



Projektdokumentation

MArC
Mixed Reality Architecture Composer

von

Laura Anger (Matrikelnr. 11086356)
Vera Brockmeyer (Matrikelnr. 11077082)
Paul Berning (Matrikelnr. 11068249)
Lukas Kolhagen (Matrikelnr. 11084355)

Durchgeführt im
Master Medientechnologie
im SS 2016 und WS 2016/17

Betreuer:

Prof. Dr. Stefan Michael Grünvogel
Institut für Medien- und Phototechnik

Inhaltsverzeichnis

1 Einleitung	5
1.1 Motivation	5
1.2 Anwendungskontext	5
1.3 Projektziel	6
2 Grundlagen	6
2.1 Architektur von Virtual-Reality-Anwendungen	7
2.2 Haptische Interaktionsmethoden in VR Umgebungen	8
2.3 Handtracking Interaktionsmethoden	8
2.4 ISO/OSI-7-Schichtenmodell	9
2.5 Bedienkonzepte in VR-Umgebungen	10
2.6 Marker Tracking	11
3 Materialien	13
3.1 Hardware	13
3.1.1 Computer zur Ausführung der Unity-Simulation	13
3.1.2 Computer zur Ausführung der Tracking-Anwendung	13
3.1.3 HTC Vive	14
3.1.4 IDS uEye 164LE-C	15
3.1.5 Leap Motion	15
3.1.6 Leap Motion SDK	16
3.1.7 ArUco Marker	17
3.1.8 Würfel Marker	18
3.1.9 Schachbrett-Kalibrierungshelfer zur Kamerakalibrierung	19
3.1.10 Kalibrierungscontroller zur Arbeitsbereichskalibrierung	20
3.2 Obsolete Hardware	20
3.2.1 Ovrvision Pro	21
3.2.2 Webcam	22
3.3 Software	22
3.3.1 Unity	22
3.3.2 Visual Studio 2015	23
3.3.3 OpenCV	23

3.3.4	Steam VR	23
3.3.5	Windows Sockets (Winsock)	24
4	System	25
4.1	Aufbau	25
4.1.1	Tracking Aufbau	25
4.1.2	Unity Aufbau	26
4.2	Systemvoraussetzungen	27
4.3	Netzwerk	28
4.3.1	Server-seitige Netzwerkanbindung	28
4.3.2	Client-seitige Netzwerkanbindung	30
4.4	Starten des Systems	32
4.5	Menüführung	32
4.5.1	Menüs	32
4.5.2	Ablauf der Menüführung	34
4.6	Kontextmenü	36
4.6.1	Marker Handles	37
4.6.2	Ein- und Ausblenden des Kontextmenüs	38
4.6.3	Berechnung der Gebäudeeigenschaften	38
4.7	Table Menü	39
4.7.1	Szenen Management	39
4.7.2	Speichern von Szenen	40
4.7.3	Laden von Szenen	41
4.7.4	Match Modus	42
5	Tracking Programm	42
5.1	uEye Ansteuerung	43
5.2	Kalibrierung	44
5.2.1	Korrespondierende Punktpaare	44
5.2.2	Definition der Koordinatensysteme	45
5.2.3	Kamerakalibrierung	46
5.2.4	Kalibrierung des Arbeitsbereichs	49
5.2.5	Kalibrierungsfehler	50
5.2.6	Mögliche Fehlerquellen	51

5.2.7	Schnittstelle der beiden Kalibrierung	51
5.2.8	Speichern und Laden der Kalibrierungen	52
5.3	Marker Detektion	53
5.3.1	Detektion der grünen Rechtecke	53
5.3.2	ArUco Marker Tracking	54
5.4	Marker Objekt Definition	54
5.5	Verfolgung und Registrierung der Marker	55
6	Ausblick	57
6.1	Trackingmethoden	57
6.2	AR Erweiterung mit der Webcam	57
6.3	Daten als Excel Tabelle	57
6.4	Validierung	57
7	Projektmanagement	57
7.1	Zeitplanung	57
7.2	Gant Chart	57
7.3	Planungsmethoden	57
7.4	Reflexion	57
7.4.1	Kommunikation im Team und nach Außen	57
7.4.2	Probleme	58
8	Zusammenfassung	58
9	Steckbrief	58
10	Anhang	58
10.1	<i>MArC</i> ReadMe-Datei	58
10.2	<code>startTCPServer()</code> -Methode	66
10.3	<code>getPointerOfMarkerVec()</code> -Methode	66
10.4	<code>interpretTCPMarkerData()</code> -Methode	67

1 Einleitung

Paul

Virtuelle und erweiterte Realität ist bereits seit einiger Zeit in aller Munde. Mit dem Erscheinen der *Oculus Rift*, der *HTC Vive* und anderen Virtual-Reality-Headsets rückt eine neue Art der Immersion beim Genuss von Videospielen in greifbare Nähe. Wenn es jedoch um die Nutzung dieser Technologien zur Effizienzsteigerung in professionellen Umgebungen geht, so sind verfügbare Anwendungen bisher nur selten anzutreffen.

Die Entwicklung von *MArC*, einem Mixed-Reality-System für die architektonische Planung bei Siedlungsbauten, soll dies ändern. Mittels eines ausgeklügelten Zusammenspiels verschiedener neuartiger Technologien ist es möglich, mit *MArC* intuitiv abstrahierte Gebäude zu erstellen, verändern und zusammen zu stellen.

Hierbei werden Marker in Form von Würfeln, welche aus Aluminium gearbeitet und mit einem Code auf der Oberfläche versehen sind, auf einem Tisch bewegt. Eine Kamera über dem Tisch trackt die Position dieser Würfel auf dem Tisch. Der Nutzer, welcher eine *HTC Vive* trägt (siehe Abschnitt 3.1.3), sieht an der Stelle der Würfel in dem Headmounted Display die virtuellen Gebäude. Er kann diese bewegen, indem er die Würfel verschiebt oder rotiert. Die Dimensionen des Gebäudes lassen sich intuitiv mit dem Finger in die gewünschte Größe ziehen. Die Gebäude können Stockwerkweise erhöht oder erniedrigt werden. Hat der Nutzer eine Szene entworfen, kann er diese Speichern und zu einem späteren Zeitpunkt wieder laden. Hierzu dient ein virtuelles Menü, welches neben dem Arbeitsbereich platziert wurde.

1.1 Motivation

Paul

Zu Beginn der Entwicklung von *MArC* lernte das Team die Arbeitsmethoden von Architekten in einem Architekturbüro kennen. Die Architekten stellten dem Team die Arbeitsmethode vor, wie zur Zeit Siedlungen geplant und entworfen werden. Des Weiteren stellte sich heraus, dass zur Zeit ein komplizierter Workflow verwendet wird, um anfängliche Entwürfe im Laufe der Zeit zu konkretisieren und zu digitalisieren. Hierbei sind extrem viele Absprachen zwischen den einzelnen Arbeitsschritten notwendig. Diese im folgenden detailliert beschriebene Ausgangssituation lässt sich durch die modernen Technologien sehr gut verbessern.

1.2 Anwendungskontext

Paul

Die bisherige Konzeptionierung zur Erschließung von Wohngebieten findet heute in Architekturbüros meist noch so statt wie vor dem Einzug der weit verbreiteten digitalen Technik in unsere Arbeitsleben.

Dazu werden simple Modelle aus leicht zu verarbeitenden Materialien – wie etwa Styropor – erstellt und als Platzhalter für die zu planenden Gebäude bei dem Entwurf verwendet.

Verschiedene Modelle werden fest miteinander verklebt und platziert. Die erstellten Modelle werden im folgenden immer weiter verfeinert und optimiert. Diese Herangehensweise macht nachträgliche Änderungen an den Gebäuden aufwendig und führt zu dem Umstand, dass ein bestimmter Zustand der Planung nur umständlich wiederhergestellt werden kann – zum Beispiel durch Fotografieren und späteren manuellen Wiederaufbau.

1.3 Projektziel

Paul

An diesem Punkt setzt MArC an, der „Mixed Reality Architecture Composer“. Um den kreativen Prozess zu optimieren, war es notwendig keine rein virtuelle Lösung zu erarbeiten. Eine haptische Komponente war von Wichtigkeit, sodass der Bezug zu den Gebäuden besser greifbar ist. Die Änderung von den abstrahierten Gebäuden in alle Dimensionen sollte für den Benutzer so leicht wie möglich Umsetzbar sein. Andere Mitarbeiter sollten den Prozess mit verfolgen können und so in den Prozess der erarbeitung mit einbezogen werden. Dies ist bei *MArC* dadurch möglich, dass die Darstellung im Headmounted Display auch auf einem Monitor verfolgt werden kann.

Ebenso sollten Szenen einfach abgespeichert werden können und bei Bedarf wieder aufgerufen werden, um Änderungen durchführen zu können oder ältere Projektstände wieder herzustellen. Dies ist durch eine elegante virtuelle Menülösung verwirklicht worden, die komplett mittels des Fingers des Benutzers bedient werden kann. Ein Absetzen des Headmounted Displays ist, nachdem das System erfolgreich eingerichtet wurde, nicht mehr notwendig.

Doch ebenso wie die Usability sollte die Performance berücksichtigt werden. Der Benutzer sollte zu jeder Zeit ein flüssiges Markertracking sowie ein reibungslose Darstellung in der virtuellen Umgebung erleben können. Um dies zu realisieren musste das Tracking der Marker und die Darstellung auf zwei verschiedenen Computern aufgeteilt werden, die untereinander Kommunizieren.

2 Grundlagen

Das hier muss auch noch wer machen

2.1 Architektur von Virtual-Reality-Anwendungen

Lukas

Bei der Entwicklung von Virtual Reality (VR) Anwendungen gibt es zwei entscheidende Unterschiede im Vergleich zu normalen Software-Anwendungen [7]:

- die virtuelle Umgebung und das damit verbundene Interface sollten auf die vorliegende Aufgabe zugeschnitten werden und
- spezielle Anforderungen an die Performanz der Anwendung müssen erfüllt sein, damit die virtuelle Realität erfolgreich präsentiert werden kann.

Als grundsätzlich unterschiedliche Architekturen bei der Entwicklung von VR-Anwendungen kann nach Hardware-Plattformen unterschieden werden:

Desktop-VR-Applikationen: hierbei handelt es sich um die leistungsstärksten VR-Applikationen, die potente Computer-Hardware häufig mit Head-Mounted-Displays wie etwa Oculus Rift oder HTC Vive verwenden.

Mobile-VR-Applikationen: mobile Anwendungen vereinen häufig (jedoch nicht immer) alle notwendige Hardware in einem Gerät, wie etwa einem Smartphone oder Tablet-Computer.

Beispiele für mobile VR-Anwendungen sind Samsung GearVR [49] und Google Cardboard [21]. Im Falle von Samsung GearVR und Google Cardboard wird zusätzlich eine Haltevorrichtung für das verwendete Gerät eingesetzt, die bei GearVR auch als Controller fungiert und durch ein integriertes Linsensystem das Sichtfeld des Benutzers erhöht.

Web-VR-Applikationen: Anwendungen für das Web wie etwa WebVR [37] erlauben den Zugriff auf Virtual-Reality-Geräte wie Head-Mounted-Displays durch einen Browser. Auf diese Weise können Web-Inhalte mit VR-Hardware konsumiert werden.

Die Entscheidung, für das vorliegende Projekt auf eine Desktop-VR-Anwendung zu setzen, wurde getroffen, weil mobile und Web-VR-Applikationen die folgenden, vom Projektteam als unverzichtbar eingestuften, Voraussetzungen nicht erfüllen konnten.

- Es sollte möglich sein, die in 2.3 beschriebenen Handtracking Interaktionsmethoden für die Verwendung von haptischen Würfel-Markern zu verwenden.
- Außerdem sollte die verwendete Architektur ein zuverlässiges Echtzeit-Tracking mit geringer Latenz von mindestens einem Dutzend Markern erlauben, wie es in 2.6 beschrieben wird.
- Und schließlich sollte *MARc* die Basis bereitstellen, um später auch mit mehreren Benutzern verwendet werden zu können.

2.2 Haptische Interaktionsmethoden in VR Umgebungen

Laura

Um das Eintauchen in die virtuelle Welt zu erleichtern, kann es bei manchen VR-Anwendungen sinnvoll sein, dem Benutzer eine Form von haptischem Feedback zu geben. Die meisten Systeme haben ihren Ansatzpunkt an den Fingern, oder genauer gesagt an den Fingerspitzen. Es gibt aber auch Systeme die mehrere Parts des Körpers oder sogar den ganzen Körper mit einbeziehen.

Man unterscheidet zwischen *Active Feedback Devices* und *Passive Feedback Devices*. Während die aktive Variante dynamisch veränderliche Kräfte erzeugt, die dem Benutzer entgegenwirken, werden die passiven Varianten immer nur durch den Benutzer selber bewegt [23].

Um eine genauere Unterteilung vorzunehmen, kann man die verschiedenen Systeme zudem nach den einzelnen haptischen Wahrnehmungen unterteilen. Eine mögliche Aufteilung ist im Folgenden zu sehen.

Force Feedback Systeme: Ausübung eines Kraftvektors auf eine Körperstelle des Benutzers, um seine Bewegung zu erschweren oder aktiv eine Bewegung zu erzwingen.

Tactile Feedback Systeme: Stimulation der Berührungsempfindung der Haut. Im Allgemeinen wird ein variabler Druck auf eine Hautstelle ausgeübt, welcher nicht direkt von der Bewegung des Körperteils abhängig ist.

Greifen: Kann als Kombination der beiden vorangegangen Ansätze angesehen werden. Befindet sich ein virtueller Gegenstand in der Hand so soll hierbei verhindert werden können, dass der Benutzer seine Hand schließt.

Proprioceptive Feedback Systeme: Im Gegensatz zu den *Tactile Feedback Systems* spürt der Anwender nicht über seine Haut, sondern über Gelenke bzw. Muskeln.

Bei den oben aufgelisteten Formen von haptischer Interaktion wird davon ausgegangen, dass der Gegenstand um den es geht rein virtueller Natur ist. Eine Besonderheit an *MaRC* ist, dass es sich um reale Objekte (siehe Kapitel 3.1.8) handelt, an der Position virtuelle Objekte gerendert werden können. In diesem *Mixed Reality*-System hat der Benutzer also einen realen haptischen Eindruck. Der reale Würfel dient somit als Anfasser und bietet die Möglichkeit das virtuelle Objekt, was durchaus kleiner oder größer als sein reales Pendant sein kann, zu transformieren.

2.3 Handtracking Interaktionsmethoden

Paul

Nr.	Schicht	Beispiel
7	Anwendung	HTTP, SMTP, FTP, DNS
6	Darstellung	HTTP, SMTP, FTP, NNTP, NetBIOS
5	Sitzung	HTTP, SMTP, FTP, NNTP, NetBIOS, TFTP
4	Transport	TCP, UDP, SPX, NetBEUI
3	Vermittlung	IP IPX
2	Sicherung	Ethernet, ATM, FDDI, TR
1	Bitübertragung	Manchester, 10B5T, Trellis

Tabelle 1: ISO-/OSI-7-Schichtenmodell

2.4 ISO/OSI-7-Schichtenmodell

Laura

Um die Kommunikation zwischen unterschiedlichsten technischen Systemen zu ermöglichen und zu vereinheitlichen dient das *ISO/OSI-7-Schichtenmodell* [27], welches in Tabelle 1 schematisch dargestellt ist. Um die Weiterentwicklung von Kommunikationsmodellen möglichst barrierefrei zu gestalten, sind in dem Modell sieben aufeinanderfolgende Schichten definiert worden, die für einen klar eingegrenzten Teilbereich der Kommunikation zuständig sind. Die Netzprotokolle, die in einer Schicht zum Einsatz kommen, müssen einheitliche Schnittstellen aufweisen, um einen reibungslosen Austausch zu gewährleisten. Entsprechende Beispiele sind der rechten Spalte von Tabelle 1 zu entnehmen.

Während die Schichten 1–4 als transportorientierte Schichten einzustufen sind, können die verbleibenden Schichten 5–7 als anwendungsorientiert angesehen werden. Da der Austausch von Daten für die Umsetzung von *MaRC* im Vordergrund steht, wird im Folgenden besonders auf die vierte Schicht eingegangen. Dabei werden die beiden Übertragungsprotokolle *Transmission Control Protocol* (TCP) und *User Datagram Protocol* (UDP) vorgestellt. Auf die Aufführung weiterer Übertragungsprotokolle wird bewusst verzichtet.

Die Transportschicht stellt eine logische Ende-zu-Ende-Verbindungen dar und dient als Bindeglied zwischen den transportorientierten und anwendungsorientierten Schichten [27].

TCP: Dieses Transportprotokoll ist verbindungsorientiert und paketvermittelt. Genaue Details können in dem Standard RFC 793 [3] von 1981 in Erfahrung gebracht werden.

UDP: Dieses Transportprotokoll wurde aus der Notwendigkeit heraus entwickelt, für die Übertragung von Sprache auf ein, im Gegensatz zur TCP, einfacheres

Protokoll zurückgreifen zu können. Genaue Details können in dem Standard RFC 768 [45] von 1981 in Erfahrung gebracht werden.

Ein wichtiger Unterschied zwischen den beiden Transportprotokollen ist, dass die Übertragung per TCP im Vergleich zu UDP sicherstellt, dass gesendete Daten korrekt übertragen und empfangen werden. **QUELLE??** Dies geschieht dadurch, dass in einem TCP-Socket-Netzwerk Fehlererkennungs- und Korrekturmechanismen enthalten sind, die fehlerhafte Übertragungen der darunterliegenden Internet Protocol (IP) Schicht ausgleichen, also entweder korrigieren oder dafür sorgen, dass fehlerhafte Daten erneut übertragen werden.

2.5 Bedienkonzepte in VR-Umgebungen

Lukas

Die Bedienung eines Systems – vor allem Dateneingabe und Objektmanipulation – in einer Virtual Reality (VR) Umgebung unterscheidet sich stark von der konventionellen Bedienung mit Maus und Tastatur [8]. Das Hauptziel einer VR-Umgebung ist Immersion, also das „Eintauchen“ des Benutzers in die virtuelle Welt. Ein Teil zur Erreichung dieses Ziels sind Methoden, mit denen der Benutzer mit der virtuellen Welt – möglichst intuitiv – interagieren kann.

Nach [50] gibt es vier unterschiedliche Manipulationsmethoden für VR-Anwendungen:

Direct User Control: Der Benutzer interagiert mit Objekten in der virtuellen Welt so, wie dieser es auch in der realen Welt tun würde. Ein Beispiel hierfür ist eine Geste wie das Schließen der Hand zu einer Faust, um einen Gegenstand in der virtuellen Welt „anzufassen“, sodass dieses Objekt anschließend der Bewegung der Hand folgt.

Physical Control: Manipulationen in der virtuellen Welt geschehen durch die Interaktion des Benutzers mit einem realen Gegenstand. Ein solcher Gegenstand kann beispielsweise ein Lenkrad oder ein Joystick sein.

Virtual Controls: Bei *Virtual Controls* handelt es sich nicht direkt um eine Manipulationsmethode, viel mehr beschreibt der Begriff die in der virtuellen Welt vorhandenen Interaktionsmodule, wie etwa Buttons, die gedrückt werden können. Um tatsächlich zu interagieren muss der Benutzer eine der drei anderen Manipulationsmethoden verwenden.

Agent Controls: Der Benutzer interagiert mit der virtuellen Welt über einen „intelligenten Mittelsmann“. Dieser Mittelsmann kann eine Person oder ein vom Computer kontrolliertes Wesen sein.

In den Abschnitten 4.6, 4.6.1 und 4.7 wird auf die in *MArC* verwendeten Interaktions- und Manipulationsmethoden eingegangen.

2.6 Marker Tracking

Vera

In einer VR oder AR Umgebung ist zur interaktiven Positionierung eines 3D Objektes die präzise Bestimmung der Orientierung und Position im 3D-Zielraum notwendig. Zur Lösung dieses Problems muss zunächst ein physisches Objekt in der realen Welt erkannt, zugeordnet und verfolgt werden. Demzufolge ist es notwendig dieses physische Objekt mit einer Videokamera auf zu nehmen. In den resultierenden Bildsequenzen werden die physischen Objekte anhand von festgelegten Merkmalen erkannt. Diese Merkmale können sowohl natürlicher Art sein oder als künstlich erstellten Codes definiert sein. Ein Objekttracking mit natürlichen Merkmalen wird in der Literatur auch als *Markerless Tracking* bezeichnet, während die Verwendung von Codes, beziehungsweise Bildmarken, als *Markerbased Tracking* bekannt ist.

Natürliche Merkmale zur Identifizierung der Objekte sind Textureigenschaften, Kanteninformationen oder sogenannte Keypoints. Eine der robustesten und simpelsten Methoden ist die Verwendung von Keypoints [4][58][9][32], die sowohl in der Ausgangs- als auch in der Kamerawelt bekannt sind. Diese werden mit Hilfe einer Homographie zur Übereinstimmung gebracht. Daraus resultiert die notwendige Transformation zwischen den Welten (siehe Kapitel 5.2). Ein bedeutender Nachteil der Verwendung von Keypoints oder Texturbasierten Verfahren ist, dass sie nur für Objekte mit einem hohen Grad an Texturmerkmalen und großen Gradientenbeträgen geeignet sind. Aus diesem Grund wurden auch Verfahren [24][12][44] entwickelt die sich speziell für texturarme Objekte eignen, wie etwa die in *MArC* verwendeten grünen Rechtecken der Würfel Marker (siehe Kapitel 3.1.8). Andere Autoren verwenden sehr rechenintensive Kantenerkennungsalgorithmen, wie den *Moving Edges Algorithm* [48]. Eine weitere Weiterentwicklung der Kantenbasierten Methoden ist das Modelbased Tracking bei dem die detektierten Kanten eines möglichen Kandidaten mit den 3D-Kantenmodellen des zu verfolgenden Objektes abgeglichen wird [56][33][60][5].

