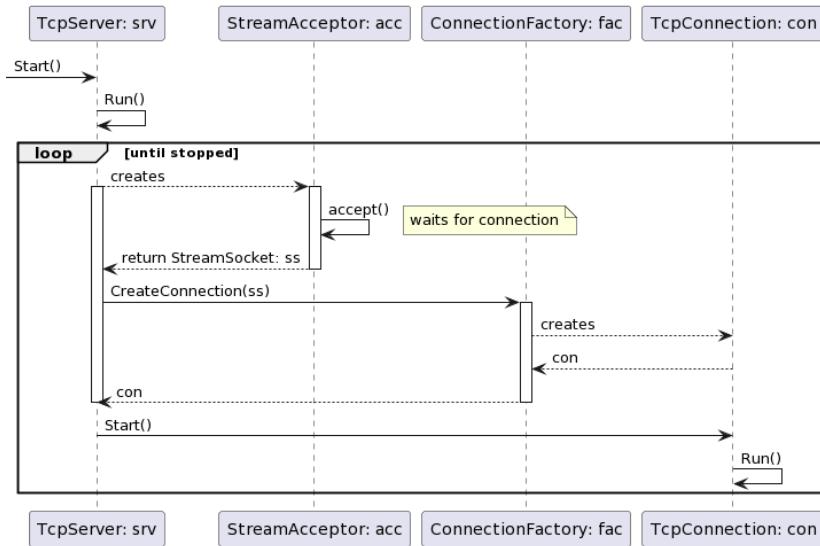


Abbildung 5.8.: Ablauf des Verbindungsaufbaus zwischen einem Acceptor und Connector aus Serversicht

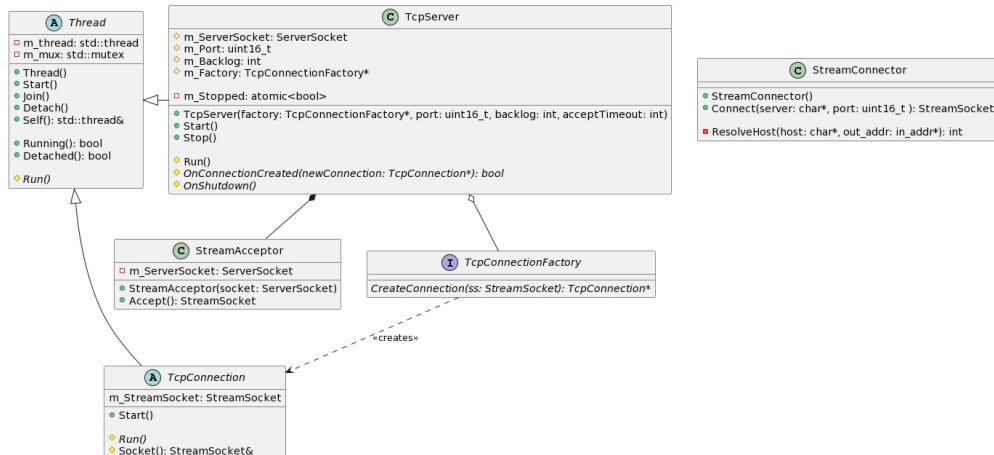


Abbildung 5.9.: (Gekürztes) Klassendiagramm der Acceptor-Connector Komponenten

Service-Handler. Eine `TcpConnection` ist also dafür verantwortlich, den Datenaustausch zwischen den zwei verbundenen Geräten abzuwickeln. Für das Empfangen und Senden von Nachrichten durch die Sockets wird eine blockierende API verwendet, weshalb die `TcpConnection` ebenfalls von `Thread` erbt, um den Hauptthread nicht zu blockieren. Für jeden verbundenen Client, wird also eine `TcpConnection` erstellt, die auf einem eigenen Thread mittels eines `StreamSockets` Nachrichten austauscht.

Um noch einmal auf den Ablauf im Sequenzdiagramm 5.8 zurückzukommen: Der

`TcpServer` nutzt den `StreamAcceptor`, um in einem Nebenthread zu warten, bis Verbindungsanfragen eingehen. Wenn eine Anfrage eingeht, erstellt der Acceptor einen `StreamSocket` für diese Verbindung. Dieser Socket wird dann der Factory weitergegeben, damit sie einen Service initialisieren kann. Dieser Service (`IcpConnection`) führt dann in seinem eigenen Thread, applikationsspezifischen Code aus, bis die Verbindung getrennt wird oder der Thread terminiert (in dem z. B. die Endlosschleife verlassen wird).

Der `TcpServer` so wie er im Netzwerk-Kern zur Verfügung steht, implementiert wirklich nur die Logik, die nötig ist, um einen Server auf einem angegebenen Port zu starten, Verbindungsanfragen zu beantworten und ggf. den Applikationscode im Service-Handler auszuführen. Die Klasse kann zwar unverändert verwendet werden, ist aber eher als abstrakte Klasse konzipiert, sodass sie durch Vererbung erweitert werden kann. Beispielsweise ist es im `TcpServer` nicht einmal möglich abzufragen, wie viele Klienten zurzeit verbunden sind oder Nachrichten an alle Klienten zu senden. Um Funktion in geerbten Klassen zu ermöglichen, existieren die überschreibbare Event-Funktionen `OnConnectionCreated(newConnection)`, welche immer aufgerufen wird, sobald ein Klient akzeptiert wurde. Auf Clientseite kann mit dem `StreamConnector` und der IP-Adresse durch den Aufruf der `Connect(ip, port)` eine Verbindung angefragt werden (Siehe API im Klassendiagramm 5.9). Die Frage ist nun: Wie bringt man die Netzwerk-adresse des Servers im lokalen Netzwerk in Erfahrung, ohne diese manuell einzugeben? Dieses Problem lösen die im Folgenden beschriebenen zwei Klassen.

### Ad-Hoc Sitzungen im lokalen Netzwerk

Um ad-hoc Sitzungen im lokalen Netzwerk erstellen zu können, musste eine Verfahren entwickelt werden, mit dem Client und Server sich selbstständig entdecken können. Hierbei war auch darauf zu achten, dass sich mehrere Instanzen des Servers im Netzwerk befinden können, ohne sich gegenseitig zu stören. Um diese Aufgabe zu bewältigen, wurden die zwei Klassen `ServerInfoDispatcher` und `ServerFinder` entworfen. Wie im Klassendiagramm in Abbildung 5.10 zu sehen, bieten die beiden Klassen eine sehr einfache Schnittstelle an. Der `ServerInfoDispatcher` kann mit `Start()` gestartet und mit `Stop()` beendet werden. Nachdem er gestartet wurde, läuft sämtliche Logik in einem eigenen Thread und beantwortet, wie in Kürze näher beschrieben, Anfragen des `ServerFinders`. Dieser bietet, wie ebenfalls in 5.10 zu sehen, nur die zwei öffentlichen Funktionen `TryFind()` und `TryFindAsync()`. Wobei letztere dieselbe Funktionalität wie die Erstere hat, nur dass sie in einem neuen Thread ausgeführt wird, um die Programmausführung nicht zu blockieren.

Der genaue Ablauf, wie Server und Client sich finden, ist im Sequenzdiagramm in Abbildung 5.11 verbildlicht. Wie bereits erwähnt, wird die Logik der `ServerInfoDispatcher`

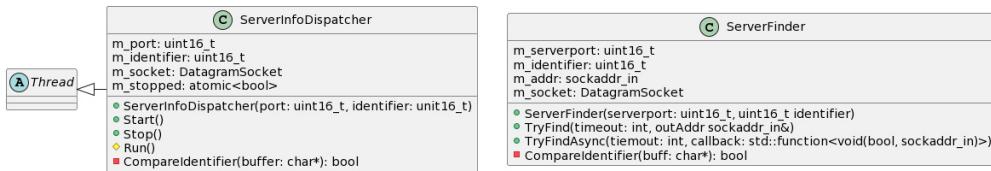


Abbildung 5.10.: Die ServerInfoDispatcher- und ServerFinder-Klassen als Diagramm

(abk. Dispatcher) auf dem Server-Gerät in einem eigenen Thread ausgeführt. Der Dispatcher bindet einen DatagramSocket an einen Port, welcher dem Server und dem Client bekannt sein muss. Dieser Socket wird dann so konfiguriert, dass Broadcast Nachrichten von beliebigen Adressen empfangen werden können. Mit diesem DatagramSocket wird nun gewartet, bis jemand im Netzwerk eine Nachricht an den Port sendet. Im Idealfall ist dies ServerFinder-Instanz eines Clients. Mit dem ServerFinder sendet der Client periodisch UDP-Broadcast-Nachrichten in das Netzwerk an den festgelegten Port. Diese Nachricht enthält einen sogenannten Identifier. Der Identifier gibt an, welche Klienten und Server zusammengehören. So können zum Beispiel mehrere Server-Instanzen im lokalen Netzwerk laufen und es würden sich die Klienten nur mit den Servern verbinden, die den gleichen Identifier benutzen. Sobald der Dispatcher eine Nachricht aus dem Netzwerk empfängt, wird diese als Identifier (16-Bit Integer) interpretiert. Ist der gesendete Identifier derselbe wie der im Server hinterlegte, wird er inkrementiert und an den Sender zurückgesandt. Er wird für den Fall inkrementiert, dass sich ein anderer Server im Netzwerk befindet, der jede Antwort unverändert zurücksendet (Echo Server). Im ServerFinder wird derweilen auf die Antwort des Dispatchers gewartet. Sobald diese ankommt, wird auch sie wieder als Identifier interpretiert, dekrementiert und mit dem eigenen Identifier verglichen. Sollten diese übereinstimmen, kann der Finder aus den Informationen der empfangenen Nachricht die IP-Adresse des Servers entnehmen und diese zur Verbindung mit dem TCP-Server verwenden.

### 5.4.3. TCP Client-Server-Framework

Mit den soeben beschriebenen Komponenten des *Netzwerk-Cores* ist sämtliche Funktionalität gegeben, um ein Client-Server-Framework zu erstellen. Das Framework bietet eine Client- und eine Serverklasse, die mit dem Ziel entworfen wurden, sie möglich einfach mit applikationsspezifischer Logik erweitern zu können. Das Framework übernimmt dabei den komplexen Teil der Nachrichtenverarbeitung. Der Austausch der Nachrichten zwischen Server und Client ist in Abbildung 5.12 bildlich dargestellt.

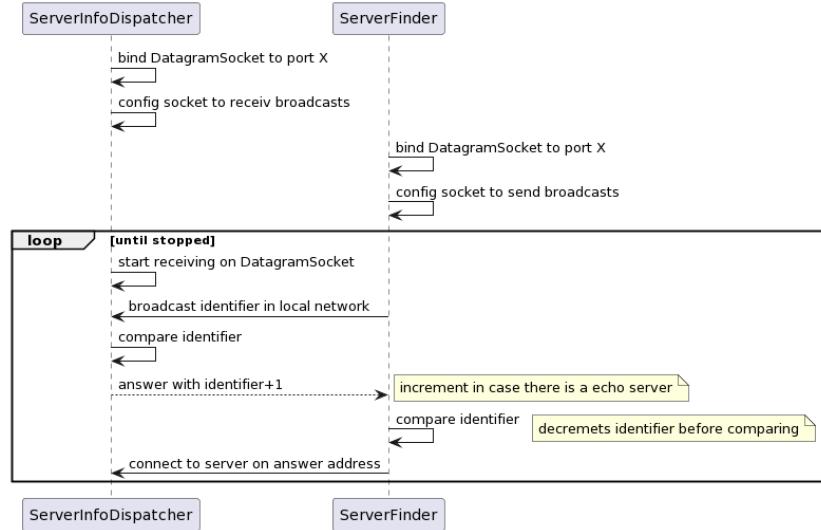


Abbildung 5.11.: Ablauf bzw. Zusammenspiel des ServerFinder und ServerInfoDispatcher, um sich im lokalen Netzwerk zu finden.

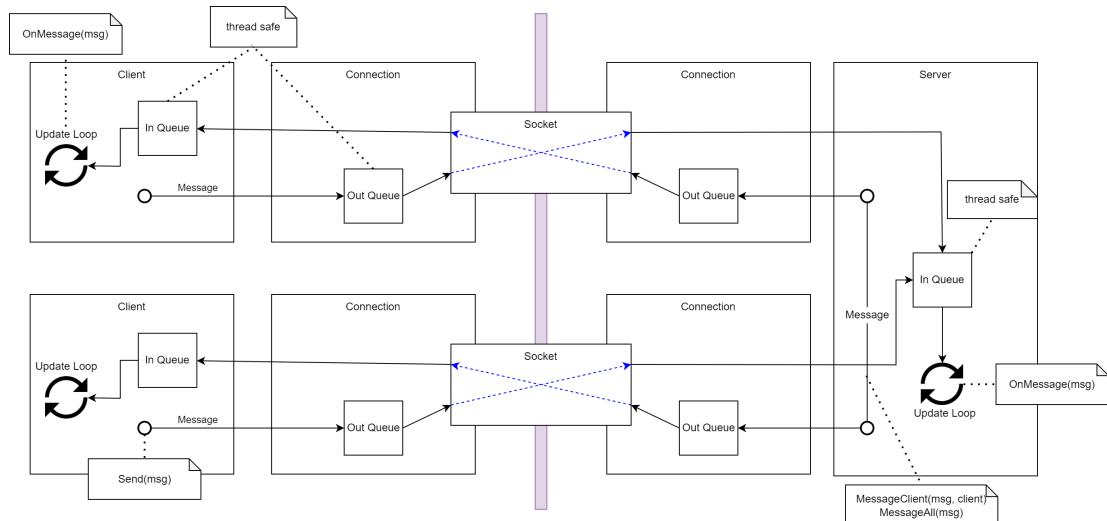


Abbildung 5.12.: Visualisierung des Nachrichtenaustausches zwischen einem Server und zwei Clients. Durch Threadsafe Queues werden die Nachrichten im Update-Loop mit dem Haupthead synchronisiert.

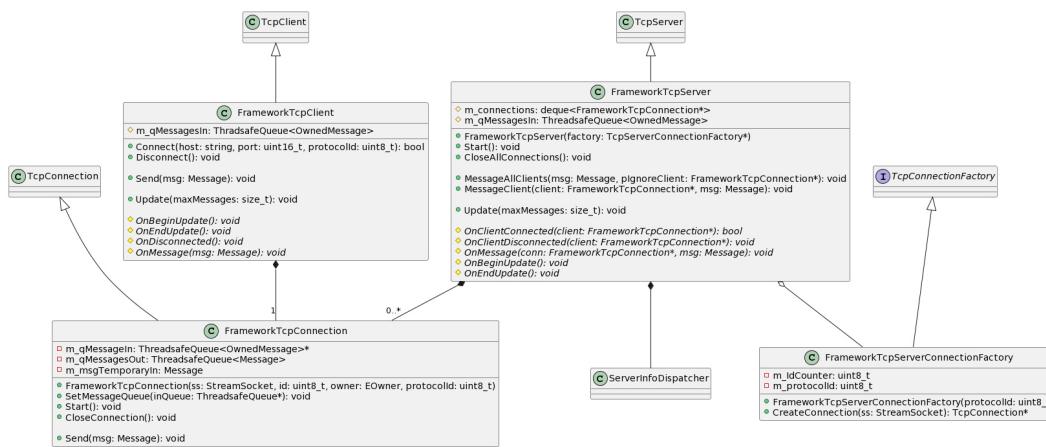


Abbildung 5.13.: Diagramm einiger vom Framework zur Verfügung gestellter Klassen

Wie früher bereits erwähnt, wird für jede Server-Client-Verbindung auf beiden Seiten eine `TcpConnection`-Instanz erstellt, die in einem neuen Thread die Datenübermittlung durchführt. Um Nachrichten zwischen den Hauptthread und den Connection-Threads auszutauschen, werden `Threadsafe Queues` verwendet. Jede empfangene Nachricht im Server oder auch Client wird in also threadsicher in eine Queue geschrieben. Um die Nachrichten im Hauptthread abzurufen, besitzt der `FrameworkTcpServer` und `FrameworkTcpClient` die öffentliche Funktion `Update()`. Diese ist dafür gedacht, in jedem `Update`-Loop der Render-Engine ausgeführt zu werden. Alle Nachrichten, die sich zum Zeitpunkt der Ausführung in der Queue befinden, werden über die überschreibbare Event-Funktion `OnMessage(msg)` erbenden Klassen zum Verarbeiten übergeben. Auf ähnliche Weise funktioniert auch das Senden von Nachrichten. Hierfür fügt ein Client oder Server durch die entsprechende Funktion Nachrichten in eine weitere threadsichere Queue. Diese wird dann im Thread der `FrameworkTcpConnection` aus der Queue entfernt und versendet. Wie im Klassendiagramm in 5.13 zu sehen, bietet der `FrameworkTcpServer` noch einige weitere, in erbenden Klassen überschreibbare Event-Funktionen wie z. B. `OnClientConnected()` und `OnClientDisconnected()`, mit denen ein Sitzungsmanagement umgesetzt werden kann. Außerdem ermöglicht der Server, mit den Methoden `MessageClient(msg, client)` und `MessageAllClients(msg)` einen spezifischen Client oder allen verbundenen Clients Nachrichten zu senden. Da ein Klient immer nur mit einem einzigen Server verbunden sein kann, besitzt der `FrameworkTcpClient`, wie ebenfalls in 5.13 zu sehen, nur eine `Send(msg)`-Funktion zur Nachrichtenübermittlung.

Wenn man die `TcpConnection`-Klasse aus 5.9, mit der erbenden Klasse `FrameworkTcpConnection` in 5.13 vergleicht, fällt auf, dass in Ersterer noch keinerlei Logik zum Senden und Empfangen von Nachrichten implementiert ist. Es wird lediglich ein `StreamSocket` zur Verfügung gestellt, der von erbenden Klassen zum

Header			Body
Protocol ID	Message Type	Body Size	Byte Buffer

Abbildung 5.14.: Aufbau des Message-Datentyps, welcher zur Nachrichtenübertragung verwendet wird.

Senden und Empfangen verwendet werden kann. Genau diese Logik wurde in der `FrameworkTcpConnection`-Klasse umgesetzt. Hierfür wurde, aufbauend auf dem TCP-Protokoll ein eigenes Protokoll entworfen. Die vom `StreamSocket` bereitgestellte API zum Senden und Empfangen von Nachrichten benötigt einen Pointer zu einem Puffer und die Anzahl der Bytes, die gesendet bzw. empfangen werden sollen. Wie im UML-Diagramm 5.13 zu sehen, erwartet die API der `FrameworkTcpConnection`, zum Senden einer Nachricht jedoch den Datentyp `Message`. Dieser Datentyp bestimmt wie der Byte-Puffer, der am Ende gesendet und empfangen wird, anzusehen hat. Eine `Message` besteht wie in Abbildung 5.14 zu sehen aus einem Header und dem dazugehörigen Body. Im Header befinden sich die Information, welche Protokollversion genutzt wird, welche Art von Nachricht gesendet wird und wie groß der zugehörige Body ist. Im Body befindet sich die eigentliche Nachricht, die vom Empfänger je nach Nachrichten-Typ entsprechend interpretiert werden muss. Beim Entwurf des `Message`-Structs wurde viel Wert darauf gelegt, die Erstellung und das Auswerten möglichst einfach zu gestalten. Hierfür wurden die zwei Operatoren `<<` und `>>` mit Templateargumenten überschrieben. Mit der Nutzung von `<<` können Daten von einfachen Datentypen zur Nachricht hinzugefügt werden und mit `>>` wieder ausgelesen. Im Pseudo-Quellcode sieht dies wie folgt aus.

```

1 Message m(EMessageTypes::Node_update);
2
3 //data to send:
4 float pos[3];
5 float rot_q[4];
6
7 //put into message
8 m << pos;
9 m << rot_q;
10
11 //on receiving side:
12 if(m.header.type == EMessageTypes::Node_update){
13     m >> rot_q;
14     m >> pos;
15 }
```

Das Feld im Header, welche die Größe des Bodys angibt, wird dabei automatisch angepasst.

Nachrichten werden in dieser Form auf der ganzen Framework-Ebene verwendet und über virtuelle Funktionen wie z. B. `OnMessage(msg)` auch an erbende Klassen weitergereicht. Nur die `FrameworkTcpConnection` muss die Nachricht als Byte-Puffer an die Socket-API weitergeben.

Beim Empfangen erwartet die Connection immer zuerst einen Header mit fixer Größe, in dem die Größe des als Nächstes zu empfangenen Bodys angegeben ist. Für diese Anzahl an Bytes wird ein Puffer alloziert, welcher vom Socket mit Daten gefüllt wird. Hierbei wird in der Connection sichergestellt, dass auch wirklich alle Daten ankommen. Dieser Puffer wird dann zusammen mit dem Header zur Weiterverarbeitung in die eingehende Message-Queue des Klienten oder Servers geschrieben.

Um das Netzwerk-Framework abzuschließen, wird noch kurz die `FrameworkTcpServerConnectionFactory` behandelt. Diese ist für die Instanziierung der `FrameworkTcpServerConnections` zuständig. Hierbei vergibt die Factory bei der Instanziierung jeder Verbindung eine einzigartige ID, welche später verwendet werden kann, um einen Benutzer zu identifizieren.

Zusammengefasst bietet das oben beschriebene Networking-Framework die Möglichkeit, multithreaded Client-Server-Verbindungen herzustellen, mit denen über eine einfache API Nachrichten erstellt und ausgetauscht werden können. Dabei wurden die Klassen so entworfen, dass sie mit applikationsspezifischem Code erweiterbar sind. Außerdem ist der `FrameworkServer` und dem `FrameworkClient` für die Zusammenarbeit mit einer Render-Engine entworfen, sodass die Nachrichten mit dem Hauptthread synchronisiert werden.

