ulm university universität **u**ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Realisierung einer Markererkennungsengine für Augmented Reality Applications auf mobilen Geräten

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Fabian Schwab
fabian.schwab@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Rüdiger Pryss

2013

Fassung 7. April 2014

Satz: PDF-LaTeX 2$_\varepsilon$

Inhaltsverzeichnis

1 Einleitung

Zu Beginn werden die Grundlagen zur Augmented Reality erläutert. Es wird der Begriff der Augmented Reality erklärt, wie dieser einzuordnen ist und was Augmented Reality charakterisiert. Zudem wird die technische Seite von Augmented Reality Systemen und welche Basis-Technologien notwendig sind, um solch ein System zu bilden, betrachtet. Außerdem wird ein Einblick gegeben, wo Augmented Reality zum Einsatz kommt. Nachdem die Grundlagen erläutert sind, wird das Thema dieser Arbeit beschrieben.

1.1. Augmented Reality

1.1.1. Was ist Augmented Reality?

Augmented Reality oder kurz auch „AR“ genannt, kann mit ”erweiterter Realität” ins Deutsche übersetzt werden.

Unter Augmented Reality versteht man eine computergestützte Wahrnehmung bei der sich die reale Welt durch computergenerierte Inhalte erweitert. Hierbei werden virtuelle Objekte, die aus Texten, Grafiken oder Tönen bestehen können, in Echtzeit direkt oder indirekt über die Realität gelegt.
Die am häufigsten zitierte Definition von Ronald Azuma [1] bezeichnet Augmented Reality als Systeme, die folgende drei Eigenschaften aufweisen:

- *Combines real and virtual*
- *Interactive in real time*
- *Registered in 3-D*

1.1.2. Augmented, Virtual und Mixed Reality

Um den Begriff Augmented Reality noch besser einordnen zu können, kann die von Paul Milgram [2] erstellte Grafik ”Reality-Virtuality-Continuum” (siehe dazu auch die Abbildung 1.1) betrachtet werden.
Hierbei wird das Verhältnis von realen und virtuellen Elementen beschrieben. Auf der rechten Seite befindet sich hier die Realität (Real Environment) und am anderen Ende eine virtuelle Realität (Virtual Environment oder Virtual Reality). Der Raum dazwischen wird als vermischte Realität (Mixed Reality) bezeichnet. Wie in Abbildung 1.1 zu sehen ist, gehört die Augmented Reality zur Mixed Reality. Da sich Augmented Reality näher an der Realität befindet, leitet sich der Grad der Vermischung von virtuellen und realen Elementen ab. Das heißt, dass sich in Augmented Reality nur relativ wenig virtuelle Elemente befinden. Im Gegensatz hierzu steht das Virtual Environment, dort wird eine komplette virtuelle Welt erschaffen, die keinen Bezug mehr zu realen Welt aufweist. Selbst physikalische Grundsätze wie die Gravitation müssen hier nicht mehr gültig sein. Oft werden in Systemen der Augmented Reality und der Virtual Reality ähnliche Technologien benutzt. In der Virtual Reality wird der Benutzer von allen Sinneseindrücken der realen Welt abgeschottet. Doch gerade bei der Augmented Reality steht weiterhin die reale Welt und ihre Sinneseindrücke im Vordergrund. Die wenigen virtuellen Elemente sollen weiterhin als Teil der realen Welt erkennbar und nutzbar sein.

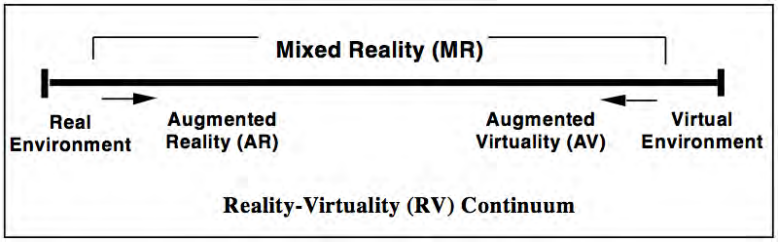

Abbildung 1.1.: Vereinfachte Darstellung des Reality-Virtuality Continuum [2].

1.2. Technologien von AR-Systemen

Die technologischen Anforderungen an die Augmented Reality sind viel höher als für die Virtual Reality. Deshalb brauchte der Bereich der Augmented Reality viel länger, um einen ähnlichen Grad der Entwicklung zu erreichen.
So unterschiedlich manche Augmented Reality Systeme seit den ersten Prototypen von Iva Sutherland [3],[4],[5] aus den 60 Jahren auch sind, so haben sie alle gewisse Technologien gemeinsam. Display- und Tracking-Technologien zählen hierbei zu den wichtigsten.

1.2.1. Display

In diesem Abschnitt werden die gängigsten Display-Technologien vorgestellt, die bei Augmented Reality Systemen eingesetzt werden, um virtuelle Elemente darzustellen.

Head-Mounted Display (HMD)

Ein HMD ist ein System, das auf dem Kopf getragen wird. Dabei ist ein kleines Display in Augennähe angebracht, oder ein Projektor projiziert ein Bild direkt auf die Netzhaut des Auges.
Beim ersten Szenario, dem augennahen Displays, gibt es zwei Prinzipien [6], [7].

- Das ”video-see-through” Prinzip
  Bei diesem Prinzip wird die reale Welt mit einer Kamera aufgenommen. Die

Aufnahme wird um die virtuellen Elemente der Augmented Reality ergänzt und anschließend auf einem Bildschirm dargestellt, siehe dazu die Abbildung 1.2.

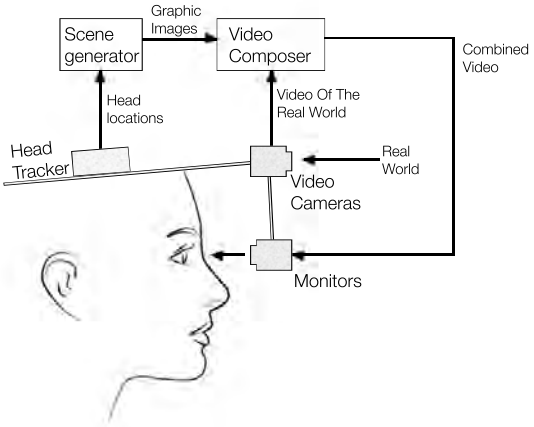

Abbildung 1.2.: video-see-through Prinzip [1]

- Das ”optical-see-through” Prinzip
  Hierbei wird, im Gegensatz zum ”video see-through” Prinzip, die Umwelt weiterhin mit dem Auge direkt wahrgenommen. Virtuelle Elemente werden auf einer halb transparenten Oberfläche, die sich vor dem Auge befindet, dargestellt. Siehe dazu die Abbildung 1.2.

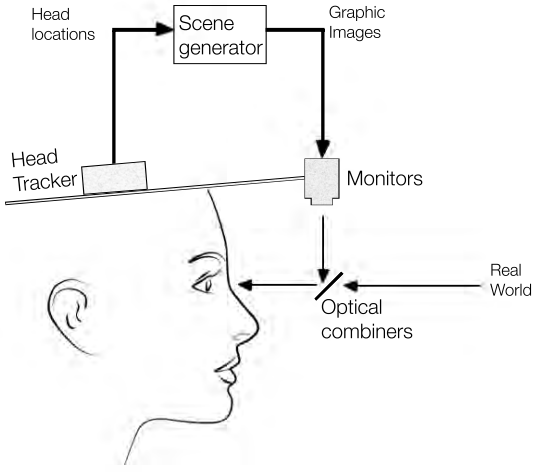

Abbildung 1.3.: optical-see-through Prinzip [1]

Eines der bekanntesten Projekte von HMDs ist das Projekt ”Google Glass” [8]. Hierbei handelt es sich um eine Brille, die Informationen in das Sichtfeld des Trägers einblendet. Ein kleiner Projektor projiziert diese auf ein halb transparentes Glassprisma. Da die Umwelt weiterhin direkt mit dem Auge wahrgenommen wird, ist dieses Projekt dem ”optical see-through” Prinzip zugeordnet.

Projektoren

Projektoren bieten in der Augmented Reality eine Vielzahl an Einsatzmöglichkeiten. Unter anderem können Informationen direkt auf Oberflächen von Objekten projiziert und so mit Informationen überlagert werden.
Diese Systeme können entweder mit fest installierten Projektoren, verbaut in Geräten wie Smartphones oder als tragbare Systeme, realisiert werden [9].

Handheld Displays

Eine andere Kategorie von Displays ist die der ”handheld” Displays. Diesen Namen tragen alle Displays, die klein und leicht genug dafür sind, dass der Benutzer sie in der Hand halten kann. Viele haben eine Kamera auf der Rückseite des Displays integriert, um mittels des ”video see-through” Prinzips, Augmented Reality zu ermöglichen. Die virtuellen Elemente werden dann zusammen mit dem Livestream der Kamera auf dem Display angezeigt.
Da viele Tablets und Smartphone diese Komponenten schon integriert haben, stehen sie verstärkt im Fokus der Entwicklung von Augmented Reality Anwendungen. Durch die verstärkte Verbreitung dieser Geräte in den letzten Jahren, erhält das ”video see-through” Prinzip immer mehr Aufmerksamkeit bei der Weiterentwicklung von Augmented Reality Frameworks.

1.2.2. Tracking-Technologien

Alle Augmented Reality Systeme müssen ihre Umwelt erfassen können. Hierzu benötigen alle Systeme Verfahren zur Erfassung der eigenen Position oder der Position eines zu beobachtenden Objekts, woraus wieder die eigene Position ermittelt werden kann. Diese Verfahren werden Tracking genannt. Das wichtigste Kriterium, welches an Tracking-Verfahren gestellt wird, ist die Performance, so dass die Interaktion in Echtzeit abläuft.
Es gibt verschiedene Arten, wie Tracking realisiert werden kann. Im Folgenden wird eine kurze Übersicht [10] gegeben, wobei der Schwerpunkt auf optischen Tracking-Verfahren liegen wird. Alle weiteren werden nur kurz angerissen.

Mechanisch

Das mechanische Tracking ist einer der frühesten Verfahren. Hierbei wird ein zu trackendes Objekt fest mit einem Gelenkarm verbunden. Dieser ist an einem festen Punkt fixiert und durch Änderungen der Winkel in den Gelenken kann so die Position des zu trackenden Objekts verfolgt werden. Die Änderung der Winkel wiederrum können optisch oder durch andere Sensoren erfasst werden. Mit diesem Verfahren lässt sich die Position eines Objektes sehr exakt und sehr schnell ermitteln. Doch sind bei dieser Technologie, bedingt durch die mechanischen Teile, Einschränkungen in der Mobilität und in den Freiheitsgraden hinzunehmen.

Trilateration und Multilateration

Die Funktionsweise basiert darauf, dass Laufzeiten von Signalen gemessen werden und daraus eine Entfernung errechnet wird. Eine eindeutige Bestimmung kann erst gemacht werden, wenn Entfernungen zu drei bekannten Punkten vorhanden sind. Hier spricht man von Trilateration [11]. Sind mehr als drei Entfernungen zur Positionbestimmung bekannt, spricht man von Multilateration [11].
Eines der hier bekanntesten Beispiele zur Bestimmung einer Position ist das ”Global

Positioning System[1]" kurz GPS [12].
GPS stützt sich dabei auf Satelliten, die kontinuierlich ihre Position und eine genaue Uhrzeit senden. GPS-Empfänger können so durch die Laufzeiten der Signale ihren eigenen Standort ermitteln. Da aber die meisten Empfängern keine Uhr besitzen, die genau genug wäre, wird ein vierter Satellit benötigt, um die genaue Zeit auch im Empfängergerät festzustellen.
Auch kann dieses Verfahren auf W-LAN[2] und Mobilfunknetze übertragen werden.

Magnetisch

Auch über das Erdmagnetfeld ist eine Bestimmung der Position möglich. Die Störanfälligkeit dieser Methode macht es jedoch schwer, genaue und zuverlässige Daten zu erhalten.
Andere Verfahren benutzen ein künstliches Magnetfeld zur Positionsbestimmung, womit sich Position und Orientierung schnell und exakt ermitteln lassen.

Optisch

Sehr verbreitet bei Augmented Reality sind die optischen Tracking-Verfahren [13]. Hierbei werden meist digitale Kameras benutzt, die das für den Menschen sichtbare oder infrarotes Licht erfassen. Voraussetzung bei diesen Verfahren ist, dass die Kamera eine direkte Sicht auf das zu erfassende Objekt haben muss.
Jedes Einzelbild, welches die Kamera dabei liefert, wird mit Algorithmen aus der Bildverarbeitung analysiert [13]. Hierbei werden die Bilder entweder nach optischem Markern oder nach so genannten Features durchsucht. In den folgenden Abschnitten wird genauer auf die verschiedenen Verfahren eingegangen.

- optische Marker
  Es gibt verschiedenste Arten von Markern, aber allen ist gemeinsam, dass sie an

[1] Die offizielle Bezeichnung ist "Navigational Satellite Timing and Ranging – Global Positioning System" (NAVSTAR GPS)
[2] Wireless Local Area Network

den zu trackenden Objekten angebracht werden müssen. Grundsätzlich gibt es zwei Arten von Markern, aktive und passive.

- Bei den passiven Markern ist die Grundform sehr oft ein Viereck mit weißem Hintergrund und schwarzem Rand. Darauf befindet sich ein schwarzes Muster, welches zur eindeutigen Erkennung nötig ist. Durch die Beschränkung auf schwarz-weiß wird der höchst mögliche Kontrast erreicht, der von Bedeutung ist, wenn es um die Bildbearbeitung durch Algorithmen geht. Da diese Marker stark vom Umgebungslicht abhängig sind, gibt es passive Marker, die aus sogenanntem retroreflexivem Material bestehen. Dieses Material wirft einen sehr großen Teil des einfallenden Lichts in Richtung der Lichtquelle zurück. Für den Fall, dass ein Augmented Reality System dauerhaft bei schlechten Lichtverhältnissen arbeiten muss, könnte dieses System mit aktiven Lichtquellen (sichtbares oder infrarotes Licht) ausgestattet werden, und so in Verbindung mit retroreflexiven passiven Markern [13] eine zuverlässige Erkennung bieten. Ein großer Vorteil, den die passiven Marker bieten, ist, dass sie an einem herkömmlichen Drucker ausgedruckt und sofort eingesetzt werden können.

- Aktive Marker benötigen kein Licht aus der Umgebung, sondern senden selbst Lichtsignale aus. Diese können im infraroten oder sichtbaren Bereich des Lichts liegen. Der Nachteil bei aktiven Markern ist, dass diese eine zusätzliche Stromversorgung brauchen, und durch den Einsatz von LED[3] und weiterer zusätzlicher Elektronik in der Herstellung teurer sind als die passiven Marker.

- Features
  Feature-Tracking wird auch als ”Markerless-Tracking” bezeichnet, da hier keine Marker benutzt werden. Bei solch einem markerlosen System müssen natürliche Eigenschaften einer bestimmten Szene aus dem Videostream der Kamera erkannt und verfolgt werden. Diese Features müssen zuvor schon bekannt sein, da sonst kein Angleich mit dem Livestream der Kamera erfolgen kann.

