Horrorspiel in der Unreal Engine

von

Linus Abegg*
von Arth, SZ, Schweiz
Klasse 6e

Maturitätsarbeit
eingereicht an der
Kantonsschule Wiedikon

betreut von
Pavel Lunin

Dezember 2022

* Linus Abegg, Zürich, Schweiz. E-Mail: abegglinus@gmail.com

Inhaltsverzeichnis

Vorwort

Du möchtest mein Spiel selbst ausprobieren? Unter dem folgenden Link kannst du das Spiel gratis herunterladen:

---

https://freshefisch.itch.io/the-hunt-for-power

QR-Code zur Spielseite.

Wichtig: Das Spiel ist ausschliesslich für Windows 10/11 Computer. Ausserdem braucht es etwa 500 Megabyte Speicherplatz und läuft auf fast jedem Computer flüssig.

Die Unreal Engine 5.0.3 Projektdatei, in welcher man den Code anschauen kann, ist unter folgendem Link verfügbar:

---

https://www.linus-abegg.ch/redirect/Maturitaetsarbeit-Projektdatei

---

1. Einleitung

Im Rahmen meiner Maturitätsarbeit habe ich zum ersten Mal ein Videospiel erstellt. Ich bin seit klein auf begeisterter „Gamer“, und das Treiben hinter den Kulissen hat mich schon immer interessiert. Vor einigen Jahren hatte mich die Spielentwicklung bereits gepackt, doch schon nach nicht allzu langem experimentieren merkte ich, dass der ganze Prozess zu komplex für mich war.

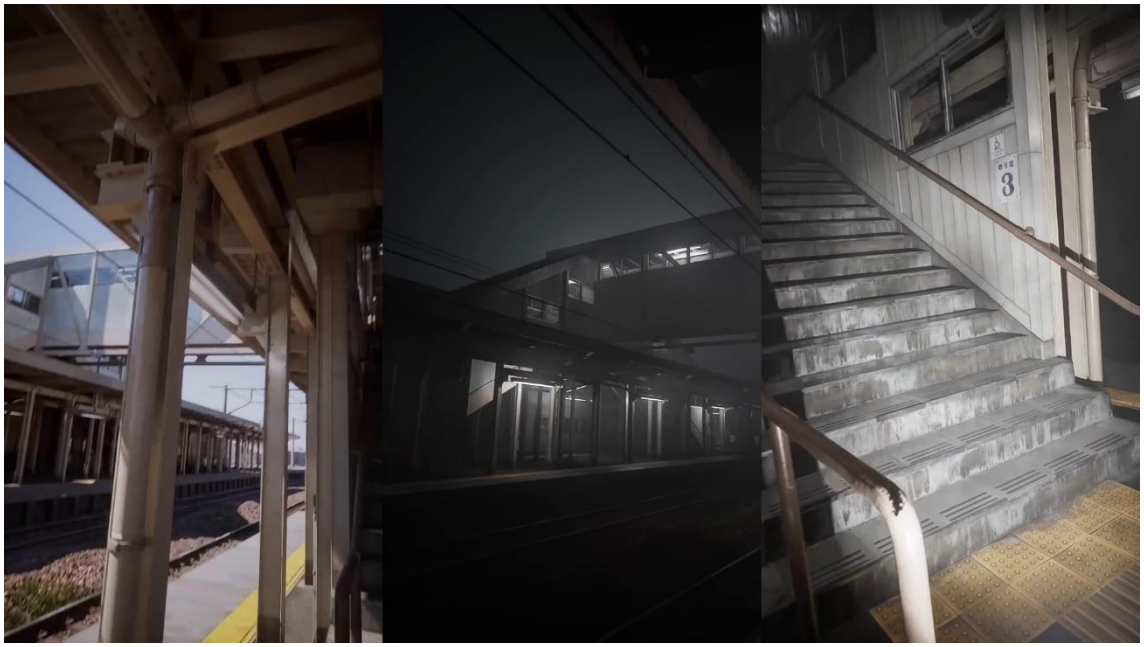
Dann, am 26. Mai 2021, wurde die Testversion von Unreal Engine 5, eine neue Vollversion der *Game Engine* Unreal Engine, veröffentlicht. Eine *Game Engine* ist ein Programm, welches das Erstellen eines Spiels stark vereinfacht, indem es viele wichtige Features eingebaut hat, die nicht von Grund auf programmiert werden müssen. Neben vielen kleinen neuen Features war in der neuen Version auch ein grosses dabei: *Lumen*. Mit dieser speziellen Beleuchtungstechnik kann realistisches Licht in Echtzeit simuliert werden. Im Internet wurde darauf sehr viel über dieses Feature diskutiert. Nutzer teilten ihre Projekte und spornten sich so gegenseitig an, noch schönere und realistischere Szenen zu erstellen.

Abbildung 1: Screenshot von einem Unreal Engine 5 Projekt

Das Bild zeigt ein Muster-Projekt, das von Epic Games selber erstellt wurde, um die neuen Features zu demonstrieren.
Unreal Engine: Welcome to Unreal Engine 5 Early Access, https://youtu.be/d1ZnM7CH-v4, Version 26.5.2021

Durch diese erhöhte Präsenz im Internet habe auch ich wieder von Unreal Engine gehört. Begeistert durch die neuen Features habe ich mich im Internet nach Projekten umgeschaut, die von den neuen Features Gebrauch machten. Ich war fasziniert von den wunderschönen und absolut echt-aussehenden digitalen Kreationen. Ich erinnerte mich an meine gescheiterten Anfangsversuche von damals, doch in den vergangenen Jahren hatte ich sehr viel neues Wissen in der Informatik erworben, zum Beispiel auch die Grundlagen im Programmieren. Also fasste ich neuen Mut und entschloss mich, als Maturitätsarbeit selbst ein Spiel zu erstellen.

Abbildung 2: Drei Ausschnitte aus «etchū-daimon station»



Nutzer «subjectn» teilte im Mai 2022 ein realistisch aussehendes Video von einem Bahnhof. Das ganze Video wurde ausschliesslich in Unreal Engine hergestellt. Auf YouTube wurde das Video über 2 Millionen Mal angeschaut.

subjectn: 【UE5】etchū-daimon station - 越中大門駅, https://youtu.be/2paNFnw1wRs, Version 6.5.2021

2. Entstehungsprozess

2.1. Planung

Als Erstes legte ich die Art des Spieles fest. Schon einige Monate vorher hatte ich begonnen, mich für Horror-Spiele zu begeistern, denn es gibt neben den bekanntesten Spielen wie zum Beispiel jene der «Resident Evil»-Reihe, hinter denen ein riesiges Team mit einem ebenso grossen Budget steckt, unzählige *Indie Games*, die von Einzelpersonen oder kleinen Teams mit meist sehr geringem oder gar keinem Budget entwickelt werden. Diese Spiele sind viel simpler als diejenigen der grossen Hersteller und kosten selten mehr als ein paar Franken, jedoch gibt es sehr viele Leute, die sich stark an ihnen begeistern.

