# Technische Dokumentation - Gods of Akragas

# 1. Einführung und Ziele

## 1.1 Aufgabenstellung

Die grundlegende Aufgabe für unser Team war die Entwicklung eines Computerspiel-Prototypen und Konzeptes in Zusammenarbeit mit dem Lehrstuhl der Archäologie der Universität Augsburg auf Grundlage Ihrer Ausgrabung in Agrigent.

Daraus entwickelte sich die Idee für Gods of Akragas, ein Metroidvania-Sidescroller, bei dem die antike griechische Götterwelt im Mittelpunkt steht.

Geplante Features:

- Ein Movementsystem zum Erkunden des Levels
- Ein LayerChangesystem zum Aufbrechen der 2 Dimensionalität
- Ein simples Kampfsystem
- Ein Dialogsystem
- ein Opfersystem, man sollte Items opfern können, um vielleicht bessere zu erhalten (Resourcenmanagment)
- Ein Heilungs- und Respawnsystem, bei welchem man “Wein” opfern oder konsumieren kann, um entweder Respawnpunkte freizuschalten, oder sich zu Heilen. (Resourcenmanagment)
- antike griechische Spiele als Minispiele
- ...und noch ein paar mehr

## 1.2 Zeitlicher Rahmen

Zur Umsetzung dieses Spiels hatten wir jedoch nur ein Semester Zeit, also c.a. 4 Monate, was für ein ganzes Spiel diesen Ausmaßes einfach zu wenig ist, daher beschlossen wir nur ein paar der vorausgehenden Features auch wirklich ins Spiel zu übernehmen und nur einen kleinen Teil des gesamten Levels zu bauen, sodass es möglich ist, den Anfang der

Story des Spiels zu erleben, und der Spieler so einen guten Einblick davon bekommt, wie das fertige Spiel aussehen würde.

Features zur Implementierung festgesetzt:

- Das Movementsystem
- Der Layerchange
- Das Kamfpsystem
- Das Dialogsystem

In diesem Dokument werden also die nur die oben genannten Features behandelt, mit Ausnahme des Dialogsystems, da dieses zum jetzigen Zeitpunkt noch nicht implementiert worden ist und sich daher das System sicherlich noch vom geplanten ändern wird. Auch nicht behandelt werden gestalterische Entscheidungen und Prozesse, da dies für die technische Seite nicht relevant ist.

## 1.3 Qualitätsziele

Die folgendes Tabelle beschreibt die wichtigsten Qualitätsziele von Gods of Akragas, wobei die Reihenfolge grob die Wichtigkeit der Ziele angibt.

| **Qualitätsziel** | **Motivation und Erläuterung** |
|---|---|
| historisch akkurate Assets | Das Spiel soll einen Einblick in die griechische Antike gewähren |
| Gut aussehendes und gut spielbares erstes „Level“ | Da wir kein ganzes Spiel entwickeln können, wollen wir so gut es geht zeigen, wie das fertige Spiel aussehen bzw. sich spielen würde. |
| Gute Atmosphäre | Soll den Spieler dazu bringen sich mehr mit dem Spiel zu beschäftigen. |
| Schnell anpassbarer Code | Computerspiele sind sehr komplex, daher sollte der Code schnell anpassbar sein um Fehler schnell beheben zu können. Aber es macht auch die Weiterentwicklung deutlich leichter. |
| Übersichtlicher Code | Damit ein anderer Entwickler schnell die Struktur des Codes versteht. Dadurch sind Fehler besser zu finden und das Spiel lässt sich besser weiterentwickeln. |
| Gute Performance | Das Spiel soll auch auf nicht ganz so guten Rechnern laufen können. |

## 1.4 Stakeholder

| **Wer?** | **Interesse, Bezug** |
|---|---|
| Der Spieler | ● Will ein Spiel, welches Spaß macht<br>● Will eine gute Atmosphäre |

| | |
|---|---|
| | ● Will eine interessante und packende Story<br>● Will eine liebevolle und detailreiche Welt zum erkunden<br>● Kennt sich mit dem antiken Griechenland nicht unbedingt gut aus |
| Studenten und Professoren des Lehrstuhl Archäologie der Uni Augsburg | ● Recherchieren für die Teammitglieder<br>● Wollen ein historisch akkurat dargestelltes Spiel<br>● Wollen, dass der Alltag der Charaktere von historischen Befunden gestützt wird<br>● Wollen, dass die antike griechische Götterwelt aus Sicht der antiken Griechen richtig dargestellt wird<br>● Wollen ein Spiel, das Spaß macht<br>● Die Technik dahinter interessiert sie nicht wirklich<br>● Besitzen wenig Kenntnisse über die technische Seite eines Spiels |
| Prof. Jens Müller | ● Berät bei gestalterischen und konzeptionellen Fragen<br>● Will ein gutes Spielkonzept<br>● Will ein Spiel, das Spaß macht<br>● Will ein gut gestaltetes Spiel |
| Prof. Dr. Thomas Rist | ● Berät bei technischen und konzeptionellen Fragen<br>● Will ein gutes Spielkonzept<br>● Will ein Spiel, das Spaß macht<br>● Will eine gute technische Ausführung |
| Teammitglieder (Studenten der Hochschule Augsburg, Projektteam Nebel von Agrigent) | ● Erstellt Assets/Konzepte/Code<br>● Wollen ein gutes Spiel erstellen<br>● Wollen ein Spiel, das Spaß macht<br>● Wollen ein gut gestaltetes Spiel<br>● Wollen eher eine gute Demo erstellen als einen Prototypen<br>● Wollen gut aussehende Assets |
| Entwickler | ● Entwickeln den Code des Spiels<br>● Wollen was Teammitglieder wollen<br>● Wollen einen übersichtlichen Code<br>● Wollen schnell anpassbaren Code<br>● Wollen guten Code schreiben |

# 2. Randbedingungen

## 2.1 Technisch

| **Randbedingung** | **Erläuterung, Hintergrund** |
|---|---|
| Breite Hardwareausstattungen | Das Spiel soll auf recht aktuellen Systemen laufen und dabei nicht zu anspruchsvoll sein, damit es eine große Anzahl an Spielern spielen könnten. Wir zielen dabei Mittelklasse Hardware an. |

| | |
|---|---|
| Betrieb auf Windows Desktop Betriebssystem | Die Teammitglieder besitzen alle Windowsgeräte, außerdem nutzen die meisten Spieler Windows. Daher ist es für uns einfacher, für Windows zu entwickeln. Eine Mac-Unterstützung wäre wünschenswert, wir besitzen die Hardware dafür aber nicht. |
| Implementierung mit der Unrealengine (UE) | Die Entwickler haben alle Erfahrung im Umgang mit dem Visual Scripting der UE, den sog. Blueprints (BP). Kaum welche mit der Unity Engine. Außerdem ist die UE die von den Betreuern bevorzugte Engine. |

## 2.2 Organisatorisch

| **Randbedingung** | **Erläuterung, Hintergrund** |
|---|---|
| Entwickler/Gestalter Team | Studenten der Hochschule Augsburg:<br>Kimberly Kneissl, Lukas Uebel, Jonas Lang, Verena Hanner, Nina Hirsekorn, Laura Käppner, Florian Geiger, Leonhard Sandler, Melody Grace Wall |
| Recherche Team | Die Studenten und Professorin des Lehrstuhl Archäologie der Uni Augsburg: Prof. Natascha Sojc, Michael Schaper, Marie Arnold, Karlotta Braun, Roman Walch, Alina Rodat, Julian Leitner |
| Zeitplan | Entwurf und Design des Spiels bis zur 3. Oktoberwoche, ab dann Implementierung der Features und Beginn der Asset-Konzeption. Produktion der Assets ab Anfang November. Assetproduktion und Programmierung sollte mit Mitte Januar abgeschlossen sein. Ab Anfang Januar bis Mitte Januar Erstellung des Levels. Ab Mitte Januar dann Polishing und Fertigstellung bis Ende Januar. |
| Vorgehensmodell Programmierung | Entwicklung iterativ und inkrementell. Einen Ansatz zur Umsetzung eines Features überlegen, diesen im Team besprechen, dann implementieren und dann testen. Darauffolgend wird die Implementierung verbessert, dann wieder getestet und verbessert bis man mit dem Feature zufrieden ist. Bietet ein hohes Risiko, da ein Ansatz auch mal nicht richtig funktionieren kann und der Code später vielleicht nochmal neu geschrieben werden muss, da man auf eine bessere Lösung kommt. So kommen wir aber mit den Features sehr schnell voran und lernen schnell, was für den Spieler funktioniert und was nicht. |
| Entwicklungswerkzeuge | Stehen den einzelnen Teammitgliedern frei selbst zu wählen. Lediglich alles was mit der UE zu tun hat wird auch mit dem Standard Editor der UE entwickelt. |
| Versionsverwaltung | Git bei Gitlab mit Large File Storage, auch wenn es hin und wieder mal Probleme macht, denn es kennen und |

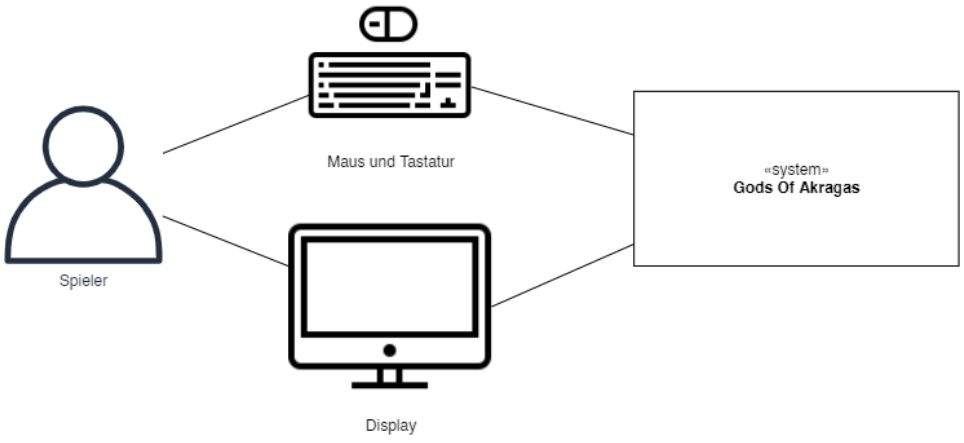
| | |
|---|---|
| | nutzen auch soweit alle Teammitglieder hin und wieder mal. |
| Veröffentlichung | Das Spiel soll später über itch.io zum Download angeboten werden, sodass auch externe Personen Spielen können. Das Spiel wird im unfertigen Zustand als Prototyp veröffentlicht. |