Im Gegensatz zu dem Markerbased Tracking benötigt diese Art des Trackings keine Veränderung der realen Welt und die Parameter, welche den Tracking-Algorithmus beeinflussen können nicht ohne weiteres kontrolliert werden [4]. Dennoch ist die markerbasierte Detektierung von Objekten sehr rechenaufwändig und eine eindeutige Zuordnung beziehungsweise Identifikation von ähnlichen oder gleichförmigen Objekten, wie den Würfel Marker ist sehr aufwändig und nahezu unmöglich. Aus diesem Grund ist es für ein System wie *MArC* zum derzeitigem Zeitpunkt sinnvoller die Würfel Marker mit codebasierten Mustern zu erweitern, die auch nach längerer Verdeckung eine eindeutige Zuordnung von Würfel Marker ermöglichen. In Abbildung 1 sind vielfältige Beispiele von binären Codes zu sehen, die zum Marker Tracking verwendet werden. Für *MArC* sind vor allem rechteckige binäre Muster vorteilhaft, da aus dem äußeren vier Ecken auch die Orientierung des Würfel Markers im Raum abgeleitet werden kann. Während der Inhalt des Muster zur eindeutigen Identifizierung des Würfels beiträgt.

Bekannte markerbasierte Verfahren mit rechteckigen binären Mustern sind das be-

kannte *QR* Verfahren, *ARToolKit*[28], *ARToolKit Plus*[59], *ARTag* [16], *BinARyID*[18] und *ArUco*[19]. Wie in Abbildung 1 zu sehen ist, sind bis auf *BinARyID* und *ArUco* alle Muster sehr detailreich und komplex. Diese Eigenschaft ist auf die höhere Bitgröße der Codes zurückzuführen macht die Erkennung in Aufnahmen aus größerem Abstand und gegebenenfalls mit Bewegungsunschärfe ungleich schwerer und es kommt zu häufiger zu Fehlinterpretationen und Ausfällen. Gerade in einem System wie *MArC* tritt Bewegungsunschärfe sehr häufig aus, wenn die Würfel Marker verschoben werden. Auch der verhältnismäßig große Abstand der Kamera zu den Würfel Markern lässt die Marker Muster im Kamerabild relativ klein werden und somit wirkt sich die Bewegungsunschärfe noch deutlicher auf die feinen Codemuster aus. Darum eignen sich hier vor allem Muster mit geringer Bitanzahl und einfacheren groben Mustern, wie zum Beispiel *ArUco* Marker mit maximal 4 bit Codierung. Diese Marker Bibliothek ist ein *OpenCV* Modul (siehe Kapitel 3.3.3), welche alle notwendigen Funktionalitäten und Ressourcen für das Tracking und die Orientierungsbestimmung enthält. Ein weiterer großer Vorteil dieser Bibliothek ist, dass die Bitgröße und die maximale Anzahl der Identitäten explizit ausgewählt werden kann.



Abbildung 1: Diverse Binäre Muster die als Code für Markerbasiertes Tracking verwendet werden. Quelle: [19]

CGPC6	Beschreibung
Prozessor	Intel Core i7 6700 CPU @ 4 × 3.4 – 4.0 GHz
Arbeitsspeicher	16 GB
Grafikkarte	NVIDIA GeForce GTX 980
Betriebssystem	Windows 10 Education 64 bit
Schnittstellen	2× USB 3.0, 5× USB 2.0, 1× HDMI

Tabelle 2: Übersicht der technischen Daten des Computers für die Unity-Simulation.

3 Materialien

In den nachfolgenden Abschnitten, werden Werkzeuge und Hilfsmittel beschrieben, die für die Fertigstellung des Projekts voneinander waren.

Des weiteren werden auch Komponenten vorgestellt, die während der Projektlaufzeit erstellt wurden, wie etwa die Würfel-Marker (s. Abb. 5).

3.1 Hardware

Zur Ausführung der *MArC*-Software sind diverse Hardware-Komponenten Voraussetzung. Diese Komponenten werden nachfolgend beschrieben und deren Kontext im System näher erläutert.

3.1.1 Computer zur Ausführung der Unity-Simulation

Lukas

Die Anwendung, welche aus Unity [52] heraus erstellt wurde, benötigt einen Host-Computer, welcher sowohl mit dem HTC Vive Head-Mounted Display kompatibel, als auch leistungsstark genug sein muss, um das Rendering der Simulation mit ausreichend hoher Bildrate ausführen zu können.

Für das vorliegende Projekt wurde seitens der Technischen Hochschule Köln ein Computer zur Verfügung gestellt. Die technischen Daten des Geräts sind in Tabelle 2 aufgeführt.

Die Hard- und Software-Voraussetzungen für die Ausführung der Unity-Anwendung in Verbindung mit der HTC Vive, welche in Tabelle 3 aufgelistet sind, werden von dem verwendeten Computer übertroffen.

3.1.2 Computer zur Ausführung der Tracking-Anwendung

Vera

Auf Grund der begrenzten Bandbreite einer USB-Karte ist es zwingend notwendig einen weiteren Rechner an das Gesamtsystem zu koppeln, welcher ausschließlich für

HTC Vive	Systemvoraussetzungen
Prozessor	mindestens Intel Core i5-4590 oder AMD FX 8350
Grafikkarte	mindestens NVIDIA GeForce™ GTX 1060 oder AMD Radeon™ RX 480
Arbeitsspeicher	mindestens 4 GB
Videoausgang	1× HDMI 1.4-Anschluss oder DisplayPort 1.2
USB	1× USB 2.0-Anschluss
Betriebssystem	Windows 7 SP1, Windows 8.1 oder Windows 10

Tabelle 3: HTC Vive Systemvoraussetzungen. [26]

Acer E5-571G-795A	Beschreibung
Prozessor	Intel Core i7-5500U CPU @ 2 × 2.4 – 3.0 GHz
Arbeitsspeicher	8.0 GB
Grafikkarte	NVIDIA GeForce 840M
Betriebssystem	Windows 10 Home, 64 bit
Schnittstellen	2× USB 2.0, 1× RJ45-Netzwerkanschluss

Tabelle 4: Auszug aus dem technischen Datenblatt des Acer E5-571G-795A.

die Ansteuerung der uEye-Kamera und die Berechnungen des Tracking-Algorithmus zuständig ist. An den Computer zur Ausführung der VR Umgebung sind gezwungenenmaßen viele externe USB Komponenten, wie zum Beispiel die *Leap Motion* und die *HTC Vive*, angeschlossen. Dies führt zu einer hohen Auslastung der Bandbreite der USB-Karte und auf Grund dessen ist es nicht mehr möglich die uEye-Kamera mit der notwendigen maximalen Framerate zu betreiben. Somit wird für ein flüssiges und real-time fähiges Tracking der Acer E5-571G-795A mit den Eigenschaften aus Tabelle 4 verwendet.

3.1.3 HTC Vive

Laura

Bei der *HTC Vive* handelt es sich um ein Head-Mounted Display, welches von *HTC* in Kooperation mit *Valve* [57] produziert wird. Vorgestellt wurde dieses am 1. März 2015 im Vorfeld des *Mobile World Congress* [36].

Die Auflösung des Displays beträgt insgesamt 2160×1200 Pixel, was 1080×1200 Pixeln pro Auge entspricht. Die Brille bietet ein Sichtfeld von bis zu 110° bei einer Bildwiederholrate von 90 Hz [25]. Alle technischen Systemvoraussetzungen können in Tabelle 3 eingesehen werden.

Zur Positionsbestimmung im Raum wird die Lighthousetechnologie von *Valve* genutzt. Zusätzlich sind neben einem Gyrosensor auch ein Beschleunigungsmesser und ein Laser-Positionsmesser verbaut. Mittels speziellen Game-Controllern wird eine Interaktion mit virtuellen Objekten ermöglicht.

3.1.4 IDS uEye 164LE-C

Vera

Die Kamera *uEye 164LE-C* wurde vom Hersteller *IDS Imaging Development Systems* entwickelt. Sie hat eine Auflösung von 1280×1024 Pixel und ermöglicht Live-Video-Aufnahmen im RGB Farbmodus mit maximal 25 fps. Der integrierte CMOS Bildsensor wird im Rolling Shutter betrieben und ermöglicht Belichtungszeiten von $37\mu s$ bis 10s. Weiterführend kann sie universell mit allen gängigen Computern oder Systemen via USB 2.0 Schnittstelle verbunden werden [1].

Die erforderliche Ansteuerung der *uEye 164LE-C* erfolgt mit Hilfe der bereit gestellten *IDS Software Suite*. In diese Suite ist die *uEye-API* integriert, welche die Entwicklung von eigenen Programmen unter *Windows* und *Linux* mit den Programmiersprachen *C++*, *.NET*, *C#* oder *C* ermöglicht [2]. Für das Tracking der Marker in *MArC* wurde eine eigene Schnittstelle in *C++* erstellt, welche die Kamera im Live Modus initialisiert und steuert.

3.1.5 Leap Motion

Paul

Bei der *Leap Motion* [29] handelt sich um ein $7,6 \times 3 \times 1,3\text{ cm}$ großes Gerät, welches es mit Hilfe von Sensoren möglich macht, Hand- und Fingerbewegungen zu tracken und diese als Eingabemöglichkeit zu nutzen. Die Idee dahinter ist, eine Eingabegerät im virtuellen Raum analog zu Maus zu schaffen, welches keinen direkten Kontakt bzw. keine Berührung benötigt. Hergestellt wird die *Leap Motion* von der amerikanischen Firma Leap Motion Inc. Gegründet wurde die Firma am 1. November 2010. Wie auf Abbildung 2 gezeigt, besteht das Gerät im wesentlich aus zwei integrierten weitwinkel Kameras und drei einfachen Infrarot LEDs. Die LEDs haben jeweils eine Wellenlänge von 850 nm . Der durch die beiden Kameras aufgespannte Interaktionsraum der *Leap Motion* ähnelt einer umgedrehten Pyramide, mit einem Flächeninhalt von knapp 243 cm^2 . Diese Reichweite ist durch die Ausbreitung der LED Lichter räumlich begrenzt. Die Lichtintensität der LEDs ist wiederum durch den maximalen Strom, der über die USB-Verbindung fließt beschränkt.

Für das Projekt wurde die *Leap Motion* zur Interaktion mit den virtuellen Menüs verwendet. Dabei wurde das Gerät an der *HTC Vive* befestigt und so ein Interaktionsraum vor dem Gesicht des Nutzers aufgespannt.



Abbildung 2: Explosionszeichnung der Leap Motion. [30]

3.1.6 Leap Motion SDK

Paul

Leap Motion Inc. bietet ein vollständiges Software Development Kit (SDK) für die *Leap* an, welches *Unity* um das Hand- und Fingertracking erweitert. Die Treibersoftware interpretiert die von den Kameras gelieferten Daten und sendet Trackinginformationen an die jeweilige Software. Das Leap SDK setzt an dieser Stelle an und verwendet diese eingehenden Daten um sie auf ein Handmodell in *Unity* zu übertragen. Das SDK ist so vorbereitet, dass der Nutzer fertige Bausteine in die virtuelle Szene einbinden kann, die sich dann um die Dateninterpretation und visualisierung als Handmodelle kümmert. Die Handmodelle, die das SDK liefert, sind mit Collidern ausgestattet. Diese abstrahierte Geometrie lässt *Unity* Kollisionen feststellen. Dieses Feature wird bei *MArC* verwendet, um die Interaktion mit den virtuellen Menüelementen zu realisieren.

Leap bietet zudem mehrere sog. "Modulen", die das SDK erweitern können. Unter anderem auch ein Modul, welches die *Unity*-internen GUI Elemente mit den Handmodellen bedienbar machen. Nach einigen Tests hat sich jedoch herausgestellt, dass diese Interaktionsmöglichkeit für *MArC* ungeeignet ist, da sie zu instabil läuft. Das um das UI Modul erweiterte SDK projiziert die Position der Fingerspitzen auf die UI Ebene, falls sich die Hand in der Nähe dieser befindet. Auf der UI wird dann ein kleiner Kreis angezeigt, um dem Nutzer ein visuelles Feedback zu gewährleisten. Im Falle von *MArC* sollten die virtuellen UI Elemente auf den Tisch platziert werden. Hierbei sind die Elemente jedoch immer ein wenig oberhalb des Tisches platziert, damit sie in jedem Fall mit der virtuellen Hand bedient werden können. Zwingend durchdringt die Hand dann die UI Elemente. Mittels des mitgelieferten UI-Modules ist eine Interaktion dann nicht mehr möglich. Daher war eine Bedienung mit Hilfe der Collider sinnvoller.

3.1.7 ArUco Marker

Vera

Die ArUco Bibliothek ist ein Marker Tracking Modul von *OpenCV* (siehe Kapitel 3.3.3) kann für Augmented Reality (AR) Anwendungen genutzt werden und stellt für diese Anwendungen alle notwendigen Funktionalitäten zum Orten und Verifizieren der Codes sowie der anschließenden Pose Estimation der ermittelten Positionen zur Verfügung [19]. Die Marker bestehen ähnlich wie QR-Codes aus einer zweidimensionalen Matrix, mit schwarzen oder weißen Feldern, welche die kodierten Daten binär, wie in Abbildung 3, darstellen. Weiterführend kann die Anzahl der Bits variabel gewählt werden (siehe Abbildung 4), je nachdem wie groß die gefragte Markeranzahl ist oder deren erforderliche Erkennbarkeit in sehr großen Entfernung sowie kleinen Bildern. Um diese Vielzahl an verschiedenen Größen und IDs händeln zu können wurden sogenannte Dictionarys eingeführt [20]. Diese Dictionarys bestehen aus Markern mit gleicher Bit-Anzahl und sind zusätzlich auf eine maximalen Anzahl von IDs begrenzt um ein möglichst hohe Performanz zu gewährleisten.

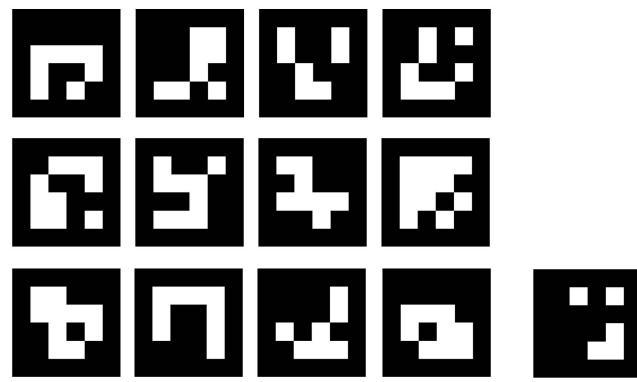


Abbildung 3: Alle genutzten 16 bit ArUco Marker des Prototypen. Links die zwölf IDs der Würfel Marker und rechts die ID, welche zur Kalibrierung benötigt wird. Die maximale Anzahl der Marker ist auf 50 begrenzt.

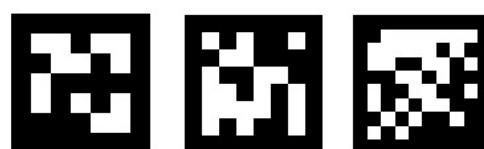


Abbildung 4: ArUco Marker mit unterschiedlicher Bitgröße. Von Links nach Rechts: $n = 5$, $n = 6$ und $n = 8$. Quelle: [19]

3.1.8 Würfel Marker

Vera

In vielen VR oder AR Umgebungen müssen die Nutzer eines Systems häufig nach virtuellen Objekten zur Interaktion greifen, die nicht real existieren. Demzufolge greifen die Personen ins Leere, was häufig die Immersion stört und zu Irritationen und Unsicherheit führt. Um dem Nutzer des Systems *MArC* für die Positionierung und Orientierung ein reales haptisches Feedback zu ermöglichen wurden zwölf Aluminiumwürfel mit aufgeklebten Markern designt. Diese Würfel können beliebig innerhalb eines zuvor festgelegten Bereiches auf dem realen Tisch verschoben und rotiert werden. An der aktuellen Position und Orientierung des jeweiligen Markers wird in der VR ein explizit zugeordnetes Objekt gerendert. Diese Position wird mit Hilfe des Tracking-Algorithmus (siehe Kapitel 5) aus den Bildern der uEye Kamera ermittelt. Alle zwölf Marker stimmen mit der Form und Farbe, sowie Material und Oberflächenbeschaffenheit aus Abbildung 5 überein. Sie haben eine Kantenlänge von 46 mm und sind in einem Winkel von 45° an allen Kanten gefräst. Das Aluminium ist glasperlgestrahlt um eine matte Oberfläche zu erzeugen, welche ungewollte Reflexionen und Überstrahlungen vermeidet, die unter Umständen den Tracking-Algorithmus beeinflussen können.

Auf die Oberseite des Markers ist mittig ein leuchtend grünes Quadrat mit einer Kantenlänge von 40 mm aufgebracht. Diese grüne Fläche wird für ein Green-Keying benötigt, welches die Verfolgung der Marker auch bei Bewegungsunschärfe ermöglicht. Die leuchtend grüne Farbe wurde ausgewählt, da sie auf Grund ihrer hohen Leuchtkraft selten in der Natur und vor allem nicht in der Hautfarbe vorkommt. So hebt sie sich stark von ihrer Umgebung ab und erleichtert das Segmentieren der grünen Fläche im Bild. Die rechteckige Form der grünen Flächen hat noch eine weitere Bedeutung. Auf Grund der Geometrie können die äußeren Eckpunkt wie bei den rechteckigen codebasierten Markern auch zur Berechnung der Orientierung des Würfelmachers im Kameraraum verwendet werden.

Ebenfalls mittig ist jeweils ein individueller 35 mm großer 16 bit ArUco-Marker plan befestigt. Alle verwendeten ArUco-Marker haben einen Rand von einem Bit hat und wurden jeweils aus dem Marker Dictionary DICT_4X4_50 des Arcuo Moduls [40] generiert. Die maximale Anzahl an IDs von 50 ist ausreichend für den Prototypen von *MArC*, da die Anzahl von 50 Würfelmakern die durchschnittliche Fläche mehr als ausfüllen würde. Ein weiterer Vorteil dieser verhältnismäßig kleinen Dictionarys ist, dass auch der Aufwand für den entsprechenden Abgleich einer ID mit den potentiellen Mustern im Dictionary erheblich reduziert werden kann. Weiterführend beinhaltet das DICT_4X4_50 Dictionary auf Grund seiner 16 bit Codierung sehr grobe und einfache Muster, welche gegen die mögliche Bewegungsunschärfe robuster ist. Bei sehr feinen Strukturen kann es schneller zu einer Verwischung des gesamten Musters kommen und die Wahrscheinlichkeit einer erfolgreichen Erkennung sinkt.



Abbildung 5: Würfel Marker mit grüner Fläche und einem ArUco Marker, die mittig auf den Aluminumwürfel aufgebracht sind.

3.1.9 Schachbrett-Kalibrierungshelfer zur Kamerakalibrierung

Laura

Der in Abbildung 6 gezeigte Kalibrierungshelfer wird für die Kamerakalibrierung (vgl. Kapitel 5.2.3 der uEye-Kamera, deren Eigenschaften in Kapitel 3.1.4 beschrieben werden, verwendet. Dazu wurde eine Tisch-artige Erhöhung gebaut, die genauso hoch ist, wie die Würfel Marker (vgl. Kapitel 3.1.8). Darauf ist ein ausgedrucktes Schachbrettmuster mit 8×10 Feldern, wobei man algorithmus-bedingt nur die inneren Felder zählt, also 7×9 Felder. Dieses steht unter http://www.mrpt.org/downloads/camera-calibration-checker-board_9x7.pdf (Abgerufen am: 25.03.2017) zum Download bereit.

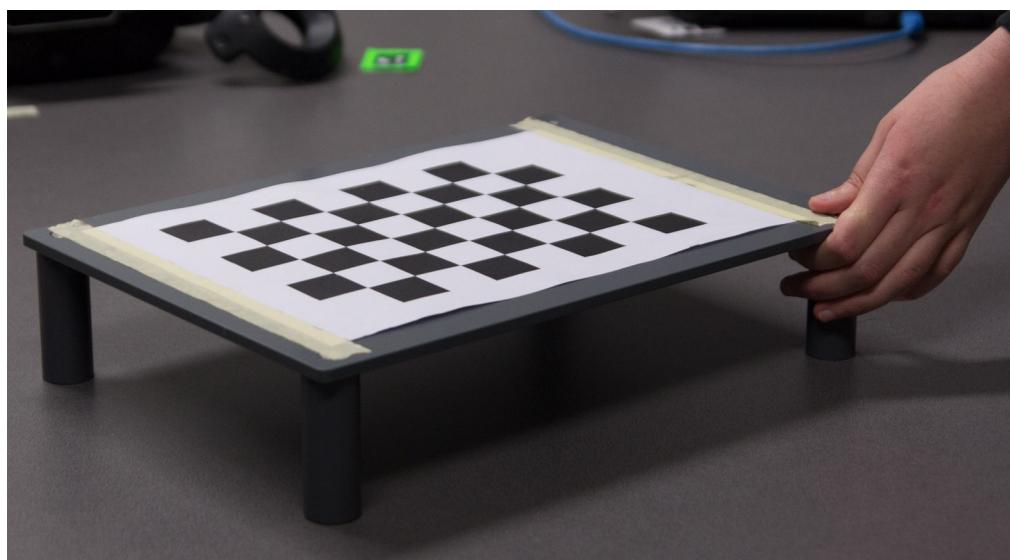


Abbildung 6: Schachbrett-Kalibrierungshelfer.

