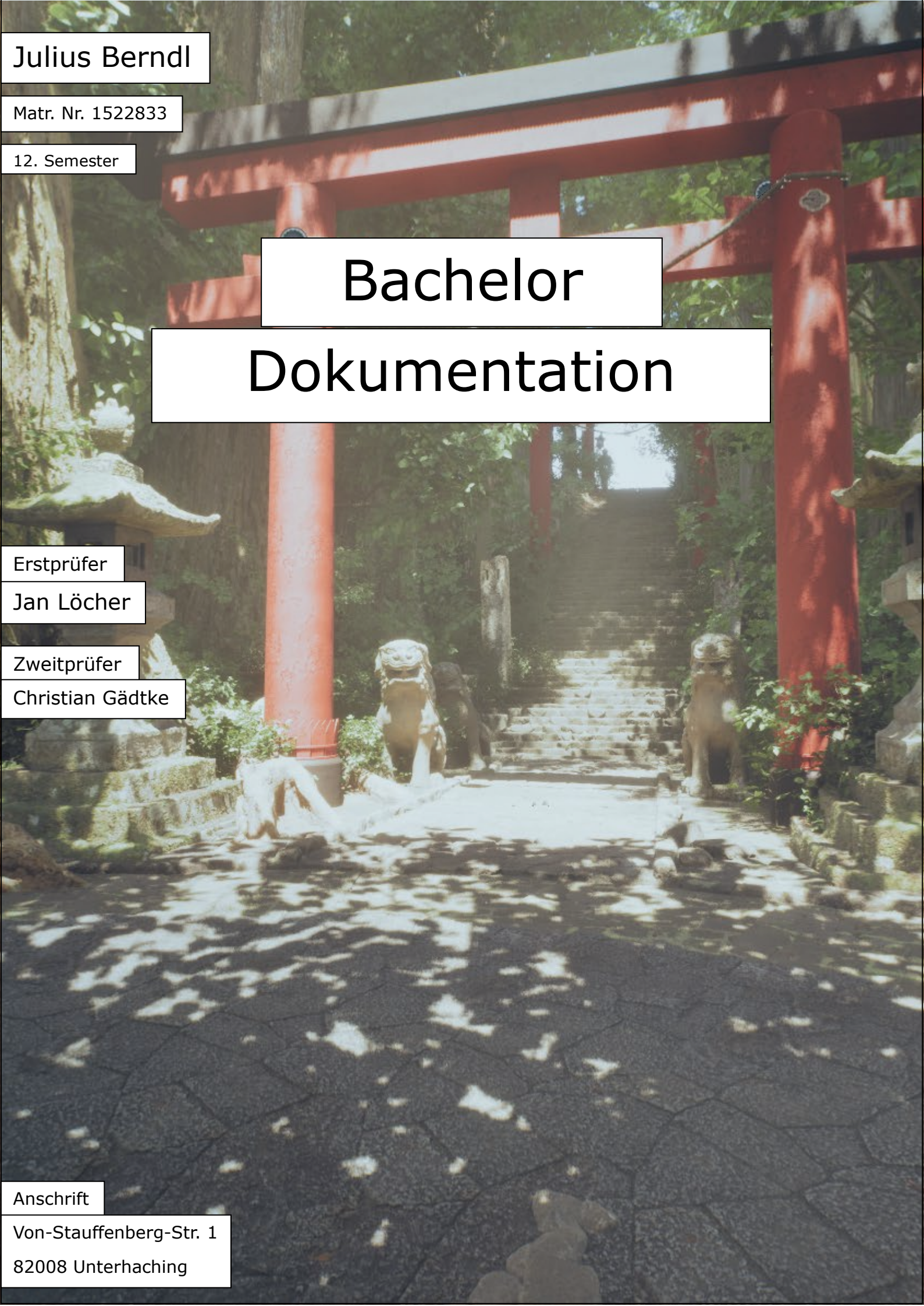
Julius Berndl

Matr. Nr. 1522833

12. Semester

Bachelor Dokumentation



Erstprüfer

Jan Löcher

Zweitprüfer

Christian Gädtke

Anschrift

Von-Stauffenberg-Str. 1

82008 Unterhaching

Inhaltsverzeichnis

Einleitung

Bei einer der vorherigen Bachelor Präsentation gab es ein sehr spannendes Konzept zu einem Third Person Action Game an dessen Prototypisierung ich interessiert gewesen wäre. Zuerst war dies auch der Plan für das anstehende Bachelorprojekt. Aufgrund von Kommunikationsproblemen, die sich bereits vor Beginn der Bachelorarbeit abzeichneten, wurde dieses Vorhaben jedoch zu Beginn der Bachelorphase verworfen. Stattdessen wurde beschlossen, in der Bachelorarbeit einen 3D-Game-Prototyp zu erstellen, der meine bisherigen Spezialisierungen vertiefen sollte. Das Ziel war es, effektive Gameplay-Mechaniken und essenzielle Spielbestandteile zu entwickeln und abzuhandeln, ohne zwingend einen vollständigen Gameplay-Loop oder Level zu erstellen.

Projektfindung

Die Schwerpunkte des Projekts liegen auf der Erstellung des Character Controllers und der Enemy AI. Dies umfasst die Anpassung aller verwendeten Assets sowie weiterer Elemente, die das visuelle und klangliche Erlebnis unterstützen. Ein weiteres Ziel war es, das Portfolio zu bereichern, was Projekte wie ein 2D-Arcade-Game, einen First-Person-Shooter und mehrere 2D-Metroidvania-Games ausschloss. Da bisher alle Projekte in der Unity Engine erstellt wurden, sollte dieses Projekt in der Unreal Engine realisiert werden, um die eigenen Fähigkeiten zu erweitern und den Anforderungen aktueller Jobangebote gerecht zu werden.

Projektideen

1. **Open World RPG**:
    - Ein Open-World-RPG nach Vorbildern wie Assassins Creed oder Horizon Zero Dawn wurde verworfen, da der Fokus nicht auf der Erstellung einer Open World oder einer umfangreichen Story liegen sollte.
2. **Third-Person-Shooter, Hack and Slash, Jump and Run und Beat Game**:
    - Eine Mischung aus verschiedenen Genres mit Beat-Mechaniken wurde in Betracht gezogen. Da jedoch die notwendigen Assets für die Visualisierung der Mechaniken fehlten und keine kostengünstigen Alternativen verfügbar waren, wurde diese Idee verworfen.
3. **Third-Person Hack and Slash**:
    - Ein simples Hack and Slash nach Vorbildern wie Devil May Cry, Nier Automata oder Ninja Gaiden wurde als Grundlage gewählt. Dies ermöglichte den Fokus auf die Entwicklung des Character Controllers und der Enemy AI. Die notwendigen Assets für die Visualisierung waren bereits vorhanden oder leicht beschaffbar.

Grobe Planung der Spielbestandteile

Player Character

- Zwei Bewegungsmodi: Normal und Targeting
- Targeting-System, das den Gegner in Blickrichtung erfasst
- Light Attacks in vier Elementartypen (Luft, Erde, Feuer, Wasser)
- Heavy Attacks (dunkle Attacken)
- Combo-Attacken
- Air Combat
- Block und Parry
- Möglichkeit, die Schadensart der Attacken zu ändern

Enemy AI

- Hit-Reactions basierend auf der Angriffsdirection
- Melee- und Ranged-Enemies
- Wahrnehmungssystem
- Koordinierte Angriffe mehrerer Gegner
- Boss Enemy mit unterschiedlichen Attacken

Damage System

- Verwaltung von Leben und Tod

UI

- Life Bar für Player und Gegner

Effekte

- Angriffs- und Impact-Effekte
- Effekte für Blocken und Parrieren

Levelbau

- Landschaftsdesign und Dojo für den Bosskampf
- Bereiche für Kämpfe gegen mehrere und patrouillierende Gegner

Sound

- Angriffsgeräusche
- Footsteps
- Atmosphärische Klänge

Die Story

Inspiriert von der Serie Loki, in der die Hüter des wahren Zeitstrahls andere Zeitsträhle stutzen, um den Zusammenbruch der Zeit zu verhindern, und dem Anime Bleach, wo Shinigamis, auch Sensenmänner genannt, gegen Hollows kämpfen, die aus den Seelen verstorbener Menschen entstehen, entfaltet sich die Geschichte in einem Universum, das von der Philosophie von Yin und Yang durchdrungen ist.

Der Spieler schlüpft in die Rolle eines Wächters, dessen Aufgabe es ist, die Bindestellen der Dimensionen zu schützen. Diese Bindestellen verknüpfen verschiedene Dimensionen und spiegeln Orte wider, die zufällig in beiden Dimensionen existieren. Diese Bindestellen existieren außerhalb von Raum und Zeit, wodurch ein Ort in einer Dimension im Jahr 900 v. Chr. und in einer anderen Dimension erst im Jahr 5000 existieren kann.

Im Tod begibt sich die Seele eines Menschen auf eine Reise durch diese Bindestellen, um in einer anderen Dimension zu einer anderen Zeit wiedergeboren zu werden. Über diese Bindestellen wacht eine mysteriöse Entität aus göttlichem Licht, deren Gestalt und Stimme nur in der inneren Wahrnehmung erkennbar ist. Zu Beginn der Zeit bewegten sich die Seelen frei und ungestört von einer Dimension zur anderen.

Doch nach Jahrhunderten tauchte ein dunkles Wesen auf, das von der Lichtentität wahrgenommen wurde, obwohl es nicht greifbar war. Dieses böse Wesen machte es sich zur Aufgabe, die Seelen, die die Bindestellen passieren, zu beschmutzen und zu verunreinigen. So wurde die Sünde und das Böse geboren. Einige dieser verunreinigten Seelen stürzten Dimensionen ins Chaos, entfachten Kriege und verursachten Klimakatastrophen auf ganzen Planeten.

Das Lichtwesen war hilflos und verwehrte den verunreinigten Seelen den Durchgang durch die Bindestellen, sperrte sie dort ein. Doch solange eine Bindestelle versperrt war, konnten keine Seelen mehr passieren. Um die Passagen zu reinigen, wählte das Wesen reine Seelen aus und gab ihnen Körper und Mittel diese Bindestellen wieder zu reinigen.

Wahl der Engine

Unreal Engine 5 wurde gewählt, da sie umfangreiche Built-In-Tools und Möglichkeiten zur Bearbeitung von 3D-Assets und Animationen bietet. Obwohl die Erfahrung mit der Unreal Engine begrenzt war, überwogen die Vorteile für die Anforderungen des Projekts. Die Entscheidung wurde nach Abwägung der Stärken beider Engines und einem kurzen Test innerhalb der Unreal Engine Blueprints getroffen.

Sammlung der externen Assets

Zu Beginn wurden Assets für den Player, Animationen und ein Schwertmodell benötigt. Zwei Animationspakete (Powerful Sword Pack und Sword Animation Pack) wurden erworben, während das Player-Modell aus dem Cyberpunk Street Boy Asset Pack verwendet wurde. Weitere Assets wurden während des Projekts ergänzt. Eine Liste der verwendeten Assets und Plugins ist am Ende der Dokumentation enthalten.

Vorwort zur Projekterstellung in Unreal Engine

Der Einstieg in die Unreal Engine war herausfordernd, da viele Built-In-Tools zunächst unbekannt waren. Einige Grundlogiken ließen sich gut von Unity und C# übertragen, während andere Unterschiede zu unerwünschten Ergebnissen führten. Ein schwerwiegender Fehler der genutzten UE5 Version zerstörte immer wieder den BP_ThirdPersonCharacter Blueprint, was zu einer Umstellung auf ein besseres Backup-System mit GitHub führte. Aufgrund der vielen Änderungen und kleinen Anpassungen während der Projekterstellung wurde beschlossen, keine detaillierte Ablaufdokumentation zu erstellen, sondern sich auf den Endzustand und die wichtigsten Schritte, die zu diesem Zustand geführt haben zu konzentrieren. Als Startpunkt des Projektes wurde das Third Person Template der Unreal Engine verwendet.

DamageSystem

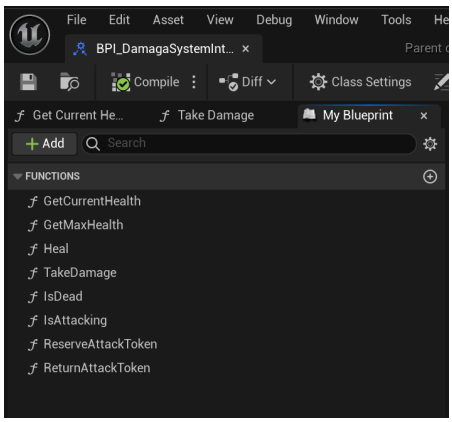
Ein Großteil der Projektzeit wurde mit dem integrierten Damage System von Unreal Engine gearbeitet. Anfangs funktionierte dies auch gut, jedoch bin ich es aus Unity gewohnt, solche Systeme selbst zu schreiben. Mit dem Fortschreiten des Projekts entstanden zunehmend Workarounds, um bestimmte Daten beim Austeilen von Schaden weiterzugeben. Dies führte zu Unübersichtlichkeit und es gab immer mehr Funktionen, wie Blocken oder Parieren, die sowohl von NPCs als auch von Charakteren genutzt werden sollten und in Verbindung mit dem Damage System standen.

Später im Projekt entstand daher der Wunsch, ein eigenes Damage System zu erstellen, das Aufgaben im Zusammenhang mit Schaden, die sich bei Spielern und NPCs überschneiden, innerhalb des Damage Systems abhandeln kann. Leider ist dieses eigene Damage System noch nicht vollständig implementiert und bietet Funktionen, die teilweise noch nicht genutzt werden. Beim Erstellen wurde versucht, es so flexibel wie möglich zu gestalten, um für alle Eventualitäten gewappnet zu sein, was dazu führte, dass einige Funktionen noch nicht vollständig genutzt werden.

Das Damage System besteht im Wesentlichen aus einer Actor Component Blueprint Class namens BPC_DamageSystem, einem Blueprint Interface BPI_DamageSystemInterface und einer Structure S_DamageInfo, ergänzt durch Enumerations.

BPI_DamageSystemInterface

Das BPI_DamageSystemInterface definiert eine Reihe von Funktionen, die von verschiedenen Blueprints implementiert werden können, um ein einheitliches Damage System zu erstellen. Diese Funktionen umfassen die Verwaltung der Gesundheit, das Verarbeiten von Schaden, das Heilen, das Überprüfen von Zuständen wie Tod und Angriff sowie das Reservieren und Zurückgeben von Angriffstokens.



GetCurrentHealth

Gibt den aktuellen Gesundheitswert des Charakters zurück. Diese Funktion hat einen Float-Output, der den aktuellen Gesundheitswert liefert.

GetMaxHealth

Gibt den maximalen Gesundheitswert des Charakters zurück. Diese Funktion hat einen Float-Output, der den maximalen Gesundheitswert liefert.

Heal

Diese Funktion nimmt eine Menge an Heilung als Float-Input und gibt den neuen Gesundheitswert nach der Heilung als Float-Output zurück. Sie wird verwendet, um den Charakter zu heilen.

TakeDamage

Diese Funktion ist für die Verarbeitung von Schaden zuständig. Sie benötigt mehrere Eingaben:

- DamageInfo: Eine Struktur, die Details über den Schaden enthält.
- DamageCauser: Der Actor, der den Schaden verursacht hat.
- AttackType: Der Typ des Angriffs.
- DamageDirection: Die Richtung, aus der der Schaden kommt.

Als Output gibt es einen Boolean-Wert WasDamaged, der angibt, ob der Schaden erfolgreich zugefügt wurde.

IsDead

Gibt einen Boolean-Wert IsDead zurück, der anzeigt, ob der Charakter tot ist.

IsAttacking

Gibt einen Boolean-Wert IsAttacking zurück, der anzeigt, ob der Charakter gerade angreift.

ReserveAttackToken

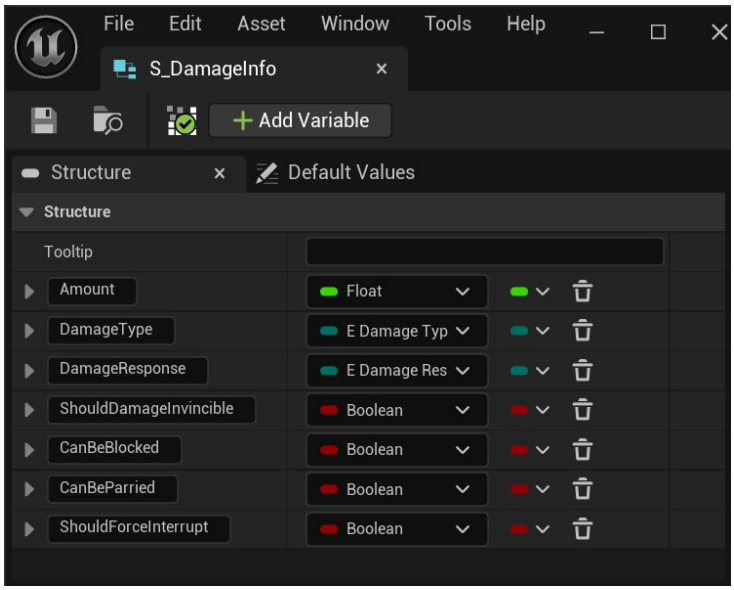
Diese Funktion ist für das Reservieren von Angriffstokens verantwortlich, die für das Gruppenverhalten der Gegner benötigt werden. Sie nimmt einen Integer-Wert Amount als Input und gibt einen Boolean-Wert Success als Output zurück, der anzeigt, ob das Reservieren erfolgreich war.

ReturnAttackToken

Diese Funktion gibt die Angriffstokens zurück, die für das Gruppenverhalten der Gegner benötigt werden. Sie hilft dabei, die Tokens wieder freizugeben, damit sie von anderen Gegnern verwendet werden können.

S_DamageInfo Structure

Die S_DamageInfo-Struktur dient als Container für alle relevanten Informationen, die zur Verarbeitung von Schaden innerhalb des DamageSystems benötigt werden. Diese Struktur ermöglicht es, detaillierte Schadensdaten zu übermitteln, einschließlich der Schadensmenge, des Schadens- und Reaktionstyps sowie verschiedener boolescher Werte, die bestimmen, wie der Schaden behandelt werden soll. Durch die Verwendung dieser Struktur kann das Damage System flexibel und präzise auf verschiedene Schadensszenarien reagieren.



Amount

Gibt die Menge des zugefügten Schadens an. Dies ist der numerische Wert des Schadens, der auf den Charakter angewendet wird.

DamageType

Bestimmt den Typ des Schadens, z.B. Melee, Projectile, Explosion, etc. Diese Information wird verwendet, um spezifische Reaktionen basierend auf dem Schadenstyp zu implementieren.

DamageResponse

Gibt die Reaktion des Charakters auf den Schaden an, z.B. HitReaction, Stagger, Stun, KnockBack. Diese Enumeration definiert, wie der Charakter auf den erhaltenen Schaden reagiert.

ShouldDamageInvincible

Ein boolescher Wert, der angibt, ob der Schaden auch dann zugefügt werden soll, wenn der Charakter unverwundbar ist.

CanBeBlocked

Ein boolescher Wert, der angibt, ob der Schaden geblockt werden kann.

CanBeParried

Ein boolescher Wert, der angibt, ob der Schaden pariert werden kann.

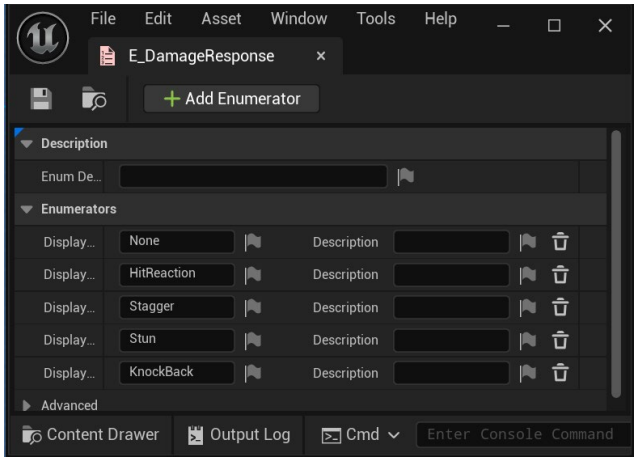
ShouldForceInterrupt

Ein boolescher Wert, der angibt, ob der Schaden die aktuelle Aktion des Charakters unterbrechen soll.

Enumerator

E_DamageType Enumerator

Der E_DamageType Enumerator ermöglicht es, verschiedene Arten von Schaden im Spiel zu klassifizieren die eine Zustands Veränderung hervorrufen sollen. Obwohl er aktuell noch nicht genutzt wird, ist er für die zukünftige Erweiterungen und Anpassungen des Damage Systems vorgesehen.



None

Keine spezifische Schadensreaktion. Dies kann als Standardwert verwendet werden, wenn keine spezifische Reaktion erforderlich ist.

HitReaction

Diese Reaktion tritt auf, wenn der Charakter getroffen wird. Sie könnte eine kurze Animation oder Verzögerung sein, die den Treffer darstellt, ohne den Charakter stark zu beeinträchtigen.

Stagger

Diese Reaktion lässt den Charakter taumeln, was ihn für kurze Zeit unfähig macht, Aktionen auszuführen. Dies simuliert den Effekt, von einem starken Angriff getroffen zu werden und das Gleichgewicht zu verlieren.

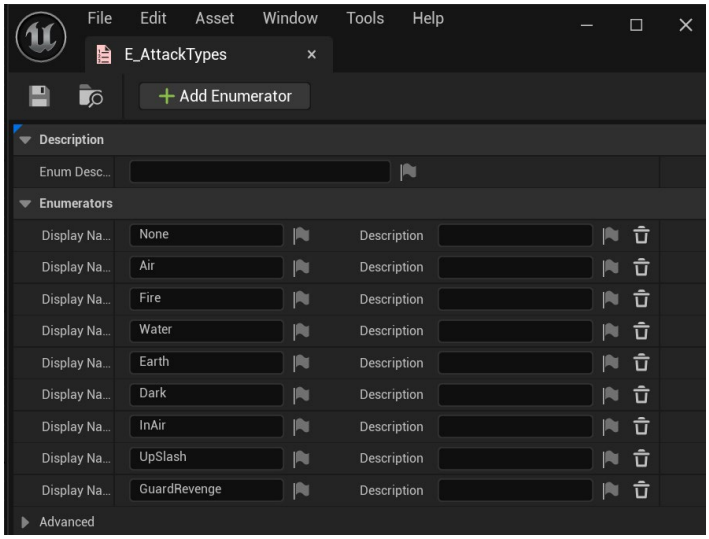
Stun

Diese Reaktion betäubt den Charakter, wodurch er für eine bestimmte Zeitspanne bewegungsunfähig wird. Dies ist eine stärkere Form der Unterbrechung als das Taumeln.

KnockBack

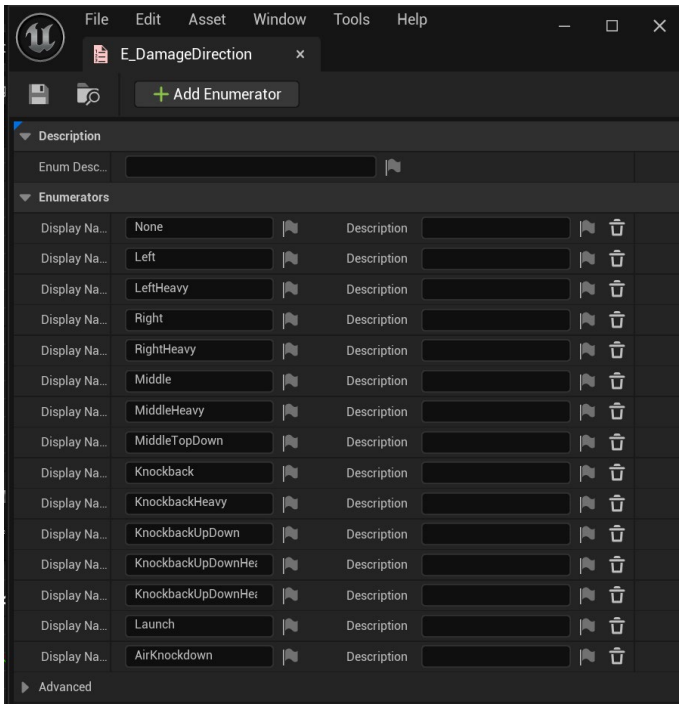
Diese Reaktion stößt den Charakter zurück, was ihn von seiner aktuellen Position wegbewegt. Dies kann verwendet werden, um Angriffe darzustellen, die den Charakter physisch zurückwerfen.

E_AttackTypes Enumerator



Der E_AttackTypes Enumerator enthält eine Vielzahl von Angriffstypen, die die Blockfähigkeit eines Charakters beeinflussen können. Die Elementarangriffe (Air, Fire, Water, Earth) haben z.b. eine spezielle Wechselwirkung, bei denen sie den Block des gegensätzlichen Elements durchbrechen können.

E_DamageDirection Enumerator



Der E_DamageDirection Enumerator enthält verschiedene Werte, die genutzt werden, um die Hit-Reaktion basierend auf der Richtung des Angriffs zu definieren. Dieser Enumerator wurde bereits vor der Implementierung des Damage Systems erstellt und sollte langfristig angepasst werden, um das Damage System übersichtlicher zu gestalten.

BPC_DamageSystem

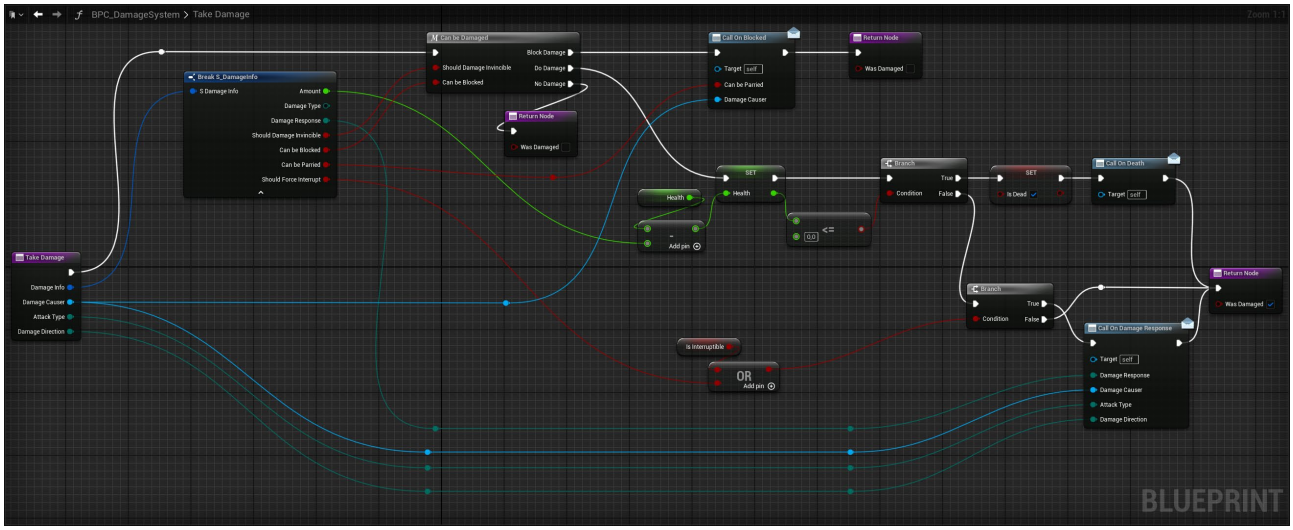
Für die Umsetzung der Logik des Damage Systems wird die Blueprint Class Actor Component verwendet, da das System sowohl auf den Player als auch auf alle anderen Actors angewendet werden soll, die Schaden nehmen können. Dabei ist es wichtig, dass die BPC_DamageSystem Actor Component den jeweiligen Blueprints als Komponente hinzugefügt wird.

Im Event Graph des BPC_DamageSystem Blueprints wird keine Logik verarbeitet; die gesamte Logik wird innerhalb der Funktionen der Komponente abgehandelt.

Funktionen

TakeDamage

Die Take Damage Funktion im BPC_DamageSystem Blueprint verarbeitet den zugefügten Schaden an einem Actor.



1. **Eingehende Parameter:**
    - Die Funktion erhält Schadeninformationen (Damage Info), den Schadenverursacher (Damage Causer), den Angriffstyp (Attack Type) und die Schadensrichtung (Damage Direction).
2. **Schaden aufschlüsseln:**
    - Der eingehende Schaden wird in seine Bestandteile zerlegt (S_DamageInfo).
3. **Schaden validieren:**

    - Mithilfe der Funktion "Can be Damaged" wird geprüft, ob der Schaden abgeblockt oder pariert werden kann und ob der Charakter unverwundbar ist. Basierend auf diesen Prüfungen wird der Schaden entweder geblockt, als Schaden durchgeführt oder es passiert nichts.

4. **Blockierung und Parierung:**
    - Falls der Schaden geblockt werden kann, wird "Call On Blocked" aufgerufen und die Funktion beendet sich, indem "Was Damaged" auf "False" gesetzt wird.
5. **Schaden anwenden:**
    - Wenn der Schaden durchgeführt werden kann, wird die aktuelle Gesundheit um den Schaden reduziert.
6. **Tod prüfen:**
    - Nach der Gesundheitsreduktion wird überprüft, ob die Gesundheit null oder niedriger ist. Wenn ja, wird das "Is Dead" Flag gesetzt und die "Call On Death" Funktion aufgerufen.
7. **Schaden Reaktion:**
    - Unabhängig davon, ob der Charakter stirbt oder nicht, wird die "Call On Damage Response" Funktion aufgerufen, die die spezifische Reaktion auf den Schaden verarbeitet.
8. **Interruptionslogik:**
    - Es wird geprüft, ob der Charakter unterbrechbar ist (Is Interruptible) oder ob der Schaden eine Unterbrechung erzwingen soll. Falls eine Unterbrechung erforderlich ist, wird die entsprechende Logik ausgeführt.

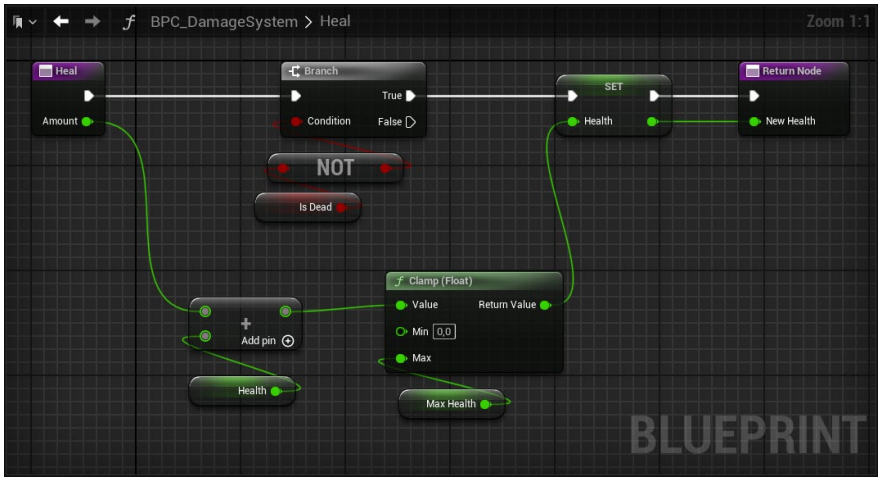
Am Ende wird "Was Damaged" auf "True" gesetzt, wenn der Charakter tatsächlich Schaden genommen hat.

Die detaillierte Logik sieht folgendermaßen aus:

- Wenn der Charakter blocken oder parieren kann, wird der Schaden entsprechend behandelt und die Funktion beendet.
- Falls der Charakter den Schaden annimmt, wird die Gesundheit angepasst und bei Bedarf Tod- und Unterbrechungs-Events ausgelöst.
- Die Funktion stellt sicher, dass der Schaden korrekt angewendet und entsprechende Reaktionen ausgelöst werden, unabhängig davon, ob der Charakter den Angriff überlebt oder nicht und gibt dabei Relevante Informationen für die Reaktion über die OnDamageResponse weiter.

Heal

Die Funktion "Heal" im BPC_DamageSystem führt eine Heilungsaktion durch und aktualisiert die Gesundheit des Charakters entsprechend.

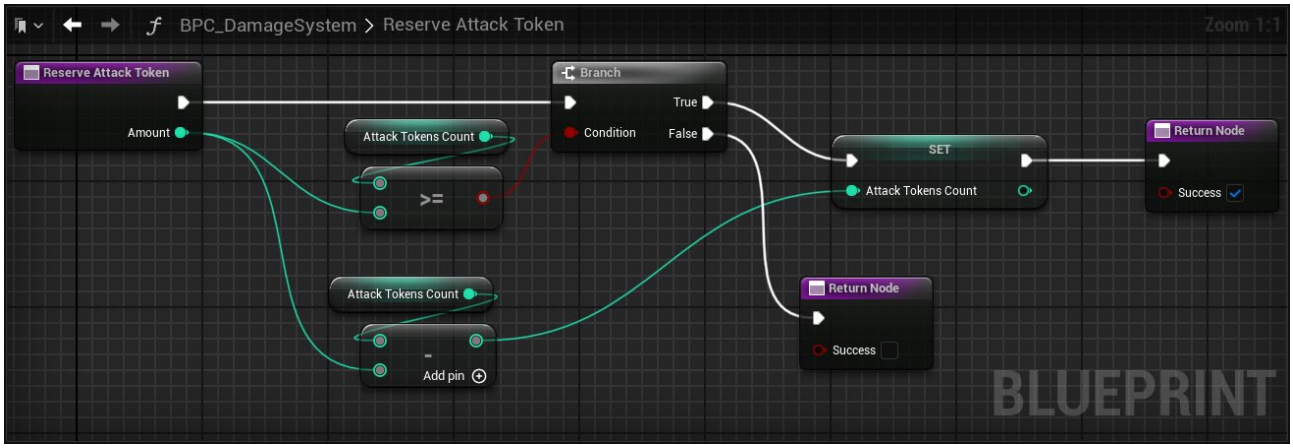


1. **Eingehender Parameter:**
    - Die Funktion erhält den Heilungsbetrag (Amount).
2. **Lebensstatus überprüfen:**
    - Ein Branch-Node prüft, ob der Charakter nicht tot ist (NOT Is Dead). Wenn der Charakter tot ist, wird die Funktion beendet und keine Heilung angewendet.
3. **Heilungsbetrag addieren:**
    - Wenn der Charakter lebt, wird der Heilungsbetrag zur aktuellen Gesundheit addiert.
4. **Gesundheitswert begrenzen:**
    - Der neue Gesundheitswert wird mit der Clamp-Funktion begrenzt, sodass er nicht unter 0 und nicht über die maximale Gesundheit hinausgeht.
5. **Gesundheit setzen:**
    - Der begrenzte Gesundheitswert wird als neue Gesundheit des Charakters gesetzt (SET Health).
6. **Rückgabewert:**
    - Die Funktion gibt die neue Gesundheit des Charakters als Rückgabewert zurück (Return Node New Health).

Zusammengefasst sorgt die Funktion dafür, dass der Charakter nur geheilt wird, wenn er lebt, und stellt sicher, dass die Gesundheit nicht über das Maximum hinausgeht.

ReserveAttackToken

Die Funktion "Reserve Attack Token" im BPC_DamageSystem überprüft und reserviert eine bestimmte Anzahl von Angriffstokens.

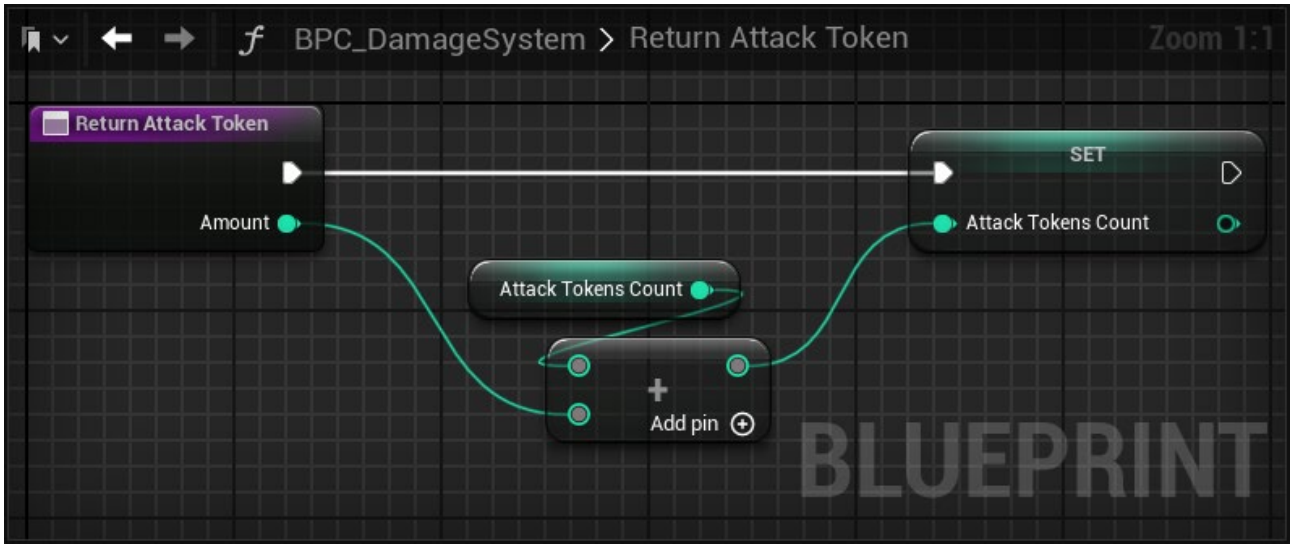


1. **Eingehender Parameter:**
    - Die Funktion erhält die Anzahl der zu reservierenden Tokens (Amount).
2. **Verfügbarkeit der Tokens prüfen:**
    - Ein Branch-Node überprüft, ob die Anzahl der aktuellen Angriffstokens (Attack Tokens Count) größer oder gleich dem benötigten Betrag (Amount) ist.
3. **Tokens reservieren:**
    - Wenn genügend Tokens verfügbar sind (True-Zweig):
        - Die Anzahl der aktuellen Angriffstokens wird um den benötigten Betrag reduziert.
        - Der neue Wert der Angriffstokens wird gesetzt (SET Attack Tokens Count).
        - Die Funktion gibt "Success" zurück, um anzuzeigen, dass die Reservierung erfolgreich war (Return Node Success).
4. **Reservierung fehlgeschlagen:**
    - Wenn nicht genügend Tokens verfügbar sind (False-Zweig):
        - Die Funktion gibt "Success" mit dem Wert "False" zurück, um anzuzeigen, dass die Reservierung fehlgeschlagen ist (Return Node Success).

Zusammengefasst überprüft diese Funktion, ob genügend Angriffstokens vorhanden sind, reserviert die angeforderte Anzahl und gibt den Erfolg oder Misserfolg der Reservierung zurück.

ReturnAttackToken

Die Funktion "Return Attack Token" im BPC_DamageSystem fügt eine bestimmte Anzahl von Angriffstokens zurück zum verfügbaren Pool hinzu.



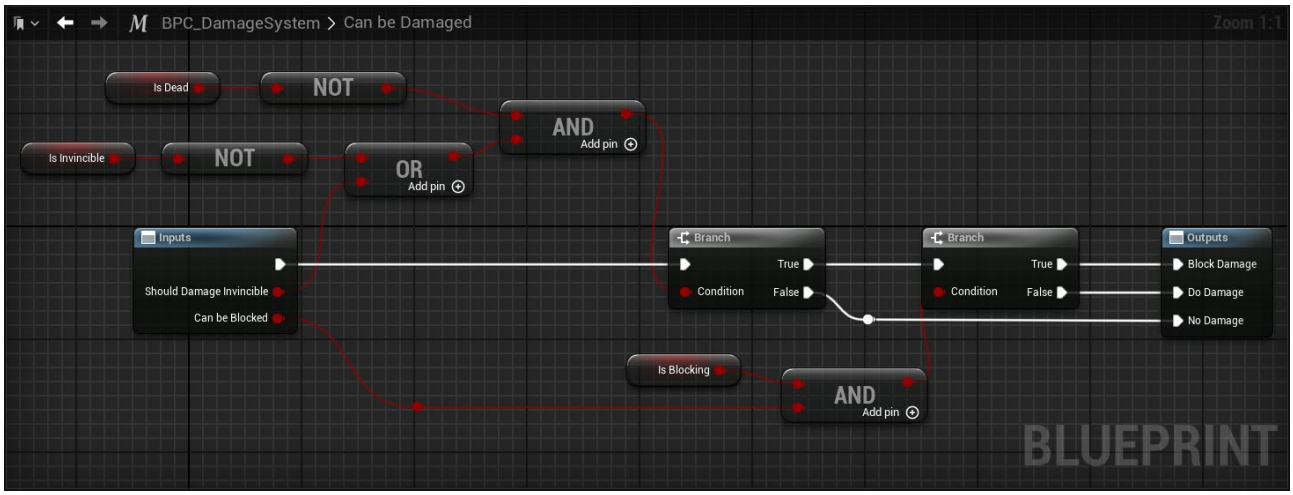
1. **Eingehender Parameter:**
    - Die Funktion erhält die Anzahl der zurückzugebenden Tokens (Amount).
2. **Tokens zurückgeben:**
    - Die aktuelle Anzahl der Angriffstokens (Attack Tokens Count) wird um den angegebenen Betrag (Amount) erhöht.
3. **Tokens aktualisieren:**
    - Der neue Wert der Angriffstokens wird gesetzt (SET Attack Tokens Count).

Zusammengefasst fügt diese Funktion die angegebene Anzahl von Angriffstokens zum Pool der verfügbaren Tokens hinzu und aktualisiert den aktuellen Stand der Angriffstokens.

Macros

CanBeDamaged

Das "Can be Damaged"-Makro im BPC_DamageSystem prüft, ob ein Charakter Schaden erleiden kann und gibt den entsprechenden Zustand zurück.



1. **Eingänge:**
    - Should Damage Invincible (Bool)
    - Can be Blocked (Bool)
2. **Zustände überprüfen:**
    - Is Dead: Ein Bool, der überprüft, ob der Charakter tot ist.
    - Is Invincible: Ein Bool, der überprüft, ob der Charakter unverwundbar ist.
3. **Überprüfungslogik:**
    - **Nicht tot und nicht unverwundbar oder Unverwundbar aber Schaden erlaubt:**
        - Ein NOT-Knoten überprüft den Zustand von Is Dead.
        - Ein weiterer NOT-Knoten überprüft den Zustand von Is Invincible.
        - Ein OR-Knoten kombiniert den Zustand von Is Invincible und Should Damage Invincible.
        - Ein AND-Knoten überprüft, ob der Charakter nicht tot ist und entweder nicht unverwundbar oder Schaden trotz Unverwundbarkeit erlaubt ist.
4. **Verarbeitung:**
    - Ein Branch-Knoten überprüft die Bedingung des AND-Knotens.
    - Wenn die Bedingung wahr ist:
        - Ein zweiter Branch-Knoten überprüft den Can be Blocked-Eingang.
        - Falls Can be Blocked wahr ist:

        - Es wird überprüft, ob der Charakter blockt (Is Blocking). Wenn ja, wird der Ausgang Block Damage gesetzt.
        - Falls Is Blocking falsch ist, wird der Ausgang Do Damage gesetzt.
    - Wenn die Bedingung falsch ist:
        - Der Ausgang No Damage wird gesetzt.

5. **Ausgänge:**
    - Block Damage (Bool): Setzt diesen Ausgang, wenn der Charakter blockt.
    - Do Damage (Bool): Setzt diesen Ausgang, wenn der Charakter Schaden nehmen kann.
    - No Damage (Bool): Setzt diesen Ausgang, wenn der Charakter keinen Schaden nehmen kann.

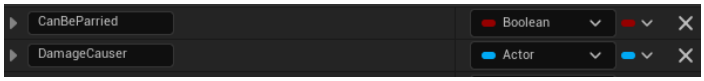
Zusammengefasst prüft das Makro, ob der Charakter Schaden nehmen kann, indem es den Zustand (tot, unverwundbar) und die aktuellen Bedingungen (Blocken, Schaden trotz Unverwundbarkeit) überprüft, und gibt entsprechend zurück, ob Schaden erlitten, blockiert oder verhindert wurde.

Event Dispatcher

OnDeath

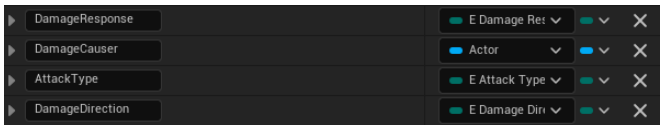
Wird ausgelöst, wenn der Charakter stirbt. Kein Input notwendig

OnBlocked



Wird ausgelöst, wenn ein Angriff geblockt wird.

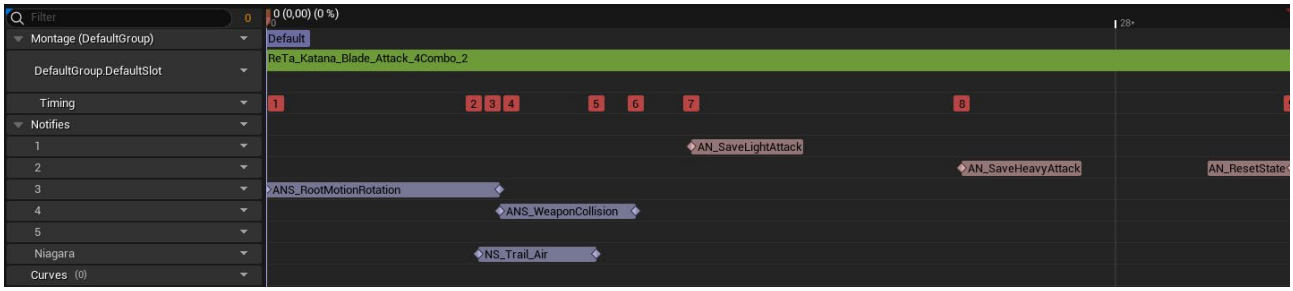
OnDamageResponse



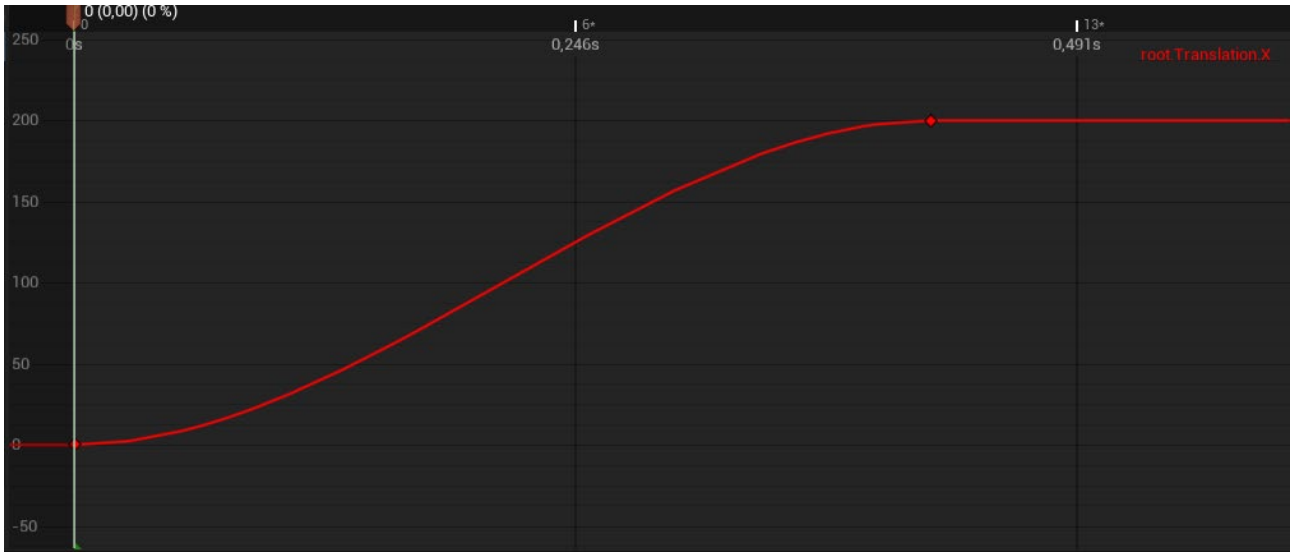
Wird ausgelöst, um zusätzliche Schadensreaktionen zu handhaben.

Player Character

Animation



Zur Umsetzung des Players werden einige Animationen benötigt, die aus externer Quelle stammen. Aus den Animationen werden Anim Montages erstellt. Innerhalb dieser Montages werden Anim Notifys und Anim Notify States genutzt, um weitere Logik abzuhandeln. Zudem werden auch Sounds und Effekte innerhalb der Montages abgehandelt. Die Root Motion der Animationen wird genutzt, um einerseits die Bewegung zu übernehmen und andererseits den Character den Input des Spielers ignorieren zu lassen. Dieser wird stellenweise dann nur manuell erlaubt.



Im Verlauf des Projektes wurde vor allem die RootMotion innerhalb vieler Animationen immer wieder spezifisch im Anim Graph editiert um Distanzen, die der Charakter bei einer Animation zurücklegt anzupassen. Auch Blend Times und Ablauf Geschwindigkeiten wurden in so gut wie jeder Animation angepasst und aufeinander abgestimmt. Anderen Ortes wurden Animation zu einer neuen Animation kombiniert.

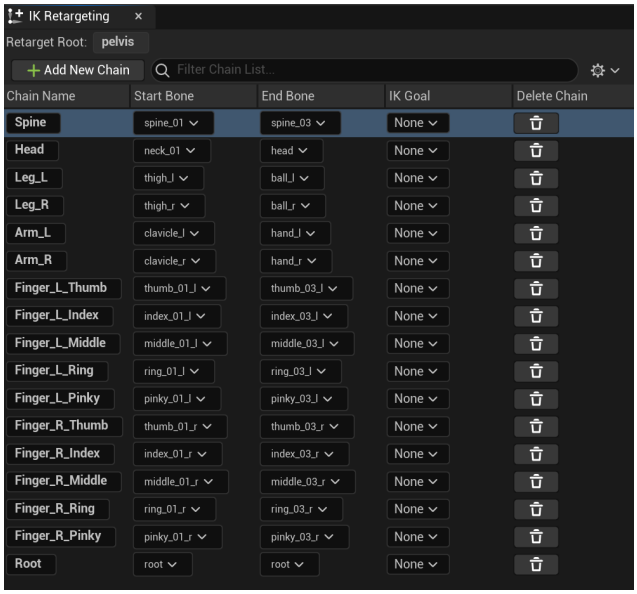
Retargeten der AnimationsAssets

Der ursprüngliche Plan sah vor, das Mesh und die Animationen des PlayerCharacters auszutauschen. Hierfür wurden Assets vom Unreal Engine Marketplace verwendet. Als Mesh für den Player wurde das "Cyberpunk Street Boy"-Asset von Fajrul Falakh eingesetzt. Die Animationen stammten aus dem "Sword Animation Pack" von 9CG und dem "2 Assets Powerful Sword Pack" von Grruzam. Dabei stellte sich heraus, dass das verwendete Mesh und die Animationen nicht miteinander kompatibel waren.

Unreal Engine ermöglicht zwar das Ausführen von Animationen auf einem nicht kompatiblen Mesh, dies führt jedoch je nach Grad der Inkompatibilität zu seltsamen Deformationen oder Fehlern beim Abspielen der Animation. In diesem Fall sahen die Animationen inkorrekt aus, und das Mesh wurde dabei merkwürdig deformiert. Daher mussten die Animationen retargeted werden.

Um dies zu erreichen, wird zunächst ein IK Rig mit dem Skeletal Mesh erstellt, das mit den Animationen kompatibel ist, sowie eines mit dem Skeletal Mesh, für das die Animationen kompatibel gemacht werden sollen. Innerhalb der neu erstellten Rigs werden unter dem Tab "IK Retargeting Chains" für die wichtigsten Knotenpunkte erstellt, die die meisten Animationen nutzen. Angefangen bei der Spine und, vor allem wichtig, um RootMotion korrekt zu retargeten, der Root des Skeletal Meshes.

Wichtig ist hierbei, in beiden Rigs die Bones exakt gleich zu benennen und die am besten passenden Start- und End-Bones in den jeweiligen Chains zuzuweisen. In diesem Fall erzielte ich die besten Ergebnisse, wenn die Chains in den beiden IK Rigs wie in Abbildungen 1 zu sehen ist konfiguriert waren.

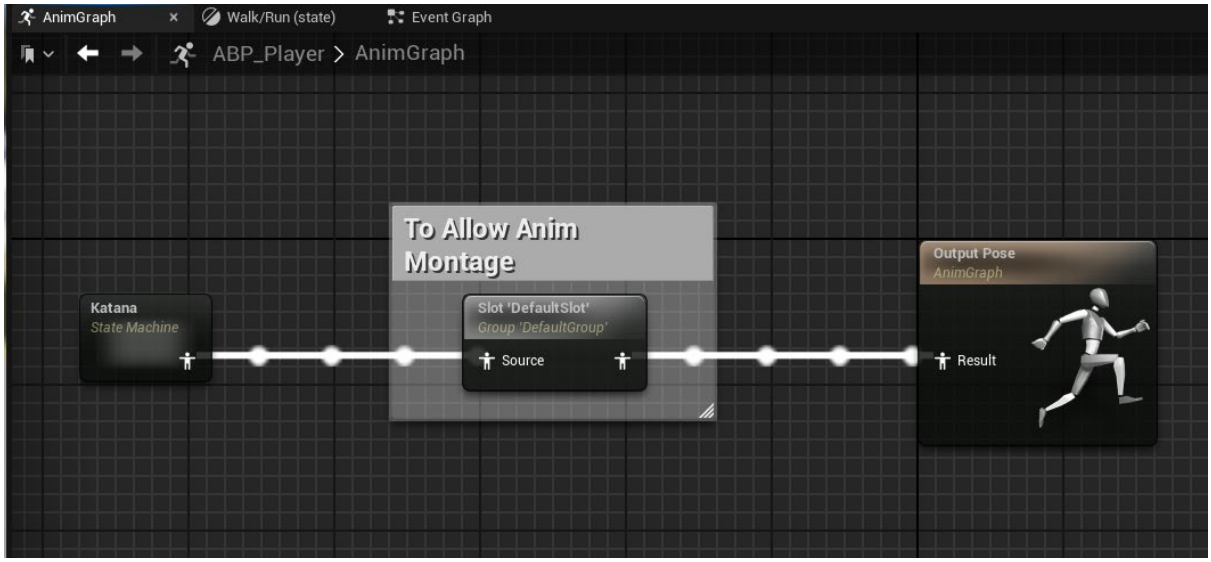


Mit beiden erstellten IK Rigs wurde anschließend ein IK_Retargeter erstellt, der die Animationen des Asset-Pakets auf das neue Mesh retargeted.

ABP_Player (Animation Blueprint)

Nach einigen Experimenten mit dem standardmäßigen Animation Blueprint aus dem Unreal Engine Third Person Pack wurde beschlossen, einen eigenen Animation Blueprint für den Player zu erstellen.

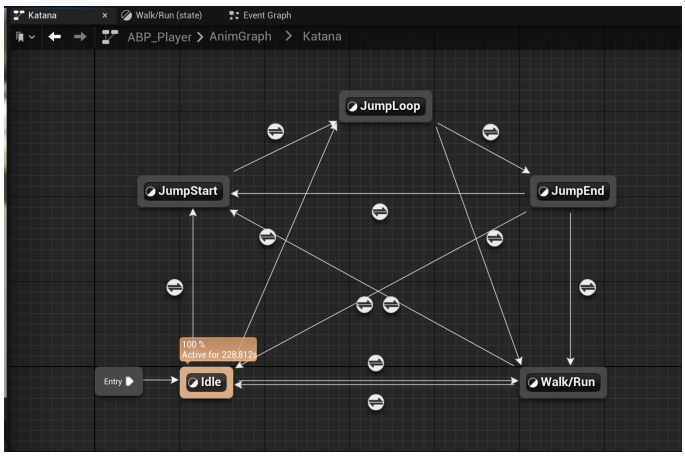
AnimGraph (ABP_Player)



Der AnimGraph besteht im Wesentlichen aus drei Bestandteilen:

1. **Katana StateMachine**: Diese definiert die verschiedenen Zustände des Charakters und gibt den jeweils aktiven Zustand als Output aus.
2. **Default Slot**: Notwendig, um Animation Montages innerhalb des Blueprints des Players zu starten und die den Output der StateMachine zu überschreiben.
3. **Output Pose**: Wendet das Resultat der vorher definierten Animationen auf das Charakter-Skelett an.

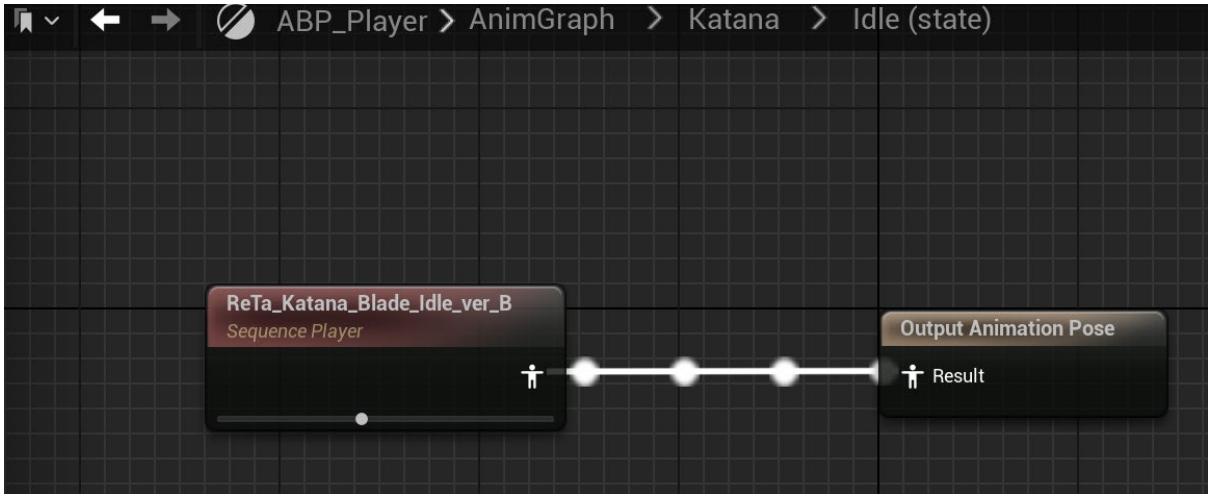
Katana StateMachine (ABP_Player)



Da Sonderaktionen wie das Combat System und Hit Reaktionen innerhalb des BP_ThirdPersonCharacter gehandhabt werden sollen, wird hier nur die

Grundbewegung des Charakters verarbeitet. Diese besteht aus den Zuständen Idle, Walk/Run, JumpStart, JumpLoop und JumpEnd. Dazwischen befinden sich sogenannte Rules, die festlegen, wann und wie zwischen den Animationen gewechselt wird.

Idle State und andere States (ABP_Player)

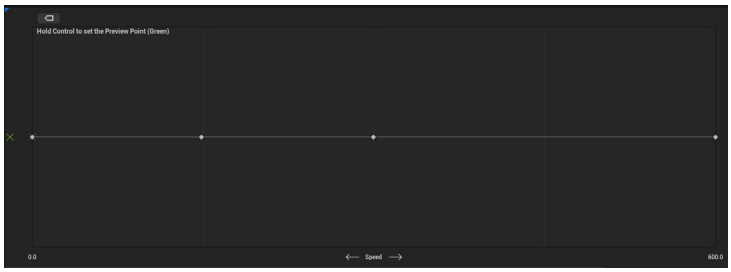


Die meisten Zustände leiten lediglich die festgelegte Animationssequenz in den Output weiter.

Walk/Run State (ABP_Player)

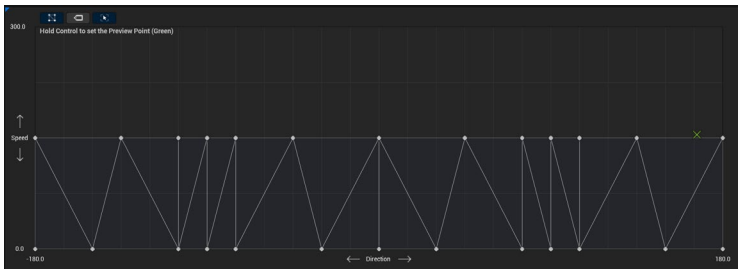
Der Walk/Run State hingegen beinhaltet zwei BlendSpaces, die die zwei Bewegungsmodi auf dem Boden abhandeln. Speed und Direction sind Floats, die die Bewegungsgeschwindigkeit und die Bewegungsrichtung enthalten. "Blend Poses by Bool" entscheidet basierend darauf, ob sich der Player gerade im Target Modus befindet, welcher BlendSpace verwendet wird, und verblendet im Falle eines Wechsels.

BS1D_Player_Walk_Run_Sprint



Nimmt den Speed Input und nutzt diesen Wert, um zwischen Idle, Walk, Run und Sprint zu blenden, basierend auf der Bewegungsgeschwindigkeit. Ein BlendSpace 1D wird verwendet, da lediglich ein Input verwertet werden muss.

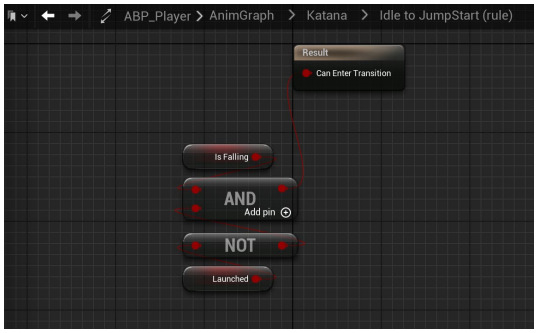
ABS_Player_DirectionalWalk



Nutzt Speed und Direction, um das Animationsverhalten des Players im TargetLock Modus korrekt darzustellen. Speed blendet dabei immer von Idle zur entsprechenden Bewegungsanimation, die durch die Direction definiert wird. Die Direction ist der Wert der Rotationsabweichung der Bewegung in Relation zur Richtung des Targets, also ein Float zwischen -180 und 180. Dabei verwenden -180 und 180 Grad die gleiche Animationssequenz, da beides eine Rückwärtsbewegung darstellt.

Animation Rules

Idle to JumpStart

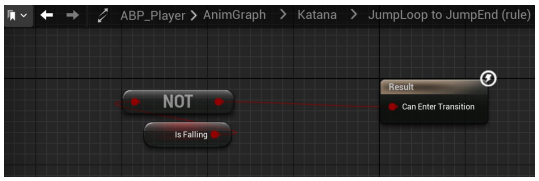


Die JumpStart-Animation wird nur gespielt, wenn der Player gerade keine LaunchAttacke durchgeführt hat oder sich bereits in der Luft befindet und fällt.

JumpStart to JumpLoop

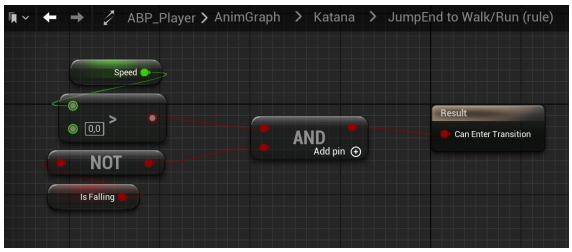
Wechselt nach Ablauf der Animation und benötigt daher keine Sonderregeln.

JumpLoop to JumpEnd



Passiert nur, wenn der Charakter nicht mehr am Fallen ist.

JumpEnd to Walk/Run

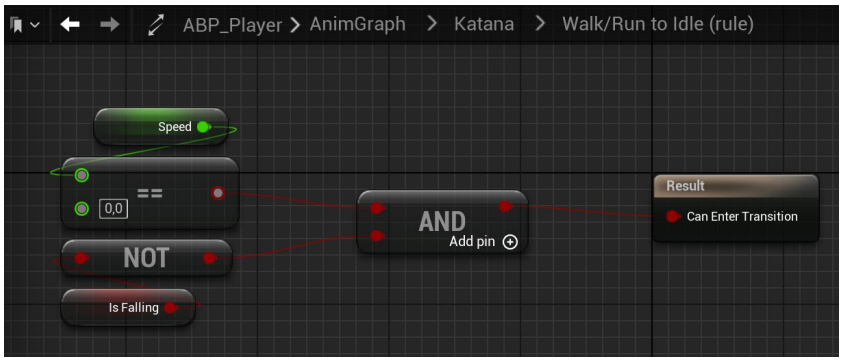


Sorgt dafür, dass die Animation nur dann in den Walk/Run übergeht, wenn der Charakter sich bewegt.

Idle to Walk/Run und JumpLoop to Walk/Run

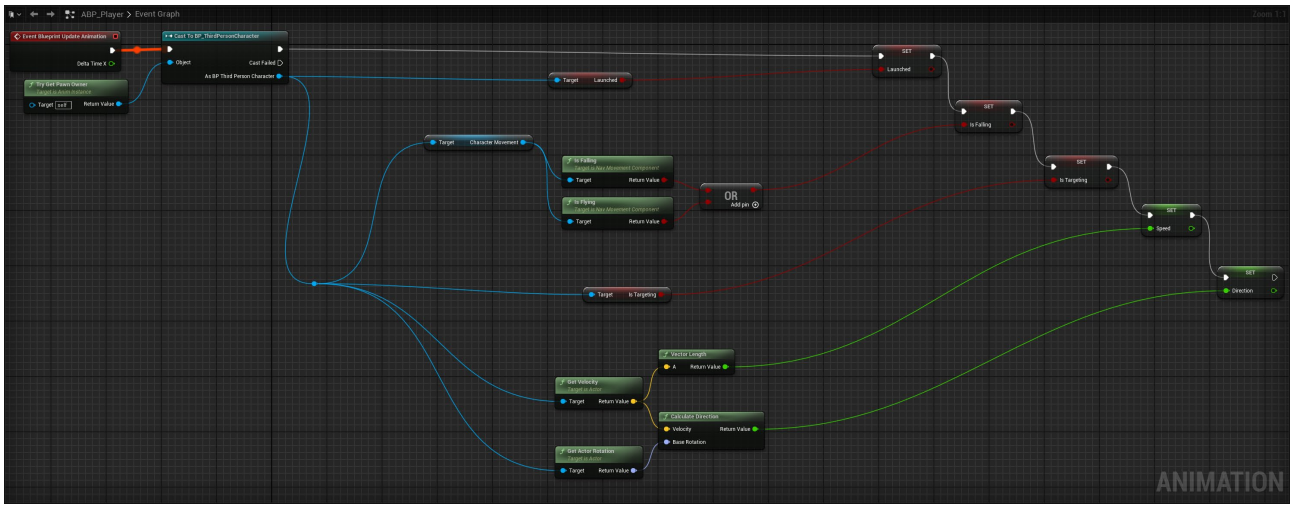
Funktionieren nach den gleichen Regeln wie JumpEnd to Walk/Run, da in allen drei Fällen ein Animation Blend zur Bewegung erfolgen soll.

Walk/Run to Idle und anderen to Idle



Die Rules von anderen Zuständen zur Idle folgen alle derselben Regel, die nur voraussetzt, dass der Player nicht mehr fällt und sich nicht mehr bewegt.

Event Graph



1. **Event Blueprint Update Animation**: Dieser Event wird jedes Frame aufgerufen und übergibt die Delta Time.
2. **Try Get Pawn Owner**: Holt sich den Besitzer des Pawns (Spieler-Charakter).
3. **Cast To BP_ThirdPersonCharacter**: Wandelt den Besitzer in den BP_ThirdPersonCharacter um.
4. **Character Movement**: Überprüft die Bewegungszustände des Charakters:
    - **Is Falling** oder **Is Flying**: Setzt den Zustand "Is Falling".
    - **Launched**: Setzt den Zustand "Launched".
5. **Is Targeting**: Setzt den Zustand "Is Targeting".
6. **Speed und Direction**:
    - Holt sich die Geschwindigkeit (Vector Length).
    - Berechnet die Richtung (Calculate Direction) basierend auf der Geschwindigkeit und der Rotation des Charakters.

Der Event Graph aktualisiert die Animationsvariablen "Is Falling", "Is Targeting", "Speed" und "Direction" basierend auf dem Zustand und der Bewegung des Spieler-Charakters um die Animation Rules und BlendSpaces anwenden zu können.

Character Controller

Der Character Controller basiert auf dem BP_ThirdPersonCharacter Controller aus dem Third Person Starter Pack. Im Laufe des Projekts wurde dieser kontinuierlich erweitert, ergänzt und umgeschrieben.

Zusammenfassung der vorherigen Versionen

Version 1:

Die gesamte Logik des Player Controllers wurde innerhalb des BP_ThirdPersonCharacter Blueprints gehandhabt. Diese Version war sehr Bool-lastig und nutzte keine Arrays, was zu extremer Unübersichtlichkeit des Codes führte. Die Combos funktionierten, bestanden jedoch aus extrem langen Branch-Verkettungen. Das Air Combat war funktional, teleportierte den Gegner jedoch an die Stelle in der Luft, an der sich der Player befand, wenn die Launch Attacke ausgeführt wurde.

Version 2:

Übergang zu einem System mit Arrays für die Combos und Anpassung der Bewegungen innerhalb des Target-Systems.

Version 3:

Eingliederung der Nutzung von Anim Notifys, Notify States und Enums. Zuvor wurde die Logik der Funktionen immer innerhalb des BP_ThirdPersonCharacters eingeleitet und ausgeführt. Mit Hilfe der Anim Notifys und Notify States wurde die Anpassung der Timings wesentlich einfacher und der Aufwand reduziert.

Dokumentierte Version: Version 4

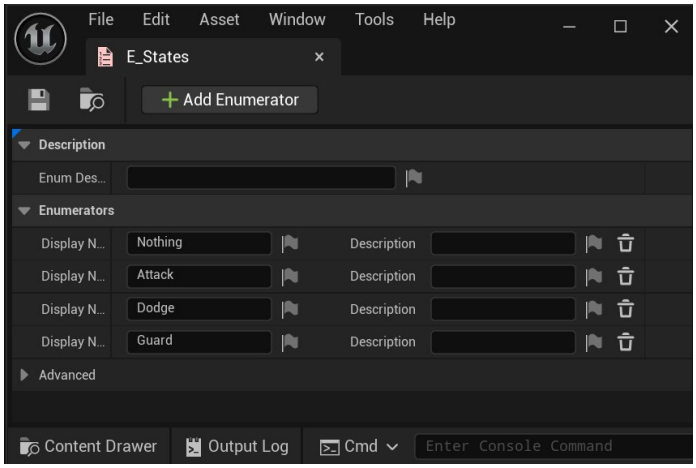
Der hier dokumentierte Player Controller ist die vierte Version. Diese Version enthält die Funktionalität des Players. Die meiste Logik läuft dabei im BP_ThirdPersonCharacter Blueprint ab. Anim Notifys und Notify States werden genutzt, um die Timings der Combos und Aktionen des Charakters besser aufeinander abzustimmen. Dies bietet mehr Flexibilität, um gameplay-relevante Anpassungen in den Combo-Sequenzen vorzunehmen, ohne neuen Code schreiben zu müssen.

Wichtige Komponenten:

- **Character Blueprint**: BP_ThirdPersonCharacter
- **Enumerator**: E_States
- **Animation Notifys**: AN_DownSlash, AN_LaunchCharacter, AN_ResetState, AN_RotationToTarget, AN_SaveDodge, AN_SaveLightAttack
- **Animation States**: ANS_RootMotionRotation, ANS_WeaponCollision
- **Damage System**: BPC_DamageSystem (Blueprint Class) und BPI_DamageSystemInterface (Blueprint Interface)

Zunächst wurde das integrierte Damage System genutzt, welches später durch ein eigenes Damage System ersetzt wurde.

