Redirected Walking in zufallsgenerierten Virtual Reality Leveln

Autor:
Karim Djemai

Matrikel: 6911548

Mensch-Computer Interaktion
Fachbereich Informatik

Erstgutachter: Prof. Dr. Frank Steinicke
Zweitgutachter: Dr. Eike Langbehn

Hamburg, 14. September 2021

Abstract

One of the greatest challenges in dealing with modern "virtual reality" (VR) technology is the limitation of the accessible area (tracking space). There are different solutions to this problem. In the experiment presented here, three of them are compared. On the one hand, conventional methods of locomotion in VR (joystick control and teleportation), and on the other hand, a relatively new method of locomotion called "redirected walking". This is an umbrella term for methods in which the user is guided through the real tracking space by subtle changes to the virtual locomotion, while maintaining the illusion that they are moving without any changes. These methods can be combined in different ways and thus are adaptable to the context. This work investigates a certain context: automatically generated levels. By combining so-called "rotational gains" and so-called "impossible spaces" and incorporating them into a level generation algorithm, I have succeeded in creating theoretically infinitely large levels through which the user can move even though she is located in a limited tracking space. This paper presents an experiment that investigates whether the user's spatial understanding forms itself better with this method of locomotion as the user moves through the level, and whether the sense of presence in virtual reality is higher, compared to the more traditional methods of locomotion.

Zusammenfassung

Eine der größten Hürden im Umgang mit moderner “virtual reality” (VR) Technologie ist die Begrenzung des begehbaren Bereiches (Trackingspace).

Es gibt unterschiedliche Lösungsansätze für dieses Problem. In dem hier vorgestellten Experiment werden drei davon miteinander Verglichen. Zum einen konventionelle Fortbewegungsarten für VR (Joystick-Steuerung und Teleportieren), zum anderen eine verhältnismäßig neue Fortbewegungsart mit dem Namen “Redirected Walking”. Hierbei handelt es sich um einen Sammelbegriff für Methoden bei denen die Nutzer:in durch subtile Änderungen an der VR-Fortbewegung durch den realen Trackingspace gelenkt wird, dabei aber die Illusion aufrecht erhalten wird, sie würde sich ohne Änderung fortbewegen. Diese Methoden lassen sich auf unterschiedliche Weisen kombinieren und sind so dem Kontext flexibel anpassbar. Diese Arbeit untersucht dabei einen konkreten Kontext: Automatisch generierte Level. Durch die Kombination von so genannten “Rotationgains” und sogenannten “Impossible Spaces” und die Inkorporation davon in einen Levelgenerirungsalgorithmus ist es mir gelungen, theoretisch endlos große Level zu erschaffen, durch die sich die Nutzer:in fortbewegen kann obwohl sie sich nur in einem Begrenzt großen Trackingspace befindet. Diese Arbeit stellt ein experiment vor, das untersucht ob das räumliche Verständnis der Nutzer:innen mit dieser Fortbewegungsart beim durchschreiten des Levels besser gebildet wird, und ob das Präsenz Gefühl in der virtuellen Realität höher ist, als mit den herkömmlicheren Fortbewegungsmethoden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Zweibeiniges Gehen bietet als Fortbewegungsart durch virtuelle Umgebungen viele Vorteile gegenüber alternativen Fortbewegungsarten.[

] beschreibt beispielsweise, ein höheres Präsensgefühl der Nutzer:innen.[

] zeigt dass weniger Motion sickness entsteht wenn sich die Nutzer:innen durch die virtuelle Welt bewegen indem sie gehen. [

] erklärt, dass beim Gehen mehr Sinne stimuliert werden als bei künstlichen Alternativen, wie zum Beispiel der Joystick Steuerung. Tiefensensibilität (Propriozeption) und Gleichgewichtssinn (vestibuläre Wahrnehmung) signalisieren, dass er gerade wirklich geht, während diese Information bei alternativen Fortbewegungsart allein vom visuellen Sinn übermittelt wird. Leider bring das reale gehen (real walking nach[

]) auch den großen Nachteil mit sich, dass es in der Regel auf einen einzelnen Raum (den Trackingspace) beschränkt ist.

Dies entsteht zum einen, durch räumlich limitierte Erfassung (auf Englisch: tracking) der Position und Rotation des Headsets und der Controller bei einigen Technologien (zum Beispiel den Modellen der "HTC VIVE"-Produktreihe), zum anderen durch die Raumgröße der meisten VR-Setups.

Zwar gibt es dazu auch Ausnahmen, (siehe z.B.[ ]), jedoch sind diese dann mit großem Aufwand verbunden und nicht für jede Endnutzer:in umzusetzen.

1.1 Redirections Techniques

Eine Herangehensweise dieses Problem zu Umgehen sind so genannte "Redirection Techniques" (besser bekannt als Redirected Walking, diese Begriffe werden oft austauschbar verwendet (nach [

])). Dies ist ein Sammelbegriff für Techniken bei denen die Nutzer:in mit Manipulationen der Fortbewegungsart durch den Trackingspace navigiert wird. So lässt sich die Nutzer:in von den äußeren Begrenzungen des Trackingspaces fern halten, und die virtuell begehbare Fläche vergrößern. Im folgenden werde ich nun zwei dieser Techniken genauer vorstellen.

1.1.1 Rotationgains

Rotationgains werden Kopfrotationen hinzugefügt sodass sich die virtuelle Kamera leicht schneller oder langsamer dreht als der reale Kopf mit dem VR-Headset. Kopfrotationen lassen sich mit der Schreibweise

$$R_{real} := (pitch_{real}, yaw_{real}, roll_{real})$$

darstellen, wobei pitch, yaw und roll die Eulerschen Winkel der Kopfrotation darstellen. Der Rotationgain wird dann als Quotient des virtuellen Winkels und des realen Winkels definiert also:

$$gR := \frac{R_{virtual}}{R_{real}}$$

Für alle 3 Winkel kann ein Rotationgain angewandt werden. Dieses Anwenden funktioniert indem der Rotationgain gR mit dem Winkel der realen Kopfrotation α multipliziert wird also:

$$gR * \alpha$$

Da für jeden Winkel der Kopfrotation ein Rotationgain definiert werden kann werden Rotation gains folgendermaßen dargestellt:

$$(gR_{pitch}, gR_{yaw}, gR_{roll})$$

In der Regel wird für Redirection ein Rotationgain auf den yaw_{real} Winkel der Kopfrotation angewandt. [ ] Durch anwenden eines rotation gains kann der virtuelle Trackingspace um den realen Trackingspace mit dem Drehpunkt der Nutzerposition herum rotiert werden. Für den Nutzer kann so die Illusion entstehen er würde über die Grenzen des Trackingspaces hinaus schreiten können, ohne dies zu tun. (Siehe grafik)

1.1.2 Impossible Spaces

Um den begehbaren Bereich eines Trackingspace noch weiter zu vergrößern haben sich Suma et al. [18] eine Technik ausgedacht bei der zwei oder mehr Räume in überlappenden Flächen liegen, allerdings nur einer zur Zeit angezeigt wird. Es gibt dann unterschiedliche Bedingungen, wann welcher der Räume angezeigt wird. Beispiels weise wird Raum A nur angezeigt wenn die Nutzer:in den überlappenden Raum durch Tür a betritt und Raum b, wenn sie ihn durch Tür b betritt. Dafür ist es also notwendig x verschiedene states zu setzen sodass immer einer x verschiedener Räume angezeigt wird. Des weiteren ist es Notwendig einen Bereich zu erschaffen in dem zwischen den states gewechselt werden kann, ohne dass die Nutzer:in es merkt.

1.2 Generierte Level

Klassischer weise werden Level in Computerspielen und virtuellen Umgebungen von Leveldesignern designed. Dies erfordert Zeit und know-how. Der Arbeitsaufwand wächst

(linear) mit der Größe des Levels, deshalb ist es unmöglich endlos große Level zu erschaffen. Eine alternative Levelerstellungsweise ist das so genannte “Prozedurale Generieren”(Auch “Prozedurale Synthese” genannt. Dabei wird das Level von einem Algorithmus erschaffen, und kann somit endlos große Welten erschaffen.

In dem hier vorgestellten Experiment

Kapitel 2

Verwandte Arbeiten

In diesem Kapitel werden wissenschaftliche Arbeiten vorgestellt mit denen diese Arbeit zusammenhängt. Dabei werde ich zunächst auf solche Arbeiten eingehen, die sich mit dem Thema Real-Walking in virtuellen Umgebungen beschäftigen, danach verschiedene redirection Techniken vorstellen und dann auf das Thema der Level-Generierung eingehen. Zunächst stelle ich generelle Arbeiten zu dem Thema Real-Walking und dann zu Redirection Techniken vor, danach gehe ich konkreter auf die in dieser Studie sehr im Fokus liegenden Rotationgains ein um danach die auch in dieser Studie genutzten Impossible Spaces vorzustellen. Als nächstes stelle ich dem Leser noch Arbeiten vor die sich damit beschäftigt haben Level auf eine automatische Art und Weise zu generieren. Dabei werde ich mich sowohl mit Artikeln über die genauen Definition dieses Bereichs beschäftigen, als auch eine Taxonomie zur Einordnung von Prozeduren zur Inhaltsgenerierung zitieren. Die Kombination von generierten Leveln und Redirection-Techniken führen zu dem sogenannten "Infinite-Walking". Mit den Arbeiten zu diesem Thema wird das Kapitel abgeschlossen.

2.1 Real-Walking

1995 zeigen Slater et al. [16], dass Proband:innen eine höheres Präsensgefühl zeigten wenn sie die von ihm vorgestellte Technik "Walking-In-Place" nutzen als wenn sie per Knopfdruck durch die Welt bewegten. Hierbei handelte es sich um eine virtual-walking Technik bei der die Proband:innen ein eine Gehbewegung simulierten die dann digital erfasst und in Virtuell Fortbewegung umgewandelt wurde. Dieses Experiment wurde 1999 von Usoh et al. [24] repliziert, wobei nun die Option wirklich zu gehen ("Real-Walking") gegeben war. Dabei hatten die Proband:innen nochmal ein signifikant höheres Präsensgefühl, als bei den beiden anderen Optionen (Virtual-Walking und Push-Button-Fly). Des weiteren zeigen Arbeiten wie [15] und [1], dass virtuelle Fortbewegungsarten, die anders als real-walking nicht den vestibulären Sinn und die Propriozeption stimulieren, wahrscheinlicher die sogenannte "Simulator-Sickness" auslösen und, dass die User:innen damit weniger effektiv navigieren.

Wenn Designer eine Real-Walking-Umgebung erstellen müssen sie dabei schon die Dimensionen des Trackingspaces kennen. Da man aber nicht davon ausgehen kann, dass unterschiedliche Nutzer:innen gleiche Trackingspacedimensionen zur Verfügung haben

entsteht ein Problem, dass Marwecki et al. in ihrer Arbeit [10] zu Lösen versuchen. Sie stellen dabei das Softwaresystem "Scenograph" vor, welches große virtuelle Umgebungen in mehrere kleinere, teilweise anders geformte, Umgebungen, mit prozedural generierten Verbindungen, aufteilt ohne dabei die narrative Struktur der Ursprünglichen Umgebung zu verändern.

Allerdings gibt es auch andere Ansätze um Nutzer:innen mit begrenztem Trackingspaceplatz Real-Walking-Erfahrungen zu ermöglichen, wie beispielsweise den virtuellen Bereich, der von der Nutzer:in begehbar ist, zu vergrößern. Eine vielversprechende Art dies zu erreichen sind Redirection-Techniken.

2.2 Redirection Techniken

Razzaque et al. [14] stellten 2001 die Technik des "Redirected-Walking" vor, bei der die Nutzer:innen unwissentlich durch den Trackingspace gelenkt werden, dabei aber die Illusion entsteht, sie würden sich über die Grenzen dessen hinausbewegen. Die Technik basiert darauf, dass der visuelle Sinn dominanter ist als andere Sinne, mit denen man seine Orientierung im Raum bestimmen kann [6]. Seit dem gibt es zahlreiche weitere Techniken um den selben Effekt zu erzielen oder um ihn weiterzuentwickeln. Der Ansatz die verschiedenen Manipulationseffekte als "Gains" zu beschreiben findet sich bei Steinicke et al. [17]. Dort wird untersucht wie subtil diese Manipulationen sein müssen um nicht von der Nutzer:in erkannt zu werden.

