



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	8
1.1 Motivation	8
1.2 Anwendungskontext	8
1.3 Projektziel	9
2 Grundlagen und Stand der Wissenschaft	9
2.1 Architektur von Virtual-Reality-Anwendungen	9
2.2 Haptische Interaktionsmethoden in VR-Umgebungen	11
2.3 Handtracking	11
2.4 ISO/OSI-7-Schichtenmodell	12
2.5 Bedienkonzepte in VR-Umgebungen	13
2.6 Marker Tracking	14
3 Materialien	17
3.1 Hardware	17
3.1.1 Computer für Unity-Simulation	17
3.1.2 Computer für Tracking-Anwendung	17
3.1.3 HTC Vive	18
3.1.4 IDS uEye 164LE-C	18
3.1.5 Leap Motion Controller	19
3.1.6 Würfel-Marker	20
3.1.7 Schachbrett-Kalibrierungshelfer zur Kamerakalibrierung	21
3.1.8 Kalibrierungscontroller zur Arbeitsbereichskalibrierung	21
3.2 Obsolete Hardware	22
3.2.1 Ovrvision Pro	23
3.2.2 Webcam	24
3.3 Software	24
3.3.1 Unity	24
3.3.2 Visual Studio 2015	25
3.3.3 OpenCV	25
3.3.4 ArUco Bibliothek	25
3.3.5 Steam VR	26

3.3.6	Windows Sockets (Winsock)	27
3.3.7	Leap Motion SDK	27
4	System	28
4.1	Aufbau	28
4.1.1	Tracking-Aufbau	28
4.1.2	Unity-Aufbau	29
4.2	Systemvoraussetzungen	30
4.3	Netzwerk	30
4.3.1	Serverseitige Netzwerkanbindung	30
4.3.2	Clientseitige Netzwerkanbindung	33
4.4	Starten des Systems	35
4.5	Menüführung	35
4.5.1	Menüs	35
4.5.2	Ablauf der Menüführung	37
4.6	Kontextmenü	39
4.6.1	Anfasser für Marker-Skalierung	40
4.6.2	Ein- und Ausblenden des Kontextmenüs	41
4.6.3	Berechnung der Gebäudeeigenschaften	42
4.7	Tischmenü	42
4.7.1	Szenen-Management	43
4.7.2	Speichern von Szenen	43
4.7.3	Laden von Szenen	45
4.7.4	Match-Modus	46
5	Tracking-Anwendung	47
5.1	uEye Ansteuerung	48
5.2	Kalibrierung	48
5.2.1	Korrespondierende Punktepaare	49
5.2.2	Definition der Koordinatensysteme	49
5.2.3	Kamerakalibrierung	50
5.2.4	Kalibrierung des Arbeitsbereichs	53
5.2.5	Kalibrierungsfehler	55
5.2.6	Schnittstelle der Kalibrierungen	56

5.2.7	Speichern und Laden der Kalibrierungen	57
5.3	Marker-Detektion	58
5.3.1	Detektion der grünen Rechtecke	59
5.3.2	ArUco Marker-Detektion	60
5.4	Definition des Marker-Objekts	61
5.5	Tracking und Registrierung der Marker	62
6	Ausblick	66
6.1	Trackingmethoden	66
6.2	Erweiterung von <i>MArC</i> als Augmented-Reality-System	66
6.3	Export von Architekturdaten	67
6.4	Evaluierung von <i>MArC</i>	67
7	Projektmanagement	68
7.1	Projektdefinition	68
7.1.1	Problemanalyse	68
7.1.2	Projektziele und Anforderungen	68
7.1.3	Lösungskonzept	69
7.1.4	Durchführbarkeitsanalyse	70
7.1.5	Projektauftragsformular	71
7.1.6	Projektorganisation	73
7.2	Projektplanung	73
7.2.1	Arbeitspakete	73
7.2.2	Projektstrukturplan	73
7.2.3	Ablaufplan, Terminplan (Gantt Chart)	73
7.2.4	Kapazitätsplan	75
7.2.5	Kostenplan	75
7.2.6	Qualitätsplan	75
7.3	Projektdurchführung	75
7.3.1	Kommunikation im Team und nach Außen	75
7.3.2	Maßnahmen zur Problemvermeidung	75
7.4	Projektabchluss	77
7.4.1	Abschlusspräsentation	77
7.4.2	Video	78

7.5	Reflexion	78
8	Zusammenfassung	79
9	Anhang	81
9.1	Schachbrett zur Kamerakalibrierung	81
9.2	<i>MArC</i> ReadMe-Datei	82
9.3	<code>startTCPServer()</code> -Methode	90
9.4	<code>getPointerOfMarkerVec()</code> -Methode	90
9.5	<code>interpretTCPMarkerData()</code> -Methode	91

Abbildungsverzeichnis

1	Binäre Muster für markerbasiertes Tracking	16
2	Aufbau des <i>Leap Motion</i> Controllers	19
3	Würfel-Marker	21
4	Schachbrett-Kalibrierungshelfer für die Kamerakalibrierung	21
5	Kalibrierungscontroller des <i>MArC</i> -Systems	22
6	<i>Orvision Pro</i> Stereokamera	24
7	Verwendete <i>ArUco</i> -Marker	26
8	<i>ArUco</i> -Marker mit unterschiedlicher Bitgröße	26
9	Montage der uEye Camera	29
10	Aufbau des <i>MArC</i> -Systems	30
11	Auszug aus der <i>MArC</i> ReadMe-Datei	31
12	Flussdiagramm der Menüführung	38
13	Zwei Beispiele für ein Kontextmenü eines Würfels mit ID 1	40
14	Collider zum Ein- und Ausblenden des Kontextmenüs	41
15	Unskalierter Standard-Marker mit geöffnetem Kontextmenü	42
16	Tischmenü von <i>MArC</i>	43
17	Match-Modus von <i>MArC</i>	47
18	Lage des Kamerakoordinatensystems	50
19	Begrenzung des Arbeitsbereichs in der <i>Unity</i> -Simulation	54
20	Kalibrierungsfehler	56
21	Screenshot der Tracking-Anwendung	60
22	Flussdiagramm des Marker-Trackings	64
23	Schachbrett zur Kamerakalibrierung	81
24	Ressourcenplan von <i>MArC</i>	93
25	Mitarbeiterressourcenplan von <i>MArC</i>	94
26	Kostenplan von <i>MArC</i>	95
27	Qualitätsplan von <i>MArC</i>	96
28	Gantt-Chart von <i>MArC</i>	97
29	Projekt Strukturplan von <i>MArC</i>	98

Tabellenverzeichnis

1	ISO-/OSI-7-Schichtenmodell	13
2	Übersicht technische Daten des Computers für <i>Unity</i> -Simulation	17
3	<i>HTC Vive</i> Systemvoraussetzungen	18
4	Technische Daten des <i>Acer E5-571G-795A</i>	18
5	Bildmodi der <i>Ovrvision Pro</i> Stereokamera	23
6	Parametrisierung der <i>uEye</i> -Kamera für das Tracking	48
7	Parameter und Berechnungsschritte der Kamerakalibrierung	51
8	Projekt-Auftragsformular von <i>Marc</i>	72
9	Identifikation der Arbeitspakete und Meilensteine	74

1 Einleitung

Virtuelle und erweiterte Realität (auch „VR“ und „AR“ genannt) sind 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 Head-Mounted Display (HMD) 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

Zu Beginn der Entwicklung von *MArC* lernte das Team die Arbeitsmethoden von Architekten in einem Architekturbüro kennen. Dabei lag der Fokus vor allem auf der Planung und dem Entwurf von Siedlungen. Dabei stellte sich heraus, dass zur Zeit ein komplizierter Arbeitsablauf notwendig ist, um die anfänglich noch sehr rudimentären 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

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 Her-

angehensweise 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 späterem manuellem Wiederaufbau. Zudem ist diese Vorgehensweise sehr platzintensiv.

1.3 Projektziel

MArC, der „Mixed Reality Architecture Composer“ soll die bisherigen Schwierigkeiten im Entstehungsprozess von Siedlungen beheben und diesen erleichtern. 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 und Stand der Wissenschaft

Im Folgenden werden die Grundlagen und der Stand der Wissenschaft von allen Teilbereichen des *MArC*-Systems erläutert. Es werden Veröffentlichungen von ähnlichen Architektur VR-Anwendungen, haptischen Interaktionsmethoden sowie Handtracking-Methoden erläutert. Des weiteren wird auf die Grundlagen der Netzwerktechnologie und den bekanntesten Bedienkonzepten in VR-Umgebungen eingegangen. Zuletzt werden die wichtigsten Methoden zu des Marker-Tracking zusammengefasst.

2.1 Architektur von Virtual-Reality-Anwendungen

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

- Die virtuelle Umgebung und das damit verbundene Interface sollten auf die vorliegende Aufgabe zugeschnitten werden, da die Komplexität der Interaktion deutlich höher ist als bei nicht-VR-Anwendungen 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. Beispiele für Desktop-VR-Applikationen sind insbesondere 3D-Spiele, die bereits ohne den Zusatz von Virtual Reality häufig hohe Anforderungen an die Computerhardware stellen.

Mobile-VR-Applikationen: Mobile Anwendungen vereinen häufig (jedoch nicht immer) alle notwendige Hardware in einem Gerät, wie etwa einem Smartphone oder Tablet-Computer. Dies führt generell zu einer geringeren Anwendungsperformanz, weshalb Anwendungen für mobile VR-Geräte tendenziell „simpler“ ausgelegt sind.

Beispiele für mobile VR-Anwendungen sind *Samsung GearVR* [60] und *Google Cardboard* [25]. 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 [46] 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 Abschnitt 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 Abschnitt 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

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 solcher Systeme mit haptischem Feedback haben ihren Ansatzpunkt an den Fingern, oder genauer gesagt an den Fingerspitzen. Dazu zählt unter anderem der Datenhandschuh *Exoskelett* von *Dexta Robotics* [26]. Es gibt aber auch Systeme die mehrere Parts des Körpers oder sogar den ganzen Körper mit einbeziehen. Ein Beispiel dafür ist der *Synesthesia Suit* [34].

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 [28].

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 (vgl. Abschnitt 3.1.6) handelt, an deren Position virtuelle Objekte gerendert werden können. In diesem sogenannten 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 größer (oder auch flacher) als sein reales Pendant sein kann, zu transformieren.

2.3 Handtracking

Handtracking ist mittels verschiedener Trackingsysteme möglich. Alle Systeme haben hierbei ihre Vor- und Nachteile.

Beispielsweise ist es möglich, die Bewegungen der Hand mittels eines Datenhandschuhs zu erfassen. Dies ist ein mit Sensoren ausgestatteter Handschuh, den der Nutzer an den Händen trägt. Eine Möglichkeit die Fingerbewegung mit einem Datenhandschuh zu erfassen sind Biegesensoren. Diese funktionieren zum Beispiel mit Glasfasern, die an den Fingern entlang laufen. Wird ein Finger gebogen, wird die Intensität des Lichts, welches durch die gebogene Glasfaser geschickt wird, leicht abgeschwächt. Hierdurch ist eine Biegung erkennbar [16] Eine weitere Möglichkeit des Handtrackings ist ein Exosklett. Hierbei wird eine Mechanik an dem Handschuh angebracht, mittels welcher Bewegungen auf Sensoren übertragen werden[16] Datenhandschuhe sind für den Nutzer einschränkend und aus hygienischen Gründen, abhängig von der Anwendung, nicht immer die Beste Wahl zum Handtracking. Eine bessere Möglichkeit ist das optische Tracking der Hände. Systeme wie der *Leap Motion* Controller erfassen mit Kameras die Hände und rechnen aus den stereoskopischen Bildinformationen die Winkel und Positionen der verschiedenen Fingerglieder aus.

Dieses System ist einfach zu benutzen. Sobald der Nutzer seine Hand in den vom *Leap Motion* Controller abgedeckten Bereich hält, werden diese erkannt.

Typisch für optische Systeme ist jedoch das Problem der Verdeckung. Wird die Hand aus einer bestimmten Richtung durch Kameras gefilmt, kann es passieren, dass einige Finger andere überdecken. Teilweise kann dann nicht exakt unterschieden werden, welches Fingerglied zu welchem Finger oder zu welchem Teil der Hand gehört. Dies wirkt sich bei der Benutzung in einem „hüpfenden“ Verhalten oder unmenschlich verrenkter Finger der gerenderten Hand aus.

2.4 ISO/OSI-7-Schichtenmodell

Um die Kommunikation zwischen unterschiedlichsten technischen Systemen zu ermöglichen und zu vereinheitlichen dient das *ISO/OSI-7-Schichtenmodell* [32], 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 jeweils klar eingegrenzten Teilbereich der Kommunikation zuständig sind. Die Netzwerkprotokolle, 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 Schich-

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

ten [32].

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 [55] 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. 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. Wegen dieser automatischen Fehlerkorrektur wurde TCP für die Kommunikation zwischen den beiden Computern bei *MArC* verwendet. Der genaue Netzwerkaufbau kann in Kapitel 4.3 nachvollzogen werden.

2.5 Bedienkonzepte in VR-Umgebungen

Die Bedienung eines Systems – vor allem die Dateneingabe und Objektmanipulation – in einer Virtual Reality (VR) Umgebung unterscheidet sich stark von der konventionellen Bedienung mit Maus und Tastatur [9]. 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 [62] 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

In einer VR- oder AR-Umgebung ist zur interaktiven Positionierung eines 3D Objektes die präzise Bestimmung der Orientierung und Position im 3D-Zielraum notwendig [23]. 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 aufzunehmen. In den resultierenden Bildsequenzen werden die physischen Objekte anhand von festgelegten Merkmalen erkannt. Diese Merkmale können sowohl natürlicher Art sein oder in Form von künstlich erstellten Codes festgelegt 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 [4].

Natürliche Merkmale zur Identifizierung der Objekte sind Textureigenschaften, Kanteninformationen oder sogenannte Keypoints. Eine der robustesten und simpelsten Methoden ist die Verwendung von Keypoints [4][71][10][40], die sowohl in der Ausgangs- als auch in der Kamerawelt bekannt beziehungsweise ermittelbar sind. Diese werden mit Hilfe einer Homographie zur Übereinstimmung gebracht. Daraus resultiert die notwendige Transformation zwischen den Welten (vgl. Abschnitt 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 [29][13][54] entwickelt die sich speziell für texturarme Objekte eignen, wie etwa die

in *MArC* verwendeten grünen Rechtecken der Würfel-Marker (vgl. Abschnitt 3.1.6). Andere Autoren verwenden sehr rechenintensive Kantenerkennungsalgorithmen, wie den *Moving Edges Algorithm* [59]. 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 [69][41][73][5].

Im Gegensatz zu *Marker-based Tracking* benötigt *Marker-less Tracking* keine Veränderung oder Anpassung der realen Welt, wie zum Beispiel das Markieren der Marker durch die genannten Codemuster. Doch die Parameter, welche den Tracking-Algorithmus beeinflussen, können nicht ohne Weiteres kontrolliert werden [4]. Dennoch ist das *Marker-less Tracking* von Objekten sehr rechenaufwändig und eine eindeutige Zuordnung beziehungsweise Identifikation von ähnlichen oder gleichförmigen Objekten, wie zum Beispiel 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 bekannte *QR*-Verfahren, *ARToolKit*[33], *ARToolKit Plus*[72], *ARTag* [20], *BinARyID* [22] und *ArUco*[23]. 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 Bitanzahl der Codes zurückzuführen macht die Erkennung in Aufnahmen aus größerem Abstand und möglicher mit Bewegungsunschärfe ungleich schwerer. Es kommt unter diesen Umständen häufiger zu Fehlinterpretationen und Ausfällen. Gerade in einem System wie *MArC* tritt Bewegungsunschärfe sehr häufig auf, da die Würfel-Marker regelmäßig bewegt werden. Auch der verhältnismäßig große Abstand der Kamera zu den Würfel-Markern lässt die Markermuster 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 groben Mustern, wie zum Beispiel *ArUco* Marker mit einer 16-Bit-Kodierung. Diese Marker Bibliothek ist ein *OpenCV*-Modul (vgl. Abschnitt 3.3.3), welche alle notwendigen Funktionalitäten und Ressourcen für das Tracking und die Orientierungsbestimmung der physische Objekt enthält. Ein weiterer großer Vorteil dieser Bibliothek ist, dass die Bitgröße und maximale Anzahl der Identitäten explizit ausgewählt werden kann.

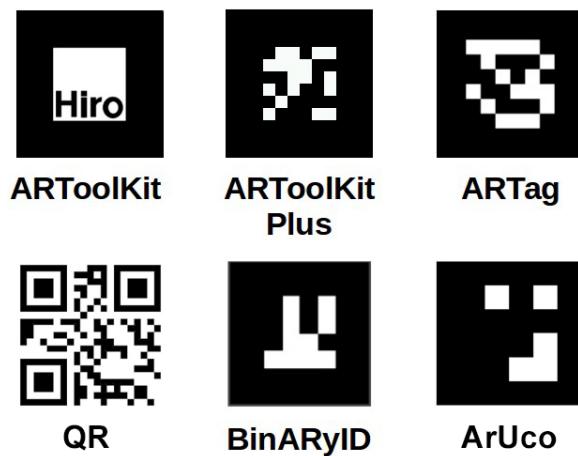


Abbildung 1: Diverse Binäre Muster, die als Code zur eindeutigen Identifizierung für markerbasiertes Tracking verwendet werden. Quelle: [23]

3 Materialien

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

Des weiteren werden auch Komponenten vorgestellt, die während der Projektlaufzeit erstellt wurden, wie etwa die Würfel-Marker (vgl. Abbildung 3).

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 für Unity-Simulation

Die Anwendung, welche aus *Unity* [65] 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 für Tracking-Anwendung

Aufgrund der Erfahrung, welche während der Entwicklung gemacht wurden, ist es zwingend notwendig einen weiteren Rechner an das Gesamtsystem zu koppeln, welcher ausschließlich für die Ansteuerung der uEye-Kamera und die Berechnungen des Tracking-Algorithmus zuständig ist. Bei Versuchen mit einem einzelnen Rechner wurde eine erhebliche Reduzierung der Framerate von den angeschlossenen Kameras beobachtet. An den Computer für die *Unity*-Simulation sind gezwungenermaßen

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.

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 [31].

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.

viele USB-Komponenten angeschlossen, wie zum Beispiel der *Leap Motion* Controller und die *HTC Vive*. Dies kann zu einer hohen Auslastung der Bandbreite von der USB-Karte führen und aufgrund dessen ist es nicht mehr möglich die uEye-Kamera mit der notwendigen maximalen Framerate zu betreiben. Somit wird für ein flüssiges Tracking der Acer E5-571G-795A mit den Eigenschaften aus Tabelle 4 verwendet.

3.1.3 HTC Vive

Die *HTC Vive* ist ein Head-Mounted Display, welches von *HTC* in Kooperation mit *Valve* [70] produziert wird. Vorgestellt wurde dieses am 1. März 2015 im Vorfeld des *Mobile World Congress* [45].

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 [30]. Alle technischen Systemvoraussetzungen können in Tabelle 3 eingesehen werden.

Zur Positionsbestimmung im Raum wird die Lighthouse-Technologie [15] von *Valve* genutzt. Zusätzlich sind neben einem Gyroskop auch ein Beschleunigungssensor und ein Laser-Positionsmesser verbaut. Mittels proprietärer Hand-Controller wird bei der *HTC Vive* eine Interaktion mit virtuellen Objekten ermöglicht.

3.1.4 IDS uEye 164LE-C

Die Kamera *uEye 164LE-C* wurde vom Hersteller *IDS Imaging Development Systems GmbH* entwickelt. Sie besitzt eine Bildauflösung von 1280×1024 Pixel und

ermöglicht Live-Videoaufnahmen im RGB-Farbmodus mit maximal 25 Bildern pro Sekunde. Der integrierte CMOS Bildsensor wird im Rolling-Shutter-Modus betrieben und ermöglicht Belichtungszeiten von $37\mu s$ bis 10 s. Weiterführend kann sie universell mit allen gängigen Computern oder Systemen via USB 2.0 Schnittstelle verbunden werden [2].

Die erforderliche Ansteuerung der *uEye 164LE-C* erfolgt mit Hilfe der bereitgestellten *IDS Software Suite*. In diese Suite ist die *uEye-API* integriert, welche die Entwicklung von eigenen Programmen unter den Betriebssystemen *Windows* und *Linux* mit den Programmiersprachen *C++*, *.NET*, *C#* und *C* ermöglicht [1]. Für das Tracking der Marker in *MarC* wurde eine eigene Schnittstelle in *C++* erstellt, welche die Kamera im Live Modus initialisiert und steuert (vgl. Abschnitt 5.1).

3.1.5 Leap Motion Controller

Der *Leap Motion Controller* [35] ist ein $7,6 \times 3 \times 1,3\text{ cm}$ großes Gerät, welches es mit Hilfe von Sensoren möglich macht, Hand- und Fingerbewegungen zu erkennen und diese als Eingabemöglichkeit zu nutzen. Die Idee dahinter ist, ein Eingabegerät für virtuelle Umgebungen analog zu konventionellen Eingabegeräten wie einer Maus zu schaffen, welches keinen direkten Kontakt bzw. keine Berührung benötigt. Hergestellt wird der *Leap Motion Controller* von der US-amerikanischen Firma *Leap Motion Inc.*, welche am 1. November 2010 gegründet wurde [35].

Wie in Abbildung 2 gezeigt, besteht das Gerät im wesentlichen 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 des *Leap Motion Controllers* ähnelt einer umgedrehten Pyramide mit einem Flächeninhalt von knapp 243 cm^2 . Die Reichweite des Geräts ist maßgeblich durch die Reichweite der LEDs begrenzt. Die Lichtintensität der LEDs ist wiederum durch den maximalen Strom, der über die USB-Verbindung fließt beschränkt [36]

Für das Projekt wurde der *Leap Motion Controller* zur Interaktion mit den virtuellen Menüs verwendet. Dabei wurde das Gerät am *HTC Vive Head-Mounted Display* befestigt und so ein Interaktionsraum vor dem Gesicht des Nutzers aufgespannt.



Abbildung 2: Aufbau des *Leap Motion Controllers* [36].

