Universität Augsburg

Fakultät für Angewandte Informatik

Lehrstuhl für Menschzentrierte Künstliche Intelligenz

Masterarbeit:

Design und Implementierung einer Virtual-Reality-Umgebung zum Testen des impliziten Lernens von motorischen sequentiellen Bewegungen

von

Nikolai Glaab

Semester: Sommersemester 2021
Adresse: Pfalzstraße 37, 86343 Königsbrunn
Master Informatik, 9-tes Fachsemester
Matrikelnummer: 1301069
Betreuer: Tobias Huber
Erstprüfer: Prof. Dr. Elisabeth André

Inhaltsverzeichnis

1 Einleitung

Wie misst man das implizite Lernen von motorischen, sequentiellen Bewegungen? Für das Messen des impliziten Lernens von Sequenzen ohne motorische Bewegungen gibt es zahlreiche Beispiele in der Literatur. In der seriellen Reaktionszeit Aufgabe müssen Probanden auf visuelle Reize (*Stimuli*), die auf einem Bildschirm erscheinen, mit dem Drücken der zugehörigen Taste reagieren. In dem ursprünglichen Experiment von Nissen & Bullemer [1] konnten die Stimuli auf einer von vier möglichen Positionen erscheinen, wobei für jede Position die zugehörige Taste gedrückt werden sollte. Das Drücken der Taste kann allerdings nicht als motorische Bewegung gezählt werden, es ist nur als Antwortaktion des Probanden auf den visuellen Stimulus zu bewerten. Die präsentierten Stimuli folgen einem sich wiederholenden Muster, einer *Sequenz*, welche üblicherweise acht bis zwölf Elemente enthält. Dabei wird der Lernerfolg über die benötigte Reaktionszeit des Drückens der richtigen Taste bestimmt. Insbesondere werden die Reaktionszeiten der sequenzierten Stimuli mit denen von Stimuli, die in zufälliger Reihenfolge erscheinen, verglichen. Bei erfolgreichem Lernen der Sequenz von Stimuli nehmen die Reaktionszeiten für die Sequenz, verglichen mit den zufällig angeordneten Stimuli ab. Das Abschneiden für solche Sequenzen verbessert sich auch bei Probanden, die sich der Sequenz nicht bewusst sind oder die Reihenfolge der Stimuli in der Sequenz nicht wiedergeben können. Dies lässt auf implizites Wissen zur Sequenz bei diesen Probanden schließen [vgl. 1, 2]. Implizites Wissen zeichnet sich dadurch aus, dass sich die Person des Wissens nicht bewusst ist und dieses auf Nachfrage oder Aufforderung nicht wiedergeben kann.

In den letzten Jahren hat die Verbreitung von Virtual Reality Headsets immer mehr zugenommen, sodass mittlerweile eine relativ große Anzahl an für VR ausgelegten Spielen von verschiedenen Spiele-Studios entwickelt wird. Levac et al. sehen virtuelle Umgebungen als geeignet an, um den Lernprozess und Transfer komplexer Fähigkeiten zu bewerten, ohne dabei auf experimentelle Kontrolle verzichten zu müssen. Insbesondere nutzen virtuelle Umgebungen modellierte Aufgaben des echten Lebens, bei denen experimentelle Beeinflussung der Probanden möglich ist, um gewünschte Verhaltensweisen der Probanden zu messen bzw. herbeizuführen. Die hierbei erreichte Präzision übersteigt die Möglichkeiten physischer Umgebungen[vgl. 3].
Am Sportzentrum der Universität Augsburg war bereits mit dem VR-Spiel „Beat Saber“ der Firma Beat Games das sequentielle Lernen von *beidhändigen* Schlagbewegungen mit verschiedenen Studien untersucht worden. Laut Swinnen und Wenderoth [vgl. 4] weist die Steuerung beidhändiger Bewegungen ein hohes Maß an Modularität auf. So ist eine Vielzahl an Bewegungsabläufen und komplexen Gesten möglich, bei denen die einzelnen Hände auch unterschiedlich stark einbezogen sein können. Die Zielbeständigkeit der aus-

führenden Bewegung wird durch eine flexible Kovariation der Bewegungen der einzelnen Gliedmaßen sichergestellt. Komplexe, beidhändige Fähigkeiten werden gerne als Beispiel zur Studie höherer kognitiver Fähigkeiten herangezogen. Außerdem stellen beidhändige Bewegungen einen Sonderfall des Multitasking dar, der Aufschluss darüber gibt, wie das Zentralnervensystem die Organisation mehrerer, gleichzeitiger Befehlsströme bewältigt. Laut mehreren Studien ist das Sequenzen Lernen beim Multitasking allerdings beeinträchtigt [vgl. 5, 6], was ein verschlechtertes Abschneiden beim gleichzeitigen Zerschlagen verglichen mit zeitlich versetzten Schlagen vermuten lässt.
Andererseits ist das Sequenzen Lernen von mehreren Aufgaben vor allem dann verschlechtert, wenn der Lernende die zwei verschiedenen Aufgaben nicht auseinanderhalten kann. Dies ist vor allem dann der Fall, wenn eine Aufgabe eine Sequenz von zufälligen Stimuli enthält, welche nicht zu der Sequenz der ersten Aufgabe passt [vgl. 7]. Da beim gleichzeitigen Zerschlagen dieselbe bzw. eine ähnliche Aufgabe zeitgleich ausgeführt werden muss, könnte diese Aufgabe gut von Probanden durchgeführt werden, die die Schlagbewegungen für die zwei Hände nicht verwechseln.
In Beat Saber zerschlägt der Spieler mit zwei Lichtschwertern im Rhythmus Kisten, die direkt auf ihn zufliegen, zu ausgewählten Songs. Je präziser die Kisten zerschlagen werden, desto mehr Punkte bekommt der Spieler für einen Schlag. Es lassen sich auch für jeden Song fünf verschiedene Schwierigkeitsgrade einstellen, welche die Anzahl der auftauchenden Kisten beeinflussen. Auf den Kisten sind Pfeilrichtungen angebracht, die beachtet werden müssen, weil sie die Schlagrichtung vorgeben. Die Anzahl der auf den Spieler zufliegenden Kisten/ Boxen variiert dabei je nach aktueller Stelle im Lied. In einer Vielzahl an Fällen muss der Spieler mit beiden Armen gleichzeitig Kisten zerschlagen. Es kann aber auch vorkommen, dass der Spieler eine Zeit lang Kisten nur mit einem Arm zerschlägt. Allerdings gibt es einige Argumente, die gegen Beat Saber als Studienumgebung sprechen. So kann durch das Zufliegen der Kisten auf den Spieler nicht die Reaktionszeit gemessen werden, da die Kisten im Rhythmus eines Liedes vor dem Spieler erscheinen und der Spieler nur eine kleine Zeitspanne hat, um die Kisten zu zerschlagen. Stattdessen muss die Präzision und Fehlerrate beim Zerschlagen der Kisten genutzt werden, um den Lernerfolg zu bewerten. Allerdings wird in den ursprünglichen Serielle-Reaktionszeit-Aufgabe-Experimenten die Reaktionszeit genutzt, um das implizite Lernen zu bewerten. Auch ist störend, dass in bestimmten Abschnitten von Songs visuelle Effekte auftauchen, welche das Bestimmen der als nächstes auftauchenden Kisten erschweren. Durch das Spieldesign ist man darauf festgelegt, den Spieler nur vor ihm auftauchende Kisten zerschlagen zu lassen. Es gibt zwar auch 360°-Level in Beatsaber, allerdings erscheinen die Kisten immer auf fixen, festgelegten Linien, wobei der Spieler sich nur wenig drehen muss, um die Kisten im Blickfeld zu haben.

In dieser Arbeit soll eine ähnliche VR-Anwendung entwickelt werden, welche mehr Kontrolle über den Ablauf des Spiels erlaubt. So sollen die Objekte zum Zerschlagen direkt vor dem Spieler erscheinen und immer nur zwei gleichzeitig von ihnen auftauchen, bevor die nächsten erscheinen. Dies hat den Vorteil, dass die Reaktionszeit ab dem Auftauchen eines Würfels bis zum erfolgreichen Zerschlagen des Würfels gemessen werden kann. Die gemessenen Reaktionszeiten können dann verwendet werden, um den impliziten Lerner-

folg des Spielers zu bewerten. Über das Bestimmen von Reaktionszeit und Präzision beim Zerschlagen werden die einzelnen Schläge mit einer Punktzahl bewertet. Je genauer und je präziser der Spieler einen Würfel zerschlagen hat, desto mehr Punkte erhält er für den gemachten Schlag. Von einem erscheinendem Würfelpaar ist stets ein Würfel für die rechte, und der andere für die linke Hand vorgesehen und kann auch nur mit der richtigen Hand zerschlagen werden. Es ist hierbei gewünscht, dass der Spieler die zwei Würfel parallel mit beiden Händen zerschlägt, da er nur auf diese Weise die höchste Punktzahl erreichen kann, indem die Reaktionszeit für das Zerschlagen beider Würfel minimiert wird. Die Würfel sind unterschiedlich gefärbt (rot und blau), um anzuzeigen, mit welchem farblich passenden Schwert der Würfel zerschlagen werden muss. Die VR-Anwendung ermöglicht es, dass Würfel im 360°-Radius um den Spieler erscheinen und bietet eine Reihe von verschiedenen Einstellungsmöglichkeiten (siehe Implementierungs-Kapitel), welche in Beat Saber nicht vorgenommen werden können. Zerschlägt der Spieler den Würfel nicht in der mit einem Pfeil gekennzeichneten Richtung, so gilt der Würfel als falsch durchschlagen und folglich erhält der Spieler 0 Punkte für diesen Schlag. Die ermittelten Daten eines Spieldurchlaufs bzw. des Studienlevels werden gespeichert, damit zu einem späteren Zeitpunkt eine Auswertung dieser Daten erfolgen kann. Der Fokus liegt darauf, dass der Spieler bestimmte Schlagbewegungen durch eine festgelegte, sich wiederholende Sequenz implizit lernt. Nach einer Trainingsphase, in welcher der Spieler die vorher festgelegte Sequenz übt, soll eine Verbesserung der Leistung des Spielers messbar sein. Es muss außerdem sichergestellt sein, dass das Lernen implizit geschieht.

Die entwickelte Anwendung kann als seriöses Spiel zu den *Exergames* gezählt werden, bei denen der Spieler während des Spielens körperlich aktiv werden muss. Hier ist der große Vorteil, dass das Spielen mit einer sportlichen Betätigung verbunden wird, was die Gesundheit des Spielers fördert. Die sportliche Betätigung muss hier allerdings mehr als das Drücken von Knöpfen eines Controllers ausmachen. Die vorliegende Arbeit beschäftigt sich sowohl mit sport-theoretischen Grundlagen zum Impliziten Lernen von Sequenzen als auch der technischen Implementierung einer VR-Anwendung, welche dazu genutzt werden kann, den Lernerfolg beim Ausüben von motorischen sequentiellen Bewegungen zu messen und speichern. Die VR-Anwendung wurde mittels der Unity Engine entwickelt. Im erstem Kapitel werden die Grundlagen für implizites Lernen behandelt und der Aufbau von bereits durchgeführten Studien zum Erforschen von implizitem Lernen beschrieben. Das zweite Kapitel beschäftigt sich mit den technischen Grundlagen für die Arbeit, hier wird vor allem die Funktionsweise einer Physik Engine beschrieben. Anschließend wird die Implementierung der VR-Anwendung ausführlich behandelt. Hier werden die wichtigsten Bestandteile der Anwendung nach Funktion gruppiert beschrieben. Zum Schluss wird eine selbst durchgeführte Studie erläutert und die mittels der Studie gesammelten Spieldaten in Bezug zur Forschungsfrage analysiert.

2 Grundlagen Implizites Lernen

2.1 Implizites Lernen einer Sequenz

Für dieses Kapitel Grundlagen von impliziten Lernen einer Sequenz halte ich mich größtenteils an Schwarb et al. [vgl. 6].

2.1.1 Definition Implizites Lernen

Implizites Lernen bezeichnet unbewusstes Lernen, bei dem sich der Lernende unbemerkt neue Verhaltensweisen aneignet. Dieses erfolgreich angelernte Wissen kann in zukünftigen Situationen von der Person angewendet werden, um besser und effizienter zu agieren [vgl. 6, S.1].
Als Beispiel für implizites Lernen kann das Sprechen-Lernen der Muttersprache angeführt werden, da man sie grammatikalisch korrekt sprechen kann, ohne die meisten grammatikalischen Regeln beschreiben zu können, welche einen korrekten Satz bilden. Ein weiteres Beispiel stellen die ersten Gehversuche im Kindesalter dar, da einem selbst (zu diesem Zeitpunkt) die genauen Bewegungsabläufe aller Muskelgruppen unbekannt sind, welche für die Fähigkeit zu gehen erforderlich sind. In beiden Beispielen kommt die menschliche Befähigung zum Tragen, sich auf Einschränkungen aus der Umwelt einstellen zu können, ohne selbst das Wissen zu besitzen, wie genau die Umsetzung der Befähigung geschieht [vgl. 8, S.1].
Frensch et al. [8, S.2] haben folgende Definition für implizites Lernen adaptiert, welche bereits von vielen anderen Forschern verwendet wurde: „Die Fähigkeit zu lernen, ohne Bewusstsein über die Produkte des Lernens zu besitzen“. Deswegen wird angenommen, dass implizites Lernen nur dann stattgefunden hat, wenn die Studienteilnehmer sich darüber unklar sind, was sie genau gelernt haben. Im Gegensatz dazu spricht man vom expliziten bzw. hypothesengetriebenen Lernen, wenn gerade nicht implizites Lernen vorliegt. Das ist dann der Fall, wenn sich die Studienteilnehmer darüber bewusst sind, was sie gelernt haben, also dann, wenn sie die Lernprodukte beschreiben können [vgl. 8, S.1f].

2.1.2 Serielle Reaktionszeit Aufgabe

Als Verfahren, um das implizite Lernen erforschen zu können, wurde 1987 von Nissen und Bullemer [vgl. 1] die serielle Reaktionszeit Aufgabe vorgestellt. Mithilfe dieses Verfahrens kann das räumliche Lernen von Sequenzen untersucht werden. Hierbei kann man zwei verschiedene Versuchsreihen unterschieden: Bei der einen haben die Studienteilnehmer nur die Aufgabe, auf eine Sequenz von Stimuli mit bestimmten Aktionen zu reagieren.

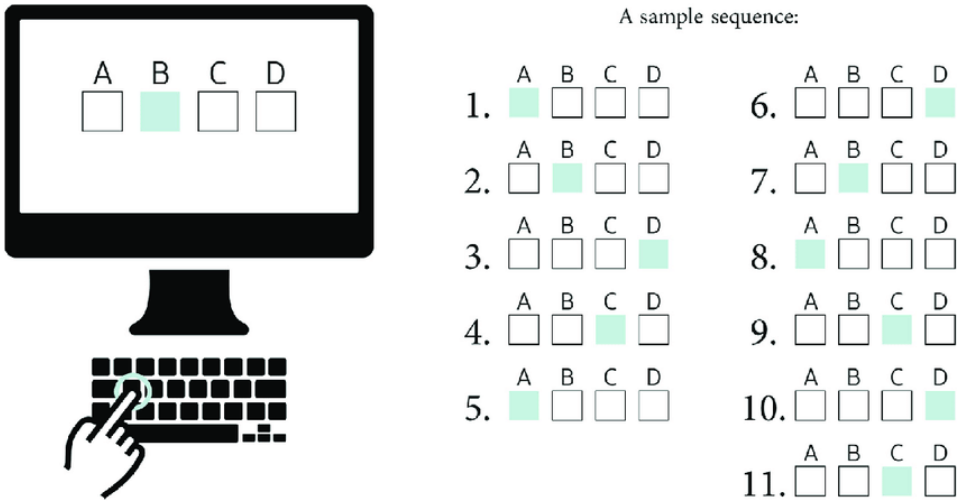


Abbildung 2.1: Versuchsaufbau für Serielle Reaktionszeit Aufgabe [9, S.7]

Bei der anderen müssen die Studienteilnehmer gleichzeitig noch eine andere Aufgabe bewältigen. Eine Sequenz besteht aus visuellen Stimuli, welche an fest vordefinierten Positionen auf einem Bildschirm erscheinen. Auf diese erscheinenden Stimuli muss der Lernende mit dem Drücken der zur Position zugehörigen Taste reagieren. In mehreren Blöcken wird aufgezeichnet, wie schnell und genau (z.B. werden auch falsche Tasten gedrückt?) ein Studienteilnehmer einer Gruppe die zur Sequenz passenden Aktionen durchführt (Tasten drücken). Dabei kann ein Block mehrere Wiederholungen derselben Sequenz umfassen, aber es sind auch Blöcke mit komplett zufälligen visuellen Stimuli möglich. Bei einem Block mit zufälligen Stimuli gilt allerdings die Einschränkung, dass auf die Position eines visuellen Stimulus nicht wieder dieselbe Position des visuellen Stimulus folgen kann.
Bei diesem ersten Experiment kann oft festgestellt werden, dass Personen der Gruppe mit einer sich wiederholenden Sequenz besser abschneiden als Personen einer zweiten Gruppe mit zufälligen visuellen Stimuli; d.h. sie erreichen genauere Eingaben und bessere bzw. kürzere Reaktionszeiten.
Bei dem zweiten Experiment müssen die Studienteilnehmer zusätzlich neben der Reaktion auf visuelle Stimuli auch noch Töne zählen. Dabei werden zwei verschiedene Töne verwendet, ein hoher Ton und ein tiefer Ton. Die Anzahl der tiefen Töne soll von den Teilnehmern gezählt werden und am Ende eines Blocks müssen sie die genaue Anzahl der vorgekommenen tiefen Töne angegeben. Eine Multitasking-Gruppe bezeichnet hier eine Gruppe, welche zwei Aufgaben parallel erledigen muss (Reaktionsaufgabe + Töne zählen). Sowohl die Personen der Multitasking-Gruppe mit zufälligen visuellen Stimuli als auch die Personen der anderen Multitasking-Gruppe mit in einer Sequenz ablaufenden visuellen Stimuli schnitten schlechter ab als die Gruppe mit nur einer Aufgabe (Reaktionsaufgabe auf visuelle Stimuli). Allerdings spielen Eigenschaften der verwendeten Sequenz durchaus eine größere Rolle, sodass implizites Lernen auch bei Multitasking stattfinden kann, wenn die Eigenschaften der Sequenz es ermöglichen. Die verschiedenen möglichen Formen einer Sequenz und wie sich diese auf das Lernen (insbesondere bei

Multitasking) auswirken, wird im nächsten Abschnitt beschrieben [vgl. 6, S.2].

2.1.3 Sequenzstruktur

Die Sequenzstruktur beschreibt den Aufbau einer Sequenz. Vereinfacht kann eine Sequenz durch aufeinanderfolgende Zahlen ausgedrückt werden, welche dann beispielsweise je nach Zahl angeben, an welcher Position der nächste visuelle Stimulus erscheint. Eine Sequenz kann *einzigartig*, *hybrid* und *mehrdeutig* sein.
Bei einer *einzigartigen* Sequenz hat jedes Element ein eindeutiges Nachfolge-Element (z.B. die Sequenz "3-4-1-7-2-6-8-5"). Bei einer *mehrdeutigen* Sequenz hat jedes Element mindestens zwei verschiedene Nachfolge-Elemente. Hier kann das Nachfolge-Element für ein Element in der Sequenz nur dann vorhergesagt werden, wenn mindestens zwei vorhergehende Elemente bekannt sind und in Betracht gezogen werden, (z.B. die Sequenz "1-3-4-2-1-2-4-3"Man beachte, dass hier auch das Element "3"durch das zyklische Wiederholen der Sequenz innerhalb eines Blocks zwei verschiedene Nachfolge-Elemente besitzt. Die Nachfolgeelemente sind in diesem Beispiel „4“ und „1“).

Eine *hybride* Sequenz ist eine Mischform zwischen *einzigartiger* und *mehrdeutiger* Sequenz. In dieser Sequenz gibt es sowohl Elemente, welche ein eindeutiges Nachfolge-Element besitzen, als auch Elemente, welche mindestens zwei verschiedene Nachfolge-Elemente aufweisen. (z.B. die Sequenz "4-2-4-1-3-6-2-5"). [vgl. 6, S.2f] Die Sequenzstruktur beeinflusst maßgeblich, wie leicht eine Sequenz von Probanden verinnerlicht / gelernt werden kann. So fanden Cohen, Ivry und Keele 1990 [vgl. 10] in einer Studie heraus, dass unter der Bedingung, dass die Probanden zwei Aufgaben gleichzeitig durchführen müssen, nur noch das Lernen von einzigartigen und hybriden Sequenzen beobachtet werden kann, wohingegen das Lernen von mehrdeutigen Sequenzen scheitert. An dieser Stelle muss allerdings auch angemerkt werden, dass das Ausmaß der Beeinträchtigung des Lernens einer Sequenz auch davon abhängt, was genau als zweite Aufgabe gewählt wird und welche Rahmenbedingungen für die zweite Aufgabe gelten. So ist die Beeinträchtigung des Lernens am geringsten, wenn die Stimuli für die zweite Aufgabe gleichzeitig mit den Stimuli der ersten Aufgabe (Reaktionsaufgabe auf visuelle Stimuli) präsentiert werden [vgl. 5, S.2].
Am einfachsten lassen sich *einzigartige* Sequenzen erlernen, weil hier nur die verschiedenen Assoziationen von zwei aufeinanderfolgenden Elementen innerhalb der Sequenz gelernt werden müssen (Eine Assoziation für ein Element in der Sequenz). Bei mehrdeutigen Sequenzen sind es schon mindestens zwei Assoziationen für ein Element in der Sequenz und es werden auch mehr Informationen (vorangehende Elemente innerhalb der Sequenz) benötigt, um die passende Assoziation für das aktuelle Element auswählen zu können [vgl. 5, S.2].
Pasquali et al. [vgl. 11, S.2] stellten einen neueren Ansatz für die Wahl einer Sequenz vor; sogenannte *bedingte Sequenzen zweiter Ordnung*. Bei diesen Sequenzen ergibt sich die nächste Zielposition (bzw. das nächste Element in der Sequenz) durch die zwei vorgehenden Zielpositionen (bzw. zwei vorhergehende Elemente in der Sequenz). D.h. eine Funktion berechnet nach bestimmten Regeln mithilfe zweier vorhergehender Elemente

innerhalb der Sequenz das nächste Element. Obwohl Studienteilnehmern nichts von den Gegebenheiten dieser Sequenz erzählt wird, stellen sie sich mit der Zeit darauf ein. Dies kann im Vergleich zu einer zufälligen Sequenz oder einer anderen Sequenz, die nicht mehr den vordefinierten Regeln der *bedingten Sequenz zweiter Ordnung* folgt, gut an verbesserten , kürzeren Reaktionszeiten beobachtet werden.
Der Vorteil dieser *bedingten Sequenzen zweiter Ordnung* liegt darin, das Vorkommen von ergänzenden Informationen zu kontrollieren und zu minimieren. Denn diese besonderen Gegebenheiten könnten zum Lernen der Sequenz ausgenutzt werden. Dies umfasst beispielsweise wie oft jede Zielposition in der Sequenz vorkommt, wie häufig Vor-und-Zurück-Bewegungen auftreten, wie lange es innerhalb der Sequenz dauert, bis jede Zielposition mindestens einmal aufgetreten ist, usw. Mit Hilfe der Verwendung von bedingten Sequenzen zweiter Ordnung kann also ausgeschlossen werden, dass die Probanden nur Häufigkeitsinformationen erlernen und mithilfe dieser bessere Reaktionszeiten erzielen, sondern die tatsächliche Sequenz(-struktur) implizit lernen [vgl. 6, S.3].

2.1.4 Maß für Lernerfolg

Um zu messen, ob Lernen einer Sequenz stattgefunden hat, wurde in den ersten Studien (von Nissen und Bullemer) [vgl. 1] die gemessenen Werte der Teilnehmer (Reaktionszeit) zwischen zwei Gruppen verglichen. In der einen Gruppe wurden die Studienteilnehmer einer Versuchsreihe mit einer festgelegten Sequenz, in der anderen Gruppe wurden die Studienteilnehmer einer Versuchsreihe mit zufalls-basierter Sequenz ausgesetzt. Mittlerweile [vgl. 10] hat man aber eine effizientere Methode gefunden, um das implizite Lernen auf eine Person bezogen zu messen. Dabei geht man so vor, dass der Proband zunächst mehrere Blöcke mit der zu lernenden Sequenz bekommt. Anschließend erhält der Proband einen Block mit einer alternativen Sequenz (typischerweise bedingte Sequenz zweiter Ordnung). Zum Schluss darf der Proband nochmal einen Block mit der zu lernenden Sequenz absolvieren. Falls implizites Lernen stattgefunden haben sollte, müssen die Reaktionszeiten für den Block mit der alternativen Sequenz deutlich schlechter ausfallen, als für die den Block umgebenden Blöcke (zuletzt kommenden Blöcke mit der zu lernenden Sequenz). Diese Beziehung von Reaktionszeiten, bekannt als **Transfer-Effekt**, ist mittlerweile die Standardmethode, um Sequenzen-Lernen bei der seriellen Reaktionszeit Aufgabe zu messen [vgl. 6, S.3f].

2.1.5 Unterscheidung von impliziten und expliziten Wissen

Bei impliziten Wissen handelt es sich um Kenntnisse, welche man nicht bewusst und nicht aktiv abrufen kann. Auf Explizites Wissen dagegen kann man jederzeit zugreifen und kann insbesondere Fragen zu dem explizitem Wissen leicht beantworten. Das Ziel bei der seriellen Reaktionszeit Aufgabe ist es, nur implizites Lernen zu messen ohne explizites Lernen zu berücksichtigen. Dennoch kann es bei Versuchen und Experimenten (in Form einer Studie) vorkommen, dass sich die Probanden explizites Wissen über die Sequenzen aneignen. Deshalb ist es gängige Praxis nach dem Test mit einem Probanden mithilfe von Fragebögen herauszufinden, wie viel explizites, bewusstes Wissen bei

einem einzelnen Probanden zur Sequenz vorliegt [vgl. 12]. In diesen Fragebögen werden Fragen gestellt wie z.B. ob man bemerkt hat, dass die Zielpositionen der erscheinenden, visuellen Stimuli einer sich wiederholenden Sequenz folgen. Eine andere oft gestellte Frage lässt den Probanden nach seinem besten Glauben raten, was das Ziel des Experiments ist. Eine andere Möglichkeit, explizites Wissen beim Probanden festzustellen, ist der Erzwungene-Wahl-Erkennungstest. Bei diesem wird mittels verschiedener auf die Sequenz bezogenen Fragen geprüft, ob die Probanden Teile der Sequenz wiedererkennen können. Auch Aufgaben für die freie Erzeugung der Sequenz sind möglich, einerseits gibt es den Inklusion-Test, andererseits den Exklusion-Test. Im Inklusion-Test müssen die Probanden die im Experiment verwendete Sequenz nachbilden. Im Exklusion-Test dagegen müssen die Probanden es vermeiden, Teile der im Experiment verwendeten Sequenz zu erzeugen. Beim Inklusion-Test werden Probanden mit explizitem Wissen zumindest Teile der Sequenz nachbilden können. Hingegen werden beim Exklusion-Test die Probanden, die die Sequenz nachbilden, unbewusst implizites Wissen über die Sequenz verwenden, obwohl sie gerade nicht diese Sequenz nachstellen sollen. Die Kombination aus Inklusion-Test und Exklusion-Test stellt eine gute Kombination dar, um die verschiedenen impliziten und expliziten Wissensanteile bei den Probanden zu bestimmen. [vgl. 6, S.3]

2.2 Funktionsweise des sequentiellen Lernens

Im folgendem Abschnitt sollen drei verschiedene Theorien vorgestellt werden, welche sich damit beschäftigen, in welchem kognitiven Verarbeitungsschritt die Sequenzen tatsächlich gelernt werden. So gibt es die *Stimulus-basierte Theorie*, die *Antwort-basierte Theorie* und die *Stimulus-Antwort-Regel-basierte Theorie*. Es wird angenommen, dass mindestens drei kognitive Verarbeitungsschritte beim Lernen einer Sequenz eine Rolle spielen. So muss der Studienteilnehmer zunächst den Stimulus enkodieren (Stimulus-Enkodieren-Prozess), dann eine passende, geeignete Antwortaktion als Reaktion auf den Stimulus auswählen (Antwort-Auswählen-Prozess) und schließlich muss er die ausgewählte Antwortaktion auch noch ausführen (Antwort-Ausführen-Prozess). Es ist möglich, dass das Lernen während einer dieser informationsverarbeitenden Phasen passiert, aber genauso gut ist es denkbar, dass das Lernen auf mehrere dieser Phasen aufgeteilt ist. Die verschiedenen aufgestellten Theorien befassen sich genau damit, in welcher dieser Phasen das Lernen geschieht. [vgl. 6, S.4] Was und wie genau wird bei der seriellen Reaktionszeit Aufgabe gelernt? Im Folgenden sollen die verschieden Theorien vorgestellt werden, um diese Frage zu beantworten.