## 2.3 Konventionen

| **Konvention** | **Erläuterung, Hintergrund** |
|---|---|
| UE Benennungskonventionen | Die Benennung von BPs, Animationen und weiteren Dingen nach UE Konventionen. So dass auf einem Blick klar ist, um was es sich dabei handelt und wofür es verwendet werden kann |
| UE Visual Scripting Konventionen | Innerhalb der BPs, wie ordnet man die Nodes an, wie verlaufen die Verbindungen der Nodes. Möglichst immer gerade, aber sie dürfen sich nie groß überschneiden. Sorgt dafür, dass ein BP verständlich und übersichtlich bleibt. |
| Sprache (Deutsch vs. Englisch) | Innerhalb von BPs oder anderen Bausteinen in der UE sollte Englisch verwendet werden. Die Sprache zur Benennung von Assets steht dem Ersteller frei. |

# 3. Kontextabgrenzung

## 3.1 Fachlicher Kontext




**Spieler (Benutzer)**

Die Person welche das Spiel spielen will. Der Spieler muss natürlich sehen, was er tut und muss das Spiel auch irgendwie steuern können.

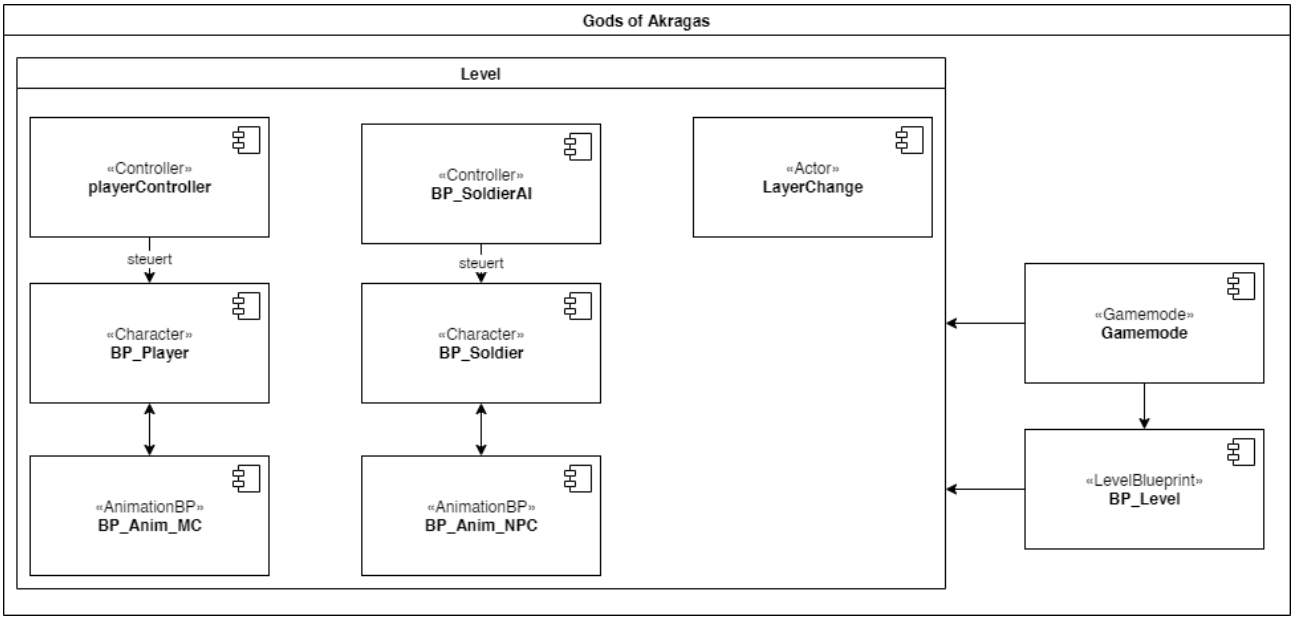
**Maus und Tastatur (Hardware)**

Der Spieler muss das Spiel steuern können, daher wird ein Eingabegerät benötigt. Die Wahl fiel hierbei auf Maus und Tastatur als Eingabegeräte, da jeder Windows-Nutzer eine Tastatur und Maus (Touchpad funktioniert auch) zum Bedienen des PCs besitzt. Und auch, da die Tastenbelegung bei Tastatur und Maus deutlich intuitiver wählbar ist als, bei einem Controller.

**Display (Hardware)**

Der Spieler muss auch sehen, was beim Spiel vor sich geht, und dafür benötigt er ein Display. So ergibt sich, dass für die Anzeige Assets benötigt werden, welche auch erstellt werden müssen. Der Spieler sieht hierbei in klassischer Sidescroller-Manier ein „zweidimensionales“ Level, welches von links nach rechts geht.

## 3.2 Technischer Kontext




**Gods of Akragas (Das Spiel an sich):**
Das Spiel an sich mitsamt der UE und ihrer Systeme.

**Gamemode (Fremdsystem):**

Regelt die grundlegenden Regeln und Einstellungen des Spiels. Wie viele Spieler gibt es, wie können sie beitreten usw. Der Gamemode regelt aber auch die grundlegende Logik des Spiels, dass heißt er initialisiert, spawned und überwacht die Actors. Wir passen hier momentan lediglich die Klasse des Pawns an und belassen den Rest beim Standard.

**BP_Level (Das LevelBP):**

Das Blueprint des Levels, darüber steuert man Levelabhängige Logik des Spiels. Bei uns übergeben wir benötigte Referenzen.

**Level (Das aktuelle Level im Spiel):**

Hier werden die Assets und Actors platziert. Es ist die Welt, durch welche sich das Spieler bewegen kann.

**playerController (Fremdsystem):**

Steuert und kontrolliert den Player Pawn. Wir ändern hieran momentan nichts. Sollten zukünftig aber etwas von der Funktionalität des Pawns hier herein verlagern.

**BP_Player (Playerpawn/Character):**

Unser eigener Playerpawn.In diesem wird der Input von Maus und Tastatur verarbeitet, und implementiert, was der Spieler alles können soll. In unserem Fall also das Movementsystem und ein Teil des Kampfsystems.

**BP_Anim_MC (AnimationBP):**

Ein BP, welches dazu dient die Animationen des Meshes des Playerpawns zu steuern. In diesen kommt also nur Logik, welche Animationen betrifft, aber auch etwas geskriptetes Movement, welches während oder direkt nach einer Animation stattfinden soll.

**BP_SoldierAI (Pawncontroller):**

Wie ein Playerpawn einen Controller benötigt, der diesen steuert, so brauchen auch alle anderen Pawns einen Controller. Dieser steuert den Pawn Soldier, welcher der Gegner im Kampfsystem ist.

**BP_Soldier (Character):**

Der Gegner des Spielers. In diesem BP wird implementiert, was der Gegner alles können soll, also auch ein Teil des Kampfsystems.

**BP_Anim_NPC (AnimationBP):**

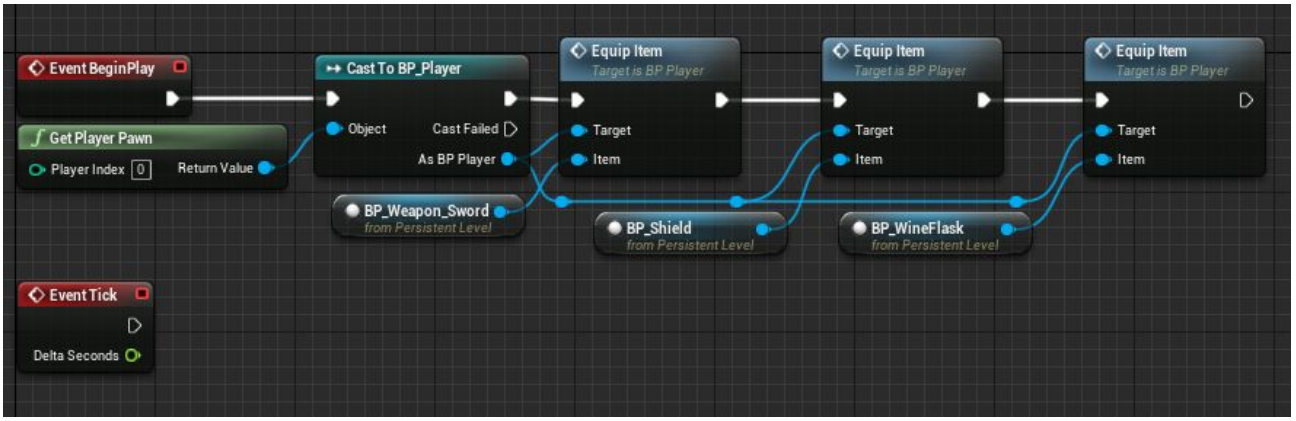
Steuert die Animationen des Meshes des BP_Soldier. Soll aber allgemein für menschliche NPCs als AnimationBP nutzbar sein.

**LayerChange (Actor):**

Die Implementierung des LayerChanges, welcher dazu dient, die 2-Dimensionalität des Levels etwas aufzubrechen. Ein LayerChange besteht aus drei Actors: dem BP_LayerChangeStart und zweimal dem BP_LayerChangeEnd (siehe späteres Kapitel).