3.1.6 Würfel-Marker

In vielen VR- oder AR-Umgebungen müssen Nutzer eines Systems häufig nach virtuellen Objekten zur Interaktion greifen, die nicht real existieren. Demzufolge greifen die Personen ins Leere, was die Immersion erheblich stört und zu Irritationen sowie Unsicherheit führt. Um den Benutzern von *MArC* für die Positionierung und Orientierung ein reales haptisches Feedback zu ermöglichen, wurden zwölf Aluminiumwürfel mit aufgeklebten Markern angefertigt. 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 virtuellen Welt ein explizit zugeordnetes Objekt gerendert. Diese Position wird mit Hilfe der Tracking-Anwendung (vgl. Abschnitt 5) aus den Bildern der *uEye*-Kamera ermittelt. Alle zwölf Marker stimmen mit der Form und Farbe, sowie Material und Oberflächenbeschaffenheit aus Abbildung 3 überein. Sie haben eine Kantenlänge von 46 mm und sind mit einer 0,5 mm/45° Fase an allen Kanten versehen. Zur Erzeugung einer matten Oberfläche ist das Aluminium glasperlgestrahlt. Dies vermeidet ungewollte Reflexionen und Überstrahlungen, 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 das Tracking der Marker auch bei Bewegungsunschärfe ermöglicht. Die leuchtend grüne Farbe wurde ausgewählt, da sie aufgrund 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 Muster-basierten Markern auch zur Berechnung der Orientierung des Würfel-Markers 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 und wurden jeweils aus dem Marker-Dictionary DICT_4X4_50 des ArUco-Moduls [49] generiert. Die maximale Anzahl von IDs ist mit 50 ausreichend für den Prototypen von *MArC*, da die Anzahl von 50 Würfel-Markern die durchschnittliche Tischplattenfläche ausfüllt. Ein weiterer Vorteil dieses verhältnismäßig kleinen Dictionarys ist, dass auch der Aufwand für den entsprechenden Abgleich einer ID mit den potentiellen Mustern im Dictionary erheblich reduziert werden kann. Weiter beinhaltet das DICT_4X4_50 Dictionary aufgrund seiner 16-Bit-Kodierung sehr grobe und einfache Muster, welche gegen die mögliche Bewegungsunschärfe der Kamerabilder robuster ist. Bei sehr feinen Strukturen kann es schneller zu einer Verwischung des gesamten Musters kommen und die Wahrscheinlichkeit einer erfolgreichen Erkennung sinkt.

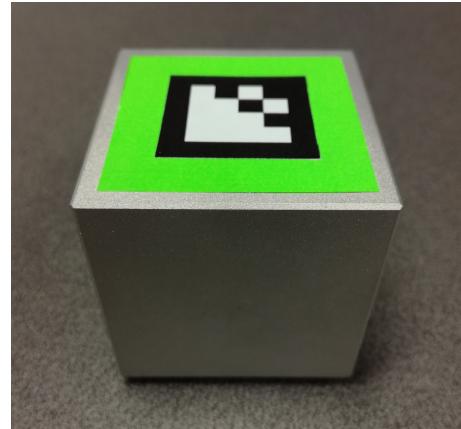


Abbildung 3: Würfel-Marker mit grüner Fläche und einem ArUco-Marker, die mittig auf dem Aluminiumwürfel aufgebracht sind.

3.1.7 Schachbrett-Kalibrierungshelfer zur Kamerakalibrierung

Der in Abbildung 4 gezeigte Kalibrierungshelfer wird für die Kamerakalibrierung (vgl. Abschnitt 5.2.3) der *uEye*-Kamera, deren Eigenschaften in Abschnitt 3.1.4 beschrieben werden, verwendet. Dazu wurde eine Tisch-förmige Erhöhung gebaut, die genau so hoch ist wie der *ArUco*-Marker, der an dem Controller zur Arbeitsbereichskalibrierung, wie in Abschnitt 3.1.8 beschrieben, angebracht ist. Darauf ist ein ausgedrucktes Schachbrettmuster mit 8×10 Feldern. Davon werden von dem verwendeten Algorithmus nur die inneren Felder gezählt, also 7×9 Felder. Das Schachbrett so wie der zugehörige Download-Link sind im Anhang in Abschnitt 9.1 zu finden.



Abbildung 4: Schachbrett-Kalibrierungshelfer für die Kamerakalibrierung.

3.1.8 Kalibrierungscontroller zur Arbeitsbereichskalibrierung

Für die Kalibrierung des Arbeitsbereichs, die in Abschnitt 5.2.4 beschrieben wird, wurde ein Controller der *HTC Vive* wie in Abbildung 5 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 der Kalibrierung wackelt.

Bei der Anbringung des eben erwähnten *ArUco*-Markers, der in Abbildung 7 zu sehen ist, ist eine genaue Positionierung wichtig. Er muss genau mittig unterhalb des ringförmigen Abschlusses des *HTC Vive*-Controllers aufgebracht werden, so wie es auf der Abbildung zu sehen ist. Dies ist entscheidend für das spätere Verfahren, welches auf der Zuordnung korrespondierender Punktepaare basiert (vgl. Abschnitt 5.2.1). Um eine leichtere Aufbringung und gute Ausrichtung des *ArUco*-Markers zu ermöglichen, wurde dafür eine kleine Auflagefläche am Controller angebracht. Diese ist so befestigt, dass der Mittelpunkt des *ArUco*-Markers sich möglichst genau an der Stelle befindet, wo auch die Position des Controllers in der *Unity*-Simulation gemessen wird.



Abbildung 5: Kalibrierungscontroller des *MArC*-Systems mit *ArUco* Marker.

3.2 Obsolete Hardware

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*

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 [53].

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 stattdessen 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

Die *Ovrvision Pro* (vgl. Abb. 6) ist eine kompakte Stereokamera, welche über USB 3.0 mit einem Computer verbunden wird [52]. 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 [51].

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 6: *Ovrvision Pro* Stereokamera montiert an *Oculus Rift* (links) und *HTC Vive* (rechts) [52].

3.2.2 Webcam

Die *Creative Senz3D* ist eine RGB-Kamera mit einer zusätzlichen Infrarot-Tiefenkamera. Das generierte RGB-Bild hat eine Auflösung von 1280×720 Pixel und ein Tiefenbild von 320×240 Pixel bei einer Reichweite von 1599 cm. Zusätzlich wurde eine Sichtfeld von 74° ermittelt. 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 [11]. Weiter ist es möglich die Kamera direkt aus Anwendungen per *Intel Perceptual Computing SDK* anzusteuern.

3.3 Software

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

3.3.1 Unity

Unity ist eine sogenannte Spiel-Engine, also eine Entwicklungs- und Laufzeitumgebung, die speziell auf die Entwicklung von 3D-Spielen ausgelegt ist. Die Software wurde am 6. Juni 2005 veröffentlicht [27] und wird von *Unity Technologies* [65] 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 [67].

Unity bietet eine sehr breite Plattformunterstützung [66] und erlaubt ebenso die Entwicklung für Head-Mounted Displays, wie etwa die *Oculus Rift* [47][68] oder auch die in diesem Projekt verwendete *HTC Vive* [68].

Die zu Beginn des Projekts verwendete Stereokamera *Ovrvision Pro* stellt ein Software-Development-Kit (SDK) für *Unity* (Version 5) zur Verfügung [51]. 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

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 Web-Applikationen und Webservices [44] programmieren und anschließend kompilieren. Für *MArC* wurde mit der Version 14.0.25123.00 Update 2 gearbeitet.

3.3.3 OpenCV

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 [12].

Für dieses System ist vor allem das Modul `calib3d` [48] und das extra Modul `aruco` [49] verwendet. Das erste Modul `calib3d` bietet alle notwendigen Funktionen zur Erstellung, Verwendung und Weiterverarbeitung von intrinsischen und extrinsischen Kamerakalibrierungen an (vgl. Abschnitt 5.2). Während das Zweite alle benötigten Ressourcen und Funktionalitäten zum Tracking von *ArUco* Markern zur Verfügung stellt (vgl. Abschnitt 3.3.4).

3.3.4 ArUco Bibliothek

Die *ArUco*-Bibliothek ist ein Marker-Tracking-Modul von *OpenCV* (vgl. Abschnitt 3.3.3), dass für Augmented Reality (AR) Anwendungen genutzt werden kann. Es stellt für diese Anwendungen alle notwendigen Funktionalitäten zum Orten und Verifizieren der Codes sowie einer Positionsabschätzung der ermittelten Positionen im Kamerraraum zur Verfügung [23]. Die Marker bestehen ähnlich wie QR-Codes aus einer zweidimensionalen Matrix, mit schwarzen oder weißen Feldern, welche die kodierten Daten binär, wie in Abbildung 7, darstellen. Weiterführend kann die Bitanzahl (vgl. Abbildung 8) und die gefragte maximale Markeranzahl variabel gewählt werden. An dieser Stelle ist eine kleine Bitanzahl, welche detailarme Muster erzeugt, für eine gute Erkennbarkeit in sehr großen Entfernung zur Kamera oder kleinen Bildern sinnvoll. Um diese Vielzahl an verschiedenen Größen und IDs zu verwal-

ten wurden sogenannte Dictionaries eingeführt [24]. Diese Dictionaries bestehen aus Markern mit gleicher Bitanzahl und sind zusätzlich auf eine maximalen Anzahl von IDs begrenzt um ein möglichst hohe Performanz während des Zuordnungsprozesses zu gewährleisten.

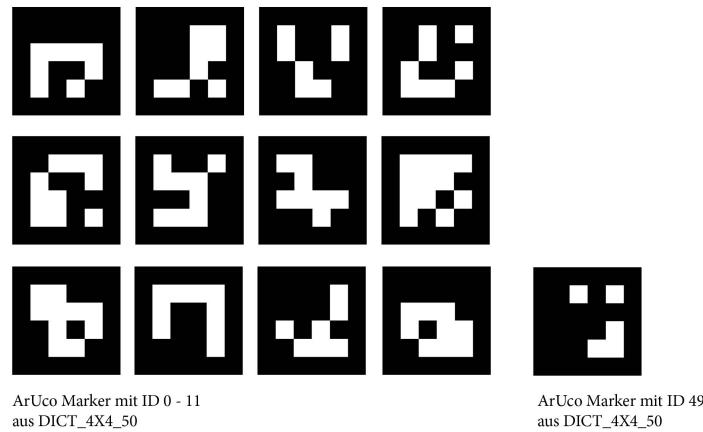


Abbildung 7: Alle genutzten 16-Bit-kodierten *ArUco*-Marker des Prototypen. Links die zwölf ID der Würfel-Marker und rechts die ID, welche zur Kalibrierung benötigt wird. Die maximale Anzahl der Marker im Dictionary ist auf 50 begrenzt.

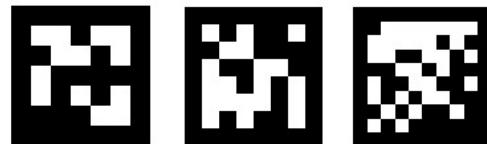


Abbildung 8: *ArUco*-Marker mit unterschiedlicher Bitgröße. Von Links nach Rechts: $n = 5$, $n = 6$ und $n = 8$. Quelle: [23]

3.3.5 Steam VR

Steam VR [63] ist die Schnittstelle zwischen der *HTC Vive* und *Unity*. Um das HMD nutzen zu können, muss *Steam VR* auf dem Computer installiert sein. Für den Nutzer ist ein kleines GUI Element auf dem Monitor sichtbar, welches den Status der Geräte der *Vive* darstellt. Hierdurch werden Fehlermeldungen kommuniziert, Kalibrierungen durchgeführt und eine Kommunikation mit dem HMD bereitgestellt, so dass das Gerät im Fall der Fälle neu gestartet werden kann.

Innerhalb von *Unity* stellt *Steam* ein Plugin zur Verfügung, welches direkt in Szenen in *Unity* eingebettet werden kann. Der Entwickler ist also in der Lage, eine vorhandene *Unity*-Szene um die VR Möglichkeit bequem per Drag-and-drop-Technik zu erweitern.

Das bereitgestellte *Unity*-Prefab beinhaltet alle notwendigen Elemente um mit der Hardware kommunizieren zu können. Dabei wird eine Positionsbestimmung ebenso

wie ein Kamera Rig für die stereoskopische Bildwiedergabe bereitgestellt, wie auch die Controllereingabe und Weiterverwendung der Daten möglich gemacht.

3.3.6 Windows Sockets (Winsock)

Windows Sockets (abgekürzt Winsock) ist eine API für den Zugriff auf Netzwerkkomponenten in Microsoft Windows Betriebssystemen [56]. 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 (vgl. Abschnitt 2.4). 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.

3.3.7 Leap Motion SDK

Leap Motion Inc. bietet ein vollständiges Software Development Kit (SDK) für den *Leap Motion Controller* an, welches die *Unity*-Applikation um das Hand- und Fingertracking erweitert [37]. Die Treibersoftware interpretiert die von den Kameras gelieferten Daten und sendet Trackinginformationen an die jeweilige Software. Das *Leap Motion 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 Motion Inc. bietet zudem mehrere sogenannte „Module“ an, 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 geben [38].

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.

4 System

Die Benutzung von *MArC* ist in dem Programm mitgelieferten ReadMe-Datei (vgl. Abschnitt 9.2) 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

Der Aufbau des *MArC*-Systems kann in zwei Teile aufgeteilt werden: Der erste Teil ist für das Tracking der Würfel-Marker verantwortlich (nachfolgend „Tracking-Aufbau“ genannt) und der zweite erzeugt die Virtual-Reality-Umgebung (nachfolgend „Unity-Aufbau“ genannt, da auf dieser Seite Unity als zentrale Software läuft). Letzterer stellt auch die notwendige Peripherie für die Interaktionen. In den beiden folgenden Abschnitten werden beide Teile des Systemaufbaus näher beschrieben.

4.1.1 Tracking-Aufbau

Wie in Abbildung 10 dargestellt ist, wird senkrecht über einem beliebigen Tisch eine Kamera installiert mit der die Würfel-Marker aus der Vogelperspektive gefilmt werden. Der Abstand zum Tisch ist so gewählt, dass die Aufnahmen noch manuell am Objektiv scharf gestellt werden können und ein möglichst großer Ausschnitt des Tisches abgebildet wird. Hier ist besondere Vorsicht geboten, da die Nutzer durch die *HTC Vive* nicht die reale Umgebung wahrnehmen können und es einer niedrigen Montage zu Kollisionen kommen kann.

Für den Prototypen wurde eine *IDS uEye 164LE-C* (vgl. Abschnitt 3.1.4) verwendet und über eine USB 2.0 Schnittstelle an den Computer mit der Tracking-Anwendung verbunden. Um diese Industriekamera zu befestigen wurde eine Halterung, wie in Abbildung 9, gebaut mit der eine Montage unter der Raumdecke möglich ist. Diese Kamera wird mit Hilfe der *uEye*-API der Tracking-Anwendung im Live-Bild-Modus initialisiert und gesteuert. In den resultierten Live Bildern werden die Würfel-Marker erkannt und verfolgt. Für jeden erkannten Würfel-Marker werden alle relevanten Informationen über die TCP Netzwerkverbindung (vgl. Abschnitt 4.3) an den Computer zur Ausführung der *Unity*-Simulation (vgl. Abschnitt 3.1.1) übertragen. Damit der Algorithmus einen Würfel-Marker erkennt muss dieser innerhalb des vorab definierten Arbeitsbereich bewegt werden. Die Definition findet während der Kalibrierung des Arbeitsbereiches 5.2.4 statt. Für eine erfolgreiche Kalibrierung ist es zudem notwendig den einzelnen *ArUco*-Marker mit der ID 49 (vgl. Abbildung 7) exakt mittig auf den Controller der *HTC Vice* zu montieren (vgl. Abbildung 5). Nur so kann gewährleistet werden, dass die erkannte *ArUco*-Marker Positionen in der Kamerawelt mit den Controller Positionen in der *Unity*-Welt korrespondieren.



Abbildung 9: Montage der uEye Camera.

4.1.2 Unity-Aufbau

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 10 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 [39].

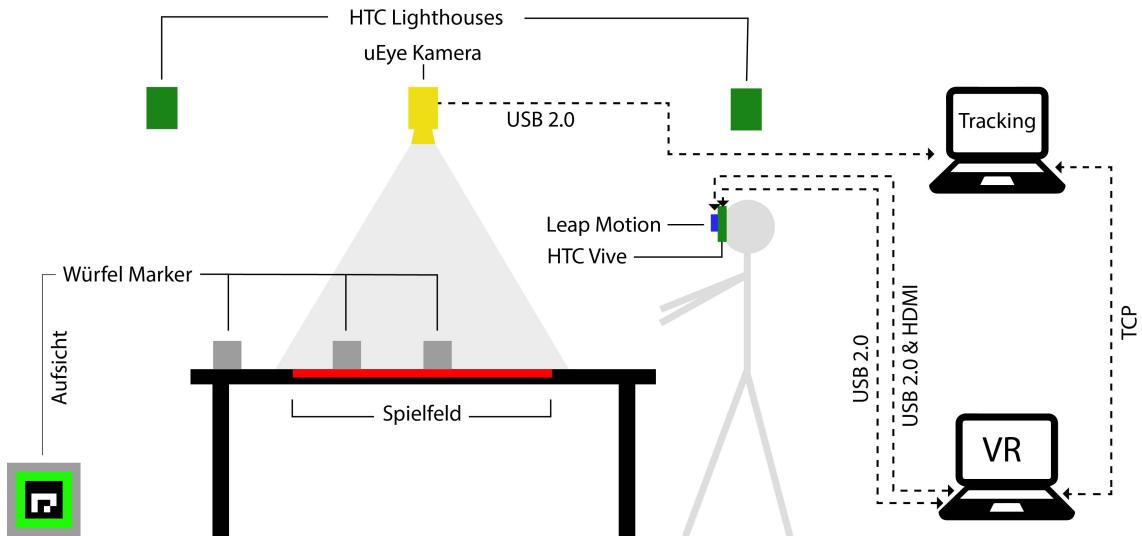


Abbildung 10: Aufbau des *MArC*-Systems.

4.2 Systemvoraussetzungen

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 9.2 zu finden, außerdem enthält Abbildung 11 einen Ausschnitt der ReadMe-Datei, welcher die Voraussetzungen beschreibt, die vor dem Starten des Systems erfüllt sein müssen.

4.3 Netzwerk

Die in Kapitel 3.3.6 beschriebene Winsock-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 Serverseitige Netzwerkanbindung

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

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 11: Auszug aus der *MArC* ReadMe-Datei (vgl. 9.2).

(vgl. Abschnitt 3.3.6) durch das Einbinden von `<winsock2.h>` und `<windows.h>` bereit und werden, wie in Quellcode-Auszug 9 (im Anhang einsehbar) gezeigt, verwendet.

Datenübertragung Die Datenübertragung kann grob 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 *HTC Vive* Controllers während der Arbeitsbereichskalibrierung (vgl. Abschnitt 5.2.4 empfangen werden.

```

88 void TCP::sendStatus(int status) {
89     const char far* markerPointer = (const char*)&status;
90     send(connectSocket, 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 (vgl. Quellcode-Auszug 2). Beide Methoden gehen von einem Status aus, der genau einen `int`-Wert enthält, also 4 Byte groß ist.