## 5.5. Die *Core*-Bibliothek

Die letzte Bibliothek im nativen Teil des Plugins ist die *Core*-Bibliothek. Sie benutzt die Komponenten aller anderen vorgestellten Bibliotheken und stellt eine API für die Render-Engine bereit. Die API wird durch die `COLTK_Facade`-Klasse bereitgestellt, welche eine Facade[77] ist, die das Zusammenspiel der konkreten Server-Client-Implementationen und dem Calibrator koordiniert. Außerdem verwaltet sie den Lebenszyklus des Plugins. Die API ist im Klassendiagramm von Abbildung 5.15 zu sehen. Die geplante Verwendung der API kann am einfachsten durch den folgenden Pseudocode demonstriert werden:

```

1 COLTK_Facade f;
2 f.Init(network_settings, calibration_settings);
3 f.ConnectOrCreateServer();
4 f.SetCalibrationPose(pose); //broadcasts calibration data to other clients
5
6 //do in render-engine tick

```

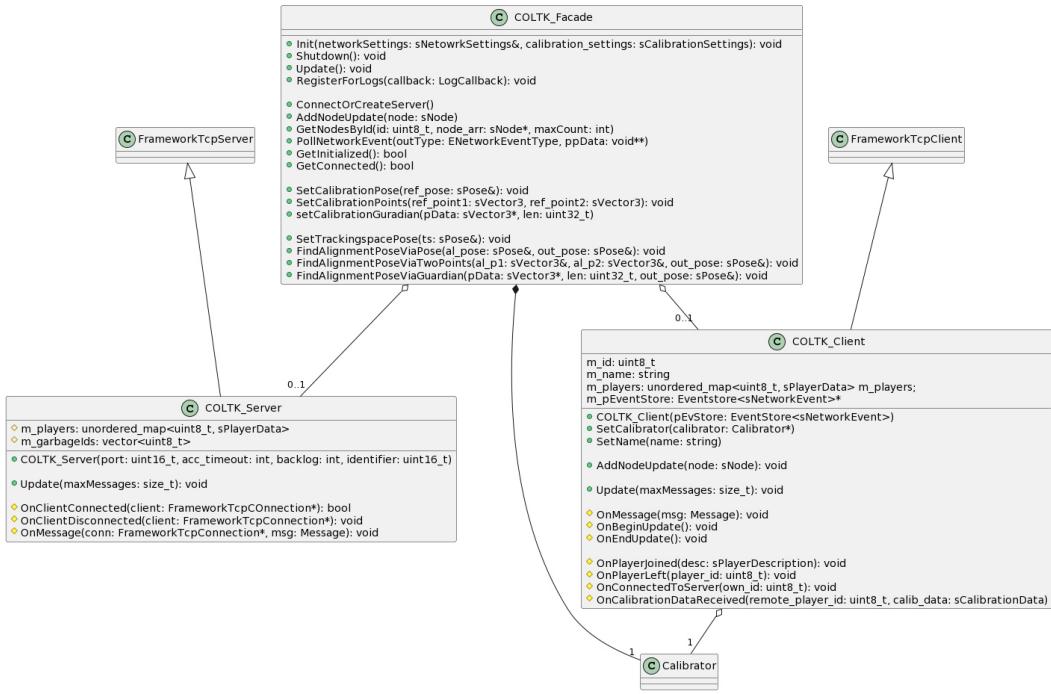


Abbildung 5.15.: (Gekürztes) Klassendiagramm der Core-Bibliothek. Hier ist zu sehen, welche API durch die Facade der Render-Engine angeboten wird.

```

7 f.AddNodeUpdate(node); //updates the pose of one local node.
8 f.Update(); //syncs server-client network messages with thread. Sends poses of all local nodes
9 f.PollNetworkEvent(outEventType, ppData); //get network events since last update()
10 f.GetNodesById(remoteId, pNodeArray, maxCount); //gets all nodes from a remote user
    mapped to local frame
11 //end render-engine tick
12
13 f.Shutdown(); //shuts down server & client and cleans up

```

Der oben gezeigte Pseudocode zeigt die wie die Verwendung der Facade auf Render-Engine-Ebene aussehen könnte. In dem Codeausschnitt werden die wichtigsten API-Aufrufe demonstriert, die nötig sind, um eine Co-Location Sitzung umzusetzen. Ganz am Anfang (Zeile 2) muss das Plugin initialisiert werden. Hierbei werden die Netzwerk- und Kalibrationseinstellungen von Render-Engine-Ebene zur nativen Ebene übermittelt. In den Netzwerkeinstellungen sind Informationen über verwendeten Port, Server-Identifier und Name des Klienten enthalten. Außerdem beinhalten die Netzwerkeinstellungen eine Timeout-Zeit, die angibt, wie lange im lokalen Netzwerk nach einem Server gesucht werden soll, bis dies aufgegeben und ein eigener Server erstellt wird. Was in den Kalibrationseinstellungen angegeben wird, wurde bereits in 5.3 beschrieben.

In der nächsten Zeile des Quellcodes wird mit `ConnectOrCreateServer()` zunächst mit der eben erwähnten Time-Out-Zeit im Netzwerk nach einem existierenden Server gesucht und falls nach Ablauf der Zeit keiner gefunden wurde, ein neuer erstellt. In echtem Code muss an dieser Stelle gewartet werden, bis eine Verbindung aufgebaut ist. Die Facade bietet hierfür die entsprechenden Getter-Methoden, um den Status abzufragen. In Zeile 4 wird gezeigt, wie aus der Render-Engine die Kallibrations-Daten gesetzt werden können. Ist der Client mit einem Server verbunden, kümmert sich die Facade darum, die Daten an alle anderen Klienten zu verteilen.

Von Zeile 7 bis 10 sind Funktionsaufrufe gezeigt, die im Update-Loop einer Render-Engine ausgeführt werden sollten. Da sich die Pose von Tracking-Nodes in der Regel nach jedem Update-Loop ändern, müssen die lokalen Posen mit der `AddNodeUpdate()`-Methode (Zeile 7) aktualisiert werden. Die Funktion muss für jeden Tracking-Node einmal pro Loop aufgerufen werden.

In der nächsten Zeile ist ein Aufruf zur `Update()`-Methode der Facade zu sehen. Der Aufruf dieser Methode ist sehr wichtig, da in durch sie alle zwischenzeitlich eingegangenen Netzwerknachrichten verarbeitet werden und die aktuellen Posen der lokalen Tracking-Nodes an den Server übermittelt werden. Wie genau der Austausch der Nachrichten abläuft, wird später behandelt.

Durch manche verarbeiteten Netzwerknachrichten werden Netzwerkevents erstellt, damit auf Render-Engine-Ebene darauf reagiert werden kann. Beispielsweise werden für jeden Klienten Events bei Beitritt und verlassen der Serversitzung erstellt. Diese Events können von der Render-Engine durch die `PollNetworkEvent()` abgefragt werden. Dieser *Polling*-Anzantz wird anstelle von Callbacks verwendet, damit die Codeausführung nur von Engine in Richtung Plugin geht. Bei jedem Aufruf der Funktion wird die das älteste Event (FIFO-Prinzip) an die Engine übergeben. Auf Engine-Ebene können diese Events dann zur Umsetzung von Sitzungslogik verwendet werden. Zum Beispiel kann beim Beitritt ein Avatar in der Szene erstellt und beim Austritt wieder entfernt werden.

Die letzte Methode im Engine-Update-Loop ist `GetNodesById()` in Zeile 10. Mit dieser Methode können die Posen aller Tracking-Nodes jedes anderen verbundenen Klienten abgefragt werden. Hierbei werden die Posen automatisch in das lokale Koordinatensystem abgebildet. Sind mehrere Klienten verbunden, muss diese Methode natürlich für jeden Klienten mit dessen ID aufgerufen werden. Welche Klienten gerade verbunden sind, ist durch die eben erwähnte Netzwerkevents oder durch einen entsprechenden API-Aufruf zu ermitteln.

Wenn sich ein Benutzer entschließt, die Anwendung zu beenden, müssen alle verwendeten Ressourcen wieder freigegeben werden. Dies geschieht durch die `Shutdown()`-Funktion der Facade (Zeile 13). Durch sie werden der Client, Server und alle anderen Ressourcen aufgeräumt, sodass die Anwendung sauber beendet werden kann.

Die Funktionen für das Ausrichten des Trackingspace werden hier nicht näher behandelt, da sie nur die Argumente an den Calibrator weiterreichen und der Rest schon in 5.3

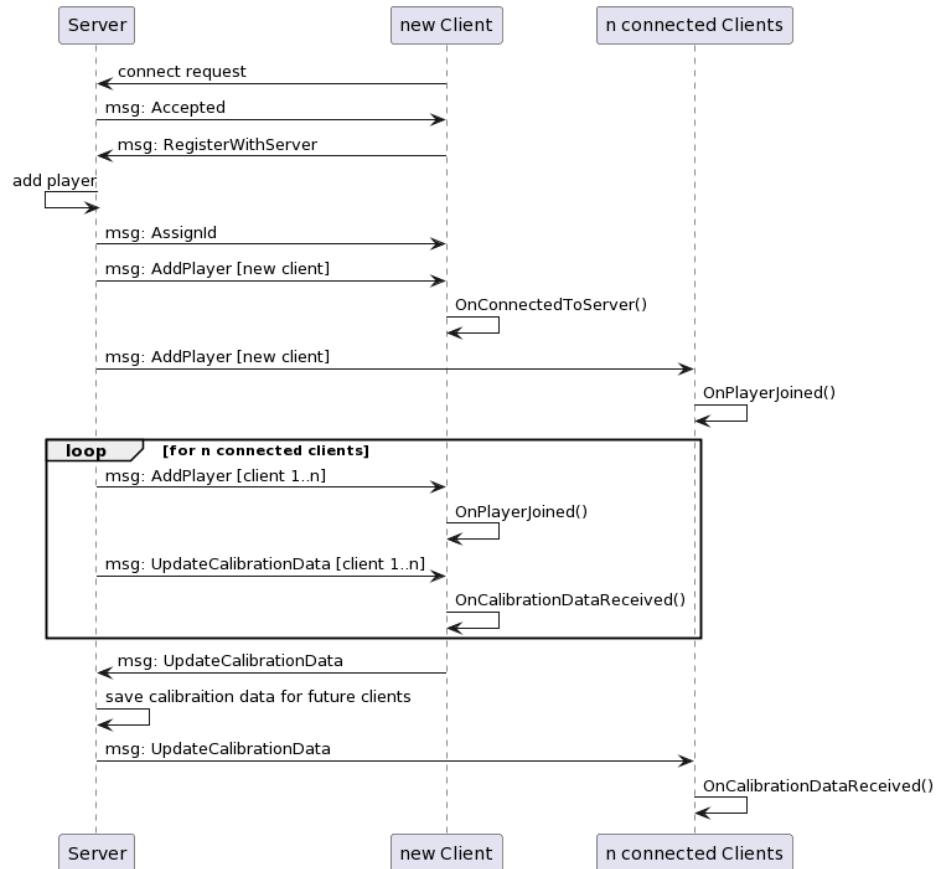


Abbildung 5.16.: Verbildlichung des Nachrichtenaustausches, wenn ein neuer Client sich beim Server registriert.

beschrieben wurde.

#### Nachrichtenaustausch

Soeben wurde beschrieben, wie die COLTK\_Facade genutzt werden kann, um Daten mit anderen Klienten auszutauschen. Wie der Datenaustausch genau abläuft, wird in Folgendem behandelt. Zunächst wird der Ablauf beim Verbinden mit dem Server erläutert. Das Sequenzdiagramm in Abbildung 5.16 stellt diesen Ablauf visuell dar. Ausgangspunkt hierbei ist ein Server, mit dem bereits  $n$  Klienten verbunden sind und ein neuer Client, der mit einer Verbindungsanfrage an den Server startet. Die Nachrichten (dargestellt mit Vorangestelltem *msg:* ) sind dabei Nachrichtentypen, die dem Server und Client bekannt sind und mittels der Message-Struktur umgesetzt wurden.

Sobald der Server eine Verbindungsanfrage erhält und diese akzeptiert, antwortet er dem Klienten mit einer Accepted-Nachricht. Der Client wiederum bittet als nächstes, um die

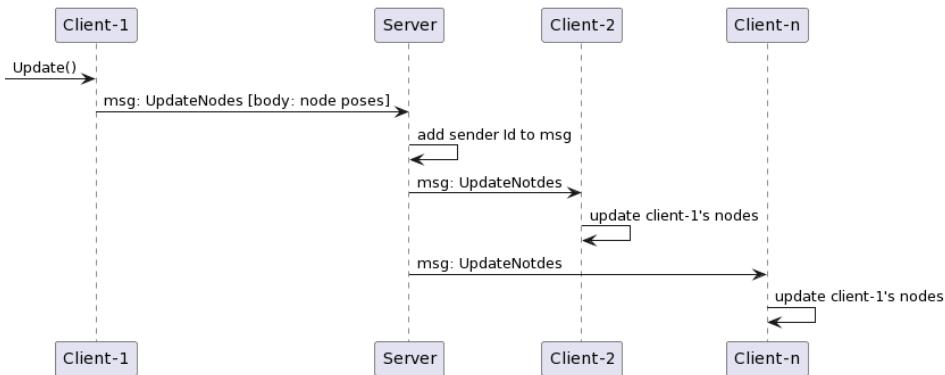


Abbildung 5.17.: Ablauf zum Aktualisieren der Tracking-Nodes eines Klienten.

Registrierung im Server. Hierfür sendet er ihm mit der `RegisterWithServer`-Nachricht seinen Namen. Auf Serverseite wird nun dem Klienten eine ID zugewiesen, die ihm mit der `AssignId`-Nachricht mitgeteilt wird. Außerdem übermittelt der Server die Daten des neuen Klienten an alle verbunden Klienten mit der `AddPlayer`-Nachricht (einschließlich des neuen Klienten). Der neue Klient erkennt hierbei, dass es sich um seine eigene ID handelt und gilt somit als registriert und erstellt das entsprechende Netzwerk-Event. Die anderen, bereits verbundenen Klienten erhalten dieselbe Nachricht, erstellen aber ein `PlayerJoined`-Event. Im nächsten Schritt werden die Daten der bereits verbundenen Klienten an den neuen Klienten gesendet. Hierfür wird für jeden verbundenen Klienten eine `AddPlayer`-Nachricht und `UpdateCalibrationData`-Nachricht an den neuen Klienten gesendet. Sobald der neue Klient die Kalibrierung durchgeführt hat, sendet er seine Daten mit der `UpdateCalibrationData`-Nachricht an den Server. Dieser speichert diese, um sie bei einer neuen Verbindungsanfrage wie eben beschrieben sofort übermitteln zu können und leitet sie an die anderen verbundenen Klienten weiter.

Die anderen wichtigen Daten, die ausgetauscht werden müssen, sind die Tracking-Node-Posen. Dieser Austausch ist sehr simpel, da der Server nur als Vermittler dient. Jedes Mal, wenn durch die Render-Engine die `Update`-Methode aufgerufen wird, sendet der Klient, falls es seit der letzten Übermittlung Änderungen der lokalen Posen der Tracking-Nodes gab, diese im Body einer `UpdateNodes`-Nachricht an den Server. Der Server fügt dieser Nachricht noch die ID des Versenders hinzu und verteilt sie dann an alle anderen Klienten. Diese Klienten bilden die Posen bei Eintreffen mithilfe der zuvor empfangenen Kalibrationsdaten in das eigene Koordinatensystem ab. Dieser Ablauf ist in Abbildung 5.17 verbildlicht.

Zusammengefasst stellt die Bibliothek also durch die `COLTK_Facade`-Klasse, der Render-Engine eine einfach zu verwendende API zur Verfügung und versteckt die komplexe Logik für den Datenaustausch zwischen Server und Client sowie Transformationsabbildungen des `Calibrators` hinter einer Facade.

## 5.6. Unity Package

In dieser Sektion wird der Entwurf des Toolkits auf Render-Engine-Ebene beschrieben. Zunächst wird gezeigt, wie die Unity-Engine mit der nativen *Core*-Bibliothek kommuniziert. Danach wird auf die Unity-Komponenten und deren Zusammenspiel eingegangen.

### 5.6.1. C# Plugin API

Die Unity-Engine verwendet als Programmiersprache C#. C# ist eine der Programmiersprachen, die Teil des .NET-Frameworks[86] von Microsoft sind. C#-Programme werden in *IL-Zwischencode* kompiliert, die von der *Common Language Runtime* (CLR), ausgeführt wird. Ein Merkmal der .NET-Sprachen ist die Verwendung von verwaltetem Code. In verwaltetem Code übernimmt die CLR die Verantwortung für die Verwaltung des Speichers und anderer Ressourcen der Programme. Diese „Verwaltung“ kann die Kontrolle der Lebensdauer von Objekten, die Garbage-Collection, erweiterte Debugging-Funktionen usw. umfassen. Im Gegensatz dazu weiß das Laufzeitsystem bei nicht verwaltetem Code wie bei C++ wenig über den vom Programm verwendeten Speicher und die Ressourcen und kann nur minimale Dienste anbieten. Es liegt in der Verantwortung des Programms bzw. des Programmierers, diese Objekte und Ressourcen zu verwalten. Eine der Schwierigkeiten bei Interoperabilität zwischen C# und C++ ist, die Daten über die Grenze zwischen verwaltetem und nicht verwaltetem Code hinweg zu bewegen. Dieser Prozess ist als *Marshaling* bekannt. Das .NET Framework bietet verschiedene Möglichkeiten, Interoperabilität zu erreichen. Eine dieser Möglichkeiten ist die Nutzung von *Platform Invocation* (abk. *P/Invoke*, deu. *Plattformaufruf*)[87]. P/Invoke ermöglicht es in verwaltetem Code, native, nicht verwaltete Funktionen aufzurufen, die als geteilte Bibliothek (Windows: DLL, Android: SO) kompiliert wurden. Diese Methode ist ideal, wenn API-ähnliche Funktionen existieren, die in C oder C++ geschrieben wurden und auf die von einem C#-Programm aus zugegriffen werden muss. Abbildung 5.18 zeigt eine Verbildlichung des P/Invoke Mechanismus. Um die Funktionen der *Core*-Bibliothek in einer geteilten Bibliothek nutzbar zu machen, wurde die komplette API der Facade mit exportierten C-Funktionen umhüllt. Um mittels P/Invoke eine API-Funktion aufzurufen zu können, muss diese in C# deklariert werden. Die zwei zusammengehörenden Deklarationen sehen dann beispielsweise wie folgt aus.

```

1 //Cpp declaration
2 extern "C" COLTK_API int FindAlignmentPoseViaPose(sPose a_pose, sPose&
3           out_result);
4 //C# declaration

```

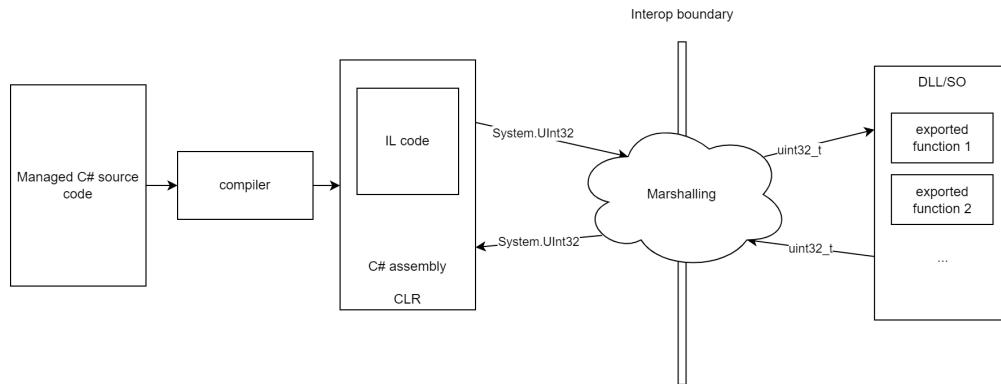


Abbildung 5.18.: Verbildlichung des Marshaling Prozesses bei Nutzung von P/Invoke

```

5 [DllImport("COLTK", CallingConvention = CallingConvention.Cdecl)]
6 public static extern int FindAlignmentPoseViaPose(sPosef a_pose, ref
    sPosef out_result);

```

Bei der Erstellung der C# und *Core*-Bibliothek API wurde sehr darauf geachtet, nur Blittbare-Datentypen zu verwenden. Blittbare-Datentypen sind Datentypen, die in C# und C++ die gleiche Bit-Darstellung haben. Da diese ohne Marshalling als Kopie in beide Richtungen übergeben werden können, entsteht nur wenig Overhead. Strukturen, die nur aus Blittbaren-Datentypen bestehen, sind selbst auch Blittbar. Bei Verwendung als Parameter einer Funktion muss die Struktur auf verwalteter Seite exakt so aufgebaut sein wie auf nicht verwalteter Seite. Die Struktur, welche auf beiden Seiten eine Pose darstellt, sieht beispielsweise wie folgt aus.

```

1 //C++ struct
2 struct sPose {
3     float position[3];
4     float rotation_quat[4];
5 }
6
7 //C# struct
8 [StructLayout(LayoutKind.Sequential)]
9 struct sPosef{
10     sVector3f position; //blittable struct containing three floats
11     sQuaternionf rotation; //blittable struct containing four floats
12 }

```

In den oben gezeigten Strukturen ist gut zu erkennen, dass Blittbare-Typen beliebig kombiniert werden können.

Wird eine Blittbare Struktur durch das *Ref*-Keyword *By-Reference* anstelle *By-Value*

übergeben, wird automatisch, durch das Marshaling, für die Dauer des Funktionsaufrufs nicht verwalteter Speicher alloziert und ein Pointer zu diesem an die API übergeben. Die native Bibliothek kann die Struktur an dieser Speicherstelle mit Daten füllen. Durch das Marshaling werden die Daten nach Funktionsaufruf wieder in verwalteten Speicher übertragen. Dies wird oft verwendet, um Daten in einem *Out*-Parameter an die Engine zurückzugeben.

Für jede API-Funktion der Bibliothek wurde also wie oben beschrieben eine Deklaration in C# erstellt. Außerdem wurden, wie ebenfalls oben beschrieben, für alle Funktionsparameter Strukturen erstellt, die mit den Datentypen auf nativer Ebene kompatibel sind. Zusätzlich wurden die API-Deklarationen auf Engine-Ebene noch einmal mit Funktionen umhüllt, welche ihre Benutzung deutlich handlicher macht, da sie die Engine-Datentypen in die benötigten Strukturen konvertiert.

Mit dieser bis hier her beschriebenen Schnittstelle ist es möglich, Co-Location Anwendungen in der Unity-Engine zu erstellen. Um die Nutzung dieser API zu erleichtern, stellt das Toolkit einige Unity-Komponenten zur Verfügung.

### 5.6.2. Der COLTKManager und Subsysteme

Die **COLTKManager**-Klasse ist eine Unity-Komponente, die auf Render-Engine-Ebene das Herz des Toolkits darstellt. Sie hat die Aufgabe, den Lebenszyklus des Plugins zu verwalten und einen zentralen Zugriffspunkt auf die Funktionalitäten des Plugin zu bieten. Die Funktionalität des nativen Plugins wurde hierfür in vier Sub-Systeme aufgeteilt.

1. **LocalNodeUpdater:** Der **LocalNodeUpdater** hat als einzige Aufgabe, in jedem *Update()*-Loop die Posen aller Objekte, die mit anderen Teilnehmern synchronisiert werden sollen, dem Plugin zu übergeben.
2. **Networking:** Das **Networking**-Subsystem ruft im Update-Loop durch *Polling* alle neu eingegangen Netzwerkevents ab und macht diese durch C#-Events dem Rest der Anwendung zugänglich. Außerdem kann hier die eigene ID und der Verbindungsstatus abgefragt werden.
3. **PlayerManager:** Der **PlayerManager** nutzt die Events des **Networking**-Systems, um einen Container mit verbundenen Spielern zu verwalten. Für jeden Spieler wird hierfür beim Beitritt eine Instanz der **Player**-Klasse erstellt und beim Austritt zur Vernichtung freigegeben. Durch Events, wird es anderen Teilen der Anwendung ermöglicht, auf Bei- und Austritt von Spielern zu reagieren. Beispielsweise können die Events zur Avatar Instanziierung und Zerstörung verwendet werden. Außerdem aktualisiert der **PlayerManager** in seinem Update-Loop die Tracking-Nodes aller verbundenen Spieler, in dem er diese vom nativen Plugin abruft.

