Fakesch, Jens:

Multi-User 3D Augmented Reality Anwendung für die gemeinsame Interaktion mit virtuellen 3D Objekten

Masterarbeit, Sommersemester 2016

Gutachter: Höhl, Wolfgang ; Fuchs, Christian und Vogel, Jörg

Fakultät für Mathematik, Informatik und Statistik
Media Informatics
Master

Ludwig-Maximilians-Universität München

https://doi.org/10.5282/ubm/epub.36648

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Department "Institut für Informatik"
Lehr- und Forschungseinheit Medieninformatik
Prof. Dr. Heinrich Hußmann

Masterarbeit

Multi-User 3D Augmented Reality Anwendung für die gemeinsame Interaktion mit virtuellen 3D Objekten

Jens Fakesch
fakesch@cip.ifi.lmu.de

Bearbeitungszeitraum:	25.01.2016 bis 25.07.2016
Betreuer:	Dr.-Ing. Wolfgang Höhl
Externer Betreuer:	Fuchs & Vogel media solutions GbR
Verantw. Hochschullehrer:	Prof. Dr. Heinrich Hußmann

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer netzwerkgestützten Multi-User Augmented Reality Anwendung für mobile Endgeräte. Ein Präsentator hat die Möglichkeit, 3D-Modelle dynamisch zu laden, auf einem Augmented Reality 2D-Tracker anzuzeigen und an einzelnen Objekten Manipulationen durchzuführen. Diese sind bereits durch die Objekthierarchie des 3D-Objekts genauer definiert. Ausführbare Manipulationen sind die Translation, Rotation, Skalierung und der Wechsel von Materialien.

Eine beliebige Anzahl von Zuschauern kann der Präsentation mit dem eigenen Gerät aus ihrem selbstgewählten Blickwinkel folgen. Befinden sich die Modelldaten nicht auf ihrem Endgerät, werden sie automatisch vom Präsentator über das Netzwerk übertragen.

Die im Vorfeld getätigten Überlegungen werden beschrieben, gefolgt von den Details der Implementierung und aufgetretenen Problemen sowie gefundenen Lösungen.

Anhand des Prototyps werden mithilfe einer Nutzerstudie Leitlinien für die Wahl der Beleuchtungsart zwischen statischer, dynamischer oder kombinierter Beleuchtung für bestimmte Anwendungsfälle festgelegt. Zusätzlich dazu wird in der Studie die Usability der Anwendung überprüft.

Abstract

This thesis covers the development of a network supported multi user augmented reality application for mobile devices. A presenter can load 3D models dynamically, display them on an augmented reality 2D tracker and is capable of manipulating certain single objects. These manipulations are already well-defined in advance through the hierarchy of the 3D object. Executable manipulations are translation, rotation, scaling and the change of materials.

Any number of spectators can follow the presentation with their own device from a point of view of individual choice. If the data of the model is not present on their own device it will automatically be transferred from the presenter via network.

Thoughts that were made in advance are described, followed by the details of implementation and the occured problems as well as chosen solutions.

With the prototype a user study was conducted to define guidelines for the choice of different kinds of lighting for certain applications. The choice is between static, dynamic and combined lighting. Additionally the general usability of the app is evaluated in the study.

Aufgabenstellung

Die Arbeit beschäftigt sich mit der Entwicklung einer leicht zu bedienenden netzwerkgestützten Multi-User Augmented Reality Präsentationsanwendung für mobile Endgeräte und den verschiedenen Beleuchtungsansätzen in der Augmented Reality.

Durch den Blick durch Tablets oder Augmented Reality Brillen können Objekte präsentiert werden, die nicht physisch vor Ort sein müssen. Wichtig ist hierbei die Vernetzung der Anwender, so dass ein Präsentator virtuell im Vorfeld festgelegte Interaktionen mit 3D-Objekten ausführen kann, die weitere Betrachter wahrnehmen können.

Projekte wie *Junaio*, *LecceAR* (Banterle et. al.), *Layar*, *Augment* oder *Wikitude* erlauben es bereits, relativ einfach 3D Objekte in der Welt zu platzieren, bieten aber standardmäßig keine Multi-User Funktionen. *Second Surface* (Kasahara et. al.) konzentriert sich auf die Kollaboration, verzichtet jedoch auf 3D Objekte und die Interaktion mit ihnen.

Als Lösung soll ein Prototyp einer Anwendung entwickelt werden, der es durch eine benutzerfreundliche Oberfläche und Vernetzung auch technisch unversierten Personen ermöglicht, Produkte mithilfe von Augmented Reality zu präsentieren oder diesen Präsentation zu verfolgen. Durch die Netzwerkunterstützung ist es mehreren Benutzern möglich, sich in der selben digitalen Welt zu bewegen und diese zu erleben.

Anhand des entwickelten Prototyps soll schließlich im Laufe einer Nutzerstudie herausgefunden werden, ob sich Leitlinien für die Auswahl hochqualitativer statischer oder anpassbarer dynamischer Beleuchtung bei bestimmten Anwendungsfällen der Augmented Reality finden lassen. Die Arbeit wird in Kooperation mit der *Fuchs & Vogel media solutions GbR* geschrieben. Die Entwicklung findet vor Ort in Ottobrunn bei München statt.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, 20. Juli 2016

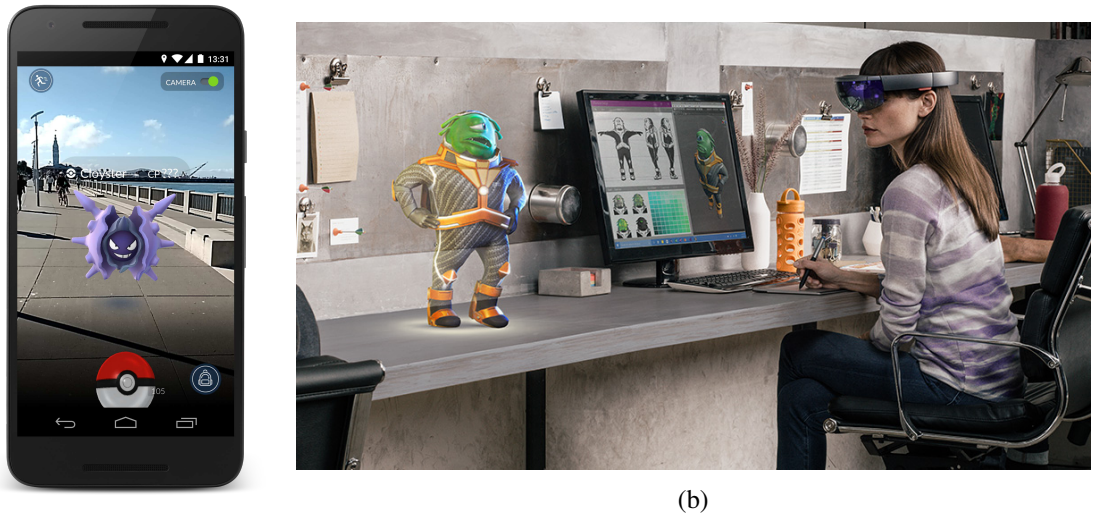
. . . . . .

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Durch die rasant steigende Leistungsfähigkeit bei stetig schrumpfender Größe der Hardware für mobile Endgeräte in den letzten Jahren gewinnt auch die Augmented Reality (auch AR) immer mehr an Bedeutung. AR-Anwendungen wurden erst in letzter Zeit wirklich praktikabel und beginnen nun, sich mehr und mehr zu verbreiten. Das beste Beispiel hierfür ist der aktuell explosionsartige Erfolg der Augmented Reality Anwendung *Pokémon GO*, bei der die Spieler in der realen Welt digitale Monster einfangen und sammeln können (siehe Abbildung 1.1a). Schon etwa eine Woche nach der Veröffentlichung hat die App mehr täglich aktive Nutzer als der weit verbreitete Kurznachrichtendienst *Twitter*. Rund 6% der *Android*-Nutzer in den USA rufen die Anwendung täglich auf. Die Spieler verbringen mit ihr mehr Zeit pro Tag als mit *Facebook* [40]. Diese jüngsten Entwicklungen zeigen eindeutig, dass die mobilen Endgeräte und vor allem die Menschen heutzutage für die Augmented Reality bereit sind.



(a) (b)

Abbildung 1.1: (a) Die Augmented Reality App *Pokémon GO* [35],
(b) Die AR-Brille *Microsoft HoloLens* [29].

Auch die fortschreitenden Entwicklungen im Bereich der mobilen Endgeräte bestätigen diesen Eindruck. *Microsoft* bringt mit der *HoloLens* [30] ein Headset auf den Markt, das rein auf AR ausgelegt ist (siehe Abbildung 1.1b) und auch *Googles Project Tango*, ein AR-Tablet mit echter Tiefenwahrnehmung [14], zeigt, dass die Augmented Reality Unterstützung von großen Herstellern und zunkunftsorientierten Tech-Konzernen hat.

Natürlich ist nicht nur die Unterhaltung ein sinnvoller Einsatzbereich für AR-Anwendungen. Auch auf dem Gebiet der Navigation, Weiterbildung und gerade auch Planung und Präsentation ergeben sich zahllose Möglichkeiten. Reale oder auch erst in Entwicklung befindliche Objekte lassen sich als 3D-Modelle abbilden und durch die Augmented Reality leichter in ihrer Struktur von allen frei wählbaren Blickwinkeln erfassen. Zwar gibt es viele Anwendungen und relativ leicht zugängliche Wege, um Modelle auf AR-Trackern in der realen Welt anzeigen zu lassen (siehe *2.1 Augmented Reality* unter *2 Stand der Technik*), jedoch vernachlässigen diese das gemeinsame Erleben der digitalen Welt, den sozialen Aspekt, welcher einer der Faktoren für den Erfolg von *Pokémon GO* ist.

Aus diesem Grund soll in dieser Arbeit eine Mehrbenutzeranwendung entwickelt werden. Alle Zuschauer, die an einer AR-Präsentation teilnehmen, sollen die selbe virtuelle Welt aus ihrem eigenen Blickwinkel erleben. Jede Veränderung auf Seiten des Präsentators hat Auswirkungen auf diese gemeinsame Welt. Der weitere Fokus liegt bei der Entwicklung auf der einfachen Definition der erlaubten Veränderungen sowie dem komfortablen Austausch der 3D-Modelle, was einen dynamischen Ladevorgang erfordert.

Die passende Beleuchtung der Modelle hat dabei einen großen Einfluss auf die Glaubhaftigkeit der AR. Anhand des Prototypen lassen sich verschiedenste Anwendungsfälle entwickeln, die auf Interaktionen mit Objekten basieren. Im Zuge dieser Arbeit soll ermittelt werden, ob es allgemein gültige Richtlinien für die Beleuchtung solcher Fälle gibt. Statische vorberechnete Beleuchtung ist zwar aufwendig umzusetzen, bietet dafür aber sehr hohe Qualität und gute Laufzeiteigenschaften. Dynamische Beleuchtung ist visuell weniger eindrucksvoll, passt sich aber verändernden Umständen an. Je nach Aufgabe, die man mithilfe einer Augmented Reality Anwendung lösen möchte, eignen sich die unterschiedlichen Ansätze mal mehr und mal weniger. Diese Arbeit soll eine Entscheidungshilfe bei der Wahl zwischen statischer, dynamischer oder einer kombinierten Beleuchtung in der Augmented Reality geben.

1.2 Überblick über diese Arbeit

Die Arbeit beginnt mit einem ausführlichen Überblick über den aktuellen Stand der Technik im Bereich der Augmented Reality und bei Beleuchtung in Gameengines im Allgemeinen. Es wird erläutert, welche Ansätze es für eine glaubhafte Beleuchtung gibt und inwiefern diese für mobile Endgeräte verwendbar sind.

Anschließend wird die Konzeption des in der Arbeit entwickelten Prototyps beschrieben. Es wird dargelegt, welche Anforderungen er erfüllen sollte und welche Entscheidungen im Vorfeld getroffen wurden. Dazu gehören technische Aspekte wie die Wahl der Entwicklungsumgebung und der Trackinglibrary, wie aber auch die nutzerzentrierten Entscheidungen, wie die verschiedenen Rollen und Interaktionsmöglichkeiten.

Der folgende große Abschnitt beschreibt die Implementierung der Anwendung. Das beginnt bei der Erstellung der 3D-Inhalte nach bestimmten Regeln, beschreibt den Weg aus der Modellierungssoftware bis in die fertige Anwendung und anschließend die (programmier-)technischen Details des Backends, sowie die gestalterischen Designentscheidungen des Frontends. Der Teil endet mit einer Erläuterung der im Laufe der Umsetzung aufgetretenen Schwierigkeiten und den dafür gefundenen Lösungen.

Das darauffolgende Kapitel behandelt die durchgeführte Nutzerstudie mit dem Schwerpunkt Beleuchtung. Es beginnt mit der Konzeption, wie der Entwicklung drei verschiedener Anwendungsfälle, des Fragebogens und dem Aufstellen der zu überprüfenden Hypothesen. Nachdem die Details der Durchführung dargelegt wurden, werden die im Laufe der Studie gewonnenen Daten ausgewertet und interpretiert. Stützen die gefundenen Ergebnisse die aufgestellten Leitlinien für die Verwendung bestimmter Beleuchtungsansätze bei Augmented Reality oder werden sie widerlegt?

Abschließend folgt ein zusammenfassendes Fazit. Es werden noch einmal kurz und klar die gewonnenen Erkenntnisse dargelegt und in Zusammenhang zueinander gesetzt. Die Arbeit endet mit einem Ausblick auf denkbare zukünftige Entwicklungen und Einsatzmöglichkeiten dieses Prototyps.

2 Stand der Technik

Im folgenden Abschnitt werden der aktuelle Stand der Technik und bedeutende wissenschaftliche Arbeiten aus dem Gebiet der AR und der Computergrafik vorgestellt. Die Grundlagen und ausführliche Geschichte der AR sollen dabei nicht Teil dieser Arbeit sein, der Fokus liegt vielmehr auf den technischen Möglichkeiten, Trackingverfahren, der grafischen Darstellung und Mehrbenutzerinteraktionen.

2.1 Augmented Reality

Im Jahr 2010 gaben Krevelen et al. einen umfassenden Überblick über die Geschichte und den damals aktuellen Stand der Augmented Reality [61]. Die Arbeit beinhaltet eine ausführliche Einordnung der verschiedenen Displays und Endgeräte, Trackingverfahren, User Interfaces und Anwendungsmöglichkeiten. Eine weitere Zusammenfassung der Entwicklung und Forschung von Augmented Reality-Technologien bis 2008 bieten Zhou et al. [73]. In ihrer Arbeit analysierten sie Beiträge der letzten zehn Jahre der ISMAR (*International Symposium on Mixed and Augmented Reality*), der bedeutendsten akademischen Konferenz auf dem Gebiet der Augmented und Mixed Reality.

Durch die Weiterentwicklung und Miniaturisierung der Technologie, insbesondere von Smartphones und Tablets, verbreiteten sich diverse Softwarelösungen und Frameworks, die auch dem Laien ermöglichten, mit relativ geringem Aufwand selbst AR Anwendungen zu entwickeln. Die Firma *Layar* wurde 2009 in den Niederlanden gegründet [26]. Sie bietet mit der gleichnamigen *App*, dem *Creator* und dem *SDK* (*Software Development Kit*) die Möglichkeit, digitale Inhalte mit Hilfe von verschiedensten Trackern anzuzeigen. Vergleichbares bot die Münchener Firma *Metaio* mit ihrem *Junaio AR Browser*, dem *Creator* und dem *SDK* an. 2015 wurde *Metaio* von *Apple Inc.* übernommen und bis auf Weiteres geschlossen [11]. Weitere AR SDKs mit vergleichbaren Funktionen sind *Wikitude* [69], *Augment* [2] oder das inzwischen ebenfalls eingestellte *String* [51]. Zuverlässige Trackingtechnologie mit API-Anbindung in C++, Java, Objective-C und .Net, sowie Anbindung an die *Unity Game Engine* bietet das *Vuforia Augmented Reality SDK* [62]. Ehemals von *Qualcomm* betrieben, wurde *Vuforia* 2015 von *PTC* übernommen [44].

2.2 Tracking

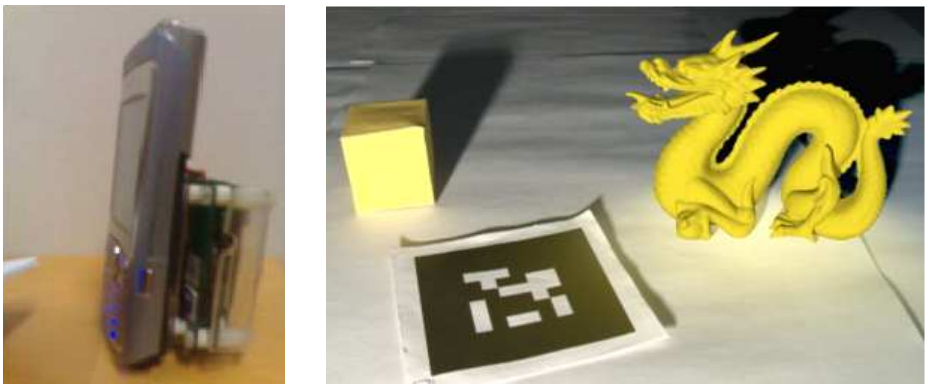
Um die Position und Orientierung der Kamera und damit des Betrachters im Raum zu ermitteln, gibt es bei Augmented Reality verschiedene Trackingverfahren. Zhou et al. [73] ordnen diese in die folgenden drei Grundkategorien ein.

2.2.1 Sensorbasiertes Tracking

Das sensorbasierte Tracking benutzt für die räumliche Einordnung verschiedenste Sensoren. Die unterschiedlichen Verfahren werden von Rolland et al. [47] detailliert vorgestellt. Magnetische Sensoren messen die Position in einem magnetischen Feld. Akustische Sensoren bieten die Möglichkeit, durch Ultraschallsender und -empfänger aktuelle Positionen zu berechnen. Neigungssensoren und Gyroskope messen die Orientierung und Beschleunigung getrackter Objekte. Im Jahr 2006 stellten Kähäri et al. *MARA* vor [21]. Hierbei handelte es sich um ein Mobiltelefon mit zusätzlich befestigter Sensorhardware, mit deren Hilfe das Tracking verbessert wurde (siehe Abbildung 2.1a). Die hier unhandlich extern befestigten Sensoren wie GPS-Empfänger und Beschleunigungsmesser sind heutzutage in modernen Mobiltelefonen und Tablets bereits intern eingebaut und können für zuverlässigeres Tracking benutzt werden.

Jede dieser nicht optischen Sensorarten bieten jedoch Vor- und Nachteile, wie Störungen durch die Umwelt oder Ungenauigkeit, so dass sie für optimale Ergebnisse kombiniert werden sollten. 2004 taten dies Newman et al. mit ihrem *Ubiquitous Tracking* [34]. Hier wurden Daten aus

verschiedenen Sensoren kombiniert, dynamisch und automatisch verrechnet und schließlich für weitere Anwendung zu Verfügung gestellt. Allgemein gab es jedoch im Bereich des rein sensorbasierten Trackings in den letzten Jahren keine herausstechenden Weiterentwicklungen, da die Technologie an sich schon seit längerem gut entwickelt ist [73].



(a) (b)

Abbildung 2.1: (a) MARA, ein Mobiltelefon mit zusätzlicher Sensorhardware [21],
(b) Optisches Tracking mit klassischem schwarzweißen 2D-Marker [22].

2.2.2 Optisches Tracking

Das optische Tracking mit Hilfe von Kameras ist nach Zhou et al. der Schwerpunkt der Augmented Reality Trackingforschung [73]. Hierbei werden Bildverarbeitungsverfahren benutzt um die Position der Kamera im Raum zu berechnen. Die optischen Verfahren lassen sich in zwei Kategorien unterteilen: *feature-based*, also merkmalbasiertes und *model-based*, modellbasiertes Tracking [43]. Beim merkmalbasierten Ansatz wird das Bild auf zweidimensionale Merkmale untersucht. Dies können unter anderem geometrische Grundformen [49] oder Konturen von Objekten sein [18]. Das modernere modellbasierte Tracking arbeitet mit einem 3D Modell des tatsächlichen zu verfolgenden Objekts. Zur Registrierung des Objekts werden jedoch auch hier 2D Ansätze wie die Kantenerkennung eingesetzt [27]. Die ersten Ansätze für optisches Tracking verwendeten vordefinierte zweidimensionale Marker (siehe Abbildung 2.1b). So auch das *ARToolKit* [24], ein weit verbreitetes Framework, welches bereits 1999 vorgestellt wurde. Andere Verfahren nutzten das Auswerten von nicht händisch platzierten Merkmalen, wie die Erkennung von in dem Bild auftretenden Linien oder Texturen von Objekten, sogenannten natürlichen Markern [38]. Seit einer zusammenfassenden Arbeit über markerbasiertes Tracking von Zhang et al. im Jahr 2002 [72] ergaben sich aber auch hier im wissenschaftlichen Bereich keine weitreichenden Neuentwicklungen mehr.

Erwähnenswert ist jedoch das *Extended Tracking* des Vuforia AR Frameworks [63]. Hierbei werden, zusätzlich zum Marker selbst, Merkmale der Umgebung verwendet, um das Tracking aufrecht zu erhalten auch wenn der Marker sich nicht mehr im Bild befindet. Das Vuforia Framework bietet neben Erkennung selbst definierter und vorgefertigter 2D Marker auch das erwähnte modellbasierte Tracking [66].

2.2.3 Hybrides Tracking

Das hybride Tracking verwendet eine Kombination der vorherigen Ansätze. Da in einigen Situationen das rein sensorische oder rein optische Verfahren nicht zufriedenstellend sein kann, verbessert das Zusammenwirken der beiden das Trackingergebnis. Nach Azuma et al. [4] ist für das Tracking in der unkontrollierten freien Umgebung der hybride Ansatz unerlässlich. Der visuelle

Kanal wird hier durch GPS und Lagesensoren unterstützt. Lang et al. stellten 2002 die Kombination von sichtbasiertem Tracking mit einem leichten günstigen zusammengesetzten System aus Beschleunigungssensoren und Gyroskop vor [25]. Durch hybride Verfahren wird das Tracking schneller und robuster und bietet die Möglichkeit der Bewegungsvorhersage [41]. Seit 2014 arbeitet Google an *Project Tango*, einer Technologieplattform, die verbessertes räumliches Tracking für mobile Geräte ermöglichen soll. Die Verwendung von zwei Kameras, statt wie sonst üblich nur einer, erlaubt Tiefenwahrnehmung. Unterstützt durch Beschleunigungsmesser und Gyroskop bietet sie zuverlässiges Motiontracking [14]. Viele der modernen AR SDKs bedienen sich des hybriden Trackings, um so ein präzises und glaubhaftes Ergebnis zu gewährleisten.

2.3 Licht und Schatten

Licht und Schatten sind im Bereich der computergenerierten Grafiken seit jeher ein Schwerpunktthema. Sie haben großen Einfluss auf den Realitätsgrad von virtuellen Welten, weshalb sie stets im Fokus der Forschung standen. Bei der Augmented Reality ist neben dem Realismus auch die Performanz entscheidend, sowie die Anpassung an die reale umgebende Welt. Die wichtigsten Arbeiten und Entwicklungen zu diesen Themen werden in den folgenden Abschnitten vorgestellt.

2.3.1 Beleuchtung in Augmented Reality

Eine Klassifizierung verschiedener Beleuchtungsmethoden nahmen Jacobs und Loscos im Jahr 2006 vor [20]. Die Arbeit bedient sich des Konzepts der Mixed Reality, einem Oberbegriff für die Kombination aus realen und virtuellen Elementen, welcher von Milgram und Kishino bereits 1994 eingeordnet wurde [31]. Da die Augmented Reality hiernach ein Teil der Mixed Reality ist, ist die Klassifizierung auch hier anwendbar. Es werden drei verschiedenen grundlegende Beleuchtungsmethoden genannt: *common illumination, relighting* und *inverse illumination*. *Common illumination* bezeichnet die Methoden, bei denen eine Beleuchtungsüberblendung stattfindet, wie zum Beispiel der Schattenwurf von realen Objekten auf virtuelle und umgekehrt. Hierbei darf die Beleuchtung nicht verändert werden und bleibt statisch. *Relighting*, also die Neubeleuchtung, ermöglicht die Änderung der Lichtverhältnisse der Szene. Hierzu wird die Beleuchtung der realen Szene analysiert und soweit möglich virtuell abgeschaltet. Das ermöglicht die virtuelle Neubeleuchtung der kompletten Szene und somit ein einheitliches Ergebnis. Bei *inverse illumination*, auch physikalisch basierte Beleuchtung genannt, wird versucht die physikalischen Eigenschaften aller Objekte zu ermitteln, sowie die Positionen und Eigenschaften der realen Lichtquellen. Anhand dieser Informationen können die virtuellen Objekte nun realistisch in die Szene eingefügt werden. Einen Überblick zu dieser Methode bieten Patow und Pueyo [39].

Die fotorealistische Beleuchtung von virtuellen Elementen in realen Szenen in Echtzeit ist mit der Weiterentwicklung der Renderverfahren und auch der leistungsfähigeren Hardware inzwischen prinzipiell möglich. Praktikabel und mobil sind die Verfahren jedoch in der Regel nicht. Die optisch hochwertigsten Ergebnisse müssen mit hohem Rechenaufwand erkauft werden, was stationäre Desktop-PCs und/oder zu geringe Frameraten bedeutet. Die direkte Anwendbarkeit für mobile AR entfällt somit meist noch.

Eine Technik, die die Umgebungsbeleuchtung bei der Berechnung von harten und zusätzlich auch weichen Schatten mit einbezieht wurde mit dem Verfahren der Lichtfaktorisierung von Nowrouzezahrai et al. 2011 vorgestellt [37]. Hier wurden hohe Frameraten von über 70 FPS erreicht, jedoch behandelte die Arbeit nur die Schattierung. Kan und Kaufmann stellten 2013 einen GPU-basierten Ansatz vor, der durch Raytracing sogar indirekte Illumination und Refraktion bot und die bisherigen vergleichbaren Ansätze in Sachen Performanz übertraf [22] (siehe Abbildung 2.2). Auf einer Nvidia GeForce GTX 690 Grafikkarte wurden bei einer geringen PAL Auflösung von 720x576 Pixeln jedoch trotz dessen nur Frameraten von im Schnitt etwa 10 bis 15 FPS erreicht. 2014 veröffentlichen Gruber et al. ein Verfahren für den effizienten und robusten Strah-

lungstransfer zwischen sich verändernden realen und virtuellen Objekten [15]. Auch hier wurden auf einer Nvidia GeForce GTX 780 bei einer noch geringeren Auflösung von 640x480 Pixeln nur Frameraten um 10 FPS erreicht.

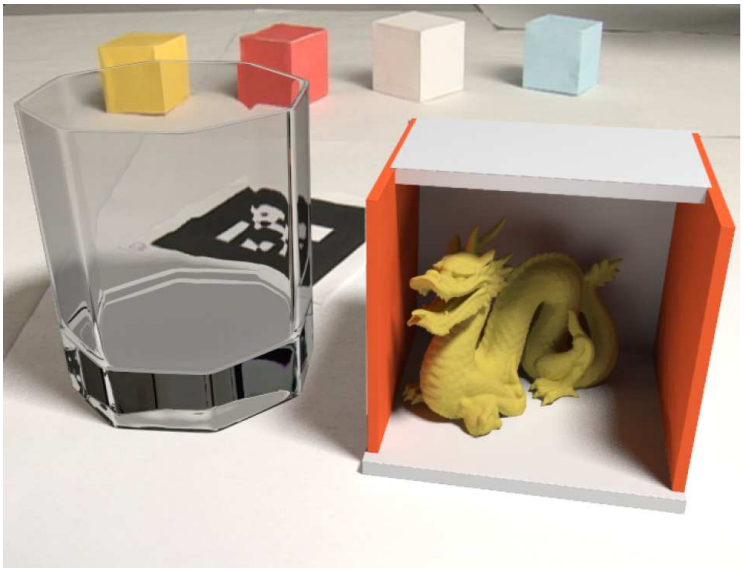


Abbildung 2.2: Die hohe Qualität der Darstellung nach Kan et al. [22] mit Refraktion im Glaskörper sowie indirekter Illumination. Beide Objekte im Vordergrund sind virtuell.

Um virtuelle Objekte glaubhaft in die reale Szene einzubetten, gibt es den Grundansatz, die aktuelle Lichtsituation zu ermitteln und diese Beleuchtung auf die künstlich erzeugten Objekte zu übertragen. Das Einfangen der Lichtverhältnisse kann auf mehrere Arten geschehen. Borg et al. erstellten bei ihrem Verfahren eine *High Dynamic Range* (HDR) Map [5]. Dabei handelt es sich um eine 360° Abbildung der Umgebung, aufgenommen durch ein Fischaugenobjektiv oder zusammengesetzt aus mehreren Einzelbildern. Durch Aufnahmen mit verschiedenen Belichtungseinstellungen und ihrer Kombination zu einer einzelnen HDR Map kann ein höherer Belichtungsbereich abgedeckt werden als bei herkömmlichen Einzelaufnahmen. Mit den Informationen aus der HDR Map kann das virtuelle Objekt schließlich realistisch ausgeleuchtet und eingefügt werden. Anstatt einer statischen, vordefinierten Map kann auch eine dynamische Map verwendet werden, die anstelle von Fotokameras mit Videokameras zur Laufzeit aufgezeichnet und verarbeitet wird (so geschehen bei [46]). Das ermöglicht sich verändernde Umgebungen und Lichtverhältnisse, der Berechnungsaufwand steigt jedoch enorm.

Die Beleuchtung mittels HDR Map wurde bereits 1998 von Paul Debevec beschrieben [10]. Er benutzte zur Erstellung der Map eine *light probe*, ein Objekt mit dessen Hilfe das Licht eingefangen wird. In seinem Fall handelte es sich dabei um ein simple verspiegelte Kugel welche fotografiert wurde. Im Laufe der Zeit wurden *light probes*, auch *shading probes* genannt, weiterentwickelt. 2013 stellten Calian et al. verschiedene Probes vor, die mittels 3D-Druck erstellt wurden und deren Konstruktion sich an der Umgebung orientierte [6] (siehe Abbildung 2.3). So unterschieden sich Probes für den Innen- und Außenraum in der Verteilung und Unterteilung von Messpunkten. Zusätzlich fingen sie nicht das einfallende Licht sondern die Schattierung direkt ein, was einen aufwendigen Berechnungsschritt ersparte.

Moderne Renderengines erlauben es heutzutage, Bilder zu berechnen, die von Fotografien so gut wie nicht mehr unterscheidbar sind. Die notwendigen Rechenschritte für fotorealistische Ergebnisse, wie Einfall und Reflexionen von Lichtstrahlen und weiteren physikalisch korrekten Effekten, benötigen trotz fortschreitender Optimierung und Weiterentwicklung der Hardware immer noch eine gewisse Rechendauer, die von einer Echtzeitdarstellung weit entfernt ist.

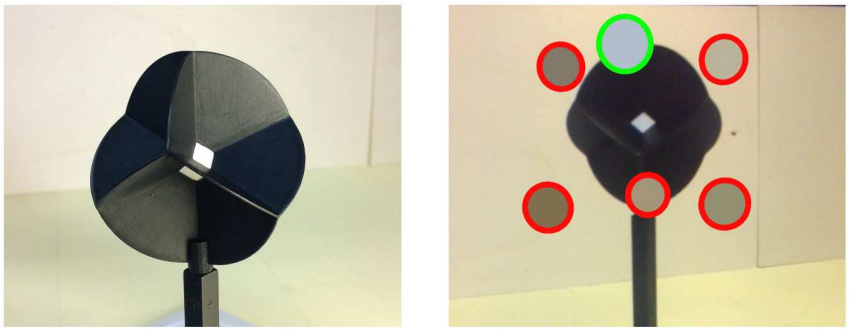


Abbildung 2.3: Eine mit 3D-Druck erstellte *Shading Probe* nach Calian et al. [6]. Das rechte Bild zeigt die ermittelten Shadingwerte der verschiedenen Messpunkte.

Die Darstellung in Echtzeitrenderengines bedient sich deshalb seit jeher mehr oder weniger präzisen Annäherungen, die zwar die Reduzierung von Rechenschritten und damit hohe Frameraten ermöglichen, dadurch aber auch eine Reduzierung der optischen Qualität zur Folge haben.