```

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 Methode zum Senden der Daten eines Markers – also dessen ID, Position, Winkel und Sichtbarkeits-Status – ist in Quellcode-Auszug 3 zu sehen. Die 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 }
```

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 umfangreicherem `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. Abschnitt 5.2.4) wird die Position des *HTC Vive* Controllers nach jedem Auslösen des Triggers von der *Unity*-Simulation an die Tracking-Anwendung gesendet. Die entsprechende Methode zum Empfangen der dreidimensionalen Position in *Unity*-Koordinaten ist in Quellcode-Auszug 5 aufgeführt. Neben dem reinen Empfang der Daten, der äquivalent zur `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(connecteSocket, 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 Clientseitige Netzwerkanbindung

Aufbauen der Netzwerkverbindung Das Aufbauen der Netzwerkverbindung vom Client zum Server, also von der *Unity*-Simulation zur Tracking-Anwendung, geschieht mit Hilfe der von Winsock (vgl. Abschnitt 3.3.6) 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 herstellten 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 vor.

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 Unitykoordinaten 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-Simulation 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 Würfel-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

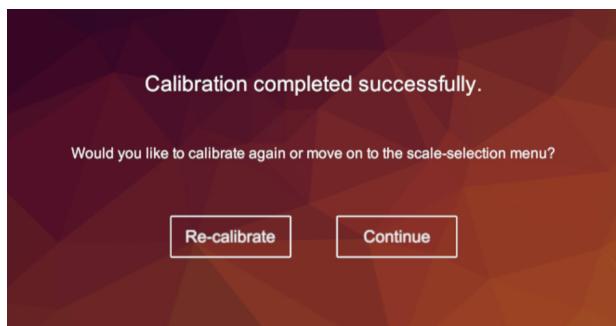
Nachdem sichergestellt wurde, dass alle in [4.2](#) beschriebenen Voraussetzungen bestehen, kann das System gestartet werden, indem zunächst die Tracking-Anwendung (auf dem einen Computer, vgl. Abschnitt [3.1.2](#)) und anschließend die aus *Unity* heraus erstellte Anwendung (auf dem anderen Computer,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

4.5 Menüführung

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

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. Zusätzlich wird die Option geboten, erneut zu kalibrieren.

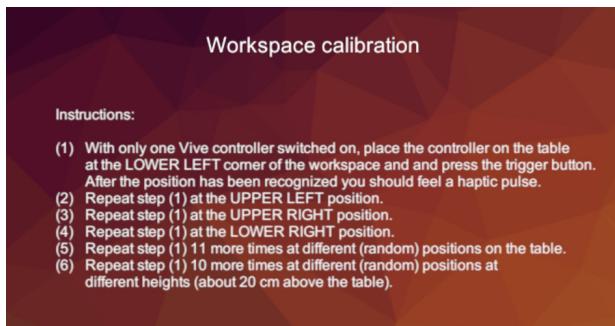


CalibrateOrNot:

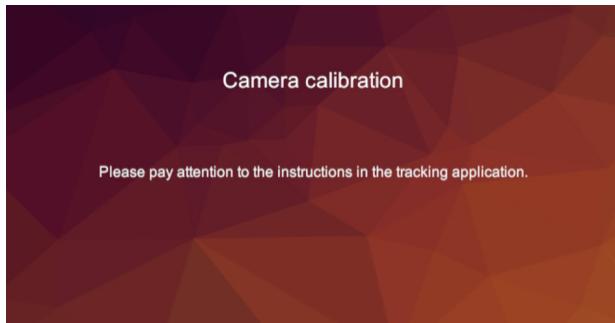
Erscheint nach dem Verlassen des Welcome-Menüs und erlaubt dem Benutzer eine Kalibrierung durchzuführen oder eine bereits durchgeführte Kalibrierung zu laden.

**ControllerNotFound:**

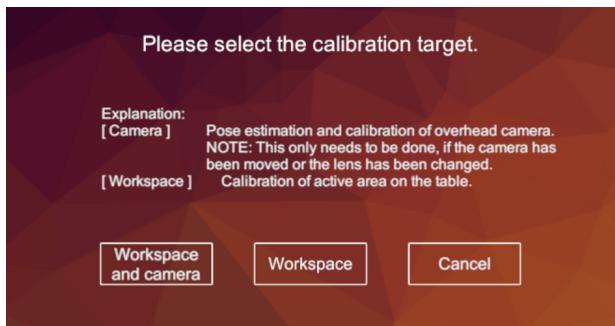
Warnt den Benutzer nach dem Starten der Kalibrierung, dass der *HTC Vive Controller*, welcher für die Kalibrierung benötigt wird, nicht eingeschaltet ist. Während das Menü angezeigt wird, kann der Benutzer den Controller einschalten und anschließend auf **Continue** klicken.

**doPlaneCalibInVS:**

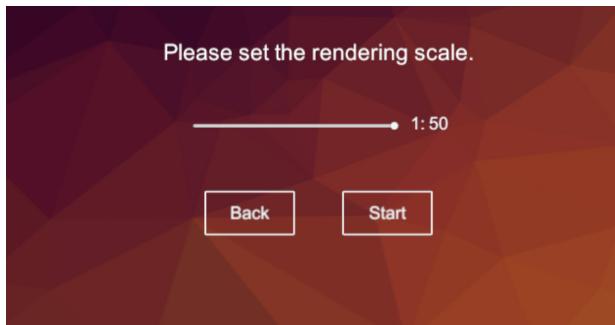
Dient dem Benutzer als Anleitung für die Durchführung der Kalibrierung des Arbeitsbereichs. Diese wird in [5.2.4](#) genauer beschrieben.

**doPoseCalibInVS:**

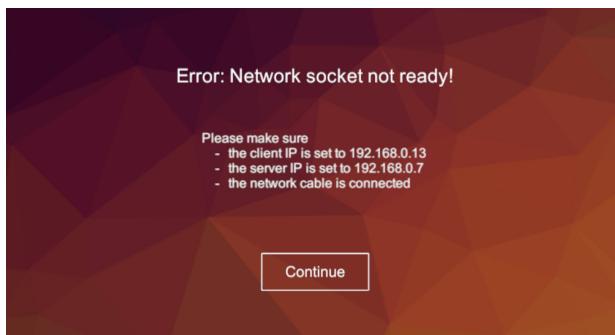
Dient dem Benutzer als Anleitung für die Durchführung der Kamerakalibrierung. Diese wird in [5.2.3](#) genauer beschrieben.

**SelectCalibrationTarget:**

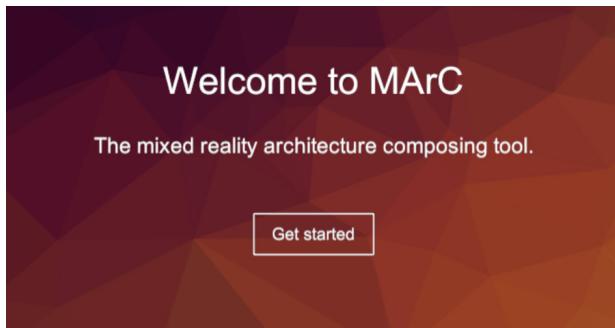
Erlaubt die Auswahl der Art der Kalibrierung. Es kann hier entweder nur der Arbeitsbereich oder sowohl der Arbeitsbereich, als auch die Kamera kalibriert werden. Die Kalibrierung ist näher in [5.2](#) beschrieben.

**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

Der Ablauf der Menüführung von *MArC* ist in Abbildung 12 dargestellt. Die einzelnen Menüs sind bereits in 4.5.1 beschrieben worden.

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`

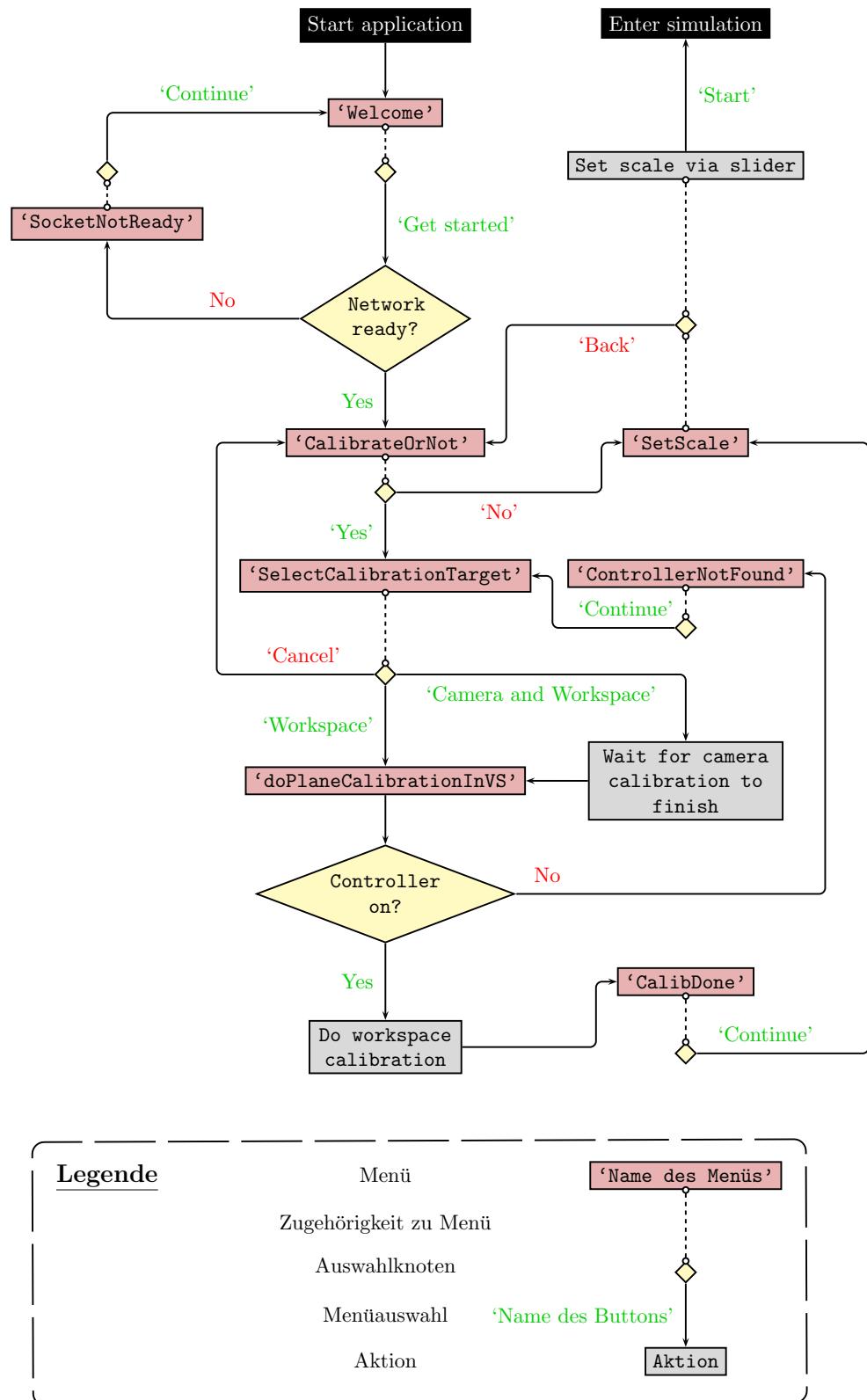


Abbildung 12: Flussdiagramm der Menüführung.

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

Die virtuellen Objekte, die den in Kapitel 3.1.6 beschriebenen Aluminiumwürfeln zugeordnet werden, sind zunächst würfelförmig und können zur Laufzeit der Anwendung vom Benutzer verschoben, rotiert und skaliert werden. Diese virtuellen Objekte repräsentieren bei *MArC* die Gebäude. 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 13 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. Abschnitt 7.1.1). 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.

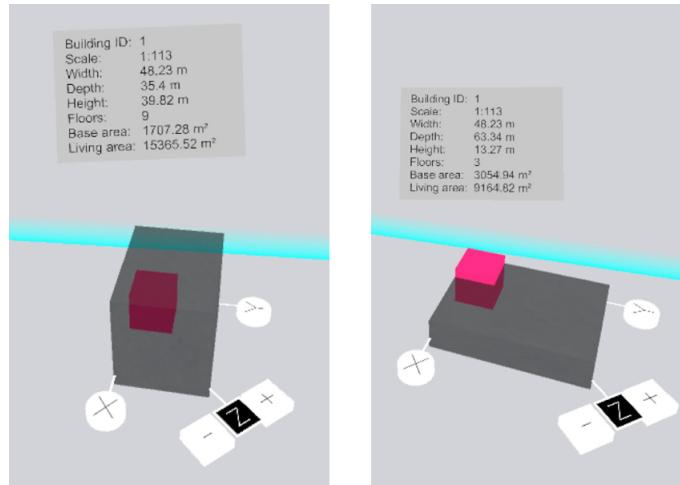


Abbildung 13: Zwei Beispiele für ein Kontextmenü eines Würfels mit ID 1.

4.6.1 Anfasser für Marker-Skalierung

Jeder Marker verfügt über die Möglichkeit in seiner Größe in der X-, Y- und Z-Achse verändert zu werden. Diese Anfasser sind sichtbar, wenn das Kontextmenü angezeigt wird (vgl. Abschnitt 4.6). Die Anfasser bestehen aus Zylindern, die mit einem Collider versehen wurden. Die X- und Y-Achse wird betätigt, indem der Benutzer mit einem Finger in den jeweiligen Anfasser hineinfasst und dann in die entsprechende Richtung zieht. Die Größe in Z-Richtung wird über Schaltflächen gesteuert, die mit „+“ und „-“ beschriftet sind. Drückt der Benutzer diese Schaltflächen, wird die Höhe des Gebäudes um je ein Stockwerk erhöht oder verringert.

Die Interaktion der Anfasser mit der Hand des Nutzers wurde über das *Leap Motion* Handmodell umgesetzt. Dieses besitzt Collider, die an den Fingern des Modells angebracht sind und sich mit der Hand des Nutzers mitbewegen. Berührt die Hand nun einen der Anfasser, wird dieser Collider des Anfassers „angesprochen“. Im Falle der X-, und Y-Anfasser 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-Richtung wird die sogenannte `localDifference` berechnet, dies ist der Abstand des Fingers zur Position des Markers. Die eigentliche Skalierung wird dann wie folgt berechnet:

```
1 pos = (startPosition.x - localDifference.y) / 2, startPosition.y, startPosition.z)
```

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

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

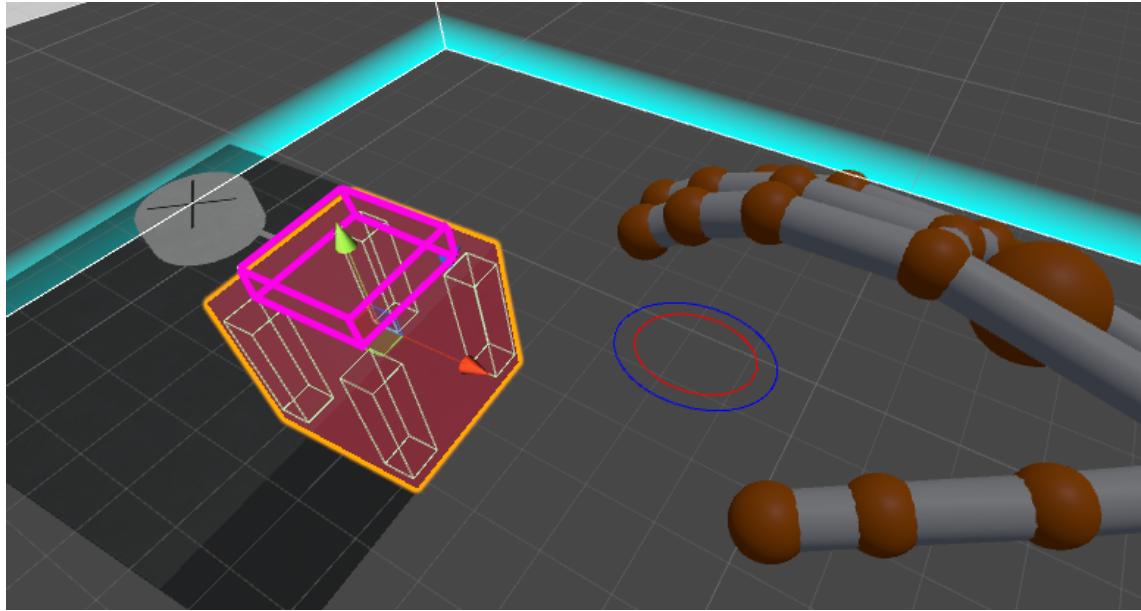


Abbildung 14: Collider zum Ein- und Ausblenden des Kontextmenüs (in Pink eingefärbt).

```
1 pos = (startPosition.x, startPosition.y, (startPosition.z - localDifference.y) / 2)
```

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

4.6.2 Ein- und Ausblenden des Kontextmenüs

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. 14). 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.

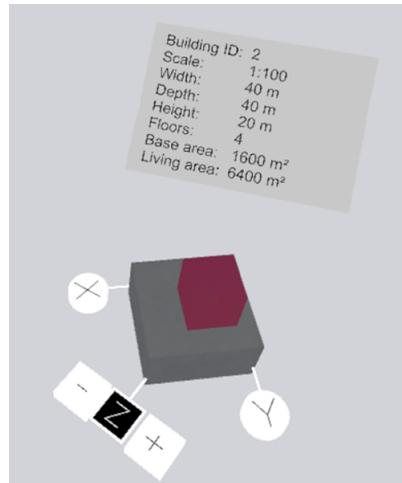


Abbildung 15: Unskalierter Standard-Marker mit geöffnetem Kontextmenü.

4.6.3 Berechnung der Gebäudeeigenschaften

Wie in Abbildung 15 zu sehen, enthält das Kontextmenü neben der anwendungsspezifischen ID des jeweiligen Gebäudes noch eine Reihe weiterer Informationen über dieses. Zunächst wird der Maßstab, den der Benutzer zu Beginn der Anwendung im `SetScale`-Menü (vgl. Abschnitt 4.5.2) ausgewählt hat, angegeben. Wenn das Objekt nicht bereits durch den Benutzer anhand der Handles transformiert wurde, sieht es aus wie in Abbildung 15. Diese zeigt einen Standard-Marker mit vier Geschossen. Über den Z-Anfasser (vgl. Abschnitt 4.6.1) können einzelne Geschosse hinzugefügt oder entfernt werden. Was im Kontextmenü zum einen natürlich direkten Einfluss auf die Geschossigkeit hat und zum anderen zur Veränderung der Höhe des jeweiligen Gebäudes führt. Neben der Höhe sind die Breite und Tiefe, sowie die Grund- und Wohnfläche von der Skalierung des virtuellen Markers abhängig. Alle im Kontextmenü dargestellten Größen, bis auf die ID und die Geschossigkeit, sind von dem ebenfalls dort aufgeführten und bereits erwähnten Maßstab abhängig.

Es ist natürlich denkbar die im Kontextmenü angegebenen Eigenschaften eines Objektes an die individuellen Bedürfnisse und Vorstellungen des Benutzers anzupassen. Die entsprechenden Berechnungen finden sich in der `ContextMenu.cs`-Klasse.

4.7 Tischmenü

Zur intuitiven Bedienung von *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. Das Tischmenü ist in Abbildung 16 zu sehen.

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



Abbildung 16: Das Tischmenü von *MArC*.

Das Tischmenü und dessen Funktionen werden nachfolgend im Detail erläutert.

4.7.1 Szenen-Management

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

- das Speichern von Szenen,
- das Laden von Szenen und aufrufen des *Match-Modus* (s. Abschnitt 4.7.4), sowie
- das Blättern durch Seiten mit gespeicherten Szenen.

4.7.2 Speichern von Szenen

Möchte der Nutzer eine Szene speichern, so drückt er mit dem Finger auf den Button „Save“ (vgl. Abbildung 16). *MArC* speichert die aktuelle Szene automatisch im Ordner \Resources\saves\ mit einem Dateinamen, der sich aus dem Speicherzeitpunkt und -datum wie folgt zusammensetzt:

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

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

Da innerhalb von *Unity* eine hierarchische Anordnung sämtlicher Elemente in Form eines Szenenraphen 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 Datenobjekten (XML Documents) und ermöglicht das hierarchische Abspeichern von geparsten oder ungeparsten Daten in Form von Textdateien. Die erzeugten Textdateien sind für den Menschen lesbar und die gespeicherte Hierarchie erfassbar. Details zu *XML* können in [6] nachgelesen werden. In *C#* sind Verarbeitungsklassen bereits implementiert. Mittels dieser sogenannten *XML Prozessoren* wird ein Datenzugriff erleichtert.

Dateiaufbau einer gespeicherten Szene Das gesamte Dokument wird mit einem Hauptknoten <AR2_COMPOSERSCENE> umspannt. Innerhalb diesem wird mit <time> ein Zeitstempel gespeichert, für den Fall, dass der Dateiname geändert worden sein sollte. Anschließend folgt der <TableObject> Knoten, innerhalb dessen die Marker gespeichert werden. Jeder Marker wird nach dem folgenden Schema gespeichert, welches den Aufbau der *Unity-GameObjects* 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>
```

Dies versteht sich als Kurzschreibweise. Innerhalb der Positions-, Rotations- und Skalierungsknoten ist jeweils der gespeicherte 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. In dieses wird zunächst sowohl der äußerste Knoten als auch der Zeitstempel eingefügt. 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.

Hat diese Funktion durch alle Kindknoten bearbeitet, wird zuletzt noch die Skalierung der Szene in dem Knoten `globalBuildingScale` gespeichert, damit der globale Maßstab, welcher im `SetScale`-Menü (vgl. Abschnitt 4.5.1) eingestellt wird, beim Laden der Szene rekonstruiert werden kann.

Darstellen von gespeicherten Szenen Die gespeicherten Szenen werden in Listenform im Tischmenü dargestellt. Zuoberst ist immer die aktuellen (jüngste) Szene. Über die Schaltflächen „1-6“, „7-12“, „13-18“ und „19-24“ kann „umgeblättert“ werden und so können weitere, in der Vergangenheit liegende Szenen zugegriffen werden. Für diese Darstellung ist das Skript `Timeline.cs` zuständig. Bei jedem Speichern wird die Darstellung aktualisiert, so dass immer alle gespeicherten Szenen in der Darstellung berücksichtigt werden.

4.7.3 Laden von Szenen

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 unter Umständen noch Marker, die im Arbeitsbereich liegen und aktuell verfolgt 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 Netzwerk-Datenstrom der Tracking-Anwendung überschrieben werden würden.

Vorgehen beim Laden Durch Auswählen einer XML-Datei im Tischmenü 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 Tischmenü hinzu. Anschließend wird die Datei eingelesen, die internen Knoten liegen dann in einer Form einer `XmlNodeList` bereit.

Darauffolgend 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 Speichern sichtbar waren.

Alle aktiven Marker werden in der `ArrayList activeMarkerIDs` gespeichert. Diese `ArrayList` wird, nachdem sie vollständig gefüllt ist, bearbeitet: Für jeden Eintrag, also jeden aktiven Marker wird ein Platzhalter Marker instantiiert. Dies ist ein Marker, der bereits alle Komponenten, also zugeordnete `GameObjects` in *Unity* wie Kontextmenü etc., die benötigt werden, angeheftet hat. Es müssen dann nur noch gespeicherten Parameter auf diesen Marker übertragen werden. Jeder Marker bekommt anschließend `Marker + (Original-ID + 100)` als Namen zugewiesen. Dies ist wichtig um später die aus der XML-Datei gelesenen Marker von den aktuell über TCP gesteuerten Marker am Namen unterscheiden zu können.

Nachdem die weiteren Parameter wie Position, Rotation und Skalierung an den Marker übergeben wurden, wird dann der Marker an das `TableObject` in *Unity* als `Kind-GameObject` angehängt 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 größer 100 haben, und die aktiven Marker die durch die laufend Tracking-Anwendung verfolgt werden.

4.7.4 Match-Modus

Der Match-Modus ist ein Systemzustand, der nach dem Laden von gespeicherten Szenen aktiviert wird. Dieser Systemzustand ist in Abbildung 17 zu sehen.

Es werden wie in Abschnitt 4.7.3 beschrieben die geladenen Marker und die laufend von der Tracking-Anwendung verfolgten 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 der virtuellen Marker aktiviert. Die markerinterne Variable `matchMode` ist zu diesem Zeitpunkt auf `false` gesetzt.

Schiebt der Benutzer nun einen beliebigen Marker an die Position des virtuellen Markers, werden bei ausreichend genauer Positionierung alle vier Collider des geladenen virtuellen Markers mit den vier Collidern des vom Benutzer bewegten virtuellen Markers getriggert. Ist dies der Fall, wird die markerinterne Variable `matchModeReady` auf `true` gesetzt. Dieses Verhalten wird von dem Skript `StopMatchMode.cs` kontrolliert.

Zeigen alle geladenen Marker durch grünes Aufleuchten, dass sie bereit sind, erscheint im Tischmenü die Schaltfläche `Apply`. Drückt der Nutzer diese, 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. Nach Beenden des *Match-Modus* existieren wieder nur die durch die Tracking-Anwendung laufend verfolgten Marker, die jetzt allerdings die Parameter aus der geladenen Datei übernommen haben und somit in der Gesamtheit die geladene Szene repräsentieren.

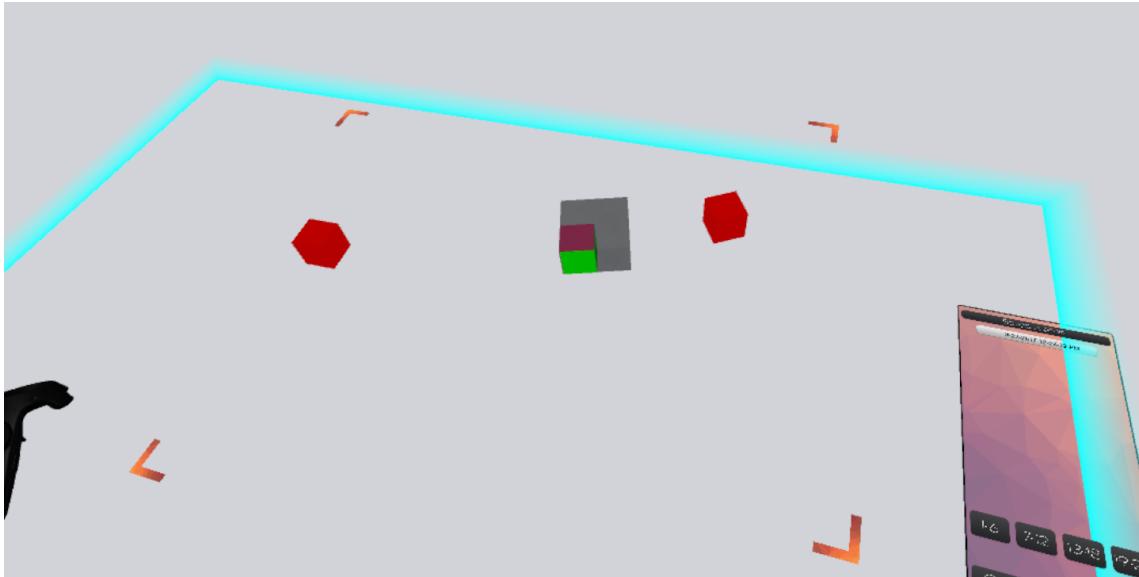


Abbildung 17: Der Match-Modus von *MARc*.