2.2.1 Stimulus-basierte Theorie

Die *Stimulus-basierte Theorie* des Sequenzen Lernens geht davon aus, dass die Sequenz über eine Aufstellung von Stimulus Assoziationen gelernt wird. Die Art der Antwort auf den Stimulus ist aber bereits nicht mehr von Bedeutung; auch dann, wenn überhaupt keine Antwortaktion gemacht wird. Außerdem besagt sie, dass das Lernen „reiz-spezifisch,

effektor-unabhängig, nicht-motorisch und rein durch die Wahrnehmung geschieht“ [6, S.4]. Ein Effektor bezeichnet „eine Zelle oder einen Zellverband mit dem Vermögen, auf einen neuronalen Impuls hin einen Effekt hervorzubringen“ [13, S.313, übernommen von 14]. Eine Muskelzelle erreicht als Effektor den gewünschten, herbeizuführenden Bewegungseffekt durch Kontraktion. In einer Studie mit paralleler Tonzählaufgabe von A. Cohen et al. [vgl. 10] wurde 1990 untersucht, inwiefern sich das Tauschen der Aktionen durchführenden Effektoren auswirkt. Zunächst nutzten die Probanden vier Finger der rechten Hand, um auf die visuellen Reize zu antworten. Anschließend durften sie nur noch einen Finger zum Antworten verwenden, was allerdings das Lernen nicht beeinträchtigte. Daraus wurde geschlossen, dass Wissen über die Sequenz nur von den dargestellten Stimuli abhängt und auch vom eingesetzten Effektor-System zum Lernen der Sequenz unabhängig ist. Daraufhin wurde diese Theorie in verschiedenen Studien und Experimenten untersucht und es wurde mehrere Thesen gegen diese Theorie aufgestellt. Einerseits könnte explizites Wissen die Ergebnisse so verfälschen, dass es nur so aussieht, als ob das Sequenzen Lernen für die Gruppe ohne Antwort-Reaktion genauso gut ausfällt wie für die Gruppe mit Antwort-Reaktion. Andererseits ist es möglich, dass die Stimuli räumlich so weit voneinander entfernt sind, dass motorische Bewegungen der Augen notwendig sind, um die Positionen der Stimuli visuell erfassen zu können. Das wiederum ermöglicht Stimulus-Antwort Assoziationen zwischen den Stimuli und den Bewegungen der Augen, sodass nicht ausgeschlossen werden kann, dass bereits die Bewegungen der Augen zu den verschiedenen Positionen der Stimuli eine Antwort-Aktion auf die Stimuli darstellen und diese Assoziationen zum Sequenzen Lernen genutzt wurden.
Eine alternative Interpretation der Stimulus-basierten Theorie besagt, dass die Möglichkeit besteht, dass durch das wiederholte Üben der Sequenz im Gehirn ein Schnellverbindung von dem Stimulus-Enkodieren-Prozess zum Antwort-Ausführen-Prozess entsteht. Diese Verbindung überspringt den Antwort-Auswählen-Prozess und verbessert auf diese Weise die benötigte Ausführungszeit für die Aufgabe und somit auch die Reaktionszeit deutlich. Ändert man allerdings das Schema, wie die Stimuli dargestellt werden, kann die Schnellverbindung nicht mehr genutzt werden. Dieser Ansicht nach spielen nur die Charakteristiken der Stimuli eine Rolle, nicht aber die Stimulus Sequenz. [vgl. 6, S.4f]

2.2.2 Antwort-basierte Theorie

Die *Antwort-basierte Theorie* schlägt vor, dass für das Lernen von Sequenzen eine motorische Komponente involviert sein muss. Zusätzlich soll auch das Ausführen der Antwort-Aktion und der Ort, wo die Antwort-Aktion durchgeführt wird, eine große Rolle beim Lernen der Sequenz spielen. Willingham untersuchte 1999 [vgl. 15] nochmal die Ergebnisse von Experiment 1 von der Studie von Howard et al. (1992) [vgl. 16] und stellte dabei fest, dass eine große Anzahl an Probanden die Sequenz explizit gelernt hatte. Es ist suggeriert worden, dass explizites und implizites Lernen als zwei Prozesse grundsätzlich anders ablaufen und unterschiedliche Hirnareale dafür genutzt werden [vgl. 17]. Willingham wiederholte die Studie von Howard und seinen Kollegen. Daraufhin analysierte er die Daten zuerst mit Studienteilnehmern eingeschlossen, die Anzeichen von explizitem Wissen zeigen. Danach analysierte er die Daten ohne Studienteilnehmer mit

Anzeichen von explizitem Wissen. Wurden die Studienteilnehmer mit explizitem Wissen mit berücksichtigt, so war das Ergebnis identisch. Jedoch konnte man ohne Berücksichtigung der Studienteilnehmer mit explizitem Wissen nur bei denjenigen Probanden den Transfer-Effekt nachweisen, welche auch wirklich eine Antwort-Aktion durchgeführt hatten. In einem weiteren Experiment (Experiment 3) [vgl. 15] unterstützte er die These vom Antwort-basierten Sequenzen Lernen. So wurden zwei verschiedene Gruppen gebildet. Beide Gruppen wurden am Anfang mit der gleichen Sequenz und denselben Antwort-Aktionen trainiert. In der Testphase blieb bei der einen Gruppe nur die Sequenz der Stimuli gleich, während die Teilnehmer dieser Gruppe veränderte Antwort-Aktionen durchführen mussten. Bei der anderen Gruppe dagegen blieb die Sequenz der Antwort-Aktionen erhalten, dafür wurde aber die Sequenz der Stimuli verändert. Die Ergebnisse legten nahe, dass die Gruppe mit konstanten Antwort-Aktionen signifikanten Lernerfolg zeigte, während die Gruppe mit konstanten Stimuli keinen Lernerfolg hatte. Daraus schloss Willingham, dass alleine das Lernen der Orte / Positionen der Antwort-Aktionen Sequenzen Lernen ermöglicht.
Anderen Autoren [vgl. 18] stimmen dieser Idee insofern zu, dass sie einräumen, dass beim Sequenzen Lernen eine motorische Komponente eine Rolle spielen kann, jedoch schließen sie aus, dass das Sequenzen Lernen nur auf die Positionen der Antwort-Aktionen beschränkt ist und gehen davon aus, dass die Reihenfolge der verschiedenen zu machenden Antwort-Aktionen (unabhängig von der Position) das Sequenzen Lernen ermöglicht. [vgl. 6, S.5]

2.2.3 Stimulus-Antwort-Regel-basierte Theorie

Die *Stimulus-Antwort-Regel Hypothese* sieht die Stimulus-Antwort-Regeln und die Auswahl der Antworten als kritische Kernelemente des Sequenzen Lernens. Sie hebt hervor, dass sowohl wahrnehmende als auch motorische Elemente von großer Bedeutung für das Sequenzen Lernen sind.
Die Stimulus-Antwort-Regel-Hypothese nimmt an, dass das Sequenzen Lernen durch die Assoziation von Stimulus-Antwort Regeln in dem Antwort-Auswählen-Prozess (siehe Kapitel Funktionsweise des sequentiellen Lernens) vermittelt wird. Eine Stimulus-Antwort Regel wählt für einen bestimmten Stimulus die passende Aktion aus, welche als Antwort auf den Stimulus ausgeführt werden soll (Abbildung von Stimulus nach Aktion) Nachdem in verschiedenen Studien inkonsistente Ergebnisse gefunden wurden (siehe Kapitel Stimulus-basierte Theorie und Kapitel Antwort-basierte Theorie), geht Schwarb et al. [vgl. 6, S.5] davon aus, dass die Stimulus-Antwort-Regel-basierte Theorie(-Hypothese) als Rahmenkonzept herangezogen werden kann. Nach dieser Hypothese (Stimulus-Antwort-Regel-basierte Theorie) wird Sequenzen Wissen über assoziative Prozesse erlangt, welche beginnen, geeignete Stimulus-Antwort Paare im Arbeitsspeicher des Gehirns zu verlinken. Während der seriellen Reaktionszeit Aufgabe verbleiben ausgewählte Stimulus-Antwort-Paare über mehrere Versuche im Arbeitsspeicher (Kurzspeichergedächtnis), wodurch es zur Co-Aktivierung von mehreren Stimulus-Antwort-Paaren kommt und sich so zeitlich übergreifende Berührungen und Zuordnungen zwischen den Paaren bilden. Interessant ist, dass räumliche Abbildungen vom Stimulus auf die gegebene Antwort

gemacht werden können. So können bestehende Stimulus-Antwort-Paare herangezogen werden, um über die räumliche Abbildung die passende Antwort-Aktion auszuwählen. Die Formel hierfür lautet:

$$R = T(S) \tag{2.1}$$

R = gegebene Antwort-(Aktion)
T = feststehende, konstante räumliche Beziehung zwischen Stimulus und Antwort
S = gegebener Stimulus

Zum Beispiel könnte diese Beziehung so aussehen, dass die Probanden bei der seriellen Reaktionszeit Aufgabe immer eine räumliche Position nach rechts auf die Stimuli antworten sollen. Es ist also eine einfache Transformation des gelernten Wissens (Stimulus-Antwort-Paare) notwendig, um es für diese Aufgabe gewinnbringend einsetzen zu können (Ort der Antwort eins nach recht verschieben).
Weitere Experimente ließen darauf schließen, dass das Sequenzen Lernen weder Stimulus-basiert noch Antwort-basiert ist, sondern über die Stimulus-Antwort-Verbindungen, welche für die Aufgabe benötigt werden, geschieht. Neuerdings (2012) gewinnt diese Theorie an Popularität, weil sich damit die Diskrepanzen und Widersprüchlichkeiten der Ergebnisse der bisher gemachten Studien erklären lassen.
Deroost und Soetens zeigten in ihrer Studie von 2006, dass das Sequenzen Lernen sich verbessert, wenn komplexe Stimulus-Antwort-Abbildungen (mehrdeutig oder indirekt) für die serielle Reaktionszeit Aufgabe benötigt werden. Dies wiederum soll stärker überwachte Antwort-Auswählen-Prozesse bedingen. [vgl. 6, S.5f]

2.3 Sequentielles Lernen von mehreren Aufgaben

Beim Sequentiellem Lernen von mehreren Aufgaben müssen während des Lernvorgangs zwei Aufgaben gleichzeitig erledigt / gelernt werden, was zu einer Beeinträchtigung der simultanen Leistung führt. [vgl. 5] Schließlich gibt es viele kontroverse Forschungsergebnisse zum Thema Sequentielles Lernen bei mehreren Aufgaben. So berichteten viele Studienergebnisse, dass das Sequenzen-Lernen vollständig gelingt, während andere Ergebnisse verschlechtertes Lernen bei mehreren Aufgaben feststellten. Ob das sequentielle Lernen noch gelingt oder verschlechtert wird, hängt stark von der zweiten gewählten Aufgabe ab und dem gesamten Versuchsaufbau. [vgl. 6]

2.3.1 Generelles Vorgehen

In diesem Abschnitt wird der generell verwendete Ansatz zur Untersuchung des sequentiellen Lernens bei mehreren Aufgaben beschrieben. Die Hauptaufgabe für die Studienteilnehmer besteht daraus, auf das Erscheinen eines visuellen Stimulus die passende Taste zu drücken. Die zweite, zusätzlich hinzugenommene Aufgabe ist gewöhnlicherweise das Zählen von Tönen. Dabei hören die Studienteilnehmer sowohl hohe als auch tiefe Töne. Während des Erscheinens eines visuellen Stimulus wird genau ein Ton gleichzeitig abgespielt (hoch oder tief). Die Versuchsprobanden müssen sich beispielsweise immer die

Anzahl der vorgekommenen hohen Töne merken und am Ende eines Blocks ihre ermittelte Anzahl berichten.
Schmidtke dagegen [5] wählte als zweite Aufgabe die Reaktion auf hohe Töne mittels eines Fußpedals, welches die Probanden nach einem hohen Ton mit dem Fuß drücken mussten. In Schmidtke und Heuers Experiment 1 wurden die Probanden in drei Gruppen eingeteilt. Alle Gruppen bekamen eine sechsstellige visuelle Sequenz präsentiert. Einer Gruppe wurde auch eine sechsstellige akustische Sequenz von Tönen vorgesetzt. Die nächste Gruppe hatte nur eine fünfstellige akustische Sequenz zur Verfügung. Für die dritte Gruppe dagegen waren alle akustischen Signale komplett zufällig.

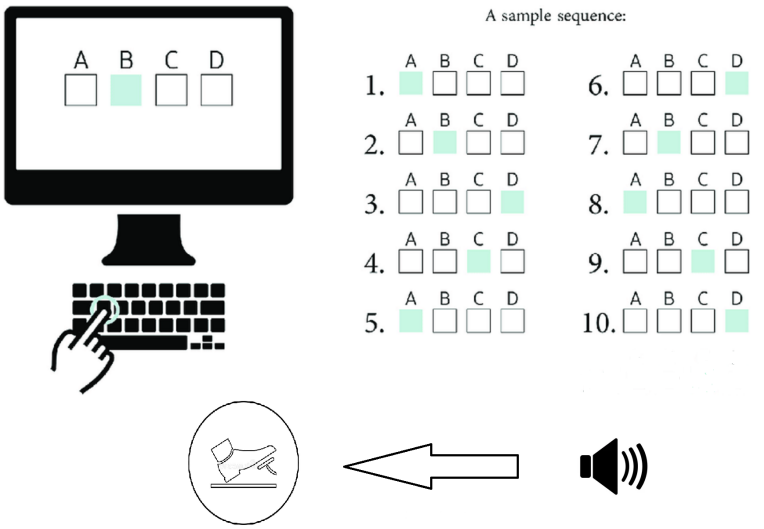


Abbildung 2.2: Versuchsaufbau mit zwei parallelen Aufgaben [9, S.7]

2.3.2 Unterschiede zwischen einer und mehreren Aufgaben

Durch das Hinzunehmen einer zusätzlichen Aufgabe wird für die Probanden die Schwierigkeit erhöht, die Sequenz zu lernen. Durch die gemessenen Reaktionszeiten bei der Hauptaufgabe wird festgestellt, inwiefern sich die Probanden über mehrere Trainingsblöcke verbessern konnten, was dafür spricht, dass die Sequenz gelernt worden ist. Die zusätzliche Aufgabe forciert Multitasking, sodass die volle Konzentration nicht mehr für die Hauptaufgabe verwendet werden kann.

2.3.3 Aufgabenintegrations-Hypothese

Die Aufgabenintegrations-Hypothese besagt, dass sequentielles Lernen, unter der Bedingung, dass zwei Aufgaben gleichzeitig erledigt werden müssen, oftmals beeinträchtigt ist. Sie erklärt es damit, dass das menschliche Gehirn versucht, die visuellen und akustischen Stimuli (Reize) in eine Sequenz aufzunehmen. Weil aber im gewöhnlichen Versuchsaufbau die Töne zufällig abgespielt werden und somit nicht von den visuellen

Stimuli abhängig sind, ist es nicht möglich, aus den visuellen und akustischen Stimuli eine sich wiederholende Sequenz aufzubauen.
Schmidtke und Heuer gehen davon aus, dass Aufgabenintegration nicht mehr stattfindet, falls die akustischen Signale keine Reaktion des Probanden erfordern. Die Ergebnisse der Experimente von Schmidtke und Heuer legen nahe, dass Aufgabenintegration dann am stärksten auftritt, wenn es von Vorteil für die Probanden ist. So konnte lediglich die Gruppe mit der sechsstelligen visuellen und sechsstelligen auditiven Sequenz einen guten Lernerfolg erzielen, während die Gruppe mit der sechsstelligen visuellen und fünfstelligen auditiven Sequenz als auch die Gruppe mit der sechsstelligen visuellen Sequenz und der zufälligen auditiven Sequenz deutlich schlechter abschnitten. Die erste Gruppe, wo die visuelle und akustische Sequenz dieselbe Länge hatten, konnte recht einfach die zwei Sequenzen in einer Sequenz abspeichern (integrieren). Dagegen konnte die zweite Gruppe nicht von den akustischen Signalen profitieren, weil sich durch die Versetzung mit einer sechsstelligen Sequenz und der fünfstelligen auditiven Sequenz eine lange komplexe Sequenz ergibt, welche kaum gelernt werden kann. Interessanterweise wies die erste Gruppe sogar höhere Werte von implizitem Wisse über die Sequenz auf als eine Kontrollgruppe, welche nur die visuelle Sequenz präsentiert bekam und überhaupt keine Töne hörte. Steht die zweite Aufgabe also in konsequenter, sinnvoller Verbindung mit der ersten Aufgabe, kann das implizite Lernen der Sequenz gesteigert werden.

2.4 OPTIMAL-Theorie für motorisches Lernen

Die OPTIMAL-Theorie beschäftigt sich damit, welche Faktoren motorisches Lernen (das Lernen von Bewegungsabläufen) beeinflussen. Genauer untersucht sie, welche Umstände oder Gegebenheiten die zukünftige Leistung verbessern, welche Variablen die Autonomie des Lerners beeinflussen und inwiefern sich ein externer Fokus auf den beabsichtigten Bewegungseffekt auswirkt. OPTIMAL steht hier als Abkürzung für „Optimizing Performance through Intrinsic Motivation and Attention for Learning“ d.h. übersetzt „Optimieren der Leistung(-sfähigkeit) durch intrinsische Motivation und Aufmerksamkeit für den Lernprozess“ [19, S.1]
Sie behauptet, dass auf Motivation und Aufmerksamkeit gerichtete Faktoren die Leistung und den Lernerfolg verbessern, indem die Kopplung von Zielen zu Aktionen gestärkt wird. Die Erwartungshaltung der Lernenden für Erfolg (z.B. erfolgreiche, richtige Ausführung eines Bewegungsablaufs) bestimmt in Kombination mit der Freisetzung einer bestimmten Menge an Dopamin, wie gut die auszuführende Bewegung letztendlich ausgeführt wird. Die Dopamin-Ausschüttung ist abhängig von der Erwartung einer positiven Erfahrung und zeitlich mit der Einübung der Fähigkeit verbunden. Die Autonomie des Lerners wirkt sich vermutlich durch einen verbesserten Erwartungspfad auf das Lernen aus. Ein externer Fokus fördert die Bildung von effizienten, funktionalen Verbindungen von verschiedenen Hirnarealen, welche wiederum förderlich für einen geschickt ausgeführten Bewegungsablauf sind.
Die Verfasser dieser Theorie vermuten, dass eine verbesserte Erwartungshaltung (für Erfolg) und ein externer Fokus die kognitiven und motorischen Systeme des Ausführenden

(einer Bewegung) vorantreiben und ein Zurückfallen (des Ausführenden) in (negative) selbstbezogene und aufgaben-fremd fokussierte Zustände vermeiden. [vgl. 19, S.1]

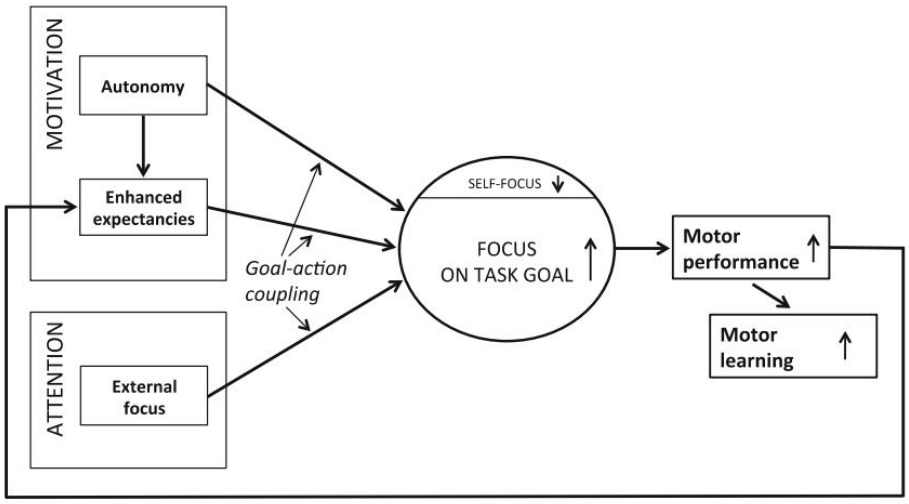


Abbildung 2.3: Skizze des OPTIMAL-Theorie-Modells [19, S.10]

Abbildung 2.3 zeigt skizziert ein Modell für die OPTIMAL-Theorie. So wirken sich Aspekte der Motivation (Autonomie und verbesserte Erwartungshaltung) sowie eine gerichtete Aufmerksamkeit (externer Fokus) positiv darauf aus, dass die Konzentration auf das selbstbezogene Ich abnimmt, während der Fokus auf das Aufgabenziel gestärkt wird. Dies wiederum führt dazu, dass die Ausführung der Bewegung gut gelingt und so auch das Lernen dieser Bewegung gesteigert wird, was wiederum die Erwartungshaltung für eine zukünftig gute (und korrekte) Ausführung dieser Bewegung stärkt. Die Kopplung von Zielen zu Aktionen wird durch Autonomie, verbesserte Erwartungshaltung und einen externen Fokus gewährleistet.

2.4.1 Erwartungshaltung für Erfolg

Motivation wirkt sich auf Energiefreisetzung, Richtung und Intensität von Verhalten aus. Der Begriff Motivation umfasst hier beispielsweise soziale Verhaltensweisen, umgebungsbedingte Beschaffenheiten sowie interne Gedankengänge, Prozesse und Gefühlsreaktionen. Es hat sich gezeigt, dass wir genau dann handeln, wenn die Zukunftserwartung ein Gefühl liefert, dass positive Ergebnisse auftreten werden, und wenn wir Autonomie wahrnehmen. Die Wahrnehmung von Autonomie bedeutet, dass wir die Akteure sind, welche diese positiven Ergebnisse bewirken. [vgl. 19, Abschnitt Motivation]
Die Erwartungshaltung überträgt die persönlichen Erfahrungsgeschichten in relevanten Momenten in einen neuen Kontext, sodass man sich auf zukünftige Ereignisse vorbereiten kann. Außerdem ist die Erwartungshaltung nicht neutral, sondern sie zieht belohnende Eigenschaften in Erwägung, inwiefern Bedürfnisse und Wünsche des Individuums erfüllt werden. Wenn man sich auf gewollte Bewegungen konzentriert, muss die erwartete

Selbstwirksamkeit betrachtet werden. Selbstwirksamkeit drückt das auf eine bestimmte Situation bezogene Selbstvertrauen eines Individuums aus, die Aktionen ausführen zu können, welche (je nach Aufgabe) die gewünschten Ergebnisse erzielen. Vergangene Erfolgserfahrungen bzw. das Fehlen dieser sind die theoretischen Hauptbestimmungsfaktoren für die Selbstwirksamkeit. Selbstvertrauen ist als Prädiktor für die Leistungsfähigkeit identifiziert worden. Außerdem stellt die durch vergangene Erfolgserfahrungen gewonnene Selbstwirksamkeit einen Prädiktor für die Leistung in darauffolgenden Tests für das motorische Lernen dar.
Im Folgenden sollen verschiedene Möglichkeiten aufgezeigt werden, welche die Erwartungshaltung für den Lernenden verbessern können. [vgl. 19, S.3f]

Positives Feedback zur erbrachten Leistung bezüglich des Aufgabenziels bei guten Versuchen und das Ignorieren von schwächeren Versuchen erzielen einen wirkungsvolleren Lernprozess. Durch das positive Feedback nach erfolgreichen Versuchen wird die Wahrnehmung von Selbstkompetenz und Selbstwirksamkeit gestärkt. Auch die Selbsteinschätzung der eigenen Leistung im Vergleich zum Durchschnitt der anderen Teilnehmer spielt eine große Rolle beim Feedback . Dieses normierte, positive Feedback, selbst wenn es überhaupt nicht zutrifft, führt zu einem verbesserten Lernprozess und sogar zu qualitativ besseren Kontrolle über die Bewegungsabläufe, was bessere Automatisierung und Effizienz der Bewegungen bewirkt. Auch soll normiertes, positives Feedback Bedenken und Nervosität über die eigene Leistung und Befähigung reduzieren und zusätzlich auch noch die Zufriedenheit mit der eigenen Leistung stärken, positive Gefühle hervorrufen und somit die Motivation (die Bewegung) zu lernen steigern.
Negatives Feedback dagegen soll Gedankengänge über das eigene Ich anstoßen und Selbstregulierungsaktivitäten bewirken, um die (durch das negative Feedback) angegriffene Selbstachtung auszugleichen (bzw. gutzumachen), welche aber für das erfolgreiche Lernen der Hauptaufgabe hinderlich sind. Selbstmodellierung in Form von editiertem Video-Feedback, welches die Beste Ausführung der gewollten Bewegung zeigt, führt auch zu verbessertem Lernen und zusätzlich noch zu gesteigerter intrinsischer Motivation sowie größerer Zufriedenheit über die gezeigte Leistung. [vgl. 19, S.4f]

Eine weitere Möglichkeit, die Erwartungshaltung auf Erfolg zu verbessern, stellt die wahrgenommene Aufgabenschwierigkeit dar. Wählt man die Maßstäbe so, dass sie vorgeblich auf eine gute Ausführung der erforderlichen Bewegung hindeuten und das Erzielen der guten Ausführung relativ leicht erscheinen lassen, kann die Erwartungshaltung der Lernenden verbessert werden. Beispielsweise kann dem Teilnehmer vor Ausführung der Aufgabe (z.B. eine Balance-Aufgabe) gesagt werden: „Eine aktive und erfahrene Person wie Sie schneidet bei dieser Aufgabe gewöhnlicherweise gut ab“. Wird die zu erfüllende Aufgabe relativ leicht gewählt, so erhöht sich die Chance auf Erfolg während der Übung und verbessert so das Lernen. Wird dagegen der Maßstab für eine gute Leistung so gewählt, dass er relativ schwer zu erreichen ist, kann das Lernen stark beeinträchtigt werden. Die Erwartungshaltung für die Leistung kann durch die Vorstellung und den Hinweis, dass bestimmte Hilfsmittel oder Geräte bei der Durchführung helfen werden, beeinflusst werden. Auch möglich sind optische Illusionen (Manipulation der

wahrgenommenen Schwierigkeit) und manchmal sogar Aberglaube (z.B. der Glaube an „glückliche Golfbälle“ oder die Effektivität von Glücksbringern). Es wird angenommen, dass auch mittels Hypnose die Erwartungshaltung verbessert werden kann, was wiederum die Leistung und das Lernen verbessern würde. Selbst eine verbesserte allgemeine Erwartungshaltung für bestimmte Stresssituationen kann in deutlich verbesserter Leistung bei der Aufgabe (z.B. verbesserte Wurfgenauigkeit) resultieren. [vgl. 19, S.5ff]

Es gibt zwei Wahrnehmungen von Befähigung/Fähigkeit z.B. für eine bestimmte Tätigkeit wie Fußball spielen. Die erste Wahrnehmung ist, dass motorische Fähigkeiten relativ festgelegt sind und nur schwer geändert oder verbessert werden können. Sogenannte Entität-Theoretiker mit dieser Ansicht neigen dazu, die Sorge zu haben, anderen ihre eigene Befähigung beweisen zu müssen, und nehmen negatives Feedback als Angriff auf sich selbst wahr, weil dadurch eine verringerte Belastbarkeit oder ein Fehlen von Befähigung/Talent suggeriert wird.
Die andere Wahrnehmung ist, dass Fähigkeiten veränderbar und formbar sind. Sogenannte Inkrementell-Theoretiker mit dieser Ansicht neigen dazu sich mehr auf den Lernprozess zu konzentrieren und ihre Leistung bei einer gewissen Aufgabe zu verbessern. Sie sind weniger von negativem Feedback (zu gemachten Fehlern) betroffen, weil sie Schwierigkeiten mit gesteigertem Einsatz entgegentreten. Des Weiteren gehen sie davon aus, dass sie sich die Fähigkeiten selbstständig erarbeiten können und sehen Fehler nur als temporär und Teil des Lernprozesses an. Die Wahrnehmung von Befähigung/Fähigkeit kann relativ leicht durch die Aufgabenanweisung oder eine Leistungsrückmeldung (Feedback) beeinflusst werden. Es sollte darauf geachtet werden, dass die Beeinflussung in Richtung von Inkrementeller Theorie gemacht wird, weil nur so die Motivation und Selbst-Einschätzung positiv gehalten werden kann. Außerdem bedingt die Wahrnehmung von erlernbarer Fähigkeit höhere Selbstwirksamkeit und mehr selbstbezogene, positive Gefühle, was wiederum zu höherem Interesse an der Aufgabe und stärkeren Verbesserungen über Versuchsreihen/Trainingseinheiten führt. [vgl. 19, S.7]

Die meisten Erfahrungen, welche verbesserte Erwartungshaltung auf Erfolg in Form von messbaren Ergebnissen bringen, werden wahrscheinlich auch von positiven Gefühlen begleitet. Die Vorahnung von positiven Gefühlen könnte eine gültige Form von verbesserter Erwartungshaltung darstellen, um Leistung und Lernerfolg zu verbessern. Ashby, Ilsen, & Turken; Dreisbach & Goschke; Lyuborminsky, King & Diener [vgl. 20–22] haben untersucht, inwiefern sich positive Gefühle auf gedankliche Prozesse auswirken, und konnten verbesserte kognitive Flexibilität und Kreativität belegen. Laut Berridge ist die Ausschüttung von Dopamin nicht mit dem hedonistischem „Mögen“ von Belohnungen verbunden, sondern mit dem „Herbeisehnen“ von Belohnungen. Daraus folgt, dass positive Gefühle an sich schon belohnend sind, auch wenn kein Bezug zu erbrachter Leistung besteht, aber gleichzeitig auch positive Gefühle nicht die Hauptursache für Belohnungseffekte bei Leistung und Lernen sein können [vgl. 19, S.8].

2.4.2 Autonomie

Über die Umgebung Kontrolle auszuüben stellt für Individuen nicht nur ein psychologisches Grundbedürfnis dar, sondern es handelt sich dabei möglicherweise sogar um eine biologische Notwendigkeit. Eitam, Kennedy und Higgins haben aufgezeigt, dass die menschliche Motivation davon abhängt, inwiefern man das Gefühl hat, dass die eigenen Aktionen einen sichtbaren, erkennbaren Einfluss auf die Umgebung haben. Verschiedene Studien der Motorisches-Lernen-Literatur haben übereinstimmend gezeigt, dass das Lernen von motorischen Fähigkeiten gesteigert wird, wenn dem Lernendem die Kontrolle über bestimmte Gesichtspunkte der Übungsbedingungen überlassen wird. Außerdem hat sich herausgestellt, dass die Art und Weise wie die Aufgaben gestellt werden (autonomie-unterstützend vs kontrollierend) sich auf das motorische Lernen auswirkt. Die autonomie-unterstützende Sprache lässt dem Lerner die Wahl beziehungsweise mehrere Möglichkeiten offen, wie die gestellte Aufgabe erledigt werden kann (beispielsweise ein Kricket-Bowling-Wurf). Dagegen zwängt die kontrollierende Sprache dem Lernenden auf, wie genau er Bewegungen durchführen muss, um die Aufgabe zu erfüllen. In verschiedenen Studien wurde gezeigt, dass die Leistung und das Lernen durch autonomieunterstützende Anweisungen verbessert wird. [vgl. 19, S.11]

Einerseits kann dem Lernenden die Wahl über relevante Übungsbedingungen überlassen werden. So kann dem Lernendem die Wahl überlassen werden, wie lange er üben / trainieren will oder wie groß die Abstände zwischen den verschiedenen Übungsdurchläufen gehalten werden. Der Lernerfolg kann selbst dann beeinflusst werden, wenn der Lernende vor die Wahl über nebensächliche Parameter bzw. Umstände (z.B. vermeintliche Hilfsgegenstände wie eine Ausgleichsstange bei einer Balance-Aufgabe, welche aber nicht bei der Aufgabe hilft) gestellt wird. In einer Studie von Wulf, Chiviacowsky, & Cardozo, (2014) durfte sich eine Gruppe der Studienteilnehmer die Farbe des Golfballs aussuchen, die andere Gruppe der Studienteilnehmer musste sich mit einem bestimmtem Golfball begnügen. Die Gruppe, welche sich die Ballfarbe aussuchen durfte, erzielte einen besseren Lernerfolg als die andere Gruppe. Auch wurde festgestellt, dass die Wurfgenauigkeit verbessert werden kann, wenn man die Studienteilnehmer die Farbe des Wurfballes aussuchen lässt. [vgl. 19, S.12]
Es ist eine Vielzahl von Gründen diskutiert worden, warum die Autonomie des Lernenden (gewährte Kontrolle des Lerners über Übungsbedingungen) das Lernen begünstigt. So soll eine mehr aktiv am Lernprozess beteiligte Person die fürs Lernen wichtigen Informationen tiefgründiger verarbeiten. Außerdem werde durch das Gewähren der Kontrolle über Übungsbedingungen die Fehlereinschätzung gefördert sowie allgemein die Motivation des Lernenden gestärkt, was sich in gesteigertem Interesse des Lernenden an der Aufgabe widerspiegeln kann. [vgl. 19, S.12f]

2.4.3 Aufmerksamkeit

Der Begriff Aufmerksamkeit beschreibt „die Überwachung der Aufgabe und Umgebung, den Umfang und die Breite von physischen oder anderen für die Aufgabe bedeutungsvol-

len Auslösereizen, das Geschick oder die Fähigkeit die Konzentration trotz widersprüchlicher Eingaben oder Ablenkungen kontrollieren zu können sowie auch die auf Inhalte von Auslösereizen, welche wichtig für bestimmte Bewegungen sind, gerichtete Konzentration“ [19, S.15].
Für die Aufmerksamkeit werden zwei verschiedene Fokussierungen unterschieden. So gibt es zum einen die externe Fokussierung, bei der die Aufmerksamkeit und der Blick auf ein externes Objekt gerichtet ist, dass nicht zu einem selbst gehört. Zum anderen gibt es die interne Fokussierung, bei der die Aufmerksamkeit auf sich selbst oder eigene Körperbewegungen gerichtet ist. Über eine Vielzahl an Experimenten hat sich herausgestellt, dass das Anwenden einer externen Fokussierung auf den gewünschten Bewegungseffekt einen Nutzen bringt. So spielt die Wortwahl bei der Aufgabenanweisung eine große Rolle. Im Optimalfall wird die Aufmerksamkeit von den eigenen Körperteilen und dem Selbst weg und gleichzeitig zum gewünschten Bewegungseffekt hingelenkt. Im anderem Fall dagegen wird die Aufmerksamkeit zu dem eigenen Körper und den Bewegungen der verschiedenen Muskelgruppen hingelenkt. [vgl. 19, S.16]

Lernen und Leistung wird bei Balance-Aufgaben durch eine externe Fokussierung verbessert, weil die (ungewollten) Bewegungen des Lernenden minimiert werden. Darüber hinaus wird die Treffergenauigkeit bestimmte Ziele zu treffen durch einen externen Fokus verbessert. Ein Bewegungsmuster wird als effizienter oder ökonomischer angesehen, wenn es dieselbe Auswirkung mit weniger verbrauchter Energie erreicht. So wurde bei Sportübungen mit externer Fokussierung der Ausführenden weniger Muskelaktivität festgestellt, wobei gleichzeitig zusätzlich auch die Präzision bei Aufgaben, bei denen Präzision benötigt wird, erhöht wurde. Darüber hinaus kann mit einer externen Fokussierung eine höhere maximale Kraftproduktion (z.B. maximale vertikale Sprunghöhe, maximale Länge eines Sprungs aus dem Stand heraus oder Diskus-Weitwurf) erreicht werden, wobei auch hier eine verringerte Muskelaktivität festgestellt werden kann. Des Weiteren lässt sich auch die Geschwindigkeit und das Durchhaltevermögen beim Rennen oder Schwimmen durch eine externe Fokussierung steigern. [vgl. 19, S.17]

Daneben soll eine externe Fokussierung die Koordination innerhalb von Muskeln (Bereitstellung von Motoreinheiten) und zwischen den einzelnen Muskeln (z.B. Co-Kontraktionen) und Muskelgruppen verbessern. Tatsächlich wird die Bewegungsform durch eine externe Fokussierung verbessert, was eine bessere technische Ausführung des Bewegungsmusters ermöglicht und so auch die erreichte Leistung steigert. Die verschiedenen nützlichen Effekte einer externen Fokussierung auf den gewünschten Bewegungseffekt können dadurch erklärt werden, dass die Automatisierung der Bewegungsabläufe gefördert wird. Liegt die Aufmerksamkeit auf einem externen Objekt und nicht dem eignen Körper und den Bewegungen der Muskeln, so kann die Steuerung der verschiedenen Muskelgruppen von unbewussten, schnellen und reflexartigen Kontrollprozessen übernommen werden und der Bewegungsablauf wird automatisiert ausgeführt.
Dagegen löst eine interne Fokussierung eine bewusste Art der Kontrolle aus, welche das Bewegungssystem einschränkt, weil es automatische Kontrollprozesse (für die Bewegungen) beeinträchtigt (eingeschränkte Handlungshypothese). Eine externe Fokussierung

auf den gewünschten Bewegungseffekt scheint auch die „funktionale Variabilität“ zu erhöhen, welche ansonsten nur bei erfahrenen, fähigen Sportlern beobachtet werden kann. Obendrein werden durch die externe Fokussierung Aufmerksamkeitsanforderungen in Bezug auf die Aufgabe reduziert, was sich durch kürzere Reaktionszeiten zeigt. [vgl. 19, S.18f]

2.4.4 Einfluss von Hormonen

Wie im ich Kapitel Erwartungshaltung für Erfolg bereits erwähnt habe, sind positive, gute Lernerfahrungen oft mit positiven Gefühlen und damit einhergehend der Ausschüttung des Hormons Dopamin verbunden. Dopamin scheint dann ausgeschüttet zu werden, wenn die Chancen auf eine positive, erfolgreiche Absolvierung der Übung gut stehen (abhängig von der Erwartungshaltung des Übenden). Das Stresshormon Cortisol wird vom Körper vermehrt freigesetzt, wenn die Autonomie des Übenden eingeschränkt wird. Für einen guten Lernerfolg sind eine hohe Menge an Dopamin förderlich, während eine hohe Menge Cortisol das Lernen behindert. Zu einer Belohnung zugehöriges Dopamin stärkt die Wiederholung oder Reaktivierung von Erinnerungen während der Erholungsphase und trägt damit zur Konsolidierung/Festigung des gelernten Wissens bei. Es ist bekannt gemacht worden, dass die Stärke der funktionalen Verbindungen in Ruhezustandsnetzwerken die Geschichte von aufgabenabhängigen Co-Aktivierungen widerspiegelt.[vgl. 19, S.8, S.22]

2.4.5 Ziel-Aktions-Kopplung

Es wird angenommen, dass beim motorischen Lernen sowohl in der Neuroanatomie als auch in funktionalen Verbindungen zwischen verschiedenen Hirnregionen strukturelle Änderungen auftreten. Wulf et al gehen davon aus, dass bei Übungen unter optimalen Bedingungen (hohe Motivation, Aufmerksamkeitsfokus) effektivere neuronale Verbindungen entstehen, welche dann die Leistung stärken und effizienteres Lernen ermöglichen. Die verbesserte Erwartungshaltung in Kombination mit Autonomie und Aufmerksamkeitsfokus weisen die sich Bewegenden mit relativer Klarheit zu ihren Aktionszielen. Außerdem könnten diese treibenden Kräfte auch unterdrücken, dass die Aufmerksamkeit auf nebensächliche Aufgaben oder sich selbst gelenkt wird.
Funktionale Konnektivität beschreibt temporale Verbindungen zwischen räumlich abgetrennten neuronalen Regionen oder Netzwerken. Diese verschiedenen Strukturen und deren Subsysteme werden in Bezug auf die (sportliche)Leistung des Lernens von Aufgaben aktiviert und deaktiviert. Die funktionale Konnektivität kann bei einer Reihe von Krankheiten und psychischen Störungen unterbrochen bzw. gestört sein (z.B. Parkinson, Depressionen, Alzheimer, Autismus oder Schizophrenie). Jedoch lässt sich über die vorhandene funktionale Konnektivität der Lernerfolg verfolgen und ausgeprägtere, funktionale Verbindungen werden mit hoher Befähigung (in dem Sport/für die Bewegung) in Verbindung gebracht. [vgl. 19, S.22]

2.4.6 Umsetzung in der Implementierung

Im Folgendem wird beschrieben, wie die Kernelemente der OPTIMAL-Theorie in die Implementierung bzw. das Design der VR-Anwendung wie auch der Durchführung der Pilot-Studie eingebracht werden können, um den mit der VR-Anwendung erzielten Lernerfolg bei den Probanden zu erhöhen. So kann Autonomie über die verschiedenen implementierten Schwierigkeitsgrade gewährleistet werden, wobei die Probanden nach jedem Block die Schwierigkeit anpassen können. Außerdem kann der Spieler den Abstand zu den Würfeln, welche zerschlagen werden sollen, im Schwierigkeitsmenü je nach Armlänge frei einstellen. Die von den Probanden zu absolvierenden Blöcke können beliebig aus zufälligen Sequenzen und festen über einen Leveleditor erstellten Sequenzen zusammengestellt werden. Hier könnte man die Probanden eine Sequenz frei zwischen mehreren Sequenzen aussuchen lassen.
Durch die Verwendung des Virtual Reality Headsets *HMD Vive* befinden sich die Probanden in einer virtuellen Umgebung, wenn sie die Würfel zerschlagen. Dies stärkt den externen Fokus, weil man sich selbst in der virtuellen Umgebung weniger stark wahrnimmt und der Fokus auf die zwei Schwerter, die mithilfe der Controller in den Händen gehalten werden, auf die zu absolvierende Aufgabe - das Zerschlagen der Würfel - gelenkt wird. Über Hintergrundmusik und auditives sowie visuelles Treffer-Feedback, wie präzise der Würfel durchschlagen wurde, wird die externe Fokussierung noch weiter gestärkt. Schließlich sind die Anweisungen (siehe AntizipationsTest) so formuliert, dass die Aufmerksamkeit der Probanden zur externen Fokussierung hingelenkt wird.
Verbesserte Erwartungshaltung für Erfolg wird durch positives Feedback auf besonders gute Schlagversuche gestärkt. So wird über verschiedene Farben visuell verdeutlicht, wenn ein Schlag besonders viele Punkte erbracht hat. Dadurch dass alle relevanten Daten eines Versuchs (Schlaggenauigkeit, Reaktionszeit, erreichte Punktzahl, etc.) gespeichert werden, kann die eigene Leistung mit der Leistung vorangeganger Probanden verglichen werden und so kann eingeschätzt werden, ob man sich besser als der Durchschnitt schlägt.
Durch angenehme Hintergrundmusik wird eine positive Stimmung beim Probanden erzeugt. Durch Anweisungen wird verdeutlicht, dass sich die Fähigkeit des schnellen und präzisen Durchschlagens der Würfel durch wiederholtes Üben erlernen und verbessern lässt.