Es konnte gezeigt werden, dass Proband:innen in virtuellen Umgebungen, die Redirection-Techniken nutzen um Real-Walking zu ermöglichen, signifikant besser unbewusst räumliches Wissen über diese Umgebungen sammelten, signifikant bessere Navigation und Wegfindung aufwiesen und die Größe der Umgebung signifikant besser einschätzen konnten als in Umgebungen, die andere Fortbewegungsarten nutzen, wie Walking-In-Place, Joystick-Steuerung oder Teleportation [13], [8].

Eine Taxonomie über die verschiedenen Redirection Techniken stellten 2012 Suma et al. [19] vor. Die unterschiedlichen Techniken werden in die Kategorien: "Repositioning" (Repositionierung) oder "Reorientation" (Reorientierung), "Subtle" (subtil) oder "Overt" (unverborgen), und "Discrete" (diskret) oder "Continuous" (kontinuierlich) unterteilt.

2.2.1 Rotation Gains

Bei Rotationgains handelt es such nach Sumas Taxonomie [19] um eine kontinuierliche, subtile Reorientierungstechnik. In der Arbeit [17] untersuchten Steinicke et al. verschiedene subtile Redirection-Techniken darauf, wie stark die Manipulation sein darf, bevor Proband:innen erkennen ob sie eingesetzt wurde oder nicht. Dazu teilt er die verschiedenen Elemente, die für Redirected Walking eingesetzt werden, in drei verschiedene Gains

ein: “Translation-Gains”, “Rotation-Gains” und “Curvature-Gains”. Es stellte sich heraus, dass Nutzer:innen physisch um bis zu 49% mehr oder um bis zu 20% weniger als die wahrgenommene virtuelle Rotation, rotiert werden können, ohne die Diskrepanz zu bemerken.

Des weiteren wurde festgestellt, dass Distanzen unbemerkt um bis zu 14% herunter- oder um bis zu 26% heraufskaliert werden können und, dass Nutzer:innen erst bemerken, dass Sie in einem Kreisförmigem Bogen durch den Trackingspace geleitet werden, wenn dessen Radius 22m oder kleiner ist.

2.2.2 Impossible-Spaces

Bei “Impossible-Spaces” handelt es sich um eine von Suma et al. [18] vorgestellte Redirection-Technik, bei der sich die Architektur der virtuellen Umgebung auf nicht-euklidische Weise verändert, sodass solche Gebiete in der Realität nicht existieren könnten. Die Räume überlappen einander, allerdings wird jeweils nur einer der überlappenden Räume angezeigt. Hierbei handelt es sich nach der schon erwähnten Taxonomie um eine subtile diskrete Redirection-Technik.

In einer Forschungsdemonstration [9] stellten Langbehn et al. eine Weise vor mit der Impossible-Spaces mit traditionelleren Redirected-Walking Methoden (in diesem Fall Curvature-Gains) kombiniert werden können, sodass beide Methoden ihren Effekt beitragen können.

2.3 Prozedural generierte Level

2.3.1 Definition

Der Artikel [21] von Togelius et al. definiert prozedurale Generierung von Spiel-Inhalten (procedural (game-)content generation oder auch PCG) als:

> “[...] creating game content automatically, through algorithmic means.”
>
> (“[...] algorithmisch, automatisch, (Computer-)spiel Inhalte erstellen.”)

In ihrer späteren Arbeit hingegen [20] definieren Togelius et al. PCG folgendermaßen neu:

> “We can therefore tentatively redefine PCG as the algorithmical creation of game content with limited or indirect user input.”

> ("Wir können PCG daher versuchsweise als die algorithmische Erstellung von Spielinhalten mit begrenzter oder indirekter Benutzereingabe neu definieren.")

um unter anderem miteinzubeziehen, dass einige PCG-Algorithmen Nutzer- oder Designerinput miteinbeziehen können und somit nicht mehr "automatisch" Inhalte generieren. Ausserdem wollen sie in der Definition festhalten, dass Nutzerinput typischerweise zumindest indirekt (beispielsweise durch Druck eines Startknopfes) erforderlich ist um Inhalte zu generieren.

Mit (Spiel-)Inhalten sind in diesen Definitionen unterschiedlichste Elemente in Videospielen gemeint. Unter anderem Texturen, Musik oder auch die Geschichte des Spiels können prozedural generiert werden. Im Rahmen dieser Arbeit hingegen beschäftige ich mich lediglich mit PCG zur Erstellung von Leveln.

2.3.2 Taxonomie

In ihrer Arbeit [21] stellten Togelius et al. eine Taxonomie für PCG vor, die aus folgenden Kategorien besteht:

"Online versus offline" (Zur Laufzeit versus während der Entwicklung),

"Necessary vs optional" (Müssen die Spieler:innen den generierten Bereich des Spiels absolvieren oder nicht?),

"Random seeds versus Parameter Vectors" (auch: "degrees of control": Wieviel Einfluss hat die Spieler:in auf den Generierten Inhalt, wird nur ein zufälliger RNG-Seed (Random-Number-Generator-Seed) als Eingabe in den Zufallsgenerator genutzt oder wird sein bisheriges Spielverhalten analysiert und bei der Generierung beachtet?),

"Stochastic vs deterministic" (Wird bei gleicher Eingabe (abgesehen vom RNG-Seed) auch der gleiche Inhalt generiert?) und

"Constructive vs generate-and-test" (Generiert der Algorthmus direkt nur korrekte Ausgaben, oder funktioniert er so, dass er fortlaufend Versuche generiert und dann validiert ob sie korrekt sind und sie dann erst ausgibt.)

Der in dieser Arbeit beschriebene PCG-Algorithmus lässt sich dementsprechend eher in diese Kategorien Taxonomie einordnen, als in ihre jeweiligen Alternativen: Online, necessary, random seeds, stochastic and constructive.

2.4 Infinite walking

Viele der redirection Techniken ermöglichen das (erlebte) hinaustreten über den Rand des Trackingspaces, doch dennoch bleibt die begehbare Fläche limitiert. Solange die virtuelle Umgebung von menschlichen Designern erschaffen werden muss ist sie begrenzt.

Wenn jedoch PCG genutzt wird um die virtuelle Umgebung zu erschaffen lässt sie sich theoretisch endlos weit durchschreiten, weil die Generierung während dem Erkunden der Welt fortgeführt werden kann. Wenn die virtuelle Umgebung also, mit Hilfe von PCG, theoretisch endlos weit erkundet werden kann spricht man vom "Infinite-(Real)-Walking". In der Regel lässt sich dieser Zustand erreichen indem man Redirection-Techniken (um über den Trackingspace hinaus gehen zu können) mit prozeduraler Levelgenerierung (um die Welt weiter zu während der Laufzeit weiter zu generieren) kombiniert.

Ein Beispiel für eine solche Technik stellen Vasylevska et al. in ihrer Arbeit [25] vor. Ihr Algorithmus generiert fortlaufend Räume, innerhalb des Trackingspaces, die einander überlappen können (Impossible-Spaces) und verbindet sie mit Korridoren, sodass die Nutzer:in von einem Raum zum nächsten gehen kann. Praktisch ist diese Technik besonders bei Umgebungen in denen der Inhalt der Räume mehr im Fokus steht als das spezifische Layout der Räume wie beispielsweise einem Museum.

Einen sehr ähnlichen Ansatz nutzt das VR-Spiel "Tea for God" [5] bei dem die Nutzer:in durch ein endlos scheinendes Labyrinth von Korridoren gehen kann. Der Entwickler Jarosław (Void Room) Ciupiński erklärt in seinen Devlogs (beispielsweise [4] oder [3]) genauer wie der Ansatz funktioniert. Die Welt besteht aus einem prozedural generierten Netz von verbundenen Zellen, die jeweils einen Raum repräsentieren und mit Korridoren verbunden sind. Auch hier basieren die Räume auf den vorher erwähnten Impossible Spaces.

Einen anderen Ansatz verfolgt das in der Arbeit [2] von Cheng et al. vorgestellte Projekt "VRoamer". Hier erkundet die Nutzer:in eine On-The-Fly generierte virtuelle Umgebung (auch hier besteht diese aus Räumen und Korridoren), während er durch die reale Welt läuft. Die Generierungssoftware erhält einen 3D-Kamera Input und kann so Wände, Säulen, Gegenstände, andere Menschen etc. beachten und dementsprechend die virtuelle Welt anpassen. Dort wird dann ein virtueller Gegenstand platziert, sodass die Nutzer:in nicht mit den Hindernissen der realen Welt kollidiert. Diese Technik ist nur bei VR-System anwendbar, die nicht auf einen Trackingspace beschränkt sind, sondern (zum Beispiel mit Kameras am HMD (Head-Mounted-Display)) ihre Umgebung, und somit auch ihre eigene Position und Orientierung benötigen. Die Möglichkeit die virtuelle Umgebung zu erkunden sind hier also nur durch den realen Platz, den die Nutzer:in zur freien Begehung zur Verfügung hat limitiert. Streng genommen gilt die Definition von Infinite-Walking hier also nicht, sie sollte an dieser Stelle aber dennoch Erwähnung finden.

2.5 Einordung dieser Arbeit

Ähnlich zu der Arbeit [25] werde ich in dieser Arbeit eine Methode vorstellen, wie mit verschiedenen Redirection-Techniken und einem PCG-Algorithmus eine virtuelle Umgebung mit Infinite-Walking erstellt werden kann. Vergleichbar mit den Arbeiten [13]

und [8] werde ich diese Methode dann in einem Experiment unter Testbedingungen mit alternativen Fortbewegungsarten, die dementsprechend kein Real-Walking ermöglichen auf verschiedene Faktoren vergleichen.

Kapitel 3

Implementierung

In diesem Kapitel werde ich die technischen Elemente für die Umsetzung des in dieser Arbeit vorgestellte Experiments vorstellen. Dabei werde ich im Grob erklären, wie die unterschiedlichen Module des Quelltext funktionieren und dementsprechend offen legen, wie eine solche Infinite-Walking Umgebung implementiert werden kann. Im darauf folgenden Kapitel wird dann detailliert die Implementierung der Levelgenerierung beschrieben. Zunächst gebe ich eine Übersicht über die Beziehungen zwischen den verschiedenen Scripts und Klassen des Projektes indem ich sie graphisch in Form eines UML-Klassendiagramm darstelle. Danach wird jede einzelne Klasse einmal Vorgestellt.

Die gesamte Programmierung für dieses Projekt ist in der Entwicklungsumgebung Unity (Version:) [23] und dementsprechend mit der Programmiersprache "C#" erfolgt. Um die Software während der Entwicklung testen zu können und um damit das Experiment durchführen zu können wurde mir freundlicherweise eine "Oculus Quest 2" [11] Datenbrille, vom Arbeitsbereich Mensch-Computer-Interaktion der Universität Hamburg zur Verfügung gestellt. Um mit der Schnittstelle davon zu interagieren nutzt das Projekt das, von Oculus frei zur Nutzung gestellte, "Oculus Integration SDK" für Unity [12].

3.1 Klassendiagramm

Um das Diagramm übersichtlich zu halten beschränkt es sich ausschließlich auf die Relationen zwischen den Klassen, es wird also nicht wie sonst in UML-Klassendiagrammen üblich die gesamte Schnittstelle aller Klassen inklusive ihren Attributen und ihren Methoden aufgelistet. Aus dem selben Grund wurden auch Grundklassen/-typen mit denen Standardmäßig in Unity gearbeitet wird (zum Beispiel "Vector3", "MonoBehaviour" oder "GameObject") weggelassen.