5 Tracking-Anwendung

Für das Tracking und die Positionsbestimmung der Würfel-Marker (vgl. Abschnitt 3.1.6) wurde ein Programm entwickelt, welches alle erforderlichen Ressourcen und Funktionen bereit stellt und verknüpft. Zunächst muss die Initialisierung und der Zugriff auf die *uEye*-Kamera (vgl. Abschnitt 3.1.4) ermöglicht werden. Im Anschluss ist das Erstellen beziehungsweise das Laden aller relevanten Kalibrierungsinformationen (vgl. Abschnitt 5.2) für einen korrekten Tracking-Prozess erforderlich. Nach erfolgreichem Abschluss beider Schritte wird das Tracking der *ArUco*-Marker und der grünen Rechtecke ausgeführt. Dies wird in Abschnitt 5.3 beschrieben. Die sich anschließende Verwaltung der registrierten und detektierten Würfel-Marker kann in Abschnitt 5.5 nachvollzogen werden. Bei Projektbeginn war es ursprünglich geplant ein *markerless Tracking* zu implementieren. Doch auf Grund der unvermeidlichen längerfristigen Verdeckungen von Würfel-Marker und deren texturarmen Oberfläche musste darauf verzichtet werden. Alle Schritte die der Kalibrierung folgen werden in einer Endlosschleife für jeden Frame ausgeführt. Während einer Iteration wird auf das aktuelle Live Bild zugegriffen, alle Marker detektiert und verfolgt, sowie alle Resultate übertragen. Die Übertragung wurde wie in Abschnitt 2.4 beschrieben per TCP realisiert und es gilt alle relevanten Daten der Marker Objekte (vgl. Abschnitt 5.4) an den *Unity*-Computer zu übermitteln. Die Tracking-Anwendung berechnet derzeit 21.7 ± 1.6 Iterationen pro Sekunde. Diese Messung wurde mit 500 Stichproben bzw. Iterationen durchgeführt, während sechs Würfel-Marker vom System getrackt und die Daten an den *Unity*-Computer per TCP übertragen wurden.

Dieses Programm wurde in der IDE *Visual Studio* (vgl. Abschnitt 3.3.2) in der Programmiersprache *C++* entwickelt. Für die Entwicklungen wurden die *uEye*-SDK (vgl. Abschnitt 3.1.4), *OpenCV* (vgl. Abschnitt 3.3.3), sowie die Standardbibliothek *Winsock* zur Hilfe genommen. In den folgenden Abschnitten wird die Vorgehensweise und der Ablauf schrittweise und detailliert erklärt.

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.

5.1 uEye Ansteuerung

Der erste Teilprozess der Tracking-Anwendung ist die Initialisierung der *uEye* Kamera im Live-Bild-Modus. Mit Hilfe des vom Hersteller bereit gestellten SDK wird die Kamera mit allen benötigten Eigenschaften parametrisiert und die notwendigen Speicher allokiert. Dies übernimmt die Funktion `inituEyeCam()` in der Klasse `uEye_input.cpp`. Die verwendeten Parameter können aus Tabelle 6 entnommen werden. Die Kamera wird mit dem maximalen Pixel Clock und der maximalen Framerate betrieben. Der Pixel Clock gibt den Takt an mit dem die Pixel vom Sensor ausgelesen werden und demzufolge kann er die maximale Framerate begrenzen. Aus diesen Einstellungen ergibt sich auch die Belichtungszeit. Hier kann noch hinzugefügt werden das eine hohe Framerate ein flüssiges Tracking ermöglicht. Um einen höheren Kontrast zur Erkennung der Marker zu erzielen wurde zusätzlich das Hardware Signal zwanzigfach verstärkt. Die letzten beiden Parameter wurden vom Entwickler mit visueller Begutachtung unter den Lichtverhältnissen im Projektraum ermittelt und müssen ggf. unter wechselnden Bedingungen angepasst werden. Im Live-Modus werden die generierten Aufnahmen fortlaufend in der selben Speicheradresse überschrieben. Auf diesen Speicherplatz kann das Programm jederzeit mit der Funktion `getCapturedFrame()` zugreifen, welcher die aktuellste Aufnahme zur Weiterverarbeitung zurück gibt. Nach der Beendigung des Tracking-Ablaufs wird zusätzlich noch die Funktion `exitCamera()` bereit gestellt, welche den verwendeten allokierten Speicher wieder freigibt.

5.2 Kalibrierung

Um das Ziel von *MarC* zu erreichen muss das System kalibriert werden, um an den exakten Positionen der Aluminiumwürfel virtuelle Würfelobjekte mit *Unity* zu rendern. Die Kalibrierung hat zum Ziel, eine Koordinatentransformation zu finden, die Positionen im Kamerakoordinatensystem in das Unitykoordinatensystem 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. Abschnitt 5.2.1) – also in zwei verschiedenen Koordinatensystemen bekannte Punkte – eine affine 3D-Transformation, welche die Abbildung vom Kamera-Koordinatensystem auf das Unitykoordinatensystem 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. Abschnitt 5.2.7), 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 Abschnitt 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 Abschnitten 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. Abschnitt 3.3.3) benutzt worden sind. Alle Grundlegenden Definitionen für die Rechenschritte können in Abschnitt 5.2.2 nachgelesen werden. Die Schnittstelle in Form der `Calibration.cpp`-Klasse wird in Abschnitt 5.2.6 eingeführt.

Trotz einer sorgfältigen Umsetzung der in den nächsten beiden Abschnitten 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 Abschnitt 5.2.5 beschrieben.

5.2.1 Korrespondierende Punktpaare

Im Zusammenhang mit der Kalibrierung des Systems von *MArC* werden von einem Marker Koordinaten in Bildkoordinaten und die entsprechenden Koordinaten in der *Unity*-Welt benötigt. Hierbei spricht man von korrespondierenden Punkten. Genauer gesagt wird für diese Zwecke der Kalibrierungscontroller (vgl. Abschnitt 3.1.8) verwendet. Auf diesem befindet sich ein *ArUco*-Marker (vgl. Abschnitt 3.3.4). Dessen Mittelpunkt, also das `estimatedCenter`, kann für jede Position über das in Abschnitt 5.3.2 beschriebene Trackingverfahren ermittelt werden und wird für jeden erkannten Marker auf Trackingseite abgespeichert. Dies geschieht bei der Kalibrierung immer, wenn der Trigger am Controller der *HTC Vive* ausgelöst wird. Somit kann in *Unity* zeitgleich die entsprechende Controller-Position abgespeichert werden. Eine Position liegt also sowohl in Bild- als auch in *Unity*-Koordinaten vor (vgl. Tabelle 7).

5.2.2 Definition der Koordinatensysteme

In Abbildung 18 sind die Zusammenhänge zwischen dem Projektionszentrumkoordinatensystem der Kamera, sowie deren Bildebene und dem Weltkoordinatensystem zu sehen. Im Gegensatz zu der Abbildung und den in Abschnitt 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 *Unity*-Koordinaten genannt.

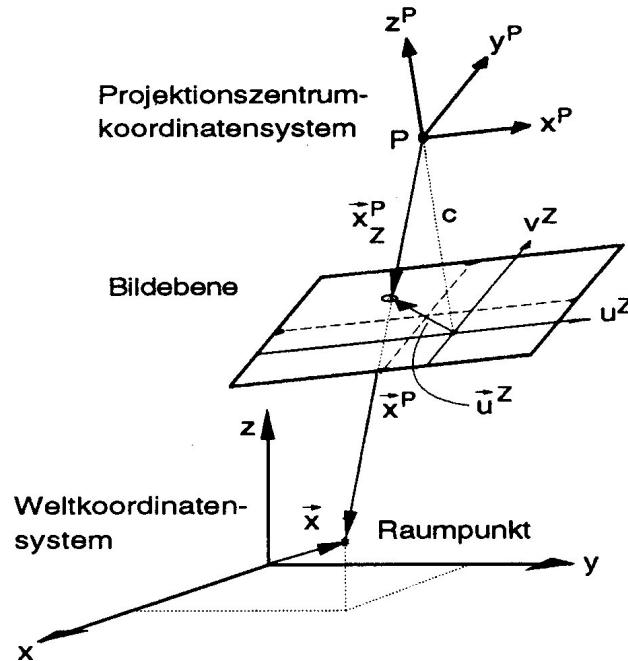


Abbildung 18: Lage des Kamerakoordinatensystems in Bezug auf Projektionsebene und Weltkoordinatensystem. [43]

Um die verschiedenen Koordinatensysteme bei den Ausführungen in den Abschnitten 5.2.3 und 5.2.4 auseinander halten zu können, wird zu Beginn eine Notation festgelegt, 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 hinzu *Unity*-Koordinaten, 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 *Unity*-Koordinaten ergänzt werden.

5.2.3 Kamerakalibrierung

Es gibt viele verschiedene wissenschaftliche Ausführungen über die Durchführung einer Kamerakalibrierung, wie z.B. [57], [74] und [19]. Dabei unterscheidet man häufig zwischen automatischen und manuellen Kalibrierungen. Im Folgenden wird der Algorithmus zur Kamerakalibrierung von *MArC* erläutert.

	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			Bildeckenverkippung
\vec{x}^V	Verkippung	$u^V, v^V, -c^V$	
\downarrow			Bildhauptpunktverschiebung
\vec{u}	Sensor	u, v	

Tabelle 7: Parameter und Berechnungsschritte der Kamerakalibrierung [43].

Im konkreten Fall soll die uEye-Kamera (vgl. Abschnitt 3.1.4), die für Tracking-Zwecke (vgl. Abschnitt 5) an der Decke des Entwicklungsraums befestigt ist, kalibriert werden. Dies kann zu Beginn der eigentlichen Anwendung durchgeführt werden. Dazu wird ein eigens für dieses Projekt angefertigter Schachbrett-Kalibrierungshelfer verwendet, der in Abschnitt 3.1.7 beschrieben wird. Eine Beschreibung der genauen Durchführung der Kamerakalibrierung kann der ReadMe-Datei (vgl. 9.2) 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, wie aus den Parametern eine geeignete Transformation von Kamerakoordinaten in Weltkoordinaten gebildet 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 Bildecke 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:

$$R_v = \begin{pmatrix} r_{v11} & r_{v12} & r_{v13} \\ r_{v21} & r_{v22} & r_{v23} \\ r_{v31} & r_{v32} & r_{v33} \end{pmatrix} = \begin{pmatrix} \cos \vartheta & \sin \vartheta \sin \varphi & -\sin \vartheta \cos \varphi \\ 0 & \cos \varphi & \sin \varphi \\ \sin \vartheta & -\cos \vartheta \sin \varphi & \cos \vartheta \cos \varphi \end{pmatrix} \quad R_v \in \mathbb{R}^{3x3} \quad (1)$$

Die negative verkippte Kamerakonstante $-c^V$, die in Tabelle 7 aufgeführt ist, berechnet sich wie in [43] beschrieben nach der Gleichung:

$$-c^V = -\frac{c + u^V \cdot r_{v13} + v^V \cdot r_{v23}}{r_{v33}} \quad (2)$$

Im zweiten Schritt wird im Allgemeinen 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 Bildbenenverkippung, lässt sich in der intrinsischen Kameramatrix $M_{intrinsisch}$ zusammenfassen und ausdrücken. 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.

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

Während eine dreidimensionale Rotation im Allgemeinen mit Matrix R aus Gleichung (4) beschrieben werden kann, reicht für die Translation ein Vektor, wie t aus Gleichung (5) aus.

$$R = R_\gamma \ R_\beta \ R_\alpha = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \quad R \in \mathbb{R}^{3x3} \quad (4)$$

$$t = (t_x \ t_y \ t_z)^T \quad t \in \mathbb{R}^{3x1} \quad (5)$$

Diese beiden Transformationen können wie in Gleichung (6) 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 $R \in SO(3)$.

$$M_{extrinsisch} = (R \ | \ t) \quad M_{extrinsisch} \in \mathbb{R}^{3x4} \quad (6)$$

Schritt 1 und 2 lassen sich vereinfachen, indem man die intrinsische Kameramatrix $M_{intrinsisch}$ und die extrinsische Kameramatrix $M_{extrinisch}$ nach der Gleichung (7) zu einer Matrix M zusammenfasst.

$$M = M_{intrinsisch} \cdot M_{extrinisch} \quad M \in \mathbb{R}^{3x4} \quad (7)$$

Die Umrechnung von Kamerakoordinaten in Weltkoordinaten kann dann mit der invertierten Matrix M , wie in Gleichung (8) berechnet werden.

$$\vec{x} = M^{-1} \cdot \vec{u} \quad (8)$$

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

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 (vgl. Abschnitt 5.2.7). Die Durchführung der Kalibrierung wird in der ReadMe-Datei (vgl. 9.2) erläutert. Grob gesagt, werden per Knopfdruck am Kalibrierungscontroller (vgl. Abbildung 5) 25 seiner Positionen sowohl in Kamerakoordinaten, als auch in *Unity*-Koordinaten 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 *Unity*-Koordinaten fungiert. Beide Teilbereiche werden in den Absätzen „Begrenzung des Arbeitsbereichs“ bzw. „Affine Transformation“ weitergehend erläutert.

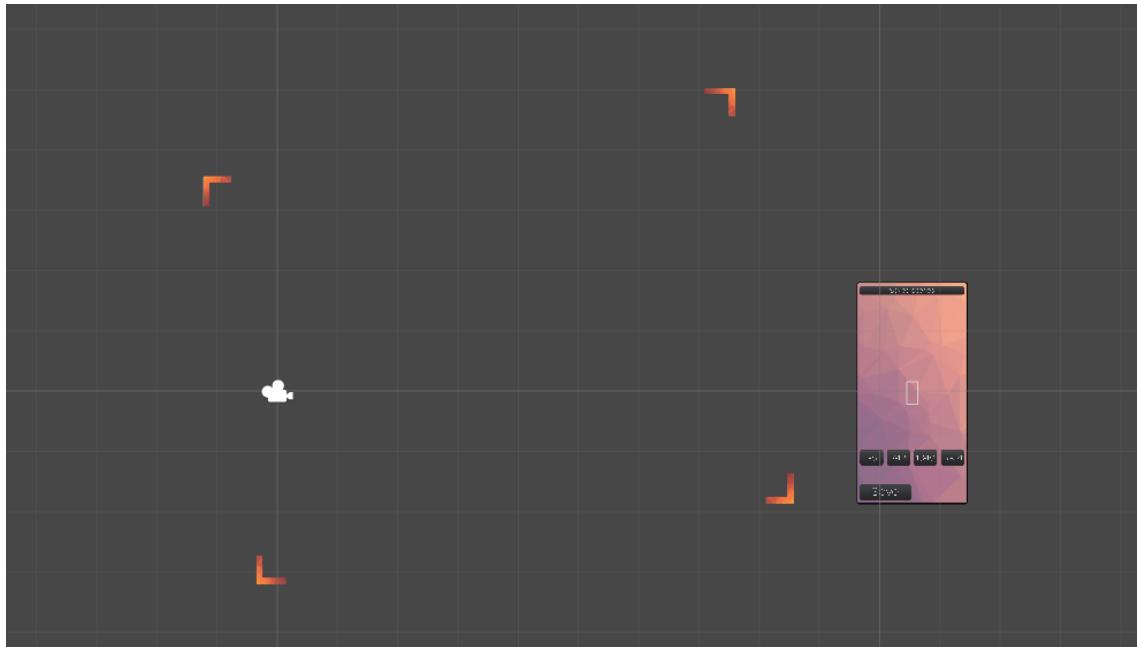


Abbildung 19: Begrenzung des Arbeitsbereichs in der *Unity*-Simulation.

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-Anwendung und *Unity*-Simulation) unterschiedlich vorgegangen, was zu leicht abweichenden Arbeitsbereichen führen kann. Es wird ausdrücklich empfohlen sich dabei an den Tischkanten zu orientiert und beispielsweise die Kante der Auflage des Kalibrierungscontrollers (vgl. Abschnitt 3.1.8) daran ausrichtet und somit eine ungefähr rechteckförmige Fläche aufzuspannen.

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 Arbeitsbereich 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. Abschnitt 5.3). In Abbildung 21 ist der so aufgespannte Arbeitsbereich rot dargestellt.

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 19 dargestellt, begrenzt wird. Zunächst war der Arbeitsbereich in der *Unity*-Simulation ebenfalls nur durch zwei Punkte definiert, sodass der Bereich immer 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 abgemeldet worden waren, oder umgekehrt. Um dieses Abbil-

dungsproblem 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 gesamten 25 Controller-Positionen werden genutzt, um eine affine Transformation von Weltkoordinaten \vec{x} in *Unity*-Koordinaten \vec{x}^U zu finden. An dieser Stelle kann davon ausgegangen werden, dass die Mittelpunkte der *ArUco* Marker über die in Abschnitt 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. Abschnitt 5.2.1) verstanden. Mit anderen Worten, soll in diesem Schritt des Algorithmus jeder dieser 25 Punkte in Weltkoordinaten, möglichst fehlerfrei auf sein entsprechendes Pendant in *Unity*-Koordinaten projiziert werden.

Es gilt also die in Gleichung (9) 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 sechs Freiheitsgrade. Zur Abschätzung der bestmöglichen Matrix M_{affin} eignet sich vor allem der *RANSAC*-Algorithmus [21]. Dieser wird ebenso für die in diesem Projekt verwendete *OpenCV*-Methode `estimateAffine3D()` verwendet. Dieser Methode können alle 25 korrespondierenden Punktpaare übergeben werden.

$$M_{affin} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \quad M_{affin} \in \mathbb{R}^{3x3} \quad (9)$$

5.2.5 Kalibrierungsfehler

In den Abschnitten 5.2.3 und 5.2.4 wird der Algorithmus zur Berechnung der Koordinatentransformation von Kamerakoordinaten in *Unity*-Koordinaten beschrieben. Beim Benutzen des fertig kalibrierten System fällt jedoch auf, dass die Aluminiumwürfel nicht immer deckungsgleich mit ihrem gerenderten Pendant sind. Der beobachtete Fehler variiert bei jeder Kalibrierung. In Abbildung 20 wird ein Kalibrierungsfehler in vertretbarer Größenordnung gezeigt. Dieser ist weiß markiert. Im Idealfall würden die beiden weißen Kreise deckungsgleich sein. Wird ein deutlich größerer Fehler beobachtet, so ist es sinnvoll neu zu kalibrieren. Im Folgenden werden mögliche Gründe für die Positionsabweichung von dem Würfel-Marker und seinem virtuellen Marker diskutiert.

Neben der Rechenengenauigkeit und Rundungsfehlern, die sich über die diversen Rechenschritte (vgl. Abschnitt 5.2.3 und 5.2.4) hinweg summieren, sollen in diesem Kapitel systembedingte mögliche Fehlerquellen aufgeführt werden. Man kann davon ausgehen, dass sich der Fehler großteils durch Fehlerfortpflanzung [61] ergibt.

Aufgrund der Auflösung der für das Tracking genutzten *IDS uEye*-Kamera (vgl. Abschnitt 3.1.4) ist die Genauigkeit des Trackings begrenzt und die Positionen der



Abbildung 20: Mit weiß markierter möglicher Kalibrierungsfehler.

Würfel-Marker können nicht beliebig genau ermittelt werden. Dieser Effekt wird durch Einflüsse wie z.B. Bildrauschen weiter verstärkt. Es ist also davon auszugehen, dass die Mittelpunkte der verfolgten Marker, die als ein Part in die korrespondierenden Punktpaare (vgl. Abschnitt 5.2.1) eingehen, schon einen gewissen Fehler aufweisen.

Da mittels des *RANSAC*-Algorithmus [21] lediglich eine Schätzung ermittelt wird, ist die affine Transformationsmatrix M_{affin} nicht gänzlich fehlerfrei.

Zusätzliche Fehler bei der Registrierung 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.

5.2.6 Schnittstelle der Kalibrierungen

Wie bereits erwähnt, dient die `Calibration.cpp`-Klasse als Schnittstelle für die Kamerakalibrierung (vgl. Abschnitt 5.2.3) und die Arbeitsbereichskalibrierung (vgl. Abschnitt 5.2.4). Diese Schnittstelle stellt alle erforderlichen Funktionen für beide Kalibrierungen 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 (3)) 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()` der Klasse `PoseEstimation.cpp` für die spätere Verwendung gespeichert.

Die Klasse `calibWithChessboard.cpp` implementiert den modifizierten Beispielcode `calibration.cpp` [50] aus dem *GitHub* Repository der *OpenCV* Bibliothek. Die Modifikation ermöglicht die Kompatibilität mit dem restlichen Programm. Vor dem Speichern und der Weiterverbreitung werden die `distCoeffs` auf null gesetzt, da die Entzerrung des Bildes nicht zuverlässig ist. Die explizite Bestimmung der intrinsischen und extrinsischen Matrizen findet in der *OpenCV*-Methode `calibrateCamera()` auf Grundlage von mehreren Aufnahmen eines fest definierten Kalibrierungsmusters (vgl. Abschnitt 3.1.7) statt.

Wie in Abschnitt 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 *Unity*-Koordinaten 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. Abschnitt 5.2.7).

Die affine Transformation wird auf Grundlage der korrespondierenden Punktpaare (vgl. Abschnitt 5.2.1) berechnet. Hierzu wird in der Methode `computeAffineTransformation()` die Funktion `estimateAffine3D()` der *OpenCV* Bibliothek aufgerufen, welche die affine Transformationsmatrix mit Hilfe des RANSAC-Algorithmus abschätzt [21].

5.2.7 Speichern und Laden der Kalibrierungen

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 drei Möglichkeiten:

1. **Beide Kalibrierungen durchführen:** In diesem Fall wird zunächst eine Kamerakalibrierung durchgeführt, wie sie in Abschnitt 5.2.3 beschrieben ist und anschließend wird eine Kalibrierung des Arbeitsbereiches (vgl. Abschnitt 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 Abschnitt 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

Nach Abschluss der erfolgreichen Kalibrierung bzw. des Ladens einer gespeicherten Kalibrierung (vgl. Abschnitt 5.2.7) beginnt das Tracking-Anwendung, sowohl damit die *ArUco*-Marker, als auch die leuchtend grünen Rechtecke zu detektieren. Die codebasierten *ArUco*-Marker ermöglichen es den Marker nach längerer Verdeckung wieder eindeutig seiner entsprechenden Registrierungs-ID (RID) zu zuordnen. Somit wird Fehlzuordnungen oder sprunghaften Vertauschungen von benachbarten Würfel-Markern vorgebeugt. Alle Methoden zur Detektierung der beiden Markertypen werden von der Klasse `MarkerDetection.cpp` zur Verfügung gestellt und mit der Funktion `runMarkerDetection()` aufgerufen.

Die detektierten Marker werden nur verfolgt wenn sie innerhalb eines festgelegten Arbeitsbereichs positioniert sind. Zur Registrierung eines Marker-Objektes (vgl. Abschnitt 5.4) muss der Würfel-Marker in den anzeigte Arbeitsbereich geschoben werden und zur Abmeldung wieder heraus. Demzufolge werden Würfel-Marker deren *ArUco*-ID (AID) noch unbekannt ist registriert und Würfel-Marker, welche den Arbeitsbereich verlassen, zurückgesetzt und die RID wieder freigeben. Der Arbeitsbereich wird während der Kalibrierung des Arbeitsbereichs (vgl. Abschnitt 5.2.4) als Rechteck definiert. Hier wird der erste detektierte Mittelpunkt des *ArUco*-Markers als unterer linker Eckpunkt des Rechtecks übernommen und der Dritte legt den oberen rechten Punkt fest. Mit diesen beiden Punkten kann der Arbeitsbereich aufgespannt werden.