# 4. Bausteine

## 4.1 BP_Level




Hier werden nur Referenzen übergeben, sodass der Playerpawn weiß, welcher Actor als seine Waffe, Schild gedacht und Weinflasche gedacht sind.

## 4.2 BP_Player

Unser eigener Playerpawn, Verarbeitung von Input und Implementierung von dem,was der Spieler können soll.

### Komponenten

| Komponentenname | Für was ist sie da |
|---|---|
| CapsuleComponent | Die umschließende Kapsel, zuständig für Kollisionen. |
| ArrowComponent | Zeigt den vorwärtsgehenden Vektor, genannt ForwardVector, des BP. |
| Mesh | Das Mesh des BP, wie dieser dargestellt werden soll. Hält auch die Referenz auf das AnimationBP. |
| SpringArm | An diesem Arm sitzt die Kamera. Er soll die Bewegung der Kamera etwas flüssiger machen. |
| Camera | Die Kamera, durch welche der Spieler das Level wahrnimmt. |
| ClimbDetection | Ein Sphere Collider, welcher erkennt, ob sich ein Objekt in der Nähe des Spielers befindet, an dessen Kante er sich festhalten können soll, oder an welchem er hochklettern können soll. |
| BP_CameraControlle r | Ein Controller, welcher die Bewegungen der Kamera steuert. |
| CharacterMovement | Steuert das Movement des Characters |

### Variablen

| Variablenname | Für was ist sie da |
|---|---|
| JumpChache: Float | Gibt an, wie weit der Boden maximal weg sein darf, damit immer noch ein Sprung ausgeführt wird, auch wenn der Pawn nicht den Boden berührt. |

| | |
|---|---|
| canHang: Bool | Gibt an, ob der Pawn sich an einer Kante festhängen kann. |
| isHanging: Bool | Hängt der Spieler gerade an einer Kante? |
| isWallInFront: Bool | Befindet sich vor dem Spieler eine Wand? |
| canChangeLayer: Bool | Kann der Spieler gerade das Layer wechseln? |
| isClimbable: Bool | Ist das Objekt vor dem Spieler hochkletterbar? |
| isClimbing: Bool | Klettert der Spieler gerade? |
| EdgePosition: Vector | Die Position der Kante, an welcher sich der Spieler festhalten können soll |
| WallNormal: Vector | Die Normal der Wand, an welcher sich die Kante befindet. |
| WallPos: Vector | Die Position der Wand, an welcher sich die Kante befindet. |
| ObjectsInReach: Int | Wie viele Objekte befinden sich gerade im Radius des Spielers, an dessen Kanten er sich potenziell festhalten könnte. |
| Health: Float | Das aktuelle Leben des Spielers |
| Damage: Float | Wie viel Schaden macht ein Angriff des Spielers |
| Weapon: BP_Weapon | Die Waffe des Spielers |
| isAttacking: Bool | Greift der Spieler gerade an? |
| isInvincible: Bool | Is der Spieler gerade unbesiegbar? |
| Shield: BP_Shield | Das Schild des Spielers. |
| LayerChangeSplines: Spline[] | Die Splines des LayerChanges, welcher der Spieler nutzen könnte. |
| LayerChangeSplineUsed: Int | Welchen Spline nutzt der Spieler gerade. |
| IsFighting: Bool | Hat der Spieler seine Waffe gezogen? |
| FollowPath: Bool | Folgt der Spieler gerade einem Spline? |
| IsBlocking: Bool | Block der Spieler gerade? |
| RespawnTransform: Tranform | Der Transform an welchem der Spieler respawnen soll. |
| IsDead: Bool | Ist der Spieler gerade tot? |
| WineFlask: BP_WineFlask | Die WeinFlask des Spielers. |
| MaxHealth: Float | Das maximale Leben des Spielers. |

## Events

**Alle Input Events werden Automatisch durch das drücken der jeweiligen Taste aufgerufen.**

| Eventname | Was macht es? |
|---|---|
| Event Tick | Je nach Zustand greift es nach einer Kante, setzt den Spieler auf die Position eines Splines und rotiert den Kopf in Richtung Ende des Splines, oder passt die Rotation des Spielers an die der Wand an, an welcher er gerade hängt. |
| InputAction Jump | Hängt der Spieler an einer Kante, klettert er diese hoch. Ansonsten schaut es, ob der Boden nah genug ist und springt dann. Je länger man die Taste hält, desto höher springt man (Sprunghöhe ist begrenzt) |
| InputAction Crouch | Wenn der Spieler sich ducken kann, dann duckt er sich solange die Taste gedrückt wird. |
| Grab | Greift nach der Kante, ist der Anfang davon, dass sich der Pawn an einer Kante festhält. |
| EndGrab | Beendet das an einer Kante festhalten. |
| ResetGrab | Setzt das Do Once am Anfang von Grab zurück. |
| InputAction Interact | Hier passiert im Moment nichts. |
| InputAction MoveInward | Hängt der Pawn an einer Kante und die Taste für “nach Oben” wird gedrückt, zieht er sich die Kante hoch. Fang mit dem Klettern an, wenn der Spieler es kann. Wenn er gerade klettert, dann bewegt es den Pawn nach oben oder unten. Wenn die Ebene gewechselt werden kann, dann wechsel zur entsprechenden Ebene. |
| Event OnWalkingOffLedge | Wird aufgerufen, wenn der Spieler über einen Kante läuft. Drückt der Spieler dabei die Taste für “nach Unten klettern”, dann fange damit an, dich an der Kante festzuhalten, wenn es möglich ist. |
| On ComponentEndOverlap(ClimbDetection) | Ein Objekt hat den Radius des Spielers verlassen. Überprüft, ob der Spieler noch Klettern bzw. sich an einer Kante festhalten kann. |
| On ComponentBeginOverlap(ClimbDetection) | Ein Objekt hat den Radius des Spielers betreten. Überprüft, ob der Spieler Klettern bzw. sich an einer Kante festhalten kann. |
| ClimbLedge | Kletter die Kante hoch. |
| ResetClimbLedge | Setzt das Do Once am Anfang von ClimbLedge zurück. |
| InputAction Attack | Hat der Spieler seine Waffe gezogen, dann greife an, wenn du es kannst. Ansonsten ziehe die Waffe. |
| InputAction Roll | Führe eine Ausweichrolle aus, wenn der Spieler seine Waffe gerade gezogen und kein Schild ausgerüstet hat. |
| ResetRoll | Setzt das Do Once von InputAction Roll zurück. |
| InputAction Block | Blocke solange der Spieler die Taste gedrückt hält, gerade seine Waffe gezogen hat und nicht angreift. |

| | |
|---|---|
| InputAxis MoveForward | Der Input welcher den Pawn nach rechts oder links laufen lässt. Hängt der Pawn an einer Kante oder klettert er gerade, prüft es, ob der Pawn sich gerade von der Wand wegbewegen will. Ist das der Fall, dann beendet es das Klettern und Hängen, aber nur, wenn nicht noch die Taste fürs hinunterklettern gedrückt wird. Folgt der Pawn gerade einem Pfad, dann lässt es den Pawn den Pfad entlang laufen. |
| InputAction Sheat | Packt die Waffe weg, wenn er es gerade möglich ist. |
| Event OnLanded | Ist der Pawn gerade gelandet, dann prüfe, ob er Fallschaden bekommt und berechne diesen. |
| Any Key | Wir irgendeine Taste gedrückt und der Spieler ist seit 2 Sekunden tot, dann respawne. |
| AllowRespawn | Setzt das Do Once am Anfang von Any Key zurück, so das ein Respawn möglich wird. |
| Dead | Wird aufgerufen, nachdem der Pawn „gestorben“ ist. Aktiviert den Respawn und Input nach 2 Sekunden wieder. |
| AnyDamage | Bekommt der Pawn irgendeinen Schaden, wird geprüft ob er ihn abgeblockt hat, ansonsten berechnet er das verbleibende Leben und startet das angemessene Feedback. |
| InputAction Heal | Wenn das Leben des Spielers weniger als das maximale Leben ist und er eine Weinflasche ausgerüstet hat, die noch Wein enthält, dann benutze diese und heile dich. |

## Funktionen

| **Funktionsname** | **Was macht es?** |
|---|---|
| ConstructionScript() | Wird beim Erstellen des Pawn aufgerufen. Setzt, wie lange die Sprungtaste maximal gehalten werden kann und übergibt eine Referenz vom SpringArm und der Kamera an den CameraController. Außerdem setzt es momentan auch den RespawnTransfrom auf den aktuellen Transform des Pawns. |
| IsNearGround() | Schickt einen Linecast direkt unter den Pawn in Richtung Boden und prüft damit, ob der Boden JumpCache weit weg ist und ob der Pawn den Boden berührt, und gibt dieses als Boolean zurück. Außerdem berechnet es auch noch die Zeit, die es braucht, bis der Pawn wieder den Boden berührt und gibt diese auch zurück. |
| IsEdge() | Prüft mit zwei LineTraces, wo sich eine Kante befindet und berechnet deren Position. |
| CanClimb() | Prüft, ob der Pawn klettern kann. Dafür müssen folgende Bedingungen erfüllt sein:<br>Es muss eine Wand in Reichweite sein, das Objekt muss kletterbar sein, der Pawn darf sich nicht in der Luft befinden, der Pawn muss entweder hängen oder sich eine Wand vor ihm befinden und er darf nicht bereits klettern. |