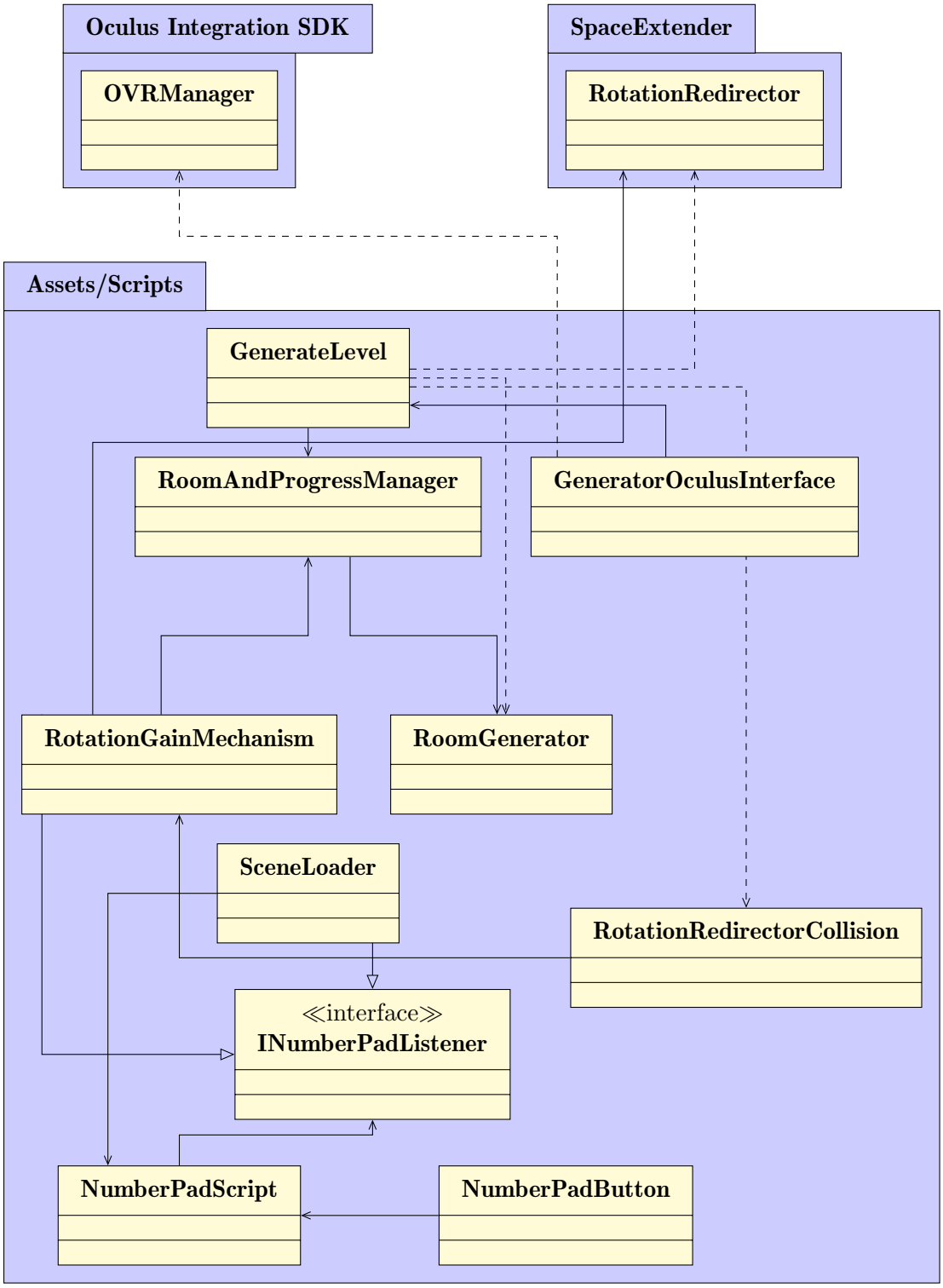

Abbildung 3.1: Ein Überblick über die Beziehungen der Klassen untereinander in Form eines UML-Klassendiagramms; Für die Übersichtlichkeit wurden Unity-Speizifische Klassen wie zum Beispiel *Vector3* oder *MonoBehaviour* herausgelassen

3.2 Vorstellung der Klassen

GeneratorOculusInterface Kommuniziert mit der Schnittstelle des "Oculus Integration SDK" um Informationen die im Projekt benötigt werden bereitzustellen. Die wichtigsten Informationen die hier geliefert werden sind die Koordinaten der Begrenzungsecken des Trackingspaces. Die Nutzer:in der Oculus Quest 2 stellt im Setup der Brille den sogenannten "Guardian" ein. Dies ist dann eine Begrenzung die sie um den begehbaren Raum in dem sie sich befindet zieht und der immer dann in der virtuellen Realität eingeblendet wird, wenn sie dieser Begrenzung mit den Hand-Controllern oder der Datenbrille zu nahe kommt. Diese Begrenzung ist allerdings nicht rechteckig, was für den hier vorgestellten Levelgenerierungsalgorithmus aber notwendig wäre. Doch das Oculus SDK bietet in seiner Schnittstelle an, das größte, in diese Begrenzung passende, Rechteck auszugeben, dies ist dann die Grundlage für die Raumgenerierung. Zudem ist diese Klasse dafür zuständig, je nach ausgewählter Testbedingung entsprechende Einstellungen an das Oculus SDK zu übergeben.

RoomGenerator Implementiert die Generierung der einzelnen Räume. Dafür liegt in dieser Klasse die zentrale "Generate"-Methode. Anhand von den vier Eckkoordinaten des Raumes, der Richtung in die der Raum zeigen soll und verschiedener Zusatzeigenschaften wie zum Beispiel, der Tiefe des zu generierenden Korridors, der Höhe der zu generierenden Türen, der Dicke der Wände und der Höhe des zu generierenden Korridors erzeugt diese Methode das GameObject für den zu generierenden Raum, erzeugt das entsprechende Mesh für den Korridor und platziert die ansonsten nötigen Objekte (zum Beispiel Türen, Tafel, Eingabefeld) an den richtigen Positionen. Um die Tür nicht immer mittig zu platzieren bekommt die "Generate"-Methode ausserdem einen Zufallswert zwischen 0 und 1 übergeben. Dieser bestimmt dann die Position der Tür, die in den Korridor hineinführt. (0 bedeutet ganz links, 1 hingegen ganz rechts). Des weiteren bietet diese Klasse mehrere Methoden an, die Berechnungen über den Raum, den sie generiert haben anstellen können. So kann man sich beispielsweise berechnen lassen ob der Korridor des Raumes auf der längeren, oder der kürzeren Kante des Grundrechtecks liegt, wie die Koordinaten des Mittelpunkts des ganzen Raumes sind und wie die Koordinaten des Mittelpunktes des Bereichs vor dem Korridor sind. (An diese Stelle im ersten Raum wird die Proband:in bei den Testbedingungen, in denen kein real-walking stattfindet, teleportiert.) Die Klasse besteht aus circa 1000 Zeilen Programmierung und ist somit die längste selbst geschriebene Klasse des Projektes.

GenerateLevel Diese Klasse implementiert den Levelgenerierungsalgorithmus der genauer in Kapitel 4 beschrieben wird. Sie ist die zentrale Klasse des Projektes. In ihren öffentlichen Attributen können die Einstellungen für die zu generierende Testbedingung verändert werden. In ihrer "Generate"-Methode implementiert sie den Levelgenerierungsalgorithmus und verbindet darin die restlichen Teilmodule des Projekts. Sie ist also nicht nur dafür verantwortlich die "Generate"-Methode der "RoomGenerator"-Klasse mit den richtigen Eingabewerten aufzurufen, sondern platziert auch die Rotation-Gain Funktionali-

tät an die richtige Stelle, und übergibt die Generierten Räume an den "RoomAndProgressManager", in dem sie in einer Liste gespeichert werden.

RoomAndProgressManager Ist dafür zuständig den Fortschritt der Proband:in zu verfolgen und entsprechend zu reagieren wenn es notwendig ist. Beispielsweise öffnet er die entsprechenden Türen, wenn ein Raum abgeschlossen ist. Des weiteren hält diese Klasse Instanzen der GameObjects für die Räume, in einer Liste, sodass sie dazu in der Lage ist an den zugehörigen RoomGenerator-Instanzen Methoden aufzurufen. Um später vergleichen zu können wie gut die Einschätzung der Proband:innen war misst diese Klasse ausserdem die von der Proband:in fortbewegte Strecke. Wenn der letzte Raum des aktuellen Versuchs erreicht wurde übergibt sie die entsprechenden Daten an den FirebaseHandler der diese dann an eine Cloud-Datenbank verschickt sodass der Versuchsdurchgang im Nachhinein ausgewertet werden kann.

RotationGainMechanism Diese Klasse implementiert den Mechanismus, der in der Real-Walking Bedingung bei den jeweiligen Räumen angewandt wird um den genauen Ablauf des RotationGains zu bestimmen. Er zeigt die Zahlen auf der Tafel an, verwertet die Eingabe des Eingabefelds, startet den Rotation-Gain und beendet ihn auch wieder. Er signalisiert dem RoomAndProgressManager wenn der Rotation-Gain vollständig ist, also sich der Trackingspace also um 90° gedreht hat, und die Proband:in dann eine richtige Eingabe tätigt, dass der Raum abgeschlossen ist, sodass sich die Tür öffnet. In den anderen Bedingungen ist er dafür verantwortlich dies nach einer zufälligen, geringen, Anzahl richtiger Eingaben in das NumberPad zu tun, sodass die Proband:in auch in diesen Bedingungen den selben Ablauf erleben.

SceneLoader Bei betreten des Levels sieht die Proband:in ein virtuelles Eingabefeld, in dass sie die verschiedenen Testbedingungen (A1-C3, siehe Kapitel) eingeben kann um den Versuch zu starten. Der SceneLoader ist dafür zuständig auf diese Eingabe entsprechend zu reagieren und die entsprechende Bedingung zu Laden. Dazu verändert der die Attribute der "GenerateLevel"-Instanz und ruft dann auf ihre die "Generate"-Methode auf.

RotationRedirectorCollision Kleine Hilfsklasse, die erkennt ob der Nutzer sich gerade in dem entsprechenden Bereich befindet, indem der Rotation-Gain aktiv sein soll.

Door Die Türen nutzen dieses Script. Auf ihm kann die Methode "OpenDoor" aufrufen um sie zu öffnen.

Dissolvable Wird von der "Door"-Klasse genutzt um der Tür die Funktionalität zu geben fließend zu verschwinden. Dafür nutzt sie einen entsprechend programmierten Shader, der mithilfe einer Wolkentextur die Transparenz des Türmaterials anpasst.

INumberPadListener Die Klassen die über Eingaben in einem NumberPad informiert werden wollen können dieses Interface implementieren um zu einem NumberPadListener zu werden. Dies geschieht also nach dem "Observer Pattern" (siehe [7])

NumberPadScript Dieses Script ist ein Component eines NumPads und ist dafür zuständig die Eingabe zu verarbeiten und dann alle angemeldeten Observer über Änderungen zu informieren.

NumberPadButton Verbessert die Knöpfe eines NumberPads indem es ihnen eine "CoolDown"-Periode gibt, sodass nicht unbeabsichtigt mehrere Eingaben entstehen und indem es einen Soundeffekt abspielt um der Nutzer:in als Feedback zu bestätigen, dass der Knopf gedrückt.

FireBaseHandler Bietet die Methode "PostResult" an, die die Daten des aktuellen Versuchs an eine Cloud-Datenbank verschickt. So werden dann die gelaufene Anzahl Räume, die gelaufene Strecke in Metern, die Fortbewegungsart, und das Datum, inklusive der genauen Uhrzeit des Abschickens gespeichert. Ausserdem wird gespeichert ob das Verschicken im Unity-Editor, also zum Testen, geschehen ist, oder ob es wirklich von der Datenbrille aus versendet wurde.

3.3 Design der Szene

Im folgenden werde ich die Designentscheidungen, die die Optik der Szene bestimmen erläutern.

Die virtuelle Umgebung, die die Nutzer:in durchschreitet ist der eines Dungeons nachempfunden. Dies ist in Computerspielen ein häufig eingesetztes Setting, weltbekannte Spiele wie "The Elder Scrolls V: Skyrim", die "Final Fantasy" Spielreihe, die "The Legend of Zelda" Spielreihe, und besonders sogenannte "Roguelike"-Spiele, wie die "Diablo" Spielreihe, "The Binding of Isaac", oder "Darkest Dungeon", beinhalte alle zumindest zum Teil Dungeonartige Level, die den Spieler herausfordern eine dunkle Umgebung zu erkunden und gegen fiese Gegner zu kämpfen. Obgleich letzteres in den hier erzeugten Leveln nicht vorkommt, ist das Ambiente bewusst ein wenig düster gehalten und erinnert optisch an einen Dungeon. Sehr hilfreich ist dafür die aus dem Unity-Asset-Store heruntergeladene Textur [22], die die Wände schmückt und ihnen ein rustikales, dungeonartiges Gefühl verleihen.