2.3.2 Schatten

Ein wichtiges Thema in der Entwicklung der Computergrafik ist die Darstellung von Schatten. Realistischer Schattenwurf trägt einen Großteil zur Glaubhaftigkeit einer computergenerierten Szene bei und ermöglicht es, Objekte räumlich besser einzuordnen. Besonders deutlich wird dies bei AR, wo sich der virtuelle Inhalt in die reale Umgebung einfügen muss. Durch den direkten Vergleich zwischen der realen und der berechneten Beschattung, stechen Diskrepanzen besonders hervor. Einen grundlegenden Überblick über verschiedenste Methoden der Schattenberechnung bieten Woo et al. [71].

Ein weit verbreiteter Ansatz zur performanten Berechnung von Schatten ist das sogenannte *Shadow Mapping*, bereits im Jahre 1974 von Catmull grundsätzlich beschrieben [7] und 1978 von Williams erweitert [70]. Eine *Shadow Map* ist ein Tiefenbuffer der Geometrie der Szene aus Sicht der Lichtquelle (siehe Abbildung 2.4a). Um herauszufinden, ob ein Punkt in der Szene im Schatten liegt, wird seine Entfernung zur Lichtquelle berechnet. Ist der Wert hierbei größer als der des entsprechenden Punktes im Tiefenbuffer, ist er aus Sicht der Lichtquelle verdeckt und liegt im Schatten. Da *Shadow Maps* von der Grafikhardware unterstützt werden, sind sie performant. Sie sind allerdings auch anfällig für Artefakte, was mit der Abtastrate, der Auflösung der Map zusammenhängt. Hierdurch entsteht eine Treppenbildung an den Kanten der Schatten, das *Aliasing* (siehe Abbildung 2.4b). Gegenmaßnahmen sind diverse Filterverfahren (zum Beispiel das *Antialiasing* von Reeves et al. [45]), Anpassungen bei der Berechnung der Map (z.B. [1]) oder simpler jedoch rechenintensiver: eine *Shadow Map* mit einer höheren Sampleanzahl.

(a) (b)

Abbildung 2.4: (a) Eine *Shadow Map* mit Tiefeninformationen aus Sicht der Lichtquelle, (b) *Aliasing* bei *Shadow Mapping* [8].

Eine weitere grundsätzliche Herangehensweise sind die *Shadow Volumes*, eingeführt 1977 von Crow [9]. Hier werden zunächst die Silhouettenkanten aller schattenwerfenden Objekte ermittelt. Die Polygone, deren Normalen der Lichtquelle zugewandt sind, werden *front-facing* genannt, die abgewandten *back-facing* Polygone. Die Kanten, die diese beiden Arten von Polygonen trennen sind die Silhouettenkanten, also die Umrisse der schattenwerfenden Objekte aus Sicht des Lichts. Sie werden nun, von der Lichtquelle weg, extrudiert und je nach Implementierung mit Abschlusskappen versehen, was einen geschlossenen Volumenkörper ergibt (siehe Abbildung 2.5). Alles was sich in diesem *Shadow Volume* befindet, liegt im Schatten. Heidmanns Ansatz nutzte 1991 [16] den *Stencil Buffer* und ermöglichte so den Gebrauch von *Shadow Volumes* zur Schattendarstellung in Echtzeitanwendungen. Der Hauptvorteil gegenüber *Shadow Maps* liegt in der Pixelgenauigkeit, jedoch muss für die Berechnung zusätzliche Geometrie erzeugt werden. Die beiden grundsätzlichen Verfahren können kombiniert werden, um effiziente hybride Renderalgorithmen zu entwickeln (z.B. bei Chan und Durand [8]).

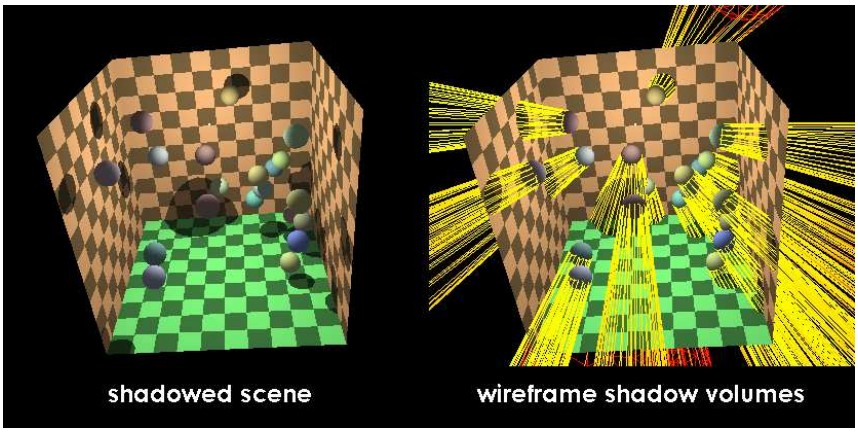
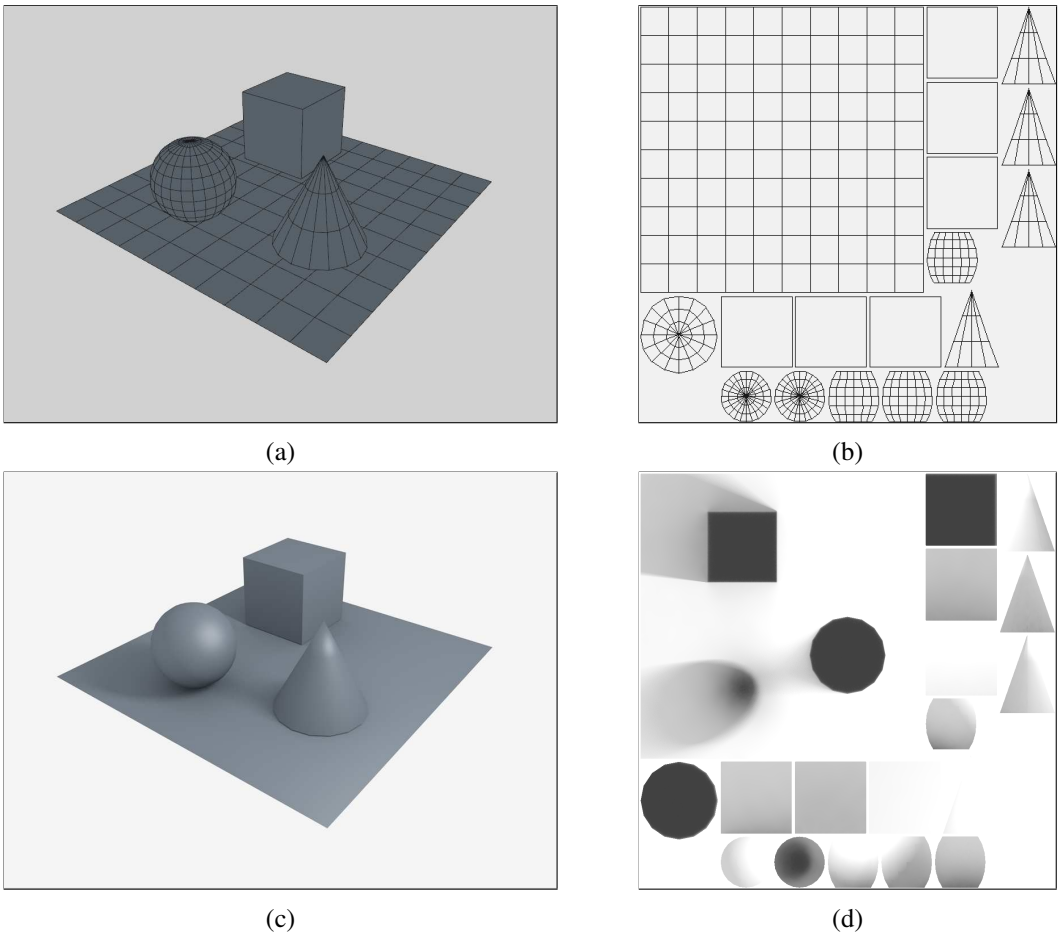


Abbildung 2.5: Berechnete *Shadow Volumes*, hier in gelb dargestellt [68].

Echtzeitschatten müssen innerhalb der Zeit eines Framewechsels berechnet und angezeigt werden. Bei einer flüssigen Framerate von 60 FPS stehen also etwa nur 17 Millisekunden pro Frame zur Verfügung. In dieser kurzen Zeitspanne ist es unmöglich, die Qualität einer Beleuchtung zu erreichen, die theoretisch unbegrenzt Zeit für das Rendering hat. Aufgrund dessen ist es üblich, Licht und Schatten für statische Objekte im Vorhinein hochqualitativ zu berechnen, in Texturen zwischenzuspeichern und performant zur Laufzeit lediglich anzuzeigen. Dieser Vorgang der Vorberechnung wird *Backen* genannt. Um Licht auf ein 3D-Modell zu backen, muss es zunächst *ab-*

gewickelt werden um eine *UV-Map* zu erstellen. Das bedeutet, dass jedes Polygon eine bestimmte Fläche eines Texturraums zugewiesen bekommt. An der entsprechenden Stelle in der Textur des Modells werden schließlich die Farb- oder Lichtinformationen für dieses Polygon gespeichert (siehe Abbildung 2.6). Der große Vorteil der Vorberechnung ist jedoch gleichzeitig auch der größte Nachteil des Ansatzes, denn er verhindert Veränderungen der Szene zur Laufzeit. Sobald sich Objekte, egal ob 3D-Modelle oder Lichter, verändern, sind die zuvor berechneten Daten ungültig. Es ist zwar möglich, die Texturen für verschiedene Situation zu backen und zwischen ihnen zu überblenden (z.B. eine Version für Tageslicht, eine für die Nacht), jedoch handelt es sich dabei nur um eingeschränkt nutzbare Sonderfälle. Echtzeitschatten können auf diese Weise also nicht gänzlich überflüssig gemacht werden.



(a) (b)

(c) (d)

Abbildung 2.6: (a) Eine Szene in Wireframeansicht, (b) Abgewickelte *UV-Map* des Modells, (c) Szene mit statischer Beleuchtung gerendert, (d) Gebackene Beleuchtungstextur des Modells.

2.3.3 Schatten in Gameengines

In den verbreiteten Gameengines ist es üblich, Echtzeitschatten und gebackene Schatten zu kombinieren, um das beste Ergebnis zu bieten. Die *Unity 5 Game Engine* nutzt für ihre Echtzeitschatten den *Shadow Maps*-Ansatz [55]. Für statische Objekte wird internes Backen inklusive *globaler Illumination* angeboten. Um den Unterschied bei der Beleuchtung zwischen statischen und dynamischen Objekten weniger sichtbar zu machen setzt *Unity* auf *Light Probes*. Diese Messpunkte können an sinnvollen Positionen in der Szene verteilt werden und fangen dort beim Backen die Lichteinstrahlung auf. Zur Laufzeit wird das Shading der dynamischen Objekte anhand der Position zu den *Light Probes* interpoliert. Dabei geht es jedoch nur um die Beleuchtung des sich bewegenden Objekts selbst, der Schattenwurf auf statische Objekte wird dabei nicht beachtet. In der Version 4.6 verwendete *Unity Dual Lightmaps* [54]. Dabei wird je nach Entfernung von der Kamera zwischen einer Entfernungs- und einer Nähelightmap überblendet, die statischen und dynamischen Objekte werden je nach Entfernung anders behandelt. Sie werfen nah an der Kamera immer Echtzeitschatten, sind sie weiter entfernt benutzen statische Objekte keine Echtzeitbeleuchtung mehr, die dynamischen hingegen schon, werfen jedoch keinerlei Schatten (siehe Abbildung 2.7). Eine simplere Möglichkeit boten die *Single Lightmaps*. Hier wurden gebackene Maps für statische und Echtzeitschatten für dynamische Objekte einfach gleichzeitig angezeigt. Die Deckkraft der Echtzeitschatten musste manuell an die der Lightmaps angepasst werden [54].

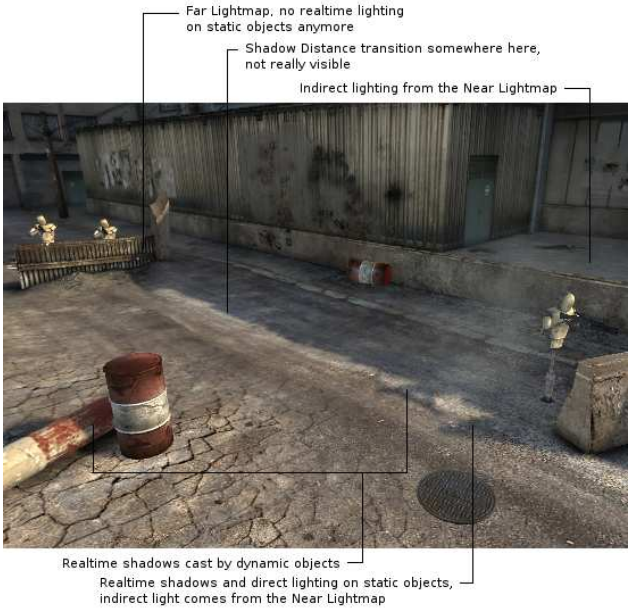
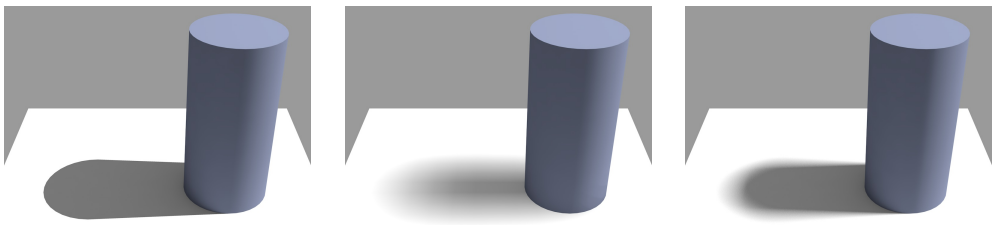


Abbildung 2.7: *Dual Lightmaps* in *Unity 4.6* [54].

Die *Unreal Engine 4* bietet vier grundsätzliche Typen des Schattenwurfs/Lichttypen [59]. *Statische Lichter* haben keinerlei Einfluss auf dynamische Objekte sondern werden nur beim Backen der Szene beachtet. *Gerichtete stationäre Lichter* oder *Whole Scene Shadows* sind vergleichbar mit dem *Dual Lightmap*-Ansatz der *Unity Engine*. Auch hier werden dynamische Schatten abhängig von der Entfernung zur Kamera in statische überblendet. *Stationäre Lichter* werden beim Backen von statischen Objekten mit einbezogen, werden aber zur Laufzeit auch bei der Beschattung dynamischer Objekte beachtet. Hier werden für jedes bewegte Objekt zwei Schatten berechnet, der Schatten der statischen Geometrie auf das Objekt und der des Objekts auf die statische Geometrie. Statische und dynamische Schatten werden schließlich verschmolzen und gleichzeitig angezeigt.

Bei dem letzten Typ handelt es sich um voll *dynamische Lichter*. Sie werfen Echtzeitschatten auf alle Objekte, werden beim Backen nicht beachtet und sind somit die teuersten Lichter in der



(a) Harter Schatten (b) Weicher Schatten (c) Flächenschatten

Abbildung 2.8: Verschiedene Arten des Schattenwurfs.

Berechnung. Um *Flächenschatten* bei beweglichen Objekten effizient anzuzeigen bietet *Unreal Ray Traced Distance Field Soft Shadows* [58]. Bei *Flächenschatten* ist die Härte der Schattenkanten abhängig von der Entfernung zum schattenwerfenden Objekt. Im Gegensatz zu harten Schatten, die gleichmäßig scharfe Kanten haben und *weichen Schatten*, welche durchgängig weiche Kante besitzen, sind *Flächenschatten* nah am Objekt hart und werden auslaufend weicher (siehe Abbildung 2.8). Dieser Schattenwurf ist am realistischsten, jedoch auch am aufwendigsten zu berechnen. *Unreal* verwendet hierfür eine *Distance Field Representation* der Szene [57]. In einem *Distance Field* wird für jedes Objekt in jeden Punkt in einem festgelegten Raster die Entfernung zur Oberfläche des Meshs gespeichert. Um den *Flächenschatten* zu berechnen, wird ein Strahl von dem zu beschattenden Punkt in Richtung Lichtquelle durch das *Distance Field* geschossen. Für sehr geringe zusätzliche Kosten kann nun mithilfe der kleinsten bekannten Entfernung zum schattenwerfenden Objekt ein Kegel ermittelt werden, durch welchen die Härte des Schattens festgelegt wird.

2.4 Interaktion

Die Interaktion ist ein weites Forschungsgebiet in der AR. Hier gelten durch die anderen Voraussetzungen und Möglichkeiten auch teils abgewandelte oder komplett neue Regeln. Man kann die Interaktion in drei große Bereiche einteilen: die Navigation im AR- oder realen Raum, die Manipulation von Objekten und die Kollaboration beziehungsweise gemeinsame Interaktion. Da sich der mögliche Bewegungsraum in der im Zuge dieser Arbeit entwickelten AR-Anwendung lediglich auf den Platz um ein einzelnes 3D-Modell herum beschränkt, liegt kein Fokus auf der Navigation im erweiterten Raum wie zum Beispiel bei Navigationsapps mit GPS-Unterstützung. Die im Prototyp gebotenen Navigationsmöglichkeiten werden im Abschnitt der Konzeptionsphase in *3.4.2 Navigationsmöglichkeiten* erläutert.

2.4.1 Manipulation

Ein spezielles und mächtiges Interaktionskonzept innerhalb der AR sind die sogenannten Tangibles. Dabei handelt es sich um reale Objekte, die vom Nutzer physisch berührt und manipuliert werden können. Diese Veränderungen werden auf verschiedenste Arten, wie durch Kameras, Tracking und weitere Sensoren, erkannt und auf die digitalen Objekte übertragen (siehe Abbildung 2.9a). Der Begriff der *TUI*s, der *tangible user interfaces*, wurde hierfür schon 1997 von Ishii und Ullmer eingeführt [19]. Der große Vorteil dieses Ansatzes liegt in dem leichten Verständnis durch den Nutzer. Wenn man die physischen Objekte mit ähnlichen Eigenschaften versieht, wie sie die digitale Repräsentation haben soll, sprich gleiche Form, gleiches Gewicht, gleiche Oberflächenstruktur, verstehen die User sofort die Verknüpfung zwischen den virtuellen und realen Repräsentationen und erkennen die möglichen Interaktionen. Die erlaubten Manipulationen werden

zusätzlich durch die sogenannten *Affordances* der Objekte schnell erkannt. Dieser Begriff wurde von Don Norman, einem der bekanntesten Interaktionsdesigner, in seinem Buch *The Design of Everyday Things* aus dem Jahr 1988 geprägt und bedeutet, dass die möglichen Interaktionen alleine durch die physischen Eigenschaften eines Gegenstands wahrgenommen werden können, wie zum Beispiel das Herunterdrücken einer Türklinke oder das Rotieren eines Drehknopfes [36]. Durch die Verwendung von *TUI*s kann diese hilfreiche Eigenschaft auch in der AR genutzt werden. Wird nun der *TUI*-Ansatz mit den durch AR gebotenen zusätzlichen digitalen Informationen verschmolzen, entsteht die sogenannte *Tangible Augmented Reality*.

(a) (b)

Abbildung 2.9: (a) *TUI* nach Ishii und Ullmer [19], (b) Durch Gesten steuerbares AR-Schach [12].

Eine der natürlichsten Möglichkeit zur Interaktion ist die klassische Handgestenerkennung [28]. So wurde 2001 zum Beispiel ein AR-Schachspiel präsentiert, das sich mit einem speziell markierten Handschuh steuern ließ [12] (siehe Abbildung 2.9b). Die Steuerung war sehr intuitiv und bereitete keine großen Probleme. Damals war die benötigte Hardware zur Erkennung der Eingaben noch groß und sperrig, es mussten spezielle Trackingmarker und stereoskopische oder eine Vielzahl von Kameras verwendet werden. Durch aktuelle Entwicklungen wie das kleine mobile Gestenerkennungssystem *Leap Motion* [33], erweitern sich die Einsatzmöglichkeiten der Gestensteuerung jedoch stetig und sie wird zunehmend auch für tragbare AR-Anwendungen attraktiver.

Zusätzlich können bei AR-Anwendungen auch alle weiteren üblichen Eingabemöglichkeiten eingesetzt werden. So zum Beispiel eine Sprachsteuerung über einzelne Befehle oder je nach anzeigendem Endgerät Neigungs- und Beschleunigungssensoren. Hat der Nutzer die Hände frei, indem er etwa ein AR-Headset trägt, können auch die Körpergestensteuerung (zum Beispiel mit der *Microsoft Kinect*) oder klassische Eingabegeräte wie Spielecontroller (Gamepad, Joystick, Lenkrad) genutzt werden.

Da das in dieser Arbeit entwickelte AR-System jedoch sehr mobil und ohne zusätzliche Einrichtungs- und Aufbauarbeit von externen Sensoren einsetzbar sein sollte, wird in dem Prototyp die Touchsteuerung über das Display des Endgeräts verwendet. Durch die Erfahrungen mit Smartphones und Tablets sind die Nutzer mit den Grundzügen der Interaktion bereits vertraut.

2.4.2 Gemeinsame Interaktion

Dieser Abschnitt konzentriert sich auf die gemeinsame Interaktion mehrerer Benutzer in einer AR-Welt. Es gab bereits früh wissenschaftliche Arbeiten, die sich mit der Interaktion oder Zusammenarbeit im virtuellen Raum beschäftigten (z.B. *Virtual Notepad* von Poupyrev et al., 1998 [42]; *Studierstube* von Schmalstieg et al., 2000 [48] oder *Boom Chameleon* von Tsang, 2002 [53]). Diese waren jedoch auch aufgrund der damaligen technischen Entwicklung und Umsetzung eher als frühe Prototypen und Machbarkeitsstudien zu verstehen und wenig mobil und zugänglich.

Keines der vorgestellten AR SDKs bietet den bereits eingebauten Mehrbenutzerbetrieb für Interaktionen. Diese Netzwerkfunktionen müssen per SDK oder API manuell implementiert werden. Die *Layar* App *Stiktu* ermöglichte es Benutzern in der realen Welt digitale Grafiken anzubringen und diese mit anderen Nutzern zu teilen, 2013 wurde ihr Betrieb eingestellt [50]. *Second Surface* von Kasahara et al. [23] bietet eine ähnliche Funktion, zusätzlich können hier jedoch mehrere User zum selben Zeitpunkt zusammen an einem digitalen Bild arbeiten. Dieses Bild wird virtuell an reale Oberflächen geheftet, ohne vordefinierte Marker ausdrucken zu müssen. Die Anwendung erlaubt jedoch keinen Import und keine Manipulation von dreidimensionalen Modellen sondern beschränkt sich lediglich auf zweidimensionale Grafiken. Die Implementierung verwendete das *Vuforia* SDK und eine Client-Server-Architektur. Auf dem Client generierter Inhalt wurde über Netzwerk an den Server geschickt, welcher die Daten verwaltete, abspeicherte und an andere Clients weiterschickte. *Scrawl*, eine App des AR Anbieters *String*, bot ähnliche Malfunktionen im dreidimensionalen Raum, jedoch nur mit vordefinierten Markern und ohne geteilte Arbeitsfläche [32].

AR-Applikationen, bei denen der Benutzer dreidimensionale Modelle dynamisch laden und manipulieren und dieses zusätzlich in Echtzeit mit weiteren Nutzern teilen kann existieren bisher nicht.

3 Konzeption des Prototyps

In diesem Abschnitt wird der Prozess der Planung dargestellt. Es wird beschrieben, welche Überlegungen und Entscheidungen im Vorhinein getätigt wurden und welche Funktionen der fertige Prototyp bieten sollte.

3.1 Technische Grundlagen

Die Auswahl der Endgeräte und damit des Betriebssystems hat großen Einfluss auf die Wahl der verfügbaren Entwicklungssoftware. Die Plattform entscheidet zudem über den Ablauf der Entwicklung, die Verfügbarkeit und die unkomplizierte Anwendbarkeit des Prototyps.

3.1.1 Endgeräte und Plattform

Eine wichtige Anforderung an den Prototypen war die Mobilität. Er sollte ohne großen Aufwand transportabel sein und keine spezielle oder teure zusätzliche Hardware benötigen. Für AR nach dem optischen Ansatz ist das Vorhandensein einer Kamera zum Einfangen der Umgebung zwingend erforderlich. Aufgrund dessen fiel die Entscheidung für eine Umsetzung als Applikation für Smartphone und Tablet. Moderne Geräte sind auch in einer günstigen Preisklasse schon ausreichend leistungsstark, haben meist eine Kamera und eine Vielzahl weiterer Sensoren bereits eingebaut und sind weit verbreitet. Auf diese Weise wird es Nutzern ermöglicht, den Prototyp schnell zu erfahren oder auch längerfristig zu verwenden ohne sich eigens für diesen Zweck neue Hardware zulegen zu müssen.

Die am weitest verbreiteten Betriebssysteme auf diesen Geräten sind *Android* (82,8% Marktanteil bei Smartphones im zweiten Quartal 2015), *iOS* (13,9%) und *Windows Phone* (2,6%) [17]. Die starke Verbreitung, Offenheit des Systems und Verfügbarkeit von Geräten in allen Preis- und Leistungsklassen führte zu der Entscheidung, vorrangig das *Android* Betriebssystem zu bedienen.

Mit fortschreitender Entwicklung und Verbreitung weiterer mobiler Endgeräte, ähnlich der mittlerweile eingestellten *Google Glass* [13] oder der *Microsoft HoloLens*, einer speziellen Augmented-Reality-Brille [30], ist prinzipiell auch die Verwendung auf weiteren Plattformen und Geräten denkbar.

3.1.2 Entwicklungsumgebung

Um für *Android* Applikationen zu realisieren gibt es mehrere Möglichkeiten. Apps können mithilfe des *Android-SDKs* programmiert werden, einem Software Development Kit mit zusätzlichen hilfreichen Tools, wie der *Android Debug Bridge ADB* zum Ansehen des Systemlogs des Endgeräts oder einem Emulator. Die verwendete Programmiersprache ist *Java*, weshalb außerdem das *Java Development Kit (JDK)* benötigt wird. Als Entwicklungsumgebung stehen verschiedene Programme zur Auswahl – *Eclipse*, *IntelliJ IDEA*, *NetBeans* oder seit 2015 das von Google entwickelte *Android Studio*. Diese unterscheiden sich nur unwesentlich im Leistungsumfang und können nach persönlicher Präferenz gewählt werden.

Eine weitere Herangehensweise ist die Verwendung einer noch höheren Entwicklungsumgebung, die dem Programmierer viele aufwendige Schritte abnimmt und damit die Arbeit erleichtert. Gerade für das schnelle Prototyping ist dies das Mittel der Wahl. Es können schneller lauffähige Apps erstellt werden und der Fokus liegt an vielen Stellen nicht mehr auf der manuellen grundlegenden Umsetzung sondern auf der Implementierung der erweiterten Funktionen. Bei solchen Umgebungen handelt es sich häufig um Gameengines, die ursprünglich für die Entwicklung und Anzeige von Computerspielen vorgesehen waren. Aktuelle Versionen erlauben es aber heutzutage, die Anwendungen ohne große Anpassungen für verschiedenste Plattformen zu kompilieren,

darunter auch die großen mobilen Betriebssysteme. Zudem sind sie nicht auf die Spieleentwicklung beschränkt sondern ermöglichen das Erstellen vollwertiger Programme jeden Anwendungsbereichs. Die bekanntesten Vertreter dieses Gebiets sind wohl die *Unreal Engine* und die *Unity Game Engine*. Beide Engines basieren auf *WYSIWYG*-Editoren (*„what you see is what you get“*), in denen Objekte in einer zwei- oder dreidimensionalen Welt platziert und manipuliert werden können. Diese werden durch das Hinzufügen vorgefertigter Bausteine oder das Schreiben eigener Skripte mit Funktionen versehen. *Unreal* lässt dem Anwender dabei die Wahl zwischen Code in *C++* oder einer grafischen Programmierung mithilfe eines Node-Editors. In *Unity* werden die Skripte in *C#* oder *JavaScript* geschrieben.

Beide Engines ähneln sich in Programmoberfläche und Konzepten sehr stark und sind im Funktionsumfang vergleichbar (siehe Abbildung 3.1). Beide können zum Beispiel mit Modellen, Shadern und Plugins aus dem *Unreal Marketplace* beziehungsweise dem *Unity Asset Store* erweitert werden. Als Stärke der *Unreal Engine* wird im Allgemeinen die grafisch hohe Qualität und Performanz gesehen, die *Unity Engine* punktet durch erhöhte Einsteigerfreundlichkeit. Aufgrund der Verfügbarkeit der gewählten AR-Library sowie bereits gewonnener Erfahrung bei der Entwicklung mit dieser Engine, fiel die Wahl für die Umsetzung des Prototyps auf die *Unity Game Engine*.

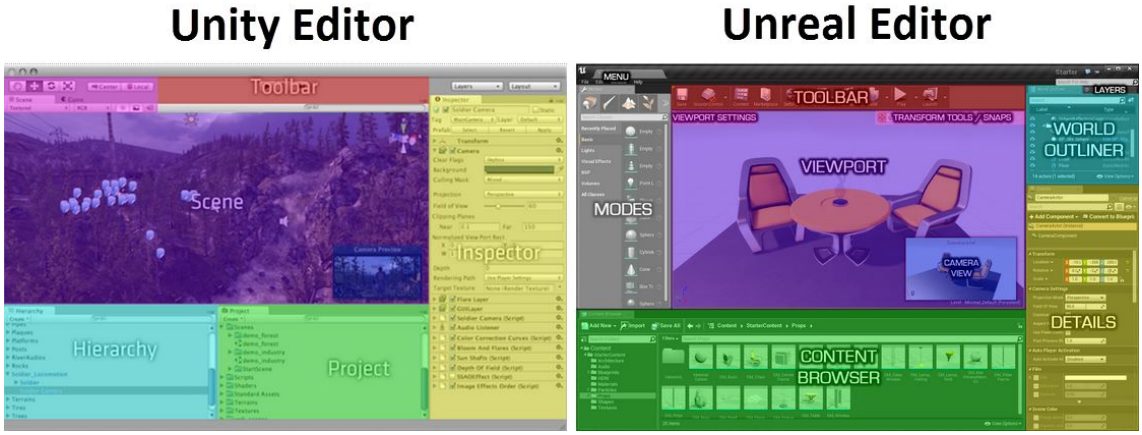


Abbildung 3.1: Vergleich der Programmoberflächen von *Unity* und *Unreal* [60].

3.2 Tracking

Die Entscheidung für eine AR-Library sowie die Art des Trackings ist von mehreren Faktoren abhängig. Dazu gehören die Zielplattform, die Einbindung in das Softwarepaket der Entwicklung sowie die Anforderungen an Vorbereitungsaufwand, Robustheit und die Anwendungsfälle des Trackings. Auch eine ausführliche Dokumentation der Library und eine aktive Internetcommunity kann ein ausschlaggebender Faktor sein.

3.2.1 Augmented Reality Library

Wie in Kapitel *2.1 Augmented Reality* vorgestellt, gibt es für die Entwicklung von AR-Apps verschiedenste Anwendungen und Libraries. Vorgefertigte Editoren, wie der *Metaio Creator*, die es auch dem Laien erlauben in wenigen Schritten Objekte auf optischen Trackern anzuzeigen sind durch ihre Eingeschränktheit in Erweiterbarkeit und Funktionen für diesen Prototyp nicht geeignet. Das Paket muss eine direkte Programmierschnittstelle (API) bieten, die zudem in einer Programmiersprache verfügbar ist, welche von der Entwicklungsumgebung verstanden wird.

Das *Layar* SDK ist kostenpflichtig und nur in einer 30-tägigen Testversion frei verfügbar. Aus diesem Grund ist es für diese Arbeit nicht nutzbar. *Metaio* hat wie bereits erwähnt den Betrieb

eingestellt, wodurch auch dieses SDK aus der Auswahl fällt. *Wikitude* besitzt durch ein Plugin eine direkte Anbindung an *Unity*, die freie Version zeigt jedoch ein auffälliges und sehr störendes Wasserzeichen auf der kompletten Kameraansicht an. Die Wahl der Augmented Reality Library fiel deshalb auf das *Vuforia Framework*. Die Starter-Lizenz bietet vollen Zugriff auf die *Vuforia*-Plattform, ein Wasserzeichen wird lediglich in der unteren linken Ecke angezeigt und beeinflusst die Erfahrung nicht negativ. Es gibt eine große Auswahl an Tracking- und Markerarten, das Tracking ist zudem sehr zuverlässig. Das SDK gibt es für direkt für *Android* (*Java/C++*) und *iOS* (*C++*), außerdem als *Unity*-Plugin (*C#*). Die ausführliche Onlinedokumentation sowie eine aktive Community im eigenen Forum sind weitere Gründe, die für dieses Framework sprachen und zu dessen Wahl führten.

