# MPSE HoloLens - Technischer Bericht

Jonas Bien B.Sc., Tim Bienias B.Sc., Fabian Brenner B.Sc., Ibrahim Cinar B.Sc.,
Ahmed Danyal B.Sc., Hakan Durgel B.Sc., Lukas Hutfleß B.Sc., Kristian Krömmelbein B.Sc.,
Arnold Lockstein B.Sc., Tobias Michalski B.Sc., Dirk Peters B.Sc. und Sven Steininger B.Sc.
*Fachbereich Informatik, Hochschule Darmstadt*
*Haardtring 100, 64295 Darmstadt*

## 1. Einleitung

Das Grafiklabor der Hochschule Darmstadt bietet Masterstudierenden die Möglichkeit beim Projekt Systementwicklung zu partizipieren. Im Wintersemester 17/18 nahm eine Gruppe von 12 Studierenden am Projekt "Bühnenbilder mit der HoloLens" teil. Dieses Projekt fand unter der Schirmherrschaft von Frau Prof. Dr. Elke Hergenröther statt und wurde zusätzlich vom Unternehmen Bosch-Rexroth durch Alexander Kunkel und Tino Inderwieß begleitet. Ziel des Projekts war es ein Programm für die Microsoft HoloLens zu entwickeln mit dem der Benutzer befähigt wird ein Augmented Reality Bühnenbild zu erstellen. Herausforderungen hierbei waren die Einarbeitung in die Technologie der Microsoft HoloLens Plattform und der Unity Game-Engine. Des Weiteren lagen Herausforderungen im Projektmanagement - beispielsweise bei der Anforderungsanalyse und Teamführung. Diese Ausarbeitung dient als Bericht des Projekts und legt die jeweiligen Themengebiete und Aktivitäten aus technischer Sicht dar. Der Bericht ist wie folgt gegliedert: Abschnitt 2 befasst sich mit dem Projektmanagement. Folgend wird in Abschnitt 3 das Assetmanagement betrachtet. Kapitel 4 erörtert das Projekt aus Sicht der Mehrspieler-Umgebung. Absatz 5 beschreibt den Raspberry Pi Asset Server. In Abschnitt 6 gehen die Autoren auf die Motorvisualisierung ein. Kapitel 7 beschäftigt sich mit dem SpectatorView und Kapitel 8 beschreibt das dreidimensionale Bildtracking mit Vuforia. Abschnitt 9 widmet sich der Beschreibung des User Interface und geht zusätzlich auf Texturierung ein. Bevor in Kapitel 11 ein Fazit gezogen wird, geht Absatz 10 auf die im Projekt eingegangenen Aspekte der Qualitätssicherung ein.

## 2. Projektmanagement

Dieser Abschnitt erläutert das Vorgehen beim Projektmanagement und allen verwandten Aktivitäten. Grundsätzlich lässt sich der Projektverlauf in drei Phasen unterteilen. Diese reichen von der Projektinitiierung, die im nächsten Abschnitt behandelt wird, über die Findungsphase bis hin zur Restrukturierung, bei der letztendlich Prozesse definiert wurden, die das Projekt dann getragen haben.

### 2.1. Projektinitiierung

Zu Beginn des Projekts wurde das Thema initial motiviert und der bisherige Entwicklungsstand präsentiert. Anschließend wurden die Product Owner und ein Scrum Master gewählt, welche nun das Projekt managen sollten. Die erste Amtshandlung der Product Owner bestand darin mit Alexander Kunkel und Tino Inderwieß erste Anforderungen und Ziele des Projekts abzuleiten. Hierbei galt es auch den Scope des gesamten Projekts zu erörtern. Aus den gewonnenen Erkenntnissen wurden Backlog Items für den ersten Sprint abgeleitet. Hinsichtlich industrieller Normen, wie z.B. ISO 21500 [1], hätte aus dieser Phase allerdings eine Dokumentation mit ausführlichen Zielvereinbarungen, User Stories und Risiken entstehen sollen. Dieser Mangel führte im weiteren Projektverlauf zu erhöhter Komplexität.

### 2.2. Findungsphase

Nach der Initiierungsphase arbeiteten sich die Projektleiter in die vorgegebenen Projektmanagement-Tools ein und versuchten in Absprache mit den Stakeholdern und Entwicklern eine Richtung für das Projekt zu finden. Durch zeitlichen Druck kam es jedoch immer häufiger zu Kommunikationsfehlern zwischen Product Ownern und Stakeholdern sowie zwischen Product Ownern und Entwicklern. Dies führte nicht nur zu geminderter Effizienz, sondern auch zu fachlich falschen Annahmen bei der Projektplanung. Infolgedessen entwickelte sich durch die genannten Mängel ein suboptimales Arbeitsklima. Durch die oben genannten Diskrepanzen zwischen diesem und einem optimal durchgeführten Projekt ergab sich, dass bis zu diesem Zeitpunkt nur sehr wenige Ziele erfolgreich umgesetzt werden konnten.

Im Anschluss an eine angemessene Projektinitiierung hätten die Product Owner iterativ detaillierte Anforderungen mit den Stakeholdern erarbeiten sollen. Zusätzlich hätten die Product Owner als Team enger zusammenarbeiten und durch ein geschlossenes Auftreten ein motivierendes Vorbild für alle Projektbeteiligten bilden sollen.

### 2.3. Restrukturierung

Nachdem das Projekt hinter die Erwartungen zurückfiel, fand nach einem umfangreichen Review, bei dem das bisherige Vorgehen analysiert wurde, eine Umstrukturierung des

Product Owner Teams statt. Die Projektleitung investierte nun Zeit, um einen sauberen Prozess für das Projektmanagement zu etablieren, d.h. standardisierte Vorgehensweisen für beispielsweise Sprint Planung oder Kommunikation mit Stakeholdern. Des Weiteren wurden alle aufgestellten Ziele nochmals den Stakeholdern vorgelegt und durch diese priorisiert. Der nun eingeführte Prozess wird folgend im Detail der Reihe nach beschrieben (siehe Abbildung 1):

1) Zielplanung - involviert Product Owner und Stakeholder
2) Projektstudie und Aufwandsschätzung durch Product Owner und Entwicklungsteam
3) Zusammenfassung zu Arbeitspaketen durch Product Owner
4) Priorisierung von Paketen durch Product Owner und Stakeholder
5) Planung von Arbeitsablauf durch Product Owner
6) Umsetzung durch Entwicklungsteam
7) Analyse der Ergebnisse durch alle Beteiligten

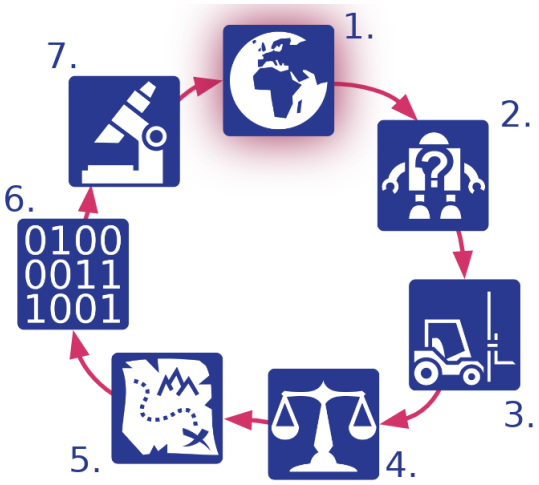



Abbildung 1. Workflow innerhalb des Projekts.

Die neue Struktur basiert im Wesentlichen auf der *PDCA*-Methode [2]. Durch die neuen, genau definierten Prozesse können Risiken schnell und zuverlässig erkannt werden und die Projektplanung ist transparenter und nachvollziehbarer. Folglich sind eine Verbesserung der Arbeitsmoral, einheitliches Auftreten und weniger Missdeutungen eingetreten.

### 2.4. Produktivität

Nach der Einführung des neuen Prozesses stieg die Produktivität und Effizienz des Teams merklich an. Alle zwei Wochen konnte nun eine neue Version der Software mit geplanten Features an die Stakeholder ausgeliefert werden. Durch diese Vorgehensweise konnten die Unwegbarkeiten der Anfangsphase wettgemacht und als lehrreiche Erkenntnisse verbucht werden.

### 2.5. Features

Der folgende Abschnitt gibt einen kurzen Überblick über die implementierten Kern-Features, welche in den nachfolgenden Abschnitten näher beschrieben werden:

- Dynamisches Laden von 3D-Assets zur Laufzeit
- Asset-Server zum Speichern von Assets
- Motorvisualisierung
- Implementiertes UX-Konzept
- Bedienung mit Controller
- Translation, Rotation und Skalierung
- Speichern und Laden von Szenen
- Texturmanagement
- Logging-Komponente

## 3. Assetmanagement

Der Workflow mit der HoloLens sieht vor, dass die Szene, welche später in der Brille dargestellt werden soll, im Unity-Editor eingestellt wird. Nachdem die Szene (jedenfalls im Start-Zustand) modelliert wurde, wird das benötigte Programm kompiliert und auf die HoloLens geladen.

Alle in diesem Schritt genutzten Assets (3D-Objekte, Texturen, Text, usw.) werden mit auf die HoloLens gepackt und in dem internen Massenspeicher abgelegt. Dies umfasst den Modus Operandi für gewöhnliche Anwendungsfälle: ein geschlossenes System, welches sich durch die programmierten Zustände äußert.

Doch was passiert, wenn nach dem Packen und Kompilieren der Szene auffällt, dass weitere Assets hinzugefügt werden sollen? In den meisten Fällen bleibt hier das Rekompilieren und ein erneutes Hochladen auf die Brille nicht aus.

Dieser Umstand ist jedoch nicht akzeptabel, wenn entweder häufige Änderungen an den Assets zu erwarten sind oder während der Laufzeit des HoloLens-Programms neue Assets hinzugefügt werden, die sich in der Szene widerspiegeln sollen.

Da die Unity-Engine jedoch keine native Laufzeitumgebung bereitstellt, in der Assets systematisch und dynamisch nachgeladen werden können, muss ein gewisser Programmieraufwand betrieben werden, um dies zu ermöglichen.

### 3.1. Anforderungen

Die Anforderungen an den Asset-Server[1] waren zum Anfang des Projekt mannigfaltig.

Neben der offensichtlichen Funktionalität, zur Laufzeit Assets[2] anbieten zu können, sollte der Asset-Server auch

- eine Versionierung von Assets bereitstellen, diese in Binär- und Textformat verwalten

1. In diesem Kontext als Modul zu verstehen und nicht als ein Implementierungsdetail. Selbiges folgt in einem späteren Unterkapitel.

2. Entitäten, welche die für das Unity-Projekt benötigten Daten kapseln. Ein Asset kann z.B. ein 3D-Objekt, eine Textdatei, eine Textur oder andere Meta-Informationen enthalten.

- eine REST-Schnittstelle anbieten, um neben der Versionierung auch direkt auf die Assets zugreifen zu können
- die Assets in einem komprimierten Zustand speichern und bei Anfrage entpacken
- PUSH-Benachrichtigungen an die HoloLens senden

Im Laufe des Projekts wurden die Anforderungen jedoch weiter spezifiziert und ein realistisches Ziel gesetzt. Die oben genannten Features wurden fallen gelassen und mit der Hauptfunktionalität ersetzt:

- Anfordern von Assets zur Laufzeit
- ein minimales Cache-System in der HoloLens, um das Laden von Assets zu beschleunigen
- Speichern/Laden von Szenen

Der letzte, aufgezählte Punkt wurde relativ spät im Projekt hinzugefügt und umfasst das Speichern/Laden der Zustandsinformationen von Objekten der aktuellen Szene.

## 3.2. Konzept

**3.2.1. Datenübertragung.** Das dynamische Laden von Assets bedingt eine (kabellose) Netzwerkkommunikation mit der HoloLens. Dies wiederum impliziert eine Client-Server-Kommunikation, die auf verschiedenste Weise implementiert werden kann.

Während sich hierbei eine Socket-Kommunikation anbietet, wie sie in gewöhnlichen Applikationen vorkommt, wäre die Implementierung für die HoloLens durch die verschiedenen Komplexitäts-Ebenen (Unity-Engine ⇔ Unity-Framework ⇔ UniversalWindowsPlatform ⇔ .NET C#), die durch den Workflow entsteht, unnötig kompliziert.

Aus Gründen der Wartbarkeit und zur Vermeidung von Kompatibilitätsproblemen wird eine HTTP-Kommunikation bevorzugt. Dies erzeugt zwar einen minimalen Overhead pro Anfrage, erleichtert jedoch den Umgang mit dem Datenaustausch auf eine standardisierte und menschlich lesbare Weise.

Während nun das verwendete Übertragungs-Protokoll bekannt ist, muss noch geklärt werden, wie das Anwendungsprotokoll auszusehen hat. Während Informationen in Klartext gesendet werden können, bedingt dies immer einen speziellen Mehraufwand, da ein gewisses Parsing vorgenommen werden muss. Um auch hier eine standardisierte und menschlich lesbare Lösung zu gewährleisten, werden sämtliche Daten in der Übertragung mit JSON dargestellt.