E_States Enumerator

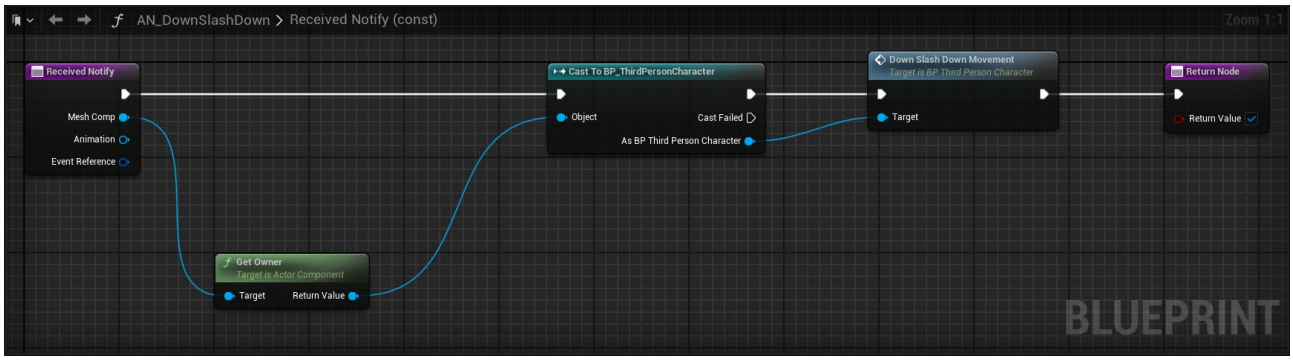


Innerhalb des Character Movements gibt es bereits definierte Movement States des Players. Dieser Enumerator definiert und erweitert die möglichen Zustände des Charakters, einschließlich Nothing, Attack, Dodge und Guard. Dabei werden diese States parallel zu denen um Character Movement genutzt.

Anim Notifys und Anim Notify States

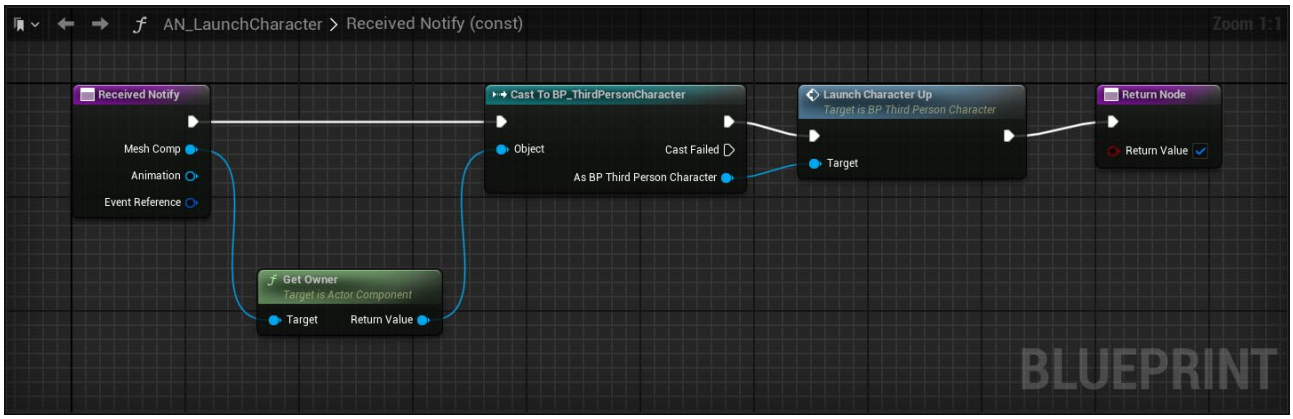
Im Projekt werden Animation Notifys (AN_) und Animation Notify States (ANS_) genutzt, um das Timing der Funktionen, die während Animationen ablaufen müssen, präzise anzupassen. Innerhalb der Anim Notifys wird die Received_Notify Funktion überschrieben, um die Logik an der entsprechenden Stelle in der Animation einzuleiten.

AN_DownSlashDown



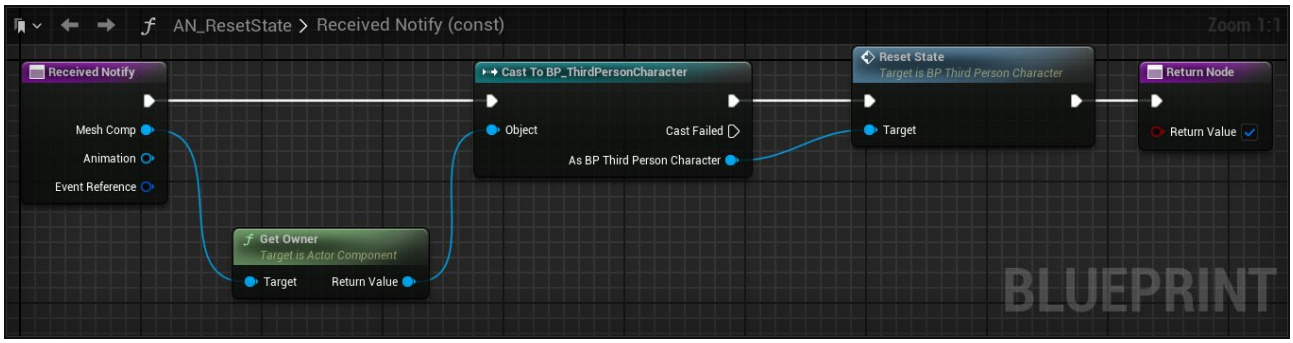
Führt die DownSlashDown Funktion auf dem Charakter aus, um die zugehörige Bewegungslogik anzuwenden.

AN_LaunchCharacter



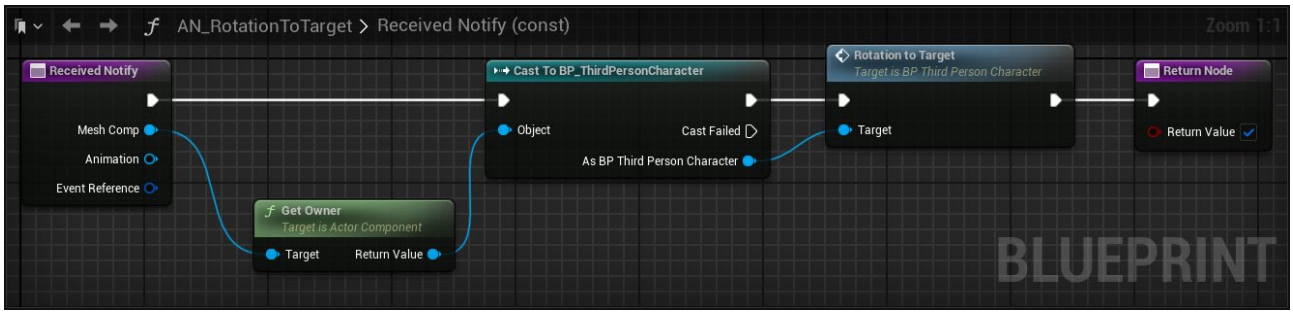
Führt die LaunchCharacterUp Funktion auf dem Charakter aus, um die zugehörige Bewegungslogik anzuwenden.

AN_ResetState



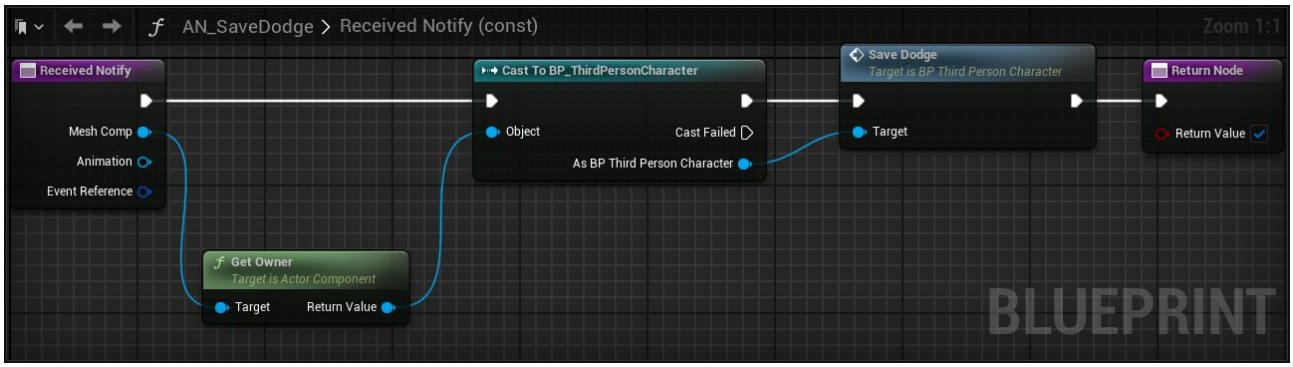
Führt die ResetState Funktion auf dem Charakter aus, um den Zustand des Charakters zurückzusetzen. Wird am Ende der meisten Animationen genutzt um andere Aktionen wieder möglich zu machen.

AN_RotationToTarget



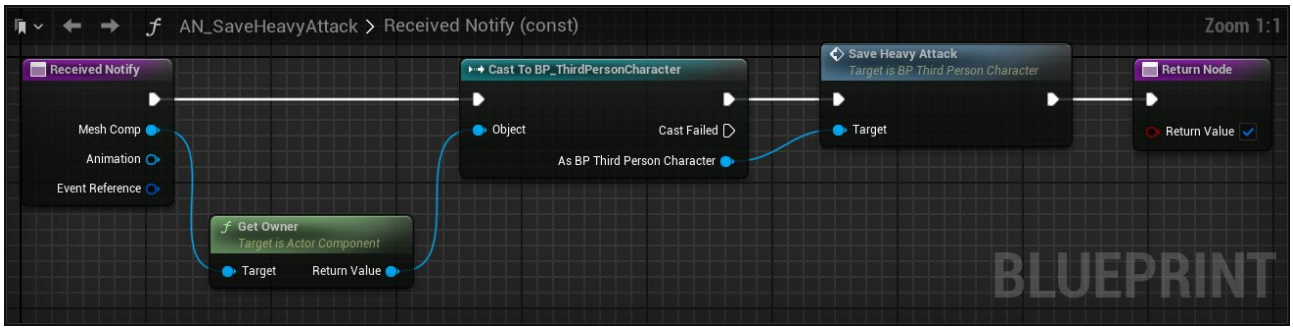
Führt die RotationToTarget Funktion auf dem Charakter aus, um ihn in die Richtung seines Ziels zu drehen. Beispielsweise am Anfang einer Angriffsanimation.

AN_SaveDodge



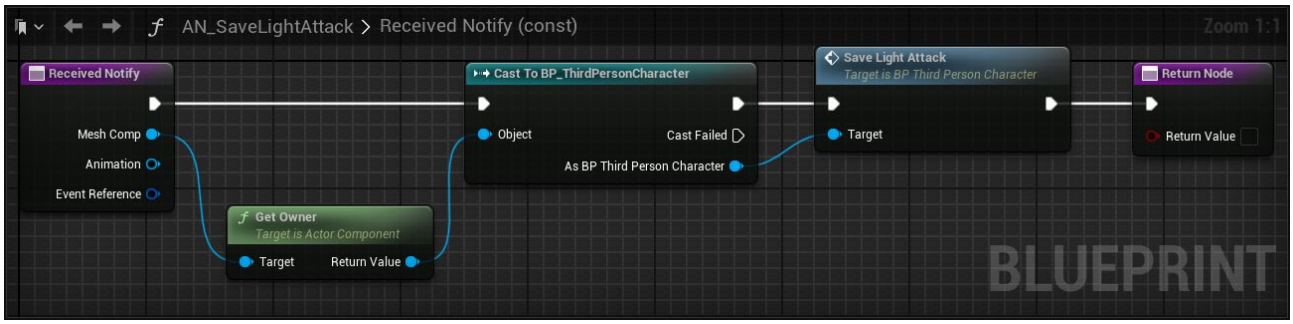
Führt die SaveDodge Funktion auf dem Charakter aus, um den zu speichern das die Ausführung des Dodges zu Ende ist und der nächste Dodge ausgeführt werden kann. Kann auch dazu genutzt werden den Dodge ab einem gewünschten Zeitpunkt innerhalb anderer Animation nutzbar zu machen bevor der State resettet wurde.

AN_SaveHeavyAttack



Führt die SaveHeavyAttack Funktion auf dem Charakter aus, um den zu speichern das die Ausführung der HeavyAttack zu Ende ist und die nächste Heavy Attack eingeleitet werden kann. Kann auch dazu genutzt werden die Heavy Attack ab einem gewünschten Zeitpunkt innerhalb anderer Animation nutzbar zu machen bevor der State resettet wurde.

AN_SaveLightAttack



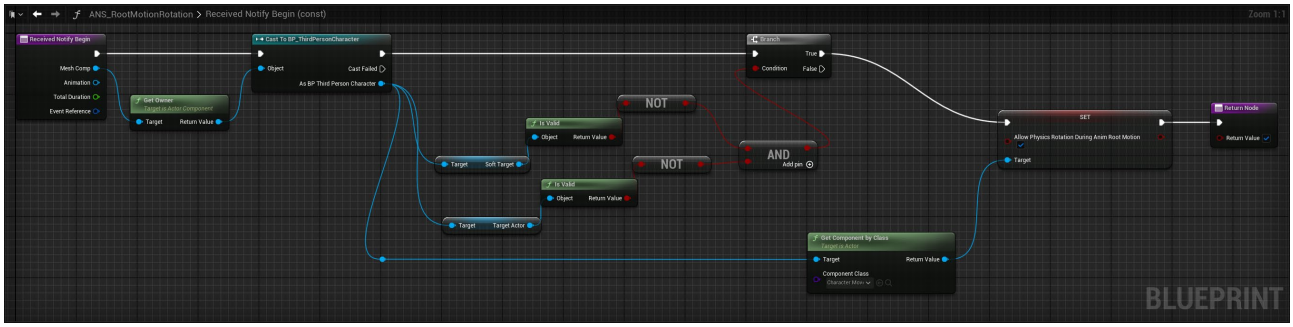
Führt die SaveLightAttack Funktion auf dem Charakter aus, um den zu speichern das die Ausführung der Light Attack zu Ende ist und die nächste Light Attack eingeleitet werden kann. Kann auch dazu genutzt werden die Light Attack ab einem gewünschten Zeitpunkt innerhalb anderer Animation nutzbar zu machen bevor der State Resettet wurde.

Ergänzung zu den AN_Saves

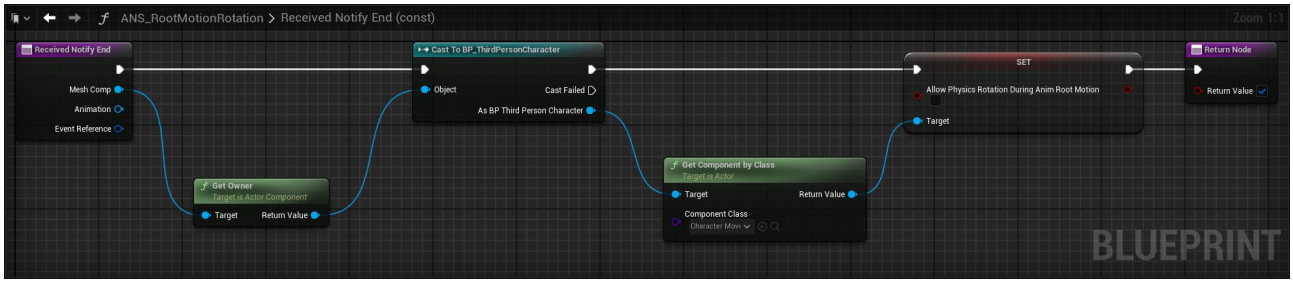
Die AN_Saves setzen schalten zwar die Nutzung der jeweiligen Aktionen Frei unterbinden dabei allerdings die Nutzung der anderen Aktionen die mit AN_Saves gehandhabt werden bis der StateReset ausgeführt wird. Auf diese Weise lassen sich Phasen innerhalb einer Animation bestimmen, in der nur eine Bestimmte Aktion ausgeführt werden kann. Das ist vor allem Hilfreich, um die Schlagabfolgen richtig zu timen und einen dynamischen Ablauf der Aneinanderreihungen der Animation zu erzeugen.

ANS_RootMotionRotation

Received_Notify_Begin



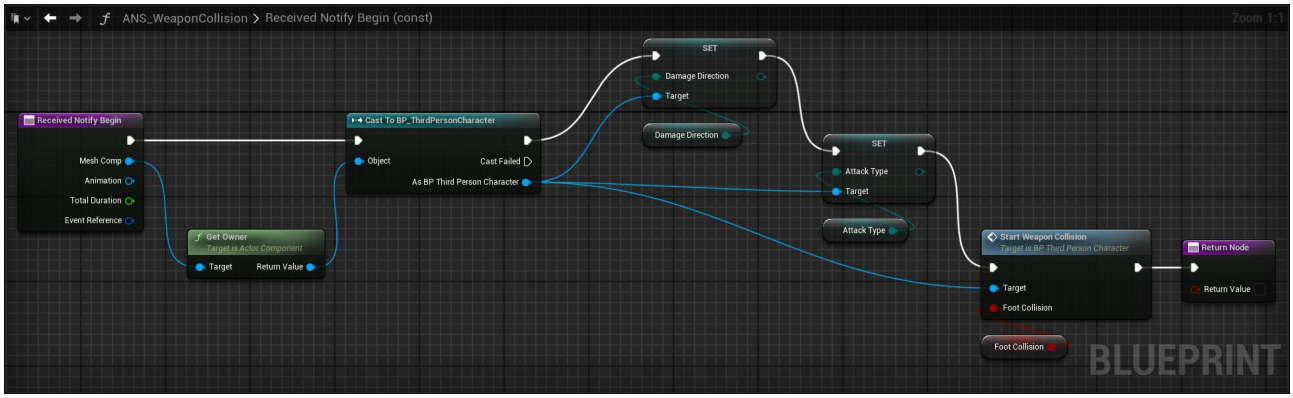
Received_Notify_End



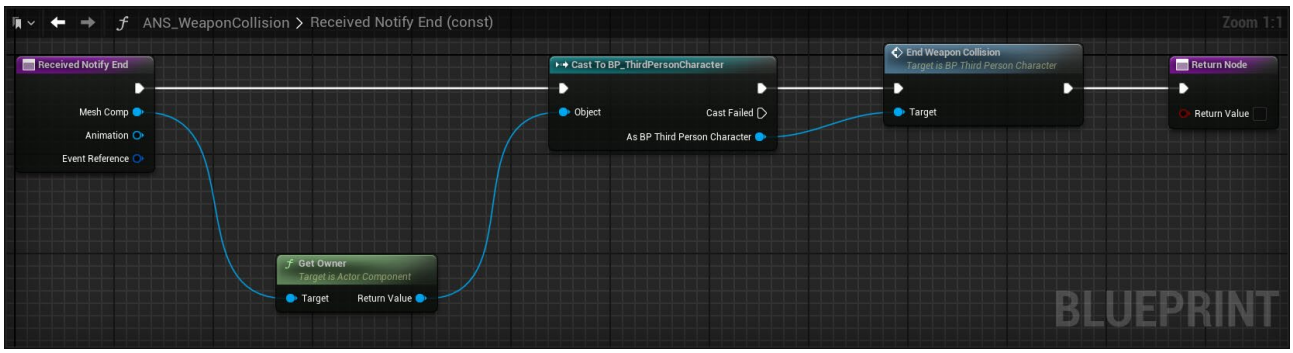
Der ANS_RootMotionRotation AnimNotifyState sorgt dafür, dass der Spieler während eines bestimmten Zeitraums der Animation die Kontrolle über die Rotation des Charakters steuern, vorausgesetzt, er befindet sich nicht im TargetLock-Modus und hat keine Soft Target-Ausrichtung. Hierzu wird innerhalb des Zeitraums des States ganz einfach die Rotation des Character Movements während einer RootMotion Bewegung erlaubt.

ANS_WeaponCollision

Received_NotifyBegin



Received_NotifyEnd

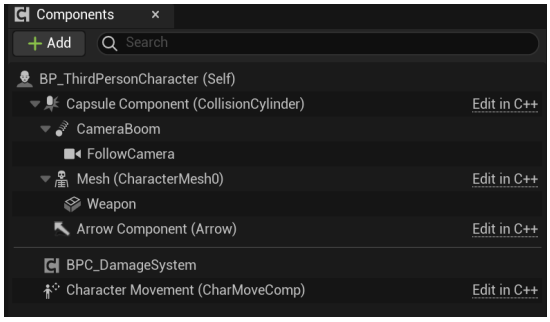


Der ANS_WeaponCollision AnimNotifyState sorgt dafür, dass während eines bestimmten Zeitraums der Animation die Kollisionsabfragen für die Waffe des Charakters gestartet und nachfolgend wieder gestoppt werden. Ausserdem setzt er die entsprechenden Variablen der Enums im BP_ThirdPersonCharacter und gibt weiter, ob es sich um eine FootCollision handelt. Beides wird dort für die Verarbeitung der Attacken gebraucht

BP_ThirdPersonCharacter

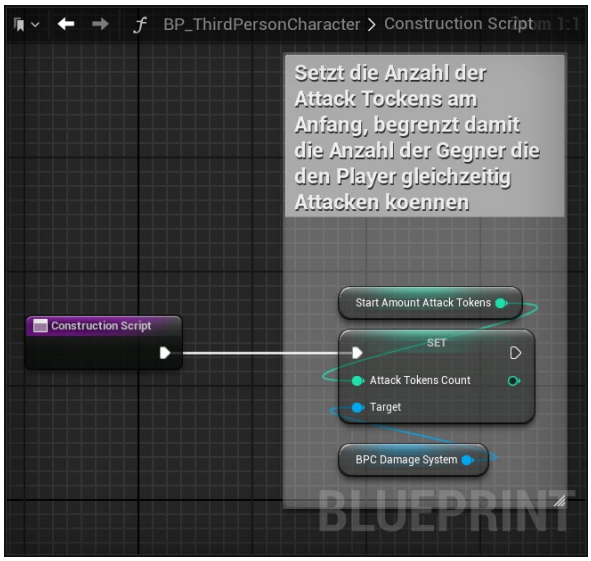
Innerhalb dieses Blueprints der Character Blueprint Class läuft die gesamte Logik des Players ab. Um die Eingaben des Players umzusetzen, wird das Enhanced Input System der Unreal Engine 5.3 genutzt, da dies eine flexible Anpassung der Eingaben im Verlauf des Projektes ermöglicht.

Die Basis hierfür bildet der BP_ThirdPersonCharacter, der mit dem Unreal Engine Third Person Paket geliefert wird.



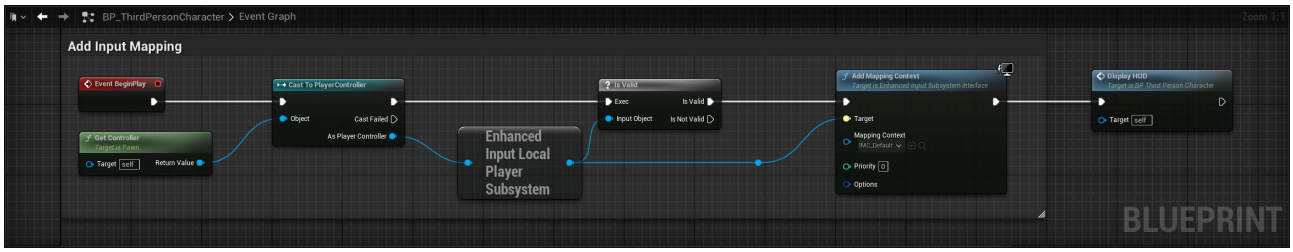
Dieser Blueprint bringt bereits einige benötigte Komponenten für den Player Character mit. Ergänzt werden diese durch die Weapon (Static Mesh Component) und das BPC_DamageSystem.

Construction Script



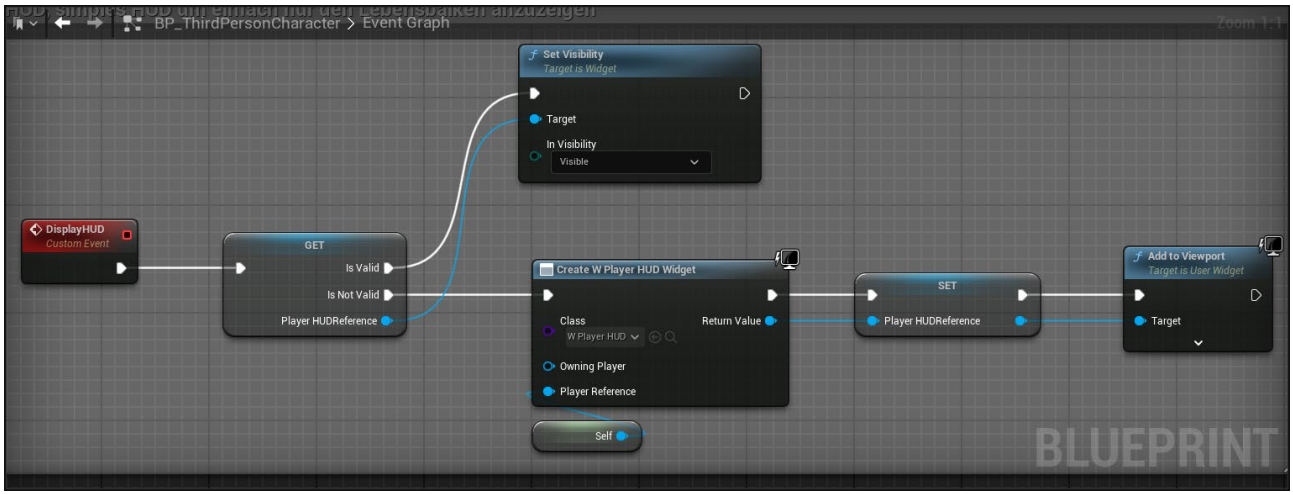
Dieses Setup stellt sicher, dass zu Beginn des Spiels die maximale Anzahl von Gegnern, die den Spieler gleichzeitig angreifen können, begrenzt ist. Es hilft dabei, das Gameplay ausgeglichen zu gestalten und zu verhindern, dass der Spieler von zu vielen Gegnern gleichzeitig überwältigt wird.

Event BeginPlay



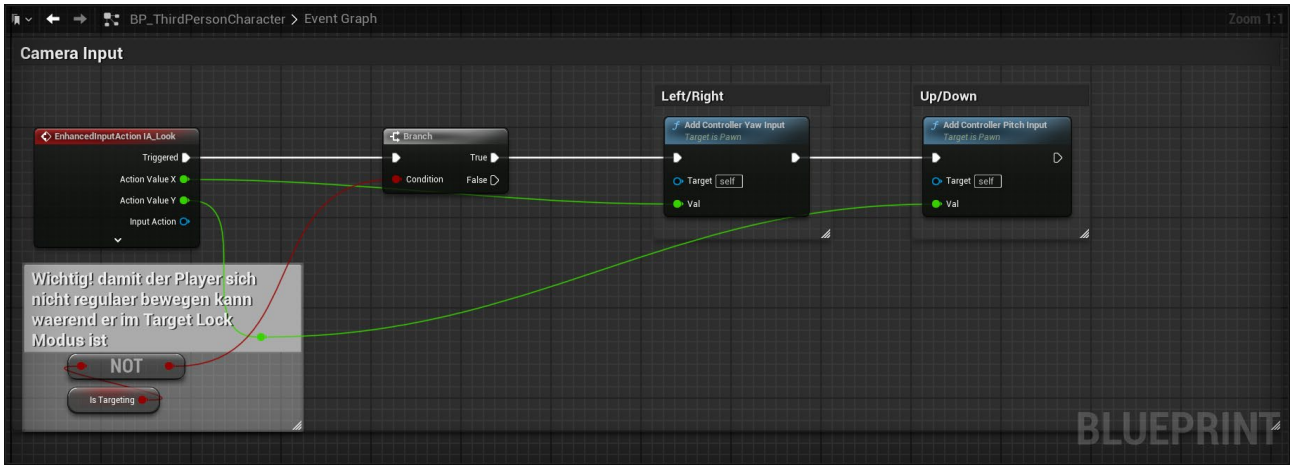
Setzt den Eingabekontext für den Charakter, wenn das Spiel beginnt. Es stellt sicher, dass der richtige Player Controller verwendet wird, überprüft die Gültigkeit des Eingabesubsystems und fügt den entsprechenden Eingabekontext hinzu. Abschließend wird das HUD des Charakters angezeigt. Bis auf die Display HUD Funktion, war dieser Abschnitt bereits im BP_ThirdPersonCharacter enthalten.

Event DisplayHUD



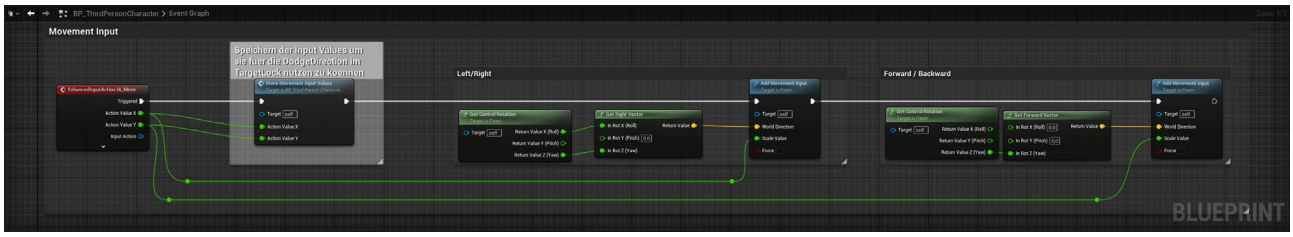
Dieses Event stellt sicher, dass das Spieler-HUD korrekt angezeigt wird. Wenn das HUD bereits existiert, wird nur die Sichtbarkeit aktualisiert. Wenn es noch nicht existiert, wird ein neues Widget erstellt und dem Viewport hinzugefügt.

Event CameraInput



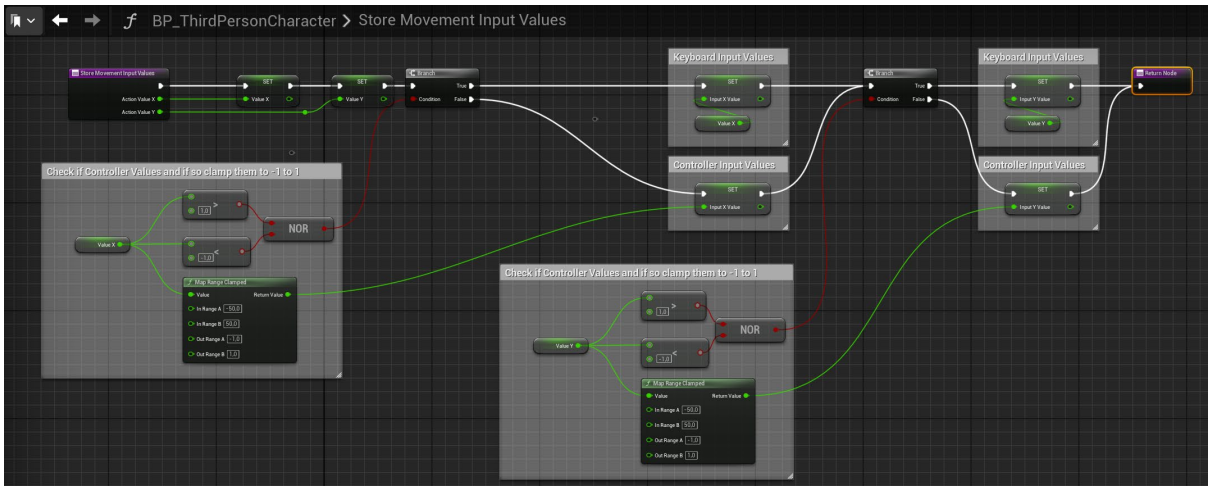
Das Camera Input Event im Event Graph des BP_ThirdPersonCharacter ist dafür verantwortlich, die Kameraeingaben des Spielers zu verarbeiten und entsprechend die Blickrichtung des Charakters zu ändern. Dieses Event war so bereits Teil des BP_ThirdPersonCharacter und wurde durch den Branch ergänzt. Der Branch stellt sicher, dass die Kameraeingaben des Spielers nur dann verarbeitet werden, wenn der Spieler nicht im Target-Lock-Modus ist.

Event MovementInput



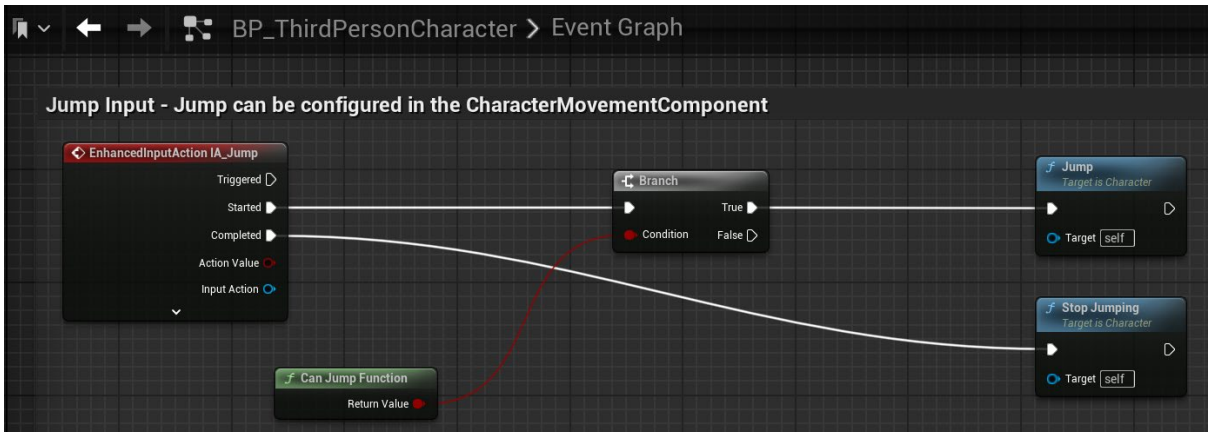
Das Movement Input Event im Event Graph des BP_ThirdPersonCharacter ist dafür verantwortlich, die Bewegungseingaben des Spielers zu verarbeiten und den Charakter entsprechend zu bewegen. Dieses Event kam schon vorneweg mit dem BP_ThirdPersonCharacter. Wurde allerdings um das Abrufen der StoreMovementInputValues Funktion ergänzt.

Funktion StoreMovementInputValues



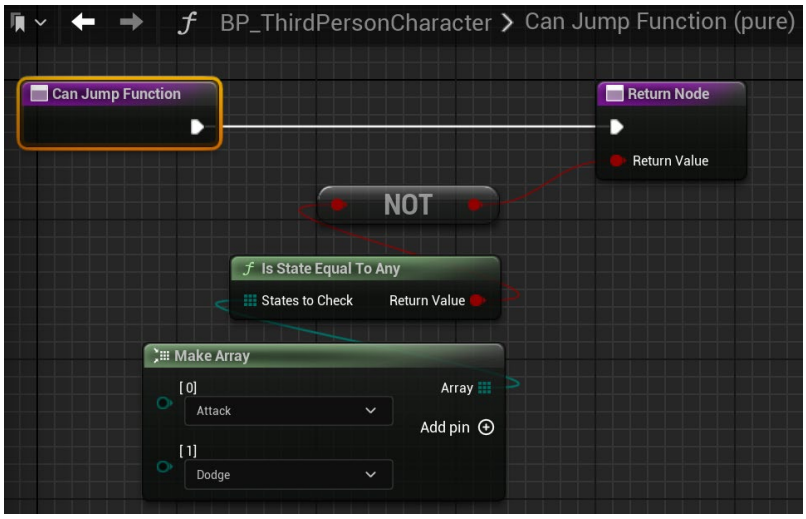
Die Funktion speichert die Bewegungseingabewerte zur späteren Verwendung, beispielsweise für die Dodge-Richtung im Target-Lock-Modus. Indem es die Eingabewerte speichert und diese dann differenziert zwischen einer Tastatur oder Controller Input Wert verarbeitet und falls notwendige auf einen Bereich zwischen -1 und 1 begrenzt und Als Input Values Y und X speichert. Mithilfe der gespeicherten Values kann anderen Ortes eine Richtungstest ausgeführt werden der die Eingabe Richtung auf eine von 4 Richtungen beschränkt.

Event JumpInput



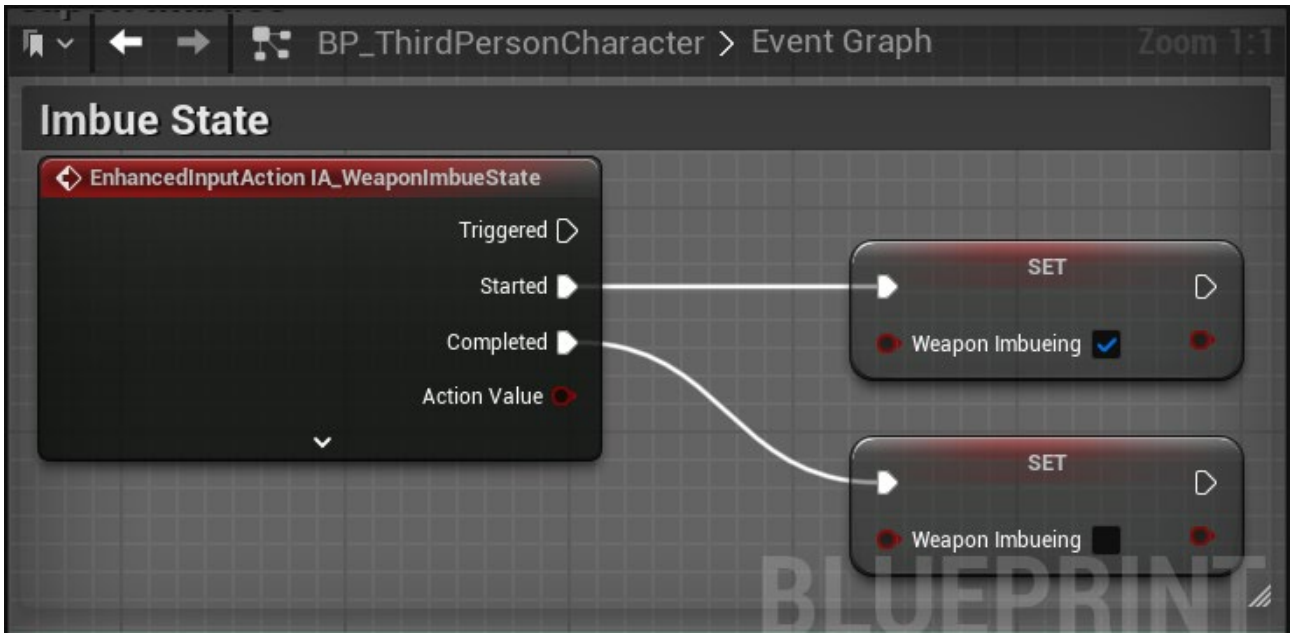
Das Jump Input Event im Event Graph des BP_ThirdPersonCharacter ist dafür verantwortlich, die Sprungeingaben des Spielers zu verarbeiten und den Charakter entsprechend springen zu lassen. Dieses Event war so bereits Teil des BP_ThirdPersonCharacter und wurde durch den Branch ergänzt. Der Branch stellt sicher das der Charakter sich in einem Zustand befindet, in dem er springen kann.

Funktion CanJumpFunction (pure)



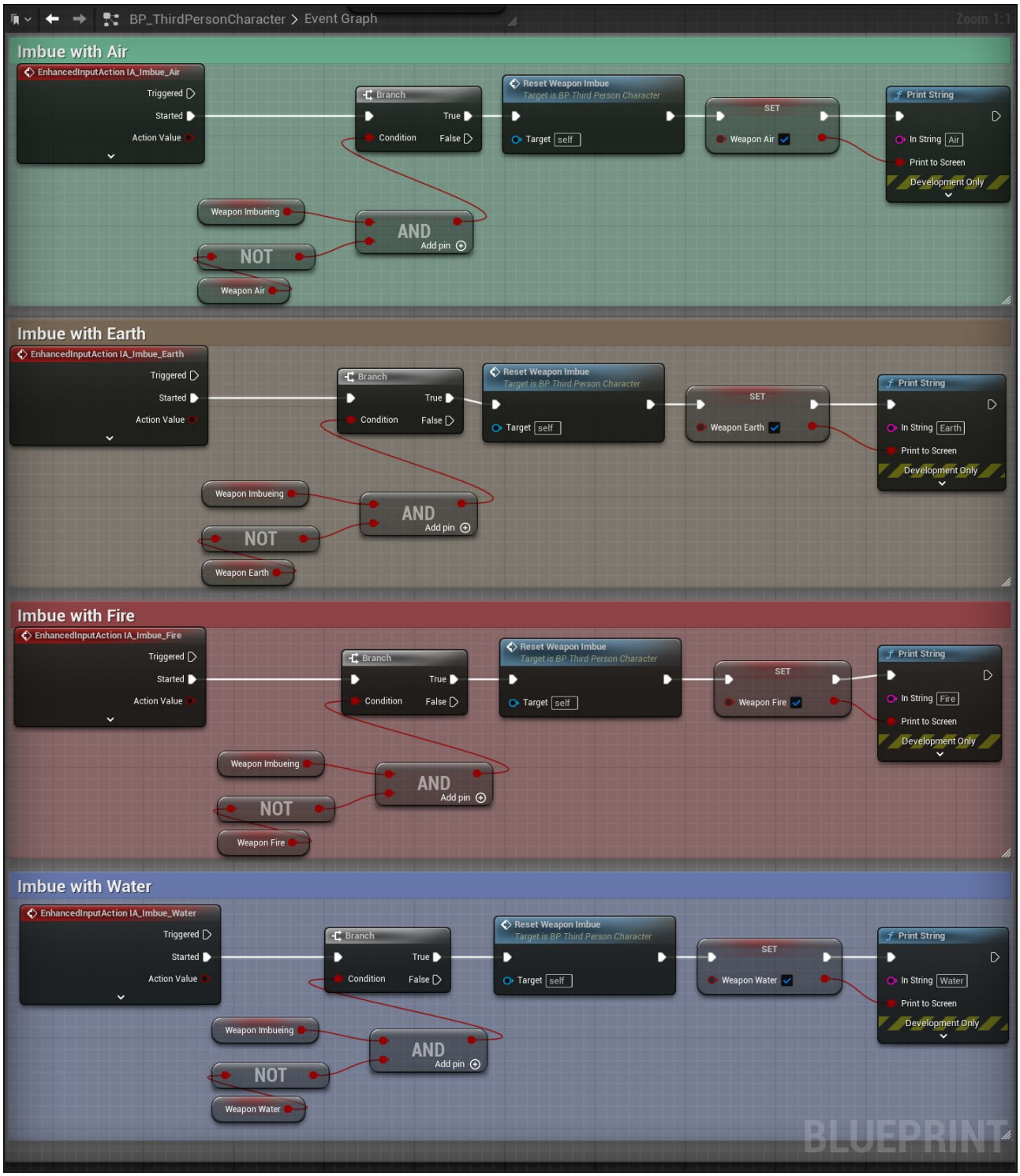
Diese Funktion stellt sicher, dass der Spieler nur dann springen kann, wenn er sich nicht im Zustand "Attack" oder "Dodge" befindet.

Event ImbueState



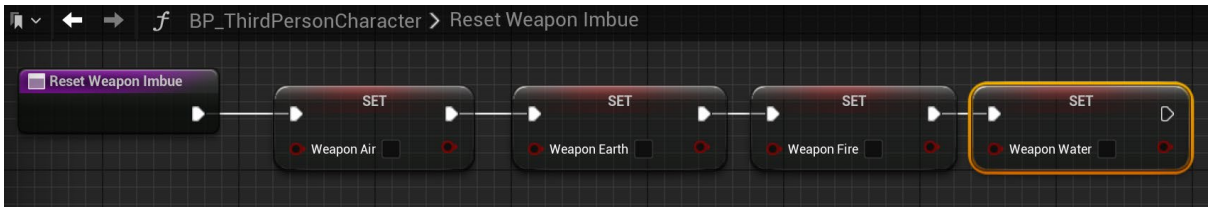
Dieses Event stellt sicher das der Spieler nur dann den Elementartyps seiner Attacken ändern kann, während er den WeaponImbueState Input gedrückt hält indem es währenddessen einen Boolean setzt.

Events ImbueWithElement



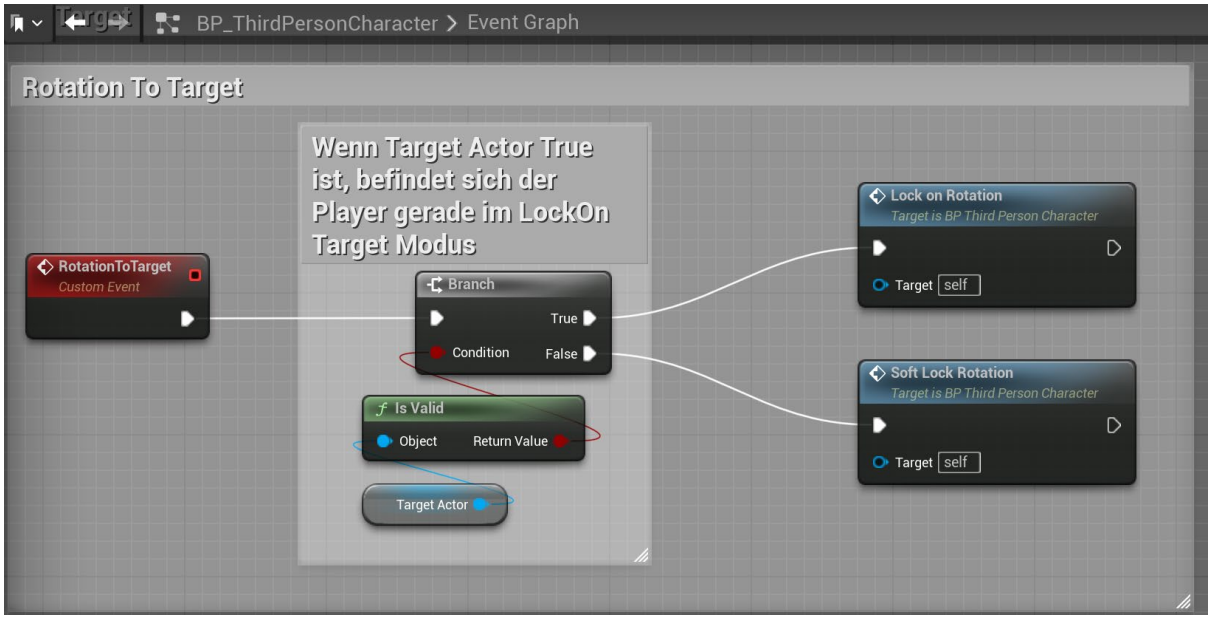
Diese Events ermöglichen es dem Spieler, die Waffe des Charakters mit verschiedenen Elementen zu infundieren, was wiederum die ausgeführten Light Attacks ändert. Jedes Event überprüft, ob der Spieler im WeaponImbueing State ist und nicht schon dasselbe Element aktiv ist, setzt gegebenenfalls die vorherigen Zustände zurück und aktiviert das gewünschte Element.

Funktion ResetWeaponImbue



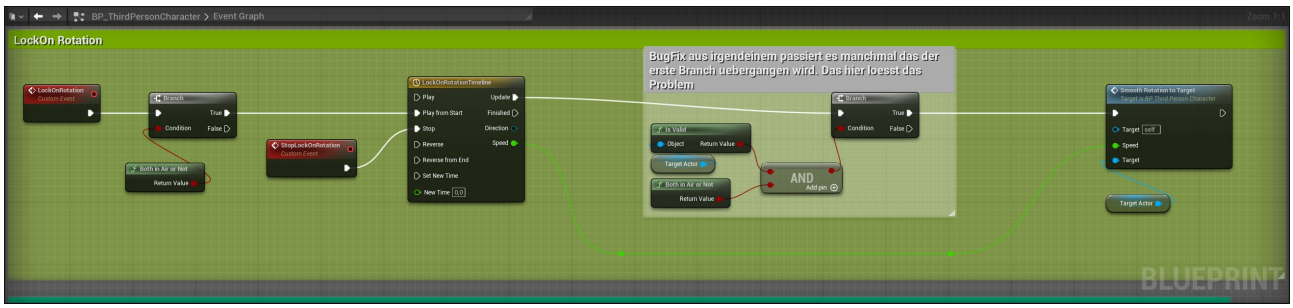
Diese Funktion sorgt dafür, dass alle Elementar-Imbuing-Zustände der Waffe auf false gesetzt werden. Dies ist ein notwendiger Schritt, bevor eine neue Infusion durchgeführt wird, um sicherzustellen, dass die Waffe nicht gleichzeitig mit mehreren Elementen infundiert ist.

Event RotationToTarget



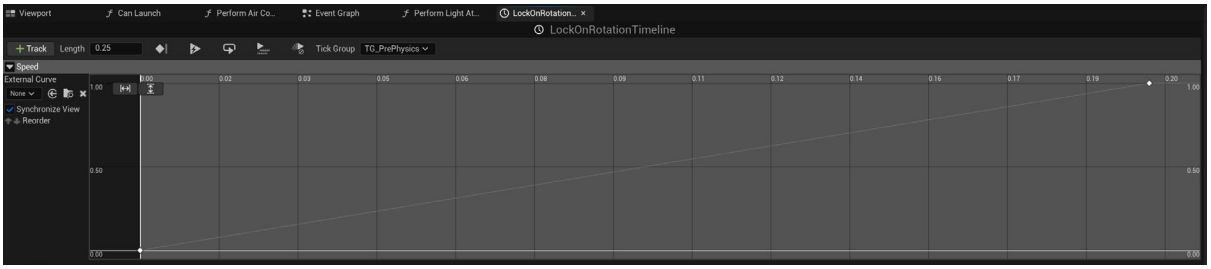
Dieses Event stellt sicher, dass der Charakter sich korrekt in Richtung seines Ziels ausrichtet, abhängig davon, ob er sich im LockOn-Modus oder im Soft Lock Modus befindet.

Event LockOnRotation



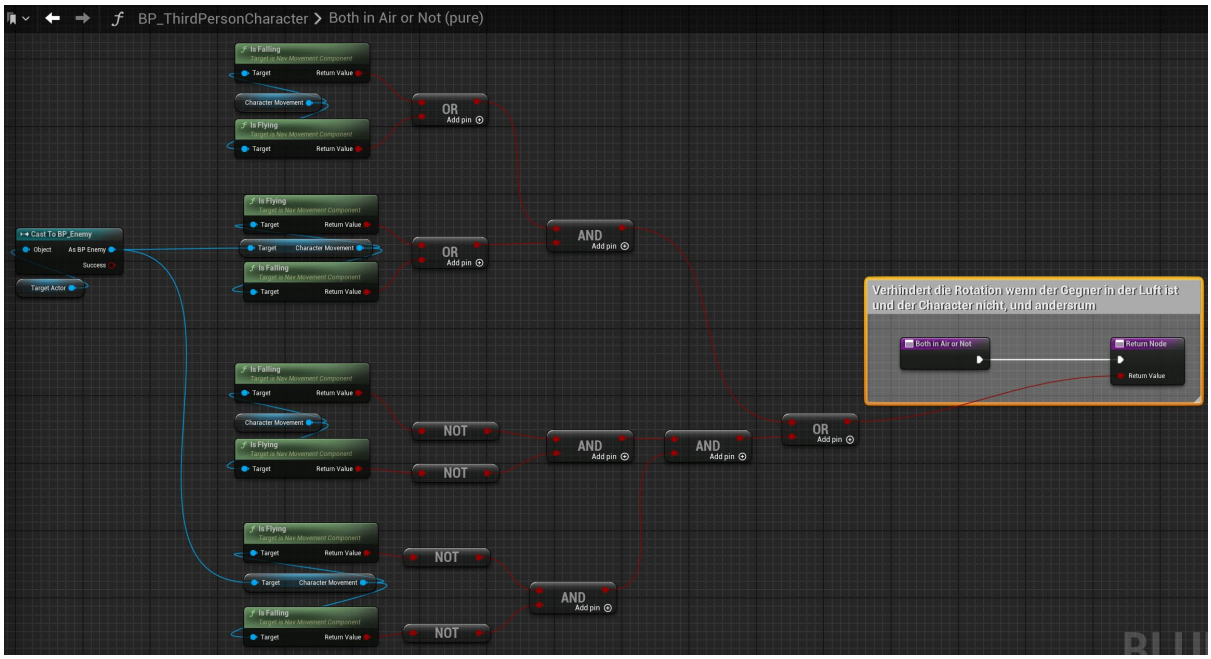
Das Event sorgt dafür, dass der Charakter im Lock-On-Modus gleichmäßig zum Ziel rotiert, solange beide Akteure entweder in der Luft sind oder nicht, und stellt sicher, dass alle Bedingungen für den Lock-On-Modus erfüllt sind. Wenn die Bedingungen nicht erfüllt sind, wird die Rotation gestoppt.

Ablauf:

1. **Custom Event LockOnRotation**: Startet die Funktion.
2. **Branch-Node**: Überprüft mit der Funktion Both in Air or Not, ob beide Akteure (Spieler und Ziel) entweder in der Luft sind oder nicht. Wenn die Bedingung nicht erfüllt ist, wird die LockOnRotationTimeline gestoppt und StopLockOnRotation aufgerufen.
3. **LockOnRotationTimeline**:
   

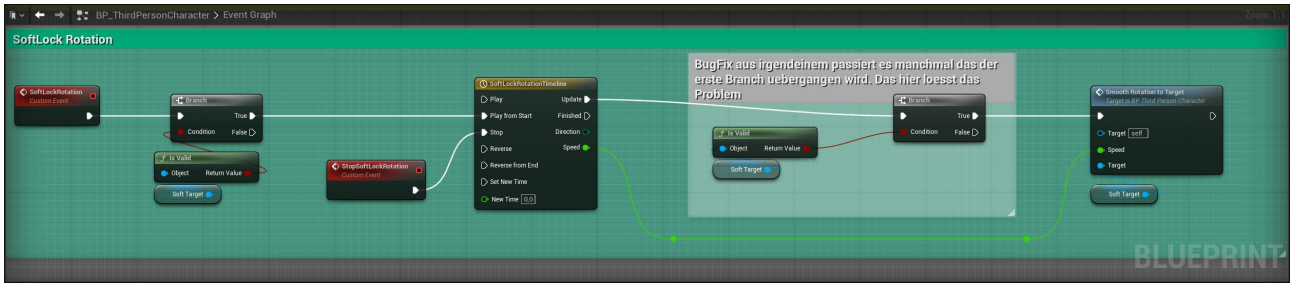
   Eine Timeline, die für die Dauer der Rotation verantwortlich ist. Sie sorgt dafür, dass die Rotation des Charakters im Lock-On Modus über eine Dauer von 0.25 Sekunden glatt und gleichmäßig verläuft.
4. **StopLockOnRotation (Custom Event)**: Führt zum Stoppen der LockOnRotationTimeline.
5. **Bugfix-Branch**: Ein weiterer Branch, um sicherzustellen, dass der Lock-On-Zustand korrekt überprüft wird. Dies ist ein Fix für ein gelegentliches Problem, bei dem der erste Branch übersprungen wird.
6. **AND-Node**: Überprüft, ob das Zielobjekt gültig ist und ob beide Akteure in der Luft sind oder nicht.
7. **Smooth Rotation to Target**: Diese Funktion führt die eigentliche Rotation des Charakters zum Ziel durch. Die Rotation erfolgt gleichmäßig und berücksichtigt die aktuelle Geschwindigkeit.

Funktion BothInAirOrNot



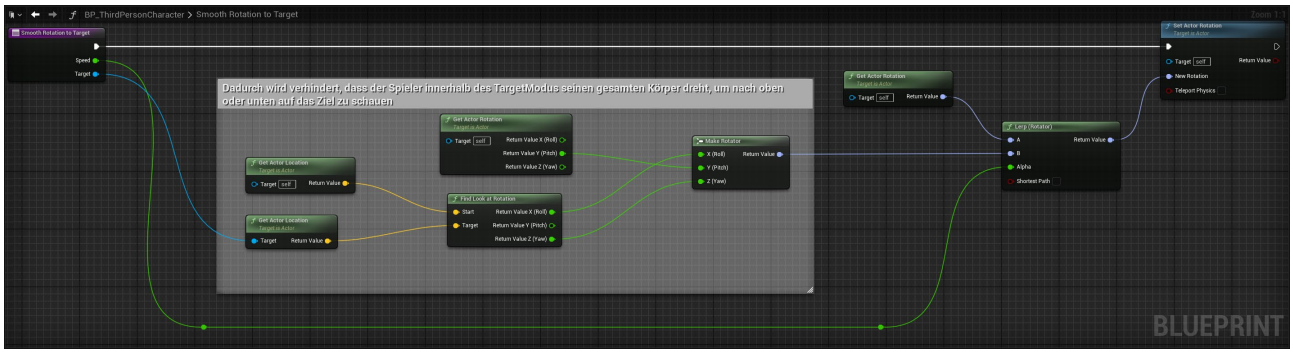
Die Funktion stellt sicher, dass der Spielercharakter und das Ziel entweder beide gleichzeitig in der Luft sind oder keiner von beiden, indem es den Movement Mode beider abfragt und IsFalling und IsFlying aus beiden miteinander abgleicht. True, wenn beide entweder in der Luft oder nicht in der Luft sind; False, wenn einer von beiden in der Luft ist und der andere nicht.

Event SoftLockOnRotation



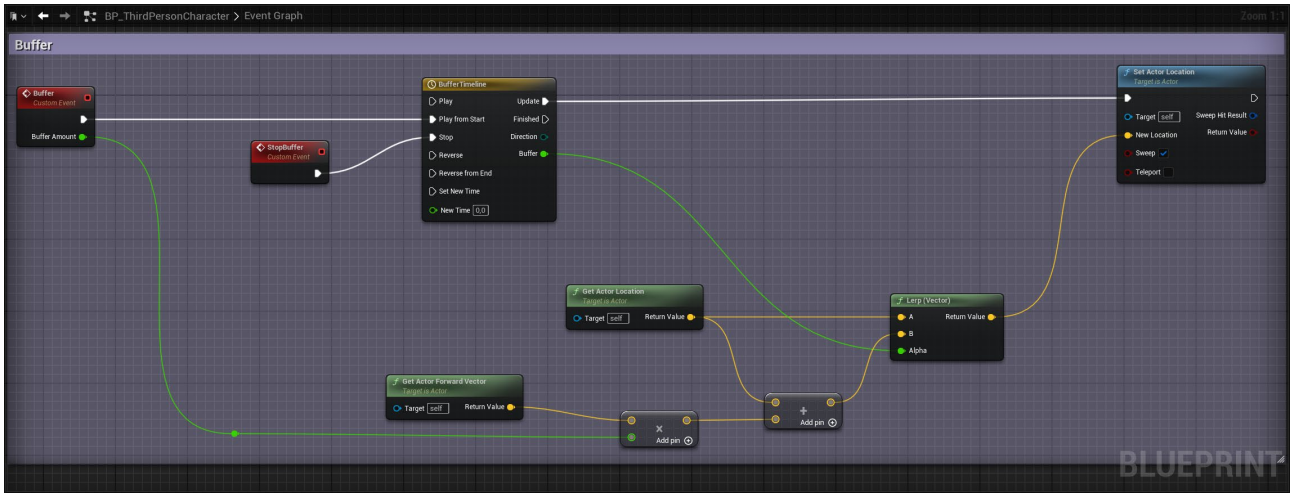
Die Funktion steuert die Rotation des Charakters, damit er sich auf das letzte getroffene Ziel konzentriert, auch wenn der TargetLock-Modus nicht aktiviert ist. Sie gleicht in vielen Punkten der LockOnRotation. Die Branches testen hier allerdings ob es ein Last Target gibt und lassen das Event nur weiterlaufen wenn dies jeweils der Fall ist. Die neue Timeline besitzt die gleichen Werte wie die der LockOnRotation. Das Event, das die Timeline stoppt, wurde durch ein neues StopSoftLockRotation ersetzt.

Funktion SmoothRotationToTarget



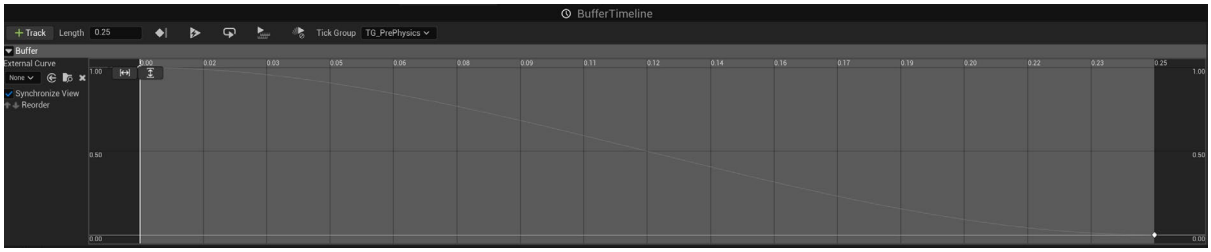
Die Funktion sorgt dafür, dass der Charakter sich flüssig in Richtung seines Ziels dreht. Dabei wird die aktuelle Rotation des Charakters beibehalten und nur die notwendige Yaw-Rotation geändert. Durch die Interpolation wird eine sanfte Rotationsbewegung gewährleistet.

Event Buffer



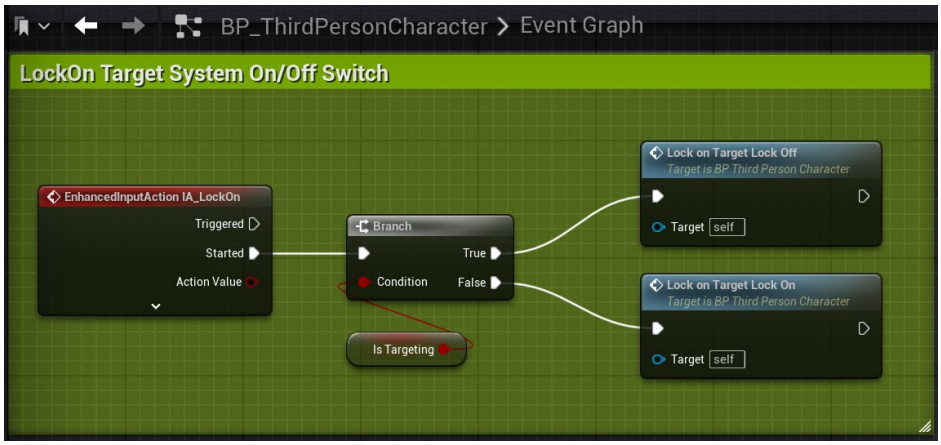
Das Buffer Event im BP_ThirdPersonCharacter ermöglicht eine sanfte Vorwärtsbewegung des Charakters über eine definierte Zeitspanne. Dies wird durch die Verwendung einer Timeline und der Interpolation der Position erreicht.

BufferTimeline



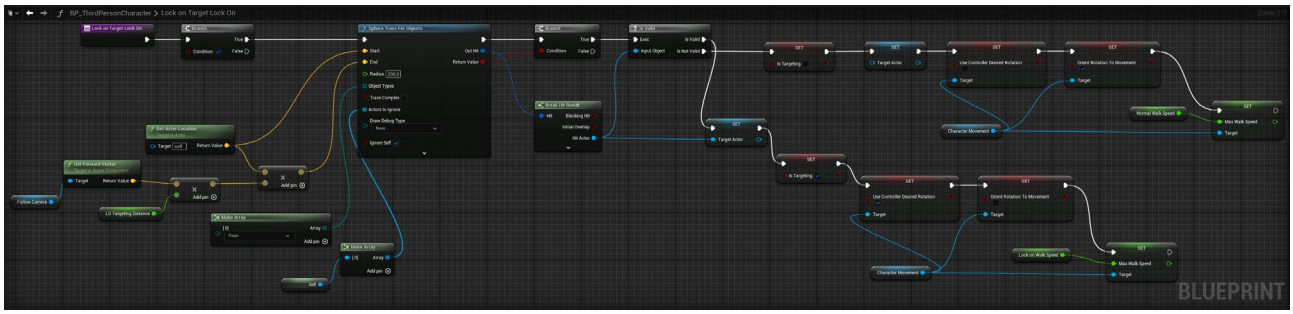
Die Timeline sorgt dafür, dass der Charakter über eine festgelegte Zeitspanne von 0.25 Sekunden vorwärtsbewegt wird. Die Bewegung beginnt schnell und verlangsamt sich allmählich, bis sie stoppt, während die Set Actor Location Node die neue Position des Charakters festlegt.

Event LockOn On/Off Switch



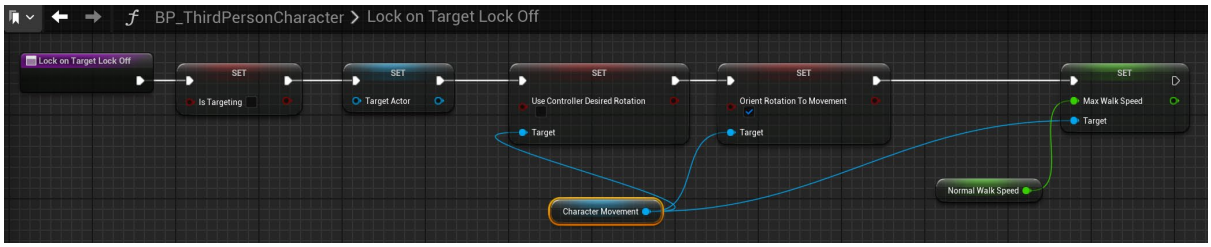
Dieses Event schaltet den Lock-On Modus des Charakters um, basierend auf der aktuellen Eingabe des Spielers. Wenn der Player bereits einen Gegner anvisiert, wird der Lock-On Modus ausgeschaltet, andernfalls wird er eingeschaltet.

Funktion LockOnTargetLockOn



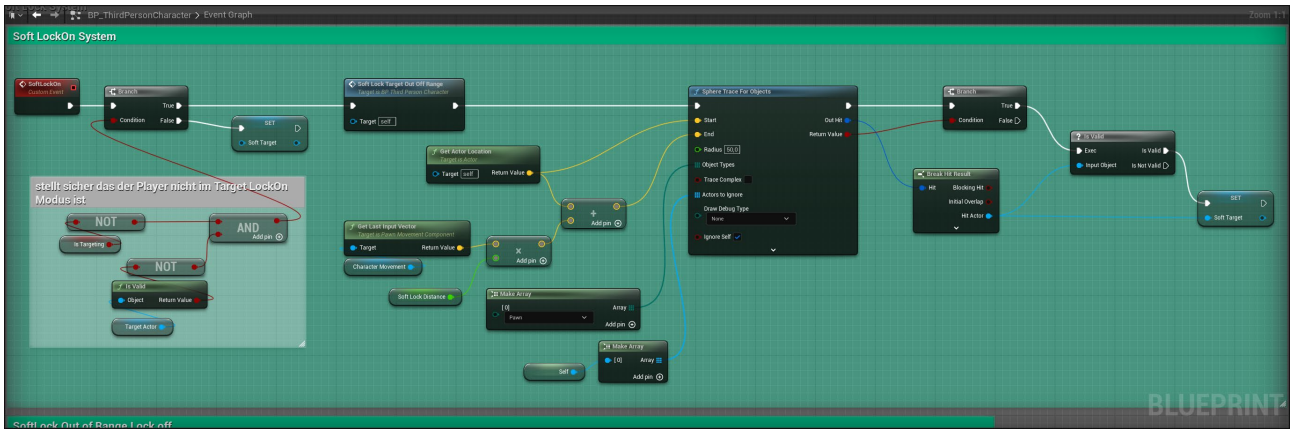
Diese Funktion nutzt einen Sphere Trace auf Basis der Position des Spielers und der Ausrichtung der Camera um einen Gegner (Pawn) zu finden der nicht er selbst ist. Dabei geht das Trace von dem Spieler aus in Blickrichtung. Wird ein Pawn getroffen und der Pawn enthält einen Actor handelt es sich um einen Gegner und er wird als Target Actor gesetzt und der Player geht in den LockOn Modus über, setzt isTargeting auf true und passt seine Bewegungsgeschwindigkeit und Orientierung entsprechend an. Ist der Actor Not Valid wird der LockOn Modus aufgelöst.

Funktion LockOnTargetLockOff



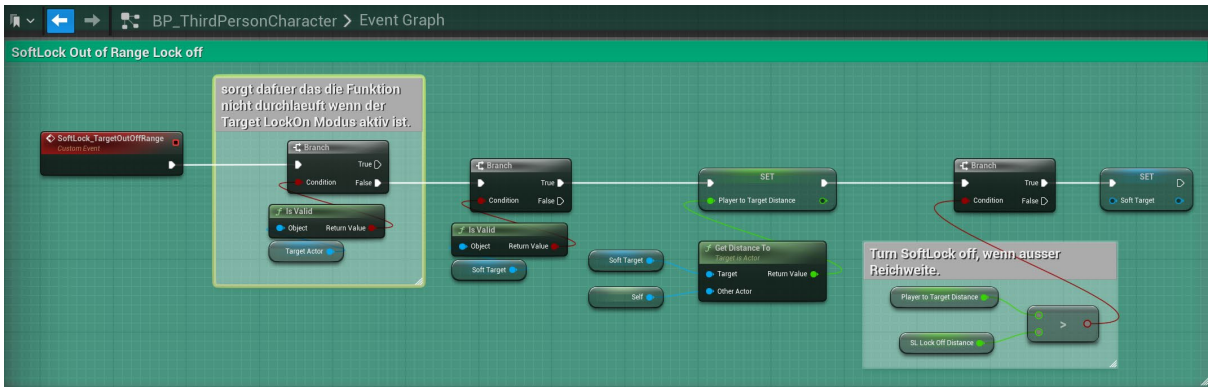
Diese Funktion deaktiviert den Lock-On Modus des Charakters und setzt die entsprechenden Variablen und Einstellungen zurück, um sicherzustellen, dass der Charakter sich wieder normal bewegen und rotieren kann.

Event SoftLockOn



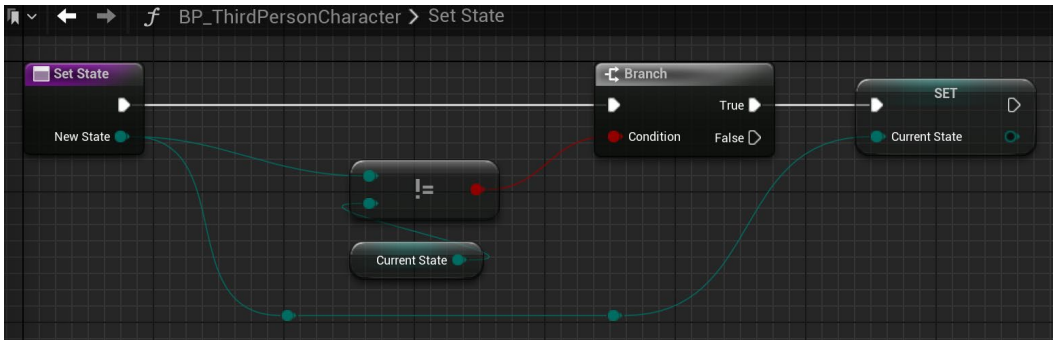
Dieses Event ermöglicht es dem Charakter, sich auf nahegelegene Ziele zu fokussieren, indem es die Position des Charakters und die letzte Eingaberichtung verwendet, um eine kugelförmige Abtastung durchzuführen und ein geeignetes Ziel zu finden, sofern sich der Charakter nicht im Target LockOn Modus befindet. Das wird Beispielweise verwendet, um den Player bei aufeinanderfolgenden Attacken auf den Enemy auszurichten. Oder um dynamisch in Gruppenkämpfen das Ziel zu wechseln, indem man nach einer Attacke die Steuerungsrichtung in die Richtung eines neuen Gegners ändert.

Event SoftLock_TargetOutOffRange



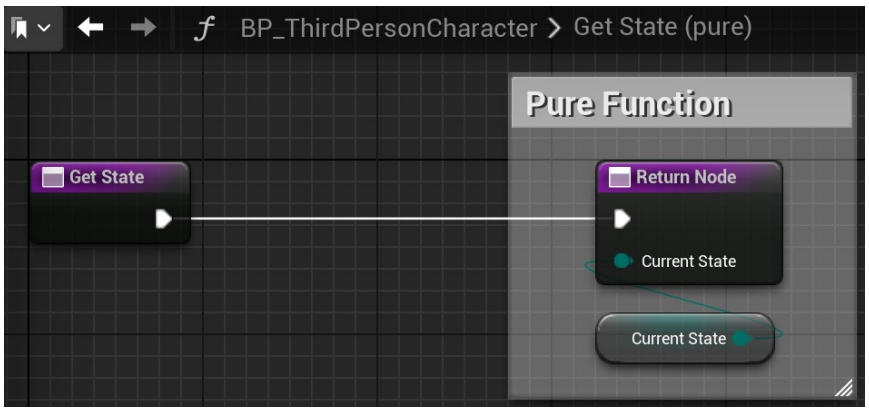
Dieses Event stellt sicher, dass der Soft Lock-On Modus deaktiviert wird, wenn das Ziel außerhalb einer bestimmten Reichweite liegt. Es überprüft zunächst, ob der Charakter im Target LockOn Modus ist. Wenn dies nicht der Fall ist, wird die Entfernung zum Ziel berechnet und verglichen. Liegt das Ziel außerhalb der definierten Reichweite, wird das Soft Target auf ungültig gesetzt, wodurch der Soft Lock-On Modus beendet wird.