Das Spiel "Rogue", von 1980, nachdem auch das Genre "Roguelike" benannt wurde ist ein Meilenstein in der Prozeduralen Levelgeneration gewesen und hat bis heute großen Einfluss auf die Welt der Computerspiele. Auch dabei geht es darum einen Dungeon zu erkunden, auch wenn dieser verständlicherweise graphisch, noch nicht besonders aufbereitet gewesen ist.

Abgesehen davon bietet die Umgebung eines Dungeons viele Freiheiten, was den Realismus der Welt angeht. Es wird nicht hinterfragt wieso ein Gebäude so strukturiert sein sollte, dass man die ganze Zeit um die Ecke geht oder wieso die Türen sich öffnen und schließen, in dem sie magisch erscheinen oder sich auflösen.

Um die Erkundungsstimmung noch ein wenig zu verstärken und eine realistische Lichtquelle in die Welt einzubauen trägt die Nutzer:in eine Kopflampe, dies ist ein einfacher Spotlight mit leicht bläulichem Licht, dessen Parent die Kamera des Spieler:innenavatars ist.

3.4 Implementierung der Rotationgains

Die Umsetung der Rotaiongains erfolgt mithilfe des freundlicherweise zur Verfügung gestelltem Unity-Package "Space-Extender". Dieses bietet das Prefab "RotationRedirector", welches im Code instanziiert werden kann und dann nachdem es richtig positioniert ist, die Dimensionen des Trackingspaces und den virtuellen Avatar der Nutzer:in und die Kooridinaten, des Punktes, um den rotiert werden soll übergeben bekommen hat, dazu in der Lage ist einen Rotationgain auf die Nutzer:in anzuwenden. Der Rotation-Redirector wird nachdem das alles geschehen ist für jeden Raum, an die Instanz des RotationGainMechanism, die für eben jenen Raum zuständig ist übergeben, sodass er

aktiviert werden kann sobald das Signal vom ensprechenden Collider kommt, dass die Nutzer:in sich über dem Rotationspunkt (P) befindet.

3.5 Der Rotationgain Inzentivierungs Mechanismus

Damit der Rotationgain seinen Effekt erzielen kann ist es wichtig, dass die Nutzer:in sich etwa auf den Punkt P stellt und ihren Kopf um die Y-Achse dreht. Damit sich dies intuitiver anfühlt und die Aufmerksamkeit nicht so konkret auf den Rotationgain gelenkt wird, habe ich mich dagegen entschieden den Nutzer:innen einfach den Punkt P zu markieren und sie zu bitten dort ihren Kopf zu drehen, sondern habe mir einen Vorwand ausgedacht, wieso es zum einen Notwendig ist an der Stelle zu stehen und zum anderen, wieso sie ihren Kopf drehen müssen.

Um diesen Effekt zu erzielen wird ein Nummern-Eingabefeld, an der Korridorinnenwand in Richtung des Raumes aus dem die Nutzer:in gerade kommt, genau vor die Position P plaziert. In dieses Eingabefeld sollen dann mehrere zweistellige Nummern, die am anderen Ende des Korridors, auf einer digitalen Tafel angezeigt werden eingegeben werden, bis die Tür sich öffnet. Auf der Tafel wird nur eine zweistellige Nummer zur Zeit angezeigt. Dies sorgt dann sowohl dafür dass die Nutzer:in sich automatisch auf den Punkt P stellt, um möglichst gut die Nummern eingeben zu können, als auch dafür, dass die Nutzer:in ihren Kopf wiederholt um die Y-Achse dreht weil sie abwechselnd ablesen muss welche Zahlen auf der Tafel hinter ihr angezeigt werden und dann die Zahlen in das Eingabefeld schräg vor ihr eingeben muss. Dies sorgt für einen intuitiven Grund den Kopf zu drehen und somit dafür den Rotationgain zu nutzen. Wenn der Fortschritt des Rotation-Gains abgeschlossen ist und ein weiteres Mal die korrekte Nummer in das Eingabefeld eingegeben wurde, öffnet sich die Tür vor der Nutzer:in (also die "Seitentür") sodass die Nutzer:in in den nächsten Raum gehen kann. Da nun der Trackingspace genau den nächsten Raum umgibt ist es wichtig, dass die Nutzer:in nicht versucht zurück in den vorherigen Raum zu gehen. Aus diesem Grund schließt sich hinter der Nutzer:in eine vorher noch nicht zu erkennenende Tür.

Im Programm-Code findet sich dieser ganze Vorgang in der Klasse "RotationGainMechanism". Dieser wird für jeden Raum erstellt und auch dem Objekt des Raumes als Component hinzugefügt. Eine Instanz des RotationGainMechanisms ist also immer für den Raum zuständig auf dem er sich befindet. Mit der "Init"-Funktion wird das Objekt direkt nach der Erstellung initialisiert, hier meldet sich die Instanz dann auch in der zugehörigen Eingabefeld-Klasse ("NumPadScript") als Observer an. Sobald sich die Nutzer:in dann in den um die Position P herum platzierten Collider wird mit der Funktion "StartMechanism" die Prozedur gestartet. Zunächst wird eine neue zweistellige Zahl mithilfe des (von Unity bereitgestellten) Pseudozufallgenerators erstellt, dann wird diese Zahl auf der dem aktuellen Raum zugehörigen Tafel angezeigt. Sobald die Bestätigungstaste des Eingabefelds gedrückt wurde ("#"), wird überprüft ob der Fortschritt des Rotation-Gains schon bei 100% ist. Falls nicht, beginnt das ganze von vorne und es

wird eine neue Zahl erstellt. Wenn der Rotation-Gain vollständig abgeschlossen ist (also eine 90° Drehung stattgefunden hat) wird dem "RoomAndProgressManager" mitgeteilt dass der aktuelle Raum abgeschlossen ist und die Tür öffnet sich.

In den Kontrollbedingungen wird kein Rotationgain angewandt. Dennoch müssen die Proband:innen auch hier Zahlen in das Eingabefeld eingeben. In diesen Szenarien wird die Türöffnung aus offensichtlichen Gründen nicht von der vervollständigung des Rotationgains abhängig gemacht, sondern wird eine Zufallszahl zwischen 4 und 8 bestimmt und mitgezählt, wie oft Zahlen bestätigt wurden. Wenn die Zufallszahl erreicht wird öffnet sich die Tür. Durch die Erfahrung beim Testen hat sich gezeigt dass es etwa zwischen 4 und 8 Eingaben bedarf, bevor der Rotationgain abgeschlossen ist.

3.6 Messmechanismen und Datenspeicherung

Um an späterer Stelle einschätzen zu können, wie gut die Proband:innen die Entfernung die sie sich fortbewegt haben geschätzt haben ist es notwendig eben dies auch zu messen. Zusätzlich dazu ist es auch notwendig diese Daten zu speichern. Im folgenden werde ich erläutern, wie dies implementiert ist.

Da es nicht darum geht die Luftlinie zwischen Start und Ende des Levels zu messen, da dann die Bewegungsweise der Proband:in keine Rolle mehr spielen würde, sondern ihre konkreten Fortbewegungsstrecke zu messen, ist diese Aufgabe nicht gänzlich trivial. Würde man einfach in jedem Frame die Kopfposition des virtuellen Avatars, der Proband:in speichern, die Distanz zum vorherigen Punkt berechnen und aufsummieren, wäre das Ergebnis sehr unexakt. Da nun jede millimeterbewegung des Kopfes aufsummiert werden würde, würde eine einfache Kopfdrehung die fortbewegte Strecke schon um einen großen Teil erhöhen. Dies würde dazuführen, dass die Messung der fortbewegten Strecke nicht mehr intuitiv einzuschätzen wäre. Um dem entgegenzuwirken habe ich die Auflösung dieser Messung deutlich veringert, indem nicht mehr jedes Frame, diesen Prozess durchläuft, sondern nur ein Frame etwa alle 2 sekunden. So werden deutlich grobere Bewegungen gemessen und die gemessenen Ergebnisse passen deutlich besser zu der intuitiven Einschätzung der Entfernung.

All dies ist im "RoomAndProgressManager" implementiert. Hier gibt es eine "StartTimeMeasure"-Methode, die zum Start der Bedingung vom "SceneLoader" aufgerufen wird und eine "AddDistance"-Methode, die in der von der "Update"-Methode alle 150 Frames einmal aufgerufen wird. Diese bestimmt dann euklidisch die Distanz zwischen der aktuellen Position der Nutzer:in und der zuletzt gemessenen und addiert diese auf eine Kummulationsvariable.

Sobald das Level der Versuchsbedingung abgeschlossen ist wird diese Kummulationsvariable an den "FirebaseHandler" weitergeleitet, der nun die Daten der aktuellen Versuchsbedingung an die private Datenbank sendet um die Daten für die spätere Auswertung zu speichern.

Kapitel 4

Levelgenerierung

Dieses Kapitel beschreibt, wie genau die Generierung eines in diesem Projekt genutzen Levels funktioniert. Nacheinander werde ich die grundlegenden Ideen hinter der Levelgenerierung erörtern und dabei jeweils genauer auf ihre Implementierung eingehen um dann den Algorithmus, der die unterschiedlichen Codeblöcke nutzt um die Levelstruktur zu generieren, zu beschreiben und vorzustellen. Zuerst wird besprochen, wie die Räume angeordnet werden müssen damit sie durch den Rotation-Gain verbunden werden können. Dann wird erklärt und vorgestellt wie die Korridore, die in den Räumen generiert werden um der Nutzer:in einen klaren Weg voran zu präsentieren und den Übergang zwischen den Räumen intuitiver zu gestalten, erstellt werden. Als nächstes beschreibe ich die Vorgänge die für die Generierung der Wände der Räume verantwortlich sind und erkläre wie die entsprechenden Berechnungen stattfinden. Um hinterher dann den Generierungs-Algorithmus und den zugehörigen Quelltext vorzustellen wird dann noch Beschrieben an welchen Stellschrauben manipuliert werden kann um heterogene Ergebnisse zu bekommen, erst hier können (Pseudo-)Zufallsfaktoren eine Rolle spielen. Den krönenden Abschluss dieses Kapitel bekommt dann der Abschnitt, der die Zusammensetzung des Algorithmusses zunächst erklärt und den entsprechenden Quelltext dann auch noch ausführlich vorstellt.

4.1 Raumplatzierung

Um zu verstehen wie die Berechnung der Ecken eines Raumes funktioniert ist es zunächst essentiell, die Grundidee hinter der Raumanordnung zu verstehen.

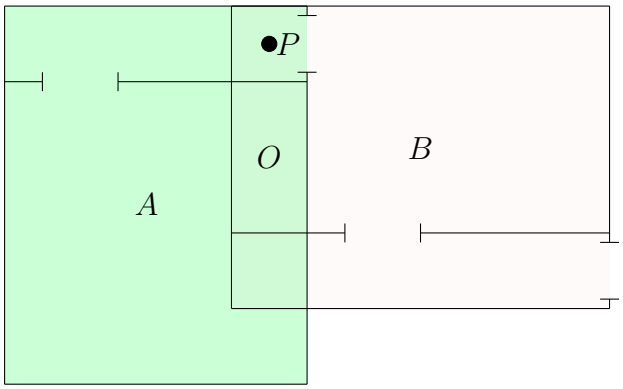

Abbildung 4.1: Veranschaulichung der Raumstellung zueinander, sodass P den Drehpunkt kennzeichnet

4.1.1 Grundidee

Wenn ein Rotation-Gain angewandt wird verdrehen sich virtuelle und reale Realität um den Drehpunkt, der durch die Position der Nutzer:in bestimmt wird. Wenn diese Position P sich in einer Ecke des (rechteckigen) Raumes befindet, zudem genau gleich weit von beiden Wänden, die diese Ecke bilden, entfernt ist und auf die Nutzer:in dann ein Rotation-Gain angewandt wird, bis genau 90° Verdrehung erreicht sind, dann steht der aktuelle Bereich in der virtuellen Realität der vom realen Trackingspace begrenzt ist A, dem vorherigen Bereich, der durch den Trackingspace begrenzt war B genau orthogonal gegenüber. Die beiden Areale A und B überlappen sich in einem gewissen Bereich, in dem die Nutzer:in nach Beendigung des Rotation-Gains dann gerade steht, aber ein Großteil von B ist Teil der virtuellen Realität der zuvor nicht zugänglich war. Auf diese Weise ist also für die Nutzer:in ein neuer Bereich der Virtuellen Welt begehbar geworden. Da ja aber ein Rotation-Gain angewandt wurde hat die Nutzer:in nicht bewusst wahrgenommen, dass die Welten sich verdreht haben. Wenn sie nun also geradeaus geht entsteht dann der Eindruck, dass nun ein weiterer Teil der Welt zugänglich geworden ist. Ziel des Algorithmus ist es eine Reihe von aufeinander folgenden Räumen zu generieren, sodass die Nutzer:in sich durch ein Dungeon-artiges Level bewegen kann um vom ersten zum letzten Raum zu gelangen. Die Grundidee hinter der Raumgenerierung ist es also, dass genau die beiden Areale A und B von Wänden umgeben werden. In Abbildung 4.1 wird ersichtlich, wie die beiden Räume zueinander Stehen müssen um diesen Effekt zu ermöglichen.