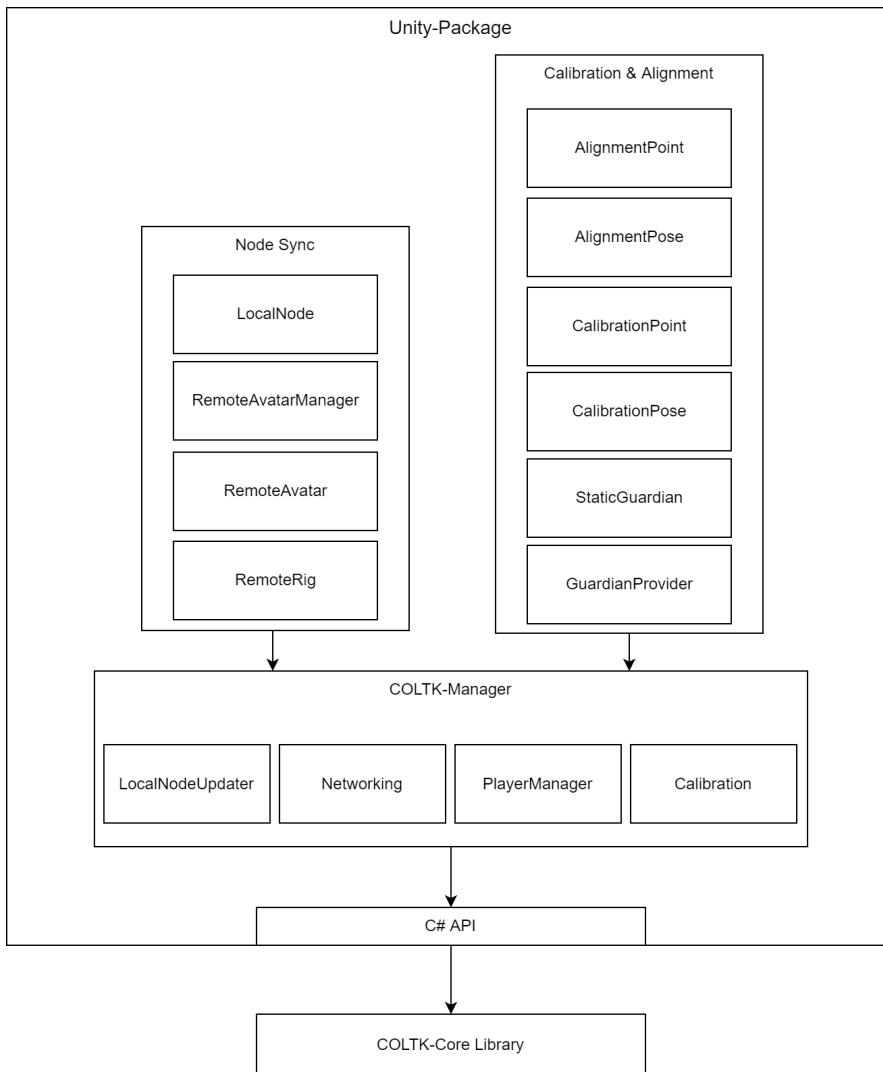


Abbildung 5.19.: Die Architektur des COLTK auf Render-Engine-Ebene.

4. **Calibration:** Das Calibration-Subsystem bietet alle Funktionen, die zur Kalibrierung und Ausrichtung von Frames notwendig sind. Die API des nativen Plugins wird an manchen Stellen vereinfacht, indem von der Unity-Engine-API gebraucht gemacht wird. Beispielsweise werden beim Aufrufen von `TryAutoCalibration()` die benötigten Referenzpunkte etc. selbstständig in der Szene gefunden.

Die Architektur des Toolkits auf Engine-Ebene ist in Abbildung 5.19 zu sehen.

Um den Zugriff auf die Subsysteme von überall Quellcode zu ermöglichen, nutzt der COLTKManager das Singleton-Entwurfsmuster[78]. Der Zugriff auf die Subsysteme kann von jeder Klasse der Anwendung z. B. wie folgt aussehen.

```

1 COLTKManager.instance.Calibrator.TryAutoCalibration();
2 COLTKManager.instance.Networking.ConnectedToServer += OnConnectedToServer;
3 COLTKManager.instance.PlayerManager.PlayerJoined += OnPlayerJoined;
4 COLTKManager.instance.LocalNodeUpdater.SetLocalNodes(nodeArray);

```

### 5.6.3. Calibration & Alignment Komponenten

Um den Kalibrierungs- und Ausrichtungsprozess so einfach wie möglich zu gestalten, werden vom Toolkit für diesen Zweck einige Unity-Komponenten bereitgestellt. Verwendet ein Toolkit-Nutzer diese Komponenten, kann die Kalibrierung und Ausrichtung jeweils auf einen einzigen Funktionsaufruf reduziert werden: `TryAutoCalibration()` und `TryAutoAlignment()`.

#### Pose- und Punkte-Methode

Für die Pose- und Punkte-Methode existiert jeweils ein `Calibration` und ein `Alignment` Skript (siehe Abbildung 5.19). Diese dienen als *Tag*, welcher die Unity-GameObjects markieren, die zur Kalibrierung bzw. Ausrichtung verwendet werden sollen. Will man beispielsweise den rechten Controller als Referenzpose zur Posen-Kalibrierung verwenden, fügt man dem GameObject, das den Controller in der Szene darstellt, das `CalibrationPose`-Skript hinzu. Das `Calibration`-Subsystem findet dieses Skript dann in der Szene und benutzt dessen Pose für die Berechnungen.

#### Guardian-Methode

Die Guardian-Methode zur Kalibrierung und Ausrichtung verwenden, ist etwas aufwendiger. Um diese Methode verwenden zu können, muss der aktuelle Guardian über die entsprechende API abgerufen werden. Diese API ist abhängig vom verwendeten Backend. Außerdem dürfen keinerlei Abhängigkeiten zu anderen Unity-Packages entstehen. Dies wäre der Fall, wenn z. B. direkt die Oculus-API verwendet wird. Um diese Probleme zu lösen, wurde die abstrakte Klasse `AbstractGuardianProver` geschaffen. Ein Entwickler muss lediglich eine erbende Klasse erstellen, welche die Methode `TryGetGuardianPointCloud()` implementiert. In dieser kann dann eine beliebige API verwendet werden, was die Methode mit allen Geräten kompatibel macht, die ein Guarden-Konzept verwenden.

Zur Frameausrichtung kann das `StaticPointCloud`-Skript verwendet werden. Existiert dieses Skript auf einem GameObject in der Szene, kann ein im JSON-Format gespeicherter Guardian visuell dargestellt und als Referenz zur Level-Modellierung genutzt werden.

Diese zwei Skripte werden ebenfalls vom `Calibration`-System gefunden und automatisch zur Kalibrierung und Ausrichtung verwendet.

#### 5.6.4. Synchronisierung der Nodes

Eine Anforderung an das Toolkit ist es, eine Funktionalität zu implementieren, die es ermöglicht, die Transforms von Unity-GameObjects mit den Posen von anderen Teilnehmern zu synchronisieren (siehe 4.4.3). Die Komponenten, die dies ermöglichen, sind in der Architekturabbildung 5.19 als *Node Sync* zusammengefasst. Zunächst ist zu klären, was genau ein Node ist. Ein *Node* setzt sich zusammen aus einer Pose (Position und Orientierung) und einer ID. Die ID gibt an, zu welchem Objekt die dazugehörige Pose gehört. Alle Objekte, die synchronisiert werden sollen, benötigen die `LocalNode`-Komponente. Im Engine-Editor kann jedem Node eine ID zugewiesen werden. Im Realfall würde man diese Komponente z. B. auf die GameObjects für das HMD und den zwei Controllern anfügen. Der `LocalNodeUpdater` kümmert sich dann darum, dass die Posen dieser GameObjects inklusive ID periodisch an das native Plugin übergeben werden.

Auf Empfängerseite müssen diese Nodes nun korrekt interpretiert werden. Hierfür können Entwickler entweder aufbauend auf der `COLTKManager` API ein eigenes Avatar-Management erstellen oder das im Toolkit enthaltene verwenden. Der enthaltene `RemoteAvatarManager` erstellt beim Beitritt eines Spielers einen Avatar und zerstört ihn bei Verlassen der Sitzung wieder. Außerdem übergibt er den Avataren die zugehörige Player-Instanz, mit dem die aktuellen Nodes abgefragt werden können.

Ein Avatar besteht aus zwei Skripten und einer beliebigen Anzahl an Gameobjekten, welche die synchronisierten Nodes darstellen. In Abbildung 6.7 ist der im Toolkit enthaltene Avatar zu sehen. Wie zu erkennen ist, sind die beiden benötigten Komponenten das `RemoteAvatar`- und das `RemoteRig`-Skript. Mit dem `RemoteRig` können empfangene IDs lokalen Unity-Transforms zugeordnet werden. Wird ein Node empfangen, wird durch das Rig die Position und Orientierung der zugeordneten Transform mit der Pose der Node ersetzt. Dieser Ablauf ist vereinfacht in Abbildung 5.20 dargestellt.

Zusammengefasst wird also eine Vielzahl von Systemen und Unity-Komponenten geboten, die die Erstellung synchronisierter Co-Location Anwendungen extrem vereinfachen. Welche Schritte insgesamt nötig sind, wird in 6.2 demonstriert.

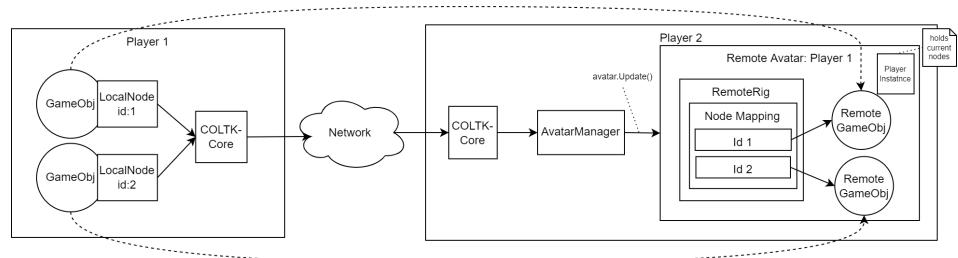


Abbildung 5.20.: Grobe Darstellung der Logik hinter der Synchronisieren von Tracking-Nodes

# 6. Evaluation

In diesem Kapitel werden mehrere Evaluationsergebnisse beschrieben. Zunächst werden die vier im Plugin enthaltenen ICP-Algorithmen miteinander verglichen. Danach wird durch eine „How-To“-Demonstration die Funktionalität des Toolkits evaluiert. Als Nächstes wird eine Methode gezeigt, wie die Anwendbarkeit einer Kalibrationsmethode für einen konkreten Raum bestimmt werden kann und welche Positionsfehler in einer Testumgebung auftraten. Zuletzt werden die Evaluationsergebnisse der Gebrauchstauglichkeit von konkreten Umsetzungen der Kalibrationsmethoden beschrieben.

## 6.1. Vergleich der ICP-Algorithmen

Für das Plugin dieser Arbeit wurden vier Algorithmen zur Punktwolkenregistrierung implementiert. In dieser Sektion werden diese miteinander verglichen, um den am besten geeigneten zu ermitteln.

Die Algorithmen werden anhand von drei Kriterien bewertet:

1. Robustheit: Fällt der Algorithmus in ein lokales Minimum?
2. Geschwindigkeit: Wie viel Zeit benötigt der Algorithmus für eine Registrierung?
3. Genauigkeit: Wie gut wurden die Punktwolken registriert bzw. wie groß ist der endgültige Fehler?

### Datensatz

Für den Vergleich der Algorithmen wurden vier Räume unterschiedlicher Form und Größe gewählt. Für jeden dieser Räume wurden vier Guardians mit unterschiedlicher Ausgangsposen aufgezeichnet. Ein Guardian für jeden der vier Räume ist in Abbildung 6.1a zu sehen. Abbildung 6.1b zeigt die vier unterschiedlichen Guardians für einen Raum. Um die Algorithmen zu bewerten, wird eine Registrierung zwischen allen Guardians eines Raumes durchgeführt. Für die vier Räume wurden also für jeden ICP-Algorithmus insgesamt 24 Registrierungen durchgeführt.



(a) Ein Guardian für jeden Evaluations-Raum, (b) Für jeden Raum wurden vier Guardians auf-dargestellt in Unity

gezeichnet

Abbildung 6.1.: Visualisierung des Datensatzes zum Vergleich der ICP-Algorithmen

### Robustheit

Durch den Brute-Force-Ansatz des Bf- und BfKdTree-Algorithmus fallen diese nicht in ein lokales Minimum und können daher dazu verwendet werden, um zu überprüfen, ob die anderen Algorithmen dies tun. Wie oben beschrieben, wurden zwischen allen Guardians des jeweiligen Raums Punktwolkenregistrierungen durchgeführt. Bei der Auswertung der Registrierungsergebnisse fällt auf, dass einige Guardian-Paare eines Raums der errechnete Fehler deutlich größer ist als bei anderen Guardian-Paaren. Der Fehler ist hierbei definiert als die Summe der quadrierten euklidischen Distanzen zwischen den nächstgelegenen Punkten beider Punktwolken. Das Vorkommen dieser großen Fehler bei Guardian-Paaren liegt daran, dass sich die Punktwolken mancher Paare mehr unterscheiden als andere. Haben die Punktwolken beispielsweise bereits vor der Registrierung in etwa dieselbe Orientierung, ist die Wahrscheinlichkeit, in ein lokales Minimum zu laufen, deutlich geringer. Die Guardian-Paare bei denen mindestens ein Algorithmus nur ein lokales Minimum erreichte, sind in der Tabelle 6.1 aufgelistet. Hierbei sind die Ergebnisse der Algorithmen, die ein globales Minimum erreicht haben, unterstrichen.

In der Tabelle ist zu sehen, dass jedes Mal, wenn der SVD-Algorithmus nur ein lokales Minimum erreichte, dies mit dem Point2Point-Algorithmus ebenfalls so war. Im Gegensatz dazu erreichte der Point2Plane-Algorithmus bei 9 von den 12 Guardian-Paaren das globale

Raum	P	Q	SVD-Fehler	Point-Fehler	Plane-Fehler
1	3	1	22,78	16,86	17,65
	2	1	21,47	15,55	16,63
	1	0	124,58	107,94	20
2	3	0	72,87	51,49	0,27
	3	1	73,89	50,27	0,29
	2	0	73,38	46,39	47,46
	2	1	75,48	44,96	46,16
3	3	0	43,17	31,17	0,19
	3	2	41,78	32,94	0,22
	2	1	39,05	28,03	0,21
	1	0	42,44	29,38	0,15
4	3	2	27,79	24,8	0,3
	2	0	20,74	18,38	0,1
	2	1	20,56	18,31	0,09

Tabelle 6.1.: Registrierungen, die in ein lokales Minimum fielen. Die unterstrichenen Werte sind globale Minima.

Minimum, was daraus schließen lässt, dass der Point2Plane-Algorithmus am robustesten ist.

Auch wenn der Point2Plane-Algorithmus in den meisten Fällen (19 von 24) ein globales Minimum erreichte, ist dies für das Plugin nicht ausreichend, da ein falsches Ergebnis zu Kollisionen von Teilnehmern führen kann. Im Kontext der Punktwolkenregistrierung wird in den meisten Fällen das Problem des lokalen Minimums durch einen *Initial-Guess* gelöst, mit dem die Punktwolken bereits grob angenähert werden.

Um einen *Initial Guess* liefern zu können, wurde der BfKdTree-Algorithmus auf den ersten Schritt reduziert. Die Punktwolke wird zunächst so verschoben, dass ihr Schwerpunkt über dem Schwerpunkt der anderen Wolke liegt. Als Nächstes wird die Wolke in mehreren Schritten rotiert und nach jedem Rotationsschritt der Fehler berechnet. Die Rotation mit dem kleinsten Fehler, zusammen mit der Translation, um den Schwerpunkt zu verschieben, ergeben den *Initial Guess*. Hierbei stellt sich die Frage: Um wie viel Grad muss die Wolke pro Schritt rotiert werden, um ein gutes Ergebnis zu erhalten. Oder anders ausgedrückt: Wie genau muss der *Initial Guess* sein?

Um dies herauszufinden, wurden wie folgt vorgegangen. Für jedes Guardian-Paar eines Raumes wurde erneut der Fehler berechnet, jedoch diesmal unter Verwendung eines *Initial Guess*. Die Schritte für eine komplette Drehung der Wolke wurde hierbei 6-mal verdoppelt. Das bedeutet im ersten Schritt wurde die Wolke zweimal um 180° gedreht,

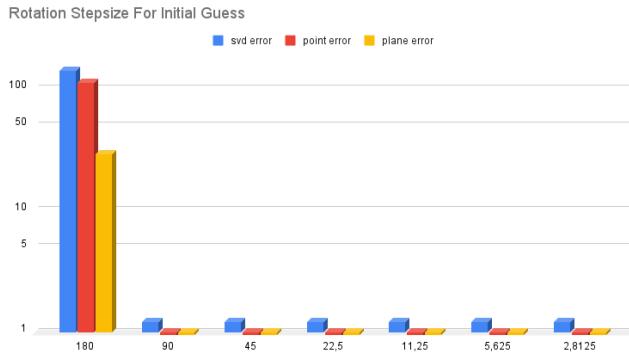


Abbildung 6.2.: Fehler eines Guardian-Paares bei immer kleiner werdendem Rotationswinkel für den *Initial Guess*

im zweiten Schritt 4-mal um  $90^\circ$ , im dritten 8-mal in  $45^\circ$ , usw. Diese Daten wurden dann ausgewertet, um den Winkel zu finden, ab dem alle Paare das globale Minimum erreichen. In der Abbildung 6.2 ist das Ergebnis eines solchen Durchlaufs zu sehen, welcher repräsentativ für alle Guardian-Paare ist. Mit einem Rotationsschritt von  $90^\circ$  erreichten alle ICP-Varianten das globale Minimum. Zur Sicherheit werden im Plugin  $45^\circ$  Schritte verwendet. Durch diesen *Initial Guess* sind also alle ICP-Algorithmen gleich robust. Es muss also anhand der Geschwindigkeit und Genauigkeit entschieden werden, welcher sich am besten für das Plugin eignet.

## Geschwindigkeit

Um zu ermitteln, welcher ICP-Algorithmus die Registrierung am schnellsten durchführt, wurden für jedes der Guardian-Paare der Räume erneut eine Registrierung durchgeführt. Hierbei wurde mithilfe eines Timers [siehe 5.2], die Ausführungszeiten in Mikrosekunden gemessen. Die Messungen wurden mit einem *Intel Core i9-129000K* Prozessor durchgeführt. Bei Ausführung auf einem XR-Gerät wie der *Meta Quest* sind die Ausführungszeiten vermutlich deutlich länger. In Abbildung 6.3 ist die durchschnittliche Dauer der 6-Registrierungen pro Raum zu sehen. Das Diagramm verwendet, um alle Werte darstellen zu können, eine logarithmische Skala. In der Abbildung ist deutlich zu sehen, dass die Brut-Force-Algorithmen auch trotz Optimierung durch einen KD-Baum, deutlich langsamer als die restlichen Algorithmen sind. Der *Point2Plane*-Algorithmus war in Raum-1 beispielsweise ca. 364-Mal schneller als die nicht optimierte Bf-Variante und immer noch ca. 93-Mal schneller als die *BfKdTree*-Variante.

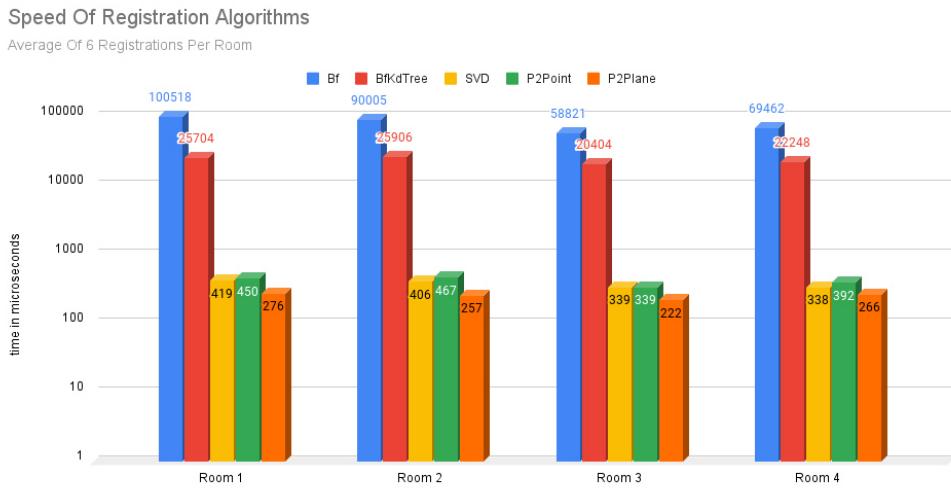


Abbildung 6.3.: Durchschnittliche Ausführungszeit der ICP-Varianten pro Raum in Mikrosekunden.

Der Abbildung ist außerdem zu entnehmen, dass der Point2Plane-Algorithmus bei allen Räumen die schnellste Registrierung durchführte. Auf Platz 2 ist der SVD-Algorithmus, knapp gefolgt vom Point2Point-Algorithmus. Da zwischen dem Rendern zweier Bilder bei Verwendung der *Meta Quest* nur etwa 11 Millisekunden vergehen, sind die Brut-Force-Varianten bei synchroner Verwendung unbrauchbar, da bereits mit dem potenten Evaluationsprozessor ca. 8-Bilder nicht gerendert würden. Dies kann beim Benutzer Motion-Sickness auslösen. Wenn der Bf-Ansatz im Vergleich keine deutlich besseren Registrierungsergebnisse liefert, beschränkt sich die Auswahl also nur noch auf den SVD-, Point2Point- und Point2Plane-Algorithmus.

## Genauigkeit

Das letzte betrachtete Kriterium ist die Genauigkeit. Dies ist das Hauptkriterium, da das Hauptziel ist, dass alle Teilnehmer so exakt wie möglich an ihren realen Positionen angezeigt werden, um z. B. Kollisionen zu vermeiden und eine gute Immersivität zu erreichen.

Zur Bewertung wurden wieder Registrierungen zwischen den vier Guardians jedes Raums durchgeführt. Die durchschnittlichen Fehler dieser sechs Registrierungen pro Raum sind in Abbildung 6.4 zu sehen.

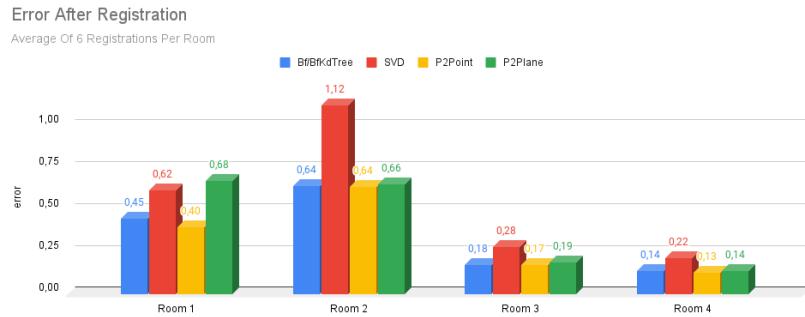


Abbildung 6.4.: Durchschnittlicher Fehler der ICP-Varianten pro Raum in Mikrosekunden.

In den Messdaten ist deutlich zu sehen, dass der Point2Point-Algorithmus nach der Registrierung für jeden Raum den kleinsten Fehler aufweist. Interessanterweise liefern die zwei Brute-Force-Algorithmen für jeden Raum ein besseres Ergebnis als SVD- und Point2Plane. SVD besitzt in den meisten Fällen die schlechteste Genauigkeit. Der Point2Plane-Algorithmus ist in den meisten Fällen minimal schlechter als Point2Point und Bf. Die beste Genauigkeit liefert also der Point2Point-Algorithmus.

### Entscheidung

Mit den eben beschriebenen Ergebnissen kann nun eine finale Wahl getroffen werden. Die Robustheit ist bei der Entscheidung zu vernachlässigen, da durch den *Initial Guess* alle Algorithmen das globale Minimum erreichen. Die Messungen haben ergeben, dass der Point2Plane-Algorithmus am schnellsten konvergiert und der Point2Point-Algorithmus das beste Registrierübungsergebnis liefert. Da ein kleinerer Registrierübungsfehler wichtiger als die Laufzeit ist, wird schlussendlich im Plugin der *Initial Guess* mit dem BfKdTree ermittelt und für die endgültige Transformationsbestimmung der Point2Point-Algorithmus verwendet.

Durch die Verwendung dieses Algorithmus wurde der Laufzeit des Ausgangsalgorithmus Bf um 99,48% reduziert. Das entspricht einer Leistungssteigerung von 19245,02%.

## 6.2. Evaluation durch "How-To" Demonstration

Wie von Ledo et al. in [88] beschrieben, wird im Folgenden eine Toolkitevaluation durch Demonstration durchgeführt.