3 Technische Grundlagen

3.1 Game Engines

Für die folgenden zwei Kapitel Game Engine und Physik Engine halte ich mich größtenteils an Gregory et al. [vgl. 23, Kapitel 13]. Eine Game Engine enthält eine gewisse Menge von Kernkomponenten. Dazu zählen beispielsweise die Rendering-Engine, das Kollisionerkennungssystem und die Physik-Engine, das Animationssystem, das Audiosystem und das Objektmodell für die Spielwelt sowie ein künstliches Intelligenzsystem. Mithilfe der von einer Game Engine zur Verfügung gestellten Tools und Komponenten kann eine Vielzahl von Spielen und Anwendungen entwickelt werden.
Ein Spiel meint hier meistens eine virtuelle, dreidimensionale Spielumgebung, in welcher der Spieler die Kontrolle über den Hauptcharakter des Spiels (ein Mensch, ein Tier oder ein Fahrzeug, etc.) erhält. Raph Koster definiert ein Spiel als „interaktive Erfahrung, welche dem Spieler eine stetig in der Schwierigkeit ansteigende Sequenz von Spielmechaniken zur Verfügung stellt, welche eventuell vom Spieler gelernt und gemeistert werden“ [24, Übersetzung übernommen von 23].

Merkmale und Eigenschaften der verschiedenen Tools der Game Engine lassen sich über Skriptsprachen erweitern und modifizieren. Heutzutage lassen sich viele Game Engines lizensieren, was den Vorteil bringt, dass nicht alle Tools der Gameengine neu entwickelt werden müssen. Allerdings kann es auch sinnvoll sein, eine Game Engine von Grund auf neu zu entwickeln, weil man so über ein größeres Verständnis verfügt, wie die Tools funktionieren und man die Tools so umsetzen kann, dass sie für das geplante, zu entwickelnde Spiel am effizientesten funktionieren bzw. die erforderte Funktionalität bieten. Normalerweise ist eine Game Engine auf ein bestimmtes Spielgenre wie beispielsweise FPS, Platformer, Rennspiel, Strategiespiel, etc. spezialisiert, doch mit den heutigen Spieleengines lassen sich auch leicht Spiele für fast alle Genres entwickeln.

3.2 Unity

Die Implementierung der Arbeit habe ich mithilfe der Game Engine Unity umgesetzt. In Unity wird die Skriptsprache C# verwendet, sodass der Hauptteil der Implementierung mit C#-Skripten umgesetzt ist. Unity unterstützt als Entwicklungsumgebung eine große Vielzahl an Plattformen (darunter mobile Endgeräte, Spielekonsolen und Desktop-PCs). Außerdem unterstützt Unity die Entwicklung von VR-Anwendungen, was für diese Arbeit eine entscheidende Rolle spielt. Der Unity Editor ist einfach verständlich und lässt sich leicht bedienen. In Unity wird jedes der Objekte, die sich in der Spielwelt befinden,

als GameObject behandelt und es können beliebig viele Skripte angehängt werden, welche von dem Skript MonoBehavior erben. MonoBehavior definiert als Basisklasse grundlegende Funktionen wie *Start* (Initialisierung von Variablen) und *Update* (Code, welcher wiederholt einmal pro Frame aufgerufen wird), welche für die Implementierung von Spielmechaniken genutzt werden können. Ebenfalls nennenswert ist das Prefab-System, über welches sich beliebig komplexe Objekte zusammenbauen lassen, welche wiederum auch auf anderen Prefabs aufbauen können (Nested Prefabs). Von diesen erstellten Prefabs (sichtbar im Asset-Explorer) können Objekt-Instanzen über Skripte instantiiert werden. Weil Unity eine von vielen Entwicklern genutzte Gameengine ist, gibt es im Internet eine gute Dokumentation und Hilfestellung zu verschiedensten Fragestellungen bezüglich der Gameengine bzw. deren Tools.

3.3 Physik Engines

Die Physikengine setzt sich aus Kollisionserkennungssystem und Starrkörperdynamik zusammen. Das Kollisionserkennungssystem hat die Aufgabe, auftretende Kollisionen zwischen den sich in der Spielwelt befindlichen Objekten zu erkennen, welche sich unter Umständen auch bewegen. Die Starrkörperdynamik wiederum berechnet, wie sich in der Spielwelt befindliche starre Körper physikalisch korrekt durch aufgetretene Kollisionen verschieben müssen, beziehungsweise welche Kräfte auf die starren Körper angewendet werden müssen. Mithilfe der Starrkörperdynamik kann eine Reihe von physikalischen Verhaltensweisen umgesetzt werden:

1. Objekte, welche gegenseitig voneinander abprallen
2. Objekte, welche sich durch Reibungskräfte verschieben
3. Objekte, welche auf dem Boden herumrollen
4. Objekte, welche nach einer Bewegungsphase zur Ruhe kommen

Für die Arbeit spielt jedoch das Kollisionserkennungsystem die entscheidende Rolle, weil Kollisionen zwischen VR-Controllern und Würfeln in der Spielwelt erkannt werden müssen, jedoch Starrkörperdynamik nicht benötigt wird. Deshalb wird das Kollisionserkennungssystem im nächsten Kapitel genauer behandelt, während die Starrkörperdynamik hier außen vor gelassen wird.

3.3.1 Kollisionserkennungssystem

Die Hauptaufgabe des Kollisionserkennungssystems ist es, festzustellen, ob und zu welchem Zeitpunkt Objekte in der Spielwelt sich berühren bzw. sich überschneiden. Um diese Hauptaufgabe erfüllen zu können, benötigen die Objekte in der Spielwelt eine oder mehrere geometrische Formen. Das Kollisionserkennungssystem berechnet dann, ob sich diese geometrischen Formen zu einem beliebigen Zeitpunkt schneiden. Die geometrischen Formen werden meistens recht einfach gewählt, sodass sie die tatsächliche Form

des Objekts nur annähern (z.B. Sphäre, Box oder Kapsel). Es können auch komplexere geometrische Formen verwendet werden, allerdings sind dann die Überschneidungsberechnungen auch teurer, d.h. die Berechnung der möglichen Überschneidungen nimmt mehr Zeit in Anspruch. Zusätzlich liefert das Kollisionserkennungssystem wichtige Informationen zu den Eigenschaften jedes Kontaktes, falls eine Kollision festgestellt wurde. Zu diesen Informationen zählt unter anderem die Menge aller Kontaktpunkte zwischen den zwei Objekten, wo die Kollision aufgetreten ist. Die Kollisionsdaten können in einer Vielzahl an Möglichkeiten verwendet werden. Beispielsweise kann man mithilfe dieser Daten verhindern, dass sich zwei Objekte gegenseitig durchdringen. Außerdem kann das Triggern eines Events ausgelöst werden, sobald die relevante Kollision auftritt, welche mithilfe der Kollisionsdaten bestimmt wird. Dieses Event könnte unter Anderem die mittels Skript ausgelöste Explosion eines Objekts in der Spielwelt sein.

Überschneidung von geometrischen Formen

Die Überschneidung zwischen zwei geometrischen Formen ist als die Menge (das Set) aller Punkte definiert, welche innerhalb beider Formen liegen. Ob ein Punkt innerhalb einer Kugel liegt, kann getestet werden, indem der Trennvektor s zwischen dem Punkt p und dem Zentrum c der Kugel berechnet wird. Der Trennvektor s wird folgendermaßen berechnet:

$$s = c - p$$

Der Punkt liegt innerhalb der Kugel, wenn die Länge des Trennvektors kleiner oder gleich dem Radius r der Kugel ist: $falls\ |s| \leq r$, dann liegt p innerhalb des Kugelradius. Um herauszufinden, ob sich zwei Kugeln schneiden, kann man den Trennvektor zwischen den zwei Zentrumspunkten der Kugeln berechnen. Falls die Länge des Trennvektors kleiner oder gleich der Summe der zwei Radien der Kugeln ist, dann schneiden sich die zwei Kugeln.

Kontaktinformationen

In den Kontaktinformationen sind alle möglichen Informationen zu entstanden Kontakten zwischen Objekten vorhanden, welche miteinander kollidiert sind. In den meisten Fällen ist es ausreichend zu wissen, ob zwei Objekte miteinander kollidiert sind, aber dennoch bietet das Kollisionssystem eine Reihe weiterer Informationen verpackt in einer geeigneten Datenstruktur. Oft ist ein Trennvektor enthalten, mit Hilfe dessen die zwei Objekte effizient aus der Kollision heraus bewegt werden können. Neben der Hauptinformation, welche zwei mit Collidern ausgestatteten Objekte miteinander kollidiert sind, werden auch noch Informationen darüber geliefert, welche verschiedenen Formen (z.B. Kugel oder Box) sich überschnitten haben und möglicherweise sogar, welche Bestandteile der einzelnen Formen von der Kollision betroffen sind.

Kontinuierliche Kollisionserkennung

Bewegen sich Objekte sehr schnell, kann es zum sogenannten Tunnelungseffekt kommen. Hier werden die aufgetretenen Kollisionen nicht mehr alle zuverlässig erkannt, d.h. Objekte, welche eigentlich kollidieren sollten, gleiten stattdessen durch das andere Objekt ohne Reaktion hindurch.
Die kontinuierliche Kollisionserkennung hilft dabei, dieses Problem zu umgehen. Das Ziel der kontinuierlichen Kollisionserkennung ist es, den frühesten Zeitpunkt des Zusammentreffens zweier sich bewegender Objekte in einem bestimmten Zeitintervall herauszufinden. Dabei wird für jedes mit Collider ausgestattete Objekt sowohl Position als auch Orientierung des vorhergehenden Zeitschritts zwischengespeichert und zusätzlich wird die Position und die Orientierung des aktuellen Zeitschritts herangezogen. So kann man Position und Orientierung unabhängig voneinander interpolieren und erhält somit für jeden beliebigen Zeitpunkt eine Annäherung für den tatsächlichen Transform (Kombination aus Position und Orientierung). Der Algorithmus sucht immer nach dem frühesten Zeitpunkt entlang des Bewegungspfads der sich bewegenden Objekte.

3.3.2 Starre Körper (Rigidbodys)

Laut Gregory ist ein starrer Körper ein „idealisiertes, unendlich hartes, nicht-verformbares, robustes Objekt“ [23, S.817]. Diese Definition erleichtert die Berechnung der Starrkörperdynamik, weil Verformungen von Objekten nicht berücksichtigt werden und somit nur die Verschiebung der Position in der Spielwelt berechnet werden muss. In Unity können verschiedene Parameter für einen starren Körper eingestellt werden. So legt die Eigenschaft *isKinematic* fest, ob bei einem Zusammenstoß mit einem anderen Objekt Kräfte durch die Physik-Engine angewendet werden sollen.
Eine weitere Eigenschaft *Use Gravity* bestimmt, ob das Objekt entlang der y-Achse herunterfallen soll, bis es auf ein solides Objekt trifft (z.B. einen mit Collider ausgestatteten Flur). Außerdem kann man einstellen, welche Kollisionserkennung für dieses Objekt angewendet werden soll. Möglich sind hier die Optionen *diskret*, *kontinuierlich*, *kontinuierlich-dynamisch* und *kontinuierlich-spekulativ*. Die Option diskret schneidet bei der Performance besser ab als die anderen Varianten, vermeidet dafür aber den Tunnelungs-Effekt überhaupt nicht.
Die Optionen kontinuierlich und kontinuierlich-dynamisch basieren auf der Sweep-basierten kontinuierlichen Kollisionserkennung. Beim Sweep-basierten Verfahren berechnet der Algorithmus mögliche Kollisionen für ein Objekt, indem er die Vorwärts-Bahnkurve des Objekts abtastet, wobei die aktuelle Geschwindigkeit des Objekts einbezogen wird. Werden Kollisionspunkte entlang der Bewegungsrichtung des Objektes gefunden, berechnet der Algorithmus den Zeitpunkt des Aufpralls und bewegt das Objekt bis zu diesem Zeitpunkt. Der Algorithmus kann ab diesem Zeitpunkt Teilschritte ausführen, bei denen die Geschwindigkeit nach dem Zeitpunkt des Aufpralls berechnet wird, und dann die Vorwärts-Bahnkurve des Objekts neu abtasten. Jedoch wird bei dieser Variante der Tunnelungs-Effekt nicht vermieden, wenn sich Objekte schnell bewegen und dabei auch noch beständig rotieren, weil die Winkelbewegung des zum Objekt gehörigen Körpers

nicht berücksichtigt wird.

Bei der Option kontinuierlich-spekulativ dagegen funktioniert die kontinuierliche Kollisionserkennung dadurch, dass während der Broad Phase die minimal achsen-ausgerichtete Box (siehe Kapitel Kollisions-Primitive) abhängig von der linearen Bewegung und Winkelbewegung des Objekts vergrößert wird. Hier sind außerdem alle von einem Solver gefundenen Kontakte spekulativ, weil sie erst während des nächsten Zeitschritts berechnet werden. Dies kann dazu führen, dass eine sogenannte Geisterkollision auftritt, die eigentlich nicht auftreten sollte. Dies macht sich dadurch bemerkbar, dass das Objekt durch den spekulativ gefundenen Kollisionspunkt beeinflusst wird und beispielsweise in irgendeine Richtung gelenkt wird. Allerdings kommt es auch hier in Einzelfällen immer noch zum Tunnelungs-Effekt, wenn sich ein schnell bewegendes Objekt durch einen anderen Collider hindurch bewegt, weil die spekulativen Kontaktpunkte nur während der Kollisionserkennungsphase berechnet werden (siehe Schritt 2 im Kapitel Funktionsweise eines Kollisions-Physik-Schrittes) [vgl. 25].

3.3.3 Funktionsweise eines Kollisions-Physik-Schrittes

Für jede auftretende Kollision, welche von der Starrkörperdynamik verarbeitet wird, erledigt die Physik-Engine bestimmte Aufgaben während jedes Update-Zeitschritts. Bei Objekten mit Trigger-Collidern ohne Rigidbody-Komponente wird für die reine Kollisionserkennung nur der 2. Schritt durchgeführt, weil dadurch die Gesamtberechnung für die Kollisionserkennung weniger Zeit kostet.

1. Die auf den Körpern liegenden Kräfte und Drehkräfte werden für die verstrichene Zeit ∇t miteinbezogen, um so die vorläufigen Positionen und Orientierungen für den nächsten Zeitschritt (das nächste Frame) feststellen zu können.

2. Der Kollisionserkennungs-Algorithmus der verwendeten Bibliothek wird aufgerufen, um festzustellen, ob neue Kontakte zwischen irgendwelchen Objekten durch die vorläufigen Bewegungen (siehe Schritt 1) entstanden sind. Es muss angemerkt werden, dass Kontaktpunkte von den zugehörigen Körpern zwischengespeichert werden, um temporale Kohärenz ausnutzen zu können. Somit muss zu jedem Zeitschritt der Simulation die Kollisions-Engine untersuchen, ob vorherige Kontakte verloren gegangen sind oder ob neue Kontakte hinzugefügt worden sind.

3. Die Kollision werden aufgelöst. Dies geschieht oft dadurch, dass Impulse oder Straf-Kräfte auf die Objekte angewendet werden. Dies kann jedoch auch im nächsten Schritt als Teil des Einschränkung Lösens erfolgen. Je nach SDK werden in dieser Phase auch die Berechnungen für die kontinuierliche Kollisionserkennung gemacht.

4. Die Einschränkungen werden vom Einschränkungs-Solver erfüllt. Die hier angeführten Einschränkungen werden für starre Körper definiert. Ein uneingeschränkter, starrer Körper kann sich entlang 3 Achsen bewegen und um 3 Achsen rotieren

(6 Freiheitsgrade). Die Einschränkungen werden verwendet, um die Bewegungsfreiheit eines Objekts teilweise oder komplett einzuschränken. Eine Einschränkung für einen schwingenden Kronleuchter könnte beispielsweise als Punkt-zu-Punkt-Einschränkung umgesetzt werden, was ähnlich wie ein Kugelgelenk wirkt. Die Punkt-zu-Punkt-Einschränkung legt fest, dass sich die betroffenen Körper solange frei bewegen können, solange ein spezifizierter Punkt auf einem Körper sich an einem spezifiziertem Punkt auf einem anderem Körper koordiniert. Der Einschränkungs-Solver ist ein iterativer Algorithmus, welcher versucht eine große Menge an Einschränkungen zur gleichen Zeit zu erfüllen, indem der Fehler/die Abweichung zwischen den aktuellen Positionen und Rotationen der Körper und ihren durch Einschränkungen definierten optimalen Positionen und Rotationen minimiert wird.

Nach der Fertigstellung von Schritt 4 könnten sich einige Körper von ihren vorherigen vorläufig berechneten Positionen fortbewegt haben, sodass neue Überschneidungen zwischen Objekten entstanden sind oder vorher eingehaltene Einschränkungen nun nicht mehr erfüllt sind. Deshalb müssen die Schritte 1 bis 4 so lange durchlaufen werden, bis entweder alle Kollisionen erfolgreich aufgelöst wurden und außerdem auch alle Einschränkungen eingehalten werden oder aber eine vorher definierte maximale Anzahl von Iterationen erreicht wurde. Im letzterem Fall werden die bisher nicht erfolgreich aufgelösten Kollisionen im nächsten Zeitschritt (Frame) berechnet, um Performance-Probleme der Anwendung (z.B. Darstellung) zu vermeiden.

3.3.4 Collider

Wenn ein Objekt in der Spielwelt für Kollisionsberechnungen miteinbezogen werden soll, muss das Objekt eine Kollisionsrepräsentation erhalten, welche die geometrische Form, die Position und Orientierung in der Spielwelt beschreibt. Ein Collider definiert genau diese geometrische Form. Es muss angemerkt werden, dass die geometrische Form (Collider) nicht der visuellen Repräsentation des Objekts entsprechen muss.
Generell sind geometrisch und mathematisch einfache Formen zu bevorzugen, damit die Berechnungszeit für Überschneidungen bzw. Kollisionen effizient und schnell bleibt. Beispielsweise kann ein menschlicher Körper annäherungsweise durch einen Zusammenschluss von verbundenen Kapseln realisiert werden. Eine komplexere geometrische Form sollte nur dann verwendet werden, wenn sich eine simplere Form nicht für den gedachten Zweck im Spiel eignet. In Unity wird zwischen zwei Arten von Collidern unterschieden, den statischen und dynamischen Collidern. An jedes Objekt in der Spielwelt können beliebig viele Collider angehängt werden. In den folgenden zwei Unterkapiteln sollen die Eigenschaften der Collider näher beleuchtet werden.

Statische Collider

Hat ein Objekt einen Collider, jedoch keine Rigidbody-Komponente (bzw. Starrer-Körper-Komponente), wird dieser Collider als *statisch* betrachtet. Statische Collider eignen sich

für Objekte, welche sich nicht bewegen und an einer festen Position in der Szene verankert sind, z.B. Mauern oder der Boden / Untergrund einer Szene. Statische Collider können auch mit dynamischen Collidern interagieren, reagieren aber nicht auf auftretende Kollisionen. Das heißt die Kollision wird vom Kollisionserkennungssystem für den statischen Collider erkannt, jedoch kommt hier keine Starrkörperdynamik zum Tragen und es werden keine Kräfte auf das zum statischen Collider zugehörige Objekt angewandt. In bestimmten Fällen kann die Physik Engine Rechenzeit für Objekte mit statischen Collidern, welche sich nicht bewegen, einsparen [vgl. 26].

Dynamische Collider

Hat ein Objekt einen Collider und zusätzlich auch noch eine Rigidbody-Komponente (bzw. Starrer-Körper-Komponente), wird dieser Collider als *dynamisch* betrachtet (Im Implementierungskapitel nenne ich diese Collider *Physik-Collider*) Hier muss nochmal zwischen kinematischen und nicht-kinematischen Rigidbody-Komponenten unterschieden werden. Ist die isKinematic-Eigenschaft der Rigidbody-Komponente aktiviert, so reagiert das zugehörige Objekt nicht auf Kollisionen und es werden keine Kräfte auf das Objekt durch die Physik Engine angewandt, jedoch kann das Objekt per Skript bewegt werden. Dagegen reagiert ein Objekt mit nicht-kinematischer Rigidbody-Komponente auf Kollisionen und Kräfte, welche über ein Skript auf das Objekt angewandt werden. Hier kommt die Starrkörperdynamik zum Tragen [vgl. 26].

3.3.5 Kollisions-Primitive

Kollisions-Primitive bezeichnen einfache Formen bzw. Körper, aus denen komplexere Formen zusammengefügt werden können. Ein Kollisionserkennungssystem kann gewöhnlicherweise mit einer geeigneten Auswahl an Kollisions-Primitiven funktionieren. Nachfolgend werden die wichtigsten Kollisions-Primitive kurz erläutert.

Sphäre Die Sphäre stellt den einfachsten dreidimensionalen Körper dar und ist für die Kollisionsberechnung bzw. Kollisionserkennung am effizientesten. Die Sphäre wird durch einen Zentrumspunkt und Radius definiert. Auch in Unity wird diese durch den *Sphere Collider* umgesetzt.

Kapsel Die Kapsel ist ein pillenförmiger Körper, welcher aus einem Zylinder besteht, der an beiden Enden mit je einer Halbkugel abgeschlossen ist. Die Kapsel wird als Datenstruktur durch zwei Punkte und einen Radius definiert, wobei die zwei Punkte jeweils das Zentrum einer der zwei Halbkugeln festlegen. Auch in Unity wird diese durch den *Capsule Collider* umgesetzt.

Achsen-ausgerichtete Box Die Achsen-ausgerichtete Box ist ein Quader, dessen Seitenflächen parallel zu den Achsen des Koordinatensystems ausgerichtet sind. Die Achsen-ausgerichtete Box lässt sich als Datenstruktur durch zwei Punkte definieren. Ein Punkt enthält dabei die minimalen Koordinaten $(x_{min}, y_{min}, z_{min})$ der Box entlang der drei

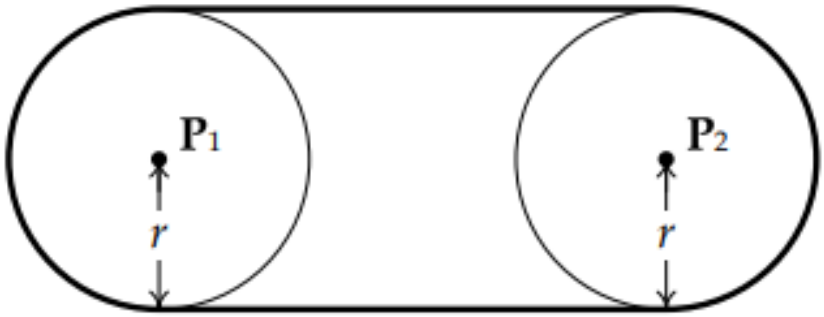


Abbildung 3.1: Schematische Skizze einer Kapsel [23, S.831]

Hauptachsen, während der andere Punkt die maximalen Koordinaten $(x_{max}, y_{max}, z_{max})$ enthält.

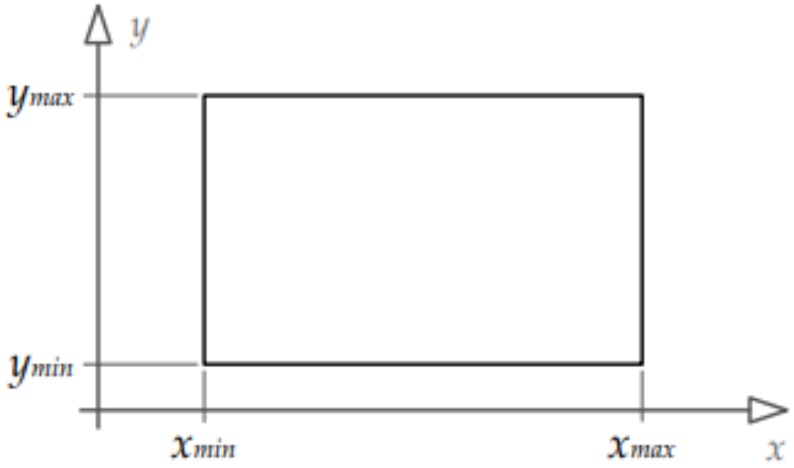


Abbildung 3.2: Skizze einer zweidimensionalen achsen-ausgerichteten Box [23, S.832]

Achsen-ausgerichtete Boxen sind von großer Effizienz, wenn man auf Überschneidung zwischen mehreren dieser Boxen testet. Jedoch ist eine große Einschränkung, dass die achsen-ausgerichteten Boxen immer zum Koordinatensystem ausgerichtet bleiben müssen, damit der Effizienzvorteil erhalten bleibt. Es sei angemerkt, dass die Verwendung einer achsen-ausgerichteten Box nur dann sinnvoll ist, wenn die Koordinatenachsen des Objekts mit den globalen Koordinatenachsen in etwa gleichgerichtet sind, weil ansonsten das Objekt wegen der Einschränkung mit der Ausrichtung der Achsen durch die Box nicht mehr gut approximiert werden kann.
In Unity sind dagegen orientierte Boxen umgesetzt, welche mehr Freiheit zu den Achsen-ausgerichteten Boxen besitzen, da hier die Boxen relativ zum lokalen Koordinatensystem

des zugehörigen GameObjects beliebig rotiert werden können. Hier ist die Datenstruktur durch drei halbierte Dimensionen (Halbe Breite, halbe Tiefe und halbe Höhe) und eine Transformation beschrieben, welche das Zentrum der Box beschreibt und die Orientierung in Relation zu den Koordinatenachsen festlegt.

3.3.6 Optimieren der Kollisionsberechnung für konvexe Körper / Formen

Definition Konvexität

Bei der Berechnung von sich schneidenden Formen muss zwischen konvexen und nichtkonvexen (z.B. konkaven) Formen unterschieden werden, da die Berechnung bei konvexen Körpern deutlich effizienter ist. Eine konvexe Form ist so definiert, dass jeder Strahl, der innerhalb der Form beginnt nicht mehrmals auf die Fläche (den Rand) der Form trifft. Alexandrov definiert die Konvexität einer Oberfläche eines Körpers folgendermaßen: „Die (Teil-) Oberfläche eines Körpers ist konvex, wenn die gerade Strecke zwischen beliebig wählbaren Punkten dieser Fläche komplett innerhalb des Körpers verläuft“ [27, übernommen von 28]. Beispielsweise sind unter zweidimensionalen Objekten Kreise, Rechtecke und Dreiecke konvex, während ein (fünfzackiger) Stern oder die Mondsichel nicht konvex sind.

Trennende-Achsen-Theorem

Das Trennende-Achsen-Theorem wird von den meisten Kollisionserkennungssystemen ausgenutzt. Es besagt, dass falls eine Achse gefunden werden kann, entlang derer die Projektionen von zwei konvexen Formen sich nicht überschneiden, man sicher davon ausgehen kann, dass die zwei Formen sich auch nicht schneiden. Im Umkehrschluss gilt auch, wenn keine Achse gefunden werden kann und die Formen konvex sind, dass sich die zwei Formen überlappen müssen.
Da das Theorem nur für konvexe Formen gilt, ist dies auch ein Grund, warum bei der Kollisionserkennung konvexe Formen gegenüber konkaven Formen bevorzugt werden. Im zweidimensionalem Raum kann man das Theorem so veranschaulichen, dass, falls eine Gerade gefunden werden kann, sodass sich Objekt A auf der einen Seite und Objekt B auf der anderen Seite befindet, dann überschneiden sie sich auch nicht. Diese Trennlinie steht senkrecht zu der trennenden Achse. Die Projektion einer zweidimensionalen konvexen Form auf eine Achse wirkt wie der Schatten, den das Objekt auf einen dünnen Draht hinterlassen würde. Im dreidimensionalem Raum wird aus der Trennlinie eine Trennebene. Die Projektion kann als Minimum und Maximum Koordinate betrachtet werden, welche zusammen das geschlossene Intervall $[c_{min}, c_{max}]$ bilden. Um Überschneidungen zwischen zwei konvexen Formen A und B, bei denen die möglichen Trennachsen offensichtlich sind, zu erkennen (beispielsweise zwei Kugeln), müssen die Formen nacheinander entlang jeder möglichen Trennachse projiziert werden. Wenn zwei Kugeln einen unendlich kleinen Abstand zueinander haben, dann liegt die einzige Trennachse parallel zu der Gerade, welche durch die zwei Zentrumspunkte der Kugeln gebildet wird. Je wei-

ter die Kugeln voneinander entfernt, desto mehr Trennachsen existieren als Rotation der ursprünglichen Trennachse. Anschließend muss dann getestet werden, ob die Projektionsintervalle $[cA_{min}, cA_{max}]$ und $[cB_{min}, cB_{max}]$ überlappen. Die Intervalle sind disjunkt, falls $cA_{max} < cB_{min}$ oder $cB_{max} < cA_{min}$ gilt. Falls die Projektionsintervalle entlang einer möglichen Trennachse disjunkt sind, wurde somit eine Trennachse gefunden und somit steht fest, dass die zwei Formen sich nicht schneiden.

Temporale Kohärenz

Eine häufig genutzte Optimierungstechnik verwendet temporale Kohärenz. Wenn sich mit Collidern ausgestattete Spielobjekte mit sinnvoller Geschwindigkeit bewegen, kommt es häufiger vor, dass sich die Positionen und Orientierungen der Spielobjekte von Zeitschritt zu Zeitschritt nicht großartig ändern. Dies kann ausgenutzt werden, indem das erneute Ausrechnen bestimmter Informationen vermieden wird, indem die Ergebnisse der berechneten Informationen für mehrere Zeitschritte zwischengespeichert werden. In der Physik Engine Havok beispielsweise bleiben Kollisionsagenten über mehrere Frames erhalten, solange die Bewegung der mit Collidern ausgestatteten Objekte die bereits gemachten Berechnungen nicht ungültig gemacht hat.