Dies hat zudem den weiteren Vorteil, dass JSON-Strings in Unity/C# de-/serialisiert werden können.

**3.2.2. Bereitstellen von Assets.** Der Asset-Server muss dediziert auf einem Rechner laufen und von der HoloLens erreichbar sein. Ist dies gewährleistet, muss der Asset-Server in der Lage sein, alle Assets, die von der HoloLens angefordert werden können, zu indexieren.

Es wird also eine Ordner-Verwaltung benötigt, in dem die Assets in einem bestimmten Format abgelegt werden, damit sie vom Asset-Server gefunden und an die HoloLens gemeldet werden können.

Diese Funktionalität ist wichtig, da die HoloLens keinerlei Informationen darüber hat, welche Assets geladen werden können. Insofern muss die HoloLens eine initiale Anfrage an den Asset-Server schicken und eine Liste der Assets geliefert bekommen können.

**3.2.3. Dateiformate.** Ein Unity-Asset kann verschiedene Arten von Informationen enthalten. Das Format selbiger ist bedingt durch das Originalformat der Daten, die zu einem Asset umgewandelt wurden. Während z.B. das Dateiformat von Bildern nur einige wenige umfasst (PNG, JPG, GIF) kann ein 3D-Modell verschiedenste Formate aufweisen, die z.T. durch das verwendete CAD-Tool vorgegeben werden.

Solange der Unity-Editor das Dateiformat lesen und auswerten kann, ist das Erzeugen eines Assets möglich. Die Unity-Engine unterscheidet später nicht mehr das Dateiformat des 3D-Models, solange es in einem Asset gebündelt ist; dieser oft zeitaufwendige und komplexe Schritt wird für den Entwickler wegabstrahiert.

Dennoch ist es wichtig, eine Vorgabe zu treffen, in welchem Format 3D-Modelle (oder auch Mesh genannt) vorliegen sollen. Dies hat den großen Vorteil, dass bestimmte Features wie Versionierung und Komprimierung besser umgesetzt werden können, wenn ein bestimmtes Dateiformat vorausgesetzt werden kann.

Da sich das FBX-Format für 3D-Models in der Entwicklerszene durchgesetzt hat, wird nunmehr vorausgesetzt, dass 3D-Models als FBX-Datei im ASCII-Format exportiert und im Unity-Editor importiert werden müssen.

## 3.3. Implementierung

Generell teilt sich die Implementierung in zwei große Teile auf: den Server- und den Client-Teil. Während der Asset-Server ein dediziertes Modul ist, wird der Client-Teil als Framework-Detail implementiert und interagiert mit den Modulen der restlichen Projektteilnehmer.

**3.3.1. Assets und Asset-Bundles.** Da schon relativ früh feststand, dass ein HTTP-Server zur Auslieferung von Assets herhalten sollte, musste nun auch geklärt werden, ob dies durch die Unity-Engine unterstützt wird. Eine kurze Suche ergab später, dass die Unity-Engine tatsächlich das Beziehen von Assets über HTTP nativ unterstützt, jedoch mit einem kleinen Haken: es können keine einzelnen Assets, jedoch aber Asset-Bundles bezogen werden.

Ein Asset-Bundle stellt ein proprietäres Archiv dar, welches Assets und andere Meta-Informationen beinhaltet. Die Besonderheit von Asset-Bundles ist zudem die Möglichkeit, Abhängigkeiten zwischen Assets, Renderinformationen, Texturen usw. auf verschiedene Bundles auszulagern, welche automatisch durch die Unity-Engine aufgelöst werden. So können z.B. alle gebräuchlichen Texturen in einem Bundle liegen, während die 3D-Objekte in einem anderen Bundle situiert sind und die Texturen referenzieren.

Während diese Features in zukünftigen Projekten und Anwendungsfällen bestimmt ihren Einsatz finden werden, finden diese keine Anwendung im aktuellen Projekt;

sie würden zu unerwünschten Steigerung der Komplexität der Implementierung führen. Insofern wurde die Design-Entscheidung getroffen, dass per Asset-Bundle immer ein einziges Asset abgelegt wird - mit dem gleichen Namen, um den Zugriff zu erleichtern.

**3.3.2. Asset-Server.** Für das Implementieren eines HTTP-Servers existieren viele verschiedene Softwarelösungen, mit ihren intrinsischen Vor- und Nachteilen. Während es wichtig ist, verschiedene Lösungen zu vergleichen und eine optimale auszuwählen, wurde für dieses Projekt lieber eine prototypische - und vor allem, schnelle - Lösung bevorzugt.

Insofern fiel die Wahl auf einen in Python geschrieben HTTP-Server mit etwas *Intelligenz*, um bestimmte Anfragen vom Client beantworten zu können. Um des Weiteren eine möglichst standardisierte Schnittstelle anbieten zu können, wurde eine REST-ähnliche API implementiert.

Die z.Z. programmierten Endpoints sehen wie folgt aus:

- GET /
  Liefert alle definierten Endpoints jeweils für alle HTTP-Methoden
- GET /Bundles
  Liefert eine Liste aller Asset-Bundles und ihre Thumbnails
- GET /Scene
  Liefert eine Liste aller gespeicherten Szenen
- GET /Bundles/`BundleName`
  Liefert das gewünschte Asset-Bundle
- GET /Scene/`SceneName`
  Liefert die gewünschte Szene
- PUT /Scene/`SceneName`
  Speichert die gewünschte Szene

Die Daten sind stets mit JSON gepackt, damit diese auf eine simple Weise in der HoloLens-Applikation deserialisiert werden können.

**3.3.3. Asset-Client.** Der Begriff *Asset-Client* umfasst eine Reihe von C#-Klassen, welche die Kommunikation zum Asset-Server herstellen und die Verwaltung der Asset-Bundles übernehmen. Insofern folgt die korrekte Interpretation, dass die HoloLens-Applikation, in der diese Klassen verwendet werden, als Asset-Client anzusehen ist.

Im Groben gibt es jedoch drei Verwaltungsklassen, mit deren Hilfe das dynamische Laden von Assets ermöglicht wird: AssetManagement, welche eine Singleton-Klasse ist und den Zugriff auf die eigentlichen Verwaltungsobjekte ermöglicht; AssetManager, welcher das Anfordern von Assets ermöglicht sowie SceneTransfer, womit die Objekte der aktuellen Szene gespeichert und wiederhergestellt werden können.

Der AssetManager implementiert die Kommunikation mit dem Asset-Server, in dem die vom Asset-Server angebotenen Endpoints bedient werden. Die auf diese Weise vom Server bezogenen Asset-Bundles werden in einem internen Cache zwischengespeichert, um einerseits eine unnötige Netzwerkauslastung zu vermeiden und die technische Limitierung zu bedienen, dass ein Asset-Bundle nicht mehrmals entpackt werden kann.

Während es in diesem Stadium des Projekts keine Rolle spielt, könnte es in zukünftigen Anwendungsfällen zu Problemen führen; sobald sich der Inhalt eines Asset-Bundles im Server ändert und diese Änderung sich im Client widerspiegeln soll, muss das vorige Asset-Bundle im Client freigegeben werden.

Die Klasse SceneTransfer ermöglicht das Speichern von Meta-Informationen über die Objekte in der aktuellen Szene. Hierzu gehören Positions- sowie Rotationsinformationen, Skalierung, genutzte Textur-IDs usw.

All diese Informationen werden in einem JSON-Konstrukt gebündelt und mit der Hilfe des AssetManagers zum Asset-Server gesandt, um dort persistent abgelegt zu werden. Das Wiederherstellen dieser Informationen folgt analog zum vorigen Schritt: Die Meta-Informationen werden vom Asset-Server abgefragt und den Objekten in der aktuellen Szene hinzugefügt. Hierzu werden jedoch die Objekte, deren Meta-Informationen geladen wurden, neu in die Szene gesetzt. Es handelt sich hierbei also um ein additives Verhalten; das Laden einer Szene bedeutet das Erzeugen neuer Objekte.

Doch welche Objekte in der Szene werden nun auf diese Weise gespeichert? Dies ist in der allgemeinen Nutzung der HoloLens die Aufgabe des Nutzers, nur dieser kann wissen, welche Objekte relevant sind. Um also die Objekte in der Szene zu spezifizieren, die abgespeichert werden sollen, wird ein Mechanismus benötigt, um eine Markierung vornehmen zu können.

Hier bedienen wir uns der Funktionalität der Unity-Engine und fügen jedem Objekt in der Szene, die durch den AssetManager geladen wurden, eine sog. Component hinzu. Diese kapselt die Informationen (in der aktuellen Version ein *boolean*) über das Objekt und markiert dieses entsprechend. Mit Hilfsfunktionen des SceneTransfers kann für jedes Objekt ein solcher Marker hinzugefügt oder entfernt werden. Nur Objekte mit einem solchen Marker werden bei der Speichern-Anfrage tatsächlich berücksichtigt.

## 3.4. Geplante Verbesserungen & Ausblick

Die aktuelle Implementierung der Asset-Behandlung ist äußerst rudimentär und prototypisch, womit die Notwendigkeit von Verbesserungen essentiell ist. In diesem Kapitel werden kurz die gröbsten Problemherde aufgezeigt und mögliche Verbesserungsmöglichkeiten aufgelistet.

**3.4.1. Asset-Server vereinheitlichen.** Die aktuelle Implementierung des Servers in Python ist allenfalls ausreichend, um die Anforderungen zu erfüllen. Da die genutzte Sprache jedoch großen Änderungen zwischen den Versionen unterliegen kann und die Eigenentwicklung einer selbstprogrammierten REST-API nicht zweckmäßig ist, muss in der nächsten Iteration eine definitive Entscheidung über die

genutzte Programmiersprache sowie benötigter Bibliotheken getroffen werden.

Die Anforderungen hierfür sind teilweise bekannt:

- Starke Typsicherheit der Programmiersprache
- Vordefinierte Bibliotheken zum Erstellen einer REST-Schnittstelle, jedoch mit der Möglichkeit, die Anfragen manipulieren zu können (kein simpler HTTP-GET-Server!)
- Kompilier- und Ausführbar unter Windows **und** Linux (bzw. x86/64 **und** ARM)

Zum Anfang der nächsten Projekt-Iteration muss eine Recherche diesbezüglich angefertigt werden, da unter den oben genannten Anforderungen einige Lösungskandidaten in Frage kommen.

**3.4.2. Szenen laden.** Das Laden einer Szene (bzw. der markierten Objekte der Szene) ist, wie in Kapitel 3.3.3 beschrieben, ein additives Verfahren. Dieses Verhalten ist jedoch nicht immer erwünscht und kann bisweilen zu äußerst verwirrenden Szenen führen.

Insofern wird es nötig sein, beim Laden einer Szene angeben zu können, ob die Objekte die Szene überschreiben oder lediglich als Objekt-Komposition hinzugefügt werden sollen. Dies kann sogar in eine eigene Funktion ausgelagert werden, die sich nur um das Speichern von Kompositionen kümmert.

**3.4.3. Komprimierung der Szene-Daten.** Kurze Tests mit dem Abspeichern von Szenen mit wenigen Objekten ($\leq 10$) und einer resultierenden Übertragungsgröße von ungefähr 200 Bytes zeigten eine nicht annehmbare Latenz von mehr als einer Sekunde. Während dies an der WLAN-Übertragung und anderen Netzwerkproblemen liegen kann, ist dieses Problem nicht ausreichend behandelt worden.

Ein Lösungsvorschlag wäre das Komprimieren der zu übertragenden Daten (z.B. durch den DEFLATE-Algorithmus). Hierzu müssen weitere Tests aufgestellt werden, welche die Übertragungsdatengröße mit der Übertragungszeit in Relation setzt, um fundierte Aussagen darüber treffen zu können, ob das Problem am genutzten Netzwerk oder der Implementierung der Übertragung liegt.

# 4. Mehrspieler-Umgebung

Eine der ersten Anforderungen war die Möglichkeit mit mehreren Nutzern die Szene betrachten und möglicherweise auch modifizieren zu können. Zeitgleich wurde darüber nachgedacht, die Applikation auf andere Endgeräte (z.B. Tablet) zu erweitern.

Zu diesem Zweck wurden zunächst zwei Ansätze verfolgt: Zum einen den von Microsoft entwickelten *SpectatorView*. Dieser stellte sich als unzureichend heraus, da die Architektur der *SpectatorView* nicht mit allen möglichen Geräten kompatibel war; das Projekt ist im weitesten Sinne eine Unity-Applikation und die *SpectatorView* ist ein externes Werkzeug, welches nicht alle von Unity möglichen Systeme unterstützt. Ferner wurde das System seit geraumer Zeit nicht mehr aktualisiert.