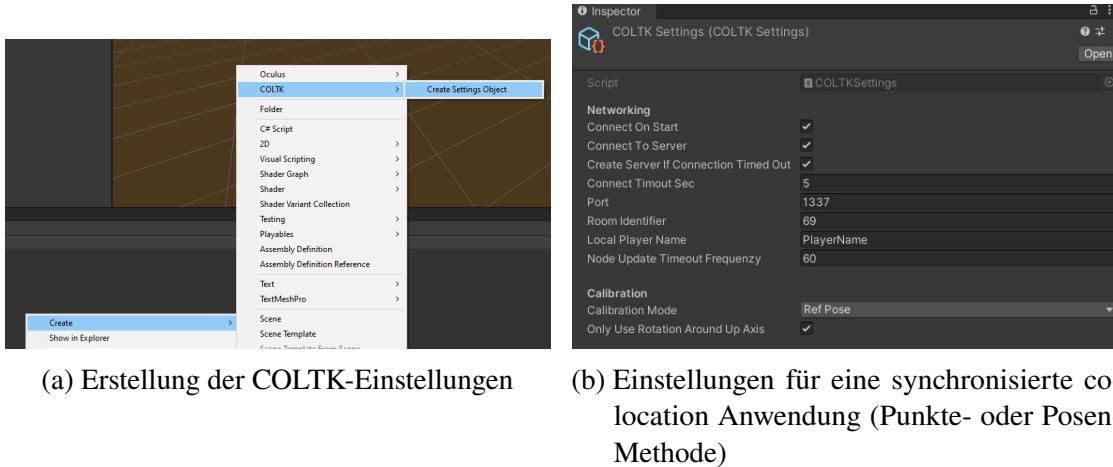


Abbildung 6.5.: Konfiguration der Kalibration und Synchronisation

### 6.2.1. Kalibrierung und Ausrichtung durch die Punkte-Methode

In dieser Demonstration wird gezeigt, wie das Toolkit verwendet werden kann, um eine co-location Anwendung zu erstellen, in der die Punkte-Methode zur Kalibrierung und zur Ausrichtung des Trackingspace verwendet wird. Hierbei wird als XR-Gerät die *Meta Quest* verwendet. Die Controller-Positionen sollen bei Ausführung der Kalibrierung als die zwei Referenzpunkte dienen. Im Folgenden wird nun Schritt für Schritt erklärt, wie dies umgesetzt werden kann. Es wird angenommen, dass das *Oculus Integrationion-Package*[89] und *COLTK*-Package im Unity-Projekt importiert wurden.

#### 1. Kalibrations- und Netzwerkeinstellungen

Als erster Schritt muss ein Settings-Objekt erstellt werden. Im Toolkit werden die Einstellungen als Assets gespeichert. Um ein solches Asset zu erstellen, kann im Asset-Browser durch Rechtsklick ein Menü geöffnet werden. Klickt man den Pfad *Create/COLTK/Create Settings Object* wird das Objekt erstellt (siehe Abbildung 6.5a). Das Einstellungs-Asset bietet diverse Optionen für die Synchronisation und Kalibration. Welche das genau sind, ist in Abbildung 6.5 zu sehen. Damit die Anwendung ad-hoc Netzwerksitzungen erstellt, ist es wichtig, dass im *Networking*-Abschnitt der Einstellungen alle Häkchen gesetzt sind. Um dem Plugin mitzuteilen, dass die Pose- oder Punkte-Methode verwendet wird, muss als *Calibration Mode* „*Ref Pose*“ ausgewählt werden. Welche dieser beiden Methoden am Ende verwendet wird, stellt das *Calibration*-Subsystem automatisch fest.

## 2. COLTKManager-Komponente Anfügen

Als Nächstes muss die **COLTKManager**-Komponente einem GameObject in der Szene hinzugefügt werden. Am besten eignet sich hierfür das Objekt, welches den Trackingspace darstellt, da der Manager automatisch alle LocalNodes in seinen Kind-Objekten findet. Ist der Manager einem Objekt hinzugefügt, muss ihm per drag'n drop, die Transform des Trackingspace und die zuvor erstellten Einstellungen zugewiesen werden. Siehe Abbildung 6.6

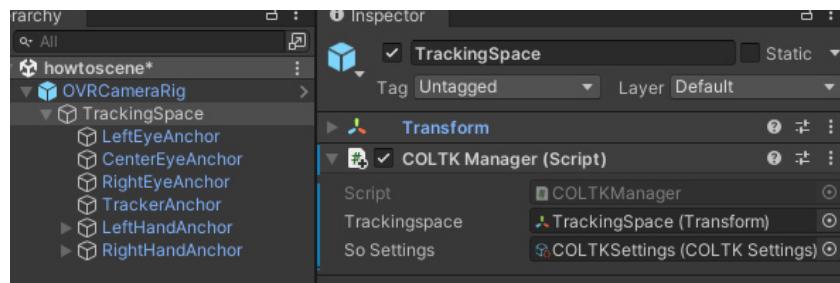


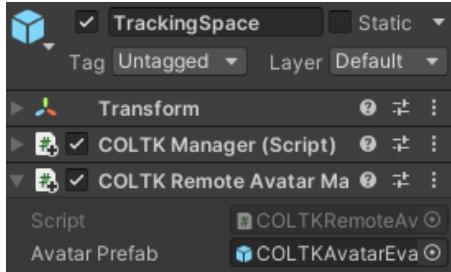
Abbildung 6.6.: Konfiguration des COLTKManager

## 3. RemoteAvatarManager-Komponente anfügen

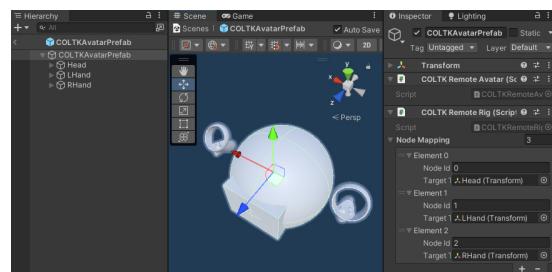
Damit andere Teilnehmer durch einen Avatar dargestellt werden, muss der **RemoteAvatarManager** in der Szene existieren. Dieser kann z. B. demselben Objekt angefügt werden wie der **COLTKManager** (siehe Abbildung 6.7a). Der Avatar-Manager stellt im Editor ein Feld bereit, in dem ihm ein Avatar-Prefab zugewiesen werden kann, welches für jeden beitretenden Spieler instanziert wird. Die einzige Anforderung an das Prefab ist, dass es die **COLTKRemoteAvatar**-Komponente besitzt. Im Toolkit ist der in Abbildung 6.7 zu sehende Avatar enthalten. Dieser kann nach Belieben angepasst werden.

## 4. Referenzpunkte zuweisen

Um die Punkte-Methode verwenden zu können, müssen in der Szene zwei Referenzpunkte definiert sein. Hierfür sollen die Controller der *Quest* verwendet werden. Dies kann umgesetzt werden, indem dem GameObjects der linken und der rechten Hand die **COLTKCalibrationReferencePoint**-Komponente hinzugefügt wird. Wie in Abbildung 6.8 zu sehen, muss hierbei im Editor mit *First* und *Second* die Reihenfolge bestimmt werden, um die Ausrichtung der virtuellen Pose zu definieren.



(a) Anfügen des RemoteAvatarManager



(b) Der vom Toolkit bereitgestellte, modifizierbare Avatar

Abbildung 6.7.: AvatarManager und zugehöriger Avatar

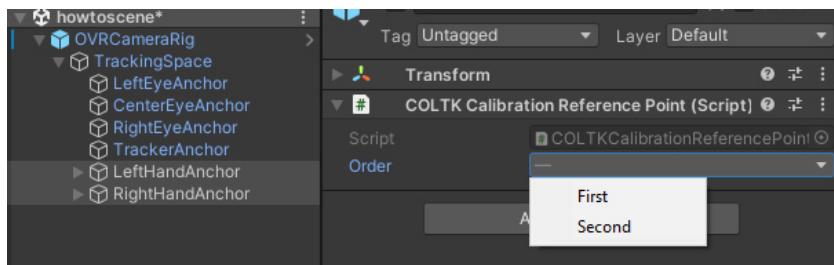


Abbildung 6.8.: Auswahl der Referenzpunkte

Nach diesem Schritt könnte bereits eine Kalibration durchgeführt werden. Für eine Ausrichtung ist jedoch noch ein zusätzlicher Schritt nötig.

## 5. Ausrichtungspunkte erstellen

Damit die Spielwelt so ausgerichtet werden kann, dass sie für alle Teilnehmer konsistent ist, müssen zwei GameObjects erstellt oder gewählt werden, die die Ausrichtungspunkte darstellen (siehe 3.3.1). Hier kann ebenfalls, wie in Abbildung 6.9 zu sehen, eine Reihenfolge festgelegt werden. Wenn dem linken Controller *First* zugewiesen wurde, muss dem Ausrichtungspunkt, der nach der Ausrichtung links sein soll, ebenfalls *First* zugewiesen werden. Ansonsten wäre das Level nach der Ausrichtung um 180° gedreht.

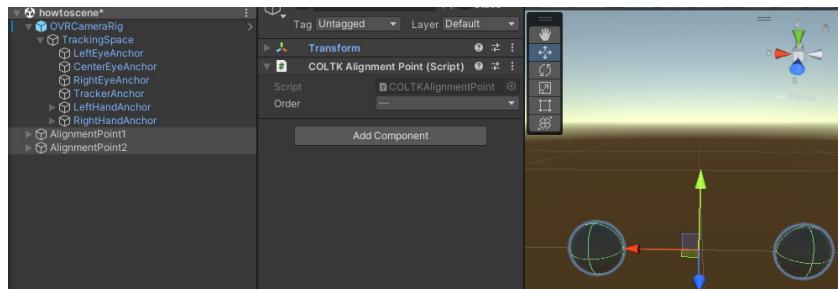


Abbildung 6.9.: Erstellung der Ausrichtungspunkte

### Ausführung der Kalibration und Ausrichtung

Nun ist der Szenen-Setup komplett. Wenn bereit, muss dem Plugin nur noch mitgeteilt werden, dass es eine Kalibration und Ausrichtung durchführen soll. In diesem Beispiel wird dies durch folgenden Code durch das Drücken des A-Knops des rechten Controllers ausgelöst.

```

1 //In Unity Component
2 private void Update() {
3     if (OVRInput.GetDown(OVRInput.Button.One, OVRInput.Controller.RTouch))
4     {
5         if (COLTKManager.instance.Calibration.TryAutoCalibration())
6             Debug.Log("Auto Calibration Success");
7
8         if (COLTKManager.instance.Calibration.TryAutoAlign())
9             Debug.Log("Auto Alignment Success");
10    }
11 }
```

Die Anwendung kann nun kompiliert und auf Geräte übertragen werden. Jeder Teilnehmer muss die zwei Controller auf dieselbe Position in der echten Welt legen und den A-Knopf drücken. Danach sehen sich die Teilnehmer gegenseitig und auch die Spielwelt an der korrekten Position.

#### 6.2.2. Ausrichtung durch die Guardian-Methode

In dieser Demonstration wird gezeigt, wie das Toolkit verwendet werden kann, um eine co-location Anwendung zu erstellen, in der die Guardian-Methode zur Ausrichtung verwendet wird. Es wird nur die Ausrichtung durchgeführt, um zu zeigen, dass es Entwickler auch möglich ist, eigene Netzwerklösungen zu verwenden. Hierbei wird als XR-Gerät ebenfalls die *Meta Quest* verwendet.

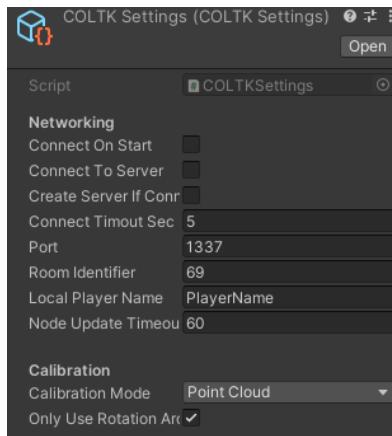


Abbildung 6.10.: Einstellungen für eine Ausrichtung mittels der Guardian-Methode

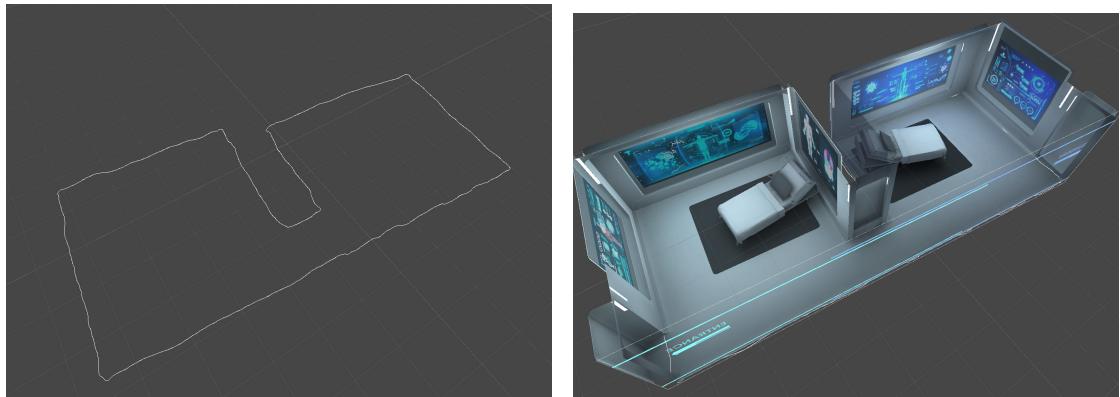
### 1. & 2. Einstellungen und Manager-Komponente

Hier sind die ersten zwei Schritte zusammengefasst, da sie im Grunde dies selben sind wie die in der vorausgegangenen Demonstration. Es muss wieder ein Settings-Objekt erstellt werden und der `COLTKManager` einem `GameObject` hinzugefügt werden. Wie in Abbildung 6.10 zu sehen, besteht der Unterschied zur vorherigen Demonstration darin, dass bei den *Networking*-Einstellungen alle Häkchen entfernt wurden und somit keine Netzwerksitzung aufgebaut wird. Außerdem für die *Calibration Methode* anstelle von *Ref Pose* nun *Point Cloud* ausgewählt.

### 3. Statischen Guardian erstellen

Um den Trackingspace mit der Guardian-Methode ausrichten zu können, wird eine statischer Guardian benötigt. Beim Ausrichten wird der Trackingspace so verschoben, dass der aktuelle Guardian des XR-Geräts und der statische Guardian überlagern. Der statische Guardian kann somit verwendet werden, um relativ zu ihm eine Szene zu modellieren. In Abbildung 6.11 ist dies demonstriert. In 6.11a ist der statische Guardians eines Raums zu sehen und in 6.11b, wie dieser zur Modellierung einer Szene genutzt wurde.

Der statische Guardian kann, wie in Abbildung 6.12 zu sehen, erzeugt werden, indem einem `GameObject` die `COLTKStaticGuardian`-Komponente hinzugefügt wird. Dieses Skript benötigt ein Text-Asset, in dem ein Guardian im JSON-Format gespeichert wurde. Der im Text-Asset enthaltene Guardian wird dann wahlweise im Editor als *Gizmo* angezeigt oder auch zur Laufzeit mittels eines *Linerendereres*. Die `StaticGuardian`-Komponente



(a) Statischer Guardian, dargestellt in der Unity- Engine (b) Relativ zum statischen Guardian modellierte Spielwelt

Abbildung 6.11.: Verwendung des statischen Guardians zur Erstellung einer Umgebung

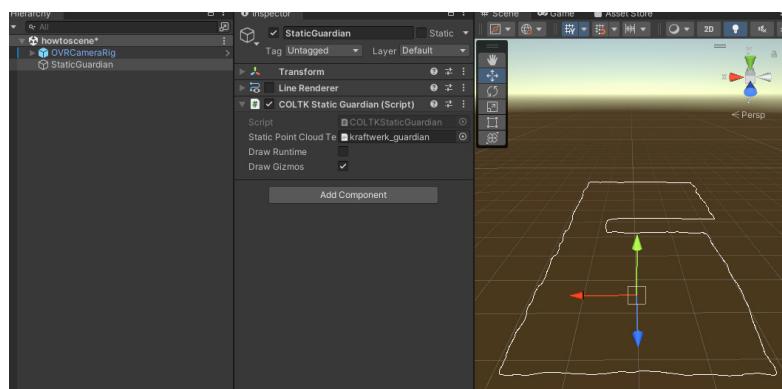


Abbildung 6.12.: Erstellung eines statischen Guardians in der Szene

wird vom Calibrator-Subsystem in der Szene gefunden und zur Ausrichtung verwendet.

#### 4. GuardianProvider bereitstellen

Damit das Toolkit den aktuellen Guardian des XR-Geräts abrufen kann, muss eine von der abstrakten Klasse `COLTKAbstractGuardianProvider` erbende Klasse existieren, welche die gerätespezifische API verwendet, um den Guardian zu erhalten. Für die *Meta Quest* kann diese Klasse wie folgt aussehen.

```

1 public class OVRGuardianProvider : COLTKAbstractGuardianProvider {
2     public override bool TryGetGuardianPointCloud(out Vector3[] pc){
3         pc = OVRManager.boundary.GetGeometry(OVRBoundary.BoundaryType.
4             OuterBoundary);
5         return pc.Length > 0;
6     }

```

Die oben gezeigte Komponente muss auf einem beliebigen GameObjekt in der Szene existieren, um vom Calibration-Subsystem gefunden zu werden.

## 5. Ausführung der Ausrichtung

Wie bei der vorherigen Demonstration muss auch hier die Ausrichtung durch das Calibration-Subsystem gestartet werden. Aber anders als bei der Punkte- und Posen-Kalibration sind alle benötigten Daten bereits beim Start der Anwendung vorhanden. Die Ausrichtung kann, wie in der letzten Demonstration gezeigt, auch über einen Knopfdruck gestartet werden, im Fall der Guardian-Methode gibt es aber auch die Möglichkeit, die Ausrichtung automatisch zu starten. Dies ist im folgenden Pseudocode demonstriert.

```

1 //unity message
2 private IEnumerator Start(){
3     yield return new WaitUntil(() => COLTKManager.instance != null);
4     COLTKManager.instance.Calibration.TryAutoAlign();
5 }

```

Wurden alle bis hierher beschriebenen Schritte ausgeführt und existiert der oben gezeigte Code-Ausschnitt in einer Komponente in der Szene, wird die Ausrichtung beim Starten der App automatisch durchgeführt. Wenn die Kalibrierung bei der Netzwerksynchronisierung auch automatisch ausgeführt werden soll, kann z. B. auf das ConnectedToServer-Event des Networking-Subsystems reagiert werden.

Zusammengefasst wurden in dieser Sektion zwei Anwendungsfälle des Toolkits demonstriert. Hierfür wurde gezeigt, welche Schritte ein Entwickler ausführen muss, um die gewünschte Funktionalität zu erhalten.

## 6.3. Vergleich der Positionsfehler

In diesem Unterkapitel werden die Positionsfehler zwischen zwei verbunden XR-Geräten betrachtet. In 3.2.3 und 3.3.2 wurde bereits der theoretisch mögliche Fehler für die Posen- und Punkte-Methode beschrieben. In der Realität hängt der Fehler jedoch nicht nur von

der menschlichen Ungenauigkeit beim Kalibrierungsprozess ab, sondern auch von der Robustheit des SLAM-Trackings, welches stark von den aktuellen Lichtverhältnissen und dem Aussehen des Raums beeinflusst wird. Außerdem soll auch das Ergebnis der neuen Guardian-Methode mit der Pose- und Punkte-Methode verglichen werden können. Um die Abweichungen zwischen den realen und den virtuellen Positionen zwischen zwei XR-Geräten zu ermitteln, wurde das im Folgenden beschriebene Verfahren entwickelt.

### 6.3.1. Ablauf und Testaufbau

Der Positionsfehler an einer bestimmten Stelle im Raum kann ermittelt werden, indem zunächst ein Tracking-Node eines Nutzers an diese Stelle gelegt und dessen Position aufgezeichnet wird. Danach legt ein zuvor kalibrierter remote Nutzer seinen Tracking-Node an die exakt gleiche Position in der realen Welt. Nun kann auch diese Position im Koordinatensystem des ersten Nutzers aufgezeichnet werden. Die euklidische Distanz zwischen den zwei aufgezeichneten Punkten, ist der Fehler an genau dieser Position im Raum. Um die Tracking-Nodes an die exakt gleiche Position in der echten Welt zu platzieren, wurde die in Abbildung 6.13 gezeigte Markierung entworfen. Diese ermöglichen eine sehr genaue Positionierung der Controller der *Meta Quest*. Die Messungenauigkeiten bei mehrfachen platzieren eines Controllers auf der Markierung lagen im Millimeterbereich. Um also einen Eindruck über den Gesamtfehler zwischen zwei Nutzern zu erhalten, können diese Messungen an mehreren Stellen im Raum stattfinden. Die Ergebnisse können für die Entscheidung für eine geeignete Kalibrationsmethode herangezogen werden.

Für die Messungen wurde in einem geeigneten Raum ein Spielareal von ca. 5 Meter Durchmesser durch Klebeband definiert. In diesem Spielareal wurden neun Markierungen wie in Abbildung 6.14 zu sehen auf dem Boden fixiert. Als Testgeräte wurden zwei *Meta Quests* der ersten Generation verwendet. Angefangen mit der zentralen Markierung wurde abwechselnd ein Controller, der zwei Quests so genau wie möglich auf jede Markierung gelegt und deren Positionen im Koordinatensystem einer der Quests aufgezeichnet. Die Reihenfolge der Aufzeichnungen ist ebenfalls in 6.14 zu sehen. Dies wurde für jede Kalibrationen-Methode zweimal durchgeführt, wobei vor jedem Durchlauf die Geräte neu kalibriert wurden.

### 6.3.2. Ergebnisse

Für stochastische Aussagen sind zwei Messungen pro Methode zu wenig. Die Messungen reichen jedoch aus, um zu beurteilen, ob eine Kalibrationsmethode für einen bestimmten Raum und Spielfeldgröße geeignet ist und um offensichtliche Eigenschaften festzustellen.

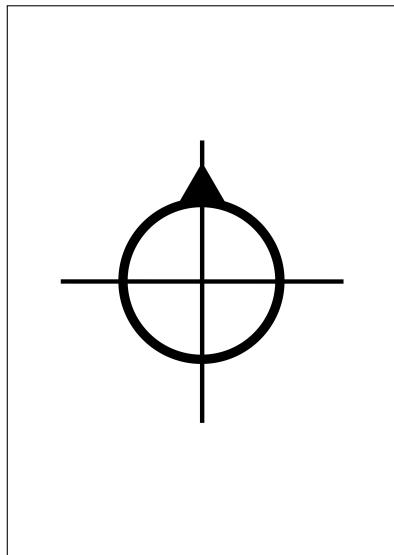


Abbildung 6.13.: Die Markierung, die zum Aufzeichnen der Koordinaten der Tracking-Nodes verwendet wurde.



Abbildung 6.14.: Testaufbau zur Positionsfehlerbestimmung. Die Nummerierung gibt die Reihenfolge bei der Aufzeichnung der Positionen an.