Örtliche Partitionierung

Die örtliche Partitionierung dient dazu, die Anzahl der Objekte mit Collidern möglichst stark zu reduzieren, welche auf Kollisionen bzw. Überschneidungen untereinander überprüft werden müssen. Somit kann insgesamt die Zeit für die in jedem Zeitschritt ablaufenden Kollisionsberechnungen verbessert werden, weil weniger Berechnungen gemacht werden müssen. Dies geschieht dadurch, dass der virtuelle Raum/die Spielwelt in eine Menge von kleineren Regionen aufgeteilt wird. Deswegen müssen Paare von Objekten, welche nicht dieselbe Region teilen, nicht auf Überschneidungen überprüft werden. Es gibt verschiedene Verfahren für die Aufteilung des Raums, welche alle hierarchisch ablaufen.
Beispielsweise können zu diesem Zweck Octrees, binäre Raumpartitionierung (BSP-Bäume), k-d-Bäume oder auch Kugelbäume verwendet werden. Diese Baumstrukturen teilen den Raum beginnend von der Wurzel des Baumes immer weiter in Regionen auf, bis genug detaillierte Größen für die einzelnen Regionen erreicht werden. Durch Herablaufen in einem Baum für ein Paar von Objekten, was auf Überschneidungen überprüft werden soll, kann die Überschneidung der zwei Objekte ausgeschlossen werden, wenn die Objekte sich in unterschiedlichen Ästen des Baumes befinden. Weil der Baum den Raum unterteilt, können Objekte innerhalb eines Astes des Baumes nicht mit Objekten eines anderen Astes kollidieren.

GjK

Der GjK-Algorithmus stellt einen äußerst effizienten Algorithmus dar, um Überschneidungen zwischen beliebigen konvexen Polytopen A und B (konvexe Polygone im Zweidimensionalem, konvexe Polyeder im Dreidimensionalem) zu erkennen.

Der Algorithmus ist nach seinen Erfindern, E.G. Gilbert, D.W. Johnson und S.S. Keerthi benannt. In dem Algorithmus spielt eine geometrische Operation, die sogenannte Minkowski-Differenz eine entscheidende Rolle. Bei dieser Operation wird von jedem Punkt, der innerhalb der Form B liegt, paarweise jeder Punkt, der innerhalb der Form A liegt, subtrahiert. Daraus resultiert dann die Set-Menge an Punkten $\{(A_i - B_j)\}$, die Minkowski-Differenz. Enthält die Minkowski-Differenz den Ursprungspunkt des Koordinatensystems ((0,0,0) im dreidimensionalem Raum), dann schneiden sich die zwei Formen, vorausgesetzt, sie sind konvex.
Der GjK-Algorithmus versucht iterativ den Ursprungspunkt innerhalb der Hülle der Minkowski-Differenz zu finden. Dabei startet der Algorithmus mit einem beliebigen ein-Punkt-Simplex aus der Minkowski-Differenz Hülle. Ein 0-Simplex ist ein Punkt, ein 1-Simplex eine Strecke, ein 2-Simplex ein Dreieck, ein 3-Simplex ein Tetraeder und ein 4-Simplex ein Pentachoron (vierdimensionale Pyramide mit tetraederförmigen Basis).
In jedem Iterationsschritt wird zunächst festgestellt, in welcher Richtung der Ursprungspunkt in Relation zum Simplex liegt. Anschließend wird die konvexe Hülle der Minkowski-Differenz in dieser Richtung nach einem Punkt durchsucht, welcher am nächsten zum Ursprungspunkt liegt. Die konvexe Hülle einer Teilmenge ist die „kleinste konvexe Form, die die Ausgangsmenge enthält“ [29, Übersetzung übernommen von 30]. Dieser Punkt wird dann zum Simplex hinzugefügt, sodass ein Simplex höherer Ordnung entsteht, welcher mehrere Punkte umfasst. Der Algorithmus endet, falls entweder der Simplex den Ursprungspunkt umschließt oder kein Punkt in der Richtung gefunden werden kann, welcher näher am Ursprungspunkt liegt. Im ersten Fall schneiden sich die zwei konvexen Formen, im zweitem Fall nicht.

3.4 HTC Vive

Die HTC Vive ist ein Virtual Reality-Headset, welches im Lieferumfang das Headset, zwei Controller und zwei Infrarot-Sender-Einheiten, welche *Leuchttürme* genannt werden, beinhaltet. Das Headset selbst deckt einen nominalen Sichtbereich von in etwa 110° ab, wobei 90° des Sichtbereiches pro Auge abgedeckt werden und mit 90 Hz (90 Bilder pro Sekunde) aktualisiert werden können. Die Pixeldichte der Vive ist momentan noch nicht hoch genug, sodass leicht einzelne Pixel entdeckt werden können, wenn man darauf achtet. Mit der Vive geht ein Bewegungserfassungssystem einher, welches die Position und Ausrichtung des Headsets in Echtzeit verfolgt und berechnet. [vgl. 31]

3.4.1 Verwendetes Modell für Controller

Als Modell für die Controller der HTC Vive werden in der im Rahmen dieser Arbeit entwickelten Simulation zwei Schwerter verwendet, welche rot und blau gefärbt sind. Abbildung 3.3 zeigt die zwei Schwerter nebeneinanderliegend. Die Farbe der Schwerter zeigt an, welche farblichen Würfel (rot, blau) mit dem farblich passenden Schwert zerschlagen werden können. Zuerst wurde als visuelle Repräsentation der Controller ein einfaches Modell der menschlichen Hand verwendet. Abbildung 3.4 zeigt die verwendeten Modelle der Hände. Jedoch stellte sich bei den Tests des Prototypen heraus, dass man mit diesem Modell der Hand die Würfel nicht so präzise zerschlagen kann. Deswegen wurde entschieden, dass man mit zwei Schwertern die Würfel zerschlagen sollte. Die Modelle für die Hände und die Schwerter wurden mit dem Programm *Blender* [32] erstellt.

3.4.2 Funktionsweise der Technik

Das System der Vive baut auf Leuchttürmen auf, welche synchronisierte Lichtstrahlen im Infrarotbereich ausstrahlen, wobei pro Leuchtturm in ausgestrahlter Richtung 120° umfasst werden. Außerdem gibt es verschiedene Tracker-Geräte (Headset, Controller, etc.), auf deren Oberfläche Photodioden angebracht sind. Diese können den Zeitpunkt des Auftreffens des von den Leuchttürmen ausgesandten Lichtpulses messen, um so die horizontalen und vertikalen Winkel zu den Leuchttürmen abschätzen zu können. Dabei kann mithilfe der Zeitunterschiede, wann die verschiedenen Photodioden der Tracker von den Lichtstrahlen getroffen werden, die Position und Orientierung des Headsets wiederhergestellt werden. Jedoch wird die aktuell verfolgte Position und Orientierung des Headsets hauptsächlich über im Headset platzierte Trägheitsmaßeinheiten mittels Pfadintegration bestimmt. Dies gewährleistet deutlich höhere Aktualisierungsraten, aber die von den Trägheitsmaßeinheiten bestimmten Werte für Position und Orientierung können noch durch die langsameren Leuchtturm-Einheiten korrigiert werden. [vgl. 31, 33]

3.4.3 Steam VR

Damit die entwickelte VR-Anwendung in Unity korrekt funktioniert, muss auf dem Rechner Steam VR über die Software Steam installiert sein. Die Hardware der HTC Vive (Headset und zwei Controller) wird über das Steam VR-Plugin getrackt und gesteuert. Das Steam-VR-Plugin wird von dem verwendeten VR-Toolkit unterstützt.

3.4.4 VRTK-Kit für Unity

Für die Implementierung der Virtual Reality-Komponenten wurde das VRTK-Kit [34] genutzt. Das VRTK-Kit ermöglicht die einfache Entwicklung von VR-Anwendungen, weil es einige vorgefertigte VR-Prefab-Komponenten anbietet, welche direkt in einer neuen Szene genutzt werden können. Es bietet auch einen VR-Simulator, mit dem man die VR-Anwendung auch ohne VR-Hardware testen kann, was vor allem zu Beginn der Entwicklung eine entscheidende Rolle spielte. Das Bewegen der Controller zum Ausführen der Schlagbewegungen über die verschiedenen Tasten der Tastatur gestaltete sich aber dennoch recht schwierig. Über das Toolkit lassen sich zahlreiche Interaktionen in der VR-Umgebung umsetzen (wie beispielsweise das Greifen, Berühren oder Werfen von Objekten). Für die implementierte VR-Anwendung wurden jedoch hauptsächlich die Pointer-Interaktionen benötigt. Dies ist beispielsweise der Fall, wenn der Nutzer über ein Menü Einstellungen vornimmt oder beim Antizipationstest mittels Pointer Würfel spawnt. Das VRTK-Kit bietet als Pointer-Interaktionen das Bedienen von UI-Elementen auf einem Canvas mittels aktivierter Pointer der Controller an. Die Pointer können einerseits in eigenen Skripten aktiviert oder deaktiviert werden. Andererseits kann auch festgelegt werden, dass das Gedrückt-Halten des Touchpads den zugehörigen Pointer für die Dauer des Haltens aktiviert. An ein Canvas muss das Skript *VRTK_UI_Canvas* angehängt werden, damit es auf Eingaben des Nutzers über aktivierte Pointer reagiert. Für die beiden Controller kann im zugehörigem *VRTK_UI Pointer*-Skript beispielsweise eingestellt werden, dass man über das Drücken des Trigger-Knopfes UI-Elemente auswählt bzw. aktiviert.

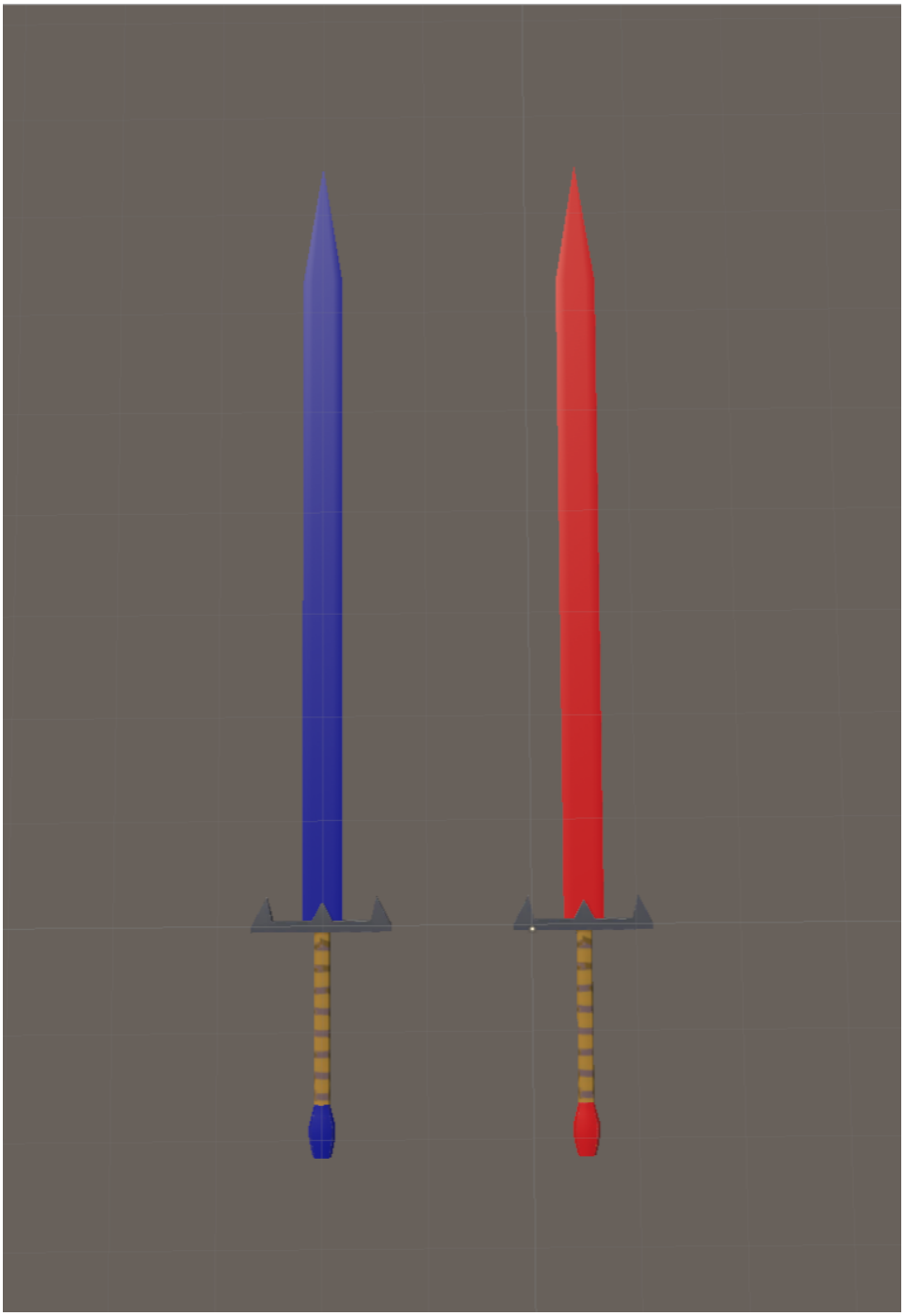


Abbildung 3.3: Modelle der Schwerter

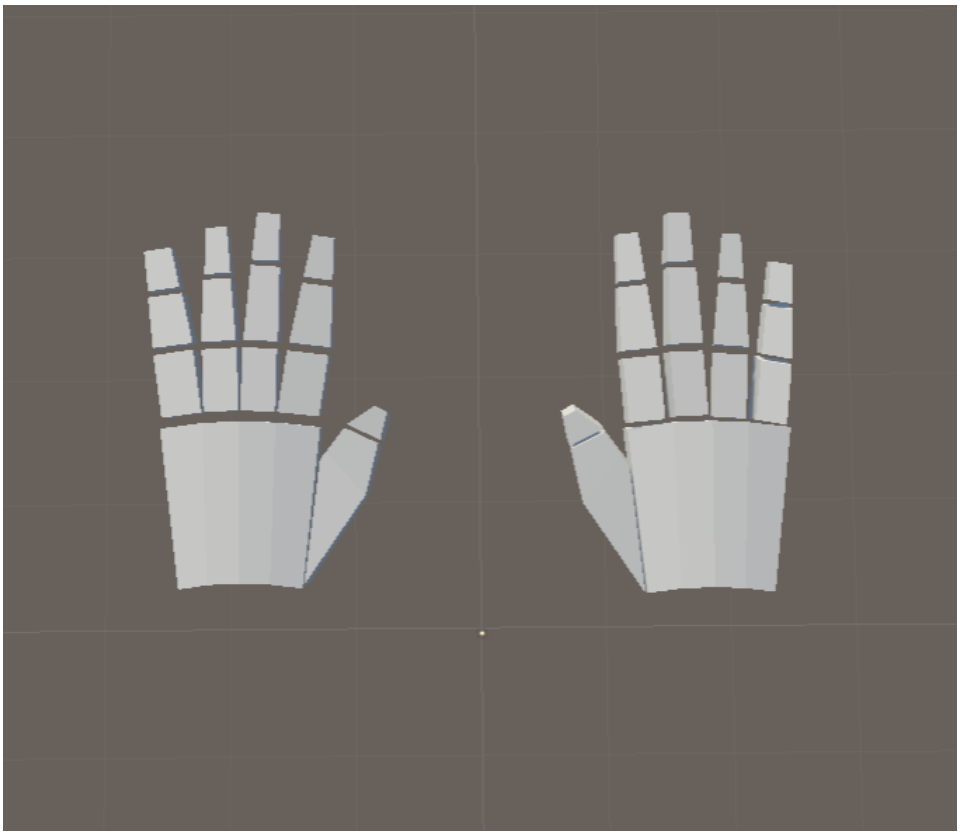


Abbildung 3.4: Modelle der Hände

4 Implementierung

Der Code der Masterarbeit wurde mittels C#-Skripten für die Unity-Umgebung umgesetzt. Die verschiedenen Skripte sind in verschiedenen Ordnern im Projekt strukturiert. Im Ordner „AnticipationTest“ sind alle zum Antizipationstest zugehörigen Skripte zu finden. Der Ordner „Data“ enthält alle Skripte zum Speichern, Lesen und Auswerten von Daten. Im Ordner „Debug“ befinden sich die leicht abgeänderten Versionen der Hauptskripte zum Analysieren und Testen von Funktionalitäten, insbesondere Treffer- und Präzisionserkennung. Alle für den Leveleditor notwendigen Skripte sind im Ordner „EditorScripts“ zu finden. Die 3 Hauptskripte der VR-Anwendung (SpawnCubes, HitInteractable und SpawnedInteractable) befinden sich im Ordner „Main“. Für die Managerskripte zum Verwalten von Schwierigkeitsgrad und Sound ist der Ordner „Manager“ angelegt. Im Ordner „Spawning“ befinden sich verschiedene Datenstrukturen, welche festlegen, welche Arten von Sequenzen vom SpawnSkript erzeugt werden können. Zu guter Letzt befinden sich im Ordner „UI“ alle Skripte, welche die verschiedenen HUD-Elemente anzeigen und steuern. Es existieren außerdem verschiedene Szenen, in denen verschiedene Skripte ausgeführt werden. So gibt es eine Hauptszene, eine Tutorial-Szene, eine Antizipationstestszene sowie eine Szene für die Debug-Skripte.

4.1 Hauptaufgabe / Spielprinzip

Die Hauptaufgabe dieser Arbeit bestand darin, eine VR-Anwendung zu entwickeln, in welcher man mit Pfeilen beschriftete Würfel zerschlägt. Die Pfeilrichtungen auf den Würfeln geben dabei an, in welcher Richtung man die Würfel zerschlagen muss. Das Zerschlagen geschieht dadurch, dass man die VR-Controller so bewegt, dass man mit den virtuellen Schwertern (als Modell für die Controller) einmal den kompletten Körper des Würfels durchschlägt. Es erscheint immer ein Würfelpaar, das es mit beiden Händen zu Zerschlagen gilt. So kann je ein Würfel nur mit dem passenden Controller zerschlagen werden, eine farbliche Kennzeichnung von Schwertern und Würfeln zeigt die Zuordnung an.
Sobald beide Würfel zerschlagen wurden oder eine festgelegte Zeitspanne verstrichen ist, erscheint das nächste Würfelpaar. Das Zerschlagen der Würfel soll nur dann als richtig anerkannt werden, wenn die vorgegebene Pfeilrichtung auf dem Würfel beachtet wurde. Wurde ein Würfel korrekt zerschlagen, so soll sowohl die Präzision als auch die verstrichene Zeit ermittelt werden, um mit diesen Daten zusammen dann den Schlag mit einer Punktzahl zu bewerten. Als Feedback wurden sowohl verschiedene Sounds für die erreichte Präzision beim Zerschlagen als auch ein HUD implementiert, welches die erreichte Punktzahl anzeigt und besonders hohe Punktzahlen farblich kennzeichnet.

So sammelt man mit jedem korrekt zerschlagenem Würfel Punkte und strebt an, eine möglichst hohe Gesamtpunktzahl, hier *Highscore* genannt, zu erreichen. Für die Positionen der erscheinenden Würfel musste sichergestellt werden, dass diese im Sichtfeld der Probanden erscheinen, so wurde einerseits die Position an die aktuelle Position der VR-Brille gekoppelt, als auch eine Funktion implementiert, welche die Positionen zum aktuell sichtbaren Quadranten des Spielers rotiert.
Da man mit der VR-Anwendung das sequentielle Lernen von Schlagbewegungen untersuchen will, müssen die erscheinenden Würfelpaare einem festen Muster entsprechen. So wird in der Hauptszene eine sich immer wiederholende Abfolge von Würfelpaaren, genannt *Sequenz*, wiederholt. Um die Eigenschaften (Position, Pfeilrichtung) der in einer Sequenz vorkommenden Würfel festlegen zu können, war es notwendig, einen Leveleditor zu implementieren, der eine erstellte Sequenz speichern kann. Für das Untersuchen des sequentiellen Lernens muss allerdings zwischen implizitem und explizitem Wissen bzw. Lernvorgang unterschieden werden. Zu diesem Zweck wurde ein Test, ein sogenannter *Antizipationstest*, implementiert, welcher untersucht, ob die Probanden explizites Wissen zu der Sequenz besitzen, aus der sie sich wiederholende Würfelpaare während des Tests zerschlagen haben. Wie gut die Probanden in der Hauptszene und im Antizipationstest abgeschnitten haben, muss gespeichert werden, damit man später diese Daten im Rahmen einer Studie auswerten kann (siehe Kapitel Ergebnisse und Diskussion).

4.2 Hauptskripte und Hauptassets

Die Hauptskripte und Hauptassets sind für das Funktionieren des wesentlichen Spielprinzips zuständig. So gibt es zunächst einmal verschiedene Assets für die erscheinenden Würfel, welche vom *Spawn-Skript* wiederholt erzeugt werden. An diese Würfel ist das *Spawned-Interactable-Skript* angehängt, welches dafür zuständig ist, dass die Schlagmechanik funktioniert und Präzision und Reaktionszeit ermittelt werden können. An die Controller (als Modell Schwerter) ist das *Hit-Interactable-Skript* angehängt. Dieses Skript ist auch dafür zuständig, dass die Schlagmechanik funktioniert und erkennt, zu welchem Zeitpunkt die Ausmaße der Schwerter die verschiedenen Collider eines Würfels betreten oder verlassen.
Das Spawn-Skript steuert das Erscheinen der Würfelpaare und prüft wiederholt, ob neue Würfel gespawnt werden müssen und erzeugt diese dann entsprechend. So können entweder beide Würfel bereits zerschlagen sein oder die maximale Zeit für das Zerschlagen der Würfel überschritten sein. Außerdem wird die Position der erscheinenden Würfel in Form von Kugelkoordinaten und einer Interpolationsfunktion von dem Skript kontrolliert. Die Eigenschaften (Pfeilrichtung, Position) der erscheinenden Würfel können über verschiedene Instanzen des Skripts BlockSequence festgelegt werden. Abbildung 4.1 (Schaubild der Hauptskripte) veranschaulicht, welchen Spielobjekten welche Skripte zugeordnet werden und die Interaktion der Elemente.

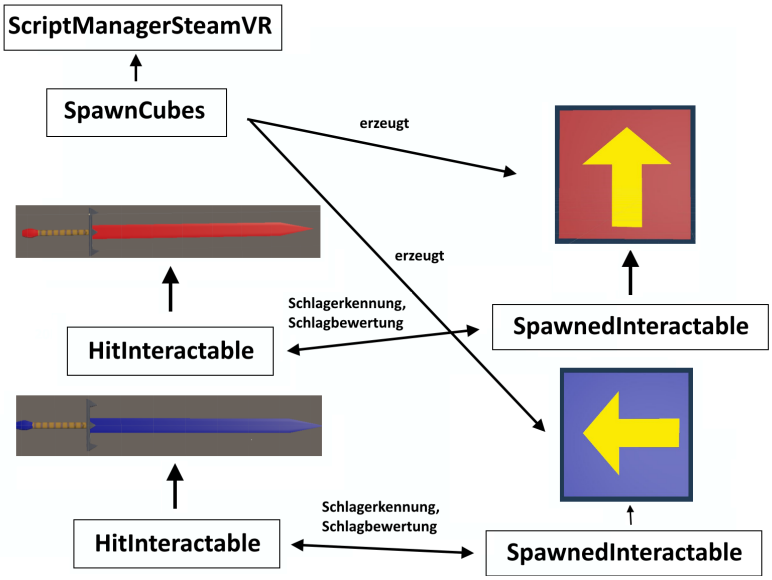


Abbildung 4.1: Schaubild der Hauptskripte

In den folgenden Unterkapiteln wird auf die wichtigsten Implementierungsdetails der Hauptskripte und Hauptassets eingegangen.

4.2.1 Würfel

Für die erscheinenden Würfel gibt es 4 vorgefertigte Elemente, sogenannte *Prefabs*, jeweils ein roter und blauer Würfel mit normalem Pfeil sowie ein roter und blauer Würfel mit diagonalem Pfeil. Prefabs sind beliebig komplexe Spielobjekte, von denen variabel viele Instanzen zur Laufzeit der VR-Anwendung instanziiert werden können. Um alle gewünschten Pfeilrichtungen erzeugen zu können, wird ausgenutzt, dass mithilfe der Rotation um die z-Achse des Würfels der Pfeil in alle Richtungen gedreht werden kann. Die folgenden 8 Pfeilrichtungen waren gewünscht:

- Pfeil nach links, Pfeil nach rechts, Pfeil nach oben, Pfeil nach unten
- Pfeil diagonal nach links oben, Pfeil diagonal nach rechts oben, Pfeil diagonal nach links unten, Pfeil diagonal nach rechts unten

Abbildung 4.2 zeigt alle verschiedenen Pfeilrichtungen für die roten und blauen Würfel. Jeder dieser Würfel-Prefabs besitzt eine Vielzahl an Collidern, welche für die Schlagerkennung eine entscheidende Rolle spielen. Die Idee ist hier, dass verschiedene Collider in einer bestimmten Reihenfolge durchlaufen werden müssen, damit der Schlag korrekt entlang der Pfeilrichtung gemacht worden ist. So besitzen die Würfel mit normalem Pfeil 3 Collidergruppen, die Würfel mit diagonalem Pfeil 5 Collidergruppen. Abbildung 4.3 zeigt den Aufbau dieser Collidergruppen für die 2 verschiedenen Würfel. Die Collidergruppen erscheinen bei dem Würfel mit diagonalem Pfeil um 45° nach rechts gedreht

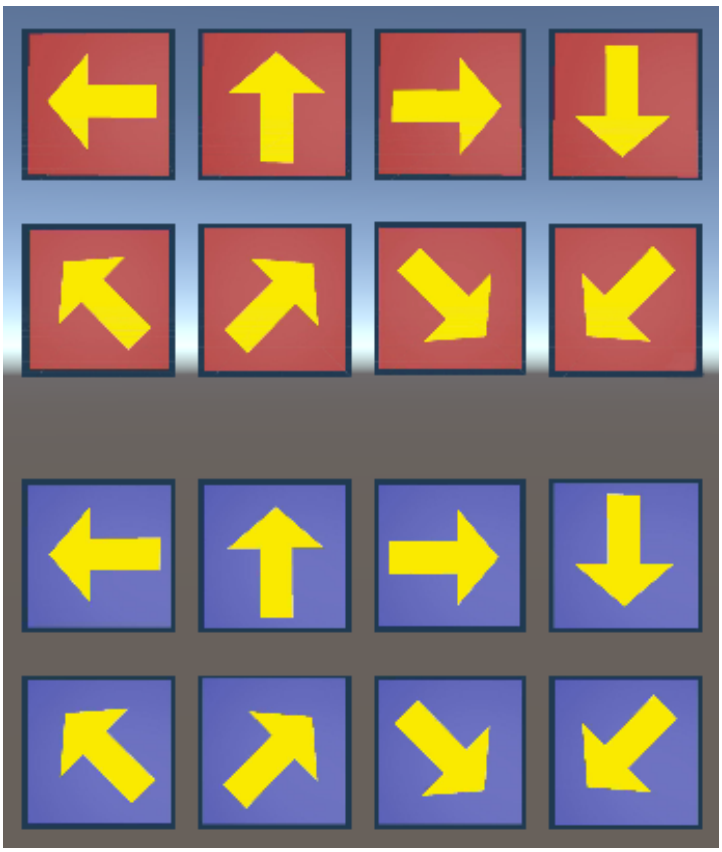
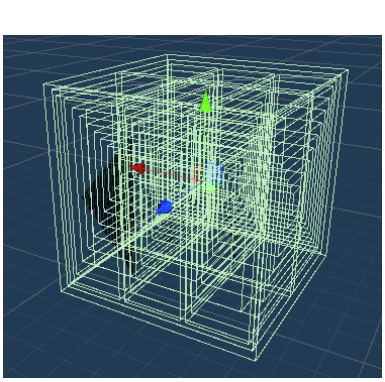
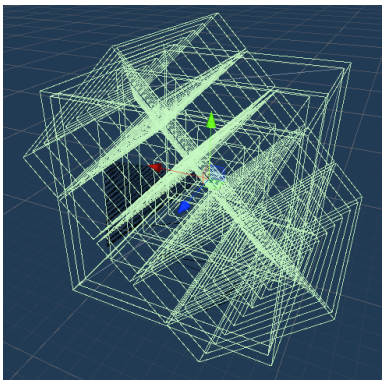


Abbildung 4.2: Schaubild aller möglichen Pfeile

zu sein. Dies liegt daran, dass auch der Pfeil in der Grundstellung verglichen mit dem normalem Pfeil um 45° nach rechts gedreht ist.



(a) Würfel mit normalem Pfeil



(b) Würfel mit diagonalem Pfeil

Abbildung 4.3: Aufbau der Collidergruppen

Die erste Collidergruppe befindet sich am Pfeilanfang, während die letzte Collidergruppe am Pfeilende vorzufinden ist. Jede Collidergruppe besitzt 10 Collider, welche für die Präzisionserkennung verwendet werden. Diese Präzisions-Collider nehmen in der Größe immer weiter ab, wobei der kleinste und letzte Präzisions-Collider mittig (sowohl in der Tiefe als auch entlang der Seite) im Würfel vorzufinden ist. Es wird der genaueste Präzisions-Collider gespeichert, welcher beim Schlag erwischt wurde. Die Gesamtpräzision wird dann als Durchschnitt aller Präzisionswerte aller Collidergruppen berechnet. Zusätzlich besitzt jeder Würfel auch noch einen Collider, welche die Form des Würfels nachstellt, damit erkannt werden kann, wann ein Schlag beginnt.
Allerdings wird ab großen Geschwindigkeiten das Betreten und Verlassen dieser Präzisions-Collider innerhalb der Collidergruppen nicht mehr zuverlässig von der Physik Engine erkannt, weshalb jeder Würfel noch einen zusätzlichen Physik-Collider besitzt. Dieser Collider besitzt einen Start- und einen Endpunkt, wobei sich der Startpunkt am Pfeilanfang und der Endpunkt am Pfeilende befindet. Außerdem ist dieser Physik-Collider in der Tiefe sehr schmal aufgebaut, damit die Winkelberechnung im Zweidimensionalem erfolgen kann. Mithilfe des Start und Endpunktes lässt sich ein Vektor der idealen Schlagrichtung bilden. Mithilfe des Eintritts- und Austrittspunktes in diesen Physik-Collider kann nun der Winkel zwischen dem idealen Schlagrichtungsvektor und dem tatsächlichen Schlagrichtungsvektor berechnet werden. Das heißt, dass bei schnellen Schlägen, bei denen die erste Erkennung nicht mehr zuverlässig funktioniert, auf diese Weise Schläge noch erkannt werden können.
Allerdings hat diese Methode auch ihre Probleme in der Hinsicht, dass Eintritts- und Austrittspunkt von der Physikengine zu ungenau ermittelt sein können, sodass in seltenen Fällen eigentlich inkorrekte Schläge als korrekt und korrekte Schläge als nicht korrekt bewertet werden, weil der Schlagrichtungsvektor durch die ungenauen Positionsdaten von Eintritts- und Austrittspunkt zu stark vom Idealvektor abweicht.

4.2.2 Spawn-Skript

Das *Spawn-Cubes-Skript* zählt mit zu den wichtigsten Skripten und kontrolliert das regelmäßige Erscheinen der im Spiel auftauchenden Würfel. In der Datei wird auch noch als Datenstruktur die Klasse *sphereCoordinates* definiert, welche für das Serialisieren der Daten (Speichern und Lesen der Daten) verwendet wird. Normalerweise erwarten die Spawn-Funktionen innerhalb des Skriptes Kugelkoordinaten, welche in Form der Kugelkoordinaten *phi und theta* für jeden Würfel eines Würfelpaares übergeben werden. Die Kugelkoordinaten werden verwendet, damit die Würfel auf einer Sphäre um den Spieler herum gespawnt werden können. Der Spieler bzw. die Position der VR-Brille definiert dabei das Zentrum dieser Sphäre. Durch die Verwendung dieser relativen Koordinaten wirkt sich die Position des Spielers nicht auf die Position der gespawnten Würfel aus. Für die berechnete Position der zu spawnenden Würfel wird auch stets die aktuelle Position des Spielers miteinbezogen.

```
public class SphereCoordinates
{
  public float phi;
  public float theta;
  public float phi2;
  public float theta2;
  public float radius;
  public float radius2;
  public float rotationZ;
  public float rotationZ2;
  public Vector3 scale;
  public Vector3 scale2;
  public SphereCoordinates(float p, float t, float p2
    , float t2)
  {
    phi = p;
    theta = t;
    phi2 = p2;
    theta2 = t2;
  }
  public SphereCoordinates()
  {
    phi = 0.0f;
    theta = 0.0f;
    phi2 = 0.0f;
    theta2 = 0.0f;
  }
}
```

Für das Skript können eine Reihe von Variablen konfiguriert werden, folgender Auschnitt zeigt einen Auschnitt der Variablen des Skriptes:

```
public float sphereRadius;
public float timeToHitGameObjects;
public float timeToWaitBetween;
public float scaleSpawnedGameObjects = 1.0f;
public bool animatedSpawning = true;
public bool useScaleFromSphereCoordinates = false;
public bool turnLoadedSequenceTowardsPlayer = false;
public bool playBackgroundMusic = true;
public bool noAutomatedSpawning = false;
public bool makeBreakBetweenBlocks = true;
public bool p_key_to_pause_game = false;
public int breakTimeInSeconds = 30;
public BlockSequence[] blockSequences;
```

So legt *timeToHitGameObjects* fest, wie viel Zeit man hat, um die erscheinenden Würfel zu zerschlagen. Der Wert dieser Variable wird zu Spielbeginn allerdings vom aktiv verwendeten Schwierigkeitsgrad festgelegt. Die Größe der erscheinenden Würfel wird über das Attribut *scaleSpawnedGameObjects* gesteuert. Auch der Wert dieser Variable

wird zu Spielbeginn vom aktiv verwendeten Schwierigkeitsgrad festgelegt. Die Variable *timeToWaitBetween* legt fest, wie viele Sekunden gewartet werden soll, bis das nächste Würfelpaar gespawnt wird. Standardmäßig ist diese Variable auf 1.0 gesetzt und ermöglicht so eine kurze Pause nach dem Zerschlagen jedes Würfelpaares. Es folgen eine Reihe von einstellbaren boolschen Variablen:

- *animated Spawning* legt fest, ob die erscheinenden Würfel vom zentralem Sichtpunkt des Spielers zum Spieler hinfliegen sollen und ist in der Voreinstellung deaktiviert.
- *useScaleFromSphereCoordinates* legt fest, ob die in der Datenstruktur spezifizierte Scale für die Würfel die globale Größe der Würfel für das Spawnen der Würfel überschreiben soll und ist in der Standardeinstellung deaktiviert.
- *turnLoadedSequenceTowardsPlayer* legt fest, ob die erscheinenden Würfel in das Sichtfeld des Spielers hingedreht werden sollen, sodass sich der Spieler nicht drehen muss, um die Wüfel zu sehen und ist in der Voreinstellung aktiviert.
- *playBackgroundMusic* legt fest, ob Hintergrundmusik gespielt werden soll. Es wird kontinuiertlich ein zufälliges Lied aus einer Auswahl von drei Liedern abgespielt und ist in der Standardeinstellung aktiviert.
- *noAutomatedSpawning* legt fest, ob sich das Skript passiv verhalten soll und somit selbstständig keine Würfel spawnt und ist in der Voreinstellung deaktiviert.
- *makeBreakBetweenBlocks* legt fest, ob zwischen 2 aufeinanderfolgenden Blöcken (ein Block enthält eine Anzahl von bestimmten Wiederholungen einer Sequenz von Würfelpaaren) eine Pause erfolgen soll und ist in der Standardeinstellung aktiviert.
- *p_key_to_pause_game* legt fest, ob der Studienleiter das Spiel durch das Drücken der -P-Taste pausieren kann. Ist diese Option deaktiviert, so wird das Spiel immer dann pausiert, wenn das Touchpad von einem der zwei Controller berührt wird. Diese Variable ist in der Voreinstellung aktiviert, damit Probanden während der Studie nicht versehentlich das Spiel pausieren können.