3.2.2 Art des Trackings

Vuforia bietet sieben Arten des Trackings mit verschiedensten Markern [64]. Dabei handelt es sich um die folgenden:

- ***Image Targets***: die klassischen flachen 2D-Marker, verwendet zum Beispiel in Printmedien. Diese können selbst definiert werden. Am Besten eignen sich Grafiken mit einer ausreichenden und markanten Verteilung an sogenannten *Features*, automatisch erkannten Trackingpunkten, die für robuste Bildverfolgung unverzichtbar sind.
- ***Multi-Targets***: eine Kombination aus mehreren *Image Targets*, welche zu einander in bestimmten räumlichen Relationen stehen. So können zum Beispiel die verschiedenen Seiten einer Verpackungsschachtel gemeinsam als Box definiert sein.
- ***Cylinder Targets***: hierbei handelt es sich um *Image Targets* welche jedoch nicht planar auf einer ebenen Fläche liegen, sondern gebogen die Form eines Zylinders ergeben. Das ist nützlich, um Bilder auf runden Objekten wie Flaschen oder Dosen zu tracken.
- ***Frame Markers***: dies sind vordefinierte Rahmen, vergleichbar eines QR- oder Barcodes, welche um jedes beliebige Bild gelegt werden können. Sie kodieren eine von 512 Zahlen und können über diese zugeordnet werden. Durch die geringe Komplexität des optischen Codes können die Tracker relativ klein sein und es können mehrere gleichzeitig erkannt und verfolgt werden.
- ***Text Recognition***: erkennt Wörter aus einem festgelegten Wörterbuch. Es umfasst rund 100 000 englische Wörter.
- ***Object Targets***: *Vuforia* erlaubt zudem das Tracking von realen Objekten. Dafür muss das Objekt zunächst mit einer eigens hierfür vorgesehenen App gescannt werden, indem es von allen Seiten aufgenommen wird. Durch die Verteilung der *Features* kann eine 3D-Hülle berechnet werden, die sich der tatsächlichen Form annähert. Diese Hülle kann schließlich für das Tracking des Objekts genutzt werden (siehe Abbildung 3.2).

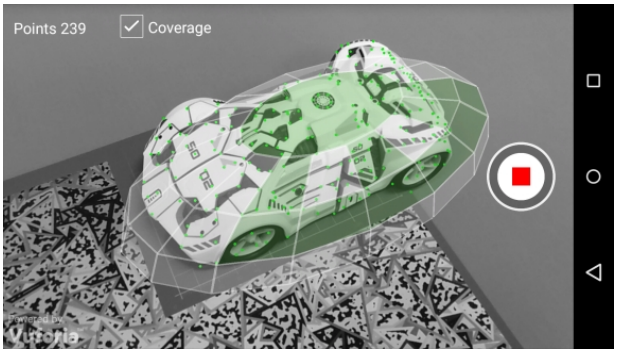


Abbildung 3.2: Mit dem *Object Scanner* erzeugte Trackinghülle eines realen Modells [67].

- ***Smart Terrain***: eine neuartige Technologie, welche es zur Laufzeit ermöglicht, ein 3D-Mesh der physischen Umgebung zu erstellen und dieses in Anwendungen zu nutzen (siehe Abbildung 3.3). Dadurch lassen sich viele neue Interaktionsweisen zwischen der erweiterten und der echten Realität umsetzen. So kann zum Beispiel durch die Positionierung und Verschiebung von realen Objekten Einfluss auf eine virtuelle Spielwelt genommen werden.

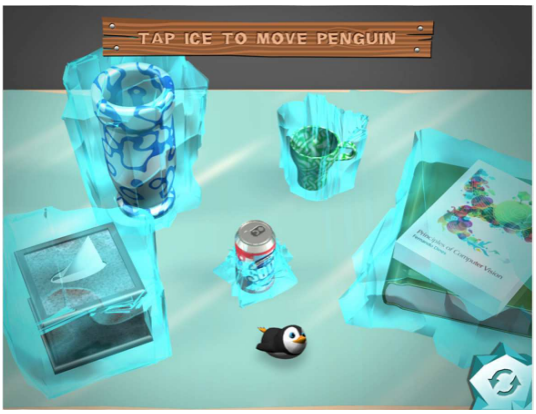


Abbildung 3.3: Einbindung der realen Umgebung in die Spielwelt mithilfe von *Smart Terrain* [65].

Vuforia arbeitet beim Tracking ohne Unterstützung des Gyroskops des Endgeräts, es handelt sich also um klassisches optisches Tracking. Für den Prototyp wurde das Tracking mit *Image Targets* gewählt, da es den Anforderungen am Besten entspricht. Virtuelle Objekte sollen auf planen Oberflächen angezeigt werden, wie einem Tisch oder dem Boden. Als Target werden selbst definierte austauschbare Grafiken verwendet. Die Interaktion findet auf dem Display des Endgeräts statt, die Erfassung von realen Objekten (*Object Targets*) oder der räumlichen Umgebung (*Smart Terrain*) ist somit nicht notwendig.

3.3 Licht und Schatten

Die Beleuchtung, insbesondere die Schattierung, ist ein wichtiger Teil dieser Arbeit. Um Erkenntnisse darüber zu gewinnen, welche Art der Beleuchtung, gebacken oder dynamisch, für welchen Anwendungsfall besonders geeignet ist, müssen beide in dem Prototypen implementiert sein. Es soll nicht das Ziel sein, die tatsächliche reale Lichtsituation und Umgebung so gut wie möglich einzufangen und die Objekte so realistisch wie möglich darzustellen. Hierfür sind nach aktuellem Stand der Technik entweder eine Vorkalibrierung durch Shading-Probes oder das Erstellen von HDRs erforderlich, zusätzlich leistungsstärkere Hardware wie vollwertige Computer mit Grafikkarte (siehe dazu *2.3.1 Beleuchtung in Augmented Reality*). Dieser Prototyp soll jedoch mobil sein und ohne umständliche Vorarbeit verwendet werden können. Der Schwerpunkt liegt deshalb auf der Interaktion, den Netzwerkfunktionen und eben der optimalen Art der Beleuchtung in den jeweiligen Anwendungsfällen.

Von den zu untersuchenden dreidimensionalen Modellen werden je drei verschiedene Versionen angefertigt: komplett mit vorgebackenen Schatten, mit rein dynamischen Schatten und mit der Kombination beider Ansätze. Die statische Version wird in einer 3D-Software ausgeleuchtet und gebacken, die dynamische Version nutzt die Lichter der *Unity Engine*. Beides wird schließlich zusätzlich kombiniert.

3.4 Interaktion

Ein weiterer Schwerpunkt der Entwicklung ist die Interaktion der Nutzer mit der Anwendung und auch untereinander. In diesem Abschnitt wird beschrieben, auf welche Weise der Benutzer mit der App interagieren kann und welche Funktionen sie ihm bieten soll.

Grundsätzlich soll es dem Nutzer ermöglicht werden, durch Touchbedienung Manipulationen an dynamisch austausch- und ladbaren Modellen vorzunehmen. Durch Verwendung des AR-Ansatzes werden diese Modelle an einer bestimmten Stelle, etwa einem ausgedruckten Marker, in der realen Welt angezeigt. Aufgrund der Betrachtbarkeit von allen Seiten durch das Endgerät als „Fenster“ soll die Erfahrbarkeit des Objekts im Vergleich zu einer einfachen per Eingaben drehbaren Ansicht verbessert werden. Zusätzlich können die Manipulationen an beliebig viele weitere Nutzer über Netzwerk gestreamt werden, sodass auch sie den Vorgängen aus ihrer eigenen Sicht ungestört folgen können. Damit sind zum Beispiel attraktive Produktpräsentationen auf Messen oder bei Kunden vorstellbar.

3.4.1 Benutzerrollen

Das entwickelte Konzept sieht vor, die Nutzer in zwei Kategorien mit unterschiedlichen Rechten und Funktionen einzuteilen:

- ***Presenter*** oder *Präsentator*
- ***Spectator*** oder *Zuschauer*

Dem Namen entsprechend, führt der *Presenter* durch die Präsentation. Er hat die Rolle des Leiters inne und hat somit die meisten Rechte. Er wählt das gewünschte Modell aus einer lokalen Datenbank auf seinem Gerät aus und führt die Manipulationen durch. Die *Spectators* können sich mit einem eigenen Endgerät in die Präsentation einloggen. Ab diesem Zeitpunkt werden jegliche Änderungen auf Seite des *Presenters*, wie das Laden und Bewegen der Modelle, an den *Spectator* übertragen und auf seiner Seite angezeigt. Dieser folgt der Präsentation aus seiner eigenen gewünschten Sicht und bleibt durch die aktive Einbindung aufmerksam. Bei einer Beschreibung als Client-Server-Architektur entspricht der *Presenter* dem Server, der den Zustand der Welt kontrolliert und verwaltet, der *Spectator* ist ein Client.

3.4.2 Navigationsmöglichkeiten

Die Bewegung in der virtuellen Welt geschieht bei AR normalerweise durch die Veränderung der Positionen des Trackers und der Kamera zueinander. Der ausgedruckte 2D-Tracker gilt als Fixpunkt der digitalen Szene. Er bleibt auf einer planen Ebene liegen, kann aber in diesem Fall auch manuell physisch bewegt werden. Das gesamte Koordinatensystem orientiert sich an seiner Position. Wichtig ist auch, dass die Perspektive des *Presenters* und der *Spectators* komplett entkoppelt sind, jeder soll die Szene aus seiner präferierten und anpassbaren Ansicht erleben. Möchte der Nutzer seine Sicht auf das Modell ändern hat er folgende Möglichkeiten:

- ***Ansicht verschieben***: Um die Szene aus einer anderen Perspektive aber aus der gleichen Richtung zu betrachten hat der Nutzer zwei Möglichkeiten. Er kann das mobile Endgerät, durch dessen Kamera er die Szene beobachtet nach rechts oder links, nach oben oder unten bewegen. Da das Modell an den Tracker gebunden ist, verändert sich seine Position auf dem Bildschirm entsprechend. Für die horizontale Bewegung gibt es außerdem die Herangehensweise, den ausgedruckten Tracker selbst nach rechts oder links zu verschieben. Der Effekt ist der selbe wie bei der Bewegung der Kamera.

- ***Ansicht zoomen***: Um das Modell aus der Nähe oder Ferne zu betrachten sollte man die Entfernung der Kamera zum Tracker verändern. Bewegt man sich mit ihr auf die Ebene zu, wird die Ansicht gleichermaßen vergrößert, um mehr Details wahrnehmen zu können. Tritt man etwas zurück, erhält man einen Überblick über die virtuelle Welt. Das Verschieben des Trackers ist hier nicht zu empfehlen, da er dafür angehoben werden müsste, was das Biegen und den eventuellen Verlust des Trackings zufolge hätte.

- ***Ansicht rotieren***: Um die Richtung, aus der der Nutzer die Szene betrachtet zu ändern, gibt es wiederum zwei Ansätze. Zunächst kann man sich mit dem Endgerät in der Hand um den Tracker herumbewegen, die Sicht auf das 3D-Modell passt sich gleichermaßen an. Dies ist jedoch nicht immer praktikabel, da es häufig nicht die Möglichkeit gibt, sich um 360° um den 2D-Tracker herumzubewegen, zum Beispiel an einem Sitzplatz an einem Tisch. Häufig ist es einfacher, per Hand den echten physischen Tracker einfach zu drehen und damit seine Sicht auf die Welt anzupassen.

Die Umsetzung dieser Interaktionsmöglichkeiten ist direkt in der AR-Grundfunktionalität von *Vuforia* enthalten. Sie müssen deshalb nicht manuell implementiert werden.

3.4.3 Manipulationsmöglichkeiten

Der Benutzer soll die Möglichkeit haben, einzelne Objekte des Modells zu Manipulieren. Die Eingabe erfolgt durch Touch-Gesten auf dem Display des Endgeräts. Die grundlegende Art und der zulässige Bereich der erlaubten Manipulation werden schon bei der Erstellung des 3D-Modells definiert und beim Laden umgesetzt. Diese Bewegungen sind auf Seiten des *Presenters* vorgesehen und werden über Netzwerk an die eingeloggten *Spectators* übertragen. Der Prototyp soll die drei Grundtransformationen erlauben:

- ***Translation***: Verschiebung auf vordefinierten Achsen zwischen festgelegten Start- und Endpunkten. Anwendungsbeispiel Schieberegler.

- ***Rotation***: Drehung um vordefinierte Achsen um einen festgelegten Punkt. Anwendungsbeispiel Lautstärkedrehregler.

- ***Skalierung***: Vergrößerung und Verkleinerung auf vordefinierten Achsen. Anwendungsbeispiel elastisches Material.

Auch die beliebige Kombination der Grundmanipulationen ist erlaubt. Zusätzlich soll das Modell in seiner Gesamtheit auf jedem Endgerät unabhängig skalierbar sein. Der Grund hierfür ist die individuelle Anpassung an die uneinheitlichen Größen der Displays unterschiedlichster Hardware. Abseits der Grundtransformationen soll es möglich sein, das ***Material eines Objekts per Touch zu ändern***.

3.4.4 Dynamisches Laden und Verteilen der Modelle

Für die vielseitige und längerfristige Verwendung ist es erforderlich, die Modelle austauschen zu können. Nur bestimmte dreidimensionale Modelle bei der Entwicklung „hardcoded“ fest einzubauen und diese nicht veränderbar zu machen ist keine sinnvolle Option, die Anwendungsfälle des Prototyps wären dadurch sehr eingeschränkt. Es soll deshalb die Möglichkeit geben, die Anwendung stets mit neuen Inhalten zu versorgen und diese dynamisch zu laden. Hierfür sollen die Modelle zur Laufzeit aus Dateien auf dem nichtflüchtigen Datenspeicher des Endgeräts ausgelesen werden. Die Auswahl des Modells geschieht über einen in die App integrierten Objektbrowser. Da die erlaubten Manipulationen bereits in der Modelldatei gespeichert werden sollen, muss der Ladevorgang dynamisch sein. Das Objekt wird dabei analysiert und die benötigten Vorgänge für die definierten Transformationen ausgeführt.

Zudem ist es nicht ausreichend, das Modell nur auf dem lokalen Gerät zu Laden. Auch die angemeldeten Clients müssen im Besitz der Objektdatei sein, um diese anzeigen zu können. Da es weiteren *Spectators* möglich sein soll an einer Präsentation teilzunehmen, ohne im Vorhinein eine Datei manuell auf den persistenten Speicher ihres Geräts kopieren zu müssen, soll die Verteilung der Datei automatisch erfolgen. Benutzer, die sich bei dem Server registrieren, sollen das dort geladene Modell bei Nichtbesitz über die Netzwerkverbindung transferiert bekommen. Die Details zur genauen Umsetzung des Ladens und Verteilens werden in den Kapiteln *4.4.3 Dynamisches Laden* und *4.4.4 Netzwerkkommunikation* erläutert.

4 Implementierung des Prototyps

Dieses Kapitel beschreibt die technischen und gestalterischen Details der Umsetzung. Welche Hard- und Software wurde benutzt? Wie wurden einzelne Aspekte konkret umgesetzt? Welche Probleme ergaben sich im Laufe der Entwicklung und wie wurden sie gelöst?

4.1 Entwicklungsumgebung

In diesem Teil wird die Hard- und Software beschrieben, die für die Umsetzung verwendet wurde. Der Prototyp wurde auf zwei verschiedenen Desktop-PCs mit folgenden Spezifikationen entwickelt:

- *PC 1*: Intel Core i7-6700K 4.00GHz, 16GB RAM, Nvidia GeForce GTX 970, Windows 8.1 Pro 64 Bit
- *PC 2*: Intel Core i7-4790K 4.00GHz, 32GB RAM, 2x Nvidia GeForce GTX 980 Ti, Windows 7 Pro 64 Bit

Die Entwicklungssoftware wurde auf beiden Geräten auf dem selben Stand gehalten. Verwendet wurde *Unity* in der Version 5.3.1p3 Personal 32 Bit inklusive *MonoDevelop* 5.9.6 als Codeeditor. Bei dem benutzten *Vuforia*-SDK handelte es sich um die Version 5.0.10. Es gab bereits zu Anfang der Umsetzung schon aktuellere Versionen beider einzelner Softwarepakete, jedoch war die garantierte Kompatibilität nur bei den verwendeten gegeben. Aufgrund dessen wurden diese Versionen benutzt und im Laufe des Projekts auch nicht aktualisiert. Zur Synchronisation der Daten auf beiden Entwicklungs-PCs wurde ein *GitHub*-Repository eingerichtet und der Stand mit dem *GitHub Desktop*-Client abgeglichen.

Als mobile Hardware wurden drei *Android*-Endgeräte mit Touchbedienung verwendet. Dabei handelte es sich um ein Smartphone und zwei baugleiche Tablets mit den folgenden Spezifikationen:

- *Smartphone*: Google Nexus 5 LG D821, Qualcomm Snapdragon S800 2.26GHz, 2GB RAM, Displaydiagonale 4.95 Zoll bei 1920x1080 Pixeln, Android-Version 6.0.1
- *Tablet*: Nvidia SHIELD Tablet K1, Nvidia Tegra K1 Grafikprozessor, ARM Cortex A15 CPU 2.2GHz, 2GB RAM, Displaydiagonale 8 Zoll bei 1920x1200 Pixeln, Android-Version 6.0

4.2 Erstellung der 3D-Inhalte

Neben der Programmierung der Anwendung war die Erstellung von geeigneten 3D-Modellen ein weiterer Schwerpunkt. Hier wird beschrieben, wie diese entstanden, wie die Interaktionsmöglichkeiten direkt integriert wurden, welche Typen von Modellen benötigt wurden und wie der Export in die *Unity Engine* geschah.

4.2.1 Software

Als 3D-Software wurde *Maxon Cinema 4D* verwendet. Hierbei handelt es sich um ein vielseitiges Werkzeug für Modellierung, Animation und Rendering. Die benutzten Versionen waren R15.064 64 Bit (siehe Abbildung 4.1) und R17.048 64 Bit. Zwar lassen sich mit dem enthaltenen *Advanced Renderer* schon hochqualitative Ergebnisse erzielen, performanter und leistungsfähiger sind allerdings externe Renderengines. Für das Backen der statischen Beleuchtungsmaps wurde deshalb der GPU-Renderer *otoy OctaneRender v3* genutzt. Die Integration in *Cinema 4D* ermöglichte das *OctaneRender for Cinema 4D*-Plugin.

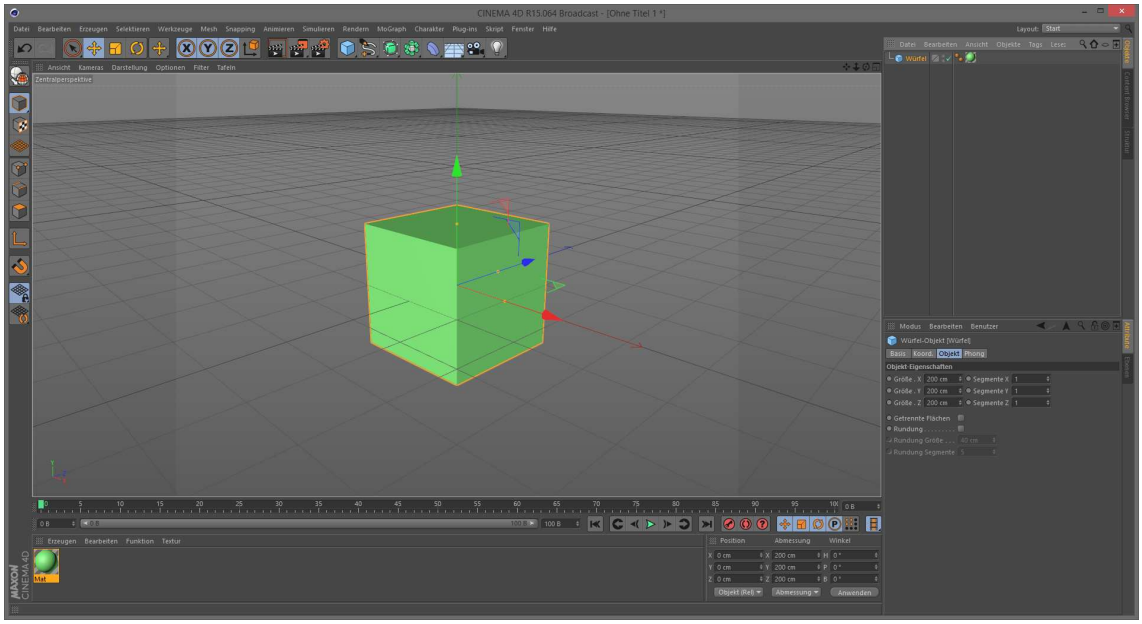


Abbildung 4.1: Die Programmoberfläche von *Maxon Cinema 4D R15*.

4.2.2 Objekthierarchie

Um den Austausch von Modellen in der Anwendung auch nach Abschluss der Entwicklung zu erlauben, war es nötig, die Objekte dynamisch zu Laden. Die erlaubten Transformationen durch den *Presenter* sind deshalb bereits in der Modelldatei enthalten. Während des Ladevorgangs wird die Objekthierarchie zur Laufzeit analysiert und anhand ihrer werden Manipulationen initialisiert und so ermöglicht. Neben der Art der Transformation wird auch der erlaubte Umfang auf diese Weise definiert. Im Folgenden werden die Ansprüche an die Objekthierarchie für einen reibungslosen Import der Manipulationen erläutert, der technische Ablauf des Ladevorgangs in der Software selbst wird in *4.4.3 Dynamisches Laden* beschrieben. Grundsätzlich muss das Objekt, welches manipulierbar sein soll, mindestens ein Unterobjekt enthalten. Am geeignetsten ist hierfür ein *Nullobjekt* das keine Geometrie besitzt sondern lediglich durch Koordinaten und Name definiert ist. Über Schlüsselwörter in dessen Namen wird dann zwischen den verschiedenen Typen von Funktionen unterschieden:

- ***Translation***: Enthält der Name des Unterobjekts das Schlüsselwort *„Translation“* ermöglicht es das Verschieben des Oberobjekts per ***Wischgeste mit einem Finger***. Über die Position des Objekts wird der erlaubte Bewegungsbereich definiert. Dabei gibt es zwei unterschiedliche Ausführungen: die Definition über ein einzelnes sowie über zwei Unterobjekte (siehe Abbildung 4.2). Wird nur ein Objekt verwendet, darf der *Presenter* das Oberobjekt später zwischen der aktuellen Startposition und der Position des Unterobjekts in gerader Linie hin- und herbewegen. Im konkreten Fall auf Abbildung 4.2 lässt sich *Objekt 2* zwischen seiner eigenen und der Position des Objekts *Translation* verschieben. Bei zwei Unterobjekten ist diese Bewegungslinie zwischen den Positionen ebendieser beiden definiert. *Objekt 1* kann zwischen *Translation 1* und *Translation 2* entlang der Z-Achse bewegt werden. Allgemein ist die Bewegung zudem nicht nur entlang einer einzelnen Achse möglich, es handelt sich also nicht nur um eine Änderung beispielsweise der Z-Position. Durch die Definition der Bewegungsendpunkte über die Objektpositionen lässt sich die Linie beliebig in den dreidimensionalen Raum legen und ist immer relativ zum lokalen Koordinatensystem des Oberobjekts. Bei einer Skalierung des gesamten Modells passen sich die Endpunkte infolgedessen korrekt an.

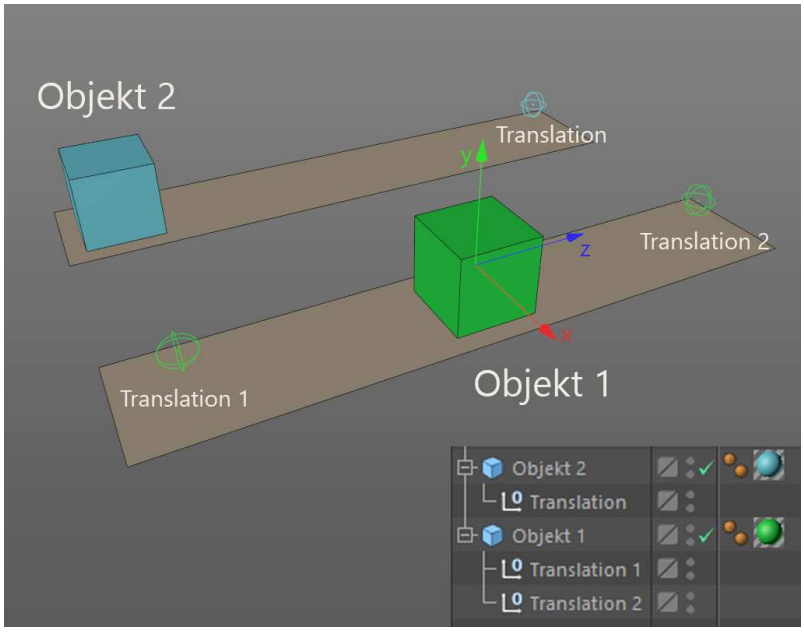


Abbildung 4.2: Die benötigte Objekthierarchie für die Translation in zwei Ausführungen.

- ***Rotation***: Durch das Schlüsselwort *„Rotation“* im Namen des Unterobjekts wird die Rotation des Oberobjekts durch ***Wischgeste mit einem Finger*** aktiviert. Hierbei wird die Position des Unterobjekts als Drehpunkt der Rotation definiert. Durch das Hinzufügen des Achsennamens im Objektnamen wird die Drehachse festgelegt. In Abbildung 4.3 lässt sich *Objekt 1* um die Y-Achse drehen, der Drehpunkt befindet sich an der lokalen Position von *Rotation Y*. Zum Ermöglichen der Drehung um alle Achsen müsste *Rotation Y* lediglich in *Rotation X Y Z* umbenannt werden.

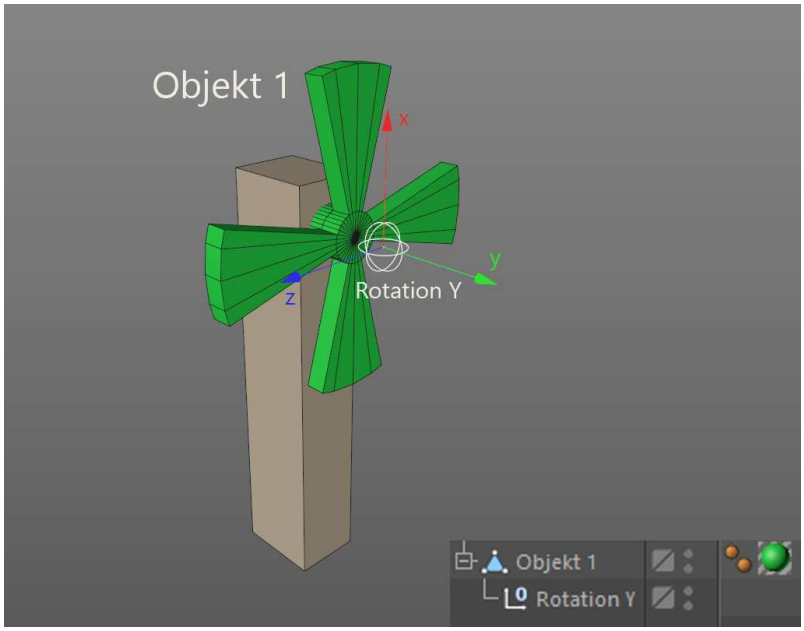


Abbildung 4.3: Die benötigte Objekthierarchie für die Rotation.

- ***Skalierung***: Enthält der Name des Unterobjekts das Schlüsselwort *„Scale“* lässt sich das Oberobjekt per ***Pinch-To-Zoom-Geste mit zwei Fingern*** skalieren. Die Position des Unterobjekts ist dabei der Ursprung der Skalierung. Ähnlich wie bei der Rotation wird über Angabe der Achse im Namen definiert, auf welchen Achsen schließlich skaliert wird. Bei der Hierarchie in Abbildung 4.4 kann der Nutzer *Objekt 1* um alle Achsen vergrößern und verkleinern, ausgehend von der Position des Nullobjekts *Scale X Y Z* in der vorderen unteren Ecke des Würfels als Ursprung.

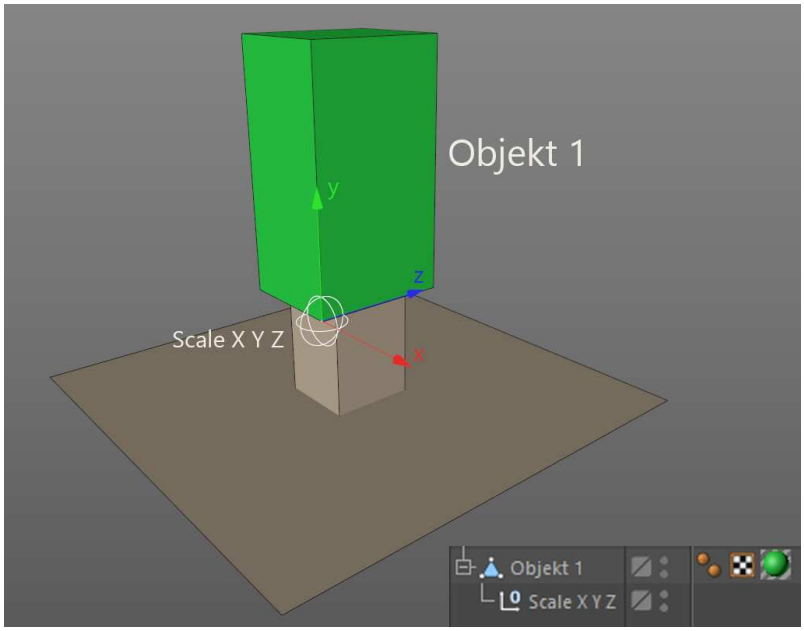


Abbildung 4.4: Die benötigte Objekthierarchie für die Skalierung.

- ***Kombination***: Zusätzlich ist die beliebige Kombination der erlaubten Transformationen möglich. Hierfür muss für jede Manipulation ein eigenes Unterobjekt erstellt werden, dabei gelten die Regeln der einzelnen Transformationen. Das *Objekt 1* in Abbildung 4.5 kann durch die Unterobjekte entlang der Schiene verschoben werden, um die Z-Achse gedreht und auf allen Achsen skaliert werden.

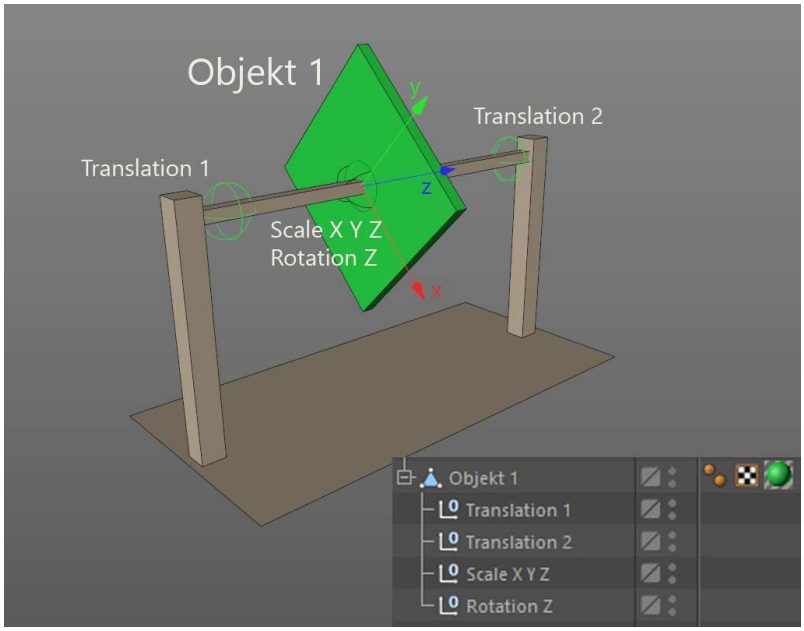


Abbildung 4.5: Die benötigte Objekthierarchie für eine Kombination der Manipulationen.

- ***Sonderbefehle***: Nicht nur die Transformationen können im Vorhinein definiert werden. Es gibt zwei weitere Schlüsselwörter, welche in den Namen eines Objekts eingefügt werden können und beim Ladevorgang bestimmte Aktionen auslösen. Die Zeichenfolge *„_delete_“* löscht ein Objekt beim Importieren. Das ist hilfreich, falls man ein Objekt in der Modelldatei behalten will, es jedoch später nicht in dem Modell sichtbar haben möchte. Der Sonderbefehl *„_noShadow_“* deaktiviert den Schattenwurf. In Abbildung 4.6 wird *Objekt 1 _delete_* beim Laden gelöscht, *Objekt 2 _noShadow_* wirft keinen Schatten.