3.1.10 Kalibrierungscontroller zur Arbeitsbereichskalibrierung

Laura

Für die Kalibrierung des Arbeitsbereiches, die in Kapitel 5.2.4 beschrieben wird, wurde ein Controller der *HTC Vive* wie in Abbildung 9 zu sehen verändert. Zum einen wurde ein *ArUco* Marker mit der ID 49 auf der Oberseite des Controllers befestigt und zum anderen wurde eine Unterlage angefertigt, die verhindern soll, dass der Controller während Kalibrierung wackelt.

Bei der Anbringung des eben erwähnten *ArUco* Markers, der in Abbildung 3 zu sehen ist, ist die genau Positionierung entscheidend. Er muss genau mittig unter dem Ring des Controllers aufgebracht werden, so wie es auf der Abbildung zu sehen ist. Dies ist entscheidend für die das spätere Verfahren, dass auf korrespondierenden Punktpaaren basiert (vgl. Kapitel 5.2.1). Um eine leichtere Aufbringung und gute Ausrichtung des *ArUco* Markers zu vereinfachen, wurde dafür eine kleine Auflagefläche am Kalibrierungscontroller angebracht. Diese ist so angebracht, dass der Mittelpunkt des *ArUco* Markers sich möglichst genau an der Stelle befindet, wo auch die Position des Controllers aus Unity heraus gemessen wird.

3.2 Obsolete Hardware

Lukas

Im Laufe eines Projekts nach Art von *MArC* ist es kaum vermeidbar, dass die gesetzten Projektziele reevaluiert werden müssen. Die Gründe hierfür können vielfältig sein. Beispielsweise könnte die Fertigstellung eines bestimmten Teils des Projekts deutlich länger gedauert haben als geplant, oder es könnte sich herausgestellt haben, dass bestimmte Komponenten zueinander nicht kompatibel sind.

Im vorliegenden Projekt trat eine Kombination der beiden oben genannten Gründe auf. Das Betreiben der *Ovrvision Pro* Stereokamera, welche in Abschnitt 3.2.1 kurz vorgestellt wird, am US-Bus verschiedener während der Entwicklung verwendeter Computer stellte sich als unberechenbar und damit leider unbenutzbar heraus. Die Kamera sorgte während der Ausführung von Unity dafür, dass mit allen anderen Geräten, die ebenfalls per USB angeschlossen waren, unterschiedlichste Probleme auftraten. Als die Situation nach dem Verbinden der Kamera in der teilweisen Zerstörung eines Mainboards gipfelte, wurde die Entscheidung getroffen, die *Ovrvision Pro* nicht länger als Gerät in der Entwicklung von *MArC* zu verwenden.

Stattdessen reifte zu diesem Zeitpunkt die Idee, eine gewöhnliche Webcam zu verwenden, um die Realisierung von Augmented Reality dennoch zu ermöglichen, wenn auch ohne den Stereo-3D-Effekt, welchen die *Ovrvision Pro* nativ bereitgestellt hätte.

Im weiteren Verlauf des Projekts führte eine, lange Zeit ungeklärte, starke Abweichung der Positionen der realen und virtuellen Marker zur Neuordnung der Projekt-prioritäten. Dies hatte zur Folge, dass letztendlich auch die Webcam als Plattform für die Umsetzung der AR-Fähigkeiten von *MArC* aufgegeben wurde und stattdes-

Örtliche Auflösung pro Auge	Zeitliche Auflösung	Bildwinkel	
		Horizontal	Vertikal
2560 × 1920 px	15 fps	115°	105°
1920 × 1080 px	30 fps	87°	60°
1280 × 960 px	45 fps	115°	105°
1280 × 800 px	60 fps	115°	90°
960 × 950 px	60 fps	100°	98°
640 × 480 px	90 fps	115°	105°
320 × 240 px	120 fps	115°	105°

Tabelle 5: Bildmodi der *Ovrvision Pro* Stereokamera. [43]

sen auf eine reine VR-Lösung der Projekt-Problemstellung umgeschwenkt wurde. Nachfolgend werden die Eigenschaften und technischen Daten beider Geräte kurz beschrieben.

3.2.1 Ovrvision Pro

Lukas

Die *Ovrvision Pro* (vgl. Abb. 7) ist eine kompakte Stereokamera, welche über USB 3.0 mit einem Computer verbunden wird. [42] Für die Kamera ist eine Vielzahl an Software-Development-Kits (SDKs) für verschiedene Plattformen und Frameworks, wie etwa Microsoft Windows, Linux, Apple Mac OS X, Unreal Engine oder Unity verfügbar. [41]

Die *Ovrvision Pro* ist speziell auf Augmented Reality (AR) Anwendungen ausgelegt. So unterstützt die Kamera natives Stereo-3D und bietet auch Bildmodi mit hohen Bildwiederholraten, welche für die Verwendung mit VR-Hardware wie Oculus Rift oder HTC Vive notwendig sind, um die Immersion des Benutzers nicht durch zu träge Bewegungswiedergabe zu beeinflussen. Die unterstützten Bildmodi der Kamera sind in Tabelle 5 aufgeführt. Wie aus Abschnitt 3.1.3 hervorgeht, stellt die HTC Vive Bildinhalte mit 1080×1200 Pixeln pro Auge bei 90 Hz Bildwiederholrate da. Diese Leistung wird von der *Ovrvision Pro* nicht erreicht.



Abbildung 7: *Ovrvision Pro* Stereokamera montiert an *Oculus Rift* (links) und *HTC Vive* (rechts). [42]

3.2.2 Webcam

Vera

Die *Creative Senz3D* ist eine RGB Kamera mit einer zusätzlichen Infrarot-Tiefenkamera. Das generierte RGB-Bild hat eine Auflösung von 1280×720 Pixel und das Tiefenbild von 320×240 Pixel bei einer Reichweite von 15 – 99 cm sowie einem Sichtfeld von 74° . Die Kamera wird über eine USB 2.0 Schnittstelle mit einem Computer verbunden und nimmt Videos mit einer Framerate von bis zu 30 fps auf [10]. Weiter ist es möglich die Kamera direkt aus Anwendungen per *Intel Perceptual Computing SDK* anzusteuern.

3.3 Software

Vera

Zur Entwicklung der *MArC*-Software sind diverse Software-Komponenten und Bibliotheken notwendig. Die Funktionalitäten und Verwendung dieser Komponenten werden in diesem Kapitel kurz erläutert.

3.3.1 Unity

Lukas

Unity ist eine sogenannte Spiel-Engine, also eine Entwicklungs- und Laufzeitumge-

bung, die speziell auf die Entwicklung von 3D-Spielen ausgelegt ist. Die Software wurde am 6. Juni 2005 veröffentlicht [22] und wird von Unity Technologies [52] entwickelt und vertrieben. In der Spieleentwicklung ist Unity weit verbreitet, so werden beispielsweise 34 % der kostenfreien Top-1000-Spiele im mobilen Sektor mit Unity entwickelt [54].

Unity bietet eine sehr breite Plattformunterstützung [53] und erlaubt ebenso die Entwicklung für Head-Mounted-Displays, wie etwa die Oculus Rift [38][55] oder auch die in diesem Projekt verwendete HTC Vive [55].

Die zu Beginn des Projekts verwendete Stereo-Kamera *Ovrvision Pro* stellt ein Software-Development-Kit (SDK) für Unity (Version 5) zur Verfügung [41]. Da das endgültige Resultat des Projekts die Verwendung der *Ovrvision Pro* nicht mehr vorsieht, wie in 3.2 beschrieben, wird auf eine weitere Beschreibung dieses SDKs verzichtet.

3.3.2 Visual Studio 2015

Vera

Micosoft Visual Studio 2015 ist eine verbreitete integrierte Entwicklungsumgebung (IDE), welche unter anderem die Programmiersprachen Visual Basic, Visual C#, und Visual C++ unterstützt. Mit Hilfe dieser IDE kann ein Entwickler Win32/Win64 Anwendungen sowie weitere WebApps und Webservices [35] programmieren sowie anschließend compilieren. Für *MArC* wurde mit der Version 14.0.25123.00 Update2 gearbeitet.

3.3.3 OpenCV

Vera

Open Source Computer Vision (OpenCV) ist eine Open Source Bibliothek für Bild- und Videoverarbeitung in der Programmiersprache C++. Vorgestellt wurde sie vor über zehn Jahren von *Intel* und wird seitdem stetig von verschiedenen Programmierern weiterentwickelt. Diese Bibliothek stellt die gängigsten Algorithmen sowie aktuelle Entwicklungen der Bildverarbeitung zur Verfügung [11].

Für dieses System ist vor allem das Modul *calib3d* [39] und das extra Modul *aruco* [40] verwendet. Das erste Modul *calib3d* bietet alle notwendigen Funktionen zur Erstellung, Verwendung und Weiterverarbeitung von intrinsischen und extrinsischen Kamerakalibrierungen an (siehe Kapitel 5.2). Während das Zweite alle benötigten Ressourcen und Funktionalitäten zum Tracken von *ArUco* Markern zur Verfügung stellt (siehe Kapitel 3.1.7).

3.3.4 Steam VR

Paul

3.3.5 Windows Sockets (Winsock)

Lukas

Windows Sockets (abgekürzt Winsock) ist eine API für den Zugriff auf Netzwerkkomponenten in Microsoft Windows Betriebssystemen [46]. Winsock wird nativ in Microsoft Windows bereitgestellt.

Für die unkomplizierte Übertragung zwischen zwei Anwendungen in einem lokalen Netzwerk bieten sich sowohl das *Transmission Control Protocol* (TCP), als auch das *User Datagram Protocol* (UDP) an. Das Erstellen von Netzwerk-Sockets für die Übertragung per TCP und UDP wird von Winsock ermöglicht.

Die Umsetzung der Netzwerkverbindung in *MArC* wird in Abschnitt 4.3 näher beschrieben.

4 System

Lukas

Die Benutzung von *MArC* ist in dem Programm mitgelieferten ReadMe-Datei (vgl. 10.1) beschrieben. Darin wird erklärt, welche Hard- und Softwarekomponenten erforderlich sind und wie das System gestartet und kalibriert wird. Auf diese Aspekte wird in den folgenden Abschnitten näher eingegangen.

Des weiteren enthält die ReadMe-Datei eine Übersicht über die enthaltenen Quellcode-Dateien.

4.1 Aufbau

Lukas und Vera

Der Aufbau des *MArC*-Systems kann in zwei Teile aufgeteilt werden. Der Teil, der für das Tracking der Würfel Marker verantwortlich ist (nachfolgend „Tracking Aufbau“ genannt) und der Teil, welcher die Virtual-Reality-Umgebung erzeugt und die notwendige Peripherie für die Interaktionen stellt (nachfolgend „Unity Aufbau“ genannt). In den beiden folgenden Abschnitten werden diese beiden Teile des Systemaufbaus näher beschrieben.

4.1.1 Tracking Aufbau

Vera

Wie in Abbildung 8 dargestellt wird senkrecht über einem beliebigen Tisch eine Kamera installiert, die Würfel Marker aus der Vogelperspektive filmt. Der Abstand zum Tisch sollte so gewählt werden, dass die Aufnahmen noch scharf sind und sich die Nutzer nicht den Kopf daran stoßen können. Hier ist besondere Vorsicht geboten, da die Nutzer durch die *HTC Vive* nicht die reale Umgebung wahrnehmen können. Für den Prototypen wurde eine *IDS uEye 164LE-C* (siehe Kapitel 3.1.4) verwendet und über eine USB 2.0 Schnittstelle an den Computer mit dem Tracking Algorithmus verbunden. Diese Kamera wird mit Hilfe der uEye-API vom Tracking Algorithmus im Live-Bild-Modus initialisiert und gesteuert. In diesen Live Bildern werden die Würfel Marker erkannt und verfolgt. Für jeden erkannten Würfel Marker werden alle relevanten Informationen über die TCP Netzwerkverbindung (siehe Kapitel 2.4) an den Computer zur Ausführung der *Unity*-Simulation (siehe Kapitel 3.1.1) übertragen. Damit der Algorithmus einen Würfel Marker erkennt muss er in dem vorab definierten Spielfeld bewegt werden. Diese Festlegung findet während der Kalibrierung des Arbeitsbereiches statt. Für diese Kalibrierung ist es notwendig den einzelnen *ArUco* Marker aus Abbildung 3 mit der ID 49 exakt wie in Abbildung 9 auf den Controller der *HTC Vice* zu montieren, da nur so gewährleistet werden kann, dass die erkannte *ArUco* Marker Position in der Kamerawelt mit den Controller Positionen in der *Unity* Welt korrespondieren.

4.1.2 Unity Aufbau

Lukas

Der Teil des Systemaufbaus von *MArC*, der sich vom Computer für die Ausführung der *Unity*-Simulation (s. Abschnitt 3.1.1) über die *Leap Motion*, bis hin zur *HTC Vive* erstreckt, ist ebenfalls in Abbildung 8 dargestellt.

Die Verbindung von *HTC Vive* zum Computer zur Ausführung der *Unity*-Simulation wird sowohl per USB 2.0, als auch per HDMI 1.4 hergestellt. Über die HDMI-Verbindung werden die Bildschirme im *HTC Vive* Head-Mounted-Display (HMD) als ein einzelnes Display auf dem verbundenen Computer eingebunden, genau so wie es auch mit einem normalen Bildschirm geschehen würde. Die USB 2.0 Verbindung des HMD dient hingegen dem Datenaustausch mit *SteamVR*, welches auf dem Computer installiert ist.

Der *Leap Motion* Controller wird über eine USB 3.0 Verbindung, welche per USB 2.0 Verlängerung angeschlossen ist, mit dem Computer verbunden. Dies ist deshalb problemlos möglich, weil die aktuelle Software des *Leap Motion* Controllers die höhere Bandbreite der vorhandenen USB 3.0 Anbindung noch nicht ausnutzt, daher gibt *Leap Motion* an, dass eine uneingeschränkte Nutzung an USB 2.0 möglich ist. [31]

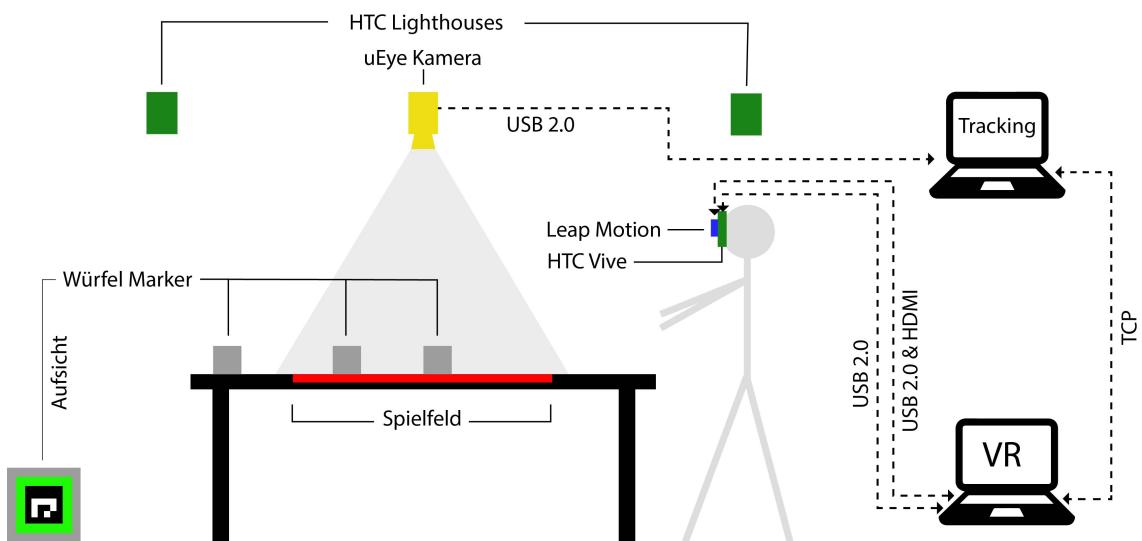


Abbildung 8: Aufbau des *MArC* System.

Starting the System

The following software is needed:

- IDS uEye Driver and SDK (<https://de.ids-imaging.com/download-ueye-win32.html>)
- Steam VR (<http://store.steampowered.com/about/>)
- Leap Driver (<https://developer.leapmotion.com/windows-vr>)
- The OpenCV Framework (<http://opencv.org/downloads.html>)

In order to start the system, make sure the following requirements are met:

- 2 Computers are available:
 - one that runs the IDS uEye tracking application [A] and
 - one for the Unity application [B]
- The HTC Vive head-mounted display has been connected to computer [B]
- The Leap Motion controller has been connected to computer [B]
- The IDS uEye software suite has been installed on computer [A]
- Computers [A] and [B] are connected via a network cable
- The IP address of the ethernet adapter of computer [A] has been set to 192.168.0.7 with the standard subnet mask of 255.255.255.0
- The IP address of the ethernet adapter of computer [B] has been set to 192.168.0.13 with the standard subnet mask of 255.255.255.0
- It is necessary to use the uEye camera with another computer than the rendering, because the performance could be worse while the computer renders the scene and does the tracking at the same time.

Abbildung 10: Auszug aus der *MArC* ReadMe-Datei (vgl. 10.1).

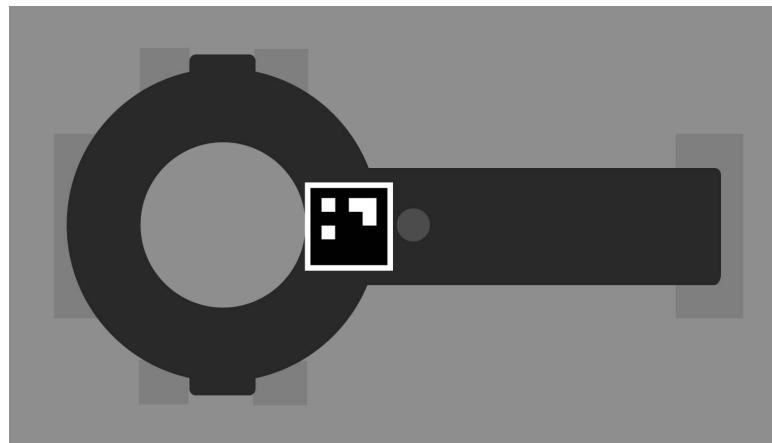


Abbildung 9: Kalibrierungscontroller des *MArC* System mit ArUco Marker.

4.2 Systemvoraussetzungen

Lukas

Die Systemvoraussetzungen für *MArC* sind in der ReadMe-Datei beschrieben, welche der im Projekt erstellten Software mitgeliefert ist. Die gesamte ReadMe-Datei ist in Abschnitt 10.1 zu finden, außerdem enthält Abbildung 10 einen Ausschnitt der ReadMe-Datei, welcher die Voraussetzungen beschreibt, die vor dem Starten des Systems erfüllt sein müssen.

4.3 Netzwerk

Lukas und Laura

Die in Kapitel 3.3.5 beschriebene API stellt die Möglichkeit bereit, den Datentransport mittels TCP oder UDP zu verwenden. Für *MArC* wurde ein TCP-Socket-Netzwerk bestehend aus (genau) einem Server und (genau) einem Client verwendet, weil der in 2.4 beschriebene Fehlerschutz des TCP ausgenutzt werden sollte, um nicht manuell überprüfen zu müssen, ob Daten fehlerfrei übertragen wurden. Da die zu übertragende Datenmenge von *MArC* hinreichend klein – und von konstanter Größe – ist, wurde die etwas höhere Geschwindigkeit durch den geringeren Overhead einer UDP-Socket-Verbindung als unnötig erachtet.

In den beiden folgenden Abschnitten wird die Umsetzung der Netzwerk-Endpunkte an den beiden für *MArC* verwendeten Computern beschrieben.

4.3.1 Server-seitige Netzwerkanbindung

Laura

Aufbauen der Netzwerkverbindung Da die Verbindung zwischen den beiden Rechnern per LAN-Kabel realisiert wurde, wird ein Server gestartet und eine Verbindung seitens des Clients ohne weitere Sicherheitsmechanismen erlaubt, so lange dieser den richtigen Port adressiert. Die entsprechenden Methoden stehen dank *Winsock* (vgl. Abschnitt 3.3.5) durch das Einbinden von `<winsock2.h>` und `<windows.h>` bereit und werden, wie in Quellcode-Auszug 9 (im Anhang einsehbar), verwendet.