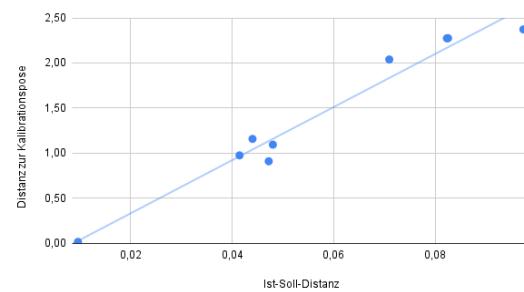
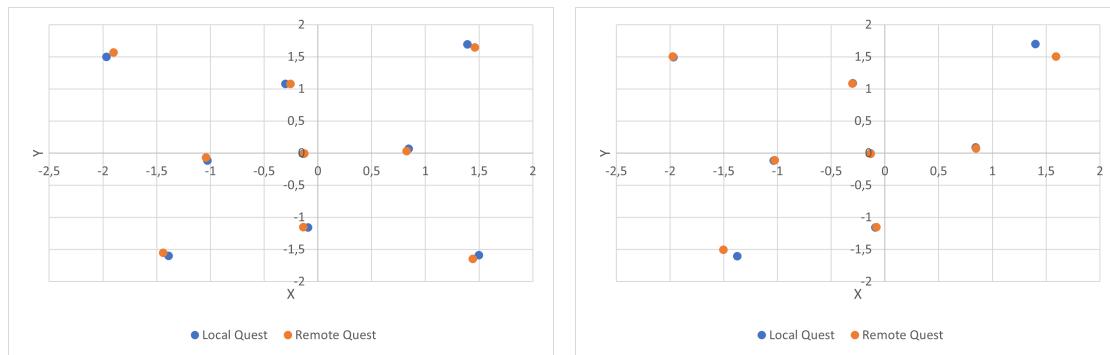
### Posen-Methode

In Abbildung 6.15a ist der erste Durchlauf der Pose-Methode als Streudiagramm dargestellt. Die blauen Punkte repräsentieren in diesem die Soll-Positionen, die durch das lokale Gerät definiert wurden. Die roten Punkte zeigen die Ist-Positionen, die für das remote Gerät aufgezeichnet wurde. Je mehr sich die blauen und roten Punkte überlager, desto kleiner ist der Positionsfehler. Wie in 3.2.3 beschrieben, wird der Fehler mit zunehmender Entfernung zum Kalibrationspose (Mitte des Spielareals) immer größer. In der Abbildung 6.15c ist der Zusammenhang zwischen der Distanz zur Kalibrationspose und der Größe des Fehlers deutlich zu sehen. In diesem Durchlauf lag der größte Fehler bei 9,73 cm und der Durchschnitt aller Fehler bei 5,81 cm.

Im zweiten Durchlauf, dessen Messwerte in 6.15b abgebildet sind, war festzustellen, dass die Abweichungen von Soll- und Ist-Positionen in den meisten Fällen deutlich kleiner als im ersten Durchlauf war. Die verwendeten Kalibrationsposen haben hier also besser übereingestimmt. Was im Diagramm außerdem auffällt, ist, dass es ganz außen zwei Ausreißer gibt. Diese sollte eigentlich nicht auftreten, da der Fehler ausgehend vom Zentrum linear zunehmen müsste. Der Punkt links oben ist aber in etwa so weit vom Zentrum entfernt wie die beiden Ausreißer, hat aber einen deutlich kleineren Fehler. Außerdem waren die Ausreißer die letzten zwei aufgezeichneten Punkte. Das lässt darauf schließen, dass vor dem Aufzeichnen der letzten zwei Punkte eines der Geräte sein internes Koordinatensystem leicht verschoben hat, beispielsweise ausgelöst durch SLAM-Probleme. Deshalb sei an dieser Stelle noch einmal gesagt, dass das der tatsächliche Fehler neben der verwendeten Methode auch stark von der Robustheit des Geräts abhängig ist. Der größte Fehler lag in diesem Durchlauf deswegen bei 27,3 cm und dementsprechend der Durchschnitt aller Fehler bei 5,14 cm.

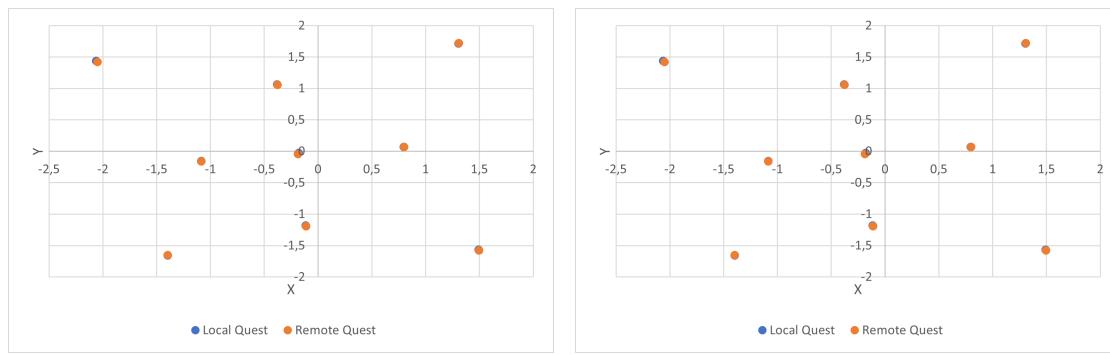
### Punkte-Methode

Bei der Kalibration wurden für diese Methode die zwei Referenzpunkte so weit wie möglich voneinander entfernt platziert, um ein möglichst gutes Kalibrationsergebnis zu erhalten. Die Ergebnisse der beiden Durchläufe sind in Abbildung 6.16 zu sehen. Beim Betrachten der Daten fällt sofort auf, dass nur noch sehr wenig der blauen Punkte zu sehen sind, da sie von den roten Punkten verdeckt werden. Die Soll- und Ist-Position lagen also in beiden Durchläufen sehr nahe beieinander. Genauer gesagt war der größte im ganzen Raum aufgezeichnete Positionsfehler im ersten Durchlauf 3,34cm und im zweiten Durchlauf nur 1,79cm. Die durchschnittlichen Fehler beliefen sich jeweils auf 1,16cm und 0,98cm. Die Punkte-Methode liefert also, wie zu erwarten, auch in der Praxis deutlich bessere Ergebnisse als die Pose-Methode.



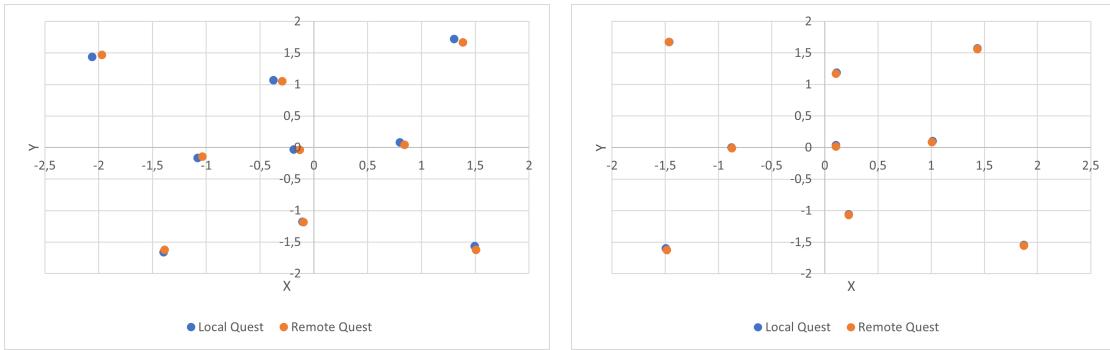
(c) Entfernung von Ist- und Soll-Positionen im Hinblick auf die Entfernung zur Kalibrationspose.

Abbildung 6.15.: Visualisierung der Messdaten der Posen-Methode



(a) Visuelle Darstellung der Lokalen- und Remote-Positionen des ersten Durchlaufs.  
(b) Visuelle Darstellung der Lokalen- und Remote-Positionen des zweiten Durchlaufs.

Abbildung 6.16.: Visualisierung der Messdaten der Punkte-Methode



(a) Visuelle Darstellung der Lokalen- und (b) Visuelle Darstellung der Lokalen- und  
Remote-Positionen des ersten Durchlaufs. Remote-Positionen des zweiten Durchlaufs.

Abbildung 6.17.: Visualisierung der Messdaten der Guardian-Methode

### Guardian-Methode

Bei der Auswertung der zwei Durchläufe für die Guardian-Methode fällt auf, dass die Ergebnisse der jeweiligen Durchläufe sich stark unterscheiden. Betrachtet man die bildlich dargestellten Messdaten in Abbildung 6.17a, stellt man fest, dass es starke Abweichungen gab. Es scheint so, als seien die Punkte des Remote-Geräts insgesamt etwas im Uhrzeigersinn rotiert. Im Gegensatz dazu überlagern sich bei der bildlichen Darstellung (Abbildung 6.17b) des zweiten Durchlaufs alle Punkte. Der durchschnittliche Fehler war beim ersten Durchlauf mit 6,03 cm sogar größer, als bei der Posen-Methode und im zweiten Durchlauf mit 1,42 cm annähernd so klein wie bei der Punkte-Methode. Da die Transformation zwischen den Geräten mittels Punktwolkeregistrierung berechnet wird, liegt die starke Abweichung im ersten Durchlauf vermutlich an einer unsauber aufgezeichneten Punktwolke. Nach jedem Durchlauf wurde der Guardian neu aufgezeichnet, weshalb vermutlich im zweiten Durchgang mit zwei sauber gezeichneter Guardians die besseren Ergebnisse erzielt wurden. Der Positionsfehler ist also sehr stark abhängig davon, wie viel Mühe sich bei der Guardianerstellung gegeben wird.

### Zusammenfassung

In diesem Unterkapitel wurde zunächst eine Vorgehensweise beschrieben, mit der für jeden Raum individuell festgestellt werden kann, welche Kalibrationsmethode geeignet ist. Mit dieser Vorgehensweise wurden alle drei Methoden unter Realwelt Bedingungen zweimal auf Positionsfehler an festgelegten Punkten überprüft. Die wichtigsten Ergebnisse sind zusammengefasst der Tabelle 6.2 zu entnehmen. Hierbei ist für jede Methode pro

Methode	Durchl.	Min.	Max.	Durchschnitt
Pose	1	0,96 cm	9,73 cm	5,81 cm
	2	0,27 cm	27,30 cm	5,14 cm
Punkte	1	0,06 cm	3,34 cm	1,16 cm
	2	0,34 cm	1,79 cm	0,98 cm
Guardian	1	1,13 cm	9,75 cm	6,03 cm
	2	0,41 cm	2,7 cm	1,42 cm

Tabelle 6.2.: Die kleinste, größte und die durchschnittliche Distanz zwischen Soll- und Ist-Positionen aller Aufnahmen eines Durchlaufs. Raumdurchmesser: ca. 5 m

Durchlauf der minimale, maximale und der durchschnittliche Fehler aller betrachteten Punkte gegeben.

In der Tabelle 6.2 ist leicht zu erkennen, dass bei den durchgeföhrten Messungen ganz klar die Punkte-Methode die besten Ergebnisse lieferte. Die Guardian-Methode kann bei sauber gezeichnetem Guardian auch ähnlich gute Ergebnisse liefern. Wenn man den unsauber gezeichneten Guardian außer Acht lässt, liefert die Pose-Methode, wie zu erwarten, die schlechtesten Ergebnisse. Bei der Wahl der Kalibrationsmethode sollte auf den maximal auftretenden Fehler geachtet werden, um zu verhindern, dass Nutzer zusammenstoßen.

## 6.4. Usability der Kalibrationsmethoden

In diesem Unterkapitel wird die Gebrauchstauglichkeit (eng. Usability) der drei Kalibrationsmethoden behandelt. Es gibt eine Vielzahl an Möglichkeiten, die vom Toolkit bereitgestellten Komponenten zu verwenden, um die Methoden konkret umzusetzen. Für die Pose- und Punkt-Kalibration kann beispielsweise der Controller, die getrackten Hände, ein visueller Marker oder auch das XR-Gerät selbst als Referenzpose/Referenzpunkte verwendet werden. Um die Gebrauchstauglichkeit bewerten zu können, wurde für jede der Kalibrationsmethoden eine konkrete Umsetzung für die *Meta Quest* implementiert:

1. Posen-Methode: Die Referenzpose ist die Pose des rechten Controllers.
2. Punkte-Methode: Die Referenzpunkte sind die Positionen des linken und rechten Controller.
3. Guardian-Methode: Die Oculus Integration API wird verwendet, um den Guardian bereitzustellen.

Bei jeder Methode wird die Kalibration durch das Drücken der Y-Taste des linken Controllers ausgelöst.

#### 6.4.1. Testablauf und Aufbau

Für jede Methode erhielten die Tester verschiedene Aufgaben. Der hierbei verwendete Testleitfaden mit den einzelnen Aufgaben und Fragen ist im Anhang A.1 zu finden.

Jedem der Tester wurde die Funktionsweise jeder Methode einmal verbal erklärt. Zusammengefasst bestanden die Aufgaben darin, ein Quest-Paar zweimal mittels jeder Methode zu kalibrieren. Dabei wurden die Tester angewiesen, ihre Gedanken laut zu äußern, um positive und negative Aspekte der jeweiligen Methode festzustellen (Qualitative Datenerhebung). Nach dem Abschluss jeder Methode beantworteten die Tester einige methodenspezifische Fragen und erhielten einen *System Usability Scale*-Fragebogen zum Ausfüllen. Nach Abschluss aller Methoden wurden noch methodenübergreifende Fragen gestellt.

Für die Tests wurden wie in Abbildung 6.14 eine Spielraumbegrenzung abgeklebt und die in Abbildung 6.13 dargestellten Markierungen zu Verfügung gestellt. Die Tests wurden mit insgesamt fünf Teilnehmern durchgeführt. Diese Anzahl reicht aus, um die meisten Gebrauchstauglichkeitsprobleme zu finden[90]. Die Reihenfolge der Methoden wurde zufällig ausgewählt.

#### 6.4.2. Ergebnisse

Die Ergebnisse der Fragebögen und die Antworten der befragten Tester befinden sich im Anhang A.2

Bevor konkret die Usability der einzelnen Methoden betrachtet wird, wird ein Problem behandelt, welches sich durch alle Methoden zog. Hierbei handelt es sich um die Anleitung zur jeweiligen Methode. Wie bereits erwähnt, wurden die Schritte, die zur Kalibrierung notwendig sind, den Testern lediglich verbal mitgeteilt. In einer echten Anwendung könnte man die einzelnen Schritte durch ein Userinterface visualisieren und Schritt für Schritt durch den Prozess leiten. Es wurde jedoch eine verbale Anleitung gewählt, um zu verhindern, dass die Tester das UI bewerten und nicht die Methode selbst.

Es stellte sich heraus, dass es für die Teilnehmer trotz Anleitung viele offene Fragen gab. Beispielsweise die Position der Markierungen im Raum. Die meisten Tester waren zu Beginn sehr unsicher, wo sie diese platzieren sollen und wünschten sich daher in der Nachbefragung mehr Details. Ein anderes Beispiel ist die Reihenfolge der Controller bei der Punkte-Methode. Einige Tester wussten nicht intuitiv, dass die Controller beider

Quest-Geräte die gleiche Reihenfolge haben müssen. Manche Tester war es nicht einmal klar, dass die Markierungen, während der Kalibration, nicht bewegt werden dürfen. Außerdem teilten die meisten Tester mit, dass sie sich nicht nur genauere Informationen darüber wünschen, wo sie die Markierungen platzieren sollen, sondern auch, warum sie dies tun sollen. Sie wollten also wissen, was der technische Hintergrund bei diesem Vorgehen ist. Beim Guardian gab es ebenfalls Beschwerden zur Anleitung. Die meisten Benutzer wussten nicht, dass sie sich bei der Guardianerstellung bewegen dürfen und zeichneten ihn deshalb von der Mitte des Raumes aus. Dies führte zu ungenauerer Kalibrierungsergebnissen.

Für eine echte Anwendung sollte also darauf geachtet werden, alle nötigen Schritte für die verwendete Methode im Detail zu beschreiben, um keine Fragen offenzulassen.

Die Markierungen der Posen- und Punkte-Methode waren ebenfalls ein großer Kritikpunkt der Tester. Da diese aus Papier waren und nur auf den Boden gelegt wurden, verrutschten sie oftmals. Außerdem nahm die genaue Positionierung der Controller auf den Markierungen viel Zeit und Mühe in Anspruch. In der Nachbefragung schlugen einige Testpersonen deshalb vor, eine Art Sockel zu verwenden, der fest im Raum angebracht wird.

Ein weiterer großer Kritikpunkt, den Posen- und Punkte-Methode gemeinsam haben, ist das Problem mit dem VR-Headset. Um die Controller genau auf die Markierung zu legen, mussten die Testpersonen das Headset abnehmen. Manche hoben das Gerät nur etwas an, um durch den Spalt die reale Welt sehen zu können, andere legten es auf einem nahe gelegenen Tisch ab. Insgesamt war dies für die Testpersonen sehr umständlich.

Für die Punkte- und Posen-Methode, so wie sie in für diese Evaluation umgesetzt wurde, müssen die Controller auf Markierungen auf den Boden gelegt werden. Dies kann für körperlich eingeschränkte Personen ein Problem sein. Außerdem könnte das knien auf den Boden manchen Menschen aus anderen Gründen unangenehm sein. Eine Probandin äußerte diesbezüglich Bedenken.

Um diesen Problemen entgegenzuwirken, könnten die Referenzpunkte auch auf einem Tisch oder Ähnliches platziert werden. Die Höhe dieser Punkte ist bei der Kalibrierung nicht relevant.

### Pose-Methode

Lässt man die bis hierher genannten Probleme außer Acht, fanden die Tester die Pose-Methode allesamt sehr einfach zu verwenden. Auf die Frage, was den Testern an dieser Methode gefiel, gab es beispielsweise folgende Antworten.

„Sehr einfach und schnell. Wenig Schritte müssen ausgeführt werden.“

„Echt unkompliziert, wenn man einmal weiß, wies geht.“

Die anderen Antworten der Tester beinhalteten in etwa die gleiche Aussage. Auch bei der Beobachtung der Tester fiel auf, dass keine der Testpersonen große Probleme bei der Verwendung dieser Methode hatte. Alle von den Testern geäußerten Kritikpunkte sind bereits in den oben genannten enthalten.

### Punkte-Methode

Mit der Punkte-Methode hatten die Probanden eindeutig die meisten Probleme. Die Funktionsweise wurde von den Testern zwar beispielsweise mit „recht simpel“ und „einfach zu verstehen“ beschrieben, die Durchführung jedoch fanden alle Probanden sehr aufwendig. In der Nachbefragung beschrieben sie diese unter anderem als „lästig“, „zeitaufwendig“, „fehleranfällig“ und „nicht intuitiv“. Ein Grund für diese Beurteilung waren wieder die Eigenschaften des XR-Headset. Die meisten Probanden gaben sich viel Mühe, die Controller genau auf die Markierung zu legen. Das Problem hierbei ist aber, dass die *Oculus Quest* automatisch Controller ausschaltet, die länger nicht bewegt wurden. Da alle Probanden die Markierungen weit voneinander entfernt platzierten, kam es also häufig vor, dass wenn Tester die Controller platziert hatten und bereit zur Kalibration waren, einer der beiden Controller bereits nicht mehr verbunden war und somit dieser Schritt wiederholt werden musste.

Eine andere Unannehmlichkeit war das Auslösen der Kalibration durch das Drücken des Y-Knopfes. Bei Drücken des Knopfes musste darauf geachtet werden, den Controller auf der Markierung nicht zu bewegen und gleichzeitig beide Controller im Blickfeld des Headset zu haben. Viele Probanden mussten sich hierfür auf dem Boden unangenehme Posen einnehmen.

### Guardian-Methode

Die Guardian-Methode war ganz klar die beliebteste Methode bei den Testern. Da das Zeichnen eines Guardians sowieso für eine Room-Scale-Anwendung notwendig ist, war der Mehraufwand für diese Methode sehr gering. Dies spiegelte sich auch in der Nachbefragung der Probanden wider. Zwei Zitate aus der Befragung:

„Sehr einfach. Großartig!“

„Sehr einfach, da keine Hilfsmittel notwendig sind.“

Trotzdem gab es vonseiten der Probanden auch negative Anmerkungen und auch Bedenken, die alle mit dem Zeichnen des Guardians zusammenhängen. Manche Tester waren sich unsicher, wie genau sie den Guardian zeichnen müssen, um ein gutes Ergebnis zu erzielen. Außerdem könnte diese Methode auch für Menschen mit Einschränkungen

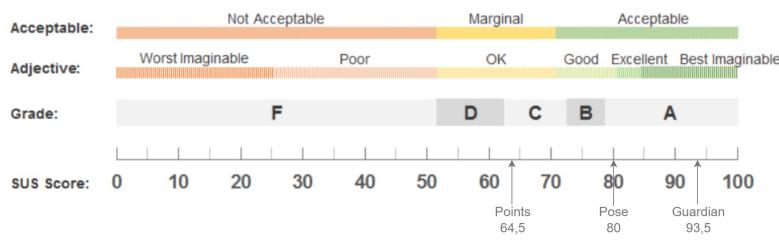


Abbildung 6.18.: Einordnung der SUS-Ergebnisse wie von Sauro[92] vorgeschlagenen.

ungeeignet sein. Zum Beispiel, wenn sie Probleme damit haben, die Hand beim Zeichnen ruhig zu halten.

#### Methodenübergreifend

Nachdem die Tests abgeschlossen waren, wurden die Probanden befragt, welche Methode ihnen persönlich am besten gefiel und welche Methode ihrer Meinung nach in Zukunft am meisten verwendet wird.

Ausnahmslos alle Probanden beantworteten erstere Frage mit der Guardian-Methode. Alle begründeten diese Entscheidung mit dem geringen Aufwand.

Die Antwort auf die zweite Frage war ebenfalls einstimmig die Guardian-Methode. Als Grund hierfür wurde genannt, dass keine zusätzlichen Komponenten benötigt werden, wenig Zeit benötigt wird und dass auch jeder ohne VR-Erfahrung diese Methode umsetzen könnte.

#### System-Usability-Scale Auswertung

Nach der Ausführung jeder Methode erhielten die Probanden einen *System-Usability-Scale*-Fragebogen. Die *SUS* ist ein technologieunabhängiger Fragebogen, um die Gebrauchstauglichkeit eines Systems zu bewerten[91]. Dem Anhang A.3 sind die ausgefüllten Fragebögen und die 10 verwendeten Fragen zu entnehmen.

Die Fragebögen liefern je Methode einen SUS-Wert zwischen 0 und 100. Auch wenn die Ergebnisse auf einer 100-Punkte Skala stehen, sind diese keine Prozentwerte und sollten auch nicht als solche behandelt werden. Jeff Sauro empfiehlt, die SUS-Werte in Quartile einzurunden und stellt hierfür eine Tabelle zur Verfügung[92]. Wie diese Werte in der gesamten Scala eingeordnet werden, ist in 6.18 zu sehen.

Durch die Tabelle von Sauro werden die Ergebnisse der Methoden wie folgt eingeordnet.

Methode	SUS	Note	Adjektiv
Punkte	64,5	C+	Ok
Pose	80	B+	Gut
Guardian	93,5	A	Exzellent

Der SUS-Interpretation spiegelt offensichtlich genau die vorher beschriebenen Beobachtung und Meinungen der Tester wider. Die Methode mit der besten Gebrauchstauglichkeit ist die Guardian-Methode. Die Gebrauchstauglichkeit der Posen-Methode ist im Vergleich immer noch gut und die Punkte-Methode ist ganz klar am unbeliebtesten.

## 6.5. Evaluation der Anforderungen

In diesem Unterkapitel wird betrachtet, inwieweit das Toolkit die Anforderungen aus Kapitel 4 erfüllen.

**4.2.1 Berechnung der Transformation zwischen lokalen und remote Koordinatensystem:** Alle in Kapitel 3 beschriebenen Methoden wurden implementiert und sind mit dem Toolkit nutzbar. Auf nativer Ebene werden die Transformationen durch die `Calibrator`-Klasse (siehe. 5.3) berechnet, indem die benötigten Lokalen- und Remote-Daten übergeben werden. Diese Anforderung wurde also erfüllt.