Abbildung 4.6: Die zwei Sonderbefehle.

Zusätzlich zu Translation, Rotation und Skalierung gibt es die Möglichkeit, ein Objekt mit verschiedenen ***Materialien*** zu versehen und bei der Ausführung ***per Tap*** zwischen ihnen zu wechseln. Da die Materialien von Objekten in *Unity* selbst final angepasst werden, um dort direkt die Optik in der Engine überprüfen zu können, wird diese Funktion nicht über die Objekthierarchie in der Modelliersoftware definiert sondern beim Export aus *Unity* heraus in die *AssetBundles*. Siehe dazu *4.2.5 Export in AssetBundles*.

4.2.3 Testmodelle

Für die Entwicklung der Anwendung wurden zunächst drei unterschiedliche Testmodelle entworfen. Anhand dieser Modelle wurden die einzelnen Funktionen implementiert und getestet. Jedes Szenario hatte dabei einen eigenen Schwerpunkt:

- ***Der Interaktionsspielplatz***: Dies war das erste Testmodell (siehe Abbildung 4.7). An ihm können alle ***drei Grundtransformationen*** ausgeführt werden. Die grünen Objekte sind interaktiv und heben sich optisch vom Rest der Geometrie ab. Der Quader im Vordergrund lässt sich skalieren, der windradähnliche Rotor erlaubt dem Nutzer die Rotation. Die beiden Quader im Hintergrund sind innerhalb der Führungsschienen verschiebbar und dienen dem Testen der Translation. Eine Schiene wurde leicht geneigt, so dass die Funktionalität der relativen Verschiebung überprüft werden konnte. Dieses Objekt muss sich bei korrekter Umsetzung nicht nur auf einer einzelnen absoluten Weltachse bewegen (z.B. lediglich Veränderung des X-Wertes) sondern auf einer gekippten Linie (z.B. Veränderung des X- und Z-Wertes). Zusätzlich wurde hier mit der Kombination von vorgebackenen Schatten auf der statischen Geometrie mit den dynamischen Schatten der beweglichen Objekte experimentiert.

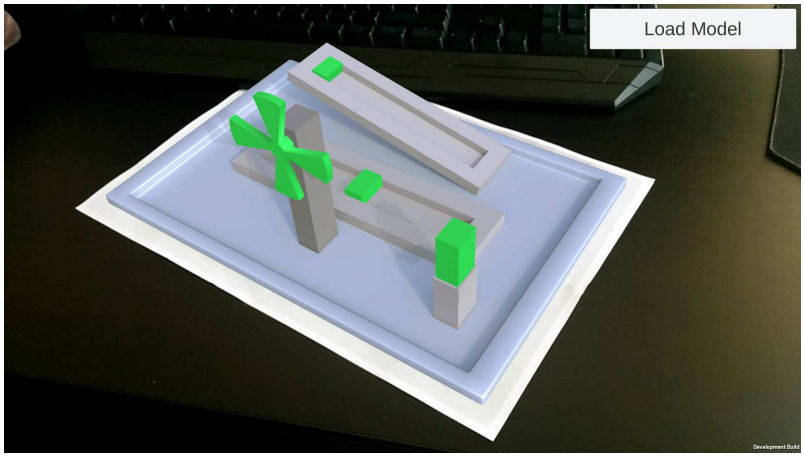


Abbildung 4.7: Der Interaktionsspielplatz

- ***Die Kombination***: Das Ziel dieses Modell war es, die Korrektheit der ***Kombination der Grundtransformationen*** sicherzustellen (siehe Abbildung 4.8). Die beiden grünen Scheiben lassen sich auf den Schienen verschieben, um ihren Mittelpunkt drehen und außerdem skalieren. Wie die Kombination schließlich im Detail implementiert wurde ist nachzulesen in *4.4.3 Dynamisches Laden*.

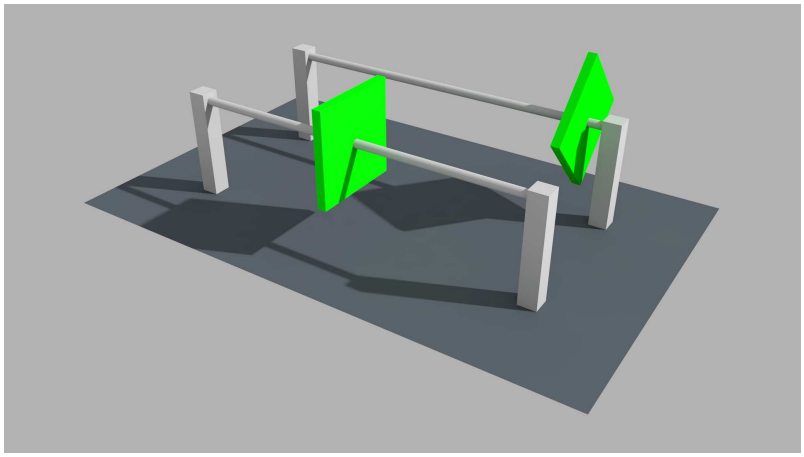


Abbildung 4.8: Die Transformationskombination

- ***Das realistische Haus***: Bei dieser Szene ging es vorrangig um die ***Optik*** und die Optimierung des Prozesses des Backens. Hier wurden die optischen Möglichkeiten ausgelotet. An diesem komplexeren Mesh wurde zunächst der Ablauf des Abwickelns der UV-Koordinaten getestet, anschließend die verschiedenen Parameter des Backens festgelegt, wie etwa eine sinnvolle Größe der Texturen (bei großen Objekten ist eine Auflösung von 2048x2048 Pixeln meist ausreichend). Auch wurden verschiedene Einstellungen der *Unity*-Shader erprobt (etwa Werte und Texturen für *Albedo, Metallic, Smoothness* und *Normalmap*).

Abbildung 4.9: Das Haus

Für die Benutzerstudie wurden später drei weitere Szenarien mit verschiedenen Anwendungsfällen erstellt, mehr dazu in *5.1 Konzeption*.

4.2.4 Backen der Beleuchtung

Im Folgenden wird der Vorgang beschrieben, der durchlaufen wurde, um hochqualitative beleuchtete vorgebackene Texturen für die 3D-Modelle zu erhalten. Der erste Schritt ist das Umwandeln aller parametrischen Objekte zu klassischen Polygonobjekten. Erst danach kann das Mesh abgewickelt werden, um sinnvolle UV-Maps zu generieren. Bei simpler Geometrie, wie zum

Beispiel Würfeln, wurde die *Cinema 4D*-interne Abwickelfunktion benutzt. Bei komplizierteren Objekten empfiehlt es sich jedoch, eine Software zu verwenden, die in diesem Bereich mehr Funktionen und einen angenehmeren Workflow bietet. In diesem Fall erwies sich *3D-COAT 4.5.19* als die richtige Wahl. Der Transfer der Geometrie in die Software und wieder aus ihr heraus geschah über das *AppLink 3D-Coat*-Plugin. Nach diesem Schritt kann mit dem Backen begonnen werden.

Wie zuvor erwähnt wurde aufgrund der besseren Renderzeit und der höheren Qualität der externe GPU-Renderer *Octane* verwendet. Um mit diesem Texturen zu backen müssen die Szene und die Objekte wie folgt konfiguriert sein:

- ***Benötigte Objekte und Tags***:

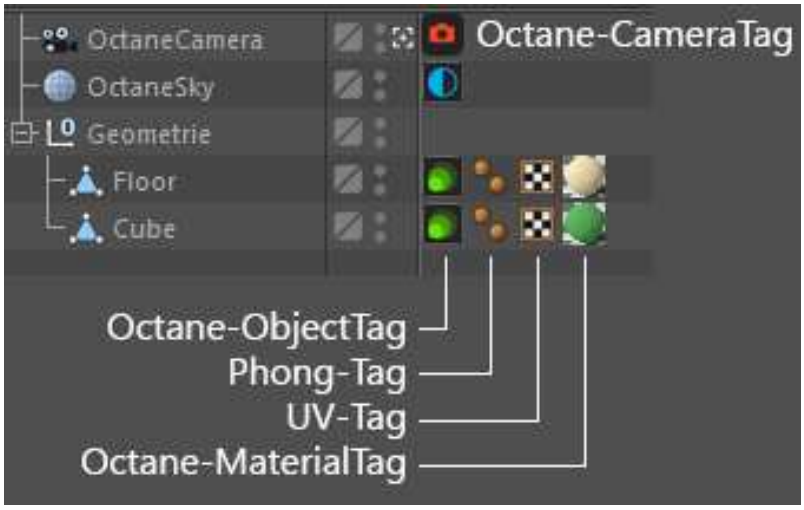


Abbildung 4.10: Objekthierarchie und Tags

Abbildung 4.10 zeigt die benötigten Objekte mit ihren dazugehörigen Tags. Tags sind in *Cinema 4D* Sammlungen von zusätzlichen Funktionen und Eigenschaften, mit denen Objekte versehen werden können. So werden zum Beispiel das Material oder das UV-Mapping über Tags definiert. Zusätzlich bietet das *Octane*-Plugin eigene Tags.

Die zu backenden Modelle sind hier **Floor** und **Cube**. Sie besitzen je vier Tags. Das ***Octane*-ObjectTag** definiert bestimmte Eigenschaften, die für die Renderengine nützlich sind. Weitere Details zu den Einstellungen folgen im eigenen Unterpunkt. Das **Phong-Tag** erlaubt Feinjustierung des Phong-Shadings, also ab welchem Grad ein Winkel als Rundung angesehen und gezeichnet wird anstatt als harte Kante. Das **UV-Tag** speichert die UV-Koordinaten die beim Abwickeln generiert wurden und das ***Octane*-MaterialTag** definiert den Shader des Objekts.

Damit sie sichtbares Licht enthält benötigt die Szene eine Lichtquelle. In diesem Fall wurde ein **Sky** benutzt, der mit einer 360°- HDR-Textur versehen ist. Anhand dieser Textur wird der Lichteinfall aus der Umgebung berechnet.

Schließlich wird noch eine **Kamera** benötigt, durch welche das finale Bild gerendert wird. Diese benötigt für weitere Einstellungen ein ***Octane*-CameraTag**. Sie muss als Renderkamera ausgewählt sein.

- ***Einstellungen ObjectTag***: Abbildung 4.11 zeigt die benötigten Einstellung des ObjectTags. Jedes zu backende Objekt muss ein solches Tag erhalten. In diesem wird unter dem Reiter *Object layer* die *Bake ID* festgelegt. Diese beginnt bei „2“ und wird für jede zu rendernde Textur inkrementiert. Das bedeutet, dass es auch möglich ist, die Texturen mehrerer Einzel-

objekte in ein gemeinsames Bild zu backen, sofern sie die selbe *Bake ID* besitzen und sich ihre UV-Koordinaten nicht überschneiden.

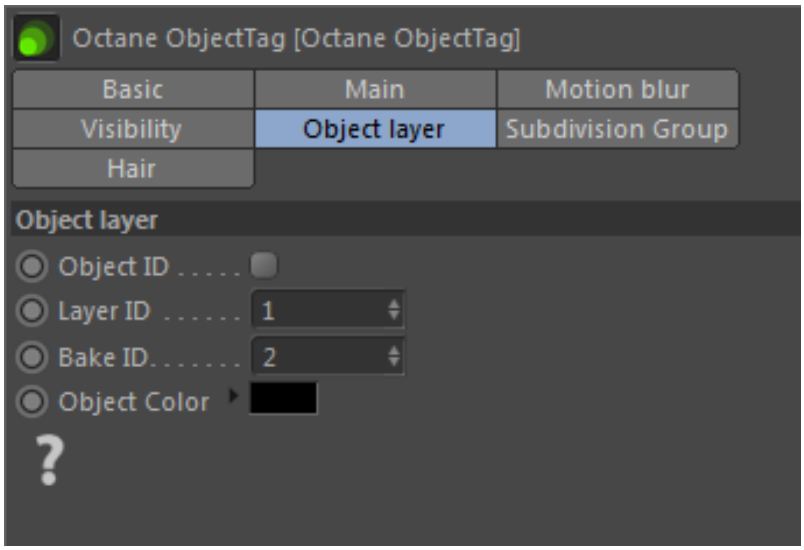


Abbildung 4.11: Das *Octane*-ObjectTag

- ***Einstellungen CameraTag***: Die Konfiguration des CameraTags der Kamera zeigt Abbildung 4.12. Zunächst wird der *Camera type* auf *Baking* gestellt. Anschließend kann im *Baking*-Reiter die *Baking group ID* definiert werden. Diese ist das Gegenstück zum Wert *Bake ID* im ObjectTag. Steht sie also wie in diesem Fall auf „2“, werden alle Objekte mit der ID „2“ in ihrem ObjectTag durch diese Kamera gerendert. Zusätzlich können hier weitere Feineinstellungen des Backvorgangs vorgenommen werden, die Standardwerte erwiesen sich allerdings als gut nutzbar.

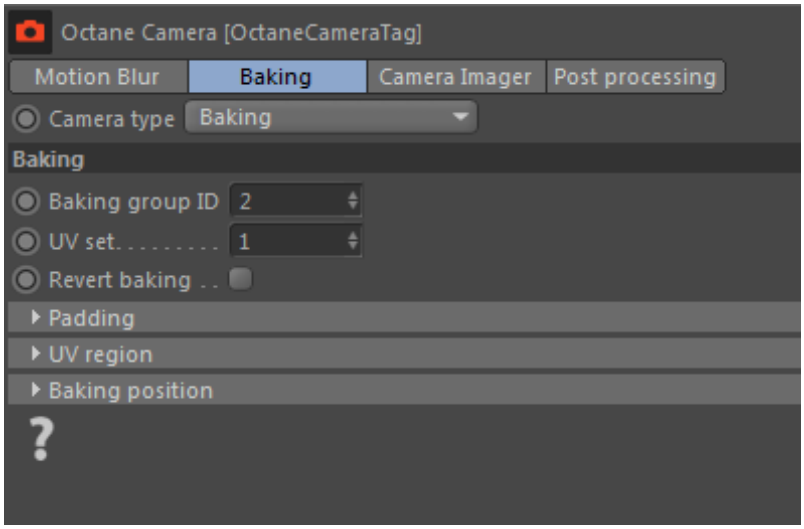


Abbildung 4.12: Das *Octane*-CameraTag

Die Auflösung der Zieltextur kann in den allgemeinen *Cinema 4D*-Rendereinstellungen festgelegt werden. Hier wurde meist 4096x4096 Pixel gewählt, da dieser Wert ein Vielfaches von zwei ist, was der Speicherverwaltung und Performanz der Engine zuträglich ist und die Auflösung noch viele Details beinhaltet. Die Größe bietet immer noch die Möglichkeit, die Texturen im Nachhinein nach Belieben kleiner zu skalieren, was bei der finalen Version des Prototyps häufig getan wurde (auf von 128x128 bis 2048x2048 Pixel).

Mit diesem Vorgang lassen sich ebenfalls Normalmaps und weitere Kanäle, wie Ambient Occlusion oder reine Farbinformationen backen.

Da der Backvorgang für viele einzelne Objekte umständlich und zeitaufwendig ist – jedes *ObjectTag* braucht seine eigene manuell zugewiesene Back-ID, jede ID muss einzeln manuell im *CameraTag* der Renderkamera eingestellt werden mit anschließendem Anpassen des Ausgabepfads und Rendern – wurde eigens hierfür ein *Python*-Skript geschrieben (im Anhang enthalten). *Cinema 4D* bietet eine Schnittstelle für *Python*, sodass Aufgaben auf diese Weise automatisiert werden können. Das Skript durchläuft alle *ObjectTags* und verteilt aufsteigende *Bake IDs*, erzeugt für jedes Tag eine Renderkamera mit Namen des entsprechenden Objekts und setzt die Frameanzahl des *Cinema 4D*-Dokuments auf die Anzahl der unterschiedlichen IDs. Die erstellten Kameras wechseln nun abhängig von der Framenummer durch (im zweiten Frame ist die Kamera mit der *Baking group ID* „2“ aktiv, es werden Objekte mit der *Bake ID* „2“ gebacken). Durch diese Herangehensweise ist es möglich, die gesamte Bildsequenz des Dokuments in einem Schritt vollautomatisch zu rendern. Für jeden Frame der zu rendernden Sequenz wird eine eigene Textur mit dem Namen der gerade aktiven Kamera (entspricht dem Objektnamen) gespeichert.

Sind die Texturen schließlich gerendert, müssen sie für den Export in ein *Cinema 4d*-Material geladen werden. Die gebackenen Objekte erhalten ein TexturTag mit entsprechendem Material. Die Projektion des Materials muss im TexturTag auf *UVW-Mapping* gestellt werden. Nicht benötigte oder gewünschte Objekte und Tags werden gelöscht und die Szene ist bereit für den Export.

4.2.5 Export in AssetBundles

In diesem Abschnitt wird der Weg eines Modells aus der 3D-Software bis in die fertige App beschrieben. Die Szene wurde in *Cinema 4D* bereits soweit vorbereitet, dass sie zum Export bereit ist. Das bedeutet, dass die gewünschten erlaubten Transformationen korrekt über die Objekthierarchie definiert sind, die UV-Maps der Meshes abgewickelt sind und die Objekte mit den entsprechenden *Cinema 4D*-Materialien versehen sind. Das Modell wird nun direkt aus der 3D-Software als FBX-Datei exportiert. Dieses Dateiformat von *Autodesk* eignet sich besonders für den Austausch von Objekten für die Spieleentwicklung [3]. Die Objekthierarchie bleibt erhalten und es besteht die Möglichkeit, Materialien mit eingebetteten Texturen zu übertragen.

Da es in der freien Version von *Unity* nicht möglich ist, ohne kostenpflichtige Plugins FBX-Dateien zur Laufzeit in gepackte Anwendungen einzulesen, ist der Weg des Direktimports der Datei in die AR-App versperrt. Die *Unity*-Entwicklungsumgebung selbst beherrscht aber den Import dieses Dateiformats. Sie legt direkt entsprechende *Gameobjects* (die engineeigene Repräsentation eines Objekts) mit den passenden Meshs und *Unity*-Materialien an. Von hier aus können *Gameobjects* anschließend komplett, inklusive ihrer Materialien und Skripte, in sogenannte *AssetBundles* gespeichert werden [56]. Diese Objektarchive sind Dateien, die mit Enginebordmitteln auch in fertigen Anwendungen eingelesen und wieder entpackt werden können. Ein weiterer großer Vorteil dieses Ansatzes ist die Tatsache, dass die 3D-Modelle vor dem Export in ein *AssetBundle* weiter angepasst werden können. So lassen sich zum Beispiel die Größe abstimmen, Shader austauschen, Lichter einfügen und auch eigene Skripte anhängen. Außerdem kann so der finale Look der Objekte in der Engine selbst überprüft werden.

Für diesen Import-Export-Vorgang (FBX nach *AssetBundle*) wurde ein eigenes *Unity*-Projekt angelegt, welches die benötigten C#-Skripte enthält. Der Export von *AssetBundles* ist zunächst nur über Code möglich, es sei denn das Projekt enthält ein Skript mit folgenden Codezeilen:

```
1 #if UNITY_EDITOR
2 using UnityEditor;
3
4 public class CreateAssetBundlesAndroid
5 {
6   [MenuItem ("Assets/Build AssetBundles for Android")]
7   static void BuildAllAssetBundles ()
```

```
8   {
9     //Erster Parameter definiert Ausgabepfad, der dritte die Zielplattform
10    BuildPipeline.BuildAssetBundles ("Assets/AssetBundles",
      BuildAssetBundleOptions.None, BuildTarget.Android);
11  }
12 }
13 #endif
```

Wird dieses kompiliert erscheint fortan in der Programmoberfläche bei einem Rechtsklick auf den *Contentmanager* ein Eintrag zum Speichern der Bundles. Das Projekt enthält außerdem zwei weitere selbst erstellte Skripte, mit denen Objekte versehen werden können:

- ***Das Thumbnail-Skript***: Wird auf das oberste Objekt des Modells in der Hierarchie angewandt. Es enthält eine frei zugreifbare Variable in der eine Textur hinterlegt werden kann. Dieses Bild wird später beim Laden der Dateien im Modellbrowser der fertigen App als Vorschaubild angezeigt.

- ***Das MaterialChanger-Skript***: Ermöglicht den Materialwechsel eines Objekts im Prototyp bei einem Tap darauf. Enthält ein Array, welches mit einer beliebigen Anzahl an *Unity*-Materialien gefüllt werden kann. Zwischen diesen Materialien wird, in im Array definierter Reihenfolge, zyklisch durchgeschaltet.

Um ein *Gameobject* einem *AssetBundle* hinzuzufügen, muss es als *Prefab* gespeichert werden. *Prefabs* sind in *Unity* so etwas wie Vorlagen von Objekten, von denen Instanzen erzeugt werden können. Zum Erstellen eines *Prefabs* muss das *Gameobject* lediglich per Drag and Drop aus dem *Hierarchie Fenster* in den *Contentmanager* gezogen werden. In dem *Prefab* lässt sich nun festlegen, zu welchem *AssetBundle* es gehört. Über die bereits per Skript erstellte Schaltfläche können nun alle Bundles exportiert und anschließend auf das Endgerät des *Presenters* kopiert werden.

4.3 Beleuchtung

Die Herangehensweise für die Verwendung der dynamischen und der vorgebackenen Beleuchtung in der Anwendung unterscheidet sich leicht. In der Hauptszene, in der Modelle angezeigt werden ist eine Grundbeleuchtung (*Ambient Light*) bereits vorhanden. Sie kann in den allgemeinen *Lighting*-Einstellungen der Szene (Hauptmenüeintrag Window > Lighting > Reiter Scene) festgelegt werden. Ohne sie wären Modelle ohne eigene enthaltene Lichtquelle nicht sichtbar. Die Vorbereitungen für Modelle mit dynamischer, gebackener oder gemischter Beleuchtung ist wie folgt:

- ***Dynamische Beleuchtung***: Die Objekte in dieser Szene erhalten nur Materialien mit Texturen, die lediglich die grundlegenden Farbwerte enthalten, nicht aber den Einfluss der Beleuchtung, wie etwa überhellte Stellen oder Schatten. Die Schattierung wird später direkt zur Laufzeit berechnet und angezeigt. Da die Grundszene der Anwendung nur das *Ambient Light* ohne Richtung und Schattenwurf enthält, muss die gewünschte Lichtquelle in dem Modell selbst enthalten sein. Dazu werden die Lichtobjekte der *Unity*-Engine benutzt. Sie werden vor dem Speichern des gesamten *Gameobjects* als *Prefab* und anschließendem Export in *AssetBundles* in der Modellszene platziert und konfiguriert. Meist wurden gerichtete Lichter verwendet (*Directional Lights)* in deren Einstellungen der Schattenwurf auf *Soft Shadows* gestellt wurde. *Soft Shadows* bieten eine leicht höhere Qualität als die *Hard Shadows*, sind aber etwas teurer in der Berechnung. Schließlich wird das gerichtete Licht so rotiert, dass es aus der präferierten Richtung kommt.

- ***Gebackene Beleuchtung***: Hier erhalten die verwendeten Materialien die Texturen, die im Vorhinein mit dem *Octane*-Renderer in *Cinema 4D* gebacken wurden. Da hier die hochqualitative Beleuchtung direkt in den Texturen fest gespeichert ist, wird im Gegensatz zu dem dynamischen Ansatz kein zusätzliches Licht zwingend benötigt. Es kann aber hilfreich

sein, ein *Ambient Light* ohne Schattenwurf in das Modell zu integrieren. So lässt sich die komplette Szene noch feinjustieren, etwa aufhellen oder einfärben, falls die gerenderten Texturen in der Engine nicht optimal erscheinen.

- ***Kombination***: Beide Herangehensweisen lassen sich auch beliebig kombinieren. So kann es sinnvoll sein, für statische Objekte eine vorgebackene Beleuchtung zu verwenden, während bewegliche Objekte dynamisch beleuchtet werden. Dafür erhalten die *Gameobjects* die Materialien mit den entsprechenden Texturen. Die Szene wird wie beim dynamischen Ansatz durch ein *Unity*-Licht beleuchtet, welches so konfiguriert wird, dass sich seine Schatten mit den vorberechneten Schatten decken. Anschließend wird die Dichte der Schatten des Lichts an die gebackenen angepasst. Für jedes Objekt kann nun festgelegt werden, ob es von dem dynamischen Licht einen Schatten wirft (bewegliche Objekte) oder nicht (statische Objekte).

4.4 Backend

Dieses Kapitel beschreibt die Entwicklung des Backends: welche Bibliotheken und Skripte im Hintergrund benutzt werden und welche wichtigen Vorgänge dort ablaufen. Zunächst wird das Einbinden und Einrichten der *Vuforia*-Trackinglibrary beschrieben, gefolgt von der Verwendung der *TouchScript*-Library für die Touchbedienung. Anschließend folgen die Details des dynamischen Ladens der Modelle zur Laufzeit und der Netzwerkkommunikation.

4.4.1 Vuforia

Für die Augmented Reality-Funktionen wurde in diesem Prototyp die *Vuforia*-Trackinglibrary verwendet. Nach Anlegen eines kostenlosen Developeraccounts lässt sie sich in verschiedenen Versionen von der *Vuforia* Website herunterladen [62]. Es besteht die Auswahl zwischen dem SDK für Android (Java/C++), iOS (C++) und *Unity*. Nach dem Download des *Unity*-SDKs wird dieses entweder durch Ausführen der heruntergeladenen Datei oder durch manuellen Import in ein vorhandenes *Unity*-Projekt integriert. Im SDK enthalten sind die benötigten Skripte für die Trackingfunktionen, Beispiele, sowie Elemente für die Anzeige, wie Schriftarten, Materialien, Shader und Texturen. Hilfreich sind außerdem die vorgefertigten *Prefabs*, wie die zwei Grundbestandteile: die ***ARCamera*** und das ***ImageTarget***. Sie können einfach in die Szene eingefügt werden und besitzen schon alle notwendigen Skripte und Einstellungen um schnell erste Ergebnisse zu erzielen.

- ***ARCamera***:

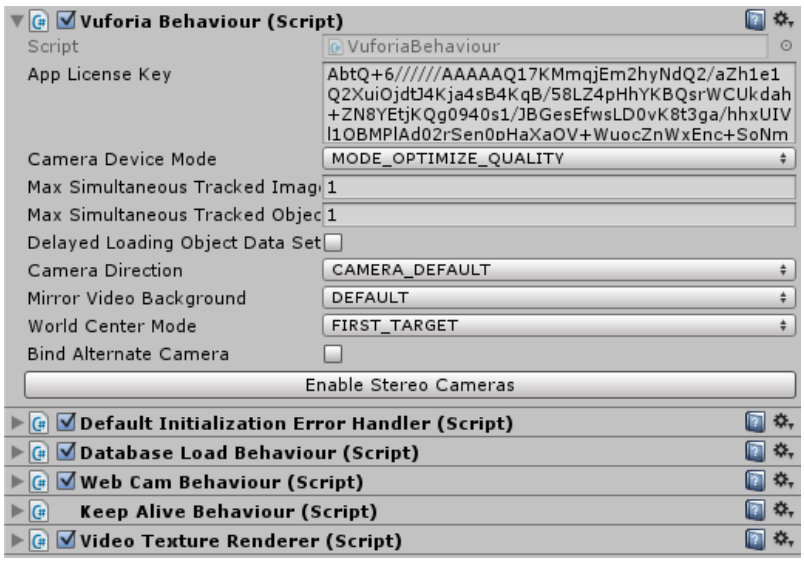


Abbildung 4.13: Die Einstellungen des ARCamera-Objekts

Das ARCamera-Objekt selbst ist keine tatsächliche *Unity*-Kamera sondern besitzt so eine Kamera als Unterobjekt, durch welche die Ansicht gerendert wird. Die Synchronisation der Position der ARCamera in der realen und der digitalen Welt wird durch Skripte von *Vuforia* anhand des optischen Trackers gewährleistet. Durch die Unterordnung der *Unity*-Kamera werden die errechneten Koordinaten immer auch auf diese übertragen. Abbildung 4.13 zeigt die zugeordneten Skripte und die Einstellungen des *Vuforia Behaviour Scripts*. Die Standardeinstellungen erwiesen sich als praktikabel, sodass höchstens kleine Anpassungen vonnöten waren. Für das Benutzen des *Vuforia*-SDKs muss über die Entwicklerwebsite ein Lizenzschlüssel beantragt werden, welcher in das Feld *App License Key* eingetragen werden muss. Er dient zur Identifikation der Anwendung bei der Zuordnung von Clouddiensten und dem Bezahlmodell.

- ***ImageTarget***:

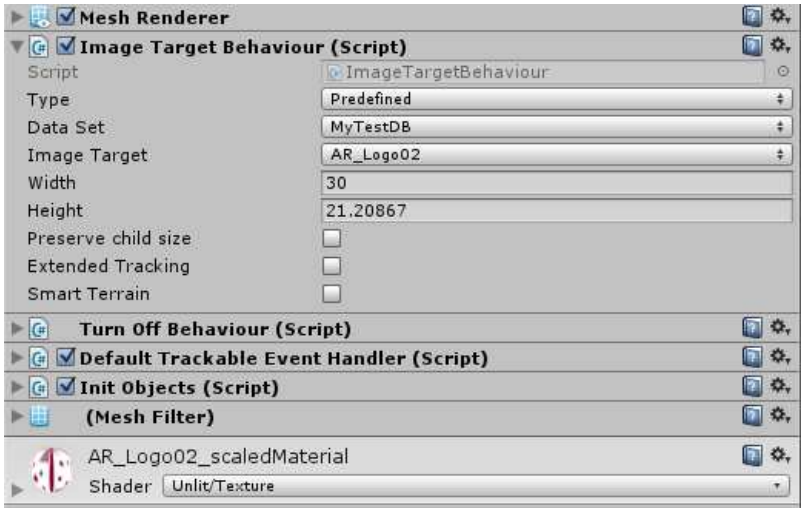


Abbildung 4.14: Die Einstellungen des ImageTarget-Objekts

Das ImageTarget ist der zweite Grundbaustein für die Trackingfunktion. Abbildung 4.14 zeigt die zugewiesenen Skripte und Einstellungen des *Image Target Behaviour Scripts*. Es handelt sich dabei um ein *Gameobject*, das den realen ausgedruckten 2D-Tracker in der digitalen Version der Welt repräsentiert. Es besitzt das Mesh einer ebenen Fläche und das Material mit dem ausgewählten Trackingbild als Textur. Die Geometrie ist jedoch eher für die Hilfe bei der Entwicklung im *Unity*-Editor gedacht, zur Laufzeit wird dieses Mesh standardmäßig ausgeblendet. Um dreidimensionale Modelle in der Anwendung auf dem Tracker an der korrekten Position in der realen Welt anzuzeigen, müssen sie lediglich in der Hierarchie dem ImageTarget untergeordnet sein. Sobald *Vuforia* den entsprechenden Tracker erkennt werden die Unterobjekte eingeblendet.

Um dem ImageTarget ein Bild als Tracker zuzuweisen ist der Umweg über die *Vuforia*-Entwicklerwebsite notwendig. Die gewünschten *Targets* werden hier in eine Datenbank hochgeladen und serverseitig auf ihre Eignung als Tracker analysiert. Besonders gut eignen sich Bilder mit hohen Kontrasten und harten Kanten. Bei dem Vorgang werden in der Textur außerdem Trackingpunkte gesetzt, an denen sich *Vuforia* später orientieren kann. Nach der Verarbeitung kann die Datenbank als Ganzes heruntergeladen und in das *Unity*-Projekt importiert werden. Erst jetzt kann in dem *Image Target Behaviour Script* des ImageTrackers dem Parameter *Data Set* die Datenbank zugewiesen werden und als *Image Target* wird eines der *Targets*, also der enthaltenen Bilder, definiert. Zusätzlich können Breite und Höhe des Trackers in Welteinheiten angepasst werden.

Nun ist das *Vuforia*-SDK vollständig in das Projekt eingebunden und kann verwendet werden.

4.4.2 Touchbedienung mit TouchScript

Für eine positive Benutzererfahrung und für die Möglichkeiten, die der Prototyp bieten sollte, war es unerlässlich, eine gut funktionierende Touchbedienung zu implementieren. Der erste Ansatz war die vollständige manuelle Programmierung aller Grundlagen über die von *Unity* zurückgegebenen Bildschirmkoordinaten eines Klicks beziehungsweise Touches. Dies erwies sich für die benötigten komplexen Funktionen als wenig zielführend und zeitlich ineffektiv. Es war sinnvoller, dafür auf eine Library zurückzugreifen, die die Grundfunktionen bereits enthält und auf der weiter aufgebaut werden konnte.