[3] Licht-emittierende Diode

Inertial

Eine weitere Methode zum Tracking der Orientierung und Ausrichtung eines Gerätes, ist das so genannte *Inertiale Tracking*. Hierbei werden interne Sensoren, die meist aus Beschleunigungssensoren und Gyroskopen bestehen [14], verwendet. Diese Sensoren erfassen die Kräfte, welche durch die Massenträgheit bedingt auf sie einwirken. Ein Gerät, das diese Komponenten vereint, wird als ”Initial Measurement Unit”, kurz IMU bezeichnet. Diese Einheit beinhaltet unter anderem drei Beschleunigungssensoren, die so angebracht sind, dass sie alle drei Dimensionen abdecken. Ebenfalls sind drei Gyroskope verbaut, in manchen findet sich zudem noch ein Magnetfeldsensor. Diese Einheiten wird als zentrale Komponente in der Luft- und Raumfahrt genutzt, um die Navigation zu unterstützen.

1.3. Smartphones als Augmented Reality Systeme

Smartphones haben viele Eigenschaften, die ein Augmented Reality System benötigt [15], schon integriert. In diesem Abschnitt wird erklärt, welche Vorzüge Smartphones als Augmented Reality Systeme haben, und wie sie ihr zum Durchbruch verholfen haben [16].
Heutige Smartphones sind mit vielen zusätzlichen Komponenten ausgestattet. Einige dieser Komponenten sind nicht nur zwingend für ein Smartphone notwendig, sondern auch für Augmented Reality Systeme. Unter anderen haben Smartphones folgende Komponenten:

- Display
- Videocamera
- Inertialsensoren
- GPS bzw. WPS[4]
- Multicore Prozessoren

[4] WLAN-basierte Ortung

- leistungsfähige Grafikunterstützung
- Lichtsensoren
- Näherungssensoren
- Mikrophone

Display

Viele Smartphones besitzen Bildschirmauflösungen, die mit denen von Laptops verglichen werden können, oder übertreffen diese sogar. Dabei sind die Displays so kompakt, dass sie noch gut mit einer Hand zu halten sind. Damit fallen sie in die Klasse der *handheld* Displays. Auch, dass ein beachtlicher Anteil der vorderen Oberfläche der Smartphones aus dem Display besteht, ist ein Vorteil. So wird der Benutzer nicht durch einen großen Rahmen um das Display gestört.

Videocamera

Nicht nur das Display, sondern auch eine oder mehrere Kameras sind auf der Rück- und Vorderseite von Smartphones fest integriert. Damit eignet sich die rückwärtige Kamera in Verbindung mit dem großzügigen Display als *see-through-system* mit optischem Tracking-Verfahren.
Die Kameras unterstützen dabei Aufnahmeauflösungen im oberen einstelligen bis zu zweistelligem Megapixelbereich. Damit ist die Bildqualität mancher Modelle schon mit der von Kompaktkameras zu vergleichen.
Aber nicht nur eine hohe Auflösung ist für optisches Tracking wichtig. Weitere wichtige Eigenschaften, die Kameras für optisches Tracking benötigen, und die die Qualität des Tracking maßgeblich beeinflussen, sind:

- ein schneller Auto-Fokus
- schnelle Blendautomatik
- hohe Wiederholungsfrequenz bei der Bildaufnahme

- verzerrungfreie Bilder an Randbereichen

Die integrierten Kameras bieten gerade in den oben genannten Punkten noch einige Schwachstellen. Gerade Bewegungsunschärfe bei schnellen Bewegungen ist hierbei ein großes Problem für optische Trackingverfahren. Momentan werden ständig neue und leistungsfähigere Kameras verbaut, sodass dieses Problem immer weiter abnimmt.

Inertialsensoren

Smartphones beinhalten Beschleunigungssensoren für X-, Y- und Z-Achsen, sowie ein Gyroskop. Mit all diesen Informationen zu Bewegung und Orientierung des Smartphones können "6-DoF[5]-Tracking" Verfahren angewendet werden. Auch können dadurch weitere Verbesserungen im optischen Tracking erzielt werden. Hierzu später mehr in Abschnitt 3.7.

GPS und WPS

Viele Augmented Reality Systeme und Anwendungen beziehen sich auf ortsbezogene Informationen. Diese Standortinformationen werden über die GPS oder WPS Module das Smartphones bezogen. GPS und WPS Systeme sind in der Lage die Position des Gerätes auf wenige Meter genau zu bestimmen.
Bei Wireless Positioning Systems werden Mobilfunkmasten und Wireless LAN Netze zur Ortsbestimmung herangezogen. Es wird festgestellt, welche Funknetze sich in der unmittelbaren Umgebung befinden und durch die Empfangsstärke die ungefähre Entfernung zu den Emittern berechnet. WPS Dienste von Anbietern wie Apple, Google [17] oder Skyhook [18], die Informationen über viele Mobilfunk und Wireless LAN Netze haben, errechnen anhand dieser Informationen die Position des Gerätes. Grundvoraussetzung ist aber, dass die Wireless LAN und Mobilfunk Module aktiviert sind und eine Internetverbindung zu diesen Diensten besteht.

[5] en. Degrees of Freedom - Freiheitsgrade

Multicore Prozessoren

Sämtliche Sensor- und Bilddaten müssen für Augmented Reality in Echtzeit verarbeitet werden. Dazu werden leistungsstarke Prozessoren benötigt. Viele Smartphones bieten auch hier schon ausreichend Leistung, um akzeptable Ergebnisse erzielen zu können. Dennoch ist der Rechenaufwand enorm und wird manchmal mit Tricks wie einer geringeren Kameraauflösung reduziert.

Grafikleistung

Auch das Berechnen von 3D Grafiken oder anderen virtuellen Elementen, die auf dem Display des Smartphones dargestellt werden müssen, ist aufwändig. Viele Smartphones bieten daher eine hardwareseitige Unterstützung zur Grafikberechnung an. In vielen Fällen ist dies eine Implementierung der OpenGL ES Spezifikation [19].

Weitere Sensoren

Eine enorme Menge an zusätzlichen Sensoren ist ebenfalls in Smartphones integriert. Mit diesen Informationen kann man Augmented Reality Systeme zusätzlich unterstützen. Alle diese Sensoren und die kompakte Bauform machen Smartphones zu einer ausgezeichneten Plattform für Mobile Augmented Reality Systeme.
Augmented Reality hat viele Anwendungsgebiete. Im Folgenden werden ausschließlich Anwendungsgebiete der Mobile Augmented Reality betrachtet.

1.3.1. Mobile Augmented Reality

Durch die in den letzten Jahren anhaltend starke Entwicklung im Smartphonesektor konnte auch die Mobile Augmented Reality - kurz MAR - profitieren. Der rasante technologische Fortschritt bringt immer leistungsfähigere Geräte auf den Mark. Die Geräte werden nicht nur immer dünner, sie beherbergen auch immer neue und bessere Technik als im Modell zuvor.

Beispielsweise nimmt die Rechenleistung ständig zu und Mehrkernprozessoren sind in fast jedem neuen Gerät, das das Fließband verlässt, verbaut. Bei höchauflösenden Displays und Kameras findet sich fast keine Grenze, und jedes neue Gerät bietet noch höhere Auflösungen und Pixeldichten. Mehrere Gigabyte an Speicher sind ebenfalls schon zum Standard geworden und die Vielfalt an Sensoren nimmt ständig zu. So lassen sich in fast jedem Gerät GPS Module, Beschleunigungssensoren und Gyrospoke finden. Der schnellen und breiten Akzeptanz dieser neuen Generation von Smatphones ist es auch zu verdanken, dass die Betriebssysteme immer effektiver und sicherer werden. Diese Entwicklung erlaubt es auch vielen Programmierern, immer komplexere Anwendungen zu entwickeln, welche die Hardware immer effizienter nutzt. Dadurch besteht die Möglichkeit, immer komplexere Berechnungen durchführen zu können.
Auch Augmented Reality profitierte in diesem Bereich. Da die Bildbearbeitung, die für Augmented Reality benötigt wird, sehr rechenintensiv ist, konnten bis vor ein paar Jahren kaum Augmented Reality Anwedungen für mobile Geräte geschrieben werden. Diese Entwicklung ermöglicht es, virtuelle Objekte auf neueren Geräten so detailgetreu darzustellen, als wären diese real. Es lassen sich also in der Mobile Augmented Reality Anwendungen schreiben, die den Eindruck hinterlassen, dass die reale und die virtuelle Welt immer mehr miteinander verschmelzen.

1.3.2. Anwendungen der MAR im Prozessmanagement

Ein mögliches Einsatzgebiet für eine Markererkennung aus größerer Distanz könnte der in Abbildung 1.4 abgebildete Prozess sein. Dieser Prozess spiegelt eine Röntgenanforderung wider, wobei die Röntgeneinheit mit einer Markererkennung ausgestatten sein könnte. Hierbei würde am Patientenbett ein Marker angebracht sein. Dieser Marker kann, verknüpft mit einer Datenbank, Informationen über den aktuellen Prozessschritt enthalten. Ein mobiles Gerät mit Markererkennug erkennt dann, dass ein bestimmter Patient eingetroffen ist und gibt das an die Prozessengine weiter. Somit würde das Prozessereignis ”Röntgenanfrage erhalten” automatisch, nach Eintreffen des Patienten, beginnen.

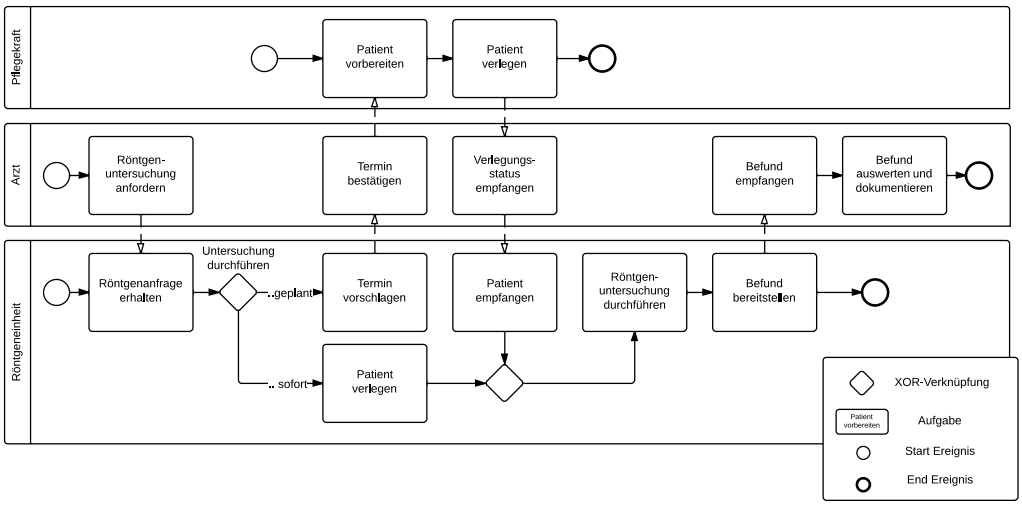

Abbildung 1.4.: Prozessmodell, welches mit Marker ergänzt werden kann [20], [21].

1.4. Thema dieser Arbeit

Diese Arbeit beschreibt einen Ansatz, wie ein Framework für Mobile Augmented Reality aussehen könnte. Die Schwerpunkte hierbei liegen auf einer stabilen Echtzeiterkennung für optische Marker. Das Framework soll, wie im nächsten Kapitel erläutert wird, Nachteile von gängigen Frameworks verbessern, oder im besten Fall gänzlich aufheben.
Ein Aspekt, der hierbei zum Tragen kommt, ist die Distanz, über die Marker erkannt werden sollen. Die Distanz sollte sich im Idealfall auf mehrere Meter belaufen, ohne die Größe der gängigen Marker zu ändern. Des Weiteren sollten die Marker mit möglichst wenig Rechenaufwand während des Tracking verfolgt werden können. Konzipiert werden soll das Framework für das mobile Betriebssystem iOS von Apple Inc.

2

Konzeption eines Markererkennungsframeworks für Mobile Augmented Reality

Im folgenden Kapitel werden zunächst die Eigenschaften guter optischer Marker aufgezeigt und analysiert. Eine kurze Übersicht über existierende Frameworks für Mobile Augmented Reality soll den Einstieg und das Verständis für die Thematik der Markererkennung erleichtern. Es werden gängige Verfahren zur Makerdetektion vorgestellt, sowie die Algorithmen, auf denen sie beruhen.
Die Stärken und Schwächen verschiedener Verfahren und einzelner unterschiedlicher Schritte werden dabei ebenfalls eine Rolle spielen. Basierend auf dem Verständnis dieser Vor- und Nachteile soll die Konzeption des eigenen Frameworks Gestalt annehmen.

2.1. Eigenschaften von optischen Markern

Bevor die Entwicklung von Verfahren optische Marker und deren Entwicklung beschrieben werden kann, stellt sich zuerst die Frage, welche Eigenschaften ein Marker zwingend besitzt, und welche einen optimalen Marker auszeichnen.
Grundsätzlich sollte man einen Marker gut aus allen Richtungen erkennen, egal wie der Blickwinkel der Kamera ist. Natürlich sind hier Grenzen gesetzt, denen man auch als Mensch unterliegt. Ist der Betrachtungswinkel zu flach, so erkennt man einen Marker nicht mehr. Um den Marker in seinem Umfeld schnell zu finden, sollte sich dieser auch stark von der realen Welt abheben. Dieses Herausstechen aus der Umgebung erleichtert es auch dem menschlichen Benutzer, das Mobile Augmented Reality System zur erstmaligen Findung eines Markers auszurichten. Nicht nur bei einem, sondern auch bei mehreren Markern erkennen mögliche Benutzer schnell, wann Augmented Reality zum Einsatz kommt und wohin das System zu richten ist. Dies ist ein Vorteil gegenüber dem Feature-Basierten-Tracking, bei dem der Benutzer nicht genau weiß, wo genau virtuelle Informationen oder Prozessaktionen zu finden sind.
Wie der optimale Marker auszusehen hat, wird in ”What is the best fiducial?” [22] beschrieben. Demnach sollten folgende Kriterien erfüllt sein:

- Form:
  Bestimmung von Position und Ausrichtung eines Markers relativ zu einer Kamera erfordert mindestens vier eindeutig unterscheidbare Punkte. Hierbei eignet sich ein Quadrat am besten.

- Farbe:
  Farbige Marker sind bei verschiedenen Kamerasystemen und unterschiedlichen Lichtverhältnissen nicht immer gleich, weshalb sich monochorme Marker am besten eignen. Ein weiterer Vorteil ist, dass sie einen hohen Kontrast bieten.

- Aufbau:
  Um den Marker in der realen Welt bestmöglich zu erkennen, sollte er einen weißen Hintergrund besitzen, und einen schwarzen breiten Rand, der 15% des gesamten

Bildes ausmacht. Somit ist die Identifikation gut möglich, und gleichzeitig der innere Bereich für ein eindeutiges Muster groß genug.