Datenübertragung Die Datenübertragung kann grob gesprochen in drei Teilbereiche unterteilt werden. Neben der bidirektionalen Übertragung von verschiedenen Status, müssen zum einen die Tracking-Daten der einzelnen sich im Arbeitsbereich befindlichen Objekte gesendet werden können und zum anderen die Positionen des Kalibrierungscontrollers während der Arbeitsbereichskalibrierung (vgl. Kapitel 5.2.4 empfangen werden.

```

88 void TCP::sendStatus(int status) {
89     const char far* markerPointer = (const char*)&status;
90     send(connectedSocket, markerPointer, 4, 0);
91     printf("Sent Status: %d \n", status);
92 }
```

Quellcode-Auszug 1: `sendStatus()`-Methode in `TCP.cpp`

Da die Übertragung der Status in beide Richtungen erfolgen muss, bedarf es einer Methode zum Senden (Quellcode-Auszug 1) und einer Methode zum Empfangen (Quellcode-Auszug 2). Beide Methoden gehen von einem Status aus, der 4 Byte

groß ist und sind ansonsten selbsterklärend.

```

107 int TCP::receiveStatus() {
108     char far* mPointer = (char*)&m;
109     recv(connectedSocket, mPointer, 4, 0);
110     printf("Received Status: %i \n", m[0].isCalibrated);
111     return m[0].isCalibrated;
112 }
```

Quellcode-Auszug 2: receiveStatus()-Methode in TCP.cpp

Die entsprechende Methode zum Versenden der Marker-Daten, also deren ID, Position, Winkel und Sichtbarkeits-Status, ist in Quellcode-Auszug 3 zu sehen. Alle eben genannten Eigenschaften eines Markers werden zu diesem Zweck in einem struct mit dem Namen **MarkerStruct** (vgl. Quellcode-Auszug 4) gebündelt.

```

99 void TCP::sendMarkerData(std::array<Marker*, 100> allMarkers, std::vector<int> takenIdVec,
100     cv::Mat frame) {
101     getPointerOfMarkerVec(allMarkers, takenIdVec, frame);
102     const char far* markerPointer = (const char*)&ms;
103     send(connectedSocket, markerPointer, 2404, 0);
104 }
```

Quellcode-Auszug 3: sendMarkerData()-Methode in TCP.cpp

Der Vollständigkeit halber befindet sich im Anhang der Quellcode-Auszug 10 der **getPointerOfMarkerVec()**-Methode. Diese dient im Wesentlichen zur Umsortierung der entsprechenden Daten aus dem umfangreicheren **allMarkers**-Array in das **MarkerStruct**-Array. Damit wird dafür gesorgt, dass nur die relevanten Daten übertragen werden.

```

13     struct MarkerStruct {
14         int id;
15         float posX;
16         float posY;
17         float posZ;
18         float angle;
19         int isVisible;
20     };
```

Quellcode-Auszug 4: MarkerStruct in TCP.h

Während der Kalibrierung des Arbeitsbereichs (vgl. Kapitel 5.2.4) wird die Position des Kalibrierungscontrollers nach jedem Auslösen des Triggers von der Unity-Anwendung an den Tracking-Rechner gesendet. Die entsprechende Methode zum Empfangen der dreidimensionalen Position in Unitykoordinaten ist in Quellcode-Auszug 5 abgebildet. Neben dem reinen Empfang der Daten, der äquivalent zu der **receiveStatus()**-Methode funktioniert, werden hier die ankommenden Daten aus dem Array in einen anderen Datentyp konvertiert, um eine Weiterverarbeitung während der Arbeitsbereichskalibrierung zu vereinfachen.

```

116 cv::Point3f TCP::receiveControllerPositions() {
117     cv::Point3d cP;
118     char far* cPPointer = (char*)&cPArray;
119     recv(connectSocket, cPPointer, 12, 0);
120     cP.x = (double)cPArray[0];
121     cP.y = (double)cPArray[1];
122     cP.z = (double)cPArray[2];
123     printf("Received Controller Points: (%f, %f, %f) \n", cP.x, cP.y, cP.z);
124     return cP;
125 }
```

Quellcode-Auszug 5: receiveControllerPositions()-Methode in TCP.cpp

4.3.2 Client-seitige Netzwerkanbindung

Lukas

Aufbauen der Netzwerkverbindung Das Aufbauen der Netzwerkverbindung vom Client zum Server, also von der Unity-Anwendung zur Tracking-Anwendung, geschieht mit Hilfe der von Winsock (vgl. Abschnitt 3.3.5) bereitgestellten Klasse System.Net.Sockets.TcpClient wie in Quellcode-Auszug 6 dargestellt. Die Netzwerkverbindung wird zu einer fest vorgegebenen IP-Adresse hergestellt.

```

69 // Set up and connect TCP socket
70 private void setupSocket(){
71     try{
72         mySocket = new TcpClient(Host, Port);
73         theStream = mySocket.GetStream();
74         socketReady = true;
75         Debug.Log("[TCP] Socket set up successfully.");
76     }catch (Exception e){
77         Debug.LogError("[TCP] Socket setup failed. Error: " + e);
78     }
79 }
```

Quellcode-Auszug 6: setupSocket()-Methode in readInNetworkData.cs

Datenübertragung Nach einer erfolgreich hergestellten Verbindung sieht das Netzwerkmodul in *MarC* sowohl eine Übertragung von verschiedenen Status zur Steuerung des Programmablaufs, als auch die Übertragung der Positionen des Kalibrierungscontrollers während der Arbeitsbereichskalibrierung (vgl. Kapitel 5.2.4) sowie die Übertragung der Tracking-Daten für die Simulation.

Die Übertragung einer Position des *HTC Vive*-Controllers während der Arbeitsbereichskalibrierung wird durch das Drücken der *Trigger*-Taste am Controller ausgelöst. Dadurch wird die Methode `setPosition()` in `TableCalibration.cs` mit der aktuellen Position des Controllers in *Unity*-Koordinaten als Übergabewert aufgerufen. Von dort werden bei erfolgreicher Beendigung der Kalibrierung die vier

Eckpunkte des Arbeitsbereichs abgespeichert und an `calibrationDone()` in `setupScene.cs` weitergegeben. In `setupScene.cs` werden diese Positionen verwendet, um den Arbeitsbereich für den Benutzer zu kennzeichnen. Dies wird in Abschnitt 5.2.4 beschrieben.

```

81     // Send status over TCP according to TCPstatus enum
82     public void sendTCPstatus(int status){
83         if (socketReady) {
84             theStream.Write(System.BitConverter.GetBytes(status), 0, 4);
85             Debug.Log("[TCP] Status sent: " + Enum.GetName(typeof(TCPstatus), status));
86         }
87         else
88             Debug.LogError("[TCP] Failed to send status, because the socket is not ready: " +
89                         status);

```

Quellcode-Auszug 7: `sendTCPstatus()`-Methode in `readInNetworkData.cs`

Das Senden und Empfangen von Status seitens der Unity-Anwendung ist in den Quellcode-Auszügen 7 und 8 dargelegt. Die Methode zum Senden eines Status ist selbsterklärend. Die `receiveTCPstatus()`-Methode prüft zunächst, ob eine Netzwerkverbindung hergestellt, also das Socket bereit ist. Anschließend werden genau vier Bytes vom Netzwerk-Datenstrom gelesen, als 32-Bit-Integer interpretiert und zurückgegeben.

```

91     // Receive status over TCP according to TCPstatus enum
92     public int receiveTCPstatus(){
93         if (socketReady){
94             while (!theStream.DataAvailable){
95                 Debug.Log("[TCP] Waiting for status to be received.");
96                 System.Threading.Thread.Sleep(1000);
97             }
98             byte[] receivedBytes = new byte[4];
99             theStream.Read(receivedBytes, 0, 4);
100            int status = System.BitConverter.ToInt32(receivedBytes, 0);
101            Debug.Log("[TCP] Status received: " + Enum.GetName(typeof(TCPstatus), status));
102            return status;
103        }
104        Debug.LogError("[TCP] Failed to receive status, because the socket is not ready.");
105        return -1;
106    }

```

Quellcode-Auszug 8: `receiveTCPstatus()`-Methode in `readInNetworkData.cs`

Das Empfangen und Interpretieren der Tracking-Daten, nachdem die Simulation gestartet wurde, gestaltet sich ein wenig komplexer und ist im Quellcode-Auszug 11 aufgeführt. Der Auszug wurde aus Platzgründen in den Anhang verschoben.

In der Methode `interpretTCPMarkerData()` wird ein Byte-Puffer konstanter Länge, welcher vorher vom Netzwerk-Datenstrom gelesen wurde, dahingehend interpretiert, dass anschließend die Daten für alle aktiven Tracking-Marker in geeigneter Form zur weiteren Verarbeitung vorliegen. Als geeignete Form ist hier die Klasse `Marker` zu nennen, welche als Attribute die ID, die X- und Y-Position, den Winkel und den Status des jeweiligen Markers enthält.

Beim Interpretieren der Marker werden zunächst die ersten vier Bytes an der von der Schleife abhängigen Pufferposition als ID interpretiert. Sofern die ID gleich -1 ist, wird der aktuelle Schleifendurchlauf beendet, da dieser Marker nicht aktiv (valide) ist. Sollte die ID gleich -2 sein, indiziert dies, dass für das aktuelle Frame alle Marker übertragen wurden und die Schleife abgebrochen werden kann. In allen anderen Fällen ist die ID valide und alle zusätzlichen Eigenschaften des Markers werden aus dem Puffer gelesen und über den BitConverter geeignet interpretiert. Zuletzt wird ein global angelegtes **Marker**-Array in jedem Schleifendurchlauf mit dem verarbeiteten Marker gefüllt. Bei anschließendem Rendern der Simulation wird dann nur noch auf die so aufbereiteten Daten – und nicht mehr auf die Netzwerkdaten – zugegriffen.

4.4 Starten des Systems

Lukas

Nachdem sichergestellt wurde, dass alle in [4.2](#) beschriebenen Voraussetzungen bestehen, kann das System gestartet werden, indem zunächst die Tracking-Anwendung (auf dem einen Computer, vgl. [3.1.2](#)) und anschließend die aus Unity heraus erstellte Anwendung (auf dem anderen Computer, vgl. [3.1.1](#)) gestartet wird. Auf letzterem Computer beginnt darauffolgend die Menüführung, welche in [4.5](#) beschrieben ist.

4.5 Menüführung

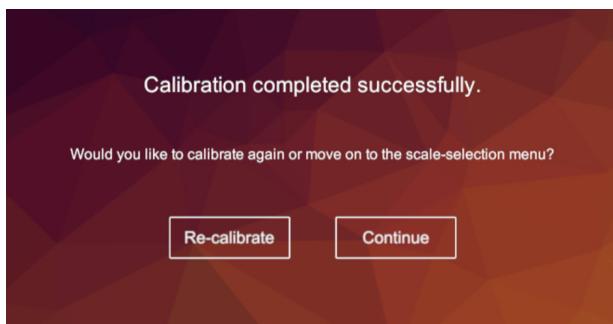
Lukas

Die Menüführung dient dazu, den Benutzer durch alle notwendigen Schritte zu leiten, die vor dem Starten der eigentlichen Simulation erforderlich sind. Im nachfolgenden Abschnitt [4.5.1](#) werden alle verfügbaren Menüs der Anwendung aufgelistet und kurz beschrieben, während im Abschnitt [4.5.2](#) der Ablauf der Menüführung erläutert wird.

4.5.1 Menüs

Laura

Die folgenden Menüs sind Bestandteil der Menüführung:



CalibDone:

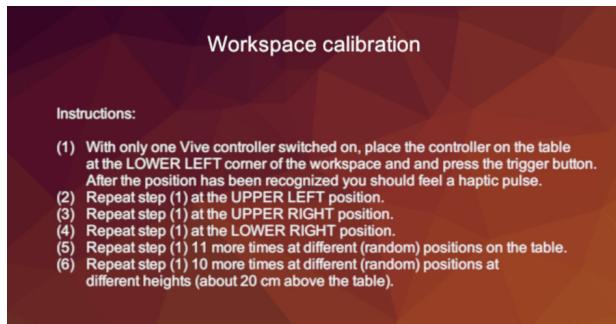
Wird aufgerufen, wenn die Kalibrierung des Arbeitsbereichs abgeschlossen ist. Es informiert den Benutzer, dass die Kalibrierung erfolgreich war und der Vorgang fortgesetzt werden kann.

**CalibrateOrNot:**

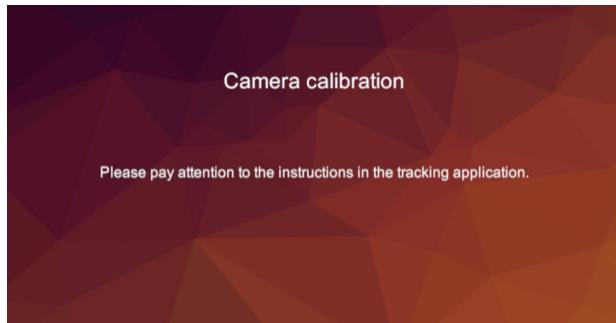
Erscheint nach dem Verlassen des Welcome-Menüs und erlaubt dem Benutzer eine Kalibrierung durchzuführen oder eine bereits durchgeführte Kalibrierung zu laden.

**ControllerNotFound:**

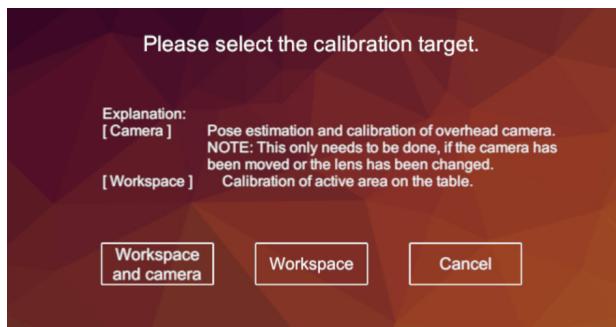
Warnt den Benutzer nach dem Starten der Kalibrierung, dass der HTC Vive Controller, welcher für die Kalibrierung benötigt wird, nicht eingeschaltet ist. Während das Menü angezeigt wird, kann der Benutzer den Controller einschalten und anschließend auf **Continue** klicken.

**doPlaneCalibInVS:**

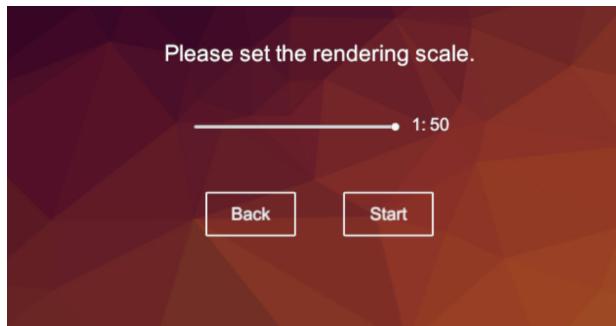
Dient dem Benutzer als Anleitung für die Durchführung der Arbeitsbereich-Kalibrierung. Diese wird in [5.2.4](#) genauer beschrieben.

**doPoseCalibInVS:**

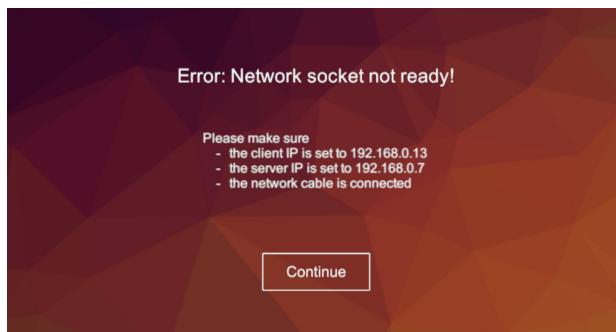
Dient dem Benutzer als Anleitung für die Durchführung der Kamera-Kalibrierung. Diese wird in [5.2.3](#) genauer beschrieben.

**SelectCalibrationTarget:**

Erlaubt die Auswahl der Art der Kalibrierung. Es kann hier entweder nur der Arbeitsbereich oder sowohl der Arbeitsbereich, als auch die Kamera kalibriert werden. Die Kalibrierung ist näher in [5.2](#) beschrieben.

**SetScale:**

Stellt das letzte Menü vor dem Starten der Simulation dar. In diesem kann der Benutzer den Maßstab der Gebäudesimulation einstellen und anschließend die Simulation starten.

**SocketNotReady:**

Warnt den Benutzer nach dem Verlassen des Welcome-Menüs, dass die Netzwerkverbindung zwischen den beiden Computern nicht bereit ist. Nach Bestätigung dieses Hinweises durch einen Klick auf Continue, kehrt der Benutzer zum Welcome-Menü zurück.

**Welcome:**

Erscheint als erstes Menü. Hier erhält der Nutzer eine kurze Information darüber, wie die Anwendung heißt und wozu sie dient.

4.5.2 Ablauf der Menüführung

Lukas

Der Ablauf der Menüführung von *MArC* ist in Abbildung [11](#) dargestellt. Die einzelnen Menüs sind bereits in [4.5.1](#) beschrieben worden.

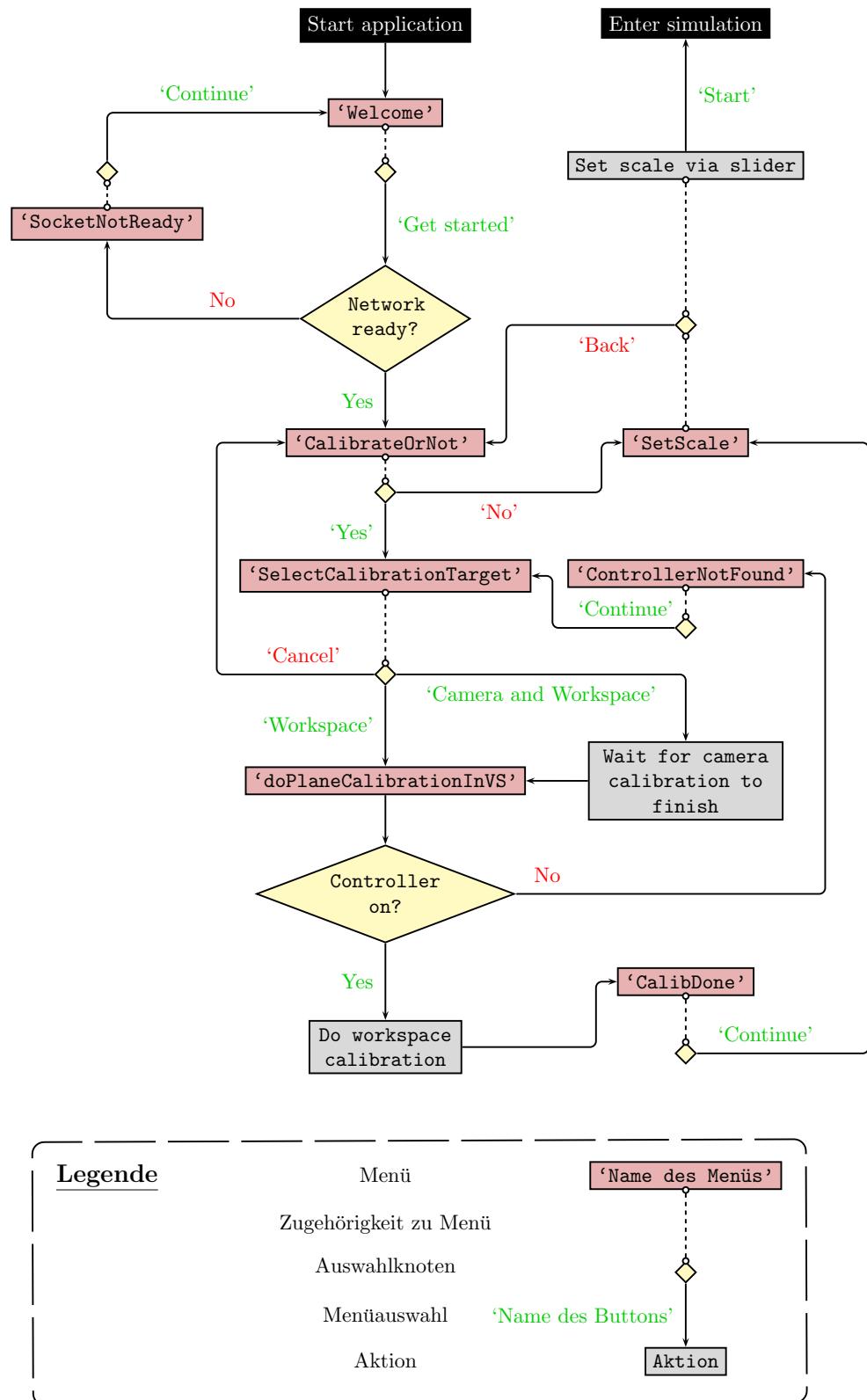


Abbildung 11: Flussdiagramm der Menüführung.

Nach dem Starten der Anwendung wird zunächst das Menü `Welcome` angezeigt. Dieses enthält nur einen Button `Get started`. Sobald dieser gedrückt wird, prüft die Anwendung, ob eine Netzwerkverbindung zu dem Computer mit der Tracking-Anwendung besteht. Sollte dies nicht der Fall sein, wird das Menü `SocketNotReady` angezeigt. Dieses verlässt der Benutzer über einen Klick auf `Continue`, anschließend wird erneut das Menu `Welcome` angezeigt. Wenn zu diesem Zeitpunkt die Netzwerkverbindung korrekt hergestellt wurde, gelangt der Benutzer zum Menü `CalibrateOrNot`, anderenfalls wird wiederholt `SocketNotReady` angezeigt.

In `CalibrateOrNot` hat der Benutzer die Auswahl zwischen den Schaltflächen `Yes` und `No`. Bei einem Klick auf `Yes` wird anschließend `SelectCalibrationTarget` angezeigt, bei einem Klick auf `No` lädt das System eine zuvor durchgeführte Kalibrierung und das Menü `SetScale` wird geöffnet.