Eine Lösung bestand in der Nutzung der generischen Mehrspieler-Funktionalität von Unity. Da Unity von sich aus die Möglichkeit hat eine Applikation für verschiedene Endgeräte zu kompilieren, ist jede native Funktion von Unity auch automatisch in der Lage auf allen diesen System zum Einsatz zu kommen.

## 4.1. Mehrspieler in Unity

In Unity-Mehrspieler-Umgebungen gibt es immer einen Server. Wenn es dafür keine dedizierte Maschine oder Applikation gibt, muss einer der Clients die Rolle des Servers übernehmen.

Der Unterschied zwischen dediziertem Server und einem Client, der diese Aufgabe übernimmt, liegt in der Art und Weise wie der Server in die Szene eingreifen kann. Ein dedizierter Server übernimmt lediglich Koordination der verschiedenen Clients, ist aber selbst nicht in der Mehrspieler-Umgebung präsent. Im Gegensatz dazu ist ein Client, der die Rolle des Servers übernimmt, als Client in der Szene vorhanden, während im Hintergrund der Server-Prozess abläuft.

Da sich der Client mit dem Server auf der selben Maschine befindet, verwendet Unity einen speziellen *LocalClient* der sich direkt zum (lokalen) Server verbindet, ohne den Umweg über das Netzwerk zu gehen. Alle anderen Clients verbinden sich über reguläre Netzwerkkommunikation (Siehe Abbildung 2).

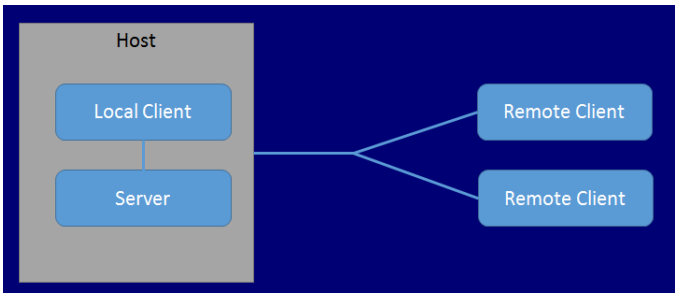



Abbildung 2. Verbindungen zwischen Server und Client [7]

## 4.2. Erzeugung von Objekten

Objekte werden in Unity typischerweise mit der Methode *GameObject.Instantiate* erzeugt. Innerhalb einer Mehrspieler-Umgebung müssen diese Objekte allerdings auf allen verbundenen Clients erzeugt werden, selbst auf solchen, die zum Zeitpunkt der Objekt-Erzeugung noch nicht mit dem Netzwerk verbunden waren.

Zu diesem Zweck müssen alle Objekte auf dem Server *gespawnt* werden [8]. Dies kann nur auf dem Server ausgeführt werden und sorgt für ein Erzeugen des Objektes auf allen verbundenen Clients.

### 4.3. Mehrspieler-Autorität

Wenn ein Objekt auf mehreren Clients erzeugt wurde, muss das Unity System für eine Synchronisation zwischen Clients sorgen. Dies wird über sogenannte *Object-State-Synchronization* erreicht, bei der der Zustand eines Objektes auf anderen Clients repliziert wird, z.B. Position.

Dabei gibt Unity vor, dass jedes Objekt innerhalb des Netzwerkes von exakt einem Client besessen wird. Dieses Konzept wird auch als *Autorität* bezeichnet. Wenn ein Objekt keinem Client zugeordnet werden kann, wird der Server verwendet. In einer typischen Unity-Applikation wie etwa einem Computerspiel ist der Server in der Regel für den Großteil der Objekte verantwortlich.

Diese Autorität diktiert welcher Client die ausschlaggebende Instanz für die Objekt-Synchronisation des einzelnen Objekts ist. Abbildung 3 erläutert dieses Konzept anhand der Spieler-Objekte.

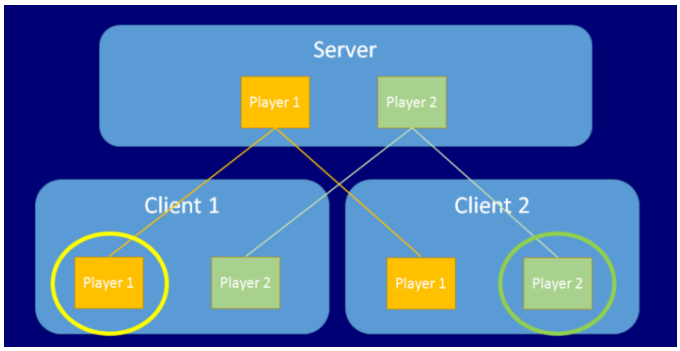



Abbildung 3. Darstellung von Mehrspieler-Autorität [7]. Der gelbe Kreis denotiert die Autorität für das *Player 1* Objekt, der grüne Kreis für das *Player 2* Objekt.

In der Abbildung ist zu sehen, wie jeder Spieler auf jedem Client existieren muss und dass jeder Spieler auch gleichzeitig Autorität über sein eigenes Spieler-Objekt hat. Der Status der Spieler-Objekte wird über das Unity System an die anderen Clients übertragen, aber die entscheidende Instanz liegt in diesem Beispiel beim jeweiligen Client selbst.

Es sei anzumerken, dass Spieler-Objekte in Unity eine besondere Bedeutung und Limitierungen haben. Dies spielt für das *HoloLens*-Projekt eine untergeordnete Rolle. Mehr kann dazu in *Player Objects* [9] gelesen werden.

### 4.4. Anforderungen an das *HoloLens*-Projekt

Die Herausforderung des Projektes bestand, abgesehen von technischen Schwierigkeiten, vor allem in der Einrichtung des Netzwerk-Systems.

Unity, bedingt durch die Spieleindustrie für die es entwickelt wurde, ist darauf ausgelegt, dass jeder Spieler mit seinem eigenen Spieler-Objekt die Szene betritt und mit ihr interagiert.

Da in einer *Augmented Reality* Umgebung der Spieler aber der echte Mensch ist, warf dieses Konzept erst einige Schwierigkeiten auf. Siehe auch 4.5.

Ferner musste eine Lösung gefunden werden, bei der Spieler in der Szene dynamisch die Autorität von Objekten verändern können. Nur so wäre es möglich, dass mehrere Nutzer die Szene verändern können.

### 4.5. Prototyp 1

Das Ziel des ersten Prototypen war die Einrichtung der Netzwerk-Architektur und Testen der Objekt-Synchronisation.

Zunächst wurde die Autorität nur statisch zugewiesen, so war nur ein Client in der Lage die Szene zu modifizieren. Es war zwar möglich das Objekt auch auf den anderen Clients zu bewegen, dies wurde allerdings nicht über das Netzwerk repliziert, es folgte eine Desynchronisation der Objekte.

Der Ansatz das Spielerobjekt dynamisch auf das jeweils zu modifizierende Objekt zu übertragen war fehlgeschlagen, da Unity dies nicht zulässt. Wie sich dann allerdings herausstellte war dies die falsche Methode um mit dem Problem umzugehen.

Anstatt das Spielerobjekt zu verändern, müsste man es lediglich verstecken (z.B. kein 3D Modell erzeugen) und stattdessen die Autorität der Nicht-Spieler-Objekte verändern. Dabei wird das Spieler-Objekt als Kommando-Puffer für den Server verwendet, denn die Autorität des Spieler-Objektes ist immer automatisch zu dem zugehörigen Client zugewiesen. Dies resultierte dann in einem zweiten Prototypen (4.6).

### 4.6. Prototyp 2

Im Folgenden wurde dann ein Prototyp entwickelt, bei dem Clients die Autorität anfordern können. Der Aufbau für diesen Prototyp war etwas komplizierter, denn die Zuweisung der Autorität kann nur auf dem Server erfolgen; der Server muss Sie absegnen.

Die Lösung für dieses Problem war die Verwendung der Unity-*Command*-RPC Methodik. Diese erlaubt es Code auf anderen Clients auszuführen und löst dementsprechend das Problem der Autoritäts-Zuweisung.

Jetzt war es möglich die Autorität auf Objektbasis dynamisch neu zu vergeben und damit die Grundlage für eine Umgebung, bei der die Nutzer jedes in der Szene erzeugte Objekt modifizieren können, geschaffen.

Objekt-Synchronisation ist wie bei Prototyp 1 immer noch nur unidirektional, allerdings ist dies durch die dynamische Neuzuweisung der Autorität für den Nutzer nicht sichtbar. In der Zukunft kann ein Objekt z.B. beim Anfassen die korrekte Autorität zugewiesen bekommen. Dies wurde im Prototyp allerdings nicht umgesetzt.

### 4.7. Weiterentwicklungen

Alle vorgestellten Konzepte konnten erfolgreich in einer entkoppelten Version des Projektes getestet werden und sind lauffähig. Das Wissen und Verfahrenstechniken zur Umsetzung von Mehrspieler im HoloLens-Projekt wurde erreicht.

Eine tatsächliche Implementierung und Integration in das Hauptprojekt wurde aus Planungsgründen noch nicht durchgeführt. Dies erfolgt im Laufe zukünftiger Entwicklungsarbeit.

# 5. Raspberry Pi Asset-Server

Wie bereits in den vorherigen Kapiteln beschrieben, bezieht die HoloLens Applikation die Assets zur Laufzeit über einen Webserver aus dem Netzwerk. Damit der Asset-Server im Netzwerk erreichbar ist, muss das Netzwerk entsprechend konfiguriert werden, was schnell zu Problemen führt, wenn man nicht die möglichen Rechte oder Kenntnisse dazu besitzt. Um dieses Problem zu umgehen wurde ein Raspberry Pi aufgesetzt, das als Wlan Hotspot ein eigenes frei konfigurierbares Netzwerk bereitstellt und gleichzeitig den Asset-Server hostet. Beim Start des Raspberry Pi wird der Hotspot sowie der Asset-Server automatisch gestartet.

Das Setup aus Raspberry Pi und Hololens ist portabel und funktioniert ohne großen Konfigurationsaufwand. Eine Internet Verbindung steht in dem Raspberry Pi Netzwerk nicht zur Verfügung, falls notwendig lässt sich dies aber entsprechend konfigurieren. Verwendet wurde das Raspberry Pi 3 in Kombination mit Raspbian Jessie Lite. Für die Hotspot Funktionalität wurde hostapd und dnsmasq genutzt. [29] Für den Asset-Server wurde zusätzlich Python 3 installiert. Eine Anleitung mit allen notwendigen Schritten sowie ein Image mit der aktuellen Konfiguration wurden erstellt.

# 6. Motorvisualisierung

## 6.1. Grundlagen

Auf einer modernen Theaterbühne existieren viele schwere Objekte wie Scheinwerfer, Vorhänge oder Bühnenteile, die durch Motoren bewegt werden. Diese Objekte können für die Darsteller auf der Bühne sehr gefährlich werden, wenn der Bühnenarbeiter diese bei schlechten Lichtverhältnissen oder aufgrund von Verdeckung blind steuert. Um diese Gefahr zu minimieren soll die aktuelle Lage dieser beweglichen Objekte, sowie deren mögliche Positionen auf der HoloLens visualisiert werden, um dem Bühnenarbeiter eine bessere Übersicht zu verschaffen und so Unfälle zu vermeiden.

Die Motoren auf der Bühne werden von einem Bühnenarbeiter über eine Motorsteuerungssoftware bedient, die die entsprechenden Steuerungssignale an die Motoren schickt. Außerdem ist die Software in der Lage die Steuerungssignale als UDP-Multicast in einem Netzwerk zu versenden. Die aktuellen Positionsdaten werden im Folgenden dynamische Motordaten genannt. Diese dynamischen Motordaten, also aktuelle Position, Geschwindigkeit und Orientierung, werden nach dem PosiStageNet Standard übertragen. [28] Die statischen Motordaten, also Motor-ID, Montageposition, Bewegungsachse und Achsenreichweite sind in einer Datenbank hinterlegt.

## 6.2. Prototypische Implementierung

Der entwickelte Prototyp umfasst eine sehr minimalistische Motorvisualisierung, einen Motorcontroller und einen Eventmanager zum Empfangen von UDP-Multicast Nachrichten aus dem Netzwerk und deren Bereitstellung als Unity Event.

Ein Motor wird in dem Prototyp durch eine rote Achse im Raum dargestellt, auf der sich der Endeffektor des Motors bewegen kann. Dabei wird die aktuelle Position des Endeffektors als grüner Punkt auf der Achse dargestellt. Abbildung 4 zeigt, wie dies im aktuellen Prototypen aussieht. Das Motor Skript hält die statischen Daten des Motors, sowie die zur Visualisierung notwendigen Objekte. Um einen Motor zu Erzeugen muss zunächst ein leeres Gameobject erzeugt werden. Diesem wird nun das Motor Skript als Komponente hinzugefügt. Mit der *InitMotor(MotorData motorData, Transform parent)* Funktion des Skripts werden nun die statischen Daten gesetzt und die entsprechende Visualisierung erzeugt. Die Klasse *Motordata* kapselt die statischen Motordaten um diese per Deserialisierung aus einer Konfigurationsdatei erzeugen zu können. Mit der MoveEffektor(float diff) Funktion lässt sich der Endeffektor auf der Motorachse verschieben.