Funktion SetState



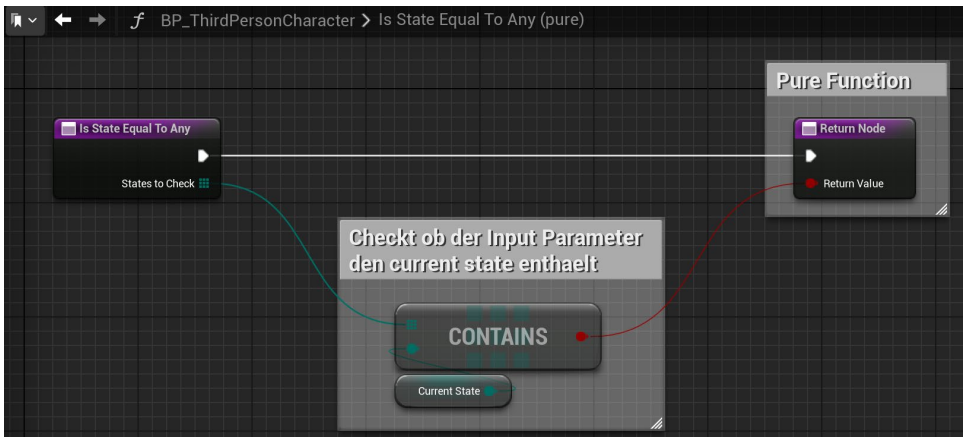
Die Set State Funktion dient dazu, den aktuellen Zustand des Charakters zu ändern. Sie verwendet den Enumerator E_States, der die Zustände Nothing, Attack, Dodge und Guard definiert. Sie stellt sicher, dass der Zustand des Charakters nur geändert wird, wenn der neue Zustand sich vom aktuellen Zustand unterscheidet.

Funktion GetState



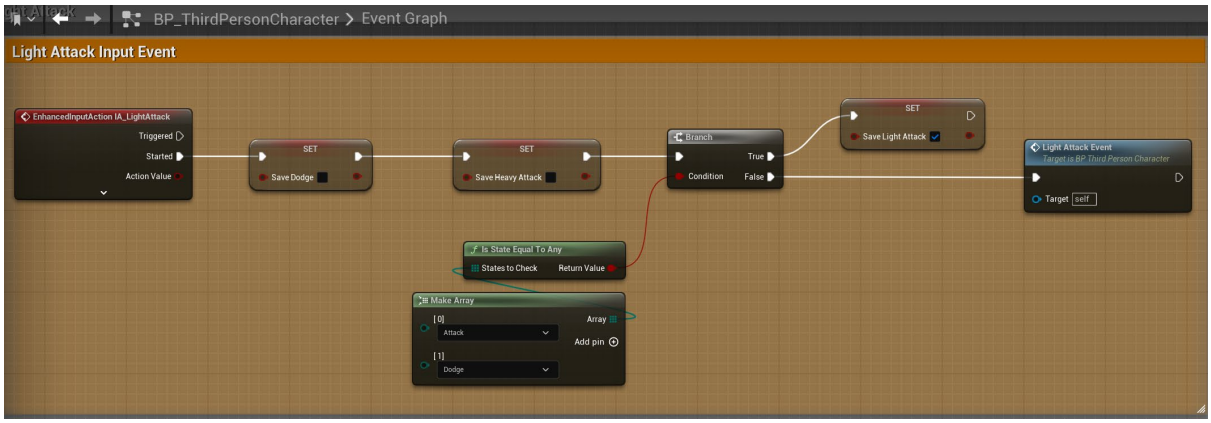
Die Get State Funktion ist eine pure Funktion, die dazu dient, den aktuellen Zustand des Charakters auf Basis des E_States abzurufen.

Funktion IsStateEqualToAny



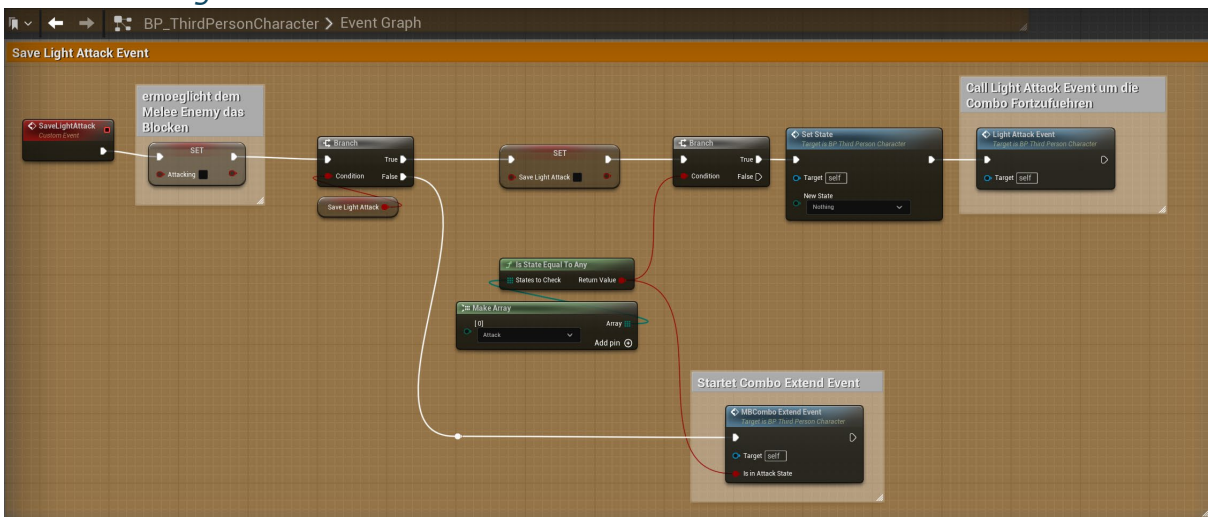
Die Funktion ist eine pure Funktion, die überprüft, ob der aktuelle Zustand des Charakters (Current State) mit einem der in den Eingabeparametern angegebenen Zustände übereinstimmt. Sie verwendet den Enumerator E_States, der die Zustände Nothing, Attack, Dodge und Guard enthält.

Event LightAttack Input



Das Event stellt sicher, dass die Eingabe für eine leichte Attacke korrekt verarbeitet wird. Die Vorherigen gespeicherten Eingaben des Dodges und der Heavy Attack werden gelöscht. Wenn der Charakter sich nicht im Zustand Attack oder Dodge befindet, wird das Light Attack Event wird aufgerufen, um die leichte Attacke auszuführen. Wenn der Charakter sich noch im Zustand Attack oder Dodge befindet, wird die leichte Attacke nicht sofort ausgeführt, sondern die Eingabe wird nur gespeichert, um sie sobald das AN_SaveLightAttack in der Animation ausgeführt wird auszuführen.

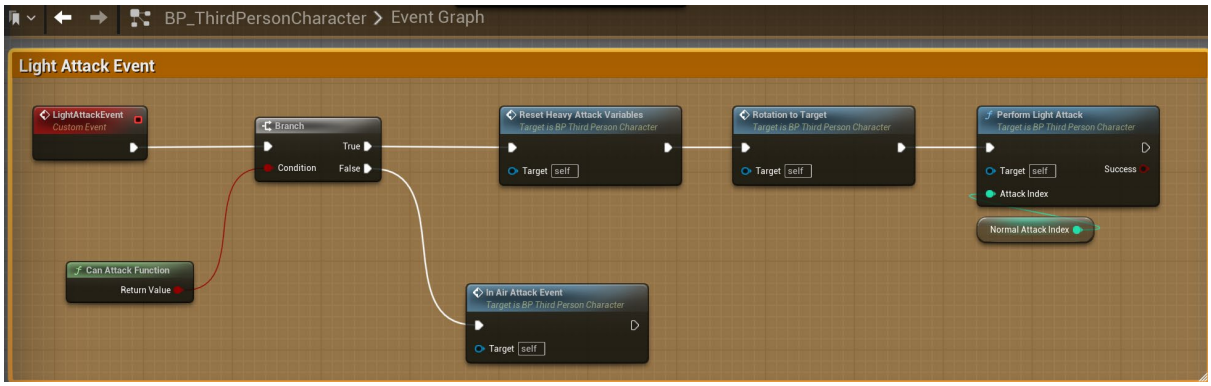
Event SaveLightAttack



Dieses Event ermöglicht es, eine gespeicherte leichte Attacke zu verarbeiten und den Angriffsstatus des Charakters zu verwalten. Es wird durch den AnimNotify AN_SaveLightAttack gestartet. Wenn eine leichte Attacke gespeichert wurde, ist die Variable Save Light Attack auf True. Ist das der Fall wird die auf False gesetzt, wenn der aktuelle Zustand Attack ist, wird das Light Attack Event ausgeführt, um die aktuelle Combo fortzuführen. Der Zustand des Charakters wird auf Nothing gesetzt, um das Ausüben der nächsten Attacke zu ermöglichen.

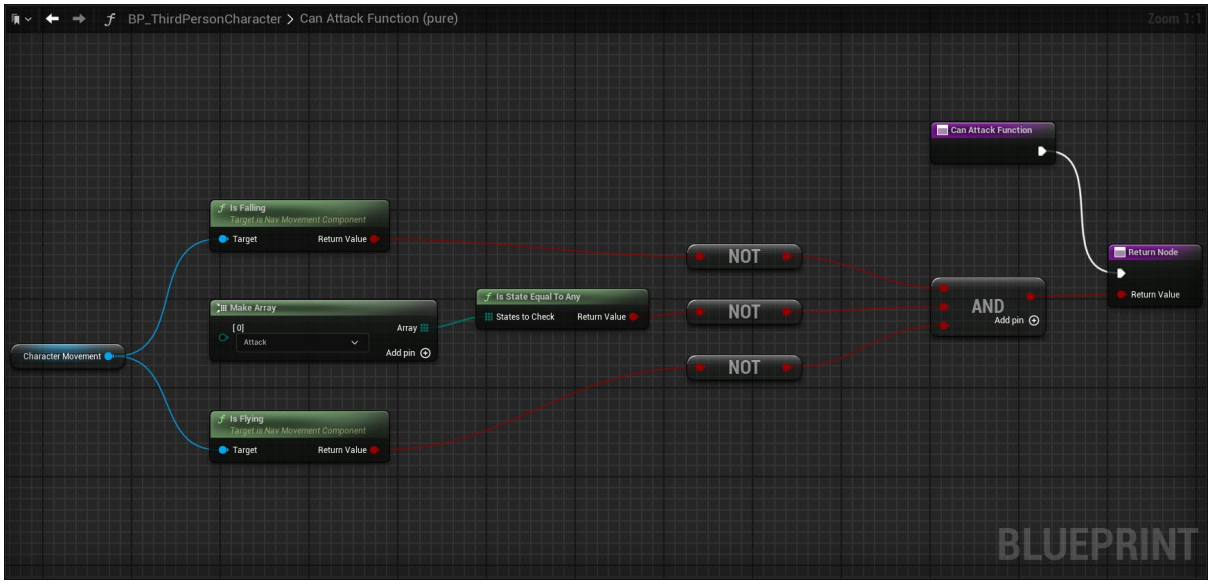
Ist die letzte gespeicherte Eingabe keine LightAttack wird das MBCombo Extend Event des Multi-Button-Combo-Systems gestartet, um die Combo fortzuführen, die mit einer Heavy Attack beginnt.

Event LightAttackEvent



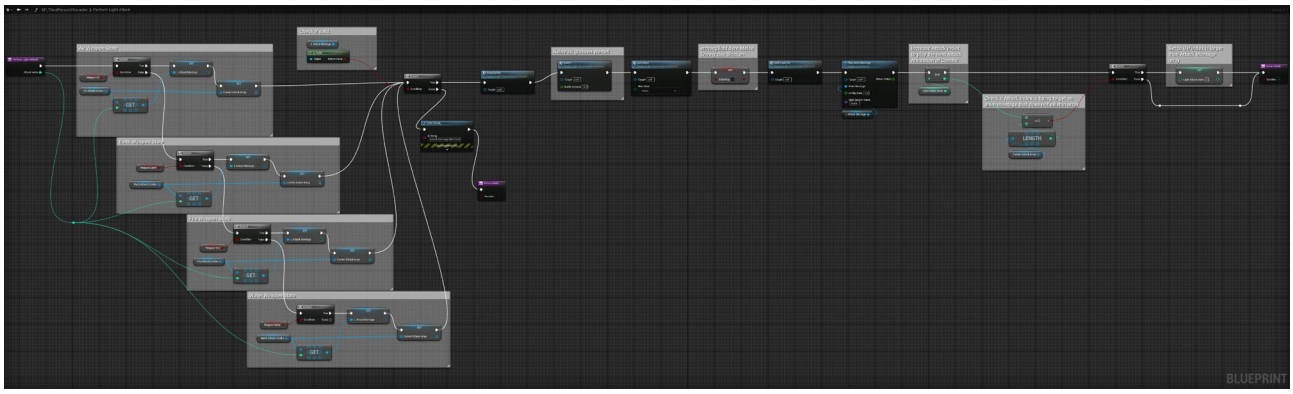
Dieses Event stellt sicher, dass eine leichte Attacke nur ausgeführt wird, wenn der Spieler derzeit in der Lage ist, anzugreifen. Wenn dies der Fall ist, werden alle relevanten Variablen zurückgesetzt, der Charakter auf das Ziel ausgerichtet, und die Attacke ausgeführt. Falls der Charakter sich in der Luft befindet, wird das In Air Attack Event aufgerufen, um die Attacke entsprechend anzupassen.

Funktion CanAttackFunction (pure)



Die Funktion Can Attack Function überprüft, ob der Charakter nicht fällt, nicht fliegt und nicht bereits im Zustand Attack ist. Wenn alle diese Bedingungen erfüllt sind, gibt die Funktion true zurück, was bedeutet, dass der Charakter mit normalen Attacken angreifen kann. Andernfalls gibt sie false zurück, was bedeutet, dass der Charakter vorausgesetzt er befindet sich in der Luft nur mit InAir Attacks angreifen kann.

Funktion PerformLightAttack



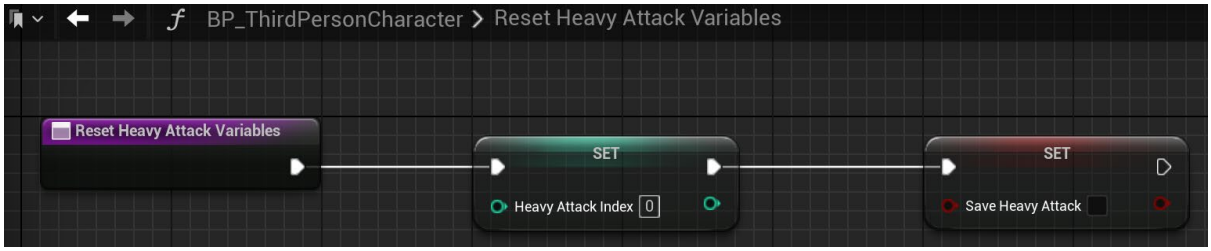
Die Funktion "Perform Light Attack" im BP_ThirdPersonCharacter ist für die Ausführung eines leichten Angriffs des Charakters verantwortlich. Sie führt einen leichten Angriff aus, indem sie die richtige Angriffsanimation basierend auf dem aktuellen Weapon State auswählt, die Angriffsanimation abspielt und den Angriffsindex aktualisiert, um Kombo Angriffe zu ermöglichen.

Hier eine kurze Zusammenfassung des Ablaufs:

1. **Custom Event: Perform Light Attack**
    - Startet die Funktion für den leichten Angriff.
2. **Überprüfen der Weapon State**
    - Es wird überprüft, in welchem Zustand sich die Waffe befindet (Air, Earth, Fire, Water).
    - Abhängig vom Weapon State wird das entsprechende Angriffskombo-Array geladen.
3. **Check if Valid**
    - Überprüft, ob der Angriff durchgeführt werden kann.
4. **Buffer**
    - Falls der Buffer noch im Gange ist wird dieser gestoppt und ein neuer gestartet.
5. **Set Attacking**
    - Setzt das "Attacking" Bool auf True, um anzuzeigen, dass ein Angriff im Gange ist.
    - Aktualisiert den State auf Attacking
6. **Soft Lock On**
    - Sorgt dafür, dass der Charakter sich auf das Ziel ausrichtet.
7. **Play Attack Montage**
    - Spielt die Angriffsanimation basierend auf dem aktuellen Angriffskombo-Array ab.

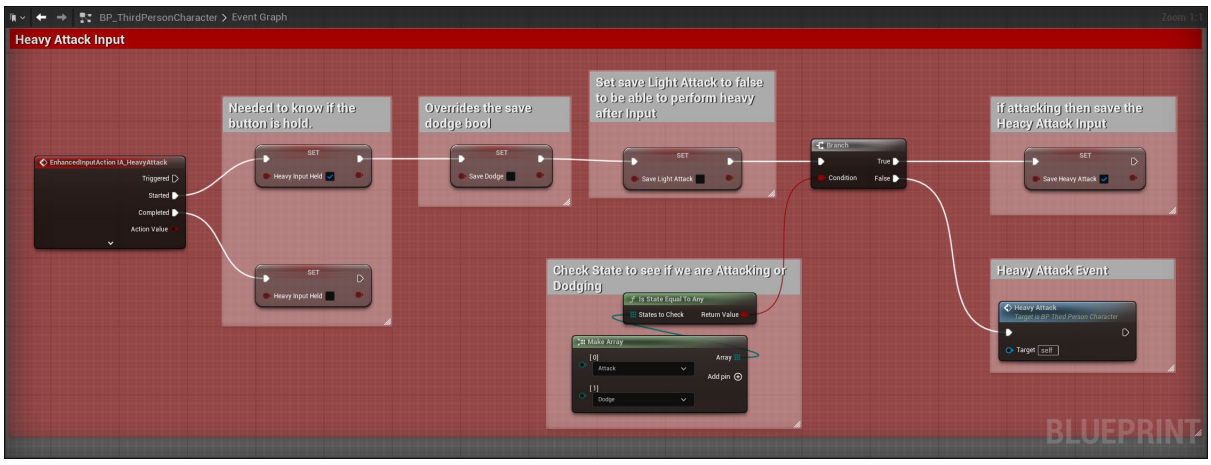
8. **Increment Attack Index**
    - Erhöht den Angriffsindex, um die nächste Angriffsanimation im Kombo zu spielen.
9. **Reset Attack Index**
    - Überprüft, ob der Angriffsindex die Länge des Angriffskombo-Arrays überschreitet. Falls ja, wird der Index zurückgesetzt.

Funktion ResetHeavyAttackVariables



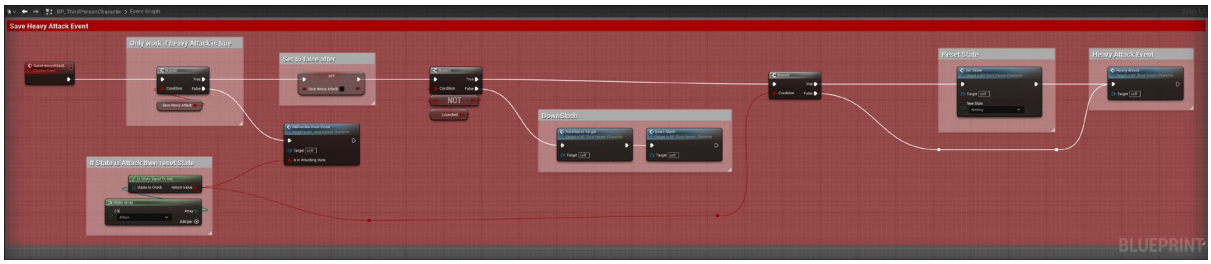
Diese Funktion stellt sicher, dass alle Variablen, die mit schweren Angriffen zu tun haben, auf ihre Standardwerte zurückgesetzt werden. Dies ist nützlich, um sicherzustellen, dass nach einer schweren Angriffskombo alle relevanten Variablen für den nächsten Einsatz korrekt initialisiert sind.

Event HeavyAttack Input



Das Event verarbeitet den Input des schweren Angriffs des Charakters. Es setzt den Boolean Heavy Input Held, um festzulegen, ob der Button gehalten wird oder nicht, löscht gespeicherte LightAttack und Dodge Eingaben und überprüft, ob der Charakter derzeit angreift oder ausweicht. Wenn der Charakter angreift oder ausweicht, wird der schwere Angriff gespeichert, andernfalls wird der schwere Angriff direkt eingeleitet.

Event SaveHeavyAttack



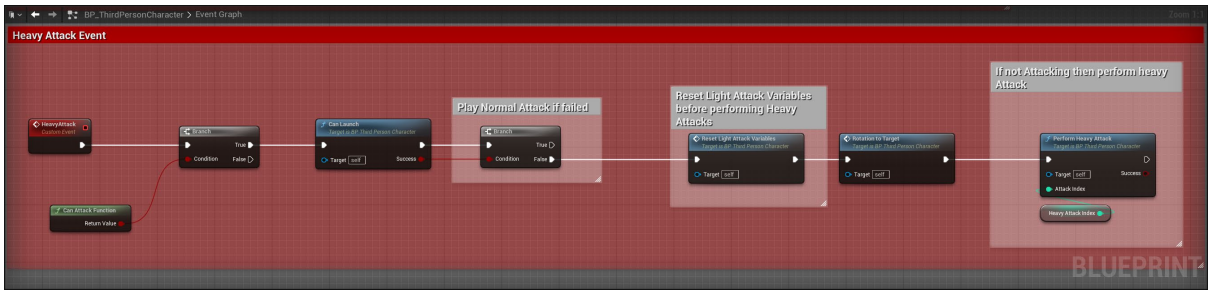
Dieses Event ermöglicht es, eine gespeicherte schwere Attacke zu verarbeiten und den Angriffsstatus des Charakters zu verwalten. Es wird durch den AnimNotify AN_SaveHeavyAttack gestartet. Wenn eine schwere Attacke gespeichert wurde, ist die Variable Save Heavy Attack auf True. Ist das der Fall wird sie auf False gesetzt und ein Branch testet, ob sich der Player im Launched Zustand befindet.

Befindet er sich nicht im Launched Zustand wird falls er sich gleichzeitig im Attack State befindet sein State gegebenfalls auf Nothing gesetzt und das Heavy Attack Event wird gestartet.

Befindet sich der Player im Launched State wird er Richtung Ziel rotiert und das DownSlash Event wird gestartet.

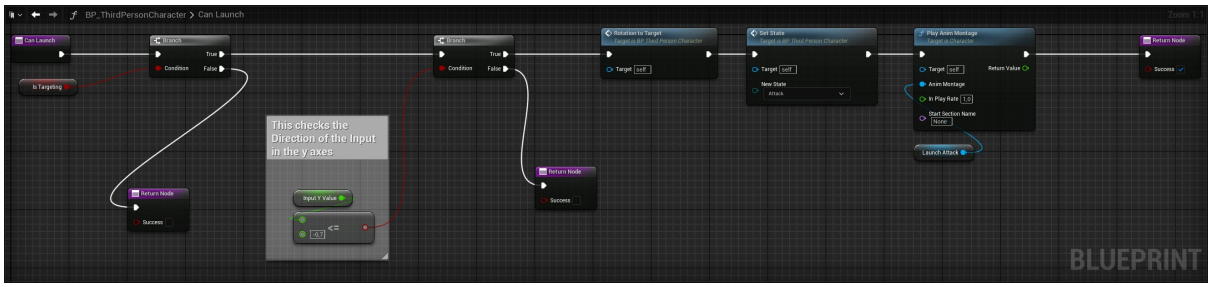
Ist die letzte gespeicherte Eingabe keine LightAttack wird das MBCombo Start Event des Multi-Button-Combo-Systems gestartet, um die Combo Multy Button Combo zu starten und die Boolean ob der Player sich im Attacking State befindet wird weitergegeben.

Event HeavyAttack



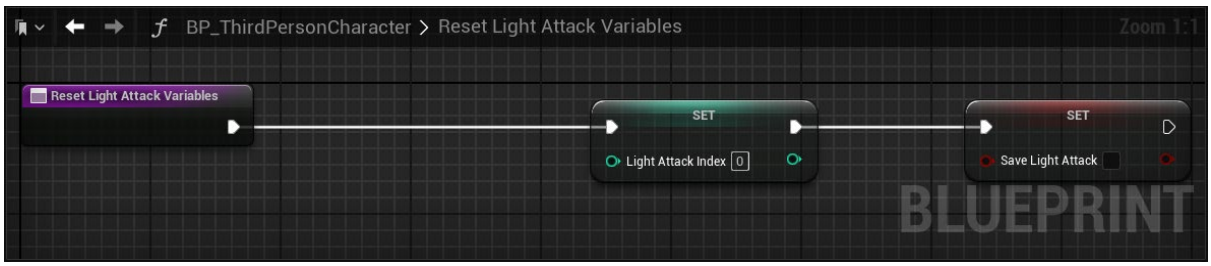
Das Heavy Attack Event prüft zuerst, ob der Charakter angreifen kann. Wenn dies nicht der Fall ist, wird überprüft, ob ein Sprungangriff möglich ist. Falls kein Sprungangriff möglich ist, werden die Variablen für leichte Angriffe zurückgesetzt, der Charakter in Richtung des Ziels gedreht und schließlich der schwere Angriff ausgeführt.

Funktion CanLaunch



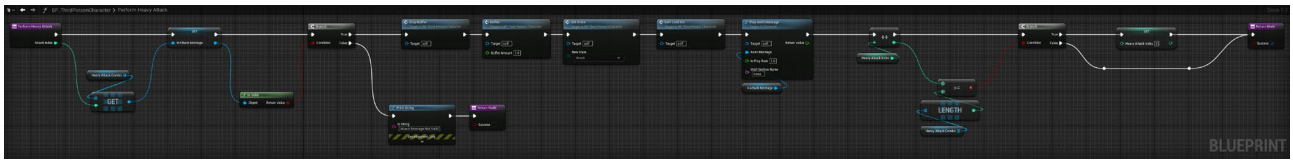
Die Can Launch Funktion prüft zuerst, ob der Charakter im Zielmodus ist. Wenn dies der Fall ist, wird die Richtung der Y-Achsen-Eingabe überprüft. Steuert der Player den Bewegungsinput während dieser Funktion nach Süden (hinten), dreht sich der Spieler in Richtung des Ziels, der Zustand wird auf Attack gesetzt, und die Angriffsanimation für die Launch Attack wird abgespielt. Andernfalls wird die Funktion frühzeitig beendet und gibt False zurück.

Funktion ResetLightAttackVariables



Die Reset Light Attack Variables Funktion stellt sicher, dass die Angriffsvariablen in einen Ausgangszustand zurückversetzt werden. Dies wird erreicht, indem der Light Attack Index auf 0 und die Save Light Attack Variable auf False gesetzt wird.

Funktion PerformHeavyAttack



Die Perform Heavy Attack Funktion verwaltet die Ausführung eines schweren Angriffs des Charakters. Sie wählt die entsprechende Angriffsanimation basierend auf dem aktuellen Heavy Attack Index, spielt diese ab und erhöht den Index für die nächste Angriffsanimation in der Combo. Wenn der Index das Ende der Combo erreicht, wird er zurückgesetzt. Die Funktion sorgt auch dafür, dass der Zustand des Charakters und die Eingabe-Buffer entsprechend aktualisiert werden.

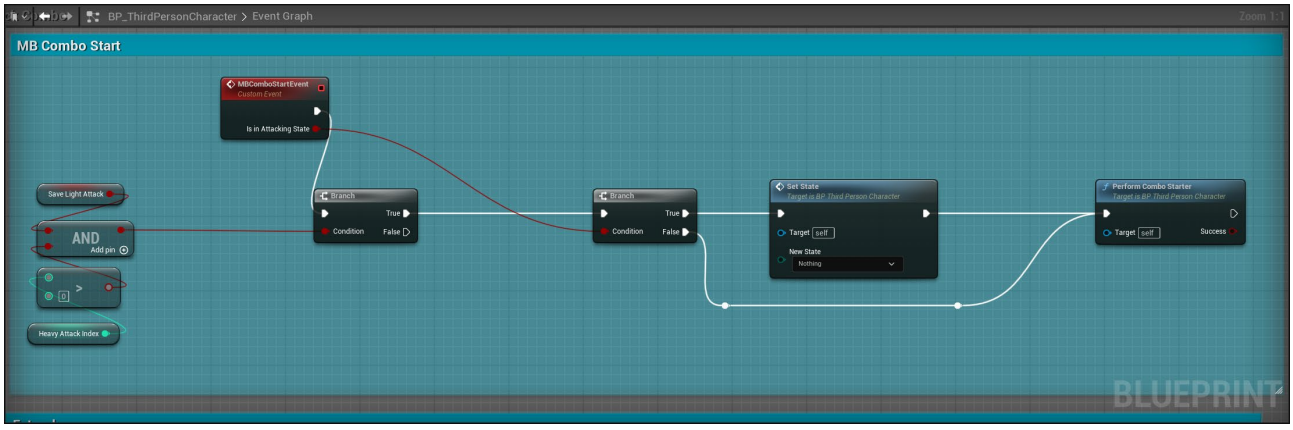
Der Ablauf:

1. **Event Start**:
    - Der Angriff wird gestartet und der aktuelle Angriff aus dem "Heavy Attack Combo" Array abgerufen.
2. **Set Heavy Attack**:

- Die Variable für den schweren Angriff wird auf true gesetzt.

3. **Angriff prüfen**:
   - Es wird überprüft, ob der Angriff ausgeführt werden kann.
   - Wenn nicht enthält das Array keine Animationen und die Funktion wird abgebochen.
4. **Buffer**:
   - Der Buffer wird gestoppt und neu gestartet.
5. **Zustand setzen**:
   - Der Zustand des Charakters wird auf "Attack" gesetzt.
6. **Soft Lock On aktivieren**:
   - Der Soft Lock-On wird aktiviert, um den Player auf das Ziel zu richten.
7. **Animation abspielen**:
   - Die Animation für den schweren Angriff wird abgespielt.
8. **Angriff Index erhöhen**:
   - Der Index des "Heavy Attack" wird um eins erhöht und überprüft, ob er das Ende des Arrays überschreitet.
   - Falls ja, wird der Index auf null zurückgesetzt.
9. **Rückgabe**:
   - Der Erfolg der Angriffsausführung wird zurückgegeben.

Event MBComboStartEvent



Das MB Combo Start-Event überprüft, ob ein leichter Angriff gespeichert wurde und vorher eine Heavy Attack ausgeführt wurde. Wenn beide Bedingungen erfüllt sind, wird geprüft, ob sich der Player noch im Attacking State befindet. Ist das der Fall wird der State auf Nothing gesetzt. In beiden Fällen wird zum Ende des Events die Funktion PerformComboStarter ausgeführt um den die MB Combo zu starten.

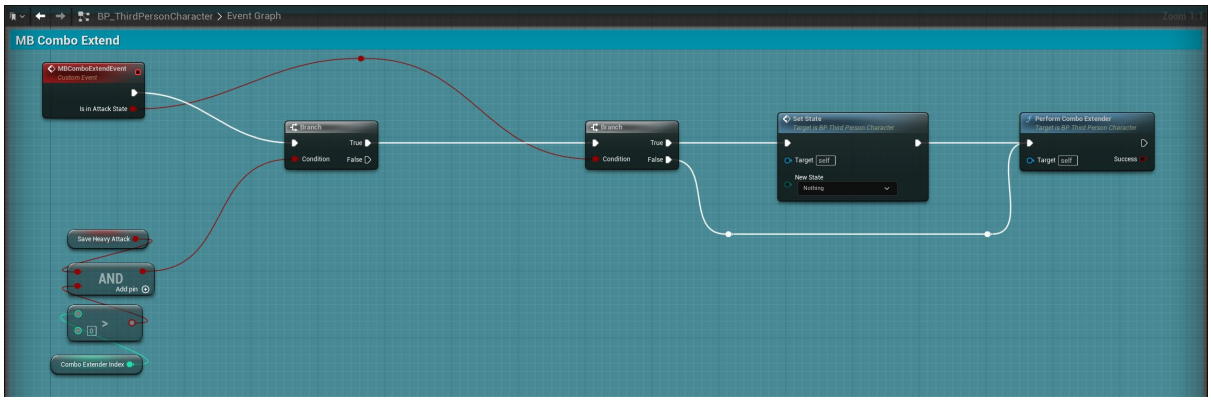
Funktion PerformComboStarter

Die Perform Combo Starter Funktion initiiert nachdem man direkt nach einer Heavy Attack eine Light Attack initiiert hat den Beginn einer Combo Abfolge. Sie wählt die entsprechende Angriffsanimation basierend auf dem aktuellen Heavy Attack Index, spielt diese ab und setzt den Zustand des Charakters entsprechend. Die Funktion stellt sicher, dass keine weiteren Eingaben gepuffert werden, während der Combo-Starter ausgeführt wird, und aktualisiert die relevanten Variablen, um die Ausführung der Combo zu ermöglichen.

Der Ablauf:

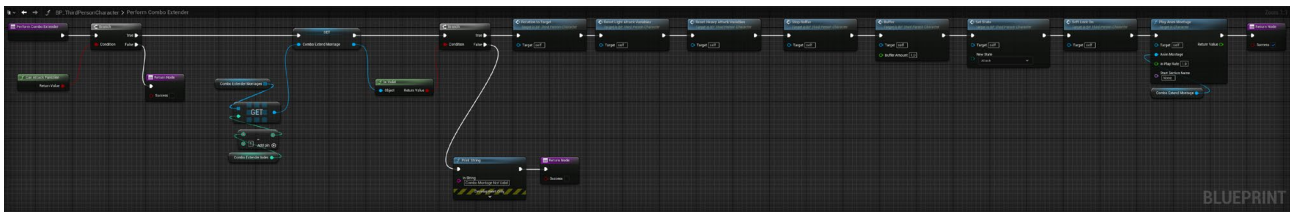
1. **Event Start**:
    - Das Event "Perform Combo Starter" wird ausgelöst.
2. **Überprüfungen**:
    - Ein Branch-Node prüft, ob sich der Player nicht im Attack oder Dodge State befindet. Wenn die Bedingung nicht erfüllt ist, wird die Funktion mit einem Success false beendet.
3. **Angriffs-Montage setzen**:
    - Wenn die Bedingung erfüllt ist, wird der Heavy Attack Index -1 gerechnet um die entsprechende Angriffs-Montage aus dem "Combo Starter Montage" Array abzurufen und damit die Attack Montage zu setzten.
4. **Aktionen**:
    - Der Charakter wird zu seinem Ziel rotiert.
    - Der Combo Extender Index für die Folgeattacke wird gesetzt.
    - Der Buffer wird gestoppt und neu gestartet.
    - Die gespeicherten Eingaben werden gelöscht.
    - Der State wird auf Attack gesetzt
    - Der "Soft Lock On" Modus wird aktiviert.
    - Die entsprechende Angriffs-Montage wird abgespielt.
5. **Ergebnis**:
    - Die Funktion gibt einen Rückgabewert "Success" zurück.

Event MBComboExtendEvent



Das MB Combo Extend-Event überprüft, ob ein schwerer Angriff gespeichert wurde und ob der Combo Extender Index gesetzt wurde. Wenn beide Bedingungen erfüllt sind, wird der Zustand des Charakters auf Nothing zurückgesetzt und die Perform Combo Extender-Funktion aufgerufen, um die Combo fortzusetzen.

Funktion PerformComboExtender



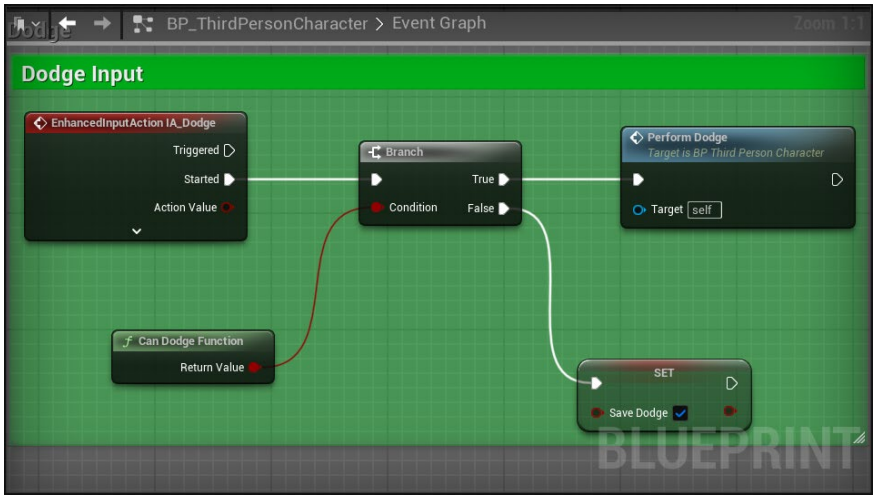
Diese Funktion verwaltet die Erweiterung einer MB Combo, nachdem der Combo-Starter erfolgreich ausgeführt wurde. Sie prüft, ob der Charakter angreifen kann, ruft die entsprechende Combo Extender Montage ab, setzt Angriffsvariablen zurück, puffert den Zustand und spielt schließlich die Anim Montage ab. Wenn alle Schritte erfolgreich sind, wird ein Erfolg zurückgegeben.

Der Ablauf:

1. **Event Start**:
    - Das Event "Perform Combo Extender" wird ausgelöst.
2. **Überprüfung der Angriffsbereitschaft**:
    - Es wird geprüft, ob der Charakter aktuell einen Angriff ausführen kann. Falls nicht, wird die Funktion beendet.
3. **Abrufen und Überprüfen der Combo Extender Montage**:
    - Die aktuelle Combo Extender Index wird verwendet, um die entsprechende Montage abzurufen. Falls diese ungültig ist, wird eine Fehlermeldung ausgegeben und die Funktion beendet.
4. **RotationToTarget**:
    - Der Player wird in die Richtung seines Ziels rotiert
5. **Rücksetzen der Angriffsvariablen**:
    - Die Variablen für leichte und schwere Angriffe werden zurückgesetzt.

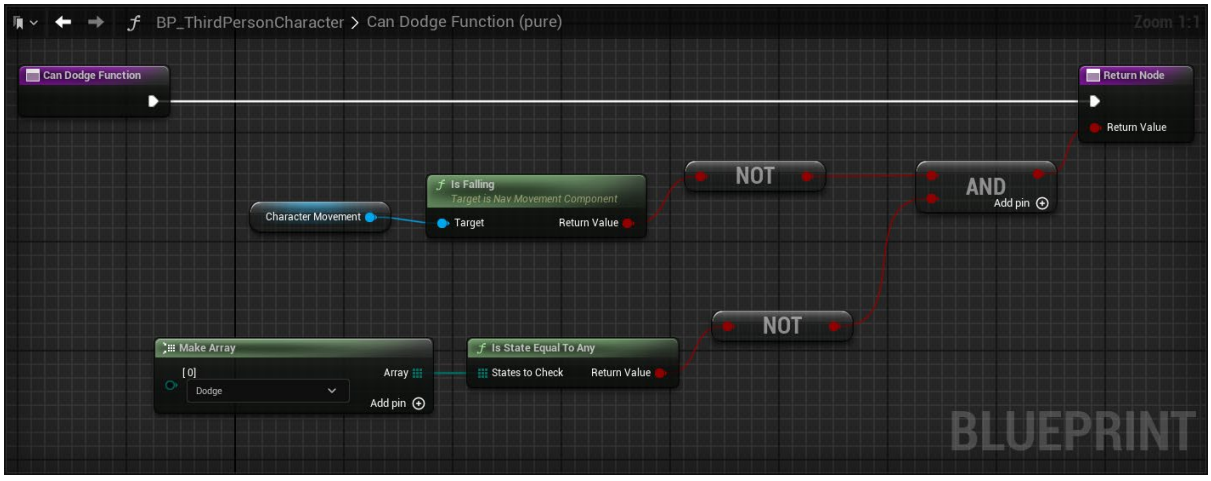
6. **Buffern und Zustand setzen**:
    - Der aktuelle Buffer wird gestoppt und ein neuer Buffer wird gesetzt. Der Zustand des Charakters wird auf "Attack" geändert.
7. **Soft Lock On**:
    - Der Charakter bleibt auf das Ziel fokussiert.
8. **Abspielen der Anim Montage**:
    - Die neu gesetzte Combo Extender Montage wird abgespielt.

Event Dodge Input



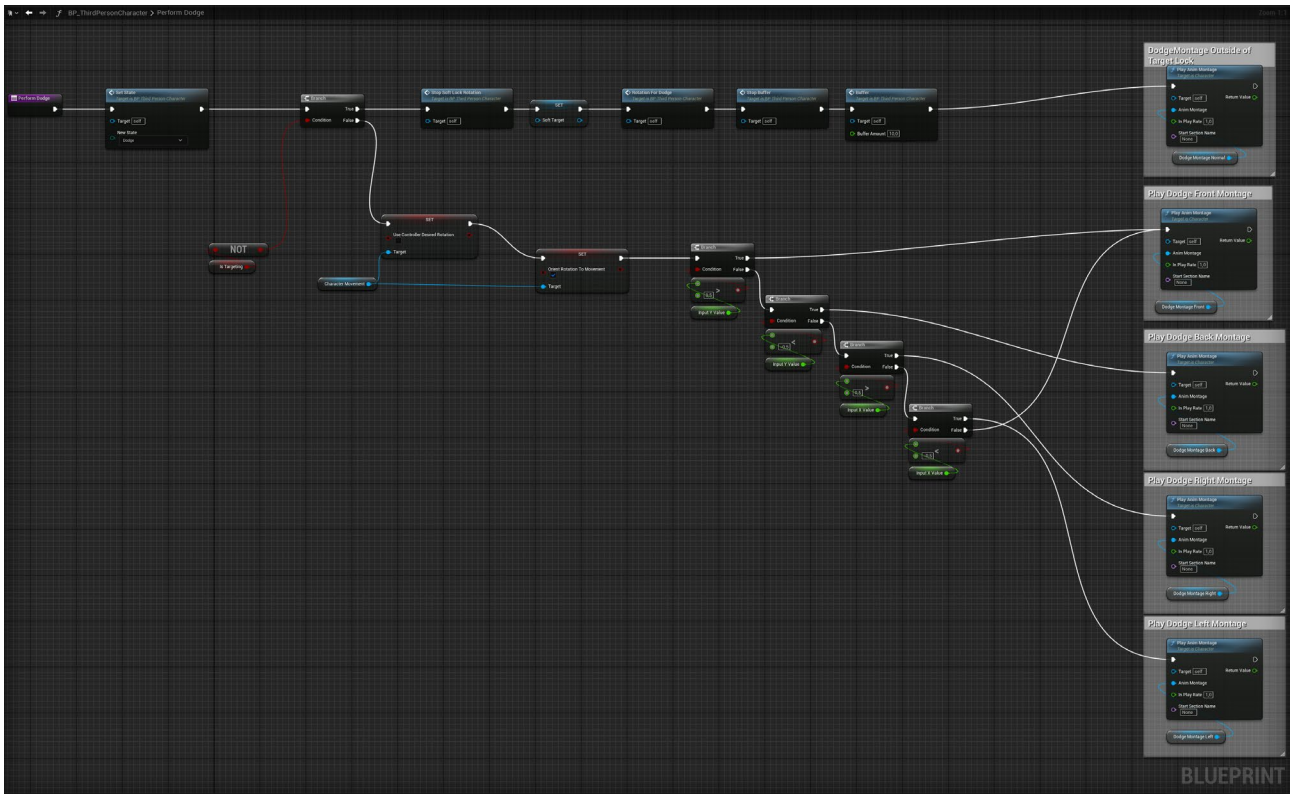
Dieses Event verwaltet den Ausweich-Input des Charakters und entscheidet, ob das Ausweichmanöver ausgeführt wird oder der Input gespeichert wird.

Funktion CanDodgeFunction (pure)



Diese Funktion testet, dass der Charakter nur dodgen kann, wenn er sich nicht im Zustand "Dodge" befindet und nicht gerade fällt.

Funktion PerformDodge



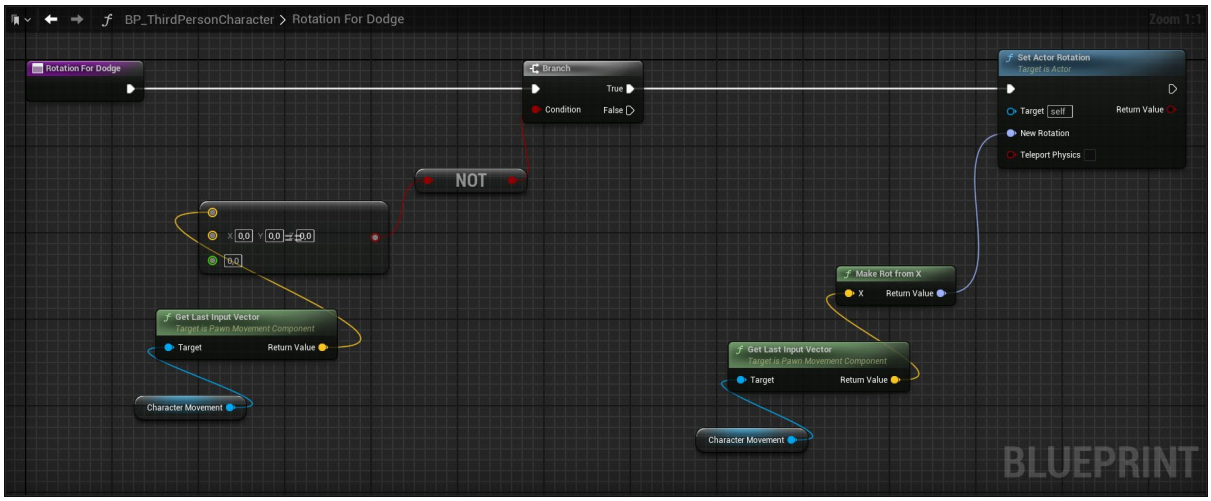
Diese Funktion führt, je nachdem, ob er sich im Zielmodus befindet oder nicht, und basierend auf der Eingaberichtung verschiedene Dodge-Aktionen aus.

Der Ablauf:

1. **Event Start**:
    - Das Event "Perform Dodge" wird ausgelöst.
2. **Setze Zustand**:
    - Der Zustand des Charakters wird auf "Dodge" gesetzt.
3. **Prüfe Target Lock**:
    - Es wird überprüft, ob der Charakter sich im Target Lock Modus befindet. Falls nicht, wird die gewünschte Rotation des Charakters verwendet und "Orient Rotation to Movement" auf false gesetzt.
4. **If True Soft Lock und Buffern**:
    - Die "Soft Lock Rotation" und "Buffertimeline" werden gestoppt.
    - Der aktuelle Buffer wird mit einem Wert von 1.0 gesetzt.
5. **If True Rotation Setzen**:
    - Der Charakter wird auf die Dodge-Position ausgerichtet.
6. **If True Buffer**:
    - Der Buffer wird gestoppt und neu gesetzt, somit die Dodge Distanz außerhalb des LockOn Modus verlängert werden.

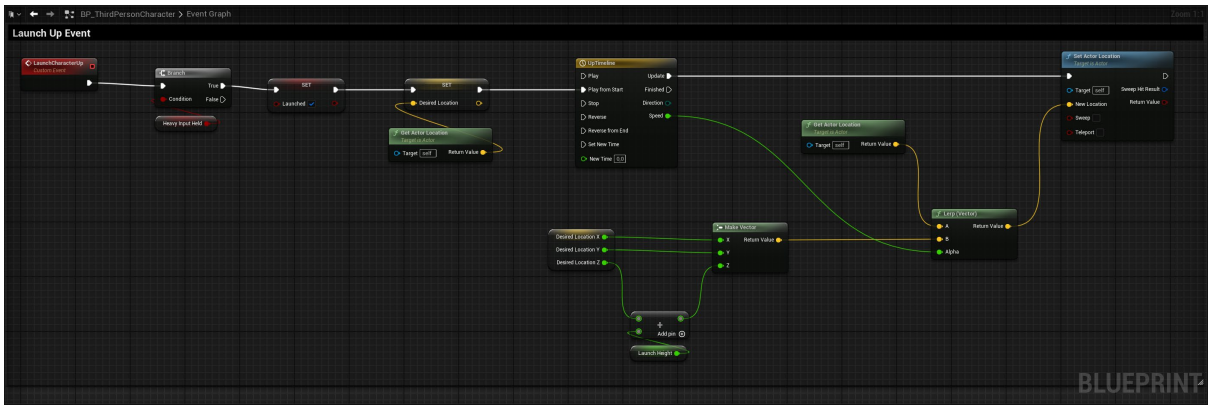
7. **If False Prüfe Eingabe und Anim Montage**:
    - Die Eingaben des Spielers werden geprüft, um die Richtung des Ausweichens zu bestimmen.
        - Wenn der Eingabewert für X positiv ist (nach vorne), wird die vordere Dodge-Montage abgespielt.
        - Wenn der Eingabewert für X negativ ist (nach hinten), wird die hintere Dodge-Montage abgespielt.
        - Wenn der Eingabewert für Y positiv ist (nach rechts), wird die rechte Dodge-Montage abgespielt.
        - Wenn der Eingabewert für Y negativ ist (nach links), wird die linke Dodge-Montage abgespielt.

Funktion RotationForDodge



Die Funktion Rotation For Dodge überprüft, ob es eine Bewegungseingabe gab, bevor der Charakter ausgewichen ist. Wenn eine Bewegungseingabe vorhanden war, wird die Rotation des Charakters entsprechend dieser Eingabe gesetzt, sodass der Charakter in die Richtung des letzten Eingabevektors schaut. Andernfalls wird die Rotation nicht geändert.

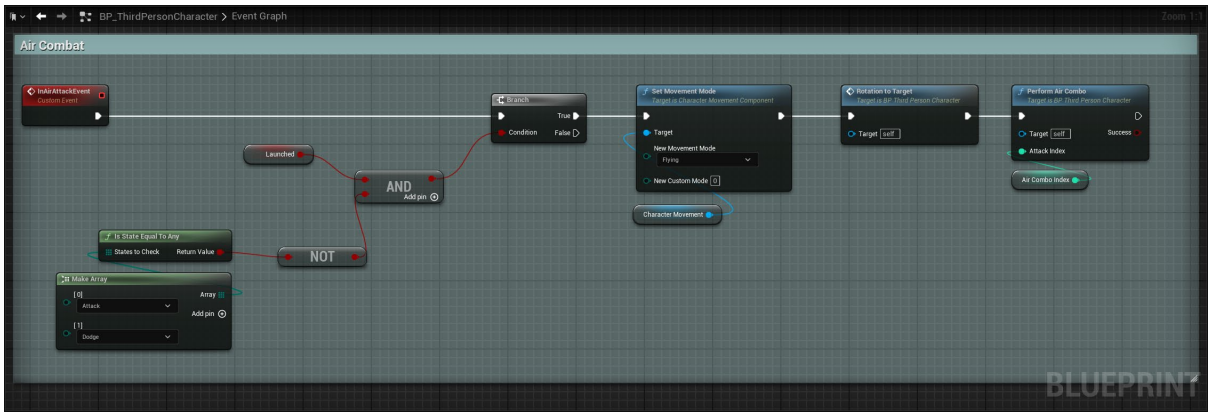
Event LaunchCharacterUp



Das LaunchCharacterUp Event überprüft, ob die Variable Heavy Input Held wahr ist. Wenn ja setzt sie Launched auf True und der Charakter wird in die Luft befördert, indem seine Position über eine Timeline interpoliert wird. Während der Timeline wird die Position des Charakters kontinuierlich aktualisiert, bis die gewünschte Höhe erreicht ist. Wenn die Bedingung falsch ist, passiert nichts und der Charakter bleibt auf dem Boden.

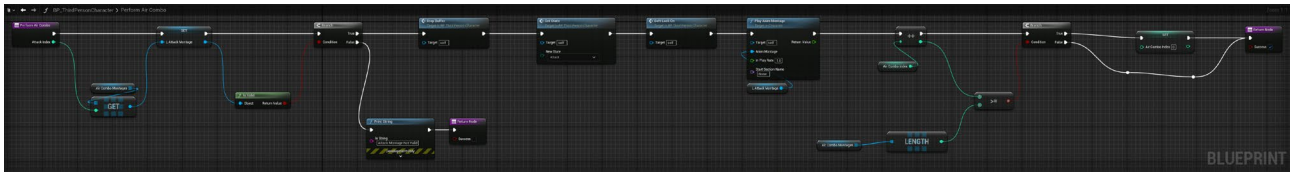
Dieses Event ermöglicht es, den Charakter auf eine vorgegebene Höhe zu befördern, wenn die Launch Attack ausgeführt wird.

Event InAirAttackEvent



Das InAirAttackEvent verarbeitet den Luftangriff des Charakters. Es überprüft, ob der Player Launched und nicht im Attack oder Dodge State ist. Wenn dies der Fall ist, wird der Bewegungsmodus auf Flying gesetzt, um die den Gravitationseinfluss auszusetzen, der Charakter dreht sich zum Ziel, und die Luftcombo-Aktion wird ausgeführt.

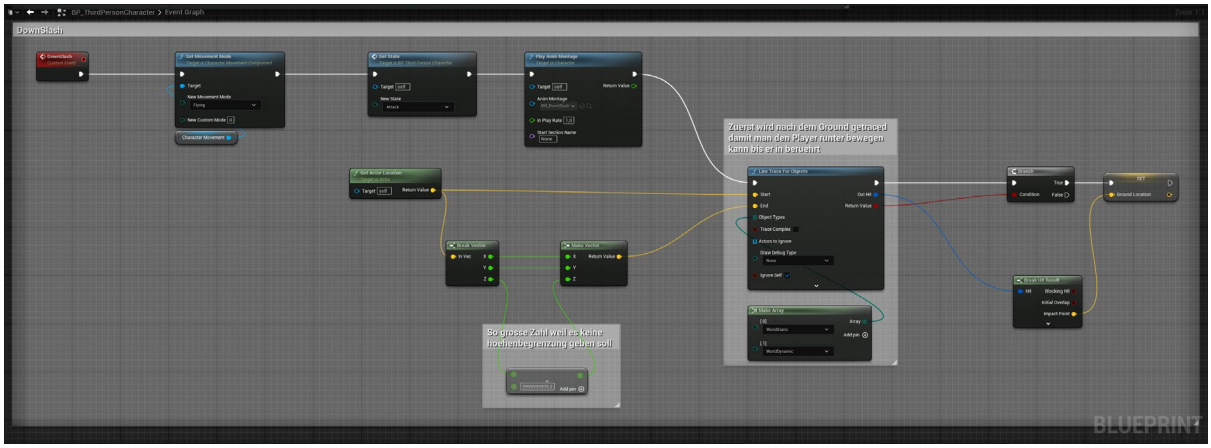
Funktion PerformAirCombo



Die Funktion Perform Air Combo führt die Luftcombo des Charakters aus. Sie ruft die entsprechende Air Combo Montage basierend auf dem Attack Index ab, setzt die lokale Variable, überprüft die Gültigkeit der Montage, stoppt den Buffer, setzt den

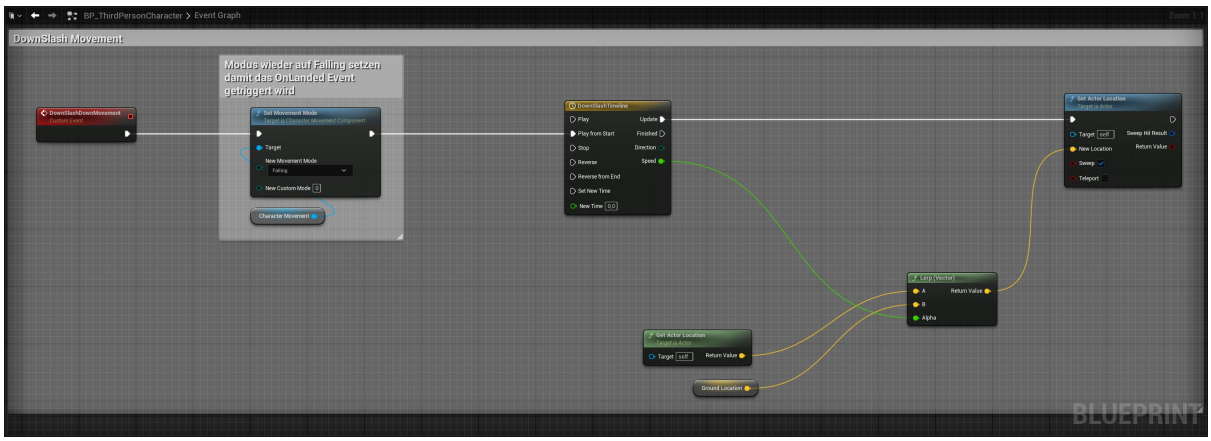
Zustand des Charakters auf Attack, aktiviert den Soft Lock On, spielt die Air Combo Animation ab und erhöht den Air Combo Index für die nächste Attacke in der Combo Reihenfolge. Wenn der Air Combo Index das Ende der verfügbaren Combos erreicht, wird der Index auf 0 zurückgesetzt.

Event DownSlash



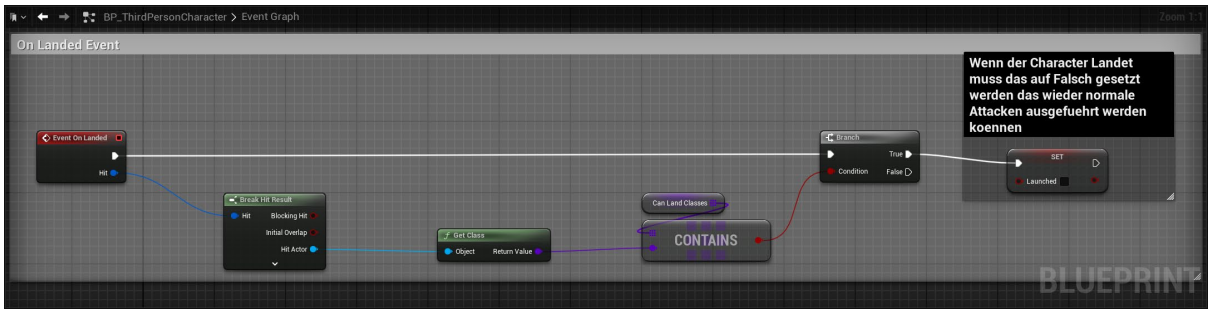
Das DownSlash-Event setzt den Bewegungsmodus des Charakters auf "Flying" um den Gravitationseinfluss auszusetzten, setzt den Zustand auf "Attack", spielt die DownSlash-Animation ab. Dann führt sie einen Line Trace vom Player nach unten durch, um die Bodenposition zu finden. Wenn der Trace erfolgreich ist, wird die Position des Bodens als GroundLocation gespeichert.

Event DownSlashMovement



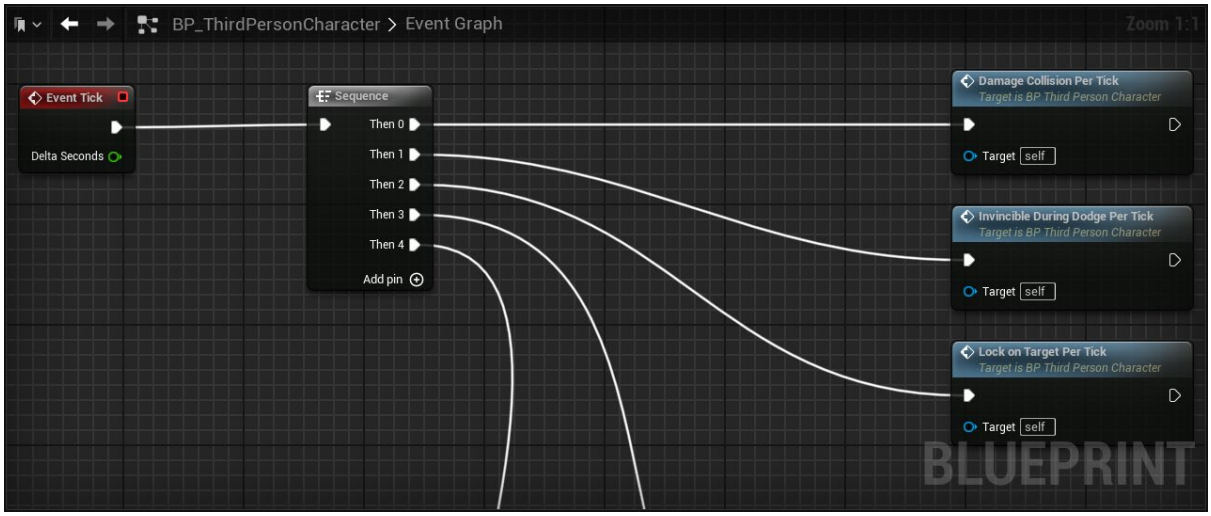
Das DownSlashDownMovement-Event setzt den Bewegungsmodus des Charakters auf "Falling", um den Gravitationseinfluss wieder zu aktivieren und das Auslösen des OnLanded-Events zu ermöglichen. Es startet eine Timeline, die die Position des Charakters von der aktuellen Position zur Ground Location interpoliert. Während der Timeline wird die Position des Charakters kontinuierlich aktualisiert, um eine fließende Bewegung nach unten zu erzeugen.

Event OnLanded



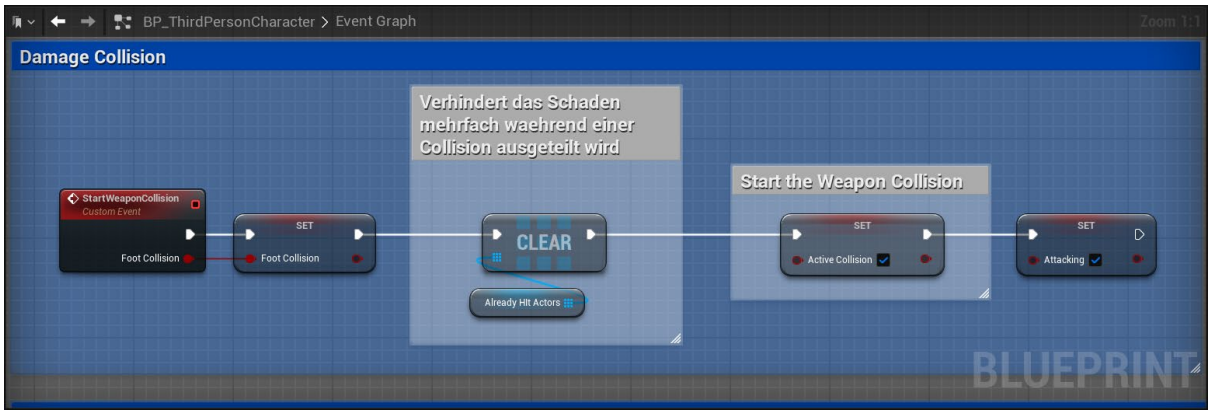
Das OnLanded-Event verarbeitet das Landen des Charakters. Es zerlegt die Informationen über den Aufprall, ermittelt die Klasse des Objekts, auf dem der Charakter gelandet ist, und überprüft, ob es sich um eine erlaubte Landeklasse handelt. Wenn dies der Fall ist, wird die Variable Launched auf false gesetzt, um die Ausführung normaler Angriffe wieder zu ermöglichen.

Event Tick



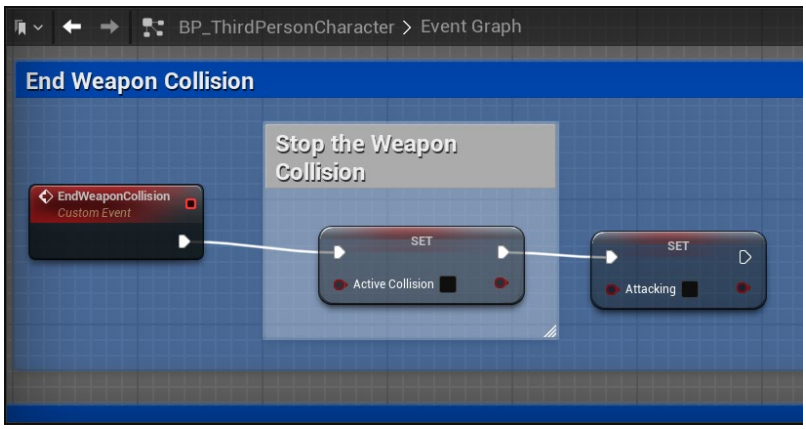
Das Event Tick im BP_ThirdPersonCharacter verwendet eine Sequenz, um mehrere Funktionen innerhalb eines Ticks nacheinander auszuführen. Zunächst wird das Damage Collision Per Event aufgerufen, um Kollisionen zu überprüfen und Schaden auszuteilen. Anschließend sorgt das Invincible During Dodge Per Tick Event dafür, dass der Charakter während eines Ausweichmanövers unverwundbar ist. Schließlich wird das Lock on Target Per Tick Event aufgerufen, um das Zielsystem des Charakters zu aktualisieren und sicherzustellen, dass das Ziel korrekt verfolgt wird.

Event StartWeaponCollision



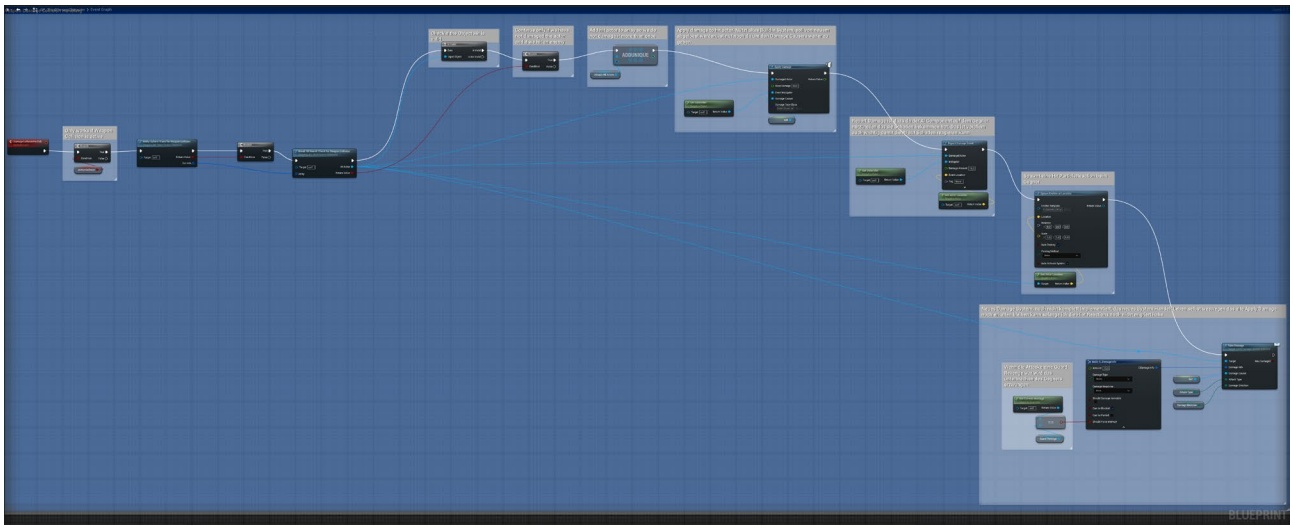
Das StartWeaponCollision Event wird durch den Beginn des Anim Notify State ANS_WeaponCollision gestartet. Es initialisiert die Kollisionserkennung für den Charakterangriff, indem es die Foot Collision setzt, das Array für bereits getroffene Feinde leert und die Booleans ActiveCollision und Attacking auf true setzt.

Event EndWeaponCollision



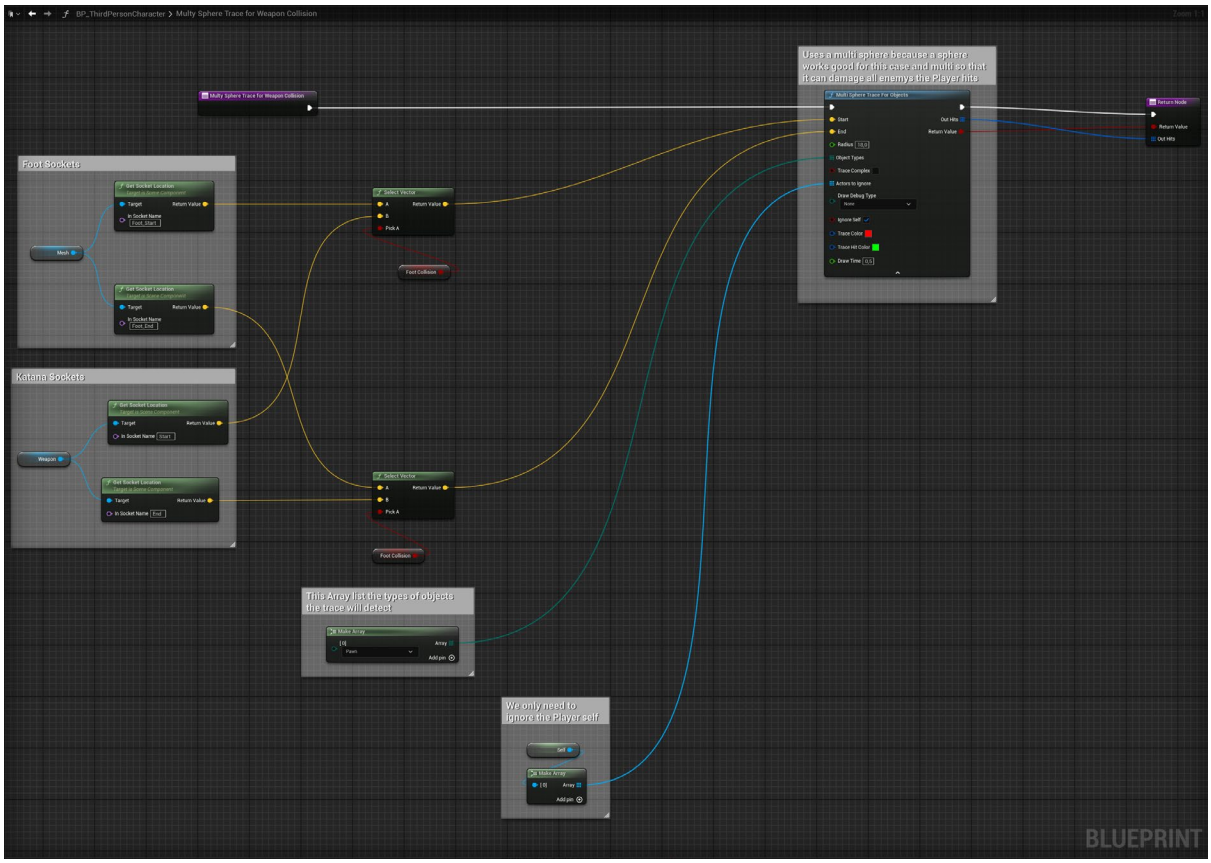
Das Event wird durch das Ende des Anim Notify State ANS_WeaponCollision gestartet. Es deaktiviert die Kollisionserkennung für den Charakterangriff und setzt die beiden Variablen zurück

Event DamageCollisionPerTick



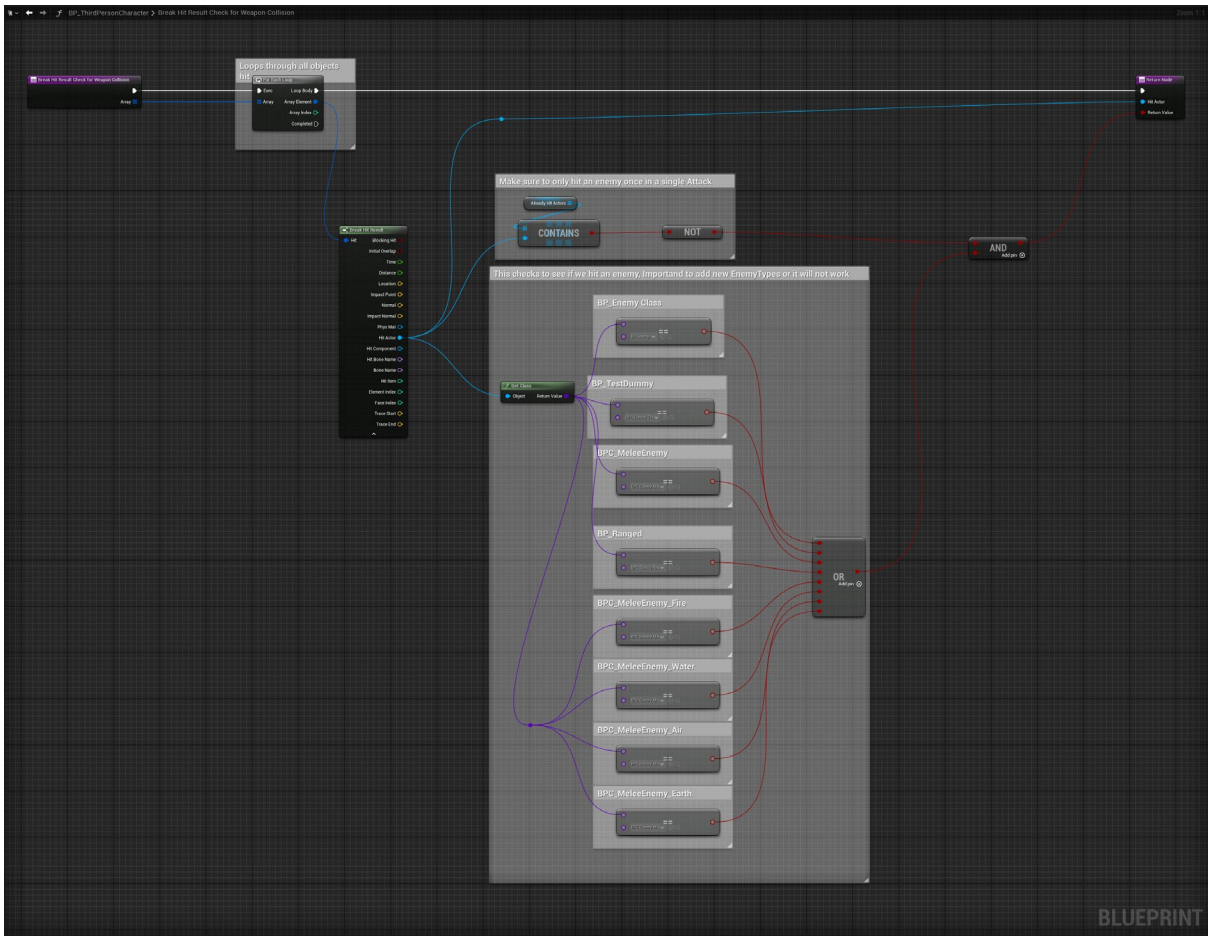
Das Event "DamageCollisionPerTick" prüft bei jedem Tick, ob sich Akteure in der Kollisionszone der Waffe befinden. Es überprüft, ob diese Akteure bereits beschädigt wurden, und falls nicht, wird Schaden angewendet und visuelles Feedback durch einen Partikeleffekt erzeugt. Der Ablauf sieht hierzu wie folgt aus: Solange die Bool ActiveCollision True ist wird die Funktion MultySphereTraceforWeaponCollision genutzt war der Trace erfolgreich wird die BreakHitResultCheckforWaeponCollision genutzt um den getroffenen Gegner auszugeben und zu validieren das es sich dabei um einen Gegner handelt. Ist das der Fall wird der getroffene Actor zu den bereits getroffenen Actors hinzugefügt, um eine mehrfache Schadensberechnung innerhalb einer Damage Collision zu verhindern. Apply Damage nach dem integrierten Damage System wird abgehandelt, was eigentlich wegen des neuen Damage Systems nicht mehr notwendig sein sollte. Allerdings verursacht das Entfernen einen unerwarteten Fehler, der noch nicht behoben werden konnte. Das Report Damage Event wird ausgeführt um die Gegner AI auf den Schaden reagieren lassen zu können. Eine Hit Effekt wird an der Position des Gegners erzeugt. Die TakeDamage Funktion des neuen DamageSystem wird eingeleitet, nimmt die notwendigen Informationen aus dem BP_ThirdPersonCharacter und gibt diese weiter. Hierzu gehören die durch die Anim Notifys gesetzen Enums und die Information das die Guard Revenge (das Parieren) den Gegner unterbrechen soll.