`SelectCalibrationTarget` stellt den Benutzer vor die Wahl entweder nur den Arbeitsbereich (*Workspace*) oder sowohl den Arbeitsbereich als auch die Kamera zu kalibrieren (*Camera and Workspace*). Außerdem besteht die Möglichkeit über `Cancel` zum Menü `CalibrateOrNot` zurückzukehren.

Wählt der Benutzer *Camera and Workspace* in `SelectCalibrationTarget` aus, so informiert die Anwendung die Tracking-Anwendung auf dem anderen Computer und wartet anschließend darauf, dass von dort die Bestätigung gesendet wird, dass die Kamerakalibrierung abgeschlossen ist. Anschließend wird das Menü `doPlaneCalibrationInVS` angezeigt, welches auch aufgerufen wird, wenn der Benutzer *Workspace* in `SelectCalibrationTarget` wählt.

Im Menü `doPlaneCalibrationInVS` wird zunächst geprüft, ob der für die Kalibrierung notwendige HTC Vive Controller eingeschaltet ist. Sollte dies nicht der Fall sein, wird `ControllerNotFound` aufgerufen. Dieses kann mit einem Klick auf `Continue` verlassen werden, woraufhin wieder `SelectCalibrationTarget` angezeigt wird. Sofern der HTC Vive Controller beim Aufruf von `doPlaneCalibrationInVS` eingeschaltet ist, wird nach Durchführung der Kalibrierung des Arbeitsbereichs das Menü `CalibDone` angezeigt.

`CalibDone` kann über einen Klick auf `Continue` verlassen werden und führt den Nutzer anschließend zu `SetScale`. Aus diesem Menü kann über den Button `Back` entweder zu `CalibrateOrNot` zurückgekehrt oder die Simulation mit dem im Menü über den Slider eingestellten Maßstab gestartet werden.

4.6 Kontextmenü

Laura und Lukas

Die virtuellen Objekte, die den in Kapitel 3.1.8 beschriebenen Aluminiumwürfeln zugeordnet werden, sind zunächst würfelförmig und können zur Laufzeit der Anwendung vom Benutzer verschoben, rotiert und skaliert werden. Die Verschiebung und Rotation kann über eine entsprechende Veränderung der Lage des realen Marker Würfels geschehen. Für die Skalierung kann wie in Kapitel 4.6.2 beschrieben das sogenannte Kontextmenü aufgerufen werden. Dieses beinhaltet neben den Marker

Handles (vgl. Kapitel 4.6.1) zur dreidimensionalen Skalierung des virtuellen Objekts auch eine Art Übersichtstafel der Gebäudeeigenschaften des zugehörigen virtuellen Markers. Beispiele für Letzteres sind in Abbildung 12 zu sehen. Im Folgenden wird, wenn nicht anders erwähnt, die Übersichtstafel mit Kontextmenü bezeichnet. Die Idee des Kontextmenüs ist aus dem Treffen mit den Mitarbeitern eines Architekturbüros zu Beginn des Projektes entstanden (vgl. Kapitel ???). Diese hatten eine detaillierte Beschreibung der einzelnen Gebäude als sinnvoll und nützlich für die Umsetzung einer Anwendung wie den *MaRC* beschrieben.

Da zu Beginn der Anwendung im SetScale-Menü (vgl. Kapitel 4.5.2) der Maßstab der Architekturszene vom Benutzer ausgewählt wird und zusätzlich wie beschrieben eine Skalierung der einzelnen Objekte durchgeführt werden kann, muss sich der Inhalt des Kontextmenüs dynamisch ändern können. Die dazu notwendigen Berechnungen der Gebäudeeigenschaften sind in Kapitel 4.6.3 erläutert.

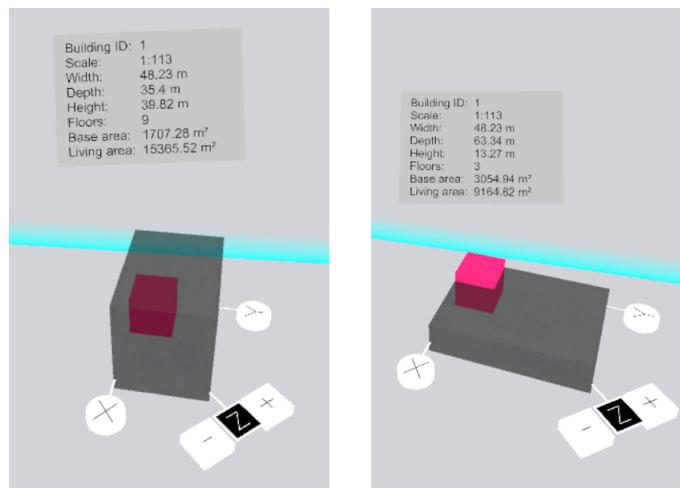


Abbildung 12: Zwei Beispiele für ein Kontextmenü eines Würfels mit ID 1.

4.6.1 Marker Handles

Paul

Jeder Marker verfügt über die Möglichkeit in seiner Größe in X-, Y- und Z-Achse verändert zu werden. Diese sog. "Handles" sind sichtbar, wenn das Kontextmenü angezeigt wird (s. Abschnitt ??). Die Handles bestehen aus Zylindern, die mit einem Collider versehen wurden. Die X- und Y- Achse wird betätigt, indem der Benutzer mit einem Finger das jeweiligen Handle berührt und dann in die entsprechende Richtung zieht. Die Größe in Z-Achse wird über "+" und "-" Button gesteuert. Drückt der Benutzer diese Buttons, wird das Gebäude um je ein Stockwerk erhöht oder verniedrigt.

Die Interaktion der Handles mit der Hand des Nutzers wurde über das Leap-Handmodell umgesetzt; dieses besitzt Collider, die an den Fingern des Modells angebracht sind und sich simultan mit der Hand des Nutzers bewegen. Berührt die Hand nun einen der Handler, wird dieser Collider des Handlers angesprochen". Im Falle der X-, und

Y-Handles wird in der `update` Funktion des Skripts `contextMenuTrigger.cs` ein Vektor berechnet, der die Start und Endposition der Bewegung miteinander verrechnet. Im Falle einer Bewegung in X Position wird die sog. "localDifference" berechnet, dies ist der Abstand des Fingers zur Position des Markers. Die eigentliche Skalierung wird dann wie folgt berechnet:

$$pos = (startPosition.x - localDifference.y)/2, startPosition.y, startPosition.z) \quad (1)$$

Das Gebäude wird dann um den Vektor pos im lokalen Objektkoordinatensystem vergrößert.

Für die Bewegung in Y berechnet sich der Vektor wie folgt:

$$pos = (startPosition.x, startPosition.y, (startPosition.z - localDifference.y)/2) \quad (2)$$

Es fällt auf, dass in der zu verändernden Achse verschiedene Achsen zur Berechnung verwendet werden müssen. Dies kommt daher, dass das Koordinatensystem der Leap Motion nicht equivalent zu dem in Unity ist.

Dass der Wert der zu verändernden Achse halbiert wird, hat sich empirisch als Sinnvoll herausgestellt. Ohne diese "Übersetzung" der Bewegung würde das Gebäude nicht equivalent zur Fingerbewegung vergrößert oder verkleinert werden.

4.6.2 Ein- und Ausblenden des Kontextmenüs

Lukas

Zum Ein- und Ausblenden des in Abschnitt 4.6 beschriebenen Kontextmenüs wird sich der Interaktionsmechanismus des *Leap Motion* Controllers bedient.

Zum Öffnen des Kontextmenüs muss der Benutzer mit einer vom *Leap Motion* Controller erkannten Hand die obere Fläche eines Markers berühren. Diese Interaktion funktioniert über einen in *Unity* erstellten Collider, welcher quaderförmig ist und in der Mitte eines jeden Markers nach oben herausschaut (vgl. Abb. 13). Sobald ein Teil der Hand des Benutzers in diesem Collider eindringt, wird in der Methode `OnTriggerEnter()` im Skript `contextMenuTrigger.cs` das Kontextmenü (dessen *GameObject* dort *CanvasTransform* heißt) sowie die in Abschnitt 4.6.1 beschriebenen Marker Handles aktiviert.

Auf die gleiche Weise wird das Kontextmenü mit den Marker Handles auch wieder geschlossen, wenn nämlich der Benutzer bei geöffnetem Kontextmenü die obere Fläche des Markers berührt. Dann werden die zum Kontextmenü gehörigen *GameObjects* wieder deaktiviert.

4.6.3 Berechnung der Gebäudeeigenschaften

Laura

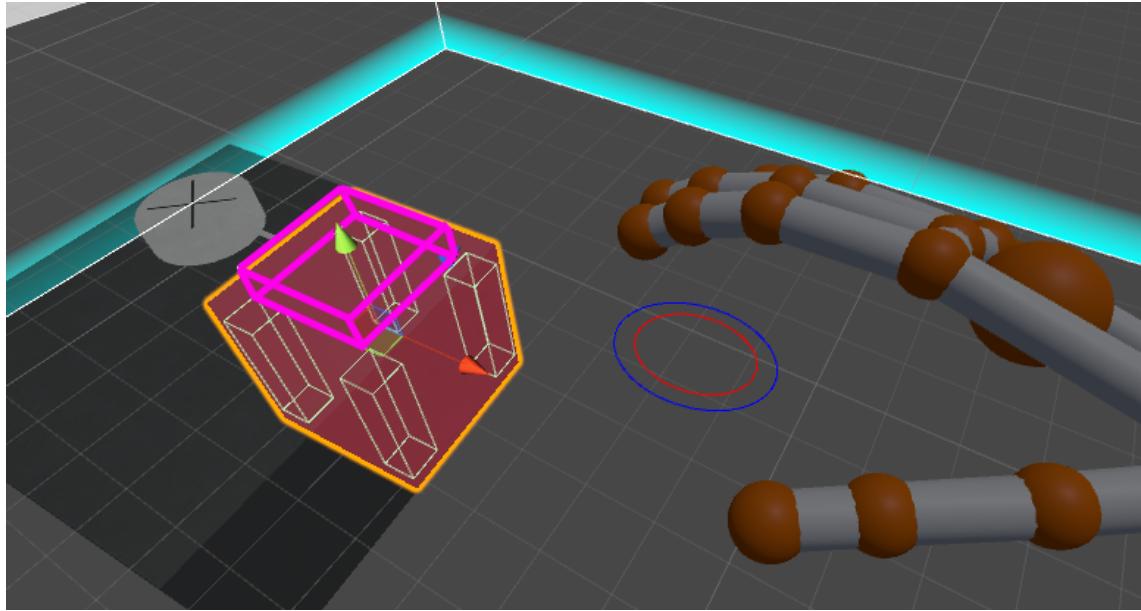


Abbildung 13: Collider zum Ein- und Ausblenden des Kontextmenüs (in Pink eingefärbt).

4.7 Table Menü

Paul

Zur intuitiven Bedienung des *MArC* wurden die Menüs, die während der Laufzeit benutzt werden können, auf dem Arbeitstisch platziert. Dies hat den Vorteil, dass zum einen diese Menüs permanent sichtbar sein können, ohne den Nutzer während der Arbeit zu stören und zum anderen ein haptisches Feedback bei der Nutzung möglich ist.

Möchte der Nutzer die Menüs bedienen, so drückt er mit dem Finger, der durch die *Leap Motion* getrackt wird auf die Buttons und damit gleichzeitig auf den Tisch. Dies erleichtert die Benutzung deutlich im Vergleich zu der Nutzung von Menüs die im Raum schweben.

Im Laufe der Entwicklung hat sich für das rechte Menü der Name *Table Menü* etabliert. Im folgenden wird dessen Funktion im Detail erläutert.

4.7.1 Szenen Management

Paul

Eine Anforderung an das System war, dass der Nutzer erstellte Szenen abspeichern und wieder aufrufen können soll. Anhand des *Table Menüs* ist dies möglich. Die Funktionen des Menüs sind:

- Speichern von Szenen
- Laden von Szenen und aufrufen des *Match Modus* (s. Abschnitt 4.7.4)
- Scrollen durch die gespeicherten Szenen

4.7.2 Speichern von Szenen

Paul

Möchte der Nutzer eine Szene speichern, so drückt er mit dem Finger auf den Button „Save“. *MArC* speichert die aktuelle Szene automatisch in dem Ordner „\Resources\saves\Dateiname.xml“.

Der Dateiname setzt sich aus aktuellem Datum und aktueller Zeit zum Speicherzeitpunkt wie folgt zusammen:

Tag-Monat-Jahr-Stunde-Minute-Sekunde.xml

Durch diese Markierung können später Dateien identifiziert werden, die auch nur Sekunden hintereinander gespeichert wurde.

Da innerhalb von *Unity* eine hierarchische Anordnung sämtlicher Elemente in Form eines Szenographen angewendet wird, bot sich eine Speicherung der Daten ebenfalls hierarchisch in form eines XML Dokumentes an.

Die *Extensible markup language (XML)* beschreibt eine Klasse von Daten Objekten (XML Documents) und ermöglicht das hierarchische Abspeichern von geparsten oder ungeparsten Daten [6]. In C# sind Verarbeitungsklassen bereits implementiert. Mittels dieser sog. *XML Prozessoren* wird ein Datenzugriff erleichtert.

Dateiaufbau einer gespeicherten Szene Das gesamte Dokument wird mit einem Hauptknoten <AR2_COMPOSERSCENE> umspannt. Innerhalb diesem wird mit texttt <time> ein Zeitstempel mit gespeichert, falls der Dateiname einmal umbenannt werden sollte. Anschließend folgt der texttt <TableObject> Knoten, innerhalb dessen die Marker gespeichert werden. Jeder Marker wird nach dem folgenden Schema gespeichert, welches den Aufbau in Unity repräsentiert:

```

1  <Name>
2    <Kindknoten>
3    <PositionX>
4    <PositionY>
5    <PositionZ>
6    <RotationX>
7    <RotationY>
8    <RotationZ>
9    <ScaleX>
10   <ScaleY>
11   <ScaleZ>
12 </Name>
```

Wobei sich dies als Kurzschreibweise versteht, innerhalb der Positions/Rotations-/Scale Knoten ist der entsprechende Wert eingetragen.

Vorgehen der Speicherung Das Skript `save.cs` ist für die Speicherung der Szenen verantwortlich. Zunächst wird der Zeitstempel gespeichert und der Dateiname

generiert. Anschließend wird ein neues XML Dokument erstellt. An dieses wird zunächst der äußerste Knoten angehangen sowie der Zeitstempel. Anschließend wird die rekursive Funktion `traverseHierarchie` mit dem TableObject der Unity Szene aufgerufen und wird für jeden Kindknoten ausgeführt.

Diese Funktion erstellt jeweils die Knoten für den Namen, die Position, Rotation und Skalierung. Die Werte werden mittels `Node.InnerText` in die Knoten gespeichert.

Ist diese Funktion durch alle Kindknoten gegangen, wird am Ende noch die Skalierung der Szene in dem Knoten `globalBuildingScale` abgespeichert, damit dieser beim Laden der Szene rekonstruiert werden kann.

Darstellen von gespeicherten Szenen Die gespeicherten Szenen werden in Listenform im Table Menü dargestellt. Zu oberst sind immer die aktuellen Szenen. Über die Buttons "1-6", "7-12", "13-18" und "19-24" kann umgeblättert werden und weitere, in der Vergangenheit liegende Szenen dargestellt werden. Für diese Darstellung ist das Skript `Timeline.cs` zuständig. Bei jedem Speichern wird die Darstellung aktualisiert, so dass immer alle neuesten gespeicherten Szenen sichtbar sind.

4.7.3 Laden von Szenen

Paul

Das Laden der Szenen ist ein komplexer Vorgang. Dies kommt daher, dass auf der einen Seite Markerdaten in der zu öffnenden .xml Datei gespeichert sind und geladen werden müssen. Auf der anderen Seite gibt es evtl. noch Marker, die auf dem Arbeitsfeld liegen und aktuell getrackt werden. Beide Informationsströme müssen beim laden der Szene berücksichtigt werden und korrekt verarbeitet werden. Ein einfaches übertragen von Positions-, Rotations- und Skalierungsdaten auf vorhandene Markerobjekte ist nicht möglich, da diese permanent durch den TCP Datenstrom überschrieben werden würden.

Vorgehen beim Laden Durch Auswählen der Datei im Table Menü mit dem Finger wird das Skript `open.cs` gestartet. Dieses setzt zuerst den Pfad aus dem aktuellem Applikationspfad mit dem Unterordner Resources und saves zusammen und fügt den Namen der zu öffnenden .xml Datei aus dem Table Menü hinzu. Anschließend wird die Datei eingelesen, die internen Knoten liegen dann in einer XmlNodeList bereit. Es wird die Funktion `crawlXML` aufgerufen.

Da aus organisatorischen Gründen alle Marker in der Datei gespeichert werden müssen, wird die XmlNodeList bearbeitet und geprüft, welche der Marker als aktiv" gespeichert wurden. Aktive Marker sind solche, die zum Zeitpunkt des Speicherns sichtbar waren. Alle aktiven Marker werden in der ArrayList `activeMarkerIDs` gespeichert.

Diese ArrayList wird nun, nachdem sie vollständig gefüllt ist, bearbeitet: Für jeden Eintrag, also jeden aktiven Marker wird ein Vorlagemarker instantiiert. Dies ist ein Marker, der bereits alle Komponenten die benötigt werden angeheftet hat und dem

nur noch die Parameter zugewiesen werden müssen. Dies wird dann getan; jeder Marker bekommt die OriginalID + 100 als Namen zugewiesen. Dies ist wichtig um später die aus der .xml Datei gelesenen Marker von den aktuell über TCP gesteuerten Marker unterscheiden zu können. Anschließend wird werden die weiteren Parameter wie Position, Rotation und Skalierung an den Marker übergeben. Schließlich wird der Marker an das "TableObject in Unity gehangen und sichtbar gemacht. Als letzter Schritt wird die Markervariable "MatchMode auf "true" gesetzt und das Skript `matchMode.cs` aktiviert. Dieses wird detailliert im Abschnitt 4.7.4 beschrieben. Die Marker die sich in diesem Match Modus befinden blinken grün und rot, um zu signalisieren, dass sie auf einen echten Marker warten. Sind alle Marker auf diese Weise geladen und parametrisiert, wird der globale Maßstab an das Skript `setupScene.cs` übergeben.

Zu diesem Zeitpunkt sind also zwei Arten von Markern sichtbar: Die geladenen aus der .xml Datei, die grün und rot blinken und eine ID > 100 haben, und die aktiven Marker die getrackt werden.

4.7.4 Match Modus

Paul

Der Match Modus ist ein Systemzustand, der nach dem Laden von gespeicherten Szenen aktiviert wird. Es werden wie im Abschnitt 4.7.3 beschrieben die geladenen Marker und die getrackten Marker gleichzeitig angezeigt. Der Benutzer muss jetzt die echten Marker an die Position der geladenen virtuellen Marker schieben. Ist ein Marker im Match Modus, sind vier Collider an den Ecken des virtuellen Markers wichtig. Die Markerinterne Variable "matchMode ist = "false" gesetzt. Schiebt der Benutzer nun einen Marker an die Position des virtuellen Markers, werden bei genauer Positionierung alle vier Collider des geladenen virtuellen Markers mit den vier Collidern des anderen virtuellen Markers, der von dem Nutzer bewegt wurde, aktiviert. Ist dies der Fall, und nur dann, wird die Markerinterne Variable "matchMode-Ready auf "true" gesetzt. Dieses Verhalten wird von dem Skript `StopMatchMode.cs` kontrolliert. Sind alle Marker bereit, werden die Parameter aus den gespeicherten Markern auf diejenigen Marker übertragen, die der Benutzer an die Position der virtuellen Marker geschoben hat. Um diese Funktionalität kümmert sich das Skript `DataHandler.cs`. Die geladenen Marker mit der ID > 100 werden im Anschluss gelöscht. Am Ende des Match Modus existieren wieder nur die TCP gesteuerten Marker, die jetzt allerdings die Parameter aus der geladenen Datei übernommen haben und somit in der Gesamtheit die geladene Szene repräsentieren.

5 Tracking Programm

Vera

Zur Verfolgung und Positionsbestimmung der Würfel Marker (siehe Kapitel 3.1.8) wurde ein Programm entwickelt, welche alle erforderlichen Ressourcen und Funktio-

Parameter	Wert
Pixel Clock	37 ms
Frame Rate	23 fps
Belichtungszeit	40 ms
Gamma	2.2
Digitale Verstärkung	20

Tabelle 6: Parametrisierung der uEye Kamera für das Tracking.

nen bereit stellt und verknüpft. Zunächst muss die Initialisierung und der Zugriff der uEye Kamera (siehe Kapitel 3.1.4) ermöglicht werden. Im Anschluss ist das Erstellen beziehungsweise laden aller relevanten Kalibrierungsinformationen unabdingbar. Nach erfolgreichem Abschluss beider Schritte wird das Tracking der Aruco Marker und der grünen Rechtecke ausgeführt und anschließend die Verwaltung aller registrierten Würfel Marker übernommen. Zuletzt werden alle relevanten Daten der Marker vom Tracking Programm per TCP Übertragung (siehe Kapitel 2.4) an den *Unity* Computer übertragen. Alle Schritte die der Kalibrierung folgen werden in einer Endlosschleife für jeden Frame ausgeführt. Die durchschnittliche Dauer einer Iteration mit der Aufnahme, dem Tracking sowie der Übertragung dauert *XXXXXXms* und einer Framerate von *XXXX* fps. Dieses Programm wurde in der IDE Visual Studio (siehe Kapitel 3.3.2) in der Programmiersprache *C++* entwickelt. Für die Entwicklungen wurden die uEye SDK, *OpenCV* (siehe Kapitel 3.3.3) sowie die Standardbibliothek *Winsock* zur Hilfe genommen. In den folgenden Abschnitten wird die Vorgehensweise im Detail erklärt.