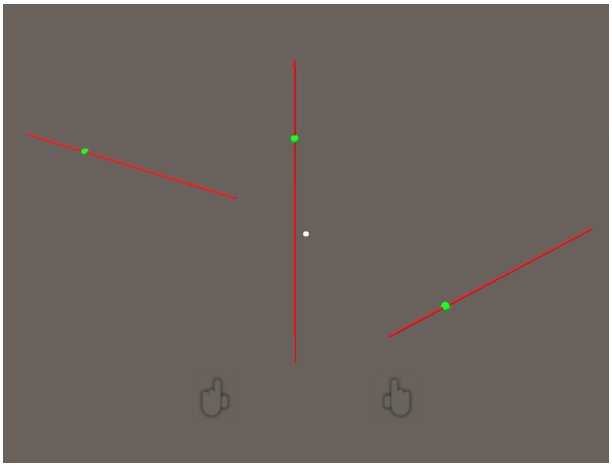


Abbildung 4. Screenshot einer Szene mit drei Motoren im Unity Editor

Das Erzeugen und Bewegen von Motoren wurde im nächsten Schritt durch einen Motorcontroller automatisiert. Der Motorcontroller ist ebenfalls ein Skript, das einem leeren Gameobjekt zugewiesen wird, das nur einmal pro Szene existieren darf. In der *Start()* Funktion des Motorcontrollers erzeugt dieser anhand von einer JSON-Konfigurationsdatei, welche die statischen Motordaten enthält, alle dort definierten Motoren wie im vorherigen Absatz beschrieben. [30] Der Motorcontroller ist dabei gleichzeitig das übergeordnete Gameobject aller Motoren, sodass diese an der Position des Motorcontrollers ihren Ursprung haben. Bei der Erzeugung der Motoren wird für jeden Motor eine Referenz auf diesen in einem Dictionary unter der jeweiligen Motor-ID gespeichert. Um die Motoren nun zu bewegen wird die Funktion *HandleMotorDataReceived(string data)* genutzt,

die übergebene Variable *data* enthält dabei die Motor-ID und die entsprechende Positionsänderung. Die Funktion sucht daraufhin im Motor Dictionary nach der jeweiligen Motor-ID und passt die aktuelle Position des referenzierten Motors entsprechend an. Die Funktion *HandleMotorDataReceived()* wird dem Event *MotorDataReceived* zugewiesen, das in der dritten Komponente, dem Eventmanager, erzeugt wird, sobald eine Nachricht mit Motordaten empfangen wird.

Die bisher einzige Aufgabe des Eventmanagers ist es, im Netzwerk UDP-Multicast Nachrichten zu empfangen, deren Inhalt zu parsen und gegebenenfalls Unity Events für die anderen Anwendungskomponenten zu erzeugen. Der Eventmanager startet dazu einen Thread, der einen Socket erzeugt und mit diesem UDP-Multicast Nachrichten empfängt. Nachrichten werden asynchron empfangen und in eine Warteschlange eingereiht. Zu jedem Frame wird die Warteschlange abgearbeitet, dazu werden die Nachrichten geparst und entsprechende Unity Events erzeugt. Der Eventmanager kann auch dazu genutzt werden, um UDP-Multicast Nachrichten zu senden, dazu steht die statische Funktion LogMessage(string data) bereit. Dies kann für spätere Log-Funktionalitäten nützlich sein. An dieser Stelle sollte noch angemerkt werden, dass mehrere verschiedene Eventmanager in einer Anwendung vermieden werden sollten, falls also bei anderen Komponenten der Anwendung Events benötigt werden, sollten diese in einem gemeinsamen Eventmanager zusammengefasst werden.

Zur Überprüfung der Netzwerk Kommunikation wurde eine einfache Python Anwendung implementiert, um entsprechende UDP-Multicast Nachrichten ins Netzwerk zu senden und alle UDP-Multicast Nachrichten im Netzwerk zu loggen. Die Kommunikation mit der Anwendung im Unity Editor funktionierte wie erwartet. Abbildung 5 zeigt die Python Anwendung und einen Ausschnitt aus dem Log, der ebenfalls als Datei gespeichert wird.

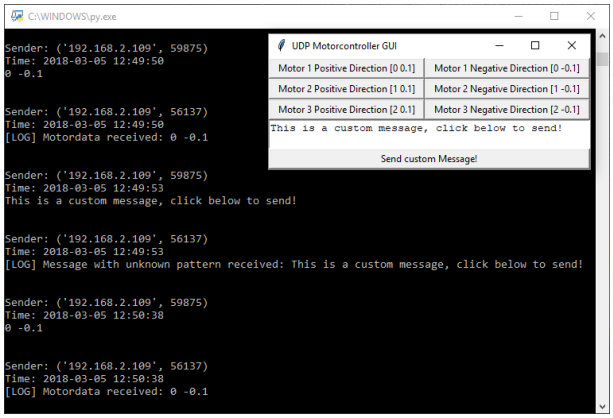



Abbildung 5. Screenshot der Python Anwendung zum Senden und Loggen von UDP-Multicast Nachrichten

Die beiden Komponenten Motorvisualisierung und Motorcontroller funktionieren bisher optimal im Unity Editor, dem HoloLens Emulator und auf der HoloLens selbst. Die Implementierung des EventManagers ist etwas komplexer und funktioniert bisher nur im Unity Editor. Ein erstes Problem gab es bei dem Versuch, die Anwendung für UWP zu bauen. Da Unity UdpCLient und UWP DatagramSocket als Socket Implementierung nutzen gab es dort zunächst Konflikte.

Das Problem lässt sich durch die Verwendung der Präprozessordirektiven *#if UNITY_EDITOR* und *#if WINDOWS_UWP* lösen. [32] Der Code für UWP wurde entsprechend angepasst und kompiliert erfolgreich. Das anschließende Testen der Kommunikation im HoloLens Emulator blieb dennoch erfolglos. Der bisher einzige gefundene Lösungsansatz, die Konfiguration des Hyper-V Netzwerkadapters anzupassen, blieb bisher ohne Erfolg. [31] Die Kommunikation konnte auf der HoloLens selbst bisher noch nicht getestet werden.

Im aktuellen Stand des Prototyps sind die UDP-Multicast Pakete sehr simpel, sie bestehen nur aus einem einfachen String, in dem die Daten getrennt durch Leerzeichen im Eventmanager mit RegExp geparst werden. In der nächsten Version des Prototyps soll dieses einfache Format auf den PosiStageNet Standard umgestellt werden. Für PosiStageNet existiert bereits eine Implementierung, die jedoch auf .NET 4.5 basiert und eine Inkompatibilität mit .NET 3.5 in Unity aufweist. Verschiedene Möglichkeiten, wie das Upgrade von Unity auf ein experimentelles .NET 4.5 oder downgraden der PosiStageNet Implementierung auf .NET 3.5 waren bisher erfolglos. Im nächsten Semester müssen hier definitiv noch weitere Möglichkeiten evaluiert und im Notfall eine eigene Implementierung des PosiStageNet Standards entwickelt werden. Das generelle Empfangen von UDP-Multicast Nachrichten auf der HoloLens hat jedoch Vorrang.

# 7. SpectatorView

## 7.1. Einleitung

Um bei der Verwendung der HoloLens eine Möglichkeit zu schaffen, dass Dritte ohne HoloLens am Erlebnis teilnehmen können hat Microsoft den sogenannten *SpectatorView* entwickelt. Dabei wird die von der HoloLens gerenderte Szene 2-dimensional mit einem normalen Bildschirm angezeigt.

## 7.2. Aufbau im Projekt

Der Aufbau im Master PSE sollte über ein Notebook und eine FullHD-Webcam realisiert werden. Mit Hilfe einer 3D gedruckten Halterung soll die Webcam auf die HoloLens montiert werden. Die Verbindung beider Geräte zum Notebook wird über USB-Kabel realisiert. Nachdem beide Geräte installiert sind und die SpectatorView Software nach einigen Startschwierigkeiten kompiliert werden konnte, müssen beide Geräte zusammen kalibriert werden. Danach sollte der SpectatorView eingerichtet und für Zuschauer verwendbar sein.

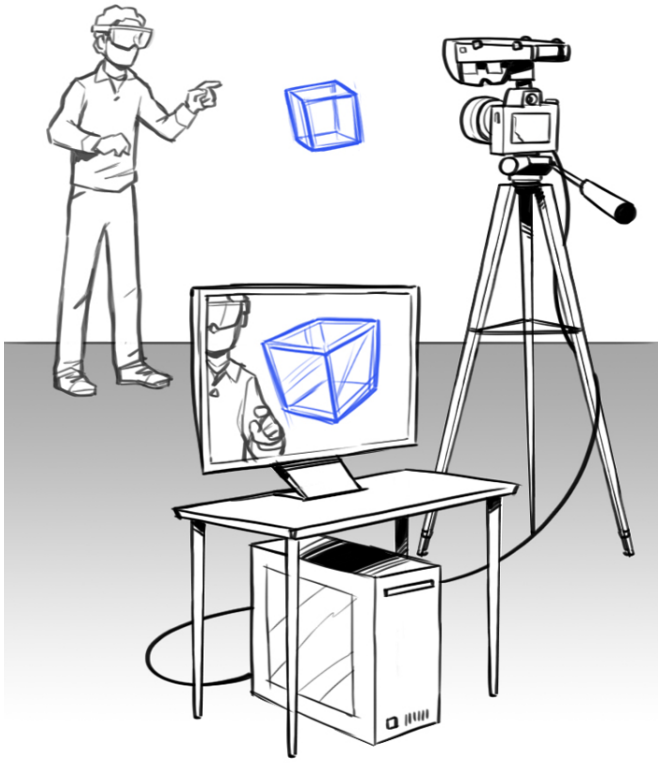


Abbildung 6. Zeichnung des SpectatorView Setups [5]

### 7.3. Problemstellungen im Projekt

Das größte Problem bei der Realisierung in unserem Projekt ist der Unterschied zwischen den Versionen von Unity. Während der SpectatorView in Unity Version 5.6 geschrieben wurde wird in unserem Projekt die Unity Version 2017.2 verwendet. Ein gravierender Unterschied dieser beiden Versionen ist, dass der *namespace* VR (Virtual Reality) in AR (Augmented Reality) umbenannt wurde, was für eine große Zahl an manuellen Ausbesserungen im Quellcode sorgte. Die meisten der dort veränderten Namespaces und zugehörigen Methoden werden in Unity 2017.2 als *deprecated* aufgezeigt und müssen mit den neueren Versionen ersetzt werden.

Die SpectatorView Applikation ist aufgeteilt in *Calibration* und *Compositor*. Während die Methoden im Calibration-Teil noch so weit ausgetauscht werden konnten, dass es möglich war zu kompilieren und zu starten, um Kamera und HoloLens miteinander zu kalibrieren, waren die Unterschiede im Compositor so groß, dass es **nicht möglich** ist diesen Teil von SpectatorView unter der Unity Version 2017.2 zu kompilieren.

## 8. Dreidimensionales Bildtracking mit Vuforia

### 8.1. Einleitung

Vuforia ist eine Augmented Reality Plattform, die vor allem für das Verfolgen von Bildern und Markern bekannt ist. Vuforia verwendet die Webcam der HoloLens, um voreingestellte Bilder und Marker zu erkennen und die Position zum Szenennullpunkt bezüglich der HoloLens zu berechnen.

### 8.2. Problemstellungen im Projekt

Wie schon beim SpectatorView ist das Problem beim Einsatz von Vuforia, dass Unity Version 2017.2 kaum unterstützt wird. In Unity wird in unserem Projekt das sogenannte *HoloToolkit* von Microsoft eingebettet, mit dem unter anderem die korrekten Kameraeinstellungen für die HoloLens in Unity automatisch integriert werden können. Der Vuforia-Service nutzt eine eigene Szenenkamera um die Webcam anzusteuern und so das Bild bzw. den Marker zu erkennen. Hierbei soll die Kamera von Vuforia als Tochterelement der HoloLens Kamera implementiert werden, um zu ermöglichen, dass die Positionierung der HoloLens Kamera auch auf die Kamera von Vuforia übertragen wird.