Funktion MultySphereTraceForWeaponCollision



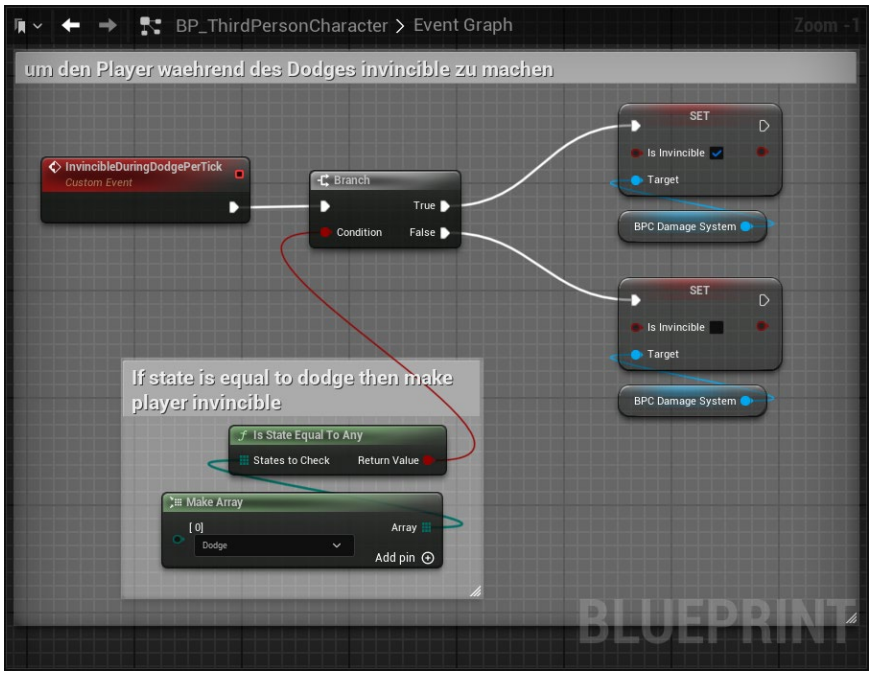
Diese Funktion führt eine Mehrfach-Sphärenverfolgung durch, um Kollisionen für Waffenangriffe und Trittanimationen zu erzeugen. Sie verwendet Select Vector Nodes, um je nach Zustand von Foot Collision entweder die Foot Sockets oder die Katana Sockets für die Verfolgung auszuwählen, und ignoriert den Spielercharakter bei der Kollisionserkennung.

Funktion BreakHitResultCheckForWeaponCollision



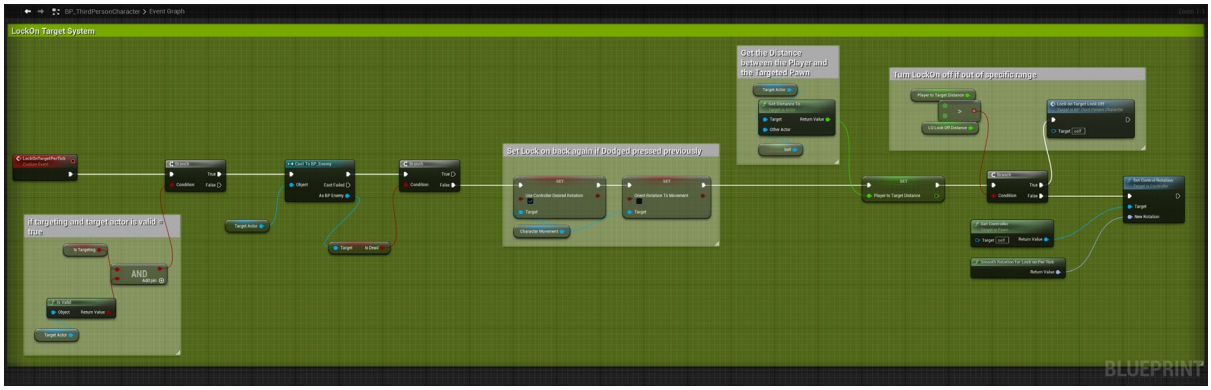
Die Funktion prüft die Trefferergebnisse der Waffenkollision auf vordefinierte Feindtypen und vermeidet doppelte Treffer gegen denselben Akteur. Nur Treffer, die diese Bedingungen erfüllen, werden weiterverarbeitet, um sicherzustellen, dass Schaden nur auf Feinde und nur einmal pro Feind und Angriff angewendet wird.

Event InvincibleDuringDodgePerTick



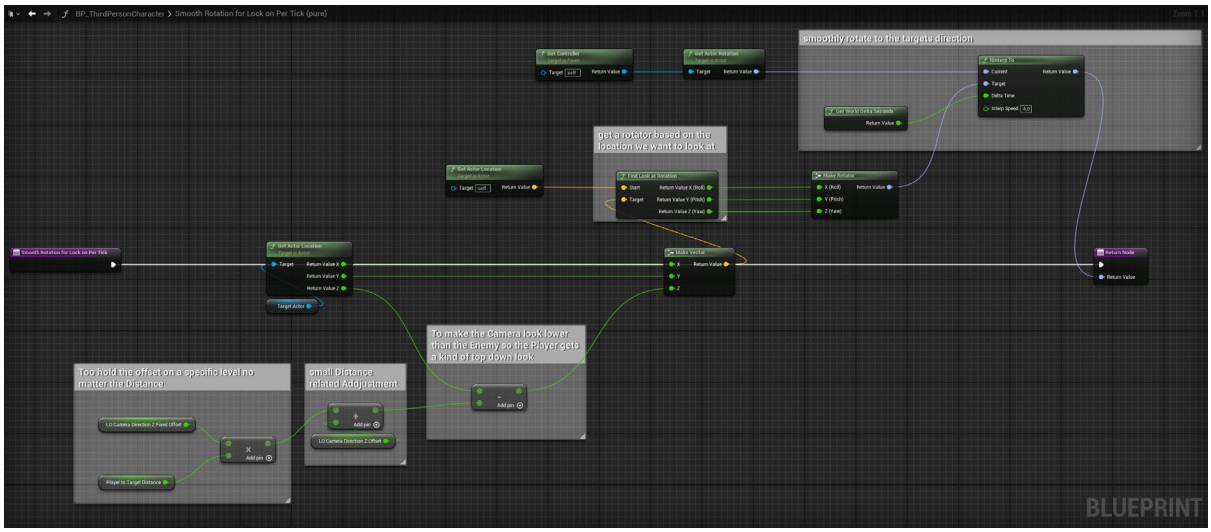
Das "InvincibleDuringDodgePerTick" Event überprüft jeden Frame, ob der Spieler sich im Zustand "Dodge" befindet. Wenn dies der Fall ist, wird der Spieler unbesiegbar gemacht. Wenn der Zustand nicht "Dodge" ist, wird die Unbesiegbarkeit des Spielers deaktiviert. Dieses Event sorgt dafür, dass der Spieler während des Ausweichens keinen Schaden erleidet.

Event LockOnTargetPerTick



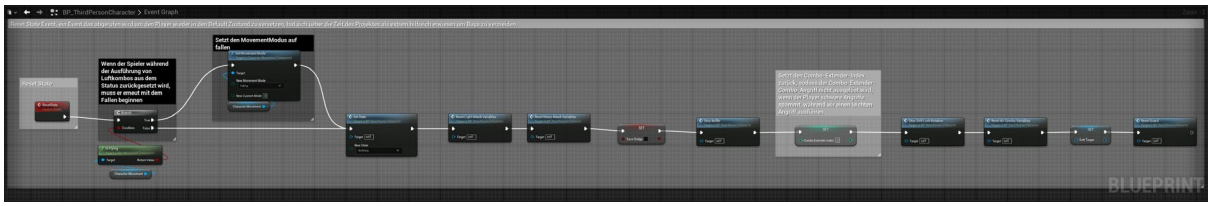
Das "LockOnTargetPerTick" Event sorgt dafür, dass das Lock-On-System des Charakters in Echtzeit aktualisiert wird. Es überprüft kontinuierlich, ob der Spieler ein Ziel anvisiert, ob dieses Ziel gültig und lebendig ist, und ob die Entfernung zwischen dem Spieler und dem Ziel innerhalb der festgelegten Grenzen liegt. Wenn alle Bedingungen erfüllt sind, wird die Rotation des Charakters entsprechend dem Ziel aktualisiert, um eine weiche und präzise Verfolgung zu gewährleisten.

Funktion SmoothRotationForLockOnPerTick



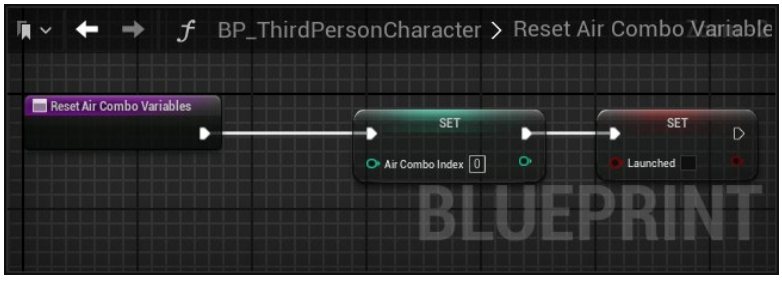
Die Funktion "Smooth Rotation for Lock on Per Tick" sorgt dafür, dass der Charakter während des Lock-Ons auf das Ziel schaut. Die Zielposition und die aktuelle Position des Charakters werden ermittelten und ein Offset für die Kamera berechnet, um dem Spieler eine optimale Ansicht zu bieten. Eine Interpolation der Rotationen sorgt für eine weiche Anpassung der Blickrichtung des Charakters in Echtzeit.

Event ResetState



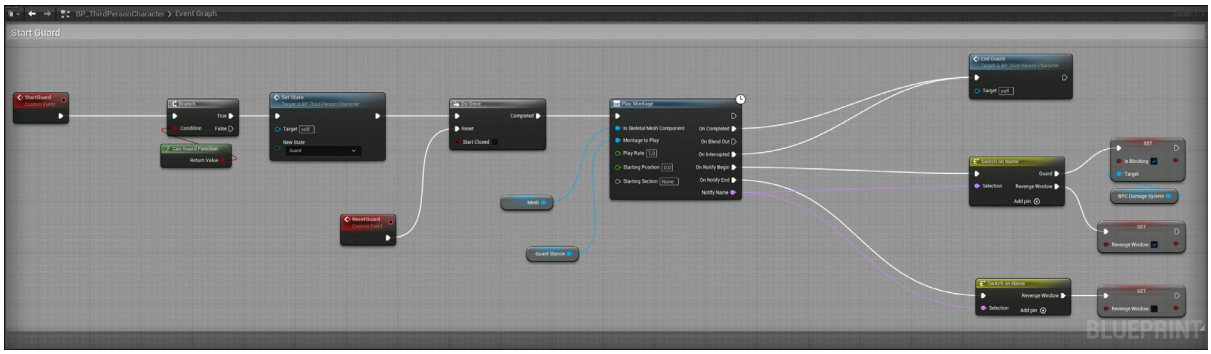
Das ResetState Event setzt den Spielerzustand zurück. Es stellt sicher, dass der Spieler erneut mit dem Fallen beginnt, wenn die InAirCombo abgebrochen wird und setzt alle relevanten Variablen und Zustände zurück, die innerhalb anderer Funktionen gesetzt worden sind und mit anderen Funktionen interferieren könnten. Dies hat sich als hilfreich erwiesen, um Bugs während des Projekts zu vermeiden. Das Event wird entweder über den dazugehörigen Anim Notify gestartet oder innerhalb anderer Funktionen, je nachdem wann ein Reset erfordert wird.

Funktion ResetAirComboVariables



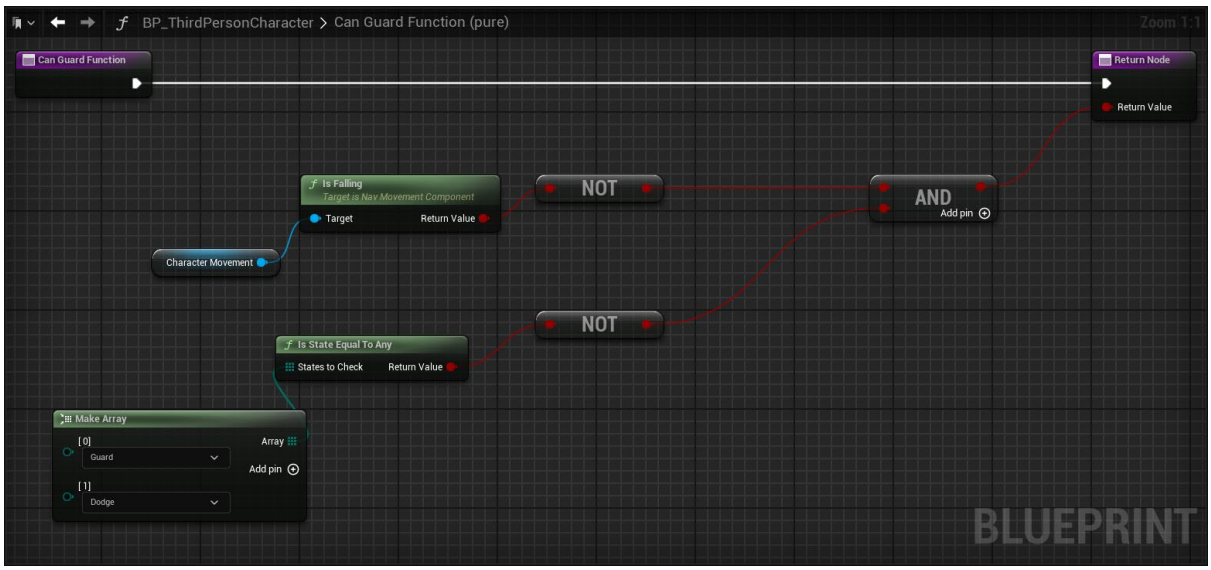
Diese Funktion stellt sicher, dass alle Variablen, die für Luft-Combos und deren Status relevant sind, auf ihre Standardwerte zurückgesetzt werden.

Event StartGuard



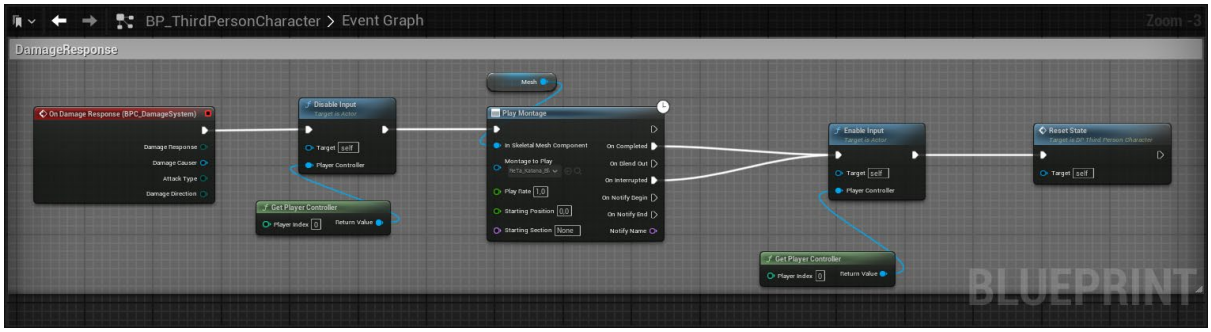
Das StartGuard Event im BP_ThirdPersonCharacter ermöglicht es dem Spieler, in den Guard-Zustand zu wechseln, indem es überprüft, ob der Spieler blocken kann. Es setzt den Zustand des Charakters auf "Guard", spielt die entsprechende Animation ab und setzt notwendige Variablen, um den Guard-Prozess zu steuern. Außerdem nutzt er den Anim Notify Guard und den Notify State Revenge Window um den Start des Blocks an das DamageSystem weiter zu geben und den Zeitraum für einen perfekten Block mit Gegenangriff festzulegen. Da diese nicht mehrfach gebraucht werden, werden dafür die Default Anim Notifys verwendet und in der Anim Montage benannt.

Funktion CanGuardFunction (pure)



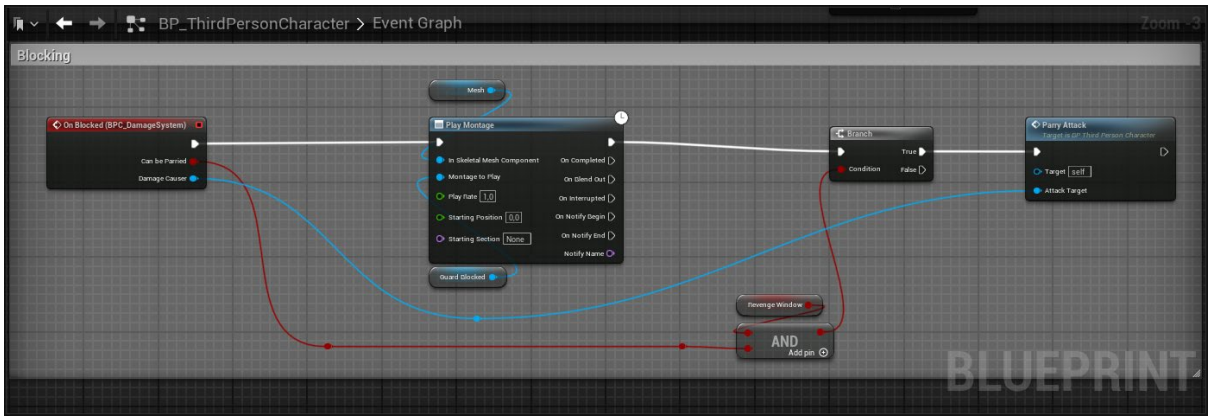
Die Funktion prüft, ob der Charakter nicht im Fallzustand ist und sich nicht im Guard- oder Dodge-Zustand befindet. Wenn beide Bedingungen erfüllt sind, wird zurückgegeben, dass der Charakter blocken kann.

Event DamageResponse



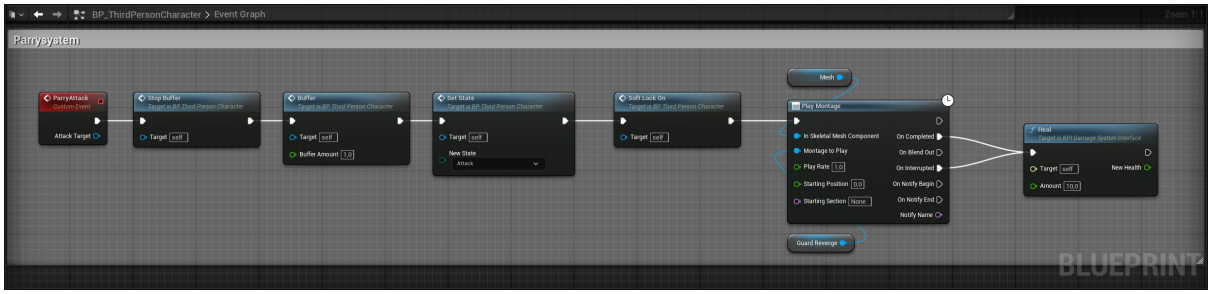
Das DamageResponse Event wird durch das neue DamageSystem ausgelöst. Es reagiert auf Schaden, den der Charakter erleidet. Es spielt eine Animation ab und deaktiviert vorübergehend die Eingaben des Spielers und führt danach einen StateReset durch.

Event OnBlocked



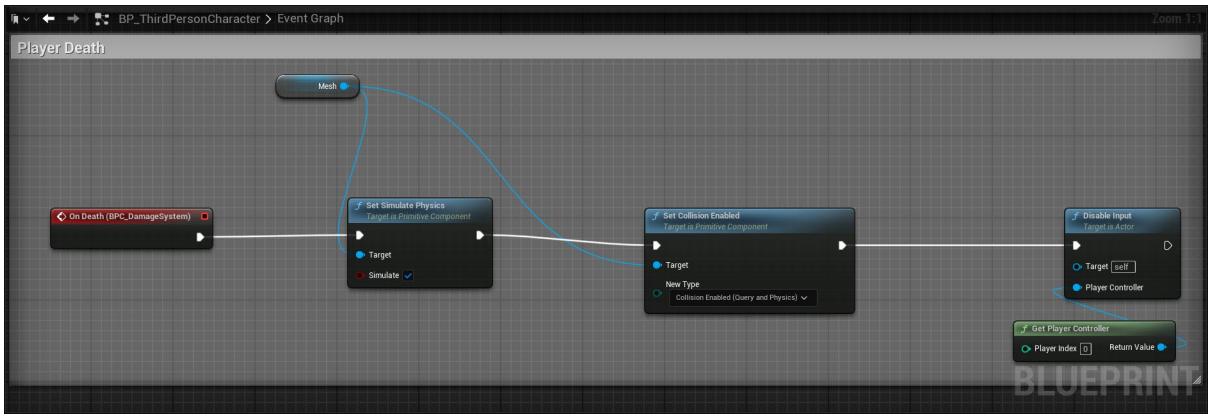
Dieses Event wird durch das neue DamageSystem ausgelöst und sorgt dafür, dass der Charakter bei einem erfolgreichen Block eine visuelle Blockreaktion zeigt und, falls die Bedingung das er sich innerhalb des Renge Windows befindet, erfüllt ist, eine Gegenattacke (GuardRevenge) ausführt.

Event ParryAttack



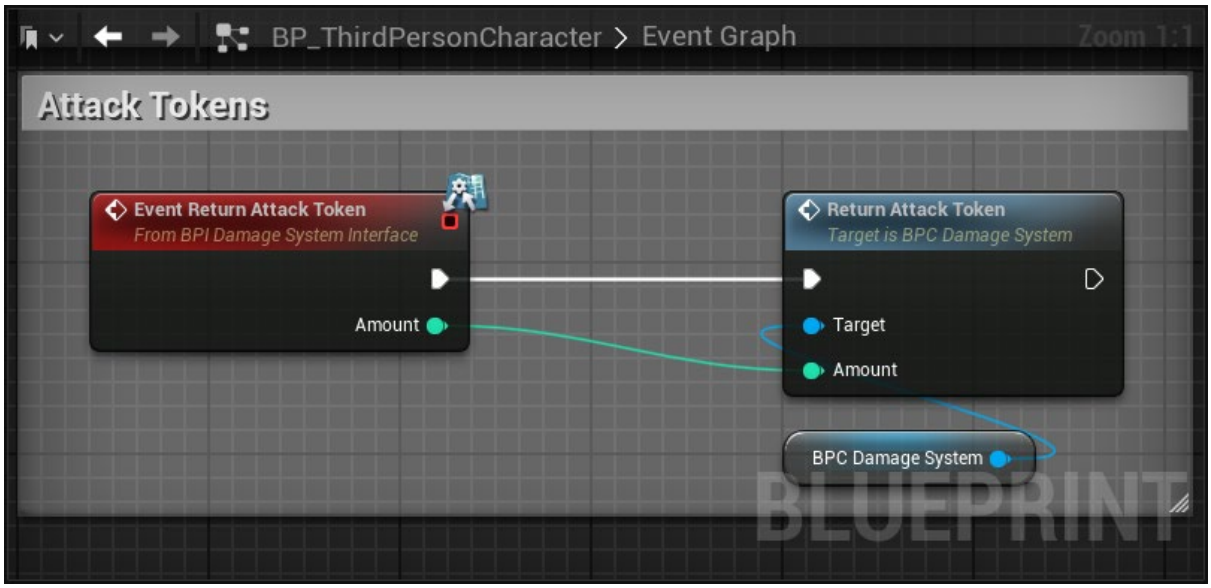
Dieses Event stellt sicher, dass bei einem Parry-Angriff die entsprechenden Animationen abgespielt, der Zustand des Charakters geändert und eine Heilung angewendet wird. Zuerst stoppt es den Buffer und startet einen neuen. Der State wird auf Attack gesetzt. Soft Lock On wird aktiviert, um den Player auf den Gegner zu orientieren. Die Animation wird abgespielt und der Spieler wird um einen Teil seines Lebens geheilt.

Event OnDeath



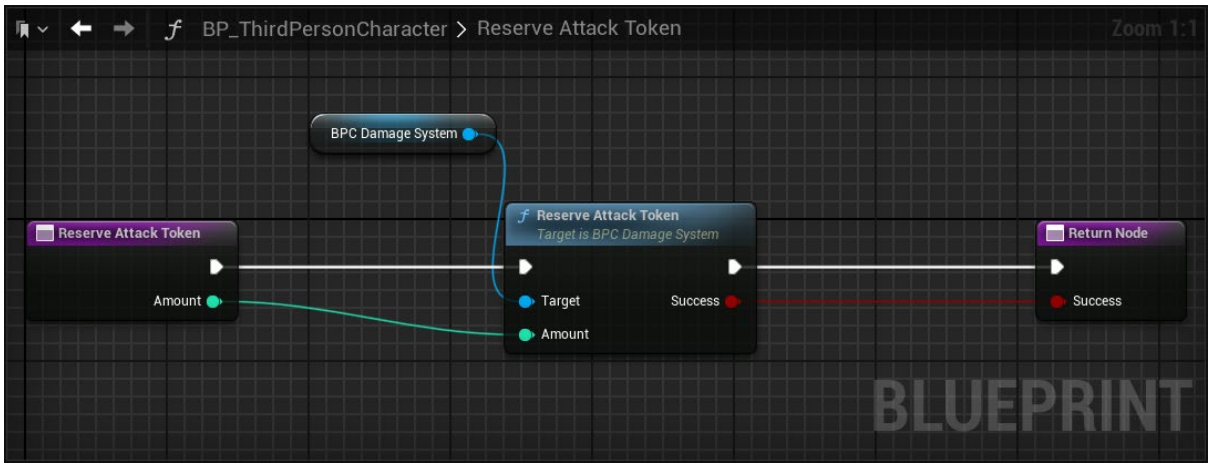
Dieses Event wird durch das neue DamageSystem gestartet, wenn das Leben des Players auf 0 fällt. Es stellt sicher, dass bei Tod des Charakters die Physik- und Kollisionssimulation aktiviert, wird um ein Ragdoll verhalten zu erzeugen und die Eingaben des Spielers deaktiviert werden.

Event ReturnAttackToken



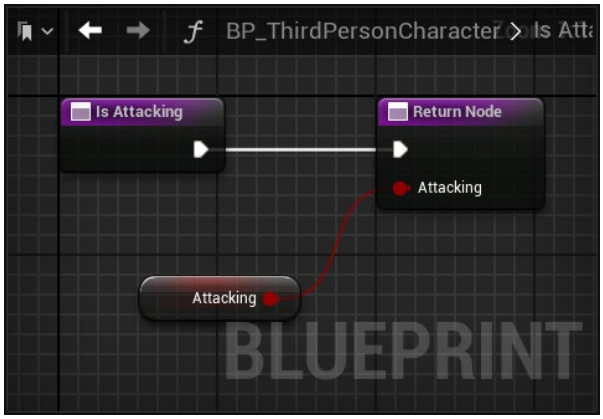
Das Return Attack Token Event wird verwendet, um Angriffstokens an das BPC Damage System zurückzugeben. Es wird vom BPI Damage System Interface aufgerufen und leitet die Menge der zurückgegebenen Tokens an das BPC Damage System weiter.

Interface ReserveAttackToken



Das Reserve Attack Token Interface im BP_ThirdPersonCharacter dient zur Interaktion mit dem BPC Damage System, um Angriffstokens zu reservieren. Es nimmt den Betrag der zu reservierenden Tokens als Eingangsparameter und gibt einen Erfolgsstatus zurück, der anzeigt, ob die Reservierung erfolgreich war.

Interface IsAttacking

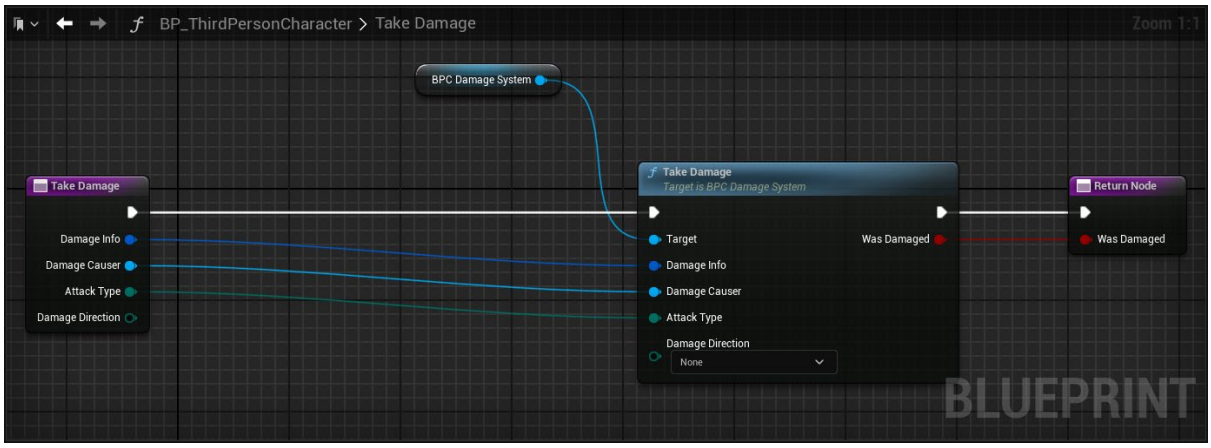


Die Is Attacking Interface-Funktion im BP_ThirdPersonCharacter gibt den aktuellen Angriffsstatus des Players zurück.

Interface IsDead

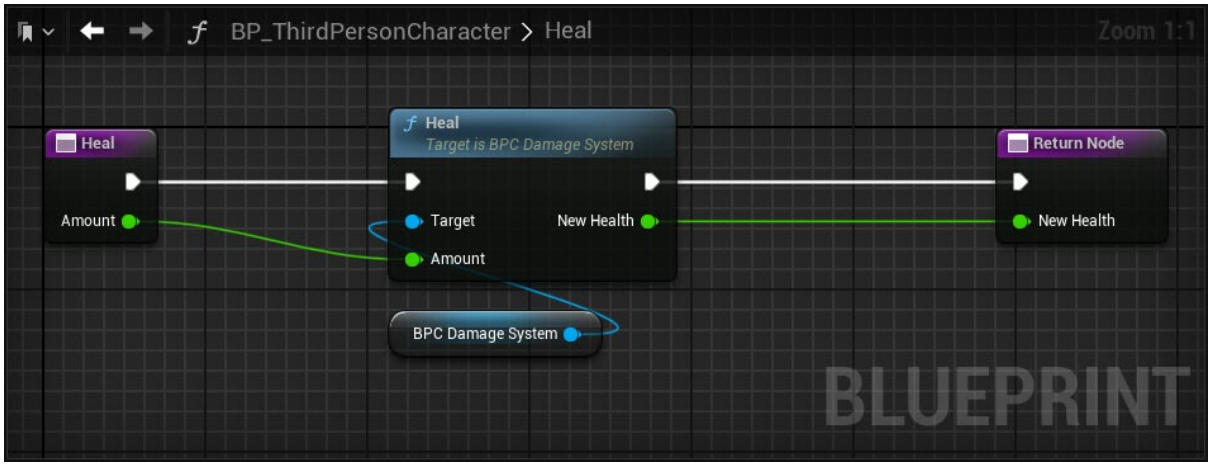
Die Is Dead Interface-Funktion im BP_ThirdPersonCharacter gibt den aktuellen Lebensstatus des Charakters zurück.

Interface TakeDamage



Die Take Damage Interface-Funktion im BP_ThirdPersonCharacter dient dazu, Schaden an den Charakter weiterzuleiten und zu verarbeiten. Sie übernimmt die Schadensinformationen und überprüft mithilfe des BPC Damage Systems, ob der Schaden erfolgreich verursacht wurde und leitet benötigte Informationen weiter.

Interface Heal

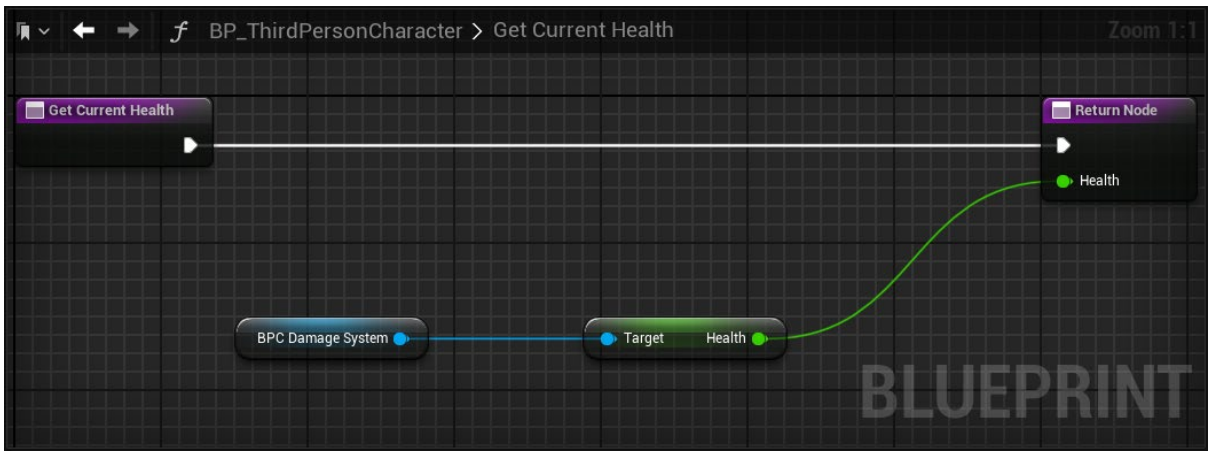


Die Heal Interface-Funktion im BP_ThirdPersonCharacter dient dazu, den Charakter zu heilen. Sie übernimmt den Heilungsbetrag und berechnet mithilfe des BPC Damage Systems die neue Gesundheit des Charakters.

Interface GetMaxHealth

Die GetMaxHealth Interface-Funktion im BP_ThirdPersonCharacter dient dazu, die maximale Gesundheit des Charakters abzurufen. Sie holt den Wert vom BPC Damage System und gibt ihn zurück.

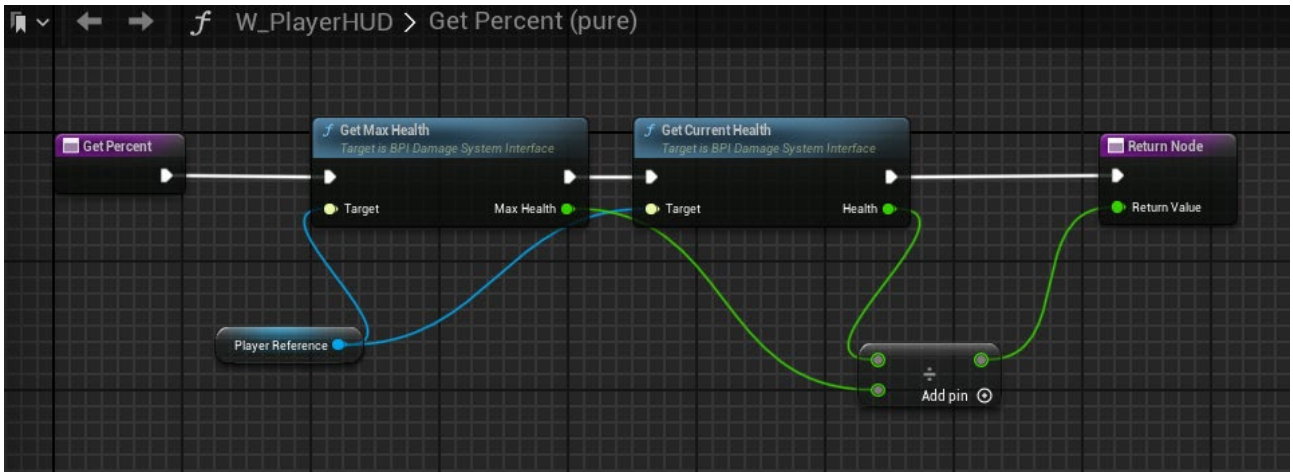
Interface GetCurrentHealth



Die GetCurrentHealth Interface-Funktion im BP_ThirdPersonCharacter dient dazu, die aktuelle Gesundheit des Charakters zu ermitteln, indem sie die Health-Variable aus dem BPC Damage System abruft und zurückgibt.

Widgets

W_PlayerHUD



Ein simples Widget das auf dem BP_ThirdPersonCharacter liegt. Enthält ein Canvas Panel mit einer Progress Bar und eine GetPercent Funktion die das aktuelle Leben als Anteil von 1, also Prozent ausrechnet. Die Progress Bar nutzt diese Funktion, um sich zu aktualisieren.

Enemies

Da alle Enemy Typen das Mesch des Unreal Engine Manny nutzen, wurden die Materials des Manny dupliziert und farblich je nach Enemy Type angepasst um im Spiel differenzierbar zu sein.

Animation

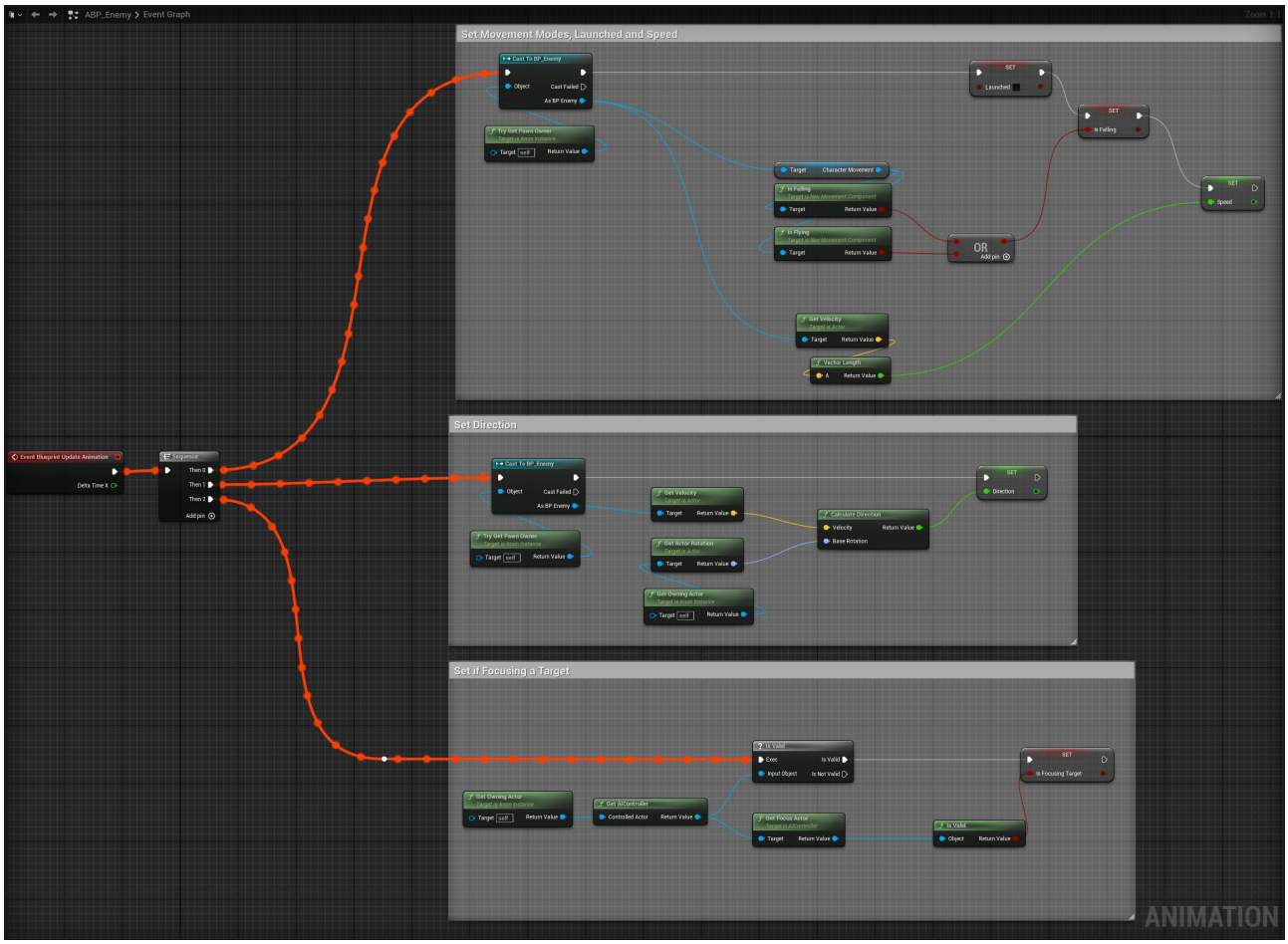
Retargeten der Animation Assets

Die Animationen der Enemies wurden mit dem vorgefertigten Retargeter der Unreal Engine für den Unreal Engine Manny kompatibel gemacht. Diese Animationen stammen aus denselben Packs wie die des Players, abgesehen von einigen speziellen Animationen für den RangedEnemy, die durch ein paar Animationen von Mixamo ergänzt wurden.

ABP_Enemy (Animation Blueprint der Melee Enemies)

Die Basis des ABP_Enemy ist ein angepasstes Duplikat des ABP_Player. Da sich aufgrund dessen einige Bestandteile überschneiden werden hier nur die Unterschiede zum ABP_Player dokumentiert.

Event Graph



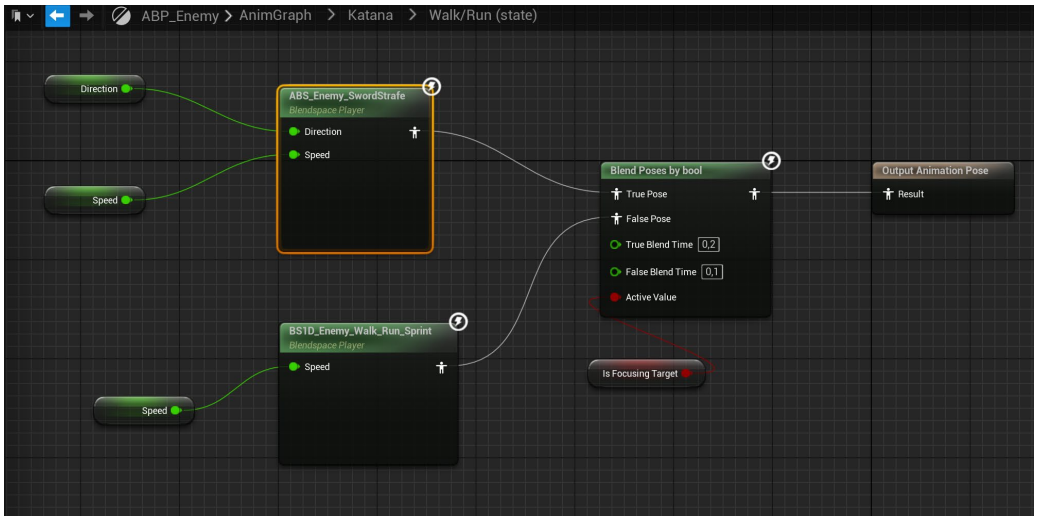
Der Event Graph im ABP_Enemy Animations-Blueprint aktualisiert regelmäßig die Bewegungsmodi, die Geschwindigkeit, die Richtung und ob der Feind auf ein Ziel fokussiert ist, indem es den Zustand des Charakters abfragt und die entsprechenden Variablen im Animations-Blueprint setzt.

1. **Set Movement Modes, Launched and Speed:**
    - Holt den Pawn Owner und castet ihn zu BP_Enemy.
    - Überprüft, ob der Charakter fällt oder fliegt, und setzt entsprechend die Variablen Launched und IsFalling.
    - Berechnet die Geschwindigkeit des Charakters und setzt die Variable Speed.
2. **Set Direction:**
    - Holt den Pawn Owner und castet ihn zu BP_Enemy.
    - Holt die Geschwindigkeit des Charakters und berechnet die Richtung basierend auf Geschwindigkeit und Rotation.
    - Setzt die Variable Direction.
3. **Set if Focusing a Target:**
    - Holt den Owning Actor und prüft, ob er ein gültiges Ziel fokussiert.

- Setzt die Variable IsFocusingTarget entsprechend.

Walk/Run State

Der Anim Graph und die Rules im Anim Graph des ABP_Enemy sind identisch mit denen innerhalb des ABP_Player. Lediglich innerhalb des Walk/Run States gab es Änderungen. Dabei wurden einfach nur die BlendSpaces gegen neue Ausgetauscht um andere Animationen und Werte zu verwenden.



BS1D_Enemy_Walk_Run_Sprint

Der BS1D_Enemy_Walk_Run_Sprint BlendSpace ist ein 1D BlendSpace, der die normale Fortbewegung animiert und von den Werten dem BS1D_Player_Walk_Run_Sprint BlendSpace gleicht.

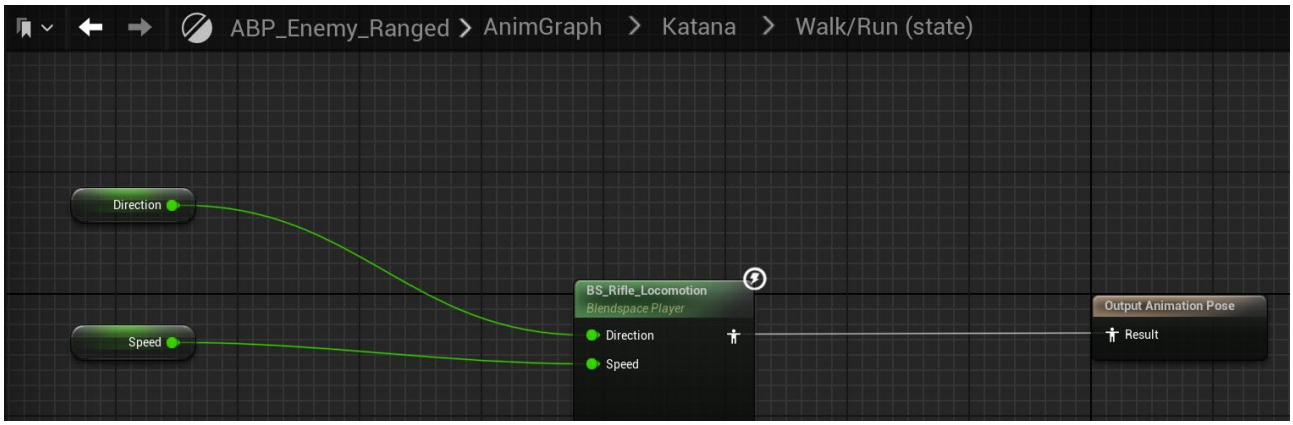
ABS_Enemy_DirectionalWalk

Der ABS_Enemy_SwordStrafe ist ein normaler BlendSpace, der die Bewegung des Melee Enemys innerhalb des Strafe-Modus animiert.

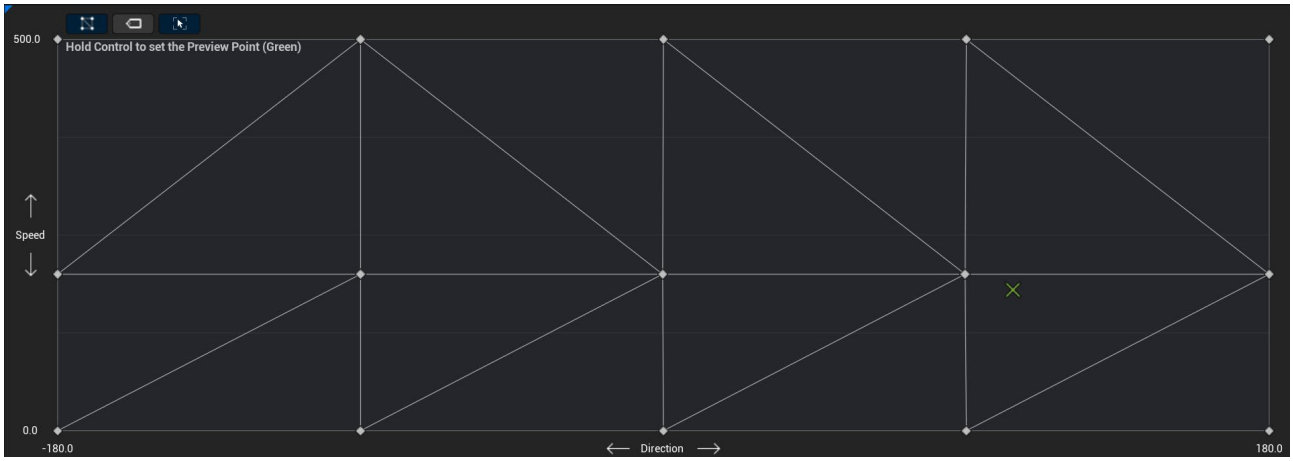
ABP_Enemy_Ranged (Animation Blueprint der Ranged Enemies)

Dieser Animation Blueprint ist eine Kopie des ABP_Enemy Blueprints mit nur einer Abweichung im Walk/Run State. Da der RangedEnemy immer die gleiche Bewegungsart haben kann, unabhängig davon, ob er den Player anvisiert oder nicht, wird nur ein BlendSpace verwendet. Dieser BlendSpace nutzt Direction und Speed, um in die entsprechende Bewegung zu blenden.

Walk/Run State



ABS_Rifle_Locomotion



Bei der Laufbewegung außerhalb des Targets ist die Direction immer immer Mittelfeld (0) weil der Enemy sich streng genommen nur in die Blickrichtung Bewegt und damit die Bewegungsrichtung gleich bleibt.

Enemy Controller

In dem Projekt sollten mehrere Gegnertypen implementiert werden: ein Fernkampfgegner, dessen Angriffe am besten durch Ausweichen abgewehrt werden, und mehrere Nahkampfgegner, die nur Attacken eines spezifischen Elements zulassen und andere Attacken blocken. Der Plan war, zum Ende des Projektes einen Boss zu erstellen, der die Verhaltensweisen der anderen Gegnertypen kombiniert und erweitert. Dieser Boss sollte einen Stance-Balken haben, der sinkt, wenn man ihn mit den richtigen Elementen angreift. Ist der Balken ganz unten, kann man ihm für kurze Zeit mit starken Attacken Schaden zufügen. Bei einem vollständigen Stance-Bruch sollte der Player ihn in die Luft schlagen können, um eine In-Air-Attack-Kombo auszuführen. Leider fehlte am Schluss die Zeit zur Umsetzung des Bossgegners, sodass es bei den normalen Gegnertypen blieb.

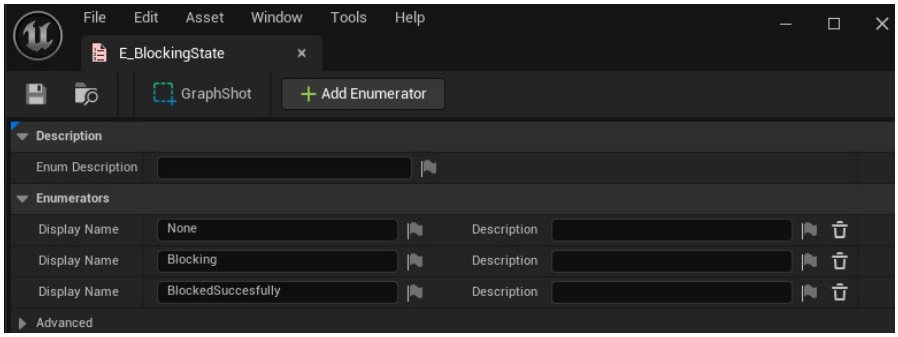
Es gibt den Melee-Gegnertypen, von dem es vier Varianten gibt, die jeweils nur von einem bestimmten Element Schaden nehmen können. Der Ranged Enemy hingegen kann von jedem Element Schaden nehmen, hält sich aber absichtlich auf Distanz, geht in Deckung und heilt sich, wenn sein Leben zu niedrig ist.

Zur Umsetzung der Gegner werden Character Blueprints, Behavior Trees, ein Detour Crowd AIController (definiert als AI Controller Class im Character BP der Enemys), ein NavMesh, ein zusätzliches Interface für gegnerübergreifende Strukturen und das Environment Query System der Unreal Engine genutzt. Zudem nutzen auch die Gegner das DamageSystem mit dem dazugehörigen Interface.

Innerhalb der Behavior Trees werden Custom Tasks, Decorator und ein Service verwendet, um die Logik abzuhandeln.

Enumerators

E_BlockingState

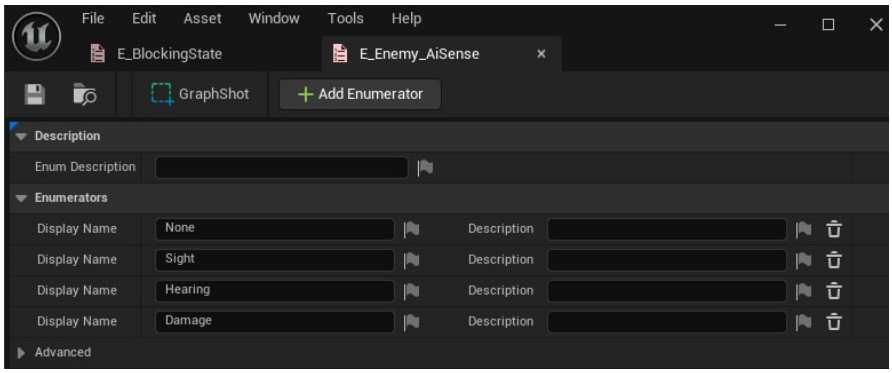


Der E_BlockingState Enumerator definiert verschiedene Zustände für das Blockieren der Nahkampf Feinde im Spiel.

1. **None:**
    - Der Zustand, wenn kein Blocken aktiv ist.
2. **Blocking:**
    - Der Zustand, wenn der Feind aktiv versucht, einen Angriff zu blockieren.
3. **BlockedSuccessfully:**
    - Der Zustand, wenn der Charakter erfolgreich einen Angriff geblockt hat.

Diese Zustände werden verwendet, um die Blockmechanik der Feinde im Spiel zu steuern und zu überprüfen, in welchem Zustand sich der Feind aktuell befindet.

E_Enemy_AiSense

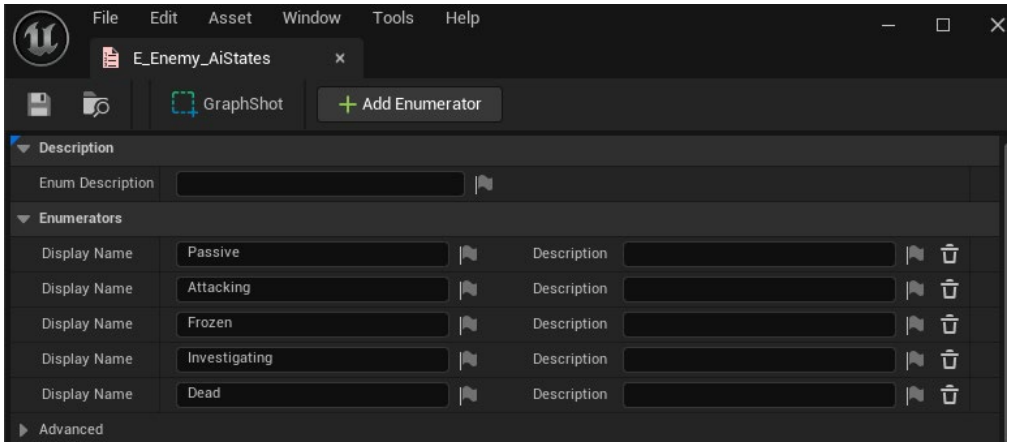


Der E_Enemy_AiSense Enumerator definiert verschiedene Sinneswahrnehmungen für die KI der Feinde im Spiel.

1. **None:**
    - Der Zustand, wenn keine spezifische Sinneswahrnehmung aktiv ist.
2. **Sight:**
    - Der Zustand, wenn der Feind visuelle Wahrnehmungen nutzt, um den Spieler zu erkennen oder zu verfolgen.
3. **Hearing:**
    - Der Zustand, wenn der Feind auditive Wahrnehmungen nutzt, um auf Geräusche zu reagieren und den Spieler zu lokalisieren.
4. **Damage:**
    - Der Zustand, wenn der Feind Schaden erleidet und darauf reagiert.

Diese Zustände werden verwendet, um die Wahrnehmungsmechanik der Feind-KI zu steuern und zu überprüfen, wie der Feind auf verschiedene Reize in der Spielwelt reagiert.

E_Enemy_AiStates

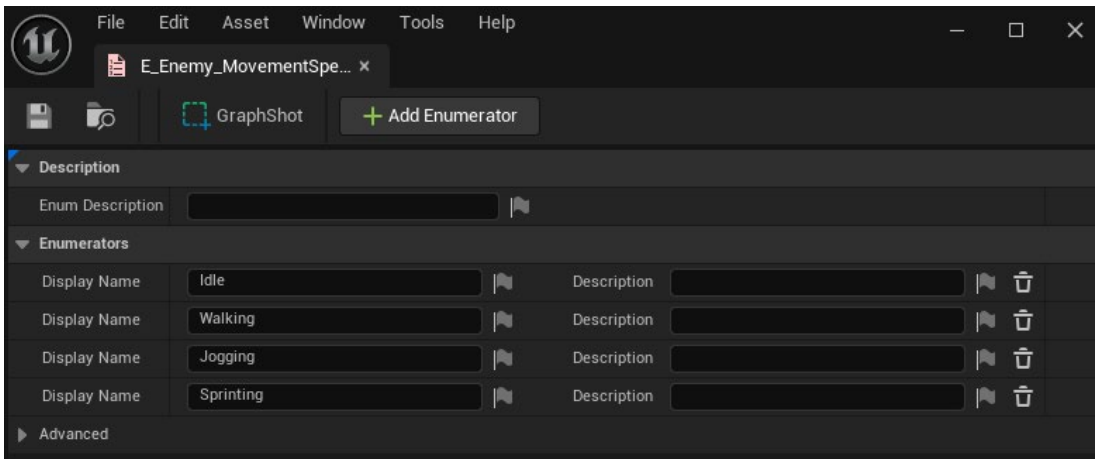


Der E_Enemy_AiStates Enumerator definiert verschiedene Zustände für die KI der Feinde im Spiel.

1. **Passive:**
    - Der Feind befindet sich in einem ruhigen Zustand und führt keine aggressiven Handlungen aus. Er könnte patrouillieren oder einfach stillstehen.
2. **Attacking:**
    - Der Feind befindet sich im Angriffszustand und greift den Spieler aktiv an.
3. **Frozen:**
    - Der Feind ist eingefroren und kann sich weder bewegen noch angreifen. Das Verhalten im Behavior Tree ist unterbrochen.
4. **Investigating:**
    - Der Feind untersucht seine Umgebung, aufgrund von Geräuschen oder visuellen Hinweisen, um die Quelle der Störung zu finden.
5. **Dead:**
    - Der Feind ist tot und kann keine weiteren Aktionen ausführen.

Diese Zustände werden verwendet, um die Verhaltenslogik der Feind-KI zu steuern und zu definieren, wie die Feinde in verschiedenen Situationen reagieren sollen.

E_Enemy_MovementSpeeds



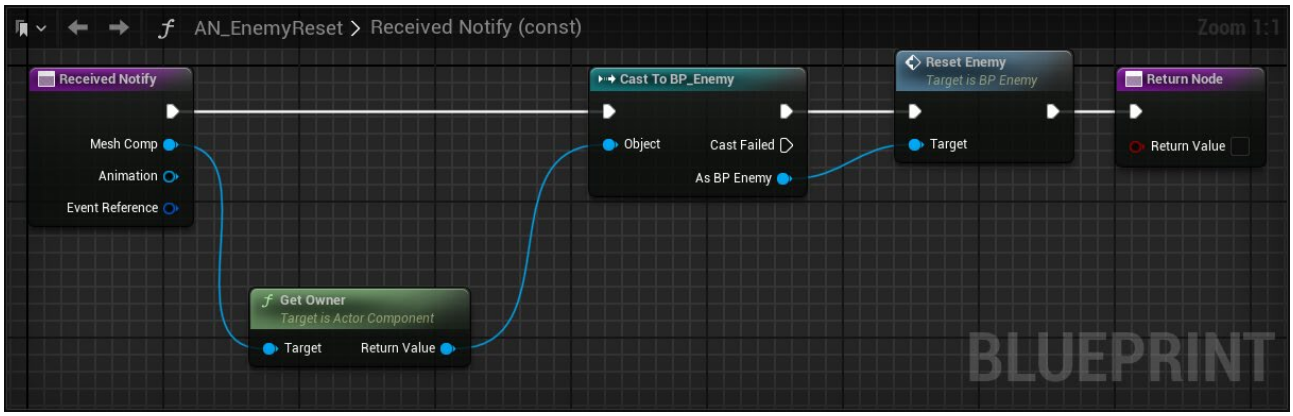
Der E_Enemy_MovementSpeeds Enumerator definiert verschiedene Bewegungsgeschwindigkeiten für die Feinde im Spiel.

1. **Idle:**
    - Der Feind steht still und bewegt sich nicht.
2. **Walking:**
    - Der Feind bewegt sich langsam, indem er geht.
3. **Jogging:**
    - Der Feind bewegt sich mit mittlerer Geschwindigkeit, indem er joggt.

4. **Sprinting:**
    - Der Feind bewegt sich schnell, indem er sprintet.

Diese Geschwindigkeiten werden verwendet, um das Bewegungsverhalten der Feinde zu steuern.

AN_EnemyReset AnimNotify



Das AN_EnemyReset Anim Notify wird am Ende der "GotHit" Animationen der Feinde ausgelöst. Es stellt sicher, dass der Feind nach einer Trefferanimation zurückgesetzt wird, um in den ursprünglichen Zustand zurückzukehren.

Interface BPI_EnemyAI

Das BPI_EnemyAI Interface definiert eine Reihe von Funktionen, die das Verhalten der Feinde im Spiel steuern. Diese Funktionen ermöglichen eine flexible und modulare Steuerung der Feind-AI, indem sie verschiedene Aspekte des Verhaltens und der Interaktion vordefinieren.

GetPatrolRoute

Output: PatrolRoute (BP_PatrolRoute)

Holt die Patrouillenroute des Feindes.

SetMovementSpeed

Input: Speed (E_EnemyMovementSpeeds)

Output: SpeedValue (Float)

Beschreibung: Setzt die Bewegungsgeschwindigkeit des Feindes und gibt den Wert der Geschwindigkeit zurück.

GetIdealRange

Outputs: AttackRadius (Float), DefendRadius (Float)

Holt den idealen Angriffs- und Verteidigungsbereich des Feindes.

EquipWeapon

Rüstet den Feind mit einer Waffe aus.

UnequipWeapon

Entfernt die Waffe des Feindes.

DefaultAttack

Input: AttackTarget (Actor)

Führt den Standardangriff des Feindes auf das angegebene Ziel aus.

AttackStart

Inputs: AttackTarget (Actor), TokensNeeded (Integer)

Output: Success (Boolean)

Startet einen Angriff auf das angegebene Ziel und gibt an, ob der Angriff erfolgreich gestartet wurde.

AttackEnd

Input: AttackTarget (Actor)

Beendet einen Angriff auf das angegebene Ziel.

StoreAttackTokens

Inputs: AttackTarget (Actor), Amount (Integer)

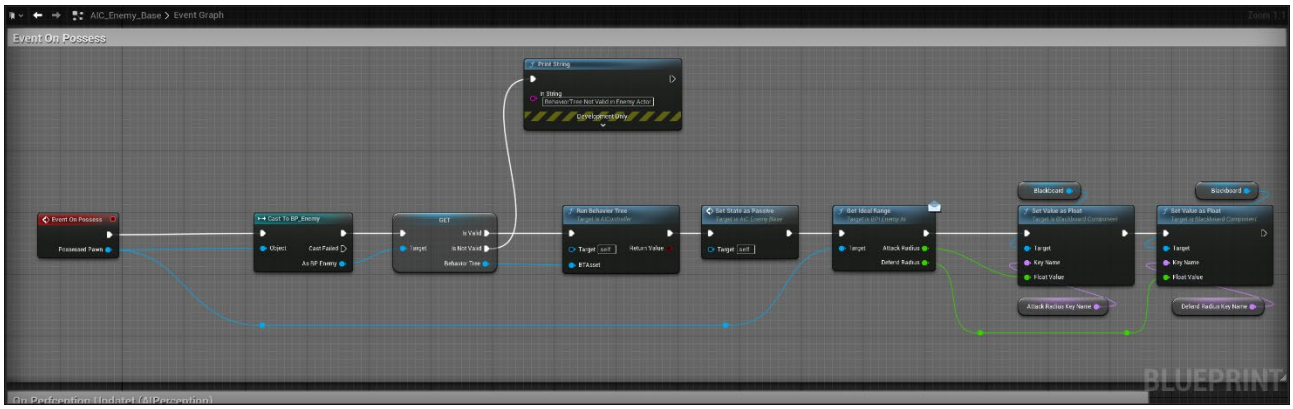
Speichert die Angriffstokens des Feindes für das angegebene Ziel.

Detour Crowd AIController AIC_Enemy_Base

Für die Bewegungslogik der Feinde wird ein Detour Crowd AIController verwendet. Diese Blueprint-Klasse eignet sich besonders gut zur Verwaltung der Bewegungen mehrerer NPCs, die sich nebeneinander bewegen. Dies wird durch Funktionen wie dynamische Kollisionsvermeidung und dynamische Pfadfindung erreicht, die auch sich bewegende Objekte einbeziehen. Da das gewünschte Gegnerverhalten eine Art Gruppenkampf ist, bei dem die Feinde den Spieler umzingeln und sich dynamisch um ihn herumbewegen, ist dieser AIController die beste Wahl als Basis für die Erstellung der AI Controller Class der Gegner. Die erstellte Klasse wird im Character Blueprint gesetzt, in diesem Fall im BP_Enemy, der Parent Class der Character Blueprints der Gegner.

Im AIC_Enemy Base findet die Navigation des Gegners statt und aktualisiert außerdem Werte im Blackboard des Gegners. Diese Werte werden dann im Behavior Tree genutzt, um das Verhalten der Gegner zu steuern. Dabei ist es wichtig, beim Definieren der Namen der Variablen keine Schreibfehler zu machen, da die Variable sonst nicht zugeordnet werden kann.

Event OnPossess



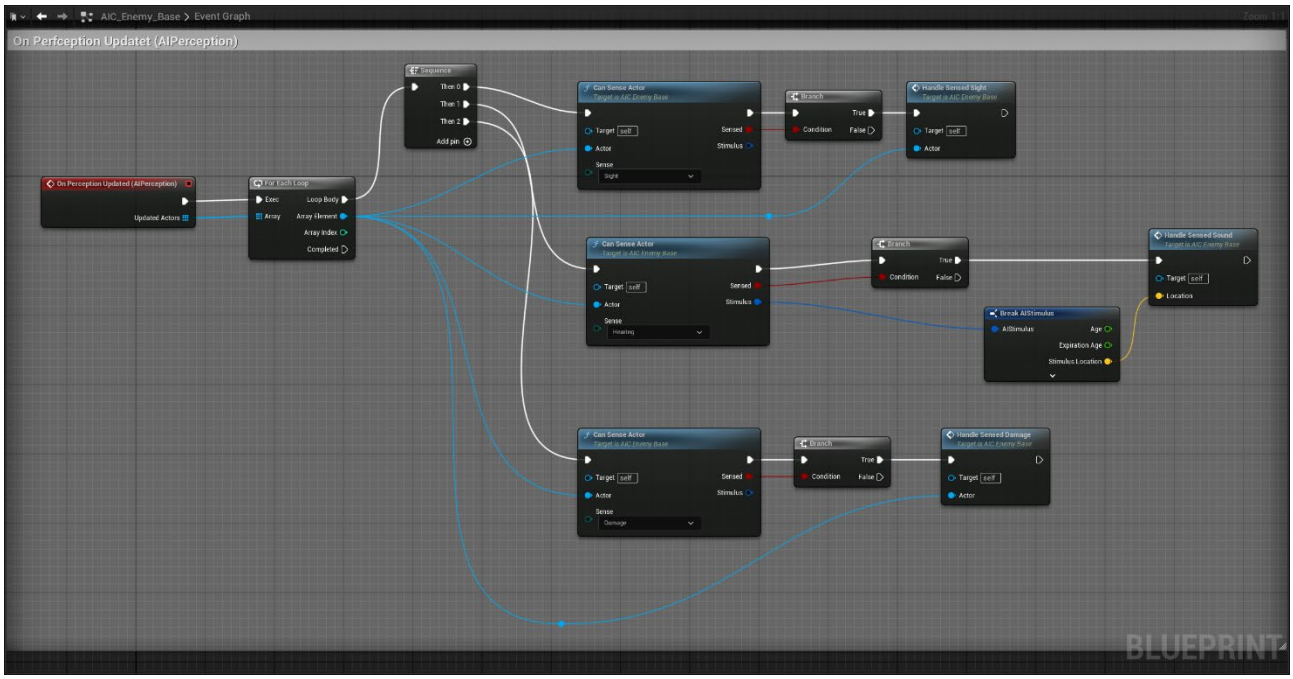
Dieses Event wird ausgelöst, wenn der AI-Controller die Kontrolle über einen Charakter übernimmt.

