UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Entwicklung eines Augmented-Reality-pARcours

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Lea Peuker

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Nils Höhner, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Oktober 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Bachelorarbeit beschreibt die Konzeption, Implementierung und Evaluation einer spielerischen Augmented Reality-Anwendung für mobile Endgeräte. Aufbauend auf dem ARCore SDK wurde das Spiel *pARcours* entwickelt, bei dem der Spieler virtuelle Objekte in der realen Umgebung platzieren kann, um sich so seinen eigenen Parcours aufzubauen. Dieser muss mit einem ebenfalls virtuellen Flugobjekt absolviert werden. Der Schwerpunkt bei der Umsetzung des Spiels lag auf der Interaktion mit den virtuellen Objekten und deren Kollisionserkennung mit realen Oberflächen. Weiterhin wurden verschiedene Eingabemöglichkeiten für das Aufbauen der Parcours und die Steuerung der Flugobjekte untersucht. Durch eine abschließende Evaluation konnten sowohl das Spiel und die verschiedenen Eingabemethoden als auch ARCore in Bezug auf die Entwicklung von Augmented Reality-Anwendungen bewertet werden.

Abstract

This Bachelor thesis describes the conception, implementation and evaluation of a playful augmented reality application for mobile devices. Building on the ARCore SDK, the game *pARcours* was developed, where the player can place virtual objects in the real environment to build their own parcours. This must be flown through with a likewise virtual aircraft. The main focus in the implementation of the game was on the interaction with the virtual objects and the collision detection of these with real surfaces. Furthermore, various input methods for building the parcours and controlling the aircrafts were examined. In a final evaluation both the game and the various input methods could be evaluated, as well as ARCore with regard to the development of augmented reality applications.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AR	Augmented Reality
COM	Concurrent Odometry and Mapping
DOF	Degrees of Freedom
HMD	Head-Mounted Display
IMU	Inertial Measurement Unit
MAR	Mobile Augmented Reality
PnP	Perspective-n-Point
RANSAC	Random Sample Consensus
SDK	Software Development Kit
SIFT	Scale Invariant Feature Transform
SLAM	Simultanous Localization and Mapping
SURF	Speeded Up Robust Features
ToF	Time-of-Flight
TUI	Tangible User Interface
UI	User Interface
VIO	Visual Inertial Odometry

1 Einleitung

Damit der Leser einen Überblick über diese Arbeit erhält, werde ich zunächst die dahinterstehende Motivation darlegen und danach die Zielsetzung und den Inhalt der Arbeit kurz beschreiben.

1.1 Motivation

Der technische Fortschritt von Smartphones und Tablets in den vergangenen Jahren und das steigende Interesse an mobilen Augmented Reality-Anwendungen lässt darauf schließen, dass Augmented Reality auch auf mobilen Geräten zukünftig eine größere Rolle spielen wird. So könnten Augmented Reality-Funktionen auf Smartphones so selbstverständlich werden wie SMS oder Anrufe [26]. Es gibt viele denkbare Anwendungsmöglichkeiten in fast allen Bereichen, welche schon heute teilweise umgesetzt sind. Zum Beispiel können manche Navigations-Anwendungen virtuelle Informationen zu den in der Nähe befindlichen Sehenswürdigkeiten, Geschäften oder Restaurants in das Kamerabild einblenden, um den Nutzer besser informieren und navigieren zu können.
Und auch die Verwendung von Augmented Reality im Bereich des Mobile Gamings besitzt viel Potential. So zeigt eine Studie, dass sogar 35% derjenigen Leute, die normalerweise keine Spiele spielen, Interesse an Augmented Reality-Spielen zeigen würden [13]. Da die Nutzung von Augmented Reality für mobile Spiele aber erst in den letzten Jahren größere Verbreitung fand (vor allem dank des Spiels *Pokemon GO*, welches für viele Menschen den ersten Berührungspunkt mit Augmented Reality darstellte [26]), lassen sich zur Zeit kaum Anwendungen finden, die über das einfache Anzeigen virtueller Informationen in der realen Welt hinausgehen. Die Interaktion bleibt dabei meist im Hintergrund. So ist nicht nur die korrekte Darstellung virtueller Objekte in der Realität eine Herausforderung, sondern auch das Entwickeln geeigneter Eingabemethoden für mobile Geräte wie Smartphones, um mit diesen interagieren zu können. An dieser Stelle setzt meine Arbeit an.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Konzeption und Umsetzung einer spielerischen und interaktiven Augmented Reality-Anwendung für mobile Geräte unter Verwendung des Software Development Kits ARCore von Google [11], wobei der Schwerpunkt auf der Kollisionserkennung von virtuellen Objekten mit realen Oberflächen und die Interaktion mit diesen liegen soll. Hierzu werden dem Leser zuerst die notwendigen theoretischen Grundlagen näher gebracht und ARCore auf seine Funktionalität hin analysiert. Auf Basis dessen wird ein geeignetes Konzept für die Anwendung entwi-

ckelt und anschließend umgesetzt. Dabei wird besonders auf die Interaktion mit den virtuellen Objekten und eine für mobile Geräte geeignete Eingabemethode Wert gelegt. Ziel ist es, ein Augmented Reality-Spiel zu entwickeln, bei dem der Spieler nicht nur virtuelle Informationen in die reale Welt einblenden, sondern auch mit diesen interagieren kann. Zum Schluss soll die entwickelte Anwendung mit Hilfe eines Nutzungstests evaluiert werden, wodurch die verschiedenen Eingabemethoden und die Tauglichkeit von ARCore in Bezug auf die Erstellung von augmentierten Anwendungen bewertet werden können.

1.3 Inhalt der Arbeit

Zunächst werden in Kapitel 2 die theoretischen Grundlagen erläutert, die für diese Arbeit von Bedeutung sind. Dazu zählen unter anderem die Definition des Begriffs Augmented Reality und die verschiedenen Teilaspekte des Trackings sowie die Besonderheiten von Augmented Reality im mobilen Anwendungsbereich. In Kapitel 3 werden die Grundkonzepte und Funktionalitäten von ARCore näher beleuchtet und einige Anwendungsbeispiele für die Verwendung von ARCore vorgestellt. Danach werden in Kapitel 4 Ziele und Anforderungen definiert, worauf aufbauend ein Konzept für die zu entwickelnde Anwendung erstellt wird. Die Umsetzung dieses Konzepts und die Implementierung der einzelnen Spielkomponenten wird in Kapitel 5 beschrieben. In Kapitel 6 wird daraufhin eine Nutzungsstudie durchgeführt, sodass die fertige Anwendung evaluiert werden kann. In Kapitel 7 wird schließlich eine Bewertung der Tauglichkeit von ARCore für die Entwicklung von Augmented Reality-Anwendungen anhand der Testergebnisse vorgenommen. Weiterhin werden Optimierungs- und Erweiterungsmöglichkeiten für die entwickelte Anwendung aufgezeigt und ein abschließendes Fazit gezogen.

2 Grundlagen

In diesem Kapitel gebe ich eine Einführung in die Grundlagen von Augmented Reality und erkläre dabei die wichtigsten Begriffe und Komponenten. Weiterhin gebe ich einen Einblick in den Bereich Mobile Augmented Reality und die Software, welche für die spätere Umsetzung der Anwendung relevant sein wird.

2.1 Begriffliche und theoretische Grundlagen

Im Folgenden werde ich die für Augmented Reality relevanten Begriffe definieren und die damit zusammenhängenden theoretischen Grundlagen erläutern, um eine Basis für das Verständnis dieser Arbeit zu schaffen.

2.1.1 Augmented Reality

Mit dem Begriff Augmented Reality (AR), was soviel bedeuted wie „erweiterte Realität“, bezeichnet man im Allgemeinen die Anreicherung der Realität durch künstliche virtuelle Inhalte, was zu einer Verschmelzung der Realität mit der Virtualität führt [36]. Im Gegensatz zur Virtual Reality, in welcher sich der Nutzer komplett in einer immersiven, virtuellen Umgebung befindet und somit von der realen Welt abgegrenzt ist, wird in AR die vorhandene Umgebung mit einer virtuellen Realität ergänzt.

Der Übergang zwischen realer und virtueller Umgebung wird in dem von Milgram et al. definierten *Reality-Virtuality Continuum* [49], auch *Mixed-Reality-Kontinuum* genannt, beschrieben (Abbildung 1). Auf der linken Seite dieses Spektrums befindet sich die reale Welt, die sich ausschließlich aus realen Objekten zusammensetzt, sei es nun bei der Betrachtung einer realen Szene durch eine reale Person oder durch ein Medium wie zum Beispiel ein Smartphone-Display. Das rechte Ende hingegen definiert Umgebungen, die nur aus virtuellen Objekten bestehen, wie es bei Computerspielen und -simulationen der Fall ist, und wird *Virtual Reality* (VR) ge-

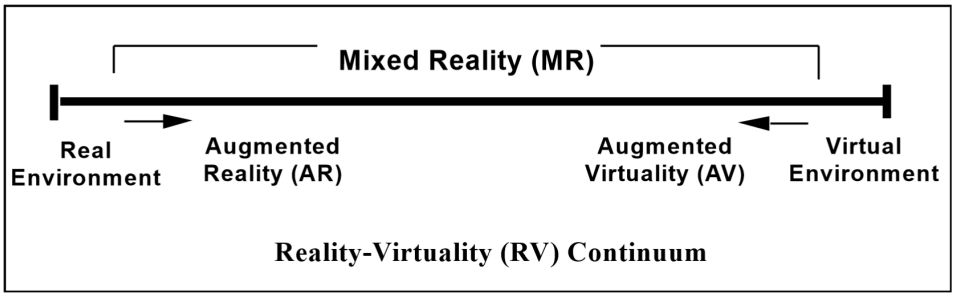


Abbildung 1: Das Reality-Virtuality Continuum nach Milgram et al. [49]

nannt. Zwischen dieser komplett virtuellen und der komplett realen Umgebung erstreckt sich das Kontinuum der „Gemischten Realität“ (*Mixed Reality*), in welcher reale und virtuelle Objekte in einer Darstellung kombiniert werden, je nachdem, wie weit man sich an einem der beiden Enden des Spektrums befindet. Bewegt man sich auf diesem von links nach rechts, so nimmt der Anteil der Realität kontinuierlich ab, während der Anteil der Virtualität zunimmt. AR ist somit in dem Bereich einzuordnen, in dem der Anteil an Realität überwiegt, jedoch ohne dass die Umgebung ausschließlich real ist. Überwiegt der Anteil an Virtualität, so wird das *Augmented Virtuality* (AV) genannt.

Auch wenn der Begriff Augmented Reality in der Literatur nicht einheitlich definiert ist, wird doch meist auf die Definition von Azuma [29] verwiesen. Dieser legt für Augmented Reality die folgenden drei Charakteristika fest:

- Kombination der realen Umgebung mit der virtuellen Realität
- Interaktion in Echtzeit
- Dreidimensionaler Bezug virtueller und realer Objekte

AR ist somit eine interaktive Überlagerung der realen Umgebung mit virtuellen Objekten, welche perspektivisch korrekt und in Echtzeit eingeblendet werden und in der die Objekte (geometrisch) registriert sind. Im Allgemeinen lässt sich eine AR-Pipeline in fünf Schritte gliedern [36]:

1. Videoaufnahme
2. Tracking
3. Registrierung
4. Rendering
5. Visuelle Ausgabe

Zur Bestimmung der Kamerapose wird zunächst ein Videobild oder eine Videosequenz der Umgebung aus der Sicht des Nutzers aufgenommen. Wichtig hierbei ist, dass die Kamera entsprechend kalibriert wurde (siehe auch [61]), um die Fehlerquote beim Tracking gering zu halten. Mit Hilfe des Kamerabildes werden dann die reale Umgebung sowie die darin befindlichen Objekte erfasst, um somit den Blickwinkel des Betrachters und/oder die Lage von Markern (Abschnitt 2.1.2) im Raum möglichst exakt und in Echtzeit erkennen und verfolgen zu können. Danach werden die virtuelle Szene und die reale Umgebung aufeinander registriert, damit die virtuellen Objekte immer perspektivisch korrekt in die Realität eingefügt

werden können. Zum Schluss wird das aufgenommene Videobild mit den virtuellen Inhalten überlagert, sodass diese in einem einzigen Bild dargestellt und auf dem entsprechenden Display ausgegeben werden können. Eine Software, die die reale Umgebung erfasst und anschließend um virtuelle Objekte ergänzt, wird *Tracking Software* oder *Tracker* genannt.
In dieser Pipeline sind vor allem das *Tracking* (Abschnitt 2.1.2), also die Berechnung bzw. Schätzung der Position und Orientierung des Betrachters (bzw. der Kamera), sowie die *Registrierung* (Abschnitt 2.1.3) und das *Rendering* (Abschnitt 2.1.4), d. h. die korrekte Darstellung der virtuellen Inhalte aus der mittels des Trackings berechneten Perspektive, wichtig, auf welche ich im weiteren Verlauf noch genauer eingehen werde.

Anwendung findet AR in vielen Bereichen: In der Medizin können schwierige Operationen leichter durchgeführt werden, wenn im Körper liegende, nicht sichtbare Elemente mittels AR dargestellt werden. Im Wartungsbereich ist es möglich, Arbeiten durch virtuell eingeblendete Instruktionen und Zusatzinformationen effizienter zu erledigen und auch im Tourismus und in der Navigation wird AR verwendet, um zum Beispiel zerstörte historische Gebäude darzustellen oder um sich besser zurechtzufinden. Auch im Bereich der Unterhaltungsindustrie gibt es mittlerweile einige AR-Anwendungen. Ein Beispiel hierfür ist das Mobile Game *Pokemon GO* des Softwareunternehmens Niantic aus dem Jahr 2016, in welchem die Spieler in ihrer realen Umgebung virtuellen Wesen (genannt Pokemon) begegnen und diese einfangen und sammeln können (Abbildung 2) [18].

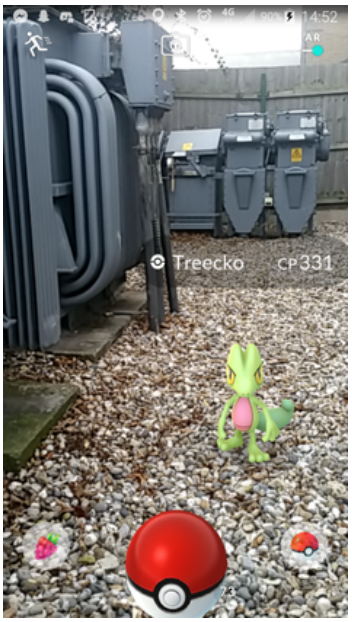


Abbildung 2: Verwendung von AR im Spiel *Pokemon GO*. [18]

2.1.2 Tracking

Im Allgemeinen bezeichnet *Tracking* (dt. verfolgen) die Erkennung und Verfolgung von Objekten [48]. Um virtuelle Objekte korrekt in die reale Umgebung einblenden zu können, muss die AR-Anwendung in der Lage sein, zu jedem Zeitpunkt die Position und Orientierung des Nutzers bzw. der verwendeten Kamera möglichst genau abschätzen zu können. Bewegt sich der Nutzer durch den Raum, muss die Größe, Position und Orientierung der virtuellen Objekte entsprechend angepasst werden. Je genauer die Bestimmung der Pose ist, desto besser können die Objekte in die reale Umgebung eingefügt werden, d. h. desto eher entsteht der Eindruck, als würden sie zur echten Welt gehören. Ist die Schätzung zu ungenau, können die Objekte deplatziert wirken, wobei sich dieser Eindruck noch verschlimmern kann, je weiter sich der Nutzer durch den Raum bewegt.
Grundsätzlich kann zwischen zwei verschiedenen Arten des Trackings unterschieden werden: *Nichtvisuelles* und *visuelles* Tracking [48]. Beim nichtvisuellen Tracking wird die Position und Orientierung des Geräts mit Hilfe von verschiedenen am Gerät befindlichen Sensoren und/oder durch Austauschen bestimmter Signale zwischen einem Empfänger und einem Sender bestimmt (abhängig vom gewählten Tracking-Verfahren). Beim visuellen Tracking wird auf dem Kamerabild mit Hilfe verschiedener Algorithmen nach entsprechend vorgegebenen Merkmalen oder Mustern gesucht, um daraus die Pose der Kamera zu ermitteln. Im Folgenden ist eine Übersicht verschiedener Tracking-Verfahren dargestellt [48, 57].

- **Nichtvisuelles Tracking:**
  - **Kompass**: Über das Magnetfeld der Erde wird die Ausrichtung relativ zu den Erdachsen bestimmt.
  - **GPS**: Die Position des Empfangsgeräts (z. B. Mobiltelefon) wird durch ein satellitengestütztes Ortungssystem berechnet.
  - **Ultraschallsensoren**: Durch die Messung der Laufzeit von Ultraschallwellen zwischen mehreren Sendern und Empfängern wird der Abstand und somit die Position zueinander ermittelt.
  - **Optoelektronische Sensoren**: Die Messung des Abstands zwischen mehreren Sendern und Empfängern erfolgt über optoelektronische Sensoren (nicht sichtbares Licht, z. B. Infrarot).
  - **Trägheitssensoren**: Über verschiedene Arten trägheitsempfindlicher Sensoren wird sowohl die Neigung (Gyroskop) als auch die Bewegung entlang einer geraden Achse (Beschleunigungssensor) gemessen.

- **Visuelles Tracking:**
  - **Markerbasiert**: Das System sucht in dem Videobild nach vorab definierten Referenzen (Markern), um mit deren Hilfe die relative Lage der Kamera oder virtueller Objekte zu bestimmen.
  - **Markerlos**: Das System erkennt innerhalb des Videobildes natürliche Merkmale (zweidimensionale Punkte) und berechnet daraus die Kamerapose.

Im weiteren Verlauf dieses Abschnitts werde ich auf die für diese Arbeit relevanten Tracking-Verfahren genauer eingehen.

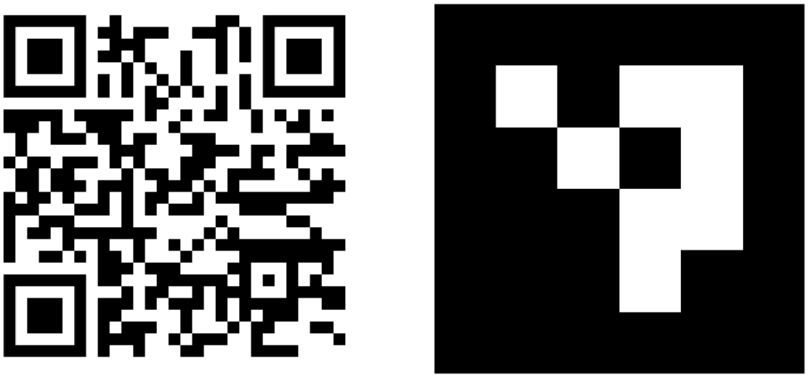
Sensorbasiertes mobiles Orientierungs-Tracking Beim sensorbasierten mobilen Orientierungs-Tracking, welches zu den nichtvisuellen Tracking-Verfahren zählt, wird die Lage des Geräts mit Hilfe verschiedener Sensoren bestimmt. Diese sind heutzutage standardmäßig in so gut wie allen mobilen Endgeräten wie Smartphones oder Tablets verbaut, und bestehen üblicherweise aus einer Kombination von drei verschiedenen Sensortypen: *Magnetometer* sowie lineare und rotatorische *Inertialsensoren* (auch Trägheits- oder Beschleunigungssensoren genannt) [36].
Eine Kombination dieser drei Sensorentypen als Einheit wird als *Inertial Measurement Unit* (IMU) bezeichnet (dt. *inertiale Messeinheit*) [58]. In der Regel werden von jedem Sensortyp je drei orthogonal zueinander angeordnete Sensoren verwendet (also insgesamt neun Sensoren) und in einem mikro-elektromechanischem System (MEMS) kombiniert, um die Orientierung bei beliebiger Lage des Endgeräts erfolgreich messen zu können.

Ein Magnetometer, oder auch elektronischer Kompass, kann die Ausrichtung des Magnetfelds der Erde um das Gerät herum elektronisch messen, woraus eine globale Orientierung folgt. Jedoch sind Magnetometer anfällig für Störungen durch lokale magnetische Felder im direkten Umfeld, wie sie zum Beispiel durch andere elektronische Geräte verursacht werden, wodurch die Lagebestimmung verfälscht und im schlimmsten Fall unbrauchbar werden kann. Daher dienen sie vor allem als Ergänzung für die Inertialsensoren. Diese dienen je nach Bauart entweder der Messung linearer Beschleunigung entlang einer Achse (*Beschleunigungssensor*) oder der Winkelbeschleunigung um eine Achse herum (*Drehratensensor*).
Ein Beschleunigungssensor (engl. *accelerometer*) misst die lineare Beschleunigung desjenigen Objekts, an welchem er angebracht ist [57]. Hierbei übt die Bewegung des Geräts, also die Beschleunigung, eine Trägheitskraft auf eine kleine Masse aus, und indem die resultierende Verschiebung entlang der entsprechenden Achse betrachtet wird, kann die Positionsänderung des Geräts (auf dieser Achse) relativ zum Startpunkt bestimmt werden. Allerdings kommt es bei dieser Art der Messung fast immer zu einem Drifteff-

fekt, wodurch Messungen über einen längeren Zeitraum ungenau werden. Deswegen sollte die absolute Position in regelmäßigen Abständen durch andere Sensoren korrigiert werden.
Ein Drehratensensor, oder auch Gyroskop genannt, misst die Rotationsgeschwindigkeit bzw. Winkelbeschleunigung, die daraus resultiert, dass ein Objekt (im Ruhezustand oder mit konstanter Bewegungsgeschwindigkeit) rotiert wird. Dadurch lässt sich bestimmen, um welchen Winkel (realtiv zum Ausgangspunkt) sich ein Objekt in einer bestimmten Zeit gedreht hat, wodurch sich dessen Orientierung ergibt. Auch hier kommt es nach und nach zum Drift.

Durch die Kombination dieser drei Sensortypen in einer IMU kann die Lage des Geräts relativ genau mit bis zu sechs *Freiheitsgraden* (6DOF) bestimmt werden. Als Freiheitsgrade (engl. *degrees of freedom*, DOF) werden voneinander unabhängige Bewegungsmöglichkeiten eines physikalischen Systems bezeichnet [36]. Um die Pose eines (realen oder virtuellen) Objekts im dreidimensionalen Raum zu bestimmen, werden gewöhnlich 6DOF benötigt: 3DOF für die Translation und 3DOF für die Rotation.

Markerbasiertes Tracking Bei Markertracking handelt es sich um ein visuelles Tracking-Verfahren, welches im AR-Bereich weit verbreitet ist. Das Ziel dabei ist es, vordefinierte Marker anhand ihrer Merkmale im Kamerabild zu detektieren und anhand dessen die Position und Orientierung der Kamera bzw. die Lage des Markers relativ zur Kamera zu bestimmen. Ein Marker ist ein zwei- oder dreidimensionales Objekt, welches durch seine Art und Form leicht durch eine Kamera getrackt werden kann [48]. Sie enthalten leicht erkennbare, asymmetrische Muster mit hohem Kontrast (meist schwarz-weiß) und sind in der Regel quadratisch oder rund mit einem komplett weißen oder schwarzen Rand, um sich von der Umgebung besser abheben zu können (Abbildung 3). Im Gegensatz zu farbigen Markern können diese auch unter wechselnden Helligkeitsbedingungen mit Hilfe von Schwellwertverfahren schnell und zuverlässig erkannt werden. Marker werden verwendet, um das Tracking zu initialisieren und zu optimieren, indem sie dabei helfen, die Kamerapose zu bestimmen, aber auch um Informationen zu transportieren. So können zum Beispiel virtuelle Gegenstände auf Markern platziert und an der entsprechenden Position in das Kamerabild eingefügt werden. Um dies zu ermöglichen, müssen vorab die Muster und Größen der verwendeten Marker bekannt sein, damit das System diese im Kamerabild suchen und verfolgen kann. Außerdem müssen diese (zumindest einmal) komplett im Kamerabild mit ausreichender Auflösung und Beleuchtung sichtbar sein, um initial erkannt zu werden. Wurde ein Marker erkannt, kann mit dessen Hilfe die Transformation zwischen Kamera- und lokalem Koordinatensystem des Markers bzw. die rela-



(a) QR-Code Marker[1] (b) ID-Marker[2]

Abbildung 3: Zwei Marker-Beispiele für markerbasiertes Tracking.

tive Lage von Kamera zu Marker und somit die Position und Orientierung der Kamera bestimmt werden.

Es gibt verschiedene markerbasierte Tracking-Ansätze für die unterschiedlichen Markerarten (siehe hierzu [44]), die aber prinzipiell nach dem gleichen Prinzip ablaufen: Zuerst nimmt die Kamera ein Videobild auf, in dem mit Hilfe von Bildverarbeitungsmethoden die Marker detektiert werden. Hierzu wird in der Regel das Eingabebild mittels Schwellwertverfahren in ein Binärbild umgewandelt, welches segmentiert und dann mittels Kantendetektion bearbeitet wird. Daraufhin wird überprüft, ob es sich tatsächlich um einen vordefinierten Marker handelt. Ist dies der Fall, kann er anhand seiner ID (seines Musters) eindeutig identifiziert werden. Die Vorgehensweise Überprüfung und Identifikation der Marker hängt dabei von der jeweiligen Markerart ab. Zuletzt werden die extrinsischen Kameraparameter bestimmt. Dabei handelt es sich um die Position und Orientierung der Kamera in Relation zu dem gefundenen Marker. Dieses Vorgehen wird kontinuierlich wiederholt, damit die virtuellen Informationen stets korrekt in die reale Welt eingeblendet werden können.

Ein großer Vorteil des markerbasierten Trackings ist der geringe Technologieaufwand und die einfache Anwendung. Marker können schnell und einfach erstellt, ausgedruckt und an den entsprechenden Stellen angebracht werden [36]. Allerdings stellt das Platzieren auch einen Nachteil dar, da die Marker nicht an allen Stellen angebracht werden können oder dürfen, oder sie stören im Bild, wenn sie das reale Objekt verdecken. Problematisch ist auch, wenn die Marker teilweise verdeckt werden, zum Beispiel bei der

[1] Erstellt auf `http://chev.me/arucogen/`, Abgerufen am 05.09.2019
[2] Erstellt auf `https://www.qrcode-generator.de/`, Abgerufen am 05.09.2019

Abbildung 4: Im Kamerabild werden visuell eindeutige Merkmale identifiziert und als Feature Points markiert (rechtes Bild). Ecken und Kanten, die einen hohen Kontrast zwischen benachbarten Pixeln vorweisen, eignen sich hierfür am besten. [7]

Interaktion mit diesen, sie zu nah oder zu weit entfernt sind, um richtig erkannt zu werden, oder wenn die Lichtverhältnisse und die Auflösung der Kamera nicht gut genug sind.

Merkmalsbasiertes Tracking Merkmalsbasiertes Tracking kann als Verallgemeinerung des markerbasierten Trackings angesehen werden [36], jedoch wird das Kamerabild statt nach (zuvor bekannten und in der Umgebung angebrachten) Markern nach allgemeinen, natürlich hervorstechenden und visuell eindeutigen Merkmalen (engl. *features*) abgesucht, welche auch als *Feature Points* bezeichnet werden (Abbildung 4). Dabei unterscheidet man zwischen *Dense Matching*, bei dem eine Korrespondenz für jedes einzelne Pixel im Bild gesucht wird, und *Sparse Matching*, wo dies nur für eine kleine, aber ausreichende Anzahl an hervorstechenden Merkmalen im Kamerabild geschieht. Im Folgenden wird nur das für diese Arbeit relevante Sparse Matching betrachtet.

Eine typische Vorgehensweise für merkmalsbasiertes Tracking, bei dem jedes einzelne Kamerabild unabhängig von den anderen nach Feature Points durchsucht wird, besteht aus fünf Schritten [58]:

1. Detektion von Feature Points
2. Erstellung der Deskriptoren
3. Deskriptor Matching

4. Auswahl robuster (valider) Korrespondenzen

5. Bestimmung der Kamerapose

Zuerst wird das Kamerabild mittels Merkmalsdetektion nach Feature Points durchsucht. Diese sind für den menschlichen Betrachter häufig kaum zu erkennen, können aber sowohl schnell als auch zuverlässig in einem Kamerabild durch entsprechende *Merkmalsdetektoren* identifiziert werden. Feature Points sollten eindeutig erkennbar, gut texturiert und nicht Teil einer sich wiederholenden Struktur sein, damit sie verlässlich erkannt werden können. Konnten genügend solcher Merkmale aus dem Kamerabild extrahiert werden, so werden diese auf Basis ihrer individuellen Beschreibung, dem sogenannten *Deskriptor*, mit der vorhandenen Beschreibung der Merkmale der zu verfolgenden 2D- oder 3D-Geometrie verglichen [36], wobei auch die umliegenden Pixel analysiert werden. Ein Deskriptor ist eine Datenstruktur, die dafür geeignet ist, Übereinstimmungen mit dem getrackten Objekt oder anderen Bildern zu finden und wird für jedes einzelne Merkmal erstellt. Er sollte idealerweise für jeden Feature Point einzigartig sein, aber identisch für verschiedene Blickwinkel und Lichtverhältnisse.
Beim Deskriptor Matching werden demnach Korrespondenzen zwischen den 2D Punkten im Kamerabild und den 3D Punkten der realen Welt gesucht. Statistische Ausreißer, also Fehlzuordnungen von Merkmalskorrespondenzen, werden dabei meistens durch die Verwendung eines *RANSAC*-Verfahrens (*RANSAC - Random Sample Consensus*) (Fischler und Bolles 1981) [38] aussortiert. Dies ist ein iterativer Ansatz, bei dem für eine möglichst kleine Menge zufällig ausgewählter Messwerte die Distanz zu dem entsprechenden Modell berechnet und anhand dessen entschieden wird, ob ein bestimmter Messwert das Modell unterstützt (d. h. ob die Distanz kleiner ist als der gewählte Schwellwert). Je größer die Anzahl an Messwerten ist, die das Modell unterstützen, desto wahrscheinlicher ist es, dass die zufällig ausgewählten Werte keine Ausreißer enthielten. Für jede (voneinander unabhängige) Iteration werden diejenigen Messwerte als *Consensus Set* gespeichert, die das jeweilige Modell unterstützen. Zum Schluss wird versucht, das Modell für die Größte so gespeicherte Menge, die im besten Fall keine Ausreißer mehr enthält, zu lösen. Ist dies nicht möglich, ist die Übereinstimmung zu gering und es gibt keine Lösung bzw. passende Zuordnung zu dem Modell.
Wurden genügend korrekte Zuordnungen von Merkmalspunkten im aktuellen Kamerabild zu denen einer vorhandenen Merkmalskarte gefunden (2D-3D Punktkorrespondenzen) (Abbildung 5), so kann auf Basis dieser Zuordnungen die Kamerapose relativ zu den bekannten Merkmalsgruppen berechnet werden. Hierzu eignet sich das Lösen eines *Perspective-n-Point*-Problems (PnP), wie zum Beispiel der *P3P*-Algorithmus, bei dem die

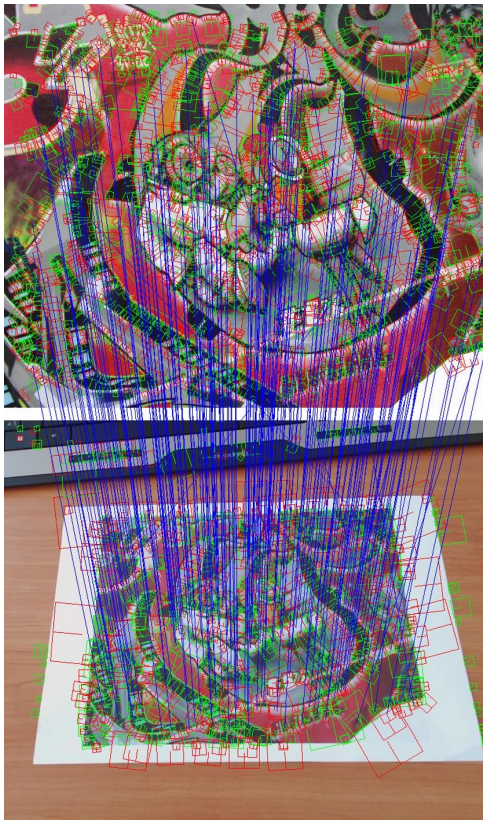


Abbildung 5: Zuordnung von Merkmalspunkten im aktuellen Kamerabild zu denen einer vorhandenen Merkmalskarte. [36]

Kamerapose mit Hilfe von nur drei Punktkorrespondenzen bestimmt werden kann (für Details zu PnP siehe [56]).

Es existieren unterschiedliche Arten von Merkmalsdetektoren, die für das merkmalsbasierte Tracking verwendet werden, welche sich in ihrer Geschwindigkeit und Zuverlässigkeit stark unterscheiden können (und auch nicht alle von ihnen bieten die entsprechenden Deskriptoren) [36]. Zwei bekannte Beispiele für Detektoren sind *SIFT - Scale Invariant Feature Transform* (Lowe 1999, 2004) [46, 47] und *SURF - Speeded Up Robust Features* (Bay et al. 2006) [31].

Ein weiteres Verfahren, welches ursprünglich aus der Robotik stammt, und bei dem zu Anfang weder die eigene Position im Raum noch die Umgebung bekannt sind, ist *SLAM - Simultanous Localization and Mapping* (dt. *simultane Lageschätzung und Kartenerstellung*) (Durrant-Whyte und Bailey, 2006) [37, 30]. Hierbei wird basierend auf der Bewegung der Kamera eine dreidimensionale Karte der Umgebung erstellt und gleichzeitig die

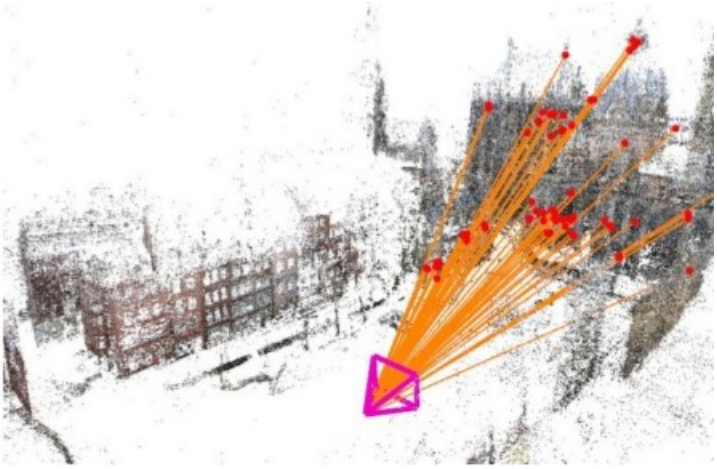


Abbildung 6: Die in den aufeinander folgenden Kamerabildern detektierten Feature Points werden zu Punktwolken zusammengefasst und ergeben eine 3D-Karte der Umgebung. [20]

Position und Orientierung der Kamera basierend auf den in der Umgebung detektierten Merkmalen neu geschätzt. Die Karte wird in der Regel durch sogenannte *Punktwolken* (engl. *pointclouds*) bestehend aus detektierten Feature Points dargestellt (Abbildung 6), wodurch sie sich leicht erweitern lässt. SLAM ist also (auf dem einfachsten Level) ein 3D-Graph von 3D-Punkten, welche durch eine Punktwolke repräsentiert werden, in der jeder Punkt mit einer zu einem Feature Point gehörigen Koordinate korrespondiert [48].

Das Erstellen der Karte geschieht inkrementell: Da die Umgebung zu Beginn nicht bekannt ist, d. h. keine Karte vorhanden ist, definiert die anfängliche Position der Kamera den Koordinatenursprung. Somit ist die absolute Position des Geräts bekannt, von welcher aus die erste Messung der Umgebung erfolgen und direkt in die Karte eingetragen werden kann. Immer nachdem die Kamera bewegt wurde, kann ein neuer Teil der Umgebung eingescannt werden. Dabei wird ein Teil der schon bekannten als auch ein Teil der neuen Umgebung vermessen (vorausgesetzt die Kamera wurde nicht zu schnell oder zu weit bewegt), und aus der Überlappung der bisherigen Karte mit der neuen Messung kann die Bewegung der Kamera und somit die neue absolute Position berechnet werden. Integriert man diese Information in die Karte, kann durch die Wiederholung der Schritte *Merkmalsextraktion* und *Umgebungsabbildung + Posebestimmung* die Karte inkrementell erweitert werden, bis man schließlich eine dreidimensionale Abbildung der kompletten Umgebung besitzt. Die Kamerapose wird also dazu genutzt, die Vermessung der Umgebung zu verbessern, und die eingescannte Umgebung wird dazu genutzt, die Kamerapose besser bestimmen

zu können. Dieses modellfreie Tracking-Verfahren kommt völlig ohne zuvor bekannte Referenzen (Marker oder Modelle) aus und ermöglicht es, eine dem System komplett unbekannte Umgebung zu erkennen und zu tracken, ist jedoch auch mit größerem Rechenaufwand verbunden.

Der Prozess, bei dem eine Sequenz von Kamerabildern auf identische visuelle Informationen überprüft wird, um so die zurückgelegte Distanz zu ermitteln und die aktuelle Lage zu schätzen, wird auch als *visuelle Odometrie* (engl. *visual odometry*) [51] bezeichnet. Sie ist ein Hauptbestandteil visueller SLAM-Verfahren (vSLAM), bei dem die Kamerapose kontinuierlich mit 6DOF relativ zum Startpunkt berechnet wird, indem die Feature Points, welche in dem aktuellen Kamerabild detektiert wurden, mit denen aus dem vorherigen Bild verglichen werden, wodurch sich die Transformationsmatrix zwischen den Bildern und somit die Änderung der Kamerapose ergibt [58]. Das Tracking basiert allein auf der visuellen Eingabe: Weder die Umgebung, noch die physische Bewegung der Kamera müssen bekannt sein. Eine Übersicht verschiedener und aktueller visueller SLAM-Verfahren findet sich in [62].

Ein Vorteil von merkmalsbasiertem Tracking gegenüber markerbasiertem Tracking ist, dass keine Marker in der Umgebung platziert werden müssen und dass es wesentlich robuster gegenüber Störeinflüssen ist (z. B. teilweise Verdeckung), da in der Regel schon ein Teil aller vorhandener Feature Points ausreicht, um das entsprechende Modell zu identifizieren. Allerdings setzt dies auch eine sehr gute Bildauflösung und gute Lichtverhältnisse voraus, da sonst keine Merkmale im Kamerabild erkannt werden können, und das Finden von validen Punktkorrespondenzen und die Berechnung der Pose anhand dieser benötigt einen größeren Rechenaufwand. Für weiteren Einblick in merkmalsbasiertes Tracking und die verschiedenen Verfahren und Algorithmen siehe [41] und [58].

Hybrides Tracking: Visuell-inertiale Odometrie Da die verschiedenen Tracking-Ansätze in unterschiedlichen Szenarien jeweils ihre eigenen Stärken und Schwächen besitzen, wodurch je nach Situation unterschiedlich gute (im schlechtesten Fall nicht verwendbare) Ergebnisse geliefert werden, bietet es sich an, verschiedene Verfahren miteinander zu kombinieren [36]. Ein Beispiel hierfür ist das markerbasierte Tracking: Wird ein Marker kurzzeitig verdeckt oder ist er nicht mehr im Kamerabild, so kann er nicht erkannt werden und es ist nicht mehr möglich, mit dessen Hilfe das Verhältnis virtueller Objekte zur Pose der Kamera zu bestimmen. In diesem Fall empfiehlt es sich, die Lageschätzung auf Basis eines alternativen Tracking-Verfahrens vorzunehmen.

Bei mobilen Systemen, wie zum Beispiel bei Smartphones oder Tablets, wird deswegen häufig auf die integrierten Lagesensoren zurückgegriffen. Liefert das visuelle Tracking nicht mehr genügend Informationen (aufgrund schlechter Lichtverhältnisse etc.), so kann die Transformation zumindest noch in Bezug auf die Lage korrekt bestimmt werden. Somit helfen die inertialen Daten, Ausfälle beim visuellen Tracking zu vermindern, und gleichzeitig grenzen sie den Suchbereich beim Merkmalsabgleich mit der erstellten Umgebungskarte ein, wodurch die Performanz verbessert werden kann. Auf der anderen Seite kann das visuelle dem sensorbasierten Tracking dabei helfen, die durch die Bewegung ermittelte Position und Orientierung des Geräts zu korrigieren, wenn (durch z. B. zu schnelle Bewegungen) ein Drift entsteht [64]. Die Kombination aus visuellem Tracking und der Verwendung von Inertialsensoren wird als *visuell-inertiale Odometrie* (VIO) (engl. *visual-inertial odometry*) [45] bezeichnet.

Time-of-Flight Sensoren Eine weitere Möglichkeit, die Stabilität der Tracking-Ergebnisse zu erhöhen, ist die zusätzliche Verwendung einer *Tiefenkamera* bzw. eines *Time-of-Flight* (ToF) Senors. Diese senden Lichtimpulse (Infrarot) in die Szene und berechnen für jeden Bildpunkt die Zeit, die das Licht braucht, um zum Objekt zu gelangen, von diesem reflektiert zu werden und wieder auf die Kamera zu treffen. Diese Zeit ist proportional zu der zurückgelegten Distanz, wodurch für jeden Bildpunkt und somit für jedes Objekt in der Umgebung die Entfernung zur Kamera berechnet werden kann. Die Tiefeninformationen einer 3D-Kamera, wie sie umgangssprachlich auch genannt wird, können dazu genutzt werden, eine 3D-Rekonstruktion der Umgebung oder eines Modells zu erstellen, sozusagen als Erweiterung der Punktwolken des merkmalsbasierten Trackings. Die Punktwolken, wie sie mittels der Tiefenkamera aufgenommen werden, besitzen eine hohe Punktdichte (siehe *Dense Matching*) und werden in ein Mesh (Polygonnetz) umgewandelt, welches „unsichtbar“ über die Szene gelegt wird, sodass die Mesh-Koordinaten mit den entsprechenden Koordinaten der realen Welt übereinstimmen, was eine bessere Interaktion der virtuellen Objekte mit der realen Umgebung ermöglicht [48]. Das erlaubt die Kollision virtueller Objekte mit reellen Gegenständen und die Verdeckung durch diese.
Wenn VIO durch die Verwendung von Tiefeninformationen eines ToF-Sensors unterstützt wird, kann das Tracking vor allem bei schlechten Lichtverhältnissen und in Umgebungen mit wenig natürlichen Merkmalen enorm verbessert werden. Allerdings funktionieren Tiefenkameras im Freien nicht so gut, da die Sonneneinstrahlung die Messung des reflektierten Lichts stören kann. Außerdem verbraucht die dauerhafte Aussendung von Lichtimpulsen einiges an Strom und die Berechnungen sind relativ aufwändig und kostenintensiv.

2.1.3 Registrierung

Mit *Registrierung* bezeichnet man im Kontext von AR das korrekte Einpassen von künstlichen virtuellen Inhalten in die Realität [36]. Dies geschieht auf Basis der Positions- und Lageschätzung des Trackings: Die Koordinatensysteme der virtuellen Objekte und der realen Umgebung müssen so in Beziehung zueinander gesetzt werden, dass die virtuellen Objekte in der Realität fest verortet (registriert) erscheinen. Werden die Objekte perspektivisch korrekt in die Umgebung eingefügt, spricht man von *geometrischer Registrierung*. Wird die Beleuchtung des Objekts der Umgebung entsprechend angepasst, nennt man das *photometrische Registrierung*.

Geometrische Registrierung Hierbei werden die virtuellen Inhalte auf Basis der durch das Tracking berechneten bzw. geschätzten Transformation zwischen Kamerapose und verfolgtem Objekt mit korrekter Position und Orientierung im Blickfeld des Nutzers (also im aktuellen Kamerabild) dargestellt. Das bedeutet, dass ein (unbewegtes) virtuelles Objekt einen festen Platz in der realen Welt zu haben scheint und dort verbleibt, auch wenn der Betrachter bzw. die Kamera ihren Standpunkt ändert. Es bewegt sich im Verhältnis zur realen Umgebung nicht, da die aktuelle Pose aus Sicht der Kamera für jedes Kamerabild durch die entsprechende Transformation angepasst wird.

Für die Korrektheit der Registrierung spielt die Geschwindigkeit und Qualität des verwendeten Tracking-Verfahrens eine entscheidende Rolle. Ist die Tracking-Rate, also die Anzahl an Tracking-Ergebnissen pro Sekunde, kleiner als die Bildwiederholfrequenz (Framerate), d. h. die Anzahl an auf dem Display dargestellten Bildern pro Sekunde, kann es passieren, dass sich die dargestellten Objekte bei Bewegung der Kamera immer kurzzeitig mit dem Kamerabild bewegen, bevor sie auf ihrer korrekten Position dargestellt werden. Dieser Effekt kann minimiert werden, indem die Bildwiederholfrequenz der Tracking-Rate angepasst wird. Jedoch kann es für den Nutzer genauso störend sein, wenn bei schnellen Bewegungen und einer geringen Framerate das komplette Bild abrupt wechselt, als nur die Position der virtuellen Gegenstände.
Ein weiteres Problem für eine korrekte geometrische Registrierung ist die Latenz, d. h. die Verzögerung zwischen dem Zeitpunkt der Bewegung (der Kamera und/oder des Objekts) und dem Zeitpunkt, an dem die durch die neue Position erhaltene Transformation zur Berechnung der korrekten Darstellung der virtuellen Objekte verwendet werden kann. Latenz entsteht meist durch ein (für die Rechenleistung des Geräts) zu rechenaufwändiges Tracking-Verfahren oder zu lange Signallaufzeiten. Eine hohe Latenz wirkt sich ähnlich aus wie eine zu niedrige Tracking-Rate: Bei Bewegungen der Kamera bewegen sich die Objekte zwar gleichmäßig und ohne „Sprünge“

mit, jedoch nur mit einer gewissen Verzögerung, d. h. mit einem geringen Offset zur eigentlichen, korrekten Position. Erst wenn die Bewegung beendet ist, kann die Registrierung wieder korrekt erfolgen. Je größer dabei der Zeitraum zwischen Schätzung der neuen Position und Orientierung und der Anwendung der Objekttransformation ist, desto größer ist die negative Auswirkung. Durch Zwischenspeichern der Kamerabilder über den Zeitraum der Latenz und Parallelisierung der Berechnungen der Transformationen können latenzbedingte Effekte reduziert werden, da so die Berechnungen auf dem jeweils zugehörigen Kamerabild erfolgen können.

Eine temporäre fehlerhafte Registrierung, ausgelöst durch eine zu niedrige Tracking-Rate und/oder zu hohe Latenz, führt generell dazu, dass der Eindruck zerstört wird, die virtuellen Objekte würden zur realen Welt gehören, und sollte unbedingt vermieden werden, da dies eine grundlegende Eigenschaft von AR darstellt.

Photometrische Registrierung Bei der photometrischen Registrierung werden die virtuellen Objekte korrekt an die Beleuchtungssituation der realen Umgebung angepasst. Das beinhaltet sowohl die (relativ einfache) Anpassung der Beleuchtung des Objekts selber als auch den Einfluss des Objekts auf die reale Umgebung (z. B. der Schattenwurf auf reale Objekte), welcher deutlich schwieriger zu realisieren ist, da hierfür grundlegende Informationen über die Topologie der Realität (Beschaffenheit von Oberflächen, Oberflächennormalen, ect.) zur Verfügung stehen müssen, was oft nicht der Fall ist. Um solche Informationen zu erhalten, könnte man zum Beispiel die Tiefeninformationen eines ToF-Sensors verwenden.
Eine Möglichkeit, die Beleuchtung der Objekte entsprechend der Szene anzupassen, ist die Schätzung der Position und Intensität der realen Lichtquelle anhand heller Bildpartien im Kamerabild und das Hinzufügen entsprechender virtueller Lichtquellen zur virtuellen Szene. Auch wenn die photometrische Registrierung im Gegensatz zu der geometrischen Registrierung nicht zwingend notwendig ist, um die Illusion der Integrität der virtuellen Objekte in die Realität zu gewährleisten, kann auch hier eine unvollständige oder fehlerhafte Registrierung dazu führen, dass diese vermindert wird. Andererseits kann gerade durch eine korrekte oder zumindest plausible photometrische Registrierung die Glaubwürdigkeit einer AR-Szene für den Betrachter enorm gesteigert werden [36].