| | |
|---|---|
| StartClimb() | Wird aufgerufen, sobald das Klettern gestartet werden soll. Prüft erst, ob der Pawn überhaupt klettern kann, indem es CanClimb aufruft. Falls ja, startet es das Klettern an einer Wand. |
| EndClimb() | Beendet das Klettern. |
| IsWallNear() | Gibt zurück, ob sich die Wand nach genug um dessen Kante greifen zu können. |
| LineTraceForward() | Ein simpler Linetrace, welcher in Laufrichtung des Pawns prüft, ob sich vor diesem eine Wand befindet, setzt dementsprechend die passenden Variablen. IsWallInFront, WallPos, WallNormal |
| CalculateEdgePosition(Vector) | Berechnet die Position der Wand aus der Wall und des Inputvektors. Übernimmt einfach die X Koordinate der WandPos, die Y-Koordinate und Z-Koordinate des Inputs, passt diese leicht an und gibt sie dann zurück. |
| CalculateLayerLocation() | Gibt die Position des nächsten Punktes zum Spieler des verfolgten Splines zurück. |
| CheckForEdgeBelow() | Prüft, ob sich unter dem Pawn eine Kante befindet.<br>Nutzt zwei Line Traces, die leicht unter dem Pawn ausgeführt werden, der eine horizontal der andere vertikal. Treffen diese auf ein Objekt, an dem der Pawn sich an einer Kante festhalten soll, updated es die WallPos und EdgePos dementsprechend. |
| StartClimbingDown() | Startet das Kletten nach unten. |
| CanGrab() | Prüft, ob der Pawn sich an eine Kante hängen kann, und gibt das als Boolean zurück.<br>Dazu prüft es, ob es überhaupt ein Objekt in der Nähe gibt, wo er sich an eine Kante hängen könnte. Falls ja, prüft es wo sich die Kante befindet und updated EdgePosition. Damit berechnet es dann den Abstand zwischen Kante und Pawn, und wenn dieser kleiner als 50 ist und der Pawn entweder in der Luft oder gerade am Klettern ist, gibt es dies als Boolean zurück. Befindet sich kein Objekt zum dranhängen in der Nähe, passiert genau das gleiche, bloß, dass die EdgePosition nicht geupdatet wird. |
| EquipItem(Object) | Prüft, was für ein Objekt übergeben wurde, attached es zum passenden Socket des Meshs und setzt die dazu passende Variable. |
| TakeDamage(Float) | Spawned ein Partikelsystem für Feedback, berechnet und setzt das übrige Leben.Fällt dieses auf oder unter Null, dann ruft es die Funktion Die auf. |
| Die() | Aktiviert das Ragdollsystem des Meshs und der Items. Deaktiviert es den Input und das Movement, setzt isDead auf True und ruft das Event Dead auf. |
| CanFight() | Prüft, ob der Spieler gerade angreifen kann. Dafür darf er weder fallen, noch gerade unsterblich sein, noch an einer Kante hängen, noch geduckt sein, noch die Waffe nicht gezogen haben. |
| DrawItems() | Setzt isFighting auf True und Attached die Items zu den passenden Sockets des Meshs. |

| | |
|---|---|
| PutItemsAway() | Setzt isFighting auf False und Attached die Items zu den passenden Sockets des Meshs. |
| SheatItems() | Aktiviert den Wegpackprozess der Items, indem es SheatSword beim BP_Anim_MC aufruft. |
| CalculateHeadRot() | Berechnet und gibt die Rotation zurück, welche der Kopf haben muss, um an das Ende des aktuell verfolgten Splines zu schauen. |
| CombatFeedback() | Simples Feedback für den Kampf, simuliert Rückschlag und sagt dem AnimBP, dass die passende Animation abgespielt werden soll. |
| Respawn() | Deaktiviert das Ragdollsystem und ruft ResetPlayer auf, außerdem setzt es den Pawn an den RespawnTransform und attached die Items an den passenden Socket des Meshs. |
| ResetPlayer() | Setzt den Spieler auf den Ausgangszustand zurück. |
| AttachWineFlask() | Attached die WineFlask an den passenden Hand Socket des Meshs |
| DetachWineFlask() | Attached die WineFlask an den passenden Spine Socket des Meshs |

### Makros

kleine Helfer

| **Makroname** | **Was macht es?** |
|---|---|
| ValueInDirection(Float) | Gibt einen Wert passend zur Laufrichtung zurück. |
| GetSplineTangent() | Gibt die Tangente an der Position des aktuell verfolgten Splines zurück, welche der aktuellen Position des Pawns am nächsten ist. |
| Turn180() | Dreht den Pawn um 180 Grad |
| BlockCheck(Object) | Gibt zurück, ob der Pawn gerade den Schaden des übergebenen Objekts blockt. Vergleicht dazu die Blickrichtung. |

## 4.3 BP_CameraController

Kontrolliert die Bewegung der Kamera und des SpringArms.

### Variablen

| **Variablenname** | **Für was ist sie da?** |
|---|---|
| isCamRotated: Bool | Wurde die Kamera bereits rotiert? |
| isCamLocked: Bool | Ist die Bewegung der Kamera gesperrt? |

| | |
|---|---|
| latestCameraClimbingShake: CamerShake | Der letzte ausgeführte CameraShake während dem Klettern. |
| latestCameraHitShake: CamerShake | Der letzte ausgeführte CameraShake, als der Parent getroffen wurde. |
| latestCameraHurtShake: CamerShake | Der letzte ausgeführte CameraShake, als der Parent verletzt wurde. |

### Funktionen

| Funktionsname | Was macht es? |
|---|---|
| RotateCamera() | Rotiert die Kamera, wenn sie nicht gesperrt oder bereits rotiert wurde. |
| RotateCameraBack() | Rotiert die Kamera zurück zur Ausgangsposition. |
| StartCameraShakeClimbing() | Startet den CameraShake fürs klettern und speichert sich die Referenz auf diesen. |
| StopCameraShakeClimbing() | Stoppt den CamerShake fürs klettern. |
| ShakeCamera() | Spielt einen kurzen CameraShake ab und speichert sich die Referenz auf diesen als latestCameraHurtShake. |
| ShakeCameraWhenHitting() | Spielt einen kurzen CameraShake ab und speichert sich die Referenz auf diesen als latestCameraHitShake. |
| JumpOvershoot() | Macht im Moment noch nichts. |

## 4.4 BI_Anim

Ein Blueprint Interface, welches die Kommunikation zwischen AnimBP und dazugehörigen Pawn erleichtern.

### Funktionen

| Funktionsname | Was macht es? |
|---|---|
| IsHanging(Bool) | Soll dem AnimBP mitteilen, dass der Pawn gerade an einer Kante hängt. |
| ClimbLedge() | Soll dem AnimBP mitteilen, dass der Pawn gerade eine Kante hochklettert. |
| IsClimbing(Bool) | Soll dem AnimBP mitteilen, dass der Pawn gerade klettert. |
| Attack() | Soll dem AnimBP mitteilen, dass der Pawn gerade angreift. |
| Roll() | Soll dem AnimBP mitteilen, dass der Pawn gerade eine Ausweichrolle ausführt. |
| IsBlocking(Bool) | Soll dem AnimBP mitteilen, dass der Pawn gerade blockt. |
| IsFighting(Bool) | Soll dem AnimBP mitteilen, dass der Pawn gerade seine Waffe gezogen hat. |

| | |
|---|---|
| SheatSword() | Soll dem AnimBP mitteilen, dass der Pawn gerade seine Waffe zieht bzw. wegsteckt. |
| SetHeadRot(Rotator) | Soll dem AnimBP mitteilen, um wie viel Grad der Kopf rotiert werden soll. |
| AbortMontages() | Soll dem AnimBP mitteilen, dass gerade alle Montagen gestoppt werden sollen. |
| GotHit() | Soll dem AnimBP mitteilen, dass der Pawn gerade getroffen wurde. |
| Reset() | Soll dem AnimBP mitteilen, dass es sich auf Ausgangszustand setzen soll. |
| Heal() | Soll dem AnimBP mitteilen, dass der Pawn sich gerade heilt. |

## 4.5 BP_Anim_MC

Ein BP, welcher dazu dient, die Animationen des Meshes des Playerpawns zu steuern. In diesen kommt also nur Logik, welche Animationen betrifft, aber auch etwas geskriptetes Movement, welches während oder direkt nach einer Animation stattfinden soll. Es implementiert BI_Anim.

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| velocity: float | Die horizontale Geschwindigkeit des zugehörigen Pawns. |
| climbVel: float | Die vertikale Geschwindigkeit des zugehörigen Pawns. |
| isCrouching: Bool | Duckt sich der zugehörige Pawn gerade? |
| isFalling: Bool | Fällt der zugehörige Pawn gerade? |
| hanging: Bool | Hängt der zugehörige Pawn gerade an einer Kante? |
| isClimb: Bool | Klettert der zugehörige Pawn gerade? |
| isRolling: Bool | Führt der zugehörige Pawn gerade? |
| isBlock: Bool | Blockt der zugehörige Pawn gerade? |
| montagesToPlay: Int | Wie viele Montagen müssen noch abgespielt werden? |
| IsFight: Bool | Hat der Pawn seine Waffe gezogen? |
| HeadRot: Rotator | Um wie viel Grad muss der Pawn seinen Kopf rotieren? |

### Events

Die AnimNotify Events sind alles Events, welcher durch die UE an einem bestimmten Frame einer Animation aufgerufen werden.