**4.2.2 Abbildung von (Remote-)Posen in lokales Koordinatensystem:** Sobald die Transformation zwischen zwei Koordinatensystemen berechnet wurde, ist es möglich, durch den `Calibrator` Posen von Remote-Systemen in das lokale System abzubilden. Diese Anforderung wurde also ebenfalls erfüllt.

**4.2.3 Rekalibrierung:** Während der Laufzeit können dem `Calibrator` beliebig oft lokale Daten übermittelt werden. Dieser berechnet automatisch für alle bereits übergebenen remote Daten die neue Transformation. Es ist also möglich, beliebig oft die Kalibration auszuführen und die Anforderung ist somit erfüllt.

**4.2.4 Ausrichtung der Frames (Opt.):** Die in 3.2.1, 3.3.1 und 3.4.1 gezeigten Ausrichtungsmethoden wurden implementiert und sind jederzeit durch die API nutzbar, um eine Ausrichtung durchzuführen. Diese optionale Anforderung ist also erfüllt.

**4.3.1 Austausch von Kalibrations-Daten:** Wie in 5.5 beschrieben, sendet der Client beim Setzen der lokalen Kalibrationsdaten diese an den Server. Dieser wiederum verteilt sie an verbundene und zukünftige Clients. Somit ist diese Anforderung erfüllt.

**4.3.2 Anwendungs- und Raumidentifikation:** Für Netzwerkschicht des Plugins verwendet keine Anwendungs-ID. Stattdessen wird diese Funktionalität implizit durch die

Festlegung des Netzwerk-Ports erreicht. Da dieser Port ohnehin allen Nutzern einer bestimmten Anwendung bekannt sein muss, fungiert diese quasi als Anwendungs-ID. Eine Room-ID wurde mit durch den in 5.4.2 beschriebenen *Identifier* realisiert. Diese ermöglicht die Aufteilung von Nutzern derselben Anwendung in beliebig viele Räume. Die geforderte Funktionalität existiert also, weswegen diese Anforderung als erfüllt betrachtet wird.

**4.3.3 Automatische Teilnehmererkennung:** Durch die in 5.4.2 beschriebenen ServerFinder- und ServerInfoDispatcher-Klassen finden alle Clients automatisch den Server im lokalen Netzwerk, falls dieser existiert. Auch diese Anforderung ist somit erfüllt.

**4.3.4 Automatische Sitzungsregistration:** Neue Clients registrieren sich wie in 5.5 beschrieben bei kompatiblen Servern und somit auch bei allen anderen Clients. Diese Anforderung wurde erfüllt.

**4.3.5 Sitzungsverwaltung:** In einer COLTK\_Server-Instanz werden alle Teilnehmer verwaltet. Sie bietet außerdem Informationen über verbindende, verbindungstrennende und zurzeit verbundene Teilnehmer. Die Anforderung 4.3.5 ist also erfüllt.

**4.3.6 Plattformunabhängigkeit:** Bei der Implementierung des Toolkits wurde darauf geachtet, dass es sowohl für Windows- als auch für Android (Linux) kompilierbar ist. Um die Netzwerknachrichten auch zwischen verschiedenen CPU-Architekturen austauschen zu können, werden alle Nachrichten beim Senden in die Network-Byte-Order und beim Empfang in Host-Byte-Order konvertiert. Diese Anforderung ist also erfüllt.

**4.3.6 Wiederverwendbares Framework (optional):** Wie in 5.4.3 beschrieben, wird durch die *Networking*-Bibliothek ein wiederverwendbares Server-Client-Framework bereitgestellt. Diese optionale Anforderung ist somit erfüllt.

**4.4.1 Unity Package:** Zusammen mit dieser Arbeit wird ein Unity-Package bereitgestellt, welche alle nötigen Unity-Skripte und das für Windows und Android kompilierte native Plugin enthält. Diese Anforderung ist dadurch erfüllt.

**4.4.2 C# API:** Wie in 5.6.1 beschrieben, wurde auf Engine-Ebene eine C# API implementiert, welche die native API des Plugins widerspiegelt. Diese ermöglicht den Zugriff auf sämtliche Co-Location-Funktionalität, womit diese Anforderung erfüllt ist.

**4.4.3 GameObject Synchronisierung:** In Kapitel 5.6.4 ist zu lesen, wie die Synchronisation von Unity-GameObjects zwischen Teilnehmern umgesetzt wurde. Es ist möglich, bis zu 255 empfangene Tracking-Nodes lokalen Unity-Objekten zuzuweisen, dessen Transforms automatisch durch das Plugin aktualisiert werden. Diese Anforderung ist somit erfüllt.

**4.4.4 Netzwerk Events:** Toolkit Nutzer werden durch den COLTK-Manager und dem Networking-Subsystem für jedes Netzwerkevent des nativen Plugins C#-Events bereitgestellt. Diese können von Entwickler zur Umsetzung eines eigenen Sitzungsmanagements verwendet werden. Die letzte Anforderung an das Toolkit ist hiermit also ebenfalls erfüllt.

Wie soeben beschrieben, wurden alle obligatorischen und optionalen Anforderungen somit erfüllt.

# 7. Zusammenfassung und Ausblick

## 7.1. Zusammenfassung

Das Ziel dieser Arbeit war es, das Co-Location-Problem für XR-Geräte mit räumlichem Gedächtnis lösen und diese Lösungen in einem engineunabhängigen Plugin umzusetzen.

In Kapitel 2 wurden zum besseren Verständnis der Arbeit wichtige Grundlagen beschrieben. Es wurden die Trackingverfahren der XR-Geräte beschrieben und geklärt, was unter einer co-located Anwendung zu verstehen ist. Außerdem wurde auf verbreitete Co-Location Lösungen eingegangen und warum diese Lösungen für dieses Toolkit nicht infrage kommen. Hauptgrund hierfür war der untersagte Kamerazugriff mancher VR-Headsets und der benötigte Internetzugriff der SDKs. Da die in der Arbeit vorgestellten Methoden allesamt das Co-Location-Problem mit Transformationsgleichungen löst, wurden in diesem Kapitel auch die wichtigsten mathematischen Grundlagen beschrieben. Es wurde beschrieben, wie Positionen, Orientierungen und Koordinatensystemen dargestellt werden können und welche Notationen in der Arbeit verwendet werden. Außerdem wurde das Konzept eines Trackingspace beschrieben. Zuletzt wurde in diesem Kapitel Literatur gezeigt, welche sich mit einer ähnlichen Problemstellung befassten. Hierbei stellte sich heraus, dass die bisher existierende Lösungen entweder zur Replikation nicht detailliert genug beschrieben wurden oder starke Game-Engine Abhängigkeiten haben. Außerdem nutzte bisher keine der Lösungen die Spielfeldbegrenzung zur Lösung des Problems.

In Kapitel 3 wurden die Methoden beschrieben, mit denen co-located Anwendungen realisiert werden können. Dabei wurde zunächst auf die grundlegende mathematische Problemstellung eingegangen und dann gezeigt, wie diese mit der Pose-, Punkte- und Guardian-Methode gelöst werden kann. Für jede dieser Methoden wurde gezeigt, wie die Transformationen berechnet werden können, mit denen Posen von remote Koordinatensystem in das lokale System abgebildet werden können. Außerdem wurde beschrieben, wie eine Frameausrichtung durchgeführt werden kann, durch die Koordinatensysteme aller Geräte aufeinander ausgerichtet werden können, um die statischen Objekte für alle Teilnehmer konsistent darzustellen. Für die Pose- und Punkte-Methode wurde außerdem beschrieben, wie der Rotationsfehler auf eine Achse beschränkt werden kann. Die

Guardian-Methode ist eine neu entwickelte Methode, die zur Berechnung einer Punktwolkeregistrierung durchführt. Wie diese Registrierung theoretisch mit einem *Iterative Closest Point*-Algorithmus durchgeführt werden kann, wurde ebenfalls beschrieben.

Die Anforderungen an das Toolkit und dem darin enthaltenen Plugin wurden in Kapitel 4 beschrieben. Hierfür wurde zunächst ein Überblick über das Projektziel gegeben und dann im Detail auf die einzelnen Anforderungen eingegangen.

In Kapitel 5 wurde der Entwurf und die Umsetzung des Toolkits beschrieben. Es wurde zunächst ein Überblick über die Gesamtarchitektur gegeben. Hierbei wurden die Abhängigkeiten der einzelnen Bibliotheken und danach auf die einzelnen Komponenten bzw. Bibliotheken eingegangen. Zunächst wurde der Entwurf der Shared-Bibliothek beschrieben, welche Klassen zur Verfügung zu stellen, die in allen anderen nativen Bibliotheken benötigt werden. Danach wurde der Entwurf der Calibration-Bibliothek behandelt, welche Klassen bereitstellt, mit denen die Transformationsberechnungen durch eine einfach zu verwendende API durchgeführt werden können, ohne mathematisches Hintergrundwissen zu benötigen. Hierbei wurden auch die ICP-Algorithmen beschrieben, die für die Punktwolkeregistrierung verwendet werden. Danach wurde der Entwurf und die Umsetzung der Networking-Bibliothek beschrieben, welche ein Server-Client-Framework bereitstellt, das den Verbindungsaufbau und Nachrichtenaustausch zwischen Server und Clients ermöglicht. Beim Entwurf der Core-Bibliothek wurde beschrieben, wie sie das Zusammenspiel zwischen der Networking-Bibliothek und der Calibration-Bibliothek koordiniert und welche API sie der Render-Engine zur Verfügung stellt. Außerdem wurden die konkreten Server und Client Implementationen und der Ablauf des Nachrichtenaustausches erläutert. Zuletzt wurde in diesem Kapitel das Unity-Package und die darin enthaltenen Komponenten beschrieben. Es wurde zunächst erläutert, wie durch P/Invoke die nicht verwalteten Funktionen des nativen Plugins aus verwaltetem C#-Code aufgerufen werden. Danach wurde beschrieben, wie das native Plugin durch den COLTKManager und dessen Subsysteme in der Engine-Ebene verwendet werden kann. Zuletzt wurden noch die Unity-Skripte beschrieben, die eine einfache Kalibrierung, Ausrichtung und Synchronisation der Teilnehmer ermöglichen.

In Kapitel 6 wurden mehrere Aspekte des Toolkits evaluiert. Zunächst wurden die vier implementierten ICP-Algorithmen miteinander verglichen, um den geeigneten zu ermitteln. Hierbei stellte sich heraus, dass der *IcpPoint2Plane* der schnellste und der *IcpPoint2Point* die besten Registrierungsergebnisse liefert. Um zu verhindern, dass die ICP-Algorithmen nur ein lokales Optimum erreichen, wurde eine modifizierte Variante des *IcpBfKdTree* entworfen, um einen *Initial Guess* zu liefern. Hierfür wurde ermittelt, um wie viel Grad die Punktwolke pro Schritt rotiert werden muss, damit der *Initial Guess* gut genug ist, damit die ICPs ein globales Optimum erreichen. Dies war für alle ICP-Varianten 90°. Als entgültiger ICP-Algorithmus wurde der *IcpPoint2Point* mit einem *Initial Guess* durch den *IcpBfKdTree*, da ein kleinerer Fehler höhere Priorität hat als

Konvergenzgeschwindigkeit.

Als Nächstes wurde die Funktionalität es Toolkits durch eine Demonstration evaluiert. Es wurde demonstriert, wie durch einen Unity-Entwickler die Kalibrierung, Ausrichtung und Synchronisierung durch die Punkte-Methode umgesetzt werden kann. Des Weiteren wurde beschrieben, wie ein Entwickler eine Frameausrichtung durch die Guardian-Methode erreichen kann, wodurch alle Hauptfunktionalitäten des Toolkits abgedeckt wurden.

Als Nächstes wurde ein Verfahren vorgestellt, mit dem die Anwendbarkeit einer Methode für einen Raum überprüft werden kann, indem die Positionsfehler an verschiedenen Stellen im Raum gemessen werden. Mit diesem Verfahren wurden die Ergebnisse der drei Kalibrationsmethoden in einer Testumgebung miteinander verglichen. Hierbei stellte sich heraus, dass die Pose-Methode die schlechtesten Ergebnisse lieferte. Die besten Ergebnisse wurden durch die Punkte-Methode erzielt. Die Ergebnisse der Guardian-Methode hingen stark von der Qualität der aufgezeichneten Guardians ab. Bei einem unsauberen Guardian war das Ergebnis vergleichbar schlecht wie bei der Posen-Methode und bei einem sauberen Guardian annähernd so gut wie bei der Punkte-Methode.

Als Nächstes wurde die Gebrauchstauglichkeit konkreter Umsetzungen der drei Kalibrationsmethoden betrachtet. Hierfür führten Probanden mit jeder Methode zweimal eine Kalibration durch, beantworteten Fragen und füllten einen SUS-Fragebogen aus. Hierbei war die Guardian-Methode klar der Favorit aller Probanden und erhielt dementsprechend die SUS-Note *A (Exzellent)*. Die Pose-Methode war wegen der Einfachheit des Verfahrens auch recht beliebt, weshalb sie die Note *B+ (Gut)* erhielt. Die Punkte-Methode war wegen des aufwändigen Kalibrationsverfahrens die unbeliebteste Methode und erhielt deswegen nur die Note *C+ (OK)*. Zuletzt wurden noch evaluiert, wie der Anforderungen aus Kapitel 4 erfüllt wurden. Hierbei wurde auf jede Anforderung eingegangen und beschrieben, warum diese erfüllt ist. Das Ergebnis dieser Evaluation ergab, dass alle obligatorischen und optionalen Anforderungen erfüllt wurden.

## 7.2. Ausblick

Das Toolkit ist im aktuellen Zustand zwar einsatzbereit, kann aber durch weitere Entwicklung noch verbessert werden. Vor einer Veröffentlichung sollten noch folgende Code-Anpassungen durchgeführt werden.

Derzeit werden alle Nachrichten zwischen Server und Clients mittels TCP übertragen. Dies eignet sich gut für Nachrichten, bei denen sichergestellt werden muss, dass sie ankommen, wie z. B. beim Übertragen der Kallibrationsdaten. Für die Übertragung von Tracking-Node-Posen aber ist TCP jedoch nicht optimal. TCP stellt nämlich nicht nur sicher, dass die Nachrichten ankommen, sondern auch, dass dies in der korrekten Reihenfolge geschieht. Das heißt, wenn ein Node-Update-Packet verloren geht und danach mehrere neuere Node-Updates ankommen, wird das verlorene Paket trotzdem

noch einmal übertragen und gewartet bis dies ankommt, obwohl eigentlich nur das neueste Posen-Update relevant ist. Dies könnte durch die Versendung der Nodes über einen UDP-Socket in Verbindung mit einem Timestamp verbessert werden. Da die Nodes mehrfach in der Sekunde gesendet werden, ist es nicht schlimm, wenn manche Pakete verloren gehen. Da UDP nicht die korrekte Reihenfolge der Pakete sicherstellt, können durch einen, in der Nachricht enthaltenen Timestamp ältere Nachrichten bei Ankunft verworfen werden. Generell sollte das Server-Client-Framework ein Konzept von *Reliable*- und *Unreliable*-Nachrichten einführen, mit denen beim Senden einer Nachricht entschieden werden kann, ob die Nachrichtenübertragung garantiert werden muss.

Eine nützliche Erweiterung des Toolkits, wäre die Möglichkeit, die Kalibration aller Teilnehmer durch eine Netzwerknachricht auszulösen. Derzeit kann jeder Klient zu jeder Zeit und beliebig oft eine Kalibration durchführen. Für manche Anwendungsfälle kann es aber nötig sein, die Kalibration für alle Teilnehmer gleichzeitig durchzuführen zu müssen. Ein Beispiel hierfür ist die Punkte-Kalibrierung, wie sie in [48] beschrieben ist. Anstatt XR-Geräte oder Controller als Referenzpunkte zu verwenden, wird in diesem Papier das Hand-Tracking benutzt. Hierfür verbergen alle Teilnehmer bis auf einen die Hände hinter dem Rücken. Die Quest-Geräte tracken dann die Posen der Hände des übrigen Teilnehmers und nutzen diese für alle XR-Geräte als Referenzpunkte. Damit ein gutes Ergebnis erzielt wird, müssen die Posen der Hände möglichst zeitnah von allen Geräten erfasst werden. Diese Erfassung könnte durch einen der Teilnehmer über eine Netzwerknachricht ausgelöst werden.

Bisher ist das Plugin darauf ausgerichtet, nur Tracking-Nodes zu synchronisieren. Jede übertragener Node gehört also zu einem Teilnehmer. Es wäre jedoch nützlich, auch die Synchronisation von *Globalen*-Nodes zu ermöglichen. Das heißt die Synchronisation von Objekten, die keinem speziellen Teilnehmer gehören. Zum Beispiel könnten Teilnehmer so gemeinsam ein 3D-Modell betrachten und dieses auch bewegen und rotieren. Die Funktionalität dies zu erreichen ist mit dem Toolkit bereits gegeben, jedoch sollte eine API hierfür bereitgestellt werden und auf Engine-Ebene Skripte erstellt werden, die die Umsetzung vereinfachen. Außerdem müssten diese Globalen-Nodes auch, um eine Skalierungs-Komponente erweiter werden, um eine umfassende Interaktion zu ermöglichen.

Die nächste nützliche Erweiterung des Toolkits wäre alle Kalibrationsmethoden miteinander kompatibel zu machen. Bisher funktioniert die Synchronisation zwischen Nutzern der Pose- und Punkte-Methode. Die Guardian-Methode ist derzeit aber noch komplett getrennt von den zwei anderen Methoden. Es gibt einige Möglichkeiten, diese Kompatibilität zu erreichen. Hier drei Beispiele:

1. Das Toolkit erlaubt es, beide Methoden zu verwenden. Benutzer der Guardian-Methode könnten beispielsweise zuerst die Guradian-Kalibration durchführen

und dann noch eine Punkte- oder Posen-Kalibration. Mehrere Methoden zur Kalibration zu verwenden, scheint zwar redundant, bildet aber den Grundstein der zwei folgenden Möglichkeiten.

2. Die Pose, an der sich die Referenzpose befinden wird, kann bereits in der Editor-Szene definiert werden, indem man einen statischen Guardian als Referenz verwendet. Nachdem eine Frameausrichtung durchgeführt wurde, kann die definierte Pose zur Posen-Kalibrierung verwendet werden.
3. Bei Durchführung der Guardian-Kalibration kann immer auch eine Posen-Kalibrierung durchgeführt werden, der einen virtuelle Pose durch den Guardian erstellt, z. B. durch den Schwerpunkt und den Eigenvektor der Punktwolke oder auch einfach nach durchgeführter Frameausrichtung, durch die Pose des Trackingspace. Auf diese virtuelle Pose müsste dann für XR-Geräte ohne Guardian eine Markierung zur Posen-Kalibration gelegt werden. Hierfür bieten die *Meta Quests* beispielsweise Pass-Through Funktionalitäten an, mit der virtuelle Objekte in der realen Welt angezeigt werden können.

Für Unity wird mit dieser Arbeit bereits ein Package bereitgestellt, das eine einfache Verwendung des nativen Plugins ermöglicht. Ein nächster Schritt wäre es, das Toolkit auch in die Unreal Engine zu integrieren. In der aktuellen Form kann das Toolkit schon in der Unreal Engine verwendet werden, weil diese ebenfalls C++ verwendet. Es sollte jedoch zur einfacheren Nutzung mehrere Komponenten und Systeme auf Engine-Ebene ähnlich wie im Unity-Packae bereitgestellt werden.

*Meta* bzw. *Oculus* erweitern stetig die Funktionalität und API deren VR-Headsets. Das Oculus-Integration-Package bieten einige Funktionen, die von anderen XR-Headset-Herstellern nicht geboten werden. Vor kurzem erweiterte *Meta* das Packet beispielsweise um *Spatial-Anchors*. Diese Raumanker ermöglichen es Posen im realen Raum festzulegen, die auch nach Neustart der Applikation die gleiche Pose im realen Raum hat, auch wenn sich der Trackingspace verändert hat. Dadurch kann zum Beispiel die Posen- und Punkte-Kalibration verwendet werden ohne die Brille absetzen zu müssen (durch Pass-Through) und muss wie bei der Guardian-Methode nur ein einziges Mal für alle Teilnehmer ausgeführt werden. Um solche und andere spezifischen APIs Entwicklern nutzbar zu machen, könnten daher spezifische Packageges erstellt werden, die auf dem API-unabhängigen COLTK aufbauen und dessen Funktionalität erweitert.

Bisher wurde die Gebrauchstauglichkeit der drei konkret umgesetzten Kalibrationsmethoden evaluiert. Da es sich bei der erstellten Software aber um ein Toolkit handelt, sollte auch dessen Gebrauchstauglichkeit noch bewertet werden. Hierfür können beispielsweise einige Entwickler wie in [88] beschrieben gewisse Aufgaben mit dem Toolkit erledigen und dann ihre Meinung dazu äußern. So lassen sich Probleme mit dem aktuellen Stand des Toolkits und auch neue Anforderungen ermitteln.

## A. Anhang: Usability Evaluation der Methoden

### A.1. Testleitfaden

Der Testleitfaden wurde vom Autor nur verwendet, um die Aufgaben verbal zu kommunizieren. Deshalb wurde nicht auf korrekte Grammatik oder Rechtschreibung geachtet.

## **Testleitfaden COLTK Kalibration - Usability Test**

Projekt: Co-Location Toolkit

### **Testdaten**

Bedingungen: Im Test-Raum, 2 Markierungen für Controller, Abgeklepter Guardian

Testgeräte: 2x Oculus Quest 1

### **Begrüßung |**

Herzlich willkommen zum Usability-Test für die Kalibrationsmethoden des Co-Location Toolkits!

Danke, dass du mitmachst. Ich bin [Name des Testleiters] und werde diesen Test mit dir durchführen. Der Test wird etwa 30 Minuten in Anspruch nehmen.

Die Kalibrationsmethoden, um die es geht, sind Teil eines Plugins für die Unity Engine. Das Plugin soll Entwicklern ermöglichen, sehr einfach VR Anwendungen für einen Shared-Space zu entwickeln. Das heißt mehrere Benutzer können sich gemeinsam in einem Raum befinden und sich in der virtuellen Welt an der Stelle sehen, wo sie sich auch in der realen Welt befinden. Dadurch können Sie gemeinsam die virtuelle Welt erleben, miteinander interagieren und dabei nicht ineinander laufen.

Um dies zu ermöglichen, wurden drei Methoden zur Kalibration der einzelnen VR-Headsets entwickelt.

Um Probleme und positive Aspekte der Methoden herauszufinden, werde ich dir im Laufe des Tests ein paar Aufgaben stellen.

Ich bitte dich, während der gesamten Zeit deine Gedanken und Überlegungen laut zu äußern. Nur dadurch, dass du laut denkst, kann ich den Ursprung eventueller Probleme erfassen.

Sagen mir einfach, was dir bei der Benutzung der Anwendung durch den Kopf geht. Dabei interessiert uns deine ehrliche Meinung. Es gibt kein Richtig oder Falsch!

Hast du noch Fragen?