Vuforia hat in allen Tests, die während unserer Entwicklungen abgelaufen sind, den Bezug zur HoloLens Kamera verloren, dabei hatte es keine Auswirkungen, ob die Vuforia-Kamera alleine, gleichgestellt mit oder als Tochterelement der HoloLens Kamera implementiert wurde. Wenn die HoloLens aufgesetzt und dann das Test-Programm gestartet wird funktioniert das Tracking eines Bildes an genau dieser Stelle und es können 3-dimensionale Objekte auf ein Bild oder Marker gesetzt werden. Das gedruckte Bild/Marker kann per Hand bewegt werden und das 3D Objekt bewegt sich mit. Es ist während der gesamten Zeit der Entwicklung nicht möglich gewesen, die Vuforia Kamera so in das Programm einzusetzen, dass es mit dem Tracking der HoloLens mitläuft. Das resultierte in dem Problem, dass das Vuforia Bildtracking nur an der Null-Position funktioniert, also der Position, in der sich die HoloLens befindet, wenn das Programm gestartet wird.

Getestet wurde in unserer Entwicklungsumgebung jede Version von Unity von 5.4 bis 2017.3, um zu überprüfen, ob die integration von Vuforia möglich ist. Der Stand bis zum Ende des PSE Kurses ist, dass es **nicht** möglich ist, Vuforia zum Imagetracking nutzen zu können.

## 9. UI (User Interface)

Die Aufgabe der UI-Gruppe besteht zum einen darin, einen Asset Store zur Verfügung zu stellen, der die Assets vom Server anzeigt und diese durch Auswahl in die Szene lädt. Zum anderen soll die Manipulation von Objekten in der Szene ermöglicht werden.

## 9.1. Konzeptphase

Zu Beginn des Semesters informierten wir uns zunächst über bestehende MR-/VR-Standards zum Thema UI. Dazu fanden wir im Wesentlichen folgende unterschiedliche Ansätze:

- **HUD (Head Up Display)**
  Dabei werden die Informationen dem Nutzer direkt ins Sichtfeld projiziert. Dieser Ansatz findet häufig Verwendung bei Piloten oder Autofahrern. Allerdings wirkt dieser Ansatz im MR-/VR-Umfeld unnatürlich, da die Immersion des Nutzers in das Umfeld behindert wird.

- **Body-locked**
  Bei diesem Ansatz werden UI-Elemente in einem festen Abstand vor dem Nutzer projiziert. Bewegt sich der Nutzer, so bewegen sich die UI-Elemente in der Welt mit.

- **World-locked**
  Bei diesem Ansatz sind UI-Elemente fest in der Welt verankert. Der Nutzer kann sich dabei um diese herum bewegen.

Für die Implementierung des Asset Stores entschieden wir uns in Rücksprache mit dem Kunden letztendlich für den World-locked Ansatz. Vorteil dabei ist, dass der Nutzer das UI an geeigneter Stelle platzieren kann und anschließend an Objekten in der Szene weiterarbeitet. Möchte der Nutzer sich nun ein neues Objekt holen, braucht er sich beispielsweise nur zum UI zu drehen und kann direkt weiterarbeiten. Beim Body-locked Ansatz hingegen müsste der Nutzer jedes mal, wenn er mit dem UI arbeiten will, dieses erst einblenden, seine Aktion ausführen und anschließend das UI wieder ausblenden. Somit wird der Arbeitsfluss gestört.

## 9.2. UI Asset Store

Die Aufgabe des UI Asset Stores liegt in der Anzeige der Assets, sowie der gespeicherten Szenen vom Server. Dem Nutzer soll es möglich sein, Assets durch Auswahl in die Szene zu laden. Nach dem Laden hängt das Objekt zunächst am Gaze. Unter dem Gaze versteht man die Position, an die der Nutzer schaut. Für den aktuellen Prototyp (vgl. Abb. 7) stand dabei die Funktionalität im Vordergrund. Der Nutzer kann mit den Pfeiltasten die Seiten durchblättern und so nach Assets suchen. Durch Anklicken eines Assets, wird es in die Szene geladen. Durch Klicken auf “SaveScene” kann der Nutzer die aktuelle Szene auf dem Server speichern. Über “LoadScene” wird die aktuelle Szene durch die gespeicherte Szene auf dem Server ersetzt. Um den Status für die Verbindung zum Server anzuzeigen, wird dem Nutzer zunächst der Ping zum Server anschaulich gemacht.

## 9.3. Manipulation von Objekten

Eine Anforderung vom Kunden ist die Manipulation von Objekten in der Szene. Darunter fallen die gängigen


Abbildung 7. Aktueller Asset Store Prototyp

geometrischen Transformationen, aber auch das dynamische Texturing von Assets. Nachdem der Nutzer ein Objekt in der Szene auswählt, erhält es zunächst eine blaue Umrandung als Feedback für den Nutzer. Unter dem Objekt erscheinen dann Icons für die geometrischen Manipulationen (vgl. Abb. 8). Im freien Modus hängt das Objekt am Gaze. Der Nutzer kann sich an eine beliebige Position bewegen und das Objekt wieder ablegen. Für eine präzisere Steuerung stehen dem Nutzer folgende Möglichkeiten zur Verfügung:

- **Translation** Änderung der Position des Objekts in x-z- bzw. x-y-Ebene.
- **Rotation** Rotation des Objekts um die x- bzw. y-Achse.
- **Skalierung** Änderung der Größe des Objekts.

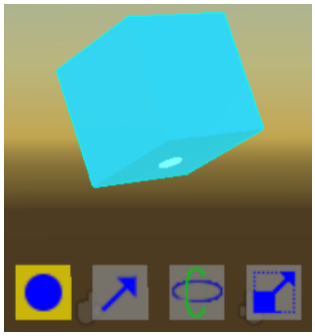


Abbildung 8. Geometrische Manipulationen eines Assets

## 9.4. Steuerung mit XBox Controller

Für eine komfortablere Bedienung haben wir uns in Rücksprache mit dem Kunden dazu entschieden die Steuerung über einen XBox Controller zu ermöglichen. Dadurch stehen dem Nutzer verschiedene Funktionalitäten durch

Knopfdruck zur Verfügung. Außerdem können Objekte dadurch wesentlich genauer manipuliert werden. Im Folgenden wird auf die implementierten Funktionalitäten des aktuellen Prototyps und die zugehörige Tastenbelegung auf dem XBox Controller eingegangen:

- **A** Durch einfaches Drücken der A-Taste wird das Objekt ausgewählt, welches vom Gaze-Pointer getroffen wird. Durch erneutes Drücken der Taste wird das Objekt ausgewählt und es erhält als Visualisierung eine blaue Umrahmung. Als Prototyp wurde ebenfalls ein Infofenster entwickelt, welches neben einem ausgewählten Objekt erscheint. Darin kann man allerlei nützliche Informationen zum Objekt anzeigen lassen.

- **B** Durch einfaches Drücken der B-Taste wird das ausgewählte Objekt abgelegt. Durch langes Drücken der B-Taste wird das ausgewählte Objekt gelöscht. Dabei erhält der Nutzer ein visuelles Feedback in Form eines Fortschritt-Kreises, sodass kein Objekt versehentlich gelöscht wird.

- **X** Durch Drücken der X-Taste wird der Asset Store in Blickrichtung neu positioniert, bzw. ausgeblendet, falls er sich im Blickfeld befindet.

- **Y** Durch Drücken der Y-Taste wird das Debug-Fenster neu positioniert.

- **Schultertasten L/R** Sobald ein Objekt ausgewählt ist, kann durch Drücken der Schultertasten der Modus für die geometrische Manipulation des Objekts (vgl. Abb. 8) gewechselt werden.

- **Control Stick** Sofern ein Objekt ausgewählt ist und der gewünschte Modus für die geometrische Manipulation eingestellt ist, erfolgt die eigentliche Manipulation um die Achsen mit dem Control Stick.

## 9.5. Gizmos

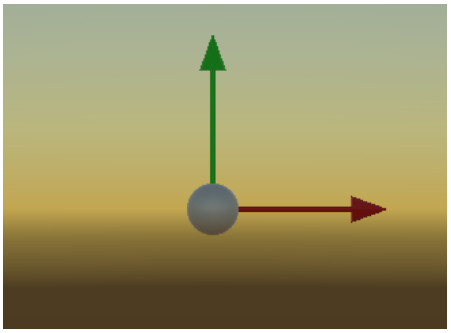
Die Anforderungen des Kunden an die Gizmos waren wie folgt. Sie sollten es ermöglichen beliebige Hologramme durch Gesten zu verschieben, zu skalieren und zu rotieren. Weiterhin sollten die Optik sowie die Handhabung dem quasi Industriestandard entsprechen, wie er in den meisten gängigen 3D Editoren wie Unity, 3ds Max, etc. zu finden ist.
Da uns selbst die nötige Erfahrung für die Entwicklung der Gizmos fehlte, begannen wir mit einer Internetrecherche. Es wurden mehrere bereits fertig entwickelte Lösungen auf Integration und Erweiterbarkeit verglichen. Letzten Endes entschieden wir uns für [4], welche eine Implementation zutage förderte, die mit einer Ausnahme genau unseren Anforderungen entsprach.

Wir entschieden uns diese Implementation als Basis für unsere eigene zu verwenden. Da der vorhandene Code auf die Benutzung mit der Maus ausgelegt war, musste dieser im ersten Schritt für die Gestensteuerung der HoloLens angepasst werden.
Die erste Frage, die sich hier stellte, war, welche der zur Verfügung stehenden Gesten genutzt werden sollten, um die Bedienung mit einer Maus zu emulieren, da wir uns an den erwähnten Bedienkonzepten verschiedener 3D Editoren orientieren sollten.
Für die Auswahl von Objekten, was einem Mausklick entsprechen würde, fiel die Entscheidung auf die “Air Tap” Geste sowie die Blickrichtung des Nutzers, um festzustellen, auf welches Objekt der Klick anzuwenden ist. Da diese Kombination intuitiv dem entspricht, was aus maus- oder touchgesteuerten Anwendungen bekannt ist sowie in fast allen HoloLens-Anwendungen zum Einsatz kommt, war diese Entscheidung naheliegend.
Weit weniger eindeutig war die Frage, welche Gesten genutzt werden sollten um die Manipulation der Hologramme, Verschieben, Rotieren und Skalieren, umzusetzen. Informationen zu den zur Verfügung stehenden Gesten sind in [3] zu finden. Von diesen fielen für uns die “Manipulation” und “Navigation” Gesten in die nähere Auswahl.
Die “Manipulation” Geste ist unter den komplexeren Gesten diejenige, die am ehesten einem “ziehen” mit der Maus entspricht. Um sie auszuführen, führt der Nutzer einen “Air Tab” durch, öffnet jedoch nicht die Hand, sondern bewegt sie, um dadurch Input für eine Anwendung zu erzeugen.
Die “Navigation” Geste startet ähnlich, mit einem gehaltenen “Air Tab”, Bewegungen der Hand vom Mittelpunkt des Sichtfeldes heraus resultieren jedoch in einem kontinuierlichen Signal zwischen -1 und 1. Die Geste entspricht damit am ehesten einem Joystick oder dem Scrollen in einem Fester unter Einsatz der mittleren Maustaste. Wir entschieden uns für die “Manipulation” Geste, da ihre Handhabung am ehesten dem entspricht, was die meisten Nutzer aus ihrer Erfahrung mit herkömmlichen Anwendungen heraus erwarten würden.

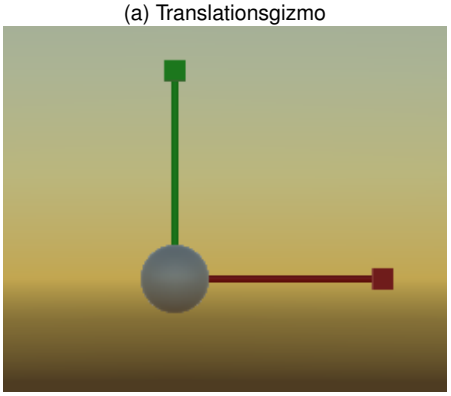
Unabhängig vom Problem der Nutzerinteraktion, musste ein neuer Layer erstellt werden, damit die Gizmos verwendet werden konnten. Dieser neue Layer wurde mit einer zweiten orthographischen Kamera verbunden. Hiermit konnten die Gizmos, unabhängig von der Position der Objekte, auf diese gerendert werden.