Die Variable *breakTimeInSeconds* spezifiert wie viele Sekunden gewartet wird, wenn ein Block endet, bevor die ersten Würfel des nächsten Blockes erscheinen. Das Array *blockSequences* enthält alle Blöcke, welche vom Skript abgearbeitet werden sollen. Die Felder müssen per Referenz im UnityEditor spezifiziert werden. Hierbei sollte jedes BlockElement eine oder mehrere Sequenzen an Würfeln enthalten. Die verschiedenen BlockElemente, welche dem Array *blockSequences* hinzugefügt werden können, werden genauer im Kapitel *Implementierte Sequenzen* beschrieben.
Die Länge des Arrays kann beliebig festgelegt werden. Es muss beachtet werden, dass jedes BlockSequence-Element innerhalb des Arrrays eine einzigartige Referenz aufweisen

muss, d.h. es sollte nicht dasselbe BlockSequence-Element im Array mehrmals vorkommen. Wird eine Referenz mehrfach verwendet, so wird das Element beim zweiten Mal nicht mehr abgearbeitet werden, weil es vom Spawn-Skript schon als abgearbeitet angesehen wird. Außerdem werden für jedes abgearbeitete BlockSequence-Element Daten über das Abschneiden in dem Block gespeichert z.B. beste oder durchschnittliche Punktzahl für die Sequenz, welche nach Abschluss der Hauptszene in eine BlockDataX.xml-Datei geschrieben werden. Wurden die letzten Würfel des letzten Blockes gespawnt, verfällt das Skript in einen inaktiven Status. Außerdem wird davor veranlasst, dass alle angesammelten Daten in die zwei Dateien *sportgamedataX.xml* und *blockDataX.xml* geschrieben werden.
Die Funktion *spawnCubessForBothHandsInSightOfCameraDirection* wird wiederholt vom Skript genutzt, um ein Würfelpaar zu spawnen. Sind keine Blöcke als Referenzparameter in der Unityszene spezifiziert worden, so werden zufällige Würfelpaare gespawnt.
Die zufällig erzeugten Würfel erscheinen immer im Sichtfeld des Spielers. Ansonsten werden die Würfel mithilfe der Kugelkoordinaten des nächsten SphereCoordinates-Elements innerhalb des aktuellen Blockes erzeugt. Damit die Würfel garantiert im Sichtfeld des Spielers erscheinen, muss *turnLoadedSequenceTowardsPlayyer* auf *true* gesetzt sein. Die interne Variable *instantiated* dient zur Kontrolle, dass ein Würfelpaar gespawnt wurde. Es wird mit jedem Abrufen der Update-Funktion des Skripts überpüft, ob das bereits gespawnte Würfelpaar noch existiert oder beide Würfel zerschlagen wurden oder die verfügbare Zeit zum Zerschlagen abgelaufen ist. In diesem Fall wird die Variable wieder auf *false* gesetzt.

```
1  private void
       spawnCubesForBothHandsInSightOfCameraDirection()
2  {
3    if (blockSequences == null || blockSequences.Length
       == 0) // no list of spawn points => randomized
      spawning of cubes
4    {
5      spawnRandomCubesForBothHands();
6      instantiated = true;
7    }
8    else   // spawning of elements in the current
      blockelement from blockarray ( => blockArray[
      blockCurrentIndex].getNextSphereCoordinates())
9    {
10     if (blockSequenceIndex >= blockSequences.Length)
      return;
11     if (!blockSequences[blockSequenceIndex].
      hasNextSphereCoordinates())
12     {
13       breakTimer = 0f;
14       if (blockSequences[blockSequenceIndex].
      waitTimeAfterCompletion > 0) actBreakTime =
      blockSequences[blockSequenceIndex].
      waitTimeAfterCompletion;
15       else actBreakTime = breakTimeInSeconds;
16       ++blockSequenceIndex;
```

```
17        return;
18      }
19      if (blockSequenceIndex >= blockSequences.Length)
        return;
20      SphereCoordinates sc = blockSequences[
        blockSequenceIndex].nextSphereCoordinates();
21      spawnCubesForSphereCoordinates(sc);
22      instantiated = true;
23    }
24  }
```

Die Interpolationsfunktion *calculatePhiOffset* berechnet ein Offset für den Phi-Wert der Kugelkoordinaten abhängig von der aktuellen Blickrichtung der Kamera (VR-Brille). Die aktuelle Blickrichtung kann über den *forward-Vektor* der Transform des HMD-Headsets eingesehen werden. Je nach Blickrichtung der Kamera (360°-Sichtfeld) erscheinen nur für bestimmte Werte von phi die Würfel im aktuellem Sichtfeld. Die Funktion dreht die Positionen in den aktuell sichtbaren Quadranten des Sichtfelds der Kamera, indem der neue Phi-Wert aus dem Original-Phi-Wert und dem Phi-Offset berechnet wird. So wird ein Phi-Offset berechnet, welches addiert mit dem aktuellem Phi-Wert sicher stellt, dass die Position der zu spawnenden Würfel im Sichtfeld der Testperson liegen.

```
1  public void spawnCubesForSphereCoordinates(
       SphereCoordinates sc)
2  {
3    ...
4    int phiOffset = calculatePhiOffset();
5    if (!turnLoadedSequenceTowardsPlayer) phiOffset =
       0;
6    ...
7    degLeftPhi = sc.phi2 + phiOffset;
8    ...
9    degRightPhi = sc.phi + phiOffset;
10 }
11 private int calculatePhiOffset() // Interpolation of
       Phioffset for current Camera Direction
12 {
13   Vector3 forward = hmd_transform.forward;
14   if (forward.x >= 0.99) return 0;
15   if (forward.z >= 0.99) return 90;
16   if (forward.x <= -0.99) return 180;
17   if (forward.z <= -0.99) return 270;
18   if (forward.x > 0 && forward.z > 0)
19   {
20     if(forward.z < 0.7) return (int) (forward.z * 45/
       0.7);
21     else                return (int) (45 + ((forward.
       z - 0.7) * 45 / 0.3));
22   }
23   if (forward.x < 0 && forward.z > 0)
24   {
25     if(forward.x > -0.7) return (int) (90 + (forward.
```

```
    x * -45 / 0.7));
26      else                    return (int) (135 + ((
    forward.x + 0.7) * -45 / 0.3));
27    }
28    if (forward.x < 0 && forward.z < 0)
29    {
30      if(forward.z > -0.7) return (int)(180 + (forward.
    z * -45 / 0.7));
31      else                    return (int)(225 + ((forward
    .z + 0.7) * -45 / 0.3));
32    }
33    if (forward.x > 0 && forward.z < 0)
34    {
35      if(forward.x < 0.7) return (int)(270 + (forward.x
      * 45 / 0.7));
36      else                  return (int)(305 + ((forward.
    x - 0.7) * 45 / 0.3));
37    }
38    else return 0;
39  }
```

Der Hauptloop (entspricht der Update-Funktion) wird nur dann ausgeführt, wenn das Spiel aktuell nicht pausiert. Falls die Option ***p_key_to_pause_game*** aktiviert ist, wird, falls die P-Taste aktuell gedrückt ist, das Spiel pausiert und ein Menü zum Einstellen des Schwierigkeitsgrades aufgerufen. Falls mindestens ein Block als Referenz im Array spezifiziert wurde und der letzte Block im Block-Array abgearbeit wurde, werden die von den Controllern (HitInteractables) gesammelten Daten geschrieben.
Des Weiteren findet sich Code zum Aktivieren eines Pausemenüs (*breakWindow.enableWindow*), welches während der Pausen zwischen den Blöcken aktiv ist. Darüber hinaus findet sich Code zum Spawnen des nächsten Würfelpaares (*spawnCubesForBothHandsInSightOfCameraDirection*) sowie auch dem Beseitigen des aktuellen Würfelpaares, wenn die maximale Reaktionszeit überschritten wurde. Die Würfelpaare werden zeitverzögert gelöscht, damit die Zerstörungsanimation (Wegfliegen der Würfel) im Spiel noch betrachtet werden kann. Auch finden sich noch einige Initialisierungen für interne Variablen wie Zeit-Counter-Variablen, welche sicherstellen, dass bestimmte Codeteile im richtigem, gewünschtem Moment ausgeführt werden.

```
1   void Update()
2   {
3     if (noAutomatedSpawning || fileWritten ||
      DifficultyManager.Instance == null ||
      DifficultyManager.Instance.gamePaused) return;
4
5     if(Input.GetKeyDown(KeyCode.P) &&
      p_key_to_pause_game)
6     {
7       difficultySettingsMenu.enableCanvas();
8       difficultySettingsMenu.pauseGame();
9       return;
10    }
```

```
11
12      if (blockSequences != null && blockSequences.Length
          > 0 && blockSequenceIndex >= blockSequences.
        Length && !fileWritten) // check if there is a
        valid blockSequenceArrray and all elements of the
        block array
13      // were allready spawned, then write the gathered
        sportgamedata to file
14      // from now on there will nothing happen here
        anymore
15      {
16        fileWritten = true;
17        serializeSportGameData();
18        serializeBlockGameData();
19      }
20
21      breakTimer += Time.deltaTime;
22
23      if (breakTimer <= actBreakTime) // activate
        breakWindow or update the remaining break time on
        the already active break window
24      {
25        breakWindow.enableWindow();
26        if (blockSequenceIndex >= blockSequences.Length)
        breakWindow.messageGameFinished();
27        else breakWindow.updateRemainingTime(actBreakTime
          - breakTimer);
28        return;
29      }
30      else breakWindow.disableWindow();
31
32      float animationTimeBonus = animatedSpawning ? 2.0f
        : 0.0f; // if the option animated spawning is
        active the cubes need exactly 2 seconds to fly to
        their destination were they can be hit
33      // so this has be taken into calculation when
        measuring the time a player needed or has to hit
        the cubes
34
35      if(instantiated && bothCubesDestroyedOrHit() )
36      // if there is currently one instantiated cube pair
        , then check if the references to the cubes are
        already destroyed or if points were rewarded for
        hitting the cubes
37      // so the timer is set over the limit time to hit
        the cubes, so that the spawning of the next cube
        pair can immediatly start in this call of update
38      {
39        timeCounter = timeToHitGameObjects +
        animationTimeBonus;
40      }
41
```

```
42    timeCounter += Time.deltaTime;
43    instantiateTimeCounter += Time.deltaTime;
44
45    if(instantiateTimeCounter >= timeToWaitBetween && !
       instantiated) // check if instantiation time has
       passed and the current cube pair was not already
       instantiated
46    {
47      spawnCubesForBothHandsInSightOfCameraDirection();
       // spawns one cube pair
48    }
49
50    if(timeCounter >= timeToHitGameObjects +
       animationTimeBonus) // Clean up remaining not hit
       cubes, if the time to hit the cubes passed, then
       reinitialise relevant variables
51    {
52      float destroyTimeLeft = 0;
53      float destroyTimeRight = 0;
54      if (lastLeftHandTarget == null ||
       lastLeftHandTarget.getPointsRewarded())
       destroyTimeLeft = 5.0f;
55      if (lastRightHandTarget == null ||
       lastRightHandTarget.getPointsRewarded())
       destroyTimeRight = 5.0f;
56
57      if(lastLeftHandTarget != null) Object.Destroy(
       lastLeftHandTarget.gameObject, destroyTimeLeft);
58      if(lastRightHandTarget != null) Object.Destroy(
       lastRightHandTarget.gameObject, destroyTimeRight);
59      timeCounter = -timeToWaitBetween;   // when
       timeToWaitBetween has passed for spawning the next
       pair, the timeCounter will be exactly zero
60      instantiateTimeCounter = 0;
61      instantiated = false;
62    }
63  }
```

4.2.3 Spawned-Interactable-Skript

Eine Instanz des *Spawned-Interactable-Skript* ist an jeden erzeugten Würfel angehängt. Die Hauptaufgabe dieses Skriptes liegt darin, über interne Zustände zu speichern, welche Präzisions-Collider bei einem Schlag durch den Würfel betreten wurden, um somit die Präzision für den Schlag berechnen zu können. Außerdem wird geprüft, ob die verschiedenen Collider-Gruppen in der erwarteten Reihenfolge beim Schlag durchlaufen wurden. Das Skript besitzt eine große Anzahl an Variablen, die meisten werden jedoch nur für eine Zerstörungsanimation benötigt, weshalb hier nur auf die wichtigsten eingegangen wird. Ein Teil der Variablen wird bereits beim Spawnen eines Würfels im Spawn-Cubes-Skript initialisiert:

```
public void spawnGameObject(bool leftHand, Vector3
    position, Vector3 rotationVector, Vector3
    scaleVector, GameObject objectToSpawn)
{
  ...
  SpawnedInteractable si = gameObject.GetComponent<
   SpawnedInteractable>();
  si.setMovedOffset(normalizedOffset);
  si.setTimeToHitObjects(timeToHitGameObjects);
  si.roundGenerated = roundGenerated;
  si.setHmd_transform(hmd_transform);
  si.lookAt();
  listOfSpawnedCubes.Add(si);
  ...
}
```

So wird die Reaktionszeit zum Zerschlagen des Würfels gesetzt, damit die verbleibende Zeit innerhalb jedes Würfels heruntergezählt werden kann. Die Runde, in welcher der Würfel generiert wurde, wird auch gesetzt. Diese wird benötigt, um bei den geschriebenen Daten herausfinden zu können, ob bestimmte Würfel verpasst wurden (d.h. kein Schlagversuch wurde innerhalb der maximalen Reaktionszeit registriert). Außerdem wird die Transform des HMD-Headsets gesetzt. Diese wird benötigt, um den Würfel mittels der Funktion *lookAt* auf die Kamera, also den Spieler ausrichten zu können. Das normalisierte Offset wird nur für die Funktionalität des animierten „Hereinfliegens“ benötigt; ist dies deaktiviert, ist das normalisierte Offset immer der Nullvektor.

Nachdem ein Würfel korrekt zerschlagen wurde, muss das Feedback dem Spieler signalisieren, dass er den Würfel korrekt zerschlagen hat. Zu diesem Zweck sind zwei verschiedene Zerstörungsanimationen implementiert. Eine dieser zwei Animationen wurde allerdings für die Studie deaktiviert, weil sie für zu viel Unübersichtlichkeit in der Szene sorgte. Sie lässt den durchschlagenen Würfel in viele kleinere Würfel zerspringen, die in alle Richtungen vor den Augen des Spielers wegfliegen.
Die andere verwendete Animation schleudert den Würfel von dem Spieler weg. Die Kraft, mit der der Würfel weggeschleudert wird, hängt dabei von der Präzision ab, wie gut der Würfel durchschlagen wurde. Da jeder Würfel eine *Rigidbody-Komponente* (Starrer-Körper-Komponente) besitzt, kann man eine Kraft auf diese Komponente einwirken. Der übergebene PositionsVektor, mit dem die Richtung der Kraft berechnet wird, ist der initiale Eintrittspunkt eines der Schwerter in dem Würfel. So kann die Richtung als Differenz aus dem Zentrumpunkts des Würfelobjektes und dem initialem Eintrittspunkt berechnet werden. Initial ist für jeden Würfel die Schwerkraft in der Rigidbody-Komponente deaktiviert, weil sonst die erscheinenden Würfel auf den Boden herunterfallen würden. Diese Eigenschaft wird hier erst aktiviert:

```
public void addForceToRigidBody(Vector3
    positionVector, int precision)
{
  rigidBody.isKinematic = false;
  rigidbodyShouldBeMoving = true;
  Vector3 direction = rigidBody.transform.position -
   positionVector;
  forceDirection = direction;
  forcePrecision = precision;
  rigidBody.AddForceAtPosition(direction * 10.0f *
   precision, transform.position);
  rigidBody.useGravity = true;
}
```

Der Code für die Schlagerkennung speichert verschiedene Zustände bezüglich der Reihenfolge, in der die verschiedenen Collidergruppen betreten wurden.
Das Betreten der ersten Collidergruppe wird nur dann als gültig erkannt, wenn die anderen Collidergruppen noch nicht betreten wurden. Das Betreten der nachfolgenden Collidergruppen wird nur dann als gültig erkannt, wenn alle vorangegangen Collidergruppen bereits betreten wurden. Daraus folgt, dass bei einem Schlag in der entgegengesetzten Richtung nur die erste Collidergruppe als betreten vom Skript markiert wurde.
Damit alle vorhandenen Collidergruppen als betreten im Skript markiert sind, müssen die Colldidergruppen des Würfels beim Schlag in der richtigen Reihenfolge durchlaufen worden sein. Allerdings lässt diese Implementierung Schläge zu, die von der entgegengesetzten Richtung den Schlag beginnen, dann aber kurz vor dem Ende des Würfels nochmal umdrehen und den Würfel korrekt zerschlagen. Andererseits sind solche Schläge in der Praxis nicht allzu oft zu erwarten, weil sie ineffizient sind und extra Zeit für das Zerschlagen des Würfels benötigen. Bei der Durchführung der Studie konnten ein paar Probanden durch diese Immplementierung falsch angesetzte Schläge noch retten, indem sie die falsche Ansatzseite während des Schlags realisierten. Da manche Würfel 3 Collidergruppen, andere aber 5 besitzen, muss die Implementierung für beide Würfelarten funktionieren. Dies wird über die Eigenschaft *uses5ColliderGroups* abgefragt:

```
public void startColliderGroupHit(string colliderName
    )
{
  if (!middleColliderHit && !endColliderHit)
  {
    startColliderHit = true;
    assignAccuracyFor(colliderName, ref
   actAccuracyStart);
  }
}
public void middleColliderGroupHit(string
    colliderName)
{
  if (startColliderHit && !endColliderHit)
  {
    middleColliderHit = true;
```

```
14      assignAccuracyFor(colliderName, ref
        actAccuracyMiddle);
15    }
16  }
17
18  public void middleColliderGroupHit2(string
        colliderName)
19  {
20    if (startColliderHit && middleColliderHit &&  !
        endColliderHit)
21    {
22      middleColliderHit2 = true;
23      assignAccuracyFor(colliderName, ref
        actAccuracyMiddle2);
24    }
25  }
26
27  public void middleColliderGroupHit3(string
        colliderName)
28  {
29    if (startColliderHit && middleColliderHit &&
        middleColliderHit2 && !endColliderHit)
30    {
31      middleColliderHit3 = true;
32      assignAccuracyFor(colliderName, ref
        actAccuracyMiddle3);
33    }
34  }
35
36  public void endColliderGroupHit(string colliderName)
37  {
38    if (( startColliderHit && middleColliderHit && !
        uses5ColliderGroups) || (startColliderHit &&
        middleColliderHit && middleColliderHit2 &&
        middleColliderHit3) )
39    {
40      endColliderHit = true;
41      assignAccuracyFor(colliderName, ref
        actAccuracyEnd);
42    }
43  }
44
45  public bool wasHitInRightDirection()
46  {
47    if(uses5ColliderGroups) return startColliderHit &&
        middleColliderHit && middleColliderHit2 &&
        middleColliderHit3 && endColliderHit;
48    else                    return startColliderHit &&
        middleColliderHit && endColliderHit;
49  }
```

Die Präzisionsgenauigkeit innerhalb der Collidergruppen wird über die Namensgebung der verschiedenen im Würfel vorkommenden Präzisions-Collider realisiert. So ist die Namensgebung so aufgebaut, dass der größte, also ungenaueste Präzisions-Collider mit „One“ endet, während der kleinste, also genaueste Präzisions-Collider mit „Ten“ endet. Wird ein Präzisions-Collider einer beliebigen Präzisionsstufe durchlaufen, so wird die Präzision für die aktuelle Collidergruppe nur dann aktualisiert, wenn sie besser als der vorherige Wert für die Präzision der Collidergruppe ist.

```
1  private void assignAccuracyFor(string colliderName,
       ref int accAttribute)
2  {
3    assignConditionalAcc(colliderName, "One", ref
       accAttribute, 1);
4    assignConditionalAcc(colliderName, "Two", ref
       accAttribute, 2);
5    assignConditionalAcc(colliderName, "Three", ref
       accAttribute, 3);
6    assignConditionalAcc(colliderName, "Four", ref
       accAttribute, 4);
7    assignConditionalAcc(colliderName, "Five", ref
       accAttribute, 5);
8    assignConditionalAcc(colliderName, "Six", ref
       accAttribute, 6);
9    assignConditionalAcc(colliderName, "Seven", ref
       accAttribute, 7);
10   assignConditionalAcc(colliderName, "Eight", ref
       accAttribute, 8);
11   assignConditionalAcc(colliderName, "Nine", ref
       accAttribute, 9);
12   assignConditionalAcc(colliderName, "Ten", ref
       accAttribute, 10);
13 }
14
15 private void assignConditionalAcc(string colliderName
       , string ending, ref int accuracy, int val)
16 {
17   if (colliderName.EndsWith(ending))
       assignIfGreaterThanActAccuracy(ref accuracy, val);
18 }
19
20 private void assignIfGreaterThanActAccuracy(ref int
       acc, int newVal)
21 {
22   if (newVal > acc) acc = newVal;
23 }
```

4.2.4 Hit-Interactable-Skript

Eine Instanz des *Hit-Interactable-Skriptes* ist jedem der zwei Steam-VR-Controller zugeordnet. Dieses Skript hat als Hauptaufgabe, die Schlagmechaniken zu steuern sowie Daten (benötigte Zeit, Präzision, Punktzahl) zu gemachten Schlägen zu sammeln, sodass diese später in die Datei (sportGameDataX.xml) geschrieben werden können. Zwei wichtige Einstellungen können in diesem Skript vorgenommen werden:

```
1 public bool oneTryForHittingCubesCorrectly = true;
2 public bool canOnlyHitMatchingCube = false;
```

1. *oneTryForHittingsCubesCorrectly* legt fest, ob die Würfel nach einem fehlerhaften Schlag mittels einer Fade-Out-Animation verschwinden oder ob sie bestehen bleiben (hier werden alle internen Variablen innerhalb des Spawned-Interactalbe-Skriptes nach einem gemachten Schlag zurückgesetzt).
   Standardmäßig ist eingestellt, dass man nur einen Versuch hat die Würfel korrekt zu zerschlagen.

2. *canOnlyHitMatchingCube* bestimmt, ob man mit dem aktuellen Controller (im Spiel ein Schwert) nur die farblich passenden Würfel zerschlagen kann; d.h. die Farbe des Schwertes und des Würfels müssen übereinstimmen. Versucht man dennoch einen farblich unpassenden Würfel zu zerschlagen, so erklingt daraufhin ein „Glockengeräusch“, dass auf die Missachtung dieser Bedingung hindeutet. Hier verschwindet der Würfel allerdings nicht. Der Schlag durch den unpassenden Würfel wird unabhängig von allen Einstellungen nicht gewertet.
   Standardmäßig ist diese Einstellung aktiviert, weil man will, dass der Spieler mit beiden Armen Schlagbewegungen durchführt.

In diesem Skript kommen sowohl OnTriggerEnter, OnTriggerExit als auch OnCollisionEnter, OnCollisionStay und OnCollisionExit-Funktionen vor. Diese Funktionen werden immer dann aufgerufen, wenn das Modell des Controllers (ein Schwert) einen Collider in der Spieltwelt betreten bzw. verlassen hat. Die OnCollision-Funktionen werden verwendet, um schnellere Schläge erkennen zu können. Hier wird der Eintritts- und Austrittspunkt in einen speziellen, in der Tiefe schmalen, Physik-Collider als Kontaktpunkt gespeichert, um dann hiermit den Vektor bestehend aus Eintritts- und Austrittspunkt mit dem Idealvektor durch den Würfel entlang des vorgegebenen Pfeiles vergleichen zu können. Der Physik-Collider ähnelt stark dem Collider, der die Ausmaße des Würfels nachbildet, ist aber eine schmale Scheibe, damit die Winkelberechnung im Zweidimensionalem mit dem Idealvektor erfolgen kann. Außerdem ist die Tiefeninformation nicht relevant dafür, ob ein Schlag korrekt entlang der Pfeilrichtung gemacht worden ist.

Im Dreidimesionalem kann die Tiefeninformation den Winkel zwischen den zwei betrachteten Vektoren verzerren. Auch die Präzisions-Collider sind als Physik-Collider umgesetzt (Kollision zwischen zwei kinematischen Starren-Körper-Objekten). Der Vorteil der Physik-Collider im Vergleich zu den Trigger-Collidern liegt darin, dass Informationen zu

Kontaktinformationen der beiden sich schneidenden Collider ermittelt werden können. Zusätzlich kann durch die Verwendung von Physik-Collidern das Auftreten von Tunnelungseffekten vermieden bzw. verringert werden.

Beim Tunnelungseffekt verpasst die Physik-Engine zu registrieren, dass sich zwei Collider geschnitten haben, weil sich beispielsweise eins der zwei Objekte mit Collidern zu schnell bewegt hat. Normalerweise sorgt die Physik-Engine dafür, dass zwei Physik-Collider sich nicht ineinander verschieben können, sondern stattdessen voneinander abprallen. Da aber ein Schlag durch die Würfel ermöglicht werden soll, müssen beide Starre-Körper-Komponenten (von dem Schwert und von dem Würfel) auf *kinematisch* gesetzt sein, damit es nicht zu unerwünschtem Kollisionsverhalten seitens der Physikengine kommt. Ermöglicht man in den Projekteinstellungen für die Physik-Engine alle Kollisionspaare, so werden die On-Collision-Funktionen auch für 2 kinematische Starre-Körper-Komponenten aufgerufen (normalerweise müsste mindestens eine der zwei Starre-Köprer-Komponenten nicht kinematisch sein, damit die OnCollision-Funktionen aufgerufen werden). Der folgende Code zeigt die Winkelberechnung zwischen dem Idealvektor und dem gemessenem Eintritts-und Austrittspunktsvektor. Ist der Winkel zwischen den 2 Vektoren kleiner gleich 45°, so wird ein Schlag als korrekt angesehen:

```
1  private bool vectorHitDirectionEqualsIdealVector(
     SpawnedInteractable si, Vector3 idealVectorlocal,
     Vector3 positionEntered, Vector3 positionLeft)
2  {
3    Vector3 calc_Vector_local = si.gameObject.transform
     .InverseTransformPoint(positionLeft) -
4    si.gameObject.transform.InverseTransformPoint(
     positionEntered);
5
6    Vector2 idealVectorlocal2D = new Vector2(
     idealVectorlocal.x, idealVectorlocal.y);
7    Vector2 calc_Vector_local2D = new Vector2(
     calc_Vector_local.x, calc_Vector_local.y);
8
9    float dot_product_2D = idealVectorlocal2D.x *
     calc_Vector_local2D.x + idealVectorlocal2D.y *
     calc_Vector_local2D.y;
10   float alpha_2D = Mathf.Acos(dot_product_2D / (
     idealVectorlocal2D.magnitude * calc_Vector_local2D
     .magnitude));
11
12   Debug.Log("Angle calculated: " + alpha_2D * Mathf.
     Rad2Deg);
13   angle = alpha_2D * Mathf.Rad2Deg;
14
15   return (alpha_2D * Mathf.Rad2Deg) <= 45;
16 }
```

Nur für den Hauptcollider, welcher die Form des Würfels nachbildet, wird ein Trigger-

Collider verwendet, über dessen Betreten ein Schlagversuch beginnt (Registration eines Schlagversuchs). Beim Verlassen dieses Colliders wird ausgewertet, welche Aktionen ausgeführt werden sollen. Folgende Ereignisse/Situationen beim Verlassen dieses Hauptcolliders sind möglich:

1. Der Schlag wird nicht als Schlagversuch gewertet, weil die Länge des Vektors von Eintritts- und Austrittspunkt in den Hauptcollider nicht mindestens 70% der Kantenlänge eines Würfels abdeckt. Dies soll verhindern, dass versehentliches Berühren des Würfels mit der Schwertspitze sofort als fehlerhafter Schlagversuch gewertet wird. Wenn beispielsweise der Eintrittspunkt ungefähr dem Austrittspunkt entspricht, so ist die Länge dieses Vektors in etwa Null.

2. Ein Schlagversuch wurde registiert, aber der Würfel wurde gemäß der Informationen in dem SpawnedInteractable-Skript nicht korrekt zerschlagen und außerdem ist der Winkel zwischen dem idealen Schlagvektor und dem Vektor aus Eintritts- und Austrittspunkt in den Physik-Scheiben-Vektor größer als 45°. In diesem Fall wird der Schlag mit 0 Punkten bewertet, ein Fehler-Geräusch ertönt, um zu signalisieren, dass der Würfel falsch zerschlagen wurde und der Würfel verschwindet mit einer Fade-Out-Animation.

3. Ein Schlagversuch wurde registiert, der Würfel wurde gemäß der Informationen in dem SpawnedInteractable-Skript nicht korrekt zerschlagen, aber hier ist der Winkel zwischen dem idealen Schlagvektor und dem Vektor aus Eintritts- und Austrittspunkt in den Physik-Scheiben-Vektor kleiner oder gleich 45°. In diesem Fall wird ein schneller Schlag mit Punkten bewertet, jedoch konnten die verschiedenen Präzisions-Collidergruppen durch die Gewschindigkeit des Schlags nicht in der richtigen Reihenfolge aktiviert werden. Deshalb wird hier die Präzision anders bestimmt. Die Präzision wird hier aus einer Kombination der Größe des Winkels zwischen den 2 Vektoren (je kleiner der Winkel, desto höher die Präzision) und der erreichten inneren Genauigkeit errechnet. Die innere Genauigkeit ist umso größer, je mittiger (auch von der Tiefe her!) der Würfel zerschlagen wurde. So gibt es 5 Collider, die dem Haupt-Collider ähneln, aber von der Größe her immer weiter abnehmen, bis nur noch das Zentrum des Würfels abgedeckt wird.

4. Ein Schlagversuch wurde registiert, der Würfel wurde nach den Informationen des SpawnedInteractable-Skriptes korrekt zerschlagen und gleichzeitig ist der Winkel zwischen dem idealem Schlagvektor und dem Vektor aus Eintritts- und Austrittspunkt in den Physik-Scheiben-Vektor (= lokaler Schlagrichtungsvektor) nicht größer oder gleich 60°. In diesem Fall wird die errechnete Genauigkeit aus dem zum Würfel zugehörigem Spawned-Interactable herangezogen. Der Schlag wird mit einer Punktzahl bewertet, welche sich aus Genauigkeit und Reaktionszeit zusammen setzt. So können unter normalem Schwierigkeitsgrad (Standardeinstellung) durch die höchste Präzisionsstufe 60 Punkte maximal erreicht werden, genauso wie theoretisch 60 Punkte durch die Reaktionszeit erreicht werden können. Dafür müsste allerdings die Reaktionszeit bei 0 Sekunden liegen, was praktisch unmöglich ist.

Ein sehr guter Schlag liegt bei über 100 Punkten (die höchste Punktzahl für einen Schlag während der Studie lag bei 107 Punkten). Die Formel für die Bepunktung lautet folgendermaßen:

$$points = (prt * 60 + ppe) * dpm$$

prt = percentRemainingTime (prozentualer Anteil verbleibender Reaktionszeit)
ppe = precisionPointsEarned (durch Präzision gewonne Punkte zwischen 0 - 60)
dpm = difficultyPointsModifier (durch den Schwierigkeitsgrad festgelegter Wert zwischen 0.25 und 3.0)

5. Es wurde versucht, einen farblich unpassenden Würfel zu zerschlagen. In diesem Fall wird der Schlag nicht als Schlagversuch gewertet und ein Glöckengeräusch ertönt, um darauf aufmerksam zu machen, dass die Farbe nicht passt.