Zunächst werden während der Marker-Detektion die grünen Rechtecke mit Hilfe eines simplen *Greenkeying* erfasst und im Anschluss die erforderlichen *ArUco*-Marker der Würfel-Marker. Letztere können bekannterweise unter Umständen nur im ruhenden Status erfasst werden, da eine mögliche Bewegungsunschärfe bei schnellen Verschiebungen (vgl. Abschnitt 2.6) eine korrekte Erfassung nahezu unmöglich macht. Aus diesem Grund werden neben den codebasierten Muster auch die leuchtend grünen Rechtecke verfolgt um einen Würfel-Marker auch bei schnelleren Bewegungen zu verfolgen.

5.3.1 Detektion der grünen Rechtecke

Das Ziel dieses Verfahren ist es eine *Oriented Bounding Box* (OBB) von jedem erkannten grünen Rechteck zu erstellen. Die OBB beinhalten Informationen über die Größe, den Mittelpunkt, 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. Bei der Detektion der grünen Rechtecke müssen diese 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 müssten, je nach Beleuchtungsverhältnisse, die Schwellwerte einer Binarisierung geeignet 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 Teilbereiche einer bestimmten Farbe zu segmentieren. Aus diesem Grund werden alle Live-Bilder vor der Binarisierung in den HSI-Farbraum transformiert.

Die Binarisierung jedes Pixel $x \in (\mathbb{R}^3 | 0 \geq x_H \leq 255, 0 \geq x_S \leq 255, 0 \geq x_I \leq 255)$ erfolgt unter den Bedingungen aus Gleichung (10), welche alle nicht grünen Bildsektionen schwarz maskiert und nur die grünen Bildbereiche weiß. 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-Weiß-Bild verringert. Das verbleibende Bildrauschen wird durch einen Median-Rangordnungsfilter mit einer Kernelgröße von 5×5 eliminiert. Sowohl die Binarisierung als auch der Rangordnungsfilter können mit Methoden aus der *OpenCV* Bibliothek umgesetzt werden.

$$I(x) = \begin{cases} 255 & \text{wenn } 65 \geq x_H \leq 85, 120 \geq x_S \leq 255, 22 \geq x_I \leq 255 \\ 0 & \text{sonst} \end{cases} \quad (10)$$

Im binarisierten Bild werden im Anschluss zur Detektierung der OBB die Rechteckkanten mit einem Canny Kanten Detektor extrahiert [8]. Im daraus resultierenden Kantenbild werden mit Hilfe eines Kantenverfolgungsalgorithmus [64] die Konturen generiert. Dieser Algorithmus erstellt eine hierarchische Sammlung von allen Punkten der äußersten Konturen eines Objektes. Aus dieser Sammlung wird die OBB mit dem *Douglas Ramer Peucker* Algorithmus [17] approximiert. Während der Approximation werden die Konturen geglättet und somit an ein ideales Rechteck angenähert. In Abbildung 21 ist sind die generierten OBB weiß dargestellt und es ist deutlich zusehen, dass die Annäherung sehr genau ist.



Abbildung 21: Screenshot der Tracking-Anwendung. Die weißen Konturen sind die detektierten grünen Rechtecke. Alle Beschriftungen der Marker zeigen die resultierenden Winkel an und der äußere rote Rahmen stellt den kalibrierten Arbeitsbereich dar.

5.3.2 ArUco Marker-Detektion

In der Klasse `MarkerDetection.cpp` werden in der Funktion `detectArucoMarker()` alle *ArUco* Marker detektiert [23], die sich im Bild befinden. Die gesamte Detektion wird durch die Methode `detectMarkers()` der *ArUco*-Bibliothek aufgerufen und als Ergebnis bekommt jeweils die vier Eckpunkte der erkannten Marker Muster. In dieser Methode werden die drei benötigten Schritte zur Detektion der Marker implementiert. Zunächst wird das Graustufenbild mit adaptiven lokalen Schwellwerten binarisiert und anschließend die Kanten des binären Bildes mit einem Canny Filter [8] herausgefiltert. Von diesen Objektkanten werden, wie bei den OBB (vgl. Abschnitt 5.3.1), die Konturen extrahiert und vervollständigt.

Der letzte Schritt sieht sowohl die Positionsabschätzung als auch die Verfeinerung der erkannten Markerpunkte vor. Zur Verbesserung der detektierten Punkte kann eine Verfeinerung der Positionen mit einer linearen Regression erreicht werden. Anschließend kann mit den generierten Konturpunkten die AID ermittelt werden, indem die Werte mit den Daten aus dem ausgewählten Dictionary abgeglichen wird.

Die Positionsabschätzung dient dazu die Markerposition mit Hilfe des Levenberg-Marquardt-Algorithmus [42] im Kameraraum zu bestimmen. Für diesen Prozess stellt die *ArUco* Bibliothek die Methode `estimatePoseSingleMarkers()` zur Verfügung, welche für eine erfolgreiche Schätzung mehrere Parameter verlangt. Zum einen werden die vier äußeren Eckpunkte des Codemusters benötigt und zum anderen die reale Kantenlänge eines Markers von 0.04 m. Weiterführend, werden auch die intrinsische und extrinschen Kameramatrizen benötigt, welche während der letzten Kamerakalibrierung (vgl. Abschnitt 5.2.3) erstellt wurden.

5.4 Definition des Marker-Objekts

Die Marker Objekte der Klasse `Marker.cpp` speichern alle benötigten Parameter und Informationen der registrierten Würfel-Marker. Zu diesen Daten gehören unter anderem die RID (`id`), die aktuelle 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 der Tracking-Anwendung verfolgt wurde.

Bei der Winkelbestimmung gibt es zwei Probleme, die es erforderlich machen die Winkel der `RotatedRects` zunächst umzurechnen. Für den Winkel ω_{RR} gilt $\omega_{RR} \in \{\mathbb{R} | -90^\circ \leq \omega_{RR} \leq 90^\circ\}$ und nicht wie für alle folgenden Komponenten des Gesamt-systems gefordert $\omega_{WM} \in \{\mathbb{R}^+ | 0^\circ \leq \omega_{WM} \leq 360^\circ\}$.

Dazu kommt noch, dass die Eckpunkte eines jeden `RotatedRect` Objektes in beliebiger Reihenfolge gespeichert werden. Jedoch sollen der Winkel ω_{RR} zuverlässig zwischen dem unteren linken Eckpunkt des Markers und einem vom Mittelpunkt des OBB ausgehenden skalierten Einheitsvektor berechnet werden. Aus diesem Grund muss gewährleistet werden, dass zuverlässig und mit geringem Kostenaufwand auf den richtigen Eckpunkt zugriffen wird. An dieser Stelle wird auf die Eckpunkte der detektierten *ArUco*-Marker zurückgegriffen, da die *ArUco*-Methode `detectMarkers()` stets die Eckpunkte im Uhrzeigersinn und beginnend mit der unteren linken Ecke des Mustercodes zurückgibt.

Um einheitliche Winkel zu garantieren ist es demzufolge nicht ausreichend den Definitionsbereich von ω_{RR} an ω_{WM} anzupassen. Deshalb wird in der Funktion `computeAngle()`, wie in Gleichung (11), der Winkel zwischen \vec{u} und \vec{o} berechnet. Der Vektor \vec{o} berechnet sich wie in Gleichung (13), wobei p_{ul} die linke untere Ecke und c_{RR} der Mittelpunkt des OBB ist. Die Konstante c kann als Skalierungsfaktor bei der Berechnung des Vektors \vec{u} verstanden werden (vgl. Gleichung (12)). Mit anderen Worten handelt es sich bei dem Vektor \vec{u} also um einen um den Faktor c skalierten Einheitsvektor.

$$\omega_{WM} = \arccos \frac{\vec{u} \times \vec{o}}{\|\vec{u}\| * \|\vec{o}\|} \quad (11)$$

$$\vec{u} = \begin{pmatrix} c_{RR}.x - c \\ c_{RR}.y \end{pmatrix} - \begin{pmatrix} c_{RR}.x \\ c_{RR}.y \end{pmatrix} \quad (12)$$

$$\vec{o} = p_{ul} - c_{RR} \quad (13)$$

Anschließend muss der Winkel ω_{WM} der bislang in Rad ist, noch in Grad umgerechnet werden und zudem auf einen Definitionsbereich von $\omega_{WM} \in \{\mathbb{R}^+ | 0^\circ \leq \omega_{WM} \leq 360^\circ\}$ abgebildet werden.

Eine andere essentielle Komponente der Klasse sind die Transformationsvektoren v_t (`tvec`), welche die Transformation jedes Markers vom Objektraum in den Kammeraraum K angeben. Diese Vektoren werden für die korrekte Abschätzung der Marker Position in der *Unity*-Welt (vgl. Abschnitt 5.2.4) benötigt und aus den vier

äußeren Eckpunkten in der Methode `estimatePoseSingleMarkers()` aus der *ArUco*-Bibliothek angenähert.

5.5 Tracking und Registrierung der Marker

Das Tracking und die Registrierung der Marker besteht zum einen aus der Bestimmung und Abschätzung von allen OBBs und zum anderen aus der Zuordnung von den detektierten AID. Alle erforderlichen Berechnungen und der logische Ablauf zur Identifikation werden in der Funktion `trackMarker()` der Klasse `MarkerManagement.cpp` durchgeführt. Relevant bei diesem Prozess ist das Wissen, dass ein Würfel-Marker nur dann verfolgt werden kann, wenn eine entsprechende OBB $r_m \in \mathbb{R}^4$ erkannt wurde. Jede r_m besteht aus den vier Eckpunkten $p_e^i \in \mathbb{R}^2$ mit $i = 4$ und einem Mittelpunkt $p_c \in \mathbb{R}^2$.

Zunächst wird für jede detektierte OBB jeweils ein Bewegungsvektor $v_m \in \mathbb{R}^2$ zwischen dem aktuellen Mittelpunkt und allen bekannten Mittelpunkten p_c^m aus der letzten Iteration nach der Gleichung (14) berechnet. Zu einem späteren Zeitpunkt werden diese Bewegungsvektoren verwendet um Marker innerhalb eines festgelegten Bewegungsradius zu verfolgen.

$$v_m = p_c - p_c^m \text{ mit } m = \text{Anzahl der reg. Marker} \quad (14)$$

Nach Abschluss aller notwendigen Vorbereitungen beginnt der eigentliche Tracking-Prozess. Dieser Prozess nutzt vier Methoden zum Tracking und zwei Funktionen zum Ermitteln der zur AID gehörenden zugehörigen RID. Diese Funktionen werden in der Reihenfolge des Flussdiagramm aus Abbildung 22 aufgerufen. Im ersten Schritt wird mittels der Methode `hasArucoID()` überprüft, ob eine r_m des aktuellen Frames eine der detektierten AID enthält. Je nachdem ob einer OBB eine AID zugeordnet werden konnte oder nicht müssen unterschiedliche Abläufe durchlaufen werden.

Zunächst wird auf den Zweig des Flussdiagramms eingegangen, indem Würfel-Marker verfolgt werden für die eine AID existiert. Doch vor dem Tracking muss diese eindeutig mit der Methode `findMatchID()` einer RID zugeordnet werden. Dafür wird die ermittelte AID mit den AIDs aller registrierten Marker abgleichen. Falls keine zugehörige RID ermittelt werden kann und der Marker innerhalb des Arbeitsbereichs liegt, wird ein neues Marker-Objekt mit der nächsten verfügbaren RID registriert. Befindet sich der Marker jedoch außerhalb des Arbeitsbereichs, so wird der detektierte Marker verworfen, da er nicht weiter von Interesse ist.

Aus diesem Grund muss auch ein Marker mit bekannter RID geprüft werden, ob er sich im Arbeitsbereich befindet. Ist dies nicht der Fall wird das entsprechende Marker-Objekt abgemeldet und alle bekannten Informationen wieder mit dem Defaultwert initialisiert. Befindet sich der Marker jedoch im Arbeitsbereich wird zunächst überprüft, ob sich der Marker noch an der selben Position befindet wie im vorangegangen Frame. Ist dies der Fall wird der Winkel ω_{WM} aktualisiert und der durch das Bildrauschen und der zum Bildrand zunehmenden Unschärfe verursachte Jitter reduziert, um eine ruhige Darstellung in der VR-Welt zu gewährleisten. Diese Reduzierung wird mit Hilfe der Schwellwerte aus der Funktion `getAngle-`

`Threshold()` der Klasse `MarkerManagement.cpp` umgesetzt, welche vom Abstand von c_{RR} zum Bildrand abhängig sind. Liegt die Differenz des Winkels ω_{WM} und dem Winkel im vorangegangen Frame jedoch oberhalb der Schwelle, so wird ω_{WM} aktualisiert.

Ein anderes Vorgehen ist notwendig, wenn für die OBB keine AID ermittelt werden konnte. Wie in Abschnitt 5.3 beschrieben wurde, ist es erforderlich auch Würfel-Marker verfolgen zu können, wenn keine AID erkannt wurde. Hierfür muss zunächst auch geprüft werden ob sich ein Würfel-Marker innerhalb des Arbeitsbereiches befindet. Für den Fall, dass der Marker an einer nahezu unveränderter Position ist, wird nur p_c aktualisiert und der Winkel ω_{WM} nicht berücksichtigt, da zur eindeutigen Bestimmung der untere linke Eckpunkt p_{ul} des *ArUco*-Markers fehlt. Dieser Umstand sollte bei einem konstant positionierten Marker in der Regel vernachlässigbar sein. In einer weiteren Situation wirkt sich dieser Umstand deutlicher aus und zwar wenn der Marker verschoben wurden. In diesem Fall muss dieser Marker in einem festgelegten Bewegungsradius mit Hilfe der vorab berechneten Bewegungsvektoren v_m gesucht werden. Konnte eine RID gefunden werden wird auch hier die Position aktualisiert und wenn nicht wird dieser Marker als verdeckt markiert und in der *Unity*-Anwendung ausgeblendet.

Zuletzt gilt es noch Würfel-Marker zu zuordnen, die keine AID haben und außerhalb des Arbeitsbereichs liegen. An dieser Stelle muss der Marker dennoch verfolgt werden, um die RID festzustellen und das entsprechende Marker-Objekt zurück zusetzen. Alle registrierten Marker-Objekte die während der aktuellen Iteration nicht zugeordnet wurden werden als verdeckt interpretiert und somit `isVis` auf den Wert 0 gesetzt.

Im Folgenden wird auf die Implementierung der Funktionen zum Marker Tracking eingegangen. Diese werden von der Klasse `IdMapping.cpp` zur Verfügung gestellt und in der Methode `trackMarker()` aufgerufen.

- `isConstantMarker()` prüft ob die Position p_c eines Markers unverändert ist und gibt die ermittelte RID zurück. Indem die Länge $\|v_m\|$ aller Bewegungsvektoren berechnet wird und prüft ob $\|v_m\| \leq 3$ Pixel ist. Mit anderen Worten gilt ein Marker als unverändert, wenn sich p_c innerhalb einer 3×3 Pixelnachbarschaft bewegt. Diese Toleranz ist notwendig, da es durch das Bildrauschen zu geringfügigen Abweichungen kommen kann.
- `isTranslatedMarker()` hat die Aufgabe Marker zu finden, die bewegt wurden. Hier wird die Zuordnung auch innerhalb eines Bewegungsradius von 100 Pixeln mit Hilfe von $\|v_m\|$ abgeschätzt. Diese Art des Trackings kann unter Umständen zu einer falschen Zuordnung führen. Doch der große Vorteil ist der gering Kostenaufwand und mit Betrachtung der Umstände sinkt die Wahrscheinlichkeit einer Fehlinterpretation auf ein vertretbares Risiko. Bei einem dieser Umstände handelt es sich um die Annahme das ein Nutzer nur maximal zwei Marker zum gleichen Zeitpunkt bewegen wird, während alle weiteren unbewegten Marker vorab als unverändert eingestuft wurden und nicht mehr während der aktuellen Iteration berücksichtigt werden.

Ein anderer Fakt ist, dass durch die Verwendung der *ArUco*-Marker zu jedem Zeitpunkt ein Marker wieder eindeutig zu geordnet werden kann. Dies

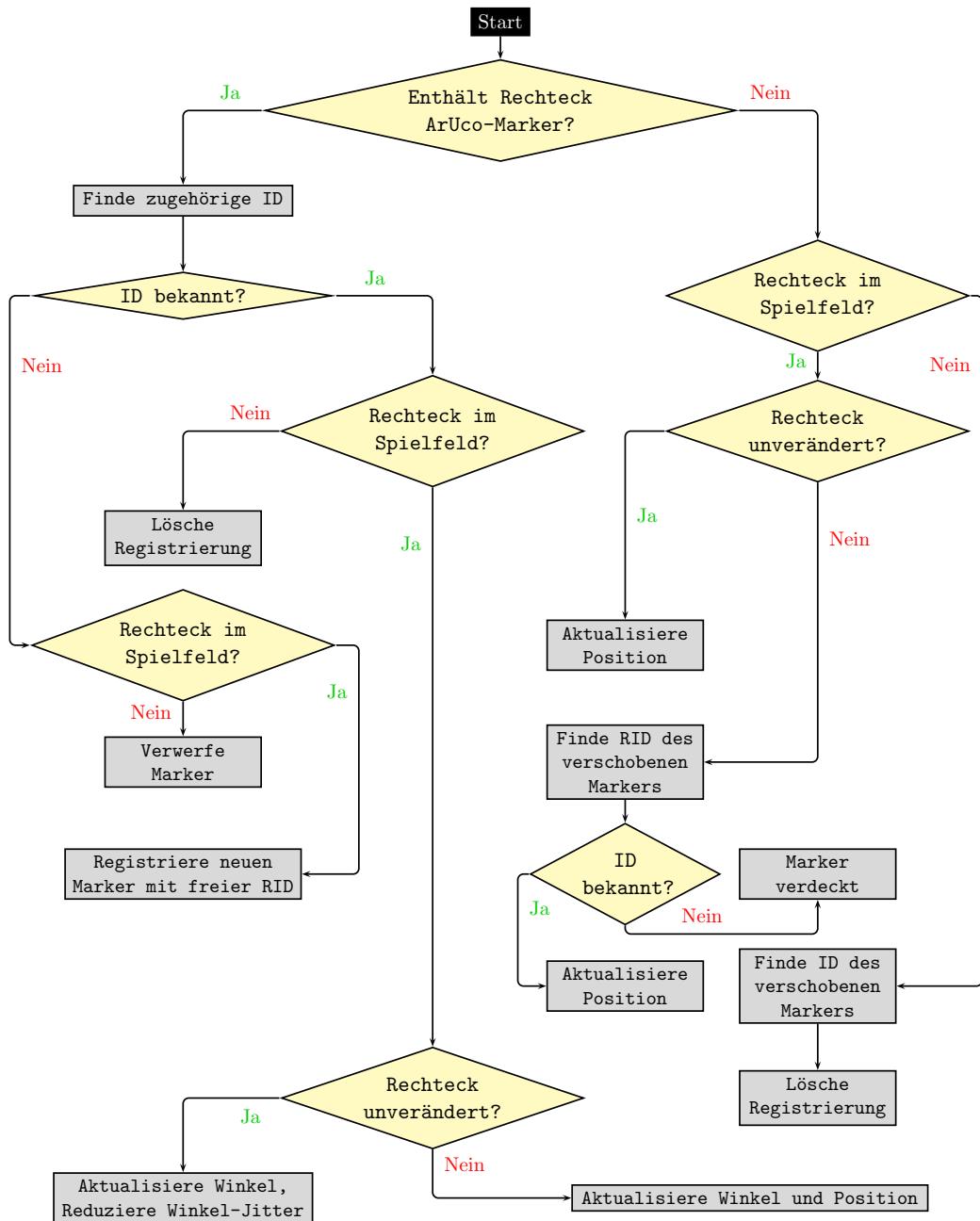


Abbildung 22: Flussdiagramm des Marker-Trackings.

ist unter anderem der Fall, wenn angenommen wird, dass der Markercode nur langsam in der näheren Umgebung anderer Würfel-Marker bewegt wird. Nach erfolgreichem Tracking wird die ermittelte RID zurückgeben.

- `isMarkerOutOfField()` und `isRectOutOfField()` testen ob sich das OBB noch innerhalb des rechteckigen Arbeitsbereichs befindet. Dieser Arbeitsbereich ist vom Typ `Rect` der *OpenCV* Bibliothek, welcher die Funktion `contains()` implementiert. Hier wird geprüft, ob ein übergebener Punkt innerhalb des bei der Kalibrierung aufgespannten Rechtecks liegt. `isMarkerOutOfField()` kann bei bekannter RID ein gesamtes Markerobjekt übergeben bekommen und `isRectOutOfField()` verlangt ein `RotatedRect` als Übergabeparameter.
- `hasArucoID()` teilt jeweils die vier Punkte des Vierecks vom *ArUco*-Marker in zwei Dreiecke auf und mit beiden wird ein Punkteinschlusstest mit Baryzentrischen Koordinaten durchgeführt [18]. Liegt der Mittelpunkt eines 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 der baryzentrischen Vorfaktoren gleich oder diese gleich Null sind liegt der Punkt im Dreieck. Der Vorteil dieses Tests ist, dass er mit wenigen kostenarmen Skalarprodukten und einfachen arithmetischen Berechnungen durchführbar ist und somit die Performanz der Tracking-Anwendung wenig beeinflusst.

Im Anschluss eines erfolgreichen Marker Trackings müssen alle p_c der registrierten Marker vom zweidimensionalen Bildraum in die dreidimensionale *Unity*-Welt transformiert werden. Die erste Transformation schätzt die Position von p_c im dreidimensionalen Kameraraum ab. Dieser `tvec` wird mit der Funktion `estimatePoseSingleMarkers()` des *OpenCV* Moduls `aruco` abgeschätzt. Das Ergebnis dieser Schätzung ist der Transformationsvektor v_t , welcher anschließend mit der affinen Transformationsmatrix M_{affin} in die Unity-Welt verschoben wird (vgl. Abschnitt 5.2.4). Der letzte Schritt wird in der Klasse `CoordsTransformation2Unity.cpp` mit der Methode `computeTransformation2Unity()` durchgeführt indem v_t mit M_{affin} multipliziert wird. Zu diesem Zeitpunkt werden die relevanten Daten der Marker Objekte per TCP an den *Unity*-Computer übertragen. Die Übertragung wird in der Klasse `TCP.cpp` aufgebaut und gesteuert. Nach erfolgreicher Übertragung beginnt eine neue Iteration (vgl. Abschnitt 4.3.1).

6 Ausblick

Lukas

Während dem Projektverlauf von *MArC* konnten einige Ideen für die Umsetzung der Anwendung reifen, welche aus Zeitgründen im vorliegenden Projekt nicht realisiert werden konnten. Dabei handelt es sich sowohl um Ideen, welche ursprünglich in der Projektplanung als Bestandteile von *MArC* vorgesehen waren, als auch um solche, die während der Arbeit am Projekt entstanden. Die folgenden Abschnitte beschäftigen sich mit den vielversprechendsten dieser Ideen. Diese können besonders dann, wenn nach dem Projektabschluss in einem anderen Rahmen noch weiter am Projekt gearbeitet werden sollte, dem nachfolgenden Team als Anhaltspunkte dienen.