Beispiel: Hinzufügen eines Scale-Gizmos zur Szene

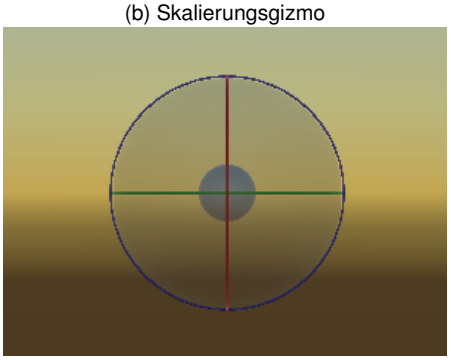
1) Ein Gizmo Layer muss angelegt werden, welcher alle Gizmos beinhaltet.
2) Eine orthographische Kamera muss erstellt werden. Diese soll nur auf den im Schritt 1 erstellten Layer gerichtet sein. Um sicherzustellen, dass die Gizmos auf den Objekten dargestellt werden, muss der “Depth”-Wert dieser Kamera höher als die Main-Kamera sein. Die Flags müssen auf “Depth Only” gesetzt werden. (Siehe Abb. 10)



(a) Translationsgizmo



(b) Skalierungsgizmo



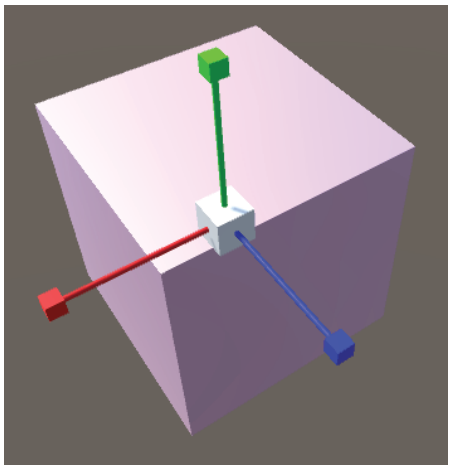
(c) Rotationsgizmo

Abbildung 9. Gizmos der ersten Version

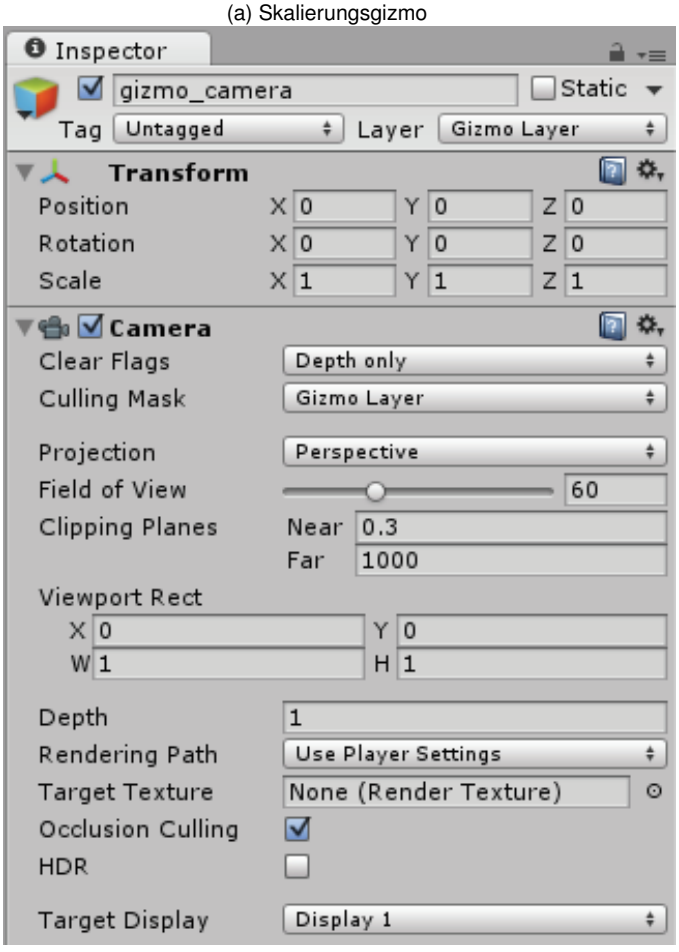
Zur Entwicklung des ersten Prototypen ist zu sagen, dass diese im HoloLens Emulator stattfand, d.h. die Gesten durch Maus und Tastatur simuliert wurden. Die Funktionsweise der Gizmos war wie folgt:

1) Der Nutzer fokussiert mit seinem Blick ein Hologramm an und führt einen “Air Tab” aus. Daraufhin erscheint das Translationsgizmo in dem ausgewählten Hologramm.
2) Mit einem erneuten “Air Tab” auf das Hologramm kann durch die verfügbaren Gizmos gewechselt werden.
3) Um ein Hologramm durch das ausgewählte Gizmo zu manipulieren, muss die oben beschriebene “Manipulation” Geste auf eine Achse des Gizmos angewandt werden. Die Achse muss dabei stets durch die Blickrichtung des Nutzers anvisiert bleiben.

Die so entwickelten Gizmos und deren Bedienkonzept



(a) Skalierungsgizmo




(b) Settings zu 5.3

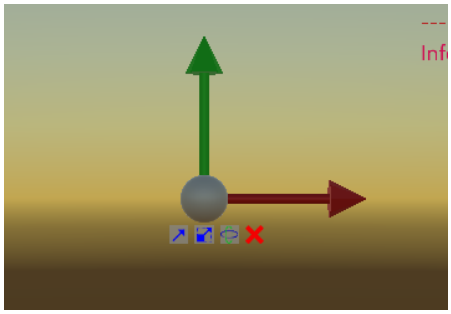
Abbildung 10. Einstellungen fürs Gizmo

erwiesen sich beim Versuch sie auf der HoloLens zu nutzen als praktisch unbedienbar. Es war zwar zu erwarten, dass die unterschiedlichen Eingabemöglichkeiten der HoloLens und des Emulators zu Differenzen in der Handhabung führen würden, doch die Schwere dieser war unerwartet.
Als grundlegendes Problem erwies sich dabei in erster Linie die mangelnde Präzision der Steuerung durch die Blickrichtung, da hierzu der Nutzer den gesamten Kopf bewegen muss. Für die Bedienung der Gizmos bedeutete dies, dass es sich als sehr schwierig erwies eine Achse des Gizmos auszuwählen und eine Bewegung auf dieser auszuführen. Weiterhin war es nicht unüblich, während der Bewegung von einer Achse auf eine andere abzurutschen, was zu unerwünschten Manipulationen des Hologramms führte.

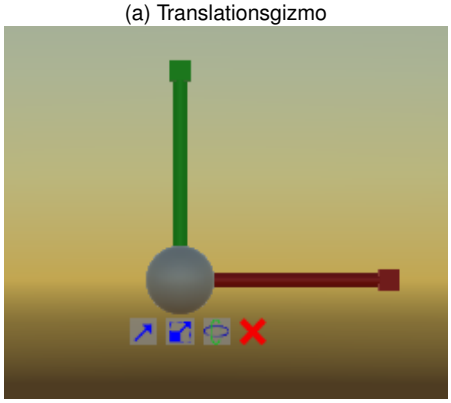
Mit diesen Erfahrungen wurde ein zweiter Prototyp entwickelt. Als Veränderung gegenüber den ersten Prototypen ist in erster Linie eine Vergrößerung der Bedienelemente der Gizmos zu nennen. Bei den Gizmos zum Bewegen und Skalieren war dies durch ein simples Vergrößern der Gizmos selbst möglich. Für das Gizmo zum Rotieren war dies nicht möglich, da es aus Ringen besteht, welche beim Skalieren zwar ihren Radius ändern, jedoch der Durchmesser des Rings gleich bleibt. Um Ringe mit größerem Durchmesser zu erhalten wurde daher ein entsprechender Torus in Blender erzeugt und importiert.
Als zweite Änderung wurde ein zusätzlicher Arbeitsschritt für die Bedienung der Gizmos eingebaut. Bevor nun eine Achse manipuliert werden kann, muss diese nun durch einen “Air Tab” auf den entsprechenden Handle selektiert werden. Anschließend findet die Manipulation wie gewohnt statt. Ein Abrutschen auf eine andere Achse hat nun jedoch keinen Effekt mehr.
Als letzte Änderung sind die Icons unter dem Hologramm zu nennen. Sie wurden in einem Versuch eingeführt den Workflow der Controller- und Gestensteuerung so einheitlich wie möglich zu gestalten, um neuen Nutzern den Einstieg zu erleichtern. Anstatt die verschiedenen Gizmos durch “Air Tabs” auf das Hologramm durchzuschalten, können diese nun durch einen “Air Tab” auf das entsprechende Icon aktiviert werden. Die Icons sind dabei die gleichen, die auch bei der Controllersteuerung zum Bewegen, Skalieren und Rotieren von Hologrammen zu finden sind. Zusätzlich wurde ein Icon hinzugefügt, mit dem das Gizmo ausgeblendet werden kann.

Tests der neuen Gizmos mit der HoloLens, insbesondere durch Nutzer, die diese vorher noch nicht benutzt hatten, zeigten, dass die Änderungen zwar die Bedienung stark vereinfachten, jedoch noch weitere Änderungen nötig sind, um diese intuitiv und einfach zu gestalten. Die vorgebrachten Mängel umfassten:

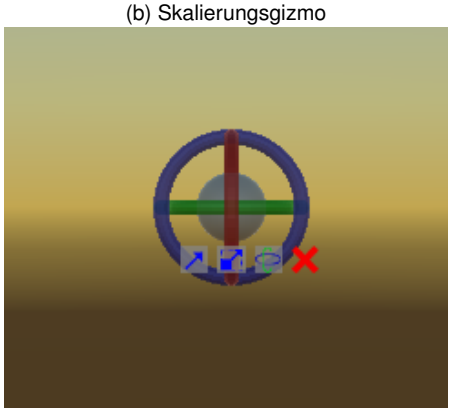
1) Bei der Auswahl einer Achse gibt es keine Rückmeldung, ob dies erfolgreich war. Diese kann fehlschlagen, wenn der Nutzer die Geste zu schnell oder zu langsam durchführt. Dieses Problem wird verschärft, da es scheinbar eine spürbare Zeitspanne gibt, die nötig ist, damit die HoloLens zwei aufeinanderfolgende Gesten als unterschiedlich wahrnimmt. Diese Beobachtung basiert bis jetzt jedoch nur auf Erfahrungswerten und wurde noch nicht empirisch untersucht. Es zeigte sich, dass der Nutzer oft versuchte eine Achse anzuwählen und anschließend eine Bewegung auf ihr ausführte, jedoch nichts passierte und der Nutzer dabei keine Rückmeldung bekam, ob er etwas beim Auswählen oder Bewegen falsch machte.
2) Es wurde der Wunsch vorgebracht, dass bei der Manipulation einer Achse die Bewegung nicht mehr auf dem Handle der Achse ausgeführt werden muss und stattdessen überall im Raum stattfinden kann.



(a) Translationsgizmo



(b) Skalierungsgizmo



(c) Rotationsgizmo

Abbildung 11. Gizmos der zweiten Version

3) Es wurde vorgeschlagen nach der Auswahl einer Achse die nicht ausgewählten auszublenden, da diese sich, insbesondere beim Gizmo zur Rotation, sich als hinderlich erwiesen.

Aufgrund dieser Erfahrungen ziehen wir als erstes Fazit, dass die Besonderheiten, die sich aus der Steuerung von AR/MR Anwendungen durch Gesten ergeben, als ein zentraler Aspekt beim Entwurf der Bedienkonzepte zu betrachten sind. Die erwähnte Ungenauigkeit der Eingabemöglichkeiten, beispielsweise der Selektion von Objekten durch die Blickrichtung, erfordert, dass die Notwendigkeit präziser Eingaben durch Gesten vermieden werden sollte. Dies kann entweder durch entsprechend großzügig dimensionierte UI Elemente oder softwareseitige Hilfestellungen, wie Snapping oder Gravitationsfelder, geschehen. Bei Letzteren ist jedoch zu bedenken, dass derartige Dinge bei erfahrenen Nutzern für Frustration sorgen können, da diese Hilfestellungen in gewissen Maß ihre Freiheit einschränken.
Weiterhin ist durch die momentane geringe Verbreitung von AR/MR Anwendungen und die nicht vorhandene Standardisierung ihrer Bedienung davon auszugehen, dass die meisten Nutzer auf keinerlei oder nur wenig Erfahrung zurückgreifen können und bei der Verwendung von AR/MR eigenen Konzepten, wie Gestenwie Gesten, dem Nutzer Rückmeldung über fehlerhafte oder erfolgreiche Eingaben gegeben werden müssen.

## 9.6. Texturing

Bereits zu Beginn der Entwicklung wurde schnell ersichtlich, dass die HoloLens gewisse Einschränkungen (hauptsächlich im Bezug auf leistungsschwache Hardware, drahtlose Verbindungen) mitbringt. Dadurch wurde es notwendig, dass das Team immer eine Optimierung des Laufzeitverhaltens bei der Umsetzung der Anforderungen im Hinterkopf behielt. Schnell stellte sich dabei das Laden der Assets und Materials vom AssetServer auf die HoloLens als sehr teure Operation heraus. Deshalb sollte durch das UI-Team eine Lösung gefunden werden, die es ermöglicht auf redundante Informationen bei der Übertragung der Assets zu verzichten.

**9.6.1. Technischer Hintergrund.** Um die Umsetzung im Detail nachvollziehen zu können zunächst ein kurzer Überblick zu Materials, Shaders und Textures in Unity. Alle Drei hängen eng zusammen.