Wird ein Schlag mit einer Punktzahl bewertet (bei falschem Zerschlagen auch 0 Punkte), so werden am Ende einige Informationen gespeichert. Darunter fallen die benötigte Zeit, um den Würfel zu zerschlagen, die erreichte Präzision für den Schlag, die erreichte Punktzahl für den Schlag, die Rundennummer, in welcher der Würfel vom Spawn-Skript generiert wurde und zuletzt der aktuell eingestellte Schwierigkeitsgrad, da sich der Schwierigkeitsgrad auch auf die Bepunktung auswirkt.

```
1  void OnTriggerExit(Collider other)
2  {
3    if (!other.gameObject.CompareTag("precisionOne"))
       return;
4
5    SpawnedInteractable si = other.gameObject.transform
       .parent.GetComponent<SpawnedInteractable>();
6
7    if (!si.isHittable()) return;
8
9    positionInitialPhysicsColliderLeft = gameObject.
       GetComponent<Collider>().ClosestPoint(si.getCenter
       ());
10   bool pointsRewarded = si.getPointsRewarded();
11
12   if (!hitOnObjectWasIntended(
       positionInitialPhysicsColliderLeft, si))
13   {
14     //SoundManager.Instance.PlayHitSound(12, 0.5f);
15   }
16
17   if (!pointsRewarded && hitOnObjectWasIntended(
       positionInitialPhysicsColliderLeft, si))
18   {
19     int precision = 0;
20     int anglePrecision = 0;
21     float precisionPercent = 0.0f;
```

```
22      float percentRemainingTime = si.
    getRemainingTimeInPercent();
23      int pointsEarned = 0;
24      int precisionPointsEarned = 0;
25      if ( (si.wasHitInRightDirection() && !
    vectorHitDirectionDeviatesTooStrong(si)) || (
    vectorHitDirectionEqualsIdealVector(si)) )
26      {
27        precision = si.getAvgAccuracy();
28        anglePrecision = getAnglePrecision(si);
29        if (anglePrecision * si.getInnerAccuracy() /
    10.0 > precision) precision = (int) (
    anglePrecision * si.getInnerAccuracy() / 10.0f);
30        precisionPercent = si.getAvgAccuracyPercent();
31
32        if (precision == 1) precisionPointsEarned = 3;
33        if (precision == 2) precisionPointsEarned = 5;
34        if (precision == 3) precisionPointsEarned = 10;
35        if (precision == 4) precisionPointsEarned = 15;
36        if (precision == 5) precisionPointsEarned = 20;
37
38        if (precision == 6) precisionPointsEarned = 25;
39        if (precision == 7) precisionPointsEarned = 35;
40        if (precision == 8) precisionPointsEarned = 45;
41        if (precision == 9) precisionPointsEarned = 50;
42        if (precision == 10) precisionPointsEarned =
    60;
43
44        pointsEarned = (int)(percentRemainingTime * 60
    + precisionPointsEarned);
45
46        pointsEarned = (int) (pointsEarned *
    DifficultyManager.Instance.getPointModifier());
47      }
48      else
49      {
50        precision = 0;
51        precisionPercent = 0.0f;
52      }
53
54      if (canOnlyHitMatchingCube && !other.gameObject.
    transform.parent.gameObject.CompareTag(tagCube))
55      {
56        SoundManager.Instance.PlayHitSound(11, 0.5f);
57        return;
58      }
59
60      SoundManager.Instance.PlayHitSound(precision, 0.5
    f);
61      interactionLocked = false;
62      siToReset = si;
63      resetHasToBeDone = true;
```

```
64      if (precision == 0 && !
    oneTryForHittingCubesCorrectly) return;
65
66      si.setPointsRewarded(pointsEarned);
67      Debug.Log("Earned Points: " + pointsEarned + "
    with precision: " + precisionPercent);
68      highScore.updateHighscore(pointsEarned, si.
    getColor(pointsEarned), tagCube);
69      destroyEffect(si, pointsEarned, precision);
70      listTimeNeededToHitObject.Add(si.
    getNeededTimeToHitObject());
71      listPrecisionWithThatObjectWasHit.Add(
    precisionPercent);
72      listEarnedPoints.Add(pointsEarned);
73      listBlockPairNumber.Add(si.roundGenerated);
74      listDifficulty.Add(DifficultyManager.Instance.
    difficulty);
75      ++countObjectsHit;
76      Object.Destroy(other.gameObject.transform.parent.
    gameObject, 10f);
77    }
78  }
```

4.3 Leveleditor zum Erstellen von Sequenzen

Der Leveleditor soll es ermöglichen, eine Sequenz von Würfelpaaren zu spezifizieren, um diese dann später vom Spawn-Cubes-Skript wiederholt spawnen zu lassen. Spawnen bezeichnet dabei das Erzeugen eines sichtbaren Objektes (hier mit Pfeilen beschriftete Würfel) in der Spieltwelt. Der Leveleditor wurde als Unity *EditorWindow* umgesetzt und kann als besonderes Fenster innerhalb von Unity über die Schaltfläche $Window \rightarrow LevelEditorWindow$ geöffnet werden. Für den Leveleditor wurde im Unity Projekt die Szene *LevelEditor* angelegt. In dieser Szene befindet sich ein Skript *Build Sequence From Level*, welches aus der bestehenden Szene die Sequenz in Form von Kugelkoordinaten bildet und diese dann in Form einer .csv-Datei speichert.
Als öffentliche Variable *Filename* kann im *Build Sequence From Level*-Skript der Name der .csv-Datei spezifiziert werden. Um dieses Skript auszuführen und somit eine Sequenzdatei zu generieren, muss die LevelEditor-Szene gestartet werden.
Abbildung 4.4 zeigt die verschiedenen Optionen, die das LevelEditorWindow bietet:

1. Über den Knopf *Spawn Red Cube* kann ein roter Würfel in der LevelEditor-Szene gespawnt werden. Der Kreiswinkel Phi kann über den Slider *Angle to spawn red Cube* zwischen 0 und 360° eingestellt werden. Der Würfel erscheint mit der eingesellten Höhe im Feld *Height to place Cubes* und mit dem eingestellten Radius im Feld *Radius*, der dem Abstand zum Ursprungspunkt des Koordinatensystems entspricht (0,0,0). Über den Knopf *Spawn Blue Cube* kann analog ein blauer Würfel in die LevelEditor-Szene gespawnt werden. Beim Spawnen wird einem Würfel der enstprechenden Farbe eine eindeutige ID vergeben (beginnend mit der ID = 1), wobei die IDs für rote und blaue Würfel unabhängig voneinander verwaltet und erzeugt werden. Ein roter und blauer Würfel mit derselben ID bilden ein Würfelpaar in der Sequenz.

2. Über eine Gruppe von Knöpfen können alle möglichen Pfeilrichtungen ausgewählt werden. Es kann zeitgleich immer nur einer der 8 Knöpfe für die Pfeilrichtung (Z-Rotation des Würfels) aktiviert sein. Ist eine Pfeilrichtung aktiviert, so erscheint der Knopf mit gelben Hintergrund. Ein nochmaliges Drücken auf einen markierten Knopf demarkiert diesen wieder. Ist einer der 8 Knöpfe aktiviert, so erhält ein roter oder blauer gespawnter Würfel über *Spawn Red Cube* oder *Spawn Blue Cube* direkt diese Pfeilrichtung. Allerdings können die Pfeilrichtungen für eine Auswahl von Würfeln (einer oder beliebig viele) jederzeit geändert werden. So wird die Pfeilrichtung aller ausgewählten Würfel beim Drücken einer der acht Knöpfe entsprechend angepasst. Wird ein Würfel mit normalem Pfeil zu einem Würfel mit diagonalem Pfeil abgeändert, so wird in der Szene auch das Modell des Würfels entsprechend ausgetauscht, damit der diagonale Pfeil nun betrachtet werden kann. Das gleiche gilt auch für einen Würfel mit diagonalem Pfeil, welcher zu einem Würfel mit normalem Pfeil abgeändert wird.

3. Über den Knopf *Select matching other colored Cube with Same Id* kann der zum Würfelpaar zugehörige, anders farbige Würfel gefunden werden, sofern er existiert.

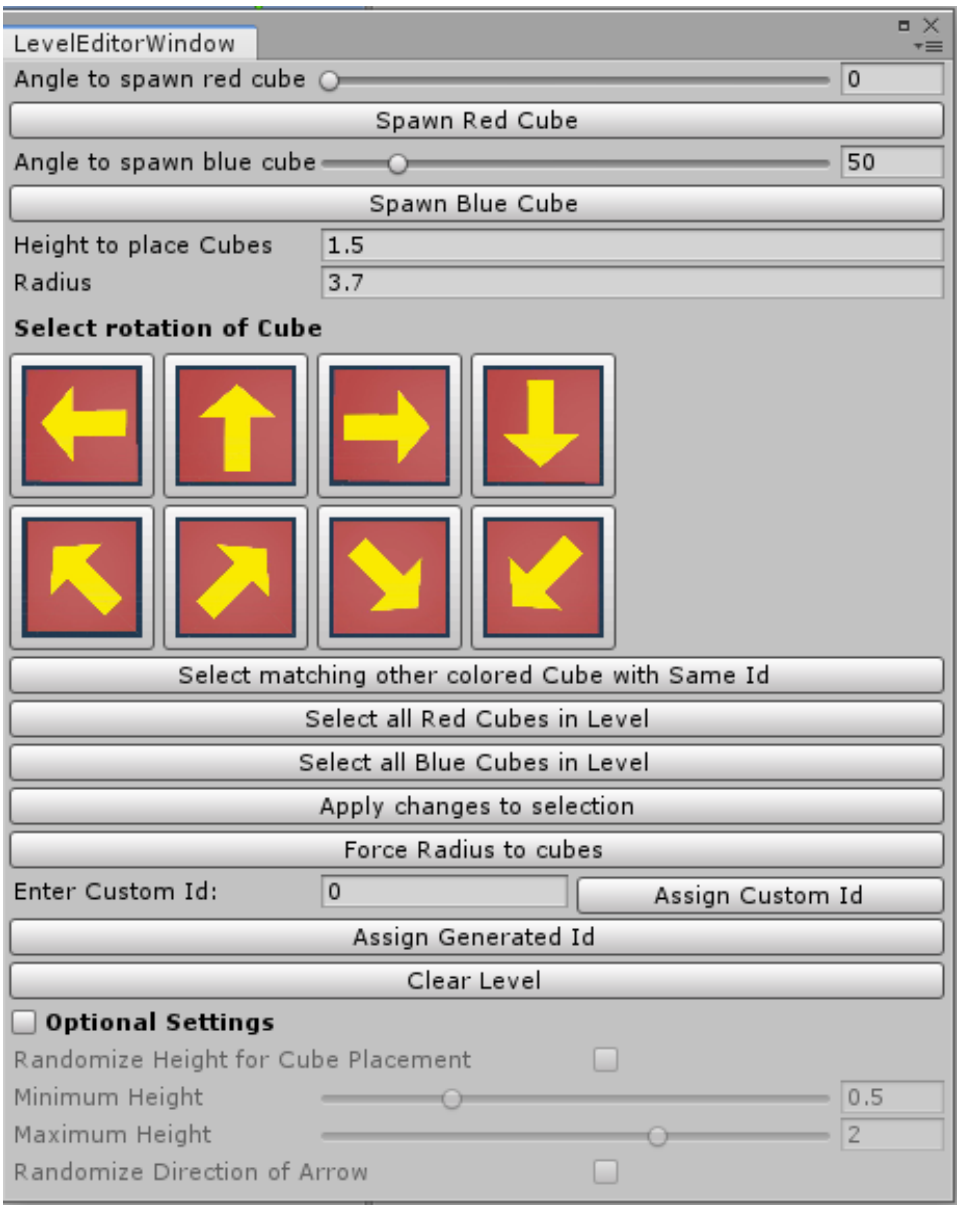


Abbildung 4.4: LevelEditor-Fenster

Hat der aktuell markierte Würfel einen Würfelpartner mit derselben ID, so sind nach Aufruf dieser Funktion beide Würfel in der Szene markiert und ausgewählt.

4. Die Schaltfläche *Select all Red Cubes in Level* wählt alle roten Würfel in der LevelEditor-Szene aus. Die Schaltfläche *Select all Blue Cubes in Level* funktioniert analog für die blauen Würfel.

5. Mittels des Knopfes *Apply changes to selection* können die Pfeilrichtung, die Höhe und der Radius aller ausgewählten Würfel in der Szene angepasst werden. Sind in den optionalen Einstellungen die beiden Checkboxen *Randomize Height for Cube*

Placement und *Randomize Direction of Arrows* aktiviert, so wird die Pfeilrichtung und die Höhe für jeden ausgewählten Würfel zufällig neu gesetzt. Für die Höhe kann ein Minimum- und Maximumwert festgelegt werden.

6. Mittels des Knopfes *Force Radius to cubes* kann der aktuell eingestellte Radius für alle in der Szene ausgewählten Würfel gesetzt werden. Der Radius beeinflusst nicht die Kugelkoordinaten, die von dem *Build Sequence From Level*-Skript generiert werden.
   Es kann beispielsweise für die roten und blauen Würfel in der Level-Editor-Szene ein anderer Radius eingestellt werden, damit man im Editor die Würfel der entsprechenden Farben besser betrachten kann.

7. Es ist auch möglich die IDs von bereits gespawnten Würfeln über das EingabeFeld *Enter Custom Id* in Kombination mit dem Knopf *Assign Custom Id* zu verändern. Es kann immer die ID von genau einem ausgewählten Würfel verändert werden. Ist die Id, welche man neu setzen will, bereits vergeben, dann erscheint eine Meldung, ob man wirklich die ID ersetzen will. Wird die ID dennoch ersetzt, so wird die alte ID des Würfels mit der neuen ID, welche ein anderer Würfel zuvor besaß, getauscht, sodass nicht zwei gleichfarbige Würfel mit derselben ID innerhalb der Szene existieren können. Möchte man bestehende Würfelpaare in der Reihenfolge der Sequenz tauschen, kann diese Funktionalität genutzt werden.

8. Mittels des Knopfes *Assign Generated Id* kann für einen ausgewählten Würfel eine neue ID generiert werden, welche noch von keinem Würfel derselben Farbe genutzt wird. Außerdem wird garantiert, dass die neue vergebene ID größer als alle bisher genutzten IDs für Würfel derselben Farbe ist. In den meisten Fällen sollte es nicht notwendig sein, diese Funktion zu nutzen, da die Würfel bereits beim Spawnen eine gültige ID (beginnend mit 1, der nächste Würfel derselben Farbe erhält die ID 2, usw.) erhalten.

9. Der Knopf *Clear Level* kann genutzt werden, um alle im Level befindlichen Würfelobjekte zu löschen. Wichtig ist anzumerken, dass alle implementierten Funktionen nur innerhalb der LevelEditor-Szene ausgeführt werden, weil sie in anderen Szenen zu unerwünschtem Verhalten führen könnten. Dies wird für jede implementierte Funktionalität zuerst überprüft, bevor der entsprechende Code ausgeführt wird.

10. In den optionalen Einstellungen, welche standardmäßig deaktiviert sind, kann eingestellt werden, dass Höhe und Pfeilrichtung von den ausgewählten Würfeln im Level über die Funktion *Apply changes to selection* randomisiert werden. Für die randomisierte Höhe der Würfel kann zusätzlich noch die untere und obere Grenze in Form von den 2 Slidern *Minimum Height* und *Maximum Height* eingestellt werden.

```
private void spawnCube(string color)
{
  GameObject toSpawn = null;
  Vector3 rotation = new Vector3(0, 0, rotationZ);
  Vector3 pos= new Vector3(0, heightToPlaceCubes, 0);
  if (color == "red")
  {
    pos = sphereToCartesianCoordinate(phiRedSlider,
   radiusDesired);
    if(rotationZ == 0 || rotationZ == 90 || rotationZ
    == 180 || rotationZ == 270)
    {
    toSpawn = cubeNormalRed;
    }
    if(rotationZ == 45 || rotationZ == 135 ||
   rotationZ == 225 || rotationZ == 315)
    {
    toSpawn = cubeDiagonalRed;
    rotation = new Vector3(0, 0, rotationZ - 45);
    }
  }
  if(color == "blue")
  {
    pos = sphereToCartesianCoordinate(phiBlueSlider,
   radiusDesired);
    if (rotationZ == 0 || rotationZ == 90 ||
   rotationZ == 180 || rotationZ == 270)
    {
    toSpawn = cubeNormalBlue;
    }
    if (rotationZ == 45 || rotationZ == 135 ||
   rotationZ == 225 || rotationZ == 315)
    {
    toSpawn = cubeDiagonalBlue;
    rotation = new Vector3(0, 0, rotationZ - 45);
    }
  }
  GameObject newSpawn = Instantiate(toSpawn, pos,
   Quaternion.identity);
  newSpawn.transform.localEulerAngles = rotation;
  StageUtility.PlaceGameObjectInCurrentStage(newSpawn
   );
  Selection.activeGameObject = newSpawn;
  assignUniqueId(newSpawn);
  lookAtCenter(newSpawn);
  EditorSceneManager.MarkSceneDirty(SceneManager.
   GetActiveScene());
}
```

Der Code zum Spawnen eines Würfels zeigt die verschiedenen Fallunterscheidungen, welche für die roten bzw. blauen Würfel mit normalem bzw. diagonalem Pfeil gemacht werden müssen. Der gespawnte Würfel ist nach dem Spawnen direkt ausgewählt, sodass

leicht über die verschiedenen Knöpfe die Pfeilrichtung angepasst werden kann.
Es besteht ein Unterschied zum „normalen“ Spawnen während einer Spielszene durch Spiellogik, da hier der Würfel in der Editorumgebung durch ein Editor-Skript erzeugt wird; D.h. extra Aufrufe wie *StageUtility.PlaceGameObjectInCurrentStage(...) oder EditorSceneManager.MarkSceneDirty(...)* werden benötigt, damit die neu zu spawnenden Würfel korrekt im Level erscheinen und Unity bemerkt, dass es neue ungespeicherte Änderungen in der LevelEditor-Szene gibt. Jeder neu gespawnte Würfel erhält eine für seine Farbe einzigartige ID, welche noch nicht vergeben wurde. Die IDs der verschiedenen Würfel entscheiden über die Rangfolge, d.h. in welcher Reihenfolge sie in der Sequenz vorkommen.
Das BuildSequenceFromLevel-Skript zum Encodieren und Schreiben der Sequenz kann die im Level befindlichen Würfel entweder nach den Namen oder der ID (Attribut in SpawnedInteractable) der Würfel sortieren. Da die Würfel entsprechend ihrer IDs sortiert werden, wird das Bilden der Sequenz auch dann funktionieren, wenn IDs ausgelassen werden. Dies trifft sowohl auf die Sortierung nach IDs wie auch auf die Sortierung nach Namen zu, weil bei der Sortierung nach Namen die Würfel im ersten Schritt die Namensendungen als IDs für sich selbst zugewiesen bekommen. Es wird empfohlen die Sortierung nach ID zu verwenden, weil die aktuell im Level gespawnten Würfel alle denselben Namen besitzen (RedLeftWA (Clone) oder BlueLeftWA(Clone)), und zusätzlich alle schon gültige IDs nach dem Spawnen besitzen. Damit die Sortierung nach Namen korrekt funktioniert, müssten zunächst die Namen der gesamten in der Szene befindlichen Würfel so umbenannt werden, dass jeder Würfel am Ende des Namens eine Zahl erhält, welche dann als ID zum Sortieren verwendet wird. Beispielsweise RedLeftWA (Clone)1, RedLeftWA (Clone)2, BlueLeftWA (Clone)1 und BlueLeftWA (Clone)2 für eine Sequenz bestehend aus 2 Würfelpaaren.

```
1 void Start()
2 {
3   List<GameObject> rootObjects = new List<GameObject
     >();
4   Scene scene = SceneManager.GetActiveScene();
5   scene.GetRootGameObjects(rootObjects);
6   List<GameObject> redCubes = rootObjects.Where(x =>
     x.name.Contains("Red")).ToList();
7   List<GameObject> blueCubes = rootObjects.Where(x =>
       x.name.Contains("Blue")).ToList();
8   rootObjects.Sort( (obj1, obj2)=>obj1.name.CompareTo
     (obj2.name));
9   ...
```

Da sich der gesamte Code des *BuildSequenceFromLevel-Skriptes* in der Start-Funktion befindet, beginnt das Skript direkt nach dem Starten der LevelEditor-Szene, die Sequenz an Würfelpaaren zu bestimmen. Im *BuildSequenceFromLevel-Skript* werden als erstes alle in der Szene befindlichen Würfel-Objekte in rote und blaue Würfel aufgeteilt (*redCubes und blueCubes*).

```
void Start()
{
  ...
  if (!sortCubesById)
  {
    foreach (GameObject o in redCubes)
    {
      SpawnedInteractable si = o.GetComponent<
   SpawnedInteractable>();
      string s = o.name;
      if (s.EndsWith(")"))
      {
        int index = s.IndexOf("(");
        string end = s.Substring(index + 1);
        int index2 = end.IndexOf(")");
        string number = end.Remove(index2);
        si.setId(int.Parse(number));
      }
      else si.setId(0);
      ...
  }
}
```

Der Code zum Sortieren nach Namensendung parst alle Zeichen nach der schließenden Klammer „)“ von Clone als Zahl, welche als ID zum Sortieren verwendet wird. Eine bereits vorhandene ID eines Würfels wird in diesem Fall überschrieben. Insofern können die Namen der im Level befindlichen Würfel auch angepasst werden, wenn man die Reihenfolge der Würfelpaare verändern will.

```
void Start()
{
  ...
  // Sorting of red and blue cubes by their defined
   ID in SpawnedInteractable
  redCubes.Sort((obj1, obj2) => obj1.GetComponent<
   SpawnedInteractable>().getId().CompareTo(obj2.
   GetComponent<SpawnedInteractable>().getId()));
  blueCubes.Sort((obj1, obj2) => obj1.GetComponent<
   SpawnedInteractable>().getId().CompareTo(obj2.
   GetComponent<SpawnedInteractable>().getId()));

    List<SphereCoordinates> list = new List<
   SphereCoordinates>();
  // build list of sphereCoordinates to write in csv-
   file
  for (int i = 0; i < redCubes.Count; ++i)
  {
  Vector2 res = cartesianToSphereCoordinate(redCubes[
   i].transform.position);
  Vector2 res2 = cartesianToSphereCoordinate(
   blueCubes[i].transform.position);
  SphereCoordinates sc = new SphereCoordinates(res
```

```
    [0],res[1],res2[0],res2[1]);
15  sc.radius = (redCubes[i].transform.position - new
     Vector3(0, 0, 0)).magnitude;
16  sc.radius2 = (blueCubes[i].transform.position - new
      Vector3(0, 0f, 0)).magnitude;
17  list.Add(sc);
18  Debug.Log(redCubes[i].name);
19  Debug.Log(blueCubes[i].name);
20  }
21  LoadFixedSequenceOfSpawns.writeCsvFromCubeLists(ref
      list, ref redCubes, ref blueCubes, filename);
22 }
```

Nach dem Sortieren der roten und blauen Würfel-Liste, wird eine Liste von Kugelkoordinaten aus diesen beiden sortierten Listen gebildet. Dabei werden die aktuellen kartesischen Koordinaten der verschiedenen Würfel in Kugelkoordinaten umgerechnet. Der aktuell eingestellte Radius für die verschiedenen Würfel wird auch gespeichert, hat allerdings nur einen Einfluß, falls man den in den Kugelkoordinaten gespeicherten Radius beim Spawnen im SpawnCubes-Skript verwendet, was standardmäßig nicht der Fall ist. Ist diese Liste an Kugelkoordinaten gebildet worden, so wird eine .csv-Datei mit allen berechneten Kugelkoordinaten geschrieben (*LoadFixedSequenceOfSpawns.writeCsvFromCubeLists(...)*).

4.4 Implementierte Sequenzen

Da die Sequenzen für das kontrollierte Spawnen der Würfel eine entscheidende Rolle bei der Untersuchung der *Forschungsfrage* (kann das implizite Lernen der sequenzierten Schlagbewegungen mithilfe der implementierten VR-Anwendung nachgewiesen werden?) spielen, wurde eine abstrakte Klasse als Datenstruktur für eine Sequenz geschaffen. Diese Klasse *BlockSequence* definiert verschiedene abstrakte Methoden, welche innerhalb vom SpawnCubes-Skript verwendet/aufgerufen werden.

```
1  public abstract class BlockSequence : MonoBehaviour
2  {
3    public abstract bool hasNextSphereCoordinates();
4    public abstract SphereCoordinates
      nextSphereCoordinates();
5    public abstract SphereCoordinates[]
      twoRandomSphereCoordinatesPairsForWholeSequence();
6    ...
7    public abstract int getCubePairCount();
8    public abstract int getIterationCount();
9    ...
10 }
```

1. Die Funktion *hasNextSphereCoordinates* gibt an, ob die Sequenz noch weitere Kugelkoordinaten für ein Würfelpaar beinhaltet oder ob schon das Ende der Sequenz erreicht wurde.

2. Die Funktion *nextSpehreCoordinates* wird genutzt, um die Kugelkoordinaten für das nächste Würfelpaar in der Sequenz zu erhalten. Es sollte vorher immer geprüft worden sein, ob die Sequenz überhaupt noch Kugelkoordinaten für ein ein weiters Würfelpaar beinhaltet.

3. Zuletzt wird die Funktion *twoRandomSphereCoordinatesPairsForWholeSequence* für den Antizipationstest benötigt. Sie bildet für die gesamte Sequenz eine doppelt so große Sequenz, wobei jeweils jedes Würfelpaar genau einmal an zufälliger Position vorkommt, immer gefolgt von dem Nachfolger-Würfelpaar in der Originalsequenz. Dabei hat das letzte Würfelpaar wieder den Anfang der Sequenz als Nachfolgerpaar.

Über die beiden Funktionen *getCubePairCount* und *getIterationCount* kann abgefragt werden, wie viele Würfelpaare die Sequenz beinhaltet bzw. wie viele Wiederholungen der Sequenz in dem dafür vorgesehenen Block erfolgen sollen.

4.4.1 Zufällige Sequenzen

Zufällige Sequenzen werden benötigt, um nachweisen zu können, dass eine spezifische Sequenz implizit von den Probanden gelernt wurde. So wird das Abschneiden in einem Block mit der gelernten Sequenz (nach einer Trainingsphase mit Blöcken der zu lernenden Sequenz) mit dem Abschneiden in einem Block zufälliger Sequenzen verglichen. Ist die Leistung im Block mit der gelernten Sequenz signifikant größer als im Block mit der zufälligen Sequenz, so deutet dies darauf hin, dass die Sequenz erfolgreich gelernt wurde. Um zufällige oder halb zufällige Sequenzen als BlockSequence zu spezifizieren, können die Skripte *RandomizedBlockSequence* und *BlockSequenceOneHandRandom* verwendet werden. Diese beiden Skripte erweitern BlockSequence und müssen daher die gerade beschriebenen, abstrakten Funktionen implementieren. Während bei der *RandomizedBlockSequence* alle Würfelpaare randomisiert werden, wird bei der BlockSequenceOneHandRandom eine Hand (links oder rechts) von einer mit dem Leveleditor erstelleten Sequenz geladen, während die andere Hand randomisiert wird. Für beide Skripte wird in der *Awake*-Funktion eine Liste an Kugelkoordinaten definierter Länge gebildet. In einem Fall wird die Länge durch *countCubePairs* bestimmt, im anderen Fall entscheidet die Länge der geladenen mit dem Leveleditor erstellten Sequenz die Länge, da hier die linke oder rechte Hand der Kugelkoordinaten durch zufällige Koordinaten ersetzt wird. Diese gebildete Liste wird so oft durchlaufen, wie mit dem Parameter *iterationCount* spezifiziert wurde. Die wichtigsten Funktionen sind folgendermaßen in *RandomizedBlockSequence* implementiert:

```
1 public override bool hasNextSphereCoordinates()
2 {
3   if (actIndexInList % sequenceOfSpawns.Count == 0 &&
      actIndexInList > 0)
4   {
5     if (pointScoreActIteration >
     maxPointScoreOneIteration)
```

```
    maxPointScoreOneIteration = pointScoreActIteration
    ;
     pointScoreAllIterations.Add(
    pointScoreActIteration);
     pointScoreActIteration = 0;
     if(allCubePairsAreRandom)
     {
       sequenceOfSpawns.Clear();
       for (int i = 0; i < countCubePairs; ++i)
       {
         generateRandomizedSphereCoordinates();
       }
     }
   }
   return actIndexInList < iterations *
    sequenceOfSpawns.Count;
}

public override SphereCoordinates
    nextSphereCoordinates()
{
   return sequenceOfSpawns[actIndexInList++ %
    sequenceOfSpawns.Count];
}
```

Es wird in interner Index verwendet, um zu speichern, an welcher Position man sich aktuell in der Liste an gebildeten Kugelkoordinaten befindet. Der Index ist somit von 0 bis $N_{IT} \cdot l_S - 1$ (wobei N_{IT} = Iterationszahl der Sequenz und l_S = Länge der Sequenz) gültig. Ist die Option *allCubePairsRandom* aktiviert, die bewirkt, dass sich keine zufällige Sequenz wiederholt, so wird nach dem Durchlaufen der zufällig gebildeten Sequenz die nächste zufällige Sequenz derselben Länge gebildet.
Für beide Skripte kann eine Reihe an Parametern eingestellt werden, welche die Erzeugung zufälliger Kugelkoordinaten steuern:

1. die Variable *randomizeElevation* legt fest, ob die Höhe (theta bei den Kugelkoordinaten) der Würfel randomisiert werden.

2. die Variable *defaultElevation* legt den Standardwert (90) für theta fest. Dies entspricht einer Position des Würfels in etwa auf Augenhöhe.

3. die Variable *offsetElevationBothSides* legt fest, wie viel Grad theta nach oben oder unten vom Standardwert abweichen kann. Wird hier 30 gewählt, so kann theta bei der zufälligen Erzeugung der Kugelkoordinaten zwischen 60 und 120 liegen.

4. die beiden Variablen *offsetLeftSideRangeMin* (15) und *offsetLeftSideRangeMax* (40) steuern den Wert für phi für den linken Würfel des Würfelpaares.

5. die beiden Variablen *offsetRightSideRangeMin* (-40) und *offsetRightSideRangeMax* (-15) steuern den Wert für phi für den rechten Würfel des Würfelpaares. Hier muss beachtet werden, dass die Werte für den rechten Würfel negativ gewählt werden,

während die Werte für den linken Würfel positiv gewählt werden, damit der rechte zufällige Würfel rot ist und der linke blau ist. Tauscht man die Werte der Grenzen für die Würfel, so wird nun der rechte gespawnte Würfel die Farbe blau besitzen, während der linke die Farbe rot aufweist.

Im *RandomizedBlockSequence*-Skript kann zusätzlich noch eingestellt werden, ob alle acht Pfeilrichtungen oder nur die normalen Pfeile ohne Diagonalpfeile vom Skript erzeugt werden können sollen (Einstellung *diagonalArrowsAreUsed*).
Außerdem kann noch eingestellt werden, dass die Farben von den gespawnten Würfeln vertauscht sein können (Einstellung *redAndBlueCubesCanBeInterchanged*). Dies hat zur Folge, dass man nun auf die Farbe der gespawnten Würfel achten muss und sich nicht darauf verlassen kann, dass immer rechts der rote Würfel spawnt, während links ein blauer Würfel spawnt. Die Einstellung *allCubePairsAreRandom* bewirkt, dass sich die zufällig gebildete Sequenz innerhalb des Blocks nicht wiederholt,d.h. nach dem ersten Durchlaufen der zufälligen erzeugten Sequenz wird eine neue zufällige Sequenz generiert. Würde man stattdessen einfach die Wiederholungszahl auf 1 setzen und die Anzahl der Würfelpaare in der zufälligen Sequenz entsprechend vergrößeren, könnte man die zufällige Sequenz nicht mehr so leicht mit einer händisch erstellten Sequenz deutlich kleinerer Länge vergleichen. So ist beispielsweise die beste bzw. durchschnittlich erreichte Punktzahl (in der Datei blockDataX.xml gespeichert, siehe Kapitel Datensammlung von Spieldaten) für eine Sequenz natürlich von der Länge der Sequenz abhängig. Daraus folgt, das nur Sequenzen derselben Länge miteinander verglichen werden sollten.

4.4.2 Mit Leveleditor erstellte Sequenzen

Um eine im Leveleditor erstellte Sequenz zu verwenden, wird das Skript *BlockSequenceFromFile* verwendet. Hier muss der Dateiname der .csv-Datei in der öffentlichen Variable *filenameCsv* spezifiziert werden, welche zuvor mit dem LevelEditor generiert wurde. Auch hier werden die Kugelkoordinaten aus der .csv-Datei innerhalb der *Awake*-Funktion in eine Liste von Kugelkoordinaten geladen, welche intern verwendet wird. Zum Laden der csv-Datei wird die Hilfsklasse *LoadFixedSequenceOfSpawns* verwendet, welche die Liste *sequenceOfSpawns* direkt modifiziert, da sie als Referenz mit *ref*-Keyword übergeben wurde.

```
1 public override void Awake()
2 {
3   base.Awake();
4   sequenceOfSpawns = new List<SphereCoordinates>();
5   if (!string.IsNullOrEmpty(filenameCsv))
     LoadFixedSequenceOfSpawns.loadSpawnSequence(ref
     sequenceOfSpawns, filenameCsv);
6 }
```

Der folgende Code bestimmt die Würfelpaare und Nachfolger-Würfelpaare in zufälliger Reihenfolge für den Antizipationstest:

```
public override SphereCoordinates[]
    twoRandomSphereCoordinatesPairsForWholeSequence()
{
  SphereCoordinates[] result = new SphereCoordinates[
    sequenceOfSpawns.Count * 2];
  List<int> usedIndexes = new List<int>();
  int resultIndex = 0;
  for(int i = 0; i < sequenceOfSpawns.Count; ++i)
  {
    int z = Random.Range(0, sequenceOfSpawns.Count);
    while (usedIndexes.Contains(z)) z = Random.Range
    (0, sequenceOfSpawns.Count);

    usedIndexes.Add(z);
    result[resultIndex++] = sequenceOfSpawns[z];
    result[resultIndex++] = sequenceOfSpawns[(z ==
    sequenceOfSpawns.Count - 1 ? 0 : z + 1)];
  }
  return result;
}
```

In einer Liste von bereits verwendeten Indexen wird gespeichert, welche Würfelpaare von der Sequenz bereits verwendet wurden, sodass jedes Würfelpaar gefolgt vom Nachfolger-Würfelpaar genau einmal im Ergebnis-Array an Kugelkoordinaten vorkommt. Zunächst wird das zufällig bestimmte Würfelpaar zum Ergebnis hinzugefügt, dann wird der Nachfolger des zufällig bestimmten Würfelpaars hinzugefügt. Da die Länge der Liste - 1 den letzten validen Index für die Liste darstellt (letztes Würfelpaar innerhalb der Sequenz), muss in diesem Fall wieder das erste Würfelpaar der Sequenz als Nachfolger herangezogen werden (Index 0). Der ResultIndex wird immer erst nach der Anweisung mittels Postinkrement um eins erhöht, sodass der erste Aufruf der Anweisung *result[resultIndex++] = sequenceOfSpawns[z];* den ersten Index im Result-Array füllt (result[0]).