6.1 Trackingmethoden

Vera

Laura

In der Zukunft kann die Tracking Software noch weiter verbessert werden. Zum einen kann geprüft werden, ob es möglich ist die OBBs der grünen Rechtecke auch mit einer Bewegungsabschätzung, wie zum Beispiel dem Kalman Filter [58] oder dem $\alpha/\beta/\gamma$ Filter [14], zu verfolgen. Doch an dieser Stelle gilt es auch zu prüfen, ob dann die Performanz der Tracking-Anwendung noch ausreichend für das Tracking mit der gewünschten Bildrate ist. Zur Verbesserung der gesamten Performanz kann auch die Möglichkeit geprüft werden die Implementierung der Verfolgung und Detektion zu parallelisieren. Gelingt dies, so kann unter Umständen auch eine Kamera mit höherer Bildrate eingesetzt werden.

Um auch bei schnellen Bewegungen einen abhängigen Winkel ω_{WM} (vgl. Abschnitt 5.4) zu ermitteln, kann auf dem Würfel Marker der untere linke Eckpunkt des grünen Rechtecks in einer anderen Farbe eingefärbt werden. Dies erfordert ein zusätzliches Keying um diese Ecke zu identifizieren und mit den generierten Eckpunkten abzuleichen.

6.2 Erweiterung von *MArC* als Augmented-Reality-System

Wie aus dem Projektmanagement in Kapitel 7 hervorgeht, war zunächst geplant, dass *MArC* als AR-Anwendung umgesetzt wird. So wie das System implementiert ist, ist die Möglichkeit nicht gänzlich verworfen worden. So ist es durchaus denkbar, nach dem Abschluss des vorliegenden Projekts die Systembasis von Virtual-Reality auf Augmented-Reality zu ändern. Hierzu müsste zunächst eine geeignete Stereo-Kamera an das *HTC Vive* Head-Mounted Display angebracht werden. Dabei ist darauf zu achten, dass man einen Kompromiss findet, sodass die Kamera zwar möglichst auf Augenhöhe des Benutzers am HMD befestigt wird, aber trotz-

dem möglichst wenige Sensoren des *HTC-Trackingsystems* verdeckt werden. Zudem muss bedacht werden, dass der *Leap Motion-Controller* ebenfalls Platz unterhalb der Kamera in Anspruch nimmt. Der Kalibrierungsansatz aus Kapitel 5.2 müsste ebenfalls entsprechend angepasst und gegebenenfalls erweitert werden.

6.3 Export von Architekturdaten

Eine weitere Erweiterungsmöglichkeit von *MArC* ist ein alternativer Export der Architekturdaten, wie sie für jedes Gebäude in dessen Kontextmenü angezeigt werden. Hat der Nutzer zum Beispiel eine Siedlung errichtet und diese gespeichert, wäre es denkbar, dass der Benutzer die Gebäudedaten in tabellarischer Form benötigt. Hierfür wäre es möglich die Gebäudedaten, die innerhalb der Szene auf den Kontextmenüs dargestellt werden zu exportieren. Denkbar wäre etwa eine übersichtliche *Excel-Tabelle*, innerhalb derer die Gebäude eindeutig benannt sind und dann deren Informationen kompakt zusammengefasst darunter notiert sind. Des weiteren könnten bestimmte Daten der einzelnen Gebäude bereits in einer Art Zusammenfassung dargestellt werden. Denkbar wäre auch, eine Excel Tabelle mit mehreren Arbeitsmappen zu erstellen, wenn mehrere Entwürfe für eine Siedlung zur Diskussion gestellt werden. Dann würde für jeden Entwurf eine Arbeitsmappe zur Verfügung stehen, innerhalb derer die Daten notiert sind. Ein direkter Datenvergleich der verschiedenen Siedlungsbauten wäre damit problemlos möglich.

6.4 Evaluierung von *MArC*

Lukas

Paul

Zur Erreichung des Gesamtziels einer jeden Produktentwicklung – nämlich ein marktreifes und kommerziell verwertbares Produkt zu erhalten – gehört für gewöhnlich auch die Beurteilung der Entwicklung mittels Benutzerstudien. Nur so lässt sich häufig eine Aussage darüber treffen, ob das Produkt in den Händen von unvoreingenommenen Testpersonen den gewünschten Zweck weiterhin erfüllt.

Eine solche Evaluierung wird für *MArC* als einen der weiteren Schritte nach Beendigung des vorliegenden Projekts ebenfalls empfohlen. Vor der Durchführung von Nutzerstudien sind sicher noch weitere Schritte nötig, um das „Produkt“ *MArC* als ebensolches abzurunden, aber schlussendlich ist eine Evaluierung durch potenzielle Endkunden unumgänglich. Im Falle von *MArC* ist der Nutzen für die Zielgruppe der Architekten nämlich keineswegs selbstverständlich. Neben dem theoretischen Nutzen, den ein perfekt funktionierendes System mit den gegebenen Eigenschaften bringen würde, sollte vor allem der tatsächliche Nutzen Gegenstand einer Untersuchungsreihe sein. Denn angenommen der theoretische Nutzen des Produktes ist immens, so könnte die unvorteilhafte Realisierung des Systems dennoch den tatsächlichen Nutzen erheblich einschränken. Dies ist insbesondere vor dem Hintergrund der Interaktionsmethoden in Mixed-Reality-Systemen zu sehen, da diese – je nach

Anwendung – keineswegs grundsätzlich intuitiv und effizient sind.

7 Projektmanagement

Dieses Kapitel beschreibt die Planung und das Management des Projektes *MArC*. Es ist in die einzelnen Projektphasen gegliedert, welche jeweils die wichtigsten Punkte der entsprechenden Phasen enthalten.

7.1 Projektdefinition

Die Projektdefinitionsphase dient in erster Linie dazu eine Übersicht über die notwendigen Schritte sowie die Problemstellungen und Risiken abzuschätzen. Dafür wird der Ist-Zustand erfasst, das Ziel des Projektes detailliert definiert und alle organisatorischen Fragen abgeklärt. Im Folgenden sind die Arbeitsschritte der Projektdefinitionsphase beschrieben.

7.1.1 Problemanalyse

Bisher erfolgte während der ersten Entwurfsphase die Visualisierung von Architekturprojekten vorwiegend mit (z.B. aus Pappkarton oder Styropor) aufwendig hergestellten Modellen. Diese werden im Laufe des Prozesses immer wieder verändert und verbessert. Um eine Darstellung von Modellen innerhalb einer bestehenden Umgebung zu ermöglichen, werden abstrahierte, ungenaue Bodenmodelle aus einzelnen Pappschichten geschnitten und übereinander geklebt. Jede Änderung des Entwurfs zieht viel Arbeit nach sich und es ist nicht oder nur sehr schwer möglich zu einem früheren Zwischenstand zurückzukehren. Weiterführend, müssen alle Zwischenstände während des Kreativprozesses umständlich mit einer Kamera dokumentiert werden. Ein einfaches flexibles Arrangieren der Objekte und ein Testen auf Licht und Schattenwurf ist nahezu unmöglich. Häufig fällt es den Architekten auch schwer Eigenschaften, wie die Anzahl der Stockwerke oder die gesamte bewohnbare Fläche abschätzen. Doch dies ist bei der Planung sehr wichtig, da in den meisten Fällen Vorgaben von den Auftraggebern festgelegt werden.

Das *MArC* Projektteam stellte den Kontakt zu einem Architekturbüro in Köln her um deren Planungsvorgehen zu sehen und nachvollziehen zu können. Anhand dieser Analyse stellten sich diese Probleme heraus und die im Anschluss zu den Projektzielen führten.

7.1.2 Projektziele und Anforderungen

Ziel des Projektes *MArC* ist es, Architekten eine neuartige Darstellung von Gebäuden und Objekten zu erlauben und ein flexibleres sowie nachhaltigeres Arbeiten in der ersten Planungsphase zu ermöglichen. Dieses System kann an einem beliebigen Tisch installiert werden, auf dem die Würfel-Marker platziert werden. Direkt über

dem Tisch wird eine Kamera montiert, welche für das Tracking der Würfel-Marker benutzt wird. Mittels Augmented oder Virtual Reality werden in dem Display einer VR-Bille (*HTC Vive*) die virtuellen Gebäude und Objekte aus den CAD Dateien extakt an die Positionen der Würfel-Marker gerendert. Diese virtuellen Objekte können per Hand mit den getrackten Würfel-Markern manipuliert werden. So entsteht eine VR oder AR Szene, die aus allen Blickwinkeln betrachtet und jederzeit durch weitere Objekte erweiterbar ist.

Zusätzlich soll eine Timeline implementiert werden, die es dem Nutzer ermöglicht jederzeit einen Zwischenstand abzuspeichern oder aufzurufen. Eine weitere Besonderheit in Form von automatischen Berechnungen der Gebäudeeigenschaften und deren Darstellung in der virtuellen Welt soll dem Architekten bei der Planung unterstützen. Diese Berechnungen sollen vom System in eine Exceldatei exportiert werden.

Das Projekt soll in einem Team von vier Personen durchgeführt werden. Hierfür stehen 12 ECTS Punkte pro Person und insgesamt 48 ECTS mit daraus resultierenden 1440 Stunden an Zeitressourcen für das gesamte Projekt zur Verfügung.

7.1.3 Lösungskonzept

MArC soll mit Hilfe von Augmented oder Virtual Reality eine intuitive und flexible Arbeit mit abstrackten Entwurfsmodellen ermöglichen. So kann per Augmented Reality Brille ein Modell auf einem Tisch visualisiert und grundlegend manipuliert werden, also skaliert in alle Achsen, verschoben und rotiert werden. Die Skalierung geschieht mittels des Kontextmenüs, welches Anfasser für jede Achse besitzt. Die Rotation wird durch die Drehung des Würfel-Markers hergerufen, eine Verschiebung durch ein Verschieben der Würfel-Marker.

Anordnungen von mehreren Modellen in einer Umgebung (z.B. Siedlung) können exportiert und so direkt weiterverarbeitet werden. Hierfür sieht *MArC* zwei Möglichkeiten vor: Zum einen werden Szenen gespeichert und in einem Menü neben der Arbeitsfläche dargestellt, in dem sie wieder aufgerufen werden können. Des Weiteren können Daten von Siedlungen als Tabellen gespeichert werden, um später auch in anderen Programmen weiter verwendet werden zu können. Durch die VR-Brille mit montierten Kamera wird ein Abbild der realen Umgebung gezeigt. Auf einem Tisch liegende Würfel werden durch das innovative System getrackt; diese können von dem Benutzer intuitiv verschoben und arrangiert werden.

Das Tracking der Würfel-Marker erfolgt wenn möglich ohne Marker (Markerless-Tracking, d.h. die Würfel-Marker die der Benutzer anfasst sind nicht mit Codes oder Mustern versehen). Hierfür wird eine Kamera über dem Tisch montiert, welche die Arbeitsfläche filmt. Diese wird an einen Computer angeschlossen, der das Tracking durchführt. Für die Markererkennung des Trackingprogramms wird *OpenCV* verwendet.

Die Teilsysteme werden über eine Netzwerkverbindung (TCP) miteinander verbunden. So kommunizieren der Computer der für das Tracking zuständig ist mit dem Computer der die Darstellung mittels *Unity* umsetzt. Markerdaten werden übertragen und diese Informationen als Basis zum Rendern der Gebäude genommen. Das

Terminziel ist das Ende des Wintersemesters 2016/2017. Die Projektabgabe soll am 28.2.2017 fertig gestellt worden sein. Die Dokumentation ist am 31.3.2017 Fertigzustellen und die Präsentation am 10.5.2017, wobei sich dieser Termin im Laufe der Arbeit ergab.

7.1.4 Durchführbarkeitsanalyse

In diesem Abschnitt soll die Durchführbarkeit des Projektes erklärt werden. Diese Durchführbarkeitsanalyse stützt sich auf die folgenden Analysen:

- Technologische Durchführbarkeit
- Ökonomische Durchführbarkeit
- Rechtliche Durchführbarkeit
- Operationale Durchführbarkeit
- Terminliche Durchführbarkeit

Technologische Durchführbarkeit Für die Technologische Durchführbarkeit stehen sowohl die Technischen Möglichkeiten, als auch das Know-how der Mitarbeiter im Zentrum. *MARc* nutzt neuartige Technologien, die einerseits zum Teil in der Hochschule vorhanden waren, teilweise jedoch in Übersee erworben werden mussten. So war die *HTC Vive* das Mittel der Wahl zur Darstellung der virtuellen Elemente. Diese war bereits in der Hochschule vorhanden und es gab einige Kommilitonen, die bereits damit Erfahrungen gesammelt hatten. Demzufolge konnte das Projektteam sporadisch auf vorhandenen Erfahrungen zurückgreifen. Ebenso verhielt es sich mit der *LeapMotion*, welche von der TH zur Verfügung gestellt wurde und ebenfalls schon in anderen Projekten verwendet wurde. Auch hier konnte zur besseren Machbarkeitsabschätzung auf bestehende Erfahrungen zurückgegriffen werden. In beiden Fällen stellte sich die Idee diese Geräte zu verwenden als gute Lösung heraus, unter anderem wegen dem Erfolg der anderen Projekte.

Anders verhielt es sich jedoch mit der *OVR Vision*. Dieses Gerät war nicht im Besitz der Hochschule und kein Teammitglied hatte Erfahrung mit diesem Gerät. Da jedoch alle weiteren Geräte funktional und mit guter Erfahrungswerten zur Verfügung standen, entschied sich das Team die *OVR Vision* zu kaufen um die Augmented Reality im System umzusetzen.

Ziel der Verwendung der *OVR Vision* war es, aus der Virtuellen Realität und haptischen Komponente der Würfel-Marker eine Augmented-Reality-Anwendung zu schaffen, welche dem Benutzer die Darstellung der virtuellen Elemente in seiner Umgebung ermöglicht.

Für das Tracking sollte eine bereits vorhandene Industriekamera von *IDS* verwendet werden. Diese wurde ebenfalls in vielen Fachbereichen in der Hochschule für die Lehre und Forschung genutzt und galt als zuverlässig.

Des Weiteren konnte das Team auf weitreichendes und individuelles Wissen zurückgreifen. So waren unter anderen bereits eine Bachelorarbeit mit *Unity* und einem

HMD erarbeitet worden und zwei andere Teammitglieder hatten tiefgreifendes Wissen über die digitale Bildverarbeitung. Letztere ist für das Marker-Tracking unumgänglich. Wissenslücken bestanden jedoch in der Umsetzung und Entwicklung der Augmented Reality. Dieser Bereich ist technologisches Neuland und im Laufe des Projektes startete Microsoft den Verkauf der *Hololens*, welches ein echtes Augmented-Reality-HMD ist. So wurden im Modul „Virtuelle und Erweiterte Realität“, welches alle der Mitglieder vor dem Projektbeginn belegten, das geforderte Know-how zur Entwicklung Augmented-Reality-Anwendungen nicht vermittelt wurde.

Ökonomische Durchführbarkeit Neben der Technologischen Durchführbarkeit war auch die Ökonomische zu erörtern. Durch die Rahmenbedingung des Masterprojektes im Studiengang Medientechnologie mussten keine Gehälter bezahlt werden und die Kosten verringern sich auf die erworbene Hardware und benötigte Zubehör. Diese Gesamtkosten überschreiten nicht das bereitgestellte Budget. Wichtiger jedoch sind die Zeitkosten der einzelnen Mitglieder. So hat jedes Mitglied ein eigenverantwortliches Zeitmanagement durchzuführen und genügend Zeit für das Projekt zur Verfügung zu stellen. Da das Masterprojekt semesterbegleitend war musste jedes Mitarbeiter in seinem Zeitmanagement auch den Aufwand andere Module berücksichtigen. Der Zeitraum des Projektes ging jedoch über zwei Semester, sodass der Aufwand im vorgegebenen Ökonomischen Rahmen machbar war.

Rechtliche Durchführbarkeit Dieser Punkt sei zur Vollständigkeit genannt. Da es sich bei *MArC* um ein studentisches Projekt handelt welches nicht verkauft werden sollte, ist das Projekt in seinem geplanten Umfang rechtlich durchführbar.

Operationale Durchführbarkeit Da jedes Teammitglied individuelles, tiefgreifendes Wissen zur Verfügung stellt und alle Mitglieder bereits im Bereich Virtual Reality gearbeitet haben, ist großes Potential für das Projekt vorhanden. Lediglich die Weiterführung von Virtueller in Agmentierter Reality ist für die Mitglieder neu und muss zusätzlich erlernt werden. Durch den Zeitrahmen ist die Operationale Durchführbarkeit gut.

Terminliche Durchführbarkeit Insgesamt wurden für das Projekt 12 ECTS $\times 4$ Personen $\times 30$ Stunden = 1440 Arbeitsstunden eingerechnet. Innerhalb diesen Rahmens ist das Projekt bis zum Ende des Wintersemesters 2016/2017 gemeinsam fertig zu stellen. Terminlich ist bei aufgeteilter Arbeitslast ein Projekt wie *MArC* machbar, wobei diese Machbarkeit mit dem betreuenden Professor abgesprochen und auf dessen Erfahrungen zurückgegriffen wurde.

7.1.5 Projektauftragsformular

Das Projektauftragsformular ist in Tabelle 8 zu finden.

Projektauftrag	
Projektname	MArC (Mixed Reality Architecture Composer)
Projektleiter	Paul Berning, Lukas Kolhagen
Projektanlass	Masterprojekt Master Medientechnologie
Projektziele	<p>Erstellung einer Augmented bzw. Virtual Reality Anwendung mit der Architekten Siedlungen in der ersten Entwurfsphase bauen und einfach nachträglich bearbeiten können.</p> <p>Dabei soll der Nutzer virtuelle Gebäude mittels getrackter Würfel verschieben und rotieren können und sämtliche Interaktion (Manipulation der Größe der Gebäude, Menü-Interaktionen) mit den Fingern durchführen können.</p>
Zu erarbeitende Ergebnisse	<p>Abgeliefert werden zwei lauffähige .exe Dateien, die auf der einen Seite das Tracking auf einem Computer, und auf dem anderen die Darstellung ermöglichen.</p> <p>Eigentümer der Hardware ist zu jeder Zeit der Auftraggeber. Das Projektteam ist dafür verantwortlich, dass die Software im vorgegebenen Rahmen funktioniert. Es wird dazu eine Readme Datei ausgeliefert, die Verwendung der Programme erklärt. Eine ausführliche Dokumentation ist bis zum 30.3.2017 abzuliefern, eine Abschlusspräsentation mit allen Ergebnissen ist am 10.5.2017 zu halten. Zusätzlich erstellt das Projektteam ein Video welches das Lauffähige Ergebnis dokumentiert.</p>
Projektbudget	Das Projekt unterliegt einem monetären Budget von 500 Euro und einem zeitlichen Budget von 4 Personen × 12 ECTS × 30 Stunden = 1440 Stunden.
Randbedingungen	Sämtliche Öffnungszeiten der Räumlichkeiten der Arbeitgeber, Verfügbarkeit des begleitenden Professors für Termine nur nach Absprache, Einhalten von Projektbudget und Zeitrahmen, insbesondere des individuellen Zeitmanagements der Mitglieder wegen anderer Universitärer Veranstaltungen
Termine und Meilensteine	Projektabgabe: 28.2.2017, Abgabe der Dokumentation: 30.3.2017, Projektpräsentation und Abgabe des Videos: 10.5.2017

Tabelle 8: Projekt-Auftragsformular von *Marc*.

7.1.6 Projektorganisation

Leitung Die Leitung des Teams wurde aufgeteilt zwischen Lukas Kolhagen und Paul Berning. Die ursprünglich gedachte Aufteilung in die beiden Semester ist einer durchgängigen Kooperation und Absprache gewichen, welche das Team organisatorisch stützte.

Team Das Team teilte sich des weiteren in zwei Unterteams auf. Basierend auf den unterschiedlichen individuellen Kompetenzen wurde das Tracking der Marker hauptsächlich von Laura Anger und Vera Brockmeyer übernommen. Auf der anderen Seite konnten Lukas Kolhagen und Paul Berning fundierte Unity Kenntnisse aufweisen und kümmerten sich um die visuelle Darstellung der Applikation.

Infrastruktur Als Arbeitsplatz wurde seitens der Hochschule ein Raum zur Verfügung gestellt, der über eine fest installierte *HTC Vive* und einem Computer, welcher die Systemvoraussetzungen der *HTC Vive* erfüllt, verfügt. In diesem wurde das Projekt durchgeführt. Optional standen dem Team weitere performante Laptops zur Verfügung und die privaten Notebooks wurden für die Entwicklung genutzt. Die benötigte *OVR Vision* sowie erforderliches Zubehör, wie Halterungen und Aluminium-Würfel wurden im Laufe der Zeit hinzu gekauft. Weiterführend, wurden die *LeapMotion* und eine Webcam von der Hochschule zur Verfügung gestellt.

7.2 Projektplanung

Die Projektplanung ist die zweite Phase des Projektes. In dieser werden die definierten Ziele systematisch in Arbeitspakete und Meilensteine zerlegt und Pläne erstellt, die zur Realisierung der darauffolgenden Phase behilflich sind. Im folgenden sind die Arbeitspakete und diversen Pläne aufgelistet.

7.2.1 Arbeitspakete

Wie in größeren Projekten üblich, wurde die Arbeit in mehrere Arbeitspakete (AP) und Meilensteine (MS) segmentiert. Die detaillierte Auflistung dieser kann in Tabelle 9 gefunden werden.

7.2.2 Projektstrukturplan

Der Projektstrukturplan ist dem Anhang in Abbildung 29 beigefügt.