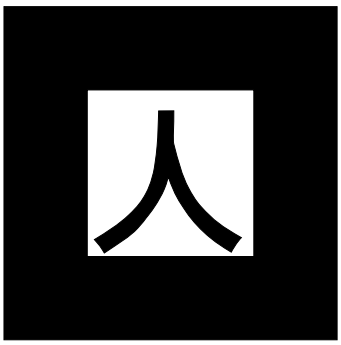

Abbildung 2.1.: Beispielmarker des ARToolKit [23].

2.2. Trackingverfahren für Marker

Jedes Augmented Reality System implementiert ein Verfahren zum Markertracking, um die Position des Objekts bzw. vom System selbst zu ermitteln. Dieser Abschnitt konzentriert sich auf die Schritte, die ein Kamerabild durchlaufen muss, bis letztendlich der Marker gefunden wurde, seine Position im Raum errechnet, und er genau identifiziert wurde. Der Ablauf, wie grundsätzlich eine Markererkennung funktioniert, richtet sich nach dem Prinzip des ARToolKit [23], welches bereits eine Markererkennung implementiert. Dort wird der Ablauf in sechs Schritten beschrieben, wie in Abbildung 2.2 zu sehen ist.

1. Original image
2. Threshold image
3. Connected components
4. Contours
5. Extract marker edges and corners
6. Fitted square

Im folgenden Absatz werden, basierend auf diesem Ablauf, Algorithmen vorgestellt, die diese Aufgaben erfüllen können.

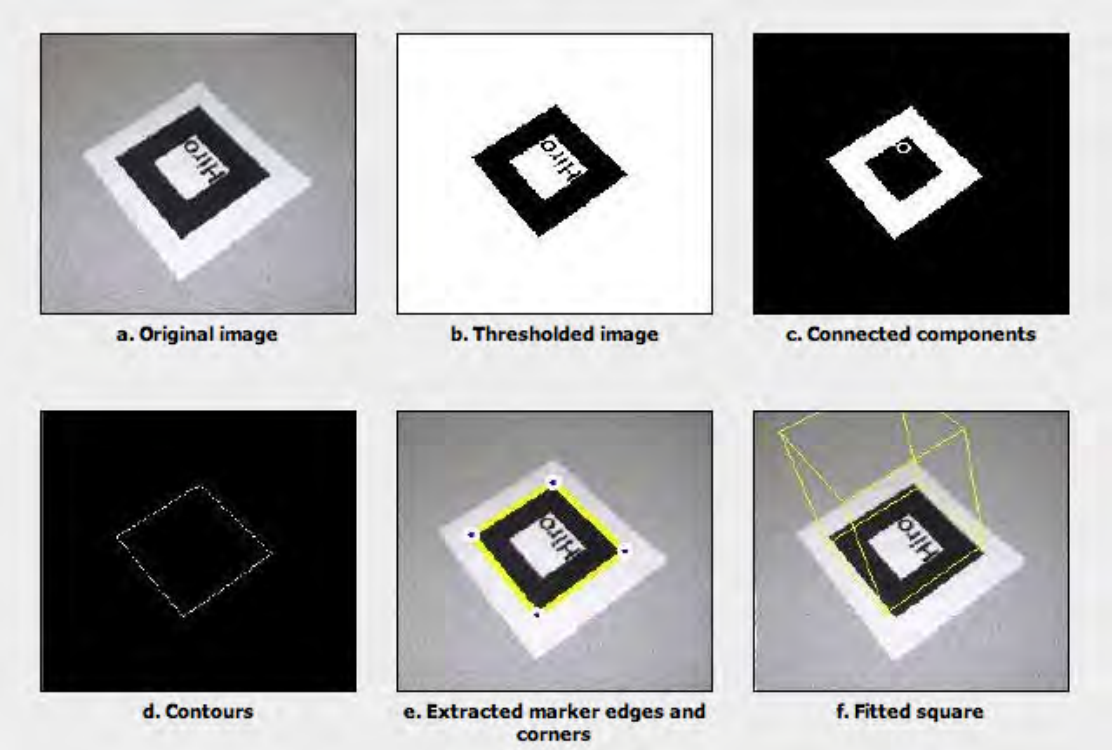

Abbildung 2.2.: Einzelschritte in der Markererkennung [23].

2.2.1. Erkennung von Markern

Das erste Ziel der Erkennung einer Markierung besteht darin, die Umrisse der potentiellen Marker zu finden. Das grundlegende Verfahren besteht aus den folgenden Schritten:

1. Graustufenbild
   Es gibt einige Verfahren, um aus einem Bild ein Graustufenbild (im folgenden GS genannt) zu erzeugen. Die gängigsten Methoden sind:
   - *Luminosity Methode*
     Der erste Algorithmus nutzt die Tatsache, dass das menschliche Auge Rot-,

Blau- und Grüntöne nicht gleich differenziert. Dies ist evolutionsbiologisch begründet, da viel von der natürlichen Umwelt grün ist, so dass die Menschen eine höhere Empfindlichkeit für grünes Licht entwickelt haben. Der auf dieser Tatsache aufbauende Algorithmus, welcher sich das zunutze macht, wird als "Luminosity Methode" bezeichnet. Dabei werden die einzelnen Farbkanäle mit verschiedenen Koeffizienten multipliziert. Bei diesen Koeffizienten gibt es verschiedene Werte, je nachdem welcher Standard benutzt wird. Zum Beispiel verwenden gängige Bildbearbeitungsprogramme wie Photoshop oder GIMP [24] ebenfalls die erste Formel, während die ITU-R [25] zwischen B.709 und B.601 unterscheidet.
GS = (Red * 0.3 + Green * 0.59 + Blue * 0.11)
GS = (Red * 0.299 + Green * 0.587 + Blue * 0.114)
GS = (Red * 0.2126 + Green * 0.7152 + Blue * 0.0722)

- *Desaturation Methode*
  Die Entsättigen eines Bildes funktioniert durch die Umwandlung des RGB-Tripel zu einem HSL[1]-Tripel, im Anschluss setzt man die Sättigung auf Null. Grundsätzlich nimmt diese Methode eine Farbe und wandelt sie in die Farbe um, die am gesättigten ist. Diese Umrechnungen sind mit einem hohen Aufwand verbunden, weshalb man eine kürzere Variante, nämlich das Auffinden der Mitte zwischen dem Maximum von (R, G, B) und dem Minimum (R, G, B), verwendet:
  GS = (Max(Red, Green, Blue) + Min(Red, Green, Blue)) / 2

- *Average Methode*
  Hierbei werden die Farbwerte der Farbkanäle eines Pixel addiert und durch drei geteilt. Das ist die am meist verwendete Graustufenkonvertierungsroutine, da sie schnell und einfach zu implementieren ist und kaum Rechenzeit in Anspruch nimmt.
  GS = (R + G + B) / 3

[1] en. Hue, Saturation, Luminance - dt. HSV-Farbraum

2. Schwarz-Weiß Bild
   In diesem Schritt muss entschieden werden, ob ein Pixel schwarz oder weiß wird. Die Art der Entscheidung ist von großer Bedeutung für das resultierende Bild. So kann man etwa definieren, dass alle Werte eines Pixels unter einem bestimmten Schwellwert schwarz, und alle darüber weiß sind. Da ein Pixel den Farbwert zwischen 0 und 255 annehmen kann, könnte man so einen Schwellwert von 128 definieren. Dieser würde genau in der Mitte liegen, aber nicht berücksichtigen, ob ein Bild zu hell oder zu dunkel aufgenommen wurde. Daher ist ein fester, globaler Wert nicht das geeignete Mittel, sondern ein dynamischer Wert erforderlich. Bekannte Methoden hierfür sind ”Adaptive Threshold” [26] und die ”Otsu Method” [27].

   - *Adaptive Threshold*
     Adaptive Schwellwertalgorithmen haben typischerweise ein Graustufenbild als Eingabeparameter, und liefern im einfachsten Fall ein binäres, also ein schwarz-weiß Bild zurück.
     Für jeden Pixel im Bild muss hier ein Schwellwert berechnet werden. Ist dann der Pixelwert unterhalb des Schwellwertes, nimmt er den Wert 0 (schwarz), ist der darüber, den Wert 255 (weiß) an.
     Um diese Schwelle zu errechnen, gibt es zwei wesentliche Ansätze [28]:

     – Chow und Kaneko Ansatz

     – lokale Schwellwerte

     Der Grundgedanke hinter den beiden Methoden ist, dass kleinere Bildbereiche eher annähernd eine gleichmäßige Ausleuchtung haben sollen, da es so besser geeignet ist für eine Schwellwertbildung.
     Chow und Kaneko teilen ein Bild in einer Reihe von überlappenden Teilbilder und finden dann den optimalen Schwellenwert für jedes Teilbild, indem sie die einzelnen Histogramme untersuchen. Die Schwelle für jeden einzelnen Pixel wird durch Interpolieren der Ergebnisse der Teilbilder gefunden.
     Ein alternativer Ansatz, lokale Schwellenwerte zu finden, ist die Untersuchung der Intensitätswerte von Nachbarpixel eines jeden Pixels. Die Statistik, die

am besten geeignet ist, hängt weitgehend von dem Eingangsbild ab. Einfache und schnelle Funktionen umfassen den Mittelwert der örtlichen Intensitätsverteilung, dem Median oder den Mittelwert der Minimal- und Maximalwerte.
Die Größe der benachbarten Region muss groß genug sein, um ausreichend Vorder- und Hintergrund Pixel abzudecken, da sonst eine falsche Schwelle gewählt wird. Werden die Regionen zu groß gewählt, sind diese nicht mehr gleichmäßig ausgeleuchtet. Diese Methode ist weniger rechenintensiv als der Ansatz von Chow und Kaneko und produziert dennoch gute Ergebnisse für einige Anwendungen.

- *Otsu Methode*
  Diese Methode bedient sich der Statistik, um einen möglichst guten Schwellenwert zu finden. Hierbei spielt das Maß der Streuung von Grauwerten, die sogenannte Varianz (Abweichung von jedem Einzelwert von Mittelwert) eine große Rolle. Die Werte des Histogramms werden als Wahscheinlichkeiten einer diskreten Zufallsvariablen gesehen. Nach einer Varianzanalyse wird so ein Grauwert bestimmt, welcher das Histogramm in zwei zusammenhängende Teile separieren lässt. Jeder Pixel wird dann mit dem errechneten Schwellwert überprüft, und dann entsprechend auf schwarz oder weiß gesetzt.

3. Bestimmung von geschlossenen Bereichen
   Auch hier gibt es viele Möglichkeiten. Die einfachste Methode wäre die weißen Bildbereiche, die zusammenhängen, zu extrahieren. Die restlichen Bildinformationen, die nicht benötigt werden, fallen so weg. Beispielsweise sind zusammenhängende Flächen, die den Sichtbereich des Systems, also auch den der Kamera verlassen, nicht interessant. Selbst wenn hier bereits Teile eines Markers zu sehen wäre, wäre es nicht möglich, anhand der wenigen Informationen, auf den gesamten Marker zu schließen.

4. Erkennung von Konturen
   Nachdem nur noch wichtigen Informationen im Bild enthalten sind, und Rauschen und andere störende Elemente verschwunden sind, kann nun ein Algorithmus genutzt werden, um verbleibende Kanten im Bild zu finden. Einige interessante und

wahrscheinlich auch die bekanntesten Verfahren hierzu sind in [29] beschrieben. Es gibt von den beschriebenen Verfahren auch verschiedenste Umsetzungen in der Computergrafik zur Bildbearbeitung. Die am häufigsten anzutreffenden Verfahren sind:

- Marr-Hildreth Edge Detector [30]
- Canny Edge Detector [31]
- Boolean function based Edge Detector [32]
- Euclidian distance and Vector Angle based Edge Detector [33]
- Depth Edge Detection using Multi-Flash Imaging [34]
- Sobel Edge Detector [35]

Die gebräuchlichsten sind jedoch Canny Edge Detector und Sobel Edge Detector.

5. Finden der Markerecken
   Auf dem nun aktuellen Bild sind idealerweise nur noch Kanten des Markers sichtbar. Nun wird mit einem weiteren Algorithmus versucht, ein Objekt zu finden, welches einem Quadrat ähnelt, was der Grundform unseres Markers entspricht. Mithilfe des Douglas-Peucker-Algorithmus werden die Ecken des Rechtecks gefunden [36]. Die noch verbleibenden Kanten im Bild werden nun vom Douglas-Peucker-Algorithmus geglättet. Ein Menge von Punkten werden auf den Kanten gewählt und durch Auslassen einiger dieser Punkte wird versucht, eine möglichst geradlinige Form des Objekts zu finden. Da es sich um die Grundform des Markers handelt, entsteht so ein Quadradt, auf welchem nur noch die Eckpunkte zu finden sind.
6. Berechung der Position
   Mithilfe der Koordinaten der Eckpunkte des Markers, lässt sich nun die Position relativ zum Augmented Reality System errechnen [6]. Um zusätzlich die Drehung des Markers im Raum bestimmen zu können, bedient man sich der inneren Markierungen, die zur Identifizierung der Marker gehört. Mit der Hough-Transformation, ein robustes Verfahren zur Erkennung von Geraden, Kreisen, Quadraten oder anderen parametrisierbaren geometrischen Figuren, kann so nun auch die Drehung bestimmt werden [37].