2.1.4 Rendering

Rendering bezeichnet im Allgemeinen die Darstellung von virtuellen, dreidimensionalen Inhalten auf einem zweidimensionalen Bildschirm. Diese Projektion vom 3D- in den 2D-Raum basiert im Fall von AR-Anwendungen

auf der Transformation, die sich durch die geometrische Registrierung ergibt, und der jeweiligen Kameraperspektive. Für die Darstellung wird das aufgenommene Kamerabild in Echtzeit und perspektivisch korrekt durch die virtuellen Inhalte überlagert, wodurch die eigentliche Augmentierung erfolgt [36]. Alternativ können virtuelle Objekte auch direkt in das Sichtfeld des Betrachters eingeblendet werden (siehe Optical-See-Through in Abschnitt 2.1.5).
Für eine korrekte Darstellung sollte die Pose der virtuellen Kamera, aus deren Sicht die virtuellen Objekte gerendert werden, im Idealfall genau mit der Pose der realen Kamera übereinstimmen. Dafür müssen das Koordinatensystem der virtuellen und der realen Umgebung bzw. der virtuellen und realen Kamera möglichst genau aufeinander registriert sein. Ist dies nicht der Fall, dann heißt das, dass die beiden Kameras den Raum aus unterschiedlichen Perspektiven betrachten, wodurch auch die virtuellen Objekte nicht mehr an der gewünschten Position dargestellt werden können. Gegebenenfalls können während des Renderings noch Anpassungen an dem virtuellen Bild vorgenommen werden, damit die Integration der virtuellen Inhalte in die Realität möglichst nahtlos verläuft. Das kann zum Beispiel die Auflösung und Schärfe des virtuellen Bildes betreffen, aber auch das Einfügen visueller Zusatzeffekte wie Beleuchtung, Schatten, Verdeckung durch reale Gegenstände oder atmosphärische Abschwächung ist möglich.

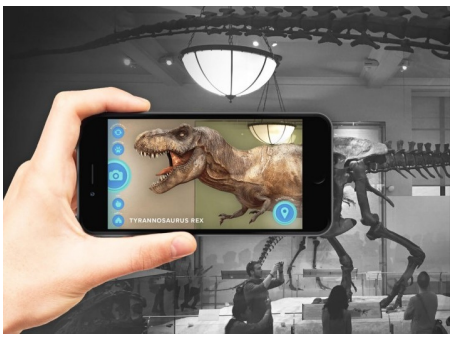
2.1.5 Visuelle Ausgabe

Grundsätzlich kann die visuelle Ausgabe von augmentierter Realität in drei Anzeigearten unterteilt werden: Das *Video-See-Through*, das *Optical-See-Through* und die *Projektionsbasierte Ausgabe*. In den ersten beiden Fällen erfolgt die Darstellung über ein Display, an welchem in der Regel auch die Kamera angebracht ist. Dies können Handheld-Geräte wie zum Beispiel Smartphones oder Tablets, aber auch Datenbrillen oder externe Monitore sein. Im dritten Fall werden virtuelle Inhalte durch einen oder mehrere Projektoren so auf bestehende reale Oberflächen projiziert, dass sich die Wahrnehmung der realen Gegenstände verändert [36]. Durch die Verwendung von externen Monitoren oder projektionsbasierter Ausgabe ist die Erschaffung des Eindrucks der nahtlosen Integrität virtueller Objekte in die Realität jedoch nur bedingt möglich, weswegen ich an dieser Stelle nur auf die beiden erstgenannten Anzeigearten genauer eingehen werde.

Video-See-Through Bei Video-See-Through wird die reale Umgebung mit einer Kamera aufgenommen und als Hintergrundbild verwendet, welches beim Rendering der Szene mit den virtuellen Inhalten perspektivisch korrekt überblendet wird. Diese Kombination von Kamerabild und darauf dargestellten virtuellen Objekten wird dann auf dem entsprechenden Display angezeigt.

Am weitesten verbreitet ist hierbei die Verwendung von mobilen Handheld-Geräten wie Smartphones oder Tablets, da diese zusätzlich zur rückseitigen Kamera zumeist auch über Inertialsensoren verfügen, sodass hybrides Tracking (siehe Visuell-inertiale Odometrie in Abschnitt 2.1.2) möglich ist, und weil sie heutzutage weit verbreitet sind. Die Augmentierung kann somit aus Sicht der Kamera perspektivisch korrekt erfolgen, jedoch nicht für die tatsächliche Blickrichtung des Betrachters, da das Gerät in der Regel in der Hand gehalten wird und sich somit nicht direkt vor dessen Gesicht befindet. Dies schränkt das Gefühl der AR-Illusion zwar etwas ein, ist aber für die meisten Anwendungen ausreichend.
Werden hingegen Datenbrillen (engl. *head-mounted displays*, HMDs) für das Video-See-Through verwendet, wie sie auch im VR-Bereich eingesetzt werden, so kann dieses Problem minimiert werden. Das Videobild der Kamera wird hierbei auf die Displays der Datenbrille eingeblendet, die sich direkt vor den Augen des Nutzers befinden, sodass der Eindruck entsteht, als würde dieser durch die Brille hindurch die reale Umgebung sehen können. Dazu sind an der Vorderseite der Datenbrille ein bis zwei Kameras angebracht, die von ihrer Ausrichtung möglichst genau auf die Blickrichtung des Nutzers abgestimmt sein sollten. Das ist jedoch nicht immer so einfach, weil dafür die Kameralinsen zu jeder Zeit unmittelbar vor den Augen in Blickrichtung liegen müssen, was technisch nur schwer umzusetzen ist. Es reicht aber meistens schon aus, wenn bei der Korrektur des Kamerabildes der translatorische und/oder rotatorische Offset der Kamera herausgerechnet wird, wodurch sich die Kamera nicht direkt vor dem Auge befinden muss. Ein Nachteil davon, dass die Realität „nur“ über ein Videobild dargestellt wird, ist, dass bei einem Systemfehler, zum Beispiel bei einem Ausfall der Kamera, kein Bild mehr vorhanden ist, um etwas darauf darstellen zu können.

Optical-See-Through Beim Optical-See-Through werden wie beim Video-See-Through Datenbrillen verwendet, jedoch sind die Displays in diesem Fall durchsichtig, sodass der Nutzer die Umgebung direkt und unmittelbar durch sie hindurch sehen kann. Hier werden lediglich die virtuellen Objekte als zusätzliche Information optisch in das Gesichtsfeld des Betrachters eingeblendet, sodass die Realität weiterhin wahrnehmbar ist. Dadurch gibt es zwar keine Einschränkung in Bezug auf die Qualität und Auflösung des Kamerabildes, aber durch die Überlagerungen und das meist nicht vollständig durchsichtige Display wird die durch die Datenbrille wahrgenommene Realität verdunkelt, wodurch ein ähnlicher Effekt wie bei einer Sonnenbrille entsteht [36]. Außerdem verdecken die virtuellen Objekte meist nicht vollständig die Realität, wodurch es zu Geisterbildern kommen kann. Weiterhin kann es bei Optical-See-Through sowohl zu dynamischen Registrierungsfehlern (Latenz bei Bewegung) als auch zu statischen Registrie-



(a) Video-See-Through [12]

(b) Optical-See-Through [16]

Abbildung 7: Verleich von Video-See-Through und Optical-See-Through am Beispiel eines Smartphones (a) und der Microsoft HoloLens 2 (b).

rungsfehlern (aufwendige Kamerakalibrierung und „nicht richtig sitzende Brille“) kommen, welche bei Video-See-Through vermieden werden können [50]. Im Gegenzug dafür wird aber die Wahrnehmung der Realität maximiert, da der Nutzer die echte Umgebung sehen kann und nicht nur ein Videobild, was auch ein natürliches Stereo zur Folge hat. Ein Beispiel für eine Optical-See-Through Datenbrille ist die Microsoft HoloLens 2 in Abbildung 7.

2.1.6 Interaktion

Der Forschungsschwerpunkt im AR-Bereich liegt bis heute meist auf technischen Aspekten wie zum Beispiel Tracking oder die Umsetzung verschiedener Display-Arten. Die Interaktion in AR-Anwendungen tritt dabei oft in den Hintergrund, und als Resultat beschränken sich viele dieser Anwendungen nur auf eine rudimentäre Interaktion mit virtuellen, in der realen Welt angezeigten Informationen, oder sie beschränken sich ganz auf die Visualisierung [32]. Möchte man mit virtuellen Objekten interagieren, muss man bedenken, dass konventionelle Eingabegeräte wie die Maus und Tastatur am Computer oft nicht geeignet sind, und dass die Brauchbarkeit und Wahl der Eingabemethode davon abhängig sind, welches Ausgabegerät bzw. welche Anzeigeart für die AR-Anwendung verwendet werden soll. Weiterhin ist zu beachten, dass die Eingaben vieler Eingabetechniken im 2D-Bereich erfolgen, welche dann auf den 3D-Raum übertragen werden müssen, um volle 6DOF-Interaktion zu gewährleisten, wie es für viele AR-Anwendungen gewünscht ist. Im Folgenden stelle ich einige AR-Interaktionstechniken im Allgemeinen vor, bevor ich in Abschnitt 2.2.4 genauer auf die Interaktionsmöglichkeiten im Bereich Mobile Gaming eingehen werde.

Tangible User Interface *Tangible User Interfaces* (TUIs) sind materielle, „anfassbare“ Benutzerschnittstellen (engl. *user interfaces*) (UIs), bei denen physische Objekte sowohl virtuelle Informationen repräsentieren als auch als Eingabemöglichkeit dienen, um diese zu manipulieren [63]. Dabei werden reale Gegenstände im Umfeld des Nutzers mit virtuellen Objekten verbunden, sodass der Zustand des realen Genstands auf den Zustand bzw. eine Eigenschaft des virtuellen Bezugsobjekts abgebildet wird [36]. Das bedeutet, dass man durch die Manipulation physischer Objekte in der realen Welt mit der augmentierten Realität interagieren kann, wodurch die Umgebung selbst zum Interface wird. Ein großer Vorteil dieser Eingabemöglichkeit ist, dass sie intuitiv ist, da die physische Manipulation von Objekten eine natürliche Handlungsweise für den Menschen darstellt.

Dabei kann die Interaktion eine direkte oder eine indirekte Form annehmen: Bei der direkten Form korrespondieren die physischen Eigenschaften eines realen Gegenstands, welcher auch als Platzhalterobjekt bezeichnet wird, unmittelbar mit denen eines virtuellen Objekts [36]. Dies ist zum Beispiel bei der Verwendung von Markern der Fall, auf welchen virtuelle Gegenstände dargestellt werden. Verändert man die Position und/oder Orientierung des Markers, so ändert sich auch gleichzeitig die Pose des darauf dargestellten Objekts. Weiterhin ist es auch möglich, durch die Verdeckung eines Markers zu interagieren, welche dem System mitteilen kann, eine bestimmte Aktion auszuführen. Die direkte Interaktion, welche sich auch auf beliebige Gegenstände erweitern lässt, wird sehr häufig in AR-Anwendungen verwendet.
Im Unterschied zur direkten Form werden bei der indirekten Form die physischen Eigenschaften eines realen Objekts auf die Attribute von virtuellen Objekten übertragen. Das bedeutet, man verändert durch die Manipulation des realen Gegenstands nicht unbedingt die Pose, sondern andere Eigenschaften wie zum Beispiel die Farbe, Form oder Größe von Objekten. Eine Voraussetzung für beide Varianten ist allerdings, dass das verwendete Tracking-Verfahren einigermaßen präzise sein sollte und dass die realen Platzhalterobjekte bei visuellem Tracking immer im Kamerabild zu sehen sein müssen bzw. bei nichtvisuellem Tracking in anderer Form vom System erfasst werden müssen.

Selektion durch Blickrichtung Die Selektion von virtuellen Objekten kann auch mittels der Erfassung der Blickrichtung des Nutzers bzw. der Kamera erfolgen. Dies kann durch *Eye-Tracking* ermöglicht werden, was bedeutet, dass der aktuell vom Betrachter fokussierte Blickpunkt erfasst wird. Jedoch ist dies relativ aufwendig, da ein präzises Eye-Tracking die Integration einer Kamera, die das Auge aufnimmt, und eine entsprechende Kalibrierung erfordert [36]. Weniger aufwändig und auch robuster ist es,

wenn man statt der tatsächlichen Blickrichtung die Orientierung des Kopfes des Nutzers (bei Verwendung einer Datenbrille) bzw. der Kamera (bei Handheld-Geräten) verwendet. Dabei wird die Orientierung so ausgerichtet, dass sich das zu selektierende Objekt in der Mitte des Bildes befindet, welche meist durch eine visuelle Markierung angezeigt wird. Danach muss es nur noch mit einer entsprechenden Auswahlaktion, zum Beispiel einer Geste oder mit einem Sprachbefehl, selektiert werden. Will man auf die Einbindung anderer Interaktionsarten verzichten, kann zur Auswahl auch eine Haltezeit eingesetzt werden, was bedeutet, dass eine entsprechende Aktion ausgelöst wird, wenn man mit der Blickrichtung lange genug auf dem Objekt verharrt.

Gesten- und Sprachsteuerung Bei der Interaktion mittels Gestensteuerung wird die Bewegung der ganzen Hand oder einzelner Finger getrackt, und durch Abgleich mit vordefinierten Gesten können dadurch unterschiedliche Aktionen ausgeführt werden (z. B. die Selektion von Objekten). Dabei kann das Tracken der Hände sowohl visuell als auch nicht-visuell erfolgen. Bei der visuellen Variante können Gesten mittels spezieller, auf dem Kamerabild basierender Gestenerkennungs-Systeme registriert werden, sobald sich eine Hand im Blickfeld der Kamera befindet. Ein Beispiel hierfür ist das auf maschinellem Lernen basierende System von Song et al. [60], und auch die Microsoft HoloLens [16] verfügt über Gestenerkennung. Eine nicht-visuelle Möglichkeit ist die Verwendung eines sogenannten Datenhandschuhs, welcher die Bewegungen der Hand und der Finger misst, um somit die Orientierung im virtuellen Raum zu bestimmen. So wird zum Beispiel die „Tinmith-Hand“ von Piekarski und Thomas [54] zur Menünavigation in ihrem AR-Interface verwendet.
Mittels Sprachsteuerung ist es möglich, mit virtuellen Objekten durch vordefinierte Sprachbefehle zu interagieren. Dazu muss zumeist erst das entsprechende Objekt selektiert werden (z. B. durch die Blickrichtung), sodass die durch den Sprachbefehl eingeleitete Aktion auf dem ausgewählten Objekt ausgeführt werden kann.

Da jede Steuerungsmöglichkeit ihre eigenen Vor- und Nachteile hat und in unterschiedlichen Situationen vielleicht nur eingeschränkt brauchbar ist, wird oft eine Kombination zweier oder mehrerer Möglichkeiten verwendet. Werden unterschiedliche Interaktionsarten in einer Anwendung miteinander kombiniert, so bilden diese ein *multimodales Interface* [32].

2.2 Mobile Augmented Reality

Die starke Verbesserung der Technologie mobiler Geräte in den vergangenen Jahren, darunter vor allem die erweiterte Rechenleistung, Akkulaufzeit und Speicherkapazität, aber auch die Qualität der eingebauten Kameras

und Sensoren, hat dazu geführt, dass auch im mobilen Bereich die Anwendung von AR ihren Einzug gefunden hat. Dabei lässt sich eine *Mobile Augmented Reality* (MAR)-Anwendung durch eine weitere, vierte Eigenschaft definieren: Zusätzlich zu den drei AR-Charakteristiken von Azuma [29] (Abschnitt 2.1.1), nämlich 1. Kombination von realen und virtuellen Objekten in einer realen Umgebung, 2. Interaktivität in Echtzeit und 3. dreidimensionaler Bezug realer und virtueller Objekte zueinander, sollte 4. die Anwendung in sich geschlossen und auf einem mobilen Gerät lauffähig und visualisierbar sein [33]. Das bedeutet, dass eine MAR-Anwendung unabhängig von externer Hardware und der Umgebung auf einem entsprechenden mobilen Gerät benutzbar sein sollte, wobei es sich dabei meist um ein Handheld-Gerät wie ein Smartphone oder Tablet handelt. Aber auch sogenannte *Wearables*, tragbare mobile HMDs wie die *Google Glass* [9], fallen unter diese Definition. Da diese aber für meine Arbeit nicht relevant sind, werde ich mich im weiteren Verlauf nur auf Handheld-Geräte beziehen.

2.2.1 Einschränkungen bei Mobile AR

Aufgrund der heutzutage standardmäßig verbauten Lagesensoren wird bei MAR meist ein hybrides Verfahren von merkmalsbasiertem und sensorbasiertem Tracking (VIO) verwendet. Das hat den Vorteil, dass das Tracking-System in sich geschlossen ist und die Umgebung zuvor nicht bekannt sein muss, was den Aspekt der Mobilität unterstützt. Bei der Erstellung von MAR-Anwendungen ist allerdings zu beachten, dass mobile Geräte im Gegensatz zu stationären Systemen (z. B. Desktop PCs) gewisse Einschränkungen mit sich bringen [32]:

- kleinerer Bildschirm
- beschränkte Rechenleistung
- beschränkte Speicherkapazität
- eingeschränkter Grafik-Support
- geringe Energiereserven
- eingeschränkte Eingabeoptionen (keine physischen Eingabegeräte)

Die Rechenleistung, Speicherkapazität und eingebaute Grafik-Hardware werden zwar zunehmend besser, jedoch sind sie aufgrund der Mobilität und geringen Größe von mobilen Geräten nicht erweiterbar, wie es bei Desktop PCs der Fall ist. Auch aus diesem Grund sollte eine MAR-Anwendung unabhängig und in sich geschlossen sein, um in einer dem System unbekannten Umgebungen zu funktionieren, da sonst relativ große

Datenmengen mit Umgebungsinformationen vorab auf dem Gerät gespeichert sein müssten. Es ist zwar möglich, solche Daten auf externe Cloud-Server auszulagern und bei Bedarf abzurufen, jedoch benötigt dies eine dauerhafte Internetverbindung, was wiederum schlecht für den Energieverbrauch ist. Da mobile Geräte während der Nutzung normalerweise nicht dauerhaft an Strom angeschlossen sind und das Tracking an sich schon viel Energie benötigt (gleichzeitige Nutzung von Kamera und Sensoren, aufwändige Berechnungen) verlieren die Akkus schnell an Energie. Auch bei der Darstellung und Interaktion muss daran gedacht werden, dass durch den kleineren Bildschirm und die fehlenden physischen Eingabegeräte Einschränkungen entstehen können.

2.2.2 Interaktion bei Mobile AR

Aufgrund der Beschränkungen von mobilen Geräten kann das Auswählen einer geeigneten Eingabemethode für AR-Anwendung mitunter schwieriger sein als für VR-Anwendungen bzw. Anwendungen ohne AR [53]. Sie haben zwar oftmals viel mit herkömmlichen auf Touch-Input basierenden Eingaben gemeinsam, bei denen die Interaktion durch Berühren des Bildschirms erfolgt, jedoch muss der Spieler die AR Welt immer im Blick haben, um visuelles Feedback auf seine Aktionen erhalten zu können, weswegen auch die Position, in der das Gerät gehalten wird, nicht vollkommen frei wählbar ist. Je mehr Steuerungs-Elemente auf dem Bildschirm platziert werden, desto wahrscheinlicher ist es, dass wichtige Spielinhalte durch diese und bei der Betätigung durch die Finger verdeckt werden [53]. Das hat dann wiederum einen negativen Effekt auf das Spielerlebnis und bereitet Schwierigkeiten besonders bei Echtzeit-Eingaben. Außerdem macht es das fehlende haptische Feedback schwierig, sich voll und ganz auf das Spiel zu konzentrieren, da man sich auch darauf konzentrieren muss, die richtigen Stellen auf dem Bildschirm zu berühren. Als alternatives Feedback könnten die Vibration des Geräts, auditive Signale oder visuelle Hinweise genutzt werden.

Weiterhin ist bei der Interaktion zu beachten, dass das mobile Gerät im Gegensatz zu HMDs gleichzeitig ein Eingabe- und Ausgabegerät ist (beides geschieht in der Regel über das Display) [32]. Da das Smartphone/Tablet meistens mit einer Hand festgehalten werden muss, ist nur noch eine Hand für die Interaktion frei, wodurch sich TUIs in den meisten Fällen nicht für die Eingabe eignen, da der Nutzer für die Manipulation physischer Objekte oft beide Hände benötigt. Wichtige Aspekte, die bei der Wahl der Interaktionsmethode für eine bessere Benutzerfreundlichkeit beachtet werden müssen, werden in [65] beschrieben:

- Reaktionsgeschwindigkeit des Systems auf Eingaben
- Ermüdung bei längerer Verwendung (wenn das Gerät beispielsweise längere Zeit vor sich hoch gehalten werden muss)
- Präzision der Eingaben
- Koordination (mit wie vielen DOF gleichzeitig erfolgt die Eingabe?)
- Intuitivität bzw. Einfachheit der Eingabemethode
- Handhabung des Eingabegeräts

Das Eingabegerät sollte nach Möglichkeit 6DOF unterstützen (und zwar ohne dass ständig zwischen verschiedenen Modi wie Translation und Rotation umgeschaltet werden muss), und sowohl intuitiv als auch einfach zu handhaben sein. Außerdem sollten Eingaben möglichst präzise auszuführen sein und schnell umgesetzt werden, damit keine Latenz entsteht. Auch eine längere Verwendung des Geräts sollte möglich sein, ohne dass Ermüdungserscheinungen auftreten.

2.2.3 AR im Bereich Mobile Gaming

Augmented Reality-Spiele besitzen das Potential, das Spielerlebnis rein virtueller Spiele zu erweitern, indem virtuelle Spielkomponenten in die echte Welt integriert werden und somit das Gefühl von Realismus und Immersion erhöhen [35]. Daher hat das Interesse an der Entwicklung von mobilen AR-Spielen in jüngerer Zeit stark zugenommen, auch wenn der große Durchbruch bei der breiten Masse noch nicht erreicht wurde [17]. Das hängt auch damit zusammen, dass obwohl viele Leute Interesse an AR-Spielen zeigen [13] (siehe beispielsweise *Pokemon GO*), diese anscheinend noch zu unausgereift sind, besonders was das Immersionsgefühl auf den kleinen mobilen Bildschirmen, die Qualität des Trackings und vor allem die Interaktionsmöglichkeiten betrifft.
Zusätzlich zu den generellen Anforderungen an ein Spiel wie Spielspaß, einfache Bedienung, Intuitivität (einfaches Erlernen) und ein klares, erreichbares Ziel gibt es auch noch spezielle Anforderungen an ein AR-Spiel, um das Spielerlebnis zu verbessern. Dazu zählen eine passende Anzeigeart, die auch für kleine Bildschirme geeignet ist, die Unabhängigkeit von der Umgebung und dem zur Verfügung stehenden Platz bzw. eine entsprechende Skalierung, das möglichst realistische Anpassen virtueller Objekte an die reale Umgebung und die Möglichkeit, mit diesen interagieren zu können, visuelle Hilfen für die 3D-Wahrnehmung (z. B. Schatten) und eine angenehme Art der Interaktion [8]. Daher ist es besonders wichtig, die richtige Eingabemethode für die jeweilige Anwendung zu wählen.

2.2.4 Eingabemöglichkeiten für Mobile AR-Games

AR benötigt ein 6DOF Tracking, um virtuelle und reale Objekte korrekt aufeinander registrieren zu können [40], und somit auch eine 6DOF Eingabe für die Steuerung bzw. Manipulation. Die Wahl der Eingabemethode hängt dabei von der entsprechenden Anwendung ab. So muss zum Beispiel bedacht werden, dass virtuelle Objekte wie ein virtueller Charakter sowohl relativ zur Kamera, relativ zu seinem eigenen Koordinatensystem oder auch absolut im Koordinatensystem der realen Welt gesteuert werden kann [53]. Im Folgenden stelle ich einige der Eingabemöglichkeiten vor, die schon in verschiedenen AR-Spielen implementiert und getestet wurden.

Touch-Input Touch-Input bedeutet die Steuerung von virtuellen Inhalten allein durch Berühren des Bildschirms. Das kann sowohl die 2D-Menünavigation als auch die 3D Manipulation von virtuellen Objekten beinhalten. Dabei besteht die Herausforderung darin, dass die 2D-Eingabe in den 3D-Raum übertragen werden muss [43], wenn man mit 6DOF steuern möchte. Dazu kann zum Beispiel zusätzlich zu den zwei Bildschirm-Achsen (rechts/links, hoch/runter) die Anzahl der Berührungen auf dem Bildschirm gemessen werden, d. h. mit wie vielen Fingern gleichzeitig der Bildschirm berührt wird. So lassen sich verschiedene Gesten für die Translation, Rotation und Skalierung von virtuellen Objekten ausführen. Dabei ist es auch möglich, einen virtuellen Charakter beispielsweise durch Wischen in die entsprechende Richtung zu navigieren [53], oder einen absoluten Punkt in der Welt durch Tippen auf den Bildschirm zu bestimmen, zu dem sich dieser selbstständig hinbewegt [55].
Touch-Input wird in vielen Spielen verwendet, entweder allein oder in Kombination mit anderen Methoden. Allerdings besteht bei dieser Interaktionsmethode der Nachteil, dass der Bildschirm während der Eingabe durch die Finger teilweise verdeckt wird, sodass möglicherweise wichtige Informationen kurzzeitig nicht zu sehen sind.

Virtuelle Buttons/Joysticks Hierbei werden virtuelle Schaltflächen (engl. *buttons*) und/oder Joysticks auf dem Bildschirm angezeigt, die mit den entsprechenden Aktionen belegt werden und ähnlich wie die Tasten auf einem physischen Controller benutzt werden können. So können zum Beispiel in [55] zwei Spieler mittels virtueller Joysticks ihre Avatare (virtuelle Spielfiguren) steuern und mit diesen in einer virtuellen Arena gegeneinander kämpfen. Die Bewegungsrichtung des Avatars wird dabei durch die relative Position des Fingers in einem auf dem Bildschirm dargestellten Kreis, welcher den virtuellen Joystick darstellt, definiert. Bei dieser Steuerungsmethode kann es allerdings als störend empfunden werden, wenn die virtuellen Knöpfe dauerhaft auf dem Bildschirm zu sehen sind, und die Verdeckung durch die Finger bei der Eingabe stellt auch hier ein Problem dar.

Markerbasierte Steuerung Wie bei einem TUI können Marker verwendet werden, um virtuelle Objekte zu steuern und zu manipulieren. Der Bildschirm ist so zwar frei von Verdeckungen, aber die Marker müssen dafür ständig im Bild und gut erkennbar sein. Außerdem hat der Nutzer meistens nur eine Hand frei, um diese zu bewegen, da er mit der anderen Hand das Gerät festhalten muss, wodurch sich die Koordination schwierig gestalten kann. In [52] kann ein virtuelles Auto in einem Rennspiel mit Hilfe eines speziellen TUIs gesteuert werden, welches aus einem Griff mit unterschiedlich angeordneten Makern besteht.

Gestensteuerung Verfügt das AR-System über eine Gestenerkennung, kann die Eingabe auch mittels Gesten erfolgen, wofür allerdings immer mindestens eine Hand im Sichtfeld der Kamera sein muss, was über längeren Zeitraum anstrengend sein kann. Als Beispiel hierfür wird in [42] Fingertracking genutzt, um die Figuren in einem virtuell dargestellten Brettspiel, welches auf einem realen getrackten Bild stattfindet, zu bewegen.

Kippsteuerung Aufgrund der heutzutage standardmäßig eingebauten Lagesensoren in mobilen Geräten kann als Eingabe auch die Bewegung des Geräts an sich verwendet werden. Diese wird von den Sensoren gemessen und durch Kippen bzw. Neigen, Rotieren oder Schütteln des Geräts können verschiedene Aktionen gesteuert werden. In [39] reagiert ein virtueller Hund auf die unterschiedlichen Ausrichtungen und Positionen des Geräts bzw. der Kamera und kann somit durch die Kombination verschiedener Bewegungen gesteuert werden. In [34] wird die Kippsteuerung dazu genutzt, ein virtuelles Raumschiff in einem Weltraumkampf gegen andere Spieler zu steuern. Ein Nachteil dieser Steuerung ist, dass der Sichtbereich während der Eingabe, also durch die Neigung oder Rotation des Geräts, eingeschränkt wird.

Steuerung mittels Kamera Ähnlich zur Kippsteuerung kann statt des gesamten Geräts auch nur die Kamera allein dazu verwendet werden, virtuelle Objekte zu steuern. So wird zum Beispiel in [55] ein Avatar gesteuert, indem der Spieler mit der Kamera in diejenige Richtung zeigt, in die er sich bewegen soll. Dabei wird in der Mitte des Bildschirms ein Fadenkreuz abgebildet und die Spielfigur folgt demjenigen Punkt in der AR-Umgebung, auf welchen dieses projiziert wird (bei gleichzeitigem Tracken eines Markers, um die Position im Raum bestimmen zu können). Die Zielposition wird dabei kontinuierlich mit dem Blickpunkt der Kamera aktualisiert.

Externe Controller Die Verwendung externer Controller für die Eingabe kann von Vorteil sein, um eine freie Sicht auf den Bildschirm zu haben und,

je nach Controller-Art, auch haptisches Feedback bei der Eingabe. Nachteil ist, dass ein zweites Gerät verwendet wird, wodurch sich Schwierigkeiten bei der gleichzeitigen Konzentration auf beide Geräte ergeben können. Ein Beispiel für einen externen Controller wäre ein physisches Gamepad, wie man es von Spielekonsolen kennt. In einem anderen Ansatz in [59] wird eine Smartwatch für diesen Zweck verwendet.

Es gibt aber auch noch einige weitere interessante Ansätze, deren Beschreibung hier zu weit führen würde. So wird zum Beispiel in der App *Tendar* [24] Gesichtserkennung für die Interaktion mit einem virtuellen Fisch verwendet. Dieser kann den Gesichtsausdruck analysieren und überlebt nur, wenn er sich von den Emotionen „ernähren“ kann. Oftmals wird eine Mischung aus verschieden Eingabemethoden gewählt, die auf die entsprechende Anwendung zugeschnitten werden, um eine bestmögliche Interaktion zu gewährleisten.

2.3 Software und Augmented Reality SDKs

In diesem Abschnitt stelle ich die zur Umsetzung meines Spiels erforderliche Software vor, bevor ich in Abschnitt 3 genauer auf die Funktionalitäten des verwendeten AR Software Development Kits (SDKs) eingehen werde.

2.3.1 Unity

Unity ist eine Laufzeit- und Entwicklungsumgebung für Spiele des Unternehmens Unity Technologies [25]. Sie steht zur kostenlosen kommerziellen Nutzung bereit, solange der Umsatz bei unter 100.000 US-Dollar im Jahr verbleibt, und bietet alle nötigen Funktionen (Grafik-Engine, Animationen, Einbindung von Sound, etc.), um ein Spiel für aktuelle Plattformen wie PC, Smartphones, Webbrowser und Spielkonsolen zu erstellen. Weiterhin können relativ leicht zusätzliche Toolkits als Erweiterungen (Plugins) in Unity eingebunden werden, darunter auch AR-SDKs wie ARCore oder Vuforia (Abschnitt 2.3.3).
Die Entwicklungsumgebung (*Unity Editor*) bietet einen 3D-Editor, mit welchem einfache 3D-Objekte erstellt, bearbeitet und einer Szene hinzugefügt werden können. Werden komplexere 3D-Modelle benötigt, können diese in einem externen 3D-Modellierungsprogramm wie *Blender* (Abschnitt 2.3.2) erstellt und in Unity als sogenannte *Assets* importiert werden.
Eine virtuelle Szene in Unity ist als Szenengraph organisiert und besteht aus *GameObjects*, welchen Komponenten (Skripte, Materialien, Sound, etc.) zugeordnet werden, die das Verhalten von diesen definieren und steuern. Diese wiederum enthalten Variablen (editierbare Eigenschaften), die über den Editor oder im Skript selber festgelegt werden können. Eine Szene enthält in der Regel eine virtuelle Kamera und eine oder mehrere Lichtquel-

len. Die Logik-Komponenten werden von den GameObjects getrennt erstellt und als Komponenten an diese angefügt. Dabei handelt es sich um sogenannte *Skripte*, die meist in der Programmiersprache C#, aber auch in JavaScript oder C++ geschrieben werden können, um das Verhalten der Objekte zu kontrollieren. Standardmäßig besitzen diese Skripte eine Start()- bzw. Awake()-Methode, die ausgeführt wird, sobald die Anwendung startet (oder das Script neu erstellt wird) und eine Update()-Methode, die bei jeder Aktualisierung des Bildes (also entsprechend der Framerate) ausgeführt wird. So können beim Start virtuelle Objekte und Komponenten erstellt und initialisiert werden, deren Verhalten durch das automatische Aufrufen von `Update()` gesteuert werden kann. GameObjects und deren Komponenten können zu *Prefabs* zusammengefasst werden, welche wie vorgefertigte Bausteine verwendet werden können, was sinnvoll ist, wenn mehrere gleichartige Objekte in einer Szene benötigt werden.
Möchte man sein Spiel testen, kann dies direkt in der *Game-View* im Editor geschehen. Dort wird die grafische Darstellung und das Verhalten des Spiels simuliert. Ist das Spiel fertig, kann eine ausführbare Anwendung für die entsprechende Zielplattform mit der Exportfunktion erzeugt werden.

2.3.2 Blender

Blender ist eine unter der GPL-Lizenz verfügbare Open Source Software für 3D-Modellierung, die auf allen gängigen Betriebssystemen läuft und Funktionen einer kompletten 3D-Pipeline bietet: Modellierung, Rigging, Animation, Simulation, Rendering, Compositing und Motion Tracking, sowie 2D-Animation und das Editieren von Videos [5]. Mit Blender ist es möglich, dreidimensionale Objekte zu modellieren, zu texturieren, zu animieren und als 3D-Modelle in verschiedenen Formaten zu exportieren, sodass sie in anderen Programmen wie Unity importiert und verwendet werden können.

2.3.3 Augmented Reality SDKs

Von verschiedenen Unternehmen und Entwicklern werden Software Development Kits (SDKs) zur Verfügung gestellt, welche die grundlegenden Funktionalitäten für die Erstellung von AR-Anwendungen, darunter vor allem das Tracking, bereitstellen, wobei jedes SDK seine eigenen Eigenschaften und Funktionen besitzt.
ARCore von Google [11] und *ARKit* von Apple [2] nutzen beide VIO-Tracking zur Poseberechnung und Oberflächenerkennung. Generell bieten diese beiden ähnliche Funktionalitäten, darunter auch das Platzieren virtueller Objekte auf Oberflächen, Bilderkennung und eine Schätzung der Lichtverhältnisse in der Szene, um virtuelle Objekte dementsprechend zu beleuchten. Allerdings wird ARKit nur auf iOS Betriebssystemen und

ARCore hingegen hauptsächlich auf Android unterstützt. *Vuforia* [27] ist ein kommerzielles AR-SDK für mobile Geräte, welches vor allem im industriellen Bereich eingesetzt wird. Dieses nutzt auch VIO für das Tracking, ist aber im Gegensatz zu ARCore und ARKit plattformunabhängig und unterstützt neben Bild- zusätzlich noch 3D-Objekterkennung. *ARToolKit* [4] ist eine plattformunabhängige Open-Soure C++ Softwarebibliothek. Im Gegensatz zu den anderen hier genannten SDKs verwendet ARToolKit kein VIO, sondern optisches Markertracking, um die Kamerapose zu ermitteln und virtuelle Objekte einzublenden.

Ich habe mich in dieser Arbeit für die Verwendung des ARCore SDKs entschieden, da dieses zu Anfang der Entwicklung meines AR-Spiels noch relativ neu war und interessante Funktionalitäten für die Erstellung von AR-Anwendungen bereithält, welche ich im nächsten Abschnitt genauer erläutern werde. Zusätzlich dazu war auch die Android-Unterstützung und Anbindungsmöglichkeit an Unity ausschlaggebend.

3 ARCore

ARCore [11] ist eine vom US-amerikanischen Unternehmen Google entwickelte AR-Plattform zur Realisierung von AR-Anwendungen auf Android-Geräten. Sie wurde im August 2017 vorgestellt und kann als Nachfolger von *Project Tango* [19] angesehen werden.

3.1 Grundkonzepte von ARCore

Im Gegensatz zu Tango, wo spezielle Tiefensensoren des Geräts genutzt wurden, um ein Tiefenbild der Umgebung zu erstellen und somit die 3D-Position im Raum ermitteln zu können, benötigt ARCore keine speziellen Sensoren oder zusätzliche Kameras, da hierbei allein mit der Standard-Hardware von mobilen Geräten und mit Software ähnlich gute Ergebnisse erzielt werden. ARCore stellt verschiedene Programmierschnittstellen (APIs) zur Verfügung, die das Erstellen von AR-Anwendungen möglich machen und vereinfachen sollen. Hierzu zählt vor allem die Verwendung der Kamera, das Tracking von realen Oberflächen und virtuellen Objekten, die Interaktion mit diesen und das Management der virtuellen Szene, d. h. der kompletten AR-Session. Startet man eine ARCore-Anwendung, so versucht das System zuerst, mit Hilfe der Kamerabilder eine Karte der Umgebung zu erstellen, und danach die relative Position des Geräts in dieser Umgebung zu bestimmen. Google definiert hierbei drei Grundkonzepte, welche von dem SDK verfolgt werden [11]: Bewegungstracking, Registrierung der Umgebung und Abschätzung der Lichtverhältnisse, auf welche ich im Folgenden näher eingehen werde.

3.1.1 Bewegungstracking

Bewegungstracking (*Motion Tracking*) ermöglicht es einem Gerät, seine Position relativ zur realen Umgebung und seine Bewegung im dreidimensionalen Raum zu erfassen. Hierfür hat Google ein eigenes Tracking-System namens *Concurrent Odometry and Mapping* (COM) [23] entwickelt, bei der visuelles merkmalsbasiertes Tracking mit der inertialen Messeinheit des Geräts kombiniert wird. Zum einen werden mit Hilfe des Kamerabildes visuell eindeutige Merkmale identifiziert. Diese werden genutzt, um mittels visueller Odometrie bzw. SLAM (Abschnitt 2.1.2) die Veränderung der Position und Orientierung der Kamera relativ zur Startposition zu berechnen. Zum anderen wird die Beschleunigung und Ausrichtung des Geräts mittels Trägheitsmessung und Messung der Rotationsgeschwindigkeit durch das eingebaute Trägheitsnavigationssystem (IMU) ermittelt. Dadurch erhält man eine zweite Auswertung der Position und Orientierung der Kamera in Bezug auf die reale Welt.

Abbildung 8: Beispielhaft dargestelltes Bewegungstracking in ARCore. Bestimmte Merkmale des Sofas werden als Feature Points identifiziert. [11]

Die Registrierungsfehler, die sich aus der einen wie der anderen Methode ergeben, sind voneinander unabhängig, da die beiden Verfahren von komplementärer Natur sind [64], wodurch es möglich ist, sie so zu kombinieren, dass das Ergebnis die kleinstmögliche Fehlermenge aufweist, sodass die Schwächen des jeweils anderen Verfahrens kompensiert werden können. Man nimmt dann das beste Ergebnis, welches durch die Kombination von beiden erreicht werden kann.

Es handelt sich hierbei also um ein auf VIO basierendes Tracking-Verfahren, wobei aus der Kombination der visuellen Odometrie und den IMU-Sensordaten die Pose der Kamera bestimmt wird, während sie sich durch den dreidimensionalen Raum bewegt und dabei immer weitere Feature Points identifiziert (beispielhaft dargestellt in Abbildung 8) und somit das Verständnis der Umgebung erweitert, wodurch das Tracking über die Zeit hinweg an Genauigkeit gewinnt. Weiterhin hat das den Vorteil, dass das System in einer komplett unbekannten Umgebung funktioniert, da die Karte der Umgebung jedes Mal von Grund auf neu erstellt wird.
Indem die Pose der virtuellen Kamera auf die so berechnete Pose der Kamera des Geräts ausrichtet wird, können virtuelle Objekte mit der richtigen Perspektive in die reale Welt eingeblendet werden. Das Kamerabild wird dann mit dem gerenderten virtuellen Bild überlagert, sodass der Eindruck entsteht, die virtuellen Objekte wären Teil der realen Umgebung.

Seit ARCore Version 1.11.0 (August 2019) ist es zudem möglich, zur Verbesserung der Tracking-Ergebnisse auf die Daten der Tiefenkamera zuzugreifen, sollte das Gerät eine solche besitzen [28]. Dadurch können zusätzlich zu den visuellen Merkmalen auch die entsprechenden Tiefenwerte hinzugezogen werden, um die Umgebung noch besser registrieren zu können.

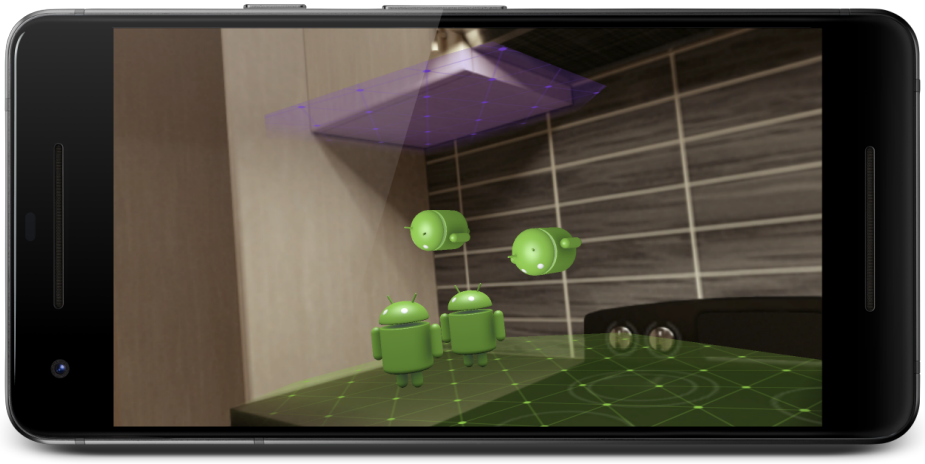


Abbildung 9: HelloAR Beispiel-App von Google. Die von der Anwendung erkannten Oberflächen werden als Grid visualisiert. [11]

3.1.2 Registrierung der Umgebung

Mit Registrierung der Umgebung (*Environmental Understanding*) wird die Lokalisierung der Größe und Position von (horizontalen und vertikalen) Oberflächen in ARCore bezeichnet. Durch fortlaufendes Identifizieren von Feature Points und Oberflächen verbessert ARCore sein „eigenes Verständnis“ der realen Umgebung. Dies geschieht, indem die Feature Points des Bewegungstrackings zu Punktwolken zusammengefasst werden, wodurch ein internes virtuelles Abbild der Umgebung entsteht. Je weiter sich die Kamera durch die Welt bewegt, desto mehr Informationen werden dort hinzugefügt und desto genauer wird das Verständnis der Umgebung. Um dies zu erreichen, sucht ARCore nach Clustern von Feature Points, die auf ein und derselben Oberfläche zu liegen scheinen, fasst diese als Ebene zusammen und erstellt ein Ebenen-Objekt (*Plane*), um sie zu repräsentieren. Dadurch ist es zum Beispiel möglich, virtuelle Objekte auf realen Oberflächen zu platzieren (Abbildung 9), oder Schatten auf diesen abbilden zu können. Außerdem ist es durch sogenannte Oriented Points (Feature Points mit Orientierung) möglich, die korrekte Orientierung virtueller Objekte in Bezug auf gekrümmte Oberflächen abzuschätzen, indem zusätzlich die umliegenden Feature Points betrachtet werden, um mit deren Hilfe die Krümmung der Oberfläche am aktuellen Feature Point abschätzen zu können. Der Neigungswinkel der virtuellen Objekte wird dann relativ zur Orientierung der Oriented Points, also dem Winkel der Oberflächenkrümmung, betrachtet.

Die Registrierung der Umgebung ist ein dynamischer Prozess, was bedeutet, dass die Größe und Position von den erkannten Oberflächen zur Laufzeit von ARCore angepasst werden, sollten weitere zur Oberfläche gehö-

renden Feature Points gefunden oder die Position der schon vorhandenen korrigiert und somit das eigene Verständnis der Umgebung verbessert werden. Ein Nachteil davon, dass Oberflächen mittels visuellen Feature Points gefunden werden, besteht darin, dass glatte Oberflächen mit wenig oder gar keiner Textur (wie zum Beispiel eine weiße Wand) nur sehr schwer bis gar nicht erkannt werden können. Außerdem kann es zu Problemen beim Tracking von Oberflächen kommen, wenn die Lichtverhältnisse in der Umgebung nicht ausreichend sind sowie auch bei transparenten, reflektierenden oder bei dynamischen Oberflächen (wie beispielsweise bei Glas, Wasser oder Gras im Wind).

3.1.3 Abschätzung der Lichtverhältnisse

Mit ARCore ist neben der geometrischen auch eine photometrische Registrierung (Abschnitt 2.1.3) möglich. Somit kann ein Bild auf die in der Szene gegebene Beleuchtung hin analysiert und daraus die Richtung der Lichtquelle, die Farbe und durchschnittliche Lichtintensität des aktuellen Kamerabildes geschätzt werden (*Light Estimation*). Im einfachsten Fall wird hierzu die durchschnittliche Helligkeit des Bildes berechnet und auf den Intensitätswert einer virtuellen Lichtquelle übertragen. Ist das Kamerabild eher dunkel, so werden auch die eingeblendeten Objekte dunkler dargestellt. Weiterführend kann auch mit mittels des Helligkeitsverlaufs im Bild die ungefähre Position und Orientierung der Lichtquelle in der realen Umgebung abgeschätzt werden, sodass die virtuelle Lichtquelle dementsprechend darauf ausgerichtet werden kann. Dadurch können virtuelle Objekte durch eine virtuelle Lichtquelle unter ähnlichen Bedingungen wie in der realen Umgebung beleuchtet werden, sodass sich der Realismus erhöht (Abbildung 10). Seit ARCore Version 1.10.0 (Juni 2019) ist auch die Berechnung von ambientem Licht, Schatten, Reflektionen und spekularen Highlights mittels maschinellem Lernen und Analyse des Kamerabildes möglich [28].

3.2 Weitere Funktionalitäten von ARCore

Zusätzlich zu den grundlegenden Funktionen des Trackings besitzt ARCore weitere Funktionalitäten, von denen ich im Folgenden die zwei wichtigsten kurz beschreiben werde, da diese auch bei der Umsetzung meines Spiels verwendet wurden.