Abbildung 3: «Resident Evil: Village»

Auf dem Bild in der Mitte ist eine der Haupt-Antagonisten des Spieles zu sehen: «Lady Dimitrescu»
Screenshot «Resident Evil: Village»

Oft wird bei diesen *Indie Games* die Bildqualität extra stark reduziert, um die Grafik von alten Spielkonsolen wie jene der *Play Station 2* zu emulieren. Unter anderem durch die reduzierte Auflösung fällt es schwerer, Dinge, die weit weg sind, zu erkennen – was sich natürlich hervorragend für Horror eignet. Zusätzlich gefällt mir dieser «Look», also wollte ich ihn auch für mein Spiel adaptieren. Ausserdem habe ich mich dafür entschieden, dass mein Spiel aus der ersten Perspektive (*First Person*) gespielt werden soll, also aus Sicht des virtuellen Charakters. Das ist die Norm bei Horror-Spielen.

Abbildung 4: «It Steals»

«It Steals» wurde von einer Person entwickelt. Trotzdem begeistert das Spiel sehr viele *Gamer* und *Gamerinnen*.
Zeekerss: It Steals, https://store.steampowered.com/app/1349060/It_Steals/, Version 22. Juli 2020

Beim *Gameplay*, also was man eigentlich im Spiel macht, habe ich mich für etwas entschieden, was bei *Indie Games* verbreitet ist: Der Spieler muss eine gewisse Anzahl Gegenstände aufspüren, während er von einem Computergegner, meist einem Monster, verfolgt wird. Nur wenn der Spieler alle Gegenstände hat, kann er dem Gegner endgültig entkommen und das Spiel gewinnen. «Slender: The Eight Pages» (2012) hat dieses Spielprinzip bekannt gemacht. Im Spiel muss der Spieler acht Blätter finden, wobei er vom «Slender Man» verfolgt wird. Meine Adaption dieser Idee sieht wie folgt aus: In meinem Spiel muss der Spieler 6 zufällig verteilte Batterien in einem labyrinthartigen Level finden, um einen Aufzug aktivieren zu können, mit dem er aus dem Labyrinth fliehen kann. Dabei wird er von einem Monster gejagt.

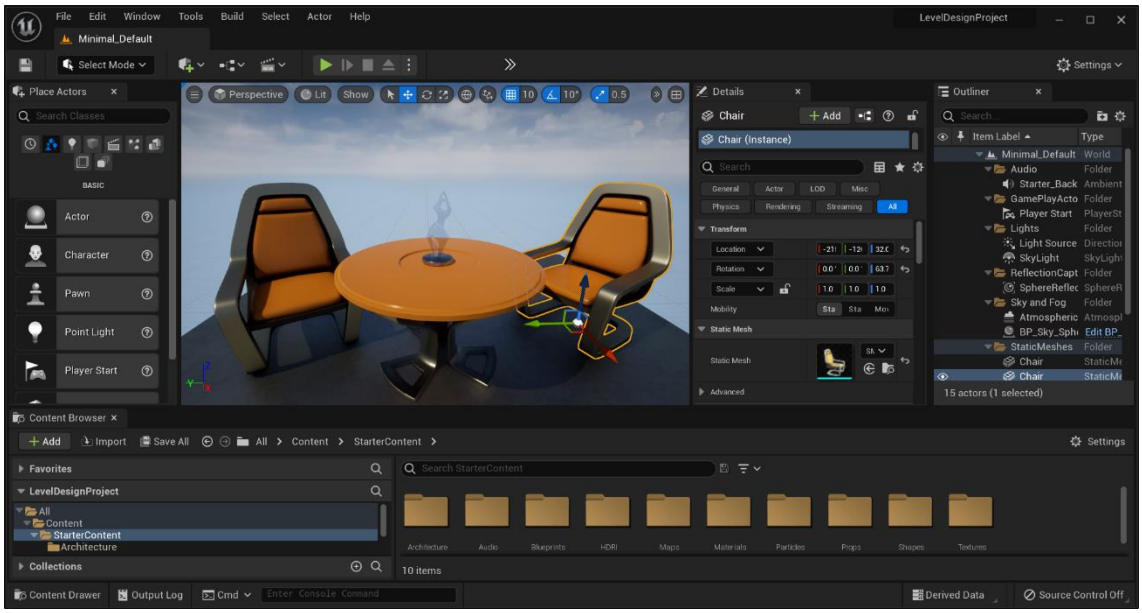
Abbildung 5: «Slender: The Eight Pages»

Ausschnitt aus «Slender: The Eight Pages». Rechts im Bild ist «Slender Man» zu sehen.
Gamer Max Channel: Slender: The Eight Pages - Walkthrough 8/8 Pages Gameplay, https://youtu.be/9BMZ6pDiHC4, Version 26.12.2016

2.2. Game Engine: Unreal Engine

Unreal Engine ist eine von Epic Games entwickelte *Game Engine*, die beim Entwickeln von Spielen genutzt wird. Das besondere an ihr ist, dass man gratis auf alle Features Zugriff hat, egal für welchen Zweck oder mit welchem Budget man sie nutzen möchte. Ein paar nennenswerte Spiele, die mit Unreal Engine entwickelt wurden, sind «Fortnite» (2017), «Sea Of Thieves» (2018), «Star Wars Jedi: Fallen Order» (2019) und «Yoshi's Crafted World» (2019).