### **Vorbereitung**

- Guardians Löschen
- Markierungen entfernen (Nicht die für Messungen)
- Erklärung wie man einen Room-Scale Guardian erstellt

## Testszenario

Testszenario	
Teilnehmer	Stell dir vor, du bist bei einem Kunden und möchtest ihm eine Co-Located Anwendung demonstrieren. Hierfür müssen die zwei Quests aufeinander kalibriert und die Anwendung gestartet werden.
[Testleiter]	

### Aufgabenstellung

- Die Reihenfolge der Kalibration muss zufällig sein
- Tester müssen wissen, wie man einen Room-Scale Guardian erstellt
- Tester müssen mit den grundlegenden Bedienelementen vertraut sein
- Leuten erklären, wie die Methode funktioniert (Nötige Schritte)

### Aufgabenstellung 1: Posen Kalibration

Aufgabenstellung 1.0: Oculus Quest Setup	
Teilnehmer	Konfiguriere für die Quest einen Room-Scale Guardian.
[Testleiter]	<ul style="list-style-type: none"> <li>• Zeit</li> </ul>

#### 1-Pose Methode:

- An dieser Stelle müssen alle Schritte der Methode einmal erklärt werden

Aufgabenstellung 1.1: Platzierung der Markierung	
Teilnehmer	Du willst nun die zwei Headsets mithilfe der 1-Posen Kalibrierung kalibrieren. Hierfür muss zunächst ein Kalibrierungspunkt im Zentrum des Raums markiert werden. Nimm eine Markierung und platziere ihn im Raum.
[Testleiter]	<ul style="list-style-type: none"> <li>• Zeit</li> <li>• Haben Tester körperliche Probleme?</li> <li>• Was wird währenddessen mit dem Headset gemacht?</li> <li>• Wo platziert der Tester den Marker?</li> <li>• Wie sicher ist der Tester beim Platzieren der Marker?</li> </ul>

- Applikation “Posen Kalibrierung” starten

Aufgabenstellung 1.2: Kalibration der ersten Quest	
<b>Teilnehmer</b>	Nachdem der Marker platziert wurde, willst du die erste Quest kalibrieren. Lege hierfür den rechten Controller mit dem Ring nach unten auf die Markierung. Achte hierbei darauf, dass der Controller in die gekennzeichnete Richtung zeigt. Je genauer du auf die Position und Rotation des Controllers achtest, desto besser wird das Ergebnis sein.
<b>[Testleiter]</b>	<ul style="list-style-type: none"> <li>• Zeit</li> <li>• Wie viel wert wird auf Genauigkeit gegeben</li> <li>• Haben Tester körperliche Probleme?</li> <li>• Was wird währenddessen mit dem Headset gemacht?</li> </ul>

Aufgabenstellung 1.3: Kalibration der zweiten Quest	
<b>Teilnehmer</b>	Jetzt willst du auch die zweite Quest kalibrieren. Gehe hierfür genauso vor wie im letzten Schritt.
<b>[Testleiter]</b>	<ul style="list-style-type: none"> <li>• Zeit (Schneller?)</li> <li>• Wie viel wert wird auf Genauigkeit gegeben</li> <li>• Geht es schneller?</li> <li>• Sicherer?</li> </ul>

Aufgabenstellung 1.4: Neu Kalibrierung beider Quests	
<b>Teilnehmer</b>	Der Kunde und du kommen gerade aus einer Pause und wollt noch einmal die Szene betrachten. Kalibriere hierfür beide Geräte neu
<b>[Testleiter]</b>	<ul style="list-style-type: none"> <li>• Zeit (Schneller?)</li> <li>• Wie viel wert wird auf Genauigkeit gegeben</li> <li>• Geht es schneller?</li> <li>• Sicherer?</li> </ul>

## Aufgabenstellung 2: Punkte Kalibration

Aufgabenstellung 2.0: Oculus Quest Setup	
Teilnehmer	Konfiguriere für die Quest einen Room-Scale Guardian.
[Testleiter]	<ul style="list-style-type: none"><li>• Zeit messen (Um zu schauen, obs später schneller geht)</li><li>• Wird der Tester sicherer? (falls nicht erste methode)</li></ul>

### Punkte Methode:

- An dieser Stelle müssen alle Schritte der Methode einmal erklärt werden

Aufgabenstellung 2.1: Platzierung der Markierungen	
Teilnehmer	Du willst nun die zwei Headsets mithilfe der Punkte Kalibrierung kalibrieren. Hierfür müssen zwei Markierungen im Raum platziert werden. Zwischen den beiden Markierungen wird das Zentrum der Szene befinden. Je weiter die Marker voneinander entfernt sind, desto genauer wird die Kalibrierung
[Testleiter]	<ul style="list-style-type: none"><li>• Zeit</li><li>• Haben Tester körperliche Probleme?</li><li>• Was wird währenddessen mit dem Headset gemacht?</li><li>• Wo platziert der Tester den Marker?</li><li>• Wie sicher ist der Tester beim Platzieren der Marker?</li></ul>

- Applikation “Punkte Kalibrierung” starten

Aufgabenstellung 2.2: Kalibration der ersten Quest	
<b>Teilnehmer</b>	Nachdem die beiden Marker platziert wurden, willst du die erste Quest kalibrieren. Lege hierfür deine beiden Controller mit dem Ring nach unten auf die platzierten Marker. Je genauer die Controller auf den Markierungen liegen, desto besser wird das Ergebnis.
[Testleiter]	<ul style="list-style-type: none"> <li>• Zeit</li> <li>• Wie viel wert wird auf Genauigkeit gegeben</li> <li>• Haben Tester körperliche Probleme?</li> <li>• Was wird währenddessen mit dem Headset gemacht?</li> </ul>

Aufgabenstellung 2.3: Kalibration der zweiten Quest	
<b>Teilnehmer</b>	Jetzt willst du auch die zweite Quest kalibrieren. Gehe hierfür genauso vor wie im letzten Schritt.
[Testleiter]	<ul style="list-style-type: none"> <li>• Zeit (Schneller?)</li> <li>• Wie viel wert wird auf Genauigkeit gegeben</li> <li>• Geht es schneller?</li> <li>• Sicherer?</li> </ul>

Aufgabenstellung 2.4: Neu Kalibrierung beider Quests	
<b>Teilnehmer</b>	Der Kunde und du kommen gerade aus einer Pause und wollt noch einmal die Szene betrachten. Kalibriere hierfür beide Geräte neu
[Testleiter]	<ul style="list-style-type: none"> <li>• Zeit (Schneller?)</li> <li>• Wie viel wert wird auf Genauigkeit gegeben</li> <li>• Geht es schneller?</li> <li>• Sicherer?</li> </ul>

### Aufgabenstellung 3: Guardian Kalibrierung

- An dieser Stelle müssen alle Schritte der Methode einmal erklärt werden

Aufgabenstellung 3.1: Setup der ersten Quest	
Teilnehmer	Du willst die zwei Quests mithilfe der Guardian Kalibrierungsmethode kalibrieren. Erstelle hierfür zunächst einen Room-Scale Guardian. Achte beim Zeichnen darauf, dass du die Form ein zweites Mal exakt nochmal nachzeichnen kannst. Je genauer du den Guardian zeichnest, desto besser wird das Kalibrations-Ergebnis
[Testleiter]	<ul style="list-style-type: none"><li>• Zeit</li><li>• Wie genau ist der Tester beim Zeichnen</li><li>• Probleme?</li><li>• Ist klar, wo der Guardian gezeichnet werden soll?</li></ul>

Aufgabenstellung 3.1: Setup der zweiten Quest	
Teilnehmer	Nachdem du die für die erste Quest einen Room-Scale Guardian erstellt hast, willst du nun das gleiche für die zweite Quest tun. Versuche hierbei den Guardian genauso zu zeichnen wie bei der ersten Quest. Hierbei gilt auch: Je sauberer gezeichnet wird, desto besser wird das Ergebnis.
[Testleiter]	<ul style="list-style-type: none"><li>• Zeit</li><li>• Wie genau ist der Tester beim Zeichnen (Beim zweiten mal ungenauer?)</li><li>• Probleme?</li><li>• Ist klar, wo der Guardian gezeichnet werden soll?</li></ul>

<b>Aufgabenstellung 3.2: Kalibration in der App</b>	
<b>Teilnehmer</b>	Um die Kalibration der zwei Quests zu starten, öffne die Applikation "Guardian Methode" auf beiden Geräten.
<b>[Testleiter]</b>	<ul style="list-style-type: none"> <li>• Reaktion der Tester</li> </ul>

<b>Aufgabenstellung 1.4: Neu Kalibrierung beider Quests</b>	
<b>Teilnehmer</b>	Der Kunde und du kommen gerade aus einer Pause und wollt noch einmal die Szene betrachten. Kalibriere hierfür beide Geräte neu, indem du die Anwendung auf beiden Geräten neu startest.
<b>[Testleiter]</b>	<ul style="list-style-type: none"> <li>• Reaktion der Tester</li> </ul>

#### **Nachbefragung (Nach jeder Methode)**

- Was fandest du bei der Benutzung dieser Methode gut?
- Was fandest du bei der Benutzung dieser Methode schlecht?
- Kannst du dir Einsatzmöglichkeiten vorstellen, für die diese Methode besonders gut geeignet ist?
- Gibt es noch etwas, was du über diese Methode sagen möchtest?

#### **Nachbefragung (Ende)**

- Welche der Methoden würdest du am liebsten verwenden?
- Welche der Methoden wird in Zukunft wohl am meisten verwendet und warum?
- Gibt es noch etwas, was du über das Toolkit sagen möchtest?



## A.2. Antworten der Tester bei der Befragung

## Usability Evaluation Befragung (Antworten in gekürzter Form):

### **Pose**

#### Gut:

- "Echt unkompliziert, wenn man einmal weiß, wies geht. Man muss nur eine Taste drücken"
- „Sehr einfach und schnell. wenig schritte müssen ausgeführt werden“
- „Einfach zu verstehen“
- "Einfach"

#### Schlecht:

- "UI wäre cool"
- "schwer die Controller genau aufeinander zu stellen"
- „Brille auf und absetzen nervt“
- „Ungenaue Ergebnisse“

#### Sonstiges:

- „Infos wo der Punkt sein soll und warum wären nützlich“
- Vorschlag: „Papier durch fixe Positionen oder Sockel austauschen“
- "Fände es cool, wenn man Markierungen durch Kamera sehen könnte"

### **Punkte**

#### Gut:

- „Easy to use. Instant Resultat is nice. man sieht gleich ob es funktioniert hat.“ findet gut, dass man sich ausbessern kann, ohne die App zu schließen
- "Recht simple, aber erstes mal braucht man klare Anweisung. Sehr schnell lernbar, ohne externe Person nutzbar wenn's mal gelernt ist"
- „Gute Ergebnisse“
- „Einfach zu verstehen“
- „Sobald einmal gemacht ist es einfach“

#### Schlecht:

- „Umständlich Y drücken. Unintuitiv sich da hinzuknieen, um die Taste zu drücken“
- „Y-Drücken war nicht einfach. Nicht einfach beide Controller zu sehen. Unsicher wie genau das auf die Markierungen gelegt werden muss“
- „Aufwändig“
- „Lästig“
- „Fehleranfällig“
- „Nicht intuitiv“
- „Fixpunkte verrutschen“
- „großer Zeitaufwand“

#### Sonstiges:

- Vorschlag: „durch Guardian bestimmen lassen, wo Markierungen landen sollen“

## **Guardian**

Gut:

- „Sehr einfach. Großartig.“
- „Man musste keine Geräte hinlegen ging auch recht schnell muss Guardian eh zeichnen, ist also kein Mehraufwand“
- „man muss sich nicht bücken“
- „keine aufwendigen extra Schritte notwendig“
- „keine Hilfsmittel benötigt“
- „sehr einfach, da es nur wenige Schritte gibt“

Schlecht:

- „unnötiges y drücken“
- „Vielleicht schwierig für manche Leute das nachzuzeichnen weil man eine ruhige Hand braucht“
- „Man weiß nicht, wie genau es sein muss“
- „Könnte schwierig sein, wenn keine Linie vorhanden ist“
- „Schwer den Guardian genau zu zeichnen“
- „Schwierig, wenn kein Guardian am Boden markiert wurde“

Sonstiges:

- „Findet es cool das es so ein Toolkit gibt, weil es sonst sehr schwer ist so was zu bauen.“
- „Hat Spaß gemacht (kleines mini game die Linien nachzuzeichnen)“

## **Übergreifend:**

Beste Methode:

- “Offensichtlich Guardian”
- „Guardian, weil kleinen spaß Faktor hat und super einfach ist. Am wenigsten aufwand im Vergleich zu den anderen Methoden.“
- „Guardian, wegen dem geringen Aufwand“
- „Guardian, wegen den wenigen Schritten und weniger Fehlerquellen“

In Zukunft am meisten verwendet:

- „Guardian, weils bei virtuellen Meetings gut verwendet werden könnte. „
- „Guardian: Jeder ohne VR Kenntnisse bekommt das easy hin“
- „Falls sich nur eine Durchsetzt, Guardian. Weil keine extra Komponenten benötigt werden“
- „Guardian, weil Personen faul sind und man sich mit dieser Methode Zeit spart“

Sonstiges

- „Ich find's cool (Guardian).“ kann sich sehr gut vorstellen, dass es viel Verwendung findet.
- Punkte & Pose benötigen bessere Markierungen

### A.3. Ausgefüllte SUS-Fragebögen

77

Guardian

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	(1)
2. I found the method unnecessarily complex.	(1)	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	(1)
4. I think that I would need the support of a technical person to be able to use this system.	!	(1)	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	(1)	!
6. I thought there was too much inconsistency in this method.	!	!	(1)	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	(1)
8. I found the method very cumbersome to use.	(1)	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	(1)
10. I needed to learn a lot of things before I could get going with this method.	!	(1)	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    (10) Extremely likely

JY Park

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	1	1	0	1	1
2. I found the method unnecessarily complex.	1	1	1	1	0
3. I thought the method was easy to use.	1	0	1	1	1
4. I think that I would need the support of a technical person to be able to use this system.	1	1	1	1	0
5. I found the various functions in this method were well integrated.	1	1	0	1	1
6. I thought there was too much inconsistency in this method.	1	1	0	1	1
7. I would imagine that most people would learn to use this method very quickly.	0	1	1	1	1
8. I found the method very cumbersome to use.	1	1	1	1	0
9. I felt very confident using the method.	1	1	0	1	1
10. I needed to learn a lot of things before I could get going with this method.	1	1	0	0	1

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

51 Pose

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	1	2	3	4	5
2. I found the method unnecessarily complex.	1	2	3	4	5
3. I thought the method was easy to use.	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system.	1	2	3	4	5
5. I found the various functions in this method were well integrated.	1	2	3	4	5
6. I thought there was too much inconsistency in this method.	1	2	3	4	5
7. I would imagine that most people would learn to use this method very quickly.	1	2	3	4	5
8. I found the method very cumbersome to use.	1	2	3	4	5
9. I felt very confident using the method.	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this method.	1	2	3	4	5

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10    Extremely likely

# Lets Go!

Please enter your participant number: \_\_\_\_\_

## System Usability Scale (SUS)

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	1	1	1	1	1
2. I found the method unnecessarily complex.	1	1	1	1	1
3. I thought the method was easy to use.	1	1	1	1	1
4. I think that I would need the support of a technical person to be able to use this system.	1	1	1	1	1
5. I found the various functions in this method were well integrated.	1	1	1	1	1
6. I thought there was too much inconsistency in this method.	1	1	1	1	1
7. I would imagine that most people would learn to use this method very quickly.	1	1	1	1	1
8. I found the method very cumbersome to use.	1	1	1	1	1
9. I felt very confident using the method.	1	1	1	1	1
10. I needed to learn a lot of things before I could get going with this method.	1	1	1	1	1

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

*Likes. Points*

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	1	2	3	4	5
2. I found the method unnecessarily complex.	1	2	3	4	5
3. I thought the method was easy to use.	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system.	1	2	3	4	5
5. I found the various functions in this method were well integrated.	1	2	3	4	5
6. I thought there was too much inconsistency in this method.	1	2	3	4	5
7. I would imagine that most people would learn to use this method very quickly.	1	2	3	4	5
8. I found the method very cumbersome to use.	1	2	3	4	5
9. I felt very confident using the method.	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this method.	1	2	3	4	5

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

Class Poso

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

Vini Guardian

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0 1 2 3 4 5 6 7 8 9 10 Extremely likely

# Vivio Points

Please enter your participant number: \_\_\_\_\_

## System Usability Scale (SUS)

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

Vivi

Pose

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

Norbert Gerdian

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0 1 2 3 4 5 6 7 8 9 10 Extremely likely

*Norbert Points*

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

Norbert Poese

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

*Paul Guardian*

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	(1)
2. I found the method unnecessarily complex.	(1)	!	!	!	!
3. I thought the method was easy to use.	!	!	!	(1)	!
4. I think that I would need the support of a technical person to be able to use this system.	(1)	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	(1)
6. I thought there was too much inconsistency in this method.	(1)	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	(1)
8. I found the method very cumbersome to use.	(1)	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	(1)
10. I needed to learn a lot of things before I could get going with this method.	(1)	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0 1 2 3 4 5 6 7 8 9 (10) Extremely likely

Paul Points

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	!
2. I found the method unnecessarily complex.	!	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	!
4. I think that I would need the support of a technical person to be able to use this system.	!	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	!
6. I thought there was too much inconsistency in this method.	!	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	!
8. I found the method very cumbersome to use.	!	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	!
10. I needed to learn a lot of things before I could get going with this method.	!	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    10 Extremely likely

Pawn C Pose

Please enter your participant number: \_\_\_\_\_

**System Usability Scale (SUS)**

This is a standard questionnaire that measures the overall usability of a system. Please select the answer that best expresses how you feel about each statement after using the website today.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1. I think I would like to use this method frequently.	!	!	!	!	(1)
2. I found the method unnecessarily complex.	(1)	!	!	!	!
3. I thought the method was easy to use.	!	!	!	!	(1)
4. I think that I would need the support of a technical person to be able to use this system.	(1)	!	!	!	!
5. I found the various functions in this method were well integrated.	!	!	!	!	(1)
6. I thought there was too much inconsistency in this method.	(1)	!	!	!	!
7. I would imagine that most people would learn to use this method very quickly.	!	!	!	!	(1)
8. I found the method very cumbersome to use.	(1)	!	!	!	!
9. I felt very confident using the method.	!	!	!	!	(1)
10. I needed to learn a lot of things before I could get going with this method.	(1)	!	!	!	!

How likely are you to recommend this method to others? (please circle your answer)

Not at all likely 0    1    2    3    4    5    6    7    8    9    (10) Extremely likely

# Abbildungsverzeichnis

2.1.	Das Meta Quest 2 VR-Headset [16] . . . . .	6
2.2.	Beispiel für die Überlagerung der virtuellen und realen Welt bei co-located Anwendungen[9] . . . . .	8
2.3.	Grafische Darstellung der Beziehungen zwischen mehreren Frames . . . . .	15
2.4.	Eine Hierarchie in Unity (links) dargestellt mithilfe der grafischen Notation als Transformationen (rechts). . . . .	17
2.5.	Vor und nach der translation und rotation des Frames $\{T\}$ . Die Frames $\{N1\}$ und $\{N2\}$ habe relativ zu $\{T\}$ noch die gleiche Transformation. . . . .	18
2.6.	Für den Kalibrations-Flow bereitgestellter Druck von Leisi et. al[46] . . . . .	20
3.1.	Die internen Koordinatensysteme zweier XR-Geräte ( $\{Q1\}$ und $\{Q2\}$ ) und die benötigten Transformationen (Pfeile) um $\{N\}$ nach $\{Q1\}$ abzubilden: $\begin{smallmatrix} Q1 \\ Q2 \end{smallmatrix} T \begin{smallmatrix} Q2 \\ N \end{smallmatrix} T$ . . . . .	22
3.2.	Visualisierung der Gleichung 3.3 . . . . .	24
3.3.	Visualisierung der Frameausrichtungs Berechnungen . . . . .	26
3.4.	Reduzierung der Rotationen auf eine Achse . . . . .	27
3.5.	Beispiel für die Durchführung einer Kalibration mittels der Posen-Methode. Durch die Positions- und Winkelabweichung gibt es am Rand des Spielfelds zwischen der Soll- und Ist-Position den Fehler $E_p$ . . . . .	29
3.6.	Positionsfehler am Rand des Spielareals laut McGill et al. aus [34]. . . . .	29
3.7.	Visualisierung der Erstellung einer der virtuellen Referenzpose. Durch $P1$ und $P2$ wird diese in beiden Koordinatensystemen definiert. . . . .	30
3.8.	Erstellung einer Guardian-Begrenzung und das Ergebnis. Die blaue Linie stellt den benutzerdefinierten Rand des Spielbereichs dar. Dieser wird intern als Punktwolke gespeichert.[16] . . . . .	33
3.9.	Punktwolkenregistrierung im Kontext zweier Guardians . . . . .	34
3.10.	Interpretation von $T^*$ als Abbildungstransformation . . . . .	35
3.11.	Mithilfe der durch Punktwolkenregistrierung ermittelten Transformation $T^*$ kann die neue Pose für den Trackingspace $T'$ berechnet werden . . . . .	36
3.12.	Caption . . . . .	38
3.13.	Das Ergebnis der Punktezuordnung. Für jeden Punkt aus einer Punktwolke (Lila) wurde durch die euklidische Distanz der nächstgelegene Punkt der zweiten Wolke (Grün) zugewiesen und somit eine Korrespondenz erstellt.	38

3.14. Bei der Point-To-Plane ICP-Variante wird der Abstand zwischen Punkten aus $P$ und deren Skalarprojektion auf den Normalenvektor des Korrespondenz-Punkts aus $Q$ minimiert.[73] . . . . .	41
5.1. Architektur des Co-Location Toolkits (COLTK) . . . . .	48
5.2. Klassendiagramm der Calibrator-Klasse. Gekürzt auf Public-API und relevante Felder . . . . .	51
5.3. Der Ablauf des IcpBf-Algorithmus bildlich dargestellt . . . . .	54
5.4. Flussdiagramm des IcpPoint2Point-Algorithmus . . . . .	59
5.5. Berechnung der Normalen für jeden Punkt einer sortierten Punktwolke. Aus [83] . . . . .	60
5.6. Die Architektur der <i>Networking</i> -Bibliothek, aufgeteilt in Schichten. . . . .	61
5.7. Klassendiagramm der Socket-Wrapper . . . . .	62
5.8. Ablauf des Verbindungsaufbaus zwischen einem Acceptor und Connector aus Serversicht . . . . .	65
5.9. (Gekürztes) Klassendiagramm der Acceptor-Connector Komponenten .	65
5.10. Die ServerInfoDispatcher- und ServerFinder-Klassen als Diagramm	67
5.11. Ablauf bzw. Zusammenspiel des ServerFinder und ServerInfoDispatcher, um sich im lokalen Netzwerk zu finden. . . . .	68
5.12. Visualisierung des Nachrichtenaustausches zwischen einem Server und zwei Clients. Durch Threadsafe Queues werden die Nachrichten im Update-Loop mit dem Haupthread synchronisiert. . . . .	68
5.13. Diagramm einiger vom Framework zur Verfügung gestellter Klassen .	69
5.14. Aufbau des Message-Datentyps, welcher zur Nachrichtenübertragung verwendet wird. . . . .	70
5.15. (Gekürztes) Klassendiagramm der Core-Bibliothek. Hier ist zu sehen, welche API durch die Facade der Render-Engine angeboten wird. . . . .	72
5.16. Verbildlichung des Nachrichtenaustausches, wenn ein neuer Client sich beim Server registriert. . . . .	74
5.17. Ablauf zum Aktualisieren der Tracking-Nodes eines Klienten. . . . .	75
5.18. Verbildlichung des Marshaling Prozesses bei Nutzung von P/Invoke .	77
5.19. Die Architektur des COLTK auf Render-Engine-Ebene. . . . .	79
5.20. Grobe Darstellung der Logik hinter der Synchronisierung von Tracking-Nodes	82
6.1. Visualisierung des Datensatzes zum Vergleich der ICP-Algorithmen .	84
6.2. Fehler eines Guardian-Paares bei immer kleiner werdendem Rotations- winkel für den <i>Initial Guess</i> . . . . .	86
6.3. Durchschnittliche Ausführungszeit der ICP-Varianten pro Raum in Mi- krosekunden. . . . .	87
6.4. Durchschnittlicher Fehler der ICP-Varianten pro Raum in Mikrosekunden.	88
6.5. Konfiguration der Kalibration und Synchronisation . . . . .	89