1. **Event On Possess:**
    - Startet das Event und erhält das besessene Pawn als Input.
2. **Cast to BP_Enemy:**
    - Versucht, das besessene Pawn in den BP_Enemy zu casten.
    - Wenn der Cast fehlschlägt, wird nichts weiter unternommen.
3. **Print String (Fehlerfall):**
    - Wenn der Cast fehlschlägt, wird eine Debug-Nachricht ausgegeben: "BehaviorTree Not Valid in Enemy Actor".
4. **Get (Behavior Tree):**
    - Holt den Behavior Tree vom BP_Enemy, wenn der Cast erfolgreich war.
    - Überprüft, ob der Behavior Tree gültig ist.
5. **Run Behavior Tree:**
    - Startet den Behavior Tree des BP_Enemy im AI-Controller, wenn der Behavior Tree gültig ist.
6. **Set State as Passive:**
    - Setzt den Zustand des AI-Controllers auf "Passiv".
7. **Get Ideal Range (von BPI_Enemy_AI):**
    - Ruft die Idealbereiche für Angriffs- und Verteidigungsradien vom Enemy AI-Interface ab.
    - Outputs: Attack Radius und Defend Radius.
8. **Set Value as Float (Blackboard Component):**
    - Setzt den Angriffsradius im Blackboard des AI-Controllers auf den abgerufenen Wert.

- Setzt den Verteidigungsradius im Blackboard des AI-Controllers auf den abgerufenen Wert.

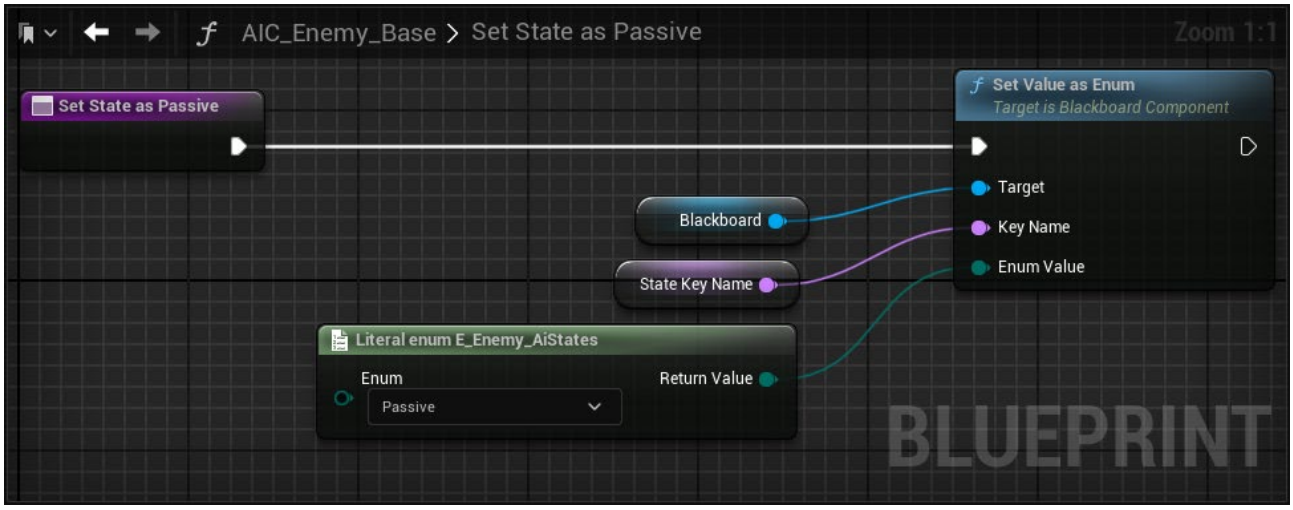
Dieser Ablauf ermöglicht es dem AI-Controller, die Kontrolle über einen Enemy-Charakter zu übernehmen, den entsprechenden Behavior Tree zu starten und wichtige Radiuswerte für Angriffs- und Verteidigungsverhalten zu initialisieren.

Event OnPerceptionUpdated(AIPerception)



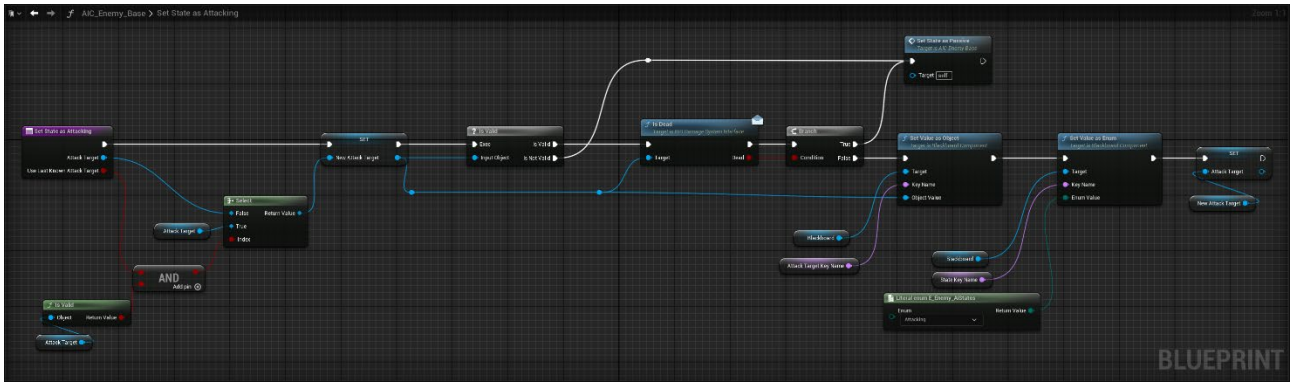
Dieses Event ermöglicht es dem AI-Controller, die Wahrnehmung von Akteuren basierend auf verschiedenen Sinnen zu aktualisieren und entsprechend darauf zu reagieren. Auch wenn in dem Projekt noch nicht von allen Wahrnehmungen komplett Gebrauch gemacht wird, werden hier die Wahrnehmungen Sight, Hearing und Damage angebunden und die Verarbeitungsfunktionen werden entsprechend gestartet.

Funktion SetStateAsPassive



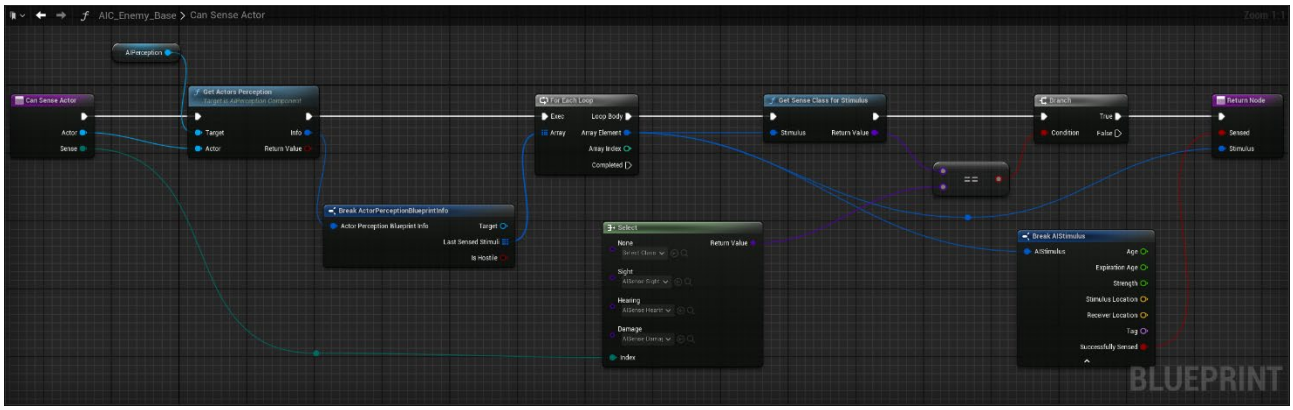
Diese Funktion ändert den AI-Zustand des Feindes in den passiven Modus, indem sie den entsprechenden Schlüsselwert im Blackboard auf den Enum-Wert "Passiv" setzt. Dies ist nützlich, um das Verhalten des Feindes dynamisch zu ändern, z.B. ihn in einen Zustand zu versetzen, in dem er keine aggressiven Handlungen ausführt.

Funktion SetStateAsAttacking



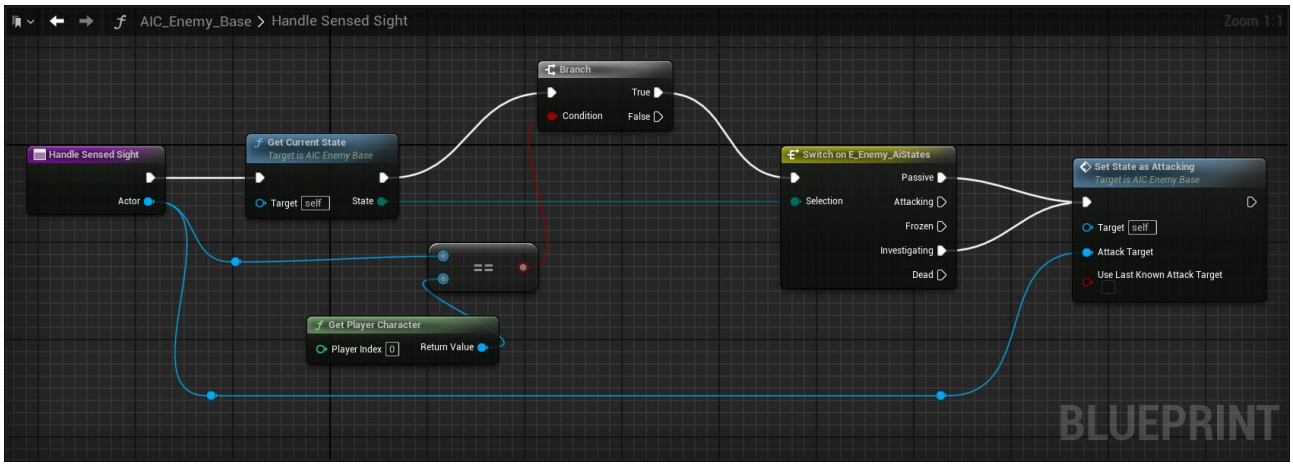
Diese Funktion wechselt den Zustand des Feindes auf "Angreifend" und legt das Angriffsziel fest. Es überprüft, ob das Ziel gültig und nicht tot ist, und aktualisiert entsprechend den Blackboard-Wert für das Angriffsziel und den Zustand. Wenn das Ziel tot oder nicht Valid ist, wird der Zustand auf "Passiv" zurückgesetzt.

Funktion CanSenseActor



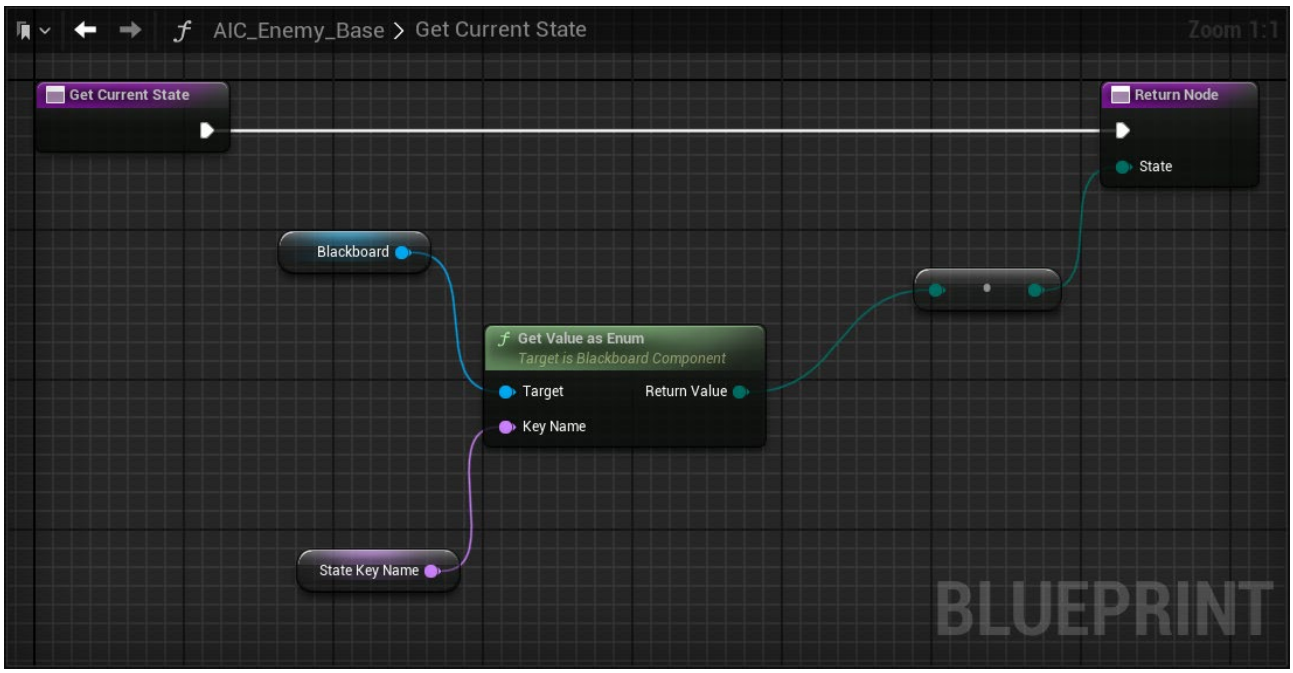
Diese Funktion überprüft, ob der AI-Controller einen bestimmten Actor mit einem bestimmten Sinn wahrnehmen kann. Es iteriert über die letzten wahrgenommenen Stimuli des Actors, überprüft die Klasse des Stimulus und vergleicht sie mit dem angegebenen Sinn. Wenn eine Übereinstimmung gefunden wird, gibt die Funktion zurück, dass der Actor wahrgenommen wurde, und liefert den entsprechenden Stimulus.

Funktion HandleSensedSight



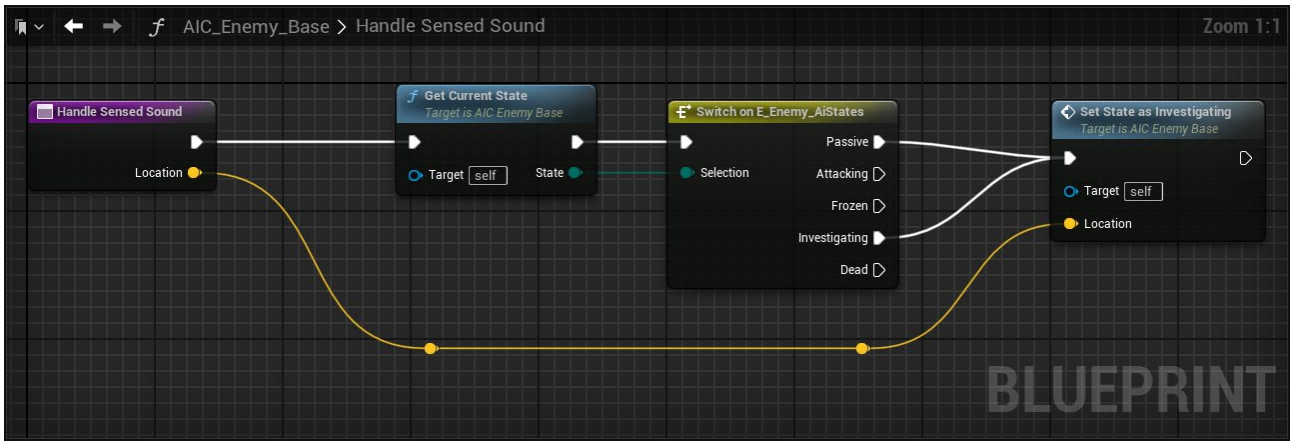
Diese Funktion überprüft, ob der AI-Controller den Player Character durch Sicht wahrgenommen hat, während er im Passive- oder Investigating-Zustand ist. Wenn dies der Fall ist, wird der AI-Zustand auf Attacking gesetzt und der Player Character wird als Angriffsziel festgelegt. Die verschiedenen möglichen Zustände des AI werden durch einen Switch-Node behandelt, um das richtige Verhalten zu bestimmen.

Funktion GetCurrentState



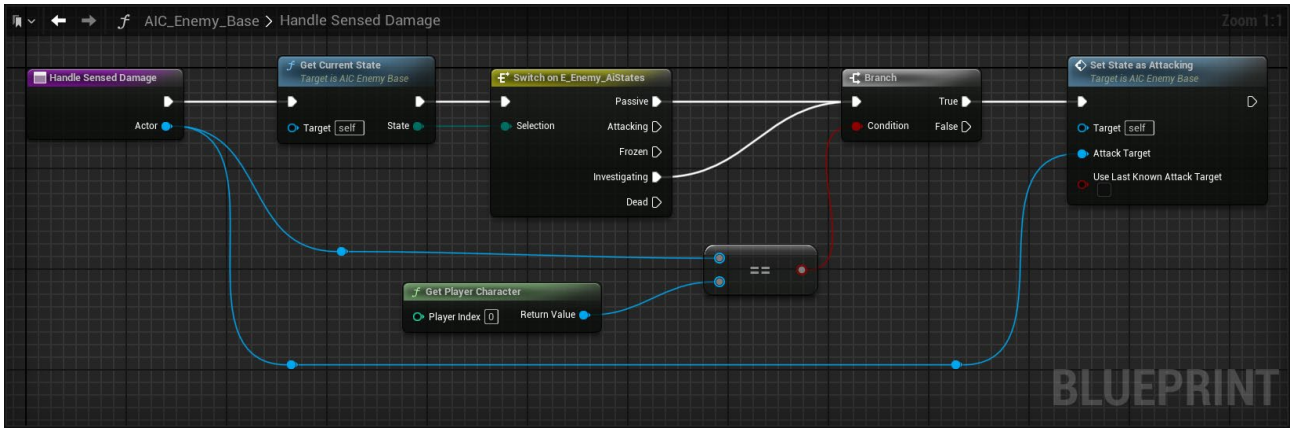
Diese Funktion ruft den aktuellen Zustand des AI-Controllers aus dem Blackboard ab. Der Zustand wird als Enum-Wert zurückgegeben. Dies ist nützlich, um basierend auf dem aktuellen Zustand des AI-Controllers entsprechende Entscheidungen zu treffen und das Verhalten des AI-Controllers zu steuern.

Funktion HandleSensedSound



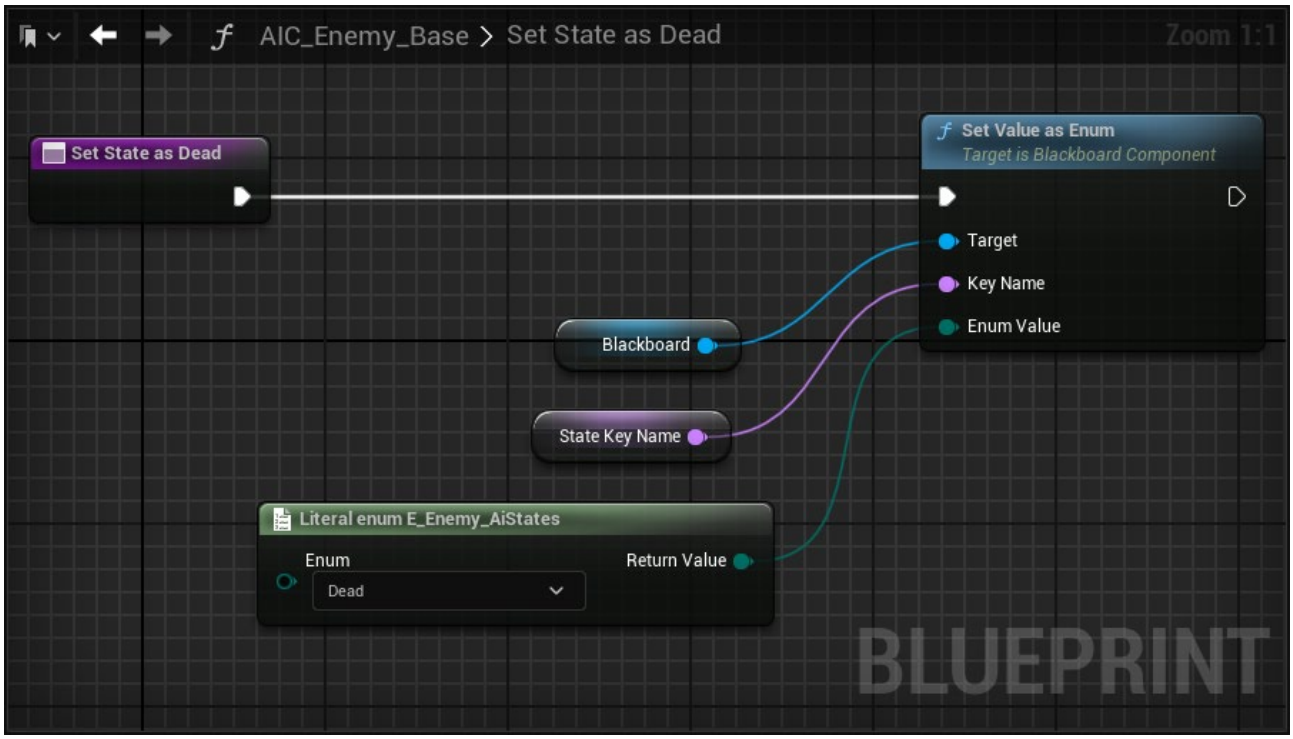
Diese Funktion überprüft den aktuellen Zustand des AI-Controllers, wenn ein Geräusch wahrgenommen wird. Abhängig vom aktuellen Zustand wird der AI-Controller in den Zustand Investigating versetzt und erhält die Position des Geräuschs. Dies ermöglicht es dem AI-Controller, auf Geräusche in der Umgebung zu reagieren und eine Untersuchung des Geräuschursprungs durchzuführen.

Funktion HandleSensedDamage



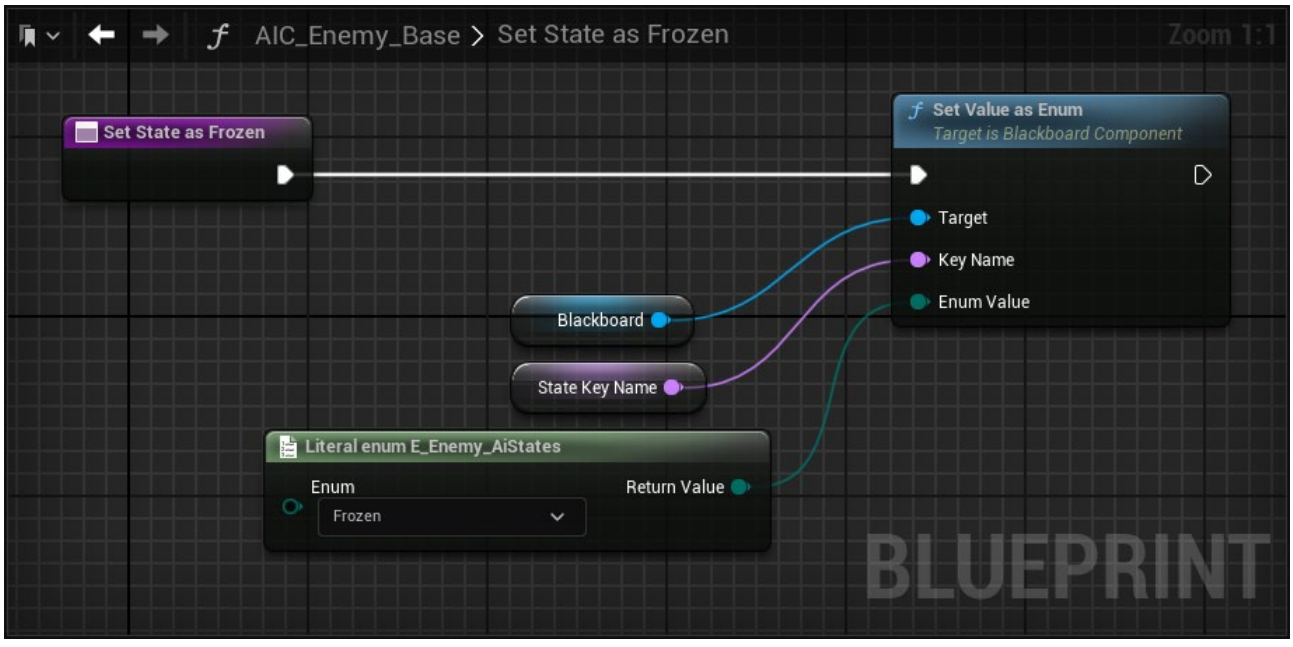
Diese Funktion überprüft den aktuellen Zustand des AI-Controllers, wenn ein Schaden erkannt wird. Wenn der Zustand Passive oder Investigating ist, wird der AI-Controller in den Zustand Attacking versetzt und erhält das Ziel des Angriffs. Dies ermöglicht es dem AI-Controller, auf Schadensereignisse zu reagieren und den Spieler anzugreifen.

Funktion SetStateAsDead



Diese Funktion wird verwendet, um den Zustand des AI-Controllers auf "Dead" zu setzen. Dies wird erreicht, indem der entsprechende Enum-Wert in der Blackboard-Komponente des Controllers gesetzt wird.

Funktion SetStateAsFrozen

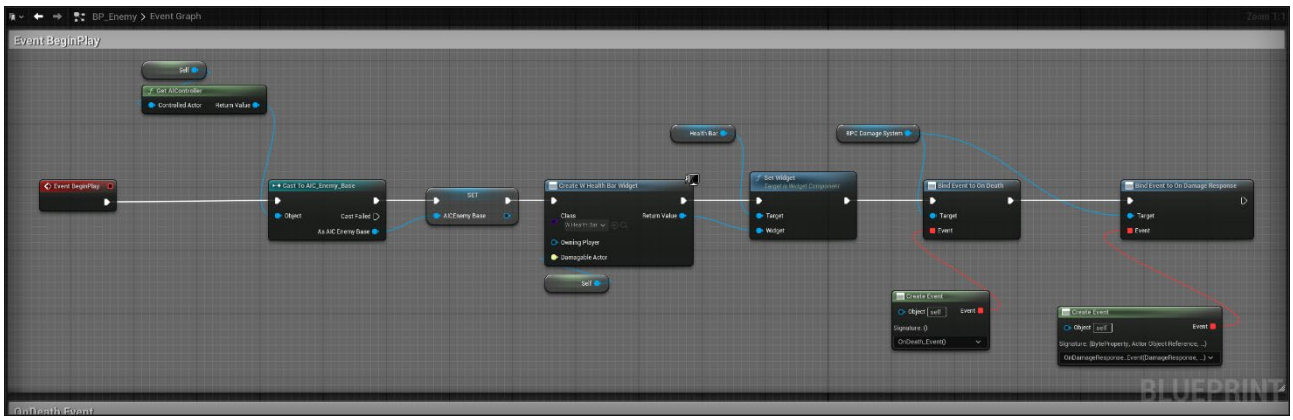


Diese Funktion wird verwendet, um den Zustand des AI-Controllers auf "Frozen" zu setzen. Dies wird erreicht, indem der entsprechende Enum-Wert in der Blackboard-Komponente des Controllers gesetzt wird. Dies ist nützlich, um das Verhalten des AI-Controllers zu ändern und ihn entsprechend reagieren zu lassen, wenn der Charakter eingefroren ist.

BP_Enemy

Der BP_Enemy ist der Parent Blueprint der verschiedenen Gegnertypen. In diesem Blueprint werden die gemeinsamen Bestandteile der Feinde definiert, die von allen Gegnern genutzt werden. Die Hit Reaktionen wurden bereits weit vor dem Enemy Behavior und dem Damage System erstellt und sind daher noch nicht in das Damage System migriert. Aufgrund dessen ist ihre Struktur und der Aufbau im Vergleich zu den neu erstellten Teilen der Blueprints relativ umständlich und unnötig groß. Leider blieb keine Zeit mehr für eine entsprechende Überarbeitung.

Event BeginPlay

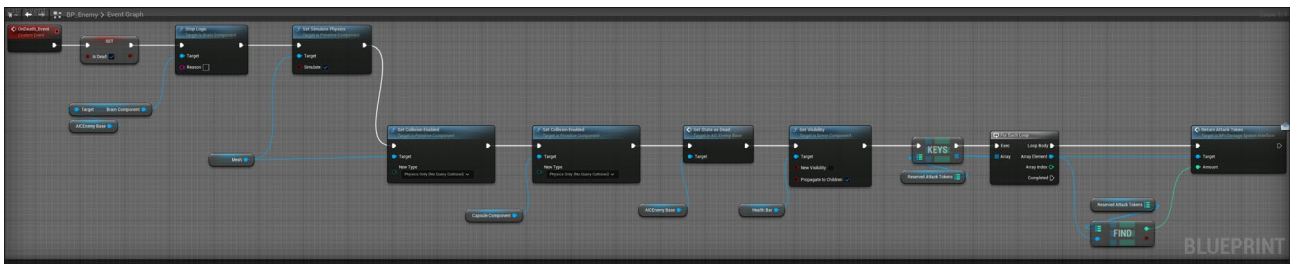


Beim Start des Spiels oder beim Erzeugen des Akteurs führt das "Event BeginPlay" folgende Aktionen aus:

- Holt den kontrollierten Akteur und überprüft, ob er vom Typ "AC_Enemy_Base" ist.
- Holt den Player Controller und erstellt ein Widget, das dem Player Controller zugewiesen wird.
- Initialisiert das Schadenssystem des Akteurs und bindet Events an die "On Death" und "On Damage Response"-Ereignisse des Schadenssystems, um entsprechende Reaktionen zu ermöglichen, wenn der Akteur stirbt oder Schaden nimmt.

Es stellt sicher, dass der Gegner korrekt initialisiert wird und auf Schaden und Tod angemessen reagieren kann, indem es die erforderlichen UI-Elemente und Eventbindungen einrichtet.

Event OnDeath_Event

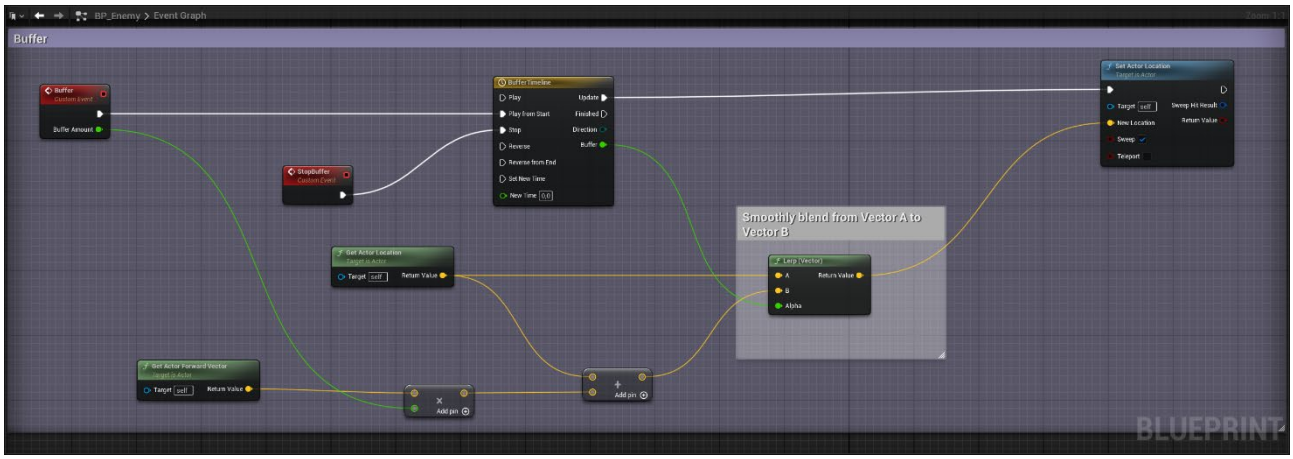


Beim Tod des Enemies führt das Event folgende Aktionen aus:

- Markiert den Akteur als tot.
- Stoppt alle AI-Prozesse.
- Aktiviert die Physiksimulation.
- Ändert die Kollisionseigenschaften der Meshs und Komponenten.
- Setzt den Zustand im "AC_Enemy_Base" auf tot.
- Macht das Health Bar Widget unsichtbar.
- Gibt alle reservierten Attack Tokens zurück.

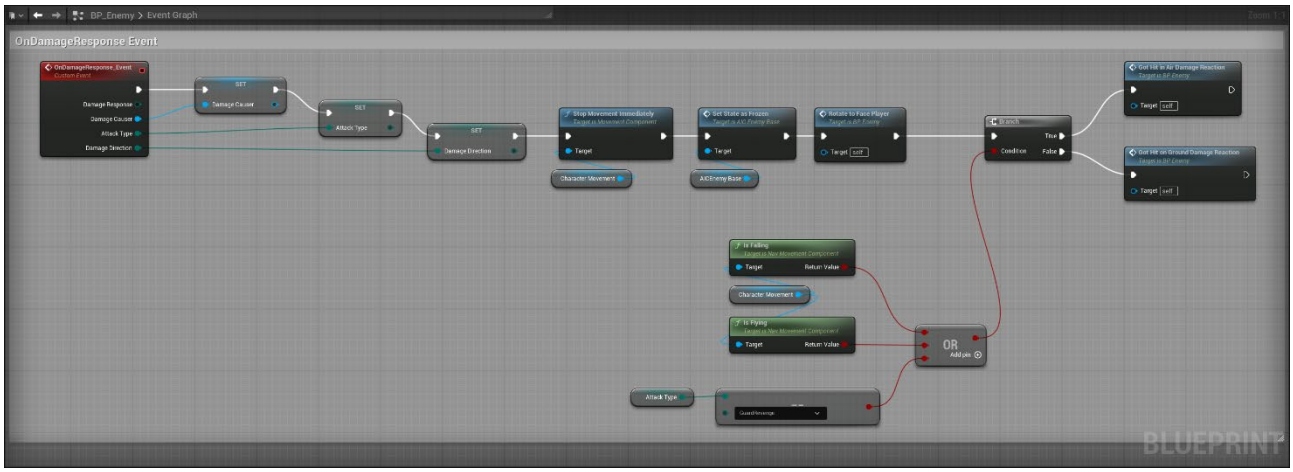
Dieses Setup stellt sicher, dass der Tod des Akteurs korrekt verarbeitet wird, indem alle notwendigen Zustandsänderungen, physikalischen Eigenschaften und Ressourcenzuweisungen gehandhabt werden. Die Änderung der Kollisionseigenschaften führt zu dem Typischen Ragdoll verhalten das man aus den meisten Videospielen kennt.

Event Buffer



Das Buffer Event ermöglicht eine sanfte Rückwärtsbewegung des Enemies über eine definierte Zeitspanne. Dies wird durch die Verwendung einer Timeline und der Interpolation der Position erreicht. Dieses Event gleicht dem Buffer aus dem BP_ThirdPersonCharacter nur das es bei dem Enemy meist mit negativen Werten verwendet wird, um eine Art Rückstoß zu erzeugen.

Event OnDamageResponse

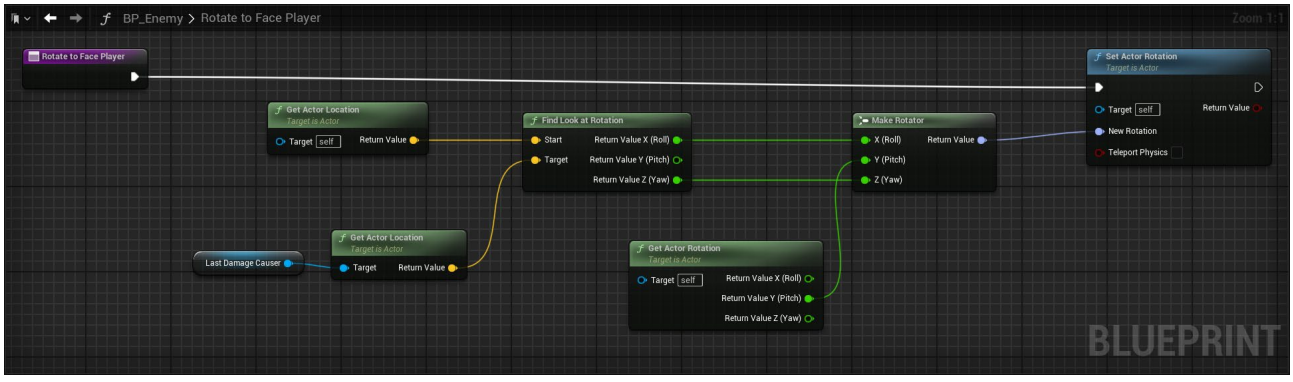


Das OnDamageResponse_Event wird ausgelöst, wenn der Charakter Schaden erleidet.

- Die relevanten Informationen zum Schaden werden in Variablen gespeichert.
- Die Bewegung des Charakters wird gestoppt und sein Zustand wird auf "Frozen" gesetzt.
- Der Charakter wird so gedreht, dass er dem Spieler zugewandt ist.
- Es wird überprüft, ob der Charakter sich in der Luft befindet, oder ob der Angriffstyp "GuardRevenge" ist, da die GuardRevenge den Enemy so zurückwirft das er sich kurz in der Luft befindet zählt die Reaktionen zu den InAir Reaktionen.
- Basierend auf den Überprüfungen wird eine entsprechende Reaktion auf den erlittenen Schaden ausgeführt (entweder eine Reaktion in der Luft oder am Boden).

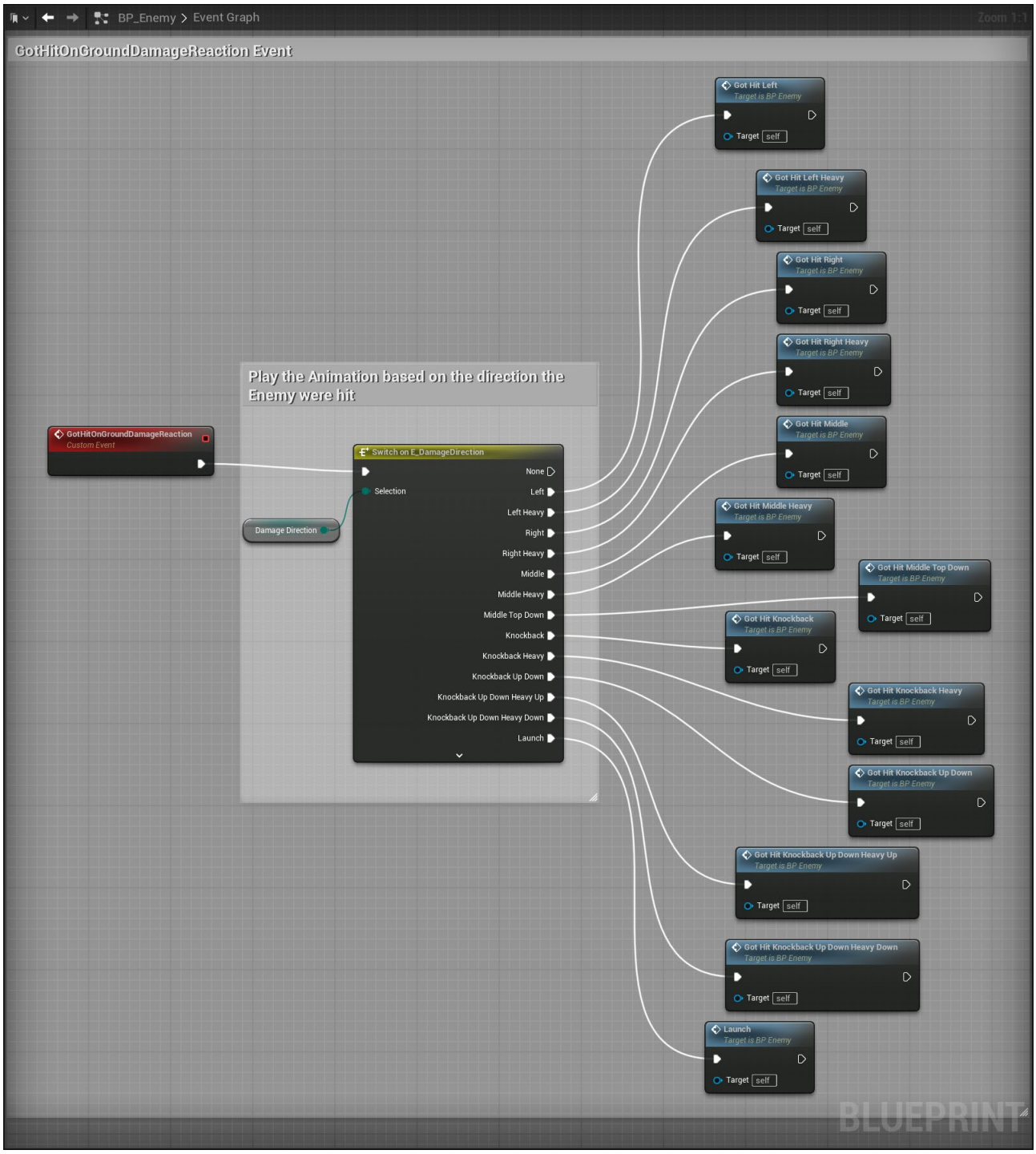
Diese Vorgehensweise stellt sicher, dass der Charakter angemessen auf erlittenen Schaden reagiert und dabei berücksichtigt, ob er sich in der Luft befindet oder es sich bei dem Schaden um eine GuardRevenge handelt.

Funktion RotateToFacePlayer



Dieser Blueprint sorgt durch Anpassung der Yaw-Wertes dafür, dass der Feind sich in Richtung des Spielers dreht, der den letzten Schaden an ihm verursacht hat.

Event GotHitOnGroundDamageReaction

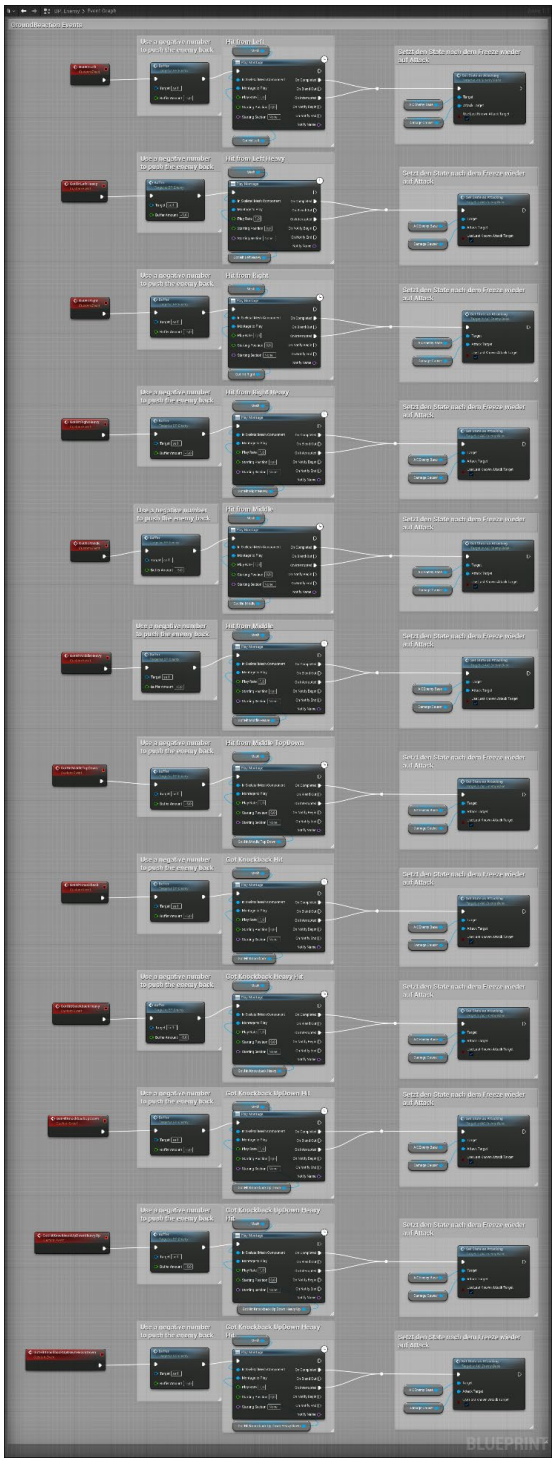


Das GotHitOnGroundDamageReaction_Event wird ausgelöst, wenn der Enemy am Boden von einer Attacke getroffen wird.

- Der Wert der Damage Direction wird verwendet, um die spezifische Reaktionsfunktion auszuwählen.
- Die entsprechende Reaktionsfunktion wird aufgerufen, um die Animation basierend auf der Richtung, aus der der Charakter getroffen wurde, abzuspielen.

Diese Vorgehensweise stellt sicher, dass der Charakter angemessen auf Treffer am Boden reagiert, indem er die passende Animation basierend auf der Richtung des Treffers abspielt.

Events GroundReactions



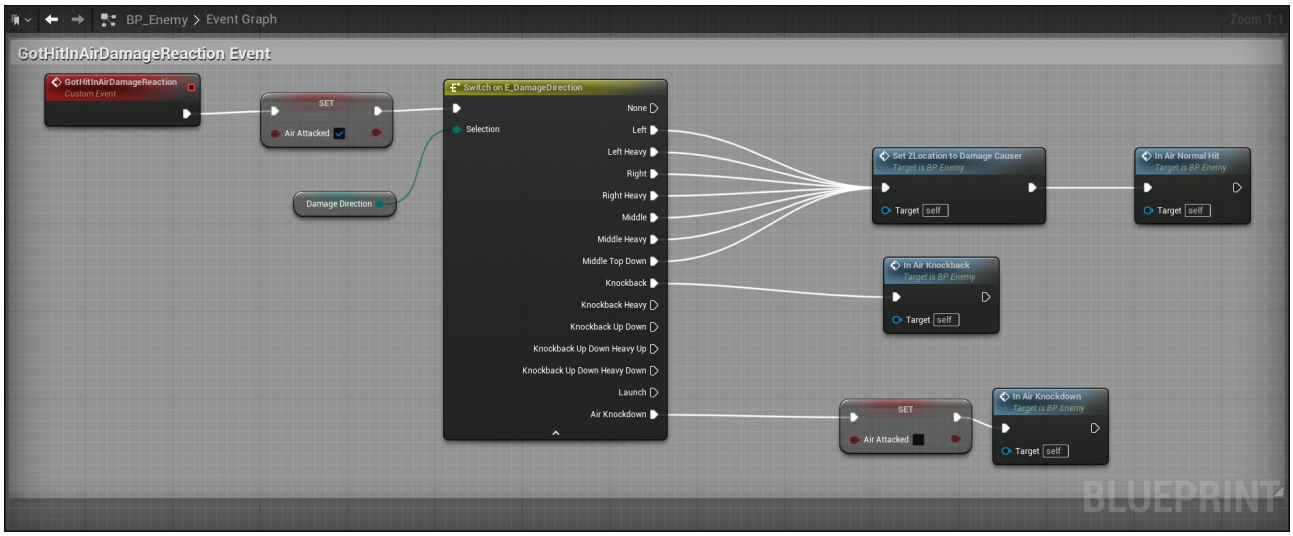
Wenn der Feind getroffen wird, wird das entsprechende Custom Event ausgelöst.

- Die Buffer Amount wird gesetzt, um den Rückstoß zu bestimmen.
- Die spezifische Trefferanimationsmontage wird abgespielt.

- Nach Abschluss der Animation wird der Zustand des Feindes auf "Attacking" zurückgesetzt, sodass er weiterhin angreifen kann.

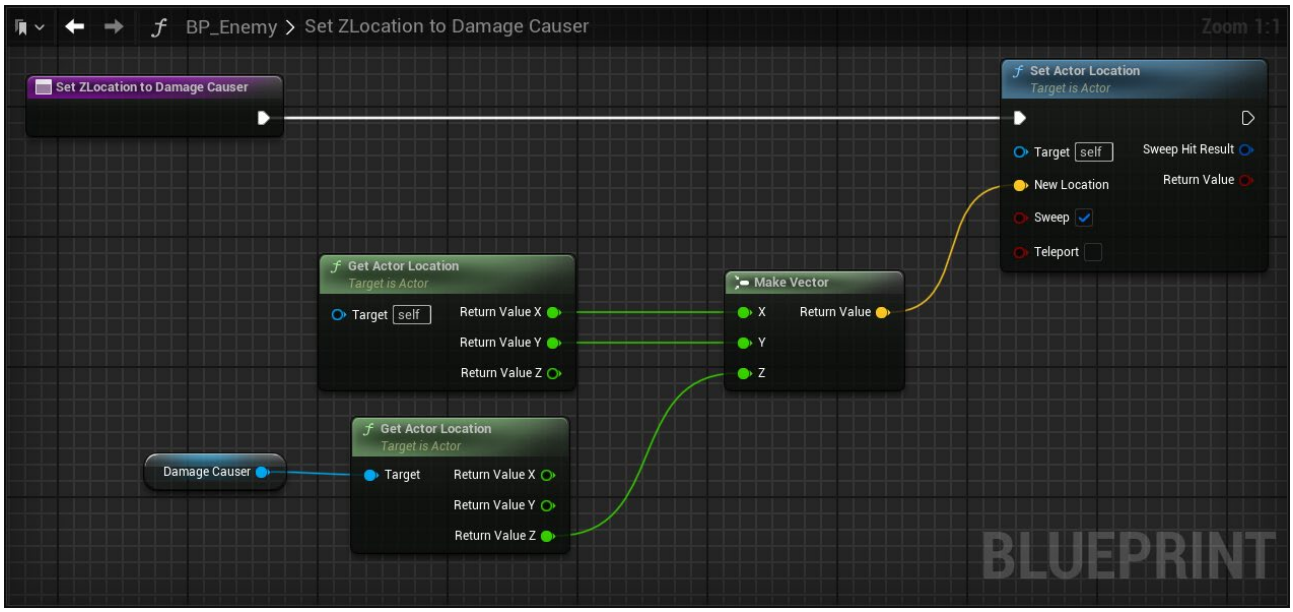
Diese Vorgehensweise stellt sicher, dass der Feind auf verschiedene Trefferarten realistisch reagiert und danach wieder in den Angriffsmodus wechselt.

Event GotHitInAirDamageReaction



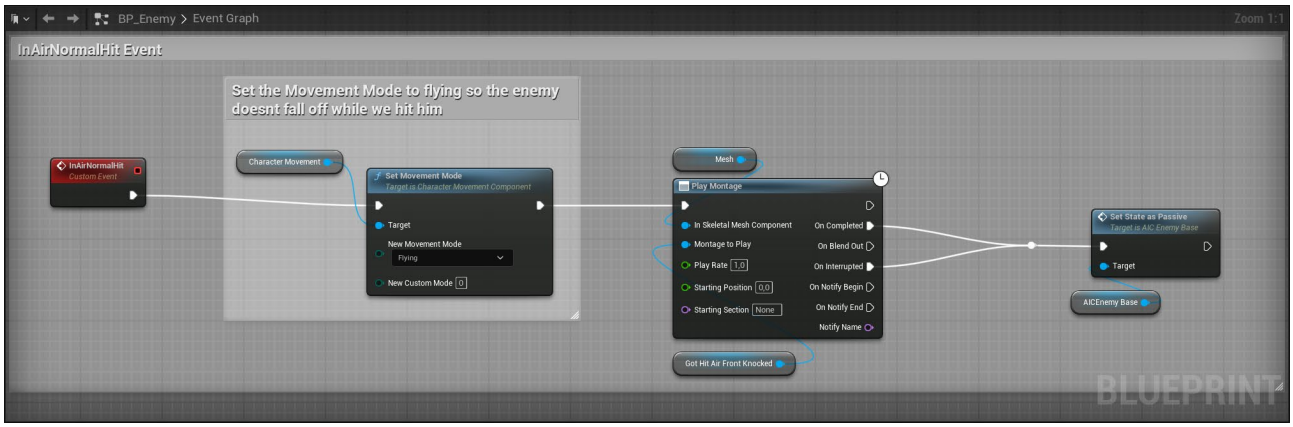
Dieses Event reagiert auf Treffer, die den Feind in der Luft treffen. Es sorgt dafür, dass die richtige Reaktion basierend auf Schwere und Art des Treffers ausgeführt wird. Dabei wird zwischen normalen InAir Treffern, dem InAir Knockback und dem InAir Knockdown unterschieden der den Enemy wieder auf den Boden bringt. Bei den Normalen InAir Treffern wird zusätzlich die SetZLocationToDamageCauser Funktion genutzt um den Enemy auf der Höhe des Schadensverursachers zu halten

Funktion SetZLocationToDamageCauser



Der Zweck dieser Funktion besteht darin, die vertikale Position (Höhe) des Feindes an die des Schadensverursachers anzupassen. Dies wird verwendet, um den Feind während der InAir Attacks auf Höhe des Spielers zu halten.

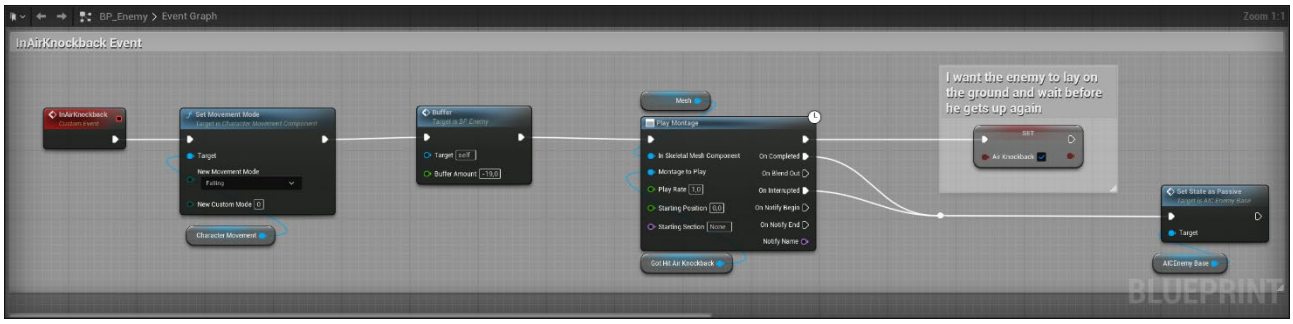
Event InAirNormalHit



Wenn der Feind in der Luft getroffen wird, löst das InAirNormalHit Event aus.

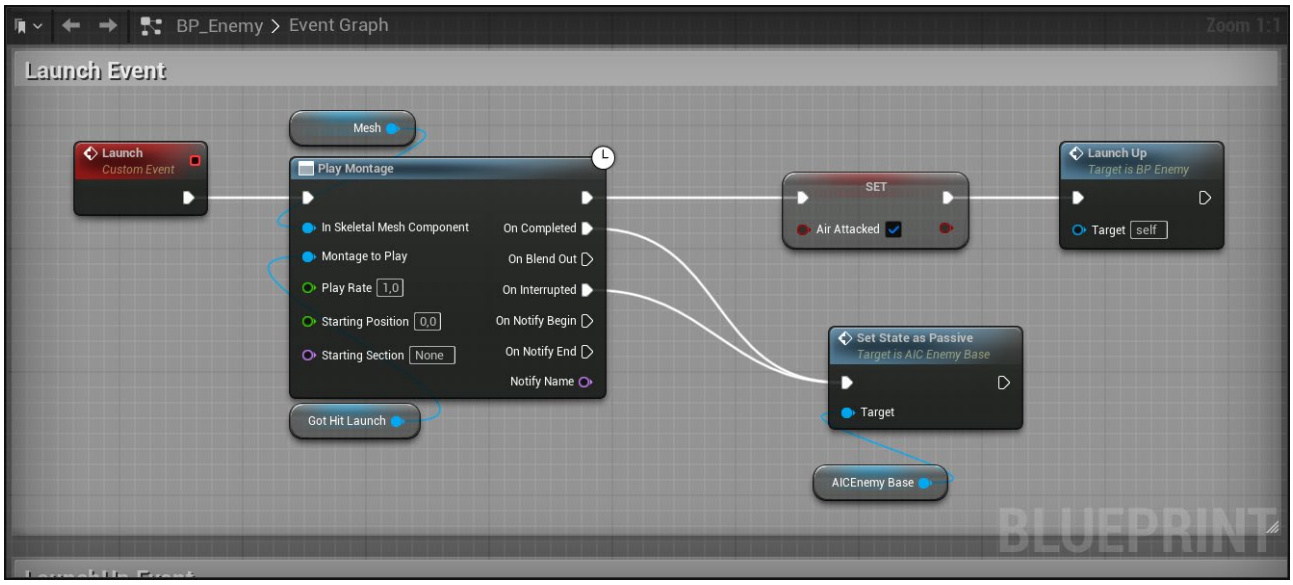
1. **Bewegungsmodus ändern**:
    - Der Bewegungsmodus des Feindes wird auf "Flying" gesetzt, um sicherzustellen, dass der Feind nicht herunterfällt.
2. **Treffer-Animation abspielen**:
    - Eine Treffer-Animation wird abgespielt, die den Treffer in der Luft visualisiert.
3. **Zustand setzen**:
    - Der Zustand des Feindes wird auf "Passive" gesetzt, indem der AI Controller des Feindes informiert wird, das sorgt dafür das der Enemy während der InAir Attacks keine Angriffe ausführt.

Event InAirKnockback



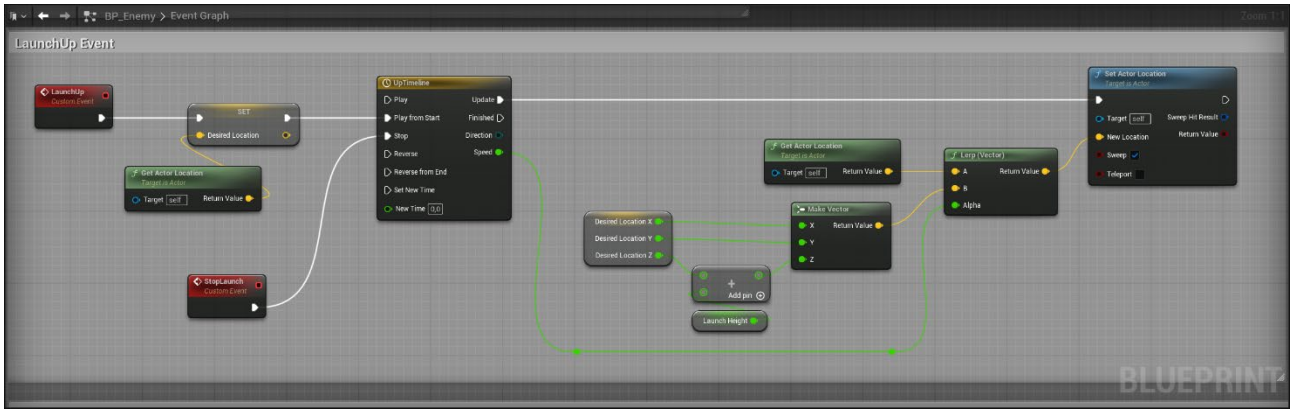
Dieses Event steuert das Verhalten des Feindes, wenn er in der Luft zurückgestoßen wird. Der Blueprint sorgt dafür, dass der Feind in den Fallmodus wechselt, gepuffert wird und eine Knockback-Animation abgespielt wird. Außerdem wird eine Bool gesetzt die das verzögerte Abspielen der Aufstehen Animation ermöglicht. Diese Reaktion wird durch den Letzten InAir Hit aus der InAirCombo ausgelöst und wirft den Gegner vom Spieler weg.

Event Launch



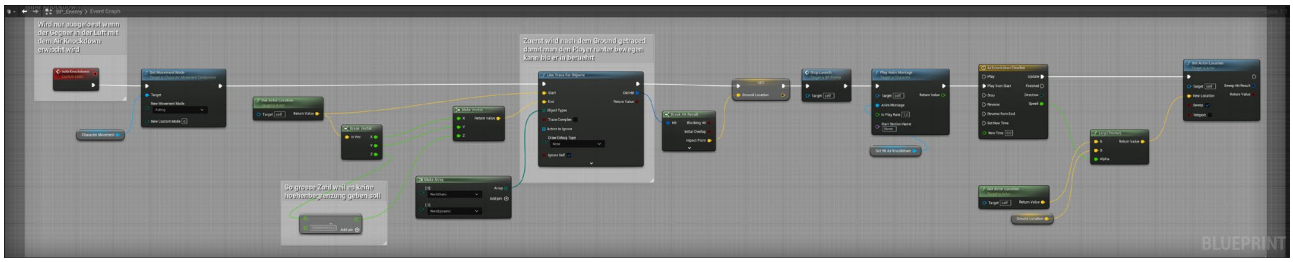
Dies stellt sicher, dass der Feind in die Luft geschleudert wird, wenn der Player den Enemy mit der Launch Attack trifft und die entsprechende Animation wird abgespielt. Der Zustand wird auch auf passiv gesetzt, was bedeutet, dass der Feind keine weiteren Aktionen ausführt, während er in der Luft ist.

Event LaunchUp



Das Event LaunchUp katapultiert den Feind nach oben, indem es eine Timeline verwendet, um die Position des Feindes über die Zeit zu interpolieren. Die gewünschte Zielhöhe wird durch Addition der aktuellen Position des Feindes mit einem festen Launch Height-Wert berechnet, der auf die Launch Attack Höhe des Players abgestimmt ist. Ein separates Event StopLaunch kann verwendet werden, um diese Aktion vorzeitig zu stoppen. Durch die Verwendung der Timeline und des Lerp wird ein natürlich wirkender Bewegungsablauf erzeugt.

Event InAirKnockdown

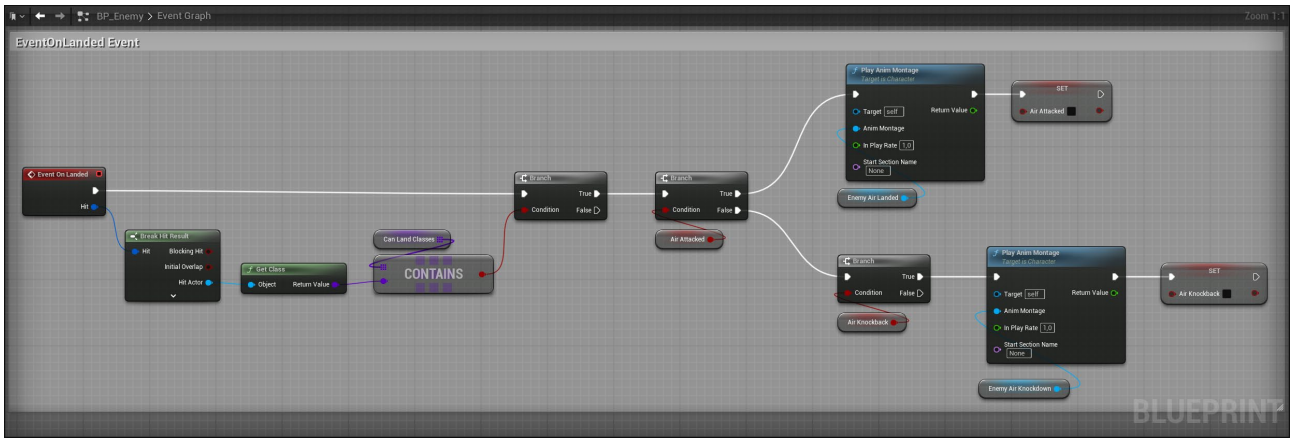


Das Event wird ausgelöst, wenn der Gegner in der Luft mit der AirKnockdown Attacke erwischt wird.

1. **Set Movement Mode**:
    - Setzt den Bewegungsmodus des Charakters auf "Falling" (fallend).
2. **Get Actor Location**:
    - Holt die aktuelle Position des Gegners.
3. **Break Vector**:
    - Zerlegt die Positionsdaten in ihre X, Y, und Z-Komponenten.
4. **Z – Z Float**
    - Subtrahiert einen Wert zur Z-Komponente der Position, um die gewünschte Höhe für den Trace zu bestimmen hier 999999999 da der Trace bei egal welcher hoehe den Ground finden soll.
5. **Line Trace for Objects**:
    - Führt eine Line Trace durch, um herauszufinden, wo der Charakter auf dem Boden aufkommen wird und speichert die Location als GroundLocation.
6. **Stop Launch**:
    - Stoppt den Launch des Gegners.
7. **Play Anim Montage**:
    - Spielt eine Animation ab, die den Luftniederschlag darstellt.
8. **AirKnockdownTimeline**:
    - Steuert die Zeitlinie für den Luftniederschlag. Die Zeitlinie sorgt für eine glatte Bewegung des Gegners von der Luft bis zum Boden.
9. **Lerp (Vector)**:
    - Interpoliert zwischen der aktuellen Position und der Zielposition über die Zeit.
10. **Set Actor Location**:
    - Setzt die Position des Gegners basierend auf der Interpolation.

Das Event sorgt dafür, dass der Gegner in der Luft niedergeschlagen wird. Es setzt den Bewegungsmodus auf "Falling", bestimmt die Bodenposition mittels eines Line Traces, stoppt den Launch, spielt eine Animation ab und bewegt den Gegner mittels einer Zeitlinie sanft auf den Boden.

Event OnLanded

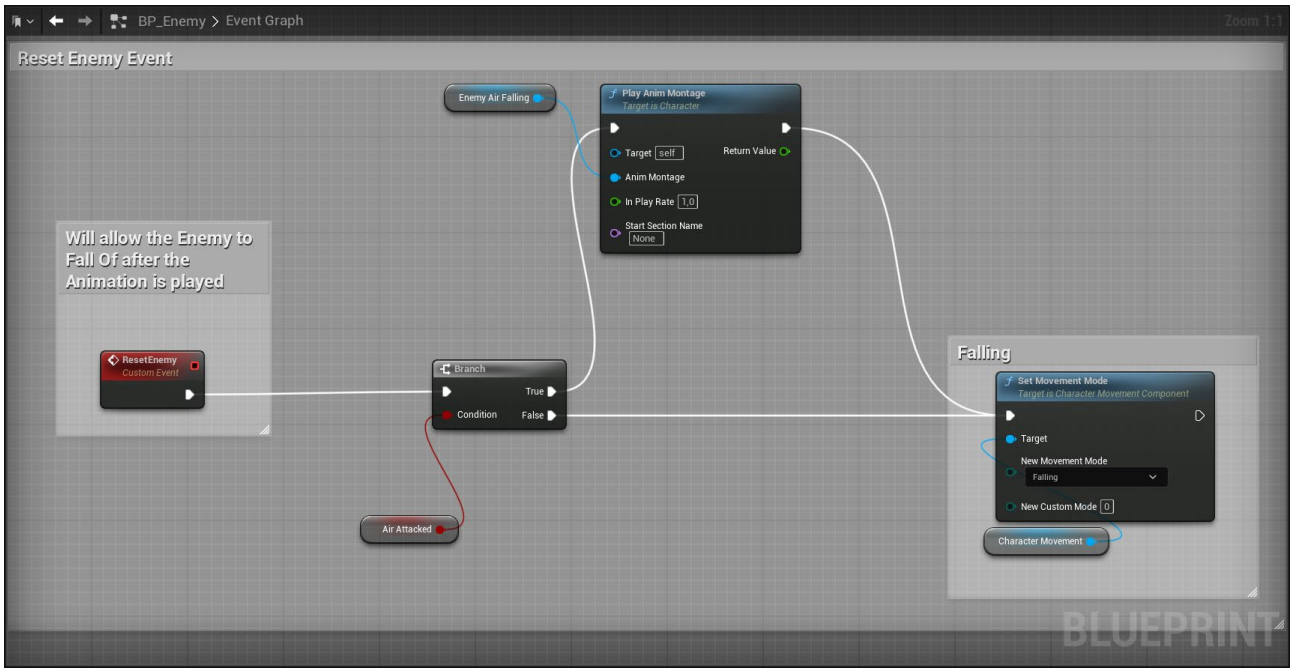


Wird ausgelöst, wenn der Gegner landet.

1. **Klasse des getroffenen Objekts überprüfen**: Prüft, ob das Objekt zu den erlaubten Landeklassen gehört.
2. **Air Attacked prüfen**: Wenn ja, spiele "Enemy Air Landed" Animation und setze "Air Attacked" auf false.
3. **Wenn nein Air Knockback prüfen**: Wenn ja, spiele "Enemy Air Knockdown" Animation und setze "Air Knockback" auf false.

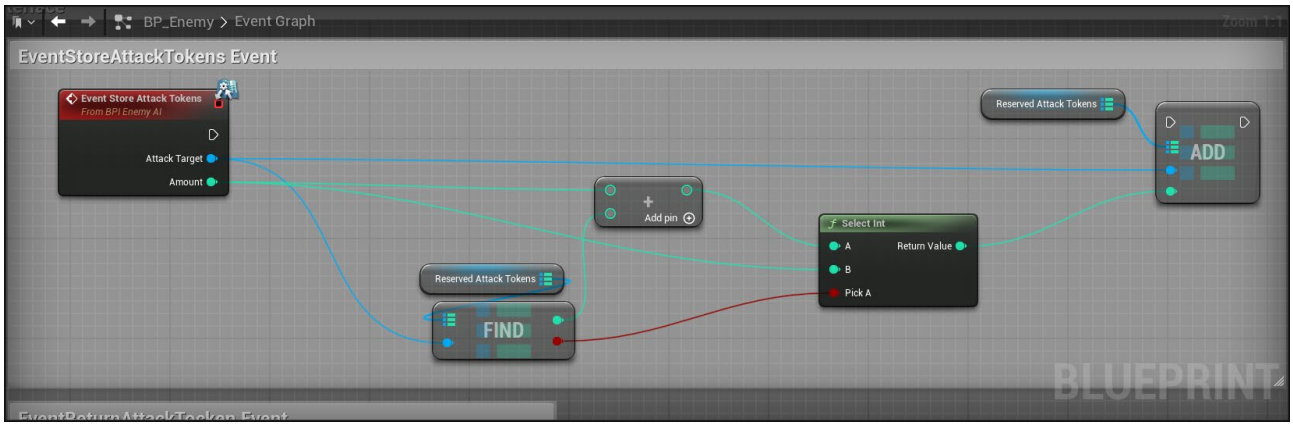
Das Event prüft, ob der Gegner auf dem Boden gelandet ist und spielt je nachdem ob er zuvor von einem AirKnockback betroffen war oder nicht die entsprechende Animation. Somit steht der Gegner schneller wieder auf wenn er nur Fallen sollte ohne von einem AirKnockback betroffen zu sein.

Event ResetEnemy



Der ResetEnemy Event Blueprint sorgt dafür, dass der Feind nach dem Abspielen einer InAir Hit Animation in den Fallzustand versetzt wird und nach einer Air Attack Hit Reaktion Animation die entsprechende Folgeanimation spielt. Wird durch den Anim Notify AN_ResetState innerhalb der Animation ausgelöst.

Interface Event StoreAttackTokens



Das Event EventStoreAttackTokens beginnt mit einem Interface-Aufruf aus dem BPI_EnemyAI Interface.

Suchen eines vorhandenen Tokens:

Die Funktion durchsucht die Reserved Attack Tokens Liste, um festzustellen, ob bereits ein Token für das angegebene Attack Target existiert.

Auswahl der Token-Anzahl:

Es wird ein Vergleich durchgeführt, um zu entscheiden, ob die vorhandene Anzahl an Tokens oder die neu hinzugefügte Anzahl verwendet werden soll. Dies wird durch den Select Int Knoten gesteuert.

Hinzufügen oder Aktualisieren der Tokens:

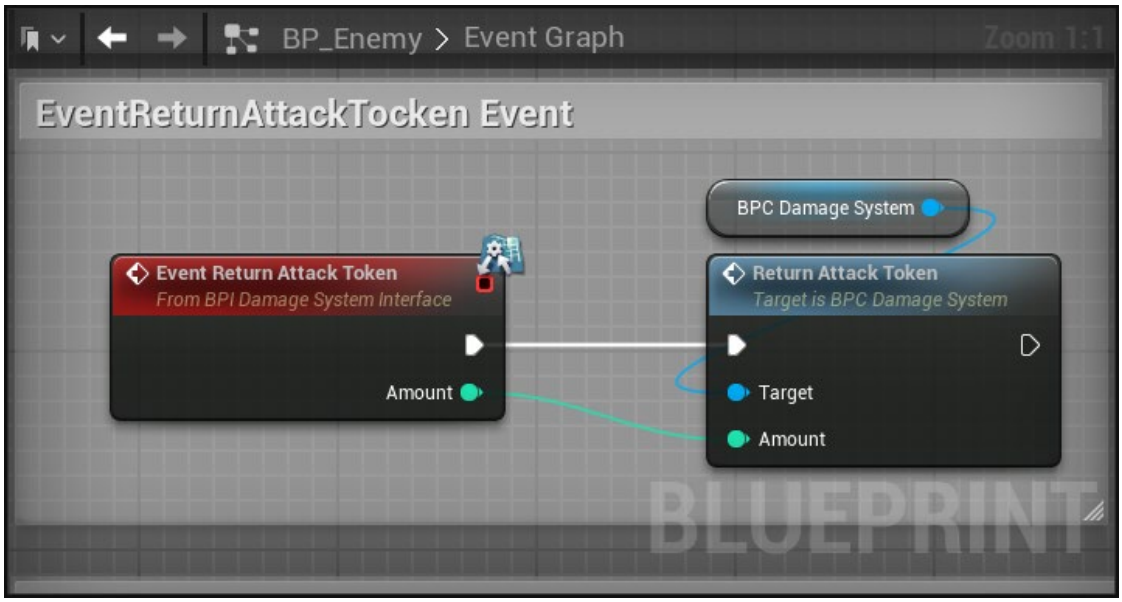
Wenn ein Token gefunden wurde, wird die vorhandene Anzahl aktualisiert. Falls kein Token vorhanden ist, wird ein neuer Eintrag mit der entsprechenden Anzahl an Tokens erstellt.