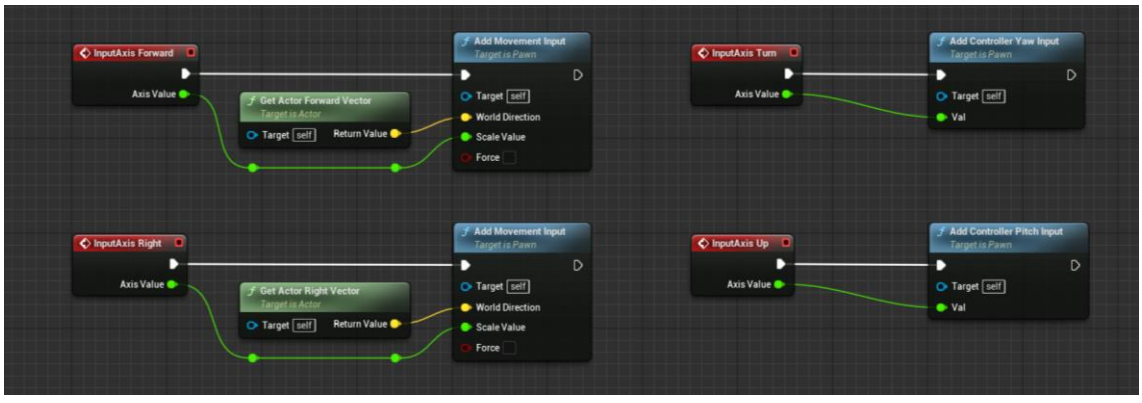
Abbildung 6: Unreal Engine 5 Benutzeroberfläche



Auf diesem Bild kann man den «Level Editor» von Unreal Engine 5 sehen. Hier wird die digitale Welt gestaltet.
Unreal Engine 5 Documentation: Level Editor, https://docs.unrealengine.com/5.0/en-US/level-editor-in-unreal-engine/, Version 17.12.2022

Programmieren in der Unreal Engine sieht meistens anders aus als in ähnlichen Programmen: Anstatt mit Text zu programmieren, wie zum Beispiel bei *Game Engine* Unity, werden sogenannte *Blueprints* verwendet. In *Blueprints* wird programmiert, indem verschiedene *Nodes* aneinandergehängt werden. Man kann aber auch mit *C++*, eine konventionelle Programmiersprache, programmieren.

Abbildung 7: *Blueprint* Ausschnitt (*Player*)



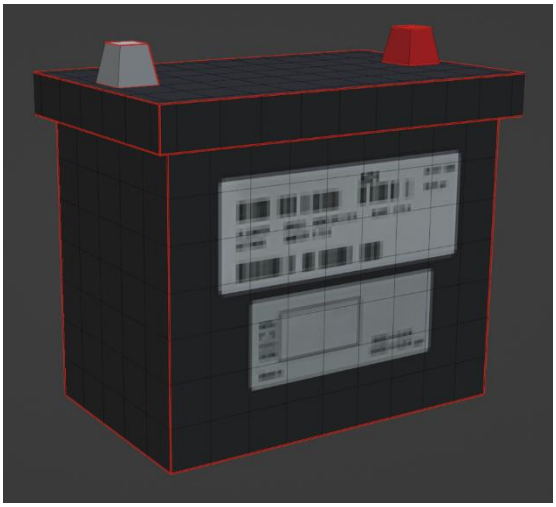
Der obige *Blueprint* stammt aus meinem Spiel. Er erlaubt dem Spieler, sich zu bewegen und sich umzuschauen.

2.3. Erstellen der 3D-Modelle

Wenn man in seinem Spiel beispielsweise eine Taschenlampe haben möchte, dann muss man ein 3D-Modell einer Taschenlampe einfügen. Ein 3D-Modell enthält Informationen zu Form und Geometrie, aber auch zur Oberfläche, wie unter anderem zu Farbe und Rauheit (auf Englisch *roughness*). 3D-Modelle können auch vom Internet heruntergeladen werden, aber für meine Arbeit habe ich alle selbst hergestellt (ausgenommen das Modell des Monsters und die Texturen der Modelle).

In Unreal Engine gibt es nur sehr einfache Werkzeuge für das Erstellen und Bearbeiten von 3D-Modellen. Viel weiter fortgeschritten ist dafür das 3D-Design- und Animationsprogramm Blender. Mit Blender kann man professionelle Animationsfilme à la Disney erstellen, aber auch 3D-Modelle designen und bearbeiten. Für meine Arbeit habe ich mich lediglich auf das letztere fokussiert.

Abbildung 8: Batterie in 3D-Designprogramm



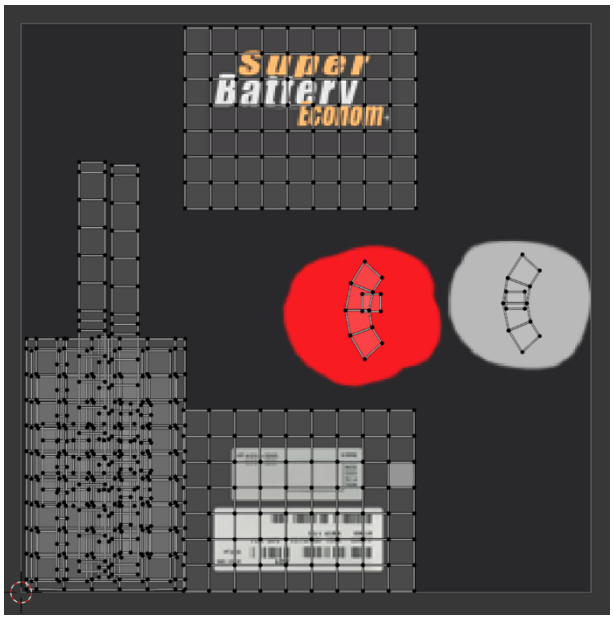
Autobatterie mit Etikette

3D-Modelle herstellen ist ein komplexer Prozess. Grob gesagt verläuft er wie folgt: Zuerst suche ich im Internet nach Bildern des Objekts, dass ich erstellen möchte. Als Nächstes beginne ich in Blender, die Geometrie zu erstellen. Dafür gibt es verschiedene Werkzeuge: Das eine ist sehr mathematisch angelegt, mit Einheiten und Koordinaten, beim anderen skulptiert man mit der Maus, wie zum Beispiel beim Töpfern. In meinem Spiel habe ich alle 3D-Modelle mit dem mathematischen Werkzeug erstellt, da die Objekte (Autobatterie, Ausgangs-Schild, etc.) sehr klar definierte Ecken haben. Das Skulptier-Werkzeug eignet sich besser für organische Objekte, wie zum Beispiel ein Brötchen oder einen Donut. Hierbei erstelle ich zunächst die Grundform des Objektes, im Fall der Autobatterie einen Würfel. Dann passe ich die Form an, verformte zum Beispiel den Würfel zu einem Quader, der mehr der Form einer Autobatterie ähnelt. Dann füge ich die kleineren Details der Oberfläche hinzu. Im Fall der Autobatterie einen überragenden Rand auf deren Oberseite und zwei Anschlüsse obendrauf. Nun ist der geometrische Teil des Modells fertig, jedoch fehlt noch das Aussehen.