Die Wahl fiel auf *TouchScript* [52]. Dieses freie *Unity*-Plugin bietet Funktionen und vordefinierte Gesten der Single- und Multitouchbedienung und lässt sich über *GitHub* oder direkt über den *Unity Asset Store* herunterladen. Im Paket enthalten sind die Skripte, Beispiele, Texturen und Shader sowie *Prefabs*. Um die Library zu verwenden muss ein Objekt der Szene das *TouchManager Script* erhalten. Dieses verwaltet die grundsätzlichen Funktionen. Die manipulierbaren Objekte selbst werden mit einem *MeshCollider Script* für das Erkennen der Kollision der Touches und mit einem *TransformGesture Script* versehen. In diesem kann die Art der Transformation auf *Translation*, *Rotation* oder *Scaling* oder einer Kombination daraus festgelegt werden. Für die Rotation hat es sich als sinnvoll erwiesen stattdessen das *PinnedTransformGesture Script* zu nutzen, da so ein Punkt fixiert ist und die Drehung mit nur einem Finger ausgeführt werden kann. Für jede Grundtransformation wird eine eigene Geste definiert, da so die Projektionsebenen der Touchbewegungen individuell anpassbar sind. Damit die erkannte Bewegung auch auf das Objekt angewandt wird, braucht es zu guter Letzt das *Transformer Script*. Abbildung 4.15 zeigt die Skripte eines korrekt eingerichteten manipulierbaren Objekts.

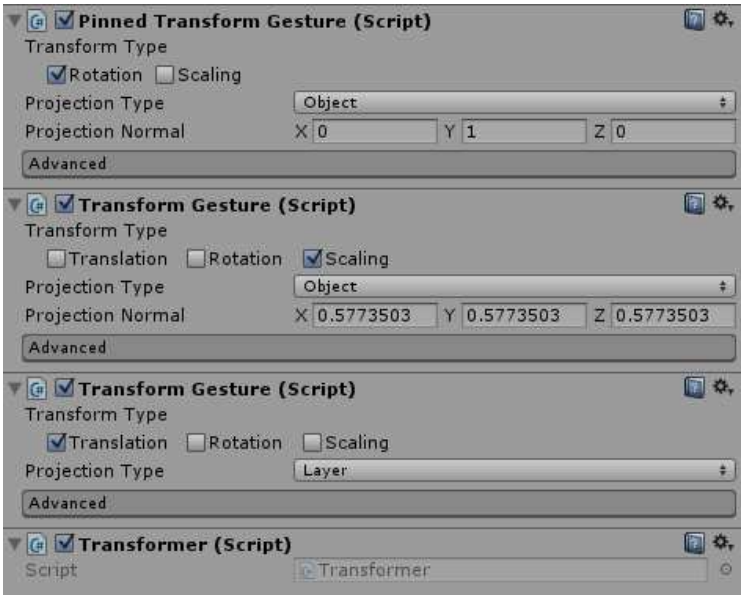


Abbildung 4.15: Die *TouchScript*-Skripte eines Objekts, das alle drei Grundmanipulationen erlaubt.

Mit *TouchScript* ist es möglich, verschiedene Gesten auf einem Objekt zu kombinieren. In etwa um dieses gleichzeitig verschieben (*TransformGesture*) und drehen (*PinnedTransformGesture*) zu können. Damit sie sich nicht gegenseitig behindern oder ausschließen müssen die verwendeten Gestenskripte beim jeweils anderen als *Friendly Gesture* eingetragen werden. In Abbildung 4.15 befindet sich diese Konfiguration hinter den eingeklappten *Advanced*-Tabs und ist deshalb dort nicht sichtbar.

Die Details der Implementierung, etwa wie Modelle beim Laden automatisch mit den benötigten Skripten und Einstellungen versehen werden, folgen im nächsten Abschnitt.

4.4.3 Dynamisches Laden

Ein wichtiges und komfortables Feature der fertigen Anwendung ist die Austauschbarkeit der Modelle. Dafür ist es unerlässlich, dass diese während des Ladevorgangs aus den *AssetBundles* analysiert und die nötigen Skripte mit den korrekten Einstellungen zur Laufzeit eingefügt werden.

In diesem Teil wird der genaue Ablauf des Ladens beschrieben. Die Netzwerkfunktion, wie etwa die Modellverteilung an die Clients, ist Teil des nächsten Abschnitts. Alle Instanzen der Anwendung, egal ob sie *Presenter* oder *Spectator* sind, bekommen die Anweisung den Ladevorgang zu starten von dem Server. Der *Presenter* ist zugleich Server und auch Client. Für die In-/Outputfunktionen, wie das Auslesen von Dateien, ist die selbstgeschriebene statische ***FileManager*-Klasse** zuständig. Sie bekommt als erste den Aufruf zum Laden eines Modells. Ihr werden der Name des gewünschten *AssetBundles*, der Name des Modells selbst und die Dateigröße des Bundles übergeben. Der Algorithmus der entsprechenden Methode in Pseudocode lautet wie folgt:

```
1  static void checkAndLoadModel(assetbundlename, modelname, filesize){
2    if (!fileFound(assetbundlename) || !sameFileSize(assetbundlename, filesize))
3    {
4      //AssetBundle nicht gefunden oder falsche Größe
5      //Verlange Bundle vom Server
6      requestAsset(assetbundlename, modelname);
7    } else {
8      //AssetBundle gefunden, lade Modell
9      loadModelFromAssets(assetbundlename, modelname);
10   }
11 }
```

Die Methode *loadModelFromAssets* lädt daraufhin das *Gameobject* aus dem *AssetBundle* und übergibt es weiter an das ***InitObjects*-Skript**. Dieses Skript regelt das Einfügen aller Objekte in die Szene, auch das der nicht manipulierbaren, einschließlich der leicht unterschiedlichen Behandlung auf Host und Clientseite. Die wichtige Methode ist hier *loadModel*. Sie liest das erste Mal die Hierarchie und die Objektnamen des Modells aus. Der stark reduzierte Pseudocode lautet:

```
1  void loadModel(gameobject){
2    interactionFound = false;
3    foreach (child in gameobject.children){
4      if (child.name.contains('_delete_'))
5        child.destroy();
6      if (child.name.contains('_noShadow_'))
7        child.disableShadows();
8      if (child.hasMaterialChangerScript())
9        interactionFound = true;
10     if (child.hasChild & child.child.name.contains('Translation'|'Rotation'|'
       Scale'))
11       interactionFound = true;
12     if (interactionFound == true){
13       child.AddMeshCollider();
14       //Skript, das für Einrichtung manipulierbarer Objekte zuständig ist:
15       child.AddInitManipulatableObjectScript();
16       if (Player == Host){
17         Server.SpawnObjectOnClients(gameobject);
18       } else if (Player == Client){
19         Client.RegisterObjectAsSpawnable(gameobject);
20         //Zerstöre Instanz und warte auf Spawnbefehl vom Server
21         gameobject.destroy();
22       }
23     }
24   }
25 }
```

Die Einrichtung der Objekte, die von dem *InitObjects*-Skript als manipulierbar erkannt wurden, wird an das ***InitManipulatableObjects*-Skript** übergeben. Jedes dieser Objekte besitzt nun dieses Skript. Es versieht die *Gameobjects* mit den passenden *TouchScript*-Skripten und stellt diese, wie im vorigen Abschnitt beschrieben, korrekt ein. Dadurch wird es erst möglich, einzelne Objekte zu bewegen:

```
1  void initManipulation(){
2    foreach (child in gameobject.children){
3      if (child.name.contains('Translation')){
4        //Lege erlaubten Bewegungsbereich anhand der Position fest
5        startPosition = gameobject.position;
6        goalPosition = child.position;
7        addAndSetupTranslationTransformGesture();
8      }
9      if (child.name.contains('Rotation')){
10       //Lege erlaubte Rotationsachsen anhand des Namens (XYZ) fest
11       setRotationPlane(child.name);
12       addAndSetupRotationPinnedTransformGesture();
13     }
14     if (child.name.contains('Scale')){
15       //Lege erlaubte Skalierungsachsen anhand des Namens (XYZ) fest
16       setScalePlane(child.name);
17       addAndSetupScaleTransformGesture();
18     }
19   }
20   //Definiere für die Kombination alle Gesten als friendly zueinander
21   addAllGesturesAsFriendly();
22 }
```

Nun ist das Modell eingelesen, die Objekthierarchie wurde durchlaufen, analysiert und die in ihr definierten erlaubten Interaktionen wurden umgesetzt. Der Nutzer ist jetzt in der Lage, das dynamisch geladene Modell zu manipulieren.

4.4.4 Netzwerkkommunikation

Die Netzwerkkommunikation war ein weiterer großer und komplexer Schwerpunkt in der Entwicklung. Die Umsetzung war sehr zeitaufwendig und forderte das Beachten vieler verschiedener Aspekte. In diesem Abschnitt werden die grundlegenden Abläufe der Multi-User-Funktionen erläutert. *Unity* bietet mit *UNet* eine Palette vorgefertigter Skripte für Multiplayeranwendungen. Im Laufe der Entwicklung hat sich jedoch gezeigt, dass diese teilweise äußerst unperformant und fehlerbehaftet sind oder für die speziellen Anforderungen dieser Anwendung nicht nutzbar waren (siehe dazu auch *4.6.2 Unity UNET* im Kapitel *4.6 Schwierigkeiten bei der Umsetzung*). Aufgrund dessen wurden einige der Skripte erweitert und andere komplett neu geschrieben.

Die grundlegende Netzwerkarchitektur ist in Abbildung 4.16 dargestellt. Es gibt einen Host, der *Presenter*, welcher zugleich Serverfunktionalitäten im Hintergrund bietet aber doch auch ein Client ist. Zu ihm können sich beliebig viele Clients (*Spectators*) verbinden.

Die Hauptfunktionen des Verbindungsaufbaus und der -verwaltung übernimmt das ***NetworkManager_Custom*-Skript** welches von der *UNet*-Klasse *NetworkManager* erbt. Diese enthält einige Methoden, die für das Öffnen und Schließen von Sessions und Verbindungen zuständig sind, außerdem lassen sich hier zahlreiche Netzwerkeinstellungen festlegen (so zum Beispiel Adresse und Port, Kanäle und Timeouts). Die erbende Klasse überschreibt viele Methoden, die bei bestimmten Netzwerkevents ausgelöst werden, wie etwa *OnServerConnect* beim Verbinden eines Clients zum Server oder *OnLevelWasLoaded* beim Wechseln der aktuellen Szene. Der Ablauf des Verbindungsaufbaus ist wie folgt, die Methoden befinden sich hier alle in *NetworkManager_Custom*:

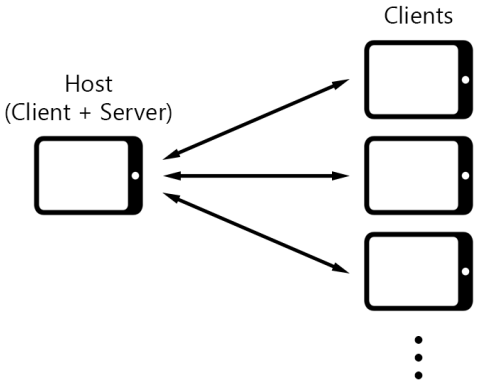


Abbildung 4.16: Die Netzwerkarchitektur.

- ***Auf Hostseite***: Nachdem im Hauptmenü die Wahl auf das Erzeugen einer Session gefallen ist, wird *CreateSession()* ausgeführt. Der Verbindungsport wird aus dem Interface gelesen und gespeichert, anschließend wird *StartHost()* im *NetworkManager* aufgerufen. Diese Methode startet das Horchen nach Verbindungen und wechselt aus dem Hauptmenü in die Hauptansicht der Anwendung. Zusätzlich wurde manuell die Methode *OnServerChatMessage* registriert, welche die Behandlung von eingehenden Nachrichten übernimmt. Beim Szenenwechsel wird der *OnLevelWasLoaded*-Event ausgelöst durch welchen Änderungen an der Oberfläche, wie das Aktualisieren von Texten, abhängig von der Rolle des Nutzers (*Presenter* oder *Spectator*) durchgeführt werden.

  Verbindet sich ein nun ein Client zum Server wird die Methode *OnServerConnect* aufgerufen, welche erkennt ob es sich um einen externen Client handelt oder um den *Presenter* selbst (der Host ist zugleich einerseits Server wie auch ein sich verbindender Client). Anschließend wird *OnServerAddPlayer* ausgelöst. Hier wird für jeden Client ein Spielerobjekt erzeugt mit dem Modell einer kleinen Kamera, die stets der Position des Spielerendgeräts folgt. Dieses Objekt stellt auch die Verbindung zu dem speziellen Nutzer dar, Nachrichten können darüber an genau diesen Client versendet werden, verlässt er die Session wird es automatisch von *UNet* gelöscht. Zuletzt wird nun die Anzahl der verbundenen Clients inkrementiert und im Interface des *Presenters* geupdated.

- ***Auf Clientseite***: Möchte sich ein Nutzer zu einer vorhandenen Session verbinden wird die Methode *JoinSession* ausgeführt. IP-Adresse und Port werden aus den entsprechenden Textfeldern gelesen und gesichert. Im *NetworkManager* wird *StartClient()* aufgerufen, auf diese Weise wird versucht, eine Verbindung zum Server aufzubauen. Währenddessen wird dem User ein Popup angezeigt, das ihn über diesen Vorgang informiert. Bei erfolgreichem Verbindungsaufbau wird auch hier ein Event für den Nachrichtenerhalt registriert (*OnClientChatMessage*), die Methode *OnClientConnect* wird ausgeführt, welche bei dem Server nachfragt, ob bereits ein Modell geladen ist und wenn ja, um welches es sich dabei handelt. Wie auf der Serverseite wird auch hier der *OnLevelWasLoaded*-Event ausgelöst, der das Interface der Rolle anpasst.

Nach diesen Schritten ist die Verbindung erfolgreich aufgebaut, die Kommunikation zwischen *Presenter* und *Spectator* regelt die weiteren Vorgänge. Für diese hat es sich als sinnvoll erwiesen, eine Mischung aus *Remote Procedure Calls* (*RPCs*) und klassischen Nachrichten zu verwenden. *RPCs* sind Methoden, die über das Netzwerk hinweg bei den anderen Endpunkten aufgerufen werden können. Sie machen die Netzwerkkommunikation einfach und übersichtlich, haben in *UNet* allerdings den sehr großen Nachteil, dass es keine Unterscheidung zwischen einzelnen Clients

gibt. Es ist nur möglich, einen *RPC* bei dem Server oder bei ALLEN Clients aufzurufen. Für Meldungen, die nur an bestimmte Nutzer gehen sollen, wurde deshalb das *UNet*-eigene Messagesystem verwendet. Hierbei gibt es entweder vordefinierte Nachrichtentypen oder man deklariert seine eigenen mit den gewünschten Nachrichtenfeldern. In diesem Fall wurde eine eigene *StatusMessage*-Klasse definiert, die die folgenden Felder enthält:

```
1 public class StatusMessage : MessageBase
2   {
3     public string message; //Typ oder Inhalt
4     public string data; //Beliebige Daten (z.B. Modellname)
5     public string filename; //Dateiname des AssetBundles
6     public long filesize; //Dateigröße des AssetBundles
7   }
```

Eine Methode zum Senden einer Nachricht zum Mitteilen des aktuell geladenen Modells von Server an Client sieht wie folgt aus:

```
1 public static void sendWhatIsLoadedServer(NetworkConnection connection){
2   //Erzeugen des Nachrichtenobjekts
3   MyMessages.StatusMessage msg = new MyMessages.StatusMessage ();
4
5   //Füllen der Nachricht mit Inhalt
6   msg.message = "WhatIsLoaded";
7   msg.filename = assetbundlename;
8   msg.data = modelname;
9   msg.filesize = filesizeLong;
10
11  //Senden über gewünschten Kanal
12  connection.SendByChannel ((short)MyMessages.MyMessageTypes.STATUS_MESSAGE,
      msg, channel);
13 }
```

Das Laden eines Modells auf Seite des *Presenters* wird per *RPC* an alle Clients übermittelt. Nachdem ein Modell im Ladeinterface ausgewählt wurde, wird im Spielerobjekt, das die Verbindung zum Server darstellt, die Methode *LoadModelToServer* zusammen mit den Parametern *AssetBundleName*, *Modelname* und *AssetBundleFileSize* aufgerufen. Diese führt per *RPC* auf dem Server eine Methode aus, welche wiederum bei allen verbundenen Clients einen *RPC* mit den selben Parametern aufruft, der ihnen den Befehl zum Laden dieses bestimmten Modells gibt. Der Ablauf ist also: *Presenter* schickt *RPC* an Server, Server schickt *RPC* an alle Clients, Clients laden das korrekte Modell.

Der Ladeaufruf erreicht die in *4.4.3 Dynamisches Laden* vorgestellte *FileManager*-Klasse, die für I/O-Funktionen zuständig ist. Sie überprüft, ob das benutzte *AssetBundle* lokal auf dem Endgerät des Nutzers vorliegt, indem es Name und Dateigröße vergleicht. Falls dem so ist, beginnt der bereits beschriebene dynamische Ladevorgang. Falls nicht, fordert sie die Datei beim Server an. Dies geschieht nicht über *RPCs*, sondern *StatusMessages*, da nur der eine Client das Modell benötigt. Die Anfrage an den Server geschieht wie folgt:

```
1 MyMessages.StatusMessage msg = new MyMessages.StatusMessage();
2
3 msg.message = "gimme";
4 msg.data = filename;
5
6 NetworkManager.singleton.client.Send((short) MyMessages.MyMessageTypes.
    STATUS_MESSAGE, msg);
7
8 //Erzeugen und Starten eines eigenen Threads
9 Thread thread = new Thread(() => sendBundleTCPClient(fullpath, NetworkManager.
    singleton.client.connection, port));
10 thread.Start();
```

Wie in dem Code zu sehen wird nach Senden der Anfrage ein eigener Thread gestartet, der nicht blockierend auf den Eingang der Datei vom Server wartet. Das Übertragen der Datei geschieht über eine klassische TCP-Filestream-Verbindung. Die *UNet* eigenen Nachrichten erwiesen sich hierfür als keineswegs geeignet, die Datenübermittlung war zwar möglich aber umständlicher und um ein Vielfaches langsamer.

Während der Dateiübertragung sieht der Nutzer ein Informationsfenster, unter anderem mit einem Fortschrittsbalken und der Menge an bereits übertragenden Daten (zu sehen in *4.5.2 Präsentationsansicht*). In dem Thread auf Seiten des Clients geschieht vereinfacht folgendes:

```
1  //Starten des Listeners, der auf die Verbindung horcht
2  var listener = new TcpListener(IPAddress.Any, port);
3  Debug.Log ("Listening...");
4
5  //Dieser Aufruf ist blockierend, Fortfahren erst nach Verbindungsaufbau
6  listener.Start();
7  Debug.Log ("Found connection...");
8
9  //Eingehenden Datenstream abgreifen
10 var incoming = listener.AcceptTcpClient();
11 var networkStream = incoming.GetStream();
12
13 //Netzwerkstream in Filestream (=Datei) kopieren
14 FileStream fileStream;
15 using (fileStream = File.OpenWrite(fullpath)) {
16   CopyStream (networkStream, fileStream);
17 }
18
19 //Verbindung schließen
20 listener.Stop();
21 Debug.Log ("Closed connection...");
```

Natürlich besitzt der Server das passende Gegenstück zum Senden der Datei. Der Sendevorgang wird gestartet, nachdem die *AssetBundle*-Anfrage von dem Client empfangen wurde und läuft analog zum Empfang ab:

```
1  var client = new TcpClient();
2
3  client.Connect(IPAddress.Parse(ip), port);
4  Debug.Log ("Connection established...");
5
6  var networkStream = client.GetStream ();
7
8  FileStream fileStream;
9  using (fileStream = File.OpenRead(fullpath)){
10   CopyStream (fileStream, networkStream);
11 }
12
13 client.Close();
14 Debug.Log ("Closed connection...");
```

Nach Beenden der Dateiübertragung versucht der Client erneut, das Modell zu laden, diesmal ist es vorhanden und der dynamische Ladevorgang beginnt. Damit ist die Kommunikation allerdings nicht beendet. Ab jetzt muss der Stand der Szene bei allen *Specators* mit dem auf der Seite des *Presenters* synchron gehalten werden, damit seine Objektmanipulationen für alle Teilnehmer der Session sichtbar sind.

UNet bietet dafür eigentlich das *Network Transform Script*, welches vollautomatisch die Synchronisation eines Objekts an allen Endpunkten übernehmen sollte. Aufgrund der besonderen Art der Modellverteilung und dem dynamischen Laden im Prototyp versagt es jedoch seinen Dienst. Hintergründe dazu finden sich unter *4.6.2 Unity UNET* im Kapitel *4.6 Schwierigkeiten bei der Umsetzung*. Aufgrund dessen wurde auch hier ein eigenes Skript geschrieben, das ***SyncScript***. Es

synchronisiert Position, Rotation, Skalierung und das Material eines Objekts über das Netzwerk, jedes manipulierbare Objekt bekommt dieses Skript zugewiesen. Dafür benutzt es *Unity SyncVars*. Bestimmte Arten von Klassenvariablen können als *SyncVar* deklariert werden. Die Werte dieser Variablen werden stets bei allen Clients abgeglichen und aktualisiert. Die Deklaration geschieht durch das Voranstellen des Schlüsselwortes:

```
1 [SyncVar]
2 private Vector3 syncPos;
3 [SyncVar]
4 private Vector3 syncScale;
5 [SyncVar]
6 private Quaternion syncRotation = Quaternion.identity;
```

Für das Übertragen der räumlichen Daten wird in der *FixedUpdate*-Methode des *SyncScripts*, welche immer in gleichem zeitlichen Abstand unabhängig von der Framerate ausgeführt wird, *TransmitTransform()* aufgerufen:

```
1 [ClientCallback] //diese Methode wird nur auf Clients ausgeführt
2 void TransmitTransform(){
3
4   if (hasAuthority &&
5     (Vector3.Distance(object.localPosition, lastPos) > 0.1 ||
6     Quaternion.Angle(object.rotation, lastRotation) > 5.0 ||
7     Vector3.Distance(object.localScale, lastScale) > 0.1 ))
8   {
9     //Client hat Autorität -> ist der Host und Transformationen überschreiten
      Schwellenwerte
10
11    //RPC auf Server, der das Objekt aktualisiert
12    CmdSendTransform(object.localPosition, object.rotation, object.localScale);
13
14    //Transformationen zwischenspeichern zum späteren Abgleich mit den
      Schwellenwerten
15    lastPos = object.localPosition;
16    lastRotation = object.rotation;
17    lastScale = object.localScale;
18  }
19 }
```

Die in dem Code per *RPC* auf dem Server aufgerufene Methode aktualisiert die *SyncVars* des Objekts mit den übermittelten Werten aus der *TransmitTransform*-Methode:

```
1 [Command] //RPC auf Server
2 void CmdSendTransform(Vector3 pos, Quaternion rot, Vector3 scale){
3   syncPos = pos;
4   syncRotation = rot;
5   syncScale = scale;
6 }
```

Da es sich bei den Variablen um *SyncVars* handelt, werden diese automatisch bei allen verbundenen Clients aktualisiert. Damit die 3D-Objekte dort nicht ruckartig springen, falls eine Bewegung zu schnell war oder übermittelte Werte wegen schlechter Netzwerkbedingungen auf dem Transportweg verloren gehen, wird zwischen den aktuellen und den übermittelten Transformationswerten per linearer Interpolation überblendet, die Methode wird in jedem Frame aufgerufen:

```
1 void LerpTransform(){
2   if (!hasAuthority) //Nur wenn es ein purer Client ist und nicht der Host
3   {
4     object.localPosition = Vector3.Lerp (object.localPosition, syncPos, Time.
      deltaTime * lerpRate);
5     object.localScale =   Vector3.Lerp (object.localScale, syncScale, Time.
      deltaTime * lerpRate);
6     object.rotation = Quaternion.Lerp (object.rotation, syncRotation, Time.
      deltaTime * lerpRate);
7   }
8 }
```

Der Abgleich der Materialwahl läuft analog ab, hier wird über eine Integer-*SyncVar* der Index des aktuellen Materials im Materialarray des *MaterialChanger*-Skripts synchron gehalten, eine Interpolation gibt es dabei nicht.

4.5 Frontend

Neben der Entwicklung des Backends wurde der Fokus auch auf das Frontend gelegt. Die App sollte optisch ansprechend und dabei leicht zu bedienen sein. Der Nutzer sollte über die Vorgänge, die ihn betreffen, ausreichend Feedback erhalten. Für die Farbgestaltung der optischen Elemente wurde eine Palette an harmonierenden Farben definiert, das entwickelte Farbschema ist in Abbildung 4.17 dargestellt.

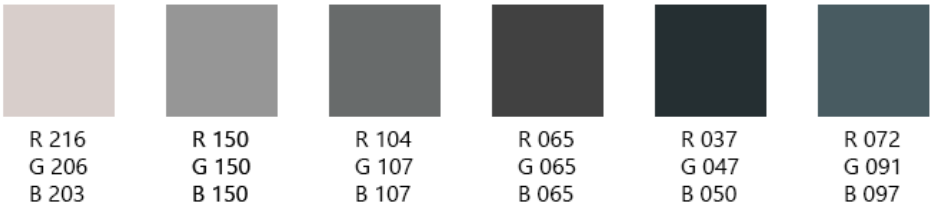


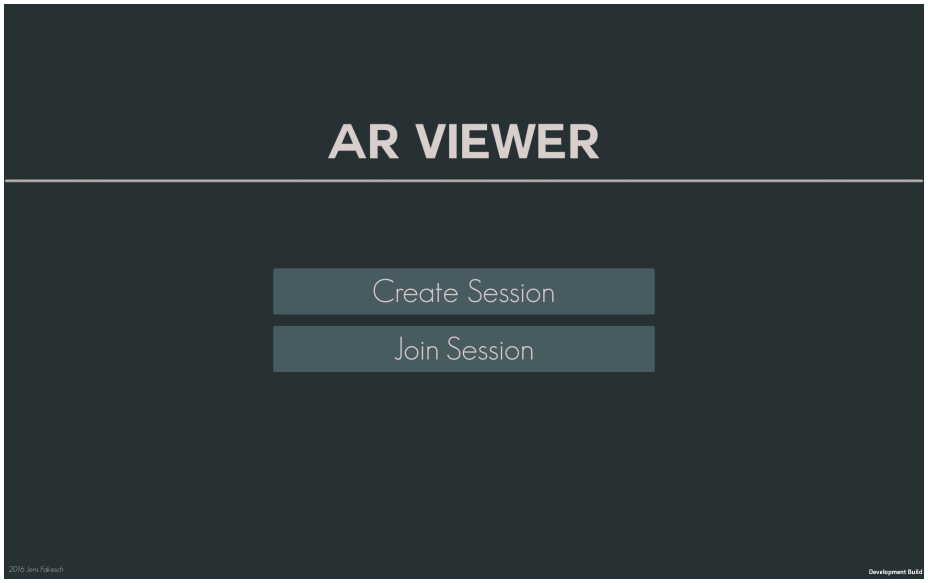
Abbildung 4.17: Die Farbpalette der Anwendung.

Zur Umsetzung des Interfaces wurde das *Unity*-eigene UI System verwendet. Es bietet die üblichen Elemente wie ein *Canvas*, *Buttons* oder *Labels*. Zusätzlich enthält es eine komfortable Layoutverwaltung mit *Grid-* und *Linear Layouts*. Auch die Eventabfrage zeigt sich leicht verständlich und meist fehlerfrei.

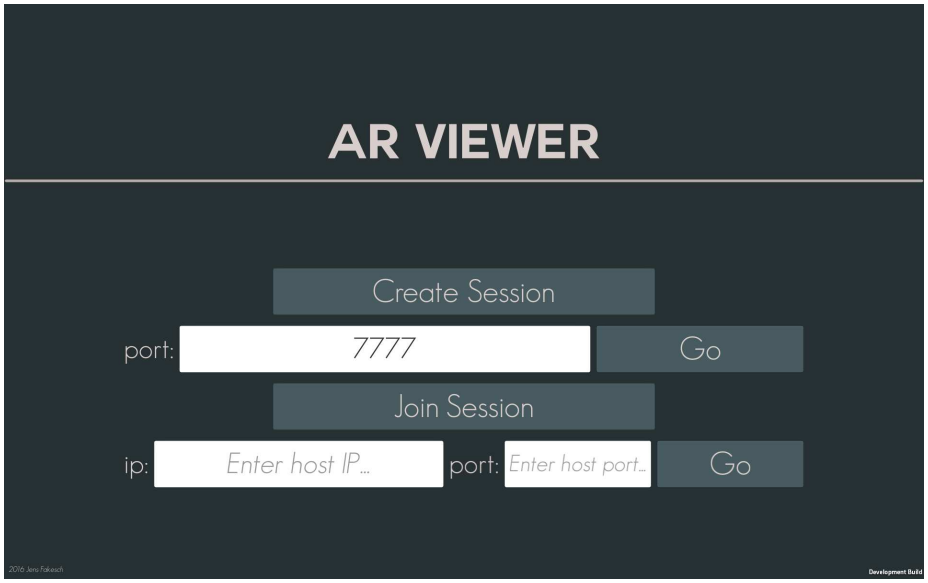
Das Userinterface ist in grundsätzlich zwei Ansichten geteilt. Die Anwendung startet in das Hauptmenü und wechselt nach erfolgreichem Erstellen oder Beitreten einer Session in die Präsentationsansicht, der eigentlichen Hauptoberfläche mit weiteren Untermenüs.

4.5.1 Hauptmenü

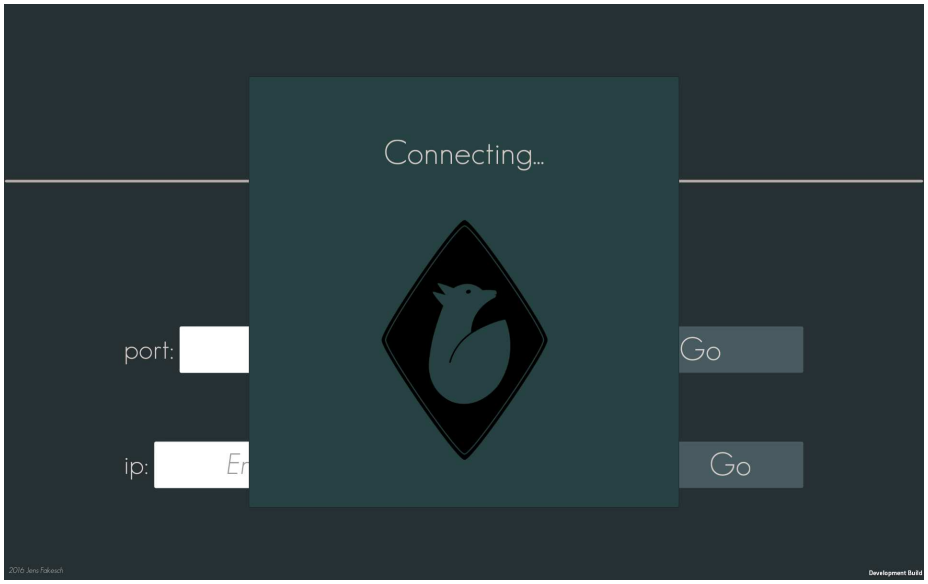
Das Hauptmenü begrüßt den Nutzer nach dem Starten der App und ist bewusst einfach und klar gehalten (siehe Abbildung 4.18a). Es gibt zunächst nur die Möglichkeit, zwischen dem Erstellen einer Session (entspricht Rolle des *Presenters*) und dem Beitreten zu einer vorhandenen Session (Rolle des *Spectators*) zu wählen. Erst nachdem man sich für eine Rolle entschieden hat, wird nach Tap auf den Button das entsprechende Untermenü mit den weiteren benötigten Eingaben animiert ausgeklappt. Die Ansicht mit beiden ausgefahrenen Untermenüs zeigt Abbildung 4.18b. Für das Hosten einer Präsentationssession muss zum Aufbau der Verbindung zu den Clients ein freier Port angegeben werden. Auf diesem Port sucht die Anwendung nach eingehenden Verbindungen. Möchte man einer bestehenden Session beitreten, wird die IP-Adresse des Hosts und ebendieser Port benötigt. Diese Informationen werden dem *Presenter* in seiner Präsentationsansicht angezeigt, sodass er diese Informationen nur an die *Spectators* weitergeben muss. Die im Hauptmenü getätigten Eingaben wie IP und Port werden zudem persistent gespeichert.