3.2.1 Anchors

Damit die Position von virtuellen Objekten über einen längeren Zeitraum korrekt getrackt werden kann, verwendet ARCore sogenannte *Anchors* (dt. Anker), eine spezielle Klasse von *Trackables*. Trackables sind diejenigen

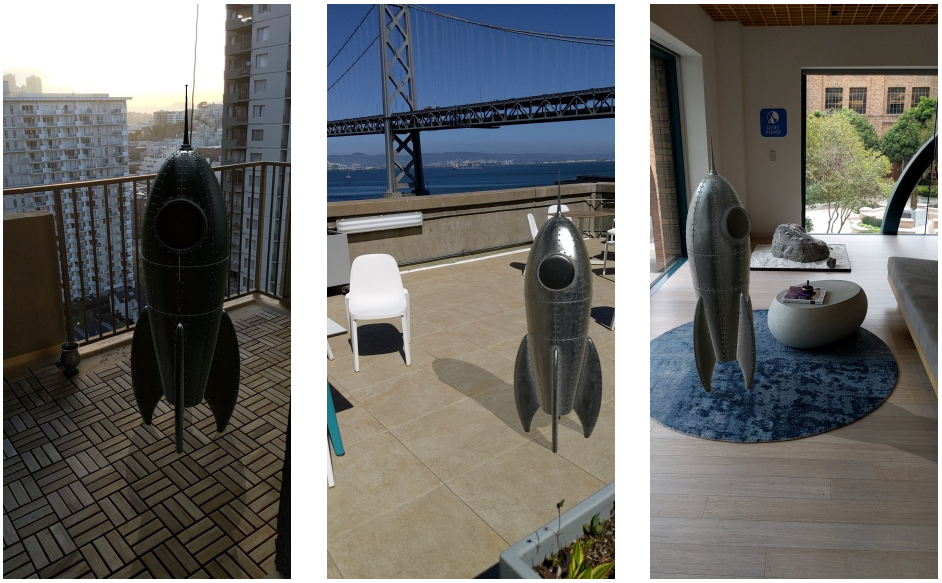


Abbildung 10: Ein virtuelles Raketen-Modell in realen, unterschiedlich beleuchteten Umgebungen. Die Position und Intensität der realen Lichtquelle werden von ARCore geschätzt und auf die virtuelle Lichtquelle übertragen. [11]

Objekte, die während der gesamten ARCore-Session gespeichert werden (auch wenn sie sich nicht aktuell im Bild befinden) und deren Position ständig aktualisiert wird, sobald sich das Verständnis der realen Umgebung erweitert (Feature Points und Planes gehören auch zu den Trackables). Da mit der Zeit immer mehr Umgebungsinformationen hinzugefügt und somit auch die Positionen von Feature Points und Oberflächen aktualisiert werden, kann es sein, dass sich die Pose von virtuellen Objekten ohne Anchor (relativ zur realen Welt) verändert, was dann so aussieht, als würden diese langsam „wegdriften“. Deshalb wird üblicherweise bei der Instanziierung eines Objekts zuerst ein Anchor an derselben Position erstellt und an dem entsprechenden Trackable (zum Beispiel auf einer Oberfläche oder an einem Feature Point) und das Objekt schließlich an dem Anchor verankert (diese werden üblicherweise nicht visualisiert, sodass sie für den Nutzer unsichtbar sind). Dadurch bleibt das räumliche Verhältnis zwischen dem speziellen Trackable und dem virtuellen Objekt stabil, auch wenn sich die Kamera bewegt. Wenn sich zum Beispiel die Position einer getrackten Oberfläche aufgrund neuer Umgebungsinformationen verändert, dann ändert das auch gleichzeitig die Position des virtuellen Objekts, welches an der Oberfläche verankert ist, sodass dieses weiterhin „an seinem Platz“ auf der Oberfläche bleibt. Dabei sollte ein guter Anchor an oder in der Nähe von möglichst eindeutigen und gut erkennbaren Merkmalen verankert werden, damit der Tracking-Algorithmus diesen auch bei unterschiedli-

chen Blickwinkeln an der korrekten Position erkennt. Er sollte außerdem auch unter veränderten Lichtbedingungen, veränderter Rotation und Skalierung und bei Bewegungsrauschen erkennbar sein.
Die Verwendung von Anchors hat zusätzlich den Vorteil, dass die Objekte weiterhin getrackt werden, auch wenn sie aktuell nicht im Kamerabild sind, sodass man zwischenzeitlich den Raum verlassen und später wiederkommen könnte, und die Objekte wären immer noch an Ort und Stelle. Andererseits sollten Anchors wenn möglich wiederverwendet oder gelöscht werden, wenn sie nicht mehr gebraucht werden, da sie durch das ständige Tracking Performanz kosten.

Weiterhin können mit Hilfe von *Cloud Anchors* Anchors und benachbarte Feature Points auch mit anderen Geräten in der Nähe geteilt werden, wenn diese ebenfalls ARCore verwenden. Dadurch können AR-Anwendungen auf unterschiedlichen Geräten simultan dieselben Anchors verwenden, was kollaborative Anwendungen ermöglicht.

3.2.2 Augmented Images

Mit Augmented Images ist es der AR-Anwendung möglich, spezifische 2D-Bilder, sogenannte *Imagemarker*, mittels merkmalsbasiertem Tracking zu erkennen und mit ihnen zu interagieren (ähnlich wie beim markerbasierten Tracking). So können virtuelle Objekte zum Beispiel auf Bildern, Postern oder Werbeplakaten dargestellt werden (Abbildung 11). Hierzu wird zuerst eine Bilddatenbank mit den entsprechenden Referenzbildern angelegt. Wird eines dieser Imagemarker aus der Datenbank im Kamerabild erkannt, wird dessen Pose berechnet und das virtuelle Objekt angezeigt. Von diesem Zeitpunkt an wird die Position, Orientierung und physische Größe des Bildes getrackt (auch wenn es sich nicht im Kamerabild befindet), da das Bild gleichzeitig auch als Anchor für das entsprechende Objekt dient.

3.3 Anwendungsbeispiele

Um aufzuzeigen, was mit Augmented Reality auf mobilen Geräten möglich ist, stelle ich im Folgenden vier Beispielanwendungen vor, in denen ARCore verwendet wird. Dies zeigt auch, wie vielfältig die Anwendungsgebiete von AR sind.

3.3.1 Streem

Die Dienstleistungs- und Kommunikationsanwendung *streem* [22] von Streem Inc. ist eine Lösung für die Fernwartung und -hilfe von Fachkräften bei Problemen mit Haushaltsgeräten oder Ähnlichem und ermöglicht

Abbildung 11: Eine Augmented Images Demonstration von Google. Das aus Würfeln bestehende virtuelle Objekt wird über das getrackte Bild gelegt, sodass es aussieht, als käme es direkt aus der Wand. [11]

es, dass sich Kunden von Zuhause aus mit den entsprechenden Fachkräften online über einen Videostream verbinden können. Die Fachkraft kann das Problem des Kunden über die Smartphone-Kamera erkennen, eventuelle Messungen vornehmen und ihn in Echtzeit durch die verschiedenen Schritte zur Reparatur bzw. Wartung leiten, indem er über Audio Instruktionen zum Lösen des Problems gibt und gleichzeitig visuelle Hilfen, wie zum Beispiel Markierungen, Pfeile (Abbildung 12) oder Laserpointer, direkt in das Kamerabild einfügt. Somit können Probleme selber gelöst und die Kosten für die Anfahrt von Fachkräften eingespart werden.

3.3.2 Curate

Die Applikation *Curate* [6] von Loft Inc. ist eine Inneneinrichtungs-Anwendung, mit deren Hilfe der Nutzer seine Wohnung virtuell nach persönlichen Vorlieben einrichten kann (Abbildung 13). Dies kann zum Beispiel möglichen Käufern eines Hauses dabei helfen, sich die Räume komplett nach ihrem Geschmack eingerichtet vorzustellen, ohne dass der Verkäufer diese für jeden potentiellen Kunden neu einrichten muss, was diesem Geld einspart und die Chance auf einen Verkauf erhöht. Oder der Nutzer probiert vor dem Einrichtungseinkauf für seine eigene Wohnung verschiedene Designs aus, um zu testen, was für eine Einrichtung ihm am besten gefällt.

Abbildung 12: Die AR-App *streem*. [22]

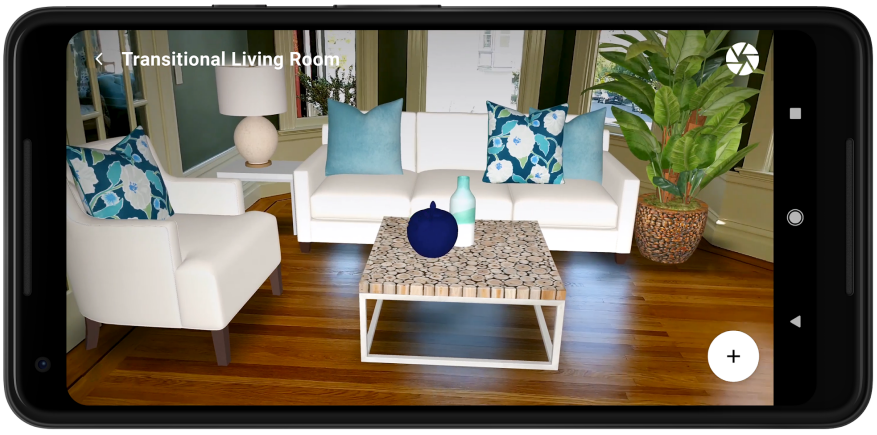


Abbildung 13: Die AR-App *Curate*. [6]

Abbildung 14: Die AR-App *Just a Line*. [14]

3.3.3 Just a Line

Mit *Just a Line* [14] von Google Creative Lab ist es einem jederzeit und überall möglich, einfache Zeichnungen in AR zu erstellen und diese dann in einem kurzen Video zu teilen. Die Zeichnungen können sowohl allein als auch zusammen mit anderen angefertigt werden, indem sie über den Touchscreen eines Smartphones vor einen in den Raum hinein gemalt werden (Abbildung 14). Um Dreidimensionalität zu erreichen, kann der Nutzer sich beim Zeichnen um die Zeichnung herum bewegen, sodass diese von verschiedenen Blickwinkeln aus erweitert werden kann.

3.3.4 A&E Crime Scene: AR

Mit dem Spiel *A&E Crime Scene: AR* [1] von A&E Television Networks Mobile kann der Nutzer in die Rolle eines Mordermittlers schlüpfen und von zuhause aus erleben, wie es wäre, an einem Tatort zu arbeiten und Mordfälle zu lösen. Innerhalb des Spiels hat er eine Auswahl an Hilfsmitteln, um Beweise zu sammeln und zu analysieren (Abbildung 15), die ihn schließlich zum Täter führen. Dabei wird der Tatort komplett in AR vor dem Nutzer dargestellt, sodass er die Möglichkeit hat, diesen aus der Nähe und von allen Seiten aus zu betrachten.

Abbildung 15: Die AR-App *A&E Crime Scene: AR*. [1]

4 Konzeptentwicklung

In diesem Kapitel werde ich zunächst auf die Ziele und Anforderungen eingehen, die bei der Konzeption und Entwicklung meiner Anwendung von Bedeutung waren. Danach werde ich die grundlegende Spielidee erläutern, bevor ich die einzelnen Aspekte und Spielkomponenten genauer beschreiben werde. Diese werden in Kapitel 5 gegebenenfalls nochmals detaillierter dargestellt und in ihrem Konzept erweitert.

4.1 Ziele und Anforderungen

Mein Ziel war es, eine mobile AR-Anwendung zu entwickeln, mit der man nicht nur virtuelle Informationen in die reale Welt einblenden, sondern auch mit diesen interagieren kann. Dabei fiel meine Wahl auf die Erstellung eines Mobile AR-Games, da es in diesem Bereich noch recht wenige wirklich interaktive Spiele gibt, die die Möglichkeiten von AR voll ausnutzen. Außerdem lässt sich bei einem Spiel eine der grundlegenden Eigenschaften von AR, die Interaktion in Echtzeit, sehr gut in verschiedenen Weisen umsetzen und testen. Da das Spiel für mobile Geräte wie Smartphones oder Tablets entwickelt werden soll, müssen die Einschränkungen, die damit einhergehen (siehe Abschnitt 2.2.1), beachtet werden.
In dem Spiel soll es möglich sein, selbstständig virtuelle Objekte in der realen Umgebung platzieren und manipulieren zu können. Dabei ist wichtig, dass diese korrekt registriert werden und auch bei längerer Verwendung an ihrem Platz in der realen Welt bleiben. Zusätzlich dazu sollen die realen erkannten Oberflächen von den virtuellen Objekten beeinflusst werden und umgekehrt. Das bedeutet es soll möglich sein, dass virtuelle Objekte mit realen Oberflächen kollidieren können, und dass sie einerseits einen Schatten auf reale Oberflächen werfen und andererseits von diesen verdeckt werden können, wenn sie sich dahinter befinden, wodurch die 3D-Wahrnehmung unterstützt und die Immersion verstärkt wird.
Eine gute Möglichkeit, das Platzieren von Objekten in ein Spiel zu integrieren, ist die Selbstgestaltung der virtuellen Spielwelt. Dadurch soll die langfristige Motivation gesteigert und der Spieler dazu bewegt werden, die Umgebung in AR zu erkunden. Weiterhin soll das Spiel ansprechend gestaltet sein, und es muss ein geeignetes Interface erstellt werden, sodass die Steuerung auch in AR intuitiv und unkompliziert ist, da sonst sehr schnell Frustration bei den Spielern auftreten kann. Ist die Steuerung zu kompliziert, um sie schnell erlernen zu können oder Spaß dabei zu haben, dann wenden sich die Spieler sehr schnell von dem Spiel ab. Außerdem muss der Nutzer der Anwendung jederzeit in der Lage sein, das Spiel an sich unter Kontrolle zu haben, was bedeutet, dass dieser den Zugriff auf verschiedene Optionen (z. B. den Schwierigkeitsgrad) benötigt oder das Spiel jederzeit beenden kann, wenn er das möchte.

Auch der Einstig in das Spiel sollte einfach gestaltet sein, sodass die Spielmechanik schnell erlernt werden kann. Dabei muss darauf geachtet werden, dass das Spiel nicht zu einfach ist, da sonst Langeweile aufkommt, aber auch nicht zu schwer, da der Spieler sonst leicht frustriert ist. Die Balance beim Schwierigkeitsgrad ist also wichtig, genauso wie ein klar definiertes und vor allem erreichbares Ziel. Weiß der Spieler nicht, was er zu tun hat oder wie, dann wird er das Spiel auch nicht weiterspielen. Zur Unterstützung dessen könnten zum Beispiel visuelle Informationen in das Sichtfeld eingeblendet werden.
Ein weiterer Aspekt ist die langfristige Motivation. Ein Spieler kann dazu bewegt werden, ein Spiel weiter zu spielen, wenn er für seine Erfolge belohnt wird. Das könnte beispielsweise beim Abschließen eines Levels oder Aufstellen einer neuen Bestzeit geschehen. Es soll auch die Möglichkeit geben, Spielinhalte anpassen zu können, damit der Nutzer in der Lage ist, das Spiel in jeder beliebigen Umgebung spielen zu können, was bei einem AR-Spiel für mobile Geräte sinnvoll ist. Wäre der Spieler auf eine bestimmte Umgebung beschränkt, so würde dies die Mobilität der Anwendung stark eingrenzen.

Generell muss auch darauf geachtet werden, dass technische Fehler und Ausfälle beim Tracking so weit es geht vermieden werden, da das Spiel auf AR aufbaut und sonst nicht mehr spielbar wäre. Kommt es zu Ausfällen beim Tracking, sollte der Spielinhalt an der letzten bekannten Position verbleiben, bis das Tracking wieder aufgenommen wird und die Position wieder aktualisiert werden kann. Müsste der Spieler das Spiel jedes Mal neu starten, wenn das Tracking unterbrochen wird, so wäre er sehr schnell demotiviert. Ein weiterer Aspekt neben der Stabilität und der Präzision des Trackings, der besonders bei der Verwendung von AR auf mobilen Geräten beachtet werden muss, ist die Performanz. Dauerhaftes Tracking, besonders wenn sich viele virtuelle Objekte gleichzeitig in der Szene befinden, benötigt einiges an Rechenleistung und Energie, welche auf mobilen Geräten mitunter sehr beschränkt sein können. Um also die Echtzeitfähigkeit gewährleisten zu können und eine stockende Darstellung zu verhindern, muss das verwendete Tracking-Verfahren sowohl effizient als auch schnell sein. Um eine Verlangsamung bei längerer Verwendung zu vermeiden, sollten virtuelle Objekte, die nicht mehr benötigt werden, wieder aus der Szene gelöscht werden. Außerdem sollten die virtuellen Spielkomponenten zwar möglichst detailliert sein, da man sich mit der Kamera frei um sie herum bewegen und sie von allen Seiten betrachten können soll, doch gleichzeitig sollten ihre Modelle so einfach wie möglich gehalten werden, damit die Darstellung dieser nicht zusätzlich die Performanz verringert.

4.2 Grundlegende Spielidee

In meinem Spiel soll die reale Umgebung des Spielers als Spielfeld dienen, welches durch virtuelle Spielkomponenten ergänzt wird. So entstand die Idee eines Parcours, welcher eine Mischung aus Renn- und Geschicklichkeitsspiel darstellt. Gewöhnliche Rennspiele mit Rennautos werden meist auf zweidimensionaler Ebene realisiert, da man mit diesen bekanntlich nur auf einem festen Untergrund fahren kann. Da ich aber ein AR-Spiel entwickeln wollte, bei dem die Steuerung in alle drei Raumrichtungen und nicht nur auf horizontaler Ebene möglich ist, habe ich mich für ein Flug-Parcours-Setting entschieden. Die Flugobjekte können in beliebige Richtungen (auch vertikal) gesteuert werden und somit ist es möglich, die vollen 6DOF von AR auszunutzen.
Der virtuelle Parcours wird hierbei vom Spieler selbst aufgebaut, was den Spielspaß die Motivation erhöhen soll und gleichzeitig die Interaktion von virtueller und realer Welt fördert. Das Ziel des Spiels ist es, den Parcours mit einem virtuellen Flugobjekt möglichst schnell zu absolvieren (Rennspiel-Komponente) und dabei so wenige Kollisionen wie möglich zu verursachen, wodurch das Feingefühl des Spielers bei der Steuerung eine wichtige Rolle spielt (Geschicklichkeitsspiel-Komponente). Dieses einfache und bekannte Spielprinzip ermöglicht einen schnellen Einstieg in das Spiel und der Spieler hat ein klares Ziel vor Augen, das es zu erreichen gilt.

4.3 Das Spiel pARcours

Dieses Spiel ermöglicht es, reale Oberflächen in der Umgebung erkennen und auf diesen virtuelle Objekte platzieren zu können, um sich so einen eigenen Parcours aufzubauen. Dieser kann anschließend mit ebenfalls virtuellen Flugobjekten absolviert werden.

4.3.1 Spielprinzip: pARcours-Modus

Den Hauptbestandteil dieses Spiels bildet der sogenannte *pARcours*-Modus. Die Aufgabe dabei ist es, einen eigenen Parcours aus virtuellen Objekten in der realen Welt zu errichten und diesen dann mit einem ausgewählten Flugobjekt zu absolvieren.
Um den Parcours aufbauen zu können, muss der Spieler aber zuerst einmal Oberflächen in seiner Umgebung einscannen, indem er die Kamera langsam umher bewegt und auf die entsprechenden Oberflächen richtet. Dabei gibt es die Option, sich die erkannten Oberflächen für eine bessere Übersicht als Grid visualisieren zu lassen. Die Oberflächenerkennung läuft während des gesamten Spieldurchlaufs im Hintergrund, sodass die vom System erkannte Umgebung auch nach dem Start des Levels noch erweitert wird. Um auf den Oberflächen das Startfeld, die Checkpoints und die

Hindernisse platzieren zu können, aus denen ein Parcours besteht, kann zwischen zwei Methoden gewählt werden: Zum einen ist es möglich, die virtuellen Objekte mittels Imagemarkern zu platzieren, welche als Ankerpunkt in der realen Welt dienen, damit die Objekte auch dann im Level verbleiben, wenn sich die Marker aktuell nicht im Kamerabild befinden. Dazu werden die entsprechenden Imagemarker so in der Umgebung verteilt, wie auch der Parcours aufgebaut sein soll. Diese werden dann der Reihe nach eingescannt, indem sie vollständig mit der Kamera eingefangen werden, bis sie von dem System erkannt wurden. Danach können die Objekte noch manipuliert werden, indem die Position und/oder Rotation der Imagemarker verändert wird, solange diese danach nochmals von der Kamera eingescannt werden.
Zum anderen kann der Parcours auch mittels Touch-Input aufgebaut werden. Dazu kann der Spieler die entsprechenden Objekte aus einer Liste auswählen und muss dann lediglich noch auf diejenige Stelle des Bildschirms tippen, an der er ein Objekt platzieren möchte. Allerdings funktioniert dies nur, wenn an der entsprechenden Stelle auch schon eine Oberfläche erkannt wurde, da das Objekt an dieser verankert werden muss, damit es an seinem Platz bleibt. Danach können die Objekte durch Touch-Gesten verschoben, rotiert und skaliert werden. Imagemarker werden bei dieser Methode nicht benötigt, können aber wahlweise hinzugenommen werden. Wie viele Checkpoints bzw. Hindernisse der Spieler momentan im Level platziert hat, wird auf dem Bildschirm angezeigt. Dadurch erhält er einen besseren Überblick über den Parcours.

Hat der Spieler den Parcours fertig aufgebaut, kann das Level gestartet werden. Dabei erscheint das ausgewählte Flugobjekt auf dem zuvor platzierten Startfeld, aber erst wenn das Startfeld verlassen wird, beginnt das Level und ein Timer fängt an zu zählen. Dann muss der Spieler, um das Level zu beenden, die Checkpoints der Reihe nach bis zum letzten durchfliegen. Wichtig hierbei ist, dass zu jedem Zeitpunkt immer nur ein Checkpoint aktiviert ist, nämlich derjenige, der als nächstes durchflogen werden muss. Inaktive Checkpoints können nicht durchflogen werden. Die Reihenfolge entspricht hierbei der Reihenfolge, in der die Checkpoints in der Szene platziert wurden. Sobald das Flugobjekt den letzten Checkpoint passiert hat, endet das Level und der Endbildschirm wird angezeigt, auf welchem die für das Level benötigte Zeit, der Levelscore und die eingesammelten Juwelen (Abschnitt 4.3.2) angezeigt werden.
Zusätzlich zu der aktuell benötigten Zeit wird auch die Bestzeit für das Level angezeigt. Stellt der Spieler eine neue Bestzeit auf, wird diese gespeichert und er erhält extra Punkte für das Absolvieren des Parcours, welche zum Levelscore dazu addiert werden. Der Levelscore ist auch während des Spiels sichtbar und zeigt an, wie viele Punkte der Spieler aktuell gesammelt hat. Immer wenn ein Checkpoint erfolgreich durchquert wurde, erhöht sich

der Levelscore entsprechend des Schwierigkeitsgrades des Checkpoints. Sollte das Flugobjekt aber mit einem Checkpoint, einem im Level platzierten Hindernis oder einer vom System erkannten Oberfläche zusammenstoßen, dann werden zur Strafe wieder Punkte vom Levelscore abgezogen (bis auf minimal null Punkte). Für das Beenden eines Levels, das Aufstellen einer neuen Bestzeit und eingesammelte Juwelen erhält der Spieler ebenfalls Punkte, die am Ende alle zum Levelscore dazugerechnet werden. Dabei hängt die Anzahl der für das Beenden des Levels erhaltenen Punkte auch von dem Schwierigkeitsgrad des Levels ab. Dieser wird intern berechnet und beim Aufbauen des Levels auf dem Bildschirm angezeigt. Er ergibt sich aus der Art und Anzahl der im Level platzierten Objekte (Abschnitt 4.3.2) und dem Schwierigkeitswert, der dem Flugobjekt zugeordnet ist. Je weniger Zeit für den Parcours benötigt wurde, je höher dessen Schwierigkeitsgrad und je kleiner die Anzahl an Kollisionen war, desto mehr wird sich der Levelscore am Ende erhöhen. Die mit dem Levelscore erhaltenen Punkte werden schließlich zum Juwelen-Konto des Spielers dazugerechnet, der sozusagen einen globalen Punktestand darstellt. Die so gesammelten Juwelen werden auch gespeichert, wenn der Spieler das Spiel beendet, sodass er diese auch beim nächsten Start noch besitzt. Diese könnten zum Beispiel als digitale Währung dazu verwendet werden, um sich spielinterne Erweiterungen oder Modifikationen für die Flugobjekte zu ertauschen. Dieses Belohnungssystem würde dazu motivieren, das Spiel öfters zu spielen und die Parcours so gut wie möglich abzuschließen.

Nachdem der Spieler die entsprechenden Punkte nach Beenden des Levels erhalten hat, kann er über ein Menü auswählen, ob er das Level wiederholen, einen neuen Parcours aufbauen, das Flugobjekt oder den Modus ändern oder ob er zurück zum Hauptmenü gehen möchte. Startet er das Level neu, wird auch der Levelscore wieder auf null gesetzt, da dieser für jeden Level-Durchgang separat aufgestellt wird. Die zuvor aufgestellte Zeit hingegen wird gespeichert, damit es möglich ist, neue Bestzeiten aufzustellen. Erstellt der Spieler aber ein neues Level, so wird auch die Bestzeit zurückgesetzt, da diese pro Level aufgestellt wird (zwischen unterschiedlich aufgebauten Parcours ist es schlecht möglich, die benötigten Zeiten zu vergleichen). Es ist auch während des Levels jederzeit möglich, über einen Menü-Button das Spiel zu pausieren, das Level zu beenden und ein neues aufzubauen oder zurück zum Hauptmenü zu gelangen.

4.3.2 Spielelemente

Zusätzlich zu den Flugobjekten sind diejenigen Elemente besonders wichtig, aus denen die Parcours aufgebaut werden: das Startfeld, die Checkpoints und die Hindernisse. Dabei ist zu beachten, dass die Größenverhältnisse der einzelnen Elemente zueinander stimmig sind. So dürfen zum

Beispiel die Checkpoints nicht zu klein sein, da man sonst mit dem Flugobjekt nicht mehr hindurchfliegen kann, aber auch nicht zu groß, weil sie sonst zu einfach zu durchqueren wären. Und auch in die reale Umgebung sollten sie sich möglichst gut einfügen können. Da der Spieler das Spiel in jeder beliebigen Umgebung spielen können soll, ist es wichtig die Möglichkeit zu haben, die Spielelemente auf die entsprechende Größe zu skalieren. Möchte er als Spielfeld einen kompletten Raum nutzen, so müssen die Objekte größer dargestellt werden, als wenn der Spieler beispielsweise nur den Platz einer Tischplatte zur Verfügung hat. Generell sollen die Objekte ansprechend gestaltet sein und es ist wichtig, dass sie visuelles und auditives Feedback geben, wenn sie miteinander interagieren (zum Beispiel bei einer Kollision).

Flugobjekte Bevor der Parcours aufgebaut wird, wählt der Spieler ein Flugobjekt aus, mit welchem er das Level spielen möchte. Ich habe mich dazu entschieden, dem Spieler mehrere Arten von Flugobjekten zur Auswahl zu stellen, die sich sowohl in ihrer Erscheinung als auch in ihrer Art zu steuern unterscheiden. Die Abwechslung hierbei kann den Spielspaß längerfristig aufrecht erhalten und der Spieler kann dasjenige Flugobjekt wählen, mit welchem er am besten zurechtkommt. Dabei unterscheiden sich die Steuerungen sowohl von der Reaktion auf die Eingaben als auch in ihrem Schwierigkeitsgrad. So gibt es leichter zu steuernde Flugobjekte, die für Anfänger geeignet sind, wie auch eines, bei dem die Steuerung eine extra Herausforderung darstellt. Die einzelnen Flugobjekte und ihre Steuerungen werden in Abschnitt 5.6.3 und 5.7 beschrieben. Weiterhin ist es wichtig, dass der Spieler ein Feedback erhält, wenn er mit den Flugobjekten interagiert, sei es visuell mittels Animationen oder auditiv durch Soundeffekte, damit er weiß, dass diese auf seine Eingaben reagieren.

Startfeld Das Startfeld ist essentiell für den Start eines Levels, da dieses der Ausgangspunkt und gleichzeitig der Anker für das Flugobjekt in der realen Welt ist, damit es in Relation zur realen Umgebung immer korrekt registriert werden kann. Ohne Startfeld kann auch das Flugobjekt nicht geladen werden. Befindet sich der Spieler zu Anfang des Levels auf dem Startfeld, ist diese sozusagen eine „Schutzzone“, die verhindert, dass der Timer anfängt zu zählen, sodass der Spieler sich erst einmal orientieren kann, bevor er losfliegt. Erst wenn das Flugobjekt diese Zone verlässt, startet das Level. Wählt der Spieler aus, das Level neu zu starten, oder möchte er die Position des Flugobjekts zurücksetzten, weil er es zum Beispiel zu weit weg gesteuert und deshalb aus den Augen verloren hat, so kehrt das Flugobjekt wieder auf seinen Ausgangspunkt auf dem Startfeld zurück.

Checkpoints Die Checkpoints, wovon immer mindestens einer im Level enthalten sein muss, machen den eigentlichen Parcours aus. Dabei handelt es sich um ringähnlich geformte Kontrollpunkte, die vom Flugobjekt der Reihe nach durchflogen werden müssen, um den Parcours zu absolvieren. Dabei muss der Spieler das Flugobjekt geschickt lenken, damit er nicht mit den Rändern der Checkpoints zusammenstößt, wodurch sich der Levelscore verringern würde. Die Reihenfolge wird dabei von der Reihenfolge, in der diese platziert wurden, festgelegt. Daher ist immer nur der nächste Checkpoint, der durchquert werden muss, aktiviert, während die anderen noch deaktiviert sind. Erst wenn der aktuell aktivierte Checkpoint durchflogen wurde, wird der nächste aktiviert. Der Status der Checkpoints wird dabei durch ihre Farben angezeigt.
Um Abwechslung ins Spiel zu bringen gibt es unterschiedliche Varianten von Checkpoints, die sich in ihrer Größe und Form und somit auch in ihrem Schwierigkeitsgrad unterscheiden. Jedem Checkpoint ist deswegen ein Wert zugewiesen, der seinen Schwierigkeitslevel beschreibt. Um wie viel sich der Levelscore erhöht, wenn der Spieler einen Checkpoint durchquert hat, hängt dabei von diesem Wert ab. Je höher der Schwierigkeitswert ist, desto mehr Punkte erhält er für den Checkpoint. Dadurch lässt sich auch der Schwierigkeitsgrad des gesamten Levels beeinflussen: Umso mehr „schwierige“ Checkpoints im Level enthalten sind, desto höher ist der Schwierigkeitsgrad dessen und desto mehr Punkte erhält der Spieler am Ende für das Abschließen des Levels.

Hindernisse Hindernisse sind für einen Parcours nicht zwingend erforderlich, können aber wahlweise hinzugefügt werden, um den Schwierigkeitsgrad des Levels zu erhöhen. Auch diese besitzen einen zugewiesenen Schwierigkeitswert, der davon abhängt, wie groß sie sind und wie sie geformt sind. Stößt das Flugobjekt gegen ein Hindernis, verringert sich der Levelscore entsprechend dem Schwierigkeitswert des Hindernisses.

Juwelen Juwelen sind einsammelbare Objekte, die abhängig vom gewählten Schwierigkeitsgrad in zufälligen Zeitabständen und Positionen um das Flugobjekt herum auftauchen können. Werden sie eingesammelt, indem sie mit dem Flugobjekt berührt werden, so erhält der Spieler einen entsprechenden Bonus bzw. Effekt, der von der Art des Juwels abhängt. Außerdem bekommt er für eingesammelte Juwelen nach Beenden des Levels zusätzliche Punkte auf den Levelscore. Werden diese nicht eingesammelt, entsteht ihm aber auch kein Nachteil und sie verschwinden nach kurzer Zeit wieder von selbst. In Tabelle 1 sind die verschiedenen Arten von Juwelen und ihr Effekt angegeben, welche (abhängig von Spielmodus und Schwierigkeitsgrad) erscheinen können.

Juwel	Beschreibung	Modus	Schwierigkeitsgrad
TimeBonus	Der Spieler erhält einen Zeitbonus auf seine aktuelle Zeit.	ARcade	Alle
TimePenalty	Der Spieler erhält eine Zeitstrafe auf seine aktuelle Zeit.	ARcade	Schwer
SpeedBonus	Erhöht die Geschwindigkeit des Flugobjekts.	Alle	Normal, Schwer
SpeedPenalty	Verringert die Geschwindigkeit des Flugobjekts.	Alle	Schwer
DoubleBonus	Der Spieler erhält für eine kurze Zeit einen doppelten Bonus (oder Strafe) für eingesammelte Juwelen (doppelte Zeit, Geschwindigkeit).	Alle	Schwer
Invulnerability	Macht den Spieler für kurze Zeit unverwundbar (kein Verringern des Levelscores bei Kollisionen).	pARcours	Normal, Schwer

Tabelle 1: Die verschiedenen Arten von Juwelen mit einer Beschreibung ihres Effekts und dem Spielmodus und Schwierigkeitsgrad, in dem sie erscheinen können.

4.3.3 Weitere Spielmodi: ARcade und FreeFlight

Neben dem in Abschnitt 4.3.1 beschriebenen Hauptmodus *pARcours* gibt es noch zwei weitere Modi in diesem Spiel, die vom Spieler ausgewählt werden können: den *ARcade*-Modus und den *FreeFlight*-Modus.

ARcade-Modus In diesem Modus geht es darum, auf Zeit so viele Juwelen einzusammeln wie möglich. Hierbei wird lediglich ein Startfeld als Ausgangspunkt benötigt, daher ist es nicht möglich, andere Objekte in der Umgebung zu platzieren. Im Gegensatz zum *pARcours*-Modus zählt der Timer die Zeit aber rückwärts, sobald der Spieler das Startfeld verlässt, wodurch das Level in dem Moment endet, in dem der Timer auf null steht. In dieser Zeit muss er versuchen, mit seinem Flugobjekt so viele der zufällig auftauchenden Juwelen zu sammeln wie möglich. Dabei kommt es aber auch darauf an, welche Juwelen eingesammelt werden: Am wichtigsten sind hierbei die *TimeBonus*-Juwelen, da diese einem einen Zeitbonus geben und damit das Ende hinauszögern. Werden genügend davon in der entsprechenden Zeit eingesammelt, kann sogar verhindert werden, dass der Timer die Null erreicht. Aber Vorsicht: Sammelt man die falschen Juwelen ein, wie zum Beispiel die *TimePenalty*-Juwelen, so kann das Spiel sehr

schnell zu Ende sein. Je länger der Spieler durchhält, bis der Timer abgelaufen ist, und je mehr Juwelen er in dieser Zeit einsammelt, desto besser ist der Levelscore am Ende.

FreeFlight-Modus Der *FreeFlight*-Modus ist dazu gedacht, dass der Spieler ohne Zeitbeschränkung frei umherfliegen und die verschiedenen Flugobjekte und Steuerungen austesten kann, ohne dass dabei ein bestimmtes Ziel erreicht werden muss. Daher gibt es hier weder einen Levelscore noch einen Timer oder Juwelen, und es gibt auch keine Punkte für diesen Modus. Allerdings ist es möglich, zusätzlich zu dem Startfeld Hindernisse in der Umgebung zu platzieren, sodass man seine Flugfertigkeiten austesten und verbessern kann.

4.3.4 User Interface und Eingabemethoden

Generell wird das Spiel über den Touchscreen des Geräts gesteuert, wie man es auch von anderen mobilen Anwendungen kennt. Das Auswählen von Funktionen im Menü und die Menünavigation wird mittels Tippen auf die entsprechenden virtuellen Schaltflächen realisiert. Startet der Spieler das Spiel, wird er zuerst danach gefragt, in welcher Dimension er spielen möchte. Damit ist die globale Skalierung gemeint, die auf alle Spielinhalte automatisch angewandt wird, damit sich diese in die Umgebung und den vorhandenen Platz einfügen. Danach gelangt der Spieler in das Hauptmenü, in dem Einstellungen (Abschnitt 5.2) wie beispielsweise die Flugsteuerung vorgenommen oder die Anleitung des Spiels eingesehen werden kann. Zudem wird hier die Anzahl der Juwelen angezeigt, die der Spieler im Laufe des Spiels gesammelt hat.

Sobald das Level aufgebaut und gestartet wurde, wird automatisch die zuvor in den Optionen ausgewählte Flugsteuerung aktiviert, womit sich das Flugobjekt steuern lässt. Dabei stehen vier Eingabemethoden zur Auswahl, die im Folgenden kurz vorgestellt und in Abschnitt 5.7 nochmal genauer in ihrer Funktionsweise erläutert werden. Die ersten drei basieren dabei auf der Eingabe über den Bildschirm, wohingegen die vierte komplett darauf verzichtet.

Virtuelle Joysticks Bei dieser Eingabemethode erscheinen zwei virtuelle Joysticks auf dem Bildschirm, mit denen das Flugobjekt sowohl in der Horizontalen (rechter Joystick) als auch in der Vertikalen (linker Joystick) gesteuert werden kann. Dazu müssen diese wie bei einem physischen Controller mit den Fingern in die entsprechenden Richtungen bewegt werden.

Touch-Input Berührt der Spieler bei dieser Methode den Bildschirm, so fliegt das Flugobjekt in die entsprechende Richtung (auf horizontaler Ebene). Das bedeutet, es fliegt zu dem Punkt in der 3D-Welt, auf den der 2D-Punkt des Touch-Inputs projiziert wird. Die Höhe wird zusätzlich mit einem virtuellen Joystick gesteuert.

Steuerung mittels Kamera Es ist auch möglich, das Flugobjekt mit Hilfe der Kamera des Geräts zu steuern. Dabei wird ein Fadenkreuz in der Mitte des Bildschirms angezeigt, welches den Zielpunkt im dreidimensionalen Raum markiert. Drückt der Spieler nun auf den auf dem Bildschirm erschienenen „Drive"-Button, fliegt das Objekt in Richtung dieses Fadenkreuzes, sodass das Flugobjekt dadurch gelenkt werden kann, dass er mit der Kamera in diejenige Richtung zeigt, in die er fliegen möchte. Dabei bleibt die Zieldistanz zur Kamera immer gleich. Möchte der Spieler diese ändern, also von der Kamera aus gesehen weiter weg oder näher heran fliegen wollen, kann er das mit einem virtuellen Joystick machen. Wird dieser vor oder zurück gezogen, so ändert sich damit die Zieldistanz (von der Kamera aus gesehen).

Gamepad Besitzt der Spieler ein mit dem Gerät kompatibles Gamepad, kann dieses für die Steuerung des Flugobjekts verwendet werden. Das funktioniert genauso wie bei den virtuellen Joysticks, nur dass hierbei ein physisches Eingabegerät in der Hand gehalten und komplett auf die Bildschirmeingabe verzichtet wird. Das hat den Vorteil, dass die Sicht auf den Bildschirm immer frei ist, zudem eignet sich diese Eingabemethode für alle Spieler, die Probleme mit der Eingabe über den Touchscreen haben.

Da die Spieler unterschiedliche Präferenzen haben, was die Eingabemethode betrifft, war es ein Ziel, mehrere Steuerungsmöglichkeiten anzubieten. Der eine Spieler kommt mit der einen Steuerung besser klar, wohingegen ein anderer Spieler mit einer anderen Steuerung besser zurechtkommen wird. Durch die Auswahlmöglichkeit ist es unwahrscheinlicher, dass ein Spieler das Spiel nur aus dem Grund nicht gut findet, weil er mit der Steuerung nicht zurechtkommt. So kann er diejenige auswählen, die ihm am besten liegt. Außerdem kann es den Spaßfaktor und die Langzeitmotivation erhöhen, wenn nicht immer dieselbe Steuerung verwendet wird, sondern auch andere ausprobiert werden können. Dies stellt eine extra Herausforderung dar.

5 Implementierung

In diesem Kapitel gehe ich auf die zentralen Aspekte bei der Umsetzung des Spiels *pARcours* ein. Dies beinhaltet die technische Umsetzung der im Konzept beschriebenen Komponenten und die Realisierung der eigentlichen Spielelemente.

Ein Hauptbestandteil dieses Spiels bildet das Erstellen der eigenen Level, was in zwei Kernaufgaben eingeteilt werden kann: zum einen das Erkennen und Tracken von realen Oberflächen (Abschnitt 5.3) und zum anderen das Platzieren von virtuellen Objekten auf diesen (Abschnitt 5.4), welche den Parcours bilden. Der grundlegende Spielablauf wird in Abbildung 16 dargestellt.

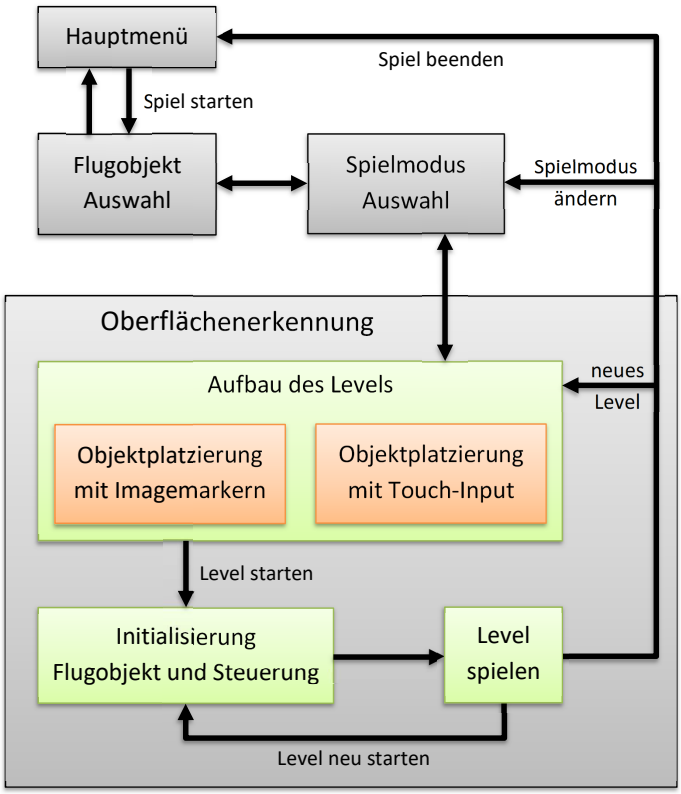


Abbildung 16: Grundlegender Programmablauf von *pARcours*.

Nachdem der Spieler das Flugobjekt und den Spielmodus ausgewählt hat, beginnt das System, nach Oberflächen in der realen Umgebung zu suchen. Auf diesen lassen sich danach die virtuellen Spielelemente für den Parcours aufstellen. Das geschieht durch Einscannen von Imagemarkern oder über den Touchscreen. Hat der Spieler das Level fertig aufgebaut, wird das entsprechende Flugobjekt mit der ausgewählten Steuerung initialisiert

und das eigentliche Spiel gestartet. Nach Beenden des Levels kann ein neuer Modus ausgewählt, ein neues Level aufgebaut oder dasselbe Level nochmal gestartet werden. Im letzteren Fall bleiben die Oberflächen und Parcours-Objekte erhalten. Während des gesamten Ablaufs ab dem Aufbauen des Levels wird parallel dazu weiterhin die Oberflächenerkennung ausgeführt, damit im System die Spielumgebung immer entsprechend aktualisiert werden kann.

Die Implementierung der verschiedenen Eingabe- und Steuerungsmöglichkeiten (Abschnitt 5.7) für die Flugobjekte sowie die Interaktion der virtuellen Objekte mit realen Oberflächen (Abschnitt 5.8) bilden weitere Kernaspekte dieser Arbeit. Des Weiteren werde ich auf die eigentliche Spiellogik hinter den verschiedenen Spielmodi (Abschnitt 5.5) und die virtuellen Spielkomponenten (Abschnitt 5.6) eingehen.

5.1 Hardware- und Software-Setup

Zur Realisierung meines Spiels habe ich mich für die Verwendung der Unity Game Engine (Abschnitt 2.3.1) entschieden, da diese alle nötigen Entwicklungsmöglichkeiten für Spiele bietet und zusätzlich benötigte SDKs einfach eingebunden werden können. Unity wurde in der Version 2018.3.14f1 verwendet, zusammen mit der Programmierumgebung Microsoft Visual Studio[3] für die Programmierung der Skripts. Die 3D-Spielkomponenten wurden mit der 3D-Modellierungssoftware Blender (v2.79) und die 2D-Grafiken (Interface, Logo) mit Inkscape[4] (v0.92) erstellt, einer freien und plattformunabhängigen Software zur Bearbeitung und Erstellung zweidimensionaler Vektorgrafiken. Zusätzlich zu den selbst erstellten Modellen wurde auf die öffentlich zugängliche 3D-Modell-Datenbank sketchfab[5] und den Unity Asset Store[6] zurückgegriffen.
Um die AR-Komponenten und die Interaktion mit der realen Umgebung zu realisieren, habe ich mich für die Verwendung des ARCore SDKs[7] entschieden, da dieses sowohl die grundlegenden Funktionalitäten des Trackings als auch der Oberflächenerkennung und die Erkennung von Touch-Gesten bereitstellt. Das SDK (zu Beginn der Arbeit Version 1.9.0, während der Entwicklung aktualisiert auf Version 1.11.0) wurde dazu in Unity eingebunden. Aufbauend auf der Oberflächenerkennung konnten somit die Platzierung von virtuellen Objekten, deren Verdeckung und die Kollisionserkennung sowie die Verwendung von Imagemarkern realisiert werden. Als Zielplattform dienen mobile Geräte (Smartphones oder Ta-

[3] https://visualstudio.microsoft.com/de/
[4] https://inkscape.org/de/
[5] https://sketchfab.com/feed
[6] https://assetstore.unity.com/
[7] https://developers.google.com/ar/develop/downloads

blets) mit Android-Betriebssystem, welche ARCore unterstützen (eine Liste aktuell unterstützter Geräte kann unter [3] eingesehen werden). Entwickelt und getestet wurde die Anwendung auf einem Honor View 20 mit Android-Version 9, einem Huawei Kirin 980 Prozessor, 8 GB Arbeitsspeicher und einer Bildschirmauflösung von 2310x1080 Pixeln. Um ARCore auf dem Gerät nutzen zu können, musste der Google Play-Dienst für AR [10] heruntergeladen und installiert werden.

5.2 Menüs und Benutzerführung

Startet der Spieler das Spiel über den entsprechenden Icon auf dem Gerät (Abbildung 17 (a)), wird zuerst kurz das Logo (Abbildung 17 (b)) eingeblendet, welches mit dem Programm Inkscape erstellt wurde. Danach kann der Spieler die gewünschte globale Skalierung auswählen, die auf alle virtuellen Objekte angewandt wird.

(a) Icon (b) Logo

Abbildung 17: Icon und Logo des Spiels *pARcours*.

Im Hauptmenü (Abbildung 18) kann gewählt werden, ob das Spiel gestartet oder beendet, weitere Optionen eingestellt oder die Anleitung gelesen werden soll. Die Eingabe und das Auswählen von Aktionen innerhalb der Menüs geschieht mittels Touch-Input. Der Spieler muss lediglich auf die entsprechenden Stellen des Bildschirms tippen. Am oberen Bildschirmrand wird immer die aktuelle Anzahl an Juwelen angezeigt, die sich der Spieler durch das erfolgreiche Beenden der Level verdient hat.