Um dann also von Raum A zu Raum B zu kommen muss die Nutzer:in sich an der Position P oft genug drehen und ein Rotation-Gain bis zu 90° angewandt werden. Danach kann sie mit der Illusion geradeaus zu gehen einen neuen Raum erkunden während sie sich eigentlich nur weiter innerhalb des realen Trackingspaces bewegt. Der reale Trackingspace umschließt also immer genau den Raum in dem sich die Nutzer:in gerade befindet.

4.1.2 Berechnung der Ecken

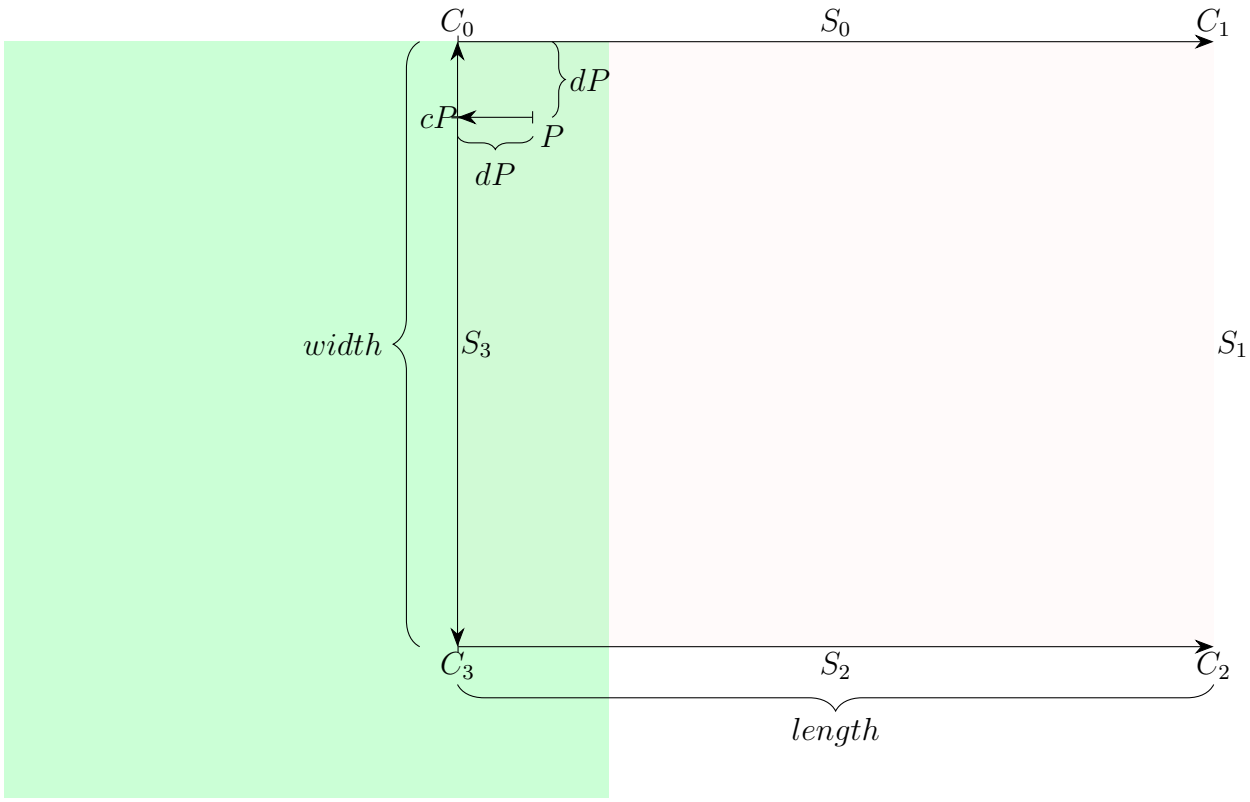

Abbildung 4.2: Zur Eckenberechnung genutzte Vektoren. Mit C_{0-3} werden die Ecken bezeichnet, mit S_{0-3} die Seiten des Raumes (Nicht die zur Eckenberechnung genutzen Vektoren).

Um dies umzusetzen müssen zunächst die Koordinaten der Ecken innerhalb der virtuellen Umgebung bestimmt werden. Für den ersten Raum ist es nicht schwierig, die virtuellen Koordinaten der Ecken der Trackingspace-Boundaries werden vom Oculus Integration SDK geliefert. In der "GeneratorOculusInterface" Klasse werden diese zu Beginn der Levelgenerierung abgefragt und der "GenerateLevel"-Klasse zur Verfügung gestellt. Um allerdings die Raumecken der folgenden Räume zu bestimmen müssen die Position P sowie der Abstand von P zu den Wänden des jeweiligen Vorraums definiert sein(dP). Die Berechnung dieser Punkte findet sich in Abschnitt 4.2. Für die weiteren Räume berechnen sich die Eckvektoren, dann folgendermaßen. Auch zu sehen ist diese Berechnung in dem in der Abbildung 4.3 gezeigen Quelltext.

$$cP = P + (-1) * sDD * dP \tag{4.1}$$

$$\tag{4.2}$$

$$C_0 = cP + (-1) * btF * dP \tag{4.3}$$

$$C_1 = C_a + sDD * length \tag{4.4}$$

$$C_2 = C_d + sDD * length \tag{4.5}$$

$$C_3 = cP + btF * (width - dP) \tag{4.6}$$

Wie die hier genutzen Variablen cP, sDD, btF, $width$ und $length$ definiert sind wird aus Abbildung 4.2 ersichtlich. cP steht dabei für den Punkt hinter P, der genau eine dP länge in Richtung des alten Raumes liegt. sDD (sideDoorDirection) ist ein normalisierter Richtungsvektor, der die Richtung des Korridors des letzen Raumes erweitert. btF (backToFront) ist auch ein normalisierter Richtungsvektor, der orthogonal zu sDD steht. Er zeigt von der hinteren Wand des Korridors des vorherigen Raumes zur vorderen Wand. $width$ beschreibt die Breite der Räume, während $length$ die Länge beschreibt. Diese beiden Variablen hängen von den Maßen des ersten Raumes ab, bei dem die Eckkoordinaten ja den Trackingspace der Nutzer:in abbilden. Wichtig ist noch zu erwähnen, dass die in der eben erwähnten Abbildung gezeigte Folge von Ecken und Seiten, nicht genau spiegelverkehrt ist wenn der Raum in die andere Richtung generiert würde (siehe Abschnitt 4.4), sondern die Reihenfolge der Ecken genau andersherum läuft. Das ganze ist also nicht nur vertikal, sondern auch horizontal gespiegelt zu der Darstellung.

Listing 4.1: calculateNewCorners.cs

```
private Vector3[] CalculateNewCorners(Vector3 outOfDoor,
    Vector3 backToFront, Vector3 rotationPoint, bool
    sideDoorIsRight, bool corridorOnWidthSide)
{
    //depending on whether the corridor is on the width
        side or on the depth side this changes
    float xLength = corridorOnWidthSide ? roomWidth :
        roomDepth;
    float yLength = corridorOnWidthSide ? roomDepth :
        roomWidth;

    float remainingX = xLength - corridorDepth;

    Vector3 wallBehindRotationPoint = rotationPoint +
        Vector3.Normalize(outOfDoor) * (corridorDepth / 2) *
        -1;
    Vector3 backFirstCorner = wallBehindRotationPoint +
        Vector3.Normalize(backToFront) * corridorDepth / 2 *
        -1; // toBack (back as in: the direction of the old
        corridor)

    Vector3 frontFirstCorner = wallBehindRotationPoint +
        Vector3.Normalize(backToFront) * corridorDepth / 2 +
        //first as in: nearer to the outdoor
                                Vector3.Normalize(backToFront)
                                    * remainingX; //toFront

    Vector3 backSecondCorner = backFirstCorner +
        Vector3.Normalize(outOfDoor) * yLength;
    Vector3 frontSecondCorner = frontFirstCorner +
        Vector3.Normalize(outOfDoor) * yLength;

    Vector3[] output = new Vector3[]
    {
        backFirstCorner,
        backSecondCorner,
        frontSecondCorner,
        frontFirstCorner,
    };

    if (!sideDoorIsRight)
    {
        Array.Reverse(output);
    }

    return output;
}
```


Abbildung 4.3: Funktion zur Berechnung der virtuellen Eckkoordinaten des nächsten Raumes

4.2 Korridore

Damit die Nutzer:in versteht wie sie in den nächsten Raum kommt sind die Räume so designed, dass es einen offensichtlichen Weg voran gibt. Um dies zu erreichen wird eine Kante des Grundecks des Raumes ausgewählt, an der ein Korridor erstellt wird (diese wird Pseudozufällig ausgewählt, siehe Abschnitt 4.4). Er kann sowohl an einer kürzeren Kante als auch an einer längeren Kante des Grundecks des Raumes erstellt werden, nur nicht an der Seite an der das Ende des Korridors des letzten Raums herausragt. Der Korridor hat zwei Türen; Eine "Haupttür", durch die die Nutzer:in in den Korridor hineinkommt und eine "Seitentür", die anfangs noch verschlossen ist und in den nächsten Raum führt.

In diesem Korridor werden dann Elemente platziert die gemeinsam für einen Mechanismus sorgen, der die Nutzer:in inzentiviert sich genug zu drehen, dass die wirkliche und die virtuelle Realität sich dank des Rotation-Gains genug umeinander drehen, dass der nächste Raum nun mit dem realen Tracking-Space übereinstimmt. Dieser Mechanismus wird in Abschnitt 3.5 genauer beschrieben.

4.2.1 Korridor-Meshgenerierung

Um einen solchen Korridor zu generieren müssen zunächst die Koordinaten der genutzten Vertices bestimmt werden. Zudem müssen dann entsprechende Faces gespannt und für jeden Vertex dann auch die UV-Koordinaten errechnet werden. All dies geschieht in der "RoomGenerator"-Klasse.

Ein Korridor besteht aus einem Boden, einer Frontalwand, zwei Seitenwänden und einer Rückwand. Bevor die Meshgenerierung stattfinden kann muss zunächst die Variable D bestimmt werden. Dabei handelt es sich um einen Wert zwischen 0 und 1, der für jeden Raum (Pseudo-)zufällig generiert wird (siehe Abschnitt 4.4). Mit diesem Wert werden die Türpositionen bestimmt, je geringer er ist desto weiter links liegt die Haupttür in der Frontalwand. Ist der Wert geringer als 0.5 befindet sich die Seitentür in der rechten Seitenwand, sonst in der linken, sodass diese immer auf der gegenüberliegende Seite der Haupttür liegt.