2.3. Existierende AR SDKs für iOS

Es gibt eine große Anzahl an bereits existierenden SDKs für die iOS Plattform. Viele davon sind zum Teil frei nutzbar, benötigen aber zum Veröffentlichen von Anwendungen eine gültige Lizenz. Ebenfalls können weitere Zusatzfunktionen gegen Lizenzgebühren erstanden werden. Wenige Projekte sind ”Open Source” Projekte, welche im vollem Umfang genützt werden können.
Im Folgenden werden einige der bereits existierenden SDKs für Augmented Reality aufgelistet und verglichen. Im Vergleich dazu stehen die unterstützten Tracking-Technologien wie GPS, Marker- und Featurebasiertes-Tracking sowie die Unterstützung für die Inertialsensoren, wie in Abschnitt 1.3 beschrieben. Die in Tabelle 2.3 aufgeführten SDKs sind für iOS, und zum Teil auch andere Betriebssysteme, ausgelegt, und enthalten nur SDKs, die auch Marker-Basiertes-Tracking erlauben.
Wie aus Tabelle 2.3 zu entnehmen ist, unterstützen nur wenige SDKs alle Eigenschaften. Die wenigen wie Wikitude, Metaio SDK und ARmedia sind kommerzielle Systeme. Bei kommerziellen und teilweise kommerziellen Systemen an Informatinen zu gelangen, mit welchen Algorithmen die Markererkennung durchgeführt wird, stellte sich als sehr schwer heraus.
Lediglich ein paar Systeme wie ALVAR verwiesen im ”User’s manual” darauf, dass OpenCV benützt wird. Unter andrem findet sich dort folgender Auszug: ”OpenCV callback functions are often used for video frame acquisition and image processing.” [57]. Doch welche Algorithmen im Speziellen benutzt werden, und in welcher Reihenfolge, wird auch bei diesem System nicht klar.
Ein SDK das speziell für Smartphones entwickelt wurde, ist Vuforia von Qualcomm. Vuforia ist ein Augmented Reality Software Development Kit für mobile Geräte, dass das Entwickeln von Augmented Reality Anwendungen erleichtert. Es benutzt Computer Vision Technologie, um einfache Marker, Bilder oder 3D Objekte in Echtzeit zu erkennen. Eine zusätzliche Funktion erlaubt es dem Entwickler, eigene Bilder als Marker zu nutzen. In Beispielanwendungen, die frei zum Download [58] stehen, ist die Leistung sehr gut. Bewegt man das Smartphone, ist die *see-through* Darstellung und Markererkennung beinahe ohne Verzögerung. Da Vuforia von Qualcomm entwickelt wird und kein Open-

Tabelle 2.1.: Vergleich zwischen AR SDKs

SDK	GPS	Marker	Feature	IMU	Lizenz
3DAR [38]	-	+	-	-	Frei, z.T. kommerziell
AR23D[39]	-	+	-	-	Frei, z.T. kommerziell
ALVAR [40]	-	+	+	-	Frei, z.T. kommerziell
ARLab [41]	+	+	-	+	Frei, z.T. kommerziell
ARmedia [42]	+	+	+	+	Frei, z.T. kommerziell
ARToolkit [43]	-	+	+	-	Open Source, ARToolKit Commercial Dual license
D'Fusion [44]	-	+	+	-	Frei, z.T. kommerziell
Koozyt [45]	-	+	-	-	Kommerziell
Metaio SDKs [46]	+	+	+	+	Frei / kommerziell
Qoncept AR [47]	-	+	+	-	andere
Robocortex [48]	-	+	+	-	Frei, z.T. kommerziell
SSTT [49]	-	+	+	-	andere
Studierstube Tracker [50]	-	+	+	-	andere
UART [51]	-	+	-	-	Open Source
Vuforia [52]	-	+	+	-	Frei, z.T. kommerziell
Wikitude [53]	+	+	+	+	Frei, z.T. kommerziell
Xloudia [54]	-	+	+	-	kommerziell
yvisoin [55]	-	+	-	-	Frei, z.T. kommerziell
Zenitum Feature Tracker [56]	-	+	+	-	kommerziell

+: wird unterstützt, -: wird nicht unterstützt

Source Projekt ist, ist auch hier ebenfalls unklar, welche Algorithmen benutzt wurden. Da aber viele aber SDKs verwandt mit dem ARToolKit sind, können hier Anhaltspunkte für Algorithmen gefunden werden.

2.4. ARToolKit

Das ARToolKit bietet den Entwicklern ein sogenanntes ”out-of-the-box” Konzept zur Realisierung von Augmented Reality Applikationen an, womit sich Anwendungen schnell und einfach realisieren lassen sollen. Der Entwickler muss hier nur eine Kamera angeschlossen haben und die im SDK enthaltende Basisanwendung auf dem eigenen System kompilieren. Diese Anwedung erkennt dann einen optischen Marker, der zuvor ausgedruckt werden muss, und kann beliebig erweitert werden.
Das ARToolkit besitzt eine Reihe von Tracking-Routinen und Funktionen zur Videoanbindung für die Darstellung von einfachen virtuellen Objekten. Das ARToolKit ist plattformunabhängig und ist mit den richtigen Treibern unter Windows, MacOS X und Linux lauffähig. Allerdings wird in den Anforderungen definiert, dass das Betriebsystem Windows genutzt werden sollte, da auch die gesammte Videoanbindung für dieses Betriebssystem ausgelegt ist. Jedoch lässt sich diese Funktionalität auch für andere Betriebsysteme hinzufügen.

2.4.1. Aufbau

Wie in Abbildung 2.3 zu sehen ist, besteht die Funktionsbibliothek aus mehreren Teilen. Da das System für die Darstellung OpenGL benutzt und dieses keine Funktionen hat, um mit einem Fenstermanager zu kommunizieren, wird für die Kommunikation mit dem Fenstersystem GLUT [2] benutzt. Das ARToolkit nutzt ebenfalls Standard-API und Video Bibliotheken, die Betriebssystemanhängig sind. Dadurch werden dann Video und Grafiktreiber geregelt.

[2] GL Utility Toolkit

Das ARToolkit selbst besteht wiederum aus drei Teilen, wie in Abbildung 2.4 erkenntlich

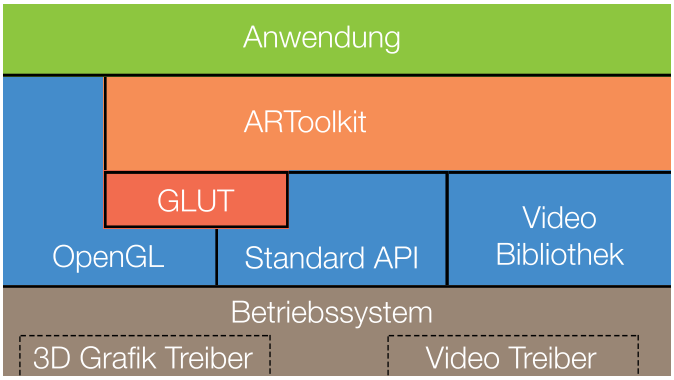

Abbildung 2.3.: Aufbau des ARToolkit

wird. OpenGL und GLUT Bibliotheken werden von dem GSUB-Modul verwendet, und stellt somit Zeichenroutinen zur Verfügung. Der zweite Teil ist das Video Modul, welches die Videoverarbeitung übernimmt und auf der Videobibliothek des Betriebsystems aufbaut. Der Kernteil ist das AR Modul. Dies beinhaltet alle Funktionen für das Tracking von Markern. Wie diese drei Teile schließlich zusammenarbeiten, kann in Abbildung 2.5 gesehen werden.
Hier wird die Kamera vom Videomodul initiiert und leitet immer das aktuelle Bild zur Verarbeitung weiter. Dieses wird dann von den Algorithmen des AR-Moduls bearbeitet. Die daraus resultierenden Trackingdaten werden dann vom GSUB-Modul genutzt, um virtuelle Objekte mit dem Videobild auszugeben.

2.4.2. Methodik

Um einen Überblick zu erlangen, kann die schematische Darstellung 2.5 helfen. Folgende Punkte werden dabei durchlaufen:

1. Von einer Kamera wird das aktuell aufgenommene Bild empfangen und zur Verarbeitung bereitgestellt.
2. Anschließend wird mittels Algorithmen nach einen Marker, genauer nach einem schwarzen viereckigen Rahmen, gesucht.

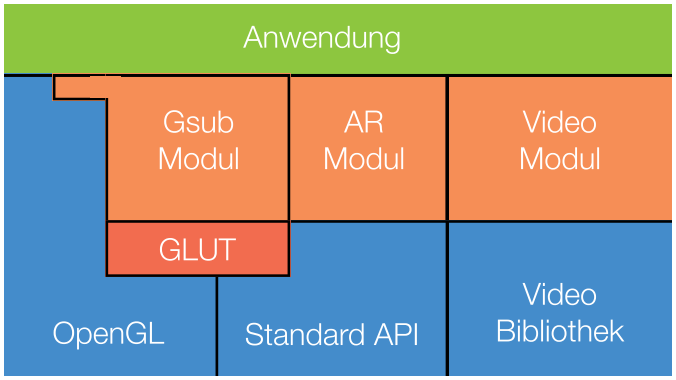

Abbildung 2.4.: Detaillierter Aufbau des ARToolKit

3. Ist dieser gefunden, wird die Position des Markers zur Kamera bestimmt.
4. Daraufhin wird der Marker mittels dem Symbol im Inneren identifiziert. Womit dann ID und Position des Markers bekannt sind und ein virtuelles Objekt daran ausgerichtet werden kann.
5. Im letzten Schritt wird dann das virtuelle Objekt in das Videobild gerendert.

Diese Schritte werden dann für jedes Einzelbild, das von der Kamera kommt, wiederholt. Die Markererkennung wird im ARToolkit mit der Funktion 'arDetectmarker' (Auszug 2.4.2) gestartet.

```
1 \\ar.h Headerdatei
2
3 int arDetectMarker (
4   ARUint8 * dataPtr,
5   int thresh,
6   ARMarkerInfo ** marker_info,
7   int * marker_num
8 );
```

Diese Funktion führt die Schwellenwertbildung, Etikettierung, Konturextraktion und die Linien-Eck-Schätzung aus. Damit ist sie mit 'arGetTransMat', welche die Berechnung der Transformation zwischen einem erkannten Marker und der realen Kamera durch-

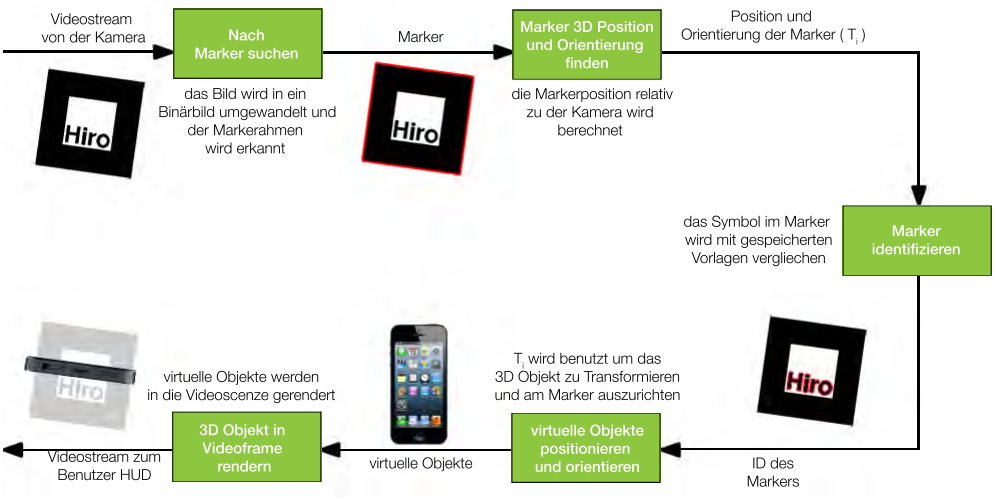

Abbildung 2.5.: Schematischer Ablauf der Augmented Reality Markererkennung

führt,eine der wichtigsten Funktoinen in den Erkennungsroutinen.

2.4.3. Analyse und Algorithmen

Im Folgenden wird der Funktionsaufruf aus 2.4.2 genauer betrachtet. Zuerst werden die Parameter und der Rückgabewert analysiert:

- *ARUint8 * dataPtr*
  Ein Zeiger auf die Farbbilddaten, in denen nach einem quadratischen Marker gesucht werden soll, wobei das Pixel-Format von der Architektur abhängt. Allgemein ist es ABGR, wobei die Reihenfolge von BGR keine Rolle spielt, da das Farbbild als Graubild gehandhabt wird. Nur die Anordung des Alpha-Kanals spielt eine Rolle.

- *int thresh*
  Ein Integer, der den Schwellenwert zwischen 0 und 255 bestimmt. Dieser wird verwendet, um das Bild in ein Binärbild umzuwandeln.

- *ARMarkerInfo ** marker_info*
  Ein Pointer auf ein Array von 'ARMarkerInfo' Strukturen. Dieses Array enthält dann alle Informationen über die gefundenen Quadrate im aktuellen Bild.
- *int * marker_num*
  Die Anzahl der gefunden Marker im Bild
- *int*
  War der Funktionsaufruf erfolgreich, wird 0 zurückgegeben, andernfalls -1.

Durch die Parameter kann schon etwas über die Implementierung und Art der Algorithmen erfahren werden.
Ein Bild im ABGR-Format hat den Nachteil, dass es schon von Anfang an mehr Speicher benötigt, da mehr Informationen enthalten sind, als zur Berechnung nicht nötig sind.
Des Weiteren wird durch den zweiten Parameter deutlich, dass es keine Funktion gibt, die den Schwellenwert dynamisch berechnet. Was den Vorteil hat, dass die Umwandlung durch einen festen Schwellenwert am schnellsten ist, was für eine gute Leistung spricht. Der Nachteil daran ist, dass wechselnde Lichtverhältnisse Probleme darstellen. Ist zum Beispiel eine Seite des Markers überbelichtet und die andere unterbelichtet, da ein Schatten über den Marker fällt, kann die Erkennung fehlschlagen.
In Abbildung 2.6 ist die Funktion ARDetectMarker dargestellt. Daran ist abzulesen, wie die einzelnen Schritte zur Markererkennung aussehen.

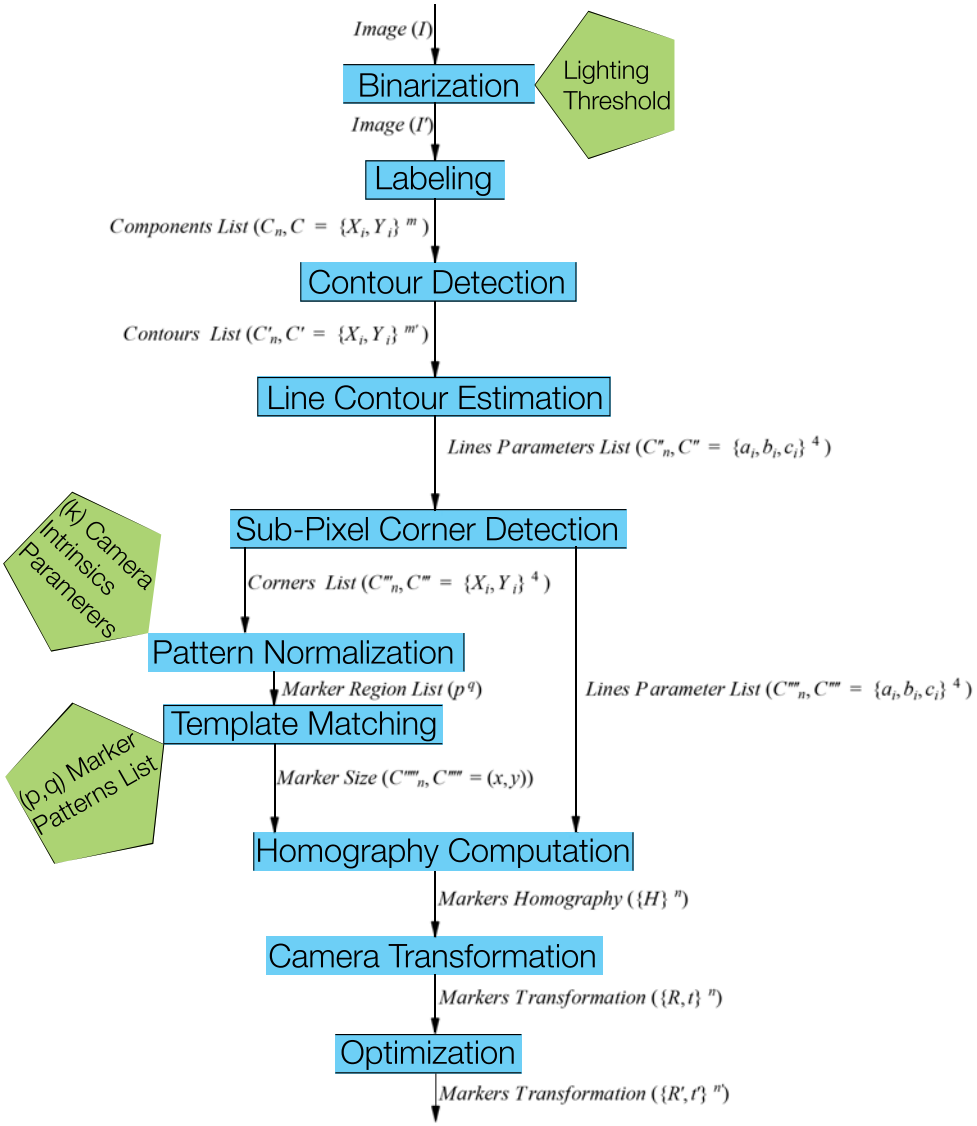

Abbildung 2.6.: Markererkenung des ARToolkit

3

Aufbau des Frameworks

3.1. Anforderungen an das Framework

Das fertige Framework sollte die in Tabelle 3.1 Anforderungen erfüllen. Zum einen sind es funktionale und nicht-funktionale Eigenschaften, die beachtet werden müssen. Das Framework sollte eine einfache Bedienung beinhalten. Durch einfache Ausrichtung des Systems müssen alle weiteren Schritte ohne Nutzerinteraktion ablaufen können. Die Darstellung der Kamerasicht ist unerlässlich für ein ”video-see-through” System und muss in Echtzeit ablaufen.
Die Logik sollte unter anderem ein modularen Aufbau aufweisen, so dass einzelne Schritte beziehungsweise die einzelnen Algorithmen ausgetauscht werden können. So ist es möglich, falls vorhanden, effizientere Algorithmen einzusetzten. Die Algorithmen an sich müssen robust sein und in jedem Fall richtige Ergebnisse liefern. Somit soll eine stabile Markererkennung gewährleistet werden.