| Eventname | Was macht es? |
|---|---|
| Blueprint Update Animation | Holt sich den Pawn und updatet Werte wie velocity, climbVel, isCrouching und isFalling. |

| | |
|---|---|
| IsHanging, IsClimbing, IsBlocking, IsFighting, SetHeadRot | Updated die dazugehörige Variable. |
| AnimNotify_climbedUp | Setzt den Pawn oben auf die Kante und ruft beim Pawn “End Grab” auf. |
| Climb Ledge | Spielt die passende Montage ab. |
| Attack | Spielt je nachdem wie häufig es aufgerufen worden ist, eine Angriffsmontage ab. |
| AnimNotify_attacked | Aktiviert das Movement wieder und setzt isAttacking beim Pawn auf False. |
| Roll | Spielt die passende Montage ab. |
| AnimNotify_rolled | Ruft beim Pawn RestRoll auf, setzt isInvincible auf False und ruft auch AnimNotify_attacked auf. |
| ResetAttack | Setzt das MultiGate am Anfang von Attack zurück. |
| SheatSword | Spielt je nachdem wie häufig es bereits aufgerufen wurde, die passende Animation ab und steck dann die Waffen weg oder zieht sie. |
| AnimNotify_Sheated | Ruft beim Pawn PutItemsAway und ResetSheat auf. |
| GotHit | Spielt die passende Montage ab und ruft AnimNotify_Attacked. |
| ResetSheat | Setzt das MultiGate am Anfang von SheatSword zurück. |
| Reset | Setzt das BP_Anim_MC auf seinen Ausgangzustand zurück. |
| Heal | Spielt die passende Montage ab. |
| AnimNotify_drank | Ruft beim Pawn DetachWineFlask und Enable Input auf. |

## Funktionen

| Funktionsname | Was macht es? |
|---|---|
| CalculateVelHor(Float, Float) | Berechnet die horizontale Geschwindigkeit des Pawns und gibt diese zurück. |

## Makros

kleine Helfer

| Makroname | Was macht es? |
|---|---|
| WaitUntilMontageIsFinished() | Schaut, wie viele Montagen noch abgespielt werden müssen, und wartet, bis diese abgespielt worden sind. |

## 4.6 BP_Anim_NPC

Ein BP, welches dazu dient die Animationen des Meshes des Playerpawns zu steuern. In diesen kommt also nur Logik, welche Animationen betrifft, aber auch etwas geskriptetes Movement, welches während oder direkt nach einer Animation stattfinden soll. Es implementiert BI_Anim.

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| velocity: float | Die horizontale Geschwindigkeit des zugehörigen Pawns. |
| isBlock: Bool | Blockt der zugehörige Pawn gerade? |
| montagesToPlay: Int | Wie viele Montagen müssen noch abgespielt werden? |

### Events

| Eventname | Was macht es? |
|---|---|
| Blueprint Update Animation | Holt sich den Pawn und updatet Werte wie velocity. |
| IsBlocking | Updated die dazugehörige Variable. |
| Attack | Spielt je nachdem wie häufig es aufgerufen worden ist, eine Angriffsmontage ab. |
| AnimNotify_attacked | Aktiviert das Movement wieder und setzt isAttacking beim Pawn auf False. |
| ResetAttack | Setzt das MultiGate am Anfang von Attack zurück. |
| GotHit | Spielt die passende Montage ab und ruft AnimNotify_Attacked. |

### Funktionen

| Funktionsname | Was macht es? |
|---|---|
| CalculateVelHor(Float, Float) | Berechnet die horizontale Geschwindigkeit des Pawns und gibt diese zurück. |

### Makros

kleine Helfer

| Makroname | Was macht es? |
|---|---|
| WaitUntilMontageIsFinished() | Schaut, wie viele Montagen noch abgespielt werden müssen, und wartet, bis diese abgespielt worden sind. |

## 4.7 BP_DealDamageNotify

Testet, nachdem dieser Notify aufgerufen wurde, mit einem Sphere Trace nach vorne, ob er einen anderen Pawn des Typs BP_Player oder BP_Enemy getroffen hat. Falls ja, dann hole den Damagewert deines Parents und rufe mit diesem bei dem getroffenen Pawn ApplyDamage auf und reporte das DamageEvent, damit es auch von einer AI wahrgenommen werden kann.

## 4.8 BP_Enemy

Soll die grundsätzlichen Funktionen eines Gegners implementieren, sodass dann jeder weitere Gegner hiervon abgeleitet werden kann.

### Komponenten

| Komponentenname | Für was ist sie da? |
|---|---|
| CapsuleComponent | Die umschließende Kapsel, zuständig für Kollisionen. |
| ArrowComponent | Zeigt den vorwärtsgehenden Vektor, genannt ForwardVector, des BP. |
| Mesh | Das Mesh des BP, wie dieser dargestellt werden soll. Hält auch die Referenz auf das AnimationBP. |
| CharacterMovement | Steuert das Movement des Characters. |

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| health: Float | Das aktuelle Leben des Pawns. |
| damage: Float | Wie viel Schaden der Pawn macht. |
| isAttacking: Bool | Greift der Pawn gerade an? |
| isBlocking: Bool | Blockt der Pawn gerade? |

### Events

| Eventname | Was macht es? |
|---|---|
| AnyDamge | Bekommt der Pawn irgendeinen Schaden, wird geprüft ob er ihn abgeblockt hat, ansonsten berechnet er das verbleibende Leben und startet das angemessene Feedback. |

### Funktionen

| Funktionsname | Was macht es? |
|---|---|
| ConstructionScript() | Macht hier nichts. |
| Attack() | Setzt ist isAttacking auf True. |

| | |
|---|---|
| Parry() | Macht hier nichts. |
| TakeDamage(Float) | Spawned ein Partikelsystem für Feedback, berechnet und setzt das übrige Leben. Fällt dieses auf oder unter Null, dann ruft es die Funktion Die auf. |
| Die() | Zerstört den Pawn. |
| StopParry() | Macht hier nichts. |
| CombatFeedback() | Simuliert Rückstoß. |

### Makros

kleine Helfer

| **Makroname** | **Was macht es?** |
|---|---|
| BlockCheck(Object) | Gibt zurück, ob der Pawn gerade den Schaden des Übergebenen Objekts blockt. Vergleicht dazu die Blickrichtung. |

## 4.9 BP_Soldier

Leitet sich von BP_Enemy ab. Ein simpler KI-gesteuerter Gegner.

### Variablen

| **Variablenname** | **Für was ist sie da?** |
|---|---|
| health: Float | Das aktuelle Leben des Pawns. |
| damage: Float | Wie viel Schaden der Pawn macht. |
| isAttacking: Bool | Greift der Pawn gerade an? |
| isBlocking: Bool | Blockt der Pawn gerade? |
| weapon: BP_Weapon | Die Waffe des Pawns. |
| shield: BP_Shield | Das Schild des Pawns. |

### Funktionen

| **Funktionsname** | **Was macht es?** |
|---|---|
| ConstructionScript() | Attached die Waffe und das Schild an die passenden Sockets des Meshs. |
| Attack() | Setzt ist isAttacking auf True und ruft Attack beim BP_Anim_NPC auf. |
| Parry() | Prüft ob eine Random Bool True ist. Dann setzt es isBlocking auf True und ruft IsBlocking beim BP_Anim_NPC auf. |
| Die() | Aktiviert das Ragdollsystem des Meshs und der Items. Außerdem deaktiviert es die Collision der CapsuleComponent. |

| | |
|---|---|
| StopParry() | Setzt isBlocking auf False und ruft IsBlocking beim BP_Anim_NPC auf. |
| CombatFeedback() | Simuliert Rückstoß und ruft beim BP_Anim_NPC GotHit auf. |
| DetachItems() | Detached die Items vom Mesh. |
| AbortAttack() | Setzt isAttacking auf false und ruft AbortMontages beim BP_Anim_NPC auf. |

## 4.10 BP_SoldierAI

Ein AIController, welcher dazu gedacht ist, BP_Soldier zu steuern. Trifft seine Entscheidungen aufgrund des Behaviourtrees(BT) BT_Soldier. Leitet sich von AIController ab.

### Events

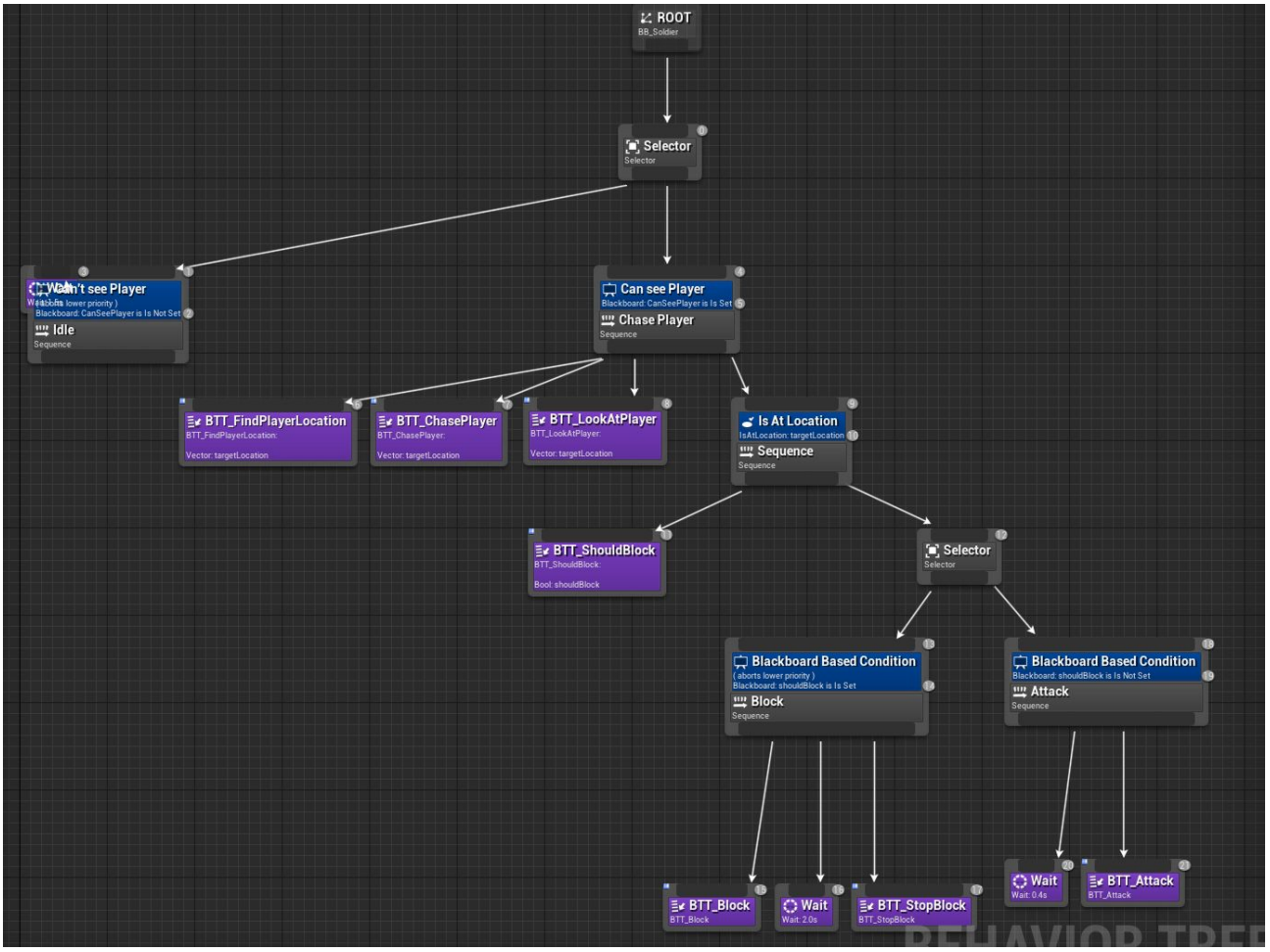
| Eventname | Was macht es? |
|---|---|
| BeginPlay | Startet den BT. |
| OnTargetPerception | Prüft, ob der über das Perceptionsystem von UE wahrgenommene Pawn vom Typ BP_Player ist, wenn ja, dann setzt es beim BlackBoard(BB) des BT CanSeePlayer auf True, wenn der andere Pawn erfolgreich wahrgenommen wurde. |