Es gibt zwei Möglichkeiten der Struktur einer Wand, abhängig davon ob in ihr eine Tür liegt oder nicht. Der simplere Fall trifft zu wenn in der Wand keine Tür liegt, dann besteht die Wand aus 8 Vertices. Die Wandstruktur der Alternative wird in Abbildung 4.4 veranschauligt. Dabei ist allerdings zu beachten, dass nur die Frontalansicht auf eine solche Wand gezeigt ist, die Innenseite derselben Wand hat allerdings die selbe Struktur. Zudem sind die Vertices der Tür jeweils mit ihren inneren Gegenspielern durch Faces verbunden, sodass man beim durchschreiten der Tür nicht ins innere der Wand blicken kann.

Vertices Das Generieren eines solchen Korridors läuft dann also wie folgt ab. Gegeben sind die die Eckkoordinaten des Raumes in den der Korridor platziert werden soll (C_{a-d}), die Seite des Raumes, an der der Korridor liegen soll S ($_{a-d}$), die Türposition D, die Tiefe des Korridors d, durch die sich der in Unterabschnitt 4.1.2 besprochene Abstand des Punktes P zu den Wänden der Ecke des Punktes ergibt: $d = 2 * dP$, die Höhe h und Dicke der Wände w sowie die Höhe (dh) und Breite (dw) der Tür .

Als erstes werden die unteren, äußeren Punkte berechnet. (F_{1-4}) Dabei wird mit den beiden hinteren, äußeren Punkte des Korridors begonnen. Da die Ecken eines Raumes immer im Uhrzeigersinn gespeichert werden lassen sich die Ecke des Raumes, die S entspricht, und die im Uhrzeigersinn darauf folgende auswählen.

$$F_1 = C_S$$
$$F_2 = C_{(S+1)\%4}$$

Damit wird der Richtungsvektur rTL bestimmt indem die rechte Ecke von der linken Ecke subtrahiert wird und das Ergebnis dann normalisiert wird.

$$rTL = F_2 - F_1$$

Um die beiden vorderen äußeren Ecken zu bestimmen muss nun ein richtungs Vektor tF berechnet werden der von der Seite S in Richtung der gegenüberliegenden Seite des Raumes zeigt. Hierfür wird das Kreuzprodukt von rTL und dem Vektor berechnet und das Ergebnis normalisiert.

$$tF = rTLx$$

Um nun die vorderen Punkte zu bestimmen wird einfach der neue Richtungsvektor mit $2 * dP$ multipliziert und auf die hinteren Punkte addiert.

$$F_3 = F_1 + tF * dP$$
$$F_4 = F_2 + tF * dP$$

Nach demselben Prinzip lassen sich nun die äußeren oberen Ecken des Korridors bestimmen, hierfür werden die unteren Ecken mit dem Vektor T addiert, der sich wie folgt berechnet:

$$T = up * h$$

Um die inneren Punkte zu berechnen werden die äußeren Ecken nach dem selben Prinzip, jeweils mit den beiden Richtungsvektoren, die beide mit w multipliziert werden addiert.

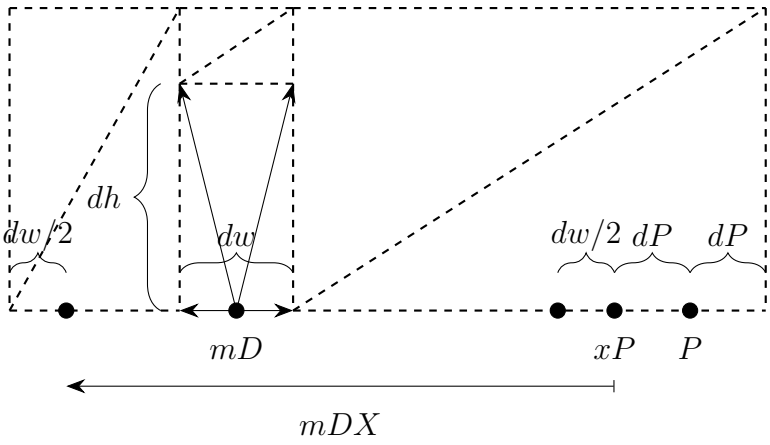

Abbildung 4.4:

Für die Tür-Vertices wird zunächst der untere Mittelpunkt der Tür (mD) berechnet. In der folgenden Erklärung wird davon ausgegangen, dass $D <= 0.5$ ist und somit die Haupttür links, und die Seitentür rechts erzeugt wird, andernfalls wären natürlich lTR und rTL vertauscht. Hierfür ist es zunächst wichtig die entsprechende Achse (mDX)zu berechnen, auf der die Tür dann erzeugt wird (siehe Abbildung 4.4)

Dafür wird erstmal der Punkt, an der Frontalseite des Korridors, diagonal hinter P berechnet (xP). Von da aus spannt sich die Achse dann bis zur anderen Seite des Korridors. Da mD ja den Mittelpunkt der Haupttür darstellen soll, muss an beiden Seiten der Achse $dw/2$ als padding eingesetzt werden, damit die Tür nicht weiter oder rechts übersteht, falls D einen extremwert annimmt. mD berechnet sich dann folgendermaßen:

$$mD = xP + D * mDX$$

Die Berechnung von xP, und mDX wird aus dem in Abbildung 4.5 dargestellten quelltext ersichtlich.

Die Berechnung der Vertices, in der Haupttür ist nun trivial und erfolgt mit der bekannten Methode und wird auch in Abbildung 4.4 ersichtlich. Türhöhe (dh), Türbreite (dw) und Wandbreite (w) werden mit den entsprechenden Richtungsvektoren multipliziert und auf mD addiert.

Listing 4.2: calcMainDoor.cs

```
bool mainDoorIsRight = mainDoorPosition > 0.5f; // main door is more right than left
    this.sideDoorIsRight = !mainDoorIsRight;

    Vector3 innerLeftToRight = leftToRight + Vector3.Normalize(rightToLeft) * wallThickness * 2;
    Vector3 innerDoorGenSpanLeftToRight = innerLeftToRight + Vector3.Normalize(rightToLeft) * rotationPointDistance * 2;

    //leftToRight but less long: whole length - inner - rotpointdistance - doorwith
    Vector3 innerDoorGenSpanLeftToRightDoorPadding = innerDoorGenSpanLeftToRight + Vector3.Normalize(rightToLeft) * doorWidth;

    Vector3 toBehindRotPointLeftRight = Vector3.Normalize(leftToRight) * rotationPointDistance * 2;
    Vector3 sideDoorCornerBehindRotPoint = sideDoorIsRight ? boRiFrCorner + -toBehindRotPointLeftRight : boLeFrCorner + toBehindRotPointLeftRight;
    Vector3 innerCrossDirection = sideDoorIsRight ? -innerDoorGenSpanLeftToRightDoorPadding : innerDoorGenSpanLeftToRightDoorPadding;
    Vector3 sideDoorCornerBehindRotPointDoorPadding = sideDoorCornerBehindRotPoint + Vector3.Normalize(innerCrossDirection) * doorWidth / 2;
    //main door origin is generated between sideDoorCornerBehindRotPoint and the other corner - inner - half of the door width

    if (sideDoorIsRight)
    {
        mainDoorPosition = 1 - mainDoorPosition;
    }

    Vector3 mainDoorOrigin = sideDoorCornerBehindRotPointDoorPadding
                             + innerCrossDirection * mainDoorPosition;
```

Abbildung 4.5:

Faces Der nächste Schritt, der für die Meshgenerierung erforderlich ist, ist das Spannen der verschiedenen Faces (Triangles). Die meisten für den Korridor erforderlichen Flächen bestehen nur aus zwei aneinanderliegenden Triangles, die somit gemeinsam die viereckige Fläche bilden. Die Ausnahmen, sind die Frontalfläche und die Seitenfläche in die die Seitentür generiert wurde (jeweils mit der zugehörigen Innenseite). Die Struktur dieser Flächen wird auch in Abbildung 4.4 ersichtlich. Auch hier werden die benötigten Vertices jeweils durch multiplizieren mit den entsprechenden Richtungsvektoren und der anschließenden Addierung auf bestehende Vertices errechnet.

Um die späteren UV Berechnungen nicht zu verunreinugen wird jeder Vertex dem Buffer pro Face einmal hinzugefügt. Die Faces werden dann gespannt indem die jeweiligen Indizes der Vertices im Buffer aufgelistet werden. Dies beides wird dann dem neu erzeugten $MeshFilter$ Objekt übergeben.

UV Die Berechnung der UV-Koordinaten erfolgt für jeden einzeln und wird nach Triangles geordnet der Reihe nach durchgeführt. Der Quelltext dazu findet sich in der GenerateUV Methode, die in Abbildung 4.6 Abgebildet ist. Die Berechnung besteht daraus die Normale des Triangles zu berechnen, diese mit den Richtungsvektoren abzugleichen und die Vektoren der Vertices dann entsprechend zu Rotieren.

Listing 4.3: uvCode.cs

```
Vector2[] GenerateUV(Vector3[] vertices, int[] triangles)
    {
        Vector2[] uv = new Vector2[vertices.Length];

        for (int i = 0; i < triangles.Length; i += 3)
        {
            int vi0 = triangles[i + 0];
            int vi1 = triangles[i + 1];
            int vi2 = triangles[i + 2];

            Vector3 normal = CalculateNormal(vertices[vi0],
               vertices[vi1], vertices[vi2]);

            if (VectorsAreParallel(normal, toFrontWall))
            {
                //DebugHighlightFace(vertices[vi0],
                   vertices[vi1], vertices[vi2], Color.magenta);

                Vector3 v0 = vertices[vi0];
                Vector3 v1 = vertices[vi1];
                Vector3 v2 = vertices[vi2];

                v0 = gameObject.transform.TransformPoint(v0);
                v1 = gameObject.transform.TransformPoint(v1);
                v2 = gameObject.transform.TransformPoint(v2);

                Quaternion rotation =
                   Quaternion.FromToRotation(leftToRight,
                   Vector3.forward);

                v0 = rotation * v0;
                v1 = rotation * v1;
                v2 = rotation * v2;

                uv[vi0] = new Vector2(v0.z, v0.y);
                uv[vi1] = new Vector2(v1.z, v1.y);
                uv[vi2] = new Vector2(v2.z, v2.y);
            }
            else if (VectorsAreParallel(normal, leftToRight))
            {
                // DebugHighlightFace(vertices[vi0],
                   vertices[vi1], vertices[vi2], Color.magenta);

                Vector3 v0 = vertices[vi0];
                Vector3 v1 = vertices[vi1];
                Vector3 v2 = vertices[vi2];
```



```
                Quaternion rotation =
                   Quaternion.FromToRotation(leftToRight,
                   Vector3.forward);

                v0 = rotation * v0;
                v1 = rotation * v1;
                v2 = rotation * v2;

                uv[vi0] = new Vector2(v0.x, v0.y);
```

4.3 Wandgenerierung

Natürlich reichen die Meshs der Korridore noch nicht aus um das Gefühl eines Raumes zu vermitteln, zumal diese ja auch nur an einer Seite des Raumes statuiert sind. Wie in Unterabschnitt 4.1.1 besprochen sollen die Räume von Wänden umgeben sein.

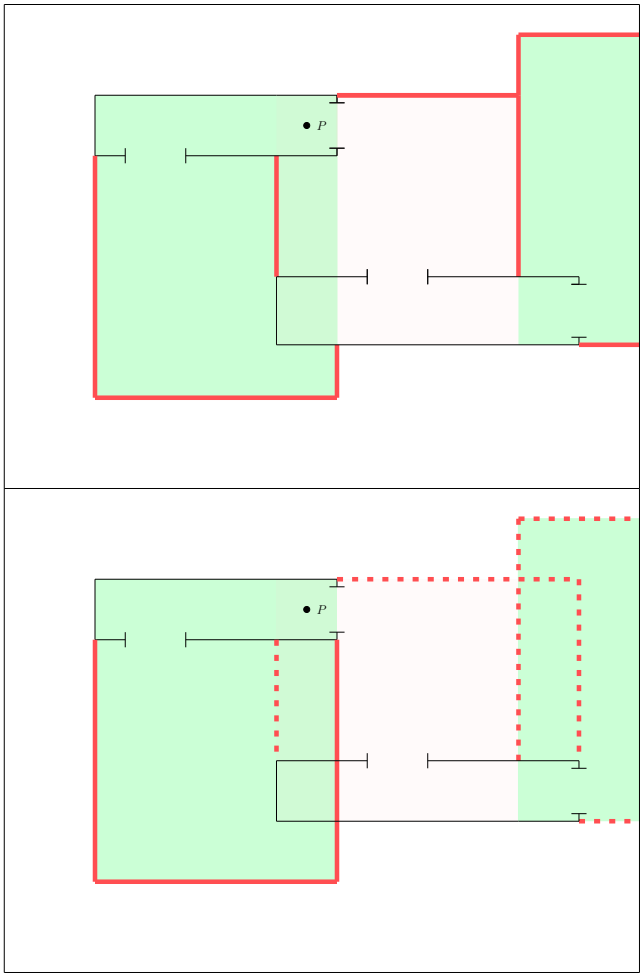

Abbildung 4.7: Vergleich der zwei Wandgenerierungsvarianten. Oben: realistische Variante, die in Joystick- und Teleportationsbedingung eingesetzt werden. Unten: unmögliche Variante, die in der Real-Walking Bedingung die Wände generiert. Gestrichelte Linien repräsentieren Wände, die erst unter bestimmten Umständen sichtbar werden.