- **Materials** definieren wie ein Oberfläche gerendert werden soll einschließlich der Texturen, die verwendet werden, Farbtönen, Rasterinformationen etc. Die verfügbaren Optionen für ein Material hängen von der Wahl des Shaders ab.
- **Shaders** sind kleine Skripte, die mathematischen Berechnungen und Algorithmen für die Farbberechnung der einzelnen Pixel, in Abhängigkeit vom Lichteinfall und der Materialkonfiguration, beinhalten.
- **Textures** sind bitmap Bilder. Ein Material kann Referenzen zu Texturen enthalten, die der Shader für diese Textur während der Berechnung der Oberfläche verwenden kann.

Ein Material spezifiziert einen Shader, der verwendet wird, dieser wiederum bestimmt welche Optionen dem Material zur Verfügung stehen. Ein Shader spezifiziert einen oder mehrere Texturen, die verwendet werden. [6]

**9.6.2. MaterialManager.** Der MaterialManager bietet dem Benutzer der Anwendung die Möglichkeit das Material eines sich in einer Szene befindlichen Objekts dynamisch auszutauschen und somit eine hohe Wiederverwendung der auf die HoloLens geladenen (und damit für die Anwendung verfügbaren) Assets zu garantieren. Diese Trennung bietet nicht nur den Vorteil die Laufzeit der Applikation und die Ladezeiten zu verbessern, sondern sie bietet auch die Möglichkeit für den Benutzer schnell das Aussehen eines Objekts anzupassen. Statt ein vorhandenes Objekt (z.B. einen Holztisch) zu löschen und dann ein neues Objekt (z.B. einen Plastiktisch) zu platzieren, kann einfach das Material geändert werden. Somit bleiben zudem Position, Ausrichtung und andere laufzeitbezogenenen Informationen erhalten, was die Softwareergonomie deutlich erhöht.


Abbildung 12. Beispiel einer Materialpalette

Das UI ist idealisiert in Abbildung 12 dargestellt. Ein einheitliches Bedienkonzept ist in Planung, aber derzeit noch nicht umgesetzt.

## 9.7. UI Ausblick

Natürlich gibt es für die UI-Gruppe weitere Aufgaben, die in der Zukunft zu tun sind. Dieser Abschnitt diskutiert Verbesserungen, neue Features und allgemeine Ziele, die im nächsten Semester implementiert werden sollen.

**9.7.1. Bugfixing.** Es sind verschiedene Probleme beim implementieren des UI aufgetreten. Diese müssen offensichtlich beseitigt werden. Einige dieser Probleme sind folgende:

- Kamerawackeln im Objekt
  Aktuell fängt die Kamera an stark zu wackeln, sobald sich die HoloLens in einem virtuellen Objekt

befindet. Nach aktuellem Stand ist es noch nicht bekannt, ob dieses Problem durch Fehler bei der Implementierung unsererseits entsteht oder auf eine fehlerhafte Implementierung seitens Microsoft zurückzuführen ist.

- Xbox Controller Mapping
  Ein essentieller Teil des UI ist der Input über einen Xbox One Controller. Jedoch gibt es hier Probleme mit dem Abbilden der einzelnen Knöpfe zu den Treiber-Funktionen. Es wurde schon viel Zeit in die Auflösung dieses Problems gesteckt, jedoch konnte noch keine Lösung gefunden werden, die das Problem vollständig behebt. Daher wurde dieser Input in der aktuellen Form des Programms in einer instabilen und etwas unkonventionellen Art implementiert und sollte somit in Zukunft verbessert werden, da dies sonst zu weiteren Problemen in anderen Teilen des Systems führen könnte.
  Dieses Problem wurde an Microsoft gemeldet damit man gemeinsam eine Lösung dafür finden kann [27].

**9.7.2. Hinzufügen eines Hauptmenüs.** Neben des aktuell vorhandenen "Asset Stores" soll es ein weiteres Menüfenster geben. Dieses soll das Hauptmenü des Systems sein. Somit sollen die wichtigsten Dinge in diesem Menü einstellbar sein, wie zum Beispiel die Modifizierung der IP des Servers, damit man zur Laufzeit verschiedene Asset Server auswählen kann. Dies bringt eine große Bereicherung in der Komfortabilität beim Testen des Programms, da das Programm dann nicht neu gebaut werden muss, um sich zu einem anderen Server zu verbinden.
Andere Dinge, die im Hauptmenü einstellbar sein sollen sind für Grafikanwendungen typische Einstellungen wie etwa die Skalierungen der Objekte. Ebenso sollen Sensibilitäten der Input-Geräte variierbar sein. Zuletzt kann es nützlich sein, dass Einstellungen bezüglich der Audio-Outputs veränderbar werden.

**9.7.3. Dynamische Vorschaubilder.** Nach Kundenanforderung ist es gewünscht, dass für jedes Objekt im Asset-Store ein Vorschaubild angezeigt wird. Dies fördert die Benutzerfreundlichkeit um ein großes Maß und sollte somit in der nahen Zukunft implementiert werden.
Der Grundstein hierfür wurde bereits gesetzt, indem die Dateistruktur wie sie vom Server an die HoloLens gesendet wird schon definiert wurde und somit nur noch implementiert werden muss.

**9.7.4. Spatial Sound.** Spatial Sound ist ein interessanter Punkt beim UI eines VR bzw. AR Systems. Das räumliche Hören kann die Benutzerfreundlichkeit stark erhöhen. Zum Beispiel können Fenster, die der Benutzer lange nicht mehr aktiv nutzte, Töne von sich geben, damit er diese schneller wiederfindet.
Dieses Feature muss sich jedoch nicht nur auf Fenster beschränken. Es kann auch dafür genutzt werden, um andere Objekte in der Szenerie wiederzufinden. Beispielsweise können Objekte, die gesucht werden Töne von sich geben, nachdem der Benutzer z.B.: "Wo sind meine Bäume?" als Voice Input eingibt. Danach können räumliche Geräusche ausgehend vom Baum dem Benutzer helfen, sich im Raum zu orientieren.

**9.7.5. Andere Verbesserungen in der Benutzbarkeit.** Es gibt noch viele andere Aspekte im UI, die bei einer guten Implementierung zur einer deutlichen Verbesserung der Benutzerfreundlichkeit führen. In diesem Abschnitt werden einige dieser Ideen erläutert.

- Spatial Mapping
  Spatial Mapping bedeutet, dass das Programm den Raum der echten Welt auf die virtuelle Ebene abbildet. Dies ermöglicht ein angenehmeres Positionieren von Objekten in der Welt. Objekte können dann nämlich entlang des realen Raums platziert werden. Dies verhindert, dass Objekte beim Verschieben hinter Wänden verschwinden und erleichtert es zum Beispiel Dinge auf einem Tisch zu platzieren.
- Raster zum Platzieren von Objekten
  Eine andere Methode die Platzierung von Objekten zu erleichtern, ist die Platzierung anhand eines Rasters. Dies ist vor allem nützlich, wenn man Objekte sehr präzise platzieren muss.
- Besser aussehende Menüfenster
  Gut aussehende Menüfenster bringen den Vorteil, dass sie dadurch automatisch auch meist übersichtlicher sind. Daher wird es nötig sein die Menüfenster besser zu gestalten.

**9.7.6. Tests durch externe Tester.** Ein Punkt der dieses Semester weniger bedacht wurde, ist das Testen des UI von anderen Personen als den Projektbeteiligten. Dies kann Erfahrungen bringen, wie unerfahrene Personen mit dem UI umgehen. Dort kann man dann eventuell nicht intuitive Dinge erfahren, damit diese dann verbessert werden können. Wir haben uns vorgenommen, das System im nächstem Semester, unter unserer Beobachtung, von anderen Testen zu lassen.

# 10. QA

## 10.1. Testing

**10.1.1. Unity Test Runner.** Um Software erfolgreich testen zu können, werden Unit Tests benötigt. Unity bietet einen Unity Test Runner für das Testen eines Projekts an. Der Test Runner benutzt die NUnit Bibliothek, welches eine Opensource Unit Test Bibliothek für .NET Anwendungen ist. Dabei wird ein C# Testskript erstellt. Der Unity Test Runner kann sowohl im Play Mode als auch im Edit Mode durchgeführt werden. Um den Test Runner in Unity zu starten, muss man in Unity unter Window und anschließend auf Test Runner klicken. In Abbildung 13 sieht man das Fenster und kann dort eigene Tests mit Button *Create EditMode test* erstellen. Dies erstellt Tests im Edit Mode. Um Tests im Play Mode zu erstellen und testen, muss man

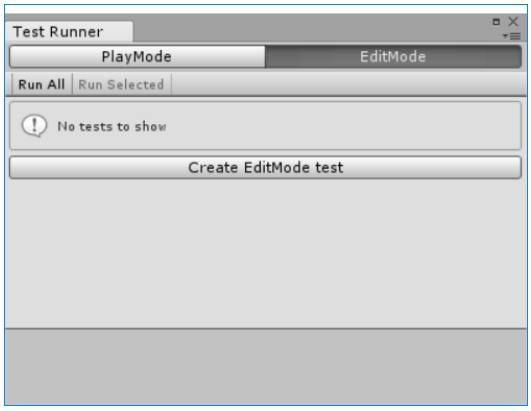



Abbildung 13. Unity Test Runner Fenster ohne Testcases [11]

auf *PlayMode* und anschließend auf den *Enable playmode tests* Button klicken. Mit den Test Runner kann man die Unit Tests ausführen. Außerdem ist es möglich mit der Konsole die Unit Tests auszuführen. Man kann die Unit Tests von der Kommandozeile ausführen. Dadurch kann man diese Befehle einem Skript hinzufügen. Der Sinn dieses Tests ist es, dass man dies als einen Prozess in einer kontinuierlichen Integration hinzufügen kann. Hier ist ein Beispiel für einen Konsolenbefehl:

```
Unity.exe -runTests
-projectPath PATH_TO_YOUR_PROJECT
-testResults C:\temp\results.xml
-testPlatform editmode
```

Argumente für diesen Befehl:

- `runTests` führt die Tests im Projekt durch
- `projectPath`: vorgegebener Projektpfad
- `testResults`: Speicherpfad für Unit Test Ergebnisse als XML-Datei
- `testPlatform editmode`: die Tests werden im Edit Mode durchgeführt

Im Unity Test Runner gibt es viele Möglichkeiten mit NUnit zu testen. Wie im NUnit können zwei Fließkommawerte (float) verglichen werden. Außerdem ist es möglich Farbwerte eines Objekts mit einem entsprechenden Farbwert zu vergleichen. Im Unity Test Runner können zwei Quaternionen oder Vektoren mit zwei, drei und vier Werten getestet werden.

**Szenenbasierte Tests**
Außerdem ist es in Unity möglich zu überprüfen, ob in einer geladenen Szene ein Objekt wie z.B. ein Würfel existiert. Dies wird mit Edit Mode getestet. Im Play Mode wird z.B. überprüft, ob ein erstellter Würfel in einer geladenen Szene den gleichen Shader hat, wie erwartet.

**MonoBehaviourTest**
Mit MonoBehaviourTest können z.B. Timeouts von bestimmten Interaktionen (die im Code als `MonoBehaviour` gekennzeichnet sind) überprüft werden. Dabei soll man bei diesem Test festlegen, wie lange dieser Test Case durchgeführt wird. In einer Schnittstelle der IMonoBehaviourTest werden die Bedingungen gesetzt, wann der Test fertig ist. Sobald die festgelegte Zeit überschritten ist, ist der Test fehlgeschlagen. Ansonsten ist er erfolgreich ausgeführt worden [10].

**UnityTestAttribute**
Im Play Mode kann ein Test in einer Koroutine laufen. Dabei kann man z.B. überprüfen, ob die Position eines Objekts in Unity verändert wurde. Im Edit Mode überprüft man, ob die Durchführung des Tests erfolgreich war und ob es keine Fehlermeldungen gibt.

**UnityPlattformAttribute**
Im UnitTestRunner ist es möglich zu überprüfen, ob man die erwartete Plattform verwendet.
Wie im NUnit Test, gibt es in Unity Tests, um die Log Nachrichten zu überprüfen. Dies wird mit LogAssert realisiert [11] [12] [13].
Die Unit Tests müssen regelmäßig gepflegt und auf aktuellem Stand gehalten werden. Alle Testskripte sollen in einem separaten Ordner liegen. Außerdem sollen die Ergebnisse der Tests in einem expliziten Ordner liegen und zur Verfügung gestellt werden. Dies wird durch die kontinuierliche Integration möglich.

**10.1.2. Unium.** Unium ist eine Bibliothek für automatisierte Tests. Sie enthält einen Webserver in Debug Builds eines Projekts, welches eine Schnittstelle an einer Szene bietet und eine Abfragesprache hat, die eine Szene steuern kann. Dadurch kann man ein bevorzugtes Test Framework für den automatisierten Test benutzen. Diese Bibliothek findet man in dem aufgeführten Github Repository [14], das man sich klonen kann. Nachdem man dies geklont hat, kann man es in ein eigenes Unity Projekt einbinden. Dabei muss folgendes gemacht werden:

1) Der Ordner (Assets/unium) in der Bibliothek muss in den eigenen Asset Ordner kopiert werden.
2) Ein leeres Objekt muss in der Szene erzeugt werden und anschließend wird die UniumComponent angehängt.
3) Die Lauffähigkeit der Szene durch Aufruf von http://localhost:8342/about überprüfen.