Tabelle 3.1.: Anforderungen an das Framework

Anforderungsbeschreibung	Zuordnung	Art der Anforderung
Einfache Bedienung	View	nicht-funktional
Video-see-through	View	funktional
Bearbeitung in Echtzeit	Logik	funktional
Modularität der Schritte	Logik	nicht-funktional
Erkennung auf größere Distanzen	Logik	nicht-funktional
Stabilität der Erkennung	Logik	funktional
Robustheit der Algorithmen	Logik	funktional

3.2. Entwicklungsumgebung

Um eine Applikation oder ein Framework für iOS zu entwickeln, wird das iOS Software Development Kit von Apple benötigt [59].

3.2.1. Xcode 4.6

Xcode ist die integrierte Entwicklungsumgebung von Apple und nur für das Betriebsystem Mac OS X erhältlich [60]. Damit lassen sich Programme für Mac OS X und iOS schreiben. Xcode ist generell für die Programmiersprachen C, C++ und Objective-C mit dem Cocoa-Framework gedacht. Das Cocoa-Framework wiederum besteht aus drei Frameworks. Dem Foundation-Framework, welches alle relevanten Basisklassen wie Strings, Arrays, Speicher-Management, Iterators usw. stellt. Der zweite Teil stellt Klassen zur Entwicklung graphischer Benutzeroberflächen, wie Fenster, Buttons oder Menüs zur Verfügung. Dieses Framework wird Application Kit genannt. Objektgraphen werden mit dem dritten Framework, dem Core Data Framework erstellt.
Die Anwendung Xcode ist ein Teil dieser integrierten Entwicklungsumgebung und wird von Apple 'Xcode IDE' genannt, während die gesammte Entwicklungsumgebung meist als 'Xcode Tools' bezeichnet wird. Die zur Entwicklung dieses Frameworks genutzte Version ist die aktuelle Version 4.6. Die Version 4.0 wurde auf der WWDC 2010 vorgestellt und wurde am März 2010 veröffentlicht. Die aktuelle Version 4.6 wurde am 28. Januar 2013 veröffentlicht und brachte unterstützung für das aktuelle iOS 6.1.

3.2.2. iPhone 4

Das Smartphone auf dem das Framework entwickelt wird, ist das von Apple entwickelte iPhone 4, welches am 24. Juni 2010 in Deutschland veröffentlicht wurde. Das iPhone 4 ist die vierte Generation der iPhone Reihe und wurde auf der WWDC 2010 vorgestellt. Die technischen Daten können aus Tabelle 3.2.2 entnommen werden.

Tabelle 3.2.: Technische Daten iPhone 4 [61]

Hersteller	Apple (Entwickler), Foxconn (Auftragshersteller)
Veröffentlichung	24.Juni.2010 (Deutschland)
Display	89 mm (3,5”), 960x640 Pixel (326 ppi)
Digitalkamera	5 MP, 720p-HD-Video (30fps)
Frontkamera	0,3MP, 480p-VGA-Video (30fps)
Betriebssystem	iOS 6.3
Prozessor	Apple A4 (ARM Cortex-A8)
RAM	512 MB eDRAM
Interner Speicher	8 GB NAND
Sensoren	3-Achsen-Gyrosensor, Beschleunigungssensor, Annäherungssensor, Umgebungslichtsensor

3.2.3. Verwendete Frameworks

Um alle nötigen Schritte der Markererkenung durchführen zu können, sind einige Frameworks der Entwicklungsumgebung nötig. Unter anderem sind dies [62]:

- AVFoundation: Dieses Framework stellt Schnittstellen für die Audio-, Videoaufnahme und Verwaltung zur Verfügung.

- CoreGraphics: Enthält Schnittstellen für Quartz 2D API. Quartz ist eine Engine für Vektorzeichnungen.

- CoreImage: Bildverarbeitungs- und Analysetechnologien, die für Echtzeitandwendungen entwickelt wurden. Hier finden sich eventuell brauchbare Filter, um einzelne Schritte der Markererkennung zu realisieren.

- CoreMedia: Das CoreMedia Framework ist eines der Basisframeworks, auf die andere, wie das AVFoundation Framework, zugreifen. Mit diesem Framework erhält man eine genauere Kontrolle über Audio- und Videoinhalte.

- CoreVideo: Dort finden sich Funktionen zum Manipulieren von Audio- und Videodaten.

- Foundation: Stellt die Basisdatentypen und Funktionen bereit.

- UIKit: Wird von jeder Anwendung implementiert. Enthält unter anderem das Benutzeroberflächenmanagement.

Ebenfalls wurde ein externes Framework hinzugezogen, welches die Bildbearbeitung bei der Markererkennung unterstützen soll - das OpenCV Framework. OpenCV (Open Source Computer Vision) ist ein Bibliothek von Funktionen, die für Bildbearbeitung in Echtzeit ausgelegt ist. OpenCV wurde unter der BSD Lizenz veröffentlicht und ist frei für akademische und kommerzielle Zwecke. Es verfügt über in C++,C, Python und Java Schnittstellen für Windows, Linux, Android und Mac. OpenCV ist für effizientes Rechnen ausgelegt und mit einem starken Fokus auf Echtzeitanwendungen und kann die Vorteile von Mehrkernprozessoren nutzen. Die Bibliothek hat mehr als 2500 optimierte Algorithmen und diese finden weltweit Anwendung mit mehr als 2,5 Milllionen Downloads. Intel startete die Entwicklung der Bibliothek und heute wird diese hauptsächlich von Willow Garage gepflegt [63].

3.3. Programmieransatz

Es soll ein Framework geschaffen werden, bei dem eine stabile Markererkennung erfolgt. Dabei ist das maßgebliche Ziel, die Größe der Marker nicht zu verändern, aber die Entfernung auf die die Marker erkannt werden, zu erhöhen. Es sollen einfache optische Marker erkannt werden und eine Schnittstelle bereitgestellt werden, die die Position der Marker relativ zur Position zum Augmented Reality System zurückgibt.
Wie in Abschnitt 2.4.3 aufgezeigt, besteht das aktuelle Problem darin, dass die Marker eine bestimmte Größe, beziehungsweise, dass der Marker eine relativ große Fläche im Display einnehmen muss, um erkannt zu werden. Ausschlaggebend hierfür war die Einstellung, welche die Auflösung der verbauten Kameras klein hält.
Die Markererkennung soll aber nicht wie üblich auf die wenigen Zentimeter von den herkömlichen SDKs limitert sein. Umgesetzt werden sollte dies, indem die native Auflösung der verbauten Kamera benutzt wird und nach der ersten erfolgreichen Markererkennung nur noch nötige Teile des Bildes umgerechnet werden.
Um möglichst effizient zu bleiben, werden optische Marker in schwarz-weiß benutzt. Dies spart Rechenzeit, da eine Featureerkennung bei höheren Eingangsauflösungen der Kamera die Grenzen der vorhandenen Hardware überschreiten würde.
Der grundsätzliche Aufbau, welcher der Performanz zugute kommt, ist in Abbildung 3.1

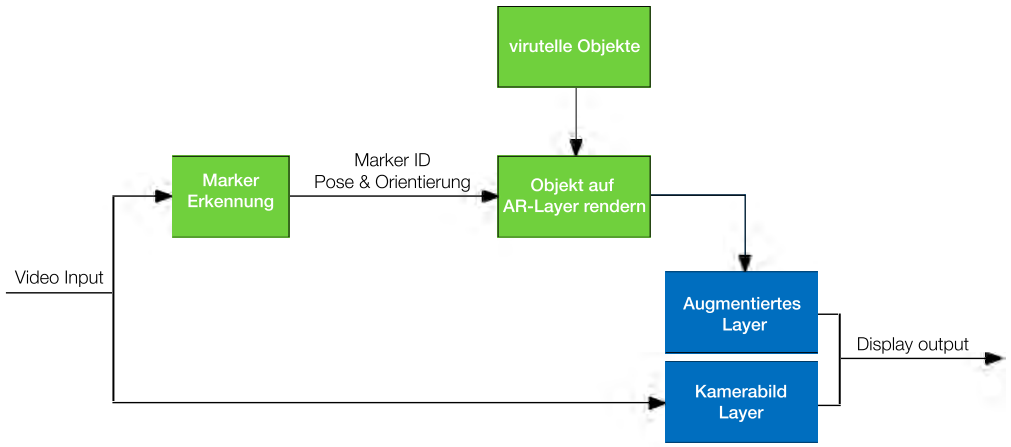

Abbildung 3.1.: Schematischer Ablauf des Augmented Reality Framework

schematisch dargestellt. Hierbei wird der Eingabestrom, welcher von der integrierten Kamera in Einzelbildern erfolgt, doppelt genutzt. Zum einen wird der Eingabestrom genutzt, um das wiederzugeben, was die Kamera in Echtzeit aufnimmt, und zum anderen wird der Datenstrom genutzt, um damit direkt die Berechungen für die Markererkennungen durchzuführen. Dieser Aufbau bietet einige Vorteile. Die Echtzeitvorschau wird auf einer Ebene dargestellt, somit kann, wie später in Abschnitt 3.4 ausführlicher erklärt wird, eine bereits implementierte Funktion herangezogen werden, welche für die Echtzeitvorschau optimiert ist. Genutzt wird diese Ebene auch in der Standard Kameraapplikation von Apple selbst.
Ein weiterer Vorteil dieses Aufbaus ist, dass nun so die einzelnen Bilder direkt bearbeitet werden können, ohne dass dies unsere Ausgabe auf dem Display beeinträchtigt. So kann die Erkennung der Marker mit einer anderen Anzahl an Bildern pro Sekunde als für die Echtzeitdarstellung des Kamerabildes erfolgen. Würde derselbe Stream zur Erkennung und zur Ausgabe benutzt werden, könnten bei niedrigen Bildraten optische Effekte wie Ruckeln entstehen. Durch die Trennung in zwei Datenströme kann auch ebenfalls direkt auf dem Buffer, der die Bilder enthält, gearbeitet werden. Der Vorteil hierbei ist das Sparen an Rechenzeit für Kopieroperationen im Speicher und das dynamische Verkleinern dieses Buffers, um weniger Daten bearbeiten zu müssen.
Somit können auf der zweiten Ebene, welche über das Echtzeitbild der Kamera gelegt wird, virtuelle Elemente gerendert werden. Da diese Ebene transparent ist, können auch durch Abstufungen der Transparenz Schatteneffekte erzeugt werden. Durch die freie Wahl der anderen Ebene, kann dies je nach Bedarf geändert werden. Dazu mehr in Kapitel 3.4.3.

3.4. Implementierung

Um eine performate Merkererkennung zu gewähren, müssen alle Schritte so effizent wie möglich ablaufen. Deshalb wird von Grund auf beachtet, dass keine zusätzlichen Schritte vorhanden sind, die unnötig Rechenzeit beanspruchen.
Das Xcode SDK bietet mit dem AVFoundation Framework, welches mit iOS4 eingeführt wurde, einen effizienzen Umgang mit Audiovisuellen (AV) Medien. Dieses Framework

stellt Eingabeströme zur Verfügung, die in Echtzeit manipuliert werden können.
Das zentrale Element in diesem Framework stellt die*AVCaptureSession* dar. Mit diesem Objekt lassen sich alle Ein- und Ausgabestöme konfigurieren und kontrollieren. Abbildung 3.2 veranschaulicht, wie Ein- und Ausgabestöme mit diesem Sessionobjekt verwaltet werden können. Ein Datenstrom, sei es Audio oder Video, wird hier von *AVCaptureInput* bereitgestellt und an das Sessionobjekt gebunden. Das Sessionobjekt wiederum liefert verschiedene Möglichkeiten der Ausgabe.
AVCaptureMovieFileOutput fasst Audio und Video zusammen und lässt dem Entwicker

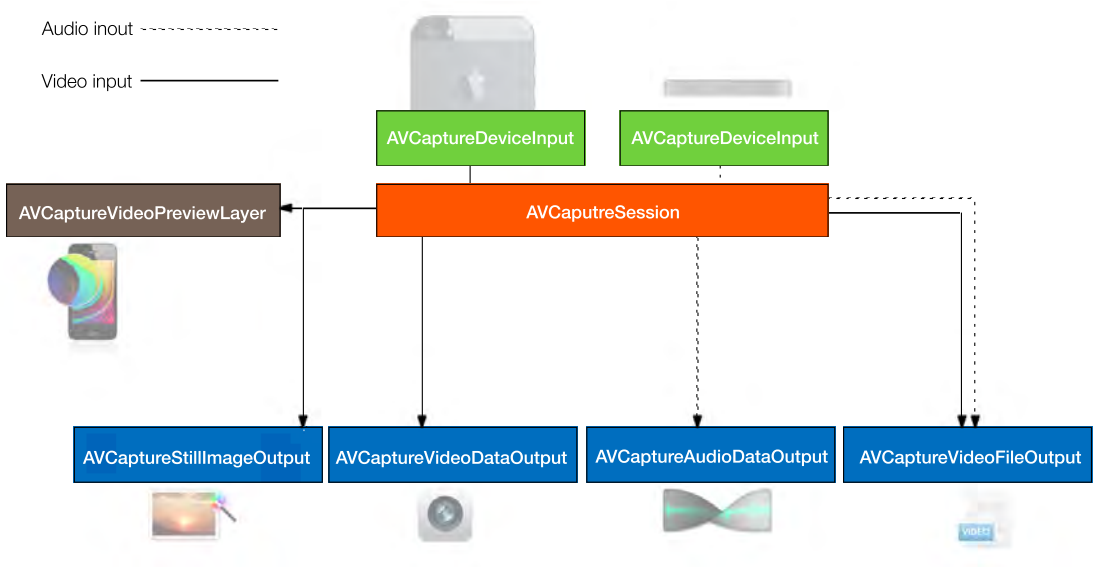

Abbildung 3.2.: Ein- Ausgabeströme die von der Session verwaltet werden

so die Möglichkeit, eine Videodatei zu erstellen. Wobei *AVCaptureAudioDataOutput* hingegen nur den Audiostream liefert. Da Audio bei der Markererkennung keine Rolle spielt, kann dieses vernachlässigt werden. Um einzelne Bider aus einem laufenden Stream entnehmen zu können, kann man sich dem *AVCaptureStillImageOutput* bedienen. Da aber jedes Bild bearbeitet werden soll, wird hier auf *AVCaptureVideoDataOutput* und auf *AVCaptureViedeoPrieviewLayer* zurückgegriffen. Letzteres spielt eine Sonderrolle.