6.6.	Konfiguration des COLTKManager . . . . .	90
6.7.	AvatarManager und zugehöriger Avatar . . . . .	91
6.8.	Auswahl der Referenzpunkte . . . . .	91
6.9.	Erstellung der Ausrichtungspunkte . . . . .	92
6.10.	Einstellungen für eine Ausrichtung mittels der Guardian-Methode . . .	93
6.11.	Verwendung des statischen Guardians zur Erstellung einer Umgebung .	94
6.12.	Erstellung eines statischen Guardians in der Szene . . . . .	94
6.13.	Die Markierung, die zum Aufzeichnen der Koordinaten der Tracking-Nodes verwendet wurde. . . . .	97
6.14.	Testaufbau zur Positionsfehlerbestimmung. Die Nummerierung gibt die Reihenfolge bei der Aufzeichnung der Positionen an. . . . .	97
6.15.	Visualisierung der Messdaten der Posen-Methode . . . . .	99
6.16.	Visualisierung der Messdaten der Punkte-Methode . . . . .	99
6.17.	Visualisierung der Messdaten der Guardian-Methode . . . . .	100
6.18.	Einordnung der SUS-Ergebnisse wie von Sauro[92] vorgeschlagenen. .	105

# Tabellenverzeichnis

6.1. Registrierungen, die in ein lokales Minimum fielen. Die unterstrichenen Werte sind globale Minima. . . . .	85
6.2. Die kleinste, größte und die durchschnittliche Distanz zwischen Soll- und Ist-Positionen aller Aufnahmen eines Durchlaufs. Raumdurchmesser: ca. 5 m . . . . .	101

# Literatur

- [1] Meta. *Horizon Community*. <https://www.oculus.com/horizon-worlds/community/>. [Online; Stand 08. Juli 2022]. 2022 (siehe S. 2).
- [2] Mozilla. *Welcome to Hubs*. <https://hubs.mozilla.com/docs/welcome.html>. [Online; Stand 08. Juli 2022]. 2022 (siehe S. 2).
- [3] Rec Room Inc. *Rec Room - Join the Club!* <https://recroom.com/>. [Online; Stand 08. Juli 2022]. 2022 (siehe S. 2).
- [4] Jan Gugenheimer, Evgeny Stemasov, Julian Frommel und Enrico Rukzio. „ShareVR: Enabling Co-Located Experiences for Virtual Reality between HMD and Non-HMD Users“. In: Mai 2017, S. 4021–4033. doi: 10.1145/3025453.3025683 (siehe S. 2).
- [5] Connor Defanti. *Co-Located Augmented and Virtual Reality Systems*. [PhD. Thesis]. 2019 (siehe S. 2, 18).
- [6] Xukan Ran, Carter Slocum, Maria Gorlatova und Jiasi Chen. „ShareAR: Communication-Efficient Multi-User Mobile Augmented Reality“. In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. HotNets ’19. Princeton, NJ, USA: Association for Computing Machinery, 2019, S. 109–116. doi: 10.1145/3365609.3365867 (siehe S. 2, 10).
- [7] Microsoft. *Spatial Anchors - Übersicht*. <https://docs.microsoft.com/de-de/azure/spatial-anchors/overview>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 2).
- [8] Miscellaneous. *ARWorldMap - ARKit*. <https://developer.apple.com/documentation/arkit/arworldmap>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 2, 10).
- [9] UploadVR. *Oculus Quest Arena ‘Colocation’ With Dead And Buried*. Video: [https://www.youtube.com/watch?v=QJXpHp\\_iQF4](https://www.youtube.com/watch?v=QJXpHp_iQF4). [Online; Stand 28. Juni 2022]. 2022 (siehe S. 2, 8, 10).
- [10] Jan Gugenheimer, Christian Mai, Mark McGill, Julie Williamson, Frank Steinicke und Ken Perlin. „Challenges Using Head-Mounted Displays in Shared and Social Spaces“. In: Apr. 2019, S. 1–8. doi: 10.1145/3290607.3299028 (siehe S. 2).
- [11] Sebastian Herscher, Connor DeFanti, Nicholas Vitovitch, Corinne Brenner, Haijun Xia, Kris Layng und Ken Perlin. „CAVRN: An Exploration and Evaluation of a Collective Audience Virtual Reality Nexus Experience“. In: Okt. 2019, S. 1137–1150. doi: 10.1145/3332165.3347929 (siehe S. 2).
- [12] Kris Layng, Ken Perlin, Sebastian Herscher, Corinne Brenner und Thomas Meduri. „Cave: making collective virtual narrative“. In: Juli 2019, S. 1–8. doi: 10.1145/3306211.3320138 (siehe S. 2).
- [13] Leihui Li, Riwei Wang und Xuping Zhang. „A Tutorial Review on Point Cloud Registrations: Principle, Classification, Comparison, and Technology Challenges“. In: Mathematical Problems in Engineering. Hindawi, 2021. doi: 10.1155/2021/9953910 (siehe S. 2, 37, 40, 41).
- [14] HTC Corporation. *HTC Vive Pro*. <https://www.vive.com/de/product/vive-pro-full-kit/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 5).

- [15] Wikipedia. *Oculus Rift*. [https://en.wikipedia.org/wiki/Oculus\\_Rift](https://en.wikipedia.org/wiki/Oculus_Rift). [Online; Stand 28. Juni 2022]. 2022 (siehe S. 5).
- [16] Meta. *Meta Quest 2*. <https://store.facebook.com/de/quest/products/quest-2/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 5, 6, 33).
- [17] Wikipedia. *Microsoft Hololens*. [https://de.wikipedia.org/wiki/Microsoft\\_HoloLens](https://de.wikipedia.org/wiki/Microsoft_HoloLens). [Online; Stand 28. Juni 2022]. 2022 (siehe S. 6).
- [18] NaturalPoint Inc. *OptiTrack - Motion Capture System*. <https://optitrack.com/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 7, 18).
- [19] Sony Interactive Entertainment Europe Limited. *PlayStation VR*. <https://www.playstation.com/de-de/ps-vr/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 7).
- [20] Valve. *SteamVR Trackingsystem*. <https://partner.steamgames.com/vrlicensing>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 7).
- [21] TonyVT SkarredGhost. *VR inside-out vs outside-in tracking*. <https://skarredghost.com/2017/08/09/vr-inside-vs-outside-tracking/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 7).
- [22] Hugh Langley. *nside-out v Outside-in: How VR tracking works, and how it's going to change*. [www.wearable.com/vr/inside-out-vs-outside-in-vr-tracking-343](http://www.wearable.com/vr/inside-out-vs-outside-in-vr-tracking-343). [Online; Stand 28. Juni 2022]. 2022 (siehe S. 7).
- [23] RedCube GmbH. *trueVRsystems*. <https://www.truevrsystems.com/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 8).
- [24] HOLOGATE. *HOLOGATE*. <https://www.hologate.com/arena/>. [Online; Stand 28. Juni 2022]. 2022 (siehe S. 8).
- [25] Wikipedia. *Simultaneous localization and mapping*. [https://en.wikipedia.org/wiki/Simultaneous\\_localization\\_and\\_mapping](https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping). [Online; Stand 28. Juni 2022]. 2022 (siehe S. 9).
- [26] Chuan-en Lin. „Introduction to Motion Estimation with Optical Flow“. In: (2019) (siehe S. 9).
- [27] Ke Huo, Tianyi Wang, Luis Paredes, Ana Villanueva, Yuanzhi Cao und Karthik Ramani. „SynchronizAR: Instant Synchronization for Spontaneous and Spatial Collaborations in Augmented Reality“. In: Okt. 2018, S. 19–30. doi: 10.1145/3242587.3242595 (siehe S. 9).
- [28] OpenCV. *Detection of ArUco Markers*. [https://docs.opencv.org/4.x/d5/dae/tutorial\\_aruco\\_detection.html](https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html). [Online; Stand 29. Juni 2022]. 2022 (siehe S. 10).
- [29] Karan Ahuja, Sujeeth Paredy, Robert Xiao, Mayank Goel und Chris Harrison. „LightAnchors: Appropriating Point Lights for Spatially-Anchored Augmented Reality Interfaces“. In: UIST ’19. New Orleans, LA, USA: Association for Computing Machinery, 2019, S. 189–196. doi: 10.1145/3332165.3347884 (siehe S. 10).
- [30] Microsoft. *Spatial anchors - Mixed Reality*. <https://docs.microsoft.com/de-de/windows/mixed-reality/design/spatial-anchors>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 10).
- [31] Ramón Argüelles, Matt Wojciakowski, Fowler Cory und McAllister Ross. *Azure Spatial Anchors overview*. <https://docs.microsoft.com/en-gb/azure/spatial-anchors/overview>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 10).

- [32] Miscellaneous. *Mit Cloud Anchors können Nutzer ihre AR-Erfahrungen teilen.* <https://developers.google.com/ar/develop/cloud-anchors>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 10).
- [33] Dominik Van Opdenbosch, Tamay Aykut, Nicolas Alt und Eckehard Steinbach. „Efficient Map Compression for Collaborative Visual SLAM“. In: *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2018, S. 992–1000. doi: 10.1109/WACV.2018.00114 (siehe S. 10).
- [34] Mark McGill, Jan Gugenheimer und Euan Freeman. „A Quest for Co-Located Mixed Reality: Aligning and Assessing SLAM Tracking for Same-Space Multi-User Experiences“. In: *26th ACM Symposium on Virtual Reality Software and Technology*. VRST '20. Virtual Event, Canada: Association for Computing Machinery, 2020. doi: 10.1145/3385956.3418968 (siehe S. 11, 19, 28, 29, 32).
- [35] Ian Hamilton. „Oculus Quest Arena-Scale Projects Persist Despite Unclear Path“. In: (2019) (siehe S. 11).
- [36] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2003 (siehe S. 11).
- [37] D.C. Lay. *Linear Algebra and Its Applications*. Pearson Education, 2003 (siehe S. 11).
- [38] J.J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley series in electrical and computer engineering: control engineering. Pearson/Prentice Hall, 2005 (siehe S. 11, 13).
- [39] Miscellaneous. *Class ARSessionOrigin*. <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@1.0-preview.8/api/UnityEngine.XR.ARFoundation.ARSessionOrigin.html>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 16).
- [40] Miscellaneous. *What is a Scene Graph?* <https://www.openscenegraph.com/index.php/documentation/knowledge-base/36-what-is-a-scene-graph>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 17).
- [41] Inc VRcade. *The VRcade: A Virtual Reality Platform for Truly Immersive Gaming*. <http://vrcade.com/>. 2015 (siehe S. 18).
- [42] The Void. *The Void: The Vision Of Infinite Dimensions*. <https://thevoid.com/>. 2016 (siehe S. 18).
- [43] A. Dushman. „On Gyro Drift Models and Their Evaluation“. In: *IRE Transactions on Aerospace and Navigational Electronics* ANE-9 (1962), S. 230–234 (siehe S. 18).
- [44] Google. *Daydream Standalone*. <https://developers.google.com/vr/discover/daydream-standalone>. [Online; Stand 29. Juni 2022]. 2019 (siehe S. 18).
- [45] Minimalisticky. *Mirror Networking*. <https://mirror-networking.com/>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 19).
- [46] Chris Elvis Leisi und Oliver Sahli. „Co-Experiencing Virtual Spaces: A Standalone Multiuser Virtual Reality Framework“. In: (Dez. 2020). doi: 10.5281/zenodo.4399217 (siehe S. 20).
- [47] Defanti. *A Quick and Easy Calibration Method*. [frl.nyu.edu/a-quick-and-easy-calibration-method/](http://frl.nyu.edu/a-quick-and-easy-calibration-method/). [Online; Stand 29. Juni 2022]. 2019 (siehe S. 20).
- [48] Dennis Reimer, Iana Podkosova, Daniel Scherzer und Hannes Kaufmann. „Colocation for SLAM-Tracked VR Headsets with Hand Tracking“. In: *Computers* 10.5 (2021). doi: 10.3390/computers10050058 (siehe S. 20, 21, 112).

- [49] P.J. Besl und Neil D. McKay. „A method for registration of 3-D shapes“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992), S. 239–256. doi: 10.1109/34.121791 (siehe S. 37, 40).
- [50] Toru Tamaki, Miho Abe, Bisser Raytchev und Kazufumi Kaneda. „Softassign and EM-ICP on GPU“. In: Nov. 2010, S. 179–183. doi: 10.1109/IC-NC.2010.60 (siehe S. 37).
- [51] Benoit Combès und Sylvain Prima. „An Efficient EM-ICP Algorithm for Non-Linear Registration of Large 3D Point Sets“. In: *Comput. Vis. Image Underst.* 191.C (Feb. 2020). doi: 10.1016/j.cviu.2019.102854 (siehe S. 37).
- [52] Dirk Holz, Alexandru-Eugen Ichim, Federico Tombari, Radu B. Rusu und Sven Behnke. „Registration with the Point Cloud Library A Modular Framework for Aligning in 3-D“. In: *Ieee Robotics amp; Automation Magazine* 22.4 (2015), S. 15. 110–124. doi: 10.1109/Mra.2015.2432331 (siehe S. 37).
- [53] S. Rusinkiewicz und M. Levoy. „Efficient variants of the ICP algorithm“. In: *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*. 2001, S. 145–152. doi: 10.1109/IM.2001.924423 (siehe S. 37).
- [54] Xian-Feng Han, Jesse S. Jin, Ming-Jie Wang, Wei Jiang, Lei Gao und Liping Xiao. „A review of algorithms for filtering the 3D point cloud“. In: *Signal Processing: Image Communication* 57 (2017), S. 103–112. doi: <https://doi.org/10.1016/j.image.2017.05.009> (siehe S. 38).
- [55] Mustafa Yilmaz und Murat Uysal. „Comparison of data reduction algorithms for LiDAR-derived digital terrain model generalisation“. In: *Area* 48 (Mai 2016). doi: 10.1111/area.12276 (siehe S. 38).
- [56] Ko Nishino und Katsushi Ikeuchi. „Robust Simultaneous Registration of Multiple Range Images“. In: *Digitally Archiving Cultural Objects*. Boston, MA: Springer US, 2008, S. 71–88. doi: 10.1007/978-0-387-75807\_5 (siehe S. 39).
- [57] Jon Louis Bentley. „Multidimensional Binary Search Trees Used for Associative Searching“. In: *Commun. ACM* 18.9 (Sep. 1975), S. 509–517. doi: 10.1145/361002.361007 (siehe S. 39).
- [58] Donald Meagher. „Geometric Modeling Using Octree-Encoding“. In: *Computer Graphics and Image Processing* 19 (Juni 1982), S. 129–147. doi: 10.1016/0146-664X(82)90104-6 (siehe S. 39).
- [59] Raouf Benjemaa und Francis Schmitt. „Fast global registration of 3D sampled surfaces using a multi-z-buffer technique1Based on "Fast global registration of 3D sampled surfaces using multi-z-buffer technique"by Raouf Benjemaa and Francis Schmitt which appeared in Proceedings of 3-D Digital Imaging and Modelling, Ottawa, May 1997 (pp. 113–120). © 1999 IEEE.1“. In: *Image and Vision Computing* 17.2 (1999), S. 113–123. doi: [https://doi.org/10.1016/S0262-8856\(98\)00115-2](https://doi.org/10.1016/S0262-8856(98)00115-2) (siehe S. 39).
- [60] Soon-Yong Park und Murali Subbarao. „An accurate and fast point-to-plane registration technique“. In: *Pattern Recognition Letters* 24.16 (2003), S. 2967–2976. doi: [https://doi.org/10.1016/S0167-8655\(03\)00157-0](https://doi.org/10.1016/S0167-8655(03)00157-0) (siehe S. 39).
- [61] Kari Pulli. „Multiview registration for large data sets“. In: Feb. 1999, S. 160–168. doi: 10.1109/IM.1999.805346 (siehe S. 39).
- [62] Su Sun, Wei Song, Yifei Tian und Simon Fong. „An ICP-Based Point Clouds Registration Method for Indoor Environment Modeling“. In: *Advanced Multimedia and Ubiquitous Engineering*. Hrsg. von

- James J. Park, Laurence T. Yang, Young-Sik Jeong und Fei Hao. Singapore: Springer Singapore, 2020, S. 339–344 (siehe S. 39).
- [63] T. Zinsser, J. Schmidt und H. Niemann. „A refined ICP algorithm for robust 3-D correspondence estimation“. In: *Proceedings 2003 International Conference on Image Processing (Cat. No.03CH37429)*. Bd. 2. 2003, S. II–695. doi: 10.1109/ICIP.2003.1246775 (siehe S. 39).
  - [64] T. Masuda, K. Sakaue und N. Yokoya. „Registration and integration of multiple range images for 3-D model construction“. In: *Proceedings of 13th International Conference on Pattern Recognition*. Bd. 1. 1996, 879–883 vol.1. doi: 10.1109/ICPR.1996.546150 (siehe S. 39).
  - [65] Silvio Giancola, Jens Schneider, Peter Wonka und Bernard Ghanem. „Integration of Absolute Orientation Measurements in the KinectFusion Reconstruction pipeline“. In: *CoRR abs/1802.03980* (2018) (siehe S. 39).
  - [66] Jong-Hee Back, Sunho Kim und Yo-Sung Ho. „High-Precision 3D Coarse Registration Using RANSAC and Randomly-Picked Rejections“. In: *MultiMedia Modeling*. Hrsg. von Klaus Schöffmann, Thanarat H. Chalidabhongse, Chong Wah Ngo, Supavadee Aramvith, Noel E. O’Connor, Yo-Sung Ho, Moncef Gabbouj und Ahmed Elgammal. Cham: Springer International Publishing, 2018, S. 254–266 (siehe S. 39).
  - [67] Ahmad Almhdie, Christophe Léger, Mohamed Deriche und Roger Lédée. „3D registration using a new implementation of the ICP algorithm based on a comprehensive lookup matrix: Application to medical imaging“. In: *Pattern Recognition Letters* 28 (Sep. 2007), S. 1523–1533. doi: 10.1016/j.patrec.2007.03.005 (siehe S. 39).
  - [68] Jiquan Liu, Maryam Rettmann, David Holmes III, Huilong Duan und Richard Robb. „A Piecewise Patch-to-Model Matching Method for Image-guided Cardiac Catheter Ablation“. In: *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society* 35 (März 2011), S. 324–32. doi: 10.1016/j.compmedimag.2011.02.001 (siehe S. 39).
  - [69] Elwan Héry, Philippe Xu und Philippe Bonnifait. „LiDAR based relative pose and covariance estimation for communicating vehicles exchanging a polygonal model of their shape“. In: Okt. 2018 (siehe S. 39).
  - [70] K. S. Arun, T. S. Huang und S. D. Blostein. „Least-Squares Fitting of Two 3-D Point Sets“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-9.5* (1987), S. 698–700. doi: 10.1109/TPAMI.1987.4767965 (siehe S. 40, 41).
  - [71] R. Bergevin, M. Soucy, H. Gagnon und D. Laurendeau. „Towards a general multi-view registration technique“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18.5 (1996), S. 540–547. doi: 10.1109/34.494643 (siehe S. 40).
  - [72] Y. Chen und G. Medioni. „Object modeling by registration of multiple range images“. In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. 1991, 2724–2729 vol.3. doi: 10.1109/ROBOT.1991.132043 (siehe S. 40).
  - [73] Ben Bellekens, Vincent Spruyt, Raf Berkvens und Maarten Weyn. „A Survey of Rigid 3D Pointcloud Registration Algorithms“. In: Aug. 2014 (siehe S. 41, 56).
  - [74] Berthold Horn, Hugh Hilden und Shahriar Negahdaripour. „Closed-Form Solution of Absolute Orientation using Orthonormal Matrices“. In: *Journal of the Optical Society of America A* 5 (Juli 1988), S. 1127–1135. doi: 10.1364/JOSAA.5.001127 (siehe S. 41).

- [75] Berthold Horn. „Closed-Form Solution of Absolute Orientation Using Unit Quaternions“. In: *Journal of the Optical Society A* 4 (Apr. 1987), S. 629–642. doi: 10.1364/JOSAA.4.000629 (siehe S. 41).
- [76] Andrew Fitzgibbon. „Robust Registration of 2D and 3D Point Sets“. In: *Image and Vision Computing* 21 (Apr. 2002), S. 1145–1153. doi: 10.1016/j.imavis.2003.09.004 (siehe S. 41).
- [77] Refactoring.Guru. *Facade*. <https://refactoring.guru/design-patterns/facade>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 47, 71).
- [78] Refactoring.Guru. *Singleton*. <https://refactoring.guru/design-patterns/singleton>. [Online; Stand 29. Juni 2022]. 2022 (siehe S. 49, 79).
- [79] Radu Bogdan Rusu und Steve Cousins. „3D is here: Point Cloud Library (PCL)“. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, Mai 2011 (siehe S. 52).
- [80] Wikipedia. *Kd-Baum*. <https://de.wikipedia.org/wiki/K-d-Baum>. [Online; 01. Juli 2022]. 2022 (siehe S. 55).
- [81] Wikipedia. *Gauß-Newton-Verfahren*. <https://de.wikipedia.org/wiki/Gauss-Newton-Verfahren>. [Online; Stand 01. Juli 2022]. 2022 (siehe S. 57).
- [82] Cyrill Stachniss. *ICP Point Cloud Registration - Part 3: Non-linear Least Squares* (Cyrill Stachniss, 2021). Video: <https://youtu.be/CJE59i8oxIE>. [Online; Stand 01. Juli 2022]. 2022 (siehe S. 57).
- [83] Niosus. *ICP*. Jupiter Notebook: <https://nbviewer.org/github/niosus/notebooks/blob/master/icp.ipynb>. [Online; Stand 01. Juli 2022]. 2022 (siehe S. 57, 60).
- [84] S. Rusinkiewicz und M. Levoy. „Efficient variants of the ICP algorithm“. In: *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*. 2001, S. 145–152. doi: 10.1109/3IM.2001.924423 (siehe S. 59).
- [85] Douglas Schmidt. „Acceptor and Connector“. In: (Nov. 1997) (siehe S. 64).
- [86] Wikipedia. *.Net-Framework*. <https://de.wikipedia.org/wiki/.Net-Framework>. [Online; Stand 02. Juli 2022]. 2022 (siehe S. 76).
- [87] Microsoft. *Plattformaufruf (P/Invoke)*. <https://docs.microsoft.com/de-de/dotnet/standard/native-interop/pinvoke>. [Online; Stand 02. Juli 2022]. 2022 (siehe S. 76).
- [88] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg und Saul Greenberg. „Evaluation Strategies for HCI Toolkit Research“. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: Association for Computing Machinery, 2018, S. 1–17. doi: 10.1145/3173574.3173610 (siehe S. 88, 113).
- [89] Oculus. *Oculus Integration*. <https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022>. [Online; Stand 03. Juli 2022]. 2022 (siehe S. 89).
- [90] usability.gov. *Recruiting Usability Test Participants*. <https://www.usability.gov/how-to-and-tools/methods/recruiting-usability-test-participants.html>. [Online; Stand 04. Juli 2022]. 2022 (siehe S. 102).
- [91] Wikipedia. *System Usability Scale*. [https://de.wikipedia.org/wiki/System\\_Usability\\_Scale](https://de.wikipedia.org/wiki/System_Usability_Scale). [Online; Stand 04. Juli 2022]. 2022 (siehe S. 105).

- [92] Wikipedia. *5 Ways to Interpret a SUS Score*. <https://measuringu.com/interpret-sus-score/>. [Online; Stand 04. Juli 2022]. 2018 (siehe S. 105).

## Erklärung zur Abschlussarbeit

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde. Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

Übersee, den 18. Juli 2022



---

Fabien J. E. Zwick