7.2.3 Ablaufplan, Terminplan (Gantt Chart)

Der Ablaufplan und Terminplan wurde der Übersichtshalber mit dem Plan über die verschiedenen Arbeitspakete und Meilensteine in Tabelle 9 aufgeführt. Das Gantt

Paketname (AP) / Meilenstein (MS)	Start	Ende	Verantwortl.:
AP1 – Organisation und Planung	21.03.2016	10.05.2017	PB/LK
AP2 - Trackingalgorithmus	2.5.2016	5.2.2017	VB/LA
MS2.01 Recherche	2.5.2016	31.5.2016	VB/LA
MS2.02 Implementierung	1.6.2016	1.7.2016	VB/LA
MS2.02 Kalibrierung entwickeln	15.12.2016	28.2.2016	VB/LA/LK
AP3 - Framework	24.5.2016	13.7.2016	PB/LK
MS 3.01 Unity und Github einrichten	24.5.2016	31.5.2016	PB/LK
MS 3.02 HTC Vive anbinden	13.7.2016	16.11.2016	PB/LK
MS 3.03 Trackingkamera besorgen	16.06.2016	1.7.2016	PB/LK
MS 3.04 Stereokamera einrichten	1.7.2016	13.7.2016	PB/LK
MS 3.05 Implementierung Tablemenü	1.11.2016	15.12.2016	PB/LK
MS 3.06 Implementierung Kontextmenü	1.11.2016	15.12.2016	PB/LK
MS 3.07 Implementierung arch. Berechn.	15.11.2016	15.12.2016	LA/LK
MS 3.08 Implementierung Netzwerkverbindung	15.11.2016	15.12.2016	LA/VB/LK
MS 3.09 Implementierung Webcam	1.1.2017	1.2.2017	TEAM
MS 3.09 Recherche Hololens	16.12.2016	20.2.2017	TEAM
MS 3.09 Leap Controller einbinden	15.11.2016	15.2.2017	PB/LK
AP4 - Hardware	31.5.2016	13.7.2016	PB/LK/LA
MS 4.01 Würfelmärker erstellen	2.5.2016	31.5.2016	LA
MS 4.02 Testbilder für Tracking erstellen	30.5.2016	17.6.2016	PB/LK
MS 4.03 Tracking Kamera Montage	1.6.2016	15.6.2016	PB/LK
MS 4.04 PC besorgen und einrichten	30.5.2016	17.6.2016	PB/LK
AP5 - CAD Import/Export	1.8.2016	13.9.2016	PB/LK
MS 5.01 CAD / 3D Formate festlegen	1.8.2016	9.8.2016	PB/LK
MS 5.02 3D Import implementieren	9.8.2016	13.9.2016	PB/LK
AP6 - Testing	10.7.2016	27.11.2016	TEAM
MS 6.01 Usability intern	10.7.2016	15.7.2016	TEAM
MS 6.02 Usability mit Experten	22.11.2016	27.11.2016	TEAM
AP7 - Dokumentation	1.6.2016	31.3.2017	TEAM
MS 7.01 Schriftliche Ausarbeitung	1.6.2016	31.3.2017	TEAM
MS 7.02 Abschlusspräsentation	24.4.2017	10.5.2017	TEAM
AP8 - Prototypen	2.5.2016	TERMIN	TEAM
MS 8.1 Prototyp1 (Paper Mockup)	2.5.2016	2.5.2016	TEAM
MS 8.2 Prototyp2 (runnable)	3.5.2016	13.7.2016	TEAM
MS 8.3 Prototyp3 (last prototype)	13.7.2016	16.11.2016	TEAM

Tabelle 9: Identifikation der Arbeitspakete und Meilensteine.

Chart ist der Übersicht halber im Anhang in Abbildung [28](#) beigelegt.

7.2.4 Kapazitätsplan

Der Kapazitätsplan enthält die Ressourcen über die Projektzeit von März 2016 bis März 2017. Der Plan ist tabellarisch sowie als Diagramm im Anhang in Abbildung [24](#) zu finden.

7.2.5 Kostenplan

Der Kostenplan ist ebenfalls dem Anhang in Abbildung [26](#) beigelegt.

7.2.6 Qualitätsplan

Der Qualitätsplan ist ebenfalls dem Anhang in Abbildung [27](#) beigelegt.

7.3 Projektdurchführung

Die Projektdurchführungsphase dient der Erarbeitung der Projektergebnisse. Hierbei ist für das Projektmanagement die zielgerichtete Lenkung der Tätigkeiten von wichtiger Bedeutung. Die Arbeit der Projektdurchführungsphase wird im folgenden beschrieben.

7.3.1 Kommunikation im Team und nach Außen

Die Kommunikation ist für den Erstellungsprozess von hoher Wichtigkeit. *MarC* wurde innerhalb des vierköpfigen Teams und in den kleineren zweiköpfigen Teams umgesetzt. Zur Kommunikation standen Email, Telefone, *Whatsapp* zur Verfügung. Der Datenaustausch der Tracking- und *Unity*-Anwendungen wurde mit einem *Github* Repository umgesetzt. Ebenfalls standen die Cloud-Dienste *Dropbox* und *Google Drive* zur Verfügung.

Die interne Team-Kommunikation lief zum Großteil über *Whatsapp* und das Telefon. Zusätzlich wurden Wochenberichte erstellt, um das Team und den Auftraggeber auf dem Stand der Dinge zu halten.

Die weitere Kommunikation mit dem Auftragsgeber wurde über den Projektleiter mittels Email gehandhabt. Bei wichtigen Entscheidungen war stets das Team im CC, sodass alle Mitglieder im Bilde waren. In dringenden Fällen wurde darüber hinaus zusätzlich über *Whatsapp* kommuniziert.

7.3.2 Maßnahmen zur Problemvermeidung

Über die für Studienprojekte vergleichsweise lange Projektlaufzeit treten gezwungenermaßen Probleme auf denen mit geeigneten Maßnahmen entgegen zu wirken ist.

Die flexible Entwicklung von Lösungen und deren Umsetzung ist ein entschiedener Prozess für eine erfolgreiche Umsetzung der Projektziele. Während der Projektaufzeit wurde versucht auch mit kurzfristig auftretenden Problemen so umzugehen, dass ein zufriedenstellendes Ergebnis realisiert wird. In diesem Abschnitt sollen einige aufgetretene Probleme exemplarisch vorgestellt werden.

Ausfall des Planungstools Bitrix24 Das kostenlose online Planungstool Bitrix24 wurde für die anfängliche Planung genutzt und stand dem Team bis September 2016 zur Verfügung. Während der Betriebsferien im September fanden keine Arbeiten am Projekt und im Planungstoll statt. Mit der Folge, dass das Projekt von den Betreibern des Web-Tools *Bitrix24* gelöscht wurde. Ein interner Mechanismus hatte gegriffen und das Projekt als obsolet erkannt. Da alle erstellten Arbeitspakete und Meilensteine im Projektplan notiert waren, konnte das Problem mit einfachen Mitteln gelöst werden. Für die fortlaufende Projektplanung wurde im Anschluss Excel verwendet und die Daten lokal gespeichert.

Ausfall von Hardware Während des Projektes fiel die *OVR Vision* wegen eines technischen Defektes aus. Das führte zu dem Problem, dass kein Videobild im HMD angezeigt werden konnte und nur die virtuelle Umgebung sichtbar war. Gleichzeitig führte der Defekt dazu, dass in einem der Arbeitslaptops die USB Ports durchgebrannt sind. Die Garantieabwicklung mit dem Hersteller gestaltete sich als schwierig, da das Gerät nach Japan verschifft werden musste. Die Gesamtdauer der Reparatur war so nicht planbar.

Es stand die Entscheidung an, ob auf die Reparatur gewartet werden soll und im Anschluss mit dem Gerät weiter gearbeitet wird oder eine Alternative zu finden. Hierbei bestand das Risiko, dass die Kamera nach der Reparatur erneut kaputt geht.

Als Lösung spaltete sich das Team für eine Woche in zwei Gruppen auf und analysierte die Alternativen. Eine erste Möglichkeit war eine einfache Webcam zu verwenden und auf den Stereoeffekt zu verzichten.

Die andere Option in Form der kürzlich veröffentlichten *Hololens* stellte sich als nicht praktikabel heraus. Die gesamte Applikation müsste auf diese Brille geladen und ausgeführt werden. Dies erforderte eine WLAN Verbindung zum Computer mit der Tracking-Anwendung statt der bereits bestehenden LAN Verbindung. Für diese Lösung fehlte es an Erfahrungen und Quellen in Verbindung mit dem Gerät und es bestanden Zweifel an der Umsetzbarkeit der Anwendung. Aus diesen war zu komplex und zeitaufwändig, um sie als Alternative zu verwenden.

Die zweite Lösung wurde jedoch weiter verwendet. Es wurde eine Webcam an das HMD angebracht und das Videobild in *Unity* vor der virtuellen Kamera gerendert. Diese Lösung wurde am Ende jedoch ebenfalls nicht in den letzten Prototypen eingebaut. Hier war das ausschlaggebende Problem die unzureichende Qualität der Darstellung sowie die mangelnde Zeit zur Umsetzung der Kalibrierung. Das Team hatte mit Positionsabweichungen zwischen den echten Markern auf dem Tisch und virtuellen Objekten zu kämpfen. Demzufolge hatte das reale Bild eine mangelhafte Deckung von Kamerabild und virtueller Welt die den Nutzer verwirrte.

Daher entschied sich das Team für eine Mixed-Reality-Anwendung als Produkt zu realisieren.

Markerless Tracking Zu Beginn des Projektes sollte die Aluminiumwürfel ohne Markierung getrackt werden (sog. Markerless tracking). Als Problem stellt sich hierbei heraus, dass diese Umsetzung innerhalb des Projektes zu zeitaufwändig und komplex war. Der Großteil der Projektzeit hätte auf diese Umsetzung angewendet werden müssen. Die größte Herausforderung war die eindeutig Identifizierung der texturarmen Marker, welche nach längerer Verdeckung nicht wieder eindeutig zugeordnet werden konnten. Die naheliegende Lösung war codebasierte Marker einer bestehenden Bibliothek zu verwenden.

7.4 Projektabschluss

Ist die Entwicklung abgeschlossen, ist damit nicht automatisch das gesamte Projekt beendet. Nach der praktischen Phase werden die Ergebnisse festgehalten und Präsentiert.

MArC begann als ehrgeiziges Projekt mit vielen Features. Von Einigen musste das Team Abstand nehmen um erfolgreich einen Prototypen zu entwickeln, welcher die Kernziele erfüllt. Anders als geplant wurde das System vor allem wegen der Probleme mit der *OVR Vision* und der weiteren Hardware nicht als Augmented-Reality-System verwirklicht. Stattdessen musste das Konzept angepasst werden und ein Mixed-Reality-System wurde weiterverfolgt. Die Projektziele mussten auch beim Marker-Tracking angepasst werden und aus dem *Markerless Tracking* wurde aus Zeitgründen ein *Markerbasiertes Tracking*, da sich das Problem der eindeutigen Identifizierung der Würfel-Marker als zu komplex hervorstellte und deutlich mehr Arbeitsaufwand erforderte, als die Planung vorsah. Für die Umsetzung des *Markerbasiertes Tracking* war lediglich eine Modifikation der Würfel-Marker notwendig und deren Tracking konnte mit einer bestehenden Bibliothek umgesetzt werden.

Ebenso konnten optionale Features wie der Import von Höhenmodellen nicht umgesetzt werden. Diese waren zur besseren Visualisierung der Siedlung und als Planungshilfe für die Architekten geplant. Doch die Implementierung konnte nicht mehr innerhalb der zur Verfügung stehenden Zeit realisiert werden, da die Kernziele mehr Aufwand erforderten als vorgesehen.

Im weiteren folgen Informationen zu der Abschlusspräsentation und dem Projektvideo. Außerdem ist eine detaillierte Reflexion im Kapitel 7.5 zu finden.

7.4.1 Abschlusspräsentation

Für das Projekt wird eine detaillierte Präsentation mit Praktischer Vorstellung des Ergebnisses erarbeitet. Diese wird am 10.5.2017 stattfinden.

7.4.2 Video

Das Team wird für die Präsentation ein Video vorbereiten. Dieses stellt das Projekt vor und zeigt die Funktionalität. Es wird bis zum 10.5.2017 stattfinden. Das Team trifft sich für die Produktion in den Projekträumen und setzt diese um.

7.5 Reflexion

Das Masterprojekt *MArC* war mit seiner Laufzeit von zwei Semestern die bislang längste praktische Projektarbeit für alle beteiligten Teammitglieder.

Das Projekt wurde fertiggestellt und das Ziel, eine funktionierende Anwendung zu entwickeln wurde erreicht, wenn auch nicht in allen Details wie ursprünglich geplant. Aufgrund mangelnder Erfahrung gab es im Projektverlauf einige Probleme, die im Nachhinein gut erkennbar und identifizierbar sind. Wichtig für die Teammitglieder ist es, diese Fehler im Nachhinein zu erkennen und zu bewerten, damit sich diese in späteren Projekten nicht wiederholen. So stellte sich im Nachhinein heraus, dass das Team am Anfang sehr viele gute Ideen sammelte, sich jedoch zu ambitionierte Ziele setzte. Mit diversen Features, die dem Benutzer gut gefallen könnten, sollte *MArC* von Anfang an ein Tool werden, welches viele Probleme des “analogen“ Arbeitsvorgangs der Architekten bei Siedlungsplanungen verhindern oder abschwächen sollte. Z.B. das Speichern und Öffnen von Szenen, was umgesetzt wurde, oder den Import von Höhenmodellen, welcher z.B. nicht umgesetzt wurde. Wie sich während der Projektdauer herausstellte, war dies für den gegebenen Zeitraum zu ehrgeizig. So wäre es ratsam gewesen zuerst ein funktionierendes Grundgerüst der Anwendung zu bauen und anschließend schrittweise neue Features einzubauen. Kurz vor Ablauf der Entwicklungszeit waren einige grundlegende Funktionen, wie z.B. die Kalibrierung (vgl. Abschnitt 5.2 noch nicht fertig gestellt und einzelne Systemkomponenten schienen sich gegenseitig zu behindern. Um das Projekt trotz der gegebenen Schwierigkeiten fertig zu stellen, wurde genau diese Herangehensweise umgesetzt. Es wurden alle Menüs und Hardware-Komponenten (bis auf die *HTC Vive*) aus der *Unity*-Simulation entfernt und erst einmal ein funktionierendes Grundgerüst gebaut. Anschließend wurden die Menüs und alle anderen Komponenten schrittweise hinzugefügt. Dies hatte zur Folge, dass Fehlerquellen einfach gefunden werden konnten und die Umsetzung der Anwendung wesentlich strukturierter und somit leichter von der Hand ging.

Des weiteren konzentrierten sich die hochmotivierten Teammitglieder zu schnell auf Detailfragen der praktischen Umsetzung. Es wäre besser gewesen, für jedes Teilproblem genaue Analyse und Recherche zu betreiben, um herauszufinden, welche Methode benötigt wird und ob diese schon einmal verwendet wurde. So hätten häufiger bereits fertige, verfügbare Umsetzungen für spezifische Probleme genutzt werden können. Mit einem breiteren und tieferen Vorwissen wären einige Fehler leichter erkennbar gewesen. Zum Beispiel beharrte das Team lange Zeit auf einem markerlosen Tracking. Die wenigen Merkmale der Würfel, die für das System verwendet werden sollten, ließen ein zuverlässiges und performantes, markerloses Tracking jedoch nicht zu, wie sich herausstellte. So wurde viel Zeit auf die Entwicklung eines markerlosen Trackings verwendet, welche besser für die Arbeit an anderen Baustellen hätte ge-

nutzt werden können. Auch fand das Team am Anfang das kostenlose Planungstool *Bitrix24*, welches scheinbar schnell und problemlos das Planen solcher Projekte erlaubte. Dass Accounts und Projekte nach einiger Zeit der Nichtbenutzung einfach gelöscht werden wurde dabei völlig übersehen. Glücklicherweise gab es die Planungsunterlagen noch an anderer Stelle, dennoch hätte so etwas nicht passieren dürfen. Bei der Planung wollte das Team drei Prototypen entwerfen. Die Meilensteine zum Erreichen dieser Prototypen waren jedoch zeitlich zu knapp hintereinander angesetzt und die möglichen technischen Schwierigkeiten wurden dabei unterschätzt. Hier hätte man auf die Erfahrung anderer Kommilitonen oder Professoren setzen müssen, um eine realistischere Einteilung zu erreichen.

Es wurde auf Hardware gesetzt, die den Ansprüchen des Projekts nicht gerecht wurde. So erlitt zum Beispiel die *Ovrvision Pro* Stereo-Kamera im Projekt einen Hardwaredefekt und musste ausgetauscht werden. Das Team hätte sich früher gegen die Lösung mit dieser Kamera entscheiden müssen. Problematisch war hier, dass es wenig Erfahrungsberichte zu der Stereokamera gab und das Team daher die Qualität dieser nicht abschätzen konnte. Ein weiteres Problem stellte die *uEye*-Kamera dar, diese hat eine zu geringe Framerate um ein absolut flüssiges Tracking zu ermöglichen. Dies hätte auch im Vorhinein durch gründliche Recherche erkannt und eine alternative Kamera verwendet werden müssen.

Die Wichtigkeit der Kalibrierung der Kameras war dem Team lange nicht bewusst. Das Team wusste nicht, wie genau eine solche Kalibrierung gemacht werden musste und kommunizierte dies nach außen ungenügend. Auch in diesem Thema hätte das Team frühzeitig tiefergehend recherchieren müssen und sich informieren müssen. Dies war bis zum Projektende ein Problem, welches das Team nur schwer entgegen kommen konnte und am Ende zeitliche Probleme bekam.

Auch auf programmiertechnischer Ebene gab es strukturelle Unzulänglichkeiten. So wurden nicht von Anfang an Klassen und Skripte, die für die Anwendung geschrieben werden sollten, intensiv geplant und durch geeignete Interfaces definiert, sondern es wurde häufig voreilig mit der Programmierarbeit begonnen.

Alles in allem kann das Team aus dieser Zeit also viele Erfahrungen mitnehmen. Ist auch nicht alles glatt gelaufen, so ist *MArC* dennoch entstanden und kann am 10. Mai 2017 präsentiert werden.

8 Zusammenfassung

Lukas

Paul

In der vorliegenden Dokumentation wurde *MArC*, der „Mixed Reality Architecture Composer“ vorgestellt und detailliert beschrieben.

Zunächst erfolgte dazu in Kapitel 1 die Einordnung des Projektvorhabens in die Ziel-Umgebung, außerdem wurde das Ziel des Projekts erläutert. Das Produkt *MArC* sollte ein Werkzeug sein, welches Architekten die frühe Entwicklung von Siedlungen erleichtert. Auf architektonische Details der einzelnen Gebäude wurde dabei bewusst

verzichtet, da *MArC* vor allem bei der Platzierung der Gebäude behilflich sein soll. In Kapitel 2 wurden Grundlagen erörtert, die für die nachfolgenden Teile der Dokumentation für den Leser relevant sind. Diese Grundlagen beinhalten Theorie für die Entwicklung von Virtual-Reality-Anwendungen und damit verbundene Interaktionsmethoden, den grundlegenden Aufbau von Netzwerk-Kommunikationssystemen, sowie eine Einführung in markerbasierte Objektverfolgung.

Alle während des Projekts verwendeten und erstellten Ressourcen, insbesondere Hard- und Software, wurden im Detail in Kapitel 3 thematisiert. Auch die in der ursprünglichen Projektplanung vorgesehene, später aber doch nicht verwendete Hardware wird dort beschrieben.

In Kapitel 4 wurde das während der Projektlaufzeit entwickelte System vorgestellt. Neben dem grundsätzlichen Aufbau und der Verwendung des Systems durch einen Benutzer wurden auch die Netzwerk-Kommunikation der beiden Haupt-Softwarekomponenten, die Menüführung vor dem Starten der *Unity*-Simulation, sowie die Interaktion mit dem System als VR-Anwendung beschrieben.

Dem Tracking Programm als einem der Hauptteile von *MArC* wurde das Kapitel 5 gewidmet. Darin wird sowohl die Anbindung der Kamera über dem Arbeitsbereich an die Tracking-Anwendung beschrieben, als auch die Kalibrierung des Systems mit den Koordinatentransformationen thematisiert, die nötig sind, um die Positionen der Würfel-Marker in den Koordinatenraum der *Unity*-Simulation zu überführen. Des Weiteren wird der Umgang der Tracking-Anwendung mit der Detektion, Verfolgung und Registrierung der verwendeten Marker behandelt.

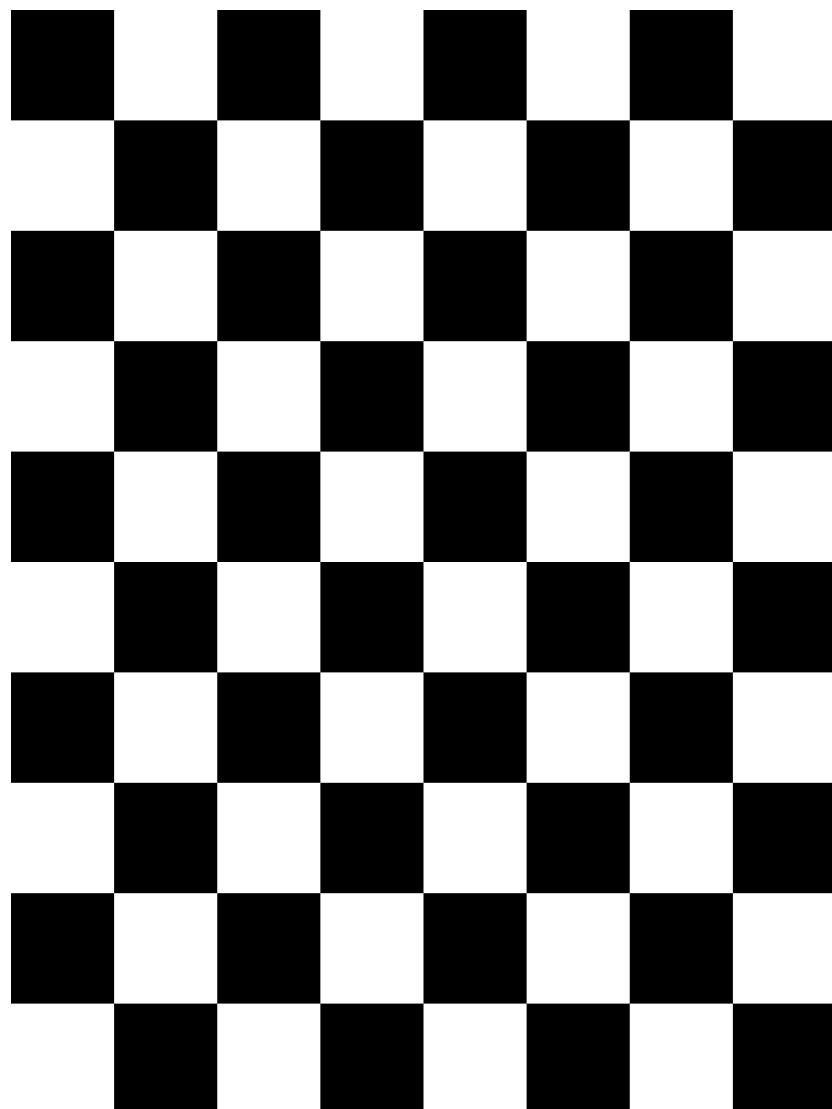
In Kapitel 6 wurde ein kurzer Ausblick darüber gegeben, welche dem Projekt folgenden, weiteren Entwicklungsmaßnahmen von *MArC* das Projektteam als sinnvoll erachtet. Dies beinhaltet neben Verbesserungen des verwendeten Marker-Tracking-Systems eine Erweiterung von *MArC* mit einer auf dem Head-Mounted-Display montierten Kamera zu einer Augmented-Reality-Anwendung, wie sie zu Beginn dieses Projekts geplant war. Im Ausblick wird zusätzlich der Import- und Export von 3D-Modell-Daten für die Interoperabilität von *MArC* mit Modellierungs- und Entwicklungssoftware behandelt, welche in Architekturbüros Anwendung findet. Außerdem wird eine Studie zur Gebrauchstauglichkeit von *MArC* empfohlen, um insbesondere die Interaktionsmethoden der VR-Anwendung daraufhin zu überprüfen, ob diese tatsächlich zu einer gesteigerten Effizienz für Architekten bei der Siedlungsentwicklung führen.