Über den „Manual“-Button gelangt der Spieler zu einer Anleitung (Abbildung 19 (a)), in der die wesentlichen Aspekte des Spiels erläutert werden, damit der Spieler beispielsweise weiß, wie er die Level aufbaut oder die Flugobjekte steuert. In den Optionen (Abbildung 19 (b)) können verschiedene Einstellungen für das Spiel vorgenommen werden. Dazu zählen unter

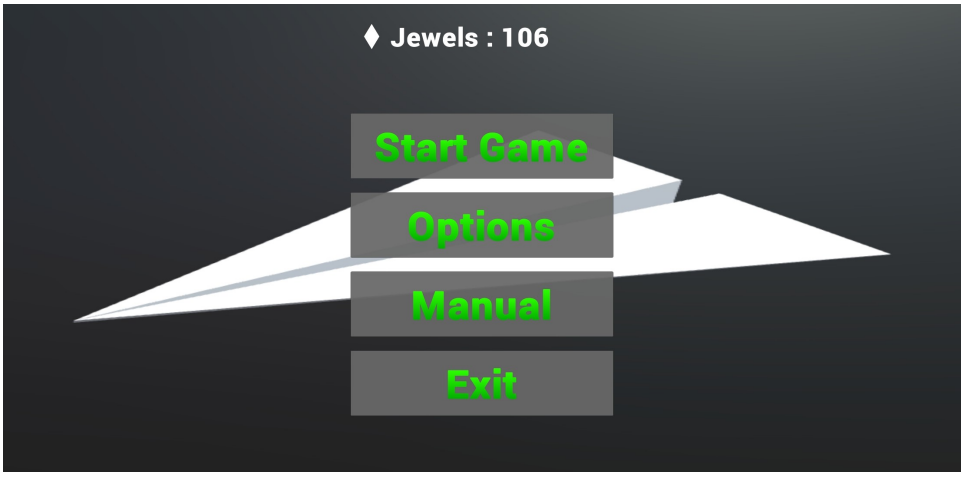
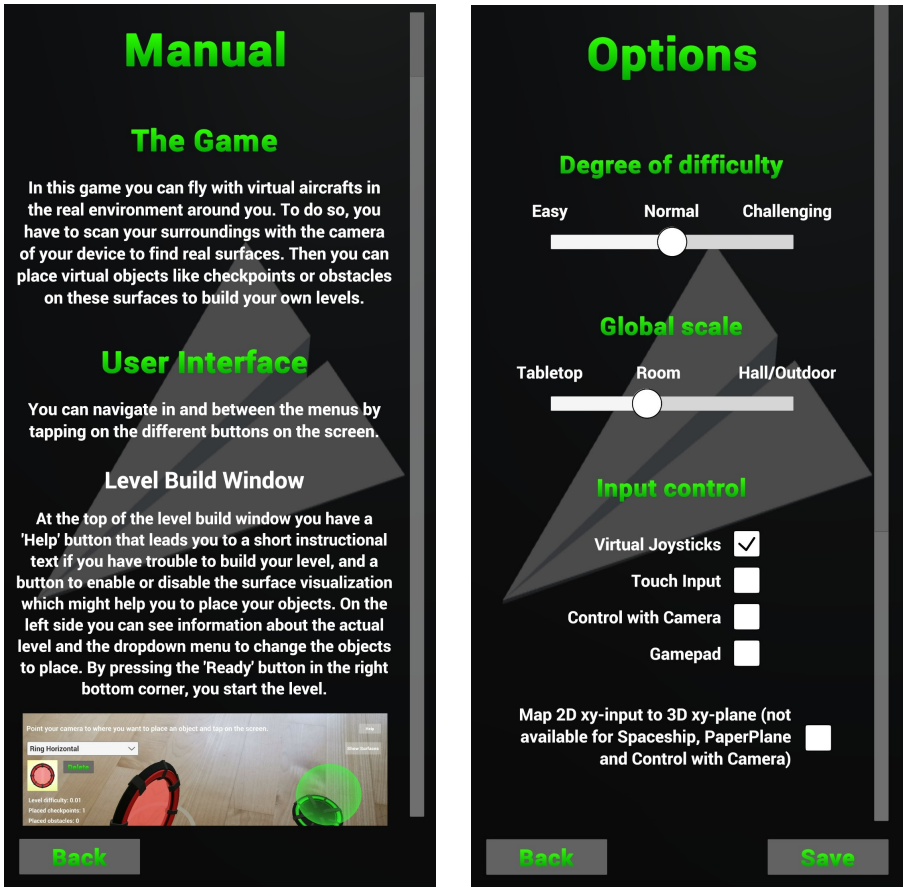


Abbildung 18: Hauptmenü des Spiels *pARcours*.



(a) Anleitung (b) Optionen

Abbildung 19: Anleitung und Optionen im Spiel *pARcours*.

Abbildung 20: Auswahl des Flugobjekts.

anderem der Schwierigkeitsgrad (Leicht, Normal und Schwer), welcher die Geschwindigkeit der Flugobjekte, die für das Beenden eines Level erhaltenen Punkte und die Art und Anzahl an Juwelen beeinflusst, die im Level erscheinen können. Außerdem kann die globale Skalierung geändert werden, mit welcher alle virtuellen Objekte im Spiel skaliert werden, damit sie sich von der Größe an den in der Umgebung vorhandenen Platz anpassen. Weiterhin können hier die Eingabemethode für die Steuerung der Flugobjekte gewählt und die verschiedenen visuellen Unterstützungen ein- und ausgeschaltet werden. Dazu zählen die Visualisierung der vom System erkannten Oberflächen, die Indikatoren, das Fadenkreuz bei der Steuerung mittels Kamera und die Instruktionen, die während des Levelaufbaus angezeigt werden. Diese sind alle standardmäßig aktiviert. Zuletzt kann das Spiel in den Optionen auf den Ausgangszustand zurückgesetzt werden. Dadurch werden alle vorgenommenen Einstellungen und die eingesammelten Juwelen auf null zurückgesetzt. Durch „Save“ können die Einstellungen gespeichert werden, welche auch erhalten bleiben, wenn das Spiel beendet und neu gestartet wird. Mit „Back“ werden die vorgenommenen Einstellungen wieder verworfen.

Startet der Spieler das Spiel, gelangt er in das Menü, in dem das Flugobjekt ausgewählt wird (Abbildung 20). Hierzu kann der Spieler mittels der Pfeile zwischen den verschiedenen Flugobjekten wechseln, welche auch direkt auf dem Bildschirm angezeigt werden, damit er diese betrachten kann. Der Name des jeweiligen Flugobjekts wird ebenfalls auf dem Bildschirm angezeigt. Bestätigt der Spieler die Auswahl des Flugobjekts über „Select“, kann er den Spielmodus auswählen und damit beginnen, das Level aufzubauen.

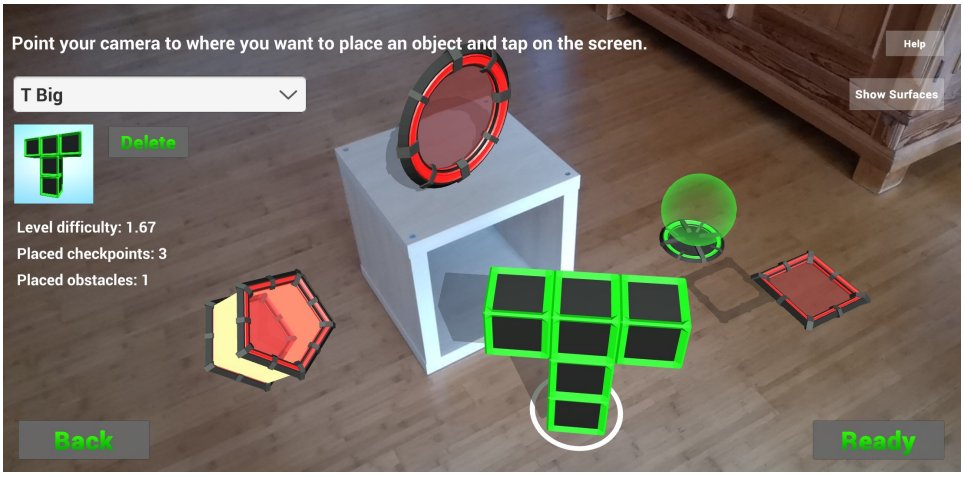


Abbildung 21: Ansicht für den Aufbau des Levels.

Hierfür wird der Levelaufbau-Modus (Abbildung 21) gestartet und das Kamerabild als Hintergrund verwendet, auf welchem Informationen als 2D UI-Elemente gerendert werden. Diese werden in einer eigenen 2D-Ebene erstellt, welche als letztes gerendert und somit zum Schluss über die gesamte Szene gelegt wird, damit die Elemente sich immer im Vordergrund befinden. Bei der Erstellung der UI-Elemente musste darauf geachtet werden, dass sich diese bei Änderung vom Portrait- in den Landschafts-Modus oder umgekehrt und auch an unterschiedliche Bildschirmauflösungen automatisch anpassen (das angezeigte Bild wird entsprechend gedreht und die Elemente neu angeordnet, je nachdem, wie der Spieler das Gerät hält). Da es sich um eine AR-Anwendung handelt, müssen sich die UI-Elemente auch immer gut vom Kamerabild abheben können, andererseits dürfen sie auch nicht zu viel des Bildes verdecken.
Am oberen Rand des Bildschirms werden Hinweise eingeblendet, wie der Spieler das Level aufbauen kann, und es erscheint ein „Help“-Button, wo weitere Informationen hierfür nachgelesen werden können. Unter diesem befindet sich ein weiterer Button, mit dem der Spieler die Visualisierung der Oberflächen aktivieren bzw. deaktivieren kann. Auf der linken Seite werden Informationen zu dem aktuellen Status des Levels angezeigt. Hier sieht der Spieler, wie viele Checkpoints und Hindernisse er schon platziert hat und welchen Schwierigkeitswert das Level aktuell besitzt, was für einen besseren Überblick sorgt. Je nachdem, ob der Spieler das Level mittels Einscannen von Imagemarkern oder über den Touchscreen aufbauen möchte, erscheint ein Menü für das Auswählen der verschiedenen zur Verfügung stehenden Objekte mit einem entsprechenden Vorschaubild. Hat der Spieler sein Level fertig aufgebaut, kann über „Ready“ das Level gestartet werden.

Abbildung 22: Ansicht für den Flugmodus.

Im Flugmodus (Abbildung 22) während des Levels erscheinen die Steuerungs-Elemente in den unteren Ecken des Bildschirms, je nachdem, welche Eingabemethode zur Steuerung der Flugobjekte gewählt wurde. Am oberen Bildschirmrand werden die eingesammelten Juwelen, der animierte Timer und der Levelscore angezeigt. Änderungen an diesen, die durch Einsammeln von Juwelen ausgelöst werden können, werden mittels eines animierten Textes angezeigt. Weiterhin erscheinen verschiedene Indikatoren auf dem Bildschirm, die den Spieler bei der Orientierung unterstützen sollen. Diese weisen in die Richtung von den entsprechenden virtuellen Objekten und zeigen deren Position an, wenn diese außerhalb des Kamerabildes sind oder von anderen Gegenständen verdeckt werden. Auch während des Levels besteht wieder die Möglichkeit, die Visualisierung der Oberflächen zu aktivieren oder zu deaktivieren. Zusätzlich kann der Spieler das Level pausieren, indem er auf den „Menu“-Button drückt, mit welchem sich ein Menü öffnet. Dort kann er das Spiel wahlweise fortsetzen, das Level zurücksetzen, den Spielmodus ändern oder zurück ins Hauptmenü gehen.
Wurde das Level erfolgreich beendet, erscheint automatisch der Endbildschirm (Abbildung 23), auf welchem dem Spieler der Levelscore, die für das Level benötigte Zeit, die Bestzeit und die eingesammelten Juwelen angezeigt werden. Um es spannender zu gestalten, werden die Errungenschaften nacheinander eingeblendet und deren Anzahlen mittels Animationen hochgezählt. Die berechnete Punktzahl, die der Spieler für das Level erhält, wird schließlich zu dessen Juwelen-Konto hinzugerechnet. Danach kann der Spieler wählen, ob er das Level neu starten, ein anderes Level aufbauen, das Flugobjekt ändern, den Modus wechseln oder zurück ins Hauptmenü gehen möchte.

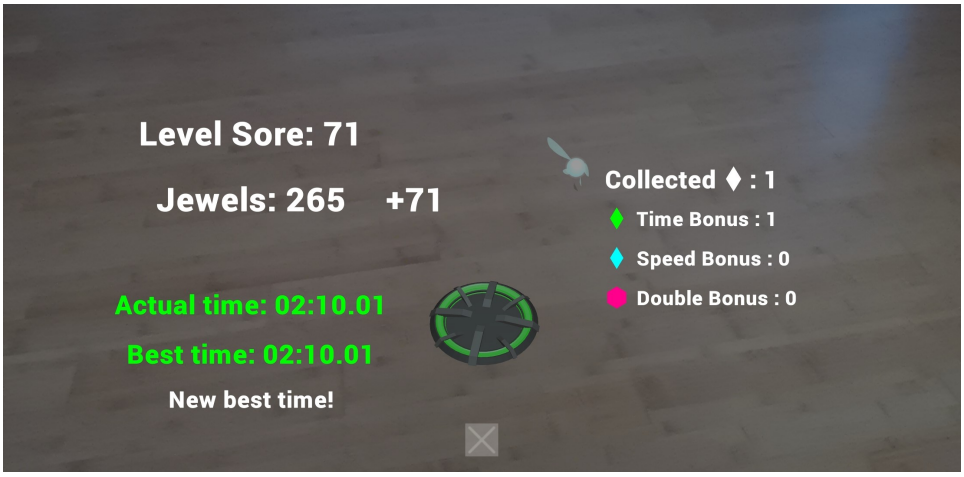


Abbildung 23: Endbildschirm mit der Wertung für das Level.

5.3 Tracking und Oberflächenerkennung

Das für AR grundlegende Tracking wird wie in Kapitel 3 beschrieben intern von ARCore übernommen. Dafür ist es allerdings notwendig, dass eine `ARCoreSession` Komponente in der Szene enthalten ist.

5.3.1 ARCore Session

Fügt man das `ARCoreDevice`-Prefab zur Szene hinzu, welches im SDK vorhanden ist, so enthält dieses bereits ein `ARCoreSession`-Skript, welches beim Starten automatisch eine `Session`-Instanz erstellt und initialisiert. Dadurch kann das Kamerabild der realen Kamera als Hintergrund der virtuellen Kamera, die ebenfalls in der Szene enthalten sein muss, gerendert werden. Diese Kamera ist die Hauptkamera (*Main Camera*) der 3D-Szene und ein Kind des `ARCoreDevice`-GameObjects. Ihre *Transformation* (lokale Position und Orientierung) entspricht der Position und Orientierung der Gerätekamera, sodass virtuelle und reale Kamera aufeinander registriert sind. Die Transformation wird hierzu zur Laufzeit von ARCore bestimmt und durch die `TrackedPoseDriver` Komponente aktualisiert. Wird die Anwendung zum ersten Mal gestartet, wird der Nutzer automatisch nach der Berechtigung für den Zugriff auf die Gerätekamera gefragt. Sollte diese Berechtigung nicht gegeben werden, kann die Anwendung nicht auf das Kamerabild zugreifen, woraus folgt, dass das Tracking nicht möglich ist.

Zusätzlich zu dem Zugriff auf die Kamera benötigt die `ARCoreSession` eine `ARCoreSessionConfig` und einen `ARCoreCameraConfigFilter`. In der `ARCoreSessionConfig` sind die Konfigurationseinstellungen der

aktuellen `Session` gespeichert. Dazu gehören der `PlaneFindingMode` (ob horizontale und/oder vertikale Oberflächen getrackt werden sollen), der `LightEstimationMode` (welche Art von Beleuchtung berechnet werden soll), der verwendete Kamera-Fokus (automatisch oder fix), die Kamera-Framerate und ob zusätzliche Funktionen wie *AugmentedImages*, *AugmentedFaces* oder *CloudAnchors* verwendet werden sollen. Diese Einstellungen können für die komplette Anwendung geändert werden.
Der `ARCoreCameraConfigFilter` definiert die Eigenschaften, welche die Anwendung vorzugsweise benutzen soll und wird dazu verwendet, zur Laufzeit eine Liste von Kamera-Konfigurationen zu erstellen, die mit dem aktuell verwendeten Gerät möglich sind, um somit automatisch die bestmögliche Konfiguration für das entsprechende Gerät auszuwählen. Dazu gehören die Kamera-Framerate und ob eine Tiefenkamera vorhanden ist und verwendet werden soll.

Die `Session`-Instanz, die von der `ARCoreSession` erstellt wird, bildet die Verbindung zwischen den ARCore-Funktionalitäten und der eigentlichen Anwendung. Sie enthält alle Informationen zum aktuellen Status von ARCore, ist für das Tracking von Oberflächen und Anchors zuständig und verwaltet alle `Trackables`. Dies ist eine Oberklasse für alle Objekte (Oberflächen, Anchors, Images), die ARCore in der realen Umgebung verfolgt. Wird die `Session`-Instanz gelöscht, werden auch alle in der Szene erkannten Oberflächen und Anchors gelöscht.

5.3.2 Oberflächenerkennung

Um ein Level aufbauen zu können, muss mindestens eine reale Oberfläche in der Umgebung erkannt worden sein, da man auf ihr die virtuellen Objekte für den Parcours platziert. Das Erkennen funktioniert in der Regel nur für annähernd horizontale oder vertikale Oberflächen, schräge oder gewölbte Oberflächen hingegen können nicht erkannt werden. Weiterhin ist zu beachten, dass Oberflächen, die gut beleuchtet und stark texturiert sind, besser gefunden werden können als welche ohne Textur oder ohne ausreichende Beleuchtung. So können zum Beispiel Fliesen oder gemaserte Holzböden besser erkannt werden als eine glatte weiße Wand oder durchsichtige Glasoberflächen.
Der Grund dafür ist, dass die Oberflächenerkennung ARCore-intern mittels merkmalsbasiertem Tracking geschieht, wie in Kapitel 3 beschrieben. Dazu wird das Kamerabild nach visuell eindeutigen Merkmalen abgesucht, welche als `FeaturePoint` abgespeichert werden. Diese sind eine Spezialisierung der `Trackable`-Klasse und stehen jeweils für einen Punkt, den ARCore in der realen Welt verfolgt. Wurden genügend Feature Points gefunden, die auf einer (horizontalen oder vertikalen) Ebene zu liegen scheinen, wird automatisch ein `DetectedPlane`-Objekt erzeugt, welches eben-

falls von `Trackable` erbt. Dies sind letztendlich die virtuellen Abbildungen der planaren Oberflächen, welche von ARCore in der realen Umgebung erkannt wurden. Diese Oberflächenobjekte besitzen jeweils einen `PlaneType` (horizontal oder vertikal) und Informationen über ihren Mittelpunkt, ihre Ausdehnung und ihr `BoundaryPolygon`, also die äußeren Randpunkte, welche die Form der Oberfläche definieren.

Da die Oberflächenerkennung während der gesamten ARCore Session ausgeführt wird, wodurch sich das Verständnis der Umgebung zur Laufzeit erweitert, ist es auch möglich, dass zuerst mehrere einzelne Cluster von Feature Points gefunden und zu voneinander getrennten Planes zusammengefasst werden, auch wenn sie eigentlich zur selben Oberfläche gehören. Merkt das System später, dass diese doch zusammen gehören, werden die entsprechenden Oberflächenobjekte zu einer einzigen `DetectedPlane` verschmolzen.
Die Oberflächen sind nach der Erkennung intern als Trackables gespeichert und werden entsprechen von ARCore weiter verfolgt und aktualisiert, jedoch ist es zu diesem Zeitpunkt noch nicht möglich, mit diesen zu interagieren. Hierzu müssen zuerst entsprechende virtuelle Oberflächenobjekte in der Szene generiert und gegebenenfalls visualisiert werden. Diese dienen als virtueller Gegenpart für die realen Oberflächen, die an derselben Stelle verortet werden, und erfüllen somit den Zweck von *Phantomobjekten* (engl. *phantom objects*) [36] (mehr dazu in Abschnitt 5.8).

5.3.3 Generierung und Visualisierung der Oberflächen

Zur Generierung der virtuellen Oberflächen-GameObjects wurde ein `SurfaceGenerator` erstellt und in die Szene eingefügt. Dieser überprüft für jeden Update()-Aufruf, ob neue Oberflächen gefunden wurden und instantiiert für jede in diesem Frame neu erkannte Oberfläche ein entsprechendes `ARSurface` GameObject. Da auf der einen Seite die Visualisierung der virtuellen Oberflächen die Immersion stören kann, es aber auf der anderen Seite schwierig ist, Objekte auf Oberflächen zu platzieren, wenn man diese nicht sehen kann bzw. wenn man nicht sehen kann, ob ARCore die Oberfläche schon erkannt hat, wollte ich dem Spieler die Möglichkeit bieten, die Visualisierung der virtuellen Oberflächen an- und ausschalten zu können. Die Visualisierung kann sowohl in den Einstellungen im Hauptmenü als auch während des Trackings in der Szene selber über einen Button umgeschaltet werden. Im `SurfaceGenerator` wird daraufhin nach der Instantiierung eines Klons des `ARSurface`-Prefabs abgefragt, ob die Oberflächen aktuell visualisiert werden sollen oder nicht, und dementsprechend wird die `ARSurface` mit einem von zwei speziell für diesem Zweck geschriebenen Shadern initialisiert.

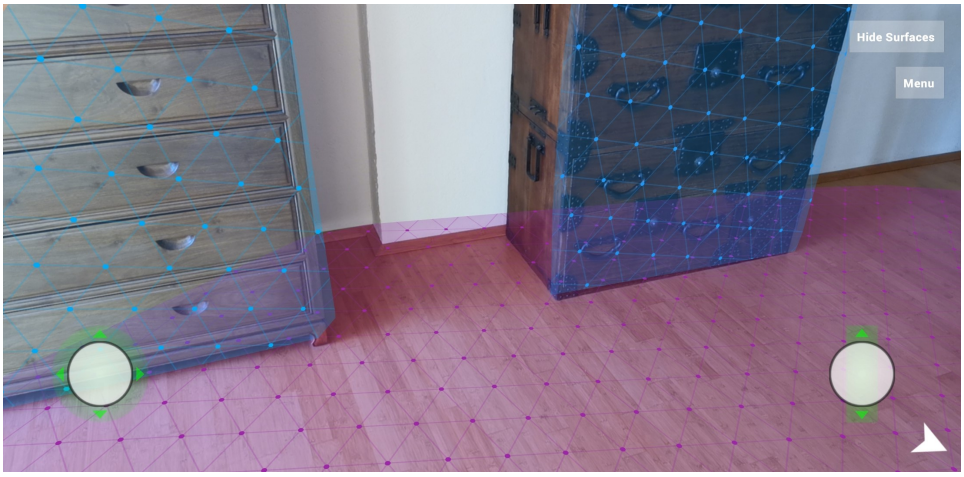


Abbildung 24: Die vom System erkannten Oberflächen werden durch ein halbtransparentes Grid visualisiert.

Eine `ARSurface` ist eine virtuelle Abbildung einer realen Oberfläche, welche jeweils einer `DetectedPlane` zugeordnet ist, was bedeutet, dass sie nur solange existiert wie auch die `DetectedPlane`. Dabei handelt es sich um ein primitives `Plane`-Objekt, dessen *Mesh Vertices* (also die Eckpunkte des Polygonnetzes des `Plane`-Modells) durch das `BoundaryPolygon` der `DetectedPlane` definiert werden. Dieses wird mittels *Mesh Renderer* (Komponente zur Darstellung des Meshs) und einem Material, welchem immer der entsprechende Shader zur Visualisierung zugewiesen wird, dargestellt. Zusätzlich besitzt die `ARSurface` einen *Mesh Collider* (Kollider-Komponente in der Form des Meshs des Objekts), damit die Interaktion mit ihr möglich ist. In jedem Frame überprüft die `ARSurface`, ob sich die ihr zugeordnete `DetectedPlane` bzw. deren `BoundaryPolygon` verändert hat. Ist dies der Fall, werden die Mesh Vertices und somit die Größe und Form der `ARSurface` entsprechend aktualisiert.

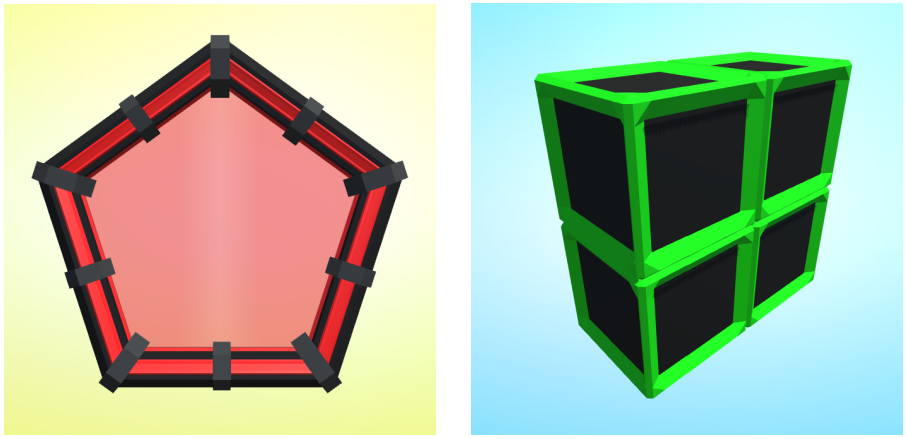
Soll die virtuelle Oberfläche visualisiert werden, wird für das Oberflächenmaterial der `PlaneGridShader` verwendet. Dieser stellt die Oberfläche durch ein halbtransparentes Grid dar (Abbildung 24), sodass der von ARCore erkannte Bereich deutlich sichtbar ist und die reale Oberfläche darunter gleichzeitig immer noch wahrgenommen werden kann. Sollen die virtuellen Oberflächen komplett unsichtbar (aber immer noch vorhanden!) sein, wird der `ARSurfaceShader` verwendet. Diese beiden Shader und ihre Funktion werden in Abschnitt 5.8 genauer erläutert. Sollte die Visualisierungsart der Oberflächen zwischendurch durch den Nutzer geändert werden, so geschieht dies über den `SurfaceGenerator`. Dieser geht in einer Schleife über alle in der Szene enthaltenen virtuellen Oberflächenobjekte und tauscht den Shader für jede `ARSurface` zur Laufzeit aus.

5.4 Aufbau der Level

Nachdem mindestens eine reale Oberfläche in der Umgebung erkannt worden ist, kann damit begonnen werden, den Parcours aufzubauen, indem die virtuellen Parcours-Objekte auf dieser platziert werden. Das ist auf allen erkannten Oberflächen möglich, also sowohl auf horizontalen als auch auf vertikalen `ARSurface`-Objekten. Alle zum Parcours gehörenden und der virtuellen Szene hinzugefügten Objekte werden dabei als Kinder des `Level`-GameObjects registriert. Dieses ist durch das angefügte Skript `GameplayController` für die Spiellogik und die Verwaltung des Levels zuständig (Abschnitt 5.5).

Ein Level benötigt in jedem Fall ein `StartField` (Startfeld) und mindestens einen `Checkpoint` (im *pARcours*-Modus). Zusätzlich dazu können weitere Checkpoints und/oder `Obstacle`-Objekte (Hindernisse) hinzugefügt werden. Versucht der Spieler, ein Level zu starten, ohne dass die benötigten Komponenten vorhanden sind, so erscheint auf dem Bildschirm ein Hinweis, welche Objekte noch platziert werden müssen, um fortfahren zu können. Soll ein Parcours-Objekt der Szene hinzugefügt werden, wird zuerst an der entsprechenden Stelle auf der Oberfläche ein `Anchor` erstellt. Die Ermittlung der Zielposition für das Objekt ist dabei abhängig von der gewählten Art der Eingabe (siehe Abschnitte 5.4.1 und 5.4.2). Der `Anchor` ist dafür zuständig, die Position des virtuellen Objekts auf der Oberfläche zu registrieren und es an dem entsprechenden `Trackable`, der `ARSurface`, zu verankern. Wird nun im Laufe des Spiels die Transformation der `ARSurface` aufgrund neuer Umgebungsinformationen aktualisiert, werden auch gleichzeitig die daran verankerten Anchors angepasst und somit auch die virtuellen Objekte, welche ihrerseits als Kinder der `Anchor`-GameObjects instantiiert werden. Somit werden alle im Level enthaltenen Objekte in der Szene verankert, wodurch ihre Registrierung in der realen Umgebung auch bei Bewegung des Nutzers weitestgehend korrekt und stabil bleibt. Zusätzlich wird jedes im Level platzierte Objekt bei seiner Instantiierung im `GameplayController` registriert, damit dieser die Spiellogik im Level übernehmen kann. Durch diesen dynamischen Levelaufbau müssen die verschiedenen Parcours nicht vorab gespeichert sein, wodurch sich der Speicherverbrauch verringert und keine Ladezeiten auftreten (um ein Level zu laden).

Ein Problem, welches auftritt, wenn man ein virtuelles Objekt auf einer `ARSurface` platziert, besteht darin, dass sich die unten liegende Fläche des Objekts und die Oberfläche überlappen, was sich in einem abwechselnden Flackern der zwei Flächen äußert, welche genau an derselben Stelle dargestellt werden sollen. Der Renderer ist dann nicht dazu in der Lage zu entscheiden, welche der Flächen pro Frame gerendert werden soll. Das Pro-



(a) Beispiel Checkpoint (b) Beispiel Hindernis

Abbildung 25: Zwei Beispiele für Imagemarker, auf welchen die entsprechenden virtuellen Objekte zu sehen sind.

blem wurde dadurch behoben, dass jede `ARSurface` bei ihrer Erstellung minimal um (übertragen auf die Realität) einem Millimeter nach unten verschoben wurde. Dieser Abstand ist zu klein, um ihn während des Spielens zu bemerken, genügt aber dennoch, dass der Renderer nur die Unterseite des Objekts darstellt und der entsprechend darunterliegende Teil der Oberfläche nicht mehr zu sehen ist.

Der Spieler kann die virtuellen Objekte auf zwei verschiedene Arten in der Umgebung platzieren: durch Einscannen von Imagemarkern oder über den Touchscreen des Geräts. So kann er selber entscheiden, auf welche Weise er das Level aufbauen möchte. Hat der Spieler zum Beispiel gerade keine Imagemarker zur Hand, kann er alternativ auch nur den Touch-Input verwenden. Die beiden Varianten werden im Folgenden näher erläutert.

5.4.1 Objektplatzierung mittels Imagemarker

Die Idee hinter dieser Variante ist relativ simpel: Jeder Imagemarker steht stellvertretend für ein virtuelles Objekt, welches im Level platziert werden kann. Verteilt der Spieler die Marker in der Umgebung und scannt diese mit der Kamera ein, so erscheinen die entsprechenden Objekte auf den Bildern und der Parcours ist aufgebaut. Dabei orientieren sich die Position und Rotation der virtuellen Objekte an denen der Imagemarker, wodurch sie sich durch die Bewegung dieser manipulieren lassen. Auf den selbst erstellten Imagemarkern sind die korrespondierenden Objekte abgebildet, damit der Spieler weiß, welches Objekt auf dem entsprechenden Marker erscheint (Abbildung 25).

Damit ARCore nach Imagemarkern im Kamerabild suchen kann, muss zunächst eine `AugmentedImageDatabase` erstellt werden. Dies ist eine Datenbank, in der alle Bilder gelistet sind, die ARCore detektieren und verfolgen soll. In ihr können bis zu 1000 Bilder gleichzeitig gespeichert werden, jedoch nimmt die Trackinggeschwindigkeit bei wachsender Anzahl gespeicherter Bilder zunehmend ab, da im schlechtesten Fall jedes Mal alle Bilder durchgetestet werden müssen, wenn die Möglichkeit besteht, dass ein Imagemarker gefunden wurde. Außerdem kann zu jedem Zeitpunkt immer nur eine `AugmentedImageDatabase` in Benutzung sein. Jeder Eintrag in der Datenbank besitzt einen eindeutigen Index, mit welchem auf das ihm zugeordnete `AugmentedImage` zugegriffen werden kann.
`AugmentedImage` ist eine Klasse für die von ARCore in der realen Welt erkannten Bilder, die von `Trackable` erbt, somit vom System verfolgt und aktualisiert wird und ähnlich wie ein `Anchor` als Ankerpunkt für virtuelle Objekte dienen kann. Für jeden Imagemarker, welcher (zum ersten Mal in der Session) in der Umgebung erkannt wurde, wird eine `AugmentedImage`-Instanz zurückgegeben, welche die Informationen zu dem jeweiligen Bild enthält. Dazu gehören die `CenterPose` (Mittelpunkt und Orientierung des Bildes), Name und Größe sowie der entsprechende Datenbank-Index für die Zuordnung. Solange die Pose und physische Größe des Imagemarkers vom System noch nicht bestimmt werden konnte (aufgrund zu weniger Umgebungsinformationen), pausiert das Tracking derjenigen `AugmentedImage`-Instanz. Erst wenn diese Eigenschaften bestimmt wurden, kann der Imagemarker verfolgt und ein virtuelles Objekt darauf angezeigt werden.

Die von mir erstellte Klasse `ImageMarkerTracker` ist für die korrekte Zuordnung und Platzierung der virtuellen Objekte auf den entsprechenden Imagemarkern zuständig. Hierfür benötigt sie eine Liste aller für den Parcours zur Verfügung stehenden Objekte, welche für eine leichtere Zuordnung in derselben Reihenfolge eingetragen wurden wie die zugehörigen Bilder in der `AugmentedImageDatabase`, und somit denselben Index besitzen. In jedem Update()-Aufruf werden über die `Session` die `AugmentedImage`-Instanzen abgefragt, welche in diesem Frame neu erkannt oder aktualisiert wurden. Denjenigen, die zum ersten Mal registriert wurden und die aktuell verfolgt werden, wird das entsprechende virtuelle Objekt zugeordnet. Hierzu wird zuerst ein `Anchor` an der `CenterPose` von dem jeweiligen `AugmentedImage` erstellt. Dadurch ist gewährleistet, dass das Objekt immer auf dem Mittelpunkt des Imagemarkers verankert bleibt. Danach wird anhand des `DatabaseIndex` des Bildes das entsprechende Objekt aus der Liste aller Parcours-Objekte ausgewählt und unter dem `Anchor` instantiiert. Zum Schluss wird das Objekt noch im `GameplayController` registriert und in einem `Dictionary` an dem jeweiligen Bildindex eingetragen, damit dem System eine eindeutige Zuord-

Abbildung 26: Die virtuellen Objekte werden nach dem Einscannen auf dem entsprechenden Imagemarker dargestellt.

nung gegeben werden kann. In Abbildung 26 sind drei virtuelle Objekte zu sehen, die mit Hilfe von Imagemarkern in der Szene platziert wurden.
Für Imagemarker, die nicht mehr aktiv verfolgt werden (weil sie zum Beispiel nicht mehr im Kamerabild zu sehen sind), wird das Modell des virtuellen Objekts deaktiviert, um Rechenaufwand zu sparen. Da es aber möglich ist, dass diese zu einem späteren Zeitpunkt wieder aktiv verfolgt werden können, werden die Objekte nicht gelöscht. Wird das Tracking für einen Imagemarker hingegen komplett beendet, wird das entsprechende GameObject aus der Szene gelöscht und durch den `GameplayController` aus dem Level entfernt.

Damit ARCore einen Imagemarker gut identifizieren kann, sollte dieser möglichst viele eindeutige Merkmale besitzen, vollständig im Kamerabild zu sehen sein und mindestens 25% des Kamerabildes ausmachen. Ist er zu weit weg, teilweise verdeckt oder ist der Betrachtungswinkel zu groß, werden zu wenige Feature Points identifiziert und das Bild somit nicht erkannt. Außerdem kann das Einscannen erschwert werden, wenn die Beleuchtung nicht ausreichend ist. Eine weitere große Einschränkung bei der Verwendung von *Augmented Images* ist, dass maximal 20 Imagemarker simultan

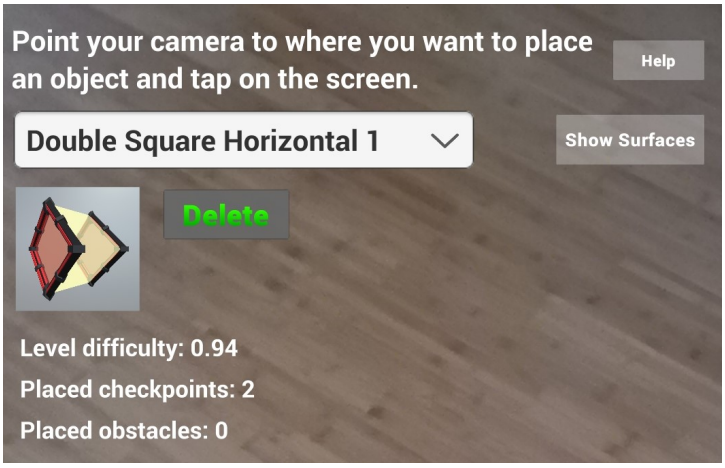


Abbildung 27: Der Spieler kann die virtuellen Objekte, die er im Level platzieren möchte, über das Dropdown-Menü auswählen.

verfolgt werden können (sowohl im Kamerabild als auch außerhalb) und dass es nicht möglich ist, mehrere `AugmentedImage`-Instanzen für dasselbe Bild zu erstellen. Versucht man, zwei identische Imagemarker parallel einzuscannen, so springt das virtuelle Objekt einfach nur zwischen diesen beiden hin und her. Im Fall meines Spiels ist diese Einschränkung aber weniger schlimm, da man zusätzlich zu den Imagemarkern die Objekte auch noch mittels Touch-Input platzieren kann. Bei dieser Variante gibt es keine Einschränkungen für die Anzahl an (identischen) Objekten.

5.4.2 Objektplatzierung mittels Touch-Input

Platziert man die virtuellen Parcours-Objekte mittels Touch-Input in der Szene, werden keine Imagemarker benötigt, da die gesamte Interaktion über den Touchscreen des Geräts vorgenommen wird. Hierfür muss lediglich auf eine Stelle einer vom System erkannten Oberfläche getippt werden, damit dort das ausgewählte Objekt erscheint. Um die verschiedenen Objekte auswählen zu können, erscheint in diesem Objektplatzierungs-Modus ein Dropdown-Menü auf dem Bildschirm, welches alle für das Level zur Verfügung stehenden Objekte beinhaltet (Abbildung 27). Wird aus dieser Liste ein Objekt durch Tippen ausgewählt, erscheint das entsprechende Objekt auf einem Vorschaubild unter dem Dropdown-Menü, damit der Spieler weiß, wie das Objekt aussieht, welches er platzieren möchte.

Die Bestimmung der Zielposition für das virtuelle Objekt in der realen Welt und die Auswahl des korrekten Modells übernimmt die Klasse `ObjectivesManipulator`. Diese besitzt wie der `ImageMarkerTracker` eine Liste aller Parcours-Objekte und zusätzlich dazu einen Indexwert für

das aktuell ausgewählte Objekt, welcher durch die Auswahl über das Dropdown-Menü geändert wird. Dieser Index wird beim Starten des Levelaufbaus immer auf null gesetzt, den Index für das Startfeld, sodass dieses zuerst ausgewählt ist, da es in jedem Level enthalten sein muss. Jedes Mal, wenn eine `TapGesture` (dt. Tipp-Geste) auf dem Bildschirm ausgeführt wird, überprüft der `ObjectivesManipulator`, ob diese Geste ein schon in der Szene existierendes Objekt oder ein UI-Element (wie zum Beispiel das Dropdown-Menü) zum Ziel hat. Ist beides nicht der Fall, kann mit der Objektplatzierung begonnen werden.

Platzierung eines Objekts Die Bestimmung der Zielposition im dreidimensionalen Raum anhand einer Tap-Geste auf dem 2D-Bildschirm erfolgt mittels *Raycasting*. Für den *Hit Test* wird ein Strahl (engl. *ray*) von der Stelle aus, wo der Spieler den Bildschirm berührt hat, geradeaus in Blickrichtung der Kamera in die Szene geschickt. Für diesen Strahl werden die Schnittpunkte mit virtuellen Objekten oder Oberflächen berechnet (falls vorhanden), wobei der (von der Kamera aus gesehen) vorderste Schnittpunkt den Zielpunkt darstellt. Wurde der vorderste Schnittpunkt ermittelt, kann die Berechnung weiterer Schnittpunkte entfallen, da diese dahinter liegen und somit vom Zielobjekt verdeckt werden müssten.
Hat der Strahl einen Gegenstand getroffen bzw. wurde ein Schnittpunkt gefunden, muss noch überprüft werden, ob an dieser Stelle ein Objekt platziert werden kann. Da die Objekte einen Ankerpunkt in der realen Welt benötigen, können diese nur auf einem `Trackable` (also auf einer Oberfläche oder einem Feature/Oriented Point) platziert werden. Daher muss zuerst überprüft werden, ob es sich bei dem mit dem Strahl getroffenen Objekt um ein solches handelt. Ist dies der Fall, wird mit Hilfe des Index das entsprechende Modell aus der Objektliste ausgewählt und an der Zielposition testweise instantiiert. Dabei wird überprüft, ob sich das Objekt mit einem anderen schon im Level befindlichen GameObject überschneidet. Dies geschieht in einer Schleife über alle `Collider` der Objekte, die schon in der Szene platziert wurden. Ein `Collider` (dt. Kollider) ist eine Komponente, die ähnlich wie eine Bounding Box ein Objekt umgibt und dazu genutzt wird, Kollisionen mit anderen Objekten, die ebenfalls einen Kollider besitzen, zu erkennen. Jeder `Collider` des zu platzierenden GameObjects wird bei dem Test mit jedem `Collider` von jedem anderen GameObject auf Überschneidungen getestet. Wird auch nur eine solche gefunden, wird das Modell wieder aus der Szene entfernt und es erscheint ein Hinweis auf dem Bildschirm, dass die Zielposition des Objekts zu nah an einem anderen Objekt gewesen ist.

Dieser Test ist dazu gedacht zu verhindern, dass die Objekte an ein und derselben Stelle aufgestellt werden können, wodurch ihre Modelle inein-

ander verschachtelt wären und der Parcours somit möglicherweise nicht mehr zu absolvieren wäre. Dabei ist allerdings das Problem aufgetreten, dass das Platzieren der Objekte auf den Oberflächen durch die Kollider derselben Oberflächen, welche für die Kollisionserkennung (Abschnitt 5.8.3) benötigt werden, verhindert wurde, da der Test immer negativ war. Deshalb werden die Oberflächenkollider während des Levelaufbaus deaktiviert und erst dann wieder aktiviert, wenn das Level gestartet wurde.
Wurde bei diesem Test keine Überlappung mit einem anderen GameObject gefunden, wird an der Zielposition ein `Anchor` für das Objekt erstellt, um dieses an der Oberfläche zu verankern, und das entsprechende Objekt wird im `GameplayController` registriert. Zusätzlich muss ein `Manipulator` für das Objekt erstellt werden, wenn die Möglichkeit gegeben sein soll, dieses zu manipulieren.

Objektmanipulation Damit die Manipulation der virtuellen Objekte ermöglicht werden kann, muss jedes dieser Objekte von einer eigenen `Manipulator`-Komponente angesprochen werden können, die ARCore Basisklasse für das Erkennen von Touch-Gesten. Hierfür wird das zu platzierende Objekt als Kind eines `Manipulator`-Prefabs und dieser als Kind unter dem `Anchor` instantiiert. Um die verschiedenen `Manipulator`-Objekte zu managen, wurde das ebenfalls von ARCore zur Verfügung gestellte `ManipulationSystem`-Prefab in die Szene integriert. Zusammen ermöglichen sie dem System die Erkennung von verschiedenen Touch-Gesten auf dem Bildschirm (*Tap* (dt. tippen), *Drag* (dt. ziehen), *Twist* (dt. drehen), *Swipe* (dt. wischen)), sodass der Nutzer virtuelle Objekte mit diesen manipulieren kann. Dazu gehören das horizontale und vertikale Verschieben, das Rotieren und Skalieren von Objekten.

Über das `ManipulationSystem` wird zudem das aktuell ausgewählte Objekt angesprochen, welches zur besseren Erkennung mit einem weißen Ring umgeben wird (Abbildung 28). Hat die Tap-Geste bei der Objektplatzierung ein schon in der Szene existierendes Objekt zum Ziel, so ermöglicht der `Manipulator`, dass dieses Objekt ausgewählt wird (wenn es das vorher noch nicht war) und mittels Touch-Gesten manipuliert werden kann. Zudem ist es sinnvoll, Objekte wieder aus dem Level löschen zu können, wenn sie beispielsweise unbeabsichtigt platziert worden sind. Hierzu wird auf dem Bildschirm ein „Delete“-Button zur Verfügung gestellt. Dieser kann betätigt werden, wenn ein bestimmtes Objekt ausgewählt wurde, woraufhin dieses aus der Szene und aus dem `GameplayController` entfernt wird. Bei der Löschung eines Objekt muss darauf geachtet werden, dass auch der zugehörige `Anchor` und `Manipulator` entfernt werden, da das System diese sonst weiterhin verfolgt.

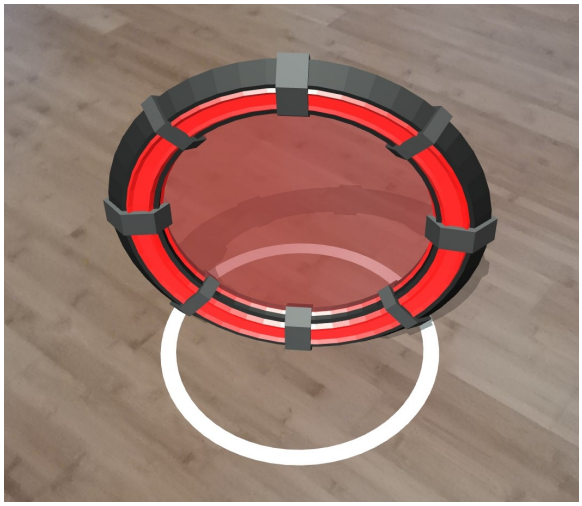


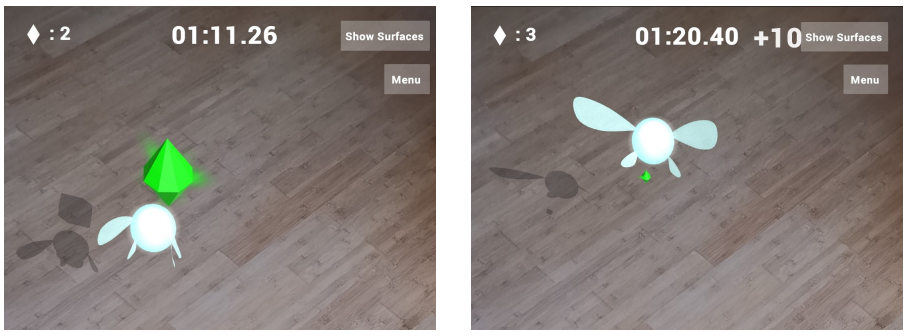
Abbildung 28: Ein ausgewähltes Objekt wird mit einem weißen Ring markiert.

5.5 Spiellogik

Alle Objekte eines Levels werden in der Szene unter dem zentralen `Level`-GameObject erstellt. An diesem ist das `GameplayController`-Skript angefügt, welches für die Verwaltung aller Parcours-Objekte, des Levelscores und der eingesammelten Juwelen, die Steuerung des Timers und die eigentliche Spiellogik im Level verantwortlich ist. Hierfür greift er auch auf weitere für das Spiel zentrale Skripts zu, welche ich in den nächsten Abschnitten erläutern werde. Danach gehe ich noch genauer auf das `GameplayController`-Skript ein, während ich den Spielablauf für die Spielmodi aus der technischen Sicht beschreibe.

5.5.1 Timer

Der `Timer` übernimmt in diesem Spiel die Funktion einer Stoppuhr und zählt demnach abhängig vom ausgewählten Spielmodus die Zeit entweder von null hoch (*pARcours*-Modus) oder von einer bestimmten Startzeit runter (*ARcade*-Modus). Der Timer beginnt zu zählen, sobald der Spieler das Startfeld mit seinem Flugobjekt verlassen hat. Damit dieser seine aktuelle Zeit zu jedem Zeitpunkt im Blick hat, wird sie während des Levels am oberen Bildschirmrand angezeigt (Abbildung 29).
Über das `Timer`-Skript kann der Timer von außen gesteuert werden. So ist es möglich, den Timer vom `GameplayController` aus zu starten, zu pausieren, zu stoppen und zurückzusetzen. Weiterhin ist es möglich, auf die aktuelle Zeit einen Zeitbonus oder eine Zeitstrafe zu addieren, wenn ein entsprechendes Juwel eingesammelt wurde. Solche Änderungen der Zeit werden durch einen animierten Text neben dem Timer angezeigt. Au-



(a) Timer vor Einsammeln des Juwels (b) Timer nach Einsammeln des Juwels

Abbildung 29: Der Timer zeigt je nach Spielmodus die schon vergangene oder die noch übrige Zeit in Minuten, Sekunden und Millisekunden an (a). Änderungen der Zeit, die durch das Einsammeln von Juwelen erfolgen, erscheinen neben dem Timer als animierter Text (b).