In der Bibliothek spielt die Abfragesprache GQL (Game Query Language) eine wichtige Rolle. Sie kann Daten der Szene aufrufen, Variablen setzen und Funktionen aufrufen. Außerdem ist es möglich die Werte der Objekte in einer Szene zu verändern z.B. können die Positionswerte ersetzt werden. Es können nur primitive Typen und Vektoren mit drei Werten verändert werden. Im Folgenden sieht man ein Beispiel für die Abfragesprache GQL:

http://localhost:8342/about/q/scene/Some/Object.

Component.SomeFunction(a,b,c)

Dieses Beispiel ruft die Funktion SomeFunction mit Argumenten in Bezug zu Object auf. Es ist außerdem ideal, um Visualisierung mit Unity zu debuggen. Mit Hilfe von JSON Daten kann man die Positionen der Objekte anzeigen lassen [15].
Diese Bibliothek hat folgende Vorteile:

- Es ist ein cross-platform Testpaket
- Man kann damit ein Automatisierungsskript schreiben und interagiert mit den Applikationen
- Ist für alle Plattformen geeignet
- Es kann in CI leicht integriert werden

[16]

## 10.2. Continuous Integration

Continuous Integration (CI) ist ein Softwareentwicklungsprozess, welcher bei jedem Push bzw. Hochladen den Codes im Softwareprojekt kompiliert und testet. Dadurch lassen sich Fehler schneller identifizieren und beheben. Der Sinn von CI ist es, dass die Entwickler frühzeitig und regelmäßig Änderungen in die Versionsverwaltung einchecken können. Die Änderungen im Softwareprojekt müssen funktionsfähig sein. Dadurch sollten die Integrationsprobleme in der gesamten Applikation überprüft werden können [17] [18].
Im Projekt wird GitLab als Versionsverwaltung benutzt. Somit bietet sich GitLab CI für die kontinuierliche Integration an. Diese verwendet eine YAML-Datei[3] (.gitlab-ci.yml) für die Konfiguration der Schritte für die CI, die bei jedem Push durchgeführt wird. Die Ergebnisse werden in Form von Reports graphisch dargestellt. Die YAML-Datei muss in dem Root-Verzeichnis des Projekts in GitLab sein. Ein GitLab Runner muss installiert und im GitLab Projekt registriert werden, damit der CI Prozess läuft und anschließend Ergebnisse an GitLab sendet. Die Anleitung für die Installation und Registrierung findet man in [20].

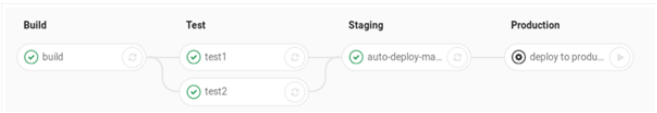



Abbildung 14. Beispiel einer Pipeline mit Jobs und Abschnitten (Stages) [21]

Für jeden Commit, der gepusht wird, generiert der CI automatisch eine Pipeline. In Abbildung 14 sieht man ein Beispiel, wie eine Pipeline aufgebaut ist. Dort wird das Projekt im ersten Abschnitt kompiliert (build). Nachdem es erfolgreich kompiliert wurde, wird der Abschnitt Test ausgeführt. Dort werden *test1* und *test2*, die als Jobs definiert sind, parallel durchlaufen. In diesem Abschnitt können Unit Tests vorhanden sein. Anschließend wird im nächsten Abschnitt *Staging* und danach der Abschnitt *Production* durchgeführt. Jeder der Abschnitte beinhaltet Jobs [22].
Ein Job ist eine Menge von Befehlen, die ausgeführt werden und jeweils einen eindeutigen Namen haben. Diese werden ebenfalls in der Konfigurationsdatei (`.gitlab-ci.yml`) definiert. Jobs können Artefakte erzeugen, welche man herunterladen kann. Somit kann man die Artefakte herunterladen und auf den Geräten testen. Artefakte können beliebige Dateien sein (Text, Bild, APK etc.). Jobs können einen der folgenden Zustände haben:

- Ignoriert d.h. der Job wird nicht ausgeführt und nicht berücksichtigt
- Laufend d.h. der Job ist am Ausführen
- Fehlgeschlagen d.h. der Job konnte erfolgreich ausführen
- Erfolgreich ausgeführt (wie in Abbildung 14 durch grünes Häkchen zu sehen)

Mehrere Jobs in einem Abschnitt können während der Durchführung der Pipeline parallel durchlaufen werden. Sobald der Abschnitt erfolgreich durchgeführt wurde, wird der nächste Abschnitt durchgeführt bzw. die Pipeline wird als erfolgreiche Durchführung markiert. Falls ein Job in einem Abschnitt fehlschlägt, schlägt auch die Pipeline fehl [23]. Jede Pipeline hat eine Menge von Artefakten, die durch die Jobs erzeugt bzw. erstellt wurden. So kann man bei einem Merge Request in der ausgeführten Pipeline die Artefakte herunterladen und testen, ob die Implementierung ordnungsgemäß funktioniert [24] [25] [26].
Im Bezug auf das HoloLens Projekt können die definierten Unit Tests über Unity als Job und in Teilen eines CI Prozesses verwendet werden. Außerdem kann man die Unium Bibliothek für die kontinuierliche Integration verwenden. Für die Einführung von Continuous Integration können folgende Jobs und Aufteilung der Abschnitte (Stage) in einer Pipeline definiert werden:

1) BUILD: Build
2) TEST: Unit Test durch Konsolenbefehl, Unium
3) STAGING: zur Verfügung stellen von kompilierten Dateien

Folgende Artefakte würden durch CI erzeugt:

- UWP Datei: Erzeugt durch Build
- Testdatei: Ergebnisse der Unit Tests in Form von XML Datei

## 10.3. Logging

Logging hilft über alle Phasen von der Entwicklung bis in den Betrieb bei der Identifikation und Rückverfolgung von Fehlern; und auch für das Monitoring lässt es sich verwenden. Noch besser ist ein zentrales Logging: Es bringt Daten- und Kontrollflüsse über mehrere Schichten, Knoten

3. YAML (YAML Ain't Markup Language) ist ein vereinfachter Standard zur Datenserialisierung [19].

und Systemkomponenten hinweg in einen Zusammenhang und eine homogene Struktur. Deshalb wurde bereits zu Beginn des Projekts beschlossen, dass im Gegensatz zu den üblichen Console-Outs und verstreuten Log-Dateien eine professionelle Lösung benötigt wird, damit eine gute Qualität gewährleistet werden kann.

Windows 10, speziell die von Microsoft entwickelte Universal Windows Platform, auf der die Applikation aufsetzt, bietet dafür bereits einen Grundstock an Funktionalität. Für die Entwicklung der LoggingLibrary wurde diese erweitert um so eine ganzheitliche Lösung für das Projekt bereitzustellen. Das Prinzip des LoggingChannels ist in Abbildung 15 skizziert. Dabei agiert der LoggingChannel als zentraler Nachrichten-Bus, auf dem Events übertragen werden. Applikationskomponenten können Sources oder Sinks von Events sein, also Events auf den Bus schreiben oder von diesem Lesen (ähnlich des Publish-Subscribe Musters). Events wiederum ist der Überbegriff für die Nachrichten, die am Bus anliegen. Inhalt kann prinzipiell von einem simplen Logtext bis zu einer serialisierten Klasse alles sein, da Events immer als Paar von Key (Identifier) und Text (Payload) erstellt werden. Zur besseren Veranschaulichung ein paar Beispiele: Die HoloLens veröffentlicht als Source in regelmäßigen Abständen ihre Temperaturdaten auf den LoggingChannel, zwei Komponenten agieren als Sink. Eine empfängt die Daten, wertet sie aus und löst einen Alarm aus, falls die Temperatur zu hoch steigt. Die andere Komponente empfängt von vielen Sources die Daten und schreibt diese lediglich gesammelt in eine Datei. Im bestimmten Fällen können Komponenten auch gleichermaßen Sink und Source sein.

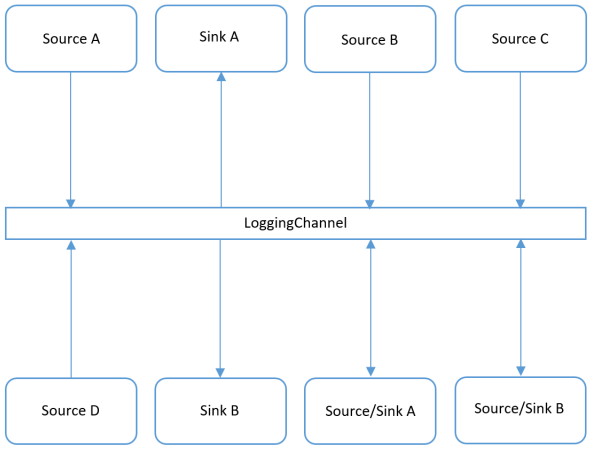



Abbildung 15. Übersicht des Prinzips des LoggingChannel

## 11. Abschlussbetrachtung

Trotz anfänglicher Startschwierigkeiten hat sich im Laufe des Semesters ein homogenes Team gefunden, welches durch kontinuierliches und fokussiertes Vorgehen die Erwartungen der Stakeholder bei weitem übertreffen konnte. Da der Großteil der Entwickler bereits Interesse an einer weiteren Teilnahme bekundet hat, sind die weiteren Ziele in realistischer Reichweite. Letztendlich konnten alle Projektteilnehmer (Stakeholder, Product Owner und Entwicklerteam) das Projekt, aufgrund der Durchführung und Erreichung aller Ziele und mehr, als großartigen Erfolg verbuchen.

## Literatur


[1] International Organization for Standardization, *ISO 21500*, https://www.iso.org/standard/50003.html, International Organization for Standardization, 2012.

[2] Wikipedia, *PDCA*, https://en.wikipedia.org/wiki/PDCA, Wikimedia Foundation Inc., 2018.

[3] Microsoft, *Homepage*, https://developer.microsoft.com/en-us/windows/mixed-reality/gestures, Microsoft, 2018.

[4] https://github.com/CaptainHillman/UnityTools

[5] https://az835927.vo.msecnd.net/sites/mixed-reality/Resources/images/spectatorView.png

[6] https://docs.unity3d.com/Manual/Shaders.html

[7] https://docs.unity3d.com/Manual/UNetConcepts.html

[8] https://docs.unity3d.com/Manual/UNetSpawning.html

[9] https://docs.unity3d.com/Manual/UNetPlayers.html

[10] https://gist.github.com/ryo-ma/db5ee4bf9945ec92748c9596639f3fa7

[11] https://docs.unity3d.com/Manual/testing-editortestsrunner.html

[12] http://www.invisiblerock.com/unity-test-runner/

[13] https://docs.unity3d.com/Manual/PlaymodeTestFramework.html

[14] https://github.com/gwaredd/unium

[15] https://github.com/gwaredd/unium/blob/master/unium.pdf

[16] https://assetstore.unity.com/packages/tools/unium-automated-test-tools-95998

[17] http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/Werkzeuge/Jenkins/#11

[18] https://about.gitlab.com/2016/08/05/continuous-integration-delivery-and-deployment-

[19] http://www.yaml.org/

[20] GitLab Runner, https://docs.gitlab.com/runner/

[21] https://docs.gitlab.com/ee/ci/img/pipelines.png

[22] https://docs.gitlab.com/ee/ci/pipelines.html#pipelines

[23] https://docs.gitlab.com/ee/ci/yaml/README.html#jobs

[24] https://docs.gitlab.com/ee/user/project/pipelines/job_artifacts.html

[25] https://about.gitlab.com/2018/01/22/a-beginners-guide-to-continuous-integration/

[26] Getting started with GitLab CI/CD, https://docs.gitlab.com/ce/ci/quick_start/README.html

[27] Unser Github Issue im Microsoft Project zum lösen des Xbox Controller Problems, https://github.com/Microsoft/MixedRealityToolkit-Unity/issues/1551

[28] PosiStageNet, An Open Protocol for On-Stage, Live 3D Position Streaming, http://www.posistage.net/

[29] Setting up a Raspberry Pi as an access point in a standalone network, https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md

[30] Json to array of objects in C#, https://answers.unity.com/questions/1190094/json-to-array-of-objects-c.html

[31] Is it possible to communicate (UDP/TCP) with the emulator?, https://forums.hololens.com/discussion/253/is-it-possible-to-communicate-udp-tcp-with-the-emulator

[32] Unity Manual - Platform dependent compilation, https://docs.unity3d.com/Manual/PlatformDependentCompilation.html