(a) Die Ansicht nach dem Start.



(b) *Create Session*- und *Join Session*-Tabs aufgeklappt.



(c) Die animierte Ladeanzeige nach dem Tap auf *Go*.

Abbildung 4.18: Der Weg durch das Hauptmenü.

Es ist als nicht nötig, diese Daten bei jedem Start neu angeben zu müssen, sofern man sich zu dem selben Host verbindet. Der anschließende Tap auf die *Go*-Schaltfläche baut die Verbindung auf und lädt die Hauptszene. Feedback darüber bekommt der User über eine Einblendung (siehe Abbildung 4.18c), die besteht bis die Anwendung in den nächsten Bildschirm wechselt, die Präsentationsansicht.

4.5.2 Präsentationsansicht

Die Präsentationsansicht ist das eigentliche Gesicht der App (siehe Abbildung 4.19). Alle Hauptfunktionen laufen über dieses Interface und der Benutzer findet sich in ihr wieder, sobald er das Hauptmenü verlassen hat. Auf der gesamten Fläche ist der Blick durch die Kamera des Endgerätes angezeigt, hier werden nach Erkennen des Trackers auch die 3D-Modelle dargestellt und bewegt.

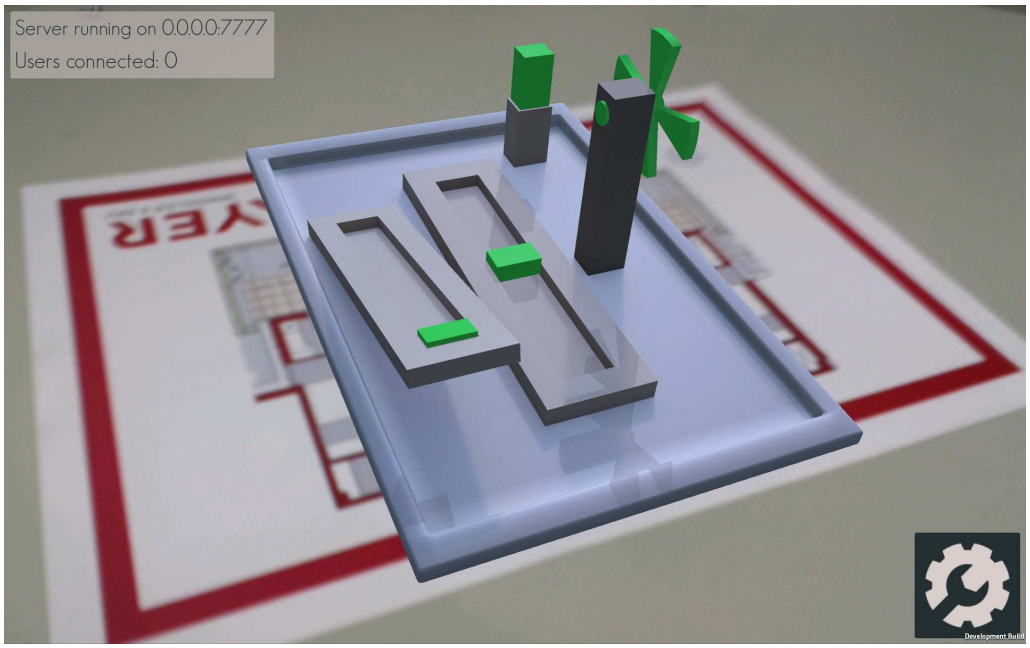


Abbildung 4.19: Die Präsentationsansicht des *Presenters*.

In der oberen linken Bildschirmecke befindet sich ein kleines Informationsfenster. Für den *Presenter* werden hier seine IP-Adresse und der gewählte Port angezeigt, so dass er diese Informationen an die Clients weitergeben kann. Zusätzlich wird ihm die Anzahl der verbundenen Nutzer angezeigt sowie eine Rückmeldung, während ein Modell per Netzwerk an einen Client übertragen wird. Dem *Spectator* wird hier stattdessen angezeigt, dass er korrekt mit einem Server verbunden ist. Der einzige Button ist der Menübutton in der unteren rechten Ecke. Ein Tap auf diese Schaltfläche mit dem Einstellungssymbol fährt das Menü von der Seite herein oder blendet es wieder aus. Abbildung 4.20 zeigt das ausgeklappte Menü mit seinen vier Buttons. Sie bieten von links nach rechts folgende Funktionen:

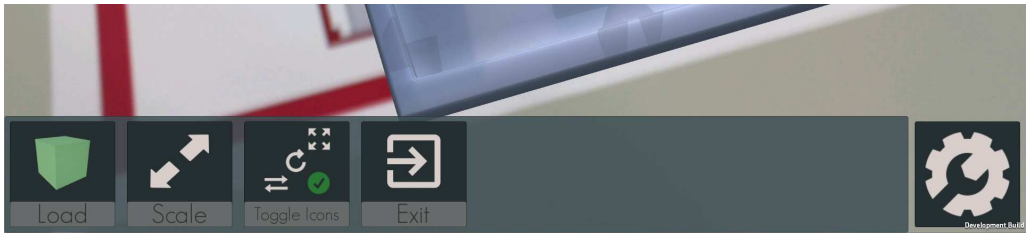


Abbildung 4.20: Das aufgeklappte Menü der Präsentationsansicht eines *Presenters*.

- ***Load***: Dieser Button ist zum Laden eines Modells vorgesehen, bei einem *Spectator* wird er nicht angezeigt, da er den Ladebefehl vom Server (*Presenter*) übermittelt bekommt. Das nach Tap geöffnete Fenster zeigt den Modellbrowser, aus welchem das gewünschte Modell aus den *AssetBundles* in die Hauptszene geladen werden kann (siehe Abbildung 4.21). Ein Textfeld am oberen Rand des Bildschirm zeigt den Dateipfad an, aus welchem die Daten gelesen werden. *AssetBundles* müssen an diesen Ort kopiert werden. Dabei handelt es sich um den Ordner *ARModels* auf der SD-Speicherkarte beziehungsweise dem internen Speicher des mobilen Endgeräts. Da die Daten beim Erstellen einer Session eingelesen werden gibt es einen *Reload*-Button in der unteren linken Ecke, um den Einlesevorgang manuell erneut zu starten, falls zur Laufzeit *AssetBundles* verändert wurden. In der rechten unteren Ecke befindet sich die *Close*-Schaltfläche zum Schließen des Modellbrowsers. Die Anzeige der in den Bundles gefundenen *Prefabs* ist scrollbar und passt sich dynamisch der Anzahl an Modellen an. Die einzelnen Modellbuttons zeigen den Namen des *Prefabs* sowie ein Vorschaubild, sofern dieses vor dem Export aus *Unity* per *Thumbnail-Script* festgelegt wurde. Wurde keines definiert wird ein Standardbild, ein grüner Würfel dargestellt. Nachdem eine Auswahl getroffen und per Tap eingegeben wurde, wird ein *Loading Model...*-Informationsfenster angezeigt, um dem User über den Ladevorgang Feedback zu geben. Der Modellbrowser wird anschließend selbstständig geschlossen und gibt wieder den Blick auf die *Präsentationsansicht* frei.

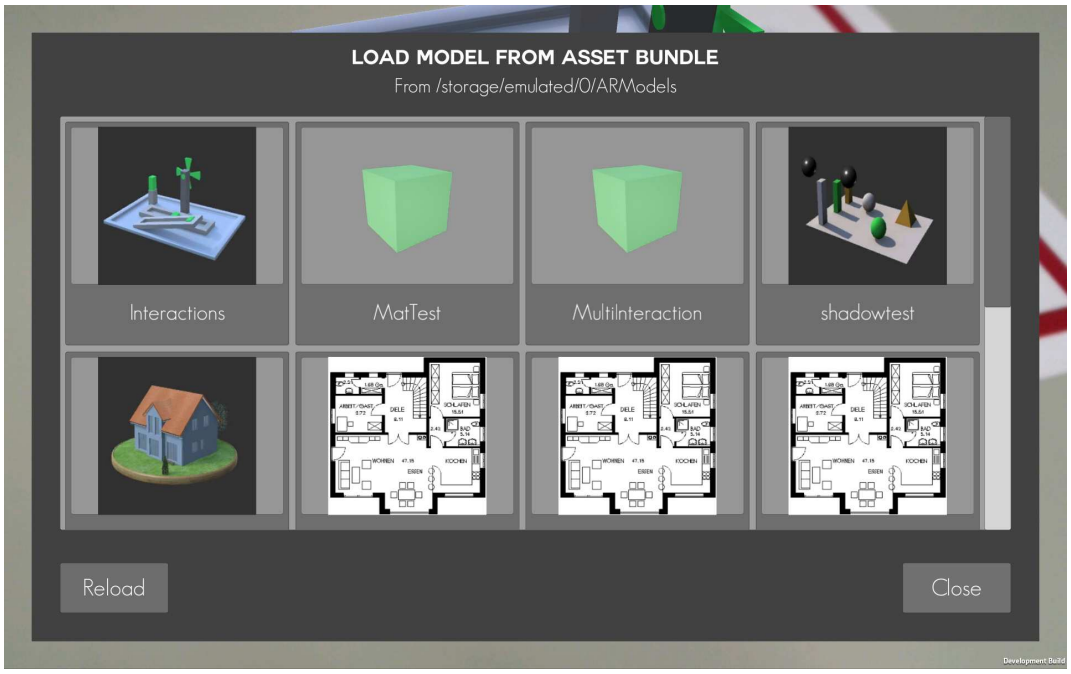


Abbildung 4.21: Der Modellbrowser nach Tap auf *Load*.

- ***Scale***: Ein Tap auf diesen Button öffnet oder schließt den *Scale*-Slider (siehe Abbildung 4.22). Mit ihm kann das Modell als Ganzes skaliert werden. Die Gesamtgröße wird nicht über das Netzwerk an alle Nutzer übertragen, da diese Einstellung dazu gedacht ist, das komplette Modell an seine eigene Bildschirmgröße oder Präferenz anzupassen. Aufgrund dessen ist die Schaltfläche bei *Presenter* sowie *Spectator* zu finden. Zusätzlich findet sich hier der *Automatic*-Button, der die Gesamtausmaße der Geometrie ermittelt und das Modell so skaliert, dass es den gesamten Tracker füllt. Diese Funktion wird auch stets automatisch beim Laden eines Modells ausgeführt.

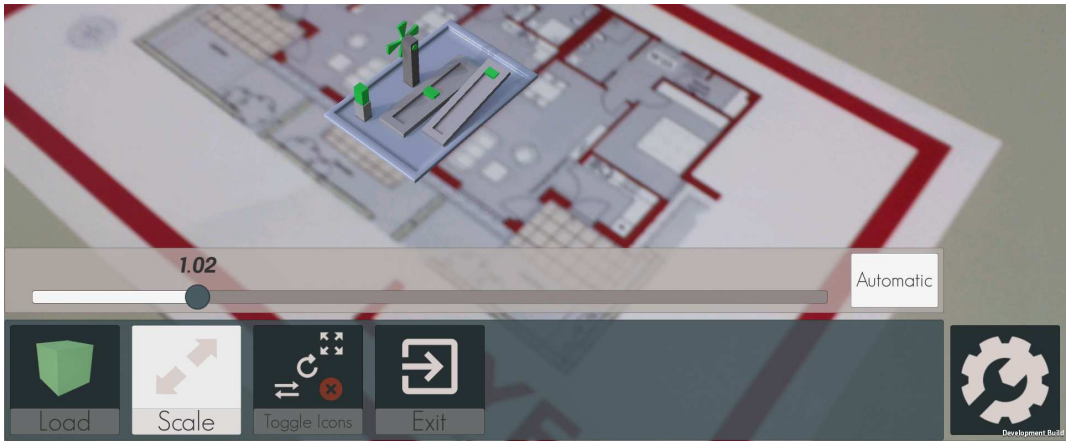


Abbildung 4.22: Der *Scale*-Slider.

- ***Toggle Icons***: Mithilfe dieser Schaltfläche können die Interaktionsicons aktiviert und deaktiviert werden. Diese werden auf den manipulierbaren Objekten angezeigt und folgen ihnen stets in passender Größe, so dass der Nutzer erkennen kann, welche Aktion er bei welchem Objekt ausführen kann (siehe Abbildung 4.23). Alle vier Möglichkeiten der Manipulation (Transformation, Rotation, Skalierung und Materialwechsel) besitzen ein eigenes leicht erkennbares Symbol. Erlaubt ein einzelnes Objekt mehrere Interaktionsarten gleichzeitig, werden die entsprechenden Icons vertikal untereinander angeordnet angezeigt. Die Interaktionsicons sind nur bei dem *Presenter* sinnvoll, weshalb auch nur er die Schaltfläche sehen kann.

Abbildung 4.23: Die Interaktionsicons.

- ***Exit***: Der letzte Button dient dem Verlassen der Session. Im Falle des *Presenters* wird die Session beendet und alle Teilnehmer finden sich in ihrem Hauptmenü wieder. Verlässt ein *Spectator* über diese Schaltfläche die Session, wird lediglich seine eigene Verbindung zum Server unterbrochen und nur er wird in das Hauptmenü geleitet. Dem *Presenter* wird das durch Dekrementieren der Clientanzahl in seinem Informationsfenster im Hauptbildschirm oben links angezeigt.

Während der Dateiübertragung beim Senden eines *AssetBundles* vom Server zum Client wird dem *Spectator* das in Abbildung 4.24 gezeigt Informationsfenster gezeigt, um ihn nicht über den Fortschritt im Unklaren zu lassen. Das Popup enthält einen sich füllenden Ladebalken, eine Anzeige der bereits übertragenen Megabyte neben der zu transferierenden Gesamtgröße der Datei sowie den Namen des *AssetBundles*. Nach erfolgreicher Übertragung schließt sich das Fenster wieder und das Modell wird angezeigt.

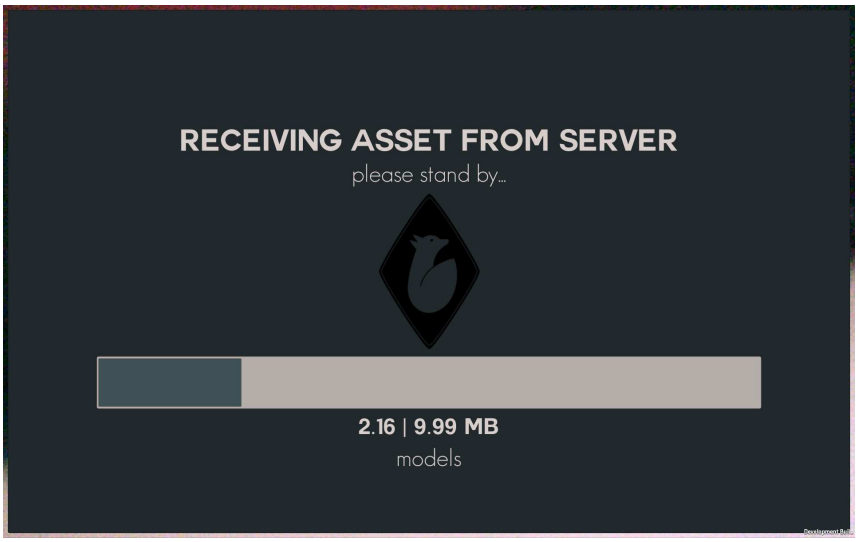


Abbildung 4.24: Das Fortschrittsinformationsfenster beim Empfangen eines *AssetBundles*.

4.6 Schwierigkeiten bei der Umsetzung

Dieser Abschnitt enthält eine Erläuterung der während der Entwicklung aufgetretenen großen Probleme und wie sie gelöst wurden. Die größten Schwierigkeiten zeigten sich bei der Umsetzung der Touchbedienung sowie den komplexen Netzwerkfunktionen mit dynamisch ladbaren Modellen.

4.6.1 Touchbedienung

Da der Nutzer die Möglichkeit haben sollte, bestimmte Objekte auf dem Endgerät zu manipulieren, war es notwendig, eine gut funktionierende Touchbedienung zu implementieren. Erste Ansätze verzichteten auf die Verwendung vom externen Bibliotheken und nutzen nur die Bordmittel *Unitys*. *Unity* liefert eine grundlegende Erkennung von Touches, so zum Beispiel Beginn, Bewegung und Ende einer Eingabe. Zurückgegeben werden die Bildschirmkoordinaten, also an welcher Stelle der Bildschirm berührt wurde. Davon ausgehend kann ein *Raycast* geschossen werden, ein Strahl der von diesem Punkt aus, abhängig von der Kamera, perspektivisch korrekt durch die dreidimensionale Szene geschickt wird. Überschneidungen mit 3D-Modellen werden erkannt und dementsprechend können Aktionen ausgeführt werden.

Die Umsetzung von komplizierteren und beschränkten Gesten, wie es bei dieser Anwendung notwendig war, ist so zwar prinzipiell möglich, die Implementierung von Grund auf ist allerdings nicht sinnvoll. Vor allem die Einschränkung der erlaubten Bewegungsbereiche durch die Positionen oder Namen der in der Objekthierarchie verwendeten leeren Objekte ist nicht trivial, ebenso die korrekte Berechnung von Zielpositionen abhängig von der Sicht auf die Szene. Objekte im dreidimensionalen Raum sollten nur auf einer Ebene, also zwei Achsen bewegt werden, drei Achsen lassen sich durch lediglich eine räumliche Ansicht nicht mehr kontrollieren. Diese Ebenen liegen jedoch nicht immer plan in der Szene auf Weltachsen sondern können auch geneigt sein, was eine zusätzliche Schwierigkeit darstellte. Es gibt diverse externe *Unity*-Libraries, welche nach Import in ein Projekt diverse Grundinteraktionen bereitstellen. Auf dieser Grundlage kann weiter

aufgebaut werden, die enthaltenen Skripte können verändert und erweitert werden, um sie den eigenen speziellen Anforderungen anzupassen.

Die Verwendung der *TouchScript*-Library erwies sich als gute Wahl. Die grundlegende Anwendung und Konfiguration wurde bereits in *4.4.2 Touchbedienung mit TouchScript* erläutert. Das Versehen von Modellen mit einfachen Interaktionen ist komfortabel und simpel. Für den Prototyp waren jedoch einige Anpassungen notwendig:

- ***Aufteilen der Grundtransformationen***: Objekte die manipulierbar sein sollen, müssen ein *Gesture Script* besitzen. Davon gibt es verschiedene Versionen, für die meisten Transformationen eignet sich das *Transform Gesture Script* (siehe Abbildung 4.25). In diesem kann unter *Projection Type* die Projektionsebene der Bewegung definiert werden. Es gibt die Auswahlmöglichkeiten *Layer* (Ebene frontal zur Kamera gerichtet), *Object* (Ebene abhängig von der Orientierung des Objekts) und *World* (Ebene abhängig von den Weltkoordinaten). Möchte man auf einem *Gameobject* mehrere Transformationen gleichzeitig erlauben, könnte man mit wenig Aufwand einfach alle Auswahlhaken des *Transform Types* aktivieren. Für den Prototyp ist das nicht anwendbar. Durch die Beschränkungen auf bestimmte Transformationsachsen über die Objekthierarchie muss für jede Grundinteraktion eine Unterscheidung stattfinden. Das Objekt erhält also per Code beim dynamischen Ladevorgang für Translation, Rotation und Skalierung ein eigenes Gestenskript mit speziell angepassten Projektionseinstellungen. Translationen sind auf die *Layer*-Ebene eingestellt, Rotation und Skalierung besitzen hingegen den *Projection Type Object*. Die Normale der Ebene wird hierbei während des Ladens anhand des Namens des leeren Transformationsunterobjekts (Name enthält die erlaubten Achsen) berechnet und angepasst. Da sich nun mehrere Gestenskripte auf einem einzigen Objekt befinden, müssen sie schließlich noch gegenseitig als *friendly* definiert werden, um sich nicht zu behindern. Das Problem der Projektionsebenen ist damit gelöst, die Einschränkung der Bewegung noch nicht.

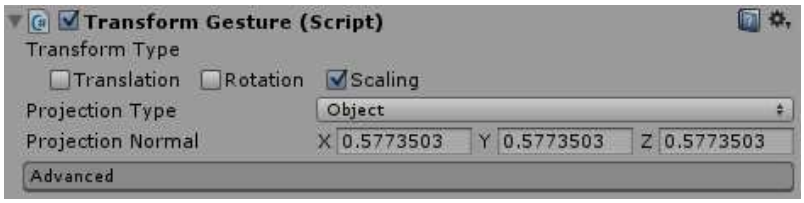


Abbildung 4.25: Das *Transform Gesture Script*

- ***Einschränken der Translation***: Da es nur erlaubt sein sollte, Objekte zwischen den über die leeren Unterobjekte definierten Positionen zu bewegen, wurden Anpassungen im Code der *TouchScript*-Skripte selbst notwendig. Das *Transform Gesture Script* regelt die Berechnung der Zielkoordinaten einer Interaktion und schließlich die Versetzung des Objekts. Um die Bewegung zu beschränken, muss also an dieser Stelle eingegriffen werden. Der folgende Code wurde kurz vor Anwendung der Bewegung eingefügt und beeinflusst die Berechnung der Zielposition, er ist leicht vereinfacht dargestellt:

```
1  if (DeltaPosition != Vector3.zero) {
2    //Erlaubter Start- und Endpunkt der Bewegung
3    Vector3 startPos = object.startGo.position;
4    Vector3 goal = object.goalGo.position;
5
6    //Projezierter Zielpunkt des Objekts auf einer
7    //Geraden durch Start- und Endpunkt
8    Vector3 projectedLocalPoint = Vector3.Project((object.position +
       DeltaPosition) - startPos, goal - startPos);
9    //Und in globalen Koordinaten
10   Vector3 projectedPoint = projectedLocalPoint + startPos;
```

```
11    //Distanz zwischen Start- und Endpunkt
12    float dist_start_goal = Vector3.Distance(startPos, goal);
13    //Distanz zwischen Ziel- und Endpunkt
14    float dist_point_goal = Vector3.Distance(projectedPoint, goal);
15    //Distanz zwischen Start- und Zielpunkt
16    float dist_start_point = Vector3.Distance(startPos, projectedPoint);
17
18    if (dist_start_goal == dist_start_point + dist_point_goal) {
19      //Zielpunkt gültig, versetze Objekt
20      object.position = projectedPoint;
21    }
22  }
```

Der Code nutzt zur Evaluation, ob der berechnete Zielpunkt zwischen den erlaubten Punkten und nicht außerhalb von ihnen liegt, die in Abbildung 4.26 dargestellten Zusammenhänge. Auf diese Weise kann das manipulierbare Objekt nur auf einer Geraden zwischen dem vorher festgelegten Start- und Endpunkt verschoben werden und schießt nicht über sie hinaus.

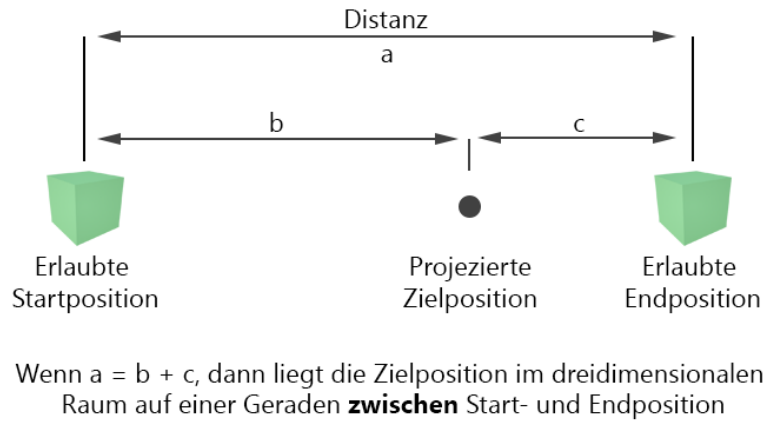


Abbildung 4.26: Hintergrund der Berechnung eines gültigen Ziels

4.6.2 Unity UNET

Auch wenn *Unity* mit seinen *UNet*-Netzwerkfunktionen viele vorgefertigte und manchmal hilfreiche Skripte bietet, waren diese gleichzeitig die Quelle vieler Probleme. Die fertigen Elemente arbeiten bei sehr genau definierten Situation durchaus praktikabel, sobald jedoch leichte Abweichungen in Spielerverwaltung oder Anwendungsstruktur auftreten versagen sie häufig ihren Dienst oder erfüllen die Ansprüche an Performanz nicht mehr. In diesen Fällen ist man damit besser beraten, die vorhandenen Skripte zu verändern, zu erweitern oder auch komplett neu zu entwickeln. Im Laufe der Implementierung des Prototyps war dies an zwei wichtigen Stellen notwendig, welche sogar Teile Hauptfunktionen der Anwendung darstellen:

- ***Beim Übertragen der AssetBundles***: *UNet* besitzt ein eigenes Nachrichtensystem, das verwendet werden kann, um Informationen und Daten zwischen bestimmten Endpunkten zu übertragen. Sogar der *QoS*-Kanal lässt sich dabei selbst bestimmen, was ein positiver Aspekt ist. *QoS* steht für *Quality of Service* und bestimmt, wie wichtig die korrekte Paketreihenfolge oder das tatsächliche Erreichen des Ziels sind. So gibt es *Reliable* (Paket erreicht auf jeden Fall den Empfänger = langsamer) und *Unreliable* (Paket darf auf dem Weg verloren gehen = schneller) Kanäle. Diese gibt es zusätzlich jeweils noch in der Version *Fragmented* und *Sequenced*. *Fragmented* erlaubt größere Nachrichten bis 32 Fragmente, *Sequenced* sorgt dafür, dass die Pakete auf jeden Fall in der korrekten Reihenfolge ihr Ziel erreichen,

was die Übertragung jedoch verlangsamt. Abbildung 4.27 zeigt die Ordnung der Kanäle. Es muss also stets zwischen Geschwindigkeit und Zuverlässigkeit abgewogen werden.

Abbildung 4.27: Die *QoS*-Kanäle von *UNet*.

Für das einfache Mitteilen von kurzen Nachrichten ist das System komfortabel. Es eignet sich jedoch keineswegs für die Übertragung größerer Datenmengen. Das zeigte sich bei den ersten Ansätzen zum Übermitteln der *AssetBundles* vom Server zu den Clients. Ein Problem dabei ist die Beschränkung der Größe der einzelnen Nachrichten. Die zu sendende Datei muss manuell in Blöcke von passender Größe aufgeteilt werden und auf der Gegenseite wieder korrekt zusammengesetzt werden. Die nichtfragmentierten Kanäle erlauben eine Blockgröße von nur 1500 Bytes, die fragmentierten immerhin bis zu 64 Kilobytes, sie werden dann intern in kleinere Pakete geteilt. Durch die Größe der Modelldaten inklusive aller Texturen wird so eine sehr hohe Anzahl an einzelnen Nachrichten benötigt. Da jede Nachricht zusätzlich einen Header enthält, welcher weitere Informationen enthält, fällt viel Overhead (Daten, die nicht der Nachrichteninhalt selbst sind) an. Zudem muss jedes Paket auf Seiten des Senders und Empfängers einzeln bearbeitet werden, das heißt in einen Zwischenpuffer oder die Datei gespeichert werden. All dies kostet viel Zeit. Auch mit dem schnellsten und „größten“ *QoS*-Kanal *UnreliableFragmented* war die Datenübertragung unabhängig vom Netzwerk selbst inakzeptabel unperformant.

Die schließlich verwendete optimale Lösung ignoriert das *UNet*-System und nutzt die klassische TCP-Datenübertragung. Die genaue Umsetzung mit Network- und Filestreams wurde in *4.4.4 Netzwerkkommunikation* erläutert. Der Ansatz zeigte sich als problemloser in der Implementierung, da durch die Verwendung der Streams keine manuelle Fragmentierung notwendig war, sowie um ein vielfaches schneller bei der Übermittlung der Daten.

- ***Bei der Synchronisation der manipulierbaren Objekte***: Auch hier wurde zunächst versucht, die *Unity*-eigenen Netzwerkfunktionen zum Abgleich der Transformation zu nutzen. Damit *Gameobjects* von *UNet* als netzwerkbeeinflusst erkannt werden, benötigen sie das fertige *Network Identity Script*. In diesem kann festgelegt werden, ob der Nutzer dieser Instanz der Anwendung die Autorität über das Objekt hat. Er gilt dann als Besitzer des Objekts und darf es verändern. Erhält das Objekt zusätzlich ein *Network Transform Script* werden die Veränderungen auf seiner Seite automatisch an alle anderen Clients, welche nicht die Autorität besitzen, übertragen. Damit die Verbindung zwischen den Objekten an den verschiedenen Endpunkten zustande kommt, muss es bei den Clients per Spawnbefehl vom Server instanziiert werden. Nur dann weiß der Server, an welche Instanzen im Netzwerk er die Updatebefehle schicken muss. Das *Gameobject* muss also schon bei der Kompilierung der Anwendung in ihr verfügbar sein und wird dann als spawnbares Objekt registriert. Der Server sendet dann nur noch den Befehl zum Einfügen dieses vorher registrierten Objekts in die Spielwelt.

  Da die Anwendung aber das dynamische Laden unterstützt sind die zu spawnenden Modelle nicht von Anfang an verfügbar. Die Zuordnung eines Objekts zu einem bestimmten Spawnbefehl muss also manuell umgesetzt werden. Das geschieht bei dem Einlesen eines Modells aus dem *AssetBundle* im *InitObjects*-Skript auf Clientseite mit den folgenden Pseudecodezeilen:

```
1  //Eindeutiger Networkhashwert wird aus verschiedenen
2  //Eigenschaften des Modells erstellt.
3  NetworkHash128 nethash = NetworkHash128.Parse (loadedObject.name + "" +
       loadedObject.position.x + "" + loadedObject.rotation.x + "" +
       loadedObject.GetComponent<MeshFilter> ().mesh.vertexCount);
4
5  //Instanz des geladenen Objekts wird erstellt
6  GameObject instance =
7  (GameObject)Instantiate (loadedObject, loadedObject.position, Quaternion.
       identity);
8
9  //und wird in einem Dictionary mit dem berechneten Hashwert als Schlüssel
       gespeichert
10 objectsDict.Add (nethash, instance);
11
12 //Das orignale Objekt wird aus der Szene gelöscht, es wird später vom
13 //Server gespawnt
14 GameObject.Destroy (loadedObject);
15
16 //Ein SpawnHandler wird registiert, der bei Spawn- und Despawnbefehl
17 //des Servers die Methoden SpawnObject und UnspawnObject aufruft.
18 ClientScene.RegisterSpawnHandler( nethash, SpawnObject, UnspawnObject );
```

Die Methode *SpawnObject* sucht aus dem Dictionary (ein Wörterbuch mit der Struktur Schlüssel -> Inhalt) das dem vom Server per Spawnbefehl gesendeten *NetworkHash* entsprechende *Gameobject* heraus und versieht es mit dem selbstgeschriebenen *SyncScript* zur Synchronisation des räumlichen Zustands. Dieses Skript wurde in *4.4.4 Netzwerkkommunikation* beschrieben. Das Object wurde nun korrekt gespawnt und die Verbindung zum Server und damit der Abgleich der Daten steht.

Wie sich gezeigt hat ist *UNet* ein zweischneidiges Schwert. Einerseits erleichtert es dem Programmierer viele Dinge durch das Bereitstellen fertiger Skripte und Methoden, zum Beispiel für den Aufbau von Sessions. So lassen sich schnell simple Netzwerkanwendungen entwickeln und erste Ergebnisse produzieren. Es ist jedoch nicht ratsam, sich zu sehr auf die eingebauten Netzwerkfunktionen zu fokussieren und zu verlassen. In einigen Fällen ist es in allen Aspekten effektiver, sich den tieferleveligen Netzwerkfunktionen der Programmiersprache selbst zu bedienen und die gewünschten Funktionen von Grund auf zu implementieren.

5 Nutzerstudie

Die Schattierungen der 3D-Modelle tragen einen sehr großen Teil zur Glaubhaftigkeit der AR-Welt bei. Wie im Kapitel *2 Stand der Technik* dargelegt gibt es zwar Ansätze, die versuchen, hochqualitative Echtzeitschatten und Lichteffekte wie globale Illumination umzusetzen, für mobile Endgeräte sind diese jedoch immer noch zu aufwendig und bei weitem nicht performant genug. Dynamische zur Laufzeit berechnete Schatten erreichen auch heute nicht die Darstellungsqualität von vorberechneten gebackenen Schatten. Es ist also bei AR-Anwendungen stets notwendig, zwischen hoher Qualität und Performanz der statischen Schatten und der geringeren Qualität und Performanz aber dynamischen Anpassungsfähigkeit der Echtzeitschatten abzuwägen.