## 4.11 BT_Soldier

Ein BT, welcher dazu genutzt wird, um mit BP_SoldierAI den BP_Soldier zu steuern.

**Variablen des BB**

| Variablenname | Für was ist sie da? |
|---|---|
| targetLocation: Vector | Wohin soll sich der Pawn bewegen? |
| CanSeePlayer: Bool | Kann er gerade den Spieler sehen? |
| shouldBlock: Bool | Sollte ich gerade blocken? |
| SelfActor: Object | Eine Referenz auf den Pawn, der durch das BP_SoldierAI des BT gesteuert wird. |




### Tasks des BT

| Taskname | Was macht sie? |
|---|---|
| BTT_Attack | Ruft beim Pawn Attack auf. |
| BTT_Block | Ruft beim Pawn nach 0,2 Sekunden Parry auf. |
| BTT_ChasePlayer | Bewegt den Pawn zur targetLocation. |
| BTT_FindPlayerLocation | Holt sich die Position des Spielers im NavMesh, sollte dieser innerhalb des NavMesh sein. Tauscht von diesem Punkt die Y und Z Koordinate durch die des Pawns aus, rechnet bei X einen Abstand von 150 mit ein und setzt diese Koordinate dann als targetLocation. |
| BTT_FindRandomLocation | Speichert sich eine Random Position als targetLocation ab. |
| BTT_LookAtPlayer | Rotiert den Pawn so, dass er den Spieler ansieht. |
| BTT_ShouldBlock | Setzt shouldBlock auf True, wenn der Spieler gerade angreift. |
| BTT_StopBlock | Ruft beim Pawn StopParry auf. |

## 4.12 BP_Shield

Ein simples Blueprint macht aus einem Mesh einen Actor, sodass es etwas besser handzuhaben ist. Besitzt also auch nur eine StaticMesh Component.

## 4.13 BP_Weapon

Ein simples Blueprint macht aus einem Mesh einen Actor, sodass es etwas besser handzuhaben ist, fügt ihm aber auch eine damage-Variable hinzu.

### Komponenten

| Komponentenname | Für was ist sie da? |
|---|---|
| StaticMesh | Das Mesh der Waffe |

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| damage: Float | Wie viel Schaden macht die Waffe? |

## 4.14 BP_Weapon_Sword

Leitet sich von BP_Weapon ab. Besitzt lediglich einen anderen damage-Wert und ein anderes Mesh.

## 4.15 BP_WineFlask

Eine Weinflasche, mit der sich der Spieler heilen kann.

### Komponenten

| Komponentenname | Für was ist sie da? |
|---|---|
| StaticMesh | Das Mesh des Actors. |

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| AmountOfWine: Float | Wie viel Wein ist in der Flasche? |
| MaxAmountOfWine: Float | Wie viel Wein passt maximal in die Flasche? |

### Funktionen

| Funktionsname | Was macht es? |
|---|---|
| Use(Object) | Heilt das übergebene Objekt um 20. Verbraucht dabei 20 Wein. |
| Refill(Float) | Füllt die Flasche um den übergebenen Wert wieder auf. |
| CanBeUsed() | Gibt zurück, ob die Flasche benutzt werden kann, also ob noch Wein da ist. |

## 4.16 BP_StatLight

Simples BP, dazu gedacht, dass man alle statischen Lichter hiervon ableitet, wie statische Fackeln und Lampen.

### Komponenten

| Komponentenname | Für was ist sie da? |
|---|---|
| StaticMesh | Wie sieht der Actor aus? |
| Fire | Eine Partikelsimulation die ein Feuer darstellen soll. |
| PointLight | Das Licht an sich. |

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| isActive | Ist die Lampe an? |

### Events

| Eventname | Was macht es? |
|---|---|
| Tick | Prüft, ob die Lampe an ist, und aktiviert diese dann. |

## 4.17 BP_DynLight

Simples BP, dazu gedacht, dass man alle dynamischen Lichter hiervon ableitet, wie Fackeln und Lampen. Leitet BP_StatLight ab.

### Komponenten

| Komponentenname | Für was ist sie da? |
|---|---|
| StaticMesh | Wie sieht der Actor aus? |
| Fire | Eine Partikelsimulation die ein Feuer darstellen soll. |
| PointLight | Das Licht an sich. |

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| isActive | Ist die Lampe an? |

### Events

| Eventname | Was macht es? |
|---|---|

| | |
|---|---|
| Tick | Prüft, ob die Lampe an ist, und aktiviert sie dann. Sorgt aber auch dafür, dass das Licht etwas flackert, indem es alle 0,08 Sekunden das Licht auf eine random Helligkeit zwischen 5000 und 10000 setzt. |

## 4.18 BP_CameraTrigger

Rotiert die Kamera um eine bestimmbare Gradzahl, während der Spieler sich im Trigger befindet.

## 4.19 BP_LayerChangeEnd

Der LayerChange dient dazu, die 2-Dimensionalität des Levels etwas aufzubrechen, und ermöglicht es den Spieler, um eine „Ebene“ nach vorne oder hinten zu wechseln.

### Komponenten

| **Komponentenname** | **Für was ist sie da?** |
|---|---|
| CollisionComponent | Der eigentliche Trigger des LayerChanges. |
| SpriteComponent | Wie soll dieses Objekt im Editor dargestellt werden? |

### Variablen

| **Variablenname** | **Für was ist sie da?** |
|---|---|
| SplineDir: Int | Der Index, unter welchem der Spline im Array Splines im BP_LayerChangeStart liegt. Gibt dadurch an, ob er nach innen oder außen geht. 0 = Er geht nach innen. 1 = Er geht nach außen. |
| SplitDir: Enum | Geht der LayerChange nach rechts oder nach links? |
| Spline: Spline | Der Spline, an welchem Ende dieser Trigger platziert ist. |

### Events

| **Eventname** | **Was macht es?** |
|---|---|
| ActorBeginOverlap | Prüft, ob es sich bei dem Actor um einen Spieler handelt, und ob dieser sich in den LayerChange reinbewegt oder raus. Geht er in den LayerChange, dann wird beim Spieler FollowPath auf True gesetzt, dem Spieler der Spline passend übergeben und SplineDir dem Spieler als genutzter Spline gesetzt.<br>Geht der Spieler aus dem LayerChange heraus, passiert nichts. |
| ActorEndOverlap | Prüft, ob es sich bei dem Actor um einen Spieler handelt, und ob dieser sich in den LayerChange reinbewegt oder raus. Geht er in den LayerChange, dann wird beim Spieler FollowPath auf False gesetzt, und die Referenz auf Splines beim Spieler entfernt.<br>Geht der Spieler aus dem LayerChange raus, passiert nichts, da er ja in Gegenrichtung den LayerChange verlässt. |

### Funktionen

| Funktionenname | Was macht es? |
|---|---|
| GoingInto(BP_Player) | Gibt True zurück, wenn der Spieler gerade in den LayerChange geht. Prüft dafür, ob der Spieler von links oder rechts den LayerChange betreten hat. |

## 4.20 BP_LayerChangeStart

Der LayerChange dient dazu, die 2-Dimensionalität des Levels etwas aufzubrechen, und ermöglicht es den Spieler, um eine „Ebene“ nach vorne oder hinten zu wechseln. Benötigt zwei BP_LayerChangeEnd Actors, um zu funktionieren.