Um dem 3D-Modell eine Oberfläche zu geben, verwendet man Bilder, sogenannte Texturen. Wenn die Textur hinzugefügt wird, wird sie zunächst meist falsch dargestellt. Die Textur hat nämlich keine Ahnung, wie sie auf das 3D-Modell projiziert werden soll. Dafür wird eine sogenannte *UV-Map* benötigt. Mit ihr kann festgelegt werden, welcher Teil der Textur auf welchen Teil des 3D-Modells projiziert werden soll. In meinem Fall der Autobatterie musste ich eine solche *UV-Map* verwenden, damit die Etikette auf der richtigen Seite und richtig ausgerichtet ist, und damit die Anschlüsse auf der oberen Seite der Batterie die richtigen Farben haben. Bei komplexeren 3D-Modellen werden oft auch noch weitere Texturen verwendet, zum Beispiel eine *Normal-Map*, die Informationen zur

Oberflächenbeschaffenheit des 3D-Modells enthält. In meinem Spiel habe ich aber solche zusätzlichen Texturen wegen der reduzierten Grafik nicht verwendet, da diese kleinen Details nicht sichtbar wären.

Abbildung 9: Textur der Batterie mit *UV-Map* in Blender



Mit der obigen UV-Map wird der Batterie an den richtigen Orten die richtige Oberfläche zugeordnet.

2.4. Programmieren mit Blueprints

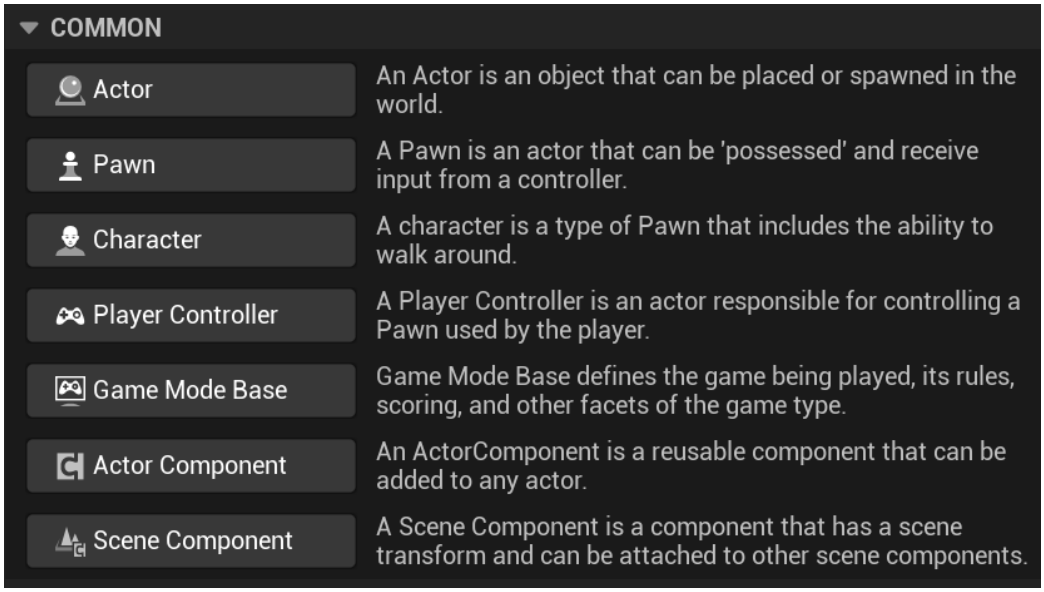
Wie schon in Abschnitt 2.2. erklärt, wird in der Unreal Engine meist mit sogenannten *Blueprints* programmiert. Weil sie mir einfacher als *C++* erschienen sind und weil die meisten Internet-Tutorials sie verwenden, habe ich mich dafür entschieden, *Blueprints* zu verwenden.

In Unreal Engine gibt es verschiedene Klassen von *Blueprints*. Viele von ihnen sind optional, das heisst, sie werden für ein funktionierendes Spiel nicht benötigt, jedoch für bestimmte Funktionen. Ein Beispiel ist der *HUD-Blueprint*, mit dem Informationen auf dem Bildschirm dargestellt werden können. Ein *HUD* ist zwar für ein Spiel nicht zwingend notwendig, wird aber in praktisch allen Spielen verwendet, zum Beispiel für das Anzeigen der Leben des Spielers.

Hier eine Liste der wichtigsten *Blueprint*-Klassen:

Blueprint-Klasse	**Nutzen**	**Beispiel**
Actor	Für Objekte, die man in seinem Level platzieren oder «spawnen» lassen möchte. Ein «Actor-Blueprint» kann Code, 3D-Modelle, Ton, etc. enthalten.	Türe, KI-Gegner, Waffe
Character	Wird für das, was der Spieler steuern sollte, verwendet.	Je nach Spiel: Mensch, Wesen, Insekt, etc.
Player Controller	Bezieht sich auf einen «Character-Blueprint». Er steuert wichtige Werte wie zum Beispiel die Geschwindigkeit des Spielers.	/
Game Mode	Hier werden die Spielregeln festgelegt. Meist wird hier der «Character-Blueprint» verwendet, beispielsweise um abzufragen, wie viele Münzen der Spieler besitzt.	Wenn der Spieler 10 Münzen hat, gewinnt er das Spiel.

Abbildung 10: Gängige *Blueprint-Klassen*

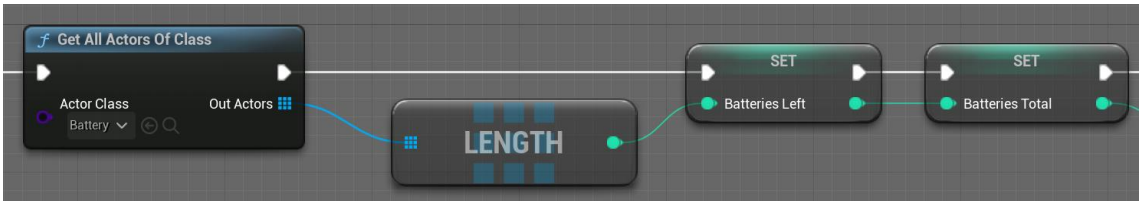


Screenshot Unreal Engine 5

In einem Blueprint wird mit *Nodes* anstatt mit Text programmiert. Die einzelnen *Nodes* in einem Blueprint kann man sich als Bausteine vorstellen. Die meisten dieser Bausteine haben jeweils einen weissen Ein- und Ausgang. So kann man den weissen Ausgang einer *Node* an den Eingang einer anderen anhängen, so dass die *Nodes* in dieser Reihenfolge ausgeführt werden, ähnlich wie bei Scratch, einem Programm, das primär genutzt wird, um Kindern das Programmieren näher zu bringen. Gewisse *Nodes* haben nur einen weissen Ausgang und keinen Eingang. Das sind in der Regel sogenannte *Event-Nodes*. Von ihnen wird eine Aktion gestartet. Beispielsweise gibt es eine *BeginPlay Node*, welche beim Start des Spiels ein Signal aussendet.