Um diese Entscheidung nicht bei jeder Entwicklung, vielleicht sogar mit aufwendigen Testläufen, neu treffen zu müssen, wäre es hilfreich, für bestimmte Anwendungsfälle Leitlinien zu haben. Für manche Szenarien ist vielleicht die hohe Qualität der statischen Schatten gefragt, andere benötigen bewegliche Schatten falls es um Veränderungen der Lichtsituation geht. In einigen Fällen kann die Kombination der Ansätze der richtige Weg sein, in wiederum anderen haben die Schatten wenig bis keinen Einfluss auf die Lösbarkeit von bestimmten Aufgaben. Für diese Arbeit soll genau dies anhand dieser Nutzerstudie herausgefunden werden. Der Fokus liegt dabei auf Anwendungen im Bereich der Architekturvisualisierung, da dies ein perfekt geeignetes und weitläufiges Feld für AR-Umsetzungen ist. Der erste Abschnitt beschreibt die Gedanken und Vorbereitungen der Konzeptionsphase, gefolgt von den Details der Durchführung. Im letzten Teil werden die gesammelten Daten ausgewertet und überprüft, ob aufgestellte Hypothesen bestätigt oder widerlegt werden konnten.

5.1 Konzeption

Im Folgenden wird die Vorbereitung der Nutzerstudie beschrieben und welche Überlegungen im Vorfeld getätigt wurden. Das beginnt bei der Wahl der entwickelten Szenarien, gefolgt von den Methoden der Datenerhebung und schließlich werden die Hypothesen erläutert, die es zu überprüfen gilt.

5.1.1 Anwendungsfälle

Im Laufe der Benutzerstudie sollten drei verschiedene Anwendungsfälle überprüft werden. Jeder deckt eine andere Möglichkeit ab, für die Augmented Reality im Bereich der Architektur sinnvoll verwendet werden könnte. Für jedes dieser drei Szenarien musste ein eigenes 3D-Modell mit den dazugehörigen gebackenen und ungebackenen Texturen erstellt werden. Jedes davon wurde in drei Ausführungen getestet: nur statische Schattierung, nur dynamische Schattierung und zuletzt eine Kombination aus beidem. So sollten die Präferenzen der Nutzer im Bezug auf die Beleuchtung ermittelt werden. Die gewählten Anwendungsfälle sind die folgenden:

- ***Sonneneinstrahlung***: Dieses Szenario ermöglicht es dem Nutzer, die Sonneneinstrahlung auf einem Grundstück während des gesamten Tagesverlaufs zu beurteilen und damit die Anordnung von großen Objekten auf selbigem sinnvoll zu gestalten. Oberhalb des Hauses befindet sich eine Himmelsscheibe mit einer stilisierten Sonne. Diese Scheibe kann gedreht werden. Die Sonne zeigt dem Nutzer vereinfacht, aus welcher Richtung der Strahl des Tageslichts kommt. Ein gerichtetes Licht bescheint die Welt stets aus der über die Himmelscheibe definierte Richtung. Bei der dynamischen Beleuchtung sind so die daraus resultierenden dynamischen Schatten sichtbar. Die Aufgabe des Studienteilnehmers besteht nun darin, die Objekte im Garten so zu verschieben, dass auf die Terrasse des Hauses im kompletten Tagesverlauf möglichst wenig Schatten fallen. Abbildung 5.1 zeigt das Szenario.

Abbildung 5.1: Das Szenario *Sonneneinstrahlung* mit dynamischer Beleuchtung.

- ***Grundriss***: Hier soll der Einfluss der Schatten auf die Beurteilungsfähigkeit eines Grundrisses überprüft werden. Nach Laden des Modells sieht der Nutzer ein großes Haus. Das Dach inklusive der oberen Etage kann abgehoben werden, sodass der Blick auf das Innere frei wird. Im Erdgeschoss befindet sich ein Labyrinth. In einer Ecke ist der per Schriftzug definierte Start, in der diagonal gegenüberliegenden Ecke befindet sich das Ziel. Der Studienteilnehmer hat die Aufgabe, einzelne Wände innerhalb des Labyrinths so zu verschieben, dass ein Pfad vom Start zum Ziel freigegeben wird. Das Haus mit angehobenem Dach ist in Abbildung 5.2 dargestellt.

Abbildung 5.2: Das Szenario *Grundriss*.

- ***Materialität***: In dieser Szene ist die einzige Manipulationsmöglichkeit das Austauschen von Materialien. Abbildung 5.3 zeigt das Grundstück dieses Anwendungsfalls. Per Tap können die Materialien des Dachs, der Hauswand und der Terrasse durchgeschaltet werden. So lässt sich zum Beispiel zwischen einer Terrasse in Holzoptik oder zweierlei Steinarten wählen. Die Aufgabe des Studienteilnehmers ist hierbei, sich für ein Materialkombination zu entscheiden, die seinem eigenen Geschmack am ehesten entspricht. Durch die drei verschiedenen Beleuchtungseinstellungen lässt sich ermitteln, ob die Qualität der Schatten einen Einfluss auf die Beurteilungsfähigkeit von Oberflächeneigenschaften hat.

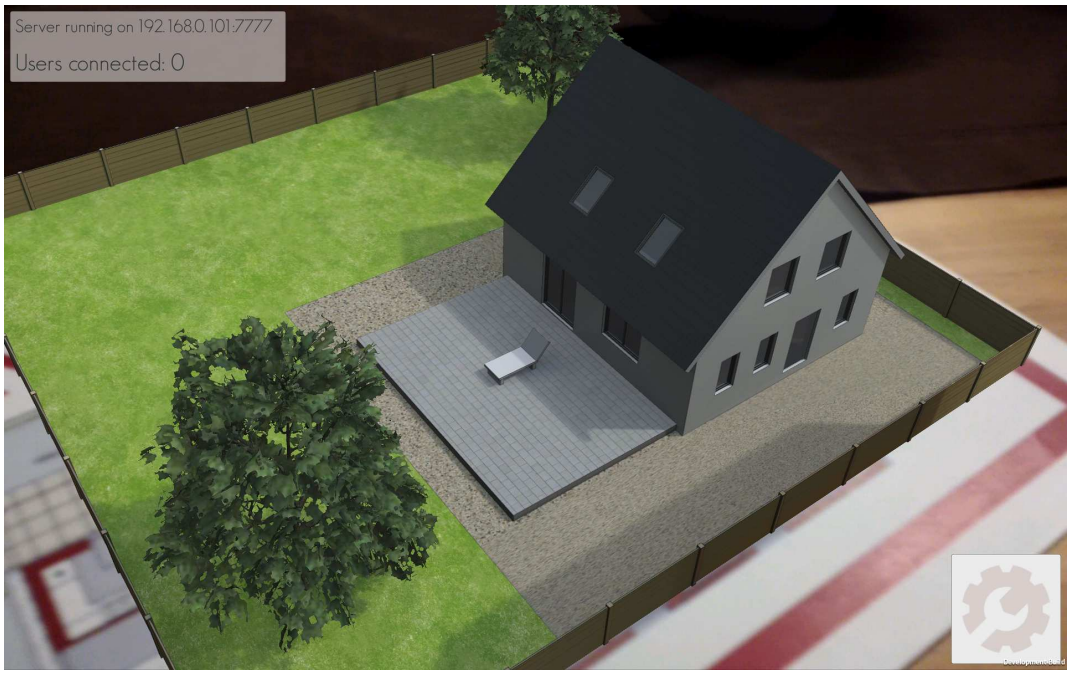


Abbildung 5.3: Das Szenario *Materialität*.

5.1.2 Nutzerbefragung

Um tiefere Einblicke in die Gedanken und Eindrücke der Teilnehmer zu erhalten ist ein Fragebogen Teil der Nutzerstudie. Nach erfolgreichem Lösen eines Anwendungsfalles werden die entsprechenden Fragen zeitnah beantwortet. Die Umfrage ist in vier große Abschnitte unterteilt:

1. ***Hintergrund***: Hier wird ermittelt, wie erfahren sich der Proband selbst im Umgang mit Smartphones, Tablets und Augmented Reality einschätzt. Die Skala reicht in fünf Punkten von *keine Erfahrung* bis *viel Erfahrung*.

2. ***Anwendung allgemein***: Fragen zur Erfahrung des Prototypen im Ganzen, unabhängig der einzelnen Anwendungsfälle. Hier werden zum Beispiel Aussagen getätigt wie „Es fiel mir leicht, ein Modell zu laden“ oder „Es fiel mir leicht, Objekte zu manipulieren“. Der Proband gibt an, wie stark er den Aussagen zustimmt. Die Skala ist in fünf Punkte unterteilt, von *stimmt nicht* bis *stimmt sehr*.

3. ***Anwendungsfälle***: Dies ist der Hauptteil des Fragebogens. Er ist in die drei einzelnen Anwendungsfälle unterteilt. Jeder dieser Abschnitte ist wiederum in drei Unterkategorien unterteilt, eine für jede Art der Beleuchtung (statisch, dynamisch, gemischt). Für jede wird ermittelt, wie leicht dem Teilnehmer die Aufgabe fiel und inwiefern er bei der Lösung durch die Schattierung irritiert oder unterstützt wurde. Bei der Version mit den kombinierten Schatten wird dabei noch zwischen den statischen und dynamischen Schatten unterschieden. Getätigte Aussagen sind zum Beispiel „Die festen Schatten haben mich behin-

dert“ oder „Die bewegten Schatten haben mir geholfen“. Der Proband gibt an, inwiefern er mit den Aussagen übereinstimmt. Die Skala besteht aus fünf Punkten von *stimmt nicht* bis *stimmt sehr*. Die Reihenfolge der Anwendungsfälle im Fragebogen lautet: Sonneneinstrahlung, Grundriss, Materialität. Nach Bearbeiten jedes Szenarios hat der Nutzer die Möglichkeit, aus den drei Beleuchtungsversionen einen Favoriten zu wählen, sofern er sich für einen entscheiden kann.

4. ***Demographie***: Der Fragebogen endet mit dem Demographieteil. Hier wird Alter und Geschlecht des Teilnehmers abgefragt.

Die komplette Nutzerbefragung besteht aus 46 Einzelfragen.

5.1.3 Hypothesen

Für die Nutzerstudie wurden drei Hypothesen aufgestellt, die mit den erhobenen Daten entweder bestätigt oder widerlegt werden sollten. Sie beziehen sich auf den Einfluss bestimmter Beleuchtungsarten auf die Lösbarkeit verschiedener Anwendungsfälle.

1. Geht es um die Position des Schattenwurfs werden klare scharfkantige dynamische Schatten präferiert.
2. Bei der Wahrnehmung eines Grundrisses spielt die Schattierung keine wichtige Rolle.
3. Bei der Beurteilung von Materialität bevorzugen Nutzer die hochqualitativen statischen Schatten.

Werden sie verifiziert, können sie als Richtlinien für den zukünftigen Einsatz verschiedener Schattenarten verwendet werden.

5.1.4 Datenlogging

Zusätzlich zu dem Fragebogen wurden quantitative Daten über eine Loggingfunktion in der Anwendung erhoben. Während der Studie wurden im Hintergrund für jedes geladene Modell folgende Informationen in einer Datei gespeichert: Datum und Uhrzeit, Name des geladenen Modells (entspricht Anwendungsfall), Anzahl der Touches und die Zeit, die der Proband für den Anwendungsfall benötigt hat. Ein Eintrag in die Logdatei sieht wie folgt aus: *2016_07_06_Wednesday_14_26_43_Model:_Material_gemischt_Touches:_33_Time Loaded:_01:35*. Mithilfe dieser Daten lassen sich eventuell zusätzliche Informationen gewinnen.

5.2 Durchführung

Die Nutzerstudie wurde über einen Zeitraum von zwei Wochen durchgeführt, an ihr nahmen insgesamt 20 Probanden teil (die genaue Zusammensetzung folgt in der Auswertung unter *5.3.1 Demographie und Hintergrund*). Der Versuchsaufbau war unkompliziert und leicht transportabel, da lediglich ein Platz an einem Tisch sowie ein ausgedruckter Tracker und ein mobiles Endgerät benötigt wurden. Als 2D-Tracker wurde aufgrund des thematischen Bezugs die Grundrisszeichnung eines Hauses verwendet. Das Motiv ist in Abbildung 5.4 zu sehen und wurde auf einem Papier in DIN A4-Größe ausgedruckt. Das Versuchsgerät war eines der bereits in der Entwicklung verwendeten *Nvidia SHIELD K1* Tablets. Per zufällig über die Website *random.org* generierter Zahlensequenz wurde bei jeder Durchführung die Bearbeitungsreihenfolge der Anwendungsfälle festgelegt. Dadurch wurde verhindert, dass die erste Aufgabe einer festen Reihenfolge im Fragebogen stets am schwierigsten zu lösen ist, da der Proband mit der Anwendung noch nicht vertraut ist. Um diesen Effekt zusätzlich abzuschwächen und erste Berührungsängste zu nehmen, wurde wertungsfrei zu Beginn das Modell *Interaktionsspielplatz* geladen, um dem Teilnehmer die Grundlagen der Augmented Reality App an sich näherzubringen.

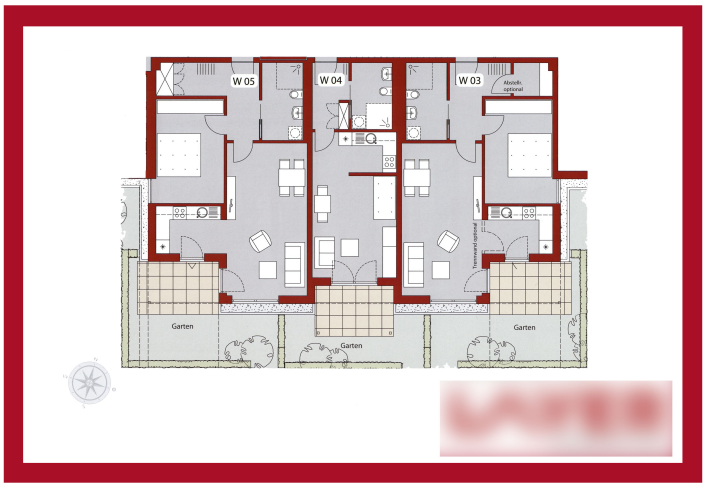


Abbildung 5.4: Der in der Nutzerstudie verwendete Tracker. Das Firmenlogo unten rechts wurde aus Lizenzgründen in dieser Abbildung unkenntlich gemacht.

5.3 Auswertung

Im Zuge der Auswertung wurden alle Fragebögen in einem *Microsoft Excel*-Dokument digitalisiert. Auf diese Weise ließen sich weitere Operationen auf den Daten ausführen und somit eine statistische Auswertung durchführen. Die ermittelten Ergebnisse werden in den nun folgenden Abschnitten erläutert.

5.3.1 Demographie und Hintergrund

An der Durchführung der Studie nahmen 20 Probanden teil. Der jüngste Teilnehmer war 21 Jahre alt, der älteste 60. Der Altersdurchschnitt betrug 29,2 Jahre. Elf davon waren weiblich, neun männlich.

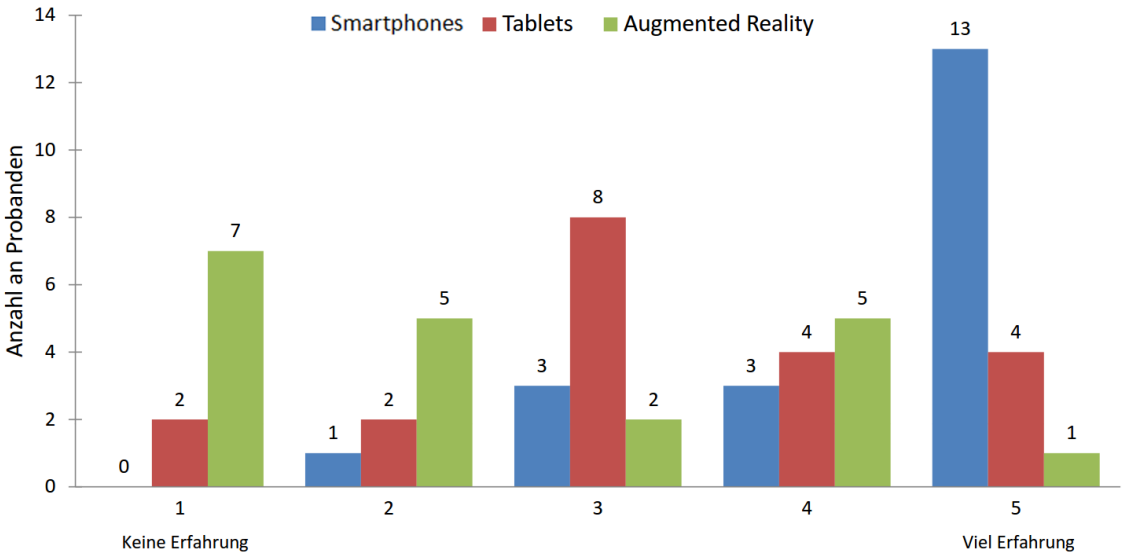


Abbildung 5.5: Erfahrungshintergrund der Probanden.

Der Großteil der Nutzer schätzte seine Erfahrung mit Smartphones sehr hoch ein. 65% gaben an, *viel Erfahrung* mit ihnen zu haben. Der Rest ordnete sich eher mittig ein mit leichter Nei-

gung in Richtung *viel Erfahrung*. Niemand gab an, sich mit einem Smartphone überhaupt nicht auszukennen. Anders stellte es sich bei Tablets dar. Hier war die Verteilung ausgeglichener, der größte Teil, nämlich 40%, schätze seine Erfahrung mit diesen Endgeräten durchschnittlich ein. Alle anderen verteilten sich relativ ausgewogen mit leichtem Schwerpunkt bei überdurchschnittlicher Erfahrung. Bei Augmented Reality zeigte sich die Verteilung grob gegensätzlich zu den Ergebnissen bei Smartphones, war jedoch etwas ausgeglichener. 35% der Teilnehmer gaben an, keinerlei Erfahrungen mit Augmented Reality zu haben. Die Anzahl sinkt in Richtung Ende der Skala ab, mit einer kleinen Spitze kurz vor *viel Erfahrung*. Abbildung 5.5 zeigt die Verteilungen übersichtlich und leicht vergleichbar in einer einzigen Grafik.

5.3.2 Anwendung allgemein

In dem Teil *Anwendung allgemein* des Fragebogens wurde die allgemeine Usability des Prototypen überprüft. Die gestellten Fragen zielten nicht auf bestimmte Anwendungsfälle ab sondern auf die Funktionen, die auf die Bearbeitung aller Aufgaben Einfluss hatten, wie zum Beispiel das Laden eines Modells oder die Freude an der Benutzung allgemein. Die Ergebnisse der Auswertung sind in Abbildung 5.6 dargestellt.

Insgesamt fiel die Beurteilung der Tester sehr positiv aus, der Schwerpunkt der Antworten befindet sich auf der zustimmenden Seite der positiven Aussagen. Besonders gut angenommen wurde die Modellladefunktion über den eingebauten Objektbrowser. 75% der Studienteilnehmer fiel es uneingeschränkt leicht, ein Modell zu laden. Die restlichen 25% ordneten sich nur knapp darunter ein. Ähnlich verhielt es sich bei dem Spaß, den die Benutzung der Anwendung verursacht hat, auch hier gab der Großteil die positivste Antwort. Leichte Probleme zeigten sich bei der Erkennung der möglichen Interaktionen. Trotz der einblendbaren Interaktionsicons auf den manipulierbaren Objekten gaben immerhin 25% der Probanden an, der Aussage „Ich habe Interaktionen leicht erkannt“ nur *mittelmäßig* zustimmen zu können. Ein Grund hierfür könnte sein, dass die Objekte stets nur auf einer vordefinierten Geraden verschoben werden konnten, wobei die Richtung der erlaubten Bewegung für einige Teilnehmer nicht direkt ersichtlich war und erst durch Probieren herausgefunden werden musste. Zusätzlich versuchten drei Probanden für eine Manipulation nur die Icons selbst zu berühren und nicht die komplette Geometrie des beweglichen Objekts. Keinmal wurden die positiven Aussagen über die Usability der Anwendung komplett abgelehnt.

	stimmt nicht	stimmt wenig	stimmt mittelmäßig	stimmt ziemlich	stimmt sehr
Es fiel mir leicht, ein Modell zu laden	0	0	0	5	15
Das Tracking funktionierte zuverlässig	0	0	0	12	8
Ich habe Interaktionen leicht erkannt	0	1	5	12	2
Es fiel mir leicht, Objekte zu manipulieren	0	1	3	11	5
Es hat mir Spaß gemacht, die App zu benutzen	0	0	1	8	11

Abbildung 5.6: Die allgemeinen Usabilityergebnisse. Die Zahlen entsprechen der Anzahl der Probanden.

Zusätzlich zu den allgemeinen Fragen wurde zu jedem Anwendungsfall ermittelt, wie leicht den Probanden die gestellte Aufgabe fiel. Für die Auswertung wurden alle Antworten der Unterszenarien der Beleuchtung zusammengezählt um eine Aussage über den Anwendungsfall an sich treffen zu können. Der Anwendungsfall *Sonneneinstrahlung* erwies sich als der schwierigste. 30% der Antworten stimmten der Aussage, dass die Aufgabe leicht fiel nur *mittelmäßig* zu, 38,3% *ziemlich* und der drittgrößte Teil von nur 21,7% *sehr*. Die anderen beiden Szenarien waren jedoch von den Nutzern sehr gut lösbar. Bei 88,3% der Antworten zum *Grundriss* wurde die „Bestnote“

vergeben, bei der *Materialität* waren es 83,3%. Die drei ablehnensten Antwortmöglichkeiten wurden bei beiden Fällen keinmal gewählt. Insgesamt lässt sich also feststellen, dass die gestellten Arten von Aufgaben mithilfe des Prototyps gut lösbar waren.

5.3.3 Hypothesen

Um die aufgestellten Hypothesen zu bestätigen oder zu widerlegen wurde der Hauptteil der Nutzerbefragung ausgewertet. Jede der Hypothesen steht für einen Anwendungsfall, weshalb für die Überprüfung die Ergebnisse des entsprechenden Szenarios herangezogen werden:

1. ***Anwendungsfall Sonneneinstrahlung: Geht es um die Position des Schattenwurfs werden klare scharfkantige dynamische Schatten präferiert.***

   Um diese Aussage zu überprüfen wurde zunächst der Einfluss der statischen Schatten auf die Lösbarkeit der Aufgabe als Ganzes untersucht. Dazu wurden die Ergebnisse der Version mit rein statischen Schatten mit ihrer Bewertung im kombinierten Szenario zusammengezählt, die Anzahl der Antworten beträgt so jeweils 40 pro Frage. Daraus ergibt sie die Wirkung auf den kompletten Anwendungsfall. Abbildung 5.7 zeigt die gegebenen Antworten der Probanden:

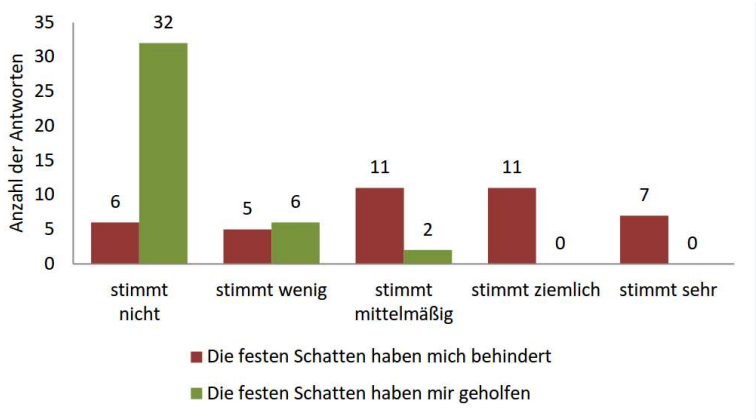


Abbildung 5.7: Die Einschätzungen der statischen Schatten im Anwendungsfall *Sonneneinstrahlung*.

Wie man sieht, wurde in 80% der Antworten angegeben, dass die festen Schatten bei der Lösung der Aufgabe nicht geholfen hätten. Die Antworten auf die Frage, ob die Schatten bei der Lösung gestört hätten sind nicht so eindeutig, sie sind relativ breit verteilt. Jeweils elfmal wurde hier *stimmt mittelmäßig* und *stimmt ziemlich* gewählt. Ganz anders ist das Bild bei den dynamischen Schatten:

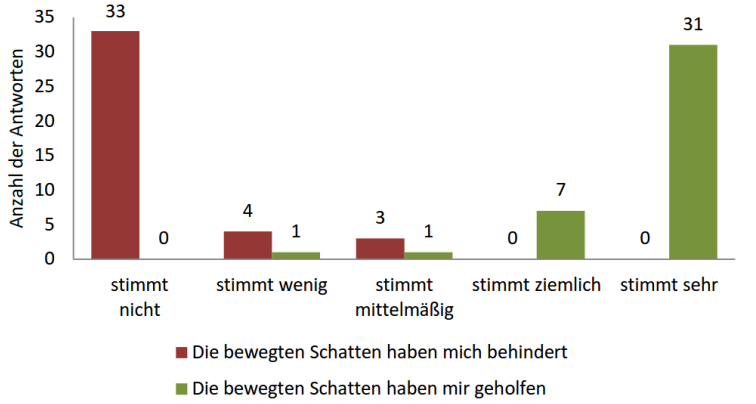


Abbildung 5.8: Die Einschätzungen der dynamischen Schatten im Anwendungsfall *Sonneneinstrahlung*.

77,5% der Antworten stimmen uneingeschränkt zu, dass die dynamischen Schatten der Lösung des Anwendungsfalls zuträglich sind. 33 von 40 mal wurde die Aussage, dass die Schatten eine Behinderung wären stark abgelehnt. Ebenso eindeutig zeigt sich die Wahl des Favoriten dieses Szenarios, dargestellt in Abbildung 5.9. Ganze 90% wählten die Version mit der rein dynamischen Beleuchtung als ihren Favoriten. **Die Hypothese ist somit wahr**, für diesen speziellen Anwendungsfall eigneten sich die dynamischen Schatten am besten.

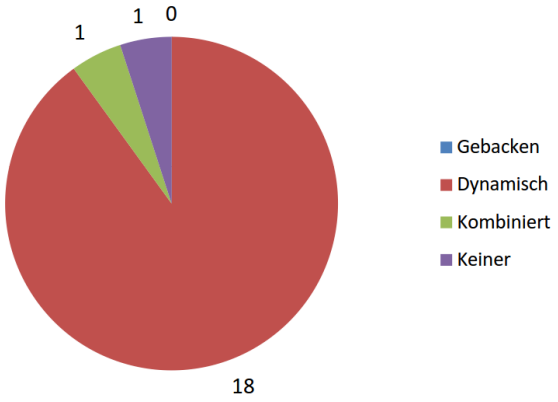


Abbildung 5.9: Die Favoritenwahl des Anwendungsfalls *Sonneneinstrahlung*.

2. ***Anwendungsfall Grundriss: Bei der Wahrnehmung eines Grundrisses spielt die Schattierung keine wichtige Rolle.***

   Betrachtet man die Antworten der Studienteilnehmer zu diesem Anwendungsfall wird schnell ein Muster sichtbar. Abbildung 5.10 zeigt dies für alle Schattenarten dieses Szenarios in der Übersicht. Wieder wurden die Antworten der einzelnen Ausarbeitungen des Modells kombiniert, um Aussagen über die Gesamtwirkung einer Beleuchtung tätigen zu können. Auf den ersten Blick wird ersichtlich, dass sich der Großteil der Antworten bei *stimmt nicht* findet. Die Behauptungen, dass beiderlei Schattenarten den Probanden bei der Lösung dieses Anwendungsfalls behindern oder unterstützen würden, wurden von der Mehrheit klar abgelehnt. **Daraus folgt, dass die Hypothese wahr ist**. Egal welche Schatten eingesetzt wurden, sie hatten keinen großen Einfluss auf die Wahrnehmung des Labyrinths und die Findung eines Wegs hindurch.

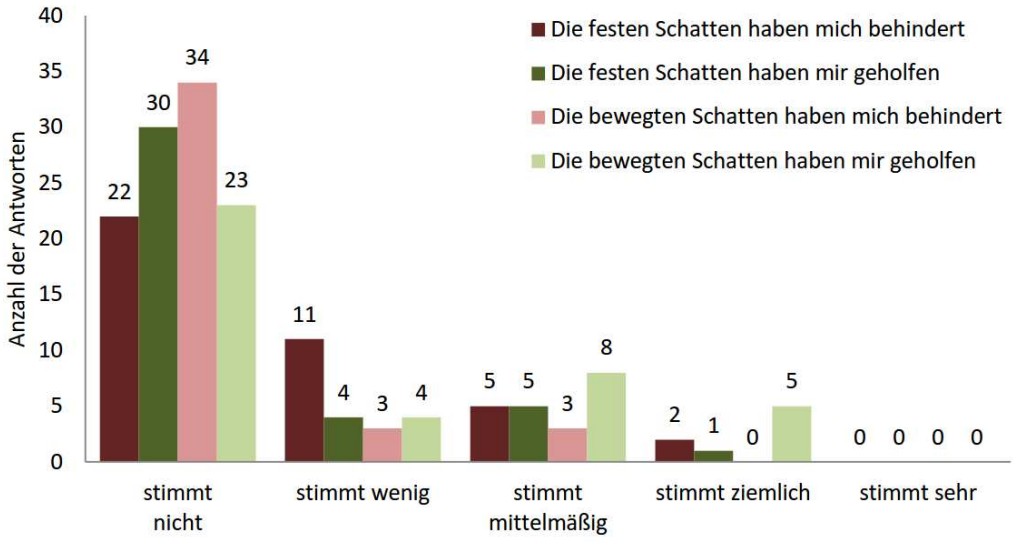


Abbildung 5.10: Die Antworten zum Einfluss der Schatten des Anwendungsfalls *Grundriss*.

Der geringe Unterschied in der Wahrnehmung der Studienteilnehmer wird von der Wahl des Favoriten bestätigt, die Ergebnisse sind in 5.11 dargestellt. 35% konnten sich nicht für einen Favoriten entscheiden. Die restlichen Nutzer entschieden sich relativ gleichmäßig verteilt zwischen dem dynamischen (35%) und dem kombinierten (25%) Ansatz. Nur einer präferierte die rein gebackene Beleuchtung mit der Begründung, dass die Schatten besser aussähen.

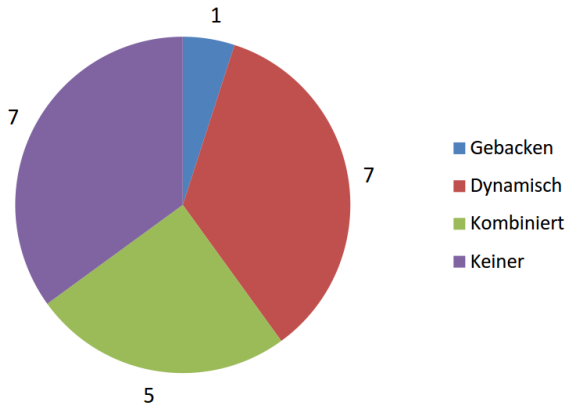


Abbildung 5.11: Die Favoritenwahl des Anwendungsfalls *Grundriss*.

3. ***Anwendungsfall Materialität: Bei der Beurteilung von Materialität bevorzugen Nutzer die hochqualitativen statischen Schatten.***

   Werden die Antworten auf die Fragen dieses Szenarios nach dem selben Verfahren wie bei vorheriger Hypothese ausgewertet, fallen sofort die Ähnlichkeiten auf. Abbildung 5.12 stellt diese dar. Wieder war der Einfluss der Schatten insgesamt sehr gering. Besonders auffällig ist die sehr hohe Ablehnung der Aussage „Die festen Schatten haben mich behindert“ von 95%. Die selbe Feststellung wird für die dynamische Beleuchtung mit 87,5% abgelehnt. Zwar fand auch der Großteil beide Schattenarten nicht hilfreich, insgesamt werden sie jedoch stärker für hilfreich erachtet als störend. Die Werte sind dabei für beide Ansätze sehr ausgeglichen, es gibt keine nennenswerten Unterschiede. Anhand des Einflusses auf die Problemlösung lässt sich also keine eindeutige Aussage für oder gegen bestimmte Schattentypen treffen, die Schatten werden nur allgemein für hilfreicher empfunden als behindernd.