4.5 Antizipationstest

Der Antizipationstest wird benötigt, um vorhandenes explizites Wissen bei den Probanden nachzuweisen. Da das Lernen der Sequenzen implizit erfolgen soll, ist gewünscht, dass die Probanden bei diesem Test schlecht abschneiden. Denn ein gutes Abschneiden bei diesem Test spricht für eine große Menge an explizitem Wissen zu der gelernten Sequenz. Die Abbildung 4.5 beschreibt grob den Ablauf des Antizipationstests, der in Kombination mit dem Antizipationsskript und den Aktionen des Spielers zustande kommt. Im Folgendem sollen die einzelnen Schritte genauer erläutert werden:

1. Zunächst spawnt das Antizipationsskript ein zufälliges Würfelpaar aus der Sequenz in der Antizipationstest-Szene. Nun muss der Spieler die erschienenen Würfel zerschlagen, was ihm als Hilfe dienen kann, sich zu erinnern, welche Würfel nach diesem Würfelpaar in der Sequenz an der Reihe sind. Sind beide Würfel zerschlagen worden, ist dieser Schritt abgeschlossen.

2. Als nächstes aktiviert das Antizipationsskript beide Pointer der Controller und lässt eine transparente Kugel zum Spieler hinschweben. Nun kann der Spieler durch das Drücken der Triggertaste die Würfel des Nachfolgerwürfelpaars an der Stelle spawnen, wo der aktivierte Pointer auf die transparente Kugel trifft. Es soll festgestellt werden, ob der Spieler die Positionen des Nachfolgerwürfelpaars in etwa abschätzen kann. Als erstes spawnt der Spieler mithhilfe des Drückens des Trigger-Buttons des rechten Controllers den roten, rechten Würfel des Nachfolgerwürfelpaars an der Stelle, die seiner Meinung nach korrekt ist. Anschließend spawnt der Spieler mithilfe des Drückens des Trigger-Buttons des linken oder rechten Controllers den blauen, linken Würfel des Nachfolgerwürfelpaars an der von ihm vorgesehenen Stelle. Sind beide Würfel vom Spieler gespawnt worden, ist dieser Schritt abgeschlossen.

3. Nun wird dem Spieler Feedback zu seinen geschätzen Positionen des Nachfolgerwürfelpaars gegeben, indem die korrekten Positionen des Nachfolgerwürfelpaars angezeigt werden. Zusätzlich werden außerdem die gespawnten Würfel je nach Abstand zu den korrekten Positionen unterschiedlich eingefärbt (Grün für geringen Abstand und Violett für zu großen Abstand). Dieser Schritt ist optional, im Diagramm durch die gestrichelten Pfeile gekennzeichnet. Nachdem der Spieler eine kurze Zeitspanne gewartet hat, um das Feedback betrachten zu können, ist dieser Schritt abgeschlossen.

4. Als letztes aktiviert das Antizipationsskript die Auswahl-UI zum Auswählen der Pfeilrichtungen des Nachfolgerwürfelpaars. Nachdem bisher nur die Positionen des Nachfolgerwürfelpaars geschätzt wurden, müssen jetzt die Pfeilrichtungen angegeben werden. Der Spieler kann nun die Pfeilrichtungen im Auswahlfenster für den roten und den blauen Würfel festlegen, indem er die seiner Meinung nach passenden Pfeilrichtungen über die verschiedenen Knöpfe auswählt. Hat der Spieler versehentlich andere Pfeilrichtungen ausgewählt, so kann er seine Auswahl noch

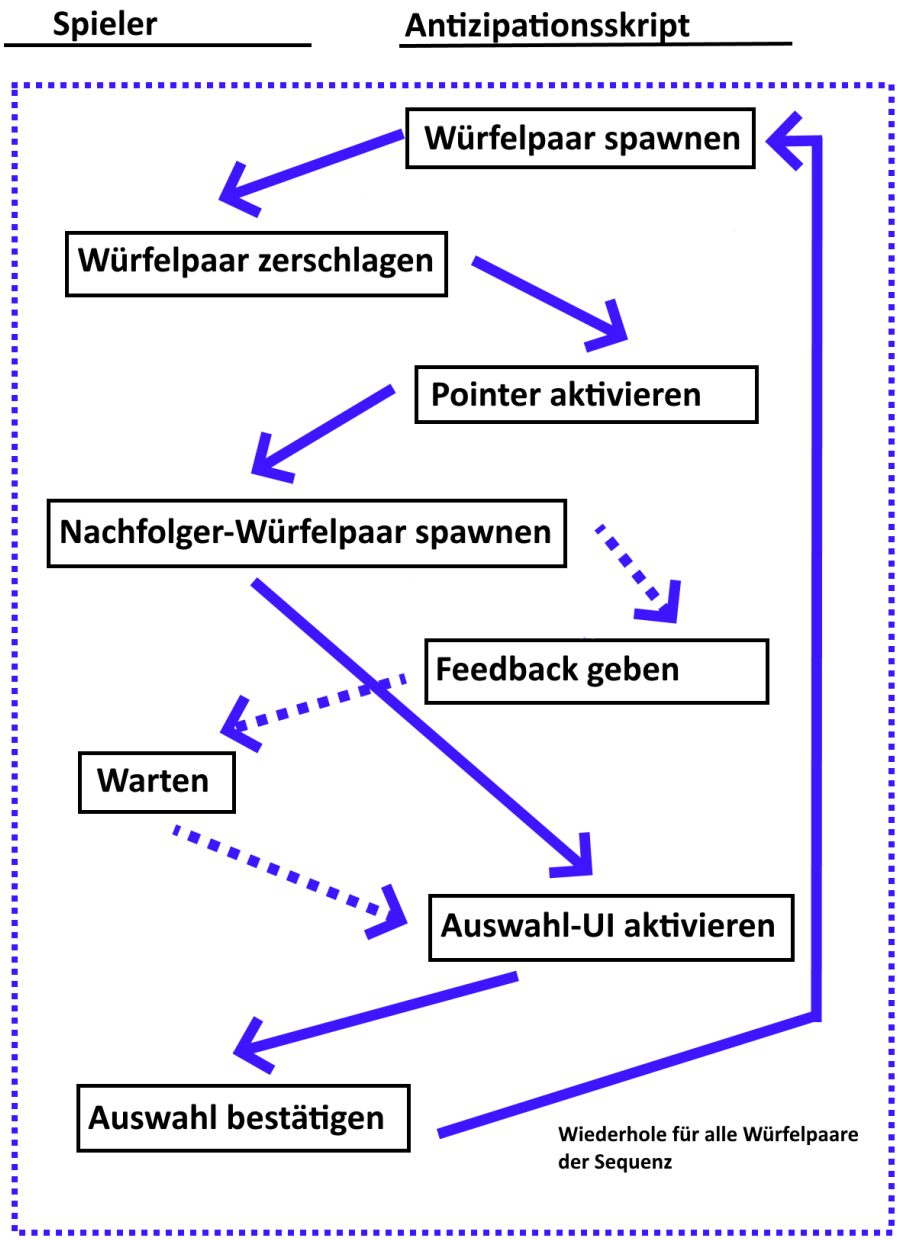


Abbildung 4.5: Ablauf des Antizipationstests

auf eine andere Pfeilrichtung abändern.
Es existieren zwei Versionen des Auswahlfensters, die erste umfasst alle acht Pfeile pro Würfel mit Diagonalpfeilen, die zweite umfasst nur vier Pfeile pro Würfel ohne die Diagonalpfeile. In der Szene kann eingestellt werden, welche Version des Auswahlfensters aktiviert sein soll. Hat der Spieler für beide Würfelfarben eine Pfeilrichtung ausgewählt, so kann er seine Auswahl durch Drücken des Buttons *Submit Answers* bestätigen. Daraufhin ist der letzte Schritt abgeschlossen.

Alle erläuterten Schritte werden für jedes in der (zu prüfenden) Sequenz vorkommende Würfelpaar genau einmal durchlaufen, d.h. die Länge der Sequenz bestimmt, wie oft die Schritte durchlaufen werden müssen. Dabei ist die Reihenfolge der Würfelpaare, in der sie beim Antizipationstest erscheinen, komplett zufällig. Als Nachfolger für das letzte Würfelpaar der Sequenz ist wieder das erste Würfelpaar der Sequenz definiert, da nach einem Durchlauf der zu lernenden Sequenz die Sequenz wieder von vorne beginnt.

Der Programm-Code für den Antizipationstest ist auf die 3 Skripte *AnticipationScript*, *SphereToSpawnGreyCube* und *DelegateButtonCallsChoiceMenu* aufgeteilt. Das *AnticipationScript* implementiert den oben beschriebenen Ablauf unterteilt in die vier Schritte innerhalb der *Update*-Funktion, indem es immer wieder darauf wartet, dass der Spieler die von ihm vorgesehenen Aktionen durchführt. Über eine Menge an internen Boolean-Variablen wird der aktuelle Zustand des Antizipationstests gespeichert, damit ersichtlich ist, in welchem der vier Schritte man sich aktuell befindet. Der folgende, vereinfachte Codeausschnitt aus dem *AnticipationScript* zeigt die Implementierung der verschiedenen Schritte:

```
1  public void Update()
2  {
3    ...
4    if (!cubePairSpawned && cubePairPointedSpawned &&
       choiceDialogeMade)
5    {
6      ...
7      spawnCubes.spawnCubesForSphereCoordinates(
       sphereCoordinates[sphereCoordinatesIndex++]);
8      ...
9      cubePairSpawned = true;
10     cubePairPointedSpawned = false;
11     choiceDialogeMade = false;
12   }
13   if(cubePairSpawned && spawnCubes.
       bothCubesDestroyedOrHit() && !
       cubePairPointedSpawned)
14   {
15     sphere.moveMeshColliderUp();
16     if (sphere.getPosition().y >= 0) sphereMoved =
       true;
17   }
18   if(sphereMoved)
19   {
20     cubePairPointedSpawned = true;
21     sphere.activateSpawningAbility();
22     sphereMoved = false;
23   }
24   if(!choiceDialogeMade && sphere.bothCubesSpawned())
25   {
26     if(spawnActualCubePositions)
```

```
27      {
28        ...
29        // FeedbackCode
30      }
31      ...
32      sphere.moveMeshColliderDown();
33      menuTimerStarted = true;
34      choiceDialogeMade = true;
35    }
36    if(menuTimerStarted)
37    {
38      menuTimer += Time.deltaTime;
39      if(menuTimer > 1.2)
40      {
41        menuTimerStarted = false;
42        activateChoiceUI(sphereCoordinates[
      sphereCoordinatesIndex++]);
43      }
44    }
45    if(choiceMenu.choiceWasMade())
46    {
47      // Save made choices
48      ...
49      // Reset different components, so this workflow
      will work for the next random cube pair of the
      sequence
50      ...
51    }
52  }
```

Das Skript *SphereToSpawnGreyCube* wird benötigt, damit der Spieler mittels der Controller kontrolliert das nachfolgende Würfelpaar spawnen kann. Es ist in der Antizipationstestszene an die transparente Kugel angehängt, welche verwendet wird, damit der Spieler die Würfel in Form passender Kugelkoordinaten innerhalb der Kugel spawnen kann. Die Stelle, wo der Pointer auf die transparente Kugel trifft, wird als die vom Spieler ausgewählte Position betrachtet, an welcher der Würfel des Nachfolgerwürfelpaars gespawnt werden soll.
Das Skript *DelegateButtonCallsChoiceMenu* steuert die verschiedenen Aktionen, welche nach Auswahl einer der Buttons des UI (z.B. Pfeilrichtungen) erfolgen und prüft, ob die vom Spieler getroffene Auswahl (teilweise) korrekt ist. Es besteht die Möglichkeit, dem Spieler Feedback zu seiner getroffenen Auswahl zu geben. Ob dem Spieler Feedback zu den getroffenen Antworten angezeigt wird, hängt von der Einstellung *showResultsChoiceMenu* im *AnticipationScript* ab. Dann werden die korrekten Antworten in Form der zugehörigen Pfeilrichtungen grün eingefärbt, während vom Spieler falsch ausgewählte Antworten rot eingefärbt werden.
Im *AnticipationScript* können zwei Einstellungen zum Feedback während des Antizipationstests vorgenommen werden:

- die Variable *showResultsChoiceMenu* entscheidet, ob auf dem Auswahl-UI für die verschiedenen Pfeilrichtungen Feedback für die getroffene Auswahl angezeigt wird.

Ist das Feedback aktiviert, so ist das Feedback dreieinhalb Sekunden für den Spieler sichtbar, ehe das Auswahl-UI verschwindet und das nächste zufällige Würfelpaar aus der Sequenz erscheint, welches der Spieler wieder zerschlagen muss.

- die Variable *spawnActualCubePositions* steuert, ob Feedback auf die vom Spieler ausgewählten Positionen des Nachfolgerwürfelpaars erscheinen soll. Die Einzelheiten für dieses Feedback sind in Schritt drei des Ablaufs des Antizipationstests aufgeführt.

4.6 Datensammlung von Spieldaten

Es werden eine Reihe von Daten in der Hauptszene und in der Antizipationstest-Szene gesammelt. Um die verschiedenen Daten zu speichern, wird Serialisierung mittels der Klasse *XmlSerializer* genutzt, welche in der Bibliothek *System.Xml.Serialization* enthalten ist. Die gesammelten Daten der Hauptszene werden in den zwei Dateien sportGameDataX.xml unter dem Verzeichnis *Assets/xml/SportGame/* und blockGameDataX.xml unter dem Verzeichnis *Assets/xml/BlockData/* gespeichert. Die gesammelten Daten der Antizipationstest-Szene werden in der Datei anticipationtestX unter dem Verzeichnis *Assets/xml/AntizipationTest/* gespeichert. Dabei werden die verschiedenen, angelegten Dateien durchnummeriert, damit keine bereits angelegte Datei überschrieben werden kann (startend mit X = 1, X = 2, X = 3, ...). Für die verschiedenen zu speichernden Daten wurden verschiedene Datenstrukturen angelegt:

```
1  public class SportGameData
2 {
3    public List<float> time_left;
4    public List<float> time_right;
5    public List<float> precision_left;
6    public List<float> precision_right;
7    public List<int> points_earned_left;
8    public List<int> points_earned_right;
9    public List<int> round_generated_left;
10   public List<int> round_generated_right;
11   public List<Difficulty> difficultyLeft;
12   public List<Difficulty> difficultyRight;
13   public int spawnedObjectCount;
14   public int countObjectsHitRightHand;
15   public int countObjectsHitLeftHand;
16 }
```

In der Datei sportGameDataX.xml sind folgende Daten enthalten:

- Die verbleibende Zeit für alle gemachten Schläge aufgeteilt nach linker und rechter Hand (*time_left* und *time_right*)
- Die erreichte Präzision für alle gemachten Schläge aufgeteilt nach linker und rechter Hand (*precision_left* und *precision_right*)

- Die erzielte Punktzahl für alle gemachten Schläge aufgeteilt nach linker und rechter Hand (*points_earned_left* und *points_earned_right*)
- Die zugehörige Rundennummer für alle gemachten Schläge aufgeteilt nach linker und rechter Hand (*round_generated_left* und *round_generated_right*) Die Rundennummer wird benötigt, um nicht gemachte Schläge feststellen zu können, d.h. der Spieler hat es zeitlich verpasst, einen Würfel eines Würfelpaars zu zerschlagen.
- Der aktuell eingestellte Schwierigkeitsgrad für alle gemachten Schläge aufgeteilt nach linker und rechter Hand (*difficultyLeft* und *difficultyRight*)
- Die Anzahl gespawnter / erzeugter Würfel (*spawnedObjectCount*)
- Die Anzahl der mit der linken bzw. rechten Hand zerschlagenen Würfel (*countObjectsHitRightHand* und *countObjectsHitRightHand*)

Es ist durchaus möglich, dass die Listen für die linke bzw. rechte Hand eine unterschiedliche Länge aufweisen, wenn mit einer Hand mehr Schläge verpasst wurden als mit der anderen Hand.

```
1  public class BlockGameData
2 {
3    public List<int> listCubePairCount;
4    public List<int> listIterationCount;
5    public List<int> listBestPointScoreAllIterations;
6    public List<int> listAvgPointScoreAllIterations;
7    public List<int> listPointScoreAllIterations;
8 }
```

In der Datei blockdataX.xml sind folgende Daten enthalten:

- Für jeden absolvierten Block in der Hauptszene ist die Anzahl der Würfelpaare der in dem Block verwendeten Sequenz gespeichert (*listCubePairCount*)
- Für jeden absolvierten Block in der Hauptszene ist die Anzahl der Iterationen gespeichert, d.h. wie oft die Sequenz innerhalb des Blocks wiederholt wird (*listIterationCount*)
- Für jeden absolvierten Block wird der beste Versuch von allen Iterationen als Punktzahl für die im Block verwendete Sequenz gespeichert (*listBestPointScoreAllIterations*)
- Für jeden absolvierten Block wird die durchschnittlich erreichte Punktzahl für die im Block verwendete Sequenz gespeichert (*listAvgPointScoreAllIterations*)
- Für alle Iterationen in allen absolvierten Blöcken wird die erreichte Punktzahl gespeichert (*listPointScoreAllIterations*) Die Liste enthält zunächst alle Iterationen des ersten Blockes, dann alle Iterationen des zweiten Blockes, etc.

```
public class AntizipationTestData
{
  public List<Vector3> positionCubeHitLeft;
  public List<Vector3> positionCubeHitRight;
  public List<Vector3> positionCubeOriginalLeft;
  public List<Vector3> positionCubeOriginalRight;
  public List<Vector3> positionCubePointedLeft;
  public List<Vector3> positionCubePointedRight;

  public List<Vector3> rotationCubeOriginalLeft;
  public List<Vector3> rotationCubeOriginalRight;
  public List<Vector3> rotationCubePointedLeft;
  public List<Vector3> rotationCubePointedRight;
  public List<Vector3> rotationCubeHitLeft;
  public List<Vector3> rotationCubeHitRight;

  public List<float> rotationZOriginalLeft;
  public List<float> rotationZOriginalRight;
  public List<float> rotationZChoosenLeft;
  public List<float> rotationZChoosenRight;
  public int scoreRightChoices;
}
```

In der Datei anticipationtestX.xml sind folgende Daten enthalten:

- Die Transform aufgeteilt in Positions- und RotationsVektor von den Würfeln, welche beim Antizipationstest von den Probanden zerschlagen wurden (*positionCubeHitLeft, positionCubeHitRight* und *rotationCubeHitLeft, rotationCubeHitRight*)
- Die Transform aufgeteilt in Positions- und RotationsVektor von den Würfeln, welche von den Probanden beim Antizipationstest gespawnt wurden. Sie stellen somit die geschätzte Position des Nachfolgerwürfelpaars dar (*positionCubePointedLeft, positionCubePointedRight* und *rotationCubePointedLeft, rotationCubePointedRight*)
- Die Transform aufgeteilt in Positions- und RotationsVektor von dem tatsächlichen Nachfolgerwürfelpaar (*positionCubeOriginalLeft, positionCubeOriginalRight* und *rotationCubeOriginalLeft, rotationCubeOriginalRight*)
- Die Pfeilrichtungen des Nachfolgerwürfelpaars codiert nach der Z-Koordinate der Rotation für den linken und rechten Würfel (*rotationZOriginalLeft*, *rotationZOriginalRight*)
- Die von den Probanden beim Antizipationstest ausgewählten Pfeilrichtungen für den linken und rechten Würfel des Nachfolgerwürfelpaars (*rotationZChoosenLeft*, *rotationZChoosenRight*)
- Die ermittelte Anzahl richtiger Antworten über die Anzahl übereinstimmender Pfeilrichtungen von gewählten Pfeilrichtungen und tatsächlichen Pfeilrichtungen (*scoreRightChoices*)

4.7 Schwierigkeitsgrade

Für die VR-Anwendung sind verschiedene Schwierigkeitsgrade implementiert. Der Schwierigkeitsgrad wirkt sich auf die Größe der spawnenden Würfel, die vorhandene maximale Reaktionszeit und einen Punkte-Multiplier aus, durch den deutlich höhere Punktzahlen bei den Schlägen erzielt werden können. Ursprünglich beeinflusste der Schwierigkeitsgrad auch den Radius der Kugelkoordinaten, aber der Radius wurde vom Schwierigkeitsgrad entkoppelt, weil er je nach Armlänge der Probanden frei eingestellt werden soll, damit ein Proband mit kürzeren Armen nicht im Nachteil ist. Aktuell gibt es die sieben Schwierigkeitsgrade *Just_Fun,Easy,Middle,Hard,Very_Hard,Extreme,Extra_Extreme*. Der Schwierigkeitsgrad *Middle* ist die Standardeinstellung, bei der der Punkte-Multiplier bei genau 1,0 liegt. Im Skript *DifficultyManager* können die Einstellungen für die verschiedenen Schwierigkeitsgrade verwaltet werden:

```
public float getPointModifier()
{
  switch(difficulty)
  {
    case Difficulty.Just_Fun: return 0.25f;
    case Difficulty.Easy: return 0.5f;
    case Difficulty.Middle: return 1.0f;
    case Difficulty.Hard: return 1.2f;
    case Difficulty.Very_Hard: return 1.5f;
    case Difficulty.Extreme: return 2.0f;
    case Difficulty.Extra_Extreme: return 3.0f;
    default: return 1.0f;
  }
}
```

Der Codeausschnitt zeigt die verschiedenen Punkte-Multiplier für alle definierten Schwierigkeitsgrade.

```
public float getTimeToHitObjects()
{
  switch (difficulty)
  {
    case Difficulty.Just_Fun: return 10f;
    case Difficulty.Easy: return 5f;
    case Difficulty.Middle: return 3f;
    case Difficulty.Hard: return 2.5f;
    case Difficulty.Very_Hard: return 2.0f;
    case Difficulty.Extreme: return 1.5f;
    case Difficulty.Extra_Extreme: return 0.7f;
    default: return 10f;
  }
}
```

Der Codeausschnitt zeigt die verschiedenen maximalen Reaktionszeiten für alle definierten Schwierigkeitsgrade.

```
 public float getScaleObjects()
{
  switch (difficulty)
  {
    case Difficulty.Just_Fun: return 0.38f;
    case Difficulty.Easy: return 0.35f;
    case Difficulty.Middle: return 0.3f;
    case Difficulty.Hard: return 0.25f;
    case Difficulty.Very_Hard: return 0.2f;
    case Difficulty.Extreme: return 0.18f;
    case Difficulty.Extra_Extreme: return 0.165f;
    default: return 0.4f;
  }
}
```

Der Codeausschnitt zeigt die verschiedenen Größen für die Würfel für alle Schwierigkeitsgrade. Bei Start der Haupt-Szene überschreibt der aktuell eingestellte Schwierigkeitsgrad die im SpawnCubes-Skript eingestellten Parameter zu Größe der Würfel und maximaler Reaktionszeit zum Zerschlagen der Würfel.

4.8 Schwierigkeiten bei der Implementierung der Schlagerkennung

Der erste Implementierungsansatz für die Schlagerkennung setzte auf verschiedene Collidergruppen, welche in fest definierter Reihenfolge durchschlagen werden mussten. Über die Reihenfolge des Betretens und Verlassens der verschiedenen Collidergruppen wurde sichergestellt, dass der mit einem Pfeil beschriftete Würfel auch gemäß der Richtung des Pfeils durchschlagen wurde. Eine Collidergruppe enthält zehn unterschiedlich große Collider, welche sich in der Form ähneln, aber absteigend immer kleiner werden. Die verschieden große Collider werden genutzt, um die Genauigkeit eines Schlags zu ermitteln. Die zehn Genauigkeitscollider in einer Collidergruppe sind so angeordnet, dass der größere Collider den nächstkleineren Collider enthält, d.h. der ungenaueste Collider einer Collidergruppe wird immer bei einem Schlagversuch, welcher den Würfel durchschlägt, getroffen. Je genauer bzw. präziser der Schlag war, desto mehr der inneren, kleineren Genaugikeitscollider werden beim Schlagversuch auch betreten und verlassen. So wird die Präzision für eine Collidergruppe dadurch ermittelt, dass das Betreten des kleinsten Genauigkeitscolliders registiert und entsprechend die Präzision gesetzt wird.

Dabei gab es zwei verschiedene Modelle von Würfeln, um 8 verschiedene Pfeilrichtungen darzustellen. Die ersten Modelle der Würfel werden für die Pfeilrichtungen links, rechts, oben und unten verwendet. Sie verwenden drei Collidergruppen, d.h. der Würfel ist in drei gleich große Segmente aufgeteilt, wobei bei einem Schlagversuch das Betreten und Verlassen der einzelnen Collider in den Collidergruppen überwacht wird. Das zweite Modell eines etwas komplexeren Würfels wurde für die Pfeilrichtungen diagonal rechts oben, diagonal links oben, diagonal rechts unten und diagonal links unten verwendet. Dieser Würfel enthält sogar fünf Collidergruppen, wobei die drei Collidergruppen des ersten Modells übernommen wurden, aber etwas kleiner skaliert wurden, sodass am Pfeilanfang und Pfeilende noch zwei zusätzliche, etwas kleinere Collidergruppen hinzugefügt werden konnten.
Das zweite, komplexere Modell für einen Würfel wurde entwickelt, um die Schlagerkennung weiter einzuschränken bzw. nur noch Schläge als gültig zu bewerten, welche auch wirklich die diagonale Schlagbewegung des vorgebenden Pfeils nachahmen.
Leider stellte sich beim Testen heraus, dass insbesondere schnelle vom Spieler durchgeführte Schläge durch den Würfel nicht immer zuverlässig erkannt werden. Das liegt wohl daran, dass bei einem sehr schnell durchgeführten Schlag die Physikengine von Unity nicht mehr zuverlässig das Betreten der verschiedenen Collidergruppen (bzw. Genauigkeitscollider einer Collidergruppe) registiert, sodass bei der Schlagerkennung eine Collidergruppe nicht betreten wurde und somit der Schlag als ungültig bewertet wird. Da die Physikengine von Unity immer nur in regelmäßigen Abständen innerhalb von fixedUpdate aufgerufen wird, kann es natürlich passieren, dass das Betreten eines Colliders verpasst wird, weil das Betreten dieses Colliders genau zwischen zwei Aufrufen von fixedUpdate passiert ist. Problematisch war außerdem, dass bei schnellen Schlägen die Auswertung für das Betreten der einzelnen Collidergruppen im Code nicht immer

synchron ausgeführt wurde, sodass beispielsweise das Betreten einer späteren Collidergruppe im Code zuerst ausgewertet wurde, obwohl das Auswerten der vorangehenden Collidergruppe im Code zuerst hätte erfolgen müssen, um die Auswertung für den Schlagversuch korrekt durchführen zu können.
Um dieses Problem zu umgehen, wurde eine zweite Erkennungsmethode entwickelt, die den Winkel zwischen zwei Vektoren berechnet und den Schlag als gültig akzeptiert, falls dieser Winkel kleiner oder gleich 45° ist. Der erste Vektor, *Idealvektor* genannt, beschreibt die perfekte Schlagrichtung, die mittig entlang des beschrifteten Pfeiles durch den Würfel hindurchgeht. Der zweite Vektor wird aus Eintritts- und Austrittspunkt in einen schmalen Physik-Collider bestimmt (siehe Kapitel Hit-Interactable-Skript).

5 Ergebnisse und Diskussion

5.1 Pilotstudie Aufbau

An der Pilotstudie nahmen 15 Probanden teil, darunter 11 männliche und 4 weibliche Teilnehmer. Das Durchschnittsalter der Probanden betrug 27,6 Jahre. Der Großteil der Probanden schätzte ihre sportliche Fitness überdurchschnittlich gut ein (Durchschnitt 4,66 auf einer Skala von 1 - 7). Mit Ausnahme von einem Probanden gaben alle Probanden an, regelmäßig Sport auszuüben. Der Großteil der Probanden hatte mit VR-Headsets wenig bis keine Erfahrung. Dementsprechend niedrig fallen die Durchschnittswerte aus, wie oft ein VR-Headset bisher aufgesetzt wurde (mit 2,066 auf einer Skala von 1 - 7), und wie gut man sich in VR zurechtfindet (mit 2,866 auf einer Skala von 1 - 7).
Im Rahmen der Pilotstudie sollte untersucht werden, ob sich die implementierte VR-Umgebung dazu eignet, das implizite Lernen von sequentiellen Bewegungen zu erforschen. Dabei musste jeder Proband einen Fragebogen mit Fragen zu biometrischen Daten (Alter, Geschlecht), sportlicher Fitness und Erfahrung in VR beantworten.
Zuerst bekam jeder Proband eine Einweisung zum Ablauf des Spiels und durfte in einem Tutorial-Level das Zerschlagen der Würfel ausprobieren. Darauf wurde die Hauptszene gestartet, in der die Probanden zehn Blöcke absolvierten mussten. In einem Block wurde eine (zuvor im Leveleditor erstellte) Sequenz bestehend aus neun Blockpaaren jeweils achtmal wiederholt. In den ersten sechs Blöcken wurde die zu lernende Sequenz von den Probanden unbewusst trainiert, nach dem Abschluss jedes Blockes gab es eine halbe Minute Pause für die Probanden. Nach Abschluss des sechsten Blockes pausierten alle Teilnehmer fünf Minuten und durften dabei auch die VR-Brille abnehmen. Die letzten vier Blöcke waren so aufgebaut, dass damit untersucht werden konnte, ob die Probanden die Sequenz implizit gelernt hatten. Dies lässt sich daran feststellen, ob die Probanden bei der gelernten Sequenz besser abschneiden als bei einer zufälligen Sequenz. In Block 7 und Block 10 zerschlugen die Probanden Würfel der gelernten Sequenz, während sie bei Block 8 und Block 9 die Würfel einer zufälligen Sequenz zerschlugen.
Bei der zufälligen Sequenz unterschieden sich sowohl die Positionen, in denen die Würfel erschienen, als auch die Pfeilrichtungen, mit denen die Würfel beschriftet waren, im Vergleich zu der gelernten Sequenz. Nachdem die Probanden den letzten Block abgeschlossen hatten, mussten sie im Anschluss daran noch einen Antizipationstest absolvieren, damit geprüft werden konnte, ob das gelernte Wissen zur Sequenz bei ihnen implizit oder explizit vorliegt.
Im Antizipationstest erschien neunmal ein zufälliges Würfelpaar aus der zu lernenden Sequenz (jedes Würfelpaar aus der Sequenz genau einmal in zufälliger Reihenfolge), welches vom Probanden zerschlagen werden durfte. Daraufhin war es die Aufgabe für die

Probanden sowohl die Positionen (über Zeigen mit Pointer) als auch die Pfeilrichtungen (über ein Auswahlmenü) des nächsten kommenden Würfelpaares zu bestimmen.
Alle wichtigen Daten (Schlagdaten mit Präzision, Reaktionszeit und Punktzahl sowie die gegebenen Antworten beim Antizipationstest) wurden aufgezeichnet und in .xml-Dateien gespeichert. Mit den erhobenen Daten soll nun statistisch geprüft werden, ob ein Lernerfolg nachzuweisen ist und falls ein Lernerfolg beobachtbar ist, ob die Sequenz auch implizit gelernt worden ist.

5.2 Zweistichproben-t-Test für verbundene Stichproben

Der t-Test prüft statistisch, ob die Erwartungswerte für die durchschnittlichen Punktzahlen der Blöcke mit der gelernten Sequenz größer sind als die durchschnittlichen Punktzahlen der Blöcke mit der zufälliger Sequenz.
Im Folgenden wird das Vorgehen beim Zweistichproben-t-Test für verbundene Stichproben beschrieben. Als Stichproben werden sowohl die durchschnittlich erreichten Punktezahlen für einen Durchlauf der Sequenz innerhalb eines Blocks als auch die besten erreichten Punktezahlen der Sequenz innerhalb eines Blocks verwendet. Da in einigen populären Sportarten nur der beste Versuch des Sportlers gewertet wird (beispielsweise Weitsprung, Diskuswurf, Hammerwerfen, etc.) wurden sowohl die durchschnittlich erreichten Punktzahlen wie auch die besten erreichten Punktzahlen eines Sequenzdurchlaufs als Leistungsdaten der Probanden betrachtet. Als Stichproben wurden die Werte aus Block 7 mit Block 8, sowie Block 9 mit Block 10, als auch Block 1 mit Block 6 verglichen. Block 7 und 10 enthielten dabei die von den Probanden gelernte Sequenz, während Block 8 und 9 zufällige Sequenzen enthielten. Vor Durchführung der Studie waren als Hypothesen/Vermutungen aufgestellt worden, dass die Probanden nach einer Trainingsphase mit der zu lernenden Sequenz in Blöcken mit dieser Sequenz besser abschneiden als in einem Block mit zufälligen Sequenzen. Die verwendeten Stichproben gelten als verbunden, weil die Werte, die miteinander verglichen werden, immer vom selben Probanden stammen (Jeder Proband absolviert hintereinander die 10 Blöcke). Um den t-Test durchführen zu können, werden als erstes die Differenzen der Ausprägungen der zwei Stichproben bestimmt:

$$Z_i = X_i - Y_i \quad \text{mit}$$

X_i : Ausprägungen der ersten Stichprobe (Block mit gelernter Sequenz)
Y_i : Ausprägungen der zweiten Stichprobe (Block mit zufälliger Sequenz)

Damit der Test aber durchgeführt werden darf, muss die Voraussetzung erfüllt sein, dass die Differenzen Z_i normalverteilt sind. Dies kann am besten durch den Shapiro-Wilk-Test geprüft werden, der auch für kleine n relativ gut funktioniert.
Mittels dieses Tests wurden für alle t-Tests die Variablen Z_i darauf getestet, ob sie normalverteilt sind (Ergebnis siehe Kapitel Ergebnisse des Zweistichproben-t-Tests). Die Nullhypothese lautet dann: $H_0 : \mu_1 \leq \mu_2$, weil dadurch der Fehler 1. Art und damit

auch die Wahrscheinlichkeit minimiert wird, fälschlicherweise die Gegenhypothese H_1 : $\mu_1 > \mu_2$ anzunehmen. μ_1 stellt den Erwartungswert für die durchschnittliche bzw. beste Punktzahl in dem Block mit der gelernten Sequenz dar. μ_2 dagegen stellt den Erwartungswert für die durchschnittliche bzw. beste Punktzahl in dem Block mit der zufälligen Sequenz dar. Die Nullhypothese H_0 ausformuliert besagt, dass der Erwartungswert für die durchschnittliche/beste Punktzahl im Block mit der gelernten Sequenz kleiner oder gleich dem Erwartungswert für die durchschnittliche/beste Punktzahl im Block mit der zufälligen Sequenz ist. Es wird also als Nullhypothese das Gegenteil dessen gewählt, was man eigentlich zeigen will. Für den T-Test muss der Wert der Teststatistik v bestimmt werden:

$$v = \frac{\bar{z}}{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(z_i - \bar{z})^2}}\sqrt{n} \quad \text{mit}$$

$\bar{z} = \frac{1}{n}\sum_{i=1}^{n} z_i$: Mittelwert der Differenzen
$s^2 = \frac{1}{n}\sum_{i=1}^{n}(z_i - \bar{z})^2$: Varianz der Differenzen Z_i

Die Nullhypothese wird genau dann abgelehnt, wenn v im Ablehnungsbereich $B = (z_{1-\alpha}; \infty)$ liegt, wobei $z_{1-\alpha}$ das Quantil der t-Verteilung mit Freiheitsgrad: n - 1 = 14 darstellt. Tabelle 5.1 zeigt die Werte der t(14)-Verteilung für einige Werte von α:

Tabelle 5.1: Ausschnitt der Werte der t(14)-Verteilung für einige Werte von α.