Neben den weissen Ein- und Ausgängen der *Nodes* gibt es auch noch weitere, je nach *Node*. Die verschiedenen Farben von den Ein- und Ausgängen stehen zum Beispiel für Vektoren (Gelb), für *Floats*, also Dezimalzahlen (Grün) und für *Booleans*, also Wahr- oder Falschwerte (Rot). Diese Werte können dann von einem Ausgang einer *Node* in den Eingang einer anderen *Node* eingeführt werden, um den Wert zu übermitteln, oder die Werte können auch in einer Variabel für den späteren Gebrauch gespeichert werden.

Abbildung 11: *Blueprint* Ausschnitt (*GameMode*)



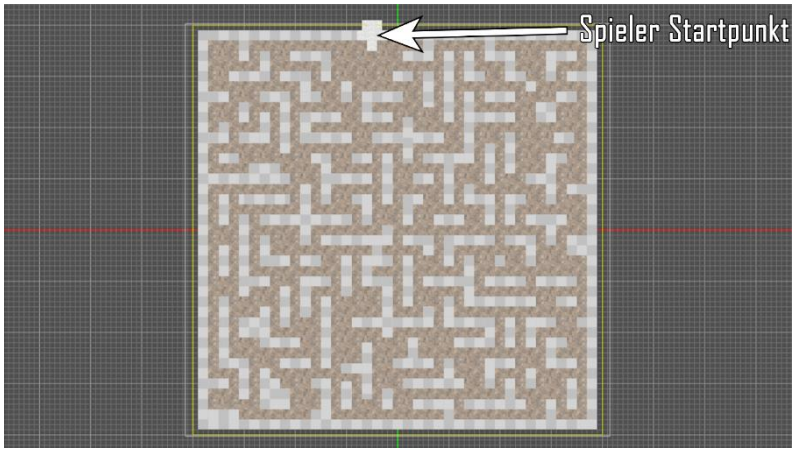
Der obige Ausschnitt aus meinem *GameMode Blueprint* sucht mit der *Get All Actors Of Class [Battery] Node* nach allen Batterien im Level. Dann findet er mit der *Lenght Node* die Anzahl heraus und speichert diese in zwei Variabeln, die verwendet werden, um dem Spieler dann mitzuteilen, wie viele Batterien er noch braucht.

In meinem Spiel habe ich eine Vielzahl von *Blueprints* verwendet, unter anderem auch von den in der Tabelle in Abschnitt 2.4. erwähnten mindestens jeden einmal. Den *Actor Blueprint* habe ich aber mit Abstand am häufigsten verwendet: Für die Lampen an den Decken, die Batterien, die der Spieler aufsammeln muss, das Monster und den Lift.

2.5. Level-Design

Beim Leveldesign bin ich wie folgt vorgegangen: Zuerst habe ich eine Grundfläche festgelegt, um darauf das Labyrinth zu erstellen. Ich habe mich nach etwas ausprobieren für 75 mal 75 Meter entschieden, damit das Level nicht riesig, aber doch gross genug ist, um die Orientation zu erschweren. Sodann habe ich die Wände hinzugefügt. Dabei habe ich mich für ein Gittersystem von 2 mal 2 Meter entschieden. Die Wände fügte ich jeweils zufällig hinzu, aber immer so, dass die maximale Sichtdistanz nicht allzu weit ist. Im nächsten Schritt habe ich das Level mehrmals virtuell erkundet und die Wände angepasst, bis ich zufrieden war.

Abbildung 12: Layout meines Spiels



Fogelperspektive von meinem Level in Unreal Engine

Nachdem ich das Grundlayout fertig hatte, wandte ich mich den kleineren Details zu. Ich habe verschiedene Graffitis und Bluteffekte im Level verteilt, um das Level etwas abwechslungsweiser zu Gestallten. Zudem habe ich einige Ausgangschilder verteilt, die jeweils in die Richtung des Lifts zeigen. So kann sich der Spieler besser orientieren und findet den Lift, wenn er alle Batterien hat. Zum Schluss habe ich noch *Target Points* hinzugefügt. Hierzu habe ich im *Level Blueprint* programmiert, dass die Batterien an einer zufälligen Auswahl dieser Punkte erscheinen. So kann sich der Spieler nicht einfach die Orte merken.

Abbildung 13: Level Design Details

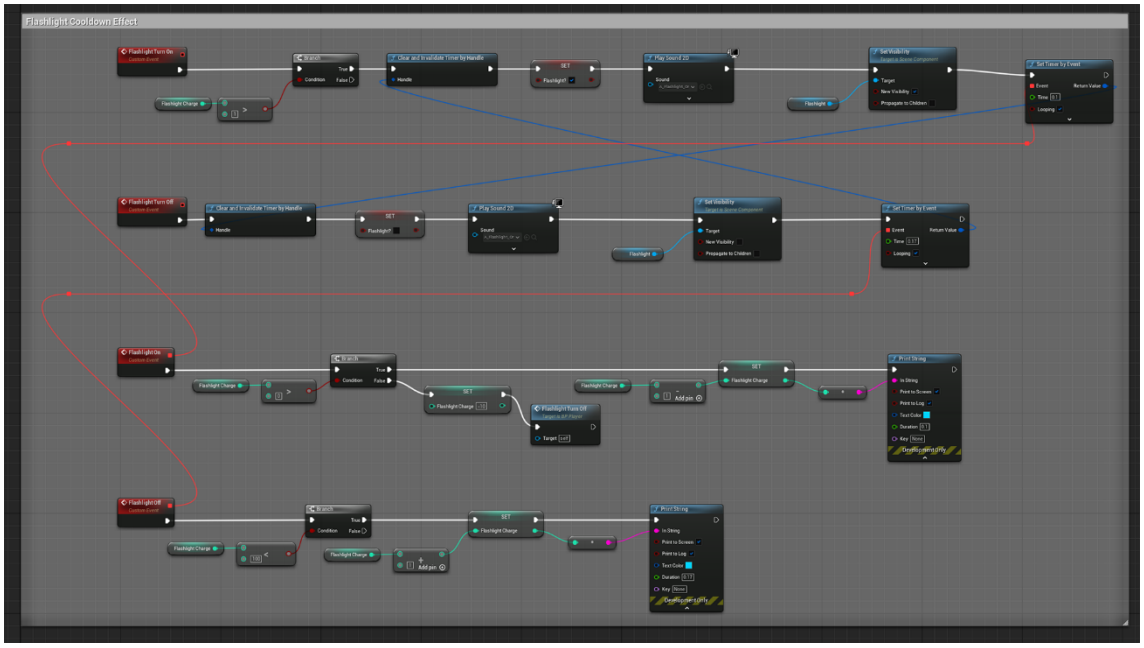
Am Boden ist eine grosse Blutlache zu sehen und an der Wand ein Graffiti.