Vera:
mes-
sen
der
Dau-
er
und
Fra-
me-
rate

5.1 uEye Ansteuerung

Vera

Der erste Prozess des Tracking Programms ist die Initialisierung der uEye Kamera im Live-Bild-Modus. Mit Hilfe der vom Hersteller bereit gestellten SDK kann die Kamera mit allen benötigten Eigenschaften parametrisiert und alle notwendigen Speicher allokiert. Dies übernimmt die Funktion `inituEyeCam` in der Klasse `uEye_input.cpp`. Die verwendeten Parameter können aus der Tabelle 6 entnommen werden. Die Kamera wird mit dem maximalen Pixel Clock und der maximalen Framerate betrieben. Aus diesen Einstellungen ergibt sich auch die Belichtungszeit. Um einen höheren Kontrast zur Erkennung der Marker zu erzielen wurde zusätzlich das Hardware Signal zwanzigfach verstärkt. Im Live-Bild-Modus werden die generierten Aufnahmen fortlaufend in der selben Speicheradresse überschrieben. Auf diesen Speicherplatz kann das Programm jederzeit mit der Funktion `getCapturedFrame` zugreifen, die die aktuellste Aufnahme zur Weiterverarbeitung zurück gibt. Nach der Beendigung des Tracking Algorithmus wird zusätzlich noch die Funktion `exitCamera` bereit gestellt, welche den verwendeten allokierten Speicher wieder freigibt.

5.2 Kalibrierung

Laura

Um das Ziel von *MArC* zu erreichen, an den Positionen der Aluminiumwürfel im Arbeitsbereich in der virtuellen Realität von Unity gerenderte Würfel darzustellen, muss das System kalibriert werden. Die Kalibrierung hat zum Ziel, eine Koordinatentransformation zu finden, die Positionen im Kamera-Koordinatensystem in das Unity-Koordinatensystem transformiert.

Zu diesem Zweck muss eine zweistufige Kalibrierung durchgeführt werden. Zunächst sorgt die Kamerakalibrierung dafür, dass Bildkoordinaten auf dem Sensor der Kamera in das 3D-Kamera-Koordinatensystem transformiert werden. Dafür wird sich einiger OpenCV-Funktionen in Verbindung mit Aruco-Markern bedient. Dieser Vorgang wird nachfolgend in [5.2.3](#) genauer beschrieben.

Der nächste Schritt, die Kalibrierung des Arbeitsbereichs, bestimmt über Punkt-Korrespondenzen (vgl. Kapitel [5.2.1](#)) – also in zwei verschiedenen Koordinatensystemen bekannte Punkte – eine affine 3D-Transformation, welche die Abbildung vom Kamera-Koordinatensystem auf das Unity-Koordinatensystem ermöglicht. Dieser Kalibrierungsschritt wird nachfolgend in [5.2.4](#) näher beschrieben.

Nach vollständiger Kalibrierung des Systems wird sowohl die Kamerakalibrierung, als auch die Kalibrierung des Arbeitsbereiches, abgespeichert (vgl. Kapitel ??), sodass der Benutzer, beim erneuten Starten des Systems, entscheiden kann, ob er auf eine erneute Kalibrierung verzichtet oder aber das System teilweise oder komplett neu kalibrieren möchte (vgl. `SelectCalibrationTarget` in Kapitel [4.5.2](#)). Dabei ist zu beachten, dass eine Kamerakalibrierung nur in Verbindung mit einer anschließenden Kalibrierung des Arbeitsbereiches durchgeführt werden kann. Eine Arbeitsbereichskalibrierung kann jedoch autonom vorgenommen werden.

Die Beschreibung des Algorithmus in den nächsten beiden Kapiteln umfasst sowohl eine mathematische Darstellung, als auch eine Darstellung der konkreten Umsetzung in Quellcode. Letzteres ist bewusst kurz gehalten, da für die Umsetzung der einzelnen Teilschritte vorwiegend Methoden aus der *OpenCV*-Library (vgl. Kapitel [3.3.3](#)) benutzt worden sind. Alle Grundlegenden Definitionen für die Rechenschritte können in Kapitel [5.2.2](#) nachgelesen werden. Die Schnittstelle in Form der `Calibration.cpp`-Klasse wird in Kapitel [5.2.7](#) eingeführt.

Trotz einer sorgfältigen Umsetzung der in den nächsten beiden Kapiteln beschriebenen Kalibrierungsschritte, ist es nicht gelungen, den Aluminiumwürfel und den gerenderten Würfel vollständig zur Deckung zu bringen. Der entstandene Fehler sowie mögliche systembedingte Fehlerquellen werden in [5.2.5](#) und in [5.2.6](#) beschrieben.

5.2.1 Korrespondierende Punktpaare

Laura

Im Zusammenhang mit der Kalibrierung des Systems von *MaRC* werden von einem Marker Koordinaten in Bildkoordinaten und die entsprechenden Koordinaten in der Unitywelt benötigt. Hierbei spricht man von korrespondierenden Punkten. Genau-

er gesagt wird für diese Zwecke der Kalibrierungscontroller (vgl. Kapitel 3.1.10) verwendet. Auf diesem befindet sich ein *ArUco MArker* (vgl. Kapitel 3.1.7). Der Mittelpunkt, also das `estimatedCenter`, kann für jede Position über das in Kapitel 5.3.2 beschriebene Trackingverfahren ermittelt werden und wird für jeden erkannten Marker auf Trackingseite abgespeichert. Dies geschieht bei der Kalibrierung genau immer dann, wenn der Trigger am Controller der *HTC Vive* gedrückt wird. So mit kann in Unity ebenfalls die entsprechende Position abgespeichert werden. Eine Position liegt also sowohl in Bild- als auch in Unitykoordinaten vor (vgl. Tabelle 7).

5.2.2 Definition der Koordinatensysteme

Laura

In Abbildung 14 sind die Zusammenhänge zwischen dem Projektionszentrumkoordinatensystem der Kamera, sowie deren Bildebene und dem Weltkoordinatensystem zu sehen. Im Gegensatz zu der Abbildung und den in Kapitel 5.2.3 erwähnten Kalibrierungsansätzen reicht die Umrechnung von Bildkoordinaten in Weltkoordinaten, im vorliegenden Fall, nicht aus. Es muss eine Umrechnung der Bildkoordinaten in den Unity Raum erfolgen, um zu gewährleisten, dass die gerenderten Würfel anschließend deckungsgleich mit den Aluminiumwürfeln sind. Koordinaten des Unity Raumes werden im Folgenden Unitykoordinaten genannt.

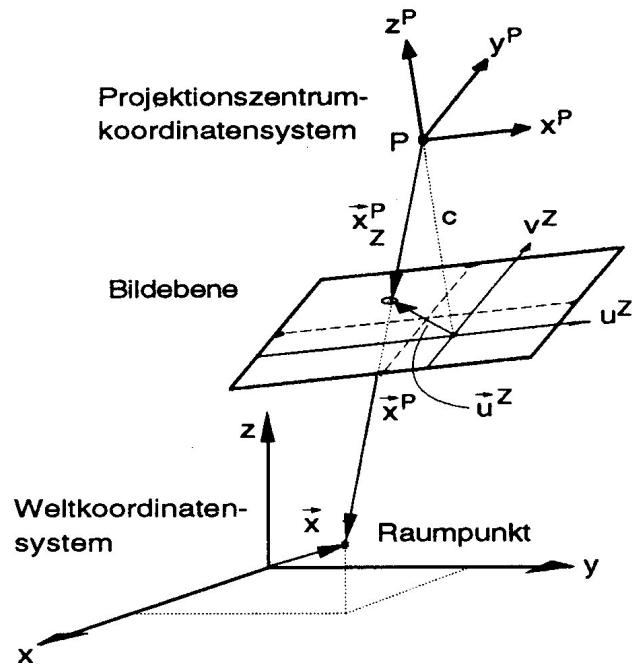


Abbildung 14: Lage des Kamerakoordinatensystems in Bezug auf Projektionsebene und Weltkoordinatensystem. [34]

Um die verschiedenen Koordinatensysteme bei den Ausführungen in den Kapiteln 5.2.3 und 5.2.4 auseinander halten zu können, wird zu Beginn eine Notation fest-

	Koordinaten	Komponenten	Transformation
\vec{x}^U	Unity	x^U, y^U, z^U	
\downarrow			Koordinatentransformation
\vec{x}	Welt	x, y, z	
\downarrow			Koordinatentransformation
\vec{x}^P	Projektionszentrum	x^P, y^P, z^P	
\downarrow			Projektion
\vec{x}_Z^P	Projektionszentrum	$u^Z, v^Z, -c$	
\downarrow			Linsenverzeichnung
\vec{x}_D^P	Verzeichnung	$u^D, v^D, -c$	
\downarrow			Bildebenenverkippung
\vec{x}^V	Verkippung	$u^V, v^V, -c^V$	
\downarrow			Bildhauptpunktverschiebung
\vec{u}	Sensor	u, v	

Tabelle 7: Parameter und Berechnungsschritte der Kamera Kalibrierung.[34]

gelegt, die in Tabelle 7 eingesehen werden kann. Zusätzlich können in der Tabelle sowohl die einzelnen Koordinatensysteme, als auch die einzelnen Berechnungsschritte der Kamerakalibrierung, sowie der Koordinatentransformation bis hin zu Unitykoordinaten, nachvollzogen werden. Lässt man die ersten zwei Zeilen der Tabelle weg, so ist eine Transformation von Weltkoordinaten \vec{x} in Kamerakoordinaten \vec{u} schematisch dargestellt. Dieses Schema muss für die Kamerakalibrierung von *MaRC* invertiert werden und um die Koordinatentransformation in Unitykoordinaten ergänzt werden.

5.2.3 Kamerakalibrierung

Laura

Es gibt viele verschiedene wissenschaftliche Ausführungen über die Durchführung einer Kamerakalibrierung, wie z.B. [47], [61] und [15]. Dabei unterscheidet man häufig zwischen automatischen und manuellen Kalibrierungen. Im Folgenden wird der Algorithmus zur Kamerakalibrierung des *MaRC* erläutert.

Im konkreten Fall soll die uEye-Kamera (vgl. Kapitel 3.1.4), die für Tracking-Zwecke(vgl. Kapitel 5) an der Decke des Entwicklungsraums befestigt ist, kalibriert werden. Dies kann, wenn gewünscht bzw. benötigt, zu Beginn der eigentlichen Anwendung durchgeführt werden. Dazu wird ein eigens für dieses Projekt angefertigter Schachbrett-Kalibrierungshelfer verwendet, der in Kapitel 3.1.9 beschrieben

wird. Eine Beschreibung der genauen Durchführung der Kamerakalibrierung kann der ReadMe-Datei (vgl. 10.1) entnommen werden. Im Folgenden wird zunächst auf die mathematische Durchführung und dann auf die konkrete Umsetzung in Quellcode eingegangen.

Mathematische Umsetzung: In diesem Abschnitt werden die mathematischen Grundlagen der Kamerakalibrierung erläutert. Dabei wird sowohl auf die intrinsischen, als auch auf die extrinsischen Kameraparameter eingegangen. Es wird ebenso gezeigt, die aus den Parametern eine geeignete Transformation von Kamerakoordinaten in Weltkoordinaten genildet werden kann.

Wie aus Tabelle 7 hervorgeht, wird im ersten Schritt der Hauptpunkt so versetzt, dass der Koordinatenursprung des Kamerakoordinatensystems mit dem Bildkoordinatensystem übereinstimmt. Dies geschieht, indem man für die eine Achse $u^V = u - \Delta u$ und für die andere Achse entsprechend $v^V = v - \Delta v$ berechnet.

Zusätzlich wird die Bildebene verkippt, so dass gilt $\vec{x}^V = R_v \cdot \vec{x}_D^P$. Bezeichnet φ den Drehwinkel um die x^P -Achse und ϑ den Drehwinkel um die y^P -Achse, so lässt sich R_v wie folgt berechnen:

$$R_v = \begin{pmatrix} r_{v11} & r_{v12} & r_{v13} \\ r_{v21} & r_{v22} & r_{v23} \\ r_{v31} & r_{v32} & r_{v33} \end{pmatrix} = \begin{pmatrix} \cos \vartheta & \sin \vartheta \sin \varphi & -\sin \vartheta \cos \varphi \\ 0 & \cos \varphi & \sin \varphi \\ \sin \vartheta & -\cos \vartheta \sin \varphi & \cos \vartheta \cos \varphi \end{pmatrix} \quad R_v \in \mathbb{R}^{3 \times 3} \quad (3)$$

Die negative verkippte Kamerakonstante $-c^V$, die in Tabelle 7 aufgeführt ist, berechnet sich wie in [34] beschrieben nach der Formel:

$$-c^V = -\frac{c + u^V \cdot r_{v13} + v^V \cdot r_{v23}}{r_{v33}} \quad (4)$$

Im zweiten Schritt wird im Allgemeinem die Linsenverzeichnung herausgerechnet. Im vorliegenden Fall wurde bewusst auf diesen Schritt verzichtet und die zugehörigen Entzerrungskoeffizienten werden für alle weiteren Berechnungen auf null gesetzt. Dieses Vorgehen wurde gewählt, da einige Tests gezeigt haben, dass die berechneten Entzerrungskoeffizienten stark voneinander abgewichen sind, obwohl dies beim Benutzen der gleichen Kameraeinstellungen und dem gleichen Objektiv nicht der Fall sein dürfte.

Weil wie bereits beschrieben, die Linsenverzeichnung vernachlässigt wurde, kann man Schritt 1 und 2 zu einem Schritt zusammenfassen und vereinfacht darstellen. Dieser erste Schritt, also die Hauptpunktverschiebung und Bildebenenverkippung, lässt sich mit Hilfe der intrinsischen Kameramatrix $M_{intrinsisch}$ zusammenfassen. Diese beinhaltet neben den Brennweiten f_u und f_v noch die Koordinaten des Hauptpunktes u^V und v^V in Bildkoordinaten und hat somit vier Freiheitsgrade.

$$M_{intrinsisch} = \begin{pmatrix} f_u & 0 & 0 & u^V \\ 0 & f_v & 0 & v^V \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad M_{intrinsisch} \in \mathbb{R}^{3x4} \quad (5)$$

Während eine dreidimensionale Rotation im Allgemeinen mit Matrix R aus Formel 6 beschrieben werden kann, reicht für die Translation ein Vektor, wie t aus Formel 7 aus.

$$R = R_\gamma \ R_\beta \ R_\alpha = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \quad R \in \mathbb{R}^{3x3} \quad (6)$$

$$t = (t_x \ t_y \ t_z)^T \quad t \in \mathbb{R}^{3x1} \quad (7)$$

Diese beiden Transformationen können wie in Formel 8 zur extrinsischen Kameramatrix $M_{extrinsisch}$ zusammengefasst werden. Diese hat sechs Freiheitsgrade, nämlich drei für den Translationsvektor t und drei für die Eulerwinkel der Rotationsmatrix R für die dementsprechend gelten muss $R \in SO(3)$.

$$M_{extrinsisch} = (R \ | \ t) \quad M_{extrinsisch} \in \mathbb{R}^{3x4} \quad (8)$$

Schritt 1 und 2 lassen sich vereinfachen, indem man die intrinsische Kameramatrix $M_{intrinsisch}$ und die extrinsische Kameramatrix $M_{extrinsisch}$ nach der Formel 9 zu einer Matrix M zusammenfasst.

$$M = M_{intrinsisch} \cdot M_{extrinsisch} \quad M \in \mathbb{R}^{3x4} \quad (9)$$

Die Umrechnung von Kamerakoordinaten in Weltkoordinaten kann dann mit der invertierten Matrix M , wie in Formel 10 berechnet werden.

$$\vec{x} = M^{-1} \cdot \vec{u} \quad (10)$$

An diesem Punkt hat man die Kamerakoordinaten vollständig in Weltkoordinaten überführt und sucht nun eine Transformationsmatrix um diese Punkte auf ihre korrespondierenden Punkte (vgl. Kapitel 5.2.1) in Unitykoordinaten abzubilden. Die dazu benötigte affine 3D-Transformation wird in Kapitel 5.2.4 erläutert.

Zu diesem Zeitpunkt sollte nun das Kamerakoordinatensystem mit dem Weltkoordinatensystem übereinstimmen und es kann sich im nächsten Schritt darum gekümmert werden, dass das Kamerakoordinatensystem verschoben und gedreht wird. Dieser Schritt und alle weiteren Schritte, die im Zusammenhang mit der Transformation von Kamerakoordinaten in Weltkoordinaten bzw. Weltkoordinaten in Unitykoordinaten stehen, werden in Kapitel 5.2.4 erläutert. Hierbei sei noch einmal angemerkt, dass eine Kamerakalibrierung nur mit anschließender Kalibrierung des Arbeitsbereiches durchgeführt werden kann.

5.2.4 Kalibrierung des Arbeitsbereichs

Laura

Wie bereits beschrieben, kann eine Kalibrierung des Arbeitsbereichs entweder mit vorheriger Kamerakalibrierung oder einzeln durchgeführt werden. In letzterem Fall wird dann die zuletzt verwendete Kamerakalibrierung geladen. Die Durchführung der Kalibrierung wird in der ReadMe-Datei (vgl. 10.1) erläutert. Grob gesagt, werden über Knopfdruck am Kalibrierungscontroller (vgl. 9) 25 seiner Positionen sowohl in Kamerakoordinaten, als auch in Unitykoordinaten gespeichert. Man erhält also 25 korrespondierende Punktpaare (vgl. Kapitel 5.2.1).

Allgemein betrachtet hat die Arbeitsbereichskalibrierung zwei verschiedene Aufgaben: Zum einen wird hierbei ein Bereich festgelegt, indem die Objekte getrackt werden und zum anderen wird mittels der verschiedenen Positionen der Kalibrierungspunkte, die affine Transformationsmatrix bestimmt, die im weiteren Verlauf als Umrechnung zwischen Kamerakoordinaten und Unitykoordinaten fungiert. Beide Teilbereiche werden in den Absätzen “Begrenzung des Arbeitsbereichs” bzw. “Affine Transformation“ weitergehend erläutert.

Begrenzung des Arbeitsbereichs Die ersten vier Punktpaare dienen dazu den Arbeitsbereich zu definieren. Sie müssen in einer klar definierten Reihenfolge, nämlich im Uhrzeigersinn und in der linken unteren Ecke beginnend, durchgeführt werden. Dabei wird bei beiden Systemen (Tracking und *Unity*) unterschiedlich vorgegangen, was zu leicht abweichenden Arbeitsbereichen führen kann. Der Unterschied ist allerdings vernachlässigbar, wenn der Benutzer sich an den Tischkanten orientiert und beispielsweise die Kante der Auflage des Kalibrierungscontrollers (vgl. Kapitel 3.1.10) daran ausrichtet und somit eine ungefähr rechteckförmige Fläche aufspannt.

Für die Tracking-Anwendung reicht es aus, den 1. und 3. Punkt zu nehmen und damit ein Rechteck aufzuspannen. Dies geschieht in der `computePlaneCalibration()`-Methode in der `PlaneAndAffineCalibration.cpp`-Klasse. Der so definierte Arbeitsbereich dient hier vor allem dazu, zu entscheiden, ob ein Marker noch im Anwendungsbereich ist und somit fortlaufend getrackt werden muss, oder ob er diesen Bereich verlassen hat und seine ID, sowie alle anderen Eigenschaften zurückgesetzt werden müssen (vgl. Kapitel ??). In Abbildung 15 ist der so aufgespannte Arbeitsbereich in rot dargestellt.

Hier habe ich (Lukas) jetzt nicht erklärt, wo das Problem (dass ein rechteckiges Spielfeld auf Trackingseite nicht einem Rechteck auf Unity-Seite entspricht) eigentlich herkommt. Weil wir das ja immer noch nicht wissen... oder?

Auf *Unity*-Seite werden die gesamten ersten vier Punkte genutzt um den Arbeitsbereich festzulegen. Dies hat den Grund, dass der Arbeitsbereich bei der Simulation an allen vier Ecken durch kleine „Winkel“, wie in Abbildung ?? dargestellt, begrenzt wird. Zunächst war der Arbeitsbereich in der *Unity*-Simulation ebenfalls nur durch zwei Punkte definiert, sodass der Bereich im-

mer ein Rechteck auf dem Tisch darstellte. Es stellte sich jedoch heraus, dass ein im Kamerabild auf Tracking-Seite definiertes Rechteck übertragen auf die *Unity*-Simulation nicht hinreichend genau als Rechteck auf dem Tisch abgebildet wurde. Dies führte dazu, dass Marker, die in der Simulationsanwendung noch innerhalb des Spielfelds zu sein schienen, auf Tracking-Seite bereits registriert worden waren, oder umgekehrt. Um dieses Abbildungsproblem zu umgehen, wurden anschließend vier Punkte für die Definition des Arbeitsbereichs in der *Unity*-Simulation verwendet. Der Nachteil hierbei ist, dass der Arbeitsbereich bei genauem Hinsehen nicht genau einem Rechteck entspricht, sondern eher einem Parallelogramm.