Abschluss:

Die aktualisierte Liste der Reserved Attack Tokens wird gespeichert.

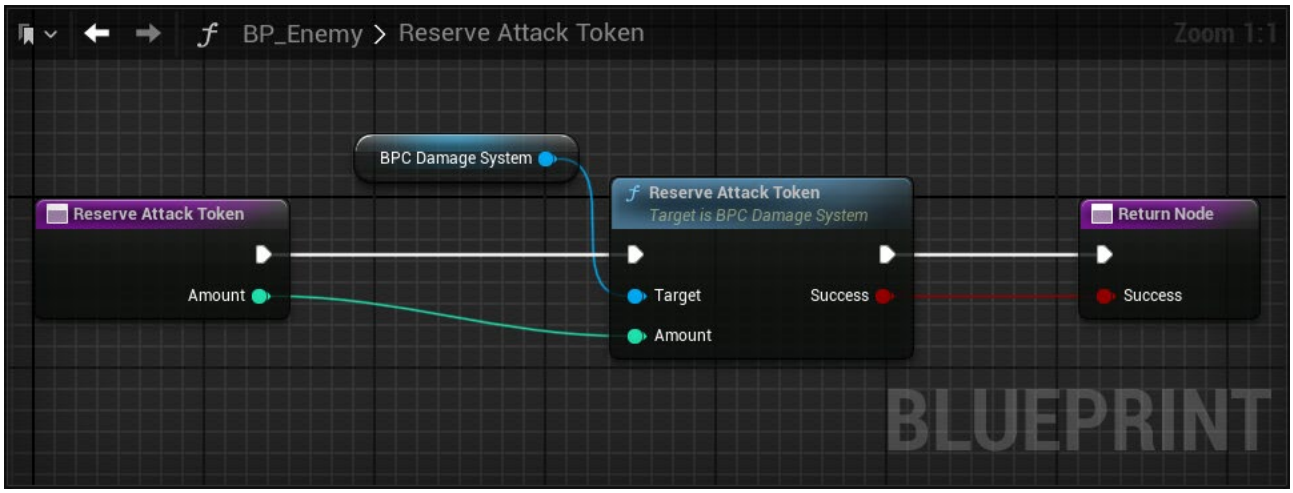
Diese Funktion ermöglicht es, die Anzahl der Angriffstokens für ein bestimmtes Ziel zu verfolgen und bei Bedarf zu aktualisieren. Sie stellt sicher, dass Tokens effizient verwaltet und verwendet werden. Sie bestimmt damit in Abhängigkeit der zu brauchenden Tokens festgelegt im Behavior Tree ob der Enemy den Player angreifen kann oder nicht.

Interface Event ReturnAttackTokens



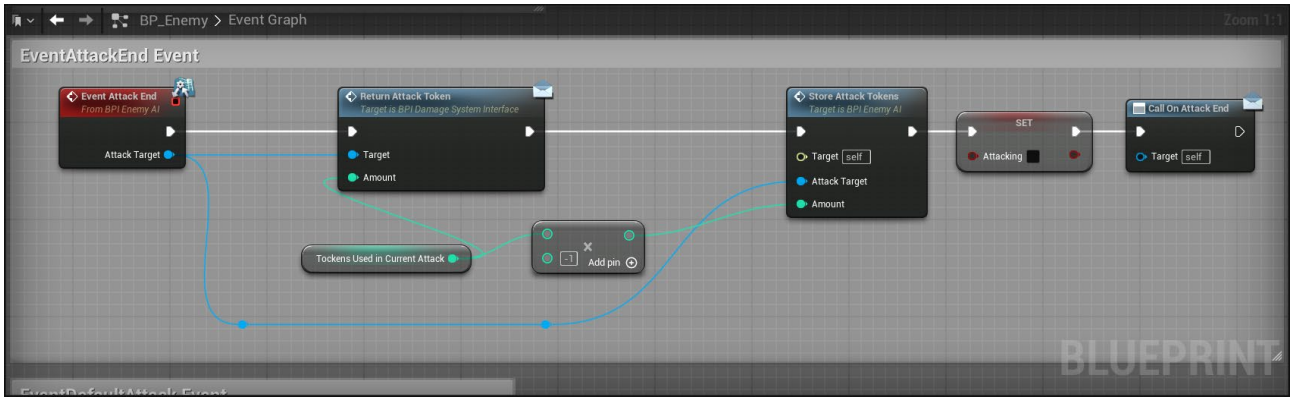
Dieses Interface Event aus dem BPI_DamageSystemInterface ermöglicht es, Angriffstokens, die nicht mehr benötigt werden, zurück an das System zu geben, um sie für zukünftige Angriffe auch anderer Enemies wiederverwendbar zu machen.

Interface Funktion ReserveAttackToken



Diese Funktion aus dem BPI_DamageSystem Interface dient dazu, Angriffstokens für den Gegner zu reservieren, um sicherzustellen, dass Angriffe nur durchgeführt werden, wenn genügend Ressourcen verfügbar sind.

Interface Event AttackEnd



Das Event EventAttackEnd beginnt mit einem Interface-Aufruf aus dem BPI_EnemyAI Interface.

Rückgabe des Angriffstokens:

Die Funktion ruft den Knoten Return Attack Token auf, um den Angriffstoken an das BPC Damage System Interface zurückzugeben. Dies geschieht z.b., wenn der Enemy eine Attacke ausgeführt hat. Die Menge wird dabei im Behavior Tree festgelegt.

Speicherung der Angriffstokens:

Nach der Rückgabe der Tokens wird der Knoten Store Attack Tokens aufgerufen, um die Angriffstokens zu speichern. Auch hier werden Ziel- und Mengenparameter verwendet.

Zurücksetzen des Angriffsstatus:

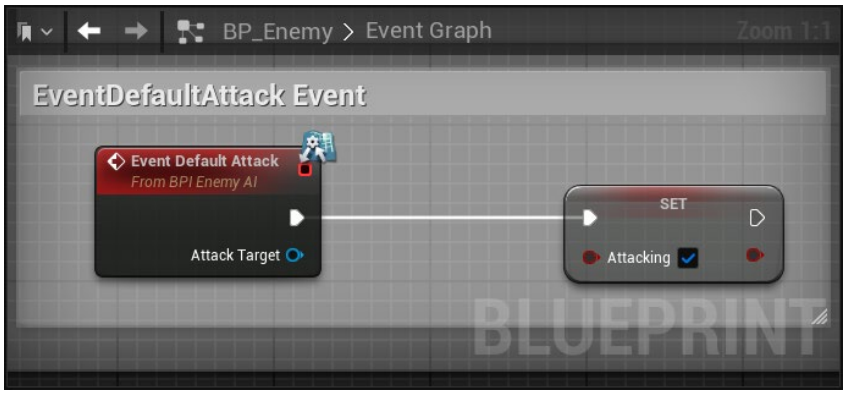
Der Boolesche Wert Attacking wird auf false gesetzt. Dies bedeutet, dass der Angriff beendet ist.

Aufruf des Angriffsende-Ereignisses:

Zum Abschluss wird der Knoten Call On Attack End aufgerufen, um das Ende des Angriffs an das System zu signalisieren.

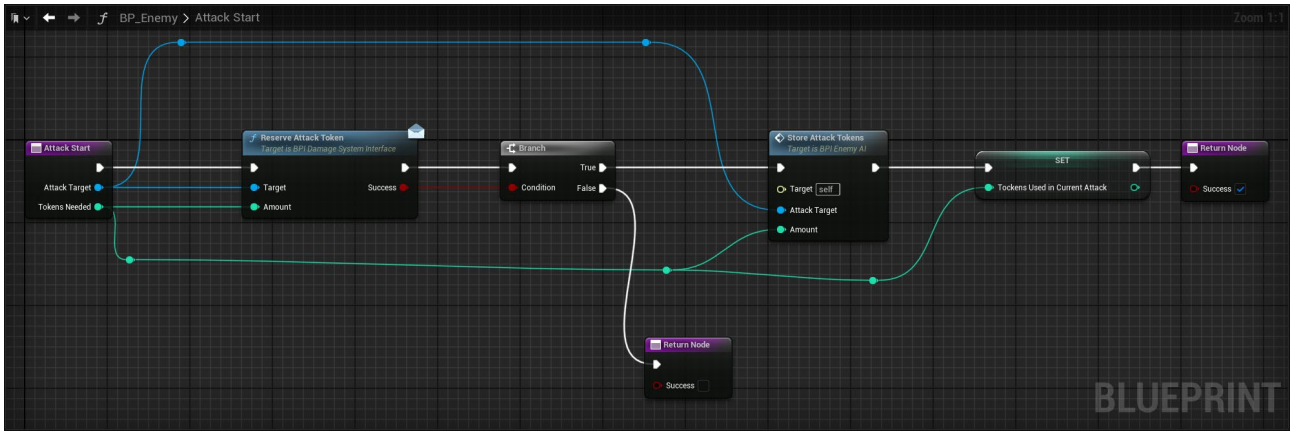
Diese Funktion stellt sicher, dass alle Angriffstokens ordnungsgemäß zurückgegeben und gespeichert werden und der Angriffszustand korrekt aktualisiert wird.

Interface Event DefaultAttack



Dieses Event Aufgerufen aus dem BPI_EnemyAI Interface stellt sicher, dass der Gegner den Angriffsstatus korrekt einnimmt, wenn ein Standardangriff initiiert wird, indem es Attacking auf true setzt.

Interface Funktion AttackStart



Das Event AttackStart beginnt mit einem Interface-Aufruf BPI_EnemyAI Interface.

Reservierung der Angriffstokens:

Die Funktion ruft den Knoten Reserve Attack Token auf, um Angriffstokens beim BPI Damage System Interface zu reservieren. Hier werden Ziel und benötigte Tokens als Parameter verwendet.

Überprüfung der Tokenreservierung:

Ein Branch-Knoten überprüft den Erfolg der Tokenreservierung. Wenn die Reservierung erfolgreich ist, wird die Ausführung fortgesetzt und die Attacke wird erfolgreich gestartet, andernfalls wird sie abgebrochen.

Speicherung der Angriffstokens:

Wenn die Tokenreservierung erfolgreich war, wird der Knoten Store Attack Tokens aufgerufen, um die Angriffstokens zu speichern. Auch hier werden Ziel und Tokens als Parameter verwendet.

Setzen der verwendeten Tokens:

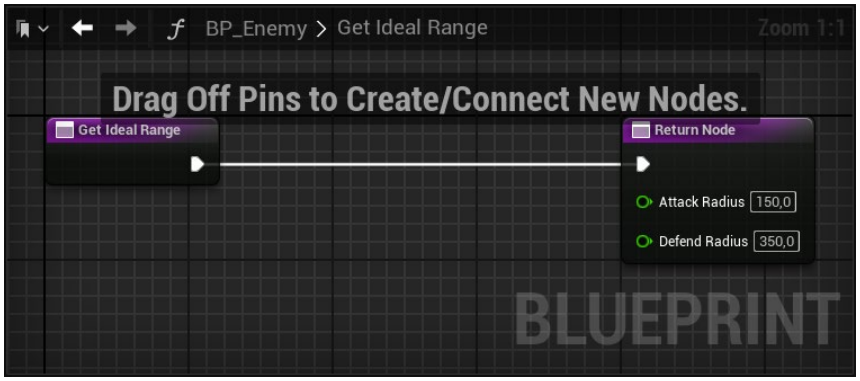
Der Wert Tokens Used in Current Attack wird auf die Anzahl der reservierten Tokens gesetzt. Dies stellt sicher, dass der aktuelle Angriff die richtige Anzahl von Tokens verwendet.

Rückgabe des Erfolgs:

Zum Abschluss wird der Return Node aufgerufen und der Erfolg des Angriffs zurückgegeben.

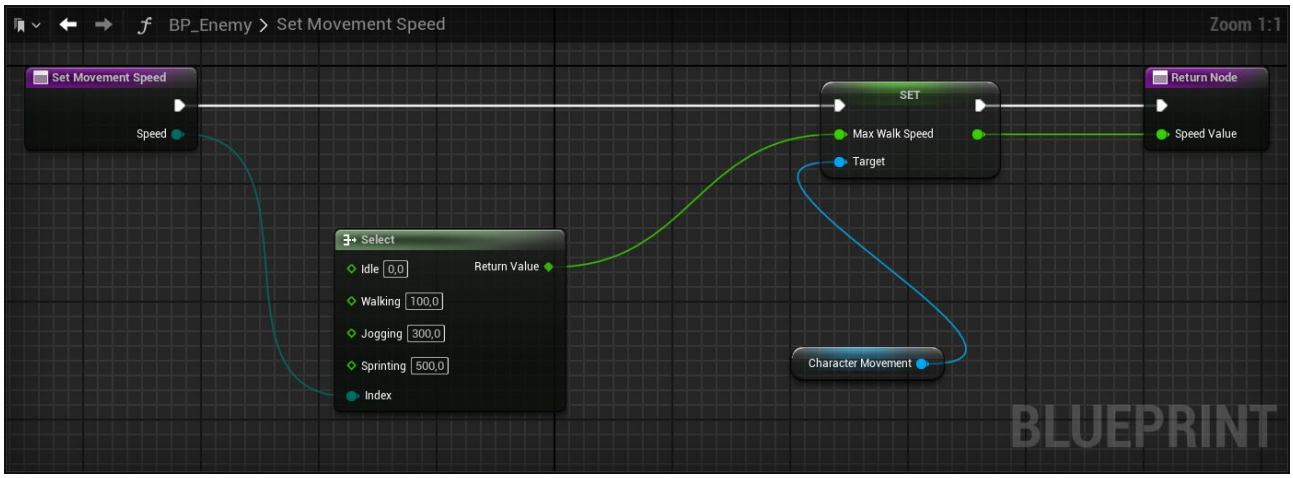
Die Funktion stellt sicher, dass alle notwendigen Angriffstokens ordnungsgemäß reserviert und beim Enemy gespeichert werden und der Angriff nur gestartet wird wenn genug Tokens geholt werden konnten.

Interface Funktion GetIdealRange



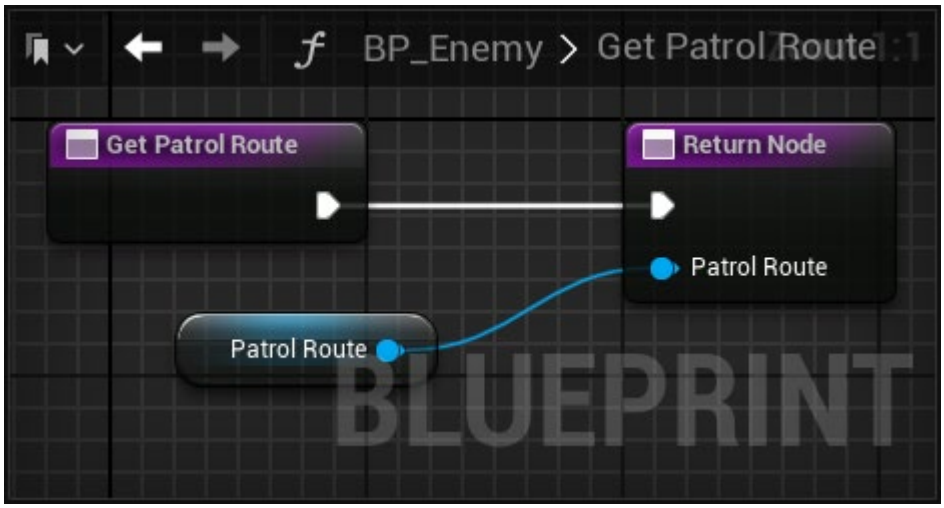
Diese Funktion wird aus dem BPI_EnemyAI Interface aufgerufen und definiert die Standard Angriffs- und Verteidigungsradien der Gegner. Der Angriffsradius gibt an, wie nah der Gegner am Ziel sein muss, um anzugreifen, während der Verteidigungsradius die Entfernung angibt, innerhalb derer der Gegner im Strafing oder Wartezustand bleibt.

Interface Funktion SetMovementSpeed



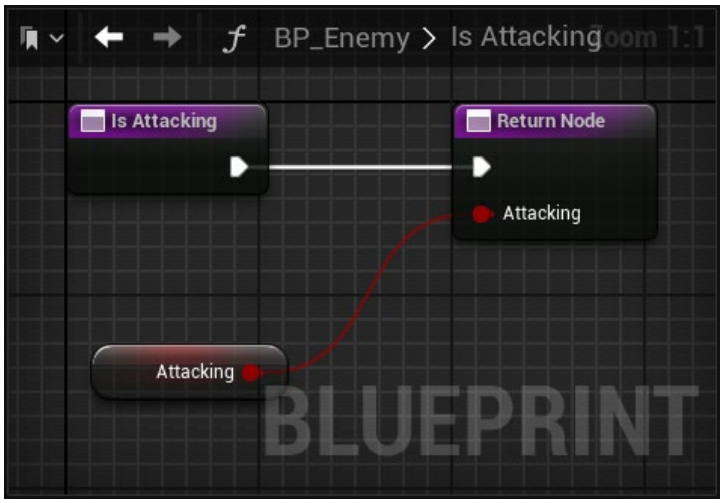
Diese Funktion aus dem BPI_EnemyAI Interface ermöglicht es, die Bewegungsgeschwindigkeit des Gegners, in Abhängigkeit des aktuellen Status des Gegners dynamisch anzupassen.

Interface Funktion GetPatrolRoute



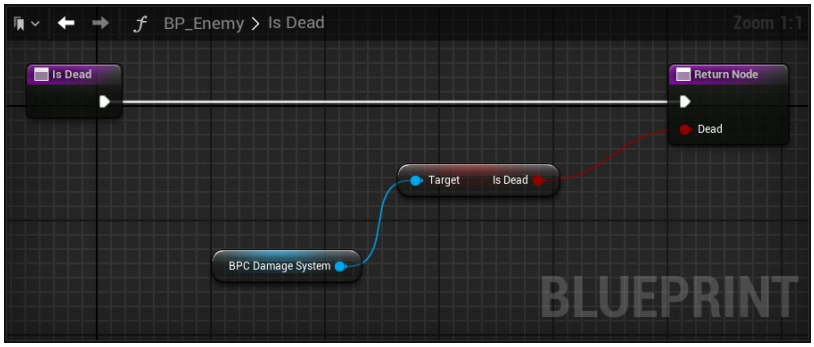
Diese Funktion aus dem BPI_EnemyAI Interface ermöglicht es, die aktuelle Patrouillenroute des Gegners zu erhalten.

Interface Funktion IsAttacking



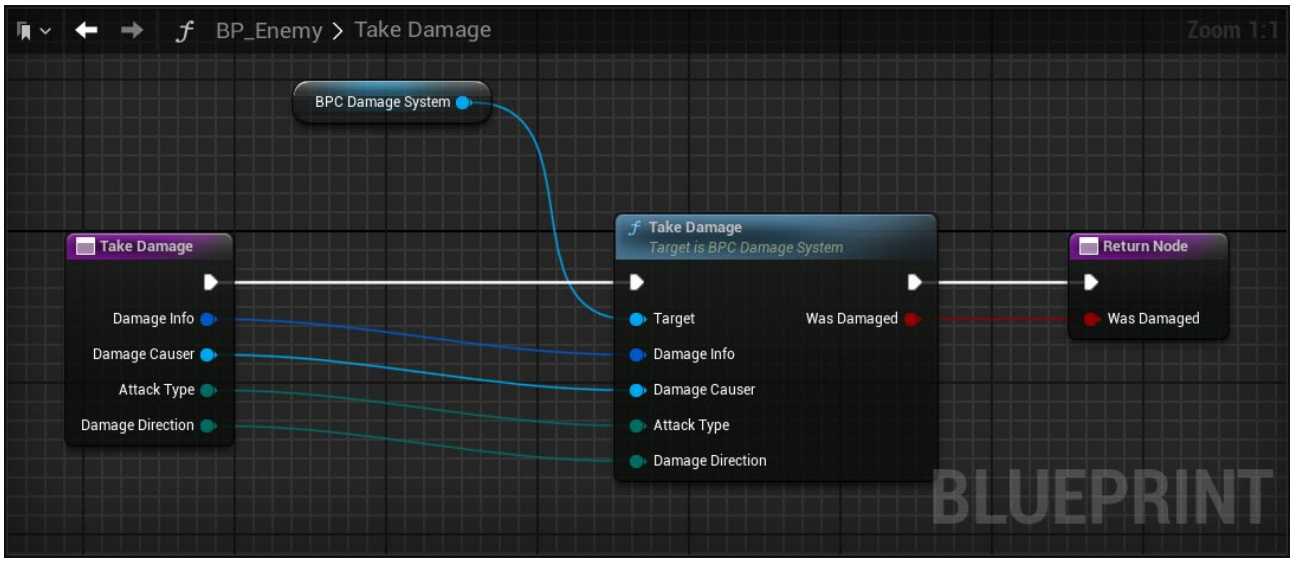
Diese Funktion aus dem BPI_DamageSystemInterface dient dazu, den aktuellen Zustand des Gegners zu überprüfen und festzustellen, ob dieser sich gerade in einem Angriff befindet.

Interface Funktion IsDead



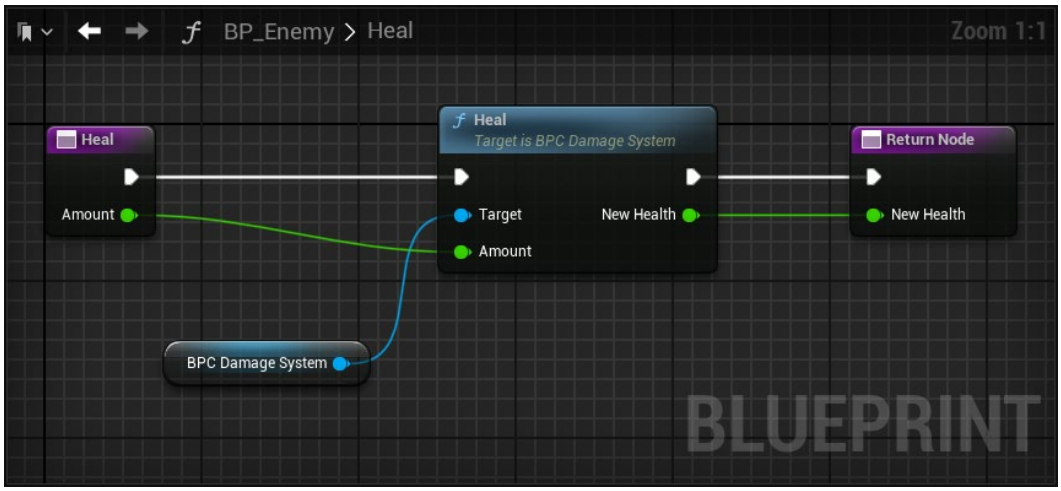
Diese Funktion dient dazu, zu überprüfen ob das Target des Enemies (der Player) tot ist.

Interface Funktion TakeDamage



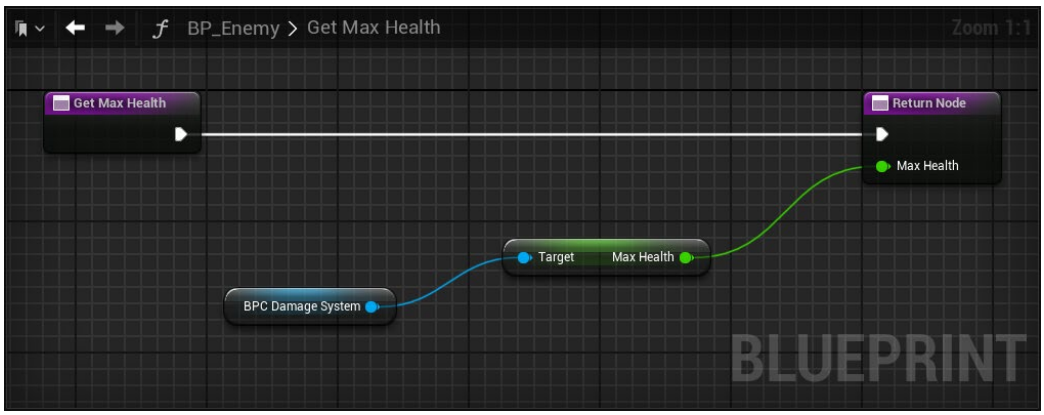
Diese Funktion aus dem BPI_DamageSystemInterface dient dazu, alle relevanten Informationen über den erlittenen Schaden zu verarbeiten und weiterzugeben ob Schaden erfolgreich ausgeteilt wurde. Vom Ablauf her der im BP_ThirdPersonCharacter fast identisch nur das hier auch die DamageDirection weitergegeben wird da sie bei der der Hit Reaktion des Enemies gebraucht wird.

Interface Funktion Heal



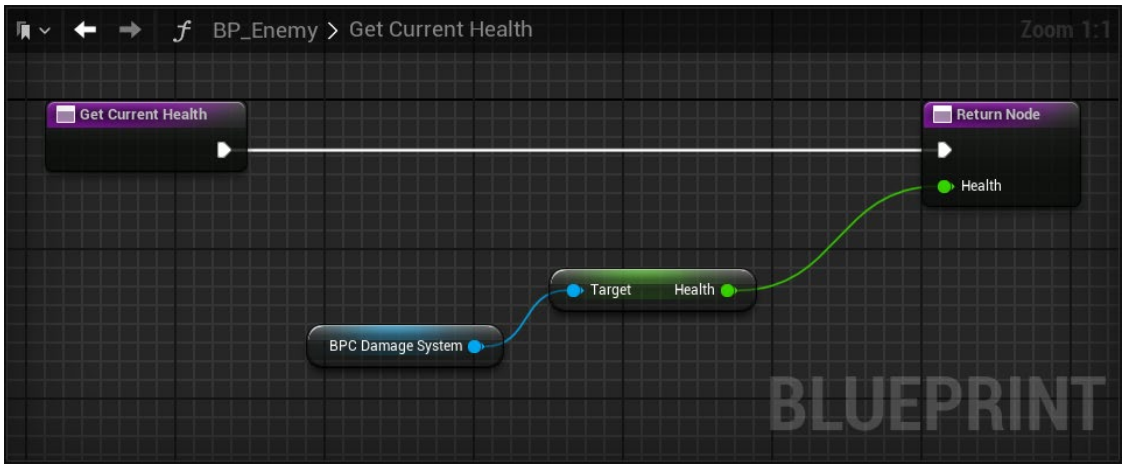
Die Heal Funktion aus dem BPI_DamageSystemInterface dient dazu, den Enemy zu heilen. Sie übernimmt den Heilungsbetrag und berechnet mithilfe des BPC Damage Systems die neue Gesundheit des Charakters.

Interface Funktion GetMaxHealth



Diese Funktion aus dem BPI_DamageSystemInterface sorgt dafür, dass die maximale Gesundheit des Feindes korrekt abgerufen und zurückgegeben wird.

Interface Funktion GetCurrentHealth



Diese Funktion aus dem BPI_DamageSystemInterface stellt sicher, dass die aktuelle Gesundheit des Feindes korrekt abgerufen und zurückgegeben wird.

Event Dispatcher

Die Event Dispatcher ermöglichen es verschiedenen Teilen des Spiels, auf spezifische Ereignisse zu reagieren, indem sie Signale senden, die andere Systeme empfangen und verarbeiten können.

OnAttackEnd

Dieser Event Dispatcher wird ausgelöst, wenn ein Angriff endet. Es signalisiert anderen Systemen, dass der Angriff abgeschlossen ist.

OnWeaponEquipped

Dieser Event Dispatcher wird ausgelöst, wenn eine Waffe ausgerüstet wird. Es signalisiert anderen Systemen, dass eine Waffe erfolgreich ausgerüstet wurde.

OnWeaponUnequipped

Dieser Event Dispatcher wird ausgelöst, wenn eine Waffe abgelegt wird. Es signalisiert anderen Systemen, dass eine Waffe erfolgreich abgelegt wurde.

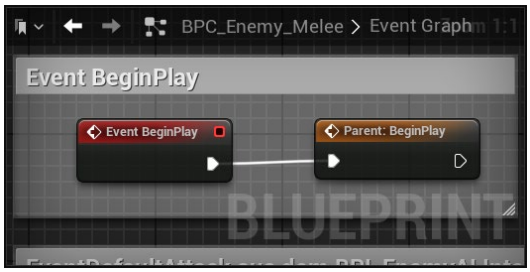
BPC_Enemy_TestDummy (Child von BP_Enemy)

Ein Dummy der verwendet wird, um die Hit-Reaktionen und Kombinationsabläufe des Spielers zu testen. Im Startbereich könnte dieser Dummy platziert werden, damit der Spieler an ihm trainieren kann.

BPC_Enemy_Melee (Child von BP_Enemy)

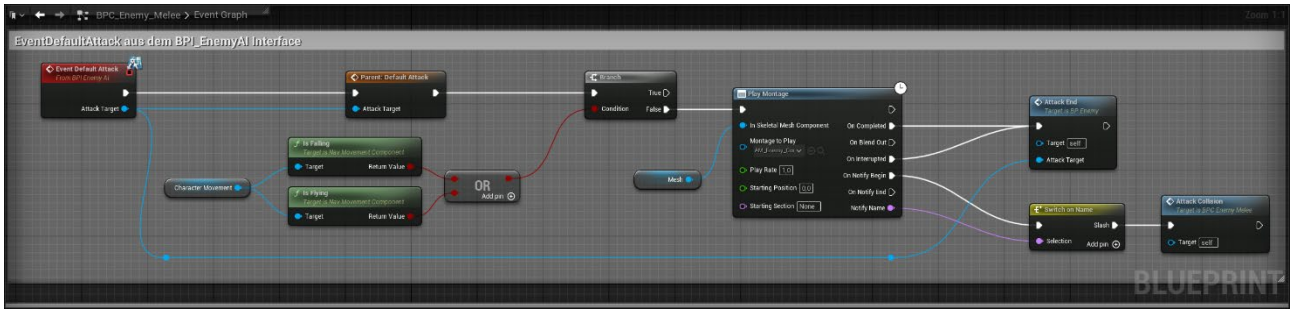
Der BPC_Enemy_Melee ist ein Child-Blueprint des BP_Enemy. Das bedeutet, dass er die Funktionalitäten von BP_Enemy erbt und zusätzlich speziell für Nahkampfgegner (Melee) angepasste und erweiterte Funktionen bietet.

Event BeginPlay



Durch den Aufruf der Parent-Funktion wird sichergestellt, dass alle Initialisierungen und Logiken, die im Eltern-Blueprint definiert sind, ebenfalls ausgeführt werden.

Interface Event DefaultAttack



Das Event EventDefaultAttack beginnt mit einem Funktionsaufruf vom BPI_EnemyAI Interface.

Aufruf der Parent-Funktion:

Es wird die Default Attack-Funktion des Eltern-Blueprints aufgerufen. Dies stellt sicher, dass alle grundlegenden Angriffsfunktionen ausgeführt werden, die im Eltern-Blueprint definiert sind.

Überprüfung der Bewegung:

- Überprüft, ob sich der Enemy gerade in der Luft befindet

Animation Abspielen:

- Spielt die AttackMontage des Melee Enemy ab

Angriff Kollision:

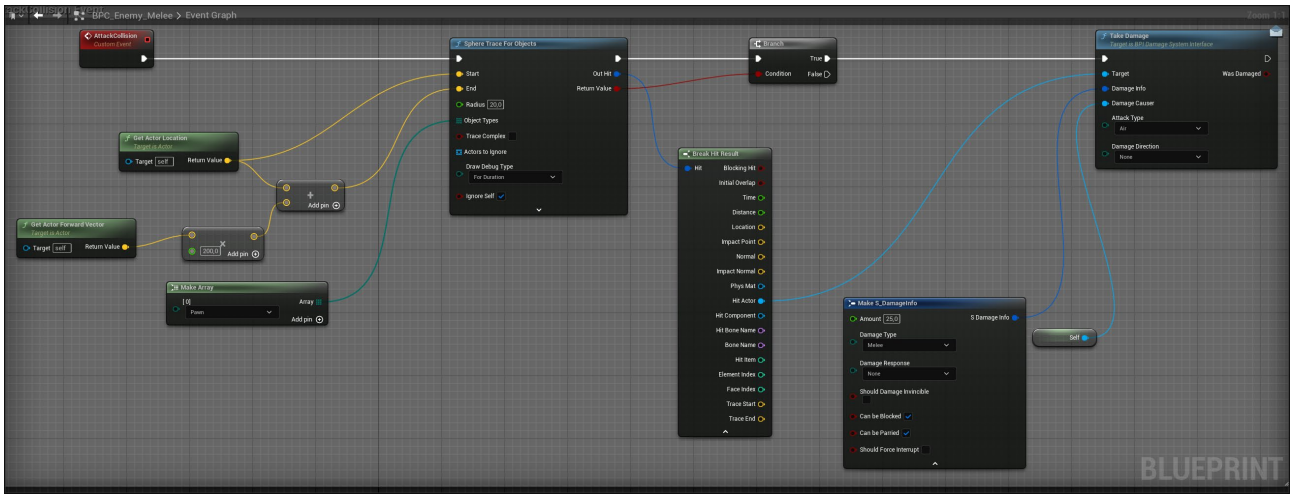
- **Switch on Name**: Schaltet auf den Namen (Slash) des Angriffs um und führt die Attack Collision durch.

Angriff Beenden:

- **Attack End**: Ruft im Falle des Endes der Animation oder des Unterbrechens das Ende des Angriffs im BP_Enemy auf und führt die entsprechenden Abschlussaktionen aus.

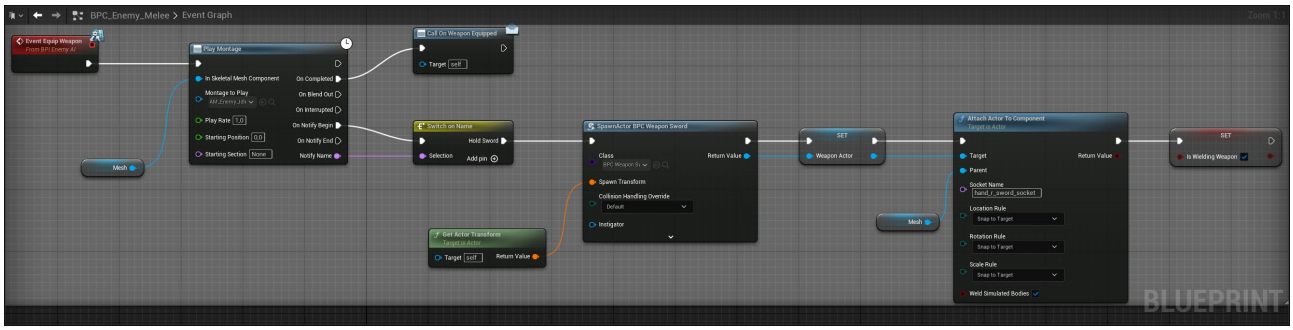
Durch die Implementierung und Anpassung der DefaultAttack-Funktion aus dem BPI_EnemyAI Interface kann das spezifische Verhalten des Nahkampfgegners während eines Angriffs definiert und gesteuert werden.

Event AttackCollision



Diese Funktion führt einen kugelförmigen Trace in Blickrichtung, vor dem Gegner durch, um Kollisionen zu erkennen und den Schaden weiterzugeben, den der Gegner verursacht. Wenn ein Treffer erkannt wird, wird der hier definierte Schaden an den getroffenen Akteur (Spieler) übertragen und gegeben falls abgehandelt.

Interface Event EquipWeapon

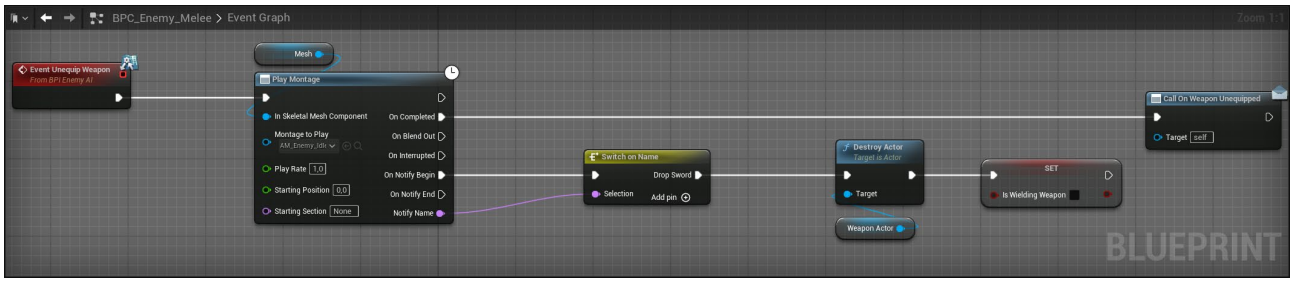


Wird aus dem BPI_Enemy_AI Interface ausgelöst, um eine Waffe auszurüsten.

1. **Play Montage**: Spielt die Montage für den Weapon Equip auf dem Mesh des Enemies ab.
2. **Call On Weapon Equipped**: Ruft die Funktion "On Weapon Equipped" auf sobald die Montage Completed ist.
3. **Switch on Name**: Prüft den Namen der auszurüstenden Waffe bei Beginn des Notify Hold Sword in der Anim Montage.
4. **Spawn Weapon**: Spawnt die Waffe "BPC_Weapon_Sword" und setzt sie als "Weapon Actor".
5. **Attach Actor to Component**: Befestigt die Waffe an der Hand (Socket: "hand_r_sword_socket").
6. **Set Is Wielding Weapon**: Setzt die Variable "Is Wielding Weapon" auf true.

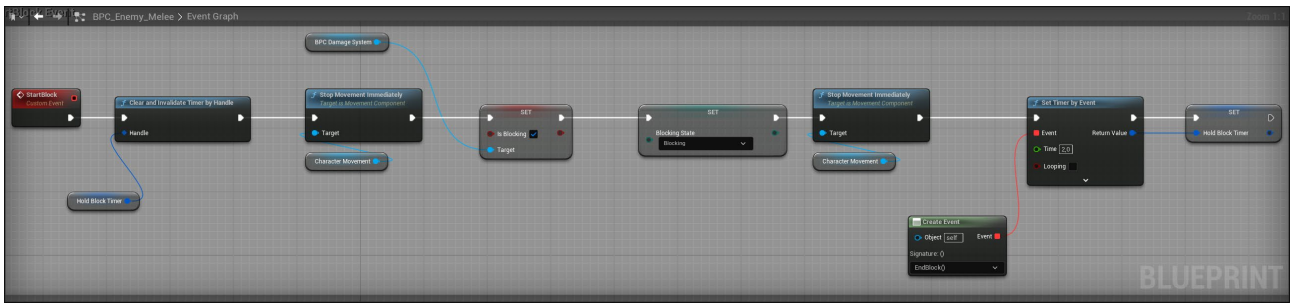
Zusammengefasst rüstet das Event eine Waffe aus, indem es eine Animation abspielt, die Waffe spawnt und an die Hand des Charakters anheftet.

Interface Funktion OnSheath



Diese Funktion aus dem BPI_Enemy_AI Interface spielt die Unequip Animation ab, zerstört die Waffe sobald der Notify „Drop Sword" in der Anim Montage ausgelöst wird und setzt den Status, dass die Waffe nicht mehr ausgerüstet ist, um anzuzeigen, dass die Waffe jetzt zurückgesteckt wurde. Nach Ablauf der Animation wird der Event Dispatcher CallOnWeaponUnequip ausgelöst.

Event StartBlock

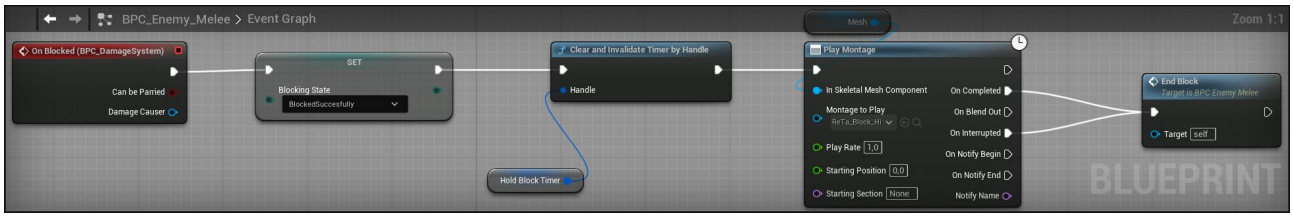


Wird ausgelöst, um den Blockvorgang zu starten.

1. **Clear and Invalidate Timer by Handle**: Löscht und deaktiviert den Timer, der durch den Handle "Hold Block Timer" repräsentiert wird.
2. **Stop Movement Immediately (Character Movement)**: Stoppt sofort die Bewegung des Charakters.
3. **Set Is Blocking**: Setzt die Variable "Is Blocking" auf true, um anzuzeigen, dass der Charakter blockiert.
4. **Set Blocking State**: Setzt den Zustand des Charakters auf "Blocking".
5. **Stop Movement Immediately (Character Movement)**: Stoppt erneut die Bewegung des Charakters.
6. **Set Timer by Event**: Setzt einen Timer für das "EndBlock"-Event auf 2 Sekunden.
7. **Set Hold Block Timer**: Speichert den Timer-Handle in der Variable "Hold Block Timer".

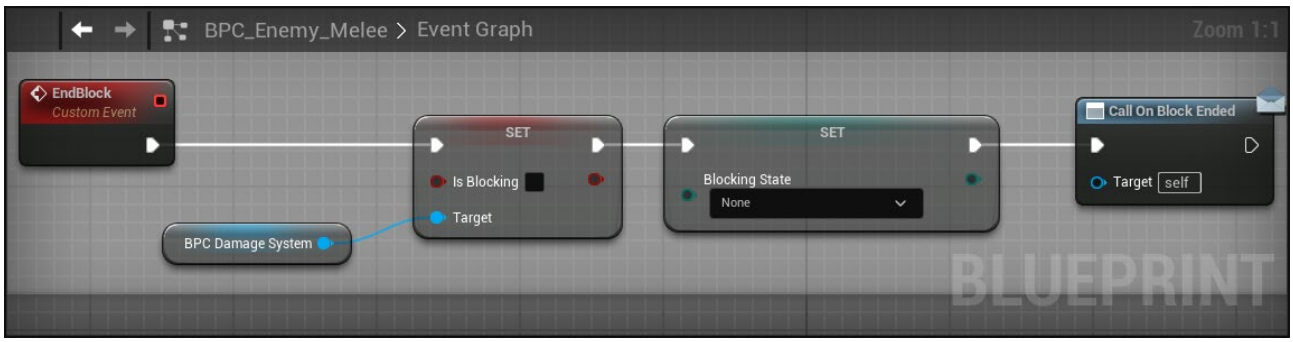
Das Event startet den Blockvorgang, indem es die Bewegung stoppt, den Blockzustand setzt und einen Timer startet, der den Blockvorgang nach 2 Sekunden beendet.

Interface Event OnBlocked



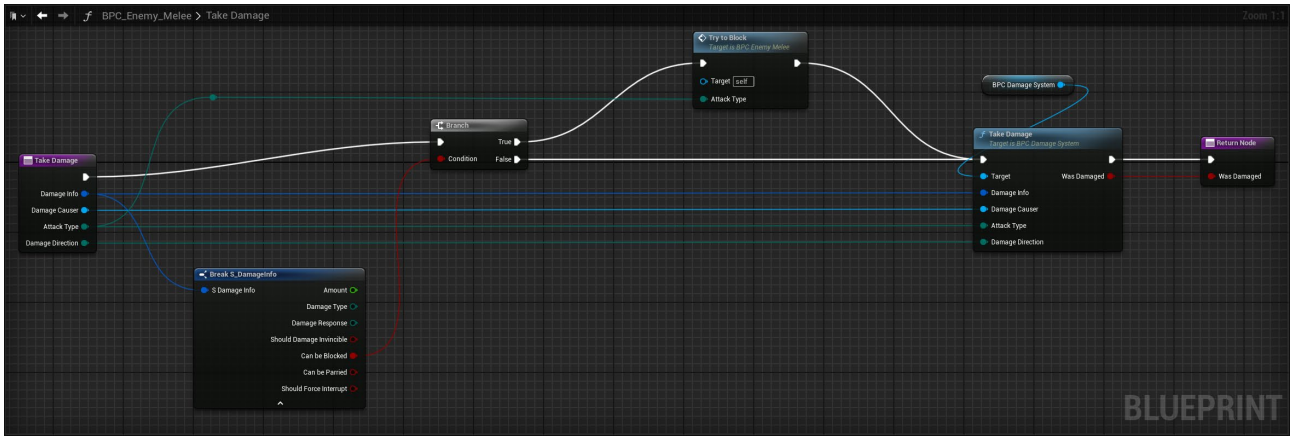
Das Event aus dem BPC_DamageSystem setzt den Blockzustand auf "BlockedSuccessfully", löscht den Block-Timer und spielt die Block-Animation ab, bevor der Blockvorgang beendet wird.

Event EndBlock



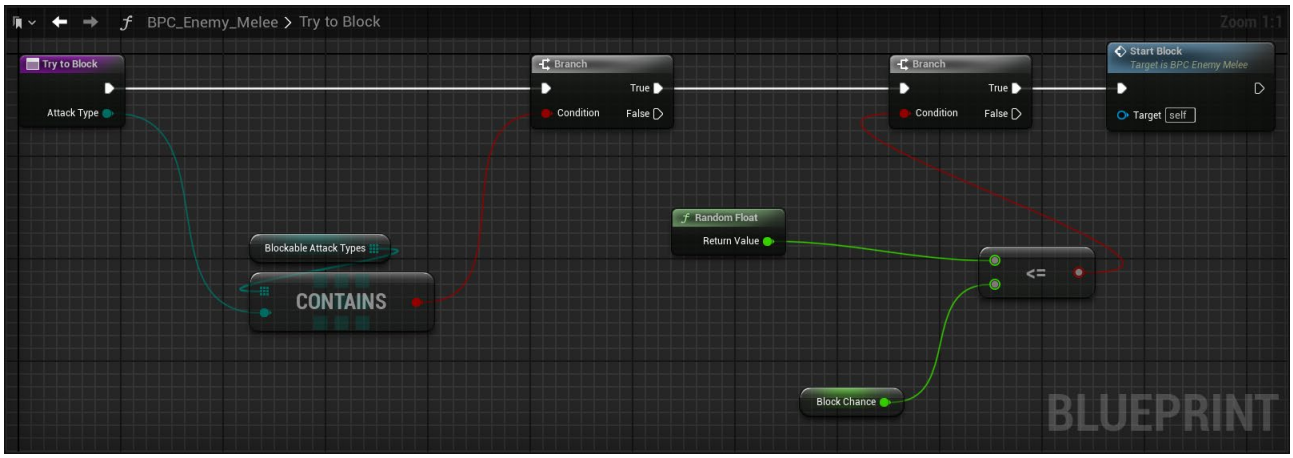
Das Event deaktiviert den Blockmodus im DamageSystem, setzt den Blockzustand zurück und informiert das System, dass der Blockvorgang beendet ist.

Override Funktion TakeDamage



Diese Funktion überschreibt die Standard-Schadensverarbeitung und fügt eine zusätzliche Überprüfung hinzu, ob der Schaden blockiert werden kann. Wenn der Schaden blockiert werden kann, wird versucht, den Schaden zu blocken, bevor die eigentliche Schadensverarbeitung durchgeführt wird.

Funktion TryToBlock



Diese Funktion versucht, eingehende Angriffe zu blockieren. Zunächst wird überprüft, ob der Angriffstyp blockierbar ist. Wenn ja, wird anhand einer Zufallsberechnung entschieden, ob der Blockversuch erfolgreich ist. Wenn die Zufallsberechnung positiv ausfällt, wird der Blockvorgang gestartet. Wird die Blockchance beim Melee Enemy 100 Prozent gesetzt werden alle blockbaren Attack Types geblockt und können keinen Schaden auf ihn ausüben.

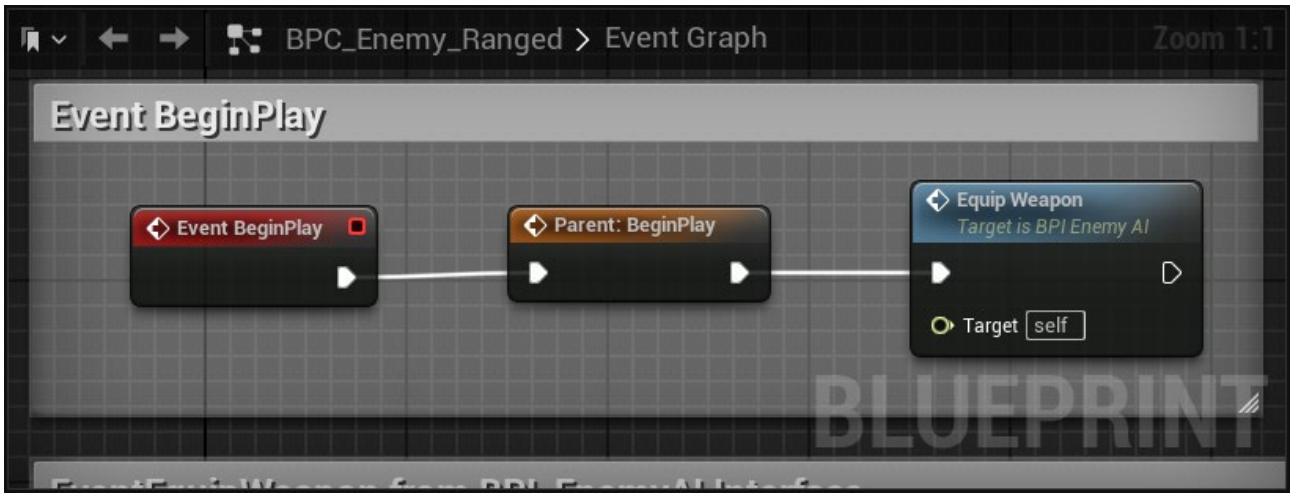
BPC_Enemy_Melee Childs

Die Blueprints BPC_Enemy_Melee_Air, BPC_Enemy_Melee_Earth, BPC_Enemy_Melee_Fire und BPC_Enemy_Melee_Water sind Child-Blueprints des BPC_Enemy_Melee. Der einzige Unterschied liegt in den festgelegten blockbaren AttackTypes. Jede dieser Variationen kann nur von Angriffen des entgegengesetzten Elements Schaden nehmen, da diese Angriffe nicht geblockt werden können.

Etwas spaeter kamen noch der BPC_Enemy_Melee_InAir, der BPC_Enemy_Melee_Heavy und der BPC_Enemy_Melee_OnlyParry hinzu. Auch Sie sind alle Childs des BPC_Enemy_Melee. Der BPC_Enemy_Melee_InAir nimmt nur schaden durch die launch und die InAir Attacken. Der BPC_Enemy_Melee_Heavy nimmt nur Schaden durch schwere Schläge und der BPC_Enemy_Melee_OnlyParry nimmt nur von GuardRevenge Attacken Schaden die der Player als Konter gegen einen Angriff ausführt.

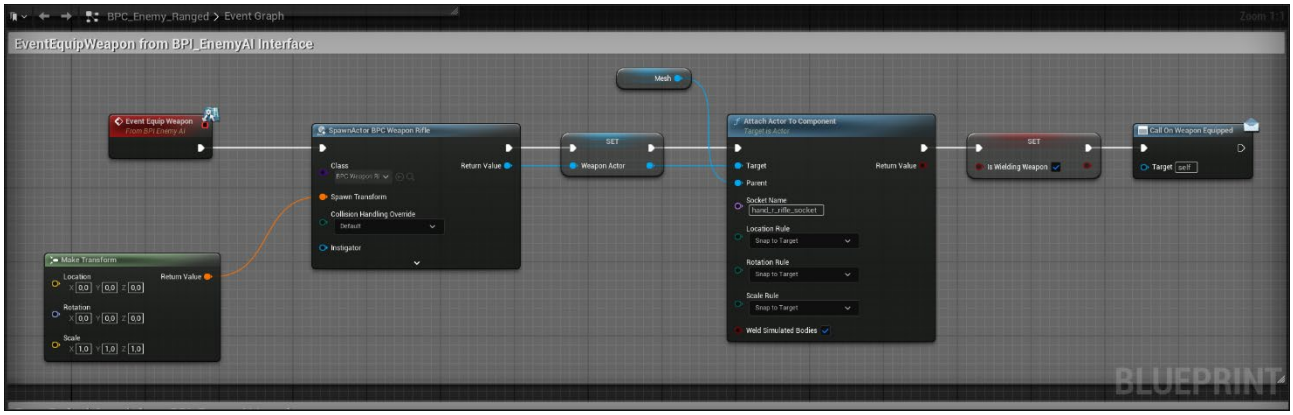
BPC_Ranged (Child des BP_Enemy)

Event BeginPlay



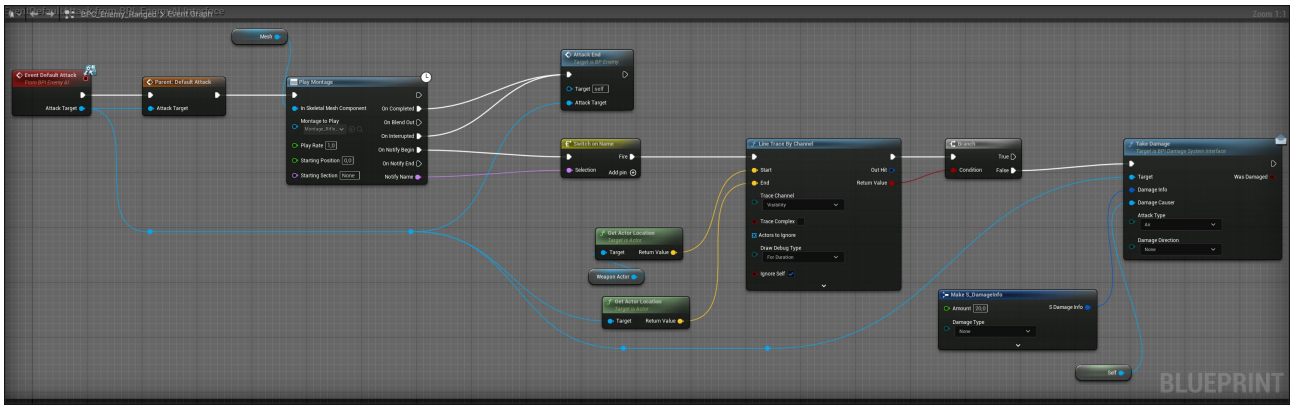
Dieses Event stellt sicher, dass der Ranged-Enemy ordnungsgemäß initialisiert und mit einer Waffe ausgerüstet wird. Die BeginPlay-Funktion der Elternklasse wird ebenfalls aufgerufen, um sicherzustellen, dass keine wichtige Initialisierung aus höheren Klassenstufen übergangen wird. Der Ranged-Enemy soll die Waffe immer ausgerüstet haben, somit kann dies gleich zu Beginn seiner Erstellung passieren.

Interface Event EquipWeapon



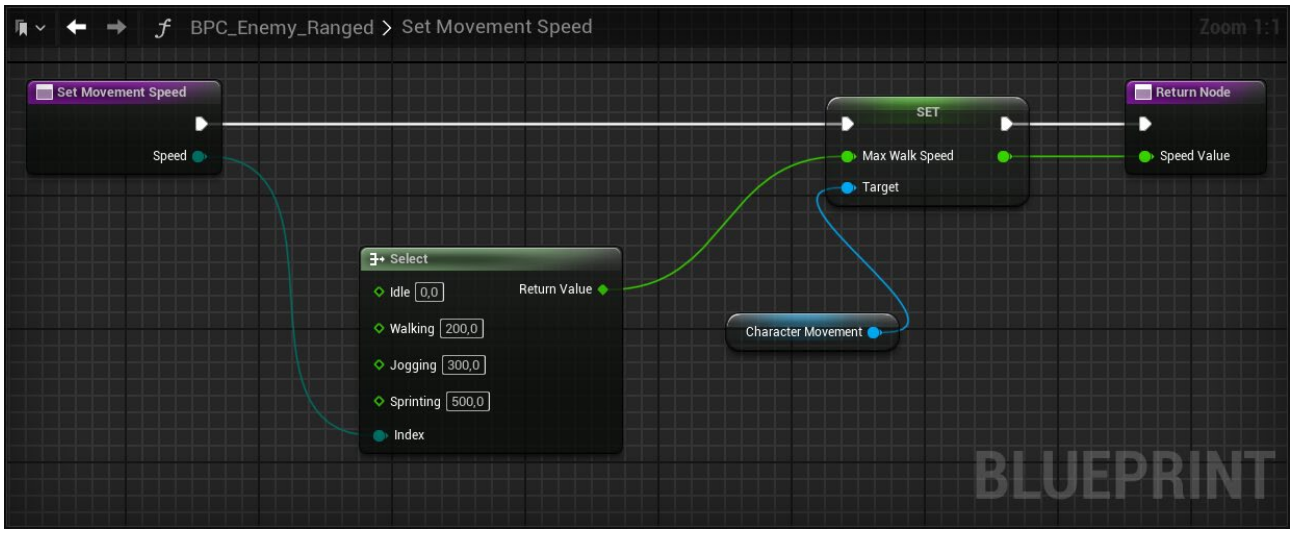
Dieses Event stellt sicher, dass der Ranged-Enemy erfolgreich mit der vorgesehenen Waffe ausgestattet wird, sobald das Event ausgelöst wird. Es beinhaltet das Spawnen der Waffe, das Zuweisen der Waffenreferenz, das Anhängen der Waffe an das Skelettmesh des Enemies und das Setzen von booleschen Werten zur Anzeige, dass der Gegner jetzt eine Waffe trägt. Schließlich wird ein Event ausgelöst, das den erfolgreichen Abschluss der Waffen-Ausrüstung signalisiert.

Interface Event DefaultAttack



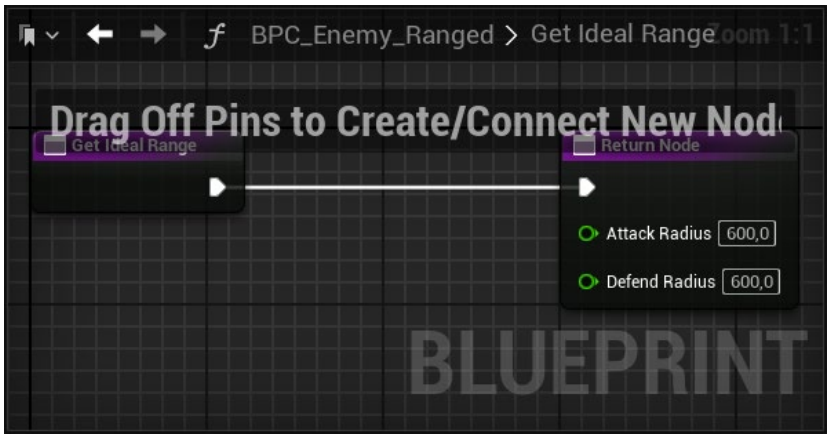
Der Event DefaultAttack aus de, BPI_EnemyAI Interface führt lässt zuerst die Logi des Parents durchlaufen, spielt dann die Angriffsmontage ab, führt durch das Auslösen der Anim Notify Fire eine Sichtprüfung von der Schusswaffe aus zum Player durch, und wendet Schaden an, wenn das Ziel sichtbar ist.

Override Funktion SetMovementSpeed



Diese Funktion überschreibt die Bewegungsgeschwindigkeiten des BP_Enemy um sie für den Ranged-Enemy neu zu definieren.

Override Funktion GetIdealRange



Diese Funktion überschreibt die Ideal Abstände des BP_Enemy um sie für den Ranged-Enemy neu zu definieren. Da der Ranged-Enemy immer eine gewisse Distanz einhalten soll und bei der Attack nicht näher zum Player kommen soll. Wurden die beiden Variablen auf den gleichen Wert gesetzt.

BP_PatrolRoute

Die BP_PatrolRoute ist eine Blueprint-Klasse, die verwendet wird, um eine Patrouillenroute für NPCs (Nicht-Spieler-Charaktere) zu definieren. Diese Klasse nutzt eine Spline-Komponente, um eine Serie von Punkten im Level zu erstellen, entlang derer NPCs patrouillieren können.

Komponenten und Variablen

Spline Component

Die Hauptkomponente der BP_PatrolRoute. Diese Spline-Komponente definiert die Punkte der Patrouillenroute. Der Designer kann diese Punkte im Level-Editor visuell anpassen, um die Route zu erstellen, die der NPC verfolgen soll.

Patrol Index

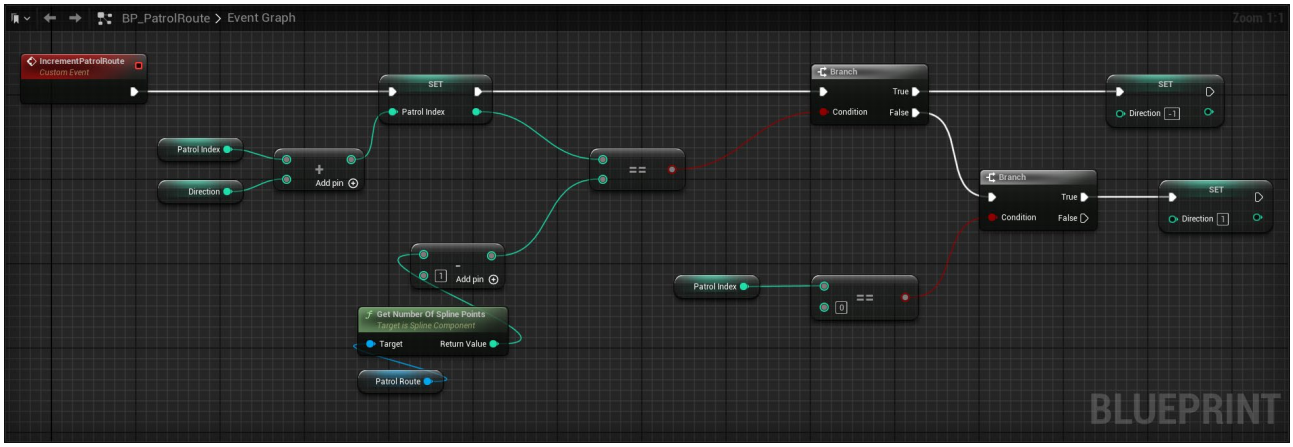
Eine Integer-Variable, die den aktuellen Index des Spline-Punkts darstellt, an dem sich der NPC befindet oder zu dem er sich bewegen soll. Dieser Index wird verwendet, um den Fortschritt des NPCs entlang der Route zu verfolgen.

Direction

Eine Integer-Variable, die die Bewegungsrichtung des NPCs entlang der Route definiert. Diese Variable hat in der Regel zwei mögliche Werte: 1 für vorwärts und -1 für rückwärts. Sie wird verwendet, um zu bestimmen, ob der NPC in Richtung des nächsten Punkts oder zurück zum vorherigen Punkt bewegt wird.

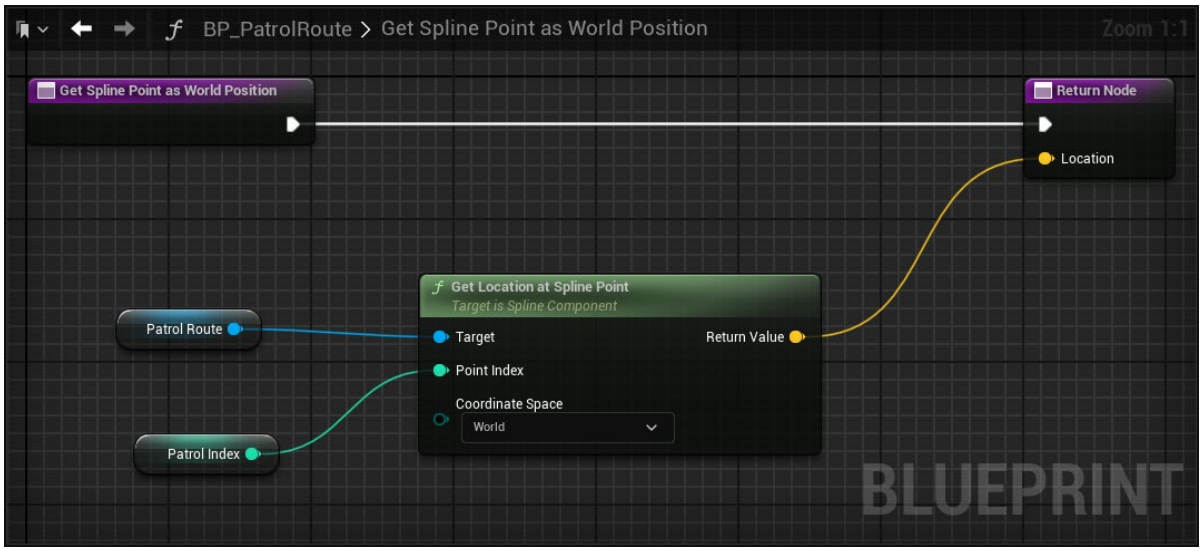
Funktionen

IncrementPatrolRoute



Diese Funktion wird aufgerufen, um den Patrol Index zu aktualisieren und die Bewegungsrichtung gegebenenfalls zu ändern. Wenn der Patrol Index das Ende oder den Anfang der Route erreicht, wird die Direction umgekehrt.

GetSplinePointAsWorldPosition



Diese Funktion gibt die Weltposition eines bestimmten Spline-Punkts zurück, basierend auf dem Patrol Index. Sie Verwendet den Patrol Index, um die Position des entsprechenden Punkts auf der Spline-Komponente zu ermitteln und gibt die Position in Weltkoordinaten zurück, damit der NPC weiß, wohin er sich als nächstes bewegen soll.

Zusammenfassung

Die BP_PatrolRoute-Klasse dient als Blueprint, um eine Patrouillenroute für NPCs zu erstellen. Die Spline-Komponente definiert die Route, während der Patrol Index und die Direction verwendet werden, um den Fortschritt des NPCs entlang dieser Route zu verfolgen und zu steuern. Mit Funktionen wie IncrementPatrolRoute und GetSplinePointAsWorldPosition kann der NPC die Route durchlaufen, die Richtungen ändern und die Positionen der Spline-Punkte ermitteln, um sich korrekt im Spiel zu bewegen.

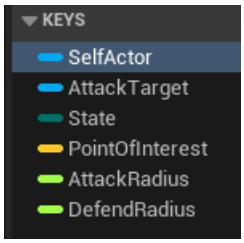
Behavior Trees

Um komplexe Behavior Trees zu erstellen, sind neben dem Behavior Tree selbst mehrere Schlüsselkomponenten erforderlich:

1. **Blackboard**:
    - Ein strukturiertes Datenhaltesystem, das Werte speichert, die von verschiedenen Knoten im Behavior Tree verwendet werden.
2. **Tasks**:
    - Grundlegende Bausteine in Behavior Trees, die konkrete Aktionen oder Arbeiten ausführen oder diese in einem anderen Blueprint starten oder beenden.
3. **Decorators**:
    - Werden verwendet, um Bedingungen festzulegen, die bestimmen, ob ein Knoten ausgeführt werden soll oder nicht.
4. **Services**:
    - Führen regelmäßig Arbeiten aus, um bestimmte Aufgaben zu erledigen. Typischerweise werden sie verwendet, um kontinuierlich Zustände zu überprüfen, Blackboard-Werte zu aktualisieren oder andere wiederkehrende Aufgaben zu erledigen, während der Behavior Tree ausgeführt wird.
5. **Enviroment Querys**:
    - Ermöglichen dynamische Abfragen in der Spielwelt, um optimale Positionen zu bestimmen, wichtige Objekte oder Akteure zu identifizieren, effiziente Pfade zu berechnen und kontextbasierte Entscheidungen basierend auf Spielzuständen zu treffen.

Diese Komponenten gibt es in vorgefertigter Form, die genutzt werden können. Allerdings beschränkt sich die Funktionalität dieser vorgefertigten Komponenten auf die Basisfunktionen eines NPC-Verhaltens. Um komplexes NPC-Verhalten umzusetzen, müssen diese um neu erstellte erweitert werden.

Blackboard BB_Enemy_Base



Das BB_Enemy_Base Blackboard enthält Schlüssel, die für die Steuerung und Entscheidungsfindung der Feind-AI unerlässlich sind. Es speichert wichtige Informationen über den eigenen Akteur, das Ziel, den Zustand, Punkte von Interesse und die Angriffs- sowie Verteidigungsradien. Diese Schlüssel werden in den verschiedenen Behavior Trees der Feinde verwendet, um das Verhalten dynamisch zu steuern und anzupassen.

Es besteht ausfolgenden Daten:

SelfActor

- **KeyType**: Actor Object
- **Beschreibung**: Repräsentiert den eigenen Akteur (Self Actor) im Spiel. Dieser Key wird verwendet, um auf den Akteur selbst zuzugreifen.

AttackTarget

- **KeyType**: Actor Object
- **Beschreibung**: Das Ziel, das der Feind angreifen soll. Dieser Key wird verwendet, um den Ziel-Akteur im Verhalten des Feindes zu definieren und zu verfolgen.

State

- **KeyType**: Enum (E_EnemyAiStates)
- **Beschreibung**: Der aktuelle Zustand des Feindes. Dieser Key verwendet das Enum E_EnemyAiStates, um verschiedene Zustände wie Patrouillieren, Angreifen, Verteidigen usw. darzustellen.

PointOfInterest

- **KeyType**: Vector
- **Beschreibung**: Ein Punkt von Interesse im Raum, den der Feind anvisieren oder zu dem er sich bewegen soll. Dieser Key wird verwendet, um Positionsinformationen zu speichern.

AttackRadius

- **KeyType**: Float
- **Beschreibung**: Der Angriffsradius des Feindes. Dieser Key speichert den Radius, in dem der Feind angreifen kann, und wird verwendet, um Angriffsentscheidungen basierend auf der Entfernung zum Ziel zu treffen.

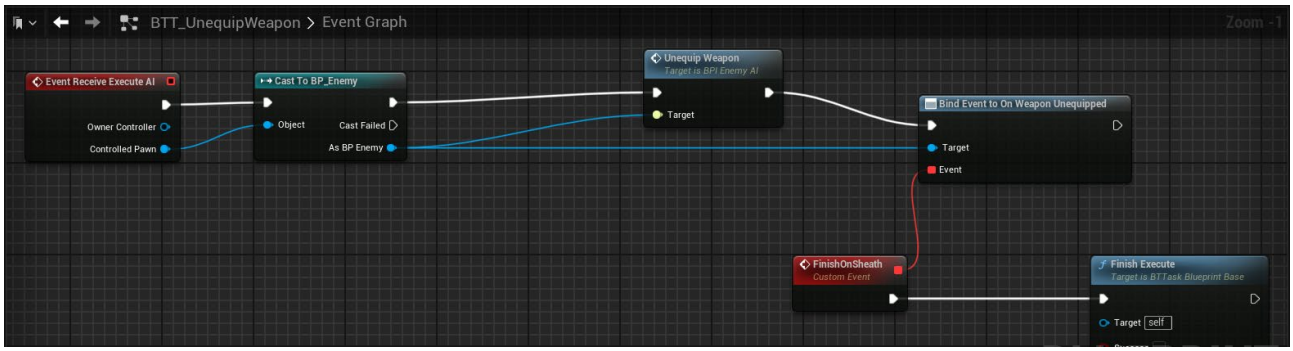
DefendRadius

- **KeyType**: Float
- **Beschreibung**: Der Verteidigungsradius des Feindes. Dieser Key speichert den Radius, in dem der Feind verteidigen kann, und wird verwendet, um Verteidigungsentscheidungen basierend auf der Entfernung zu Bedrohungen zu treffen.