3. Endprodukt

Wie erwähnt, ist der Spieler in meinem Spiel in einem von einem Monster bewohnten Labyrinth gefangen und muss 6 Batterien finden, um einen Lift zu aktivieren, mit dem er entkommen kann.

Der Spieler selbst hat kein 3D-Modell, was aber gar nicht benötigt wird, da das Spiel in *First Person* stattfindet. Der Spieler kann sich durch das Bewegen der Maus umschauen und mit den Tasten W, A, S und D in alle 4 Richtungen laufen. Ausserdem kann er mit der Taste F eine Taschenlampe aktivieren, um sich auch in den dunklen Teilen des Levels orientieren zu können. Die Taschenlampe basiert auf einem *Cooldown Prinzip*, das heisst, dass die Taschenlampe nur für eine begrenzte Zeit verwendet werden kann und dann wieder passiv aufgeladen werden muss, indem sie nicht verwendet wird. Ich habe das so programmiert, damit der Einsatz der Taschenlampe nur im Notfall erfolgt und der Spieler sich immer überlegen muss, ob er sie wirklich braucht. Die Taschenlampe wurde ausserdem mit einem Knopfdruck-Soundeffekt ausgestattet, der beim Ein- und Ausschalten ertönt.

Abbildung 14: Blueprint der Taschenlampe



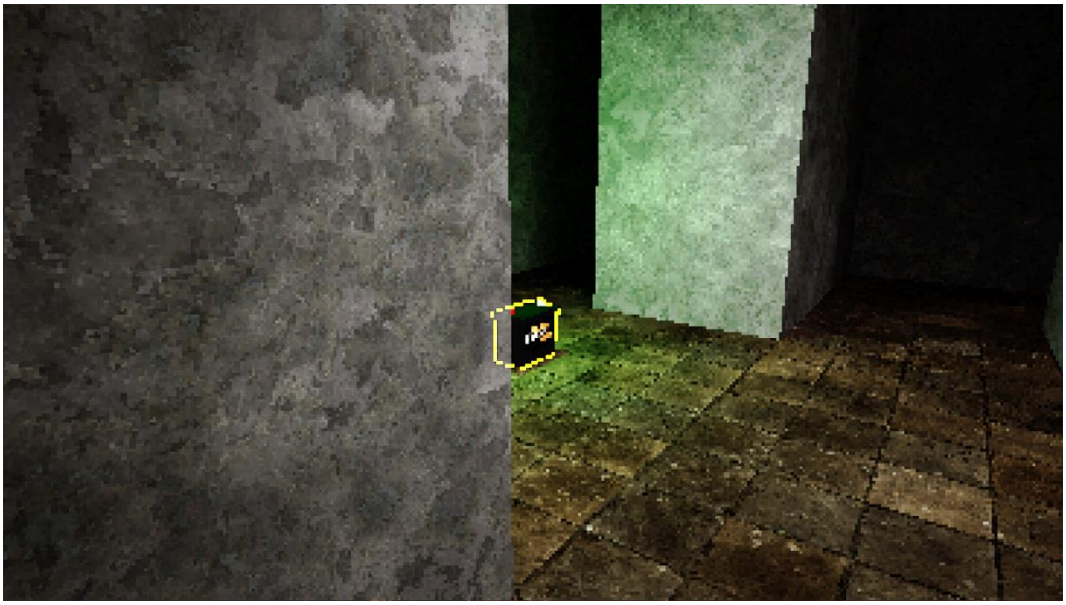
Der vollständige Code meiner Taschenlampe. Der Code kann zum genaueren Anschauen heruntergeladen werden (siehe Vorwort).

Um die Bewegung des Spielers noch etwas realistischer zu machen, fügte ich beim Bewegen ein geringes Kamerawackeln ein, um ein Nicken des Kopfes zu simulieren. Im gleichen Intervall fügte ich auch Schritt-Soundeffekte ein, welche das Laufen realistischer erscheinen lassen. Ein Problem war dabei, dass die Schritteffekte viel zu repetitiv waren, da immer dieselbe Audiodatei abgespielt wurde. Deshalb habe ich schliesslich insgesamt 6 leicht verschiedene Audiodateien importiert und programmiert, dass jeweils eine Audiodatei zufällig ausgewählt und abgespielt wird. Zusätzlich habe ich eine zufällige Variation von Lautstärke und Tonhöhe der Datei programmiert. Damit erscheint insgesamt die Bewegung des Spielers sehr realistisch.

Bei den Batterien habe ich mich für Autobatterien entschieden, da diese eine gewisse Grösse aufweisen und für den Spieler gut sichtbar sind. Beim Lift habe ich selbst ein simples 3D-Modell von einem industriellen Lift angefertigt, also ohne Türen oder Wänden, da ich fand, dass das besser zum Level passt.

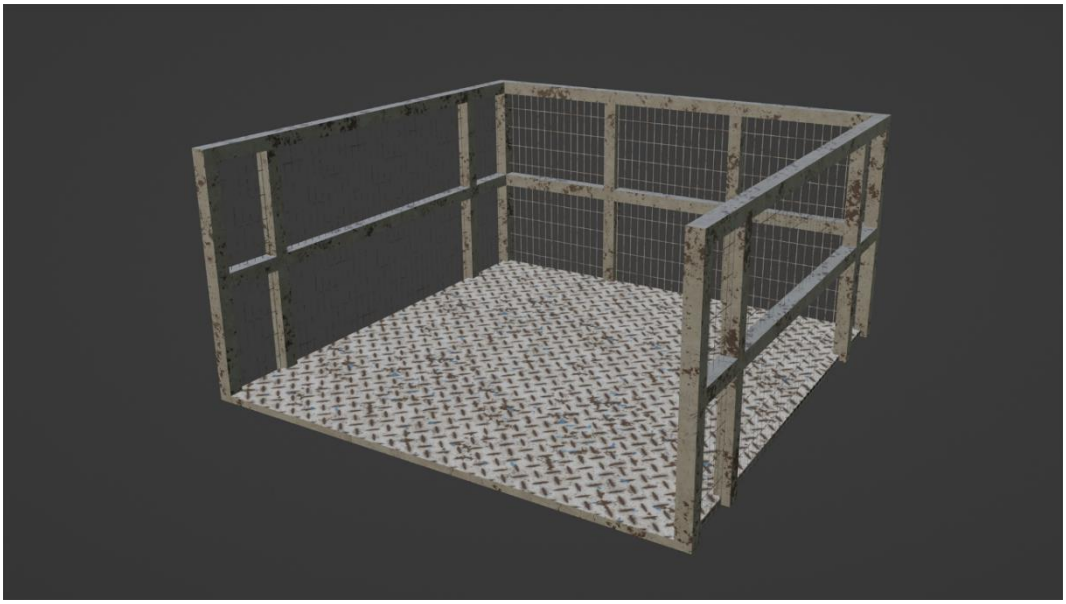
Die Verteilung der Batterien erscheint dem Spieler, wie bereits erwähnt, zufällig. Das Spiel entscheidet sich beim Spielstart für eine von mehreren möglichen Verteilungen der Batterien. Die Batterien sind also nicht vollständig zufällig platziert, damit die Batterien gleichmässig verteilt sind. Der Spieler wird zu Beginn des Spiels jeweils im Lift gespawnt.