Affine Transformation Die restlichen 21 Controller-Positionen werden genutzt, um eine affine Transformation von Weltkoordinaten in Unitykoordinaten zu finden. An dieser Stelle kann davon ausgegangen werden, dass die Mittelpunkte der *ArUco* Marker über die in Kapitel 5.2.3 beschriebene Transformation bereits in Weltkoordinaten vorliegen. Dabei werden die Position nicht wie bei der Begrenzung des Arbeitsbereichs für beide Systeme (Tracking und Unity) getrennt verarbeitet, sondern sie werden als korrespondierende Punkte (vgl. Kapitel 5.2.1) verstanden. Mit anderen Worten, soll in diesem Schritt des Algorithmus jeder dieser 21 Punkte in Weltkoordinaten, möglichst fehlerfrei auf sein entsprechendes Pendant in Unitykoordinaten projiziert werden.

Es gilt also die in Gleichung 11 der Vollständigkeit halber aufgeführte Matrix zu bestimmen. Da es geeignete Werte für a, b, c und d , sowie die Translationsparameter t_x und t_y zu finden gilt, hat die gesuchte Transformation 6 Freiheitsgrade. Zur Abschätzung der bestmöglichen Matrix M_{affin} eignet sich vor allem der RANSAC-Algorithmus [17]. Dieser wird ebenso für die in diesem Projekt verwendete *OpenCV*-Methode `estimateAffine3D()` verwendet. Dieser Methode können alle 21 korrespondierenden Punktpaare übergeben werden.

$$M_{affin} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \quad M_{affin} \in \mathbb{R}^{3 \times 3} \quad (11)$$

5.2.5 Kalibrierungsfehler

Laura

In den Kapiteln 5.2.3 und 5.2.4 wird der Algorithmus zur Berechnung der Koordinatentransformation von Kamerakoordinaten in Unitykoordinaten beschrieben. Beim Benutzen des fertig kalibrierten System fällt jedoch auf, dass die Aluminiumwürfel nicht an allen Positionen auf dem Spielfeld deckungsgleich mit ihrem gerenderten Pendant sind. Bei genauerer Betrachtung ...Wie sieht Fehler aus? immer noch an kalibrierten Ecken und Diagonale ok?

Im nächsten Kapitel werden mögliche Gründe für diese fehlerhafte Abbildung aufgeführt und diskutiert.

5.2.6 Mögliche Fehlerquellen

Laura

Bei der Registrierung der Würfelobjekte kommt es zu dem in [5.2.5](#) beschriebenen Fehler. Neben der Rechenungsgenauigkeit und Rundungsfehlern, die sich über die diversen Rechenschritte (siehe Kapitel [5.2.3](#) und [5.2.4](#)) hinweg aufaddieren, sollen in diesem Kapitel systembedingte mögliche Fehlerquellen aufgeführt werden.

- 1) Zum Einen Tracking Vive 2) Lösen DLT 3) Fehlerfortpflanzung 4) Inside out 5) Trackingfehler 6) Kameraauflösung begrenzt

Zusätzliche Registrierungsfehler können durch Latenz entstehen. Nämlich immer dann, wenn eine Zeitdifferenz zwischen der Messung der Position in Kamerakoordinaten und der eigentlichen Darstellung nach entsprechender Verarbeitung entsteht. Solche Systemverzögerungen werden allerdings nur bei Bewegungen sichtbar und sind deshalb für die beobachteten Fehler bei ruhendem Aluminiumwürfel nicht als mögliche Fehlerquelle auszumachen.

5.2.7 Schnittstelle der beiden Kalibrierung

Laura

Wie eingangs erwähnt, dient die `Calibration.cpp`-Klasse als Schnittstelle für die Kamerakalibrierung (vgl. Kapitel [5.2.3](#)) und die Kalibrierung des Arbeitsbereichs (vgl. Kapitel [5.2.4](#)). Diese Schnittstelle stellt Methoden für beide Kalibrierungen, in denen alle erforderlichen Funktionen ausgeführt werden, und die benötigten Zugriffe auf die generierten Daten zur Verfügung.

Die `runPoseEstimation()`-Methode ist zuständig für die Generierung der Kameramatrix `camMatrix` ($M_{intrinsisch}$ aus Gleichung [5](#)) und die Koeffizienten zur Behebung der Linsenverzeichnung `distCoeffs`. Beide werden mit dem Modul `calib3D` der *OpenCV* Bibliothek in der Klasse `calibWithChessboard.cpp` berechnet und anschließend in der Methode `generateCamMatAndDistMat` in Klasse `PoseEstimation.cpp` für spätere Verwendung gespeichert und allen anderen Klassen der Tracking Software zur Verfügung gestellt. Vor dem Speichern und der Weiterverbreitung werden die `disCoeffs` auf null gesetzt. Die eigentlichen Bestimmung der Matrizen findet in der *OpenCV*-Methode `calibrateCamera` auf Grundlage von mehreren Aufnahmen eines fest definierten Kalibrierungsmusters statt. In diesem Fall wird das Schachbrett aus Abbildung [6](#) verwendet.

Wie in Kapitel [5.2.4](#) erklärt werden mittels der Arbeitsbereichskalibrierung zwei doch recht unterschiedliche Ziele verfolgt. Zum einen wird der tatsächliche Arbeitsbereich festgelegt und zum anderen wird die affine Transformation von Weltkoordinaten in Unitykoordinaten berechnet. Beide Aufgaben werden in der Schnittstelle mit der Funktion `generateAffineAndPlaneCalib` aufgerufen, welche die entsprechenden Implementierungen in der Klasse `PlaneAndAffineCalibration.cpp` ansteuert und die Ergebnisse in Textfiles abspeichert. Die Methoden `loadAffineTransform` und

`loadImagePlane` laden bei keiner neuen Kalibrierung, die zuletzt abgespeicherten Daten (vgl. Kapitel 5.2.8).

Die affine Transformation wird auf Grundlage der korrespondierenden Punktpaare (vgl. Kapitel 5.2.1) berechnet. Hierzu wird in der Methode `computeAffineTransformation` die Funktion `estimateAffine3D` der *OpenCV* Bibliothek aufrufen, welche die affine Transformationsmatrix mit Hilfe des RANSAC Algorithmus abschätzt [17].

5.2.8 Speichern und Laden der Kalibrierungen

Laura

Sowohl die Kamerakalibrierung als auch die Arbeitsbereichskalibrierung werden automatisch nach der Erstellung abgespeichert. Das gibt dem Benutzer beim nächsten Ausführen des System die Möglichkeit, folgenden 3 Möglichkeiten:

- 1. Beide Kalibrierungen durchführen :** In diesem Fall wird zunächst eine Kamerakalibrierung durchgeführt, wie sie in Kapitel 5.2.3 beschrieben ist und anschließend wird eine Kalibrierung des Arbeitsbereiches (vgl. Kapitel 5.2.4) vorgenommen. Alle daraus resultierenden Kamerakalibrierungs-Parameter werden mittels der `saveCameraParams()`-Methode (vgl. `PoseEstimation.cpp`-Klasse) gespeichert. Die notwendigen Eckpunkte des Arbeitsbereichs werden direkt nach ihrer Erstellung über die `computePlaneCalibration()`-Methode abgespeichert und die affine Transformation wird über die `saveAffineTransform()`-Methode (vgl. `PlaneAndAffineCalibration.cpp`-Klasse) in ein Textfile gesichert. Somit stehen alle benötigten Parameter - wenn gewünscht - für den nächsten Systemstart bereit.
- 2. Nur die Arbeitsbereichskalibrierung durchführen :** Entscheidet sich der Benutzer dafür, dass eine Kamerakalibrierung nicht notwendig ist, so können die entsprechenden Parameter über die `loadCameraMat()`-Methode aus der `PoseEstimation.cpp`-Klasse geladen werden. Der Arbeitsbereich kann dann wie in Kapitel 5.2.4 beschrieben kalibriert werden und die neue Kalibrierung wird wie unter 1. beschrieben abgespeichert. Die alte Kamerakalibrierung wird allerdings beibehalten und muss nicht erneut gesichert werden.
- 3. Keine Kalibrierungen durchführen :** Wenn der Nutzer unter den gleichen Bedingungen, wie bei der letzten Systemverwendung weiterarbeiten möchte, ist es sinnvoll die komplette Kalibrierung zu laden. Für die Kamerakalibrierung geht man dabei wie unter 2. beschrieben vor. Für das Laden der Arbeitsbereichskalibrierung stehen hierfür die Methoden `loadImagePlane()` und `loadAffineTransform()` aus der `PlaneAndAffineCalibration.cpp`-Klasse zur Verfügung.

5.3 Marker Detektion

Vera

Nach Abschluss der erfolgreichen und vollständigen Kalibrierung beziehungsweise des Importieren aller notwendigen Kalibrierungsdaten beginnt das Tracking Programm sowohl die ArUco Marker, als auch die leuchtend grünen Rechtecke zu detektieren. Die codebasierten *ArUco* Marker ermöglichen es den Marker nach langerer Verdeckung wieder eindeutig seiner entsprechenden ID zu zuordnen und somit Fehlzuordnungen oder sprunghaften Vertauschungen von benachbarten Würfel Markern vorzubeugen. Alle Methoden zur Detektierung der beiden Markertypen werden von der Klasse `MarkerDetection.cpp` zur Verfügung gestellt und mit der Funktion `runMarkerDetection`. Zunächst werden die Rechtecke mit Hilfe eines simplen Greenkeying erfasst und im Anschluss die erforderlichen *ArUco* IDs der Würfel Marker. Letztere können bekannterweise unter Umständen nur im ruhenden Status erfasst werden, da eine mögliche Bewegungsunschärfe bei schnellen Verschiebungen (siehe Kapitel 2.6) eine korrekte Erfassung nahezu unmöglich macht. Aus diesem Grund werden neben den codebasierten Muster auch die leuchtenden Rechtecke verfolgt um einen Marker auch bei schnelleren Bewegungen verfolgen zu können.

5.3.1 Detektion der grünen Rechtecke

Vera

Das Ziel dieses Verfahren ist es eine *Object Oriented Bounding Box*(OBB) von jedem erkannten Würfel Marker zu erstellen. Die OBB beinhalten Informationen über die Größe, den Schwerpunkt, die vier Eckpunkte sowie den Rotationswinkel im Verhältnis zum Bildkoordinatensystem. Alle genannten Informationen werden in einem `RotatedRect` gespeichert, welches eine Objektklasse der *OpenCV* Bibliothek ist. Zur Detektion der grünen Rechtecke müssen sie zunächst mit Hilfe einer Binarisierung segmentiert werden. Jedoch ist die Aufnahme zu diesem Zeitpunkt ein Bild mit RGB Farbmodus dessen Grundfarben von der Helligkeit abhängig sind. Somit müssen je nach Beleuchtungsverhältnisse die Thresholds einer Binarisierung angepasst werden. Im Gegensatz zum RGB Farbraum ist der HSI Farbraum unabhängig von der Beleuchtung, da er seine Grundbestandteile aus den Werten von Farbton (H), Sättigung (S) und Helligkeit (I) definiert Folgerichtig ist es theoretisch ausreichend nur den Farbton zu beachten um Objekte oder Teillbereiche einer bestimmten Farbe zu segmentieren. Die Binarisierung jedes Pixel $x \in (\mathbb{R}^3 | 0 \leq x_H \leq 255, 0 \leq x_S \leq 255, 0 \leq x_I \leq 255)$ unter den Bedingungen aus Formel 12, welche alle nicht grünen Bildsektionen schwarz maskiert. An dieser Stelle wurden neben dem Farbton auch die Sättigung und Helligkeit betrachtet. Durch die Eingrenzung der Sättigung wird sichergestellt, dass nur grüne Bereiche mit einer hohen Leuchtkraft berücksichtigt werden. Während die leichte Einschränkung der Helligkeit das Rauschen im Schwarz/ Weißbild verringert. Das verbleibende Bildrauschen wird durch einen Median Rangordnungsfilter mit einer Kernelgröße von 5×5

eliminiert. Diese beide Methoden sind Teil der *OpenCV* Bibliothek.

$$I(x) = \begin{cases} 255 & \text{wenn } 65 \geq x_H \leq 85, 120 \geq x_S \leq 255, 22 \geq x_I \leq 255 \\ 0 & \text{sonst} \end{cases} \quad (12)$$

Im binarisierte Bild werden im Anschluss zur Detektierung der OBB die Rechteckkanten mit einem Canny Kanten Detektor extrahiert. Im resultierenden Kantenbild wird mit Hilfe eines Kantenverfolgung Algorithmus [51] die Konturen generiert. Dieser Algorithmus erstellt eine hierarchische Sammlung von allen Punkten der äußensten Konturen eines Objektes. Aus dieser Sammlung wird die OBB mit dem *Douglas Ramer Peucker* Algorithmus [13] approximiert. Während der Approximation werden die Konturen geglättet und somit an ein ideales Rechteck angenähert. In Abbildung 15 ist sind die generierten OBBs weiß dargestellt und es ist deutlich zusehen, dass die Annäherung sehr genau ist.



Abbildung 15: Screenshot des Tracking Programms. Die weißen Konturen sind die detektierten grünen Rechtecke. Alle Beschriftungen der Marker zeigen die resultierenden Winkel an und der äußere rote Rahmen stellt das kalibrierte Spielfeld dar.

5.3.2 ArUco Marker Tracking

Vera

5.4 Marker Objekt Definition

Vera

Die Marker Objekte der Klasse `Marker.cpp` speichern alle benötigten Parameter und Informationen der jeweiligen Würfel Marker. Zu diesen Daten gehören unter anderem die Registrierungs-ID (RID)`id`, die aktuelle *ArUco* ID (AID) `arucoID` des zugeteilten Würfel Markers sowie die entsprechende OBB `rect` des letzten Frames. Weitere boolesche Variablen wie `isVis` und `isTrack` geben Auskunft über die Sichtbarkeit

der registrierten Würfel Marker in der letzten Aufnahme und ob diese Registrierung bereits in der aktuellen Iteration des Tracking Algorithmus verfolgt wurde.

Für die Winkel ω_{RR} , welche von den `RotatedRect` initialisiert werden gilt $\omega_{RR} \in \{\mathbb{R} | -90^\circ \geq \omega_{RR} \leq 90^\circ\}$. Doch alle folgenden Komponenten des Gesamtsystems von *MArC* erfordert einen Winkel $\omega_{WM} \in \{\mathbb{R}^+ | 0^\circ \geq \omega_{WM} \leq 360^\circ\}$, der vom unterem linken Eckpunkt des Würfel Markers abhängt. Dies wird gewährleistet indem mit der Funktion `computeAngle` der kompatible Winkel zwischen dem entsprechenden Eckpunkt des *ArUco* Markers p_{ul} und dem Schwerpunkt des Rechtecks c_{RR} mit der Formel 13 berechnet wird. Dieser Umstand wird darin begründet, dass eine einheitliche Ausrichtung aller Würfel Marker gewährleistet sein muss um die generierten Daten für weitere architektonische Planungsschritte zu verwenden. Während die Reihenfolge der *ArUco* Marker zuverlässig an dem unterem linken Punkt orientiert ist, stellte sich heraus, dass die Reihenfolge der vier Eckpunkte eines `RotatedRect` unbeständig ist. Somit ist es unvermeidbar die Eckpunkte der *ArUco Marker* als Referenzpunkt zu setzen und im Falle einer nicht Detektion den Winkel der letzten Iteration zu verwenden.

$$\omega_{WM} = \arccos \frac{u \times o}{\|u\| * \|o\|} \quad (13)$$

$$u = c_{RR} - \begin{pmatrix} const \\ 0 \end{pmatrix} - c_{RR} \quad (14)$$

$$o = p_{ul} - c_{RR} \quad (15)$$

Eine andere essentielle Komponente der Klasse sind die Transformationsvektoren v_t , welche die Transformation jedes Markers vom Objektraum in den Kameraraum M angeben. Diese Vektoren werden für die korrekte Abschätzung der Marker Position in der *Unity*-Welt (siehe Kapitel 5.2.4) benötigt und aus den vier äußeren Eckpunkten in der Methode `estimatePoseSingleMarkers` aus der *ArUco* Bibliothek angenähert.

5.5 Verfolgung und Registrierung der Marker

Vera

Die Verfolgung und Registrierung der Marker besteht zum Einen aus der Bestimmung und Abschätzung von allen OBB und zum Anderen aus der Zuordnung von den detektierten *ArUco*-Identitäten. Auf alle erforderlichen Berechnungen und der logische Ablauf zur Identifikation wird in der Funktion `trackMarker` der Klasse `MarkerManagement.cpp` verwiesen. Relevant bei diesem Prozess ist das Wissen, dass ein Würfel Marker nur Verfolgt werden kann, wenn eine entsprechende OBB $r_m \in \mathbb{R}^4$ erkannt wurde. Jede r_m besteht aus den vier Eckpunkten p_e^i mit $i = 4$ und einem Schwerpunkt p_c .

Zunächst wird für jede detektierte OBB jeweils ein Bewegungsvektor $v_m \in \mathbb{R}^2$ zwischen dem Schwerpunkt allen bekannten Schwerpunkten p_c^m aus der letzten Iteration $n-1$ nach der Formel 16 berechnet. An einem späteren Zeitpunkt werden diese Bewegungsvektoren verwendet um Marker innerhalb eines festgelegten Bewegungsradius

zu verfolgen.

$$v_m = p_c - p_c^m \text{ mit } m = \text{Anzahl der reg. Marker} \quad (16)$$

Nach Abschluss aller notwendigen Vorbereitungen beginnt der eigentliche Verfolgungsprozess. Dieser Prozess besteht aus vier Methoden zur Verfolgung und zwei Funktionen zum Suchen der zur AID gehörenden zugehörigen RID. Diese Funktionen werden in der Reihenfolge des Blockdiagramms aus Abbildung 16 abgerufen.

Der erste Schritt prüft mit der Methode `hasArucoID`, ob ein r_m des aktuellen Frames eine der detektierten AID enthält. Für diesen Test werden jeweils alle vier Eckpunkte des *ArUco*-Markers in zwei Dreiecke aufgeteilt und mit beiden wird ein Punkteinschlusstest mit Baryzentrischen Koordinaten durchgeführt [14]. Liegt der Schwerpunkt der OBB in einem der beiden Dreiecke umschließt das Rechteck einen *ArUco*-Marker. Bei diesem Test werden die Vorzeichen der Koordinaten betrachtet, denn ein Punkt liegt im Dreieck falls alle Vorzeichen gleich oder null sind liegt der Punkt im Dreieck. Der Vorteil dieses Tests ist, dass er mit wenigen kostenarmen Vektorberechnungen durchführbar ist und somit die Performanz des Tracking Algorithmus wenig beeinflusst.

Zunächst gehen wir auf das Ablaufdiagramm ein in dem Würfel Marker verfolgt werden für die eine AID existiert. Doch vor der Verfolgung muss diese eindeutig mit der Methode `findMatchID` einer RID zugeordnet werden, in dem die ermittelte AID mit den AID aller registrierten Marker abgleichen wird.



Abbildung 16

6 Ausblick

6.1 Trackingmethoden

Vera

6.2 AR Erweiterung mit der Webcam

Laura

6.3 Daten als Excel Tabelle

Paul

6.4 Validierung

Luke

7 Projektmanagement

7.1 Zeitplanung

Paul

7.2 Gant Chart

Paul

7.3 Planungsmethoden

Paul

7.4 Reflexion

7.4.1 Kommuniktion im Team und nach Außen

Paul

7.4.2 Probleme

Paul

8 Zusammenfassung

Lukas

9 Steckbrief

Vera

10 Anhang

10.1 *MarC* ReadMe-Datei

MArC ReadMe

Masterprojekt Lukas Kolhagen, Vera Brockmeyer, Laura Anger, Paul Berning

Starting the System

The following software is needed:

- IDS uEye Driver and SDK (<https://de.ids-imaging.com/download-ueye-win32.html>)
- Steam VR (<http://store.steampowered.com/about/>)
- Leap Driver (<https://developer.leapmotion.com/windows-vr>)
- The OpenCV Framework (<http://opencv.org/downloads.html>)

In order to start the system, make sure the following requirements are met:

- 2 Computers are available:
 - one that runs the IDS uEye tracking application [A] and
 - one for the Unity application [B]
- The HTC Vive head-mounted display has been connected to computer [B]
- The Leap Motion controller has been connected to computer [B]
- The IDS uEye software suite has been installed on computer [A]
- Computers [A] and [B] are connected via a network cable
- The IP address of the ethernet adapter of computer [A] has been set to 192.168.0.7 with the standard subnet mask of 255.255.255.0
- The IP address of the ethernet adapter of computer [B] has been set to 192.168.0.13 with the standard subnet mask of 255.255.255.0
- It is necessary to use the uEye camera with another computer than the rendering, because the performance could be worse while the computer renders the scene and does the tracking at the same time.