ßerdem ist der `Timer` für das Speichern der Bestzeit zuständig. Sobald der Timer stoppt, wenn das Level beendet wurde, wird die aktuell benötigte Zeit mit der vorherigen Bestzeit des Levels verglichen. Wenn eine neue Bestzeit aufgestellt wurde, wird dies auf dem Endbildschirm angezeigt und die neue Bestzeit gespeichert.

5.5.2 TokenSpawner

Abhängig vom ausgewählten Spielmodus und Schwierigkeitsgrad ist es möglich, dass um den Spieler herum an zufälligen Positionen Juwelen auftauchen, welche eingesammelt werden können, um spezielle Boni zu erhalten (mehr zu den verschiedenen Effekten der Juwelen in Abschnitt 4.3.2). Diese verschwinden nach kurzer Zeit jedoch wieder, wenn sie nicht eingesammelt werden. Für das *Spawnen* (dt. Erzeugen) der Juwelen ist das `TokenSpawner`-Skript zuständig. Dieses enthält für die unterschiedlichen Kombinationen aus Spielmodus und Schwierigkeitsgrad verschiedene vordefinierte Listen von `Token`-Prefabs, damit in den unterschiedlichen Situationen die richtigen Arten von Juwelen mit der richtigen Wahrscheinlichkeit auftauchen können (`Token` ist eine Oberklasse für die verschiedenen Juwelenarten). Die verschiedenen Juwelen werden in einer Subroutine erzeugt, welche vom `GameplayController` aus gesteuert werden kann. Sie erscheinen nach dem Start in zufälligen Zeitabständen (welche aber in einem vorgegebenen Zeitintervall abhängig vom Spielmodus liegen) an einer zufälligen Position innerhalb einer gewissen `spawnRange`. Sobald die maximale Anzahl an Juwelen, welche im Skript für die verschiedenen Modi festgelegt ist, im Level erreicht wurde, wird gewartet, bis wieder welche verschwunden sind, bevor neue Juwelen erzeugt werden.

Die `spawnRange` ist eine quaderförmige Zone, in der die Juwelen auftauchen können. Sie wird global skaliert und befindet sich immer um das Flugobjekt herum, sodass sie sich mit dem Flugobjekt mit bewegt und dieses zu jedem Zeitpunkt im Zentrum liegt. Der Grund dafür ist, dass der Spieler nicht auf einen bestimmten Bereich beschränkt sein, sondern sich mit seinem Flugobjekt frei im Raum bewegen können soll. Wäre die `spawnRange` fest im Raum definiert, wäre der Spieler auf diesen Bereich beschränkt, und würde man die `spawnRange` überdimensional groß machen, wäre der abgedeckte Bereich zwar riesig, aber durch die zufällige Verteilung wäre es unwahrscheinlich, dass die Juwelen in der Nähe des Spielers auftauchen. Durch eine dynamische `spawnRange` hingegen wird abgesichert, dass die Juwelen immer in der Nähe des Spielers erscheinen und dass trotzdem der komplette Bereich abgedeckt wird, in dem sich der Spieler bewegen kann (d. h. theoretisch unendlich). Die einzige Beschränkung der `spawnRange` sind die realen Oberflächen. Damit die Juwelen nicht unter dem Boden erscheinen können, wo sie unerreichbar wären, wird die `spawnRange` nach unten hin von dem y-Wert der tiefsten in der Szene gefundenen Oberfläche beschränkt.

Für die Instantiierung des Juwels wird zuerst unter dem zentralen `Level`-GameObject ein `Anchor` in der Szene an der entsprechenden Zielposition erstellt. Danach wird ein zufälliges Juwel aus der zum Spielmodus und Schwierigkeitsgrad passenden `Token`-Liste ausgewählt und als Kind des Anchors instantiiert. Die Wahrscheinlichkeit des Erscheinens eines bestimmten Juwels kann hierbei durch das Anzahl-Verhältnis dieses Juwels in der entsprechenden Liste beeinflusst werden. Betrachtet man zum Beispiel die `Token`-Liste für den *pARcours*-Modus bei normalem Schwierigkeitsgrad in Abbildung 30, so erkennt man, dass die Liste insgesamt zehn Juwelen enthält und dass das *SpeedBonus*-Juwel dort sechs Mal vorkommt. Dadurch beträgt die Wahrscheinlichkeit pro Spawn-Durchgang, dass dieses spezielle Juwel im Level erscheint, 60%. Möchte man, dass eine bestimmte Juwelenart gar nicht im Level vorkommt, wird sie einfach nicht in die entsprechende Liste eingefügt. Eine Besonderheit beim *ARcade*-Modus ist hierbei, dass dort als jedes zweite Juwel ein *TimeBonus*-Juwel erscheint, damit der Modus spielbar ist (siehe Abschnitt 5.5.5)
Weiterhin wird bei der Instantiierung für jedes Juwel getestet, ob es an der zufällig bestimmten Position einen Mindestabstand zu allen anderen Objekten in der Szene einhalten würde. Das ist wichtig, damit Juwelen nicht in oder zu nah an Parcours-Objekten oder Oberflächen erscheinen, weil sie sonst gar nicht oder nur sehr schwer einzusammeln wären, wenn man eine Kollision vermeiden möchte. Bei dem Test wird für jeden anderen `Collider` in der Szene überprüft, ob ob die Distanz zu dem `Collider` des Juwels ausreichend ist. Ist das Juwel an der entsprechenden Zielposition zu nah an einem anderen Objekt, wird es direkt wieder zerstört.

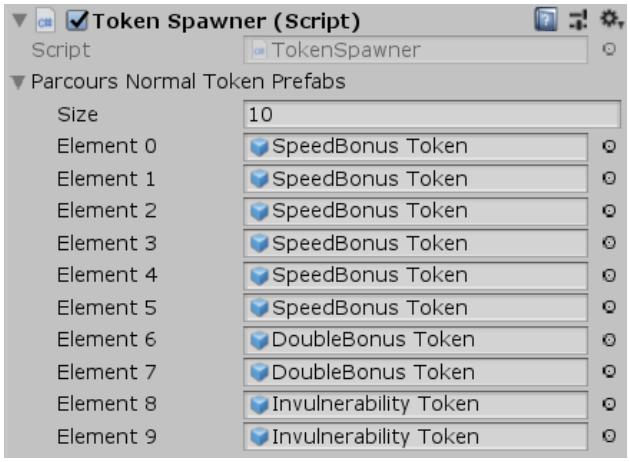


Abbildung 30: Die `Token`-Liste für den *pARcours*-Modus bei normalem Schwierigkeitsgrad. Sie enthält Prefabs für die verschiedenen Juwelenarten, die im Level erscheinen können.

5.5.3 Indikatoren

Die Indikatoren, welche vom `IndicatorArrow`-Skript erstellt und verwaltet werden, sind visuelle Hinweise für den Spieler, die auf dem Bildschirm angezeigt werden und bei der Navigation helfen sollen, falls ein Objekt von anderen Gegenständen verdeckt wird oder es ganz aus dem Kamerabild verschwindet. Ohne Indikatoren wäre es für den Spieler schwieriger zu wissen, wo er als nächstes hinfliegen soll, oder das Flugobjekt wiederzufinden, wenn er es aus den Augen verloren hat. Sie erscheinen nach Start des Levels und weisen dem Spieler die Richtung, sind aber auch in den Optionen im Hauptmenü deaktivierbar, falls sie als störend empfunden werden.
Es gibt grundsätzlich zwei Arten von Indikatoren, welche wiederum für die verschiedenen Zielobjekte unterschiedlich eingefärbt werden: Die *Onscreen*-Indikatoren und die *Offscreen*-Indikatoren (Abbildung 31). Sie werden, wenn erforderlich, für das Flugobjekt (weiß), den nächsten Checkpoint, der durchflogen werden muss (rot) und die verschiedenen Juwelen im Level (in der Farbe des Juwels) angezeigt, solange diese nicht (vollständig) im Kamerabild sichtbar sind. Befinden sich die Objekte wieder im Kamerabild, wird die Visualisierung der entsprechenden Indikatoren deaktiviert.

Onscreen-Indikator Der Onscreen-Indikator dient dazu, die Position von Objekten, die sich zwar im Kamerabild befinden, aber dennoch nicht sichtbar sind, weil sie von einem anderen Gegenstand verdeckt werden, für den Spieler erkenntlich zu machen. So ist es möglich, diese wieder ins Sichtfeld

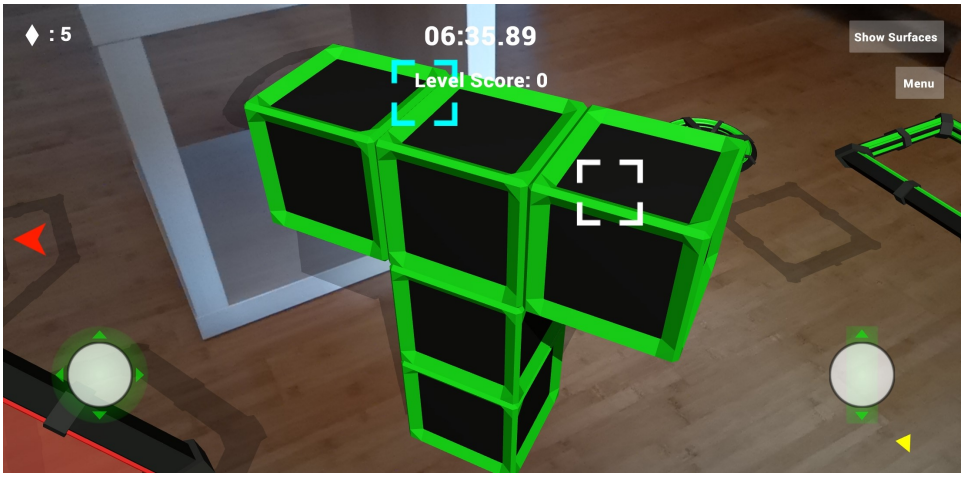


Abbildung 31: Die Indikatoren zeigen die Positionen der verschiedenen virtuellen Objekte an. Die Offscreen-Indikatoren werden als Pfeile am Bildschirmrand dargestellt und weisen in die entsprechende Richtung, die Onscreen-Indikatoren erscheinen als vier eingrenzende Ecken an der Position des verdeckten Objekts.

zu steuern, ohne sie dabei wirklich sehen zu können. Der Indikator für das entsprechende Objekt wird dabei als 2D-Sprite an derselben Stelle wie dieses angezeigt, nur dass es sich direkt auf dem UI-Layer des Bildschirms befindet und daher nicht durch die Szenenobjekte verdeckt werden kann. Um die Position auf dem Bildschirm zu berechnen, an welcher der Indikator angezeigt werden soll, wird zunächst mittels `WorldToScreenPoint()` die 3D-Position des Objekts auf die 2D-Koordinaten des Bildschirms übertragen. Anhand dieser `screenPos` (Bildschirmposition) kann überprüft werden, ob sich das Objekt im Kamerabild (Onscreen) oder außerhalb (Offscreen) befindet. Onscreen bedeutet, dass die x- und y-Werte der `screenPos` größer als null und kleiner als die Höhe bzw. Breite des Bildschirms sein müssen und dass der z-Wert positiv sein muss, um im Bereich vor der Kamera zu liegen. Ein negativer z-Wert würde bedeuten, dass sich das Objekt hinter der Kamera und somit nicht im Kamerabild befindet.
Wurde festgestellt, dass sich das Objekt im Kamerabild befindet, wird mit Hilfe von Raycasting ein Strahl von der Kamera zu dem entsprechenden Objekt geschickt, um so zu testen, ob ein anderes Objekt dazwischen liegt. Das wird daran erkannt, dass der Strahl nicht auf das eigentliche Objekt, sondern auf ein anderes (dessen `Collider`) trifft, was bedeutet, dass das Zielobjekt durch dieses verdeckt wird (mit Ausnahme der halbtransparenten Kraftfelder, siehe Abschnitt 5.6.1). Nur wenn dies der Fall ist, wird der Indikator auf die `screenPos` des Objekts gelegt und visualisiert. Wird das Objekt nicht verdeckt, wird die Visualisierung deaktiviert.

Offscreen-Indikator Der Offscreen-Indikator wird angezeigt, wenn sich das entsprechende Objekt nicht im Kamerabild befindet, damit der Spieler dieses leichter wiederfindet und die Navigation im 3D-Raum unterstützt wird. Dadurch ist es zudem möglich, den Spieler auf Sachen hinzuweisen, die außerhalb seines Sichtfeldes liegen, damit dieser sich nicht dauernd suchend umdrehen muss. Die Offscreen-Indikatoren werden durch Pfeile visualisiert, die am Bildschirmrand auftauchen und in die Richtung des entsprechenden Objekts außerhalb des Bildschirms weisen. Die Position dieser Indikatoren auf dem Bildschirm wird mit Hilfe der `screenPos` bestimmt, nur dass diese im Gegensatz zu den Onscreen-Indikatoren jetzt negative Koordinaten besitzt oder Koordinaten, die größer sind als die Bildschirmgröße, da sich das Objekt außerhalb des Kamerabildes befindet. Daher werden die Werte der `screenPos` auf einen Bereich geclippt, der um eine zuvor definierte Randbreite kleiner als die wirkliche Bildschirmgröße ist, damit die Pfeile auf dem Bildschirm, aber nicht unmittelbar am Rand dargestellt werden, sodass sie besser erkennbar und nicht so leicht zu übersehen sind. Der Winkel, mit dem die Indikatoren-Sprites gedreht werden müssen, damit sie in die entsprechende Richtung der Objekte zeigen, wird mit Hilfe der Atan2-Funktion (Arkustangens) desjenigen Vektors bestimmt, welcher vom Mittelpunkt des Bildschirms zur aktuellen `screenPos` zeigt.

5.5.4 pARcours-Modus

Nachdem die wichtigsten Komponenten vorgestellt wurden, kann nun auf die Funktion des `GameplayController`-Skripts und die Spiellogik hinter dem *pARcours*-Modus eingegangen werden.
Über den `GameplayController` werden alle während der Objektplatzierung erstellten Objekte im Level registriert und können über ihn auch wieder aus dem Level entfernt werden. Er ist dazu da, das Level vorzubereiten, zu initialisieren und zu starten, zu beenden und zurückzusetzen oder komplett zu löschen. Außerdem instantiiert er den `Player`, verwaltet den Levelscore, berechnet die Punkte für das Beenden eines Levels und zählt die eingesammelten Juwelen.

Initialisierung und Start des Levels Bei der Auswahl des Spielmodus werden die verschiedenen Einstellungen für den vom Spieler ausgewählten Modus und Schwierigkeitsgrad vorgenommen. Dazu gehört die Zählrichtung des Timers (vorwärts, rückwärts oder deaktiviert), die `Token`-Liste für den `TokenSpawner` und die maximale Juwelenanzahl im Level. Während des Levelaufbaus wird simultan die Levelschwierigkeit durch den `GameplayController` berechnet und auf dem Bildschirm angezeigt. Je höher diese ist, desto mehr Punkte erhält der Spieler für das Absolvieren des Levels. Der Schwierigkeitswert des Levels steigt durch:

- die Anzahl der Checkpoints und deren Schwierigkeitswert
- die Anzahl der Hindernisse und deren Schwierigkeitswert
- die Distanz der einzelnen Checkpoints zueinander
- den Flugwinkel von einem Checkpoint zum nächsten (befindet sich der nächste Checkpoint in entgegengesetzter Flugrichtung, steigt der Schwierigkeitswert)
- und das Platzieren von Hindernissen zwischen zwei aufeinanderfolgenden Checkpoints

Ist der Spieler mit dem Aufbau des Levels fertig und betätigt den „Ready"-Button, um das Spiel zu starten, wird zunächst überprüft, ob alle für den Modus erforderlichen Objekte (Startfeld und mindestens ein Checkpoint) in der Szene vorhanden sind. Ist dies nicht der Fall, erscheint ein Hinweis auf dem Bildschirm, welche Objekte fehlen, und dass diese vorher noch platziert werden müssen. Ist das Level vollständig, werden der Levelscore, der Timer und die Zähler für die verschiedenen einzusammelnden Juwelen auf null gesetzt. Danach werden alle `Manipulator`-Objekte aus der Szene entfernt (falls welche vorhanden waren), damit der Spieler den Parcours nicht während des Levels noch verändern kann.
Weiterhin werden die Checkpoints für das Level vorbereitet. Hierfür wird `passedCheckpoints` auf null gesetzt, ein Zähler für die Anzahl an Checkpoints, die der Spieler in einem Level schon durchflogen hat. Außerdem wird der erste Checkpoint aktiviert und als Ziel des Checkpoint-Indikators registriert. Danach werden den Checkpoints die entsprechenden Materialien zugewiesen, die mittels verschiedener Farben den Status von diesen anzeigen (mehr dazu in Abschnitt 5.6.1). Zuletzt wird der `Player` instantiiert. Dafür wird ein `Anchor` an dem Mittelpunkt der äußeren `Collider`-Grenzen des Startfeldes und an diesem das `Player`-Prefab (Abschnitt 5.6.3) erzeugt, welches die verschiedenen Flugobjekte und Steuerungs-Skripts (Abschnitt 5.7) enthält. Erst wenn all diese Sachen erfolgreich erledigt worden sind, kann das ausgewählte Flugobjekt aktiviert und das Level letztendlich gestartet werden.

Absolvieren des Levels Der `GameplayController` ist auch für das Aktivieren und Deaktivieren der Checkpoints und deren Reihenfolge zuständig. Diese ist abhängig von der Reihenfolge, in der die Checkpoints im Level platziert wurden. Dafür wird jeder Checkpoint bei seiner Erstellung in eine Liste eingefügt, sodass der `GameplayController` während des Levels nur noch diese Liste von Anfang bis Ende abarbeiten muss, um genau zu wissen, welcher Checkpoint als nächstes dran ist und wann das Level beendet wurde.

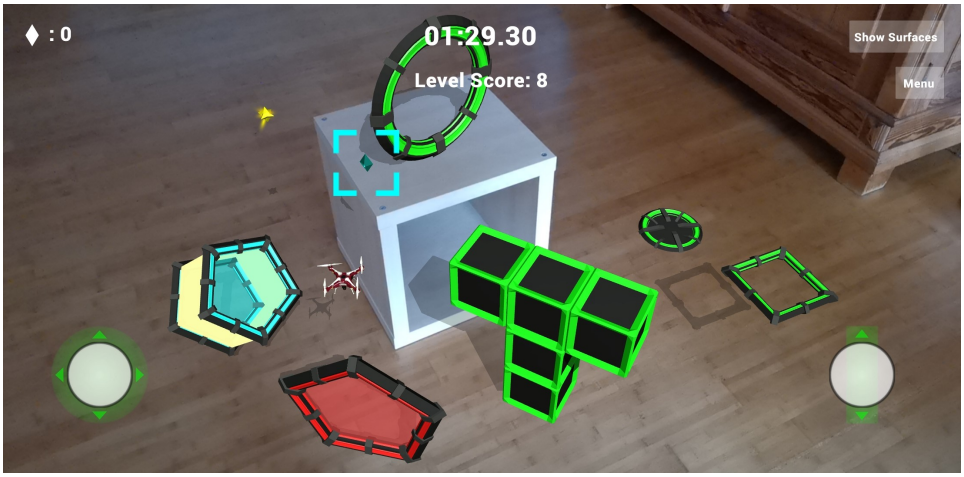


Abbildung 32: Durch die Farbe der Checkpoints wird ihr aktueller Status symbolisiert. Die grünen Checkpoints wurden schon durchflogen, der blaue ist aktuell aktiviert und muss als nächstes durchquert werden, der rote ist noch deaktiviert.

Wurde das Level gestartet, gilt es, alle Checkpoints der Reihe nach in möglichst kurzer Zeit zu durchfliegen. Der jeweils nächste Checkpoint wird hierbei immer durch seine blaue Färbung und den Checkpoint-Indikator angezeigt. Wurde ein Checkpoint erfolgreich passiert, wird der Checkpoint-Zähler inkrementiert, der nächste Checkpoint aktiviert und als neues Ziel des Indikators registriert, und die Checkpoints ändern ihr Material entsprechend ihrem Status, der durch die verschiedenen Farben symbolisiert wird (Abbildung 32). Außerdem wird der Levelscore anhand des Schwierigkeitswertes des gerade durchquerten Checkpoints erhöht. Mit Hilfe von `passedCheckpoints` kann dann in der Liste immer auf den nächsten Checkpoint entsprechend der Reihenfolge zugegriffen werden. Erreicht der Wert von `passedCheckpoints` die Anzahl aller im Level enthaltenen Checkpoints (also die Länge der Liste), weiß der `GameplayController`, dass der letzte Checkpoint durchquert wurde und ruft die `Victory()`-Methode auf, die das Level beendet.
Kollidiert der Spieler während des Levels mit einer Oberfläche oder einem Objekt, werden ihm dafür wieder Punkte abgezogen. Beides wird durch einen animierten Text neben dem eigentlichen Levelscore angezeigt. Sammelt der Spieler während des Levels Juwelen ein, wird der Zähler für diejenige Juwelenart und die Gesamtanzahl eingesammelter Juwelen erhöht.

Beenden des Levels Sobald das Level beendet wurde, werden sowohl der `Timer` als auch der `TokenSpawner` angehalten und der Endbildschirm angezeigt. Auf diesem werden die aktuelle Zeit, die (alte oder neue) Bestzeit, der Levelscore und die eingesammelten Juwelen angezeigt. Für den

visuellen Effekt werden diese Werte aber nicht sofort dargestellt, sondern nacheinander eingeblendet und (außer der Zeit) in einer Subroutine bis zu ihrem eigentlichen Wert hochgezählt. Schließlich werden die Punkte berechnet, die der Spieler für das Beenden des Levels erhält, und auf seinen Juwelen-Kontostand addiert. Der Spieler erhält (abhängig von dem Schwierigkeitswert des Levels, des gewählten Flugobjekts und dem generellen Schwierigkeitsgrad) Punkte für das generelle Beenden des Levels, das Aufstellen einer neuen Bestzeit und die Gesamtanzahl eingesammelter Juwelen.

Zurücksetzen des Levels Das Zurücksetzen des Levels kann durch den Spieler selbst eingeleitet werden, wenn dieser zum Beispiel unzufrieden mit seiner Leistung ist und nochmal von vorne beginnen möchte, oder er das Flugobjekt zu weit weg gesteuert und aus den Augen verloren hat. Will er das Level andererseits nach erfolgreichem Beenden neu starten, wird dieses automatisch zurückgesetzt. Dabei werden der Timer, der Levelscore und die Juwelen-Zähler wieder auf null gesetzt und auch die Checkpoints werden wieder in ihren ursprünglichen Zustand zu Beginn des Levels versetzt, damit der Spieler das Level von Neuem beginnen kann. Zuletzt wird noch das `Player`-GameObject zurückgesetzt, indem sein `Anchor` gelöscht und im Mittelpunkt des Startfeld-Kolliders wieder neu erstellt wird. Das soll verhindern, dass nach längerem Spielen und neu hinzugefügten Umgebungsinformationen doch noch ein kleiner Drift auftritt, sodass die Position des Anchors nicht mehr zu der des Startfeldes identisch wäre und das Flugobjekt somit nicht vom Startfeld aus starten würde. Dadurch kann der `Player` zusammen mit seinen Flugobjekten wieder sauber instantiiert und das Level neu gestartet werden.

Löschen des Levels Das Level wird automatisch komplett gelöscht, sobald der Spieler den Spielmodus wechselt oder zurück ins Hauptmenü geht. Dabei werden sowohl der `Player` als auch alle GameObjects, welche Kinder des `Level`-GameObjects sind, aus der Szene entfernt. Das beinhaltet zum einen das Löschen der Modelle und Anchors aller Objekte im Level und zum anderen das Leeren der Listen des `GameplayController`-Skripts, in denen diese Objekte für die Logik des Levels registriert waren.

5.5.5 ARcade-Modus

Der *ARcade*-Modus wird ebenfalls wie der *pARcours*-Modus über den `GameplayController` gesteuert, allerdings mit einigen Unterschieden. Da es bei diesem Spielmodus darum geht, auf Zeit so viele Juwelen wie möglich einzusammeln und daher keine Checkpoints oder Hindernisse benötigt werden, entfällt sowohl die Berechnung der Levelschwierigkeit als

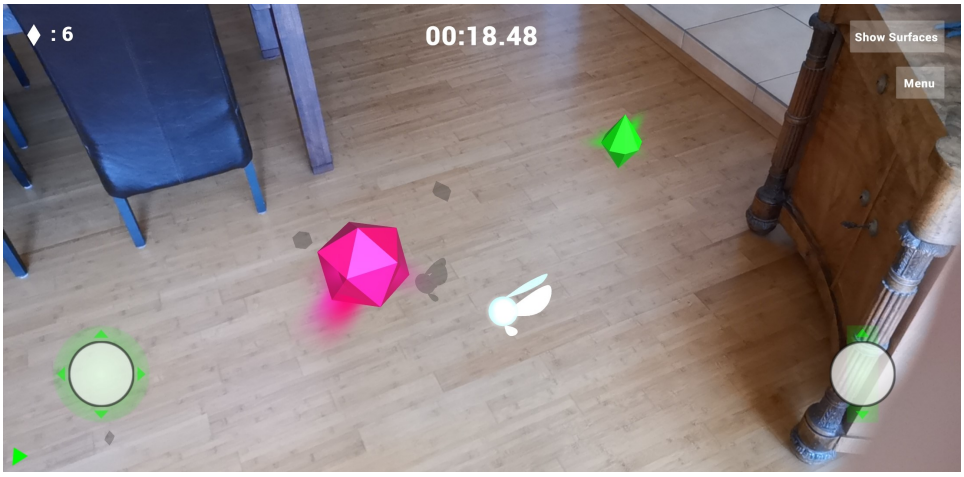


Abbildung 33: Im *ARcade*-Modus erscheinen um den Spieler herum Juwelen, die eingesammelt werden müssen, während die Zeit abläuft.

auch die Verwaltung der Checkpoints. Auch das Verhalten des `Timer`- und des `TokenSpawner`-Skripts ändert sich bei diesem Modus. Der Timer startet bei zwei Minuten und zählt die Zeit rückwärts, bis die Null erreicht wurde, womit das Level automatisch endet. In dieser Zeit tauchen weit mehr Juwelen auf als im *pARcours*-Modus und bei jedem zweiten davon handelt es sich unabhängig von der `Token`-Liste um ein *TimeBonus*-Juwel, da so viele wie möglich von diesen eingesammelt werden müssen, um die Zeit am Ablaufen zu hindern. Wäre dies nicht der Fall, könnte es aufgrund dessen, dass die Juwelen nach ihrer Wahrscheinlichkeit zufällig aus der entsprechenden Liste ausgewählt werden, dazu kommen, dass während des Levels kein einziges *TimeBonus*-Juwel vorkommt, was für diesen Modus fatal wäre. Da aber garantiert jedes zweite Juwel ein *TimeBonus*-Juwel ist, gibt das dem Spieler die Chance, das Ablaufen des Timers so lange wie möglich hinauszuzögern und dabei so viele Juwelen wie möglich einzusammeln. In Abbildung 33 ist der *ARcade*-Modus zu sehen, in dem zwei Juwelen in der Nähe des Spielers aufgetaucht sind, die mit Hilfe des Flugobjekts eingesammelt werden können.

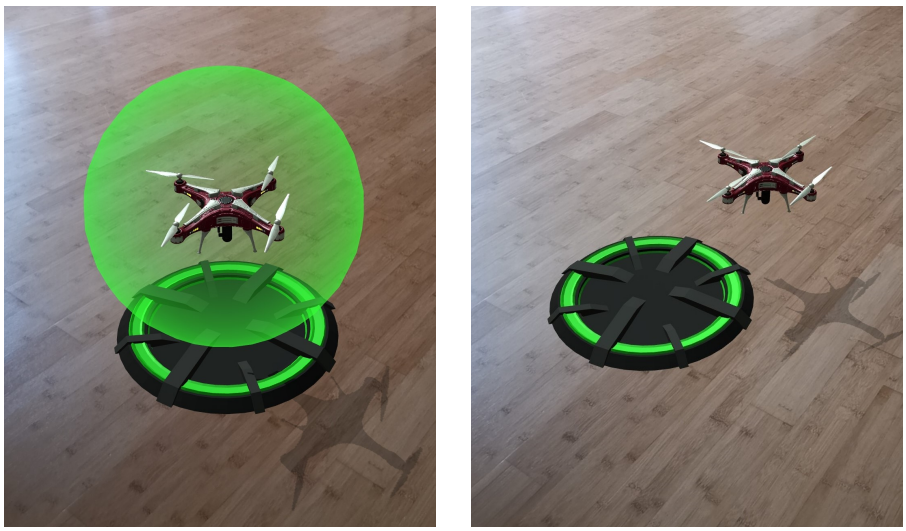
5.6 Virtuelle Spielkomponenten

Die virtuellen Spielkomponenten, welche sich in die Parcours-Objekte, die Juwelen und den Player mit seinen Flugobjekten einteilen lassen, sind sowohl für die Spiellogik als auch für das generelle Erscheinungsbild des Spiels essentiell. Eine Vielzahl an Objekt-Varianten soll dabei für ein abwechslungsreiches Spielerlebnis sorgen, wobei der Schwierigkeitsgrad durch die Wahl der Objekte individuell angepasst werden kann. Außerdem werden alle im Spiel enthaltenen Objekte mittels des zu Spielbeginn

bzw. in den Optionen gewählten `globalScale`-Werts global skaliert, damit sich die Objekte und somit das gesamte Spiel in die reale Umgebung einfügen können.
Der Großteil der 3D-Modelle für die virtuellen Objekte wurde in Blender erstellt, in Unity importiert und dort final angepasst. So wurden zum Beispiel zu allen Objekten noch entsprechende `Collider`-Komponenten für die Kollisionserkennung hinzugefügt, welche die Form der Modelle so gut wie möglich approximieren. Da es sich hierbei um ein AR Spiel handelt und es somit möglich ist, mit der Kamera sehr nah an die Objekte heran zu gehen und sie von allen Seiten zu betrachten, sollten diese möglichst detailliert sein, aber ihre Modelle gleichzeitig auch möglichst einfach (geringe Mesh-Größe), damit das Gerät nicht durch die gleichzeitige Darstellung vieler Objekte überfordert ist. Daher habe ich mich für relativ einfache Formen (Ringe, Würfel, etc.) entschieden, aus denen die Objekte zusammengebaut wurden.

5.6.1 Parcours-Objekte

Zu den Parcours-Objekten zählen alle Objekte, die während des Levelaufbaus durch den Spieler in der Szene platziert und somit unter dem `Level`-GameObject erstellt werden. Sie bilden den Parcours, der absolviert werden muss, und sind Unterklassen der `Objectives`-Oberklasse.
Jedes Parcours-Objekt besteht aus einem Wurzelknoten, welcher das Skript für den entsprechenden `Objectives`-Typ beinhaltet und dessen Transformation bei der Objektplatzierung manipuliert wird. Außerdem wird auf diesen die globale Skalierung angewandt, damit sich das Objekt von der Größe her der Umgebung anpasst. Durch die Manipulation auf dem Wurzelknoten werden alle Kindknoten automatisch mit transformiert. Unter dem Wurzelknoten befindet sich das eigentliche Modell des Objekts, dessen Kinder alle Einzelteile darstellen, aus denen es besteht. Jedem dieser Teile sind die der Funktionalität entsprechenden Komponenten zugeordnet. Dies beinhaltet die `Collider`-Komponenten, Skripts für die Kollisionserkennung und/oder Trigger (dt. Auslöser) für das Auslösen spezieller Funktionen oder Effekte.
Zu den Parcours-Objekten zählen das Startfeld, die Checkpoints und die Hindernisse (insgesamt wurden bis zum Zeitpunkt der Abgabe dieser Arbeit 47 unterschiedliche Parcours-Objekt für das Spiel kreiert), wobei jedes einzelne Objekt in der oben beschriebenen Form im Voraus erstellt, als Prefab abgespeichert und in die Objekt-Listen von `ImageMarkerTracker` und `ObjectivesManipulator` eingefügt wurde. Somit können sie während des Spiels als fertige Bausteine dienen und müssen vom Spieler nur noch ausgewählt und in der Umgebung platziert werden, wobei jedes Mal ein Klon des eigentlichen Prefabs erstellt wird. Im Folgenden werde ich näher auf die einzelnen Objekt-Typen eingehen.

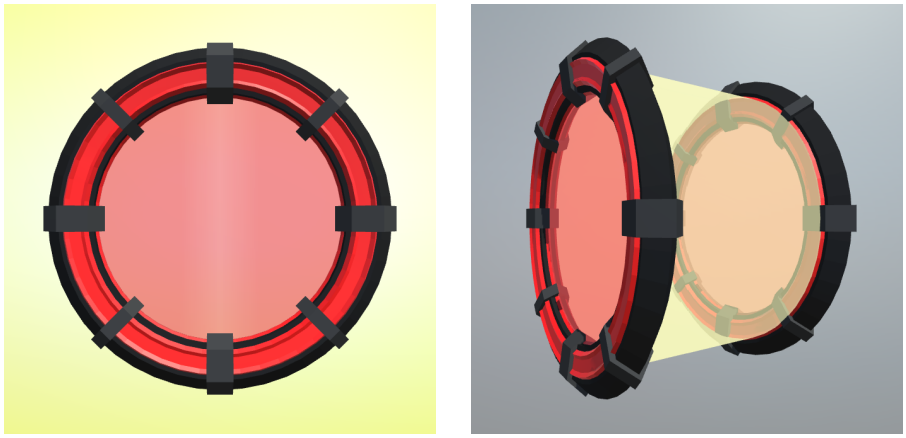


(a) Startfeld mit aktiviertem Kraftfeld (b) Startfeld mit deaktiviertem Kraftfeld

Abbildung 34: Das Flugobjekt wird zu Beginn des Levels im Zentrum des Startfeldes erstellt (a). Verlässt es das Kraftfeld, wird dieses deaktiviert (b) und das Level gestartet.

Startfeld Ein Level wird gestartet, sobald der Spieler das Startfeld mit seinem Flugobjekt verlässt. Um dies zu erkennen, besitzt das `StartField`-Prefab ein Kraftfeld (engl. *force field*) in Form einer Kugel, in dessen Mitte das Flugobjekt beim Start des Levels erstellt wird (Abbildung 34 (a)). Dieses `ForceField`-GameObject besitzt eine `SphereCollider`-Komponente, welche als Trigger gekennzeichnet ist, und ein `ForceFieldController`-Skript. In dem `ForceFieldController` wird mit Hilfe der Unity eigenen `OnTriggerExit()`-Methode erkannt, wann der Spieler das Startfeld verlässt, woraufhin die `OnPassThroughForceField()`-Methode des `StartField`-Skripts aufgerufen wird. Hier wird zuerst sicherheitshalber überprüft, ob es sich bei dem entsprechenden Objekt um ein Flugobjekt gehandelt hat. Ist dies der Fall, so wird das `ForceField` deaktiviert (Abbildung 34 (b)), ein Sound als Feedback abgespielt und das Level entsprechend dem Spielmodus und Schwierigkeitsgrad gestartet.

Die Kraftfelder spielen eine zentrale Rolle bei der Interaktion der Flugobjekte mit den virtuellen Parcours-Objekten (nicht nur beim Startfeld, sondern auch bei den Checkpoints). Jedes `ForceField`-GameObject, welches Teil eines Parcours-Objektes ist, besitzt einen `ForceFieldController`, der je nach Funktion dafür zuständig ist, entweder eine Kollision zu erkennen oder eine Aktion beim Durchfliegen zu triggern. Außerdem besitzen sie ein der Funktion entsprechendes Material mit einem speziellen



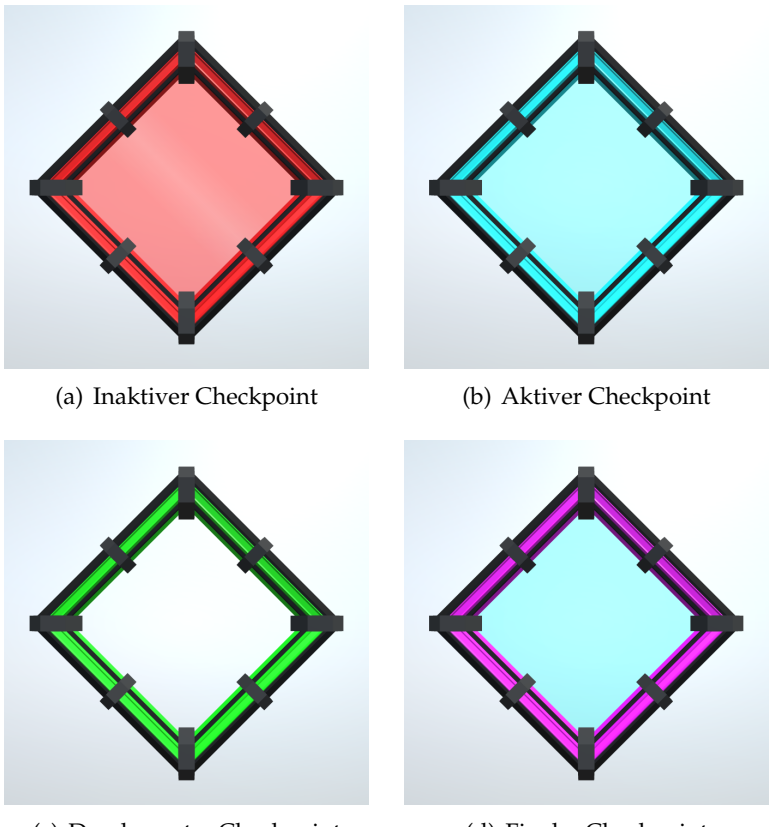
(a) Einfacher Checkpoint (b) Doppelter Checkpoint

Abbildung 35: Zwei Beispiele für Checkpoints. Der einfache Checkpoint (a) besteht aus einem einzelnen Ring, wohingegen der doppelte Checkpoint (b) aus zwei Ringen besteht, die einen Tunnel bilden.

`ForceFieldShader`, mit dem die Erscheinung der Kraftfelder wahlweise durch Verwendung von Transparenz, einer Textur, einem Farbgradienten, Emission und einem Rim-Effekt (der Rand wird weniger transparent dargestellt als die Mitte) angepasst werden kann. Durch diese halbtransparenten Kraftfelder wird das Flugobjekt nicht vollständig verdeckt und somit ist es nicht nötig, an diesen Stellen einen Offscreen-Indikator zu visualisieren, weshalb diese beim Test auf Verdeckung (Abschnitt 5.8.1) ignoriert werden.

Checkpoints Jeder Checkpoint besitzt einen individuellen Schwierigkeitswert, welcher von seiner Größe und Form abhängt und der sowohl den Punktabzug bei Kollisionen als auch den Schwierigkeitsgrad des Levels und somit die für das erfolgreiche Beenden erhaltenen Punkte beeinflusst. Dabei gibt es grundsätzlich zwei Arten von Checkpoints: Die einfachen und die doppelten Checkpoints (Abbildung 35), wobei der Schwierigkeitswert von Letzteren generell höher ist.
Ein einfacher Checkpoint besteht aus einem äußeren Ring (dieser kann unterschiedlich geformt sein) und einem Kraftfeld in dessen Mitte. Der Ring besitzt mehrere Kollider, die dessen Form approximieren und für die Kollisionserkennung zuständig sind. Stößt der Spieler mit seinem Flugobjekt gegen den äußeren Ring, wenn er beispielsweise versucht, durch ihn hindurch zu fliegen, werden ihm Punkte von seinem Levelscore entsprechend dem Schwierigkeitswert des Checkpoints und dem allgemeinen Schwierigkeitsgrad abgezogen. Das erhöht die Schwierigkeit beim Absolvieren eines Parcours. Das Kraftfeld in der Mitte der Checkpoints ist ein Trigger

und dient zur Erkennung, ob ein Flugobjekt den Checkpoint passiert. Hierzu wird die `OnPassThroughForceField()`-Methode des `Checkpoint`-Skripts aufgerufen. Gehört das `TriggerForceField` zu einem aktiven Checkpoint (Abbildung 36 (b)) und wurde es erfolgreich durchflogen, so wird das GameObject des Kraftfeldes deaktiviert und ein Sound abgespielt. Daraufhin deaktiviert der `GameplayController` den aktuellen Checkpoint, erhöht den Levelscore anhand des individuellen Schwierigkeitswertes und aktiviert den nächsten Checkpoint. Die Richtung, in der die einzelnen Checkpoints durchquert werden, ist dabei egal.
Ist der Checkpoint allerdings zu diesem Zeitpunkt nicht aktiv (Abbildung 36 (a)), bildet das Kraftfeld eine feste Barriere, gegen die das Flugobjekt stößt, anstatt durch sie hindurch zu fliegen, und statt der Trigger- wird die Kollisions-Methode aufgerufen und der Levelscore verringert. Die Aktivierung und Deaktivierung der Checkpoints und die Zuweisung der entsprechenden Materialien wird über den `GameplayController` vorgenommen. In Abbildung 36 werden die Checkpoints mit ihren unterschiedlichen Materialien dargestellt, die deren Status anzeigen.



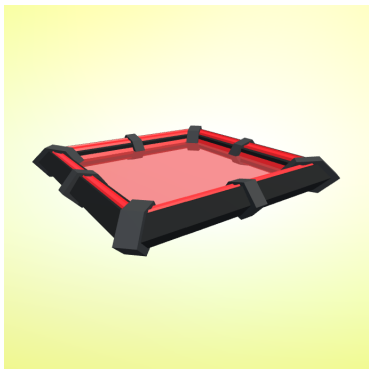
(a) Inaktiver Checkpoint (b) Aktiver Checkpoint

(c) Durchquerter Checkpoint (d) Finaler Checkpoint

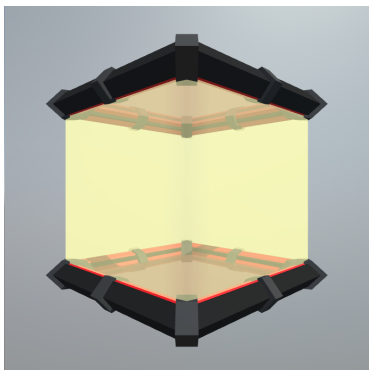
Abbildung 36: Die Farben der Checkpoints zeigen ihren Status an.

Ein doppelter Checkpoint besteht aus zwei einfachen Checkpoints, die zusammen einen Tunnel bilden, durch den der Spieler fliegen muss (Abbildung 37 (c) und (d)). Dabei besitzen die Ringe auf beiden Seiten jeweils ein eigenes Kraftfeld, sodass beide Kraftfelder durchquert werden müssen, damit der nächste Checkpoint vom `GameplayController` aktiviert wird. Der Tunnelrand bei doppelten Checkpoints ist zwar auch ein Kraftfeld, welches der Spieler allerdings nicht durchfliegen kann. Stößt er gegen den Tunnelrand, so werden ihm dafür wie bei den äußeren Ringen Punkte vom Levelscore abgezogen.
Für mehr Abwechslung wurden unterschiedliche Varianten von Checkpoints erstellt (Abbildung 37). Sowohl bei den einfachen als auch bei den doppelten Checkpoints gibt es als Formen den `Ring`, das `Pentagon` und das `Square` (dt. Quadrat), wobei die Größe vom Erstgenannten zum Letzten abnimmt und der Schwierigkeitswert zunimmt. Weiterhin gibt es alle Varianten in unterschiedlichen Rotationen (horizontal, vertikal und diagonal), damit sie in unterschiedlichen Richtungen durchflogen werden müssen. Insgesamt wurden 22 Varianten von Checkpoints erstellt.

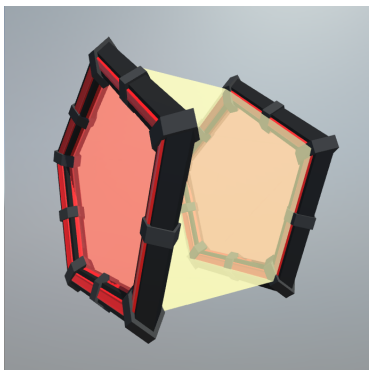
(a) Ring Diagonal



(b) Square Vertical

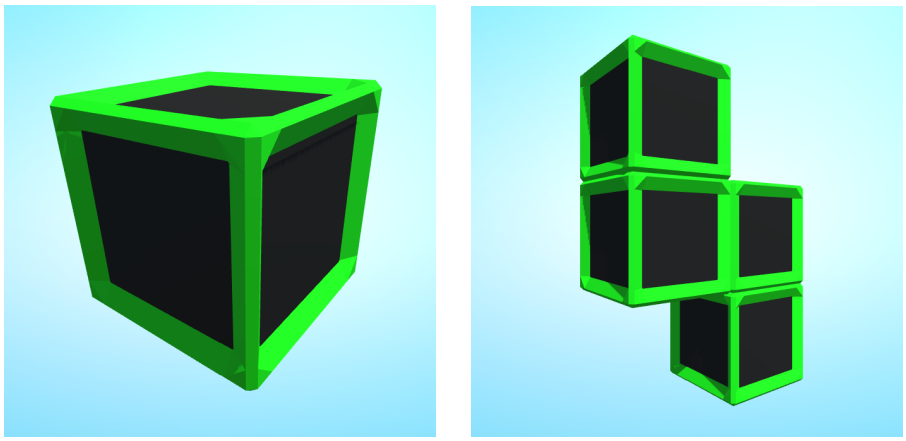


(c) Double Square Vertical



(d) Double Pentagon Horizontal

Abbildung 37: Beispiele für unterschiedliche Checkpoint-Varianten.



(a) Einfacher Würfel (b) Zusammengesetztes Hindernis

Abbildung 38: Die Hindernisse (b) werden aus Würfeln (a) zusammengesetzt, damit ohne großen Aufwand eine Vielzahl an Hindernis-Varianten erstellt werden kann.

Hindernisse Die Hindernisse (engl. *obstacles*) besitzen ebenfalls wie die Checkpoints individuelle Schwierigkeitswerte, welche von deren Größe und Form abhängen und den Punktabzug und die Levelschwierigkeit beeinflussen. Allerdings besitzen sie keine Kraftfelder und ihre Funktion beschränkt sich allein auf die Kollisionserkennung. Hierzu wird die Form der Hindernisse ebenfalls durch mehrere `Collider` angenähert. Kollidiert der Spieler mit einem Hindernis, wird die `OnCollisionWithObstacle()`-Methode im `Obstacle`-Skript aufgerufen, welches bei jedem Hindernis-Prefab vorhanden ist. Ist der Spieler zum Zeitpunkt der Kollision nicht unverwundbar, wird sein Levelscore wie bei den Checkpoints entsprechend dem Schwierigkeitswert des Hindernisses und dem allgemeinen Schwierigkeitsgrad verringert.