Tasks

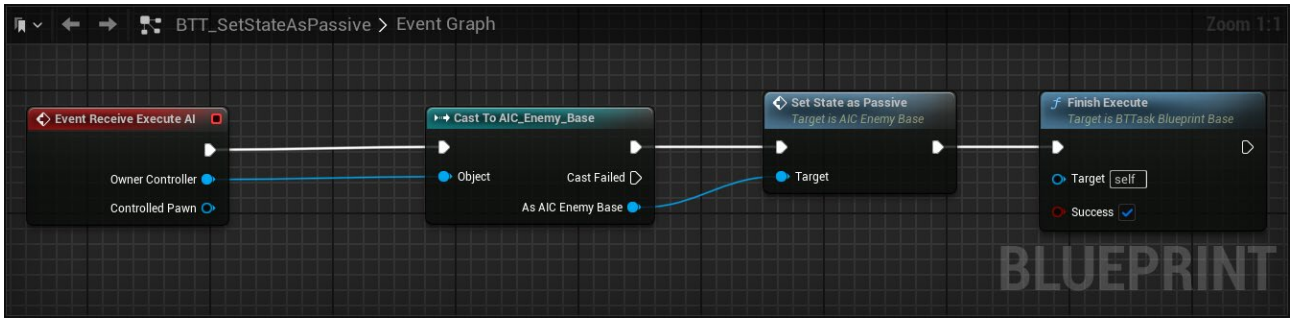
Für die Umsetzung des Enemy Verhaltens wurden neue Tasks erstellt. Die Tasks nutzen den Parent BTTask Blueprint Base und beginnen mit einem Funktionsaufruf **Event Receive Execute AI**, welcher vom Behavior Tree ausgeführt wird, wenn der Task aktiviert wird und Enden mit dem Finish Execute der den Task im Behavior Tree beendet.

BTT_UnequipWeapon



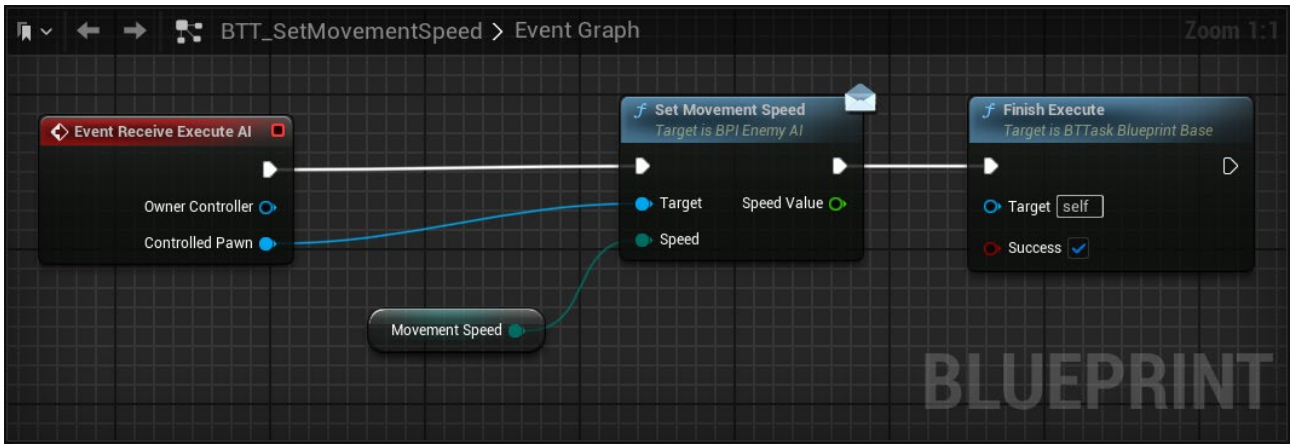
Dieser Task kümmert sich um das Entfernen der Waffe von einem BP_Enemy Charakter. Er überprüft zunächst, ob der kontrollierte Pawn tatsächlich ein BP_Enemy ist, führt die Unequip Weapon Funktion aus und wartet dann auf das FinishOnSheath Event, bevor er den Task erfolgreich beendet.

BTT_SetStateAsPassive



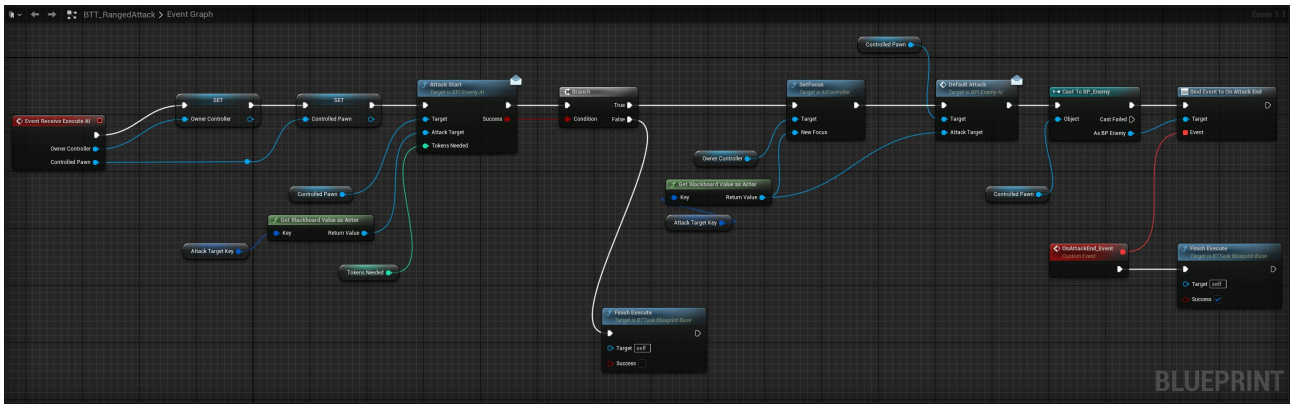
Dieser Task kümmert sich darum, den Status eines AIC_Enemy_Base Charakters auf passiv zu setzen. Er überprüft zunächst, ob der kontrollierte Pawn tatsächlich ein AIC_Enemy_Base ist, führt die **Set State as Passive** Funktion aus und beendet dann den Task erfolgreich.

BTT_SetMovementSpeed



Dieser Task kümmert sich darum, die Bewegungsgeschwindigkeit eines BPI_Enemy AI Charakters zu setzen. Er wird vom Behavior Tree ausgeführt, setzt die Bewegungsgeschwindigkeit auf den festgelegten Wert und beendet dann den Task erfolgreich.

BTT_RangedAttack



Das Event beginnt mit einem Funktionsaufruf **Event Receive Execute AI**, welcher vom Behavior Tree ausgeführt wird, wenn der Task aktiviert wird.

Setzen der Variablen:

- **SET**: Setzt die Variablen **Owner Controller** und **Controlled Pawn**.

Abrufen der Blackboard-Werte:

- **Get Blackboard Value as Actor**: Holt den Wert des **Attack Target Key** aus dem Blackboard und setzt ihn als **Attack Target**.

Starten des Angriffs:

- **Attack Start**: Ruft die Funktion auf, um den Angriff zu starten.
  - **Target**: Setzt das Ziel für diese Funktion auf den kontrollierten Pawn.
  - **Attack Target**: Übergibt das Ziel des Angriffs.
  - **Tokens Needed**: Übermittelt die benötigten Tokens für den Angriff.

- **Branch**: Überprüft, ob der Angriff erfolgreich gestartet wurde. Falls nicht wird Finish Execute mit Success False ausgelöst.

Fokussieren des Ziels:

- **Set Focus**: Setzt den Fokus des Controllers auf das **New Focus** Ziel.

Ausführen des Angriffs:

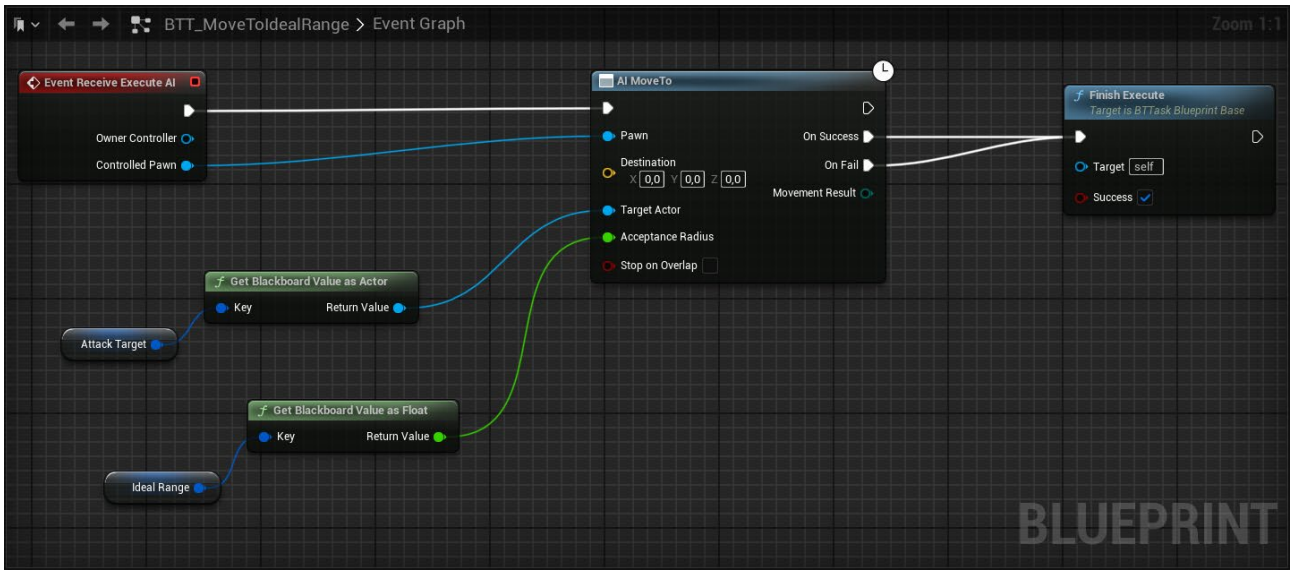
- **Default Attack**: Ruft die Standardangriffs-Funktion auf.
    - **Target**: Setzt das Ziel für diese Funktion.
    - **Attack Target**: Übergibt das Ziel des Angriffs.
- **Cast To BP_Enemy**: Überprüft, ob der kontrollierte Pawn vom Typ **BP_Enemy** ist.
    - **Object**: Der kontrollierte Pawn wird als Eingabeobjekt verwendet.
    - **Cast Failed**: Verarbeitet die Bedingung, wenn der Cast fehlschlägt.
- **Bind Event to On Attack End**: Bindet ein Event an den Dispatcher **On Attack End**, um auf das Signal zu warten, dass der Angriff beendet ist.
    - **Target**: Das Ziel ist der BP_Enemy.
    - **Event**: Das Event, das ausgelöst wird, wenn der Angriff beendet ist.

Beenden des Tasks:

- **OnAttackEnd_Event**: Ein benutzerdefiniertes Event, das ausgelöst wird, sobald der Angriff beendet ist.
- **Finish Execute**: Beendet den Task im Behavior Tree erfolgreich.
    - **Success**: Gibt an, dass der Task erfolgreich abgeschlossen wurde.

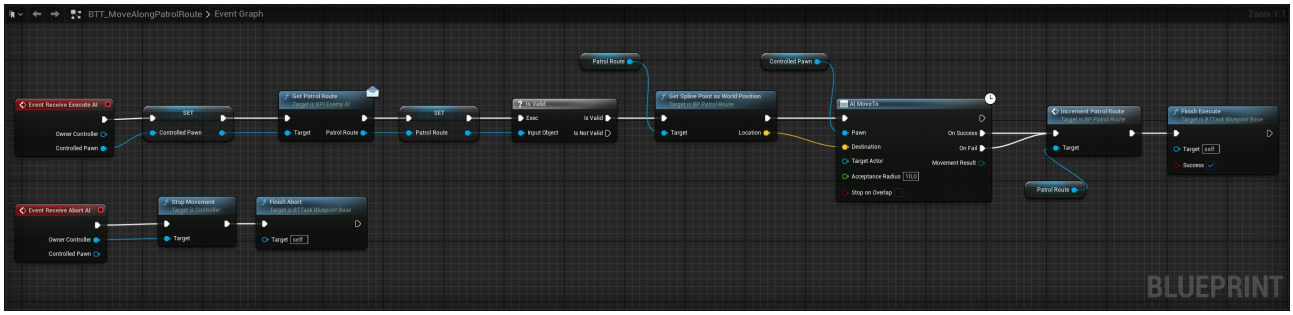
Zusammengefasst kümmert sich dieser Task um den Fernkampfangriff eines **BP_Enemy** Charakters. Er wird vom Behavior Tree ausgeführt, setzt den Fokus auf das Angriffsziel, startet den Angriff und wartet auf das **On Attack End** Event, bevor er den Task erfolgreich beendet.

BTT_MoveToIdealRange



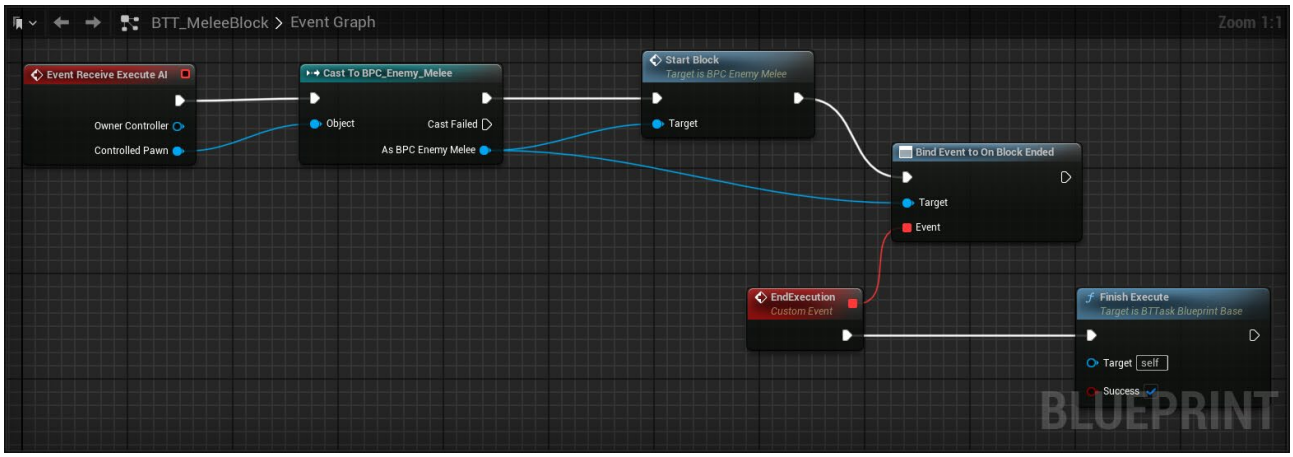
Dieser Task kümmert sich um das Bewegen eines **BP_Enemy** Charakters auf die Ideal Range für einen Angriff. Er wird vom Behavior Tree ausgeführt, holt die notwendigen Werte aus dem Blackboard, bewegt den Enemy so dass sich das AttackTarget im Acceptance Radius befinden und beendet den Task erfolgreich, wenn die Bewegung abgeschlossen ist.

BTT_MoveAlongPatrolRoute



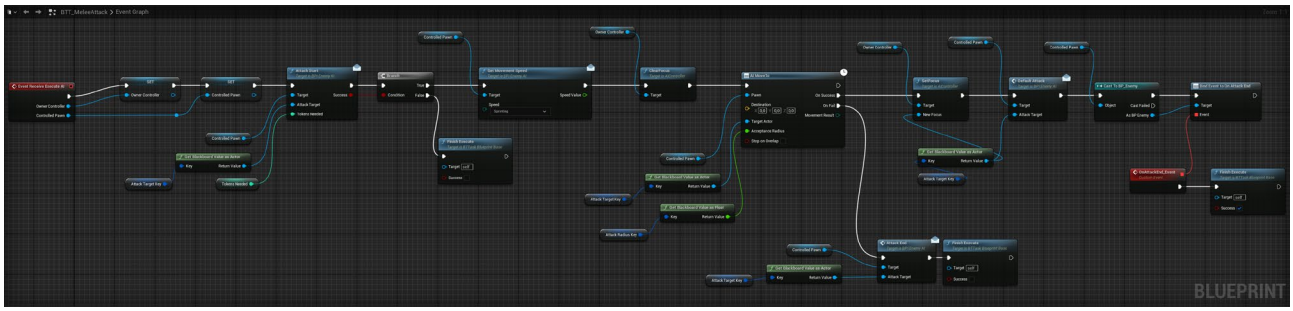
Dieser Task kümmert sich um das Bewegen eines BP_Enemy Charakters entlang einer vordefinierten Patrouillenroute. Er wird vom Behavior Tree ausgeführt, holt die notwendigen Informationen über die Patrouillenroute, bewegt den Charakter zum nächsten Punkt auf der Route und aktualisiert die Route für die nächste Bewegung. Der Task wird erfolgreich beendet, sobald die Bewegung abgeschlossen ist. Außerdem stoppt er beim Abbruch des Tasks die Bewegung des AI-Controllers und gibt leitet anschließend den Finish Abort ein.

BTT_MeleeBlock



Dieser Task kümmert sich um das Blocken eines BP_Enemy Charakters. Er überprüft zunächst, ob der kontrollierte Pawn tatsächlich ein BPC_Enemy_Melee ist, führt die Start Block Funktion aus und wartet dann auf das EndExecution Event, bevor er den Task erfolgreich beendet.

BTT_MeleeAttack



Das Event beginnt mit einem Funktionsaufruf, welcher vom Behavior Tree ausgeführt wird, wenn der Task aktiviert wird.

Setzen von Variablen:

- **SET Owner Controller**: Setzt den Owner Controller.
- **SET Controlled Pawn**: Setzt den kontrollierten Pawn.

Zielsetzung und Angriffsausführung:

- **Attack Start**: Ruft die Angriffsfunktion aus dem BPI_Enemy AI Interface auf.
    - **Target**: Setzt den Angreifer.
    - **Attack Target**: Das Angriffsziel aus dem Blackboard.
    - **Tokens Needed**: Die benötigten Tokens für den Angriff.

Bedingungsprüfung:

- **Branch**: Überprüft, ob die Attacke gestartet werden konnte
    - **True**: Logik läuft weiter.
    - **False**: Finish Execute Success False.

Bewegungssteuerung:

- **Set Movement Speed**: Setzt die Bewegungsgeschwindigkeit auf Sprinten.

Fokus setzen:

- **Clear Focus**: Entfernt den aktuellen Fokus.
- **Set Focus**: Setzt einen neuen Fokus.
    - **Target**: Das Ziel, auf das fokussiert werden soll.
    - **New Focus**: Der neue Fokus.

Bewegung zum Ziel:

- **AI MoveTo**: Bewegt den Enemy zum Ziel.
    - **Pawn**: Der Enemy.
    - **Destination**: Das Ziel, zu dem sich der Enemy bewegen soll.
    - **Acceptance Radius**: Der Radius, in dem das Ziel als erreicht gilt.
    - **OnFail**: Die Attacke wird beendet und der Execute mit False beendet

Zielwert aus dem Blackboard:

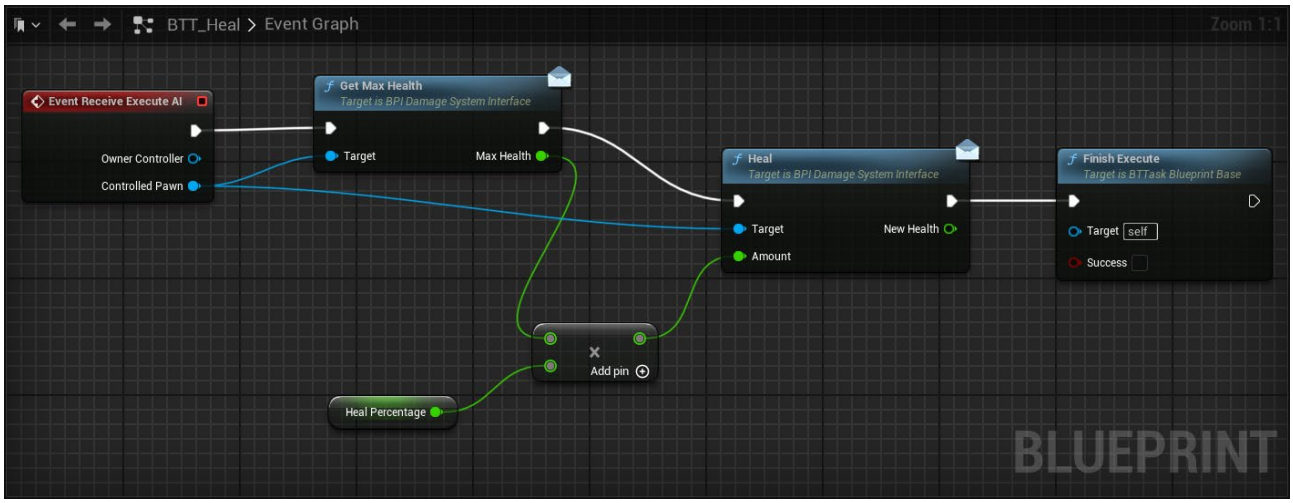
- **Get Blackboard Value as Actor**: Holt den Zielwert aus dem Blackboard.
    - **Key**: Der Schlüssel für den Zielwert.
    - **Return Value**: Der zurückgegebene Wert.

Angriffsfunktion und Abschluss:

- **Default Attack**: Führt den Standardangriff aus.
    - **Target**: Das Ziel des Angriffs.
    - **Attack Target**: Das Angriffsziel.
- **Cast To BP_Enemy**: Überprüft, ob der kontrollierte Pawn vom Typ BP_Enemy ist.
    - **Object**: Der kontrollierte Pawn wird als Eingabeobjekt verwendet.
    - **Cast Failed**: Verarbeitet die Bedingung, wenn der Cast fehlschlägt.
- **Bind Event to On Attack End**: Bindet ein Event an den Dispatcher On Attack End, um auf das Signal zu warten, dass der Angriff beendet ist.
    - **Target**: Das Ziel ist der BP_Enemy.
    - **Event**: Das Event, das ausgelöst wird, wenn der Angriff beendet ist.
- **Finish Execute**: Beendet den Task im Behavior Tree erfolgreich.
    - **Success**: Gibt an, dass der Task erfolgreich abgeschlossen wurde.
    -

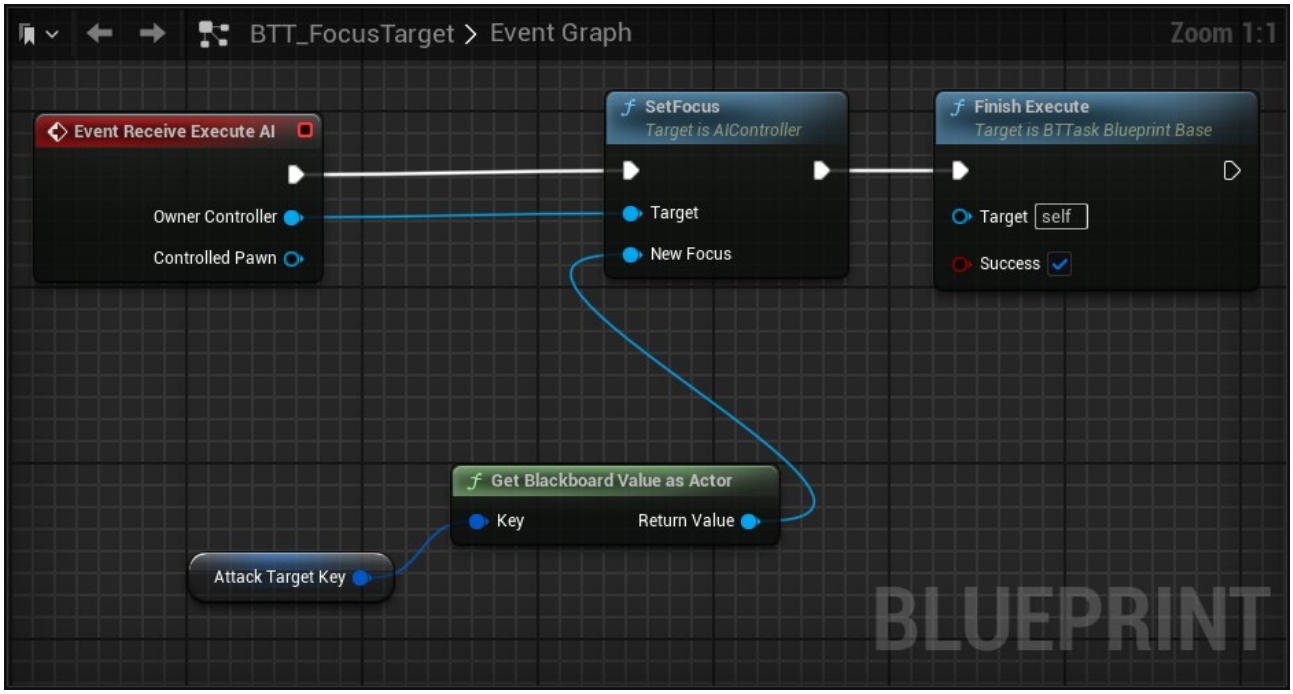
Zusammengefasst kümmert sich dieser Task um den Nahkampfangriff eines Enemy. Er überprüft zunächst, ob der kontrollierte Pawn tatsächlich ein BP_Enemy ist, führt den Angriff aus und wartet dann auf das On Attack End Event, bevor er den Task erfolgreich beendet.

BTT_Heal



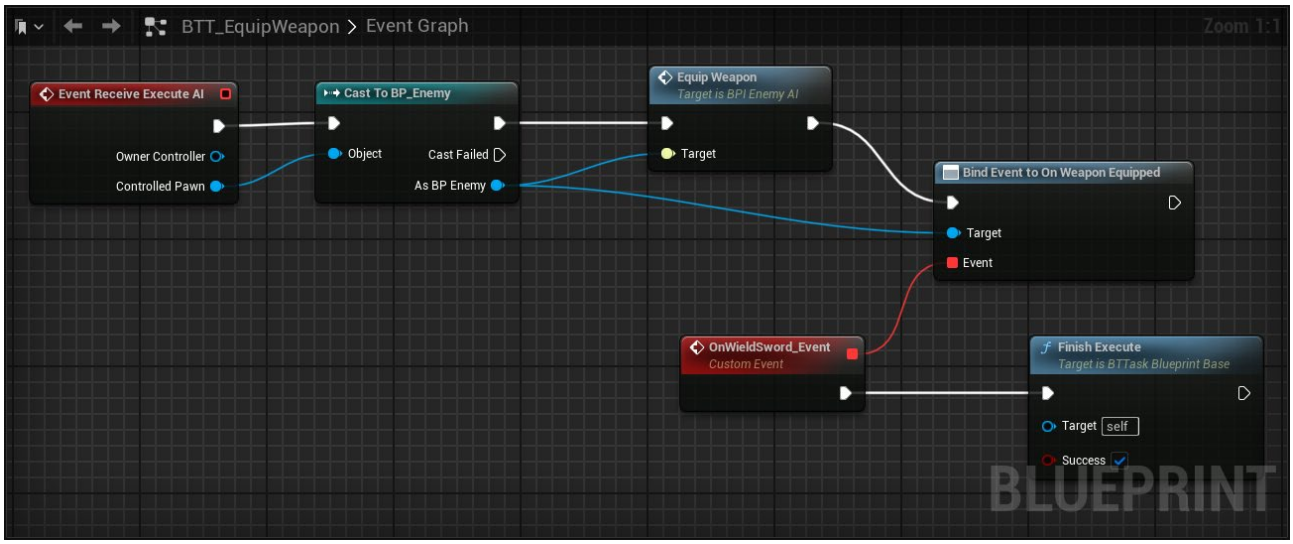
Dieser Task kümmert sich um das Heilen eines Enemies. Er ermittelt zunächst die maximale Gesundheit des Charakters, berechnet den Heilungsbetrag basierend auf einem vordefinierten Prozentsatz und ruft dann die Heilungsfunktion auf, um diesen Betrag anzuwenden. Abschließend wird der Task erfolgreich beendet.

BTT_FocusTarget



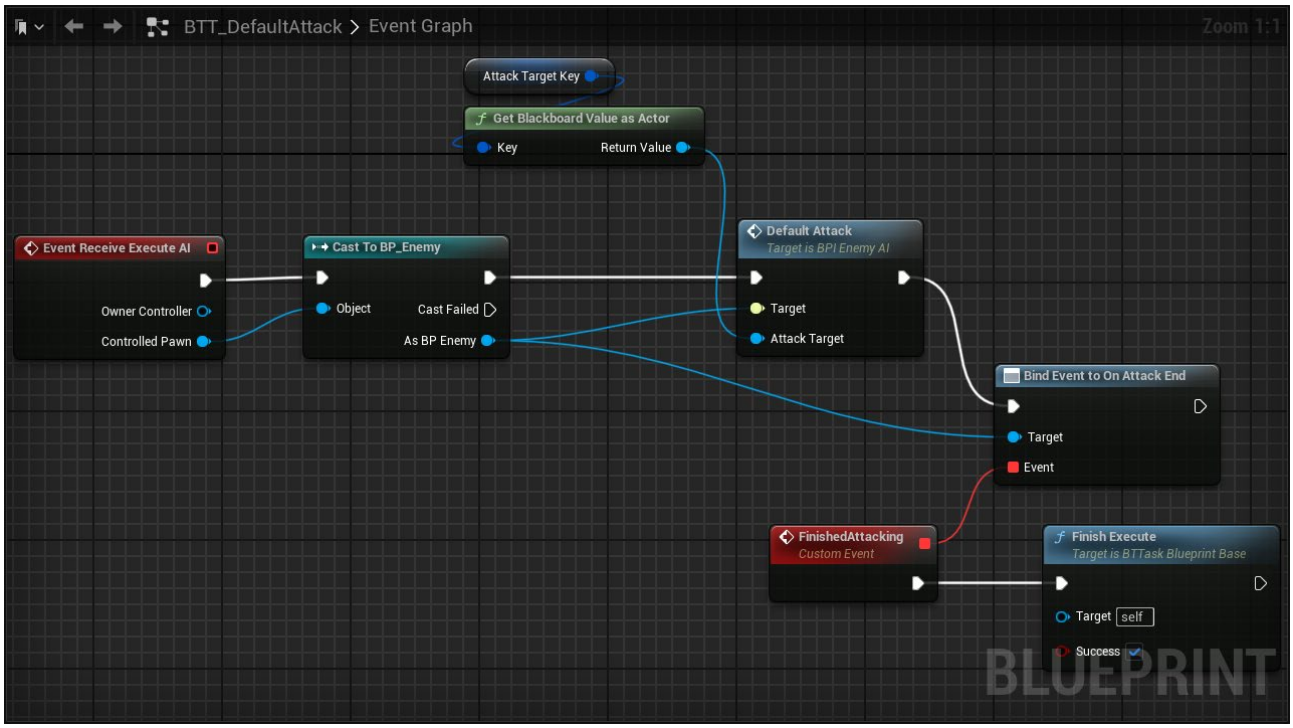
Dieser Task kümmert sich darum, dass ein AI-Charakter ein bestimmtes Ziel fokussiert. Er ruft das Ziel aus dem Blackboard ab, setzt den Fokus auf das Ziel und beendet den Task erfolgreich.

BTT_EquipWeapon



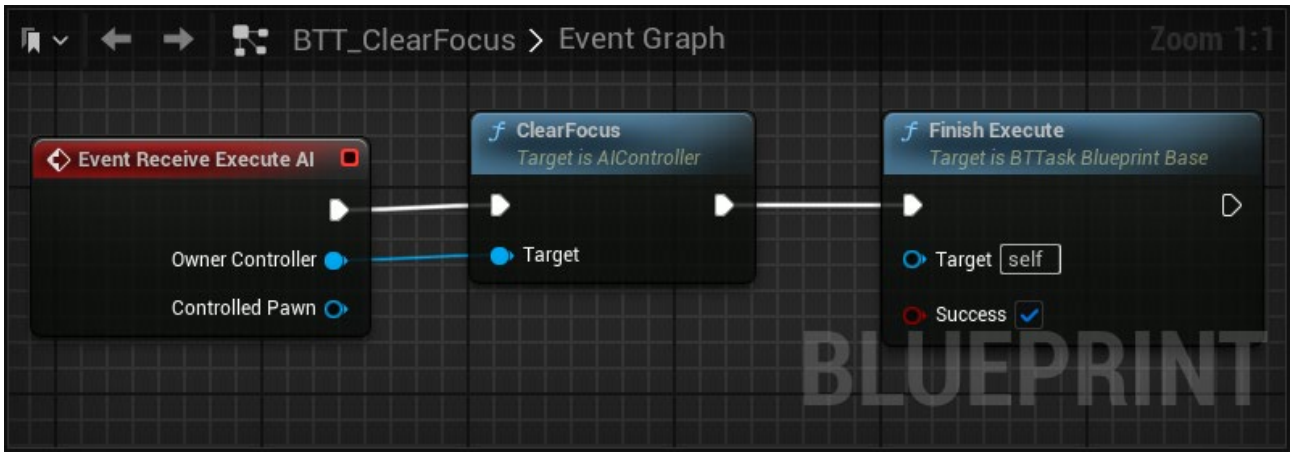
Dieser Task kümmert sich um das Ausrüsten einer Waffe für einen BP_Enemy Charakter. Er überprüft zunächst, ob der kontrollierte Pawn tatsächlich ein BP_Enemy ist, führt die Equip Weapon Funktion aus und wartet dann auf das OnWieldSword Event, bevor er den Task erfolgreich beendet.

BTT_DefaultAttack



Dieser Task kümmert sich um das Ausführen eines Standardangriffs für einen BP_Enemy Charakter. Er überprüft zunächst, ob der kontrollierte Pawn tatsächlich ein BP_Enemy ist, holt das Angriffsziel aus dem Blackboard, führt die Default Attack Funktion aus und wartet dann auf das FinishedAttacking Event, bevor er den Task erfolgreich beendet.

BTT_ClearFocus

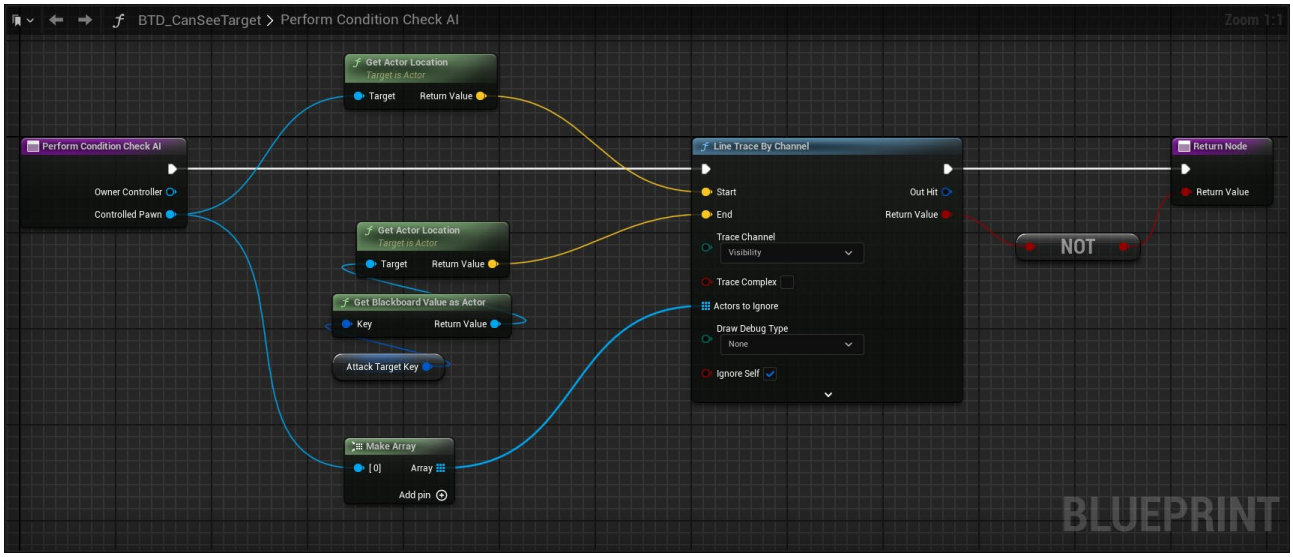


Dieser Task entfernt den aktuellen Fokus eines AI-Controllers. Er wird aktiviert, indem er vom Behavior Tree aufgerufen wird und führt die ClearFocus-Funktion aus, um den Fokus im AIController zu entfernen. Nach der erfolgreichen Ausführung dieser Funktion beendet er den Task und gibt an, dass der Task erfolgreich abgeschlossen wurde.

Decorator

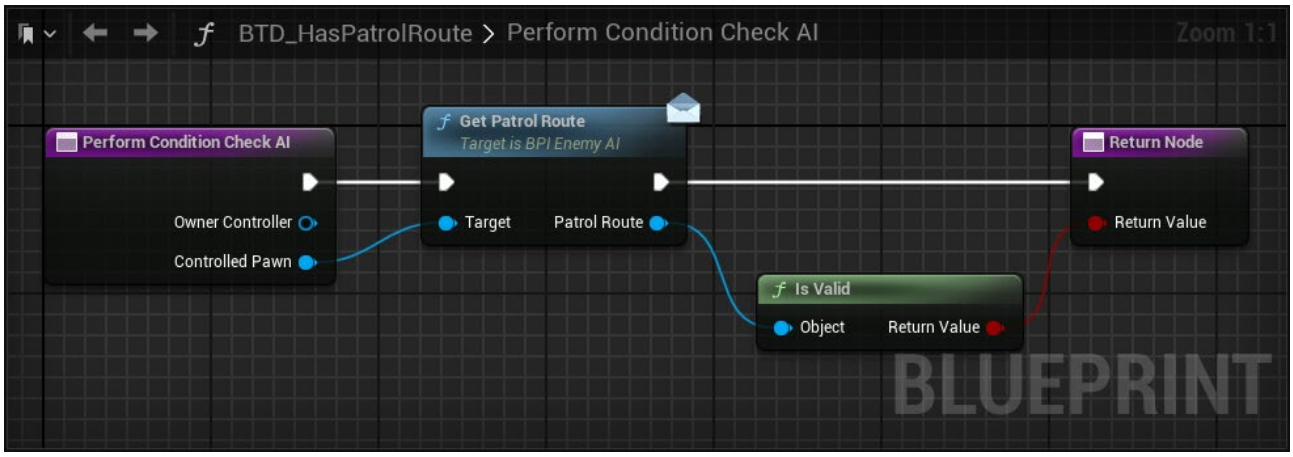
Zur Erweiterung der möglichen Bedingungen zu den Ausführungen einzelner Zweige innerhalb der Behavior Trees wurden neue Decorator erstellt. Dazu wird als Basis zur Erstellung jeweils der BTDecorator Blueprint Base verwendet und dessen Funktion PerformConditionCheckAI überschrieben.

BTD_CanSeeTarget



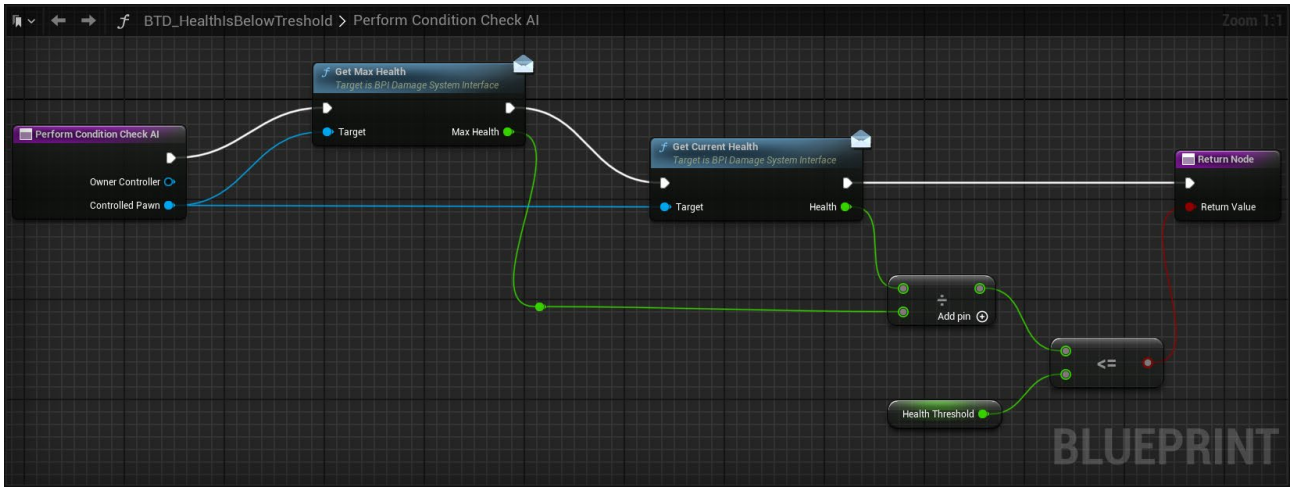
Dieser Decorator überprüft, ob der kontrollierte Akteur (Enemy) das Angriffsziel sehen kann. Er führt eine Linienverfolgung zwischen dem Akteur und dem Ziel durch, um festzustellen, ob eine direkte Sichtlinie besteht. Das Ergebnis wird invertiert und zurückgegeben, um anzuzeigen, ob das Ziel sichtbar ist.

BTD_HasPatrolRoute



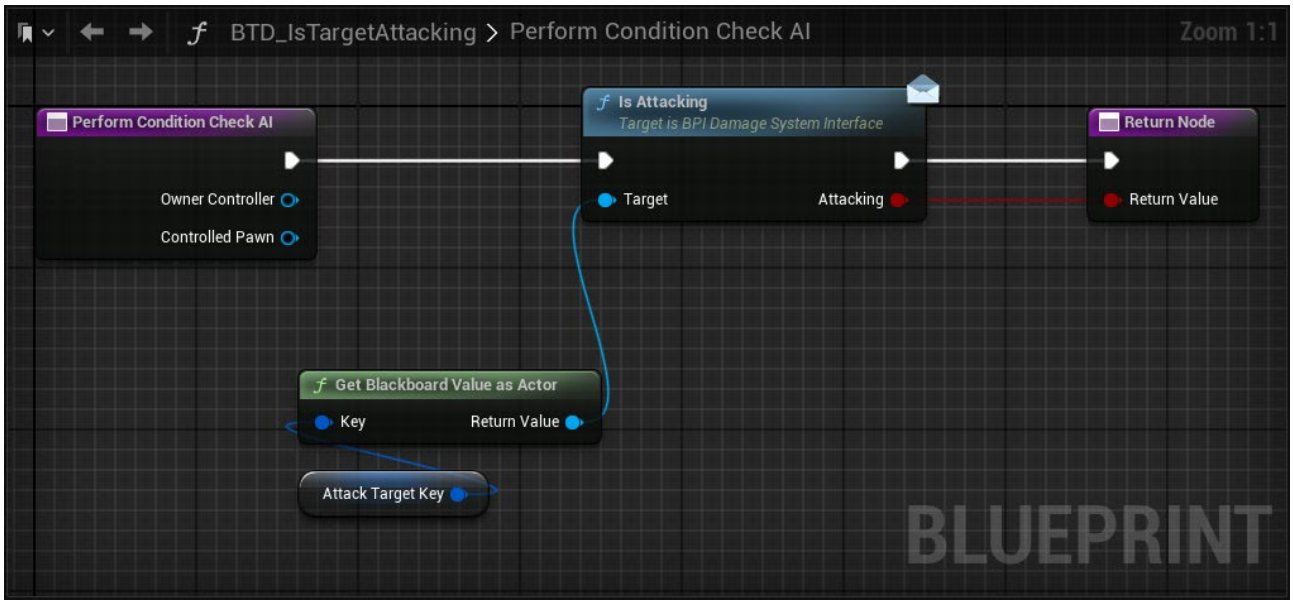
Dieser Decorator überprüft, ob der kontrollierte Akteur (Enemy) eine gültige Patrouillenroute hat. Er ruft die Patrouillenroute des Akteurs ab und überprüft dann, ob diese gültig ist. Das Ergebnis wird zurückgegeben, um anzuzeigen, ob der Akteur eine Patrouillenroute hat, der er folgen kann.

BTD_HealthIsBelowTreshold



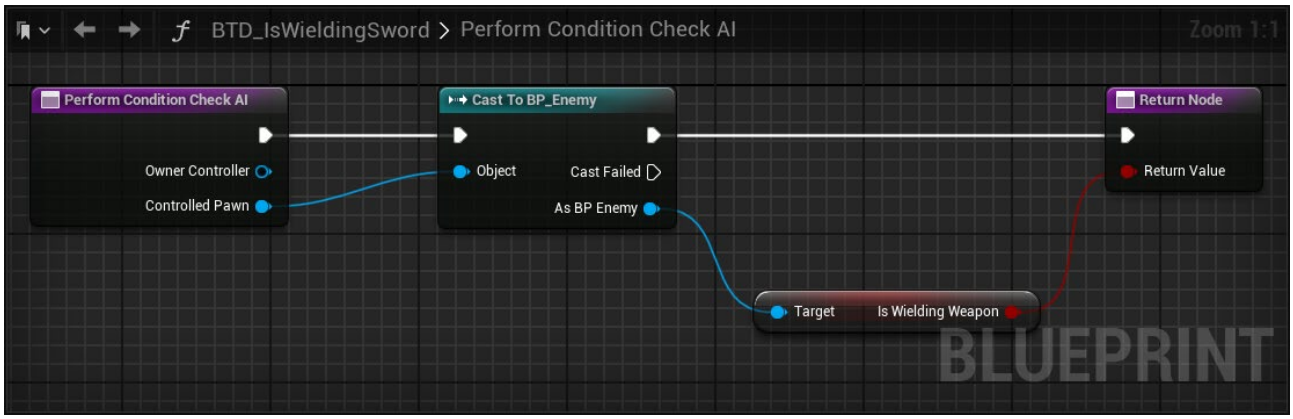
Dieser Decorator überprüft, ob die aktuelle Gesundheit des kontrollierten Akteurs unter einem definierten Schwellenwert liegt. Er ruft die maximale und aktuelle Gesundheit des Akteurs ab, berechnet das Verhältnis der aktuellen zur maximalen Gesundheit und vergleicht dieses Verhältnis mit einem definierten Schwellenwert. Das Ergebnis wird zurückgegeben, um anzuzeigen, ob die Gesundheit des Akteurs unter dem Schwellenwert liegt.

BTD_IsTargetAttacking



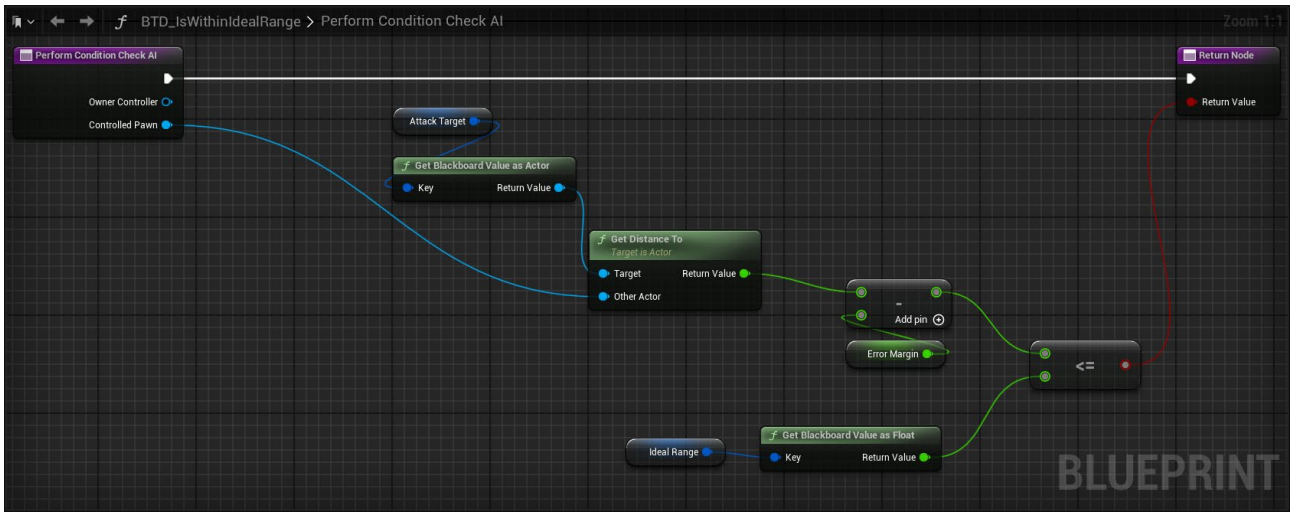
Dieser Decorator überprüft, ob das aktuelle Angriffsziel des kontrollierten Akteurs (Player) gerade einen Angriff ausführt. Er ruft den Actor des Angriffsziels aus dem Blackboard ab und überprüft mittels der "Is Attacking"-Funktion, ob dieser Actor gerade angreift. Das Ergebnis wird als boolescher Wert zurückgegeben, um anzuzeigen, ob das Angriffsziel gerade angreift.

BTD_IsWieldingSword



Dieser Decorator überprüft, ob der kontrollierte Pawn, der ein BP_Enemy ist, eine Waffe führt. Er führt zunächst einen Cast auf BP_Enemy durch, um sicherzustellen, dass der kontrollierte Pawn tatsächlich ein BP_Enemy ist. Anschließend wird die Funktion "Is Wielding Weapon" aufgerufen, um zu überprüfen, ob der BP_Enemy eine Waffe führt. Das Ergebnis wird als boolescher Wert zurückgegeben, um anzuzeigen, ob er eine Waffe führt.

BTD_IsWithinIdealRange

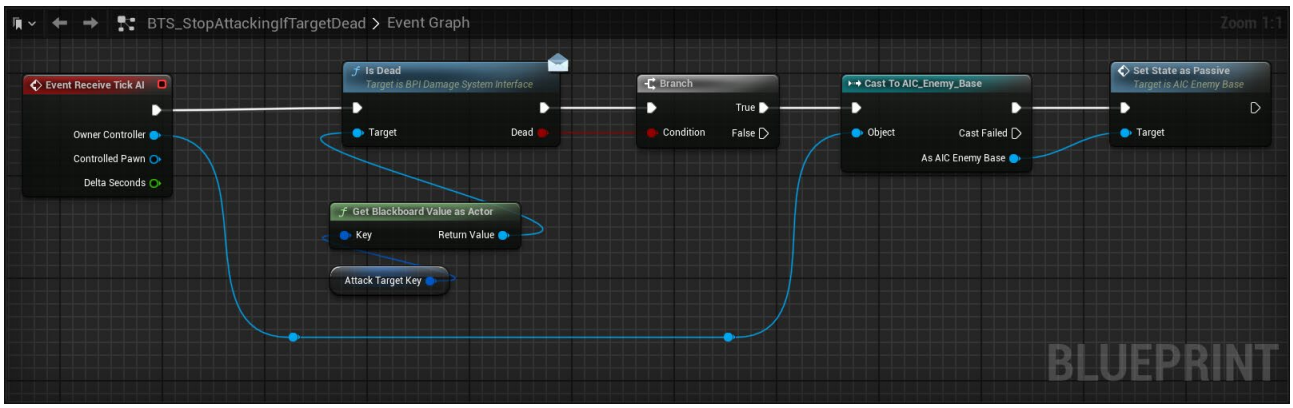


Dieser Decorator überprüft, ob der kontrollierte Enemy sich innerhalb eines idealen Angriffsbereichs (Ideal Range) zu seinem Angriffsziel (Attack Target) befindet. Er holt den Wert des Angriffsziels aus dem Blackboard, berechnet die Entfernung zwischen dem kontrollierten Pawn und dem Angriffsziel, und vergleicht diese Entfernung mit der Ideal Range, wobei eine gewisse Fehlermarge (Error Margin) berücksichtigt wird. Das Ergebnis wird als boolescher Wert zurückgegeben, um anzuzeigen, ob sich der Pawn innerhalb des idealen Bereichs befindet.

Services

Services sind mit OnTick Events vergleichbar. Das bedeutet das sie zwar sehr praktisch aber durch ihre häufige Ausführung Ressourcen lastig sind. Aus diesem Grund wurden sie so sparsam wie möglich eingesetzt. Ein Service musste trotzdem erstellt werden.

BTS_StopAttackingIfTargetDead



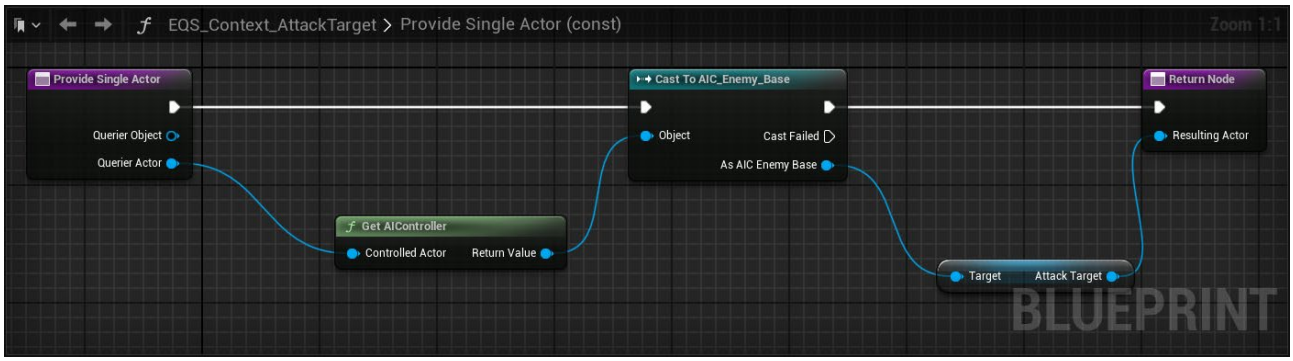
Dieser Service überprüft kontinuierlich (jeden Tick), ob das Angriffsziel (Attack Target) des kontrollierten Pawns tot ist. Wenn das Ziel tot ist, wird der Zustand des kontrollierten Pawns auf passiv gesetzt, indem die Set State as Passive Funktion aufgerufen wird.

Durch die Verwendung des Is Dead Checks und der Bedingungsprüfung wird sichergestellt, dass der Pawn (Enemy) nicht weiter angreift, wenn das Ziel bereits tot ist.

Enviroment Querys

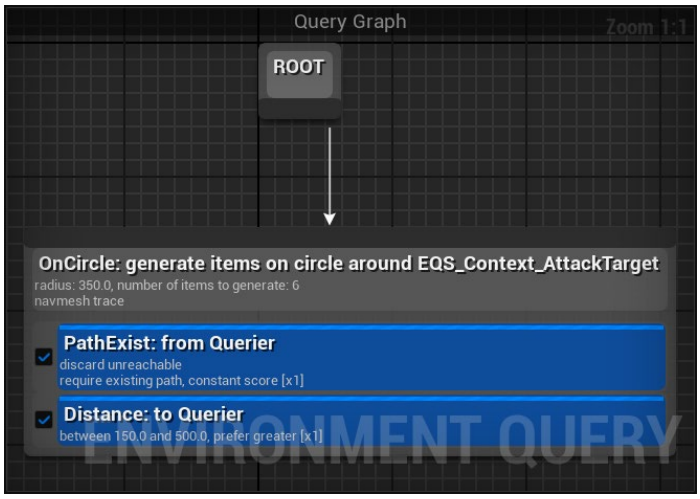
Um die Zielposition verschiedener Bewegungsformen der Enemies zu bestimmen, wird das Environment Query System (EQS) verwendet. Damit das EQS den Player als Zentrum für die Abfragen nutzen kann, ist ein Custom Env Query Context Blueprint erforderlich.

EQS_Context_AttackTarget



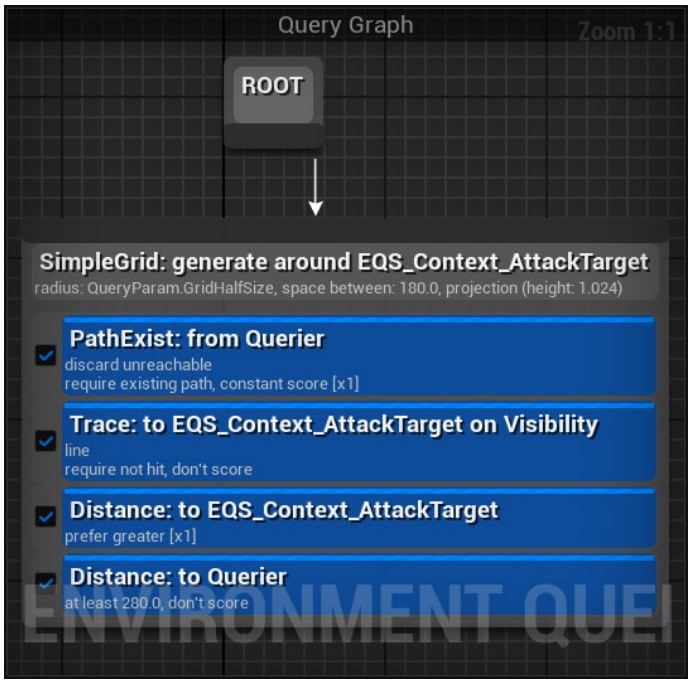
Dieser EQS Context liefert das aktuelle Angriffsziel des Akteurs, der die Abfrage durchführt. Der Ablauf beginnt mit dem Holen des AI-Controllers des Akteurs, überprüft, ob der AI-Controller vom Typ AIC_Enemy_Base ist, und holt dann das Angriffsziel aus diesem AI-Controller. Das Angriffsziel wird dann als Ergebnis der Abfrage zurückgegeben.

EQS_Strafe



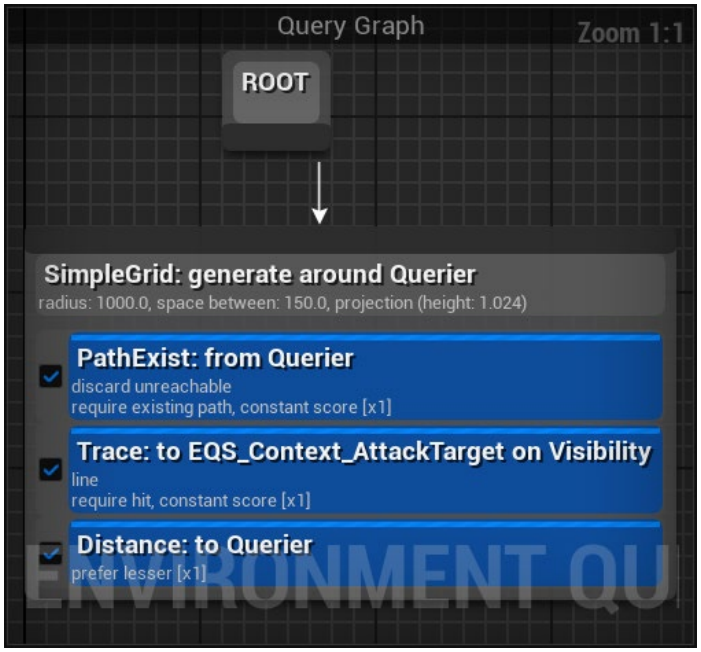
Dieses Environment Query (EQS_Strafe) generiert 6 Punkte in einem Kreis mit einem Radius von 350.0 Einheiten um das Angriffsziel (EQS_Context_AttackTarget). Die Query überprüft dann, ob ein Pfad von dem Anfrager (Querier) zu diesen Punkten existiert und verwirft diejenigen, die nicht erreichbar sind. Schließlich bewertet sie die Punkte basierend auf ihrer Entfernung zum Anfrager, wobei Entfernungen zwischen 150.0 und 500.0 Einheiten bevorzugt werden, und bevorzugt größere Entfernungen innerhalb dieses Bereichs.

EQS_FindIdealRangedLocation



Dieses Environment Query (EQS_FindIdealRangedLocation) generiert Punkte in einem Gitter um das EQS_Context_AttackTarget. Die Query überprüft dann, ob ein Pfad vom Anfrager (Querier) zu diesen Punkten existiert und verwirft diejenigen, die nicht erreichbar sind. Sie überprüft auch, ob eine Sichtlinie vom generierten Punkt zum EQS_Context_AttackTarget besteht und verwirft Punkte ohne Sichtlinie. Schließlich bewertet sie die Punkte basierend auf ihrer Entfernung zum EQS_Context_AttackTarget (bevorzugt größere Entfernungen) und zur Entfernung zum Anfrager (bevorzugt Entfernungen von mindestens 280.0 Einheiten).

EQS_FindRangedCover



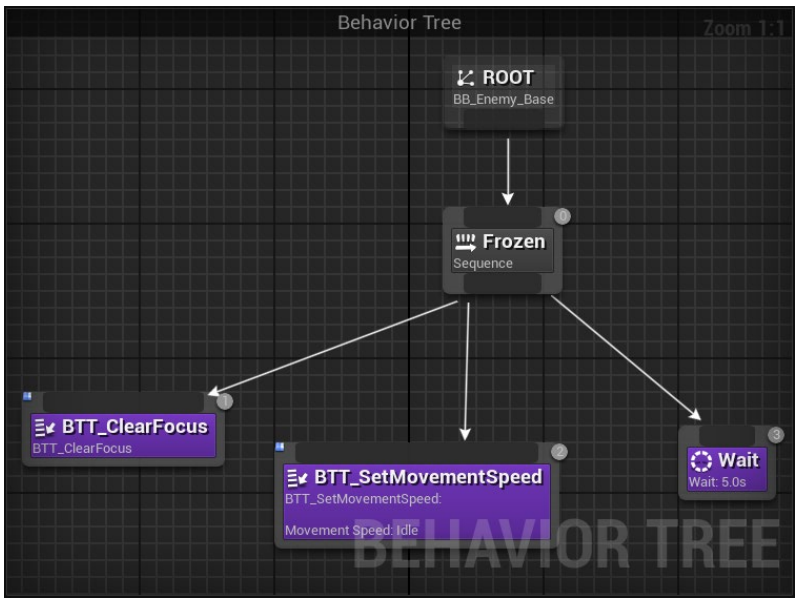
Dieses Environment Query (EQS_FindRangedCover) generiert Punkte in einem Gitter um den Querier. Die Query überprüft dann, ob ein Pfad vom Querier zu diesen

Punkten existiert und verwirft diejenigen, die nicht erreichbar sind. Sie überprüft auch, ob eine Sichtlinie vom generierten Punkt zum EQS_Context_AttackTarget besteht und vergibt Punkte, wenn diese Sichtlinie existiert. Schließlich bewertet sie die Punkte basierend auf ihrer Entfernung zum Querier (bevorzugt kleinere Entfernungen).

SubTrees

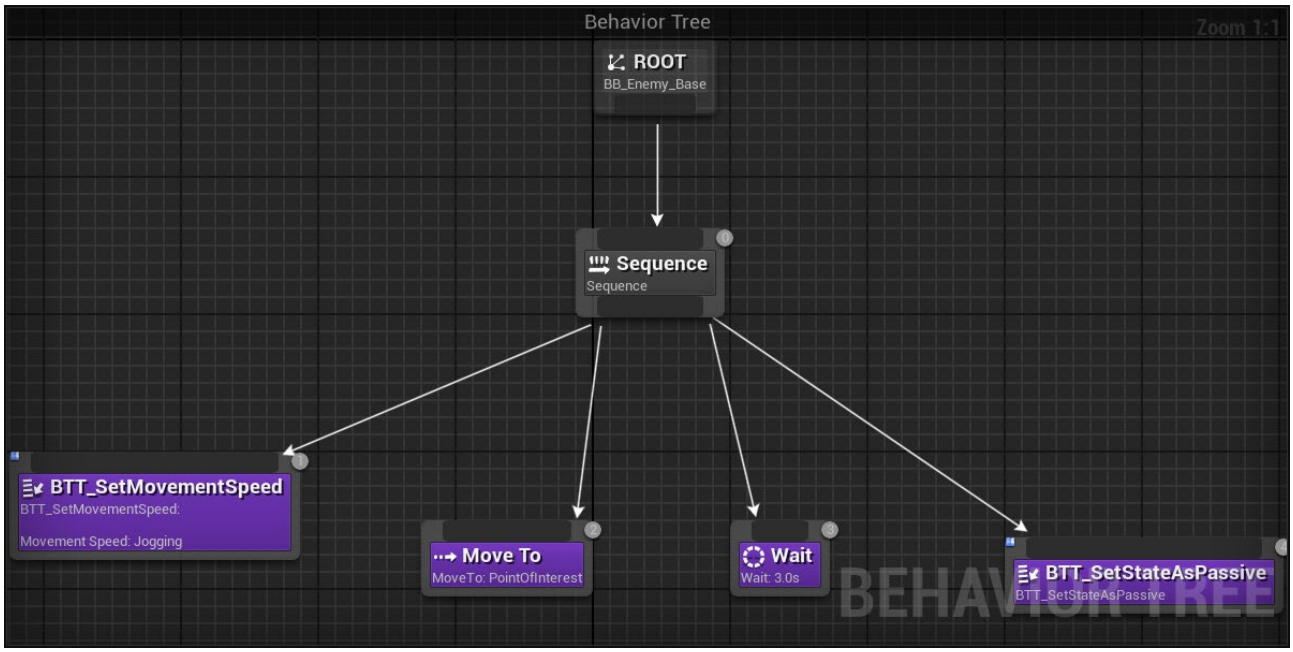
Behavior Subtrees beinhalten Abläufe, die auf eigene Behavior Trees ausgelagert wurden, da sie in mehreren Enemies identisch ablaufen. Diese Subtrees können dann einfach im jeweiligen Behavior Tree der Gegner als SubTree verwendet werden.

BT_SubTree_Frozen



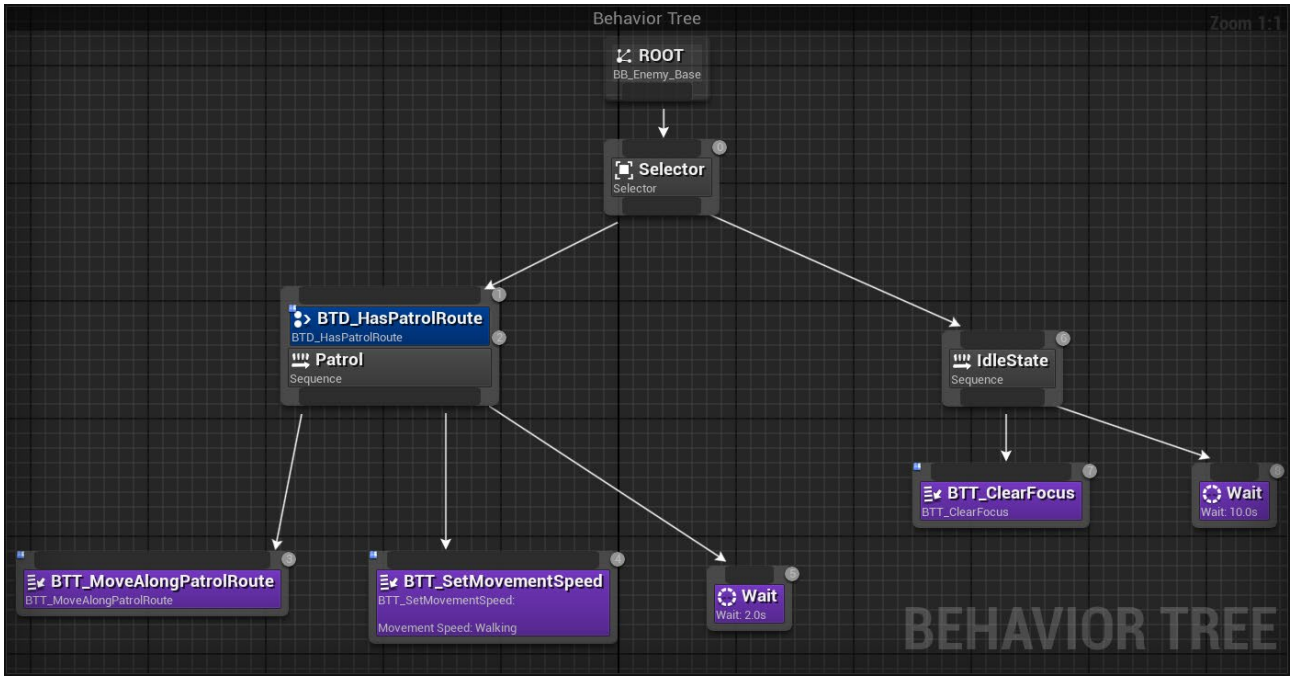
Wenn der NPC sich im "Frozen"-Zustand befindet, löscht er seinen Fokus, bewegt sich nicht und wartet 5 Sekunden lang.

BT_SubTree_Investigating



Wenn der NPC sich im "Investigating"-Zustand befindet, bewegt er sich mit einer gemäßigten Geschwindigkeit zu einem Punkt von Interesse, dort bleibt er für eine Weile und geht anschließend in einen passiven Zustand über.

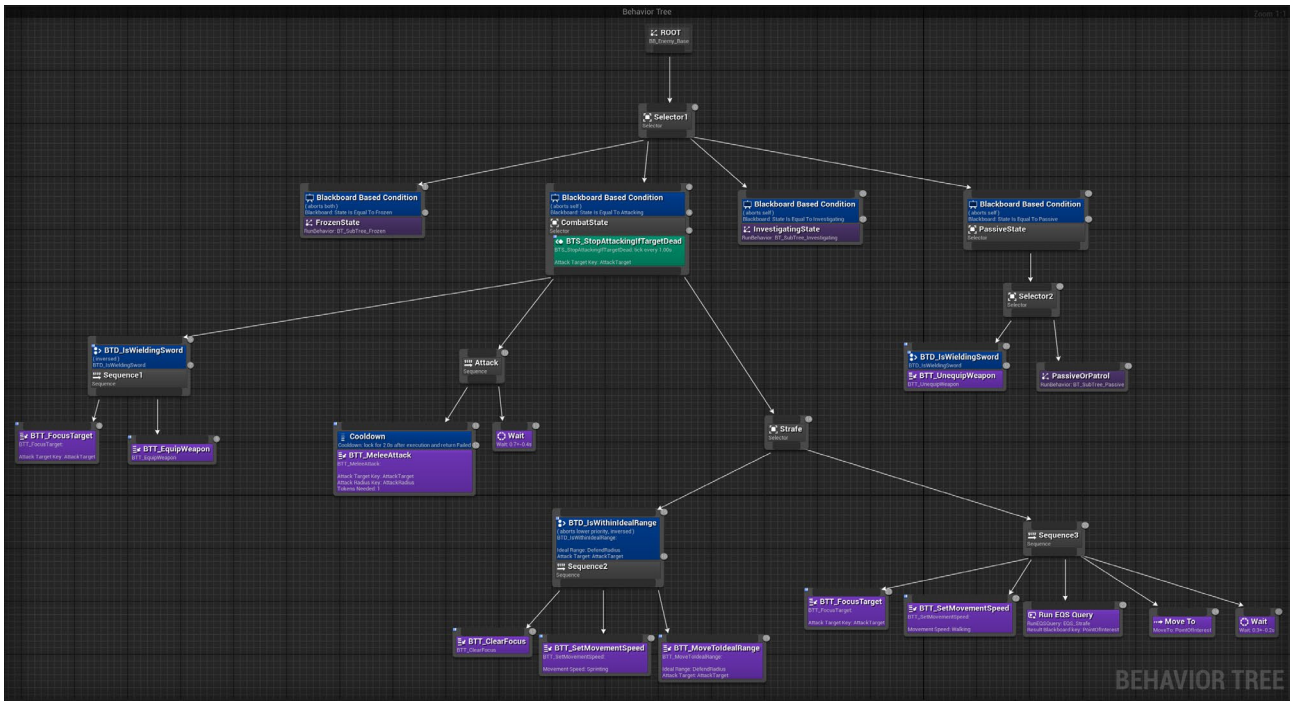
BT_SubTree_Passive



Überprüft, ob eine Patrouillenroute vorhanden ist.

- Wenn ja, folgt der Patrouillenroute, setzt die Bewegungsgeschwindigkeit auf Gehen.
- Wenn nicht, bleibt im IdleState, setzt den Fokus zurück und wartet.