Abbildung 15: Batterie in meinem Spiel



Screenshot mit Batterie in der Mitte

Abbildung 16: Lift in 3D-Designprogramm

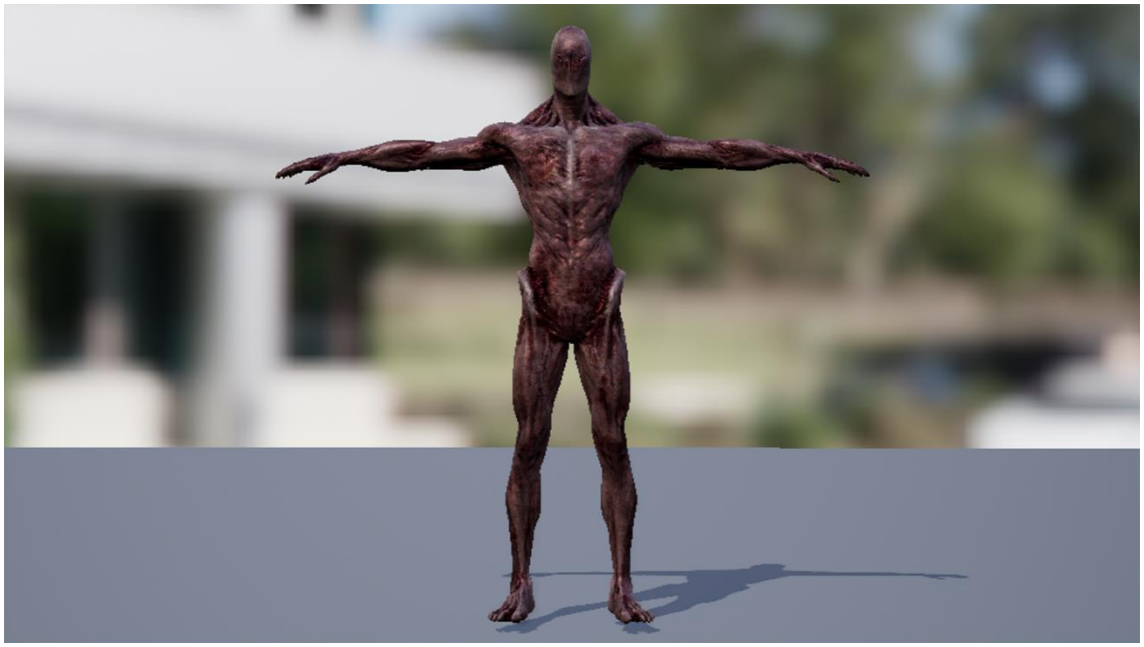


Mein Aufzug in Blender

Das Monster existiert bei Spielbeginn noch gar nicht. Erst wenn der Spieler die ersten Batterien einsammelt, wird das Monster in einer zufälligen Ecke des Labyrinths gespawnt. Das Monster stösst in einem ungefähr 8 Meter grossen Radius ein lautes Geräusch aus, womit der Spieler erahnen kann, woher es kommt. Es bewegt sich zunächst mit der Hälfte der Geschwindigkeit des Spielers auf den Spieler zu. Mit jeder Batterie, die der Spieler einsammelt, erhöht sich die Geschwindigkeit des Monsters, bis es am

Schluss, wenn der Spieler alle Batterien hat, gleich schnell wie der Spieler ist. Der Spieler kann sich nun keinen Fehler mehr erlauben und muss direkt zum Lift rennen. Dabei helfen ihm die spärlich an der Decke gehängten Ausgangsschilder, die in die Richtung des Lifts zeigen. Wenn der Spieler dem Monster zu nahekommt, erscheint eine kleine Game-Over Sequenz, in der sich der Spieler zum Monster umdreht und das Monster in die Richtung des Spielers schlägt. Dann wird der Spieler zurück ins Hauptmenu geführt, wo er einen neuen Versuch starten kann. Falls der Spieler erfolgreich entkommt, wird auf dem Bildschirm gross «Escaped» dargestellt, und der Spieler wird danach ebenfalls zurück ins Hauptmenu geführt.

Abbildung 17: Monster 3D-Modell



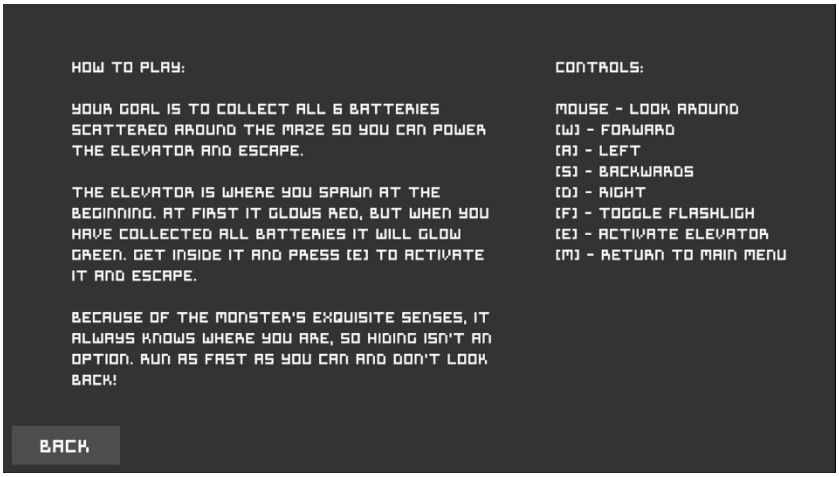
Das 3D-Modell inklusive Texturen und Animationen wurde von mixamo.com heruntergeladen.