### Komponenten

| Komponentname | Für was ist sie da? |
|---|---|
| CollisionComponent | Der eigentliche Trigger des LayerChanges. |
| SpriteComponent | Wie soll dieses Objekt im Editor dargestellt werden? |
| InwardSpline | Der Pfad, der den Spieler nach innen führen soll. |
| OutwardSpline | Der Pfad, der den Spieler nach außen führen soll. |

### Variablen

| Variablenname | Für was ist sie da? |
|---|---|
| TriggerEnd: BP_LayerChangeEnd[] | Die Enden des LayerChange. |
| onePathMode: Bool | Soll nur einem bestimmten Pfad gefolgt werden? |
| Splines: Spline[] | Die Splines des LayerChanges. |
| splineUsed: Int | Welcher Spline soll standardmäßig benutzt werden, bzw. welcher wurde zuletzt verwendet? |
| SplitDir: Enum | Geht der LayerChange nach rechts oder nach links? |

### Events

| Eventname | Was macht es? |
|---|---|
| BeginPlay | Übergib dem BP_LayerChangeEnd eine Referenz auf seinen passenden Spline, den Index des Spline im Array und die Richtung des LayerChanges. |
| ActorBeginOverlap | Prüft, ob es sich bei dem Actor um einen Spieler handelt, und ob der Spieler bereits einem Pfad folgt. Wenn nicht, dann übergibt es eine Referenz der Splines an den Spieler und setzt den benutzen Spline beim Spieler auf den Wert von splineUsed. Befindet sich der LayerChange nicht im onePathMode, dann wird beim Spieler auch CanChangeLayer auf True gesetzt. |

| | |
|---|---|
| ActorEndOverlap | Prüft, ob es sich bei dem Actor um einen Spieler handelt, und ob dieser sich in den LayerChange reinbewegt oder raus. Geht er in den LayerChange, dann wird beim Spieler CanChangeLayer auf False, FollowPath auf True, und SplineUsed beim LayerChange auf den SplineUsed vom Spieler gesetzt.<br>Geht der Spieler aus dem LayerChange raus, wird beim Spieler CanChangeLayer und FollowPath auf False gesetzt, die Referenz auf Splines beim Spieler entfernt und SplineUsed beim LayerChange auf den SplineUsed vom Spieler gesetzt. |

### Funktionen

| **Funktionenname** | **Was macht es?** |
|---|---|
| ConstructionScript() | Speichert sich die Splines im Array Splines ab. Dabei wird der Inward Spline immer unter dem Index 0 und der Outward Spline unter dem Index 1 abgelegt. |
| GoingInto(BP_Player) | Gibt True zurück, wenn der Spieler gerade in den LayerChange geht. Prüft dafür,ob der Spieler von links oder rechst den LayerChange betreten hat. |

# 5. Konzepte hinter den Features

Das Ziel dieses Abschnittes ist es, dem Leser verständlich zu erklären, wie diese Features funktionieren. Daher beschreibe ich die eher die Konzepte und gehe teilweise etwas weniger auf die direkte Implementierung ein, da man diese ja auch unter Bausteine nachschlagen kann.

## 5.1 Movementsystem

### Allgemein

Das Movementsystem besteht aus laufen, springen und aus spezielleren Movements wie klettern, sich an einer Kante festhalten und diese hochklettern zu können. Das komplette Movementsystem wurde innerhalb vom BP_Player und BP_AnimMC umgesetzt. Das grundlegende Movement ist hierbei UE Standard, aber über die Einstellungen der CharacterMovement Component etwas angepasst worden, und auch das Springen wurde etwas abgewandelt. Beim CharacterMovement haben wir die Sprunggeschwindigkeit auf 350 heruntergesetzt, damit die Sprunghöhe etwas realistischer ist. Wir haben auch die AirControl auf 5 gesetzt, sodass man den Pawn auch immer noch ganz gut in der Luft steuern kann. Und mit am wichtigsten, wir haben das Movement mit ConstrainToPlane auf die X Plane beschränkt.

### Springen

Springen wurde folgendermaßen angepasst: 1. Wenn man die Sprungtaste gedrückt hält, springt man etwas höher (dies musste beim CharacterMovement aktiviert werden). 2. Wurde eine Graceperiode eingebaut, sodass man beim Landen nicht frame perfect die Leertaste drücken muss, um wieder direkt zu springen. Man kann dies nun auch schon etwas vor der Landung tun und springt dann, sobald man den Boden berührt. Das funktioniert, indem jedes mal wenn die Sprungtaste gedrückt wird und der Spieler im

Moment nicht an einer Kante hängt, die Funktion IsNearGround aufgerufen wird. Diese Funktion prüft mit einem LineTrace, welcher direkt nach unten geht, ob sich in einer Entfernung (die nicht größer als JumpChache ist) ein Objekt unter dem Spieler befindet. Befindet sich ein Objekt unter dem Spieler, dann wird zum einen True zurückgegeben, zum anderen die Zeit berechnet, wie lange der Spieler noch fallen wird, und diese zurückgegeben. Wenn nun True zurückgegeben wurde, wird diese Zeit gewartet und dann ein Sprung mit einer erneuten Überprüfung, ob der Spieler auch wirklich auf dem Boden aufgekommen ist, gesprungen. Sollte False zurückgegeben werden, dann wird vorsichtshalber auch nochmal geprüft, ob der Spieler sich gerade auf dem Boden befindet, und falls ja, gesprungen.

## Sich an einer Kante festhalten

Hierfür wurde ein eigener Trace Channel mit dem Namen Edge angelegt, damit sich dann in den Kollisionseinstellungen festlegen lässt, wie ein Objekt auf Kollisionen mit diesem Trace Channel reagiert. So können wir bestimmen, an welchen Objekten sich der Spieler an den Kanten festhalten können soll und an welchen nicht.

Über die ClimbDetection wird zunächst geprüft, ob sich überhaupt ein Objekt in der Nähe befindet, an dem ich klettern oder mich an die Kante hängen können soll. Die ClimbDetection ist ein SphereCollisionComponent. Sobald sich ein Objekt in diese Sphere begibt, wird geprüft, ob es sich dabei um ein Objekt handelt, welches der Spieler hochklettern können soll. Diese werden im Folgenden Kletterobjekt genannt. Ist das der Fall, wird isClimbable auf True gesetzt. Egal, ob es sich um ein Kletterobjekt handelt oder nicht, wird CanHang auf True gesetzt und ObjectsInReach hochgezählt. Verlässt ein Objekt die Sphere, wird wieder überprüft, ob es sich um ein Kletterobjekt handelt, falls ja wird isClimable auf False gesetzt. Auch wie oben wird, egal ob es sich um ein Kletterobjekt handelt oder nicht, ObjectsInReach runtergezählt, sobald ObjectsInReach den Wert Null hat, CanHang auf False gesetzt.

Das Ganze nochmal etwas simpler: Befindet sich ein Kletterobjekt oder ein Kantenobjekt innerhalb der Sphere wird es vermerkt und in einer Variable abgespeichert. Da sich mehr als ein Objekt in die Sphere rein- und rausbewegen können zähle ich, wie viele Objekt sich momentan in der Sphere befinden. Sobald sich dann also keines mehr in der Sphere befindet, wird das auch mit Hilfe einer Variablen vermerkt. Es wird nochmal überprüft, ob es sich um ein Kletterobjekt handelt, da sich in diesem Fall in der Regel nur ein Objekt im Radius befindet. Ich weiß dann also direkt, dass ich das Objekt hochklettern könnte. (Dazu später mehr unter Klettern). Diese Überprüfung durch den Sphere soll Ressourcen sparen, denn so ein LineTrace, wie ich ihn gleich noch brauche, braucht viele Ressourcen. Wenn sich also keine Kantenobjekte oder Kletterobjekte in der Nähe befinden, brauche ich auch keinen LineTrace, um eine Kante zu suchen. Wie erkennt man nun genau, wo sich die Kante befindet?

Wenn ein Kantenobjekt in dieser Sphere ist, dann wird mit jedem Tick des Spiels geprüft, wo sich diese befindet. Dafür wird die Funktion CanGrab aufgerufen. In dieser Funktion wird mit zwei LineTraces die Position der Kante ermittelt.

Dafür wird zunächst mit der Funktion LineTraceForward geprüft, ob sich vor dem Spieler eine Wand befindet. Hierfür wird ein LineTrace vor den Spieler geschickt, der den Channel

Edge nutzt. Jedes Kantenobjekt blockiert den Trace, sodass dieser nicht noch weiter nach vorne gehen kann. Wenn nun also der Trace auf eine Wand trifft, weiß man, wo sich diese Wand befindet und die Position und die Normal dieser Wand wird abgespeichert. Dann wird mit einem zweiten LineTrace, welcher vor dem Spieler von oben versucht, einen Boden zu treffen, wo das obere Ende (Plattform) der Kante ist. Wenn beide Traces ein Objekt getroffen haben, wird aus der Wandposition und der Position, wo die Plattform getroffen wurde, die Position der Kante berechnet. Wenn sich diese Kante nun nicht weiter als 50 Einheiten vom Spieler entfernt befindet und dieser gerade fällt oder klettert, dann greift er nach dieser Kante und ruft das Event Grab auf. Grab macht mehrere Sachen, am wichtigsten ist jedoch, dass isHanging geupdatet und das AnimBP des Spielers darüber informiert wird. Sämtliches Movement wird gestoppt, sonst fliegt der Spieler über die Kante hinweg. Der Movementmodus wird außerdem auf Flying gesetzt, damit die Gravitation nicht mehr auf den Spieler wirkt, und der Spieler wird schließlich an die Kante gesetzt.

Klettert der Spieler nun die Kante hoch, wird eine Montage mit Rootmotion abgespielt, welche den Spieler die Kante hochklettern lässt, und dann, sobald der Spieler hochgeklettert, ist den Movementmodus wieder auf Falling stellt und die dazugehörigen Variablen updatet. Für die Implementierung siehe im BP_AnimMC: ClimbUp, AnimNotify_climbedUp und im BP_Player: EndGrab.