Alle Hindernisse sind aus einzelnen `Cube`-Modellen (dt. Würfel-Modellen) (Abbildung 38 (a)) zusammengebaut, welche zu unterschiedlichen Formen zusammengesetzt werden und somit eine Vielzahl verschiedener Hindernisse ergeben können (Abbildung 38 (b)). Das einzelne `Cube`-GameObject wurde hierfür, um sich doppelte Arbeit zu sparen, als Prefab abgespeichert und musste für die verschiedenen Hindernisse nur noch dupliziert und unterschiedlich angeordnet werden. Dadurch ist es auch möglich, dem Spiel auch in Zukunft ohne viel Arbeit weitere Hindernis-Objekte hinzuzufügen. Weiterhin gibt es besondere Hindernisse, in deren Modell zusätzlich ein Juwel integriert ist, welches vom Spieler eingesammelt werden kann (Abbildung 39). Aber Vorsicht: Diese befinden sich meist an schwer zugänglichen Stellen, sodass der Spieler abwägen muss, ob er das Juwel einsammeln und

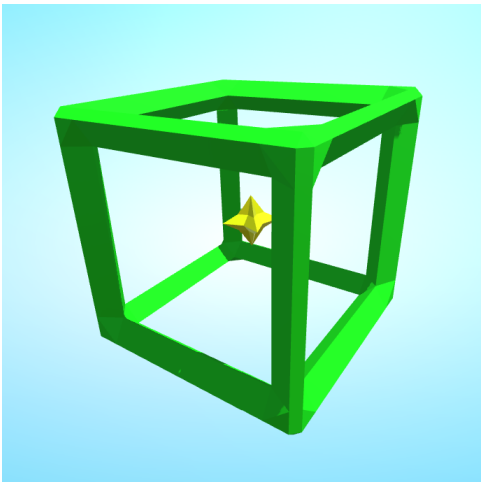


Abbildung 39: Spezielle Hindernisse enthalten Juwelen, die vom Spieler eingesammelt werden können.

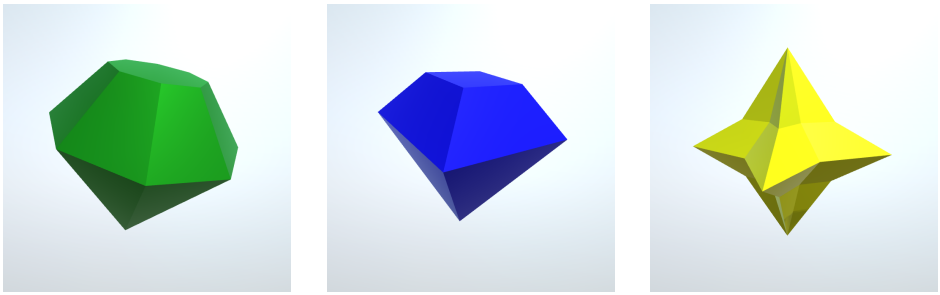
dabei eine Kollision riskieren möchte. Insgesamt wurden 24 Varianten von Hindernissen für ein abwechslungsreiches Spielerlebnis erstellt.

5.6.2 Juwelen

Die Juwelen wurden ebenfalls als Prefabs erstellt, damit sie während des Spiels ohne großen Aufwand vom `TokenSpawner` aus den entsprechenden Juwelen-Listen als Kopien erstellt werden können (Abschnitt 5.5.2). Ein Juwel besteht aus einem Wurzelknoten, an dem das für die Juwelenart entsprechende Skript angehängt ist. Diese Skripts erben alle von der `Token`-Klasse, welche die Oberklasse für Juwelen darstellt. Dieses ist dafür zuständig, die `lifetime` (Lebensspanne) zu verwalten, die jedes Juwel besitzt und abhängig vom generellen Schwierigkeitsgrad ist. Sobald ein Juwel im Level durch den `TokenSpawner` erstellt wurde, wird dessen anfängliche `lifetime` durchgehend reduziert, bis diese die Null erreicht, wodurch das Juwel automatisch zerstört wird. Unter dem Wurzelknoten befindet sich das eigentliche Modell des Juwels, dessen Form und Farbe die Zugehörigkeit zur entsprechenden Juwelenart anzeigen (Abbildung 40).

Das Erscheinen und Verschwinden eines Juwels ist jeweils durch eine Subroutine animiert. Damit die Juwelen nicht „plötzlich da und wieder weg“ sind, werden sie beim Erstellen von null hoch bzw. bei der Zerstörung von dem eigentlichen Größenwert wieder auf null runter skaliert, sodass sie langsam erscheinen und wieder verschwinden. Während sie sich im Level befinden, rotieren die Juwelen-GameObjects um die vertikale Achse, damit sie durch die Bewegung im Raum besser gesehen werden können.

(a) TimeBonus-Juwel (b) SpeedBonus-Juwel (c) DoubleBonus-Juwel



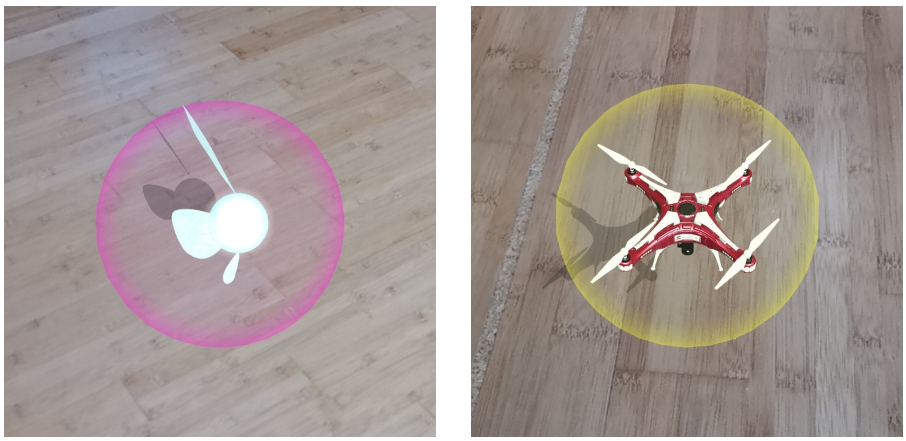
(d) TimePenalty-Juwel (e) SpeedPenalty-Juwel (f) Invulnerability-Juwel

Abbildung 40: Die verschiedenen Juwelenarten besitzen unterschiedliche Farben und Formen, damit der Spieler sie auseinanderhalten kann.

Weiterhin besitzt jedes Juwel einen Trigger, damit es möglich ist, sie mit dem Flugobjekt einzusammeln, wenn sie mit diesem in Berührung kommen. Dabei wird im Skript der Juwelenart die entsprechende Methode aufgerufen, um den jeweiligen Effekt (siehe Tabelle 1 in Abschnitt 4.3.2) auszulösen, im `GameplayController` die Anzahl der eingesammelten Juwelen zu erhöhen und danach das Juwel zu löschen. Damit der Spieler weiß, ob er ein positives oder negatives Juwel eingesammelt hat, wird für die beiden ein unterschiedlicher Sound abgespielt.

TimeBonus und TimePenalty Die beiden Juwelenarten *TimeBonus* und *TimePenalty* enthalten jeweils einen `timeBonus`- bzw. `timePenalty`-Wert, der abhängig vom gewählten Schwierigkeitsgrad ist (je leichter, desto größer). Dieser erhöht bzw. verringert beim Einsammeln des Juwels die aktuelle Zeit im `Timer`-Skript, was auf dem Bildschirm neben dem Timer angezeigt wird.

SpeedBonus und SpeedPenalty Diese beiden Juwelen enthalten einen `speedBonus`- bzw. `speedPenalty`-Wert, der ebenfalls abhängig vom gewählten Schwierigkeitsgrad ist (je leichter, desto kleiner). Dieser erhöht



(a) Doppelter Bonus-Kraftfeld (b) Unverwundbarkeits-Kraftfeld

Abbildung 41: Die Wirkungszeit der durch das *DoubleBonus*-Juwel (a) oder das *Invulnerability*-Juwel (b) ausgelösten Effekte werden mit Hilfe verschiedenfarbiger Kraftfelder um das Flugobjekt herum angezeigt.

bzw. verringert beim Einsammeln des Juwels den aktuellen `speedLevel` der Flugobjekte im `PlayerController`-Skript (Abschnitt 5.6.3) und wird ebenfalls kurz auf dem Bildschirm angezeigt.

DoubleBonus Die `doubleBonusTime` des *DoubleBonus*-Juwels gibt die Zeit in Sekunden an, die der Effekt des doppelten Bonus anhält. Dieser Wert ist ebenfalls abhängig vom gewählten Schwierigkeitsgrad (je leichter, desto größer). Durch Einsammeln wird die `GivePlayerDoubleBonus()`-Methode im `PlayerController` aufgerufen, die dafür sorgt, dass dem Spieler für die entsprechende Zeit für weitere eingesammelte Juwelen der doppelte Zeitbonus/-strafe, Geschwindigkeitsbonus/-strafe oder die doppelte Zeit bei Unverwundbarkeit oder anderen *DoubleBonus*-Juwelen gegeben wird. Solange der Effekt anhält, wird dies durch ein violettes Kraftfeld um das Flugobjekt herum visualisiert (Abbildung 41 (a)).

Invulnerability Die `invulnerabilityTime` des *Invulnerability*-Juwels gibt die Zeit in Sekunden an, für die der Spieler unverwundbar wird. Auch dieser Wert hängt vom gewählten Schwierigkeitsgrad ab (je leichter, desto größer). Durch Einsammeln wird die `MakePlayerInvulnerable()`-Methode im `PlayerController` aufgerufen, die dafür sorgt, dass dem Spieler für die entsprechende Zeit bei einer Kollision mit anderen Objekten keine Punkte vom Levelscore abgezogen werden. Solange der Effekt anhält, wird dies durch ein gelbes Kraftfeld um das Flugobjekt herum visualisiert (Abbildung 41 (b)).

5.6.3 Player und Flugobjekte

Die verschiedenen Flugobjekte und ihre Steuerungen werden mit Hilfe des `Player`-Prefabs im Level erstellt und von diesem verwaltet. Dieses Prefab wird beim Start des Levels als Kind eines Anchors in der Mitte des Startfeldes instantiiert und enthält alle zur Auswahl stehenden Flugobjekte als Kinder. Da die Flugobjekte einzeln und nicht der gesamte `Player` vom Spieler gesteuert werden, dient er als zentraler Ausgangspunkt, sodass die Flugobjekte beim Zurücksetzen zur Ausgangsposition zurückkehren können. Weiterhin enthält der `Player` das `AircraftSelection`-Skript und ist somit dafür verantwortlich, dass der Spieler sein Flugobjekt wechseln kann. Das ebenfalls angefügte `PlayerController`-Skript ist für die eigentliche Verwaltung der Flugobjekte zuständig. Dieser enthält einen `speedLevel`-Wert, der abhängig vom ausgewählten Schwierigkeitsgrad und der globalen Skalierung die allgemeine Geschwindigkeit der Flugobjekte bestimmt. So muss zum Beispiel der `speedLevel` reduziert werden, wenn eine kleine Skalierung gewählt wurde, damit die Geschwindigkeit der Größe der Objekte angepasst wird. Je höher der Spieler dagegen den Schwierigkeitsgrad wählt, desto schneller werden die Flugobjekte. Der `speedLevel` kann durch das Einsammeln von *SpeedBonus*-Juwelen positiv und durch *SpeedPenalty*-Juwelen negativ beeinflusst werden. Weiterhin werden in diesem Skript die jeweiligen Eingabe-Skripte für die ausgewählte Steuerung aktiviert und die Effekte von *DoubleBonus*- und *Invulnerability*-Juwelen auf den Spieler übertragen (visualisiert durch die entsprechenden Kraftfelder).

Es stehen vier verschiedene Arten von Flugobjekten zur Auswahl, die unter dem `Player` instantiiert werden: eine Fee, eine Drohne, ein Raumschiff und ein Papierflieger. Diese besitzen jeweils ein eigenes Skript, welches für die individuelle Steuerung der Bewegung zuständig ist. Dadurch ist es möglich, für jedes Flugobjekt ein zu dem jeweiligen Objekt passendes Steuerungsverhalten zu implementieren, sodass diese unterschiedlich auf die entsprechenden Eingaben des Spielers reagieren können. Zum einen wollte ich damit erreichen, dass der Spieler die Möglichkeit hat, zwischen verschiedenen Flugobjekten zu wechseln, die nicht nur anders aussehen, sondern sich auch anders steuern lassen, sodass durch die Abwechslung der Spielspaß für längere Zeit erhalten bleibt. Zum anderen sollte es den Realismus erhöhen, dass sich das Steuerungsverhalten dem Flugobjekt anpasst. So wäre zum Beispiel ein Papierflieger, der rückwärts fliegen kann, unrealistisch, wohingegen das bei einer Drohne durchaus möglich ist.
Weiterhin ist den Flugobjekten jeweils ein Schwierigkeitswert zugeordnet, der sich auf das individuelle Steuerungsverhalten und die Geschwindigkeit der jeweiligen Flugobjekte bezieht (jedes Flugobjekt hat einen individuellen `movementSpeed`-Wert, der dessen generelle Geschwindigkeit be-

schreibt). So besitzt die Fee den kleinsten Schwierigkeitswert, weil sie am einfachsten zu steuern ist, und der Papierflieger den größten, da sich dieser am schwersten steuern lässt. Der Schwierigkeitswert des gewählten Flugobjekts beeinflusst die Punkte, die der Spieler für das erfolgreiche Beenden eines Levels erhält (je größer der Wert, desto mehr Punkte).

Jedes GameObject eines Flugobjekts besteht aus einem Wurzelknoten, welcher das entsprechende Skript für die Bewegungssteuerung und einen `Rigidbody` enthält. Diese Komponente ist dafür gedacht, die Bewegung von Objekten mit Hilfe von Unitys *Physics Engine* steuern zu können. Dazu gehören beispielsweise das Anwenden einer physikalischen Kraft auf Objekte oder das Erkennen von Kollisionen. Ohne einen `Rigidbody` könnte der Spieler nicht mit anderen Objekten in der Szene zusammenstoßen, weshalb diese Komponente für jedes Flugobjekt zwingend erforderlich ist. Zusammen mit dem individuellen Skript für die Bewegungssteuerung kann das Flugobjekt somit gesteuert werden. Hierfür enthält jedes dieser Skripte eine `GetDirection()`-Methode, mit welcher der Richtungsvektor (`InputVector`) des Skripts der entsprechend ausgewählten Eingabemethode abgefragt wird (mehr dazu in Abschnitt 5.7). Dieser beschreibt die Bewegungsrichtung des Flugobjekts im dreidimensionalen Raum für die Eingabe des Spielers und ist für alle Flugobjekte identisch. Anhand dieses Vektors kann die Bewegung der Flugobjekte in deren Skripts auf unterschiedliche Arten realisiert werden.
Unter den Wurzelknoten befinden sich die jeweiligen Modelle der Flugobjekte. Deren Form wird wie bei den anderen virtuellen Objekten durch entsprechende `Collider` approximiert, damit die Kollisionserkennung möglichst genau erfolgen kann. Um einen Zusammenstoß zu registrieren, besitzt das Flugobjekt ein `PlayerCollision`-Skript, mit welchem die Kollisionen erkannt und ein entsprechender Kollisions-Sound abgespielt werden kann. Dieser Sound sowie der, welcher bei Bewegung der Flugobjekte abgespielt wird, sind für die verschiedenen Flugobjekte unterschiedlich, sodass die Individualität jedes Flugobjekts hervorgehoben wird. Im Folgenden werde ich das Steuerungsverhalten der einzelnen Flugobjekte genauer beschreiben.

Navi Navi ist eine Fee, die ursprünglich aus dem Spiel *The Legend of Zelda: Ocarina of Time* vom japanischen Spieleentwickler und Spielkonsolenhersteller Nintendo stammt [15]. Das verwendete Modell[8] stammt aus der 3D-Modell-Datenbank sketchfab und besteht aus einem kugelförmigen, halbtransparenten Körper mit `ShereCollider`, welcher von innen mittels einer `Light`-Komponente beleuchtet wird, und zwei Flügel-Objekten mit

[8] Verfügbar unter `https://sketchfab.com/3d-models/fairy-the-legend-of-zelda-botw-ce8425ea1cfb40e492906bb62d970969`, Abgerufen am 28.07.2019

Abbildung 42: Das Flugobjekt Navi in Bewegung.

entsprechender Textur. Das ganze Modell wird durch eine kontinuierliche Auf- und Abwärtsbewegung animiert, während die Flügel in sanftem Rhythmus schlagen (Abbildung 42). Durch die Bewegungen soll der Eindruck entstehen, Navi würde tatsächlich und aus eigener Kraft durch den Raum fliegen.

Für die Bewegungssteuerung von Navi ist das `NaviMovement`-Skript zuständig. Der mittels `GetDirection()` erhaltene Richtungsvektor `direction` wird dazu verwendet, Navi immer in die Richtung zu rotieren, in die sie fliegen soll. Dazu wird ein neuer Vektor `moveDirection` erstellt, welcher die Bewegungsrichtung auf die Blickrichtung der Kamera überträgt (die Kamera blickt hierbei entlang der positiven z-Achse). Damit ist garantiert, dass sich das Flugobjekt immer relativ zur Kamera bewegt, auch wenn der Spieler seine Position im Raum ändert. Dabei wird allerdings die y-Komponente des Vektors auf null gesetzt, da das Modell der Fee auch bei Höhenänderung immer in horizontaler Ausrichtung verbleiben soll. Ist die Länge des Bewegungsvektors `moveDirection` groß genug (was bedeutet, dass durch den Spieler eine horizontale Eingabe erfolgte), wird Navi in die entsprechende Bewegungsrichtung rotiert.
Danach wird die y-Komponente von `direction` zur Bewegungsrichtung `moveDirection` hinzugenommen, um Navi durch den Raum zu bewegen, da diese auch aufwärts bzw. abwärts fliegen können soll. Damit Navi in die gewünschte Richtung $\vec{d}$ fliegen kann, wird der `moveDircetion`-Vektor $\vec{m}$ noch mit einer vorgegebenen Geschwindigkeit s multipliziert, die für jedes Flugobjekt individuell gesetzt werden kann und welche zusätzlich von der globalen Skalierung, dem gewählten Schwierigkeitsgrad und dem `speedLevel` des `PlayerController`-Skripts beeinflusst wird. Weiterhin wird die seit dem letzten Frame vergangene Zeit t hinzugerechnet, damit die Bewegungsgeschwindigkeit auch bei unterschiedlichen Frameraten identisch bleibt.

$$\vec{d} = \vec{m} \cdot s \cdot t$$

Abbildung 43: Das Flugobjekt Drohne.

Ist durch den Spieler keine Eingabe erfolgt, wird die Geschwindigkeit von Navi langsam auf null reduziert, damit die Bewegungen sanft ausklingen und nicht abrupt enden.
Navi kann sich somit in alle Raumrichtungen bewegen und folgt präzise den Eingaben des Spielers. Die Bewegungsrichtung wird immer relativ zur Kamera berechnet, wodurch es dem Spieler leichter fällt, die exakte Flugrichtung vorzugeben. Stoppt der Spieler die Eingabe, so bleibt Navi an der entsprechenden Stelle im Raum stehen, bis weitere Eingaben erfolgen.

Drohne Bei der Drohne (Abbildung 43) handelt es sich um einen Quadrocopter, ein Flugobjekt mit vier nach unten gerichteten Rotoren, dessen Modell[9] aus dem Unity Asset Store stammt. Über ein Skript, welches zusätzlich jedem der vier Rotoren angefügt wurde, werden diese dauerhaft rotiert, damit es realistischer erscheint, wenn die Drohne durch die Luft fliegt. Die Drohne lässt sich ebenfalls in alle Raumrichtungen bewegen, ist jedoch schwieriger zu steuern, da sie durch auf den `Rigidbody` einwirkende Kräfte bewegt wird. Daher enden ihre Bewegungen nicht so schnell wie die der Fee, sobald der Spieler die Eingabe stoppt, sondern sie wird durch den Schwung der Bewegung wie bei einer echten Drohne noch ein Stück weiter bewegt. Dadurch soll ein möglichst realistisches Flugverhalten simuliert werden.
Die Bewegungen werden durch das `DroneMovement`-Skript gesteuert. Ähnlich wie bei der Fee werden die horizontalen Bewegungen mit Hilfe des `direction`-Vektors bestimmt, jedoch wirkt die Kraft, die auf die

[9] Verfügbar unter `https://assetstore.unity.com/packages/tools/physics/free-pack-117641`, Abgerufen am 28.07.2019

Drohne mittels `AddRelativeForce()` ausgewirkt wird, um diese zu bewegen, relativ zu ihrem eigenen Koordinatensystem. Die seitlich auf die Drohne wirkende Kraft $right$ und die nach vorne wirkende Kraft $forward$ werden hierbei ebenfalls durch die gewünschte Geschwindigkeit s und die Zeitdifferenz t zum letzten Frame beeinflusst.

$$right = direction.x \cdot s \cdot t$$
$$forward = direction.z \cdot s \cdot t$$

Da die Drohne zusätzlich von der Schwerkraft beeinflusst wird, muss ihre Gewichtskraft g bei der vertikal wirkenden Kraft up mit eingerechnet werden.

$$up = g + direction.y \cdot s \cdot t$$

Ist keine Eingabe vorhanden, dann schwebt die Drohne auf der Stelle, indem die nach oben gerichtete Kraft der Gewichtskraft entspricht. Die so berechneten Kräfte werden dann mit Hilfe von `AddRelativeForce()` auf die Drohne übertragen, sodass sie sich durch den Raum bewegen kann. Damit die Drohne nicht zu schnell werden kann, wenn immer weiter Kraft aus einer Richtung auf sie ausgeübt wird, wird ihre Geschwindigkeit auf eine vorgegebene `movementSpeed` beschränkt.
Wenn durch den Spieler keine Eingabe mehr erfolgt, wird die Geschwindigkeit der Drohne langsam bis auf null reduziert. Dadurch entsteht der Eindruck, sie würde sich durch ihren Schwung noch ein Stück weiter bewegen, bevor sie durch den Luftwiderstand zum stehen kommt.

Zuletzt muss die Drohne noch um die vertikale Achse rotiert werden, damit sie immer relativ zur Kamera gesteuert werden kann. Der Rotationswert entspricht dabei dem y-Wert des Rotationswinkels der Kamera. Dadurch wird die Drohne immer in dieselbe Richtung gedreht wie die Kamera, sodass beispielsweise die Vorwärts-Eingabe des Spielers immer (in Blickrichtung) weg von der Kamera führt.
Zusätzlich wird ein `tiltAmountForward` bzw. `tiltAmountSideways`-Wert berechnet, welche die Neigungswinkel der Drohne bei der Vor- bzw. Rückwärtsbewegung und bei seitlicher Bewegung darstellen. Mit Hilfe dieser Werte wird die Drohne in die jeweilige Flugrichtung geneigt, damit das Flugverhalten noch realistischer dargestellt werden kann.

Raumschiff Das Raumschiff-Modell[10] stammt ebenfalls aus dem Unity Asset Store, wobei dieses dahingehend modifiziert wurde, dass drei `TrailRenderer`-Komponenten hinzugefügt wurden, jeweils eine an der

[10]Verfügbar unter `https://assetstore.unity.com/packages/3d/vehicles/space/star-sparrow-modular-spaceship-73167`, Abgerufen am 28.07.2019

Abbildung 44: Das Flugobjekt Raumschiff.

Position eines Triebwerks des Modells. Diese ziehen bei Bewegung eine „Spur“ (engl. *trail*) hinter sich her, sodass mit ihnen der Ausstoß der drei Triebwerke des Raumschiffs dargestellt werden kann, wenn sich dieses bewegt (Abbildung 44). Dadurch wirkt die Bewegung realistischer und lebendiger und der Spieler erhält ein visuelles Feedback auf seine Eingaben.

Das Raumschiff lässt sich zwar auch relativ zur Kamera steuern, aber im Gegensatz zu Navi oder der Drohne kann es (in Bezug zu sich selbst) nur vorwärts fliegen. Daher muss es zuerst in diejenige Richtung rotiert werden, in die es sich bewegen soll, bevor es sich in diese Richtung entsprechend bewegen kann. Dies geschieht im `SpaceshipMovement`-Skript. Die Bewegungsrichtung `moveDirection` wird wie bei der Fee mit Hilfe des Eingabevektors `direction` bestimmt, nur dass der Richtungsvektor in diesem Fall keine absolute Richtung angibt, sondern eine relative Richtung zur aktuellen Rotation. Da das Raumschiff nicht in der Lage sein soll, Loopings zu fliegen, wird der y-Wert der `moveDirection` auf den Bereich [-0.8, 0.8] beschränkt. Mit `RotateTowards()` wird das Raumschiff dann von seiner aktuellen Rotation zur neuen Bewegungsrichtung rotiert.
Nachdem das Raumschiff in die gewünschte Richtung $\vec{r}$ rotiert wurde, wird es entsprechend seiner Rotierung, der Länge des Bewegungsvektors $\vec{m}$ und der Geschwindigkeit s unter Berücksichtigung von t vorwärts bewegt.

$$\vec{d} = \vec{r} \cdot |\vec{m}| \cdot s \cdot t$$

Stoppt der Spieler die Eingabe, so wird auch hier die Geschwindigkeit langsam auf null reduziert, damit das Raumschiff nicht plötzlich anhält.

Abbildung 45: Das Flugobjekt Papierflieger.

Papierflieger Das Modell des Papierfliegers wurde selber in Blender erstellt und besteht lediglich aus einem sehr flachen, gefalteten Würfel (Abbildung 45), so wie ein realistischer Papierflieger auch nur aus einem Blatt Papier gefaltet wird. Dieser wird mittels des `PaperPlaneMovement`-Skripts ähnlich wie das Raumschiff gesteuert und kann somit auch nur vorwärts fliegen. Allerdings wird der Papierflieger nicht relativ zur Kamera, sondern relativ zu seinem eigenen Koordinatensystem gelenkt und ist in seiner Rotation um die entsprechenden Achsen nicht beschränkt, wodurch es möglich ist, mit ihm Rollen oder Loopings zu fliegen. Dadurch wird es aber auch schwieriger, ihn zu kontrollieren und in die gewünschte Richtung zu steuern. Die Rotationswinkel yaw, $pitch$ und $roll$ (dt. Gieren, Nicken und Rollen) um die entsprechenden Achsen, welche sich aus dem Eingabevektor `direction` und einer vorgegebenen Rotationsgeschwindigkeit r ergeben, werden hierbei immer zu der aktuellen Rotation des Papierfliegers dazu gerechnet, sodass sich dieser auch um die eigenen Achsen drehen kann, wenn die Eingabe dauerhaft in dieselbe Richtung erfolgt und die Winkel somit größer als 360 Grad werden.

$$\begin{aligned} yaw &= direction.x \cdot r \cdot t \\ pitch &= direction.y \cdot r \cdot t \\ roll &= direction.x \cdot r \cdot t \end{aligned}$$

Sobald der Papierflieger in die gewünschte Richtung rotiert wurde, wird er wie das Raumschiff entsprechend seiner Rotierung vorwärts bewegt. Allerdings wird hierbei die Bewegung langsamer gestoppt, wenn keine Eingabe mehr erfolgt, wodurch sich der Papierflieger noch realistischer bewegt und der Schwierigkeitsgrad gesteigert wird.

5.7 Steuerung der Flugobjekte

Zusätzlich zu den unterschiedlichen Steuerungsverhalten der Flugobjekte kann der Spieler auch zwischen verschiedenen Arten der Eingabe für die Steuerung wählen. Dadurch soll gewährleistet werden, dass für jeden Spieler eine Eingabeart dabei ist, mit welcher er zurechtkommt, da jeder Spieler seine eigenen Vorlieben hat. Außerdem bringt dies weitere Abwechslung in das Spiel. Um zu testen, welche der Steuerungsmethoden für ein AR-Spiel dieser Art bei den Spielern gut ankommt, ist dies auch ein zentraler Punkt in dem von mir durchgeführten Nutzungstest (Kapitel 6).

Eine Herausforderung bei der Implementierung war es, die Eingabe, die meist über den 2D-Touchscreen des Geräts erfolgt, auf den dreidimensionalen Raum zu übertragen, um die Flugobjekte mit 6DOF steuern zu können. Außerdem war es wichtig, dass die Eingaben möglichst einfach zu handhaben, präzise und schnell auszuführen sind und das System direkt auf diese reagiert, da sonst eine Steuerung in Echtzeit nicht möglich ist.
Weiterhin sollte eine Eingabemethode jeweils auf alle Flugobjekte angewandt werden können, auch wenn diese sich damit unterschiedlich bewegen lassen. Daher wird für jede Eingabeart in dem entsprechenden Skript ein `InputVector` für die Eingabe des Spielers erstellt, der auf den Bereich [-1, 1] normalisiert wird und den gewünschten Richtungsvektor im dreidimensionalen Raum darstellt. Mit diesem Vektor, der für alle Flugobjekte identisch ist, kann somit jedes von diesen in seinem eigenen Skript individuell gesteuert werden, wobei es egal ist, aus welcher Eingabemethode der `InputVector` stammt. Das sollte verhindern, dass jede einzelne Eingabe für jedes einzelne Flugobjekt separat erstellt werden muss. Der Richtungsvektor wird für jedes Flugobjekt mittels `GetDirection()` aus dem entsprechenden Eingabe-Skript geholt.

Zu den implementierten Eingabemethoden gehören die virtuellen Joysticks, der Touch-Input, die Steuerung mittels der Gerätekamera und das physische Gamepad. Eine Kippsteuerung wurde ebenfalls testweise implementiert, jedoch ergab sich dabei das Problem, dass das Gerät gekippt werden musste, um das Flugobjekt zu bewegen, der Spieler deshalb jedoch nicht mehr sehen konnte, wohin er eigentlich fliegt. Wurde dann, um dieses Problem zu umgehen, die Sensitivität der Kippsteuerung soweit gesteigert, dass nur eine minimale Bewegung ausreichte, um das Flugobjekt zu bewegen, ließ sich dieses aber nicht mehr präzise steuern, da schon die kleinste Neigung dazu führte, dass es sehr weit weg gesteuert wurde. Dieses Problem könnte zwar durch eine Anpassung der Fluggeschwindigkeit behoben werden, jedoch ist für ein Spiel dieser Art nicht optimal, dass der Spieler das Gerät absolut gerade halten muss, um das Flugobjekt auf Position zu halten, wodurch die Bewegungsfreiheit des Spielers eingeschränkt

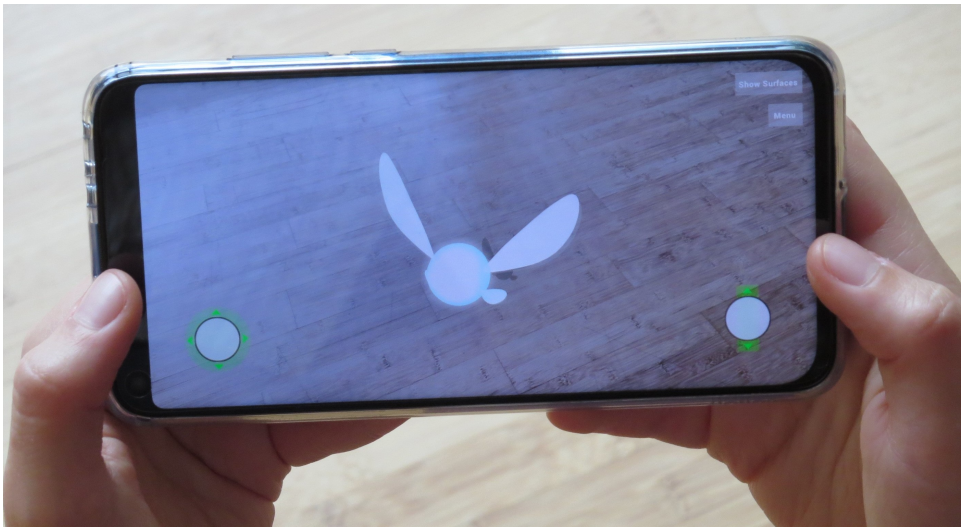


Abbildung 46: Die zwei virtuellen Joysticks werden in den unteren Bildschirmecken dargestellt und können dadurch mit beiden Daumen gleichzeitig bewegt werden.

wird. Da der Spieler aber auch um den Parcours herum gehen und die AR-Szene von allen Blickwinkeln aus betrachten können soll (was wiederum die Steuerung der Flugobjekte und die 3D-Wahrnehmung vereinfacht), habe ich diese Eingabemethode erstmal verworfen, auch wenn sie viel Potential besitzt und den Spielspaß steigern könnte.

5.7.1 Virtuelle Joysticks

Bei dieser Eingabemethode erscheinen zwei virtuelle Joysticks auf dem Bildschirm, die jeweils aus einem größeren, auf dem Bildschirm fest platzierten Hintergrund und einem kleineren „Joystick“ bestehen, der innerhalb des Bereichs des Hintergrunds bewegt werden kann. Diese sind so angeordnet, dass der Spieler sie mit beiden Daumen gleichzeitig betätigen kann, während er das Gerät in der Hand hält (Abbildung 46). Damit sie auf dem Bildschirm nicht als störend empfunden werden, sind ihre Sprites relativ klein und halbtransparent.
Will der Spieler sein Flugobjekt in eine bestimmte Richtung lenken, muss er die Joysticks wie bei einem echten Controller in die entsprechende Richtung ziehen. Dabei ist der linke Joystick für die horizontale Bewegung zuständig und kann mittels des Skripts `JoystickFourDirections` in alle Richtungen bewegt werden, wohingegen der rechte Joystick, der für die vertikale Bewegung zuständig ist, durch `JoystickTwoDirections` nur hoch oder runter bewegt werden kann. Hierfür mit mit Hilfe der Methoden `OnPointerDown()`, `OnPointerUp()` und `OnDrag()` vom System

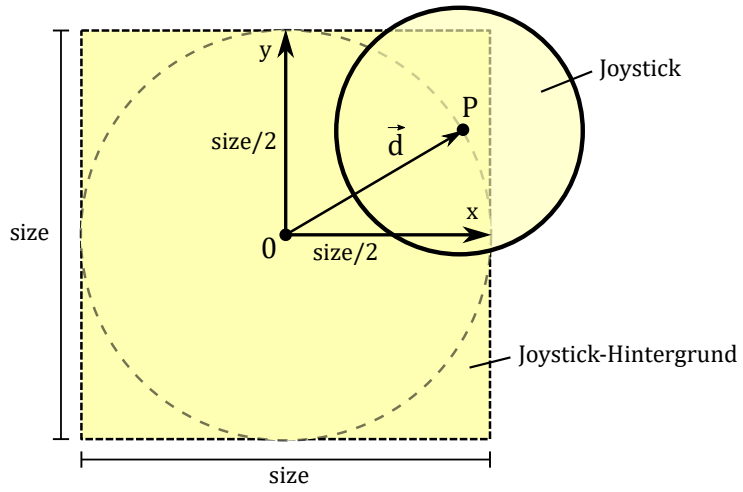


Abbildung 47: Funktionsweise des virtuellen Joysticks für vier Richtungen.

erkannt, wann der Spieler welchen Joystick benutzt und wieder loslässt. Während der Spieler einen Joystick betätigt, wird der lokale Punkt P für die Berührung auf dem Joystick-Hintergrund bestimmt (berührt der Spieler den Bildschirm außerhalb dieses Bereichs, geschieht nichts). Da der Mittelpunkt des Hintergrunds den Ursprung von dessen lokalem Koordinatensystem darstellt und P in Pixelkoordinaten vorliegt, liegen die Werte dieses Punktes im Bereich [-$size$/2, +$size$/2], wobei $size$ die Höhe bzw. Breite des Hintergrunds in Pixeln darstellt (weil dieser rechteckig ist, ist $size$ für Höhe und Breite identisch). Da der gesuchte `InputVector` derjenige Vektor $\vec{d}$ sein soll, der vom Mittelpunkt des Joystick-Hintergrunds zu der entsprechend berührten Stelle P zeigt (Abbildung 47) und im Bereich [-1, 1] liegen soll, müssen die Werte zuerst umgerechnet werden.

$$\vec{d} = (\vec{p} \cdot \frac{1}{size}) \cdot 2$$

Zusätzlich wird $\vec{d}$ normalisiert, damit der Joystick nicht in einem Quadrat, sondern in einem Kreis um den Mittelpunkt herum bewegt werden kann.

Wird der Joystick wieder losgelassen, kehrt er in seine Ausgangsposition zurück und die Länge des `InputVector` ist null. Der einzige Unterschied zum `JoystickFourDirections` ist, dass `JoystickTwoDirections` den x-Wert von $\vec{d}$ immer auf null setzt, damit sich der Joystick nur hoch und runter bewegen lässt. Durch das Zusammenspiel beider Joysticks können die Flugobjekte in alle drei Raumrichtungen gesteuert werden, indem die horizontale und vertikale Eingabe des Spielers im `direction`-Vektor zusammengefasst werden:

```
direction.x = JoystickLeftFourDirections.InputVector.x;
direction.y = JoystickRightTwoDirections.InputVector.y;
direction.z = JoystickLeftFourDirections.InputVector.y;
```

5.7.2 Touch-Input

Beim Touch-Input fliegt das Flugobjekt auf horizontaler Ebene in Richtung desjenigen Punktes, den der Spieler auf dem Touchscreen berührt (Abbildung 48). Hierzu werden die x- und y-Koordinaten des Bildschirms auf die x- und z-Koordinaten des 3D-Raums übertragen.

Abbildung 48: Das Flugobjekt fliegt auf horizontaler Ebene zu dem Punkt, den der Spieler auf dem Touchscreen berührt.

Dies geschieht im `TouchInputMovementControl`-Skript, welches dem `Player`-Prefab angefügt wurde. Hier wird zuerst überprüft, ob der Spieler den Bildschirm berührt. Ist dies der Fall und ist das Ziel der Berührung kein UI-Element, dann werden die Bildschirmkoordinaten F des Flugobjekts bestimmt. Der `InputVector` ist dann derjenige Vektor $\vec{d}$, der auf dem Bildschirm von der Position F des Flugobjekts zu der Position P zeigt, an welcher der Bildschirm berührt wurde (Abbildung 49).

$$\vec{d} = \vec{p} - \vec{f}$$

Dieser wird noch normalisiert, wenn seine Länge größer als eins war. Wird keine Berührung auf dem Bildschirm registriert, werden alle Werte vom `InputVector` auf null gesetzt. Die Höhe des Flugobjekts kann bei dieser

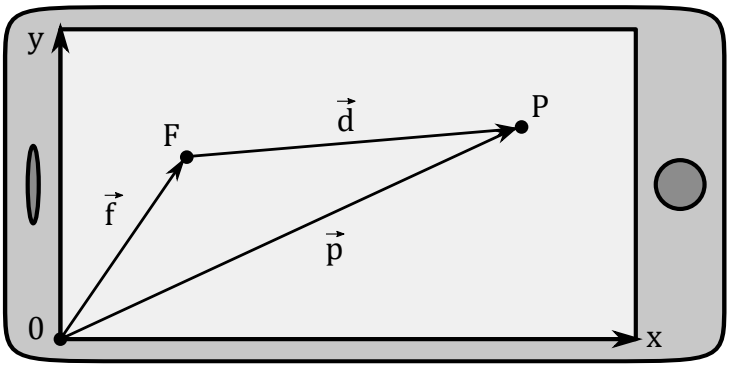


Abbildung 49: Funktionsweise des Touch-Inputs.

Eingabemethode durch einen separaten Joystick geregelt werden, der genauso funktioniert wie bei der Eingabe mit den virtuellen Joysticks, sodass auch hier in alle drei Raumrichtungen gesteuert werden kann:

```
direction.x = TouchInputMovementControl.InputVector.x;
direction.y = JoystickLeftTwoDirections.InputVector.y;
direction.z = TouchInputMovementControl.InputVector.y;
```

5.7.3 Steuerung mittels Kamera

Bei dieser Eingabemethode wird ein Fadenkreuz in der Mitte des Bildschirms dargestellt, welches den Zielpunkt für die Bewegung des Flugobjekts im dreidimensionalen Raum markiert (Abbildung 50). Weiterhin erscheint in der rechten unteren Ecke des Bildschirms ein „Drive“-Button, welcher das Flugobjekt in Bewegung setzt.
Betätigt der Spieler diesen „Drive“-Button, fliegt das Flugobjekt in Richtung des Fadenkreuzes (d. h. zur Mitte des Bildschirms), sodass er die Bewegung dadurch lenken kann, dass er mit der Kamera immer in die entsprechende Richtung zeigt, in die das Flugobjekt fliegen soll. Dabei bleibt die Distanz der Zielposition bzw. des Fadenkreuzes zur Kamera immer gleich. Möchte der Spieler die Zieldistanz ändern, so geschieht das mittels eines Joysticks, wie er für die vertikale Bewegung der beiden vorherigen Eingabemethoden verwendet wurde. Allerdings steuert dieser das Flugobjekt nicht direkt, sondern nur die Distanz des Fadenkreuzes zur Kamera und somit die Distanz der Zielposition, welche für eine bessere Übersicht am unteren Bildschirmrand angezeigt wird (in Metern).

Die Berechnung der Zielposition für das Flugobjekt übernimmt das Skript `ControlWithCameraInput`, welches am `Player`-Prefab angehängt ist.

Abbildung 50: Die Zielposition des Flugobjekts wird im dreidimensionalen Raum durch ein Fadenkreuz angezeigt. Durch Betätigen des „Drive“-Buttons in der rechten unteren Bildschirmecke bewegt sich das Flugobjekt in die entsprechende Richtung.

Wird der Joystick betätigt, ändert dies entsprechend die Zieldistanz zur Kamera mit einer konstanten `distanceChangeSpeed`, welche aber noch mit der globalen Skalierung verrechnet werden muss, damit sich das Fadenkreuz beispielsweise bei einer kleineren Skalierung langsamer bewegt. Außerdem wird der minimale Wert der Zieldistanz auf 0,2 Meter und der maximale Wert auf 20 Meter gesetzt, damit dieser nicht zu klein oder zu groß werden kann, wodurch das Fadenkreuz bzw. das Flugobjekt in oder hinter die Kamera fliegen würde (bei einer Distanz ≤ 0) oder so weit weg wäre, dass es zu klein zum Darstellen ist.

Nach Berechnung der Zieldistanz $distance$ wird die Zielposition P_w im dreidimensionalen Raum berechnet, unter der Voraussetzung, dass der „Drive“-Button gedrückt wird. Diese ergibt sich aus der aktuellen Kameraposition C_w in Weltkoordinaten (die während des Trackings von ARCore bestimmt wird und hier zur vereinfachten Darstellung dem Mittelpunkt des Kamerabildes entspricht, siehe Abbildung 51) und der Zieldistanz $distance$ in Blickrichtung $\vec{z_c}$ der Kamera.

$$\vec{p_w} = \vec{c_w} + \vec{z_c} \cdot distance$$

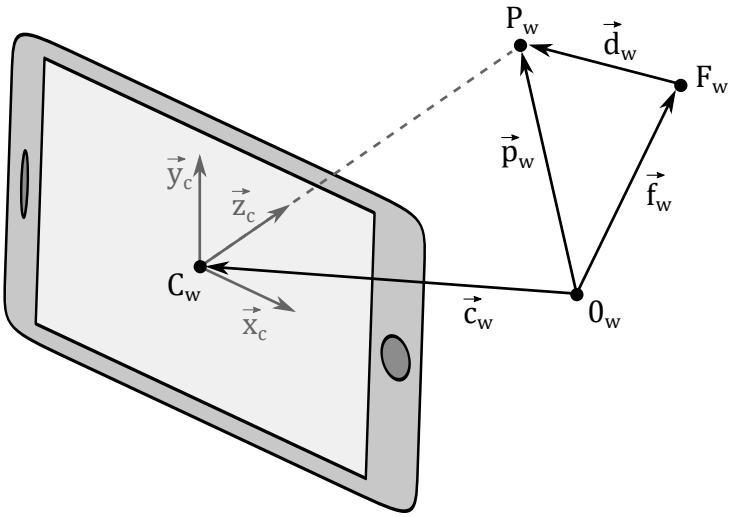


Abbildung 51: Funktionsweise der Steuerung mittels Kamera.

Der Vektor $\vec{d_w}$, welcher von der Position des Flugobjekts F_w zu der so berechneten Zielposition P_w zeigt, ist der gesuchte Richtungsvektor.

$$\vec{d_w} = \vec{p_w} - \vec{f_w}$$

Auch dieser wird noch, wenn erforderlich, normalisiert. Gab es keine Eingabe durch den Spieler, werden die Werte vom `InputVector` auf null gesetzt. Im Gegensatz zu den zwei zuvor beschriebenen Eingabemethoden handelt es sich hierbei schon um einen 3D-Richtungsvektor in Weltkoordinaten, sodass dessen Werte direkt auf den `direction`-Vektor übertragen werden können:

```
direction.x = ControlWithCameraInput.InputVector.x;
direction.y = ControlWithCameraInput.InputVector.y;
direction.z = ControlWithCameraInput.InputVector.z;
```

Allerdings muss beachtet werden, dass dieser Vektor aufgrund der Art der Steuerung in Weltkoordinaten vorliegen muss, sodass der entsprechende Schritt zur Umrechnung in Kamerakoordinaten in den Steuerungs-Skripts der Flugobjekte entfallen muss, sollte die Eingabe mittels Kamera gewählt worden sein.

5.7.4 Gamepad

Bei dieser Eingabe hält der Spieler einen physischen Controller in der Hand, an dem das Gerät befestigt ist (Abbildung 52). Hierzu wurde der *Wireless Game Controller* der Firma STOGA verwendet, welcher sich über Bluetooth

Abbildung 52: Das Smartphone wurde am Controller befestigt, wodurch der Spieler beide Hände für das Festhalten des Gamepads und die Steuerung des Flugobjekts nutzen kann.

mit einem Smartphone verbinden lässt. Dieses wird in der vorgesehenen Halterung eingeklemmt und kann somit auch bei Bewegung stabil am Gamepad befestigt werden, damit der Spieler beide Hände für das Festhalten des Controllers und die Benutzung der beiden Joysticks frei hat.
Die Bewegung der Flugobjekte kann hierbei wie mit den virtuellen Joysticks gesteuert werden. Der linke Joystick steuert die horizontale, der rechte Joystick die vertikale Bewegung. Um die Eingaben des Controllers in Unity nutzen zu können, mussten die entsprechenden Input-Achsen allerdings erst in den Input-Settings des Unity-Projekts definiert werden (in Abbildung 53 beispielhaft dargestellt für die horizontale Achse des linken Joysticks). Danach konnten die Eingaben, die über diese Achsen erfolgen, als Richtungsvektor verwendet werden:

```
direction.x = GetAxis("GamepadLeftJoystickHorizontal");
direction.y = GetAxis("GamepadRightJoystickVertical");
direction.z = GetAxis("GamepadLeftJoystickVertical");
```

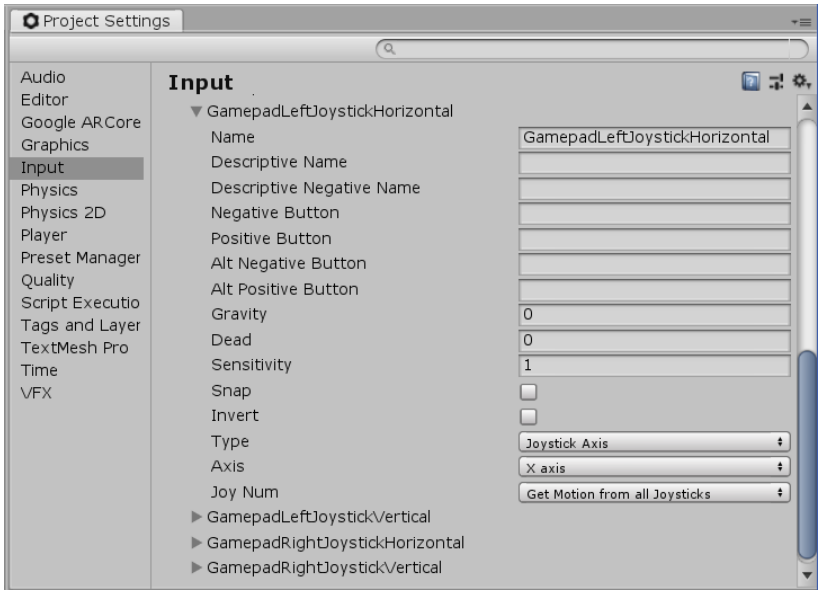


Abbildung 53: Die Zuordnung der horizontalen Achse des linken Gamepad-Joysticks zur virtuellen x-Achse in den Input-Settings in Unity.