Die Wandgenerierung erfolgt über ein Prefab, dass instanziiert und dann gestreckt, positioniert und rotiert wird. Es besteht aus einem Würfel mit der entsprechenden Textur, der Höhe der Wandhöhe h, und der Breite und Tiefe der Wände w. Dies geschieht in der Methode “GenerateWall”. Diese bekommt als Eingabe zwei Punkte, und instanziiert dann eine Wand die von dem einen dieser Punkte zum anderen Reicht. Abgesehen davon besteht der Prozess der Wandgenerierung also nur noch daraus für jeden Raum, für jede

Wand, jeweils die zwei Endpunkte der Wand zu berechnen. In dem hier vorgestellten Projekt werden die Wände auf zwei verschiedene Arten generiert. In der Abbildung 4.7 werden die beiden Varianten miteinander Verglichen.

4.3.1 Realistische Wandgenerierung

Die erste Art, fortlaufend "realistische" Variante genannt, generiert die Wände zwischen den Räumen so, wie sie auch in real existieren könnten. Dazu gibt es einen ausgefeilten Algortithmus, der alle Eventualitäten der Raumgenerierung kennt und die Wände so zwischen den Räumen platziert, dass die Räume noch möglichst viel Platz bieten, aber auch rundherum von Wänden umgeben sind. Der Code für diese Generierung wird in der "RoomGenerator"-Klasse in den beiden "GenerateWallsLegacy"-Methoden ausgeführt. Diese Variante wird in den beiden Kontrollbedingungen "Teleport" und "Joystick" verwendet, weil diese keine Impossible Spaces beinhalten.

Bei der realistischen Wandgenerierung ist die Idee, dass die Wände genau so zwischen die Korridore platziert werden dass der Nachfolgende Raum die Priotität über den Platz hat. Der Bereich O der in der Abbildung 4.1 zu sehen war, gehört bei dieser Variante also immer zu dem Raum B und nicht zu dem Raum A.

Davon ob der Korridor des Raumes an der längeren, oder der kürzeren Seite des Rechtecks generiert wurde hängt ab, wieviele Wände generiert werden. Ist der Korridor an der längeren Seite des Raumes, werden ausgehend vom Korridor des letzten Raumes, zwei Wände generiert, da die nreite Seite des nächsten Raumes als Wand genutzt werden kann. Ist jedoch der Korridor des aktuellen Raumes an der kürzeren Seite des Rechtecks, so muss eine dritte Wand generiert werden, die verhindert, dass eine offene Stelle zwischen dem aktuellen und dem nächsten Raum entsteht.

Um die Wände generieren zu können ist es wichtig, die Punkte diagnonal vor und hinter dem Punkt P des vorherigen Raumes zu kennen. Dazu werden diese nach der Generierung als Iterationsübergreifende Variablen gespeichert und somit an den jeweils nächsten Raum übergeben, siehe Unterabschnitt 4.5.2.

4.3.2 Unmögliche Wandgenerierung

Die andere Variante der Levelgenerierung, hier "unmögliche" Variante genannt, umschließt alle Räume mit drei Wänden, zeigt aber immer nur die Wände des Raums, in dem sich die Nutzer:in gerade befindet an, und hat dementsprechend die Freiheit die gesamte Raumgröße zu umschließen, weil keine Kompromisse für den sich überlappenden Bereich O gemacht werden müssen (siehe Abbildung 4.1). Der Code für diese Generierung wird in der "RoomGenerator" -Klasse in den beiden "GenerateWalls"-Methoden ausgeführt. In dem Versuch wird diese Variante ausschließlich in der Real-Walking Bedingung eingesetzt. Durch sie werden die Räume erst zu Impossible Spaces, da die Räume sich nun überlappen.

Die Umsetzung der unmöglichen Wandgenerierung erfolgt recht ähnlich, wenn auch um einiges minimalistischer als, die der realistischen. Anstatt die Wandgenerierung auf die verschiedenen Räume aufzuteilen, werden die Wände so generiert, dass sie abgesehen von den Korridoren, des aktuellen, und des vorherigen Raumes den ganzen Raum umschließen. Das ein- und ausblenden, der Wände wird dann vom "RoomAndProgress-Manager", der ja Überblick darüber wo die Nutzer:in sich gerade befindet geregelt. Es werden immer nur die Wände des aktuellen Raumes, angezeigt.

4.4 Zufallsfaktoren in der Levelgenerierung

Der hier beschriebene Levelgenerierungsalgorithmus nutzt das Element des Pseudozufalls um durch Verändern verschiedener Variablen das Ergebnis zu beeinflussen. So ist es möglich viele unterschiedliche Level zu generieren. Welche Variablen sich dafür unterscheiden können werde ich an dieser Stelle vorstellen.

4.4.1 Korridorrichtung

Zu Beginn der Korridorgenerierung wird eine Seite des Raumes als Korridorseite S bestimmt. Diese wird pseudozufällig ausgewählt. Im ersten Raum ist es noch vollkommen beliebig, welche Seite ausgewählt wird[1]. Bei den weiteren Räumen hingegen können bestimmte Seiten nicht ausgewählt werden, weil der Korridor nach der Generierung sonst mit dem Ausgang des Korridors des vorherigen Raumes überlappt. Durch die Art und Weise, wie die Eckenberechnung der Räume funktioniert, lässt sich feststellen, dass S_3 in allen Umständen immer die Seite ist, aus der der Korridor des letzten Raumes herauszeigt (siehe Unterabschnitt 4.1.2). S_3 kann also zur Korridorgenerierung nicht ausgewählt werden. Falls die Seitentür des vorherigen Raumes auf der rechten Seite liegt (wie in der Abbildung 4.2) muss ausserdem S_0 blockiert werden, da die Nutzer:in sonst von einem Korridor direkt in den nächsten läuft. Ist die Seitentür aber auf der linkens Seite ist ja die Ecken- und Seitenbelegung, wie in Unterabschnitt 4.1.2 beschrieben, nicht nur horizontal, sondern auch vertikal spiegelverkehrt. Deshalb muss in dem Fall die Seite S_2 blockiert werden. Damit diese Seiten bei der zufälligen Auswahl ignoriert werden, Funktioniert die Methode dafür ("RandomDirectionBlocked", siehe Abbildung 4.8) folgendermaßen. Als Eingabe wird eine Liste LB mit den blockierten Richtungen übergeben. Alle Richtungen (representiert von Zahlen von 0-3), die darin vorkommen werden aus der Liste aller möglichen Richtungen entfernt. Mit dem Unity-Objekt *Random* lässt sich durch Abrufen der Eigenschaft *value* eine zufällige Zahl zwischen 0 und 1 generieren[2]. Um nun die Zahl aus der Liste verbleibender Richtungen LD auszuwählen wird das Ergebnis nun mit $4 - length(LB)$ multipliziert um den Index zu berechnen.

[1] Im Versuchsaufbau wird dabei aber immer die selbe Seite ausgesucht um das Erscheinen im Raum einheitlicher zu gestalten. (Die Nutzer:in guckt nach dem Start des Versuchs in Richtung des Korridors).

[2] Falls tatsächlich mal eine volle 1 ausgewählt würde dies tatsächlich dazu führen, dass eine zu hohe Zahl ausgewählt werden würde, deshalb wird im Code mithilfe der Min-Funktion dafür gesorgt, dass

Listing 4.4: RandomDirectionBlocked.cs

```
private int RandomDirectionBlocked(List<int> block)
    {
        if (block.Count == 0)
        {
            return RandomDirection();
        }

        List<int> directions = new List<int>
        {
            0, 1, 2, 3
        };

        foreach (int i in block)
        {
            if (i == -1)
            {
                return RandomDirection();
            }
            else
            {
                directions.Remove(i);
            }
        }

        float rand = Random.value;
        rand = Math.Min(rand, 0.99999f);

        return directions[(int) (rand * ( 4 - block.Count))];
    }
```

Abbildung 4.8:

4.4.2 Türposition

Die Frage in welche Richtung der jeweils nächste Raum generiert wird, hängt gänzlich davon ab, wie die Haupttür des Korridors positioniert ist. In Unterabschnitt 4.2.1 wird erklärt wie die Haupttür auf der Frontalwand des Korridors generiert wird und abhängig von dem Wert D auf der links-rechts Achse positioniert wird. Falls D einen Wert von 0 hat wird die Tür ganz links positioniert, bei einem Wert von 1 ganz rechts. Davon ist dann auch die positionierung der Seitentür Abhängig. Ist $D <= 0.5$ wird die Seitentür in die rechte Seitenwand des Korridors generiert, sonst in die linke. Davon ist dann

dieser Fall nicht eintreten kann, siehe Abbildung 4.8

natürlich auch wieder Abhängig in welche Richtung der folgende Raum generiert wird, denn die Nutzer:in soll ja aus der Seitentür heraus-, in den Raum hineintreten. Man kann den Wert D also sowohl als einen verstehen, der graduell die position auf der Frontalwand verstellt, als auch als einen wichigen Wert, der die Richtung des folgenden Raumes vorherbestimmt.

Auch hier müssen wieder einige Varianten ausgeschlossen werden, um die Begehbarkeit der erzeugten Level zu garantieren.

S_1 stellt unabhängig der Richtung immer die Seite gegenüber der Seite aus der man kommt dar, wenn man aus dem vorherigen Raum in den aktuellen Raum tritt. Wenn der Korridor an diese Seite generiert wurde, kann die Türposition nicht mehr so sein, dass der als nächstes Generierte Raum Überschneiungen mit der Seite des aktuellen Raumes hat, aus der der Korridor des vorherigen Raumes kommt, da sonst der Weg aus dem Korridor des vorherigen Raumes nicht frei ist weil der nachfolgende Raum darüber generiert wurde. Jeh nach Raumdimensionen würden sich dann sogar potientell 3 Räume überlappen.

Genauso wichtig ist, dass das Ende des Korridors des aktuellen Raumes, nicht in Richtung des vorherigen Raumes zeigt.

Diese Varianten werden bei der Levelgenerierung mit Fallunterscheidungen ausgeschlossen. Dazu hat die Methode "GenerateRandomDoorPosition" in der "GenerateLevel"-Klasse, die für die Berechnung von D verantwortlich ist zwei Parameter, $forceLeftSideDoor$ und $forceRightSideDoor$, die zwar beide $false$ sein dürfen, (dann wird nichts verhindert), aber nicht beide $true$ sein dürfen.

Die Berechnung von D erfolgt nun wieder über das Aufrufen von Unitys $Random.value$ Attribut, welches einen Pseudozufälligen Wert von $0 - 1$ zurückgibt.

Soll die rechte Seite forciert werden wird der Wert durch 2 geteilt, falls die linke Seite forciert werden soll wird nach dem durch 2 teilen nochmal $0,5$ addiert. Falls nichts blockiert werden soll wird der Wert direkt wiedergegeben.

4.5 Umsetzung der Levelgenerierung

Nachdem ich nun nacheinander die einzelnen Komponenten der Levelgenerierung erklärt habe werde ich nun erklären wie diese zusammenspielen um den Levelgenerierungsalgorithmus zu bilden. Diesen werde ich zunächst in seiner Vorgehensweise erläutern und anschließend die Grundstruktur des Quelltextes darstellen.