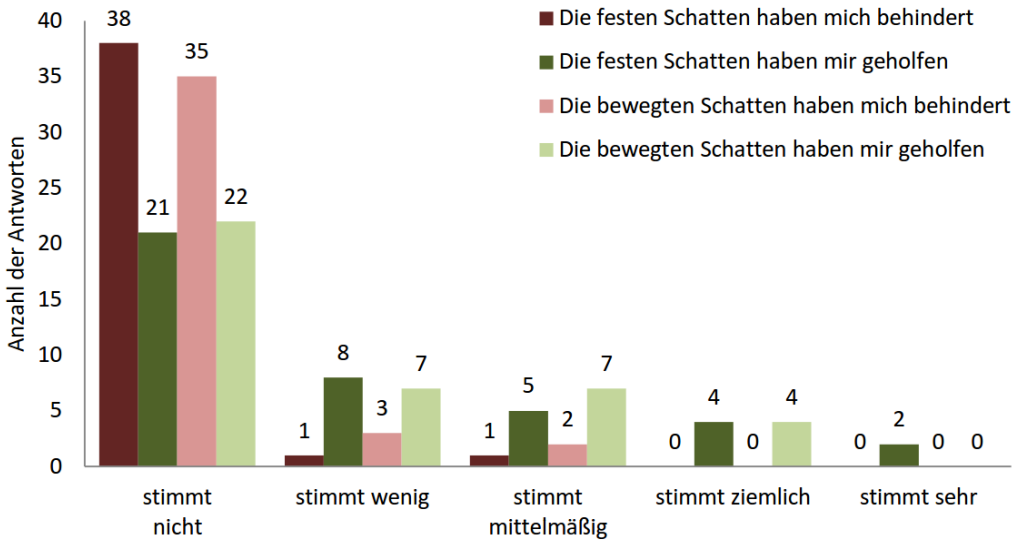


Abbildung 5.12: Die Antworten zum Einfluss der Schatten des Anwendungsfalls *Materialität*.

Etwas klarer werden die Präferenzen jedoch beim Blick auf die Favoritenverteilung (siehe Abbildung 5.13). Der Großteil der Probanden von immerhin 45% zieht die Ausarbeitung mit den gebackenen Schatten vor, einige Male wurden Aussagen getätigt wie „[...] weil es am schönsten aussah“ oder „das war am realistischsten“. Ein relativ hoher Anteil von 30% hatte keine Präferenz, nur einmal wurde die dynamische Version gewählt. **Die Hypothese lässt sich also nur teilweise bestätigen**. Auf die Beurteilung der Materialität direkt hatte die Schattenart keinen sehr starken Einfluss, wegen der Optik wurde sie insgesamt trotzdem präferiert.

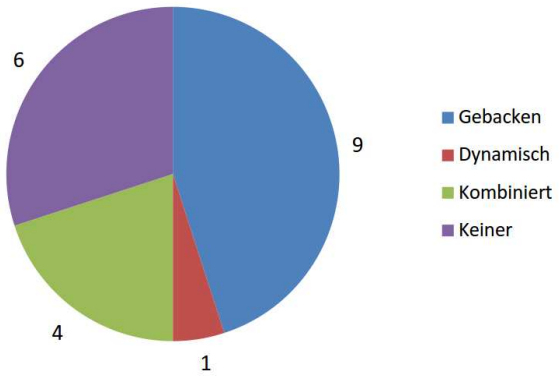


Abbildung 5.13: Die Favoritenwahl des Anwendungsfalls *Materialität*.

5.3.4 Datenlogging

Damit die im Laufe der Studie im Hintergrund gesammelten quantitativen Daten korrekt ausgewertet werden konnten, mussten die Logdateien zunächst ebenfalls in eine *Microsoft Excel*-Tabelle übertragen werden. Anschließend mussten die Daten gesäubert werden. Einträge, bei denen ein nicht der Studie zugehöriges Modell geladen wurde mussten entfernt werden, ebenso wie die Fälle, in denen ein Modell zu kurz geladen war, da ein falsches Modell ausgewählt wurde oder der Prototyp sich selbst beendet hatte. Zuletzt musste noch eine Fallunterscheidung zwischen den einzelnen Anwendungsfällen und ihren drei Unterkategorien durchgeführt werden. So ergaben sich die in Abbildung 5.14 dargestellten Werte.

Anwendungsfall	Sonneneinstrahlung			Grundriss			Materialität		
Ausführung	statisch	dynamisch	kombiniert	statisch	dynamisch	kombiniert	statisch	dynamisch	kombiniert
Anzahl der Touches (Ø)	14,6	28,2	20,7	13,3	13,3	14,9	13,4	23,4	23,6
Zeit zum Lösen (in Minuten)	0:59	1:40	1:24	0:32	0:36	0:47	0:41	0:48	0:52

Abbildung 5.14: Die Ergebnisse des Datenloggings.

Betrachtet man den Anwendungsfall *Sonneneinstrahlung* mit den Informationen aus der Auswertung des Fragebogens im Hinterkopf, fällt auf, dass bei der statischen Version die Zahl der Touches sowie die Bearbeitungszeit am geringsten war, obwohl sie den Probanden am schwierigsten fiel. Die größten Werte findet man bei der dynamischen Beleuchtung, obwohl diese die bevorzugte und am besten angenommene war. Diese Gegebenheit lässt sich wahrscheinlich dadurch erklären, dass die Nutzer an den dynamischen Schatten am meisten Freude hatten, sich deshalb länger mit dem Szenario beschäftigten und versuchten, die Aufgabe so gut wie möglich zu lösen. Denn die hier gemessene Zeit kam nicht deshalb zustande, dass die Aufgabe mit den dynamischen Schatten schwer zu lösen gewesen sei – eher ist das Gegenteil der Fall.

Die für den Anwendungsfall *Grundriss* bereits durch den Fragebogen ermittelte Ausgeglichenheit der Beleuchtungsmethoden zeigt sich auch hier, die aufgestellte Hypothese wird durch die Logdateien erneut bestätigt. Die durchschnittliche Anzahl an Touches ist extrem gleichmäßig verteilt. Auch bei der Bearbeitungszeit zeigt sich bei den beiden Extremen, statische und dynamische Schatten, kein nennenswerter Unterschied.

Bei der *Materialität* zeigt sich die bei der Auswertung schon festgestellte Ausgeglichenheit bei der sehr ähnlichen Bearbeitungszeit. Die wenigsten Touches wurden bei der statischen Beleuchtung benötigt, welche auch die Präferenz der Probanden darstellt. Die anderen beiden Ansätze, die dynamischen und die kombinierten Schatten, liegen mit 23,4 und 23,6 Berührungen im Mittel extrem nah beieinander.

Insgesamt stützen die quantitativen Logdaten die durch Nutzerbefragung gewonnenen Erkenntnisse und Aussagen.

6 Fazit

6.1 Zusammenfassung

In diesem Teil werden die Erkenntnisse, die im Laufe der Arbeit gewonnen wurden, kurz und prägnant zusammengefasst. Der Abschnitt teilt sich in die Erfahrungen mit dem Prototyp an sich, wie erwähnenswerte Probleme oder Auffassung durch die Nutzer und in die Ergebnisse der durchgeführten Studie in Bezug auf die Wahl der optimalen Beleuchtungsart für die getesteten Anwendungsfälle.

6.1.1 Prototyp

Die Umsetzung des Prototyps zeigte sich als zeitaufwendige und komplexe, jedoch belohnende und lehrreiche Aufgabe. Im Laufe der Implementierung und Auswertung sind einige Dinge aufgefallen und im Gedächtnis geblieben.

Zum einen ist es heutzutage kein Hexenwerk mehr, selbstständig Augmented Reality Anwendungen zu erstellen. Durch die leichte Zugänglichkeit von Entwicklungsumgebungen, wie *Unity*, die direkt den Export als mobile Anwendung erlauben, ist es relativ einfach geworden, Apps für verschiedenste Plattformen zu erstellen.

Das große Angebot an AR-Frameworks hält für fast jede gewünschte Programmierumgebung APIs oder Plugins bereit. Das AR-Tracking mit *Vuforia* funktioniert erstaunlich zuverlässig und bietet eine große Auswahl an Trackingverfahren und Anpassungsmöglichkeiten. Bei der Zugänglichkeit und Robustheit hat sich hier allgemein in den letzten Jahren sehr viel getan.

Ähnlich verhält es sich mit der Touchgestenbedienung von Anwendungen. Es ist nicht mehr nötig, die grundlegenden Funktionen „from scratch“ selbst zu implementieren. Touchlibraries wie *TouchScript* übernehmen das für einen und bieten eine gute Grundlage, um auf ihnen weiter aufzubauen und sie seinen eigenen Bedürfnissen und Ansprüchen anzupassen.

Wie sich gezeigt hat, ist es aber nicht immer klug, sich vollends auf fertige Pakete zu verlassen. *Unitys UNet*-Netzwerkfunktionen sind im aktuellen Zustand ein gutes Beispiel dafür. Zwar können auch hier die grundlegendsten Dinge von bereitgestellten Skripten übernommen werden, sobald die Anforderungen aber komplexer werden, sollte man möglichst schnell andere Wege finden, wie die selbständige Umsetzung mit den Netzwerkfunktionen der gewählten Programmiersprache. So lässt sich in manchen Fällen sicherlich viel Zeit und Ärger ersparen.

Ein zudem positiver Aspekt war auch die sehr gute Annahme des Prototyps durch die Nutzer der Studie. Die allgemeine Usabilityumfrage zeigte klar, dass der Großteil sehr viel Spaß bei der Benutzung hatte. Das Thema Augmented Reality weckt eine Faszination, viele sind von den angezeigten digitalen Modellen in der realen Welt begeistert und beschäftigen sich gerne mit ihnen. Einige Probanden hatten sogar schon sehr große Freude daran, sich die Modelle lediglich von allen Seiten anzuschauen oder einzelne Objekte ohne bestimmten Sinn dahinter hin- und herzubewegen.

Durch die Nutzerstudie zeigte sich schließlich auch, dass der Prototyp gut geeignet ist, um die gestellten Aufgaben zu lösen. Er kann problemlos im architektonischen Bereich verwendet werden, um zum Beispiel Planungsaufgaben zu übernehmen, wie das Anpassen eines Grundrisses oder das Auswählen von bestimmten Materialien und Farben. Dem Großteil der Probanden fielen die gestellten Aufgaben leicht und sie konnten sich für die Anwendung begeistern. Die dynamische Ladefunktion und Austauschbarkeit der Modelle macht sie bereits im Prototypenstadium zu einem nützlichen Werkzeug bei der AR-gestützten Planung und Präsentation.

6.1.2 Beleuchtungsarten

Zusätzlich wurde in dieser Arbeit der entwickelte Prototyp dazu genutzt, Erkenntnisse über die Verwendung verschiedener Beleuchtungsansätze in der Augmented Reality zu gewinnen. Durch die Durchführung und Auswertung der Nutzerstudie ist dies gelungen. Drei Anwendungsfälle für

eine AR-App wurden entwickelt, mit je drei Schattentypen umgesetzt und schließlich evaluiert. Dabei konnten folgende Erkenntnisse gewonnen werden, welche künftig als Leitlinien für die Wahl einer Beleuchtungsart bei entsprechenden Aufgaben genutzt werden können:

1. **Geht es um die Position des Schattenwurfs werden klare scharfkantige dynamische Schatten präferiert**: Diese Aussage konnte bestätigt werden.
2. **Bei der Wahrnehmung eines Grundrisses spielt die Schattierung keine wichtige Rolle**: Auch diese Aussage spiegelte sich in den Ergebnissen der Studie wider. Es kann also eine Überlegung wert sein, in so einem Szenario sogar gänzlich auf Schatten zu verzichten, um Vorarbeit und Rechenkapazität einzusparen.
3. **Bei der Beurteilung von Materialität bevorzugen Nutzer die hochqualitativen statischen Schatten**: Diese Hypothese konnte teilweise bestätigt werden. Zwar halfen die statischen Schatten nicht direkt bei der Beurteilung des Materials, sie wurden aufgrund der Optik jedoch präferiert.

6.2 Verbesserungen und Ausblick

Natürlich zeigten sich auch Punkte, die zukünftig Ansätze für Verbesserungen bieten. Die Anwendung ist ein Prototyp und deshalb nicht vollends optimiert. So stürzte er ungefähr einmal pro Studie ab und das ohne jegliche Fehlermeldungen im Logsystem. Eventuell läuft bei der Benutzung der Arbeitsspeicher des Endgeräts voll und das Betriebssystem beendet die Anwendung von außen. Auch der Akkuverbrauch des Prototyps ist noch relativ hoch, Performanzprobleme wie Ruckeln traten jedoch nicht auf. Die meiste Zeit lief das System durchaus zuverlässig und flüssig.

Abseits von den erwähnten Fehlerbehebungen sind auch eine Reihe von Funktionserweiterungen denkbar. So wäre der Direktimport von FBX-Modell-Dateien sinnvoll, denn so muss der Ersteller der Modelle nicht den Weg über *Unity* gehen, wenngleich dieser Weg auch Vorteile bei der Feinanpassung bietet. Um nur schnell ein Modell anzuzeigen, wäre die erweiterte Funktion nützlich. Umsetzbar wäre dies zum Beispiel über die Nutzung eines kostenpflichtigen *Unity*-Plugins.

Statt die 3D-Modelle nur lokal auszulesen wäre zusätzlich eine Anbindung an das Internet oder eine Cloud denkbar. User könnten ihre eigenen Modelle dort hochladen, bewerten und sich die Kreationen anderer Nutzer in AR ansehen.

Auch die nachträgliche Austauschbarkeit des Trackers ist aufgrund der von *Vuforia* vorgeschriebenen Vorgehensweise über die Entwicklerwebsite zur Registrierung der Tracker zum jetzigen Stand nicht gegeben, wäre jedoch ein angenehmes Feature.

Aus Usabilitysicht hat sich gezeigt, dass einige der Probanden trotz der einblendbaren Interaktionsicons Probleme damit hatten, erlaubte Interaktionen zu erkennen. Dieser Punkt könnte mit einer Anzeige der erlaubten Bewegungsbereiche durch Grafiken oder der Objekte als „Ghosts“, also halb durchsichtiger Kopien, verbessert werden. Um die manipulierbare Geometrie kenntlich zu machen würden sich Konturlinien oder ein Glüheffekt um dieselben eignen.

Für die weitere Zukunft bieten sich auch Umsetzungen für neue Endgeräte an. Sobald AR-Brillen oder AR-spezifische Tablets tatsächlich marktreif sind und breite Akzeptanz finden, ist bei der Erfahrung der virtuellen Welt auch mit dieser Anwendung sicherlich ein Mehrwert gegeben.

Anhang

Python-Skript zur Automatisierung des Backvorgangs in *Cinema 4D*

```
import c4d
from c4d import gui
# Runs through Octane Object Tags and increments Bake ID.
# copies octane baking camera for each frame
# to camswitcher, changes its Baking Group ID and cycles according.

# REQUIREMENTS (see .c4d file for example):
# - Octane Camera with baking enabled
# - Object(s) to bake with Octane Object Tag
# - camswitcher object with xpresso from .c4d file
# - Render save path with 'Current Camera' token ($camera)

class ObjectIterator :
  #class src: http://cgrebel.com/2015/03/c4d-python-scene-iterator/
  def __init__(self, baseObject):
    self.baseObject = baseObject
    self.currentObject = baseObject
    self.objectStack = []
    self.depth = 0
    self.nextDepth = 0

  def __iter__(self):
    return self

  def next(self):
    if self.currentObject == None :
      raise StopIteration

    obj = self.currentObject
    self.depth = self.nextDepth

    child = self.currentObject.GetDown()
    if child :
      self.nextDepth = self.depth + 1
      self.objectStack.append(self.currentObject.GetNext())
      self.currentObject = child
    else :
      self.currentObject = self.currentObject.GetNext()
      while( self.currentObject == None and len(self.objectStack) > 0 ) :
        self.currentObject = self.objectStack.pop()
        self.nextDepth = self.nextDepth - 1
    return obj

class TagIterator:
  #class src: http://cgrebel.com/2015/03/c4d-python-scene-iterator/
  def __init__(self, obj):
    currentTag = None
    if obj :
      self.currentTag = obj.GetFirstTag()

  def __iter__(self):
    return self

  def next(self):
    tag = self.currentTag
    if tag == None :
      raise StopIteration

    self.currentTag = tag.GetNext()
```

```
60      return tag
61
62  def main():
63
64    doc.StartUndo()
65
66    obj = doc.GetFirstObject()
67    scene = ObjectIterator(obj)
68
69    bakeCam = None
70    camSwitcher = doc.SearchObject("camswitcher")
71    if camSwitcher != None:
72      #delete old cams
73      for child in camSwitcher.GetChildren():
74        doc.AddUndo(c4d.UNDOTYPE_DELETE, child)
75        child.Remove()
76
77    bakeobjects = []
78
79    counter = 2
80    bakingcamcounter = 0
81
82    for obj in scene:
83
84      tags = TagIterator(obj)
85      for tag in tags:
86        if tag.GetType() == 1029603: #octaneobject tag
87          doc.AddUndo(c4d.UNDOTYPE_CHANGE, tag)
88          tag[c4d.OBJECTTAG_BAKEID] = counter
89          bakeobjects.append("ID_"+str(counter).zfill(2)  + "_" + obj.GetName())
90          print obj.GetName(), "| Bake ID:", counter
91          counter += 1
92        elif tag.GetType() == 1029524 and tag[c4d.OCT_CAMERA_TYPE_SELECT] == 2: #
      OctaneCam and baking
93          if bakingcamcounter == 0:
94            bakeCam = obj
95          bakingcamcounter +=1
96
97    if bakingcamcounter > 1:
98      gui.MessageDialog("Found " + str(bakingcamcounter) + " baking cameras.
      Using the first one.")
99
100   fps = doc[ c4d.DOCUMENT_FPS ]
101   doc.SetMinTime(c4d.BaseTime(2,fps))
102   doc.SetMaxTime(c4d.BaseTime(counter - 1,fps))
103   doc.SetLoopMinTime(c4d.BaseTime(2,fps))
104   doc.SetLoopMaxTime(c4d.BaseTime(counter - 1,fps))
105
106   if bakeCam != None and camSwitcher != None:
107
108     #reverse list
109     bakeobjects.reverse()
110     index = len(bakeobjects) + 1
111
112     for bakeobj in bakeobjects:
113       newcam = bakeCam.GetClone()
114       newcam.SetName(bakeobj)
115       newcam.InsertUnder(camSwitcher)
116
117       #change baking ID
118       tags = TagIterator(newcam)
119       for tag in tags:
```

```
120             if tag.GetType() == 1029524: #OctaneCam
121                 tag[c4d.OCT_BAKECAMERA_GROUP_ID] = index
122         index -= 1
123 
124         doc.AddUndo(c4d.UNDOTYPE_NEW, newcam)
125 
126     doc.EndUndo()
127     c4d.EventAdd()
128 
129 
130 if __name__=='__main__':
131     main()
```

Abbildungsverzeichnis

Inhalt der beigelegten CD

- Elektronische Version der Arbeit im LaTeX-Originalformat und als PDF
- Quellcode der Anwendung mit *Unity*-Projekt.
- Prototyp als Android-APK-Datei
- Entwicklungs- und Studien-*AssetBundle*
- Benutzter Tracker zum Ausdrucken
- Excel-Dokumente der gesammelten Daten aus der Nutzerstudie
- Zitierte Quellen in elektronischer Form
- Präsentationsfolien der Antrittsrede
- Inhaltsverzeichnis und Hilfe zur CD

Literatur

[1] Aila, Timo and Samuli Laine: *Alias-free shadow maps.* Rendering techniques, 2004:15th, 2004.

[2] Augment. http://www.augment.com/. Aufgerufen am 22.05.2016.

[3] Autodesk: *Fbx*. http://www.autodesk.com/products/fbx/overview. Aufgerufen am 02.07.2016.

[4] Azuma, Ronald T, Bruce R Hoff, Howard E Neely III, Ronald Sarfaty, Michael J Daily, Gary Bishop, Vern Chi, Greg Welch, Ulrich Neumann, Suya You, *et al.*: *Making augmented reality work outdoors requires hybrid tracking*. In *Proceedings of the First International Workshop on Augmented Reality*, volume 1. Citeseer, 1998.

[5] Borg, Mathias, Martin M Paprocki, and Claus B Madsen: *Perceptual evaluation of photorealism in real-time 3d augmented reality*. In *Computer Graphics Theory and Applications (GRAPP), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[6] Calian, Dan A, Kenny Mitchell, Derek Nowrouzezahrai, and Jan Kautz: *The shading probe: fast appearance acquisition for mobile ar*. In *SIGGRAPH Asia 2013 Technical Briefs*, page 20. ACM, 2013.

[7] Catmull, Edwin: *A subdivision algorithm for computer display of curved surfaces*. Technical report, DTIC Document, 1974.

[8] Chan, Eric and Frédo Durand: *An efficient hybrid shadow rendering algorithm.* Rendering Techniques, 2004:15th, 2004.

[9] Crow, Franklin C: *Shadow algorithms for computer graphics*. In *Acm siggraph computer graphics*, volume 11, pages 242–248. ACM, 1977.

[10] Debevec, Paul: *Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography*. In *ACM SIGGRAPH 2008 classes*, page 32. ACM, 2008.

[11] Donath, Andreas: *Apple kauft Münchner Augmented-Reality-Firma*. http://www.golem.de/news/metaio-apple-kauft-muenchner-augmented-reality-firma-1505-114330.html. Aufgerufen am 22.05.2016.

[12] Dorfmuller-Ulhaas, Klaus and Dieter Schmalstieg: *Finger tracking for interaction in augmented environments*. In *Augmented Reality, 2001. Proceedings. IEEE and ACM International Symposium on*, pages 55–64. IEEE, 2001.

[13] Google: *Glass*. https://www.google.com/glass/start/. Aufgerufen am 05.06.2016.

[14] Google: *Project Tango*. https://www.google.com/atap/project-tango/. Aufgerufen am 22.05.2016.

[15] Gruber, Lukas, Tobias Langlotz, Pintu Sen, Tobias Hoherer, and Dieter Schmalstieg: *Efficient and robust radiance transfer for probeless photorealistic augmented reality*. In *Virtual Reality (VR), 2014 iEEE*, pages 15–20. IEEE, 2014.

[16] Heidmann, Tim: *Real shadows, real time*. Iris Universe, 18:28–31, 1991.

[17] IDC: *Smartphone os market share, 2015 q2*. http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Aufgerufen am 05.06.2016.

[18] Isard, Michael and Andrew Blake: *Contour tracking by stochastic propagation of conditional density*. In *Computer Vision-ECCV'96*, pages 343–356. Springer, 1996.

[19] Ishii, Hiroshi and Brygg Ullmer: *Tangible bits: towards seamless interfaces between people, bits and atoms*. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 234–241. ACM, 1997.

[20] Jacobs, Katrien and Céline Loscos: *Classification of illumination methods for mixed reality*. In *Computer Graphics Forum*, volume 25, pages 29–51. Wiley Online Library, 2006.

[21] Kähäri, Markus and David J Murphy: *Mara: Sensor based augmented reality system for mobile imaging device*. In *5th IEEE and ACM International Symposium on Mixed and Augmented Reality*, volume 13, 2006.

[22] Kan, Paul and Hannes Kaufmann: *Differential irradiance caching for fast high-quality light transport between virtual and real worlds*. In *Mixed and Augmented Reality (ISMAR), 2013 IEEE International Symposium on*, pages 133–141. IEEE, 2013.

[23] Kasahara, Shunichi, Valentin Heun, Austin S Lee, and Hiroshi Ishii: *Second surface: multi-user spatial collaboration system based on augmented reality*. In *SIGGRAPH Asia 2012 Emerging Technologies*, page 20. ACM, 2012.

[24] Kato, Hirokazu and Mark Billinghurst: *Marker tracking and hmd calibration for a video-based augmented reality conferencing system*. In *Augmented Reality, 1999.(IWAR'99) Proceedings. 2nd IEEE and ACM International Workshop on*, pages 85–94. IEEE, 1999.

[25] Lang, Peter, Alben Kusej, Axel Pinz, and Georg Brasseur: *Inertial tracking for mobile augmented reality*. In *Instrumentation and Measurement Technology Conference, 2002. IMTC/2002. Proceedings of the 19th IEEE*, volume 2, pages 1583–1587. IEEE, 2002.

[26] Layar. `https://www.layar.com/about/`. Aufgerufen am 22.05.2016.

[27] Lowe, David G.: *Fitting parameterized three-dimensional models to images*. IEEE Transactions on Pattern Analysis & Machine Intelligence, (5):441–450, 1991.

[28] Malik, Shahzad, Chris McDonald, and Gerhard Roth: *Hand tracking for interactive pattern-based augmented reality*. 2002.

[29] Microsoft. `https://www.microsoft.com/microsoft-hololens/en-us/why-hololens`. Aufgerufen am 14.07.2016.

[30] Microsoft: *Hololens*. `https://www.microsoft.com/microsoft-hololens/en-us`. Aufgerufen am 05.06.2016.

[31] Milgram, Paul and Fumio Kishino: *A taxonomy of mixed reality visual displays*. IEICE TRANSACTIONS on Information and Systems, 77(12):1321–1329, 1994.

[32] Mings, Josh: *Excuse me, scrawl. your 3d drawings are crowding my reality*. `http://www.solidsmack.com/3d-cad-technology/excuse-me-scrawl-your-3d-drawings-are-crowding-my-reality/`. Aufgerufen am 29.05.2016.

[33] Motion, Leap. `https://www.leapmotion.com/`. Aufgerufen am 14.07.2016.

[34] Newman, Joseph, Martin Wagner, Martin Bauer, Asa MacWilliams, Thomas Pintaric, Dagmar Beyer, Daniel Pustka, Franz Strasser, Dieter Schmalstieg, and Gudrun Klinker: *Ubiquitous tracking for augmented reality*. In *Mixed and Augmented Reality, 2004. ISMAR 2004. Third IEEE and ACM International Symposium on*, pages 192–201. IEEE, 2004.

[35] niantic. https://www.nianticlabs.com/img/posts/Encounter0.png. Aufgerufen am 14.07.2016.

[36] Norman, Donald A: *The design of everyday things: Revised and expanded edition*. Basic books, 2013.

[37] Nowrouzezahrai, Derek, Stefan Geiger, Kenny Mitchell, Robert Sumner, Wojciech Jarosz, and Markus Gross: *Light factorization for mixed-frequency shadows in augmented reality*. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 173–179. IEEE, 2011.

[38] Park, Jun, Suya You, and Ulrich Neumann: *Natural feature tracking for extendible robust augmented realities*. In *Proc. Int. Workshop on Augmented Reality*, 1998.

[39] Patow, Gustavo and Xavier Pueyo: *A survey of inverse rendering problems*. In *Computer graphics forum*, volume 22, pages 663–687. Wiley Online Library, 2003.

[40] Perez, Sarah: *Pokémon go tops twitter's daily users, sees more engagement than facebook*. https://techcrunch.com/2016/07/13/pokemon-go-tops-twitters-daily-users-sees-more-engagement-than-facebook. Aufgerufen am 14.07.2016.

[41] Pinz, Axel, Markus Brandner, Harald Ganster, Albert Kusej, Peter Lang, and Miguel Ribo: *Hybrid tracking for augmented reality*. ÖGAI Journal, 21(1):17–24, 2002.

[42] Poupyrev, Ivan, Numada Tomokazu, and Suzanne Weghorst: *Virtual notepad: handwriting in immersive vr*. In *Virtual Reality Annual International Symposium, 1998. Proceedings., IEEE 1998*, pages 126–132. IEEE, 1998.

[43] Pressigout, Muriel and Eric Marchand: *Hybrid tracking algorithms for planar and non-planar structures subject to illumination changes*. In *Proceedings of the 5th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 52–55. IEEE Computer Society, 2006.

[44] PTC: *PTC Adds Augmented Reality Leader Vuforia to Portfolio*. http://www.ptc.com/about/history/vuforia. Aufgerufen am 22.05.2016.

[45] Reeves, William T, David H Salesin, and Robert L Cook: *Rendering antialiased shadows with depth maps*. In *ACM Siggraph Computer Graphics*, volume 21, pages 283–291. ACM, 1987.

[46] Rohmer, Kai, Wolfgang Buschel, Raimund Dachselt, and Thorsten Grosch: *Interactive near-field illumination for photorealistic augmented reality on mobile devices*. In *Mixed and Augmented Reality (ISMAR), 2014 IEEE International Symposium on*, pages 29–38. IEEE, 2014.

[47] Rolland, Jannick P, Larry Davis, and Yohan Baillot: *A survey of tracking technology for virtual environments*. Fundamentals of wearable computers and augmented reality, 1:67–112, 2001.

[48] Schmalstieg, Dieter, Anton Fuhrmann, and Gerd Hesina: *Bridging multiple user interface dimensions with augmented reality*. In *Augmented Reality, 2000.(ISAR 2000). Proceedings. IEEE and ACM International Symposium on*, pages 20–29. IEEE, 2000.

[49] Shi, Jianbo and Carlo Tomasi: *Good features to track*. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

[50] Stiktu. http://blog.stiktu.com/. Aufgerufen am 22.05.2016.

[51] String. http://string.co/. Aufgerufen am 22.05.2016.

[52] TouchScript. http://touchscript.github.io/. Aufgerufen am 01.07.2016.

[53] Tsang, Michael, George W Fitzmaurice, Gordon Kurtenbach, Azam Khan, and Bill Buxton: *Boom chameleon: simultaneous capture of 3d viewpoint, voice and gesture annotations on a spatially-aware display*. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 111–120. ACM, 2002.

[54] Unity3D: *Documentation 4.6 lightmapping in-depth*. http://docs.unity3d.com/460/Documentation/Manual/LightmappingInDepth.html. Aufgerufen am 29.05.2016.

[55] Unity3D: *Documentation shadow overview*. http://docs.unity3d.com/Manual/ShadowOverview.html. Aufgerufen am 29.05.2016.

[56] Unreal-Engine: *Assetbundles*. https://docs.unity3d.com/Manual/AssetBundlesIntro.html. Aufgerufen am 01.07.2016.

[57] Unreal-Engine: *Distance field ambient occlusion*. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/DistanceFieldAmbientOcclusion/index.html. Aufgerufen am 29.05.2016.

[58] Unreal-Engine: *Ray traced distance field soft shadows*. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/RayTracedDistanceFieldShadowing/. Aufgerufen am 29.05.2016.

[59] Unreal-Engine: *Shadow casting*. https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/Shadows/. Aufgerufen am 29.05.2016.

[60] Unreal-Engine: *Unreal engine 4 for unity developers*. https://docs.unrealengine.com/latest/INT/GettingStarted/FromUnity/. Aufgerufen am 05.06.2016.

[61] Van Krevelen, DWF and R Poelman: *A survey of augmented reality technologies, applications and limitations*. International Journal of Virtual Reality, 9(2):1, 2010.

[62] Vuforia. http://www.vuforia.com/. Aufgerufen am 22.05.2016.

[63] Vuforia: *Extended Tracking*. https://developer.vuforia.com/library/articles/Training/Extended-Tracking. Aufgerufen am 22.05.2016.

[64] Vuforia: *Getting Started*. https://developer.vuforia.com/library/getting-started. Aufgerufen am 10.06.2016.

[65] Vuforia: *How To Play the Penguin App* . https://developer.vuforia.com/library/articles/Training/Vuforia-Object-Scanner-Users-Guide. Aufgerufen am 10.06.2016.

[66] Vuforia: *Object Recognition*. https://developer.vuforia.com/library/articles/Training/Object-Recognition. Aufgerufen am 22.05.2016.

[67] Vuforia: *Vuforia Object Scanner* . https://developer.vuforia.com/library/articles/Solution/How-To-Play-the-Penguin-App. Aufgerufen am 10.06.2016.

[68] Wikimedia. https://commons.wikimedia.org/wiki/File:Shadow_volume_illustration.png. Aufgerufen am 29.05.2016.

[69] Wikitude. http://www.wikitude.com/about/. Aufgerufen am 22.05.2016.

[70] Williams, Lance: *Casting curved shadows on curved surfaces*. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.

[71] Woo, Andrew, Pierre Poulin, and Alain Fournier: *A survey of shadow algorithms*. Computer Graphics and Applications, IEEE, 10(6):13–32, 1990.

[72] Zhang, Xiang, Stephan Fronz, and Nassir Navab: *Visual marker detection and decoding in ar systems: A comparative study*. In *Proceedings of the 1st International Symposium on Mixed and Augmented Reality*, page 97. IEEE Computer Society, 2002.

[73] Zhou, Feng, Henry Been Lirn Duh, and Mark Billinghurst: *Trends in augmented reality tracking, interaction and display: A review of ten years of ismar*. In *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, pages 193–202. IEEE Computer Society, 2008.