Um auch wichtige Informationen wie eine Spielanleitung und die Quellen der nicht von mir selbst hergestellten Elemente meines Spieles in meinem Spiel unterbringen zu können, habe ich mich entschieden, ein einfaches Hauptmenu zu erstellen. Es gibt einen Knopf «Start» (Spiel starten) und «Quit» (Spiel beenden) und daneben auch «Help», welcher eine Spielanleitung zeigt, sowie «Credits», welcher meine Quellen anzeigt.

Abbildung 18: Hauptmenu meines Spiels

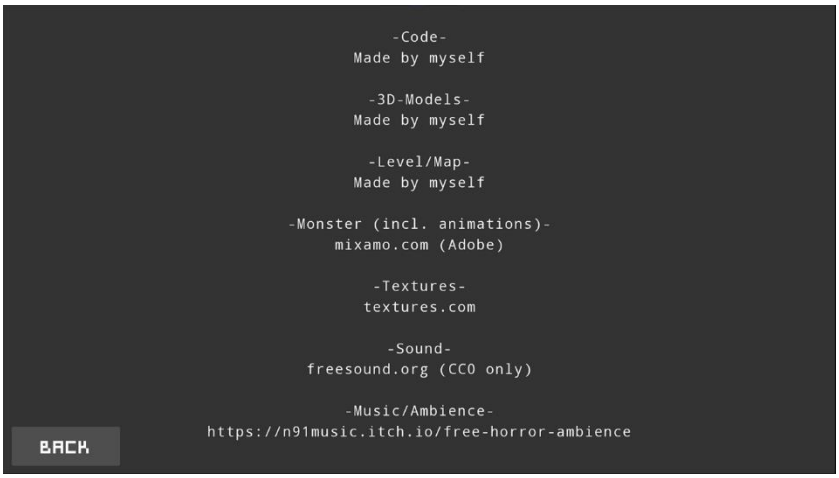
Das Hauptmenu mit verschiedenen Optionen.

Abbildung 19: «Help» Menu



«Help» Menu

Abbildung 20: «Credits» Menu



«Credits» Menu

4. Fazit

Bereits in meinem Grobkonzept habe ich detaillierte Ideen für das *Gameplay* festgelegt. Beispielsweise das Monster sollte verschiedene Phasen haben, je nachdem wie viele Batterien der Spieler hat und ob es den Spieler sieht oder nicht. Beim Programmieren selbst aber habe ich schnell gemerkt, dass meine Ideen, die ich für simpel hielt, sehr schwierig umzusetzen waren. Passende Anleitungen im Internet verwendeten eine Vielzahl der kompliziertesten Funktionen von Unreal Engine, und schliesslich habe ich mich entschieden, die Funktionalität meines Monsters neu zu überdenken. So habe ich mich auf das in Abschnitt 3. beschriebene Prinzip festgelegt.

Meine Grundkenntnisse im Programmieren haben mir zwar sehr geholfen, um das Prinzip des Programmierens mit *Blueprints* besser zu verstehen, jedoch haben sie nicht ausgereicht. Programmieren in Unreal Engine, oder besser gesagt in *Game Engines* allgemein, ist sehr viel komplexer als beispielsweise die Programmiersprache Python, mit der ich mich relativ gut auskenne. Zu den Features wie zum Beispiel der Taschenlampe gehört noch viel mehr als nur der Code des *Cooldown-Effekts*. Die Taschenlampe muss an der Richtigen Stelle hinzugefügt werden, damit sie sich mit dem Spieler bewegt, sie muss die richtige Helligkeit und den richtigen Winkel haben, und so weiter. Um zu verstehen, wie *Unreal Engine* richtig genutzt wird, habe ich zahllose Videoanleitungen auf YouTube angeschaut, welche oft sehr gut gemacht sind und für welche man nicht bezahlen muss.

Beim Verwirklichen meines Spieles gab es oft Momente, in denen etwas nicht funktioniert hat. Durch meine Grundkenntnisse im Programmieren war mir aber die Vorgehensweise beim *Debugging* (Englisch für Fehler beseitigen) bereits klar und so konnte ich in den meisten Fällen eine Lösung finden. In seltenen Fällen aber konnte ich ein Problem nicht lösen. Beispielsweise wollte ich eine Sprint-Funktion einbauen, jedoch habe ich es nicht geschafft, dass die Schritteffekte schneller abgespielt werden, und so sah das Sprinten nicht gut aus. Schlussendlich habe ich mich dann dafür entschieden, die Grundgeschwindigkeit des Spielers zu erhöhen und dafür keine Sprintfunktion hinzuzufügen. Rückblickend denke ich, dass das die richtige Entscheidung war. Ich hätte noch sehr viel mehr Zeit aufwenden können, nur um am Schluss möglicherweise trotzdem ohne Lösung dazustehen.

In der finalen Phase meines Spiels habe ich noch ein Paar meiner Freunde das Spiel ausprobieren lassen. Insgesamt kam das Spiel gut an und die Schwierigkeit schien angemessen zu sein. Lediglich einige kleinere Änderung nahm ich darauf noch vor.

Insgesamt habe ich mein Ziel erreicht. Ich habe ein funktionierendes Videospiel mit Unreal Engine 5 erstellt. Auch konnte ich meine ursprüngliche Idee (siehe Abschnitt 2.1.) verwirklichen.

Beim Arbeiten an meinem Spiel habe ich unzählige Dinge gelernt, vom Erstellen von 3D-Modellen über Leveldesign bis zum Programmieren mit Blueprints. Im Vergleich zu damals, als ich zum ersten Mal mit Unreal Engine experimentiert habe, bin ich nun zuversichtlich gestimmt für zukünftige Projekte. Möglicherweise werde ich auch Projekte mit anderen *Game Engines* ausprobieren, wie zum Beispiel Godot oder Untity, in welchen nicht mit *Blueprints*, sondern mit traditionellem Code programmiert wird. Ausserdem kann ich mir nun ein Studium in *Game Design* oder in einer ähnlichen Richtung sehr gut vorstellen.

Abbildung 21: Freund spielt mein Spiel

Abbildungsverzeichnis

Bestätigung der Eigentätigkeit

Ich bestätige mit meiner Unterschrift, dass ich meine Maturitätsarbeit selbständig verfasst und in schriftliche Form gebracht habe, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind. Mir ist bekannt, dass eine Maturitätsarbeit, die nachweislich ein Plagiat gemäss der in der Maturitätsarbeitsbroschüre gegebenen Definition darstellt, als schwerer Verstoss im Sinne des Maturitätsprüfungsreglements gewertet wird.

Unterschrift	Ort und Datum