3.4.1. Eingabestrom

Nachdem die Applikation gestartet und alle Konstruktoren aufgerufen wurden, wird der erste Funktionsaufruf gemacht. Nachdem die View geladen wurde, wird die Funktion *setupAVCapture* aufgerufen. In dieser Funktion wird das *AVCaptureSession* Objekt initialisiert und Eingabe- und Ausgabeströme werden konfiguriert.

Listing 3.1: Konfiguration der Kamera

```
1  - (void) setupAVCapture{
2    .
3    ..
4    AVCaptureDevice *camera =
5     [AVCaptureDevice defaultDeviceWithMediaType:
6      AVMediaTypeVideo];
7    [camera lockForConfiguration:nil];
8    [camera setFocusMode:AVCaptureFocusModeContinuousAutoFocus];
9    [camera setWhiteBalanceMode:
10    AVCaptureWhiteBalanceModeContinuousAutoWhiteBalance];
11   [camera unlockForConfiguration];
12   ..
13   .
14 }
```

- Der erste Schritt ist das Erstellen eines *AVCaptureDevices*. Ein *AVCaptureDevice* Objekt repräsentiert die physikalisches Aufnahmegerät, wie eine Kamera oder ein Mikrofon, und die Eigenschaften, die mit diesem Gerät verbunden sind. Mit diesem Objekt lassen sich die Eigenschaften der zugrunde liegenden Hardware konfigurieren. Dieses Objekt stellt auch die Eingangsströme für das Sessionobjekt zur Verfügung. Auszug 3.1 zeigt das Konfigurieren der Kamera. Der Befehl, der in Zeile 4 beginnt, wählt die rückwärtige Kamera aus, welche in der Entwicklungsumgebung xCode als *defaultDevice* von Typ Video aufgeführt wird.
  Die Hardware wird in den darauffolgenden Zeilen konfiguriert. Da nicht nur auf

eine nahe DistanzMarker erkannt werden sollen, sondern auch in der Entfernung, wird der Autofokus aktiviert, welcher sich kontinuierlich neu einstellt. Damit wird ein scharfes Bild erreicht, welches bei der Darstellung auf dem Display dem Nutzer gezeigt wird, und die Erkennung von Markern vereinfacht. Ebenfalls wird der automatische Weißabgleich aktiviert, wecher primär für die Markererkennung Vorteile bringt. Da durch den Abgleich ein möglichst hoher Konstrast zwischen schwarz und weiß entsteht, können Algorithmen, die beim Thresholding zum Einsatz kommen, einfach gehalten werden.

- Im zweiten Schritt wird das konfigurierte *AVCaptureDevice* an einen *AVCaptureInput* gebunden, welcher das Sessionobjekt mit dem Datenstrom versorgt. Damit ist der Eingabestrom komplett und kann an das Sessionobjekt gebunden werden (Auszug 3.2).

Somit liefert die Kamera nun den Datenstrom an die Session. Da die Kamera immer die volle Auflösung liefert, gibt es eine Möglichkeit, die Äuflösung direkt an der Session einzustellen, so dass diese die gewünschte Ausgangsauflösung liefert. Da die Session so wenig Rechenzeit wie möglich beanspruchen soll, und die volle unveränderte Bildgröße benötigt wird, wird auch die Session entsprechend konfiguriert.

Listing 3.2: Hinzufügen der Kamera zur Session

```
1 .
2 ..
3 AVCaptureInput *videoInput = [[AVCaptureDeviceInput alloc]
4  initWithDevice:camera error:nil];
5 [self.session addInput:videoInput];
6 ..
7 .
```

3.4.2. Ausgabeströme

Wie bei der Eingabe, muss auch die Ausgabe konfiguriert werden. Wie bereits in Abschnitt 3.4 angesprochen, wird ein Objekt vom Typ *AVCaptureVideoDataOutput* benötigt,

um direkten Zugriff auf jedes einzelne Frame zu bekommen. Um Arbeitsschritte zu sparen und dadurch die Leistung zu verbessern, konfiguriert man den Output wie in 3.3.

Listing 3.3: Konfiguration für Datenausgabe

```
1  .
2  ..
3  AVCaptureVideoDataOutput* videoOutput =
4   [AVCaptureVideoDataOutput  new];
5
6  NSDictionary *options;
7  options = @{(id)kCVPixelBufferPixelFormatTypeKey :
8   [NSNumber numberWithInt:
9    kCVPixelFormatType_420YpCbCr8BiPlanarFullRange]};
10 dispatch_queue_t captureQueue =
11  dispatch_queue_create("catpureQueue", NULL);
12
13 [videoOutput setVideoSettings:options];
14 [videoOutput setSampleBufferDelegate:self queue:captureQueue];
15 [videoOutput setAlwaysDiscardsLateVideoFrames:YES];
16
17 [self.session setSessionPreset:AVCaptureSessionPresetHigh];
18 [self.session addOutput:videoOutput];
19 ..
20 .
```

Durch Zeile 9 wird das Format bestimmt, welches der *PixelBuffer* haben soll, in dem später jedes einzelne Frame gespeichert wird. Bei diesem YCbCr Farbmodell teilen sich die Farbinformationen in die Grundhelligkeit Y und die zwei Farbkomponenten Cb (Blue-Yellow Chrominance) und Cr (Red-Green Chrominance) auf. Wie in Abbildung 3.3 zu sehen ist, enthält die Helligkeitsebene schon ein Graustufenbild, was für die weitere Verarbeitung benötigt wird. Dazu mehr in Abschnitt 3.4.2.

Ein weiterer Punkt zur schnellen Verarbeitung ist das Setzen der in Auszug 3.3, Zeile

Abbildung 3.3.: Originalfarbbild und Aufspaltung in die Komponenten Y, Cb und Cr. [64]

15 gesetzten Eigenschaft. Falls noch ein Frame in der Routine der Markererkennung ist, so werden neu ankommende Frames verworfen, bis der aktuelle abgeschlossen ist. Nachteil dieses Framedroppings, ist das es bei den virtuellen Elementen letztendlich zu sprunghaftem Positionswechsel kommen kann. Dieses Problem muss später beim Ausgeben der Positionen des Markers berücksichtigt werden.

Ausgabe der Kamerasicht

Da eine Echtzeitdarstellung bei der Augmented Reality unerlässlich ist, muss das was die Kamera erfasst, auf dem Display dargestellt werden. Hierbei wird das in Abschnitt 3.4 erwähnte *AVCaptureVideoPreviewLayer* verwendet. Damit dieses Layer auch Daten zur Darstellung erhält, wird es in Zeile 3 (Auszug 3.4) mit der aktuellen Session initialisiert. Damit die Echtzeitansicht auch wie gewohnt das ganze Display einnimmt, wird die Layergröße auf die Bildschirmgröße gesetzt (Auszug 3.4, Zeile 9).

Listing 3.4: Kameralivestream

```
1  .
2  ..
3  self.previewLayer = [[AVCaptureVideoPreviewLayer alloc]
4   initWithSession:self.session];
5  [self.previewLayer setBackgroundColor:
6   [[UIColor blackColor] CGColor]];
7  [self.previewLayer setVideoGravity:
8   AVLayerVideoGravityResizeAspectFill];
9  [self.previewLayer setFrame:[[UIScreen mainScreen] bounds]];
10
11 [self.previewView.layer addSublayer:self.previewLayer];
12 ..
13 .
```

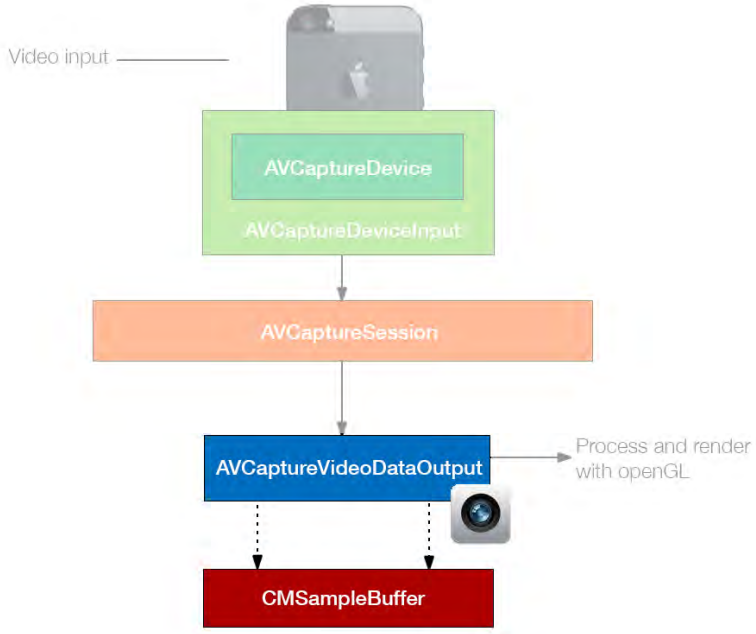

Abbildung 3.4.: Im CMSampleBuffer befindet sich immer das aktuelle Bild der Kamera

Verarbeitung der Frames

Nachdem die Konfiguration für das Mobile Augmented Reality System fertiggestellt ist, kommt die Erkennung der Marker. Durch das Konfigurieren der Session mit dem Output von Typ *AVCaputreVideoDataOutput* muss die Funktion 3.5 implementiert werden. Diese wird immer von der Session gerufen, wenn ein neuer Videoframe in einen Buffer geschrieben wurde. Dieser Buffer, siehe dazu auch die Abbildung 3.4, enthält dann idealerweise ein Bild indem sich ein Marker befindet.

Listing 3.5: Funktion für neue Frames

```
1 - (void) captureOutput:(AVCaptureOutput *)
2         captureOutput didOutputSampleBuffer:
3         (CMSampleBufferRef)sampleBuffer fromConnection:
4         (AVCaptureConnection *)connection{
5                 //Framebearbeitung
6 }
```

Nun werden die einzelnen Schritte, die in Abschnitt 2.2.1 vorgestellt wurden, durchlaufen und optimiert.

Graustufenbild

Die im Buffer abgelegten Rohdaten haben dann das Format, welches bei der Konfiguration in Absatz 3.3 Zeile 9 angegeben wurden. Da es sich hier um das Format *420YpCbCr8BiPlanarFullRange* handelt befindet sich wie in Absatz 3.4.2 beschrieben schon ein Graustufenbild irgendwo in diesem Buffer.
Da ein Graustufenbild zur weiteren Verarbeitung nötig ist, ist es ein entscheidender Vorteil, genau die Speicherstellen anzusprechen, die das Graustufenbild enthalten. Wird dies erreicht, so kann der erste Schritt, welcher das Farbbild mithilfe von Berechnungen ermittelt, welche in Absatz 2.2.1 erläutert wurden, eingespart werden. Nachdem die Speicherstellen ermittelt wurden, müssen sie zu weiteren Verarbeitung in ein Format des OpenCV Frameworks gebracht werden. Wie in 3.2.3 erklärt, ist der Datentyp *Mat* als Parameter für weitere Bildbearbeitungsalgorithmen nötig.
Bevor man jedoch die Bildinformationen aus dem *CMSampleBufferRef* Objekt lesen kann, muss dieses verstanden werden. Dieses Objekt ist aus dem Core Foundation Framework, es enthält null bis mehrere komprimierte oder unkomprimierte Exemplare von einem bestimmten Medientyp wie Audio-, Videodaten sowie beides kombiniert. Dieses Objekt enthält Referenzen auf ein *CMBlockBuffer* für ein oder mehrere Medienstücke oder eine *CVImageBuffer* Referenz. Diese Referenz beschreibt das Format des *CMSampleBuffer* und enthält wichtige Innformationen, wie die Größe, welche es im späteren Verlauf erlauben, den Datentyp zu wechseln.
Für den weiteren Schritt wird das Core Video Framework benötigt, welches den Datentyp *CVPixelBufferRef* enthält. Die Bildinformationen, die der Buffer enthält, werden in diesen Datentyp gewandelt. Auszug 3.6 zeigt den Aufbau dieses Datentyps. Darin enthalten sind Größe und Breite, die im späteren Verlauf gebraucht werden. Die Umwandlung erfolgt sehr einfach mit dem in Auszug 3.7 gezeigten Quellcode.

Listing 3.6: Aufbau Pixelbuffer

```
1 CVReturn CVPixelBufferCreate (
2    CFAllocatorRef allocator,
```

```
3     size_t width,
4     size_t height,
5     OSType pixelFormatType,
6     CFDictionaryRef pixelBufferAttributes,
7     CVPixelBufferRef *pixelBufferOut
8 );
```

Listing 3.7: Pixelbuffer von Samplebuffer

```
1 CVPixelBufferRef pixelBuffer =
2   CMSampleBufferGetImageBuffer(sampleBuffer);
```

Da nun das Farbbild des ursprünglichen Buffers hier enthalten ist, kann man auf relevante Speicherstellen zugreifen. Im ersten Schrittt werden Höhe und Breite des neuen Pixelbuffers benötigt. Diese lassen sich einfach durch die *Get* Funktionen des Objekts abfragen. In Auszug 3.8 ist dann zu sehen, wie die Startspeicheradresse der Y Ebene des Farbbildes referenziert wird. Bevor jedoch dieser Funktionsaufruf funktioniert, muss die Basisadresse gesperrt werden, um nicht gewollte Nebeneffekte zu verhindern.

Listing 3.8: Speicheradresse des Graubildes

```
1 CVPixelBufferLockBaseAddress(pixelBuffer, 0);
2
3 int bufferWidth = CVPixelBufferGetWidth(pixelBuffer);
4 int bufferHeight = CVPixelBufferGetHeight(pixelBuffer);
5
6 unsigned char *pixel = (unsigned char *)
7   CVPixelBufferGetBaseAddressOfPlane(pixelBuffer, 0);
```

Nun kann mithilfe eines Konstruktors des OpenCV Frameworks und des Standard Datentyps für Bilder das Graustufenbild extrahiert werden. In Auszug 3.9 ist der betreffende Code zu sehen. Übergeben werden hierbei die Variablen Höhe und Breite des *CVImageBufferRef* Objekts, der Zeiger auf die Startadresse der Y Ebene und eine Konstante. Letztere gibt an, dass es sich um *8Bit unsigned* Werte handelt. Dies ist wichtig, da

sonst die falschen Speicherstellen abgerufen werden und ein unvollständiges oder mit Fragmenten versehenes Bild entsteht.