5.8 Interaktion mit realen Oberflächen

Durch die Interaktion von virtuellen Objekten mit realen Oberflächen kann die Immersion gesteigert werden. Zudem hilft sie dabei, die Position von Objekten im dreidimensionalen Raum besser einschätzen zu können, und dient somit als visuelle Unterstützung für die 3D-Wahrnehmung, welche in meinem Spiel von großer Bedeutung ist. Hierzu wurden die Verdeckung virtueller Objekte durch reale Oberflächen, der Schattenwurf auf reale Oberflächen und die Kollision mit realen Oberflächen implementiert.
Um die Interaktion zu ermöglichen, benötigen die realen Oberflächen eine möglichst genaue virtuelle Abbildung in der Szene, deren Erstellung in Abschnitt 5.3.3 und 5.3.2 beschrieben wurde. Diese `ARSurface`-Objekte dienen als *Phantomobjekte* für die realen Oberflächen. Das sind virtuelle Objekte, welche die Größe und Form der entsprechenden realen Objekte möglichst genau abbilden und an derselben Stelle in der Szene registriert werden, damit sie diese möglichst korrekt überlagern. Phantomobjekte werden in der Regel nicht gerendert, sondern dienen lediglich dazu, die korrekte Verdeckung zwischen virtuellen und realen Objekten zu ermöglichen [36].

Verdeckung und Schatten werden mittels zweier Shader realisiert, die für die visuelle Darstellung der `ARSurface` zuständig sind und je nach gewünschter Visualisierung der Oberflächen ausgetauscht werden. Dabei unterscheidet sich der `PlaneGridShader` von dem `ARSurfaceShader` nur in der Hinsicht, dass bei ersterem zusätzlich zu Verdeckung und Schatten

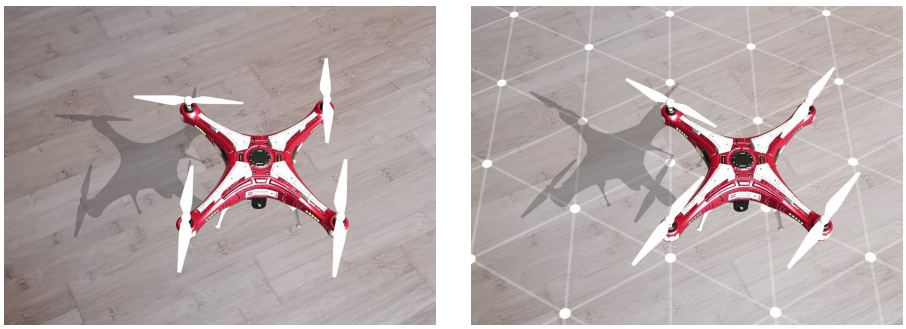
die Oberfläche in einem vorangehenden Shaderpass zusätzlich noch mittels einer Grid-Textur visualisiert wird, damit der Spieler sehen kann, welche Oberflächen schon vom System erkannt wurden.
Beim `ARSurfaceShader` hingegen sind die Oberflächen unsichtbar, sodass ein möglichst realitätsnaher Eindruck entsteht. Da Verdeckung und Schatten in beiden Shadern auf gleiche Weise realisiert wurden, gelten die folgenden Erklärungen für beide Fälle gleichermaßen.

5.8.1 Verdeckung

Ohne die Verdeckung virtueller Objekte durch die vom System erkannten Oberflächen wäre es schwierig einzuschätzen, ob sich ein Objekt vor oder hinter einer solchen befindet. So würde ein Objekt, welches sich hinter einer Oberfläche befindet, dennoch so aussehen, als würde es vor dieser schweben (Abbildung 54 (b)). Um die Verdeckung zu realisieren, wird im Shader `ZWrite on` verwendet. Dadurch wird die `ARSurface` in einem separaten Renderdurchlauf vor allen anderen Objekten gerendert, wobei lediglich der Tiefenbuffer beschrieben wird. In dem eigentlichen Renderdurchlauf können dann die z-Werte der darzustellenden Objekte durch einen Tiefentest mit den im Tiefenbuffer eingetragenen Tiefenwerten der Oberflächen verglichen werden. Ist ein Tiefenwert eines Objekts größer als der Tiefenwert einer Oberfläche, so bedeutet das, dass dieser Teil des Objekts hinter einer Oberfläche liegen muss und daher nicht gerendert wird. Dadurch ist das Kamerabild an den entsprechenden Stellen sichtbar, sodass es aussieht, als würde das Objekt an diesen Stellen durch einen realen Gegenstand verdeckt werden (Abbildung 54 (a)).

(a) Verdeckung (b) Keine Verdeckung

Abbildung 54: Vergleich des Effekts, welcher durch die Verdeckung erreicht wird. In (a) wird das Flugobjekt durch die Oberfläche verdeckt, sodass dessen Position korrekt eingeschätzt werden kann. In (b) findet keine Verdeckung statt, wodurch der Eindruck entsteht, das Flugobjekt würde über der Oberfläche schweben, wobei es sich auch hier eigentlich hinter dieser befindet.



(a) Schatten auf realer Oberfläche (b) Schatten auf Grid-Visualisierung

Abbildung 55: Durch den Schatten, der auf realen Oberflächen dargestellt wird, lässt sich die Position von virtuellen Objekten im Raum besser einschätzen.

5.8.2 Schatten

Die Darstellung von Schatten auf Oberflächen kann den 3D-Eindruck verstärken und dabei helfen, die Position von virtuellen Objekten im Raum besser einschätzen zu können. Besitzt ein Objekt keinen Schatten, kann das den Betrachter irritieren, auch wenn ihm vielleicht zunächst gar nicht auffallen mag, dass das am fehlenden Schatten liegt.
Im Fragmentshader der Oberflächen wird daher zuerst mit Hilfe von `UNITY_LIGHT_ATTENUATION` ermittelt, welche Fragmente im Schatten liegen, und anhand dessen wird der Alpha-Wert für das Fragment bestimmt. Der Farbwert des Schattens wird aus den zuvor definierten `_ShadowColor` (schwarz) und `_ShadowIntensity` berechnet, welche ihrerseits mit dem Alpha-Wert verrechnet werden. Die finale Fragmentfarbe ergibt sich dann mittels Alpha-Blending aus der Farbe des bereits vorliegenden Hintergrunds (also des Kamerabildes) und der zuvor berechneten Schattenfarbe. Als Ergebnis wird die `ARSurface` an den Stellen, auf die ein Schatten fällt, mit einem halbtransparenten Schwarz überblendet (Abbildung 55). Dies ist zwar eine relativ simple Methode, Schatten zu berechnen, jedoch reicht sie völlig aus, damit dem Spieler ihr positiver Effekt auffällt.

5.8.3 Kollisionserkennung

Eine Kollisionserkennung von virtuellen Objekten mit den realen Oberflächen ist nötig, da diese sonst nicht miteinander interagieren könnten. Ohne sie wäre es zum Beispiel möglich, mit dem Flugobjekt durch die Oberflächen hindurch zu fliegen, was die AR-Illusion stark beeinträchtigen würde. Um eine Kollision zu ermöglichen, besitzen die `ARSurface`-Objekte genauso wie die Flugobjekte einen oder mehrere der Form entsprechende `Collider`-Komponenten (im Fall der `ARSurface` handelt es sich um

einen `MeshCollider`, damit die Form der Oberfläche genau abgedeckt wird). Mit Hilfe der `OnCollisionEnter()`-Methode, welche von Unity automatisch aufgerufen wird, sobald eine Kollision zweier `Collider` erkannt wurde, können die Auswirkungen umgesetzt werden. Da das Flugobjekt das einzige Objekt ist, welches sich während des Levels bewegt, sollte auch nur dieses mit einer Oberfläche zusammenstoßen können. Daher wird in `OnCollisionEnter()` im `ARSurface`-Skript abgefragt, ob es sich bei dem kollidierenden GameObject um ein Flugobjekt handelt und ob der Spieler momentan verwundbar ist. Ist beides der Fall, wird als negativer Effekt der Levelscore des Spielers für den Zusammenstoß reduziert (im *pARcours*-Modus). Ist der Spieler zum Zeitpunkt der Kollision aber durch den Effekt des *Invulnerability*-Juwels unverwundbar, werden ihm keine Punkte abgezogen. Weiterhin wird ein Kollisions-Sound als auditives Feedback für den Spieler abgespielt, damit dieser genau weiß, dass er mit der Oberfläche zusammengestoßen ist. Der Sound unterscheidet sich hierbei von Flugobjekt zu Flugobjekt.

5.9 Sound

Durch das Hinzufügen von verschiedenen Soundeffekten kann der Gesamteindruck für das Spiel positiv verstärkt werden. Es macht nicht nur mehr Spaß, wenn der Spieler zusätzlich ein auditives Feedback auf seine Aktionen erhält, sondern es hilft auch erheblich dabei, ausgeführte Aktionen oder sonstige Events im Spiel besser einschätzen zu können. Würde zum Beispiel kein Sound abgespielt werden, wenn der Spieler mit seinem Flugobjekt gegen ein anderes Objekt fliegt, würde ihm das möglicherweise gar nicht auffallen.

Dabei sollten die Soundeffekte möglichst passend zu den jeweiligen Situationen oder Objekten ausgewählt werden. So erhielt jedes Flugobjekt seinen eigenen Sound, damit dieser dessen „Antriebsweise“ und dessen Material entspricht. Bei der Drohne handelt es sich um ein Rotorengeräusch, welches zu den rotierenden Propellern passt, wohingegen Navi den Sound aus dem ursprünglichen Zelda-Spiel besitzt. Bei dem Raumschiff ertönt ein Triebwerksgeräusch, während sich dieses vorwärts bewegt und dabei die Spuren des Triebwerks hinter sich her zieht, und beim Papierflieger der Sound eines leichten Windes. Weiterhin ist auch der Sound, der bei einer Kollision abgespielt wird, für jedes Flugobjekt unterschiedlich.
Die Kraftfelder und Juwelen besitzen ebenfalls einen eigenen Sound: Die Kraftfelder erzeugen ein leises, dauerhaftes elektrisches Summen, während sie aktiv sind, und es wird ein Deaktivierungs-Sound abgespielt, sobald sie abgeschaltet werden, wenn das Flugobjekt sie durchquert. Weiterhin wird ein Sound für das Einsammeln von Juwelen abgespielt, der an das Einsammeln von Münzen in alten Konsolen-Spielen erinnert. Dabei ist dieser

Sound höher, wenn ein Juwel mit positivem Effekt und tiefer, wenn ein Juwel mit negativem Effekt eingesammelt wurde, damit der Spieler ein zusätzliches Feedback auf die Art des eingesammelten Juwels erhält.

Die Sounddateien werden in Unity als Komponenten an die entsprechenden GameObjects angefügt und über die Skripts angesprochen, wo das Abspielen dieser gestartet oder beendet wird. Dabei können verschiedene Einstellungen vorgenommen werden, wie zum Beispiel die Lautstärke oder ob der Sound in Dauerschleife abgespielt werden soll. Eine für AR-Anwendungen wichtige Einstellung sind hierbei die *3D Sound Settings*, bei denen man eine minimale und maximale Distanz einstellen kann, in der der Sound zu hören ist. Dadurch kann der Effekt erreicht werden, dass ein Sound einen bestimmten Ursprung im dreidimensionalen Raum hat, von welchem er sich ausbreitet und bei zunehmender Entfernung von diesem Punkt an Lautstärke verliert. Somit entsteht der Eindruck, ein Geräusch würde von einem bestimmten Objekt ausgehen, und je näher sich der Spieler diesem Objekt nähert, desto lauter wird der Sound abgespielt. Ist er hingegen zu weit entfernt, kann er den Sound nicht mehr hören, wodurch die Immersion stark gesteigert wird. Der Sound für die verschiedenen Objekte und Situationen wurde zum einen Teil selbst aufgenommen und zum anderen Teil der Soundeffekt-Datenbank ZapSplat[11] entnommen. Die Sounds für Navi stammen von Youtube[12].

[11] `https://www.zapsplat.com/`, Abgerufen am 20.08.2019

[12] `https://www.youtube.com/watch?v=wOFVrjL-XBM`, Abgerufen am 20.08.2019

6 Evaluation der Anwendung

In diesem Abschnitt soll das erstellte Spiel im Rahmen einer Nutzungsstudie untersucht werden. Hierfür werden zuerst verschiedene Ziele definiert, worauf aufbauend ein entsprechender Test geplant und durchgeführt wird. Danach werden die Testergebnisse anhand dieser Ziele ausgewertet und das AR-SDK hinsichtlich der Tauglichkeit für die Entwicklung von AR-Anwendungen bewertet.

6.1 Zielsetzung

Durch den Nutzungstest sollen die verschiedenen Kernaspekte des Spiels *pARcours* untersucht werden, wobei der Fokus auf der Umsetzung des Spiels mittels ARCore, dem Levelaufbau und den verschiedenen Steuerungen liegt. Es soll getestet werden, inwieweit die Verwendung von AR für ein Spiel dieser Art gewinnbringend ist und wie gut sich das ARCore SDK für die Umsetzung eignet. Dabei kommt es vor allem auf die Stabilität, Präzision und Performanz des von ARCore zur Verfügung gestellten Trackings an. Weiterhin sollen die zwei verschiedenen Arten, wie der Spieler die Level aufbauen kann, betrachtet werden, sodass eine Aussage darüber getroffen werden kann, ob eine, und wenn ja, welche dieser beiden von den Spielern bevorzugt wird und aus welchem Grund. Die vier verschiedenen Eingabe- bzw. Steuerungsmöglichkeiten sollen auf ihre Eignung für ein AR Mobile Game überprüft werden, wobei besonders die Intuitivität, das Handling und der Spaßfaktor eine große Rolle spielen. Zuletzt sollen durch die Meinung der Nutzer über die allgemeine Umsetzung des Spiels und den Spielspaß Rückschlüsse darüber gezogen werden, ob AR im Bereich Mobile Gaming auch in Zukunft eine große Rolle spielen könnte. Dabei bezieht sich der Test allein auf den *pARcours*-Spielmodus, da dieser den Hauptteil des Spiels ausmacht und allen nötigen Funktionen in diesem getestet werden können. Durch den Nutzungstest können außerdem Schwachstellen erkannt werden, woraus sich verschiedene Optimierungsmöglichkeiten ergeben, die zukünftig umgesetzt werden können.

6.2 Planung und Durchführung

Um ein möglichst genaues Bild über die Meinungen der Teilnehmer der Nutzungsstudie zu erhalten, wurde ein Fragebogen entwickelt, der nach einem Testdurchlauf des Spiels von diesen ausgefüllt werden sollte. Der Fragebogen besteht aus vier verschiedenen Themenblöcken mit insgesamt 45 Fragen zum Spiel und zwei abschließenden, personenbezogenen Fragen. Der erste Block bezieht sich auf die Funktionen des ARCore SDKs und die Qualität und Geschwindigkeit des Trackings in Bezug auf die verschiedenen Objekte (Oberflächen, virtuelle Objekte und Imagemarker), die von

ARCore erkannt und verfolgt werden können. Im zweiten Block soll herausgefunden werden, welche Art des Levelaufbaus bei den Spielern gut ankommt und weshalb. Im dritten Block werden die einzelnen Steuerungsvarianten bewertet und einander gegenübergestellt und im vierten Block wird die allgemeine Meinung zur Spielidee, dem Spielprinzip und dem Spielspaß abgefragt.
Dabei gibt es sowohl offene als auch geschlossene Fragen. Bei den geschlossenen Fragen handelt es sich um Hypothesen, die mittels einer vierstufigen Antwortskala bewertet werden können, sodass die Meinung der Testpersonen zu den einzelnen Themenblöcken durch eine Likert-Skala [21] gemessen werden kann. Dabei kann die Antwort pro Aussage zwischen absoluter Ablehnung und vollkommener Zustimmung liegen. Hierbei wurde auf die neutrale Mitte verzichtet, sodass sich die Teilnehmer für eine Seite entscheiden mussten. Zusätzlich wurden einige offene Fragen gestellt, bei denen die Probanden frei antworten und ihre Meinung äußern konnten. Dadurch können weiterführende Aspekte festgehalten werden, die nicht Teil der geschlossenen Fragen sind. Der vollständige für den Nutzungstest verwendete Fragebogen befindet sich im Anhang A.1.

Durchgeführt wurde die Nutzungsstudie mit 13 Testpersonen, die aus verschiedenen Altersgruppen ausgesucht wurden und unterschiedliche Erfahrungen im Bereich von Computerspielen und/oder Mobile Games haben sollten. Dadurch sollte überprüft werden, ob das Spiel nur für „Hardcore-Gamer“ oder auch für Gelegenheitsspieler interessant ist oder ob es sogar Leute, die eigentlich gar nicht spielen, dazu animieren könnte. Der Test wurde an mehreren Tagen in unterschiedlichen Umgebungen zu unterschiedlichen Uhrzeiten durchgeführt, da dieses Spiel auch dafür gedacht ist, dass es jederzeit und überall gespielt werden kann. Dadurch kann es zwar bei bestimmten Fragen (umgebungs- und lichtbedingt) zu Abweichungen einzelner Meinungen vom Durchschnitt kommen, jedoch ist dies dann ein Hinweis darauf, dass das Spiel in bestimmten Situationen noch nicht so optimal funktioniert wie in anderen, sodass dahingehend Verbesserungen vorgenommen werden können.
Da die Testpersonen aufgrund unterschiedlicher Vorerfahrungen mit Mobile Games unterschiedlich lange brauchen, um sich in das Spiel einzuarbeiten, und auch ein verschiedenes Maß an Kreativität und Enthusiasmus zeigen, wenn es um das Aufbauen von eigenen Leveln geht, wurde bei der Durchführung kein Zeitlimit angesetzt. Die Testpersonen sollten so lange ausprobieren und rumspielen können, wie sie Lust hatten, aber auch nicht gezwungen werden, länger zu spielen, als sie wollten. Die Nutzungsdauer des Spiels variierte bei dem Test zwischen 20 Minuten und einer Stunde. Vor dem Test erhielt jeder Proband eine kurze Erklärung zum Ziel dieser Nutzungsstudie, zu dem Spiel an sich und dem Spielablauf. Weiterführende Erklärungen zu den verschiedenen Modi, Steuerungen und Objekten im

Level erhielt die Testperson während der Durchführung, wenn diese benötigt wurden. Dadurch wurde die Testperson dazu angeregt, die verschiedenen Funktionen selbstständig auszuprobieren, wodurch sich Rückschlüsse auf die Selbstbeschreibungsfähigkeit des Spiels ziehen lassen. Die einzigen Vorgaben, die gemacht wurden, waren, dass der Proband beide Arten des Levelaufbaus im *pARcours*-Modus und alle vier Eingabemöglichkeiten austesten sollte, damit die entsprechenden Blöcke im Fragebogen beantwortet werden konnten. Sonst stand es den Testpersonen frei, mehr auszuprobieren und zu spielen. Nachdem das Spiel getestet worden war, wurde den Teilnehmern der Fragebogen ausgehändigt, welchen sie im Anschluss ausfüllen sollten. Weiterhin hatten sie die Möglichkeit, zusätzliche Anmerkungen und Verbesserungsvorschläge zu äußern.

6.3 Auswertung der Testergebnisse

An der Nutzungsstudie nahmen insgesamt 13 Testpersonen aus unterschiedlichen Altersgruppen teil, wobei die jüngste Testperson 20 Jahre und die älteste 58 Jahre alt war. Das durchschnittliche Alter betrug 32,6 Jahre. Drei der Testpersonen gaben an, gar keine Spielerfahrung mit Computerspielen und/oder Mobile Games zu haben und die restlichen zehn ordneten ihre Erfahrung bei hoch ein. Dadurch konnten verschiedene Beobachtungen für die „Anfänger“ und die „Fortgeschrittenen“ gemacht werden. Dabei fiel auf, dass die älteren Teilnehmer durchschnittlich weniger Spielerfahrung angaben als die jüngeren. Zudem fiel es den weniger erfahrenen Spielern oft schwerer, sich mit dem Spiel und den verschiedenen Steuerungen vertraut zu machen und benötigten daher durchschnittlich mehr Zeit zum Testen. Weiterhin wurde von zwei der drei Probanden ohne Spielerfahrung geäußert, dass sie Probleme mit der 3D-Wahrnehmung und der Einschätzung der Position des Flugobjekts im Raum hätten. Das könnte darauf zurückzuführen sein, dass Personen, die häufiger Computerspiele oder ähnliches spielen, mit der Zeit ein besseres Gefühl für die Einschätzung der Position von virtuellen Objekten in dreidimensionalen Szenen, die auf einem zweidimensionalen Bildschirm angezeigt werden, entwickeln als Personen, die keine solchen Spiele spielen.
Die vier Themenblöcke werden separat ausgewertet, wobei für jeden Block in einem Balkendiagramm visualisiert wird, wie die Testpersonen die verschiedenen Aussagen der geschlossenen Fragen bewertet haben. Dabei stellt die x-Achse die Anzahl der Testpersonen und die y-Achse die verschiedenen Aussagen des jeweiligen Blocks dar. Mittels verschiedener Farben werden die vier Abstufungen der Bewertungsskala verdeutlicht, mit denen die einzelnen Aussagen bewertet werden konnten. Unter der Annahme, dass die vier Abstufungen gleichmäßige Abstände untereinander besitzen, konnte die Bewertungsskala als eine Intervallskala betrachtet werden, wodurch sich der durchschnittliche Wert für die einzelnen Aussa-

gen berechnen ließ. Dabei wurden die Stufen mittels Zahlen codiert: -2 für „stimme gar nicht zu“, -1 für „stimme eher nicht zu“, 1 für „stimme eher zu“ und 2 für „stimme voll zu“. Durch Berechnung des Mittelwerts kann somit für jede Aussage eine (unterschiedlich starke) Ausprägung entweder in positive oder negative Richtung festgestellt werden, die daraufhin interpretiert bzw. ausgewertet werden kann (die kompletten Auswertungstabellen befinden sich im Anhang A.2).

Block 1 - ARCore, Tracking und Performanz In diesem Block (Abbildung 56) geht es vor allem um die Funktionalitäten von ARCore und die Qualität des Trackings für die virtuellen Objekte, die Imagemarker und die Oberflächen, auf welchen das Spiel aufbaut. So fanden die Testpersonen, dass Oberflächen vom System im Durchschnitt generell schnell erkannt worden sind, wobei horizontale Oberflächen, wie der Boden oder Tischplatten, deutlich schneller erkannt wurden als vertikale, wie zum Beispiel Wände. Das kann daran liegen, dass in einigen Testumgebungen hauptsächlich glatte weiße Wände zur Verfügung standen, auf denen das System keine Feature Points finden konnte und diese deshalb nicht oder nur sehr langsam erkannt wurden. Böden oder Tischplatten hingegen besitzen in der Regel mehr Struktur, wodurch die Erkennung leichter und schneller vonstatten geht. Im Ganzen lief die Oberflächenerkennung in annehmbarer Geschwindigkeit. Dass die erkannten Oberflächen in ihrer Größe und Form mit den realen weitestgehend übereinstimmen, wurde zwar von keiner Testperson komplett verneint, jedoch stimmten die meisten hier nur teilweise zu. Das zeigt, dass die Oberflächenerkennung von ARCore zwar schon ganz gut, aber immer noch ausbaufähig ist. Eine breitere Varianz an Meinungen gab es hingegen bei der Aussage, dass die Imagemarker schnell erkannt wurden. Das kann darauf zurückzuführen sein, dass in den verschiedenen Testumgebungen unterschiedliche Lichtbedingungen vorlagen, wodurch die Imagemarker mal schneller und mal weniger schnell erkannt wurden. Das zeigt, dass die Erkennung von diesen sehr stark von den Lichtbedingungen abhängig ist. Andererseits könnte auch die subjektive Einschätzung der Testpersonen, was die Geschwindigkeit angeht, variieren. Dass die Imagemarker richtig erkannt wurden und dass das generelle Tracking von Objekten auch über einen längeren Zeitraum stabil blieb, wurde dagegen von allen Testpersonen ausschließlich positiv bewertet. Auch kam es durchschnittlich kaum zu Ausfällen des Trackings oder Rucklern während des Spiels. All dies zeigt, dass das merkmalsbasierte Tracking von ARCore im Großen und Ganzen schon sehr präzise, performant und stabil funktioniert.
Auch die Visualisierungshilfe für die getrackten Oberflächen und die Darstellung von Schatten und Verdeckung wurde als hilfreich bewertet, um die Lage von virtuellen Objekten besser einschätzen zu können. Dabei ist

Abbildung 56: Auswertung von Block 1 - ARCore, Tracking und Performanz.

aufgefallen, dass einige Testpersonen den Schatten von virtuellen Objekten erst bemerkt haben, als im Fragebogen explizit danach gefragt wurde. Daraus lässt sich folgern, dass die Darstellung von Schatten von den meisten Leuten als selbstverständlich wahrgenommen wird, wodurch gar nicht mehr auf diesen geachtet wurde. Fehlt der Schatten jedoch, würde das viele Menschen irritieren. Da die Level auf erkannten Oberflächen aufgebaut werden, war für die meisten Testpersonen die Visualisierung der Oberflächen wichtiger als die Darstellung von Schatten und Verdeckung, da es ihnen laut eigener Aussage schwerer gefallen wäre, ein Level aufzubauen, wenn sie nicht gesehen hätten, wo schon Oberflächen erkannt worden sind. Die durchschnittlich guten Bewertungen in diesem ersten Block zeigen, dass das Tracking des ARCore SDKs aus Sicht der meisten Probanden im Allgemeinen schon sehr gut funktioniert, auch wenn einige Funktionen bei weniger optimalen Bedingungen noch verbesserungswürdig sind.

Block 2 - Aufbau der Level Im zweiten Block (Abbildung 57) wurden der Levelaufbau mittels Imagemarkern und mittels Touch-Input gegenübergestellt. Dabei zeigte sich, dass die Platzierung von virtuellen Objekten mittels der Touch-Input-Methode von den Testpersonen sowohl als intuitiver als auch als besser funktionierend eingestuft wurde, wobei beide Methoden eine deutlich positive Tendenz aufweisen. Dass die Imagemarker-Methode nicht ganz so gut eingestuft wurde, kann daran liegen, dass das Erkennen der Imagemarker, wie im vorherigen Abschnitt gezeigt wurde, nicht in allen Situationen optimal funktioniert hat. Die Testperson mit der einzigen negativen Bewertung für den Touch-Input gab als Grund dafür an, dass sie generell mit den Touch-Gesten auf mobilen Geräten nicht so

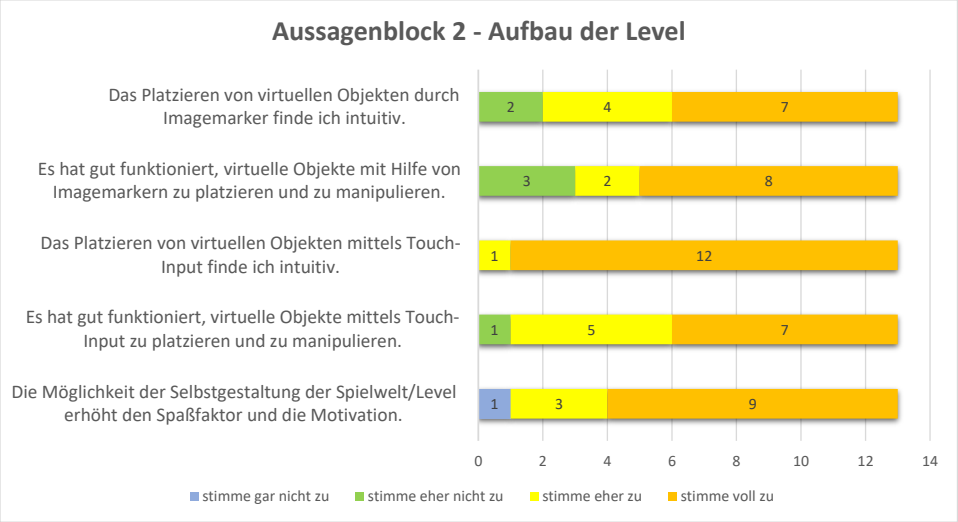


Abbildung 57: Auswertung von Block 2 - Aufbau der Level.

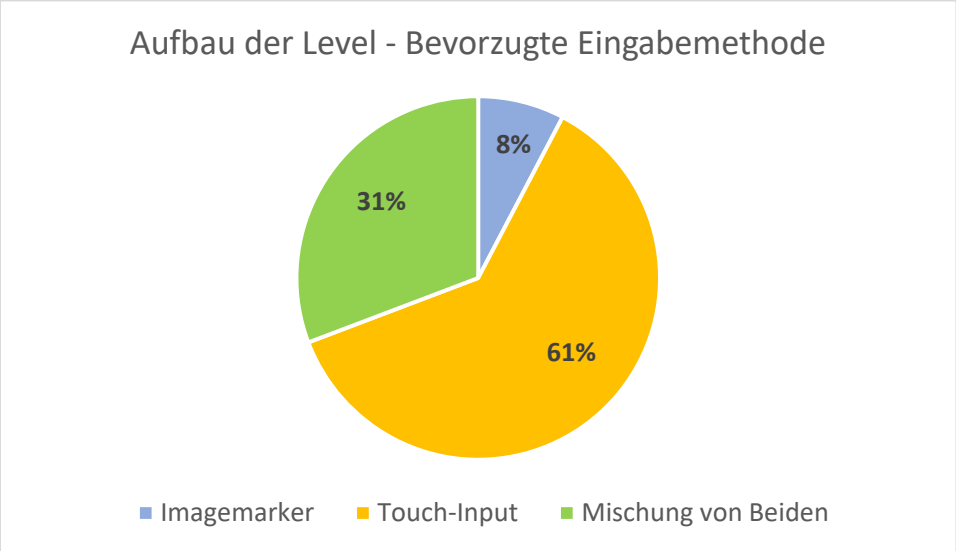


Abbildung 58: Auswertung von Block 2 - Bevorzugte Eingabemethode für den Aufbau der Level.

gut zurechtkäme, und dies somit nicht an meinem Spiel liegen würde. Die Testpersonen stimmten auch der Aussage, dass die Möglichkeit, die Spiellevel selbst gestalten zu können, den Spaßfaktor und die Motivation erhöht, größtenteils komplett zu, da dies eine flexible Gestaltung der Level und auch des Schwierigkeitsgrades mit sich bringen würde, und so für mehr Abwechslung sorgt. Auch hier zeigen die durchschnittlich positiven Bewertungen, dass die Selbstgestaltung der Spielwelt sowohl gut bei den Spielern ankommt als auch (mit wenigen Ausnahmen) gut funktioniert.

Zuletzt wurden die Testpersonen noch nach ihrer persönlichen Meinung gefragt, welche Variante zur Selbstgestaltung der Level sie bevorzugen würden und aus welchem Grund (Abbildung 58). Die Mehrheit von acht Probanden würde hierbei den Touch-Input bevorzugen, da dieser sowohl schneller und einfacher in der Handhabung als auch zuverlässiger beim Tracking wäre und weniger Bewegung des Spielers erfordert. Weiterhin wurde angegeben, dass es als nervig empfunden wurde, wenn die Erkennung der Imagemarker unter manchen Bedingungen nicht sofort funktioniert hat, und dass die Verwendung von diesen auch nicht vorteilhaft ist, wenn man zum Beispiel draußen spielen möchte. Vier der Testpersonen würden eine Mischung aus beiden Varianten bevorzugen, je nach Situation und in welcher Umgebung gespielt wird. So würden sie beispielsweise für Kinder oder in größeren Umgebungen die Imagemarker wählen, da es als spaßig empfunden wurde, in der Gegend herum zu laufen und seinen Parcours einscannen zu können, aber für den schnellen Spielspaß oder bei

kleiner Umgebung eher den Touch-Input, da dieser weniger Zeit und Platz benötigt. Nur eine Testperson würde sich für die Imagemarker-Variante entscheiden, da sie einen größeren Bildschirm (z. B. Tablet) für sinnvoller hält, wenn man die virtuellen Objekte über den Touchscreen auswählen, platzieren und manipulieren möchte. Dies zeigt, dass obwohl beide Varianten durchschnittlich positiv bewertet wurden und auch ihre Vor- und Nachteile haben, die Touch-Input Variante eher genutzt werden würde als die Imagemarker, aus dem Hauptgrund, dass diese von den Testpersonen als bequemer und schneller erachtet wird.

Block 3 - Steuerung der Flugobjekte Im dritten Block wurden die Intuitivität, das Handling und der Spielspaß bei der Verwendung der vier verschiedenen Steuerungen bewertet und miteinander verglichen (Abbildung 59). Dabei konnten die Testpersonen die Reihenfolge, in der sie die Steuerungen testen wollten, selber bestimmen. Betrachtet man die Durchschnittswerte, stellt man fest, dass sich trotz persönlicher Präferenzen der Testpersonen in Bezug auf einzelne Steuerungen eine klare Reihenfolge bei der Beliebtheit dieser abzeichnet. So wird das physische Gamepad mit einer durchschnittlichen Bewertung von 1,846 klar als beste Steuerungsmethode bewertet, gefolgt von den virtuellen Joysticks mit 1,18 und dem Touch-Input mit 0,615. Die Steuerung mittels Kamera mit einem Wert von -0,487 wurde hierbei in allen drei Kategorien (schnell zu erlernen, selbsterklärend und Spaßfaktor) durchweg negativer bewertet als die anderen drei Eingabemethoden. Das kann daran liegen, dass dies eine für die meisten Spieler ungewohnte Steuerung war, die zudem komplizierter ist und mehr Geschick erfordert. Am schlechtesten wurde dabei die Selbstbeschreibungsfähigkeit bewertet, was zeigt, dass viele der Probanden gar nicht wussten, wie man mit dieser Methode steuern sollte. Eine Verbesserung hierfür wäre, eine genauere Anleitung oder ein Tutorial anzubieten, in dem der Spieler die Steuerung erlernen kann. Auch die Eingabe mittels Touch-Input wurde als wenig selbsterklärend eingestuft, wobei diese schneller erlernt wurde, sobald die Testperson herausgefunden hatte, wie das Flugobjekt in welche Richtung gesteuert wird. Somit wurde auch der Spaßfaktor bei dieser Eingabe mehr positiv als negativ eingeschätzt. Durchweg positiv in allen Kategorien wurde das Gamepad bewertet, sowohl von den Testpersonen ohne als auch von denen mit viel Spielerfahrung. Dadurch zeigt sich, dass diese Steuerung schnell zu erlernen und selbsterklärend ist und für viele Nutzer angenehmer in der Handhabung. Auch wenn die virtuellen Joysticks mit dem gleichen Prinzip funktionieren, schnitten diese doch weniger gut ab als das Gamepad. Das haben die Testpersonen damit begründet, dass es leichter ist, für die Eingabe ein physisches Gerät in der Hand zu halten als die reine Bildschirmeingabe. Weiterhin wurde durch das Festhalten eines physischen Controllers bei den meisten Probanden auch das

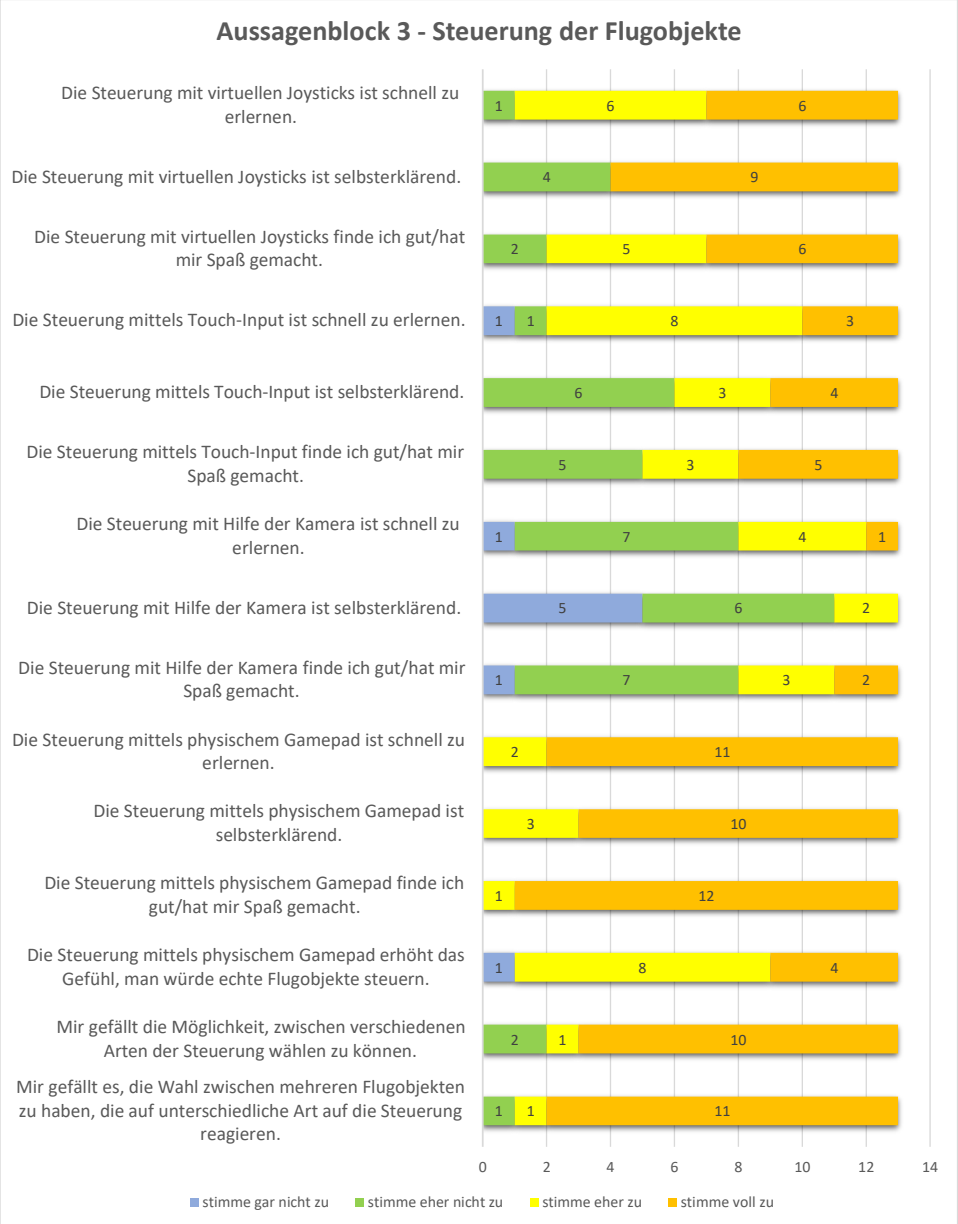


Abbildung 59: Auswertung von Block 3 - Steuerung der Flugobjekte.

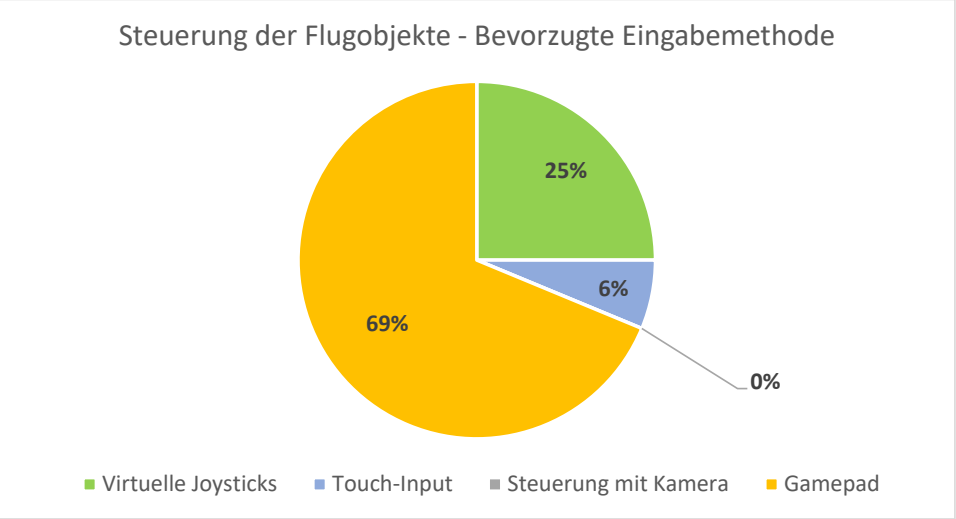


Abbildung 60: Auswertung von Block 3 - Bevorzugte Eingabemethode für die Steuerung der Flugobjekte.

Gefühl erhöht, als würden sie ein echtes Flugobjekt steuern. Die Möglichkeit, sowohl zwischen verschiedenen Steuerungsarten als auch zwischen mehreren Flugobjekten mit unterschiedlichem Flugverhalten wählen zu können, fand die Mehrzahl an Testpersonen sehr gut. Die Auswertung dieses Blocks zeigt, dass bei allen Eingabemethoden der Punkt der Selbstbeschreibungsfähigkeit tendenziell am schlechtesten bewertet wurde, was darauf schließen lässt, dass sich die Testpersonen mehr Erklärung zu diesen innerhalb des Spiels gewünscht hätten, besonders bei der Steuerung mittels Kamera. Auch ist aufgefallen, dass die Probanden meistens eine bestimmte Steuerung bevorzugt und somit positiver bewertet haben, und die anderen Steuerungen dann vergleichsweise schlechter.

Die Frage nach der bevorzugten Steuerung für die Flugobjekte (Abbildung 60) haben elf der dreizehn Teilnehmer mit dem Gamepad beantwortet, da dieses für sie die einfachste und präziseste Steuerungsart gewesen ist. Die Testpersonen fanden es gut, einen physischen Controller in der Hand zu halten und konnten durch die haptischen Joysticks auch besser einschätzen, welche Eingabe das Flugobjekt wie im Raum bewegt. Zudem konnten beide Hände gleichzeitig verwendet werden, ohne den Bildschirm zu verdecken. Drei der Testpersonen, die das Gamepad bevorzugen, haben auch gleichzeitig die virtuellen Joysticks angegeben, da sie fanden, dass sich diese genauso präzise verhalten wie die physischen Joysticks und das Steuerungsprinzip identisch ist. Insgesamt wurden die virtuellen Joysticks von vier Personen genannt, mit der Begründung, dass für diese keine zusätzliche Hardware benötigt wird und sie angenehm zu bedienen sind. Au-

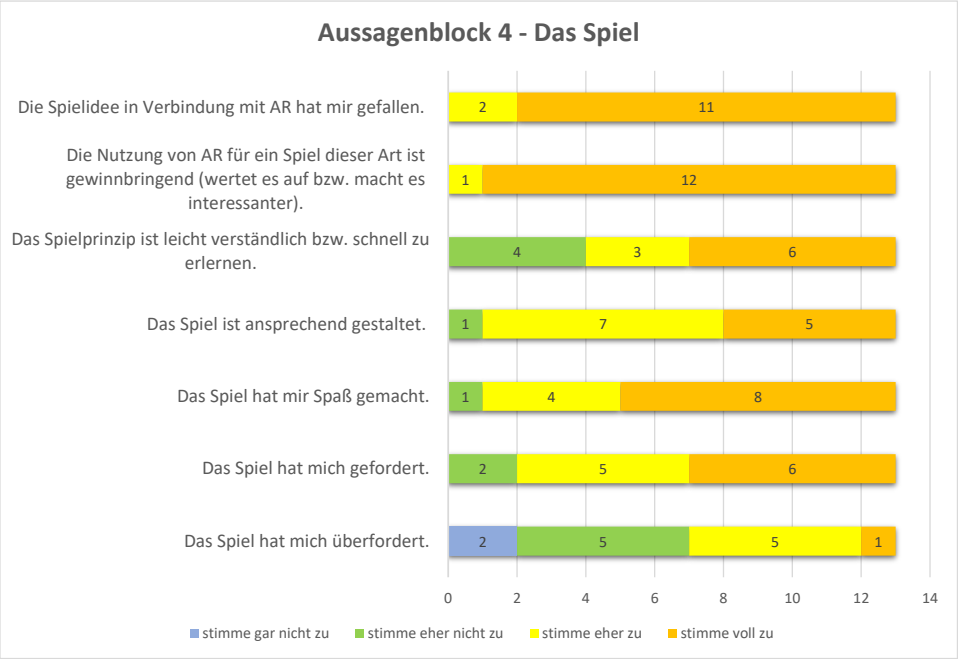


Abbildung 61: Auswertung von Block 4 - Das Spiel.

ßerdem waren die Testpersonen eine solche Steuerung aus anderen Spielen schon gewohnt. Der Touch-Input wurde von nur einer Testperson aufgrund seiner einfachen Bedienung bevorzugt. Die Steuerung mittels Kamera würde keiner der Probanden wählen, da diese als gewöhnungsbedürftig und zu unpräzise eingestuft wurde. Außerdem sei das Fadenkreuz in manchen Situationen nur schwer zu erkennen gewesen. Dieses Problem könnte zum Beispiel durch eine auffälligere Farbe behoben werden. Durch dieses Ergebnis lässt sich folgern, dass die Leute beim Spielen lieber auf die altbekannten Eingabemethoden (physische oder virtuelle Joysticks) zurückgreifen, da sie diese gewohnt sind und sie gut funktionieren.

Block 4 - Das Spiel Im vierten Block (Abbildung 61) wurden allgemeine Fragen zu dem Spiel gestellt. Außerdem hatten die Testpersonen die Möglichkeit aufzuschreiben, was ihnen besonders gut oder besonders schlecht gefallen hat, und sie konnten Verbesserungsvorschläge machen. Der großen Mehrheit von elf Testpersonen hat die Spielidee in Verbindung mit AR sehr gut gefallen, und sogar zwölf haben der Aussage voll zugestimmt, dass die Nutzung von AR das Spiel interessanter macht und aufwertet. Daraus kann gefolgert werden, dass AR in Verbindung mit Mobile Gaming in Zukunft einen größeren Stellenwert einnehmen kann, besonders weil sich die visuellen Tracking-Verfahren für mobile Geräte auch weiterhin verbessern werden. Darüber hinaus wurde auch die Gestaltung des Spiels als überwie-

gend positiv bewertet, jedoch fanden vier der Testpersonen, dass das Spielprinzip im Allgemeinen nicht so leicht verständlich und nicht so schnell zu erlernen ist, wobei es sich dabei überwiegend um die Testpersonen ohne vorherige Spielerfahrung gehandelt hat. Durch ein einführendes Tutorial, welches das Spielprinzip und die grundlegenden Funktionen erklärt, könnte dieses Problem behoben werden. Die Mehrheit der Testpersonen hatte Spaß an dem Spiel, wobei sie gleichzeitig angaben, dass das Spiel sie auch gefordert hat. Die knappe Hälfte der Testpersonen war sogar etwas überfordert, was sich dadurch erklären lässt, dass zu Anfang nur wenige Erklärungen gegeben wurden, da sie die einzelnen Steuerungen und Funktionen erst einmal selber ausprobieren sollten. Hierbei ist auch aufgefallen, dass die Testpersonen ohne Spielerfahrung generell schneller überfordert waren, da diese besonders Probleme mit den ungewohnten Steuerungen hatten und mehr Zeit benötigten, um in das Spiel rein zu finden. Dennoch zeigt die Auswertung, dass das gesamte Spiel im Durchschnitt relativ positiv bewertet wurde, auch wenn bei manchen Probanden mehr Einarbeitungszeit erforderlich war.