α	t-Verteilungs-Wert
5%	1,7613
2,5%	2,1448
2%	2,2638
1,5%	2,4149
1%	2,6245
0,1%	3,7874

Wird die Nullhypothese abgelehnt, so kann mit $(1 - \alpha)$-prozentiger Wahrscheinlichkeit davon ausgegangen werden, dass der Erwartungswert der Stichproben mit der zu lernenden Sequenz größer als der Erwartungswert der Stichproben mit der zufälligen Sequenz ist. Ist dies der Fall, konnte ein statistisch signifikanter Lernerfolg nachgewiesen werden. Im Folgenden wird für die Tests das Signifikanzniveau $\alpha = 0.01$ gewählt.

5.3 Ergebnisse des Zweistichproben-t-Tests

5.3.1 Prüfung der Voraussetzung für den Zwei-Stichproben-t-Test

Da als Voraussetzung für den Zweistichproben-t-Test die zugrundeliegende Grundgesamtheit normalverteilt sein muss, sind in Tabelle 5.2 die Ergebnisse des Shapiro-Wilk-Tests aufgelistet. Der Shapiro-Wilk-Test testet die Stichprobenwerte darauf, ob sie normalverteilt sind. Die Nullhypothese bei diesem Test lautet, dass eine Normalverteilung vorliegt. Solange der Wert der Teststatistik W größer als der kritische Wert $W_{kritisch}$ ist, bleibt die Annahme der Normalverteilung erhalten. Bei p = 0.01 ist $W_{kritisch} = 0.8350$, bei p = 0.05 ist $W_{kritisch} = 0.8810$, bei p = 0.10 ist $W_{kritisch} = 0.9010$ Die Wahrscheinlichkeit p drückt aus, wie wahrscheinlich es ist, dass die überprüften Daten tatsächlich aus einer normalverteilten Grundgesamtheit stammen.

Tabelle 5.2: Ergebnisse des Shapiro-Wilk-Tests

Blöcke	Wert der Teststatistik W	Wahrscheinlichkeit p
Blöcke 7 und 8 (avg)	0,94681	0.47567
Blöcke 10 und 9 (avg)	0,94353	0.42879
Blöcke 6 und 1 (avg)	0,95090	0.53868
Blöcke 7 und 8 (best)	0,95852	0.66672
Blöcke 10 und 9 (best)	0,97604	0.93526
Blöcke 6 und 1 (best)	0,93411	0.31400

5.3.2 Teststatistikwerte und Schlussfolgerungen

Da für den Zweistichproben-t-Test $\alpha = 0.01$ gewählt wurde, ergibt sich bei den Tests der Ablehnungsbereich B = $(2.6244, \infty)$. Es kann angemerkt werden, dass es sich bei den gemachten t-Tests um einseitige Tests handelt, weil hier nicht auf Gleichheit der Erwartungswerte geprüft wird, sondern darauf, dass einer der zwei Erwartungswerte größer als der andere ist (siehe vorheriges Kapitel). In Tabelle 5.3 sind die Werte der Teststatistik für alle gemachten Tests und die daraus folgende Schlussfolgerung für den Zweistichproben-t-Test aufgeführt:

Da bei allen Tests der Wert der Teststatistik v im Ablehnungsbereich B liegt, wird immer die Nullhypothese verworfen, dass der Erwartungswert für die Punktzahl im Block mit der gelernten Sequenz kleiner oder gleich der Punktzahl im Block mit der zufälligen Sequenz ist. Daraus folgt, dass die Gegenhypothese H_1 zu $\alpha = 0.01$ gilt. Also konnte statistisch gezeigt werden, dass der Erwartungswert für die (durchschnittliche/beste) Punktzahl im Block mit der gelernten Sequenz größer als im Block mit der zufälligen Sequenz ist. Zusätzlich wurden noch die Blöcke 1 und 6 während der Trainingsphase

Tabelle 5.3: Ergebnisse des Zweistichproben-t-Tests

Blöcke	Wert der Teststatistik v	Schlussfolgerung
Blöcke 7 und 8 (avg)	4,6642	H_0 wird verworfen
Blöcke 10 und 9 (avg)	6,3544	H_0 wird verworfen
Blöcke 6 und 1 (avg)	6,9213	H_0 wird verworfen
Blöcke 7 und 8 (best)	4,7701	H_0 wird verworfen
Blöcke 10 und 9 (best)	5,6402	H_0 wird verworfen
Blöcke 6 und 1 (best)	4,6875	H_0 wird verworfen

verglichen, wo in beiden Blöcken die zu lernende Sequenz verwendet wurde. Hier sollte gezeigt werden, dass durch einen erzielten Lern- und Trainingserfolg die Punktzahl im 6. Block stets größer als im 1. Block ist, da der Proband die Sequenz zuvor noch nicht gesehen und gelernt hat.

5.4 Bravais-Pearson-Korrelationskoeffizient

Um die Aussagekraft der Ergebnisse der Zweistichproben-t-Tests noch zu erhöhen, kann die Korrelation zwischen den Punktzahlen aus den zwei Blöcken, die miteinander verglichen wurden, bestimmt werden. Der Bravais-Pearson-Korrelationskoeffizient erfordert außerdem ein metrisches Skalenniveau (Kardinalskala) für die verwendeten Daten, was hier mit den von den Probanden erzielten Punktzahlen in den jeweiligen Blöcken gegeben ist. Der Korrelationskoeffizient ist folgendermaßen definiert:

$$Corr(X,Y) = \frac{Cov(X,Y)}{\sqrt{Var(X) \cdot Var(Y)}}$$

$$r = \frac{s_{X\,Y}}{s_X\ s_Y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})\ (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2\ \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Der Korrelationskoeffizient kann Werte zwischen -1 und 1 annehmen und misst den linearen, nicht kausalen Zusammenhang der beiden Merkmale. Dabei beudetet der Wert 1 perfekte Korrelation, $[0.7, 1)$ starke Korrelation, $[0.3, 0.7)$ mittlere Korrelation, $(..., 0.3)$ schwache Korrelation und ≈ 0 keine Korrelation. Negative Werte bedeuten eine inverse, entgegengesetzte Korrelation der beiden Merkmale (nimmt ein Merkmal zu, so nimmt das andere immer ab). In Tabelle 5.4 sind die Korrelationskoeffizienten für alle untersuchten Blockpaare (einmal mit durchschnittlichen Punktzahlen und einmal mit den besten Punktzahlen) aufgeführt.

Bei der Untersuchung der Blockpaare, bei denen durchschnittliche Punktzahlen verwendet wurden, lässt sich stets eine starke Korrelation zwischen den untersuchten Blöcken feststellen. Bei der Untersuchung der Blockpaare, bei denen die besten Punktzahlen herangezogen wurden, fällt dagegen die Korrelation deutlich schwächer aus und kann nur

Tabelle 5.4: Korrelationskoeffizientwerte

Blöcke	Korrelationskoeffizient r	Deutung von r
Blöcke 7 und 8 (avg)	0,8059	starke Korrelation
Blöcke 10 und 9 (avg)	0,7804	starke Korrelation
Blöcke 6 und 1 (avg)	0,7510	starke Korrelation
Blöcke 7 und 8 (best)	0,6521	mittlere Korrelation
Blöcke 10 und 9 (best)	0,6671	mittlere Korrelation
Blöcke 6 und 1 (best)	0,4302	mittlere Korrelation

noch als mittlere Korrelation gemessen werden. Die starke Korrelation bei den durchschnittlichen Punktzahlen zeigt, dass sich die allgemeine Leistungsfähigkeit eines Probanden in beiden Blöcken (gelernte Sequenz und zufällige Sequenz) auf die erzielten Punktzahlen auswirkt. Hat ein Proband eine sehr hohe durchschnittliche Punktzahl im Block mit der gelernten Sequenz erreicht, so hat er immer noch eine verhältnismäßíg recht hohe Punktzahl im Block mit der zufälligen Sequenz erreicht. Umgekehrt gilt auch, dass, wenn ein Proband im Block mit der zufälligen Sequenz eine recht niedrige durchschnittliche Punktzahl erzielt, er im Block mit der gelernten Sequenz nur eine etwas bessere Punktzahl erzielt, die verglichen mit den Punktzahlen anderer Probanden immer noch niedrig einzustufen ist. Folglich gibt es durchaus Probanden, die im Block mit der zufälligen Sequenz eine höhere durchschnittliche Punktzahl erreicht haben als andere Probanden im Block mit der gelernten Sequenz.

5.5 Analyse der Antizipationstestdaten

Die gesammelten Daten während des Antizipationstests müssen darauf untersucht werden, ob bei den Probanden explizites Wissen über die Sequenz vorhanden ist. Denn eigentlich soll mit der Studie getestet werden, ob sich die Probanden während der Studie das Wissen über die Sequenz implizit (unbewusst) angeeignet haben. Tabelle 5.5 zeigt die wichtigsten Daten für alle Teilnehmer der Studie zusammengefasst, sortiert nach aufsteigendem durchschnittlichem Abstand (Anzahl richtiger gegebener Antworten zur Pfeilrichtung für einen Würfel, Anzahl richtiger gegebener Antworten zur Pfeilrichtung für ein Würfelpaar, durchschnittlicher Abstand der geschätzten Position zur tatsächlichen Position und zuletzt die Anzahl richtiger geschätzter Positionen (hier werden Abstandswerte kleiner oder gleich 0.5 akzeptiert)).

Die beobachtete, mittlere Anzahl von richtigen Vorhersagen für die Pfeilrichtung des nächsten erscheinenden Würfels der entsprechenden Farbe beträgt 5,4.
Da bei der zu lernenden Sequenz 4 Pfeilrichtungen verwendet wurden (keine diagonalen Pfeile), beträgt die Wahrscheinlichkeit die Pfeilrichtung korrekt zu erraten 25%. Bei einer Sequenz bestehend aus 9 Würfelpaaren beträgt somit die erwartete Anzahl an richtiger Antworten durch Raten $\frac{1}{4} * 18 = 4.5$. Der Durchschnitt von 5,4 liegt hiermit leicht über dem erwarteten Wert von 4,5, was für eine gewisse Menge an explizitem Wissen

Tabelle 5.5: Antizipationstestdaten

Antworten (Würfel)	Antworten (Würfelpaar)	Avg. Abstand	korrekte Positionen
6	3	0,6783	6
6	0	0,7004	3
7	1	0,7120	4
10	4	0,7122	3
5	0	0,7592	3
6	2	0,7676	5
3	0	0,7853	3
2	0	0,8132	4
2	0	0,8288	1
9	3	0,8458	3
5	0	0,8470	5
5	1	0,8477	4
3	0	0,8608	1
5	1	0,8669	2
7	3	0,9795	2

sprechen könnte (insbesondere bei den Probanden mit 10 und 9 richtigen Antworten). Bei den richtig gegebenen Vorhersagen für die Pfeilrichtungen eines ganzen Würfelpaares (Pfeilrichtung für roten und Pfeilrichtung für blauen Würfel) liegt der Durchschnitt nur noch bei 1,2. Durch reines Raten wäre ein Durchschnitt von $\frac{1}{4} * \frac{1}{4} * 9 = \frac{9}{16} = 0.5625$ zu erwarten. Es ist leicht erkennbar, dass hier der Durchschnitt mit 1.2 deutlich über dem erwarteten Wert von 0.5625 liegt.
Das könnte darauf hinweisen, dass zumindest die Probanden mit jeweils 3 und 4 richtigen Würfelpaaren explizites Wissen zu einer Teilmenge der Sequenz besitzen. Während der Studie erhielt ich von einigen Probanden das Feedback, dass sie auch Würfelpaare korrekt erraten hätten, aber manchmal gaben sie auch an, sich an gewisse Bestandteile (Würfelpaare) der Sequenz erinnern zu können und waren sich sicher, dass ihre Auswahl der Pfeilrichtungen korrekt ist, bevor das Ergebnis gezeigt wurde.
Der Durchschnittswert für den Abstand der geschätzten Positionen zu den tatsächlichen Positionen der Würfel des nächsten Würfelpaares beträgt 0,80. Es wurden alle Positionen als richtige Positionen gewertet, welche einen Abstand kleiner oder gleich 0,5 zu den tatsächlich als nächstes kommenden, zugehörigen Würfel aufweisen. Durchschnittlich wurden 3,266 richtige Positionen von den Probanden gewählt (Minimum 1, Maximum 6). Für die Positionsdaten ist es schwieriger eine Grenze zu setzen, welche Anzahl an richtigen Antworten ohne explizites Wissen für die Sequenz zu erwarten ist. An den Einzelabstandswerten für alle Probanden ist auffällig, dass teils relativ niedrige Abstände (im Bereich von 0.25-0.4) vorkommen, aber die Mehrheit mehr im Bereich von (0.7-1.1) anzusiedeln ist. Die Kantenlänge eines während der Studie gespawnten Würfels betrug 0.3 (Unity-Längeneinheiten). Die Kantenlänge der gespawnten Würfel wird durch den eingestellten Schwierigkeitsgrad festgelegt (siehe Kapitel Schwierigkeitsgrade). Der

Großteil der von den Probanden geschätzten Positionen weist somit eine relativ große Entfernung zu den richtigen Positionen auf. Es ist also davon auszugehen, dass explizites Wissen zu den Positionen der erscheinenden Würfel in der Sequenz kaum bis nicht vorhanden ist. Die Steigerung in der Reaktionszeit für die gelernte Sequenz kann durch implizites Wissen zur Sequenz erklärt werden, welches die Probanden während des Tests unbewusst ausnutzen.

5.6 Weitere Analyse der Studiendaten

Untersucht man die Präzisionsdaten und Reaktionszeiten genauer, die sich ausschlaggebend auf die erzielten Punkte in den verschiedenen Blöcken auswirken, erkennt man, dass die im Durchschnitt erzielten Präzisionswerte in den Blöcken 7 und 10 nur leicht höher als die Präzisionswerte in den Blöcken 8 und 9 sind (0.72 und 0.72 zu 0.70 und 0.70). Dagegen fällt der Unterschied bei den Reaktionszeiten deutlich größer aus, hier stehen Reaktionszeiten von 0.98 und 0.98 denen von 1.21 und 1.25 gegenüber. Also wird durchschnittlich ein Würfel in einem Block mit der gelernten Sequenz etwa 0.2 Sekunden schneller zerschlagen als in einem Block mit einer zufälligen Sequenz. Hieraus lässt sich schlussfolgern, dass die erhöhten Punktzahlen in den Blöcken mit der gelernten Sequenz im Vergleich zu den Blöcken mit der zufälligen Sequenz vor allem durch die deutlich schnelleren Reaktionszeiten zustande kommen.
Die Durchschnittswerte für Präzision und Reaktionszeit für alle 15 Probanden sind in Tabelle 5.6 aufgeführt. 1.0 stellt dabei die perfekte Präzision dar und 0.0 bedeutet, dass der Würfel falsch zerschlagen wurde. Bei korrekt zerschlagenen Würfeln sind Präzisionswerte zwischen 0.1 und 1.0 in Abstufungen von 0.1-Schritten möglich. Bei den Reaktionszeiten sind theoretisch Werte zwischen 0.0 und 3.0 Sekunden möglich, allerdings ist für einen Probanden in der Praxis ein Wert von unter 0.7 Sekunden als durchschnittlicher Wert für die Reaktionszeit nur schwer zu erreichen.

Tabelle 5.6: Durchschnittswerte für Präzision und Reaktionszeit

Blocknummer	Mittelwert für Präzision	Mittelwert für Reaktionszeit
Block 1	0,6234	1,2021
Block 2	0,6832	1,0532
Block 3	0,6762	1,0385
Block 4	0,7003	1,0301
Block 5	0,7374	1,0222
Block 6	0,7247	0,9814
Block 7	0,7250	0,9775
Block 8	0,7069	1,2107
Block 9	0,7014	1,2475
Block 10	0,7281	0,9764

Für die zu lernende Sequenz verringert sich die Reaktionszeit konstant mit jedem weiteren Übungsblock, bis im letzten Block 10 das Minimum mit 0.9764 erreicht wird. Bei den Präzisionswerten fällt dagegen auf, dass nach den initialen ersten 3 Blöcken die Präzision sich stets auf einem Niveau von mindestens 0.70 bewegt und leichte Schwankungen zu beobachten sind. Die initial deutlich niedrigeren Präzisionswerte lassen sich vermutlich dadurch erklären, dass die Probanden am Anfang Übungszeit benötigen, um mit der Technik und Funktionsweise des VR-Spiels vertraut zu werden (insbesondere das Zerschlagen der Würfel). Das Maximum wird hier in Block 5 mit 0.7374 erreicht, aber die Präzisionswerte in den späteren Blöcken 7 und 10 liegen mit 0.7250 und 0.7281 auch nur leicht darunter.

Tabelle 5.7 enthält Informationen darüber, wie gut sich die einzelnen Probanden bei der Studie geschlagen haben, sortiert nach Abschneiden mit dem Highscore(der Gesamtpunktzahl). Hier ist die erreichte Highscore (Summe aller Punkte über alle gemachten Schläge), der beste Versuch (als Punktzahl) für eine Sequenz von Würfeln (9 Würfelpaare), die Anzahl sehr guter Schläge (Schläge, welche mehr als 100 Punkte eingebracht haben, was nur mit sehr hoher Präzision und schneller Reaktionszeit möglich ist), die Anzahl verpasster Würfel (also Würfel, welche nicht rechtzeitig innerhalb von 3 Sekunden zerschlagen wurden) und die Anzahl falsch zerschlagener Würfel (misslungene Schläge) aufgeführt:

Tabelle 5.7: Highscoreliste der Probanden

Highscore	Bester Versuch	gute Schläge	verpasste Würfel	misslungene Schläge
121482	1710	324	2	27
118511	1666	194	9	29
117779	1650	221	5	14
117010	1694	227	5	41
114671	1674	268	4	60
113603	1626	141	2	23
111465	1609	107	25	15
109101	1603	197	1	104
108396	1693	264	31	156
106693	1621	158	5	99
105061	1624	112	16	96
101893	1595	113	7	115
95565	1556	164	10	158
95555	1464	76	24	88
86901	1516	115	5	258

Die Werte für die Highscore rangieren von 86900 bis 121400 (größter Unterschied 29%), beim besten Versuch für eine Sequenz von Würfelpaaren liegen die Werte im Bereich von 1464 und 1710 (größter Unterschied 15%).

Erwartungsgemäß haben die Probanden weit oben in der Highscore-Liste eine hohe Anzahl an sehr guten Schlägen mit einer Punktzahl über 100 Punkte für einen zerschlagenen Würfel. Hier bewegt sich die Zahl für alle Probanden zwischen 76 und 324. Die Anzahl an verpassten Würfeln fällt bei allen Probanden sehr gering aus (1 - 31), weil 3 Sekunden in den meisten Fällen mehr als genug Zeit bieten, die zwei Würfel zu zerschlagen. Bei der Anzahl an misslungenen Schlägen fällt auf, dass während die oberen in der Highscoreliste eine relativ geringe Anzahl an Fehlschlägen (14 - 60) aufweisen, die untere Hälfte deutlich mehr Fehlschläge (88 - 258) verzeichnen muss. Da Jeder Fehlschlag mit 0 Punkten bewertet wird, wirkt sich die Anzahl an Fehlschlägen deutlich auf die am Ende erzielte Highscore aus. Der beste Versuch für eine Sequenz an Würfelpaaren kann daher als Alternative für das Abschneiden der Probanden betrachtet werden. Aber auch hier zeigt sich die Tendenz, dass die Probanden mit einem hohen Highscore in den meisten Fällen einen besseren Versuch aufweisen als die Probanden in der unteren Hälfte mit vielen Fehlschlägen. Folglich würde sich die Reihenfolge der Probanden tendenziell nicht stark ändern (Probanden, welche vorher oben in der Liste waren, sind auch hier immer noch oben. Probanden, welche vorher eher unten in der Liste zu finden waren, sind es auch hier mit Ausnahme von einem Probanden).

5.7 Konklusion

In dieser Arbeit wurde ein funktionsfähiges VR-Exergame entwickelt, mit dessen Hilfe getestet werden kann, ob Probanden implizit Sequenzen motorischer Schlagbewegungen lernen. Im Rahmen der Durchführung der Pilotstudie wurde getestet, inwiefern sich die implementierte VR-Anwendung zum Untersuchen der Forschungsfrage eignet. Aus dem gesammelten Daten ging durch statistisch signifikantem Zweistichproben-t-Test hervor, dass die durch die Sequenz vorgegebenen Schlagbewegungen von den Probanden gelernt werden. In den Trainingsblöcken mit der zu lernenden Sequenz konnte stets ein Leistungszuwachs beobachtet werden und die Leistung in den Blöcken mit der gelernten Sequenz war konstant besser im Vergleich zu einem Block mit zufälligen Sequenzen. Die Analyse der Antizipationstest-Daten legen nahe, dass geringe Anteile von explizitem Wissen zur Sequenz vorhanden sind. Insgesamt war bei den Probanden mehr explizites Wissen für die Pfeilrichtungen als für die Positionen der erscheinenden Würfel zu verzeichnen. Die Analyse der Präzision- und Reaktionszeitdaten in den Blöcken zeigt, dass sich das implizite Lernen der Sequenz vor allem durch verbesserte Reaktionszeiten bemerkbar macht. So konnten die Probanden nach der Trainingsphase mit der zu lernenden Sequenz (Blöcke 1 bis 6) in den Blöcken mit der gelernten Sequenz vor allem durch deutlich schnellere Reaktionszeiten verglichen mit dem Block einer zufälligen Sequenz mehr Punkte erzielen. Auch wenn geringe explizite Wissensanteile bei wenigen Probanden vorhanden sind, kann kann davon ausgehen, dass das Lernen der Sequenz von Schlagbewegungen bei den Probanden implizit erfolgt ist. Bei der Durchführung der Studie wurde darauf geachtet, dass die Probanden mit der untersuchten Forschungsfrage nicht vertraut sind. Weiterhin gaben die meisten Probanden an, den sequenzierten Aufbau der Blöcke nicht zu bemerken, was sich als starkes Indiz für implizites Lernen

erweist.
Das in dieser Arbeit entwickelte Spiel bietet einige Möglichkeiten für zukünftige Arbeiten. So könnten noch weitere Studien durchgeführt werden, in denen alle acht Pfeilrichtungen verwendet werden und in denen die Probanden die Möglichkeit haben, den Schwierigkeitsgrad nach Belieben anzupassen. Außerdem könnte man das Abschneiden der Probanden mit selbst eingestelltem Schwierigkeitsgrad mit dem durch Reinforcement Learning dynamisch angepasstem Schwierigkeitsgrad vergleichen.

Literatur

[1] Mary Jo Nissen und Peter Bullemer. „Attentional requirements of learning: Evidence from performance measures“. In: *Cognitive Psychology* 19.1 (Jan. 1987), S. 1–32. ISSN: 00100285. DOI: 10.1016/0010-0285(87)90002-8. URL: https://linkinghub.elsevier.com/retrieve/pii/0010028587900028 (besucht am 19. 01. 2021).

[2] Willem B. Verwey und Benjamin A. Clegg. „Effector dependent sequence learning in the serial RT task“. In: *Psychological Research Psychologische Forschung* 69.4 (März 2005), S. 242–251. ISSN: 0340-0727, 1430-2772. DOI: 10.1007/s00426-004-0181-x. URL: http://link.springer.com/10.1007/s00426-004-0181-x (besucht am 05. 10. 2021).

[3] Danielle E. Levac, Meghan E. Huber und Dagmar Sternad. „Learning and transfer of complex motor skills in virtual reality: a perspective review“. In: *Journal of NeuroEngineering and Rehabilitation* 16.1 (Dez. 2019), S. 121. ISSN: 1743-0003. DOI: 10.1186/s12984-019-0587-8. URL: https://jneuroengrehab.biomedcentral.com/articles/10.1186/s12984-019-0587-8 (besucht am 05. 10. 2021).

[4] Stephan P. Swinnen und Nicole Wenderoth. „Two hands, one brain: cognitive neuroscience of bimanual skill“. In: *Trends in Cognitive Sciences* 8.1 (Jan. 2004), S. 18–25. ISSN: 13646613. DOI: 10.1016/j.tics.2003.10.017. URL: https://linkinghub.elsevier.com/retrieve/pii/S1364661303002961 (besucht am 05. 10. 2021).

[5] V. Schmidtke und H. Heuer. „Task integration as a factor in secondary-task effects on sequence learning“. In: *Psychological Research* 60.1 (Juni 1997), S. 53–71. ISSN: 0340-0727, 1430-2772. DOI: 10.1007/BF00419680. URL: http://link.springer.com/10.1007/BF00419680 (besucht am 05. 01. 2021).

[6] Hillary Schwarb und Erich Schumacher. „Generalized lessons about sequence learning from the study of the serial reaction time task“. In: *Advances in Cognitive Psychology* 8.2 (28. Juni 2012), S. 165–178. ISSN: 18951171. DOI: 10.5709/acp-0113-1. URL: http://www.ac-psych.org/en/download-pdf/volume/8/issue/2/id/113 (besucht am 05. 01. 2021).

[7] Eva Röttger u. a. „Why Does Dual-Tasking Hamper Implicit Sequence Learning?“ In: *Journal of Cognition* 4.1 (7. Jan. 2021), S. 1. ISSN: 2514-4820. DOI: 10.5334/joc.136. URL: http://www.journalofcognition.org/articles/10.5334/joc.136/ (besucht am 06. 10. 2021).

[8] Peter A Frensch und Dennis Rünger. „Implicit Learning“. In: *CURRENT DIRECTIONS IN PSYCHOLOGICAL SCIENCE* 12.1 (2003), S. 6.

[9] Barbara J. Knowlton, Alexander L.M. Siegel und Teena D. Moody. „Procedural Learning in Humans “. In: *Learning and Memory: A Comprehensive Reference*. Elsevier, 2017, S. 295–312. ISBN: 978-0-12-805291-4. DOI: 10.1016/B978-0-12-809324-5.21085-7. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780128093245210857 (besucht am 07.04.2021).

[10] Asher Cohen, Richard I Ivry und Steven W Keele. „Attention and Structure in Sequence Learning“. In: *SEQUENCE LEARNING* (1990), S. 14.

[11] Antoine Pasquali, Axel Cleeremans und Vinciane Gaillard. „Reversible second-order conditional sequences in incidental sequence learning tasks“. In: *Quarterly Journal of Experimental Psychology* 72.5 (Mai 2019), S. 1164–1175. ISSN: 1747-0218, 1747-0226. DOI: 10.1177/1747021818780690. URL: http://journals.sagepub.com/doi/10.1177/1747021818780690 (besucht am 06.01.2021).

[12] David R. Shanks und Theresa Johnstone. „Implicit knowledge in sequential learning tasks“. In: *Handbook of implicit learning*. Thousand Oaks, CA, US: Sage Publications, Inc, 1998, S. 533–572. ISBN: 978-0-7619-0197-6.

[13] Jürgen Zervos-Kopp. *Ergotherapie Prüfungswissen – Anatomie, Biologie und Physiologie*. 2. Aufl. 2009.

[14] *Effektor (Physiologie)*. In: *Wikipedia*. Page Version ID: 210046679. 21. März 2021. URL: https://de.wikipedia.org/w/index.php?title=Effektor_(Physiologie)&oldid=210046679 (besucht am 07.10.2021).

[15] Daniel B. Willingham. „Implicit motor sequence learning is not purely perceptual“. In: *Memory & Cognition* 27.3 (Mai 1999), S. 561–572. ISSN: 0090-502X, 1532-5946. DOI: 10.3758/BF03211549. URL: http://link.springer.com/10.3758/BF03211549 (besucht am 20.01.2021).

[16] James H. Howard, Sharon A. Mutter und Darlene V. Howard. „Serial pattern learning by event observation“. In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 18.5 (1992). Place: US Publisher: American Psychological Association, S. 1029–1039. ISSN: 1939-1285(Electronic),0278-7393(Print). DOI: 10.1037/0278-7393.18.5.1029.

[17] Neal J. Cohen und Howard Eichenbaum. *Memory, amnesia, and the hippocampal system*. MIT Press, 1993. ISBN: 978-0-262-53132-0 978-0-262-03203-2. URL: https://experts.illinois.edu/en/publications/memory-amnesia-and-the-hippocampal-system (besucht am 20.01.2021).

[18] Thomas Goschke. „Implicit learning of perceptual and motor sequences: Evidence for independent learning systems“. In: *Handbook of implicit learning*. Thousand Oaks, CA, US: Sage Publications, Inc, 1998, S. 401–444. ISBN: 978-0-7619-0197-6.

[19] Gabriele Wulf und Rebecca Lewthwaite. „Optimizing performance through intrinsic motivation and attention for learning: The OPTIMAL theory of motor learning“. In: *Psychonomic Bulletin & Review* 23.5 (Okt. 2016), S. 1382–1414. ISSN: 1069-9384, 1531-5320. DOI: 10.3758/s13423-015-0999-9. URL: http://link.springer.com/10.3758/s13423-015-0999-9 (besucht am 05.01.2021).

[20] F. G. Ashby, A. M. Isen und U. Turken. *A neuropsychological theory of positive affect and its influence on cognition.* 1999.

[21] G. Dreisbach und T. Goschke. *How positive affect modulates cognitive control: Reduced perseveration at the cost of increased distractibility.* Bd. Journal of Experimental Psychology: Learning, Memory, and Cognition, 30, 343–353. 2004.

[22] S. Lyubomirsky, L. King und E. Diener. *The benefits of frequent positive affect: Does happiness lead to success?* 2005.

[23] Jason Gregory. *Game engine architecture.* Third edition. Boca Raton: Taylor und Francis, CRC Press, 2018. 1 S. ISBN: 978-1-351-97427-1 978-1-351-97428-8.

[24] Raph Koster. *A Theory of Fun for Game Design.* Phoenix, AZ: Paraglyph Press, 2004.

[25] Unity Technologies. *Unity - Manual: Continuous collision detection (CCD).* URL: https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html (besucht am 03. 10. 2021).

[26] Unity Technologies. *Unity - Manual: Colliders.* URL: https://docs.unity3d.com/Manual/CollidersOverview.html (besucht am 03. 10. 2021).

[27] A Alexandrov D. *Die innere Geometrie der konvexen Flächen.* Akademie-Verlag Berlin, 1955.

[28] *Konvexe und konkave Fläche.* In: *Wikipedia.* Page Version ID: 211784230. 9. Mai 2021. URL: https://de.wikipedia.org/w/index.php?title=Konvexe_und_konkave_Fl%C3%A4che&oldid=211784230 (besucht am 04. 10. 2021).

[29] Franco P. Preparata und Michael Ian Shamos. *Computational Theory - An Introduction.* Springer-Verlag, 1985. ISBN: 0-387-96131-3.

[30] *Konvexe Hülle.* In: *Wikipedia.* Page Version ID: 215648181. 16. Sep. 2021. URL: https://de.wikipedia.org/w/index.php?title=Konvexe_H%C3%BClle&oldid=215648181 (besucht am 04. 10. 2021).

[31] Diederick C. Niehorster, Li Li und Markus Lappe. „The Accuracy and Precision of Position and Orientation Tracking in the HTC Vive Virtual Reality System for Scientific Research“. In: *i-Perception* 8.3 (Juni 2017), S. 204166951770820. ISSN: 2041-6695, 2041-6695. DOI: 10.1177/2041669517708205. URL: http://journals.sagepub.com/doi/10.1177/2041669517708205 (besucht am 02. 07. 2021).

[32] Blender Foundation. *blender.org - Home of the Blender project - Free and Open 3D Creation Software.* blender.org. URL: https://www.blender.org/ (besucht am 03. 10. 2021).

[33] Ethan Luckett. „A Quantitative Evaluation of the HTC Vive for Virtual Reality Research“. In: (), S. 31.

[34] *VRTK - Virtual Reality Toolkit.* VRTK - Virtual Reality Toolkit. URL: https://vrtoolkit.readme.io/ (besucht am 03. 10. 2021).