Listing 3.9: OpenCV Datentyp

```
1 Auszug D
2
3 cv::Mat image = cv::Mat(bufferHeight,bufferWidth,CV_8U,pixel);
```

Die Information darüber, welche Werte ungefähr in dem *CVImageBufferRef* hält, konnte von der Dokumentation von Apple entnommen werden. Dabei musste beachtet werden, welchen Typ der Session bei der Konfiguration übergeben wurde.
Mit diesen nicht ganz einfach zu erlangenden Informationen konnte ein Farbbild, der von der Session kommt, ohne aufwendige und rechenzeitintensive Multiplikationen oder Divisionen in ein Graufstufenbild gebracht werden. Ein weiterer Vorteil ist, dass das Graufstufenbild gleich in einem Format ist, mit dem im OpenCV Framework gearbeitet werden kann.
Zur Kontrolle, das auch im Bildformat des OpenCV Frameworks die richtigen Daten enthalten sind, wurden die in Anhang A aufgeführten Hilfsfunktionen benutzt. Diese geben das Bild auf ein neues Layer aus. Dies dient lediglich der Kontrolle, da die Effizienz dieser Funktionen und des Layers, auf die das Bild ausgegeben werden, sehr schlecht sind. Die Funktionen sowie das Layer sind nicht auf Echtzeit, beziehungsweise auf schnell ändernde Bilder, ausgelegt.

Threshold

Der nächste Schritt in der Bildbearbeitung nützt vorhandene Funktionen des OpenCV Frameworks, da es zu komplex ist, solche Algorithmen zur Bildbearbeitung selbst zu schreiben. Es wurden viele Algorithmen getestet, um zu sehen, welche die besten Ergebnisse liefern. Als Ausschlusskriteruim wurde die in Anschnitt 3.5 genannte Funktion implementiert, die von der Session aufgerufen wird, wenn ein Frame aufgrund laufender Berechnungen nicht bearbeitet werden kann. Der Ausschluss wurde an nicht bearbeiteten Frames pro Sekunde festgemacht.
Für das Thresholding wurde unter anderem folgende Funktion aus dem OpenCV Fram-

work genutzt.
Die Funktion *void adaptiveThreshold(InputArray src, OutputArray dst, double maxValue, int adaptiveMethod, int thresholdType, int blockSize, double C)* wird häufig im Computer Vision Umfeld benutzt. Diese Funktion wendet einen adaptiven Schwellenwert auf ein beliebiges Array an. Um die Funktion besser zu verstehen, sollte man alle Parameter verstanden haben. Diese sind im Folgenden aufgelistet.

- src: Quellbild, der bis jetzt erlangte Graustufenbild in 8Bit
- dst: Ziel, muss ebenfalls 8Bit sein und dieselbe Größe haben. Aus Speichergründen ist Quelle und Ziel immer identisch.
- maxValue: Nicht-Null-Wert der den Pixeln zugeordnet wird, für die die Bedingung erfüllt ist.
- adaptiveMethod: Benutzer adaptiver Thresholding Algorithmus. Zum Beispiel: *ADAPTIVE_THRESH_MEAN_C* oder *ADAPTIVE_THRESH_GAUSSIAN_C*.
- thresholdType: Schwellenwerttyp, *THRESH_BINARY_INV*.
- blockSize: Größe eines Blockes von benachbarten Pixeln, die benutzt werden, um den Schwellenwert zu berechnen.
- C - Konstante subtrahiert vom Mittelwert oder gewichteten Mittelwert.

Bei dem vierten Parameter wird, wie bereits erwähnt, die Art des Verfahrens angegeben, in diesem Fall handelt es sich um die in Kapitel 2 erwähnte Otsu Methode, die mit *CV_THRESH_BINARY | CV_THRESH_OTSU* übergeben wird.

Listing 3.10: Schwellwert

```
1 double thresh =
2   cv::threshold(nz,nz,0,255,CV_THRESH_BINARY | CV_THRESH_OTSU);
3 cv::threshold(origImg,origImg,thresh,255,CV_THRESH_BINARY_INV);
```

Der hier aufgeführte Code (3.10) berechnet sehr gute Schwarz-Weiß Bilder bei unterschiedlichsten Lichtverhältnissen. Jedoch kam es bei manchen Situationen, wie zum Beipiel bei schnell wechselnden Lichtverhältnissen zu sogenannten Framedropping. Da

auch bei weiteren Test mit verschieden Parametern immer wieder ähnliche Probleme auftraten, wurde die Auflösung der Bilder, die die Session liefert, nach unten korrigiert, um die nächsten Schritte besser ausführen zu können.

Geschlossene Bereiche und Konturen
Die weiteren Schritte in der Markererkennung stellten sich als sehr schwierig und rechenintensiv heraus, wobei auch viele fehlschlugen. Beim Erkennen von geschlossenen Bereichen führten Ansätze, die nicht Algorithmen aus der OpenCV Bibliothek nutzten, zu keinen nennenswerten Ergebnisse oder die Leistung nahm so ab, dass es selbst bei herkömmlichen Auflösungen von Markerdetekton SDKs zu Framedropping kam. Einige Ansätze scheiterten an der Umsetzung in Objectiv-C, da einige Funktionen in Python oder anderen Programmiersprachen geschrieben sind.
Bei den Algorithmen zur Konturenerkennung war der Leistungseinbruch nicht so gravierend, dennoch wurden kaum mehr als 10 Frames pro Sekunde erziehlt. Getestet wurden hier, wie in Kapitel 2 bereits erwähnt, folgende Algorithmen:

- Canny-Edge-Detector [31]
- Sobel-Edge-Detector [35]

Finden der Markerecken und Positionsbestimmung
Wie bereits in den vorherigen Punkten erwähnt, führte die Wahl der benutzten Algorithmen zu gravierendem Leistungseinbruch und es entstanden weite Probleme, welche in Abschnitt 3.5 beschrieben werden. Da sich die weitere Implementierung im Rahmen dieser Arbeit als zu aufwendig herausstellte, musste die Implementierung an diesem Punkt eingestellt werden.

3.4.3. Virtuelle Elemente

Durch die am Ende erlangte Positionsmatix des geschriebenen Frameworks, kann ein virtuelles Objekt mithilfe von OpenGL ES 2.0 auf ein *EAGLLayer* gerendert werden. Dieses Layer sollte verwendet werden, da es am effizientesten arbeitet und der Applikations-

entwickler somit den Rechenaufwand niedrig hält, was wiederum der Markererkennung zugute kommt. Durch das Überlagern beider Layer entsteht so eine Augmented Reality.

3.5. Probleme

Im Laufe der Implementierung traten viele Probleme auf. Die schwierigsten und die, die zu Lösung viel Zeit in Anspruch genommen haben, werden hier nun diskutiert.
Da in den Frameworks, die der Entwicklungsumgebung Xcode beiliegen, nur wenige Algorithmen zur Bildverarbeitung enthalten sind, musste eine alternative Bibliothek gefunden werden. Dabei stellte sich OpenCV als beste Option heraus.
Die erste Herausforderung bestand darin, die verschiedenen Formate der Bilder zwischen iOS und OpenCV zu vereinen. Dies brauchte viel Zeit, da der von Apple genutzte *CMSampleBuffer* nur als Referenz beschrieben ist, und nicht der genaue Aufbau. Somit mussten Umwandlungen in andere Typen vollzogen werden, um eine bessere Transparenz zu erreichen. Dies wurde teilweise durch Reverse Engineering realisiert. Da das Zielformat in OpenCV klar war, wurden Zwischenschritte beziehungsweise temporäre Formate gesucht, welche mit OpenCV vereinbar waren. Nachdem dies gefunden war, musste eine Umwandlung des *CMSampelBuffer* in dieses Format gefunden werden, welche in Absatz 3.4 realisiert wurde.
Weitere große Probleme bestanden darin, die unterschiedlichen Sprachen wie C/C++ und Python, in denen OpenCV geschrieben ist, mit dem Objectiv-C Projekt zu verbinden. Teilweise ist es möglich, Objektiv-C und C++ zu verschachteln. Um Objectiv-C und C++ Quellcode zusammen innerhalb einer Datei benutzen zu können, muss die Dateiendung der betreffenden Datei in *.mm* umbenannt werden. Allerdings gilt dies nur für Code der betreffenden Datei die Header-Dateien müssen jedoch der gleichen Programmiersprache angehören. Ein Ausweg können hier sogenannte Wrapperklassen bieten, was aber sehr aufwändig ist, wenn es um viele Klassen geht. Einige Funktionsaufrufe und Parameterübergaben waren nicht in beide Richtungen möglich, was wiederum zu Komplikationen führte.
Die Algorithmen der Bildbearbeitung sind sehr rechenintensiv, was sich durch zunehmende Bildgröße schnell bemerkbar machte. Ein Bild in voller Auflösung kann unter

Umständen 3,5 MByte an Datenvolumen erreichen, wobei die Aufnahme der Kamera mit 30 Bildern pro Sekunde abläuft. Schnell wird hier klar, dass Algorithmen zu Bildverarbeitung nicht viel Zeit benötigen dürfen oder mehr Rechenleistung zur Verfügung haben müssen, als derzeit ein iPhone 4 bieten kann.
Um die Datengröße zu reduzieren, können verschiedene Techniken angewendet werden, welche im nächsten Abschnitt erläutert werden.

3.6. Abgleich der Anforderungen

Tabelle 3.3.: Abgleich der Anforderungen an das Framework

Anforderungsbeschreibung	Zuordnung	Bewertung
Einfache Bedienung	View	sehr gut
Video-see-through	View	sehr gut
Bearbeitung in Echtzeit	Logik	schlecht
Modularität der Schritte	Logik	gut
Erkennung auf größere Distanzen	Logik	schlecht
Stabilität der Erkennung	Logik	keine Aussage möglich
Robustheit der Algorithmen	Logik	gut

3.7. Weiterführende Ideen

Markererkennung mit voller Auflösung führt zu erheblichen Datenmengen, die vom Testgerät nicht in Echtzeit bearbeitet werden können. Daher muss vor der eigentlichen Markererkennung das Datenvolumen, zum Beispiel durch Kompression, reduziert werden. Um ein effizienteres Arbeiten zu ermöglichen, wurden Überlegungen angestellt, um die Datenmengen zu reduzieren. Folgende Ansätze wurden erarbeitet, die im Einzelnen oder in Kombination umgesetzt werden können.

3.7.1. Mehrkernbetrieb

Viele aktuelle Smartphones inklusive der aktuelle iPhone Generation haben mehr als einen Prozessorkern. Diesen Vorteil nutzten nicht nur einige Algorithmen, die in der OpenCV Bibliothek vorhanden sind, sondern man könnte auch diese Tatsache nutzen, um ein Bild auf zwei Kernen gleichzeitig berechnen zu lassen. Ein Ansatz wäre, das gesammte Bild in kleinere Bilder zu unterteilen, so dass ein Kern mehrere dieser Teile parallel zu einem anderen Kern bearbeiten kann. Abbildung 3.5 veranschaulicht dieses Vorgehen.

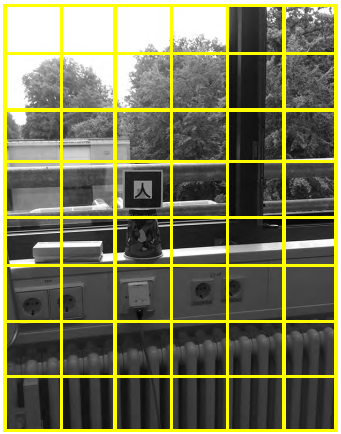

Abbildung 3.5.: Jedes gelbe Quadrate kann parallel verarbeitet werden.

3.7.2. Inertialsensorik

Ebenfalls kann mit der Inertialsensorik viel an Rechenleistung eingespart werden. Wie in Abbildung 3.6 zu sehen ist, können nur relevante Bildausschnitte verarbeitet werden. Ist der Marker nach dem Start der Applikation einmal gefunden, so kann er mithilfe von Gyroskop und Beschleunigungssensor verfolgt werden.
Durch das Auslesen dieser Sensorik kann berechnet werden, in welchem Bildausschnitt der Marker im folgenden Bild sein sollte. Dadurch muss nicht das ganze Bild bearbeitet werden, sondern nur ein kleiner Ausschnitt, welcher viel weniger an Speicher benötigt und somit auch schneller verarbeitet werden kann. Die dadurch erreichte Verkleinerung

der Datenmenge können kritische Algorithmen nutzen, um mehr Bilder pro Sekunde verarbeiten zu können. Dies verhilft wiederum zu einer flüssigen und schnelleren Augmented Reality Applikaiton.

Ebenfalls ist es möglich, Marker zu verfolgen, die den Fokus der Kamera verlassen.

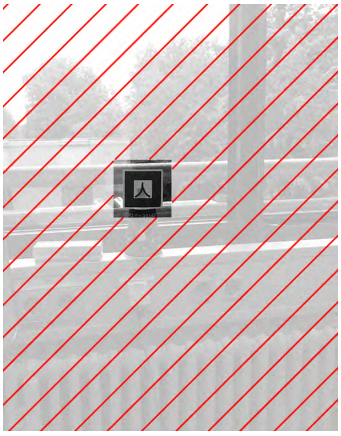

Abbildung 3.6.: Der rot markierte Bereich wird aus dem Buffer gelöscht.

Durch das interne Gyroskop kann die Ausrichtung des Smartphones errechnet werden. Gelangt der Marker anschließend wieder in den Fokus der Kamera, so muss nur das nähere Umfeld nach diesem Marker bearbeitet werden und nicht das ganze Bild, dass von der Kamera geliefert wird. Dies würde zu einer noch stabileren Augmented Reality Applikation führen und würde Fehler von nicht robusten Algorithmen verzeihen. Wird zum Beispiel der Marker durch einen zu geringen Sichtwinkel, von den Algorithmen nicht mehr erkannt, so kann durch die Sensorik immer noch die Bewegung, beziehungsweise die relative Positionsverschiebung, ermittelt werden.

Eine Kombination aus Mehrkernbetrieb (3.7.1) und Intertialsensorik (3.7.2) ist der beste Ansatz, um Mobile Augmented Reality auch auf große Distanzen.

4

Erkenntnisse

Abschließend werden in diesem Kapitel die Erkenntnisse zusammengefasst, die in dieser Ausarbeitung erzielt wurden.
Die Augmented Reality hatte ihren Ursprung in den 60 Jahren und ist bis heute noch ein Gebiet auf dem immer wieder neue Fortschritte gemacht werden. Mit dem gehäuften Aufkommen von Smartphones, die immer leistungsstärker werden, gewann auch die Mobile Augmented Reality erneut an Fahrt. Dennoch lassen sich die Desktopanwendungen nicht einfach auf mobile Geräte übertragen. Die Algorithmen in der Computer Vision, die über die Jahrzente weg entstanden sind, nutzten stets taktstarke Desktopprozessoren. Solche taktstarken Prozessoren stehen für Smartphones noch nicht bereit und deshalb müssen speziell optimierte Algorithmen entworfen werden, die die aktuelle Hardware effizient nutzen.
Unter anderem ist die Tatsache zu berücksichtigen, dass heutige Smartphones immer mehr Prozessorkerne mit niedriger Taktung besitzten, statt einen Kern mit hoher Taktung.