## Klettern

Das Klettern baut auf die Kantenerkennung auf, daher sollte man sich erst mit der Kantenerkennung vertraut machen. Ein Kletterobjekt wird mit Hilfe der ClimbDetection und des LineTraceForward, welcher im Tick Event aufgerufen wird, erkannt, und dessen Position festgestellt. Wenn ich nun also theoretisch Klettern kann, da sich ein Kletterobjekt in der Sphere befindet, und der Spieler die entsprechenden Tasten drückt, wird StartClimb aufgerufen. Dann prüfe ich in StartClimb, ob die Wand weniger als 150 Einheiten vom Spieler weg ist (IsWallNear) und sich vor dem Spieler befindet. Außerdem prüfe ich, ob der Spieler sich nicht schon in einem Movement wie Klettern oder im Hängen befindet, und dass er nicht fällt oder sein Schwert gezogen hat. (Diese Überprüfung findet in CanClimb statt). Ist das alles erfüllt, dann fange ich mit dem Klettern an. Es wird Feedback aktiviert usw. Das wichtigste ist hier auch wieder, dass Variablen geupdatet werden und das AnimBP des Spielers darüber Informiert wird. Der Movementmodus wird hier auch wieder auf Flying gesetzt, damit der Spieler nicht mehr von der Gravitation beeinflusst wird, und der Spieler an die Wand platziert. Sobald der Spieler klettert, wird nun über die InputAxis MoveInward der Input in Z Richtung hinzugefügt. Erreicht der Spieler die obere Kante, greift er diese automatisch, hängt sich an die Kante und das Klettern wird beendet. Deshalb wird in Grab auch EndClimb aufgerufen.

## Anmerkungen

Läuft der Spieler über eine Kante (OnWalkingOffLedge)und drückt dabei die Taste für das Klettern nach unten, wird mit zwei LineTraces geprüft, ob sich unter dem Spieler ein Kantenobjekt befindet(CheckforEdgeBelow). Ist das der Fall, greift der Spieler nach dieser Kante(Grab).

Der Spieler kann auch vom an-eine- Kante-hängen aus das Klettern starten. Die Überprüfungen sind alle gleich, aber deshalb wird, sobald der Spieler das Klettern anfängt, auch EndGrab aufgerufen.

## 5.2 Der LayerChange

Im Folgenden wird die Implementierung des LayerChanges beschrieben, welcher dazu dient die 2-Dimensionalität des Levels etwas aufzubrechen. Ein LayerChange besteht aus drei Actors, dem BP_LayerChangeStart und zwei mal dem BP_LayerChangeEnd.

Diese beiden Actors leiten sich vom BoxTrigger ab. Ihre Funktion ist größtenteils die selbe wie von einem Trigger, jedoch sind sie Fallsensitiv, da je nach Zustand etwas anderes gemacht werden muss. Der BP_LayerChangeStart besitzt zwei Splines, genannt InwardSpline und OutwardSpline. An den Ende der jeweiligen Splines befindet sich jeweils ein BP_LayerChangeEnd. Man beachte hierbei, dass der BP_LayerChangeStart am Start der beiden Splines sitzt. Diese Splines repräsentieren die Pfade, die der Spieler nehmen kann. Geht der Spieler über einen BP_LayerChangeStart oder -End in den LayerChange hinein, dann wird beim BP_Player die Variable FollowPath auf True gesetzt, und er folgt einem Spline. Das tut er, indem der InputVektor des Inputs auf die Tangente des nähesten Punkts des Splines zum Spieler gesetzt wird. So läuft dieser den Spline entlang. Noch dazu wird er jeden Tick auf den Punkt des Splines gesetzt, der dem Spieler am nähesten ist, sodass er auch wirklich dem Pfad folgt.

Wie wird festgelegt, welchem Spline er folgt? Die Splines liegen in einem Array, in dem der Spline der nach hinten führt immer beim Index 0 liegt, und der, der nach vorne führt, auf dem Index 1. Der BP_Player besitzt die Variable LayerChangeSplineUsed, welche Ihm mitteilt, welchen Spline er nutzen soll. Geht der Spieler nun über einen BP_LayerChangeEnd in den LayerChange, dann wird ihm eine Referenz auf den Spline, an dessen Ende der BP_LayerChangeEnd platziert worden ist, an die passende Stelle in sein Spline Arrray gelegt und die Variable LayerChangeSplineUsed auf den passenden Index gesetzt. Geht der Spieler über einen BP_LayerChangeStart in den LayerChange, dann wird ihm eine Referenz auf beide Splines als Array übergeben. Beim Spieler wird die Variable LayerChangeSplineUsed auf den Wert der Variable SplineUsed von BP_LayerChangeStart gesetzt. Solange der Spieler sich im Trigger des BP_LayerChangeStart befindet, kann er den Spline über das Drücken der passenden Taste wechseln, was dann den Wert von LayerChangeSplineUsed auf 0 oder 1 ändert, je nachdem, welchen Wert diese Variable vorher hatte.

Es gibt noch ein paar Kleinigkeiten, die man beim Verwenden des LayerChanges beachten sollte:

Um feststellen zu können, ob der Spieler gerade in einen LayerChange rein oder heraus geht, wird geprüft, aus welcher Richtung er den LayerChange betritt. Dazu muss aber auch festgelegt werden, in welche Richtung sich der LayerChange „aufteilt“. Das geht über die Variable SplitDir.

Außerdem kann man den LayerChange auch im OnePathMode verwenden. Befindet sich der LayerChange in diesen Modus, folgt der Spieler immer während eines LayerChanges dem unter SplineUsed angegebenen Spline.

## 5.3 Das Kamfpsystem

Der Spieler kann angreifen, blocken und ausweichen.

Der Spieler kann nur Angreifen und Blocken, wenn er auch seine Waffen gezogen hat. Hat er diese im Moment nicht gezogen und er drückt trotzdem die Angriffstaste, dann zieht er seine Waffen. Will der Spieler angreifen, dann wird isAttacking auf True gesetzt, und der BP_Player benachrichtigt den BP_Anim_MC, dass eine Angriffsanimation abgespielt werden soll, bzw. BP_Soldiert benachrichtigt den BP_Anim_NPC. Je nachdem, wie häufig angegriffen wurde, wird dabei eine andere Animation abgespielt. An dem Punkt, wo während einer Animation der Spieler vermeintlich den Gegner treffen würde, wird ein BP_DealDamageNotify aufgerufen. Dieser testet mit einem SphereTrace, ob sich vor dem Pawn, der den Angriff ausgeführt hat, ein anderer Pawn befindet. Falls ja, ruft es die Funktion ApplyDamage auf und übergibt den getroffen Pawn als DamagedActor, den Verursacher des Schaden (also den anderen Pawn) als DamageCauser und den Damage-Wert des Verursacher-Pawns als Damage. Damit das Damage-Event auch als solche von der KI erkannt wird, wird es noch reported. Die ApplyDamage Funktion ist eine von der UE vorgegeben Funktion, sie ruft bei dem als Damaged Actor übergebenen Pawn das AnyDamage Event aus. Das AnyDamage Event macht tendenziell im BP_Soldier und BP_Player das gleiche. Es prüft erst, ob der Schaden auch wirklich verrechnet werden muss: dafür prüft es, ob der Pawn gerade blockt und den andern Pawn ansieht, wenn ja, wird der Schaden nicht verrechnet. Genauso auch, wenn der Pawn gerade unbesiegbar ist, da er zum Beispiel gerade eine Ausweichrolle durchführt. Wenn der Schaden verrechnet werden soll, dann wird die Funktion TakeDamage aufgerufen. Diese berechnet und aktualisiert das Restleben. Fällt das Leben dabei unter Null wird die Funktion Die aufgerufen, mehr dazu bei dem jeweiligen BP.

Das Blocken und die Ausweichrolle funktionieren, indem das jeweilige AnimBP darüber informiert wird, dass es jetzt Blocken oder Rollen soll und eine Variable geupdatet wird. Eine Ausweichrolle kann nur ausgeführt werden, wenn der Pawn kein Schild ausgerüstet hat.

# 6. Technische Schulden

Im nächsten Abschnitt werden Dinge angesprochen, die momentan nicht wirklich gut gelöst sind und verbessert werden müssten, aber auch mögliche Probleme, auf welche man in der Zukunft treffen könnte, wenn das Spiel oder gewisse Systeme noch komplexer werden. Diese sind aber nicht nach ihren Prioritäten geordnet.

Der BP_Player wirkt momentan etwas überladen, daher sollte man sich nochmal genauer überlegen, was wirklich in den BP_Player muss und was nicht in einen PlayerController oder eine Komponente ausgelagert werden könnte, wobei alles, was mit Inputs und Movement zu tun hat, auf Grund der technischen Vorgaben von UE im BP_Player verbleiben muss. Man könnte zum Beispiel das geplante Inventarsystem in den PlayerController verlagern, dann wäre ein Respawn auch mehr UE Konventionen konform möglich.

Was löst man als Funktion und was als Makro? Hier gibt es keine Konvention, daher ist es immer wieder ein Problem. Insgesamt packt man beim Verwenden von BPs im Vergleich zu wirklichem Code viel mehr kleinere Aufgaben in Funktionen, da sie schnell sehr groß wirken. Man hat recht schnell 15 Nodes zusammen, die aber eigentlich eine ganz simple Berechnung durchführen. Hier müsste man sich nochmal mehr Gedanken dazu machen was man wann verwendet, und eigene Konventionen aufstellen. Man sollte aber weiterhin die Node-Anzahl in den Funktionen und Makros überschaubar halten, ansonsten wird es sehr schnell unübersichtlich.

Man sollte die Booleans, die den Status des BP_Player überwachen, vielleicht teilweise durch ein Enum ersetzen, welches angibt, in welchem Zustand er sich gerade befindet. Er kann ja zum Beispiel nicht gleichzeitig Klettern und sein Schwert gezogen haben. Oder an einer Kante hängen und Klettern. Hier müsste man aber nochmal genauer prüfen, wie und ob das machbar wäre. Momentan führt es dazu, dass an gewissen Stellen recht viele Booleans geprüft werden müssen, was es etwas unübersichtlich macht zu erkennen, wann der Code jetzt genau ausgeführt werden soll.