The start procedure is as follows:

- Start the application [Masterprojekt.exe] located in the [MArC_Tracking] directory on computer [A]
- Make sure the opened command window indicates that the server socket on port 10000 is set to listen
- Start the Unity build [MArC.exe] located in the [MArC_Unity] directory on computer [B]
- Follow the instructions on the screen of computer [B]
- If there is an offset between the table plane and the leap hand model, place the right hand onto the table and press the trigger button of the Vive Controller. This eliminates the height offset

Calibrating the System

- After starting the application on computer [B], the user will be asked whether he wants to calibrate the system
 - The user can choose between calibrating “camera and workspace” or only the “workspace”
- 1) Calibration of the uEye camera:
- NOTE: It is explicitly recommended to calibrate the camera only when it is needed!
 - Once chosen, a camera window will be opened on computer [A]. The following steps can be controlled on this computer, while computer [B] will be activated again, when it comes to the calibration of the workspace
 - Use the elevated chessboard to calibrate the system
 - Follow the instructions on the console of computer [A] and press “g”
 - The system will do 25 captures automatically and the user will hear a beep after each capture. (Taking the first 15 captures, while placing the elevated chessboard on the desk and the last 10 in a height of 20 cm leads to the best results)
 - After a longer beep (signal that the user has finished the capturing. Process) the calculation of the intrinsic camera parameters and distortion coefficients is made and a text file will be saved
 - This text file allows the user to use the system without making a new camera calibration by loading the parameters directly from the text file
 - After the calculation is completed, the calibration of the workspace can be done (see 2) Calibration of the workspace)
- 2) Calibration of the workspace:
- Follow the instructions displayed on computer [B]
 - Use the HTC Vive controller with the attached Aruco Marker (ID 49, dictionary: DICT_4X4_50) and press the button as described
 - 25 captures are necessary (each capture is confirmed by a haptic impulse)
 - NOTE: The order of the first four captures is important!
 - The user have to make sure to form up a rectangle which is equal to his desired workspace. Starting in the lower left corner and going clockwise
 - it is recommended to place the controller on different positions on the table for the next 11 captures as well
 - the last 10 captures can be done lifting the controller up to 20 cm and vary its position
 - After capturing all positions follow the instructions on computer [B]

Glossary of recurring terms:

Marker Cubes:

- There are two types of marker cubes: real ones that the user can touch and interact with and virtual ones that are rendered at the positions of the real world marker cubes

Context Menu:

- Every (virtual) marker cube has a menu that lets you scale the building associated with the marker and offers context information on the building (such as the living area or the number of floors). This is called the context menu.

Table Menu:

- On the right side of the workspace is a menu dedicated to saving and loading scenes. This is called the table menu.

Match Mode:

- When loading a previously saved scene, the match mode is entered. This mode is necessary to ensure that each virtual marker cube can be associated with a marker cube in the real world. In order to connect a virtual marker with a real one, position the real marker cube so that its position coincides with the virtual marker cube that is pulsating in red. As soon as the color of the virtual marker turns to green, the position is close enough and you may move on to the next marker (if there are more). Once this has been done for all markers the “Apply”-button appears on the table menu, which lets the user exit the match mode.

File overview of Unity assets:

1. Plug-In assets

/Assets/LeapMotion	Leap Motion core assets
/Assets/SteamVR	SteamVR core functionality necessary to run the HTC Vive as part of the Unity simulation

2. Script assets

/Assets/ObjectMenu/Scripts

ContextMenu.cs	Fills the object menu canvas with informations
contextMenuTrigger.cs	Handles the user input for the context menu
markerScale.cs	Changes the size of the cube when the context menu handlers are moved
MatchMode.cs	Controls the MatchMode of the markers
MatchModeReady.cs	Is attached to markers at the MatchMode, tells the system if a tcp-controlled marker is set correctly onto a loaded marker position
StopMatchMode.cs	Tells the system the status of the 4 triggers inside every cube while the MatchMode is running

/Assets/Skripts/

ControllerPos.cs	Handles the Vive controller button input
Marker.cs	Getter and setter for the markers
OutputNormalizedControllerPos.cs	Calculates the normalized position of the Vive controller
printFPS.cs	Debug script, shows FPS on an GUI
readInNetworkData.cs	Handles the incoming TCP datastream

setupScene.cs	Dynamic scene setup based on the calibration data
TableCalibration.cs	Calibrates the table in unity
DataHandler.cs	Handles the data copy process to tcp-controlled markers during the MatchMode
LeapHandCalib.cs	If there is an offset between the leap tracked hand and the plane, the offset is removed by this script

/Assets/Menus/

	Contains the StartMenu GUI scenes
/scripts/	Contains scripts for the startMenu GUI

/Assets/TableMenu/scripts

FileBrowser.cs	Contains functions for loading files
open.cs	Loads scenes from the saved xml documents
save.cs	Saves scenes as XML documents in /Resources/saves/
Timeline.cs	Handles the filebrowser ("timeline") in the table menu
TableMenuTrigger.cs	Handles the user input for the table menu

File overview Tracking:

1. External Sources

OpenCV Library	Image Processing Library
Aruco Library	part of OpenCV Library
uEye SDK	SDK runs, actuates and configures the uEye Camera
TCP Lib	Winsock

2. Framework

Classes

TCP.cpp	configures and runs TCP data connection, normalizes the marker pixel coordinates in order to the workspace size
uEye_input.cpp	configures the uEye parameters and runs the capturing
PoseEstimation.cpp	runs the OpenCV camera calibration and saves or loads the camera matrix and the distortion coefficients files
PlaneAndAffineCalibration.cpp	generates the workspace in picture space; calculates affine transformation from uEye camera space and unity world
CoordsTransformation2Unity.cpp	transforms marker center in uEye camera space to unity with affine Transform mat
Calibration.cpp	offers access to the Camera Calibration and Plane Calibration, offers all required attributes of both calibrations
MarkerManagement.cpp	organizes the current Markers and actualises the current position and angle; either matches recently detected Markers with current Markers or register new Marker; delete Markers if they out of workspace
MarkerDetection.cpp	detects the green rectangles and the

	aruco markers
IdMapping.cpp	offers all requested functions to compute matches of recently detected Markers and current Markers
Marker.cpp	Objectclass with all marker attributes
CreateCharucoBoard.cpp	generates CharucoBoard for Camera Calibration
ArucoCodeGenerator.cpp	generates new Aruco Codes to print

10.2 startTCPServer()-Methode

```

22 int TCP::startTCPServer()
23 {
24     SOCKADDR_IN addr;
25     // start Winsock
26     rc = startWinsock();
27     if (rc != 0) {
28         printf("ERROR: startWinsock, code: %d\n", rc);
29         return 1;
30     }
31     else {
32         printf("Winsock started!\n");
33     }
34     // build Socket
35     serverSocket = socket(AF_INET, SOCK_STREAM, 0);
36     if (serverSocket == INVALID_SOCKET) {
37         printf("ERROR: The Socket could not be build, code: %d\n", WSAGetLastError());
38         return 1;
39     }
40     else {
41         printf("Socket built!\n");
42     }
43
44     //define port
45     memset(&addr, 0, sizeof(SOCKADDR_IN));
46     addr.sin_family = AF_INET;
47     addr.sin_port = htons(10000);
48     addr.sin_addr.s_addr = ADDR_ANY;
49     rc = bind(serverSocket, (SOCKADDR*)&addr, sizeof(SOCKADDR_IN));
50     if (rc == SOCKET_ERROR) {
51         printf("ERROR: bind, code: %d\n", WSAGetLastError());
52         return 1;
53     }
54     else {
55         printf("port nr. is set to 10000\n");
56     }
57     // set server Socket in listen modus
58     rc = listen(serverSocket, 10);
59     if (rc == SOCKET_ERROR) {
60         printf("ERROR: listen, code: %d\n", WSAGetLastError());
61         return 1;
62     }
63     else {
64         printf("server Socket is set to listen....\n");
65     }
66
67     //accept connection
68     connectedSocket = accept(serverSocket, NULL, NULL);
69     if (connectedSocket == INVALID_SOCKET)
70     {
71         printf("EROOR: accept, code: %d\n", WSAGetLastError());
72         return 1;
73     }
74     else {
75         printf("New connection accepted!\n");
76     }
77 }
```

Quellcode-Auszug 9: startTCPServer()-Methode in TCP.cpp

10.3 getPointerOfMarkerVec()-Methode

```

131 void TCP::getPointerOfMarkerVec(std::array<Marker*, 100> allMarkers, std::vector<int>
132     takenIdVec, cv::Mat frame) {
133
134     myfile.open("logNorm.txt", std::ios::out | std::ios::app);
135     myfile << "Current Frame " << c << "\n";
136     c++;
137     myfile << "\t allMarkersSize() " << allMarkers.size() << "\n";
138
139     for (int i = 0; i < allMarkers.size(); i++) {
140         ms[i].id = allMarkers[i]->getId();
141
142         if (allMarkers[i]->getId() > 0)myfile << "\t tid " << ms[i].id << "\n";
143
144         ms[i].posX = allMarkers[i]->getEstimatedCenter().x;
145         ms[i].posY = allMarkers[i]->getEstimatedCenter().y;
146         ms[i].posZ = allMarkers[i]->getEstimatedCenter().z;
147
148         if (allMarkers[i]->getId() > 0) {
149             myfile << "\t posX " << allMarkers[i]->getEstimatedCenter().x << "\n";
150             myfile << "\t posY " << allMarkers[i]->getEstimatedCenter().y << "\n";
151             myfile << "\t posZ " << allMarkers[i]->getEstimatedCenter().z << "\n";
152         }
153
154         ms[i].angle = allMarkers[i]->getAngle();
155         if (allMarkers[i]->getId() > 0)myfile << "\t angle " << ms[i].angle << "\n";
156         ms[i].isVisible = allMarkers[i]->isVisible();
157         if (allMarkers[i]->getId() > 0)myfile << "\t isVisible " << ms[i].isVisible << "\n";
158     }
159
160     ms[allMarkers.size()].id = -2;
161
162     myfile.close();
163 }

```

Quellcode-Auszug 10: getPointerOfMarkerVec()-Methode in TCP.cpp

10.4 interpretTCPMarkerData()-Methode

```

141 // Convert byte[] data received over TCP to usable marker data
142 private void interpretTCPMarkerData(){
143     for (int i = 0; i < readBufferLength; i += bytesPerMarker){
144         int curID = System.BitConverter.ToInt32(readBuffer, i); // Convert the marker ID
145         if (curID == 0 && printMarkerDebugInfo){
146             Debug.Log("[READ IN NETWORK DATA] Start of frame " + frameCounter + ".");
147             continue;
148         }
149         if (curID == -1) { // Marker is empty
150             markers[i / bytesPerMarker] = new Marker(-1, 0.0f, 0.0f, 0.0f, 0);
151             continue;
152         }
153         if (curID == -2){ // End of frame reached
154             if(printMarkerDebugInfo)
155                 Debug.Log("[READ IN NETWORK DATA] Last marker reached, suspending loop for
156                 current frame.");
157             frameCounter++; // This is counted even if showMarkerDebugInfo is false, so that
158             // it can be enabled at any time
159             markers[i / bytesPerMarker] = new Marker(-2, 0.0f, 0.0f, 0.0f, 0); // Set last
160             // marker as EOF (end of frame)
161             break; // and suspend
162             loop;
163     }

```

```
159     }  
160     else if (curID < -2 || curID > markersToReceive){ // For debugging, this should  
161         not happen during normal operation  
162         Debug.LogError("[READ IN NETWORK DATA] Marker ID not valid: " + curID);  
163     }else{ // ID is valid and does not mark the end of the frame  
164         float curPosX = System.BitConverter.ToSingle(readBuffer, i + 4); // Convert the  
165             x-position  
166         float curPosY = System.BitConverter.ToSingle(readBuffer, i + 8); // Convert the  
167             y-position  
168         float curAngle = System.BitConverter.ToSingle(readBuffer, i + 12); // Conver the  
169             angle  
170         int status = System.BitConverter.ToInt32(readBuffer, i + 16); // Convert the  
171             status of the marker  
172         markers[i / bytesPerMarker] = new Marker(curID, curPosX, curPosY, curAngle,  
173             status); // Add new marker to array  
174         oneMarkerSet = true; // Give permission to use marker array since at least  
175             // one marker has been set for the current frame  
176         TCPText.text = markers[i / bytesPerMarker].ToStr(); // Set text on object menu  
177             canvas  
178         if (printMarkerDebugInfo)  
179             Debug.Log(markers[i / bytesPerMarker].ToStr()); // Print debug message  
180                 containing marker data  
181     }  
182 }  
183 }  
184 }
```

Quellcode-Auszug 11: interpretTCPMarkerData()-Methode in
readInNetWorkData.cs

Literatur

- [1] Unity. https://www.1stvision.com/cameras/IDS/dataman/uEyeLE_Brochure_english_ND.pdf. Aufgerufen: 13. März 2017.
- [2] Unity. <https://de.ids-imaging.com/ids-software-suite.html>. Aufgerufen: 13. März 2017.
- [3] RFC 793: Transmission Control Protocol. <https://rfc-editor.org/rfc/rfc793.txt>, 1981. Aufgerufen: 21. März 2017.
- [4] Iñigo Barandiaran, Céline Paloc, and Manuel Graña. Real-time optical markerless tracking for augmented reality applications. *Journal of Real-Time Image Processing*, 5(2):129–138, 2010.
- [5] G. Blasko and P. Fua. Real-time 3d object recognition for automatic tracker initialization. In *Augmented Reality, 2001. Proceedings. IEEE and ACM International Symposium on*, pages 175–176, 2001.
- [6] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210*. <http://www.w3.org/TR/1998/REC-xml-19980210>, 16:16, 1998.
- [7] Steve Bryson. Approaches to the successful design and implementation of vr applications. *Virtual reality applications*, pages 3–15, 1995.
- [8] Chi-Cheng P Chu, Tushar H Dani, and Rajit Gadh. Multi-sensory user interface for a virtual-reality-based computeraided design system. *Computer-Aided Design*, 29(10):709–725, 1997.
- [9] A. I. Comport, E. Marchand, M. Pressigout, and F. Chaumette. Real-time markerless tracking for augmented reality: the virtual visual servoing framework. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):615–628, July 2006.
- [10] Creative. Creative Senz3D, Tiefen- und Gestenerkennungskamera für PCs. <http://de.creative.com/p/web-cameras/creative-senz3d>. Aufgerufen: 19. März 2017.
- [11] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek. A brief introduction to opencv. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1725–1730, May 2012.
- [12] Dima Damen, Pished Bunnun, Andrew Calway, and Walterio Mayol-cuevas. Realtime learning and detection of 3d texture-less objects: A scalable approach. In *in British Machine Vision Conference (BMVC)*, 2012.
- [13] David H. Douglas and Thomas K. Peucker. *Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature*, pages 15–28. John Wiley and Sons, Ltd, 2011.

- [14] Gerald Farin and Diane Hansford. *Lineare Algebra: Ein geometrischer Zugang*. Springer-Verlag, 2013.
 - [15] O. Faugeras. *Three-dimensional Computer Vision: A Geometric Viewpoint*. Artificial intelligence. MIT Press, 1993.
 - [16] M. Fiala. Designing highly reliable fiducial markers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(7):1317–1324, July 2010.
 - [17] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.
 - [18] Daniel Flohr and Jan Fischer. A lightweight id-based extension for marker tracking systems. In *Eurographics Symposium on Virtual Environments (EGVE) Short Paper Proceedings*, pages 59–64, 2007.
 - [19] S. Garrido-Jurado, R. Mu noz Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.
 - [20] S. Garrido-Jurado, R. Mu noz Salinas, F.J. Madrid-Cuevas, and R. Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51:481 – 491, 2016.
 - [21] Google. Cardboard. <https://vr.google.com/cardboard/>. Aufgerufen: 20. März 2017.
 - [22] John Haas. *A History of the Unity Game Engine*. PhD thesis, Worcester Polytechnic Institute, 2014.
 - [23] V. Hayward, R. O. Astley, M. Cruz-Hernandez, D. Grant, and G. Robles-De-La-Torre. Haptic interfaces and devices. *Sensor Review*, 24(1), 2004. Aufgerufen: 22. März 2017.
 - [24] S. Hinterstoisser, C. Cagniart, S. Ilic, P. Sturm, N. Navab, P. Fua, and V. Lepetit. Gradient response maps for real-time detection of textureless objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(5):876–888, May 2012.
 - [25] HTC. Htc vive. <https://www.vive.com/>. Aufgerufen: 30. November 2016.
 - [26] HTC. HTC Vive – Für Vive geeignete Computer. <https://www.vive.com/de/ready/>. Aufgerufen: 18. März 2017.
 - [27] ITU-T. Data Networks and Open System Communication. Open Systems Interconnection - Model and Notation. <http://handle.itu.int/11.1002/1000/2820>, 1994. Aufgerufen: 21. März 2017.
-

- [28] H. Kato and M. Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Augmented Reality, 1999. (IWAR '99) Proceedings. 2nd IEEE and ACM International Workshop on*, pages 85–94, 1999.
 - [29] Leap Motion. Leap Motion. <https://www.leapmotion.com/>. Aufgerufen: 30. November 2016.
 - [30] Leap Motion. Leap Motion Blog. <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>. Aufgerufen: 2. Januar 2017.
 - [31] Leap Motion. Support – USB 3.0. <https://support.leapmotion.com/hc/en-us/articles/223783688-Would-the-Leap-Motion-Control-work-on-USB-3-0->. Aufgerufen: 2. Januar 2017.
 - [32] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
 - [33] H. Álvarez, I. Aguinaga, and D. Borro. Providing guidance for maintenance operations using automatic markerless augmented reality system. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 181–190, Oct 2011.
 - [34] Andreas Meisel. *3D-Bildverarbeitung für feste und bewegte Kameras*. PhD thesis, Braunschweig [u.a.], 1994. Zugl.: Aachen, Techn. Hochsch., Diss., 1993 u.d.T.: Meisel, Andreas: 3D-Bildverarbeitung für feste und bewegte Kameras auf photogrammetrischer Basis.
 - [35] Microsoft. Introducing Visual Studio. [https://msdn.microsoft.com/en-us/library/fx6bk1f4\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/fx6bk1f4(v=vs.90).aspx). Aufgerufen: 18. März 2017.
 - [36] Mobile World Congress. Mobile World Congress. <https://www.mobileworldcongress.com/>. Aufgerufen: 30. November 2016.
 - [37] Mozilla. Introducing the WebVR 1.0 API Proposal. <https://hacks.mozilla.org/2016/03/introducing-the-webvr-1-0-api-proposal/>. Aufgerufen: 20. März 2017.
 - [38] Oculus. Oculus – Unity Intro. <https://developer3.oculus.com/documentation/game-engines/latest/concepts/unity-intro/>. Aufgerufen: 14. März 2017.
 - [39] OpenCV. Camera Calibration and 3D Reconstruction. http://docs.opencv.org/3.1.0/d9/d0c/group__calib3d.html. Aufgerufen: 18. März 2017.
 - [40] OpenCV. Detection of ArUco Markers. http://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html. Aufgerufen: 18. März 2017.
-

- [41] Ovrvision Pro. Informationen für Entwickler. <http://ovrvision.com/setup-en/>. Aufgerufen: 14. März 2017.
 - [42] Ovrvision Pro. Ovrvision Pro. <http://ovrvision.com/>. Aufgerufen: 30. November 2016.
 - [43] Ovrvision Pro. Produktdetails. <http://ovrvision.com/product-en/>. Aufgerufen: 9. März 2017.
 - [44] Y. Park, V. Lepetit, and W. Woo. Texture-less object tracking with online training using an rgb-d camera. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 121–126, Oct 2011.
 - [45] J. Postel. RFC 768. User Datagram Protocol. <https://tools.ietf.org/html/rfc768>, August 1980. Aufgerufen: 21. März 2017.
 - [46] Bob Quinn. *Windows sockets network programming*. Addison-Wesley Longman Publishing Co., Inc., 1998.
 - [47] T. Rahman and N. Krouglisoff. An efficient camera calibration technique offering robustness and accuracy over a wide range of lens distortion. *IEEE Transactions on Image Processing*, 21(2):626–637, Feb 2012.
 - [48] Éric Marchand and François Chaumette. Feature tracking for visual servoing purposes. *Robotics and Autonomous Systems*, 52(1):53 – 70, 2005. Advances in Robot Vision.
 - [49] Samsung. Samsung Explores the World of Mobile Virtual Reality with Gear VR. <http://www.samsungmobilepress.com/press/Samsung-Explores-the-World-of-Mobile-Virtual-Reality-with-Gear-VR?2014-09-03>. Aufgerufen: 20. März 2017.
 - [50] William R Sherman and Alan B Craig. *Understanding virtual reality: Interface, application, and design*. Elsevier, 2002.
 - [51] Satoshi Suzuki and Keiichi A be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32 – 46, 1985.
 - [52] Unity Technologies. Unity. <https://unity3d.com/de>. Aufgerufen: 8. März 2017.
 - [53] Unity Technologies. Unity – Multiplatform. <https://unity3d.com/unity/multiplatform>. Aufgerufen: 14. März 2017.
 - [54] Unity Technologies. Unity – Public Relations. <https://unity3d.com/public-relations>. Aufgerufen: 14. März 2017.
 - [55] Unity Technologies. Unity – VR Overview. <https://unity3d.com/de/learn/tutorials/topics/virtual-reality/vr-overview>. Aufgerufen: 14. März 2017.
-

- [56] L. Vacchetti, V. Lepetit, and P. Fua. Stable real-time 3d tracking using online and offline information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1385–1391, Oct 2004.
- [57] Valve. Valve Software. <http://www.valvesoftware.com/>. Aufgerufen: 30. November 2016.
- [58] Daniel Wagner, Gerhard Reitmayr, Alessandro Mulloni, Tom Drummond, and Dieter Schmalstieg. Real-time detection and tracking for augmented reality on mobile phones. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):355–368, May 2010.
- [59] Daniel Wagner and Dieter Schmalstieg. Artoolkitplus for pose tracking on mobile devices, 2007.
- [60] Li-Chen Wu, I-Chen Lin, and Ming-Han Tsai. Augmented reality instruction for object assembly based on markerless tracking. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D ’16, pages 95–102, New York, NY, USA, 2016. ACM.
- [61] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, Nov 2000.