Dies ist darauf zurückzuführen, dass Energie immer noch der limiterende Faktor bei Smartphones ist. Viele Smartphones der Oberklasse können mit den verbauten Akkumulatoren ein bis zwei Tage bei durchschnittlicher Nutzung betrieben werden. Kommen rechenintensive Anwendungen, wie Augmented Reality Applikationen hinzu, sinkt damit die Laufzeit der Smartphones schnell unter ein paar Stunden.
Wie sich anhand der Tabelle in Kapitel 2.3 erkennen lässt, gibt es viele Firmen und andere Anbieter, die Source Development Kits bereitstellen und weiterentwickeln. Dabei ist in Tabelle 2.3 nur ein kleiner Auszug zu sehen. Es gibt noch weitere SDKs, die sich im speziellen auf andere Trackingverfahren, wie Feature-, oder GPS-Tracking spezialisiert haben und durch die fehlende Unterstützung für Markererkennung nicht mit aufgeführt werden.
Unternehmen wie Qualcomm, welches das Vuforia SDK entwickeln zeigen, dass auch Augmented Reality sehr gut auf mobilen Geräten funktioniert. Das Entwicklen mit dieser Plattform erfordert lediglich eine Registrierung, womit dann auch das Framework heruntergeladen werden kann.
Die Möglichkeiten, Mobile Augmented Reality Applikationen zu schreiben, sind vielfältig. Im Zuge dieser Ausarbeitung wurde evaluiert, dass weitere Untersuchungen notwendig sind, um limitierende Faktoren, wie Distanz, zu bereinigen. Die weiterführenden Ideen in Absatz 3.7 können hier den Einstiegspunkt für weitere Arbeiten darstellen.

A

Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

Listing A.1: Wird nur in Verbindung mit A.2 und A.3 benutzt

```
1 dispatch_sync(dispatch_get_main_queue(), ^{
2     _imageView.image = [self UIImageFromCVMat:image];
3     });
```

Listing A.2: Hilfsfunktion: Wandelt OpenCV Bild in iOS UIImage Bild um

```
1 -(UIImage *)UIImageFromCVMat:(cv::Mat)cvMat{
2   NSData *data = [NSData dataWithBytes:
3     cvMat.datalength:cvMat.elemSize()*cvMat.total()];
4   CGColorSpaceRef colorSpace;
5   if (cvMat.elemSize() == 1) {
```

```
6       colorSpace = CGColorSpaceCreateDeviceGray();
7     } else {
8       colorSpace = CGColorSpaceCreateDeviceRGB();
9     }
10    CGDataProviderRef provider =
11      CGDataProviderCreateWithCFData((__bridge CFDataRef)data);
12    // Creating CGImage from cv::Mat
13    CGImageRef imageRef =
14      CGImageCreate(cvMat.cols,
15      cvMat.rows,
16      8,
17      8 * cvMat.elemSize(),
18      cvMat.step[0],
19      colorSpace,
20      kCGImageAlphaNone|kCGBitmapByteOrderDefault,
21      provider,
22      NULL,
23      false,
24      kCGRenderingIntentDefault
25    );
26    UIImage *finalImage = [UIImage imageWithCGImage:imageRef];
27    CGImageRelease(imageRef);
28    CGDataProviderRelease(provider);
29    CGColorSpaceRelease(colorSpace);
30    return finalImage;
31  }
```

Listing A.3: Hilfsfunktion: Wandelt iOS UIImage in OpenCVBild um

```
1   -(cv::Mat)cvMatFromUIImage:(UIImage *)image{
2     CGColorSpaceRef colorSpace =
3       CGImageGetColorSpace(image.CGImage);
```

```
4    CGFloat cols = image.size.width;
5    CGFloat rows = image.size.height;
6    cv::Mat cvMat(rows, cols, CV_8UC4);
7      CGContextRef contextRef = CGBitmapContextCreate(cvMat.data,
8      cols,
9      rows,
10     8,
11     cvMat.step[0],
12     colorSpace,
13     kCGImageAlphaNoneSkipLast |
14     kCGBitmapByteOrderDefault
15   );
16   CGContextDrawImage(contextRef, CGRectMake(0, 0, cols, rows),
17     image.CGImage);
18   CGContextRelease(contextRef);
19   CGColorSpaceRelease(colorSpace);
20   return cvMat;
21 }
```

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] AZUMA, Ronald. A Survey of Augmented Reality. In: *Presence* 6 (1997), S. 355–385

[2] MILGRAM, Paul ; TAKEMURA, Haruo ; UTSUMI, Akira ; KISHINO, Fumio. Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum. 2351 (1994), S. 282–292

[3] SUTHERLAND, I. *Sketchpad: A man-machine graphical communication system*. Spartan Books, 1964

[4] SUTHERLAND, I. *The ultimate display*. Spartan Books, 1965

[5] SUTHERLAND, I. A head-mounted three dimensional display. (1998)

[6] KATO, Hirokazu ; BILLINGHURST, Mark. Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System. In: *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*. Washington, DC, USA : IEEE Computer Society, 1999 (IWAR '99). ISBN 0–7695–0359–4.

[7] MÖHRING, Mathias ; LESSIG, Christian ; BIMBER, Oliver. Video See-Through AR on Consumer Cell-Phones. In: *Mixed and Augmented Reality, IEEE / ACM International Symposium on* 0 (2004), S. 252–253. DOI 10.1109/ismar.2004.63. ISBN 0–7695–2191–6

[8] *Google Glass*. Version: 06 2013. `http://www.google.com/glass/start/what-it-does/`, Abruf: 17.06.2013

[9] RUKZIO, E. ; HOLLEIS, P. ; GELLERSEN, H. Personal Projectors for Pervasive Computing. In: *Pervasive Computing, IEEE* 11 (2012), Nr. 2, S. 30–37. DOI 10.1109/MPRV.2011.17. – ISSN 1536–1268

[10] TEGTMEIER, Andre. *Augmented Reality als Anwendungstechnologie in der Automobilindustrie*. 2007

[11] YANG, Zheng ; LIU, Yunhao. Quality of Trilateration: Confidence Based Iterative Localization. In: *icdcs* 0 (2008), S. 446–453. DOI 10.1109/icdcs.2008.59. – ISSN 1063–6927

[12] *Global Positioning System*. Version: April 2013. `http://www.gps.gov`, Abruf: 28.04.2013

[13] YILMAZ, Alper ; JAVED, Omar ; SHAH, Mubarak. Object tracking: A survey. In: *ACM Comput. Surv.* 38 (2006), Nr. 4. DOI 10.1145/1177352.1177355. – ISSN 0360–0300

[14] *IMU*. Version: Februar 2013. `http://www.starlino.com/imu_guide.html`, Abruf: 14.02.2013

[15] GEIGER, Philip. Entwicklung einer Augmented Reality Engine am Beispiel des iOS. Bachelor thesis, University of Ulm.

[16] WAGNER, Daniel ; REITMAYR, Gerhard ; MULLONI, Alessandro ; DRUMMOND, Tom ; SCHMALSTIEG, Dieter. Real Time Detection and Tracking for Augmented Reality on Mobile Phones. In: *IEEE Transactions on Visualization and Computer Graphics* 99 (5555), Nr. 1. DOI 10.1109/tvcg.2009.99. – ISSN 1077–2626

[17] GIULIANI, Gregory ; NATIVI, Stefano ; LEHMANN, Anthony ; RAY, Nicolas. WPS mediation: An approach to process geospatial data on different computing backends. In: *Computers and Geosciences* (2011). DOI 10.1016/j.cageo.2011.10.009. – ISSN 00983004

[18] *Wireless Positioning System*. Version: März 2013. `http://www.skyhookwireless.com/howitworks/xps.php`, Abruf: 01.03.2013

[19] *OpenGL*. Version: Mai 2013. `http://www.opengl.org`, Abruf: 21.05.2013

[20] PRYSS, Rüdiger ; LANGER, David ; REICHERT, Manfred ; HALLERBACH, Alena. Mobile Task Management for Medical Ward Rounds - The MEDo Approach. In:

1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops, Springer, September 2012 (LNBIP 132), pp. 43–54

[21] LANGER, David. Mobile Technik und Prozessmanagement zur Optimierung des Aufgabenmanagements im Kontext der klinischen Visite. Diploma thesis, University of Ulm.

[22] OWEN, C.B. ; XIAO, Fan ; MIDDLIN, P. What is the best fiducial? In: *Augmented Reality Toolkit, The First IEEE International Workshop*, 2002.

[23] *ARToolkit*. Version: 06 2013. `http://www.hitl.washington.edu/artoolkit/documentation/vision.htm`, Abruf: 13.06.2013

[24] *Grayscale Algorithm*. Version: Februar 2013. `http://www.johndcook.com/blog/2009/08/24/more-on-colors-and-grayscale/`, Abruf: 01.02.2013

[25] *ITU*. Version: Februar 2013. `http://www.itu.int/pub/R-REC`, Abruf: 21.2.2013

[26] HE, X.C. ; YUNG, N. H C. Curvature scale space corner detector with adaptive threshold and dynamic region of support. In: *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on* Bd. 2, 2004. – ISSN 1051–4651, S. 791–794 Vol.2

[27] SEZGIN, Mehmet ; SANKUR, Bülent. Survey over image thresholding techniques and quantitative performance evaluation. In: *Journal of Electronic Imaging* 13 (2004), Nr. 1, S. 146–168. DOI 10.1117/1.1631315. – ISSN 10179909

[28] NAKAGAWA, Yasuo ; ROSENFELD, Azriel. Some experiments on variable thresholding. In: *Pattern Recognition* 11 (1979), Nr. 3, 191 - 204. ISSN 0031–3203

[29] NADERNEJAD, E. Edge Detection Techniques: Evaluations and Comparisons. In: *Applied Mathematical Sciences* (2008)

[30] MARR, D ; HILDRETH, E. Theory of Edge Detection. In: *Proceedings of the Royal Society of London. Series B, Biological Sciences* 207 (1980), Nr. 1167, S. 187–217. DOI 10.2307/35407. – ISSN 00804649

[31] CANNY, John. A Computational Approach to Edge Detection. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-8 (1986), Nr. 6, S. 679–698. DOI 10.1109/tpami.1986.4767851. – ISSN 0162–8828

[32] AHMAD, M.B. ; CHOI, Tae-Sun. Local threshold and Boolean function based edge detection. In: *Consumer Electronics, IEEE Transactions on* 45 (1999), Nr. 3, S. 674–679. DOI 10.1109/30.793567. – ISSN 0098–3063

[33] DONY, R.D. ; WESOLKOWSKI, S. Edge detection on color images using RGB vector angles. In: *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on* Bd. 2, 1999. – ISSN 0840–7789, S. 687–692 vol.2

[34] RASKAR, Ramesh ; TAN, Kar-Han ; FERIS, Rogerio ; YU, Jingyi ; TURK, Matthew. Non-photorealistic camera: depth edge detection and stylized rendering using multi-flash imaging. In: *ACM Trans. Graph.* 23 (2004), August, Nr. 3, 679–688. DOI 10.1145/1015706.1015779. – ISSN 0730–0301

[35] KITTLER, J. On the accuracy of the Sobel edge detector. In: *Image and Vision Computing* 1 (1983), Nr. 1, 37 - 42. DOI http://dx.doi.org/10.1016/0262–8856(83)90006–9. – ISSN 0262–8856

[36] RAMER, Urs. An iterative procedure for the polygonal approximation of plane curves. In: *Computer Graphics and Image Processing* 1 (1972), Nr. 3, S. 244–256. DOI 10.1016/s0146–664x(72)80017–0. – ISSN 0146664X

[37] GONZALEZ, Rafael ; WOODS, Richard. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0201180758

[38] *3dAR*. Version: Februar 2013. `http://3dar.us`, Abruf: 16.02.2013

[39] *AR23D*. Version: Februar 2013. `http://www.ar23d.com`, Abruf: 16.02.2013

[40] *ALVAR*. Version: Februar 2013. `http://virtual.vtt.fi/virtual/proj2/multimedia/alvar/index.html`, Abruf: 16.02.2013

[41] *ARLab*. Version: Februar 2013. `http://www.arlab.com`, Abruf: 16.02.2013

[42] *ARMedia*. Version: Februar 2013. `http://www.inglobetechnologies.com/en/`, Abruf: 16.02.2013

[43] *ARToolkit*. Version: Februar 2013. http://www.artoolworks.com, Abruf: 17.02.2013

[44] *D'Fusion*. Version: Februar 2013. http://www.t-immersion.com, Abruf: 17.02.2013

[45] *Koozyt*. Version: Februar 2013. http://www.koozyt.com, Abruf: 17.02.2013

[46] *Metaio*. Version: Februar 2013. http://www.metaio.com/home/, Abruf: 18.02.2013

[47] *Qoncept*. Version: Februar 2013. http://qoncept.jp, Abruf: 18.02.2013

[48] *Robocortex*. Version: Februar 2013. http://www.robocortex.com, Abruf: 18.02.2013

[49] *SSTT*. Version: Februar 2013. http://technotecture.com/augmentedreality, Abruf: 18.02.2013

[50] *Studierstube*. Version: Februar 2013. http://studierstube.icg.tugraz.at/handheld_ar/stbtracker.php, Abruf: 19.02.2013

[51] *UART*. Version: Februar 2013. https://research.cc.gatech.edu/uart/, Abruf: 19.02.2013

[52] *Vuforia SDK*. Version: Februar 2013. https://developer.qualcomm.com/mobile-development/mobile-technologies/augmented-reality, Abruf: 19.02.2013

[53] *Wikitude*. Version: Februar 2013. http://www.wikitude.com, Abruf: 19.02.2013

[54] *Xloudia*. Version: Februar 2013. http://xloudia.com, Abruf: 19.02.2013

[55] *Yvision*. Version: Februar 2013. http://www.yvision.com, Abruf: 19.02.2013

[56] *Zenitum*. Version: Februar 2013. http://www.zenitum.com/en/research/, Abruf: 19.02.2013

[57] *User's Manual (v.2.0)*. : *User's Manual (v.2.0)*

[58] *Qualcomm Vuforia*. Version: Juni 2013. https://www.vuforia.com, Abruf: 01.07.2013

[59] *iOS SDK*. Version: März 2013. https://developer.apple.com/devcenter/ios/index.action, Abruf: 02.03.2013

[60] *Xcode*. Version: Februar 2013. https://developer.apple.com/xcode/, Abruf: 01.02.2013

[61] *iPhone4 Daten*. Version: Juni 2013. http://www.apple.com/de/iphone/iphone-4/specs.html, Abruf: 23.06.2013

[62] *Apple Frameworks*. Version: Juni 2013. http://developer.apple.com/library/ios/#documentation/miscellaneous/conceptual/iphoneostechoverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html, Abruf: 01.07.2013

[63] *OpenCV*. Version: Juni 2013. http://opencv.willowgarage.com/wiki/, Abruf: 01.07.2013

[64] Version: April 2013. http://de.wikipedia.org/wiki/YCbCr-Farbmodell, Abruf: 28.04.2013

Name: Fabian Schwab Matrikelnummer: 690469

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Fabian Schwab