Bei den abschließenden offenen Fragen gaben fünf der Testpersonen an, dass ihnen die Auswahl der Flugobjekte besonders gut gefallen hat, besonders da diese alle ein unterschiedliches Flugverhalten besitzen. Auch die große Auswahl an virtuellen Objekten mit optisch passenden 3D-Modellen und die Möglichkeit, die Spiellevel selber zu gestalten, gefiel den meisten Testpersonen. Dadurch ließe sich das Spiel flexibel gestalten und es würde keine eintönige Routine entstehen. Sieben der Probanden fanden die Idee gut, ein Flugspiel mit AR zu verbinden und somit die eigene Umgebung als Spielwelt nutzen zu können. Dadurch ergäbe sich oftmals das Gefühl, man würde mit seinem Flugobjekt wirklich mit den Gegenständen in der Realität zusammenstoßen können. Weiterhin wurde von zwei Testpersonen angegeben, dass ihnen die Qualität des Trackings und die Verdeckung von virtuellen Objekten durch reale Oberflächen gut gefallen hat. Auch als positiv genannt wurden die verschiedenen Steuerungen, das Spielprinzip, ein ansprechendes UI-Design, die akustischen Effekte, die Juwelen mit ihren unterschiedlichen Effekten und dass man dieses Spiel überall, sogar draußen, spielen kann.

Nicht so gut fanden die Testpersonen, dass bei Änderung der Steuerung oder des Spielmodus das Level gelöscht wurde und deshalb wieder neu aufgebaut werden musste. Auch der Schwierigkeitsgrad in Bezug auf das Erlernen des Spielprinzips und der Steuerungen wurde bemängelt. Andererseits wurde auch von einer Testperson als negativ angesehen, dass es keine Möglichkeit gab, zu verlieren, was wiederum darauf schließen lässt, dass sich diese Person einen höheren Schwierigkeitsgrad gewünscht hätte. Auch das Tracking der Imagemarker wurde als noch etwas zu langsam und

unpräzise bezeichnet. Bei kleiner Skalierung der virtuellen Objekte wurden Probleme beim Einschätzen der Kollider genannt, wodurch die Kollisionserkennung und das Durchqueren der Checkpoints fehleranfällig wurden. Durch Anpassen der Kolliderеinstellungen konnte dieses Problem jedoch behoben werden.

Zuletzt konnten die Testpersonen Verbesserungsvorschläge und Anregungen für das Spiel angeben. Drei der Testpersonen nannten hierbei die Möglichkeit, erstellte Level abspeichern und wiederverwenden zu können, damit diese nicht jedes Mal neu aufgebaut werden müssen. Weiterhin könnte man so gegebenenfalls auch entweder das Level einem Freund schicken, damit sich dieser am selben Level versuchen kann, oder sogar zusammen spielen. Zwei Testpersonen wünschten sich eine Anleitung mittels Tutorial, welches das Spielprinzip und die Steuerungen kurz erläutert, und eine weitere Testperson hätte die schon vorhandene Anleitung lieber auf Deutsch gehabt (die Anleitung war wie das Spiel nur auf Englisch). Der Proband, der bemängelt hatte, dass man nicht verlieren könne, hat vorgeschlagen, ein Lebenspunktesystem und sich bewegende Hindernisse oder Feinde einzubauen, damit dies möglich wird. So könnte der Spieler bei Kollisionen mit Hindernissen oder Feinden Lebenspunkte verlieren, und wären diese auf null, hätte er automatisch verloren. Eine andere Testperson wünschte sich bei Kollisionen nicht nur einen akustischen, sondern auch einen optischen Effekt. Auch wurde angegeben, dass bei der Skalierung und Spielen auf Tischgröße die Objekte der Größe der Imagemarker entsprechen sollten, damit man sich diese bei der Levelgestaltung besser vorstellen kann.

7 Ausblick und Fazit

Im Rahmen dieser Arbeit wurde ein AR-Spiel aufbauend auf dem ARCore SDK konzipiert, implementiert und evaluiert. In den folgenden Abschnitten gebe ich zuerst einen Ausblick auf Optimierungs- und Erweiterungsmöglichkeiten für die Anwendung, bevor ich ARCore hinsichtlich seiner Tauglichkeit für die Entwicklung von AR-Anwendungen bewerte und die erworbenen Kenntnisse in einem abschließenden Fazit zusammenfasse.

7.1 Optimierungs- und Erweiterungsmöglichkeiten

Mit Blick auf die Testergebnisse der Nutzungsstudie können vorhandene Funktionen des Spiels *pARcours* optimiert und das Spielkonzept erweitert werden. So haben sich viele der Testpersonen ein einführendes Tutorial gewünscht, welches unter anderem die Spielmechanik erklärt. Dabei könnte der Levelaufbau visuell mehr unterstützt werden, indem eine Schritt für Schritt Anleitung eingeblendet wird und entsprechende Stellen auf dem Bildschirm markiert werden. Es wäre auch ein kleines Tutorial für jede der Eingabemethoden von Vorteil, damit der Spieler diese besser erklärt bekommt und unter Hilfestellung ausprobieren kann.

Weiterhin könnte die Möglichkeit eingebaut werden, ein fertig gestaltetes Level speichern zu können, damit dieses wiederverwendet werden kann. Das funktioniert allerdings nur, wenn ARCore die Anchors der Objekte im Hintergrund weiter verfolgen kann, da diese an realen Oberflächen verankert werden. Wird das Tracking beendet, haben die Objekte keinen festen Platz mehr in der Umgebung. Daher könnten die Level nur innerhalb einer Session gespeichert werden, da bei einem erneuten Start der Anwendung die Umgebung komplett neu erkannt werden muss.
Eine andere Methode wäre, ein Level in Bezug auf einen Imagemarker abzuspeichern, sodass das komplette Level und alle enthaltenen Objekte durch erneutes Einscannen von diesem geladen werden könnten. So besteht aber die Gefahr, dass sich die virtuellen Objekte nicht mehr in die Umgebung einfügen, da sie nur noch einen zentralen Ankerpunkt besitzen und wenn dieser nur leicht verändert wurde, sind auch die Positionen der Objekte verändert. Außerdem würden sich diese bei Erweiterung der Umgebungsinformationen nicht mehr mit anpassen, und wenn der Spieler versuchen würde, das Level in einer komplett neuen Umgebung zu laden, würde dieses eventuell gar nicht mehr dort hineinpassen (außer es wurde auf einer komplett ebenen Fläche aufgebaut, wie sich auch eine in der neuen Umgebung befindet).

Mit Hilfe der *Cloud Anchors* von ARCore könnte realisiert werden, dass ein Spieler sein Level mit Freunden in der Umgebung teilen und mit diesen

zusammen spielen könnte. Dabei wäre ein Mehrspielermodus sowohl für eine Art Rennspiel denkbar, bei dem mehrere Spieler gegeneinander antreten und versuchen, schneller als die anderen durch die Checkpoints zu fliegen, als auch eine Art Unterstützer-Modus, wo der eine Spieler den anderen unterstützen kann, indem er für diesen beispielsweise Juwelen einsammelt.

Weiterhin könnte ein Lebenspunktesystem eingebaut werden, damit der Spieler auch verlieren kann. Kollidiert der Spieler mit seinem Flugobjekt, könnten ihm dafür Lebenspunkte abgezogen werden, und wenn diese auf null sind, wäre das Spiel verloren. Dabei könnte man für einen erhöhten Schwierigkeitsgrad bewegte Hindernisse oder Gegner einbauen, die versuchen, den Spieler daran zu hindern, durch die Checkpoints zu fliegen. Durch Hinzufügen eines AI-Systems (künstliche Intelligenz) für die Gegner wäre es sogar möglich, dass diese den Spieler verfolgen und abschießen können. Außerdem bietet das die Möglichkeit, einen computergesteuerten Gegner einzubauen, mit dem der Spieler um die Wette fliegen kann.

Es besteht auch die Möglichkeit, weitere Flug- oder Parcours-Objekte einzufügen, um mehr Abwechslung in das Spiel zu bringen. Mit den durch das erfolgreiche Beenden von Leveln erhaltenen Juwelen könnte es dem Spieler ermöglicht werden, sich weitere Flugobjekte freizuschalten oder die vorhandenen zu verbessern. So könnte ein Individualisierungssystem eingebaut werden, bei dem der Spieler mit den gesammelten Juwelen neue Farben oder neue Varianten seiner Flugobjekte kaufen kann, oder beispielsweise einen besseren Antrieb, damit diese schneller werden. Durch die Möglichkeit der individuellen Erweiterung der Flugobjekte könnte zusätzlich die Langzeitmotivation gesteigert werden, da der Spieler so die Flugobjekte sammeln und seine eigene „Flugstaffel“ aufbauen könnte.

Die vorhandenen Eingabemethoden, besonders die Steuerung mittels Kamera, sollten in Hinblick auf eine bessere Benutzerfreundlichkeit optimiert werden. Es wäre auch möglich, weitere Steuerungen zu implementieren, oder die in Abschnitt 5.7 beschriebene Kippsteuerung zu verbessern und einzufügen, um den Spielern eine weitere Alternative der Eingabe zu präsentieren und somit weitere Abwechslung ins Spiel zu bringen.

Zuletzt könnten verschiedene Aspekte des Trackings verbessert werden. So ist es momentan möglich, dass sich virtuelle Objekte, die mittels Imagemarkern platziert wurden, überschneiden können, wenn die entsprechenden Marker zu nah nebeneinander gelegt wurden. Dies sollte wie bei der Platzierung mittels Touch-Input erkannt werden, damit der Spieler darauf hingewiesen werden kann oder die entsprechenden Objekte erst erstellt werden, wenn die Imagemarker weit genug voneinander entfernt sind.

Auch würde es die Immersion fördern, wenn zusätzlich zu den Oberflächen auch spezifischere Modelle erkannt werden könnten, sodass man sogar einen Parcours mit realen Hindernissen aufbauen könnte. Es würde den Realismus und wahrscheinlich auch den Spaßfaktor erhöhen, wenn das virtuelle Flugobjekt nicht nur mit flachen Oberflächen, sondern mit beliebigen Objekten zusammenstoßen könnte, wodurch die komplette Umgebung zum eigentlichen Parcours werden würde. Zusätzlich könnte das Tracking der Hände hinzugefügt werden, wie es zum Beispiel bei der HoloLens der Fall ist, damit Gesten erkannt werden können und es dadurch möglich wäre, die virtuellen Objekte nicht nur über den Bildschirm, sondern mittels der eigenen Hände zu manipulieren, was auch wiederum mehr Spielspaß bringen könnte.

7.2 Bewertung der Eignung von ARCore für AR-Anwendungen

Grundsätzlich bietet ARCore ein SDK, mit welchem es Entwicklern einfach gemacht wird, eigene AR-Anwendungen für mobile Geräte zu implementieren, ohne auf zusätzliche Hardware zurückgreifen zu müssen. Dabei kann ARCore sehr leicht in Spielentwicklungsumgebungen, wie in diesem Fall Unity, eingebunden und verwendet werden. Es bietet ein verhältnismäßig schnelles, präzises und stabiles Tracking, welches allein auf dem Kamerabild arbeiten kann, sodass die Anwendung unabhängig ist und keine externe Hardware benötigt wird. Da das Spiel *pARcours*, in welchem eine Vielzahl von Objekten im Level gleichzeitig getrackt werden musste, auf dem Smartphone trotzdem flüssig lief, zeigt, dass das Tracking auch auf Smartphone-Hardware performant ist. Auch die Erkennung von realen Oberflächen lief größtenteils schnell und stabil und ist auf jeden Fall für Anwendungen, die die Umgebung nicht auf den Zentimeter genau erkennen müssen, geeignet. Jedoch ist sie noch verbesserungswürdig in Bezug auf merkmalsarme Oberflächen und die Präzision bei der Form und Größe der erkannten Oberflächen. Da es aber seit dem Update im August 2019 auch möglich ist, dass ARCore auf die Tiefenkamera des Geräts zugreifen kann, könnte dies die Präzision des Trackings zukünftig stark verbessern.

Die Funktion, dass Imagemarker für markerbasiertes Tracking verwendet werden können, ist zwar von Vorteil, aber hier muss ganz klar noch die Geschwindigkeit und Stabilität der Erkennung verbessert werden, damit dies auch unter nicht so optimalen Lichtbedingungen zuverlässig funktioniert. Auch die maximale Anzahl von 20 gleichzeitig verfolgten Bildern kann hier je nach Anwendung ein Nachteil sein. Vergleicht man die Imagemarker-Funktion mit der Markererkennung von Vuforia, so würde ich die von Vuforia vorziehen, da diese (zu diesem Zeitpunkt und meiner persönlichen Einschätzung nach) wesentlich schneller und stabiler läuft. Dafür besitzt ARCore aber weitere zusätzliche Funktionalitäten, mit denen sich auch an-

dere Arten von Anwendungen entwickeln lassen. So können zum Beispiel die *Cloud Anchors* für lokale Mehrspieler-Anwendungen genutzt werden, oder die *Augmented Faces*-Funktion, um Gesichtsregionen im Kamerabild erkennen zu können und diese beispielsweise mit virtuellen Modellen oder Texturen zu überlagern.

Ich finde, dass durch ARCore ein einfacher Einstieg in die Entwicklung von mobilen AR-Anwendungen ermöglicht wird. Das SDK bietet alle nötigen Funktionen, um die ersten Schritte zur eigenen AR-Anwendung zu machen und auch die Oberflächenerkennung besitzt viel Potential, was die Vielfältigkeit an Anwendungen betrifft, die darauf aufbauen können (siehe beispielsweise die Anwendungsbeispiele in Abschnitt 3.3). Somit ist ARCore auf jeden Fall für die Entwicklung von AR-Anwendungen auf mobilen Geräten geeignet und wird vermutlich auch in Zukunft durch Google weiter verbessert werden.

7.3 Fazit

Mit dem Ziel, aufbauend auf ARCore eine spielerische AR-Anwendung für mobile Geräte zu entwickeln, wurden zuerst die Grundlagen von Augmented Reality und Tracking betrachtet sowie die Funktionsweise des ARCore SDKs und seine Möglichkeiten. Danach wurde ein Konzept für einen AR-Parcours entwickelt, aus dem das Spiel *pARcours* entstanden ist, welches daraufhin unter Verwendung von ARCore umgesetzt wurde. Schließlich wurde die Anwendung mittels einer Nutzungsstudie evaluiert, wodurch sich Rückschlüsse auf die Tauglichkeit von ARCore für die Entwicklung von AR-Anwendungen ziehen ließen und mögliche Verbesserungen für das Spiel dargelegt werden konnten. Somit wurde das Ziel der Implementierung einer interaktiven AR-Anwendung mittels ARCore für mobile Geräte erreicht.
Das Ergebnis ist ein voll funktionsfähiges AR-Spiel für Android-Geräte, welches eine Kombination aus Renn- und Geschicklichkeitsspiel darstellt (Abbildung 62). Dieses wurde visuell ansprechend gestaltet und ist durch das merkmalsbasierte Tracking von ARCore unabhängig von externer Hardware und in jeder Umgebung spielbar, zwei Grundvoraussetzungen für mobile AR-Anwendungen. Durch die Möglichkeit der Selbstgestaltung der Spielwelt wird die Langzeitmotivation gefördert und der Spieler kann den Schwierigkeitsgrad selber bestimmen. Auch die verschiedenen Eingabemöglichkeiten und Steuerungsverhalten der Flugobjekte sorgen für Abwechslung und erhöhen den Spielspaß. Durch Fortführung des Spielkonzepts kann das Spiel auch in Zukunft um zusätzliche Funktionen erweitert werden, die in weiteren Nutzungsstudien getestet werden könnten, um repräsentativere Ergebnisse zu erhalten.

Abbildung 62: Das Spiel *pARcours* während der Nutzung durch einen Spieler.

Da sich schon jetzt ein steigendes Interesse an AR-Anwendungen in vielen Anwendungsbereichen abzeichnet und die Soft- und Hardware dahingehend immer weiter verbessert wird, kann man davon ausgehen, dass AR auch in Zukunft ein großes Thema sein wird. Wie ich in meiner Arbeit gezeigt habe, kann AR auch für den Bereich Mobile Gaming eine Bereicherung darstellen, sodass ich annehme, dass zukünftig weitere Spiele für mobile Geräte entwickelt werden, die AR nicht nur verwenden, um zusätzliche Informationen anzuzeigen, sondern wo AR einen Hauptbestandteil des Spiels darstellt.
Führt man die Idee eines AR-Flugspiels weiter, wäre auch die Entwicklung eines AR-Flugsimulators möglich, der nicht nur im Spiele-, sondern auch im Trainings-Bereich eingesetzt werden könnte. Da Drohnen viele Einsatzgebiete haben, wie zum Beispiel für Aufnahmen aus der Vogelperspektive an einem Filmset oder für die Landesvermessung bzw. Kartografie, Aufklärungsflüge bei der Polizei und dem Militär und sogar im Sport beim Drone-Racing, wäre es nützlich, einen AR-Drohnen-Flugsimulator zu benutzen, um Anfängern die Steuerung und das Flugverhalten erst einmal an einer virtuellen Drohne näher zu bringen oder eine bestimmte Route oder einen Parcours vorher testweise abzufliegen, ohne dabei Schäden an einer echten Drohne zu riskieren, da diese auch nicht gerade billig sind. So können die Personen erst mehr Sicherheit im Umgang mit der virtuellen Drohne bekommen, bevor sie eine reale Drohne steuern. Wie dieses Beispiel zeigt, sind die Einsatzmöglichkeiten für AR im mobilen Bereich vielfältig, sodass man darauf hoffen kann, dass auch in Zukunft weitere innovative AR-Anwendungen entwickelt werden, besonders auch weil in den letzten Jahren die nötigen technischen Voraussetzungen geschaffen wurden, um AR der breiten Masse zugänglich zu machen.

Literatur

[1] *A&E Crime Scene: AR by A&E Television Networks Mobile*. https://play.google.com/store/apps/details?id=com.aetn.xr.android.history.crimescene. – Abgerufen am 30.08.2019

[2] *Apple Inc. ARKit*. https://developer.apple.com/augmented-reality/. – Abgerufen am 10.09.2019

[3] *ARCore - Supported Devices*. https://developers.google.com/ar/discover/supported-devices. – Abgerufen am 16.09.2019

[4] *ARToolKit*. http://www.hitl.washington.edu/artoolkit/. – Abgerufen am 10.09.2019

[5] *Blender*. https://www.blender.org/. – Abgerufen am 10.09.2019

[6] *Curate by Sotheby's International Reality*. https://www.sothebysrealty.com/deu/curate. – Abgerufen am 30.08.2019

[7] *Feature detection (computer vision)*. https://en.wikipedia.org/wiki/Feature_detection_(computer_vision). – Abgerufen am 27.09.2019

[8] *Google AR-Designguidelines*. https://designguidelines.withgoogle.com/ar-design/augmented-reality-design-guidelines/introduction.html. – Abgerufen am 28.08.2019

[9] *Google Glass*. https://www.google.com/glass/start/. – Abgerufen am 08.09.2019

[10] *Google LLC. ARCore*. https://play.google.com/store/apps/details?id=com.google.ar.core&hl=de. – Abgerufen am 04.05.2019

[11] *Google LLC. Google developers*. https://developers.google.com/ar/. – Abgerufen am 28.08.2019

[12] *How we used Augmented Reality to bring Museums to life*. https://www.queppelin.com/2018/05/how-we-used-augmented-reality-to-bring-museums-to-life/. – Abgerufen am 07.09.2019

[13] *Is augmented reality (AR) the next level of gaming? - An Ericsson ConsumerLab insight report*. https://www.ericsson.com/4932a7/assets/local/trends-and-insights/consumer-insights/reports/gaming_report_cl_screen_aw.pdf. – Abgerufen am 09.09.2019

[14] *Just a Line by Google Creative Lab*. https://justaline.withgoogle.com/. – Abgerufen am 30.08.2019

[15] *The Legend of Zelda: Ocarina of Time*. https://de.wikipedia.org/wiki/The_Legend_of_Zelda:_Ocarina_of_Time. – Abgerufen am 21.09.2019

[16] *Microsoft HoloLens 2*. https://www.microsoft.com/de-de/hololens. – Abgerufen am 07.09.2019

[17] *Mobile AR games have been decades in the making but they're still waiting for a breakthrough*. https://blog.applovin.com/mobile-ar-games-history-future/. – Abgerufen am 09.09.2019

[18] *Pokemon GO von Niantic*. https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go. – Abgerufen am 01.09.2019

[19] *Project Tango*. https://en.wikipedia.org/wiki/Tango_(platform). – Abgerufen am 29.08.2019

[20] *SLAM, Visual Odometry, Structure from Motion, Multiple View Stereo*. https://www.slideshare.net/yuhuang/visual-slam-structure-from-motion-multiple-view-stereo. – Abgerufen am 27.09.2019

[21] *Statista - Statistik-Lexikon: Definition Likert-Skala*. https://de.statista.com/statistik/lexikon/definition/82/likert_skala/. – Abgerufen am 23.09.2019

[22] *streem by Streem Inc.* https://www.streem.pro/. – Abgerufen am 30.08.2019

[23] *System and Method for Concurrent Odometry and Mapping*. https://patents.google.com/patent/US20170336511A1/en. – Abgerufen am 29.08.2019

[24] *Tendar by Tender Claws*. http://tenderclaws.info/sheet.php?p=tendar. – Abgerufen am 09.09.2019

[25] *Unity Technologies. Unity3d*. https://unity3d.com/de/unity. – Abgerufen am 04.05.2019

[26] *Verleiht Mobile der erweiterten Realität (AR) Flügel?* https://www.silicon.de/blog/verleiht-mobile-der-erweiterten-realitaet-ar-fluegel. – Abgerufen am 26.09.2019

[27] *Vuforia Engine - Developer Portal*. https://developer.vuforia.com/. – Abgerufen am 10.09.2019

[28] *What's new in ARCore*. https://developers.google.com/ar/whatsnew-arcore. – Abgerufen am 11.09.2019

[29] AZUMA, Ronald: A Survey of Augmented Reality. In: *Presence: Teleoperators and Virtual Environments.* 6, Bd. 4 (1997), S. 355–385

[30] BAILEY, Tim ; DURRANT-WHYTE, Hugh: Simultaneous Localisation and Mapping (SLAM): Part II State of the Art. In: *IEEE Robotics Automation Magazine* Bd. 13 (2006), Nr. 3, S. 108–117

[31] BAY, Herbert ; TUYTELAARS, Tinne ; VAN GOOL, Luc: SURF: Speeded Up Robust Features. In: *Computer Vision-ECCV 2006*, Springer, Berlin Heidelberg, 2006, S. 404–417

[32] BILLINGHURST, Mark ; KATO, Hirokazu ; MYOJIN, Seiko: Advanced Interaction Techniques for Augmented Reality Applications. In: *Virtual and Mixed Reality. VMR 2009. Lecture Notes in Computer Science* Bd. 5622, Springer-Verlag Berlin Heidelberg, 2009, S. 13–22

[33] CHATZOPOULOS, Dimitris ; BERMEJO, Carlos ; HUANG, Zhanpeng ; HUI, Pan: Mobile Augmented Reality Survey: From Where We Are to Where We Go. In: *IEEE Access* Bd. 5 (2017), S. 6917–6950

[34] CHEHIMI, Fadi ; COULTON, Paul: Motion Controlled Mobile 3D Multiplayer Gaming. In: *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, 2008, S. 267–270

[35] CORDEIRO, Diogo ; JESUS, Rui ; CORREIA, Nuno: ARZombie: A Mobile Augmented Reality Game with Multimodal Interaction. In: *Proceedings of the 2015 7th International Conference on Intelligent Technologies for Interactive Entertainment (INTETAIN)*, 2015, S. 22–31

[36] DÖRNER, Ralf ; BROLL, Wolfgang ; GRIMM, Paul ; JUNG, Bernhard: *Virtual und Augmented Reality (VR/AR). Grundlagen und Methoden der Virtuellen und Augmentierten Realität*. Springer-Verlag Berlin Heidelberg, 2013

[37] DURRANT-WHYTE, Hugh ; BAILEY, Tim: Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms. In: *IEEE Robotics Automation Magazine* Bd. 13 (2006), Nr. 2, S. 99–110

[38] FISCHLER, Martin A. ; BOLLES, Robert C.: Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. In: *Commun. ACM* Bd. 24 (1981), Nr. 6, S. 381–395

[39] HARVIAINEN, Tatu ; KORKALO, Otto ; WOODWARD, Charles: Camera-based interactions for augmented reality. In: *Proceedings of the International Conference on Advances in Computer Enterntainment Technology*, 2009, S. 307–310

[40] HENRYSSON, Anders: *Bringing Augmented Reality to Mobile Phones*, Linköping University, Visual Information Technology and Applications (VITA), The Institute of Technology, Diss., 2007. http://www.diva-portal.org/smash/get/diva2:16967/FULLTEXT01.pdf. – Abgerufen am 09.09.2019

[41] HERLING, Jan ; BROLL, Wolfgang: Markerless Tracking for Augmented Reality. In: *Handbook of Augmented Reality*, Springer, New York, 2011, S. 255–272

[42] HÜRST, Wolfgang ; VRIENS, Kevin: Mobile Augmented Reality Interaction via Finger Tracking in a Board Game Setting. In: *Proceedings of Mobile HCI 2013, AR-workshop "Designing Mobile Augmented Reality"*, 2013

[43] HÜRST, Wolfgang ; WEZEL, Casper: Gesture-based interaction via finger tracking for mobile augmented reality. In: *Multimedia Tools and Applications* 62 (2013), S. 233–258

[44] KÖHLER, Johannes ; PAGANI, Alain ; STRICKER, Didier: *Detection and Identification Techniques for Markers Used in Computer Vision*. 2010 http://drops.dagstuhl.de/opus/volltexte/2011/3095/pdf/6.pdf. – Abgerufen am 02.09.2019

[45] LI, Mingyang ; MOURIKIS, Anastasios: Vision-aided inertial navigation for resource-constrained systems. In: *Intelligent Robots and Systems (IROS)* (2012 IEEE/RSJ International Conference on Intelligent Robots and Systems), S. 1057–1063

[46] LOWE, David G.: Object Recognition from Local Scale-Invariant Features. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision* Bd. 2 (1999), S. 1150–115

[47] LOWE, David G.: Distinctive Image Features from Scale-Invariant Keypoints. In: *International Journal of Computer Vision* Bd. 60 (2004), Nr. 2, S. 91–110

[48] MEHLER-BICHER, Anett ; STEIGER, Lothar: *Augmented Reality: Theorie und Praxis*. 2. Auflage. Oldenbourg Wissenschaftsverlag GmbH, 2014

[49] MILGRAM, Paul ; TAKEMURA, Haruo ; UTSUMI, Akira ; KISHINO, Fumio: Augmented Reality: A class of displays on the reality-virtuality

continuum. In: *Telemanipulator and Telepresence Technologies* Bd. 2351 (1994), S. 282–292

[50] MÜLLER, Stefan: *Vorlesungsfolien zur Vorlesung "Virtuelle Realität und Augmented Reality"*. Universität Koblenz, 2019 https://www.uni-koblenz-landau.de/de/koblenz/fb4/icv/agmueller/lehre/ws1819/arvr. – Abgerufen am 14.03.2019

[51] NISTÉR, David ; NARODITSKY, Oleg ; BERGEN, James: Visual odometry. In: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004, S. 652–659

[52] ODA, Ohan ; LISTER, Levi J. ; WHITE, Sean ; FEINER, Steven: Developing an Augmented Reality Racing Game. In: *Proceedings of the 2nd International Conference on Intelligent Technologies for Interactive Entertainment*, ICST, 2008

[53] PICKLUM, Mareike ; MODZELEWSKI, Georg ; KNOOP, Susanne ; LICHTENBERG, Toke ; DITTMANN, Philipp ; BÖHME, Tammo ; FEHN, Volker ; JOHN, Christian ; KENKEL, Johannes ; KRIETER, Philipp ; NIETHEN, Patrick ; PAMPUCH, Nicole ; SCHNELLE, Marcel ; SCHWARTE, Yvonne ; STARK, Sanja ; STEENBERGEN, Alexander ; STEHR, Malte ; WIELENBERG, Henning ; YILDIRIM, Merve ; YÜZÜNCÜ, Can ; POLLMANN, Frederic ; WENIG, Dirk ; MALAKA, Rainer: Player Control in a Real-Time Mobile Augmented Reality Game. In: *Entertainment Computing - ICEC 2012. Lecture Notes in Computer Science* Bd. 7522, Springer-Verlag Berlin Heidelberg, 2012, S. 393–396

[54] PIEKARSKI, Wayne ; THOMAS, Bruce: Tinmith-Hand: Unified User Interface Technology for Mobile Outdoor Augmented Reality and Indoor Virtual Reality. In: *Proceedings of the IEEE Virtual Reality Conference* (2002), S. 287–288

[55] POLLMANN, Frederic ; WENIG, Dirk ; PICKLUM, Mareike ; MALAKA, Rainer: Evaluation of Interaction Methods for a Real-Time Augmented Reality Game. In: *Entertainment Computing – ICEC 2013. Lecture Notes in Computer Science* Bd. 8215, Springer-Verlag Berlin Heidelberg, 2013, S. 120–125

[56] QUAN, Long ; LAN, Zhongdan: Linear N-point camera pose determination. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* Bd. 21 (1999), Nr. 8, S. 774–780

[57] ROLLAND, Jannick ; BAILLOT, Yohan ; GOON, Alexei: *A survey of tracking technology for virtual environments*. 2001 https://www.researchgate.net/publication/242415577_A_survey_

of_tracking_technology_for_virtual_environments. – Abgerufen am 01.09.2019

[58] SCHMALSTIEG, Dieter ; HÖLLERER, Tobias: *Augmented Reality: Principles and Practice*. Addison-Wesley, 2016

[59] SCHÜRG, Tobias: *Development and Evaluation of Interaction Concepts for Mobile Augmented and Virtual Reality Applications Considering External Controllers*. https://www.researchgate.net/publication/283944830_Development_and_Evaluation_of_Interaction_Concepts_for_Mobile_Augmented_and_Virtual_Reality_Applications_Considering_External_Controllers. Version: 2015. – Abgerufen am 09.09.2019

[60] SONG, Jie ; SÖRÖS, Gábor ; PECE, Fabrizio ; FANELLO, Sean ; IZADI, Shahram ; KESKIN, Cem ; HILLIGES, Otmar: In-air Gestures Around Unmodified Mobile Devices. In: *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST)*, ACM, 2014, S. 319–329

[61] SZELISKI, Richard: *Computer Vision : Algorithms and Applications*. Springer, 2010

[62] TAKETOMI, Takafumi ; UCHIYAMA, Hideaki ; IKEDA, Sei: Visual SLAM algorithms: a survey from 2010 to 2016. In: *IPSJ Transactions on Computer Vision and Applications* 9 (2017)

[63] ULLMER, Brygg ; ISHII, Hiroshi: Emerging frameworks for tangible user interfaces. In: *IBM Systems Journal* 39 (3-4) (2000), S. 915 – 931

[64] YOU, Suya ; NEUMANN, Ulrich ; AZUMA, Ronald: Hybrid Inertial and Vision Tracking for Augmented Reality Registration. In: *Proceedings of the IEEE Virtual Reality*, IEEE Computer Society, 1999, S. 260–267

[65] ZHAI, Shumin: User Performance in Relation to 3D Input Device Design. In: *ACM Siggraph Computer Graphics* Bd. 32 (1998), Nr. 4, S. 50–54

A Anhang

A.1 Fragebogen der Nutzungsstudie

Fragebogen zu pARcours

Anmerkung zur Beantwortung des Fragebogens

Bitte lesen Sie sich die Fragen sorgfältig durch und versuchen Sie, diese vollständig und wahrheitsgemäß zu beantworten. Benutzen Sie dafür die leeren Felder und Linien, bzw. orientieren Sie sich an der vorgegebenen Bewertungsskala und setzen Sie ein Kreuz an die entsprechende Stelle.

Bewertungsskala	
stimme gar nicht zu	--
stimme eher nicht zu	-
stimme eher zu	+
stimme voll zu	++

1) ARCore, Tracking und Performanz

	--	-	+	++
Horizontale reale Oberflächen wurden schnell erkannt.				
Vertikale reale Oberflächen wurden schnell erkannt.				
Die von ARCore erkannten Oberflächen stimmen (in Form und Größe) weitestgehend mit den realen überein.				
Die Imagemarker wurden schnell erkannt.				
Die Imagemarker wurden zuverlässig erkannt (d.h. keine „falsch erkannten“ Imagemarker).				
Das Tracking von Oberflächen, virtuellen Objekten und Imagemarkern war auch über einen längeren Zeitraum nach Spielstart und Entfernung vom „Startpunkt“ noch stabil (Objekte bleiben an ihrer Position).				
Es kommt kaum zu Ausfällen beim Tracking von realen Oberflächen.				
Es kommt kaum zu Ausfällen beim Tracking von virtuellen Objekten.				
Es kommt kaum zu Ausfällen beim Tracking von Imagemarkern.				
Das Spiel lief flüssig und ohne Ruckler.				
Die Möglichkeit, die getrackten Oberflächen visualisieren zu können, finde ich hilfreich.				
Der Schatten, der auf getrackten Oberflächen dargestellt wird, hilft dabei, die Position von virtuellen Objekten im Raum besser einschätzen zu können.				
Die Verdeckung von virtuellen Objekten durch getrackte Oberflächen (z. B. wenn sich das Flugobjekt hinter einer getrackten Wand befindet) hilft dabei, die Position im Raum besser einschätzen zu können.				

2) Aufbau der Level

	--	-	+	++
Das Platzieren von virtuellen Objekten durch Imagemarker finde ich intuitiv.				
Es hat gut funktioniert, virtuelle Objekte mit Hilfe von Imagemarkern zu platzieren und zu manipulieren.				
Das Platzieren von virtuellen Objekten mittels Touch-Input finde ich intuitiv.				
Es hat gut funktioniert, virtuelle Objekte mittels Touch-Input zu platzieren und zu manipulieren.				
Die Möglichkeit der Selbstgestaltung der Spielwelt/Level erhöht den Spaßfaktor und die Motivation.				

Welche Variante zur Selbstgestaltung der Level würden Sie bevorzugen?

☐ Imagemarker ☐ Touch-Input ☐ Mischung von Beiden

Begründung:

_____

_____

_____

_____

3) Steuerung der Flugobjekte

	--	-	+	++
Die Steuerung mit virtuellen Joysticks ist schnell zu erlernen.				
Die Steuerung mit virtuellen Joysticks ist selbsterklärend.				
Die Steuerung mit virtuellen Joysticks finde ich gut/hat mir Spaß gemacht.				
Die Steuerung mittels Touch-Input ist schnell zu erlernen.				
Die Steuerung mittels Touch-Input ist selbsterklärend.				
Die Steuerung mittels Touch-Input finde ich gut/hat mir Spaß gemacht.				
Die Steuerung mit Hilfe der Kamera ist schnell zu erlernen.				
Die Steuerung mit Hilfe der Kamera ist selbsterklärend.				
Die Steuerung mit Hilfe der Kamera finde ich gut/hat mir Spaß gemacht.				
Die Steuerung mittels physischen Gamepads ist schnell zu erlernen.				
Die Steuerung mittels physischen Gamepads ist selbsterklärend.				
Die Steuerung mittels physischen Gamepads finde ich gut/hat mir Spaß gemacht.				
Die Steuerung mittels physischen Gamepads erhöht das Gefühl, man würde echte Flugobjekte steuern.				
Mir gefällt die Möglichkeit, zwischen verschiedenen Arten der Steuerung wählen zu können.				
Mir gefällt es, die Wahl zwischen mehreren Flugobjekten zu haben, die auf unterschiedliche Art auf die Steuerung reagieren/sich unterschiedlich steuern lassen.				

Welche Variante zur Steuerung der Flugobjekte würden Sie bevorzugen?

☐ Virtuelle Joysticks ☐ Touch-Input ☐ Steuerung mit Kamera ☐ Gamepad

Begründung (Was war bei welcher Steuerung gut/schlecht?):

_____

_____

_____

_____

_____

4) Das Spiel

	--	-	+	++
Die Spielidee in Verbindung mit AR hat mir gefallen.				
Die Nutzung von AR für ein Spiel dieser Art ist gewinnbringend (wertet es auf bzw. macht es interessanter).				
Das Spielprinzip ist leicht verständlich bzw. schnell zu erlernen.				
Das Spiel ist ansprechend gestaltet.				
Das Spiel hat mir Spaß gemacht.				
Das Spiel hat mich gefordert.				
Das Spiel hat mich überfordert.				

Allgemein

1) Das hat mir besonders gut gefallen:

_____

_____

_____

_____

2) Das hat mir nicht so gut gefallen:

_____

_____

_____

_____

3) Verbesserungsvorschläge/Anregungen?

_____

_____

_____

_____

Personenbezogene Fragen

1) Wie alt sind Sie? ☐

2) Wie schätzen Sie ihre Spielerfahrung mit Computerspielen/Mobile Games ein?

☐ gar keine ☐ gering ☐ mittel ☐ hoch

Vielen Dank für die Teilnahme!

A.2 Auswertung der Nutzungsstudie

Auswertung des Fragebogens

Auswertungsskala

stimme gar nicht zu	stimme eher nicht zu	stimme eher zu	stimme voll zu
-- / -2	- / -1	+ / +1	++ / +2

Aussagenblock 1 (ARCore, Tracking und Performanz)

	-- / -2	- / -1	+ / +1	++ / +2	Ø	
A 1.1	-	-	5	8	1,615	1,154
A 1.2	1	2	7	3	0,692	
A 1.3	-	3	7	3	0,769	
A 1.4	2	2	7	2	0,385	-
A 1.5	-	-	-	13	2,0	
A 1.6	-	-	1	12	1,923	
A 1.7	1	-	3	9	1,462	
A 1.8	1	-	1	11	1,615	1,487
A 1.9	1	1	1	10	1,385	
A 1.10	-	1	1	11	1,692	
A 1.11	-	1	1	11	1,692	-
A 1.12	-	1	9	3	1,077	
A 1.13	-	2	6	5	1,077	
					1,337	

Aussagenblock 2 (Aufbau der Level)

	-- / -2	- / -1	+ / +1	++ / +2	Ø	
A 2.1	-	2	4	7	1,231	1,193
A 2.2	-	3	2	8	1,154	
A 2.3	-	-	1	12	1,923	1,654
A 2.4	-	1	5	7	1,385	
A 2.5	1	-	3	9	1,462	-
					1,431	

2) Aufbau der Level - Bevorzugte Eingabemethode und Begründung

- Imagemarker (1)
    - Für die Touch-Input Variante halte ich einen größeren Bildschirm (z. B. Tablet) für sinnvoll. Ansonsten geht das Erstellen der Level mit den Markern wesentlich schneller im Vergleich zu den Auswahl-Menüs (vielleicht noch anpassbar). Das Scannen könnte allerdings noch präziser/schneller sein.
- Touch-Input (8)
    - Einfachere Handhabung, schneller
    - System hat Imagemarker nicht immer erkannt
    - Einfache Handhabung, z. B. draußen
    - Faulheit (bezogen auf die zum Platzieren der Marker nötige Bewegung)
    - Ich würde den Touch-Input bevorzugen, da sich mit ihm die Level schneller bauen lassen
    - Geht etwas schneller und zuverlässiger
    - Weil es bequemer ist
    - Marker sind hakelig und nervig
    - Kaum Bewegung notwendig, schnell; Markererkennung nicht notwendig (Fehlerquelle und lahm); Skalierung und Rotation möglich
- Mischung von Beiden (4)
    - Beide Varianten intuitiv
    - Wenn man mehr Marker hat, erspart man sich beim Aufbau des Levels mehr Arbeit, aber beim Umplatzieren von Markern muss man diese erneut tracken. Wäre cool, wenn man die nachträglich per Touch manipulieren könnte.
    - Die Situation ist ausschlaggebend. Für Kinder finde ich den Imagemarker gut, für den schnellen Spielspaß das Touch-Input.
    - Beide Varianten haben Spaß gemacht. Die Auswahl würde ich abhängig von der Umgebung vornehmen. In einer kleineren Umgebung wäre die Variante der Imagemarker unpraktisch, wohingegen bei großen Räumen man sich direkt den Parcours strukturieren kann und drauf los scannen kann.

Aussagenblock 3 (Steuerung der Flugobjekte)

	-- / -2	- / -1	+ / +1	++ / +2	Ø	
A 3.1	-	1	6	6	1,308	1,18
A 3.2	-	4	-	9	1,077	
A 3.3	-	2	5	6	1,154	
A 3.4	1	1	8	3	0,846	0,615
A 3.5	-	6	3	4	0,385	
A 3.6	-	5	3	5	0,615	
A 3.7	1	7	4	1	-0,231	-0,487
A 3.8	5	6	2	-	-1,077	
A 3.9	1	7	3	2	-0,154	
A 3.10	-	-	2	11	1,846	1,846
A 3.11	-	-	3	10	1,769	
A 3.12	-	-	1	12	1,923	
A 3.13	1	-	8	4	1,077	-
A 3.14	-	2	1	10	1,462	
A 3.15	-	1	1	11	1,692	
					0,913	

3) Steuerung der Flugobjekte - Bevorzugte Eingabemethode (mehrere möglich) und Begründung

- Virtuelle Joysticks (4)
    - + Virtuelle Joysticks verhalten sich genau wie physische Joysticks!
    - + Bei virtuellen Joysticks braucht man keine extra Hardware
    - + Der virtuelle Joystick war angenehm zu bedienen und ist die Art Steuerung, die mir bei einem Handyspiel gefällt
    - + Man kann schnell einschätzen, wie man sich mit welchem Input im Raum bewegen kann
    - + Die Steuerung war wie in anderen Spielen gewohnt
    - + Hat präzise reagiert
    - - Probleme bei Orientierung im Raum
- Touch-Input (1)
    - + Bei Touch-Input kann man direkt einen Zeitpunkt für die Bewegung festlegen
    - + Einfache Bedienung
- Steuerung mit Kamera (0)
    - - Zu unpräzise
    - - Kamera-Steuerung ungewohnt und das „Steuerkreuz“ war manchmal schwer zu erkennen bei den Objekten im Raum
    - - Die Steuerung mit Kamera war etwas gewöhnungsbedürftig, gerade auch, weil ich diese Art der Steuerung zuvor nicht kannte
    - - Die Steuerung mit der Kamera könnte alternativ vielleicht besser funktionieren mit einem Zeiger/Marker, den man selbst bewegt, wohin das Flugobjekt fliegen soll
- Gamepad (11)
    - + Gewohnte Steuerung
    - + Gamepad-Steuerung verläuft erwartungskonform
    - + Kam mit Gamepad am besten zurecht, habe gerne etwas in der Hand
    - + Man kann besser einschätzen, wie man sich bewegt
    - + Man kann schnell einschätzen, wie man sich mit welchem Input im Raum bewegen kann
    - + Die Steuerung war wie in anderen Spielen gewohnt
    - + Hat präzise reagiert
    - + Das Gamepad-Feeling war noch ein Ticken cooler
    - + Gamepad ist für einen Anfänger leichter zu bedienen
    - + Gamepad war bezüglich der Steuerung am angenehmsten, vermutlich auch wegen der Gewohnheit
    - + Gamepad ist physisch
    - + Kaum Bewegung notwendig, beide Hände gleichzeitig
    - + Bildschirm ist nicht verdeckt
    - + Explizitere Anweisung als bei Touch-Input und Kamera
    - + Gamepad war am einfachsten/gewohntesten, …
    - - …aber die meisten Leute besitzen keins

Aussagenblock 4 (Das Spiel)

	-- / -2	- / -1	+ / +1	++ / +2	Ø	
A 4.1	-	-	2	11	1,846	1,885
A 4.2	-	-	1	12	1,923	
A 4.3	-	4	3	6	0,846	
A 4.4	-	1	7	5	1,231	
A 4.5	-	1	4	8	1,462	-
A 4.6	-	2	5	6	1,154	
A 4.7	2	5	5	1	-0,154	
					-	

Allgemein

1) Das hat den Testpersonen besonders gut gefallen:

- Diverse Flugobjekte mit unterschiedlichen Flugeigenschaften
- AR Idee
- Kombi Hindernisse; Checkpoints in Varianten
- Juwelen mit unterschiedlichen Eigenschaften
- Verschiedene Flugobjekte, vor allem Navi :)
- Alle Flugobjekte haben verschiedenes Steuerungsverhalten (vor allem Papierflugzeug)
- Eine große Auswahl an Objekten, die für den Parcours getestet werden können
- Ansprechendes UI-Design
- Meine erste Erfahrung im AR-Bereich
- Spielprinzip leicht erlernbar
- Flexible Möglichkeiten
- Akustische Effekte
- Mir hat besonders gut gefallen, dass es bei der Bewegung durch den Raum oftmals das Gefühl gab, man würde in der Realität mit den Hindernissen zusammenstoßen können
- Die eigene Umgebung als Spielwelt zu haben
- Die Idee, ein Flugspiel mit AR zu verbinden
- Verschiedene Flugobjekte
- Viele Hindernisse und Checkpoints
- Gute Steuerung
- Die Flugobjekt-Auswahl
- Kann man im Freien spielen
- Die Spielidee fand ich genial
- Ich finde es gut, dass die Flugobjekte Auswirkungen auf die Steuerungen haben und dass man sich mit unterschiedlichen Objekten eine kleine Spielwelt selbst bauen kann, dadurch erhält man keine eintönige Routine
- Das Spiel erlaubt eine flexible Gestaltung des Schwierigkeitsgrades und der Level
- Gute Verdeckung, Tracking
- AR ist lustiges Gimmick
- Stabiles Tracking
- Passende 3D-Modelle

2) Das hat den Testpersonen nicht so gut gefallen:

- Bei Änderung der Modi muss das Level neu gestaltet werden
- Spiel erfordert gute räumliche Vorstellung, die mir leider fehlt
- Der Verlust des Papierfliegers aufgrund seiner beschränkten Lenkfähigkeit
- Dass man nicht verlieren kann
- Es dauert etwas, das Spielprinzip/Steuerung ohne Erklärung zu verstehen
- Für mich zu schwierig/gehe lieber in den Wald
- Dass beim Marker Tracken Gegenstände ineinander liegen und das Level neu gestaltet werden muss
- Die Hitboxen waren an einigen Stellen schwer einzuschätzen
- Bugs, Steuerung, Marker erkennen
- Skalierungsprobleme und kleine Bugs
- Menü etwas umständlich

3) Verbesserungsvorschläge/Anregungen:

- Speichern des Levels, damit man bei Modus-/Steuerungs-Änderung dieses nicht neu aufbauen muss
- Die Drohne und das Papierflugzeug haben Probleme, durch die Checkpoints zu fliegen
- Das Spaceship sollte im Allgemeinen etwas schneller sein
- Möglichkeit, erstellte Parcours zu speichern, um sie jederzeit wiederzuverwenden oder ggf. Freunden zu schicken
- Anleitung per Tutorial
- Bei Kollision optischer Effekt
- Markierung auf den Imagemarkern, welche zeigen, mit welcher Orientierung das Objekt im Raum platziert wird (es ist schwer, durch den Checkpoint zu fliegen, wenn man diesen z. B. vor einer Wand platziert hat)
- Sich bewegende Hindernisse/Feinde
- Lebenspunkte
- Tutorial, welches das Spielprinzip und die Steuerung kurz erklärt
- Level speichern können, zumindest um andere Steuerung wählen zu können
- Anleitung auf Deutsch
- Eventuell Fehlermeldung beim Imagemarker, wenn Objekte ineinander stehen
- Für die Variante auf einem Tisch zu spielen würde ich mir wünschen, dass die Objekte in etwa der Markergröße entsprechen zur besseren Vorstellung bei der Levelgestaltung

Personenbezogene Fragen

1) Alter der Testpersonen:

- Summiert: 47 + 22 + 57 + 58 + 25 + 29 + 20 + 21 + 49 + 30 + 21 + 22 + 23 = 424
- Durchschnitt: 424/13 = 32,615

2) Spielerfahrung mit Computerspielen/Mobile Games:

- gar keine (3)
- gering (0)
- mittel (0)
- hoch (10)