Mit dem Projektmanagement, welches bei einem Masterprojekt wie *MArC* eine zentrale Rolle spielt, befasst sich das Kapitel 7. Dort wird die zu Beginn des Projekts erarbeitete Projektdefinition ebenso thematisiert wie die Projektplanung mit Arbeitspaketen, Meilensteinen und verschiedenen Plänen, die helfen sollen, die Planung und den Fortschritt des Projekts zu quantifizieren. Der Durchführung des Projekts mit dem Fokus auf der Zusammenarbeit im Projektteam und dem Projektabschluss sind ebenfalls Abschnitte gewidmet. Als weiteren wichtigen Teil der Analyse des Projekts ist die Reflexion in Abschnitt 7.5 anzusehen. Da die Erwartungen an das Projekt *MArC* mehrfach während der Projektdauer angepasst und gesenkt werden mussten, weil viele unerwartete – und teils selbst verschuldete – Probleme auftraten, war es im Interesse des Projektteams, das Projekt besonders im Hinblick auf diese Schwierigkeiten kritisch zu analysieren und diese Reflexion in die Dokumentation

mit einfließen zu lassen.

9 Anhang

9.1 Schachbrett zur Kamerakalibrierung



7x9 checkerboard for camera calibration

Abbildung 23: Um den Faktor 0.6 skaliertes Schachbrett zur Kamerakalibrierung.

Download: http://www.mrpt.org/downloads/camera-calibration-checker-board_9x7.pdf

9.2 *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

9.3 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

9.4 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

9.5 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         }
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

Abbildung 24: Ressourcenplan von *MArC* in tabellarischer Form.

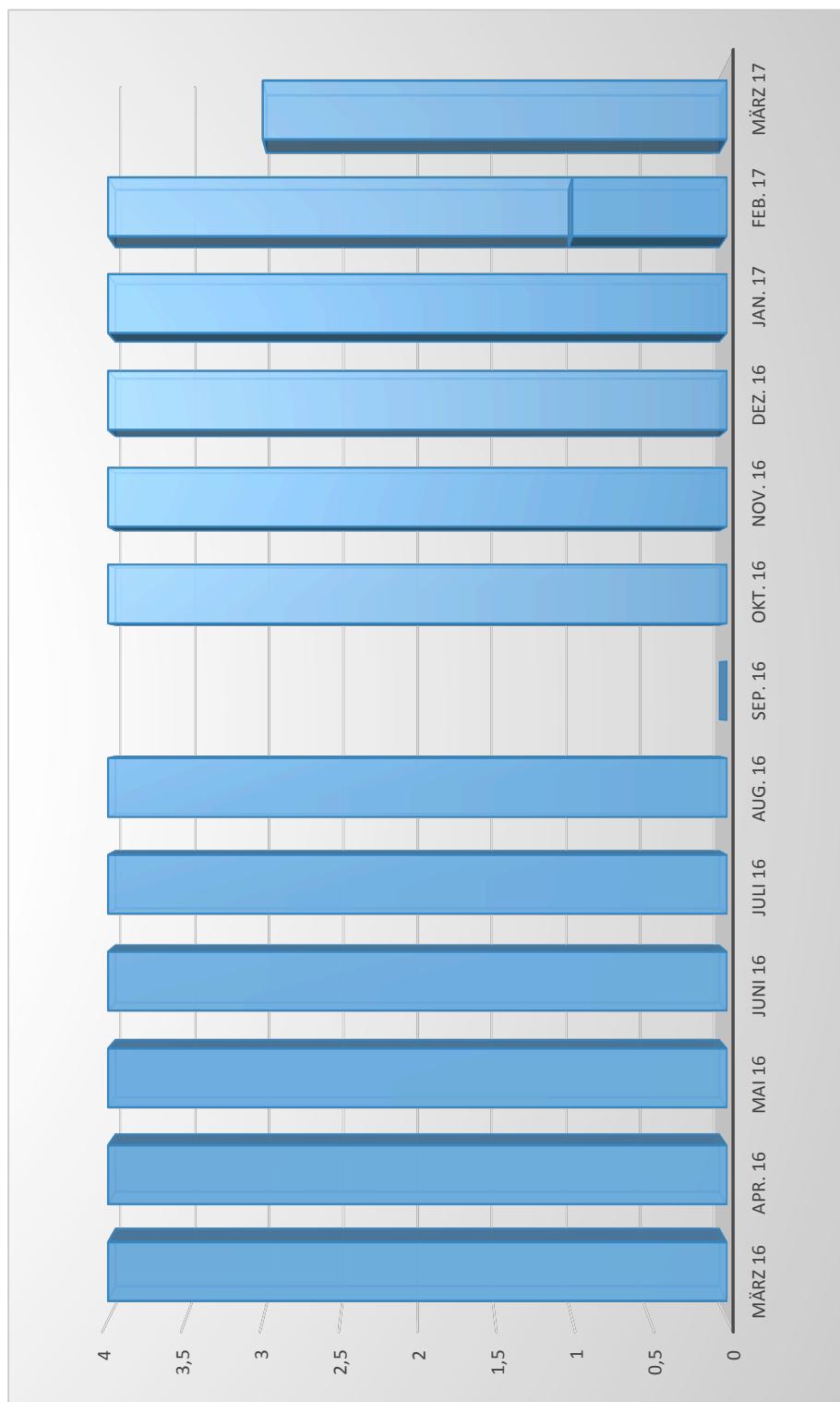


Abbildung 25: Mitarbeiterressourcenplan von MArC als Diagramm.

Kostenplan MArC		
Arbeitspakete	Kriterium	Kosten
AP1 - Organisation und Planung	Büromaterialien	< 25 EUR
AP2 - Trackingalgorithmus	-	-
AP3 - Framework	Software: Unity, Visual Studio	0 EUR, 564 EUR
AP4 - Hardware	Objektmarker, Montagematerial, HTC Vive, Leap Motion, OVR Zubehör	Vive: 899 EUR, LEAP: 279,65 EUR, OVR 415,45 EUR
AP5 - CAD Import/Export	-	-
AP6 - Testing	-	-
AP7 - Dokumentation	Druckkosten pro Seite (0,208 EUR/S)	20,8 EUR
AP8 Prototypen	-	-

Abbildung 26: Kostenplan von MArC.

Qualitätsplan MArC			
Qualitätsziel	Kriterium	Weg zur Qualitätssicherung	Qualitätskontrolle
Flüssiges Rendering	Frames pro Sekunde > 25	Darstellung an PC Leistung anpassen, wenn nicht anders Möglich, performanteren PC besorgen	Ausgabe der FPS
Flüssiges Markertracking	Frames pro Sekunde: Maximal mögliche FPS	Leistungsfähige Trackingkamera Nutzen	Ausgabe der FPS
Genaues Markertracking	Auflösung der Kamera > 1024 x 1024	Leistungsfähige Trackingkamera Nutzen	Bei fertigem Tracking: Marker langsam verschieben und rotieren
Flüssiges Handtracking	Ruckeln beim Tracking/ FPS wenn diese angegeben werden können,	Leistungsfähige Trackinghardware Nutzen	Testen der Hardware im Zusammenhang mit der Software und anderer Hardware
Genaues Handtracking	Der Nutzer soll in der Lage sein, Buttons und Menüs mittels Finger auszuwählen. Dabei soll die Fehlerrate möglichst gering sein	Leistungsfähige Trackinghardware Nutzen	Testing intern
Usability der Anwendung	Anwender sollen explorativ die Funktionen der Anwendung schnell erlernen können	Testing intern und mit Experten, anschließende Verbesserungen	Tersting intern

Abbildung 27: Qualitätsplan von MArC.

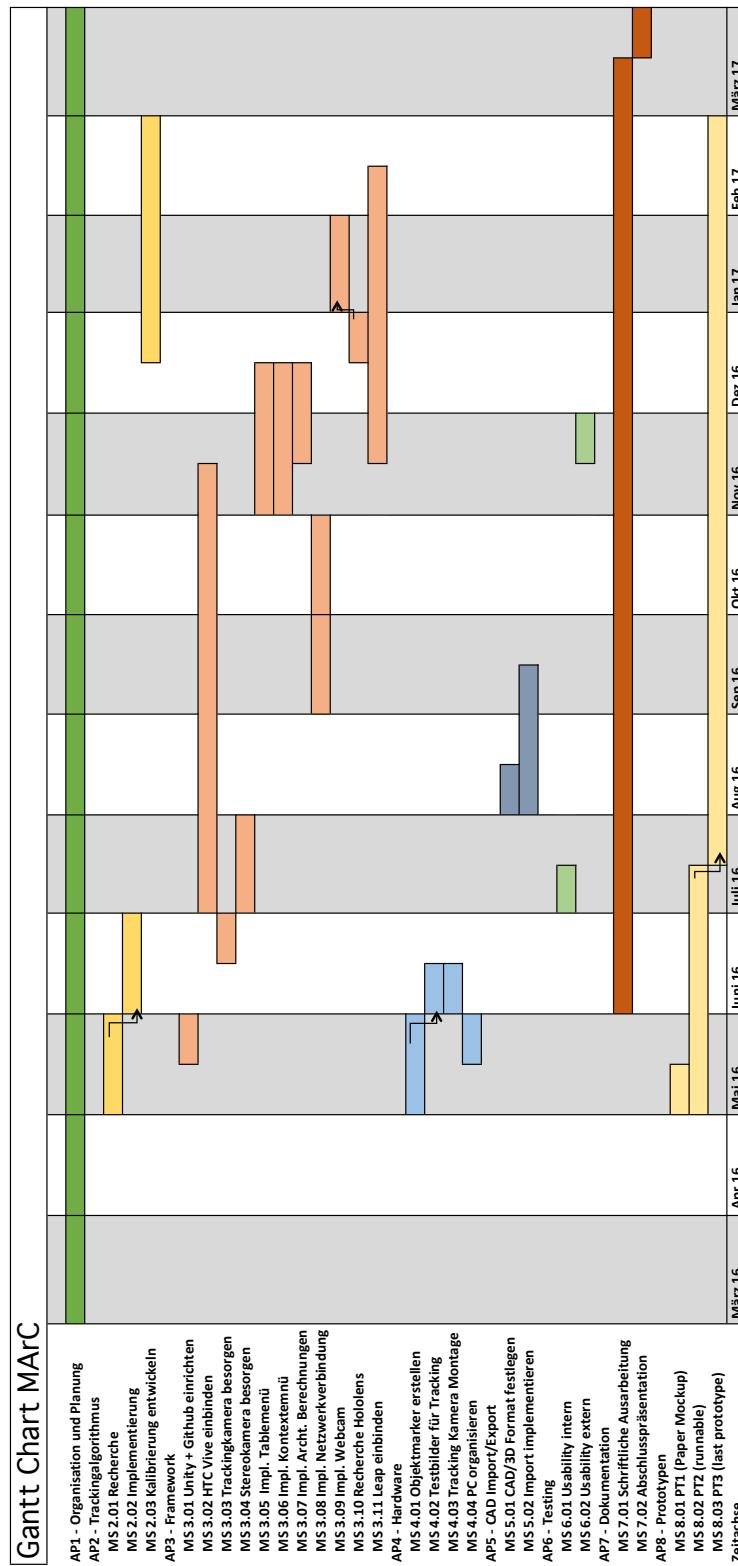


Abbildung 28: Gantt-Chart von MArC.

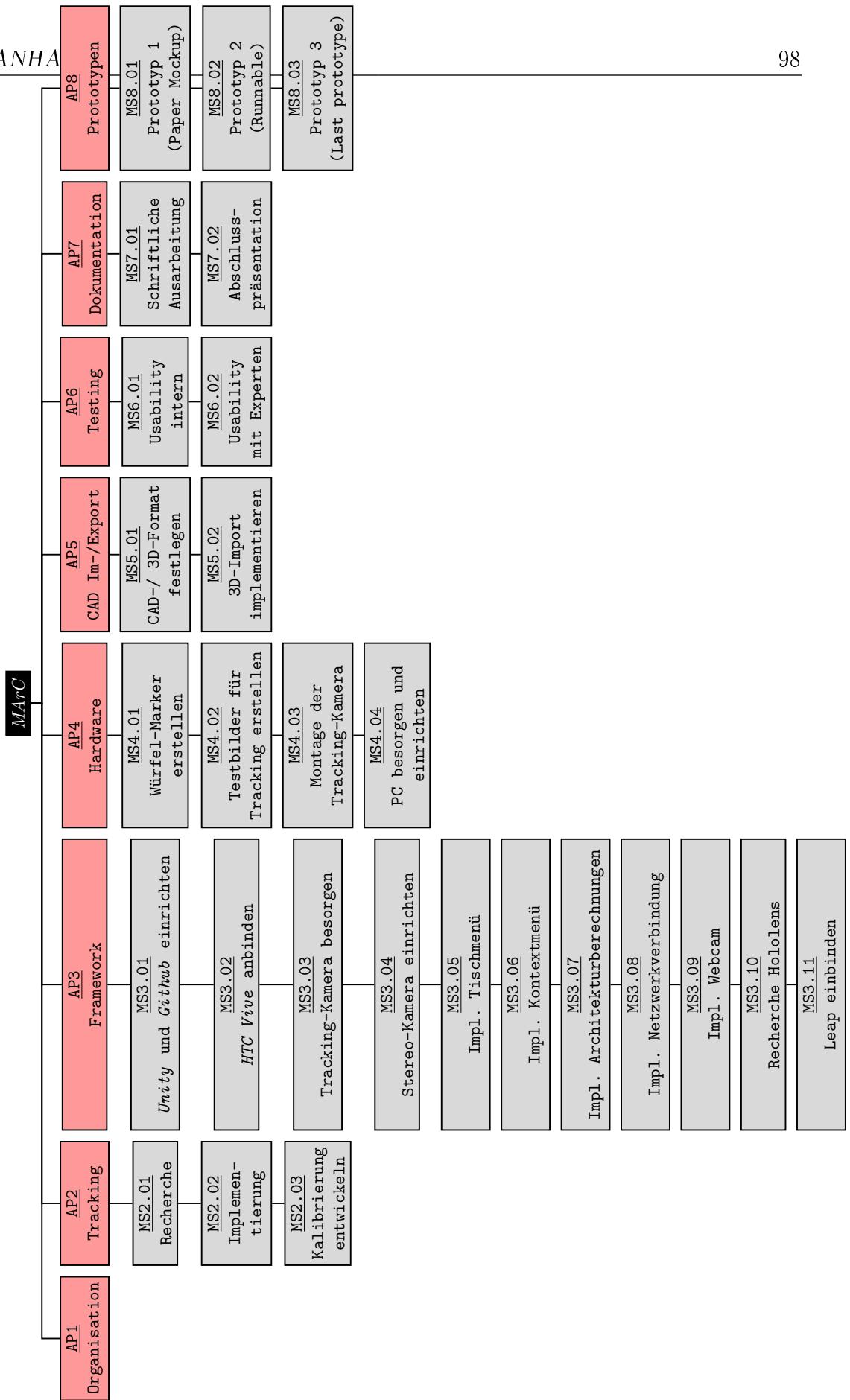


Abbildung 29: Projekt Strukturplan von MArC.

Literatur

- [1] IDS Suite. <https://de.ids-imaging.com/ids-software-suite.html>. Aufgerufen: 13. März 2017.
 - [2] uEye LE – Technical Specifications. https://www.1stvision.com/cameras/IDS/dataman/uEyeLE_Brochure_english_ND.pdf. 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. 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>, 16:16–16, 1998.
 - [7] Steve Bryson. Virtual reality applications. chapter Approaches to the Successful Design and Implementation of VR Applications, pages 3–15. Academic Press Ltd., London, UK, UK, 1995.
 - [8] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986.
 - [9] 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.
 - [10] 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.
 - [11] Creative. Creative Senz3D, Tiefen- und Gestenerkennungskamera für PCs. <http://de.creative.com/p/web-cameras/creative-senz3d>. Aufgerufen: 19. März 2017.
 - [12] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek. A brief introduction to opencv. pages 1725–1730, May 2012.
 - [13] Dima Damen, Pished Bunnun, Andrew Calway, and Walterio Mayol-cuevas. Realtime learning and detection of 3d texture-less objects: A scalable approach. 2012.
-

- [14] K. Daniilidis, C. Krauss, M. Hansen, and G. Sommer. Real-time tracking of moving objects with an active camera. *Real-Time Imaging*, 4(1):3 – 20, 1998.
 - [15] Doc-Ok.org. Lighthouse tracking examined. <http://doc-ok.org/?p=1478>. Aufgerufen: 30. März 2017.
 - [16] Ralf Dörner, Paul Broll, Wolfgang Grimm, and Bernhard Jung, editors. *Virtual und Augmented Reality (VR/AR)*, chapter 4.4, pages 114–117. eXamen.press. Springer Vieweg, Berlin, 2013.
 - [17] 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.
 - [18] Gerald Farin and Diane Hansford. *Lineare Algebra: Ein geometrischer Zugang*, chapter 8.1, pages 139–141. Springer-Verlag, 2013.
 - [19] O. Faugeras. *Three-dimensional Computer Vision: A Geometric Viewpoint*. Artificial intelligence. MIT Press, 1993.
 - [20] M. Fiala. Designing highly reliable fiducial markers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(7):1317–1324, July 2010.
 - [21] 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.
 - [22] Daniel Flohr and Jan Fischer. A lightweight id-based extension for marker tracking systems. pages 59–64, 2007.
 - [23] S. Garrido-Jurado, R. Munoz-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.
 - [24] S. Garrido-Jurado, R. Munoz-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.
 - [25] Google. Cardboard. <https://vr.google.com/cardboard/>. Aufgerufen: 20. März 2017.
 - [26] Xiaochi Gu, Yifei Zhang, Weize Sun, Yuanzhe Bian, Dao Zhou, and Per Ola Kristensson. Dexmo: An inexpensive and lightweight mechanical exoskeleton for motion capture and force feedback in vr. pages 1991–1995, 2016.
 - [27] John Haas. *A History of the Unity Game Engine*. PhD thesis, Worcester Polytechnic Institute.
 - [28] 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.
-

- [29] 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.
 - [30] HTC. Htc vive. <https://www.vive.com/>. Aufgerufen: 30. November 2016.
 - [31] HTC. HTC Vive – Für Vive geeignete Computer. <https://www.vive.com/de/ready/>. Aufgerufen: 18. März 2017.
 - [32] 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.
 - [33] H. Kato and M. Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. pages 85–94, 1999.
 - [34] Yukari Konishi, Nobuhisa Hanamitsu, Kouta Minamizawa, Ayahiko Sato, and Tetsuya Mizuguchi. Synesthesia suit: The full body immersive experience. pages 71:1–71:1, 2016.
 - [35] Leap Motion. Leap Motion. <https://www.leapmotion.com/>. Aufgerufen: 30. November 2016.
 - [36] Leap Motion. Leap Motion Blog. <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>. Aufgerufen: 2. Januar 2017.
 - [37] Leap Motion. Leap Motion SDK. <https://developer.leapmotion.com/unity#100>. Aufgerufen: 28. März 2017.
 - [38] Leap Motion. Leap Motion SDK UI Modul. <https://gallery.leapmotion.com/ui-input-module/>. Aufgerufen: 28. März 2017.
 - [39] 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.
 - [40] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
 - [41] H. Álvarez, I. Aguinaga, and D. Borro. Providing guidance for maintenance operations using automatic markerless augmented reality system. pages 181–190, Oct 2011.
 - [42] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.
-

- [43] 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.
 - [44] 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.
 - [45] Mobile World Congress. Mobile World Congress. <https://www.mobileworldcongress.com/>. Aufgerufen: 30. November 2016.
 - [46] 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.
 - [47] Oculus. Oculus – Unity Intro. <https://developer3.oculus.com/documentation/game-engines/latest/concepts/unity-intro/>. Aufgerufen: 14. März 2017.
 - [48] OpenCV. Camera Calibration and 3D Reconstruction. http://docs.opencv.org/3.1.0/d9/d0c/group__calib3d.html. Aufgerufen: 18. März 2017.
 - [49] OpenCV. Detection of ArUco Markers. http://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html. Aufgerufen: 18. März 2017.
 - [50] OpenCV. GitHub Repository von OpenCV. <https://github.com/opencv/opencv/blob/master/samples/cpp/calibration.cpp>. Aufgerufen: 26. März 2017.
 - [51] Ovrvision Pro. Informationen für Entwickler. <http://ovrvision.com/setup-en/>. Aufgerufen: 14. März 2017.
 - [52] Ovrvision Pro. Ovrvision Pro. <http://ovrvision.com/>. Aufgerufen: 30. November 2016.
 - [53] Ovrvision Pro. Produktdetails. <http://ovrvision.com/product-en/>. Aufgerufen: 9. März 2017.
 - [54] Y. Park, V. Lepetit, and W. Woo. Texture-less object tracking with online training using an rgb-d camera. pages 121–126, Oct 2011.
 - [55] J. Postel. RFC 768. User Datagram Protocol. <https://tools.ietf.org/html/rfc768>, August 1980. Aufgerufen: 21. März 2017.
 - [56] Bob Quinn. *Windows sockets network programming*. Addison-Wesley Longman Publishing Co., Inc., 1998.
 - [57] T. Rahman and N. Krouglicof. 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.
-

- [58] Kalman RE and Asme. J. A new approach to linear filtering and prediction problems. *Basic Eng*, 82(1):35–45, 1960.
- [59] Éric Marchand and François Chaumette. Feature tracking for visual servoing purposes. *Robotics and Autonomous Systems*, 52(1):53 – 70, 2005. Advances in Robot Vision.
- [60] 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.
- [61] D. Schmalstieg and T. Hollerer. *Augmented Reality: Principles and Practice*, pages 191–191. Addison-Wesley usability and HCI series. Addison Wesley Professional.
- [62] William R Sherman and Alan B Craig. *Understanding virtual reality: Interface, application, and design*. Elsevier, 2002.
- [63] Steam. Steam VR Internetauftritt. <http://store.steampowered.com/steamvr?l=german>. Aufgerufen: 26. März 2017.
- [64] Satoshi Suzuki and Keiichi Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32 – 46, 1985.
- [65] Unity Technologies. Unity. <https://unity3d.com/de>. Aufgerufen: 8. März 2017.
- [66] Unity Technologies. Unity – Multiplatform. <https://unity3d.com/unity/multiplatform>. Aufgerufen: 14. März 2017.
- [67] Unity Technologies. Unity – Public Relations. <https://unity3d.com/public-relations>. Aufgerufen: 14. März 2017.
- [68] Unity Technologies. Unity – VR Overview. <https://unity3d.com/de/learn/tutorials/topics/virtual-reality/vr-overview>. Aufgerufen: 14. März 2017.
- [69] L. Vaccagni, 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.
- [70] Valve. Valve Software. <http://www.valvesoftware.com/>. Aufgerufen: 30. November 2016.
- [71] 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.

- [72] Daniel Wagner and Dieter Schmalstieg. Artoolkitplus for pose tracking on mobile devices, 2007.
 - [73] Li-Chen Wu, I-Chen Lin, and Ming-Han Tsai. Augmented reality instruction for object assembly based on markerless tracking. pages 95–102, 2016.
 - [74] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, Nov 2000.
-