4.5.1 Funktionsweise des Levelgenerierungsalgorithmus

Im Rahmen des in dieser Arbeit vorgestellten Experiments war es notwendig, Level zu generieren, die eine vorher festlegbare Länge L, im Sinne der Anzahl der Räume gemeint,

hatten und dementsprechend nicht unendlich lang waren. Jedoch ist es aber durchaus trivial den folgend vorgestellten Algortithmus leicht zu verändern, sodass er die Räume fortlaufend generiert, während die Nutzer:in bereits dabei ist das Level zu erkunden. Nachdem dann L also ausgewählt wurde (in diesem Fall indem die Proband:in eine der Versuchsbedingungen auswählt, siehe Abschnitt x) kann das Level generiert werden.

An dieser Stelle möchte ich noch einmal die Werte auflisten, die vorher von der Designer:in festgelegt werden sollen und kurz zusammenfassen was sie bedeuten.

d : Tiefe der zu generierenden Korridore
h : Höhe der Wände, und des Korridors
w : Die Dicke der Wände und der Wände des Korridors
dw : Die Breite der zu generierenden Türen
dh : Die Höhe der zu generierenden Türen

Grundlegend besteht der Algorithmus aus der Deklarierung einiger Variablen, die den Zustand über die Iterationen hinweg speichern, und einer Schleife, die pro Iteration einen neuen Raum generiert. In der hier vorgestellten Form des Algorithmus wird diese Schleife zu Beginn der Versuchsbedingung ganz ausgeführt, sodass das Level fertig generiert wurde, bevor die Proband:in angefangen hat es zu erkunden. Um den Prozess so umzubauen, dass das Level fortlaufend weiter generiert wird, ist es lediglich notwendig den Quellcode, der innerhalb der Schleife steht in eine Funktion auszuverlagern, die immer dann, wenn die Nutzer:in einen Raum betritt einen weiteren erzeugt. Damit dies nicht bemerkt wird, ist es an dieser Stelle ratsam einen kleinen Puffer von etwa 2 oder 3 Räumen einzubauen, sodass die neu erzeugten Räume nicht direkt vor den Augen der Nutzer:in auftauchen. Dies lässt sich einfach erzielen, indem man zu Beginn des Levels einige wenige Räume erzeugt, und die neuen Räume erst hinter diesen generiert werden.

4.5.2 Inhalt einer Iteration

Eine Iteration besteht aus zwei Bereichen. Im ersten wird zunächst der Raum anhand des aktuellen Zustands der Iterationsübergreifenden Variablen generiert indem die “Generate” Methode der “RoomGenerator” Klasse aufgerufen wird. Der zweite Teil ist dann dafür verantwortlich die nächste Iteration vorzubereiten. Dies hat den Zweck, dass der erste Teil der ersten Iteration, den Ursprungszustand der Variablen nutzen kann. Ein Beispiel dafür sind die vom Oculus Integration SDK übergebenen Eckkoordinaten des realen Trackingspaces, oder die, vom Designer ausgewählte, erste Korridorrichtung, die das Erscheinen im ersten Raum vereinheitlichen soll.

Generierungs-Teil Darin werden dann zunächst die Meshs des Korridors und der ihm innewohnenden Türen erzeugt (siehe Unterabschnitt 4.2.1) und als GameObjects instan-

ziiert. Zudem werden diese mit den entprechenden Materialien, für die Optik, und den entsprechenden Scripts versehen (Beispielsweise werden die Türen mit “Dissolvable”-Scripts ausgestattet, sodass sie sich öffnen, beziehungsweise schließen können).

Dann werden die für den Rotationgain erforderlichen Objekte instanziiert, positioniert und die entsprechenden Mechanismen in die Wege geleitet Zudem wird die Innenausstattung des Korridors instanziiert, positioniert, mit den entsprechenden Scripts ausgestattet und mit den eben erzeugten Mechanismen, die den Rotationgain überwachen verknüpft.

Die Generierung der Wände wird als nächstes in die Wege geleitet, jeh nach Variante wird zwischen den verschiedenen Methoden unterschieden, die danach aufgerufen werden um die Wandpunkte zu berechnen und dann Wandobjekte initialisieren. Hierfür ist zudem erforderlich, das in Abschnitt 4.3 besprochene Prefab, das vorher von der Designer:in erstellt werden muss, zur Verfügung zu stellen.

Berechnungsteil Der zweite Teil einer Iteration ist dann für die Berechnung der Zustandsvariablen zuständig. Diese müssen berechnet werden, weil der erste Teil der nächsten Iteration keinen Zugriff mehr auf die Objekte der aktuellen Iteration hat. Hier findet die in Unterabschnitt 4.1.2 erklärte Eckenberechnung statt. So können die Richtungsvektoren und andere Informationen von dem Raum, der in der aktuellen Iteration generiert wurde genutzt werden um die Generation des nächsten Raumes vorzubereiten. Auch die in Abschnitt 4.4 besprochenene Pseudozufallsvariablen und die in Unterabschnitt 4.3.1 besprochenen Punkte diagonal vor und hinter dem Punkt P des vorherigen Raumes werden hier berechnet, sodass sie in der nächsten Iteration schon feststehen und genutzt werden können.

Kapitel 5

Experiment

5.1 Versuchsaufbau

5.2 Hypothesen

5.3 Versuchsbedingungen

5.3.1 Kontrollbedingung I Joystick

5.3.2 Kontrollbedingung II Teleportation

5.3.3 Versuchsbedingung Real-Walking

Kapitel 6

Ergebnisse

6.1 Teilnehmer

6.2 Methoden

Kapitel 7

Diskussion und Fazit

Kapitel 8

Acknowledgments

Bibliographie

[1] Sarah Chance, Florence Gaunet und Andrew Beall. “Locomotion Mode Affects the Updating of Objects Encountered During Travel: The Contribution of Vestibular and Proprioceptive Inputs to Path Integration”. In: *Presence* 7 (Apr. 1998), Seiten 168–178. DOI: 10.1162/105474698565659.

[2] Lung-Pan Cheng, Eyal Ofek, Christian Holz und Andrew D. Wilson. “VRoamer: Generating On-The-Fly VR Experiences While Walking inside Large, Unknown Real-World Building Environments”. In: *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 2019, Seiten 359–366. DOI: 10.1109/VR.2019.8798074.

[6] Henry Duh, Don Parker und James Philips. “‘Conflicting’ Motion Cues to the Visual and Vestibular Self-Motion Systems Around 0.06 Hz Evoke Simulator Sickness”. In: *Human factors* 46 (Feb. 2004), Seiten 142–53. DOI: 10.1518/hfes.46.1.142.30384.

[7] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[8] Eike Langbehn, Paul Lubos und Frank Steinicke. “Evaluation of Locomotion Techniques for Room-Scale VR: Joystick, Teleportation, and Redirected Walking”. In: *Proceedings of the Virtual Reality International Conference - Laval Virtual*. VRIC ’18. Laval, France: Association for Computing Machinery, 2018. ISBN: 9781450353816. DOI: 10.1145/3234253.3234291. URL: https://doi.org/10.1145/3234253.3234291.

[9] Eike Langbehn, Paul Lubos und Frank Steinicke. “Redirected Spaces: Going Beyond Borders”. In: *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 2018, Seiten 767–768. DOI: 10.1109/VR.2018.8446167.

[10] Sebastian Marwecki und Patrick Baudisch. “Scenograph: Fitting Real-Walking VR Experiences into Various Tracking Volumes”. In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST ’18. Berlin, Germany: Association for Computing Machinery, 2018, Seiten 511–520. ISBN: 9781450359481. DOI: 10.1145/3242587.3242648. URL: https://doi.org/10.1145/3242587.3242648.

[13] Tabitha C. Peck, Henry Fuchs und Mary C. Whitton. “An evaluation of navigational ability comparing Redirected Free Exploration with Distractors to Walking-in-Place and joystick locomotio interfaces”. In: *2011 IEEE Virtual Reality Conference*. 2011, Seiten 55–62. DOI: 10.1109/VR.2011.5759437.

[14] Sharif Razzaque, Zachariah Kohn und Mary C. Whitton. “Redirected Walking”. In: *Eurographics 2001 - Short Presentations*. Eurographics Association, 2001. DOI: 10.2312/egs.20011036.

[15] Roy A. Ruddle und Simon Lessels. “The Benefits of Using a Walking Interface to Navigate Virtual Environments”. In: *ACM Trans. Comput.-Hum. Interact.* 16.1 (Apr. 2009). ISSN: 1073-0516. DOI: 10.1145/1502800.1502805. URL: https://doi.org/10.1145/1502800.1502805.

[16] Mel Slater, Martin Usoh und Anthony Steed. “Taking steps: The influence of a walking technique on presence in virtual reality”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 2 (Dez. 1995), Seiten 201–219. DOI: 10.1145/210079.210084.

[17] Frank Steinicke, Gerd Bruder, Jason Jerald, Harald Frenz und Markus Lappe. “Estimation of Detection Thresholds for Redirected Walking Techniques”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.1 (2010), Seiten 17–27. DOI: 10.1109/TVCG.2009.62.

[18] Evan A. Suma, Zachary Lipps, Samantha Finkelstein, David M. Krum und Mark Bolas. “Impossible Spaces: Maximizing Natural Walking in Virtual Environments with Self-Overlapping Architecture”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.4 (2012), Seiten 555–564. DOI: 10.1109/TVCG.2012.47.

[19] Evan Suma Rosenberg, Gerd Bruder, Frank Steinicke, David Krum und Mark Bolas. “A taxonomy for deploying redirection techniques in immersive virtual environments”. In: *Proceedings - IEEE Virtual Reality* (März 2012), Seiten 43–46. DOI: 10.1109/VR.2012.6180877.

[20] Julian Togelius, Emil Kastbjerg, David Schedl und Georgios Yannakakis. “What is Procedural Content Generation? Mario on the borderline”. In: (Jan. 2011). DOI: 10.1145/2000919.2000922.

[21] Julian Togelius, Georgios Yannakakis, Kenneth Stanley und Cameron Browne. “Search-Based Procedural Content Generation”. In: Apr. 2010, Seiten 141–150. ISBN: 978-3-642-12238-5. DOI: 10.1007/978-3-642-12239-2_15.

[24] Martin Usoh, Kevin Arthur, Mary Whitton, Rui Bastos, Anthony Steed, Mel Slater und Frederick Brooks Jr. “Walking > Walking-in-Place > Flying, in Virtual Environments”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques ACM* (Juni 1999). DOI: 10.1145/311535.311589.

[25] Khrystyna Vasylevska, Hannes Kaufmann, Mark Bolas und Evan A. Suma. “Flexible spaces: Dynamic layout generation for infinite walking in virtual environments”. In: *2013 IEEE Symposium on 3D User Interfaces (3DUI)*. 2013, Seiten 39–42. DOI: 10.1109/3DUI.2013.6550194.

Electronic

[3] Jarosław (Void Room) Ciupiński. *Open world with impossible spaces*. Apr. 2019. URL: https://void-room.itch.io/tea-for-god/devlog/76304/procedural-level-generation-using-impossible-spaces (besucht am 01. 07. 2021).

[4] Jarosław (Void Room) Ciupiński. *Open world with impossible spaces*. März 2021. URL: https://void-room.itch.io/tea-for-god/devlog/235914/open-world-with-impossible-spaces (besucht am 01. 07. 2021).

[5] Jarosław (Void Room) Ciupiński. *Tea for God*. URL: https://void-room.itch.io/tea-for-god (besucht am 01. 07. 2021).

[11] *Oculus Quest 2 Website*. URL: https://www.oculus.com/quest-2/ (besucht am 07. 07. 2021).

[12] *Oculus Quest 2 Website*. URL: https://developer.oculus.com/downloads/package/unity-integration/ (besucht am 07. 07. 2021).

[22] *Unity Asset Store: Dungeon Ground Texture*. URL: https://assetstore.unity.com/packages/2d/textures-materials/floors/dungeon-ground-texture-33296 (besucht am 14. 09. 2021).

[23] *Unity Website*. URL: https://unity.com/ (besucht am 07. 07. 2021).

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek eingestellt wird.

Ort, Datum

Unterschrift

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Mensch-Computer-Innteraktion selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

_____ _____

Ort, Datum Unterschrift