BT_Enemy_Melee



Er steuert das Verhalten der Melee Enemies

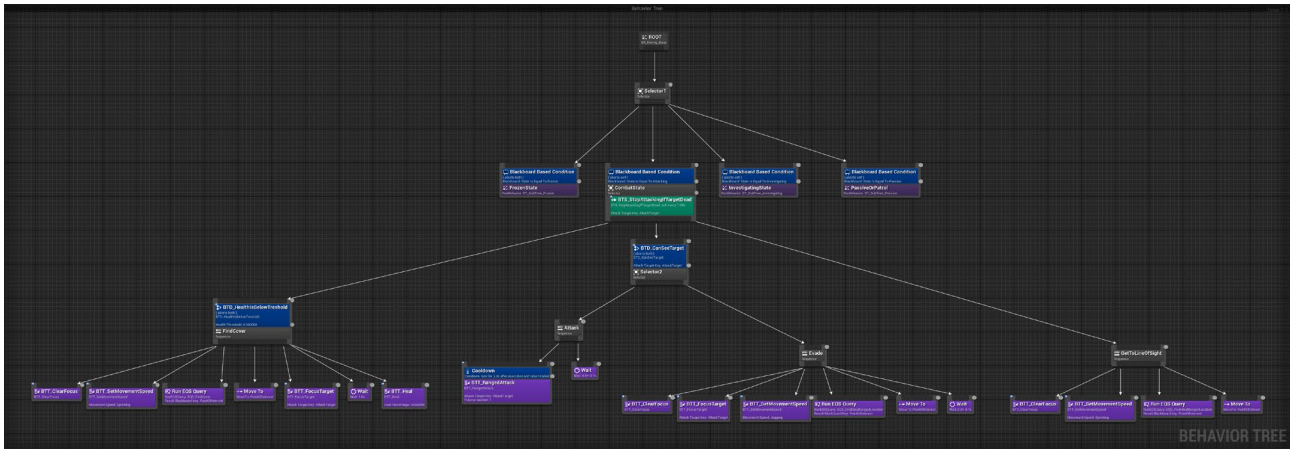
Das Verhalten im Detail:

1. **Selector1** überprüft den Zustand (State) im Blackboard.
2. Wenn der Zustand Frozen ist, führt der Baum das Sub-Tree BT_SubTree_Frozen aus.
3. Wenn der Zustand Attacking ist, wählt er den CombatState Selector.
4. Im CombatState wird überprüft, ob das Ziel (AttackTarget) tot ist (BTS_StopAttackingIfTargetDead).
5. Wenn das Ziel lebt, wird überprüft, ob der Gegner das Schwert trägt (BTD_IsWieldingSword inversed).
6. Falls nicht, fokussiert der Gegner das Ziel (BTT_FocusTarget) und rüstet das Schwert aus (BTT_EquipWeapon).
7. Anschließend greift der Gegner an (Attack Sequence):
    - Der Gegner versucht, einen Token vom Spieler zu holen.
    - Bei Erfolg führt er einen Nahkampfangriff durch (BTT_MeleeAttack).
    - Nach dem Angriff gibt es eine Cooldown-Phase (Cooldown).
    - Danach wartet der Gegner kurz (Wait).
8. Falls der Gegner nicht in der idealen Reichweite ist (BTD_IsWithinIdealRange inversed), bewegt er sich in die ideale Reichweite (Sequence2):
    - Der Fokus wird zurückgesetzt (BTT_ClearFocus).

- Die Bewegungsgeschwindigkeit wird auf Sprinten gesetzt (BTT_SetMovementSpeed).
- Der Gegner bewegt sich in die ideale Reichweite (BTT_MoveToIdealRange).

9. Wenn der Gegner in der Reichweite ist (Sequence3):
   - Er fokussiert erneut das Ziel (BTT_FocusTarget).
   - Die Bewegungsgeschwindigkeit wird auf Gehen gesetzt (BTT_SetMovementSpeed).
   - Eine EQS-Abfrage wird durchgeführt (Run EQS Query), um eine geeignete Position zu finden.
   - Der Gegner bewegt sich zur gefundenen Position (Move To) und wartet kurz (Wait).
10. Wenn der Zustand Investigating ist, führt der Baum das Sub-Tree BT_SubTree_Investigating aus.
11. Wenn der Zustand Passive ist, wählt der Selector den PassiveState.
12. Im PassiveState Selector wird überprüft, ob der Gegner das Schwert trägt (BTD_IsWieldingSword):
    - Falls ja, legt er es ab (BTT_UnequipWeapon).
13. Anschließend führt der Baum das Sub-Tree BT_SubTree_Passive aus, wo der Gegner entweder patrouilliert oder eine passive Aktion durchführt.

BT_Enemy_Ranged



Er steuert das Verhalten der Ranged Enemies

Das Verhalten im Detail:

1. Der Selector1 überprüft zuerst, ob der State auf Frozen gesetzt ist. Wenn ja, wird der BT_SubTree_Frozen ausgeführt, welcher den Fokus löscht, die Bewegungsgeschwindigkeit auf Idle setzt und 5 Sekunden wartet.
2. Ist der State auf Attacking gesetzt, wird der CombatState-Zweig ausgeführt.

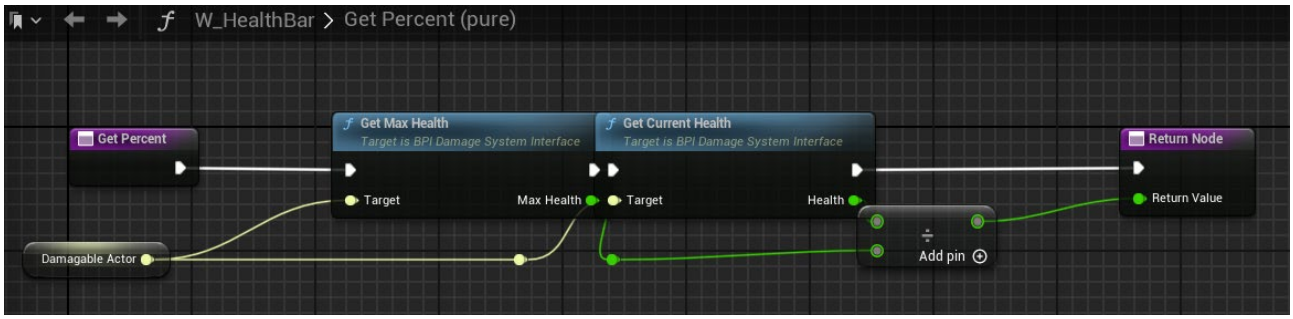
3. Der BTS_StopAttackingIfTargetDead Service überprüft jede Sekunde, ob das Angriffsziel tot ist.
4. Wenn die Gesundheit unter 30 % fällt (BTD_HealthIsBelowTreshold), wird die FindCover-Sequenz ausgeführt, die den Fokus löscht, die Bewegungsgeschwindigkeit auf Sprinting setzt, eine EQS-Abfrage (EQS_FindCover) ausführt und sich zur gefundenen Deckung bewegt. Danach fokussiert der Feind das Angriffsziel und wartet 1 Sekunde, bevor er 5 % seiner Gesundheit regeneriert (BTT_Heal).
5. Wenn der Feind das Ziel sehen kann (BTD_CanSeeTarget), wird der Selector2 ausgeführt.
6. Wenn der Feind angreifen kann, wird die Attack-Sequenz ausgeführt, die eine Abklingzeit von 2 Sekunden hat (Cooldown), einen Fernkampfangriff (BTT_RangedAttack) ausführt und 0.3 Sekunden wartet.
7. Wenn der Feind nicht angreifen kann, wird die Evade-Sequenz ausgeführt, die den Fokus löscht, das Angriffsziel fokussiert, die Bewegungsgeschwindigkeit auf Jogging setzt, eine EQS-Abfrage (EQS_FindIdealRangedLocation) ausführt und sich zur gefundenen Position bewegt.
8. Wenn der Feind über dem Treshold Leben besitzt und den Player nicht mehr direkt sehen kann , führt er die GetToLineOfSight-Sequenz aus, die den Fokus löscht, die Bewegungsgeschwindigkeit auf Sprinting setzt, eine EQS-Abfrage (EQS_FindIdealRangedLocation) ausführt und sich zur gefundenen Position bewegt.

BT_Enemy_Base

Dabei handelt es sich lediglich um einen ungenutzten Behavior Tree, der als Ausgangslage für neue Enemies dienen soll. Da er keine neue Logik enthält, wird er hier nicht weiter behandelt.

Widgets

W_HealthBar



Ein simples Widget das auf dem BP_Enemy liegt. Enthält ein Canvas Panel mit einer Progress Bar und eine GetPercent Funktion die das aktuelle Leben als Anteil von 1, also Prozent ausrechnet. Die Progress Bar nutzt diese Funktion, um sich zu aktualisieren.

Effekte

Weapon Trails

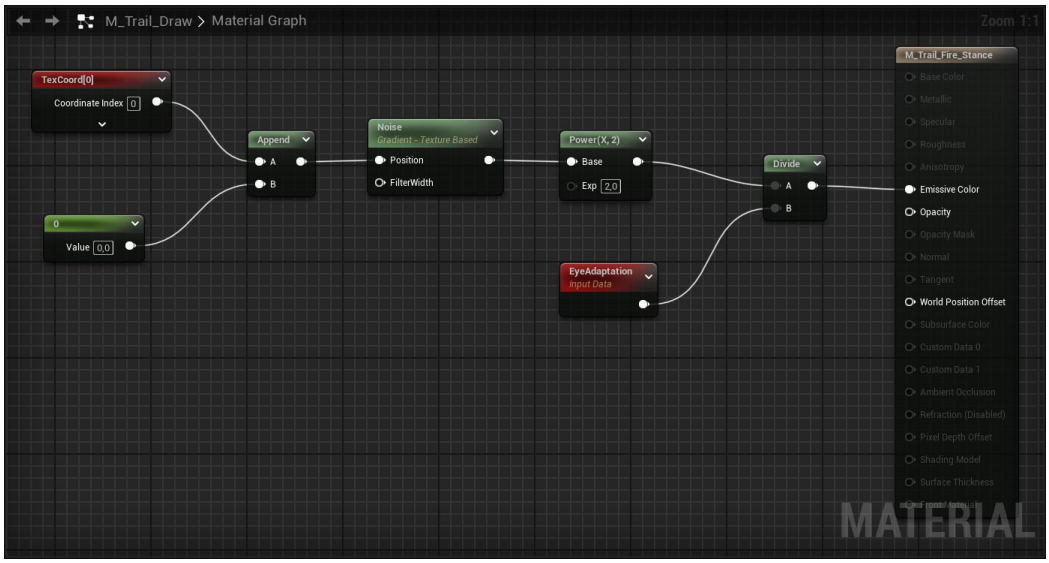
Weapon Trails sind Element der meisten Action Combat Games, in denen es Nahkampfwaffen gibt. In diesem Projekt sollen sie die visuelle Repräsentation des aktiven Elementes auf der Waffe Übernehmen.

Die Weapon Trail sollen einerseits das Bild visuell brechen, also eine Refraction erzeugen, auf der anderen Seite aber auch leuchtende schlieren in passenden Farben ziehen.

Den Anfang der Weapon Trails macht der Fire Trail der zuerst erstellt wurde. Dieser wurde letztendlich dupliziert und angepasst, um die Weapon Trails der anderen Elemente darzustellen.

Bei der Erstellung wurde die Noise Texture bewusst innerhalb der Engine erstellt und nicht mit einem externen Tool, um die Möglichkeiten der Engine auszutesten.

M_Trail_Draw

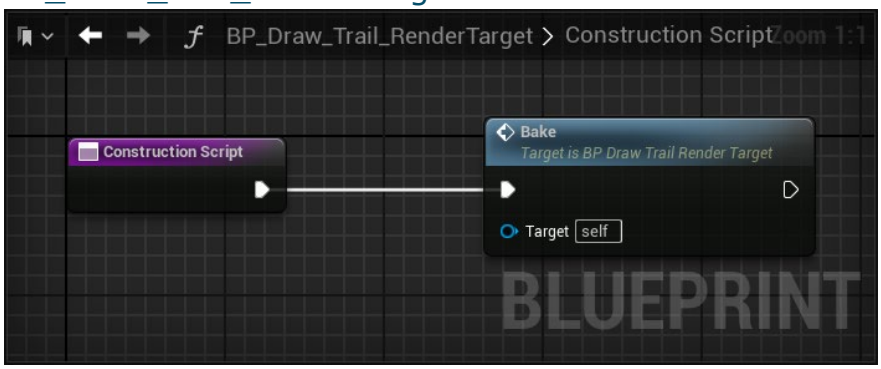


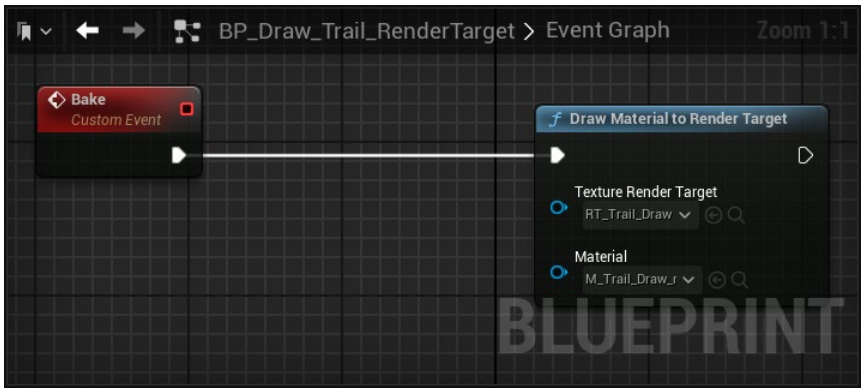
Zuerst wurde ein neues Material erstellt.

Dieses Material generiert ein Rauschmuster, das quadriert und dann normalisiert wird, bevor es als Emissionsfarbe verwendet wird. Die Emissionsfarbe wird durch die Augenanpassung beeinflusst, was bedeutet, dass die Helligkeit des Materials in verschiedenen Beleuchtungsszenarien variiert. Wenn die Szene sehr hell ist, wird die Helligkeit des Materials reduziert, und wenn die Szene dunkel ist, wird die Helligkeit des Materials erhöht.

Danach wurde ein RenderTarget RT_Trail_Draw erstellt.

BP_Draw_Trail_RenderTarget



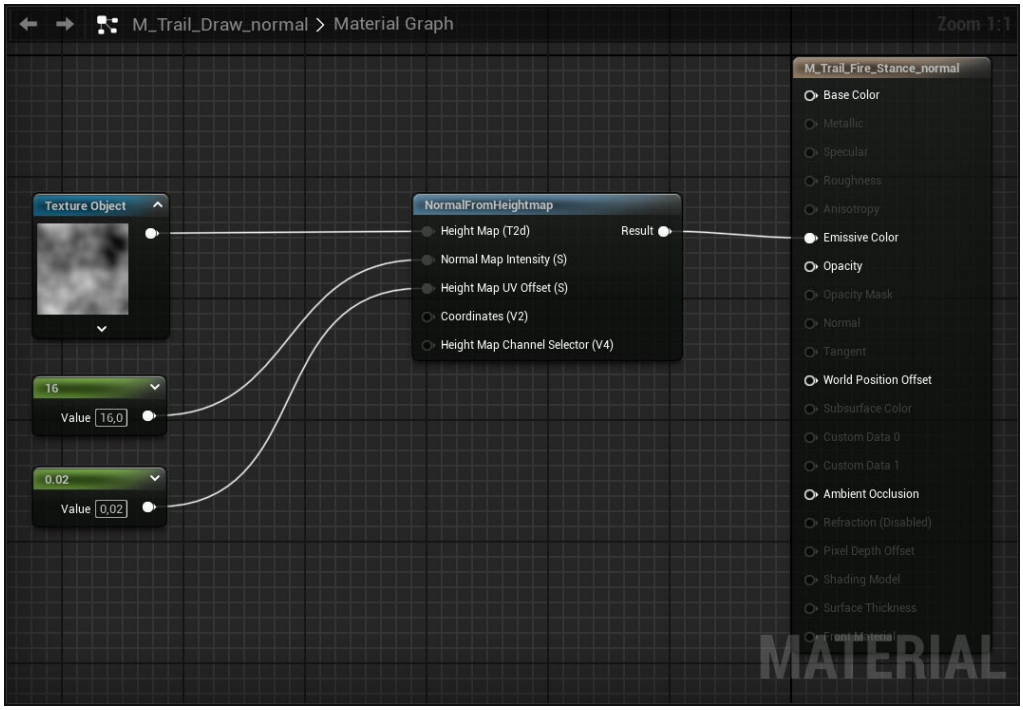


Um das Material auf das RenderTarget zu zeichnen, wird der BP_Draw_Trail_RenderTarget Blueprint erstellt. Er hat die Aufgabe, ein Material auf ein Render-Target zu zeichnen und dieses Render-Target aktuell zu halten.

Mit ihm kann man Texturen aus dem Render Target erzeugen, hierzu tauscht man einfach das Material in der Node und erzeugt dann aus dem Render Target eine Static Texture.

Aus dem Render Target RT_Trail_Draw wird eine Static Texture RT_Trail_Draw_Tex erzeugt. Auf diese Weise bekommt man eine Art Noise Texture.

M_Trail_Draw_normal



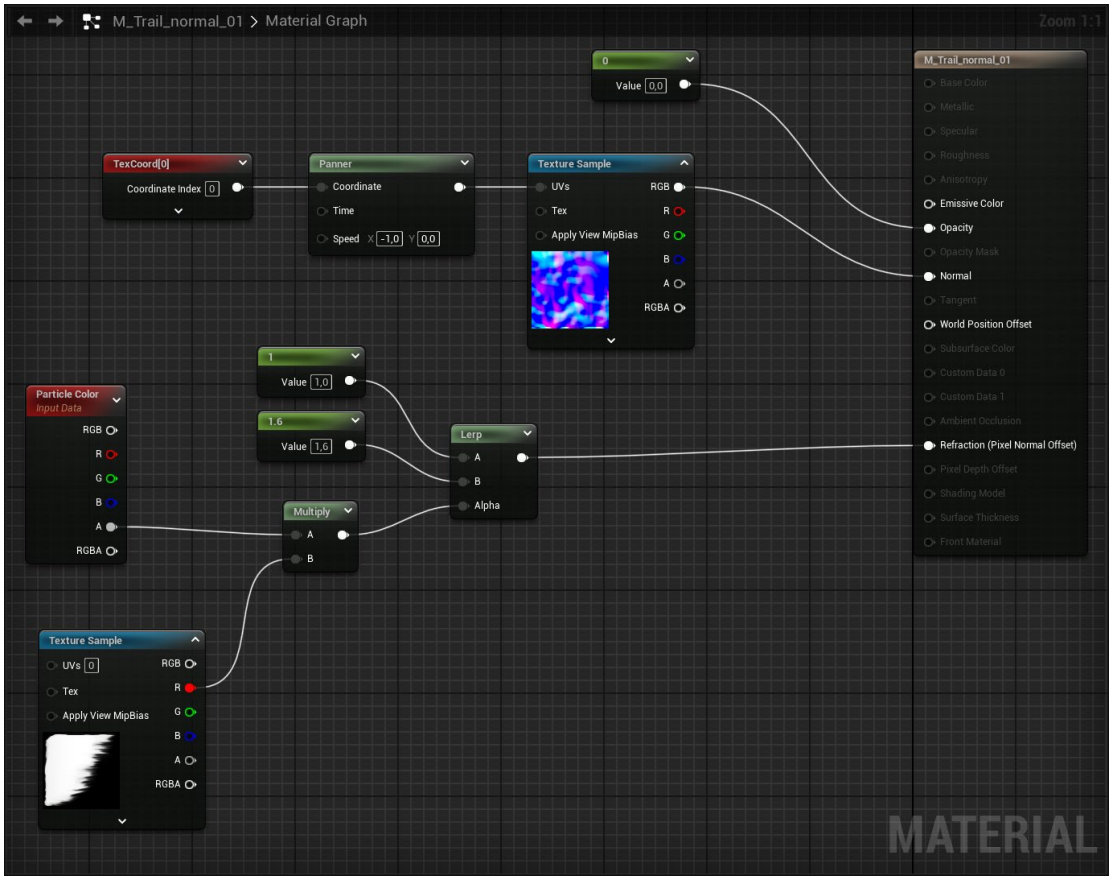
Mit der neuen Texture wird ein neues Material erstellt.

Die erzeugte Normalmap, die normalerweise Licht und Schatten beeinflusst, bewirkt subtile Änderungen im Aussehen des Materials basierend auf der Lichtquelle und dem Betrachtungswinkel und erzeugt so einen Refraction-Effekt. Mithilfe des BP_Draw_Trail_RenderTarget wird, wie zuvor, eine neue Textur aus dem M_Trail_Draw_normal erstellt. Damit sind die beiden Texturen, die für die Erstellung des Weapon Trails benötigt werden, erzeugt.

TrailTexture

Für die Verlaufsform wurde eine TrailTexture in Photoshop erstellt und in Unreal Importiert.

M_Trail_normal_01

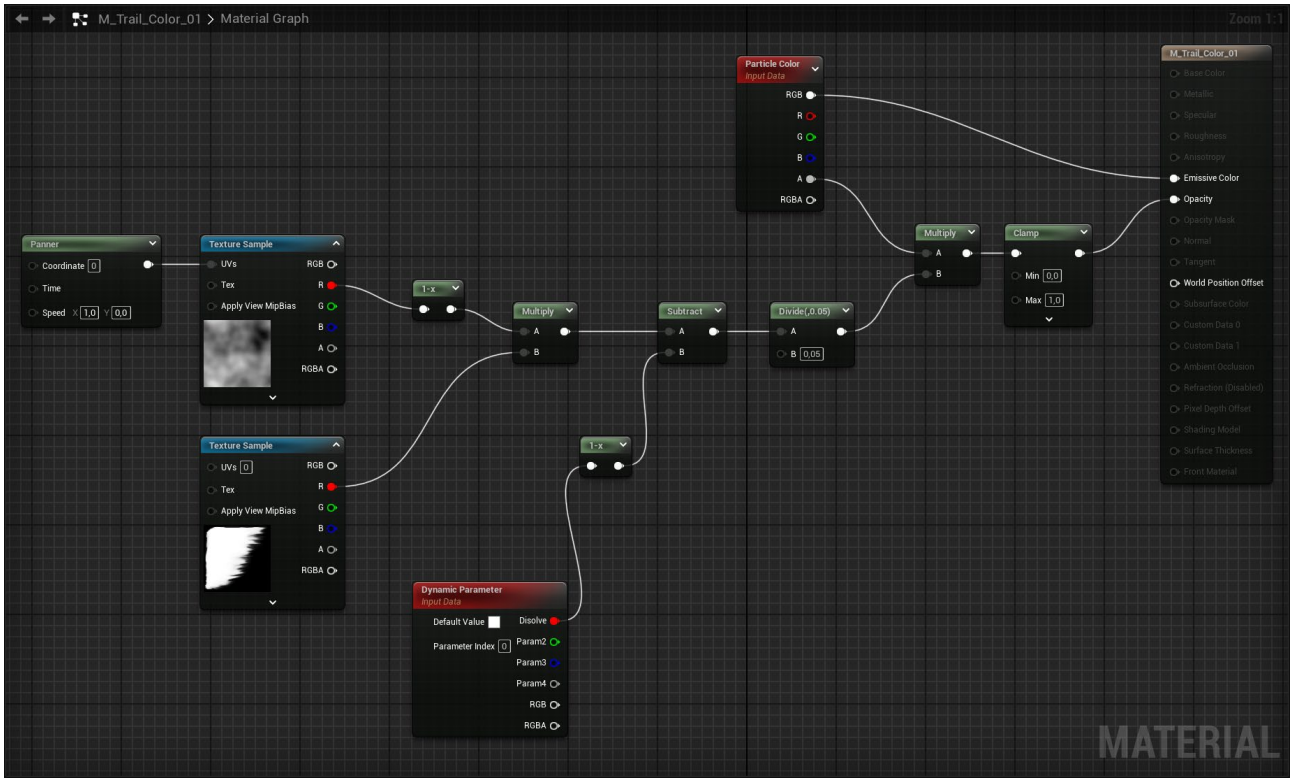


Wird benötigt, um mithilfe des Niagara Systems den Refractionsteil des Weapon Trails zu erzeugen.

1. Der Particle Color Node liefert die aktuelle Farbe und den Alpha-Wert des Partikelsystems.
2. Ein Texture Sample Node mit der Trail-Shape-Textur wird verwendet, um die Form des Trails zu definieren.
3. Der Alpha-Wert der Trail-Shape-Textur wird mit der Partikelfarbe multipliziert.
4. Der Lerp Node interpoliert zwischen den Werten 1.0 und 1.6 basierend auf dem Ergebnis der Multiplikation.
5. Das interpolierte Ergebnis wird an den Refraction (Pixel Normal Offset) Eingang des Materials angeschlossen, um den Brechungseffekt zu erzeugen.

Zusammengefasst nutzt das Material M_Trail_normal_01 die Partikelsystemdaten und eine Normalmap zusammen mit einer Shape-Textur, um dynamische Brechungseffekte für den Weapon Trail zu erzeugen. Die Bewegung der Textur und die Brechung basieren auf den durch das Niagara-System bereitgestellten Partikeldaten.

M_Trail_Color_01



Wird benötigt, um mithilfe des Niagara Systems den farbigen Teil des Weapon Trails zu erzeugen.

1. **Texturkoordinaten und Panning**:
    - Der TexCoord[0] Node liefert die Texturkoordinaten der zu rendernden Fläche.
    - Der Panner Node verschiebt diese Texturkoordinaten in der X-Richtung mit einer Geschwindigkeit von -1, um eine kontinuierliche Bewegung der Textur zu simulieren.
2. **Erster Texture Sample**:
    - Der erste Texture Sample Node liest die Normalmap-Textur, die durch den Panner Node bewegt wird. Diese Normalmap wird verwendet, um Details wie Wellen oder Muster in den Trail einzufügen.
3. **Zweiter Texture Sample (Shape)**:
    - Ein zweiter Texture Sample Node lädt eine Shape-Textur, die die Form des Trails definiert. Diese Textur gibt die Grundform des Trails vor, die durch das Panning und die Normalmap bewegt und verzerrt wird.
4. **Dynamic Parameter**:
    - Der Dynamic Parameter Node erhält zur Laufzeit dynamische Werte, die das Aussehen des Materials steuern. In diesem Fall wird der Dissolve Parameter verwendet, um Effekte wie das Auflösen des Trails zu steuern.
5. **Mathematische Operationen**:

- Mehrere mathematische Operationen kombinieren die Werte der geladenen Texturen und des Dynamic Parameters:
    - **Multiply**: Multipliziert die Ergebnisse der vorherigen Operationen, um die Intensität zu steuern.
    - **Subtract**: Subtrahiert bestimmte Werte, um Kontraste und Übergänge zu erzeugen.
    - **Divide**: Teilt die Ergebnisse durch einen festen Wert (0.05), um die Helligkeit und Intensität des Effekts anzupassen.
    - **Clamp**: Beschränkt die Werte auf einen Bereich von 0 bis 1, um sicherzustellen, dass die Endwerte gültige Farb- und Transparenzwerte sind.

6. **Emissive Color**:
    - Der Particle Color Node liefert die aktuelle Farbe der Partikel, die durch das Niagara-System bestimmt wird.
    - Diese Partikelfarbe wird mit den Ergebnissen der vorherigen mathematischen Operationen multipliziert, um die endgültige Emissionsfarbe zu berechnen.
    - Das Ergebnis dieser Berechnung wird an den Emissive Color Eingang des Materials angeschlossen. Dadurch wird der farbige Effekt des Trails erzeugt.

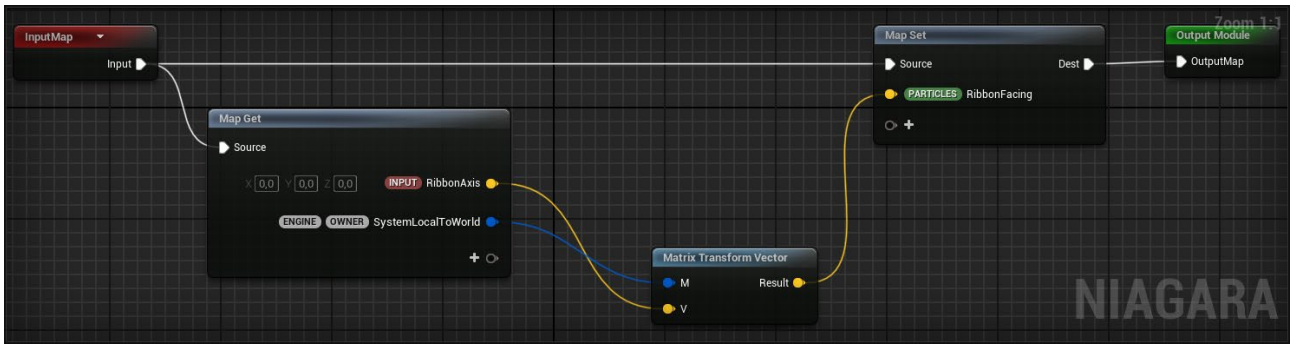
Zusammengefasst: Der Prozess beginnt mit der Bewegung der Texturkoordinaten durch den Panner Node, was zu einer dynamischen Bewegung der Normalmap führt. Diese Normalmap wird mit einer Shape-Textur kombiniert, um die Form des Trails zu definieren. Dynamische Parameter und Partikelfarben werden verwendet, um das Aussehen des Trails während der Laufzeit zu steuern. Verschiedene mathematische Operationen kombinieren diese Elemente, um die finale Emissionsfarbe des Materials zu berechnen, die dann den farbigen Weapon Trail darstellt.

Niagara System

Da sich die Weapon Trails im Aufbau sehr ähnlich sind wird hier einmal am Beispiel des Fire Weapon Trails erklärt, wie der Effekt erstellt wird.

NS_Trail_Fire

- Als Basis wird das Niaga System Fountain genutzt.
- Zuerst wird der Refraction Trail mit der Fountain Node erstellt
- Shape Location, Add Velocity, Gravity Force, Drag und Scale Forces and Velocity werden geloescht da sie nicht benoetigt werden.
- NS_Trail_Fire Node wird das Loop Behavior auf Once umgestellt sowie die Loop Duration auf 1. Die Dauer des Effektes wird über den Anim Graph innerhalb der Animation Montages gesteuert.
- Der Sprite Renderer in der Fountain Node wird durch einen Ribbon Renderer ersetzt.
- Als Material des Ribbon Renderers wird der M_Trail_normal_01 eingesetzt
- Im Initialize Particle wird der Ribbon Width Mode auf DirectSt umgestellt, um die Breite des Ribbon Effektes setzen zu können. In diesem Fall 140.
- Der Life Cycle Mode wird auf System umgestellt.
- Adden eines Scratch Pad Modules



Das Scratch Pad Module transformiert den RibbonAxis Vektor von lokalen in Weltkoordinaten und setzt diesen Wert als RibbonFacing der Partikelbänder im System. Das stellt sicher, dass die Ausrichtung der Partikelbänder korrekt in Bezug auf die Weltkoordinaten ist und somit korrekt gerendert werden.

- Der Facing Mode wird auf Custom Side Vector umgestellt
- Eine Ribbon Link Order wird hinzugefuegt und mit der Order One Minus Float und der Float NormalizedLoopAge versehen. Das sorgt dafür, dass die Bänder von den älteren zu den jüngeren Partikeln verlaufen.
- Anpassung des Width Segment Counts auf 3 anstatt eins um die Höhenunterschiede in der Refraction zu verstaerken (Tessellation Factor)
- Anpassen der Scale Color um den Verlauf der Farbe während der Erzeugung zu steuern.

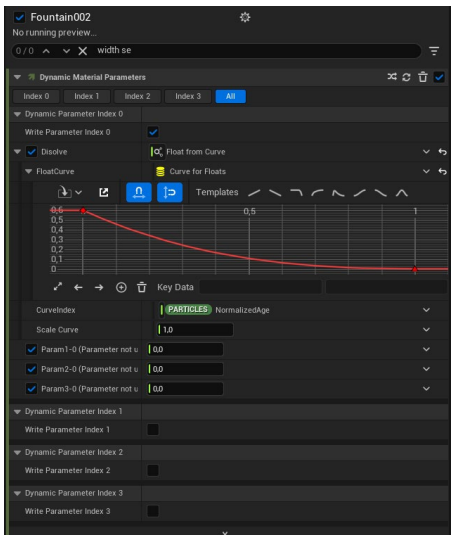
Mittlere Lebensdauer:

- Während der mittleren Lebensdauer der Partikel wird die Alpha-Skalierung basierend auf der Kurve angepasst. Dies bedeutet, dass die Partikel im Verlauf ihrer Lebensdauer allmählich transparenter werden.
- Gleichzeitig erhöht die untere Kurve die Farbintensität der Partikel, sodass sie heller erscheinen.

Ende der Lebensdauer:

- Gegen Ende der Lebensdauer der Partikel sinkt die Farbintensität wieder, und die Partikel werden noch transparenter, bis sie schließlich ganz verschwinden.

- Um die Flammen darzustellen, wird die Node des Refraction Trails kopiert und das Material des Ribbon Renderers durch das M_Trail_Color_01 ersetzt.
- Hinzufuegren eines Dynamic Parameters der das Dissolven innerhalb des Materials steuert, das führt dazu das der Trail Mithilfe der Kurve Dissolved.



- Des Weiteren werden mehrere kleinere Anpassungen an den Werten vorgenommen, die vorher schon bei dem Refraction Trail verwendet worden sind.
- Diese Node wird dann zweimal kopiert und in Farbe und anderen Werten angepasst, um einen Vielschichtigen Effekt zu erzeugen der mit Flammen assoziiert werden kann.

Restliche NS_Trail Effekte

Die weiteren Weapon Trail Effekte (NS_Trail_Air, NS_Trail_Earth, NS_Trail_Water, NS_Trail_GuardRevenge, NS_Trail_HeavyAttacks, NS_Trail_Neutral, NS_Trail_InAirAttacks) basieren alle auf derselben Basis wie der NS_Trail_Fire und werden nur in Farbe und anderen Werten angepasst oder ergänzt um zu dem jeweiligen Angrifftyps zu passen.

Implementierung

Zur Implementierung der Effekte werden diese einfach innerhalb der jeweiligen Anim Montages als Timed Niagara Effekt mit dem entsprechenden Niagara System gesetzt, in dem sie Ablaufen sollen. Um die Position zu bestimmen, wird ein neuer socket FX_Weapon_Base Socket im Skeletal Mesh mit der Position in der Mitte der Klinge der Waffe erstellt. Dieser wird im Timed Niagara Effekt definiert.

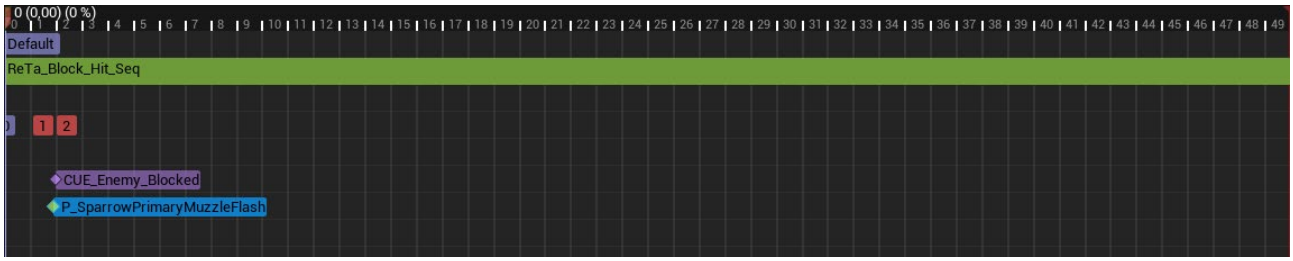
Nicht Verwendete

Während des Projektes wurde auch das Niagara Fluid System ausprobiert, um einen Weapon Trail Effekt und einen Raucheffekt für den Dodge des Players zu erstellen. Die Effekte funktionierten für sich genommen gut, jedoch funktionierte das Attachen der Effekte an den Player noch nicht wie gewünscht. Daher wurden diese Effekte nicht in das Projekt aufgenommen. Dabei handelt es sich um NS_Smoke und NS_CharacterSmokes.

Weitere Effekte

Die anderen verwendeten Partikeleffekte wurden nicht selbst erstellt. Sie wurden entweder direkt innerhalb eines Blueprints implementiert, wie zum Beispiel der Hit-

Effekt, oder innerhalb der AnimMontage mithilfe eines Anim Notify Play Particle Effekts oder Play Niagara Particle Effekts.



Dabei können Anpassungen wie Position, Rotation und Skalierung des Effekts vorgenommen werden. Eigentlich war geplant, alle Effekte innerhalb der Animation zu steuern. Aufgrund der Vielzahl der anzupassenden Effekte fehlte jedoch die Zeit, dies vollständig umzusetzen.

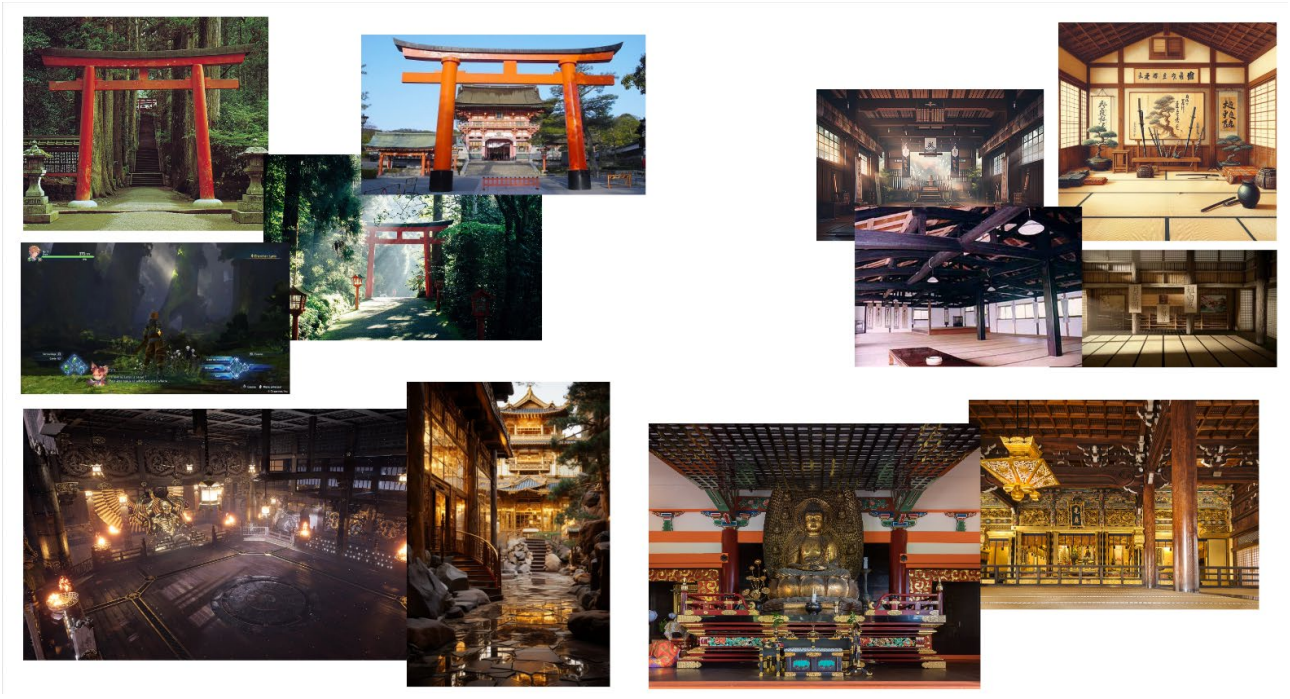
Scene Building

Das Grundkonzept sieht die Bindestellen der Dimensionen als Level vor, was eine große Freiheit in der Levelgestaltung ermöglicht. Das Projekt sollte zwei Level umfassen, die im Stil des feudalen Japan gestaltet sind: ein Level, in dem sich der Spieler durch ein Gebiet bewegt, in dem mehrere Gegner patrouillieren, und ein Level für den Bosskampf.

Zur Erstellung der Scenes wurden nur die Tools innerhalb der Unreal Engine verwendet.

BossLevel

Zunächst wurde das Bosslevel erstellt. Dieses Level sollte ein kleines Startareal haben, von dem aus Treppen zu einem Dojo führen. Zur Orientierung wurde ein MoodBoard erstellt.



Zum Bau des Levels wurde das Asset Pack "Fighting Stage" von Laertes verwendet. Die Assets daraus wurden genutzt, um eine eigene Fighting Stage zu erstellen. Zur Erleichterung des Platzierens der Assets kam das Plugin "Ultimate Level Art Tool" von Laerte zum Einsatz.

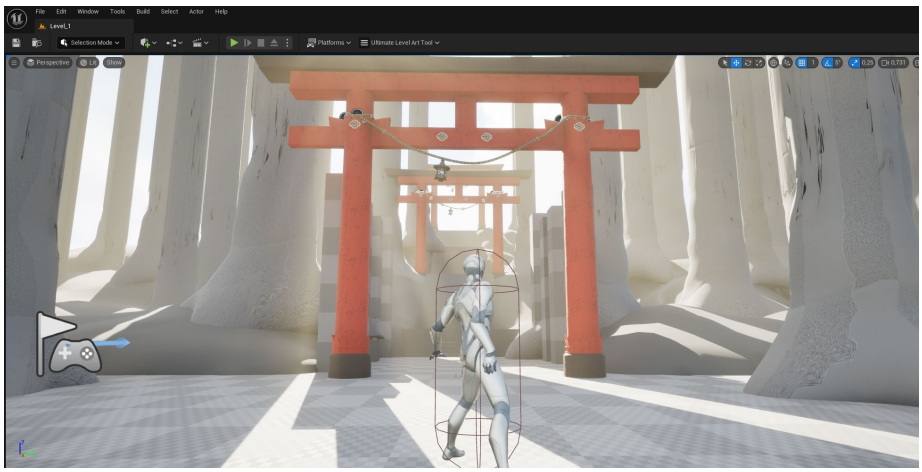
Bevor mit den Dojo-Assets und dem Gebäude selbst begonnen wurde, musste die Umgebung, in der das Dojo steht, gestaltet werden. Der Plan sah vor, dass der Spieler an einem Punkt startet und sich dann zum Dojo und dem dortigen Encounter bewegt.

Als Vorbild für die Umgebung diente eine typisch japanische Landschaft mit Toriis, den großen roten Holztoren, die oft zu Tempeln führen. Da das Dojo durch die verwendeten Assets einen buddhistischen Tempelflair erhält, passte diese Umgebung sehr gut und schuf eine passende Atmosphäre für den Weg zu einem Kampf.

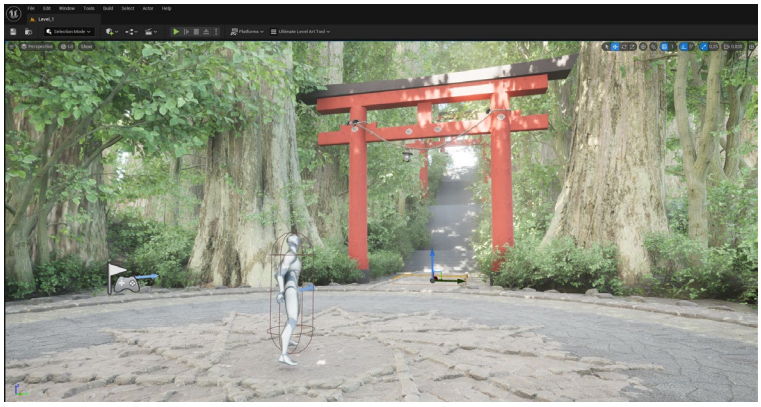
Geplant war die Erstellung eines kleinen Übungsplatzes, auf dem ein Dummy zum Testen bereitsteht. Von dort führt eine Treppe durch den Wald und Toriis zum Dojo, wo der Gegner wartet. Zur Gestaltung der Landschaft wurden Mega Scans Assets verwendet.

Zunächst wurde die Szene mithilfe des Modelling Tools von Unreal Engine grob blockiert. Dabei wurde ein Tor als geladenes Asset genutzt, und die Szene wurde modelliert und skulptiert, beginnend mit dem Treppenaufgang, der zum Dojo führen sollte.

Zum Ausprobieren der Szenenkomposition wurden in der Engine große Baumstümpfe skulptiert. Dabei wurde auch mit dem Lighting und dem Exponential Height Fog experimentiert, um ein passendes Ambiente für die Szene zu schaffen.



Bäume und Texturen für die Landschaft werden aus dem Quixel Content bezogen. Ziel ist es, ein ansprechendes Startareal zu schaffen, von dem aus eine Treppe durch die Tore zum Dojo führt. Für zusätzliche Foliage wird ein Megascans Asset Pack für Wald-Foliage verwendet. Diese Assets werden im Level verteilt, um eine realistischere Atmosphäre zu erzeugen.



Um die Baumkronen und Baumstümpfe zu verdecken, werden Bäume aus demselben Asset Pack verwendet, an den Spitzen der Baumstümpfe platziert und entsprechend skaliert. So wird sichergestellt, dass selbst wenn der Player nach oben schaut, ein gutes Bild entsteht. Es war wichtig, die großen Baumstümpfe zu verwenden, da sie einen mystischen Flair hinzufügen, den normale Bäume nicht bieten könnten. Das Platzieren des Foliages war insgesamt recht aufwendig, da fast alles einzeln gesetzt werden musste, um ein besseres Szenenbild zu erzielen. Die Szene war zu klein, um das Paint Tool effektiv zu nutzen. Als nächstes mussten die Platzhalter-Stufen durch passende Assets ersetzt werden.

Nach dem Austauschen der Treppen wirkte die Szene immer noch etwas roh. Daher wurden verschiedene Nature Props aus Megascans genutzt, um die Szene realistischer zu gestalten. Dies nahm viel Zeit in Anspruch, da man genau auf die Komposition achten musste, die der Player während des Spielens sieht.

Post-Processing: Nachdem die Landschaftsszene zufriedenstellend war, wurde das Post-Processing vorgenommen, um die Atmosphäre der Landschaft auch im Dojo einzufangen. Hier werden mehrere Color Filter, Exposure und Bloom angepasst um ein Atmosphärisches Ergebnis zu erzielen.

Lighting: Das Lighting der Szene musste angepasst werden, und die Schreinlaternen erhielten Lichtquellen, was zur Atmosphäre beitrug und ein besseres Kontrastverhältnis im Lighting ermöglichte.

Während der Arbeit an der Landschaft wurden alle Texturen in Nanite oder maximaler Auflösung aus MegaScans geholt. Dies führte jedoch zu erheblichen Performance-Einbußen und musste behoben werden. Normalerweise würde das durch Filtern im Content Browser und Bulk Edit geschehen, jedoch führte ein Bug in der verwendeten Unreal Engine 5 Version dazu, dass die Engine beim Versuch, die maximale Texture Size im Bulk Edit zu ändern, einfriert und abstürzt. Nachfolgend mussten teilweise kaputte Texturen neu importiert werden. Alle Texturen wurden auf 2048 gestellt, was immer noch mehr als ausreichend ist. Auch die Lichtquellen wurden optimiert, indem der Optimization View für Light Complexity genutzt wurde und weitere Optimierungen in anderen Views vorgenommen wurden. Im Quad Overdraw Viewmode wurde festgestellt, dass vor allem die Bäume in der Szene auf Basis ihrer Polygonzahl zu viele Ressourcen verbrauchen. Diese wurden mit dem Mesh Simplify Tool aus dem Modelling Tool der Unreal Engine vereinfacht.

Der Radius der Lichter wurde zugunsten der Performance reduziert, um teure Overlaps zu vermeiden, und die Quads wurden so weit wie möglich reduziert. Die Performance der Szene war danach zufriedenstellend allerdings immer noch nicht optimal.

Da die meisten Assets ohne Collider importiert wurden, musste die Collision hinzugefügt werden. Bei manchen Assets lohnt es sich, die Collision über das Modelling Tool zu erstellen, da Mesh to Collision mehr Feinabstimmungsmöglichkeiten bietet. Bei anderen ist das Hinzufügen der Collision in den Optionen des Static Meshes der beste Weg, je nach Detailgrad und Anpassungsbedarf an das Mesh. Die Treppen sollten präzise sein, während der Boden möglichst flach sein muss, um die Kamera stabil zu halten und optische Fehler bei der Fußstellung zu vermeiden.

Leider trat auch hier wieder ein Bug in Unreal Engine v5.3 auf, bei dem die Engine beim Anpassen der Collision wiederholt abstürzte.

Ursprünglich war geplant, das Dojo selbst zu bauen. Dabei traten jedoch zwei Probleme auf: Die verfügbaren Assets waren so spezifisch, dass sie kaum für den Bau von etwas Neuem genutzt werden konnten und architektonisch dem Original aus dem Asset Pack zu ähnlich sahen. Zudem wurde die Zeit knapp, und die AI des Gegners musste noch fertiggestellt werden. Nach einer Woche des Zeitverlusts beim Bau des Dojos wurde beschlossen, einen Teil der Assets zu nutzen, um einen kleinen Boss Raum zu erstellen der möglichst performant ist um die Performance zumindest in dem Raum, in dem der Kampf stattfindet zu stabilisieren.

Endergebnis


Perspective
Lit
Show


Perspective
Lit
Show

Fazit

Die Performance der Szene blieb trotz aller Anpassungen unbefriedigend. Dies lag hauptsächlich an der großen Menge an Foliage und den zahlreichen einzelnen Objekten. Zudem war die Nutzung von Nanite für das Foliage und die Bäume in meiner Szene keine optimale Wahl. Nanite ist in der aktuellen Version eher für Stadtszenen und geometrische Levels ausgelegt. Je feiner die Objekte und deren Überlappungen, desto schlechter die Performance.

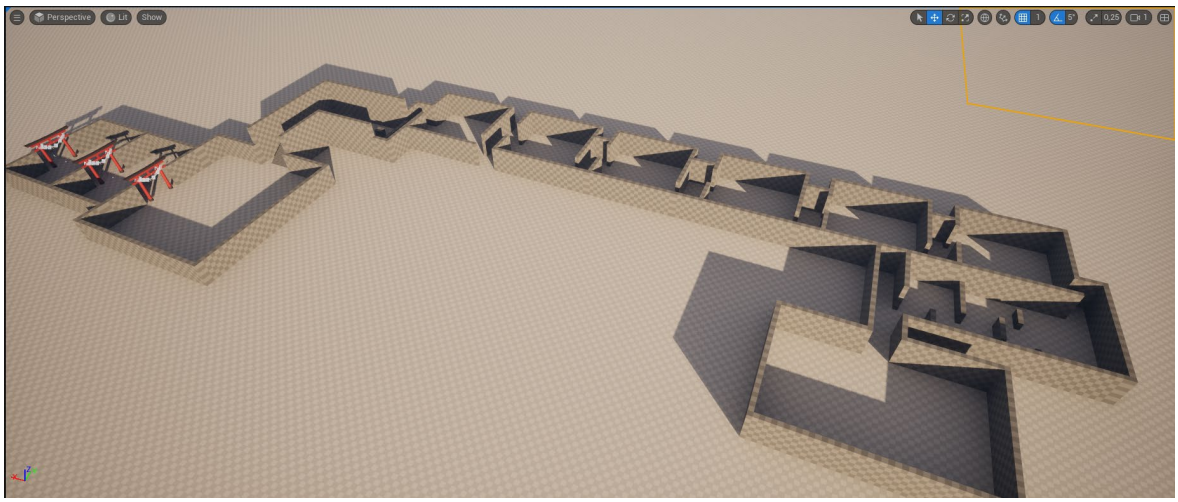
Leider wurde dieser Nachteil zu spät erkannt. Die beste Lösung wäre es, die Szene neu zu bauen und dabei das normale LOD-System zu verwenden, um die Level of Detail genauer bestimmen zu können. Da der Großteil der verwendeten Assets in der Nanite-Version importiert wurde, müssten sie in niedrig aufgelösten Varianten reimportiert werden. Dies macht jedoch wenig Sinn, da die AI des Bosses vermutlich nicht mehr rechtzeitig fertiggestellt werden kann.

Tutorial Level

Dieses Level ist dazu da die Funktionalitäten der Enemies und des Character Controllers zu Zeigen. Im wesentlichen heist das das der Aufbau wichtiger ist als die finale Optik.

Greyboxing

Deswegen wurde mithilfe des Modeling Modes der Unreal Engine zunächst ein funktionaler Aufbau erstellt, ein Prozess, der oft als Greyboxing bezeichnet wird. Als Basis wurde ein Open World Level genutzt. Dabei wurde vor allem die Modelling-Funktion "Create Cube Grid" verwendet, da sie ein schnelles Layouten des Levels ermöglicht.



Der Aufbau des Levels sieht dabei wie folgt aus:

1. Dummy Area
2. ParryTest Area
3. Perception Test Area
4. Jump Test
5. Water Enemy
6. Fire Enemy
7. Earth Enemy
8. Air Enemy
9. InAir Enemy
10. Heavy Enemy
11. Ranged Enemy
12. Gruppenkampf gegen mehrere Gegner

NavMesh

Die NavMeshes für die Raume werden so gesetzt das die Enemies ihren jeweiligen Raum nicht verlassen können.

Setzten der Gegner und Patrol Routes

Der Dummy in Raum 1 und der Enemy in Raum 2, der nur durch GuardRevenge-Attacken Schaden nehmen kann, sind schnell platziert. Anschließend werden die patrouillierenden Gegner im Perception-Raum positioniert. Die Routen werden erstellt, gelegt und den jeweiligen Gegnern zugeordnet.

Danach folgt eine kurze Passage, die man durch geschicktes Springen überqueren kann.

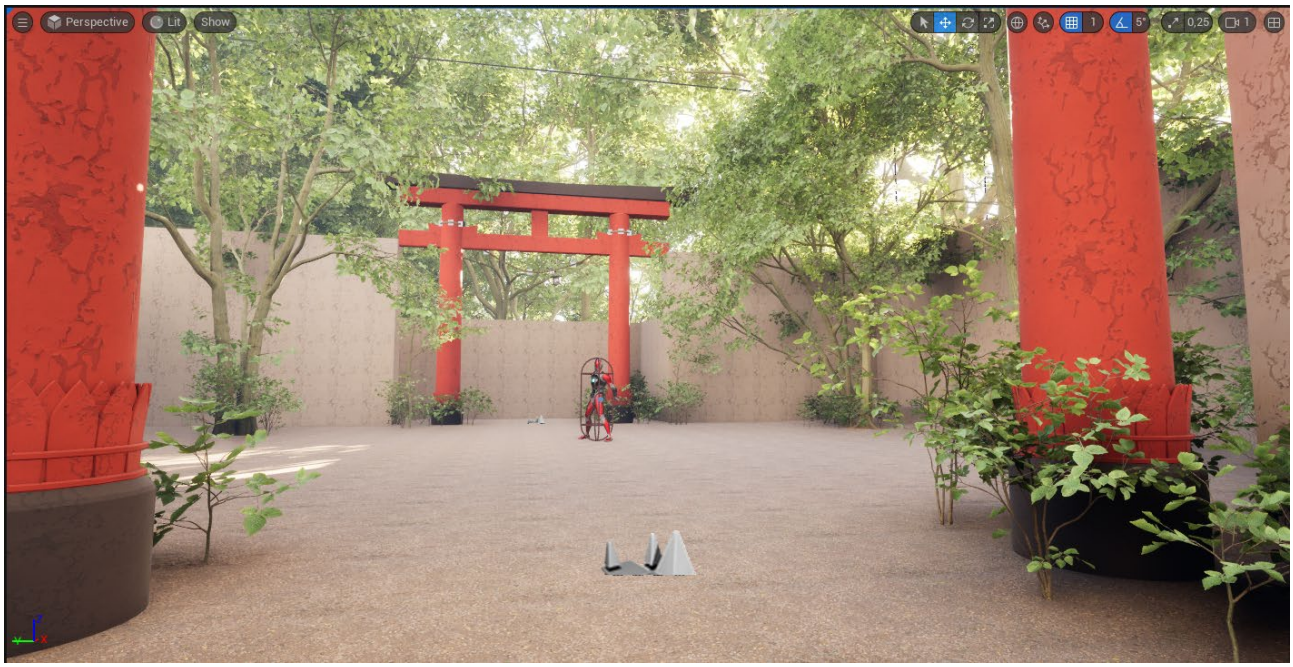
In den anderen Räumen werden die Angriffs-Typ spezifischen Gegner jeweils in der Mitte ihres Raumes gesetzt. Im vorletzten Raum werden einige Fernkampf Gegner mit Cover gesetzt. Der letzte Raum besteht aus mehreren Gegnern.

Foliage und Materials

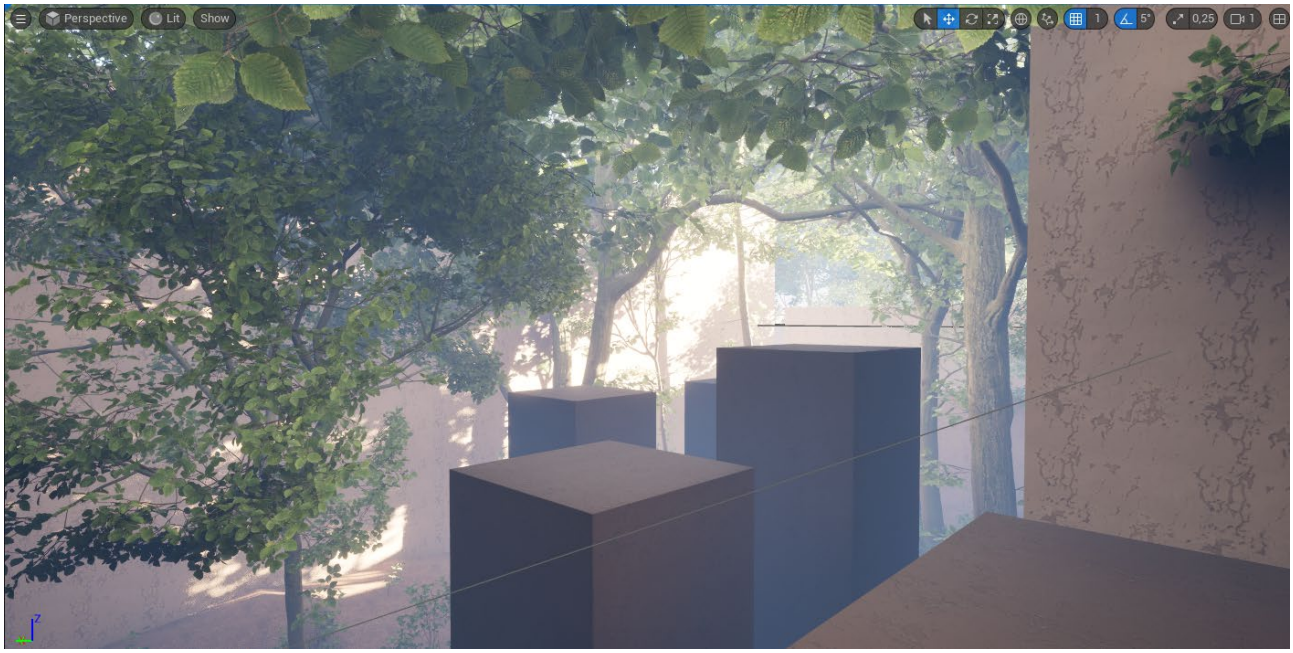
Das Material der Landschaft und der Wände wurde ersetzt, und es wurde Foliage platziert, um die Szene optisch aufzuwerten. Um die Performance zu schonen, wurde das Foliage im Single-Modus einzeln gesetzt, um nur das Notwendigste zu platzieren.

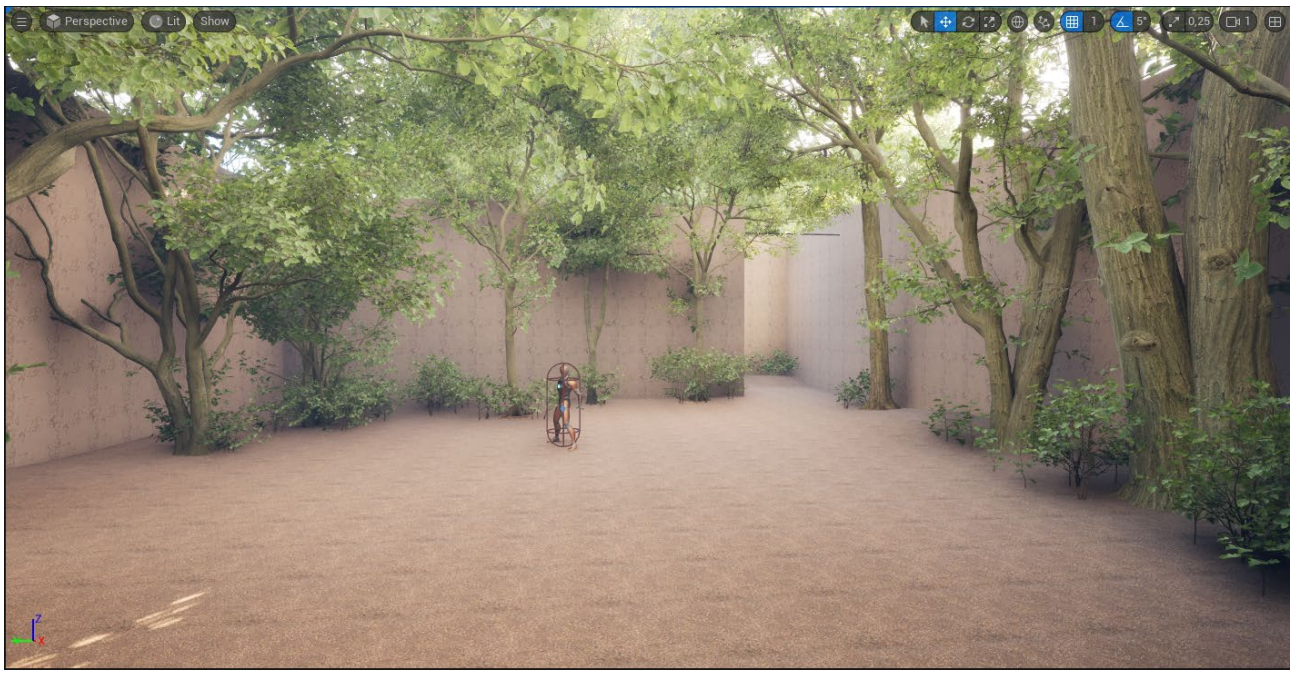
Leider lief die Zeit aus, und das Level musste in diesem Zustand verbleiben.

Endergebnis




Perspective
Lit
Show

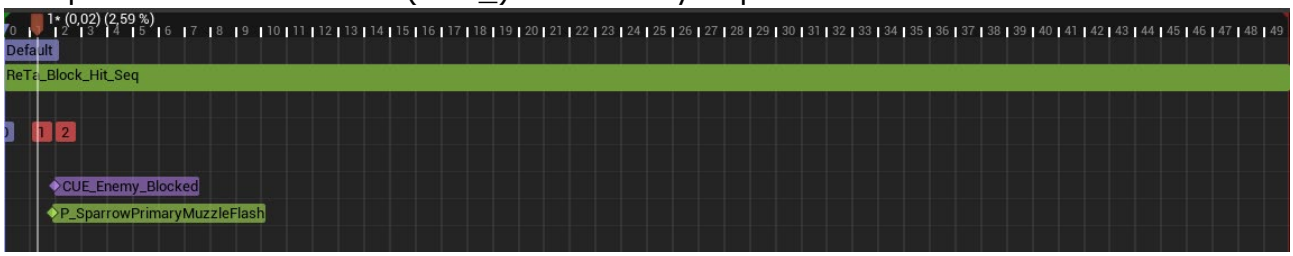

Perspective
Lit
Show


Perspective
Lit
Show


Perspective
Lit
Show

Sound

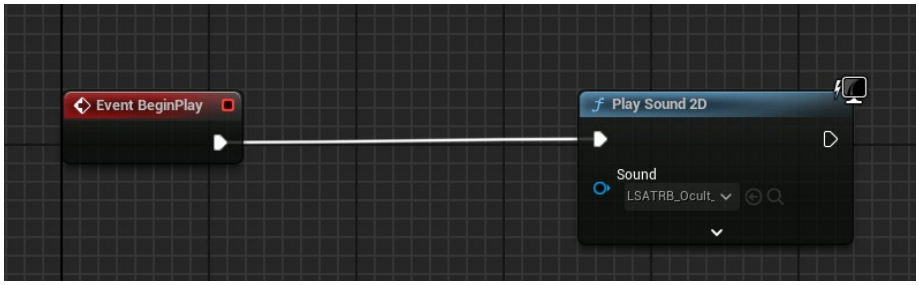
SFX wird innerhalb der genutzten Anim Montages durch einen Play Sound mit der entsprechenden Soundcue (CUE_) Anim Notify implementiert.



Einige Sounds stammen aus unterschiedlichen Quellen, die am Ende der Dokumentation aufgelistet werden. Diese wurden in Ableton so angepasst, dass sie

besser zusammenpassen, und anschließend in das Projekt importiert, um daraus eine Sound Cue zu erstellen. Andere Sounds wurden mithilfe von VSTs spezifisch für die Animation erstellt. Jeder extern beschaffte Sound wurde durch einen EQ und ein Gate angepasst. Zudem kamen Techniken wie Layering, Resampling und Pitchshifting zum Einsatz.

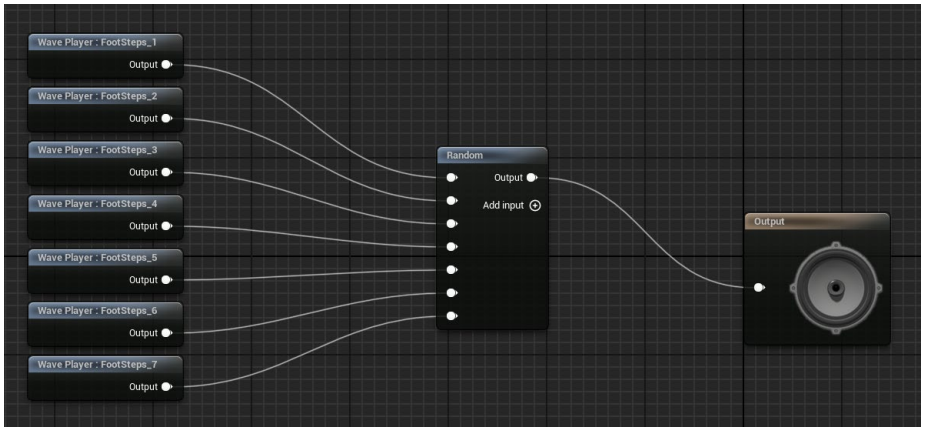
LevelSound



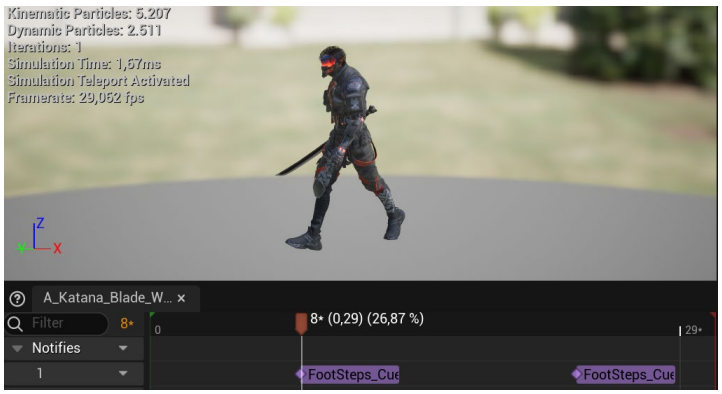
Der Level-Sound besteht aus dem Ambient Sound der von einem befreundeten Sound Designer erstellt wurde und wird im Blueprint der Levels bei Event Begin Start gestartet.

FootSteps

Hierzu wurden Footsteps aus den Foley-Aufnahmen der HS genommen, mit einem Gate und EQ nachbearbeitet und mit weiteren Foley-Aufnahmen gelayert. Diese wurden dann in einzelne Steps aufgeteilt und in der Sound Cue mit einer Random Node verknüpft, um die Footsteps natürlicher wirken zu lassen.



Die Cue wird innerhalb der Bewegungsanimationen immer bei Eintritt eines neuen Bodenkontakts mit einem Fuß ausgelöst.



DodgeSounds

Die Dodge Roll entsteht aus angepassten Foley Sounds und einem in Rise&Hit erzeugten Sound für das Woosh.

Air Attacks

Die Air Attacks wurden mit Rise&Hit erstellt, um einen passenden Swoosh-Sound zu bekommen dessen länge auf die Animation abgestimmt ist.

Fire Attacks

Bestehen aus drei Sound, einem Fire Ambient, klirrenden naegeln und einem Umgebungsrauschen. Dann mit Texture Warp verzerrt und mit Raum angepasst.

Water Attacks

Angepasster Water-Splash Sound von Freesounds.

GotHit Sounds

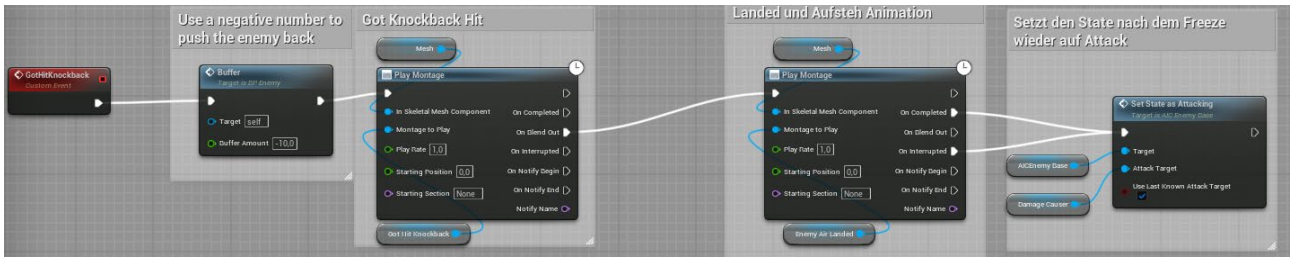
Gelayerte angepasste Sounds von Freesounds. Auch per Zufall in der Cue.

Soundeffekte Rest Zusammenfassung

Nach einigen Experimenten wurde beschlossen, die Klangumsetzung der Elementareffekte vorerst zurückzustellen und sich zunächst auf die Verbesserung der Slash-Sounds zu konzentrieren. Dazu wurden neue Sound Cues erstellt und dynamisch pro Animation zugewiesen, je nachdem, was am besten passte. Die Audio-Umsetzung der Elemente sollte langfristig als eigener Sound gelayert in der Engine hinzugefügt werden.

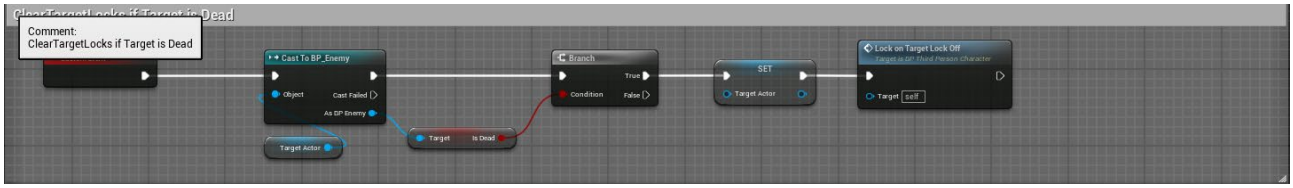
Bug Fixing und Finale Anpassungen

Guard Revenge Knockback im BP_Enemy



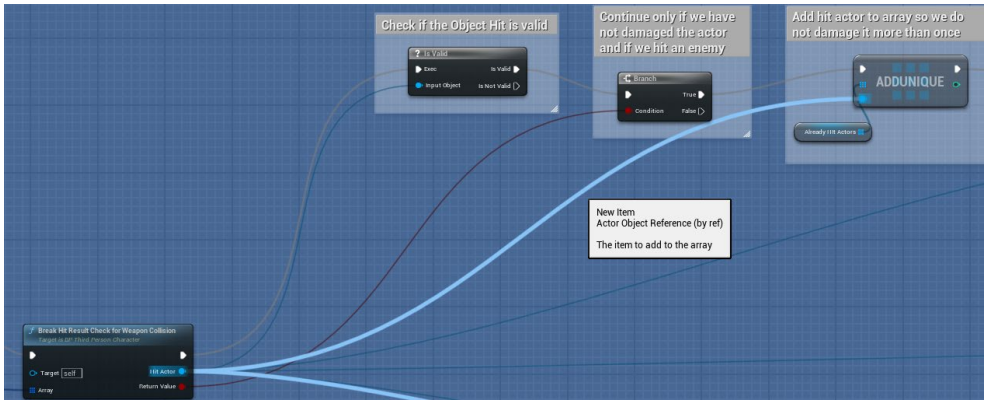
Wenn Gegner vom Guard Revenge Knockback getroffen wurden, trat das Problem auf, dass die Logik des Gegners nicht mehr richtig funktionierte. Dies lag daran, dass der Guard Slash Knockback über den InAir Knockback abgerufen wurde. Daher wurde der Knockback auf die On Ground Hit Knockbacks verschoben und um die Aufstehen-Animation ergänzt. Zusätzlich wurde der GuardSlash Test im OnDamageResponse Event des BP_Enemy entfernt.

ClearTargetLockIfItsDead im BP_ThirdPersonCharacter



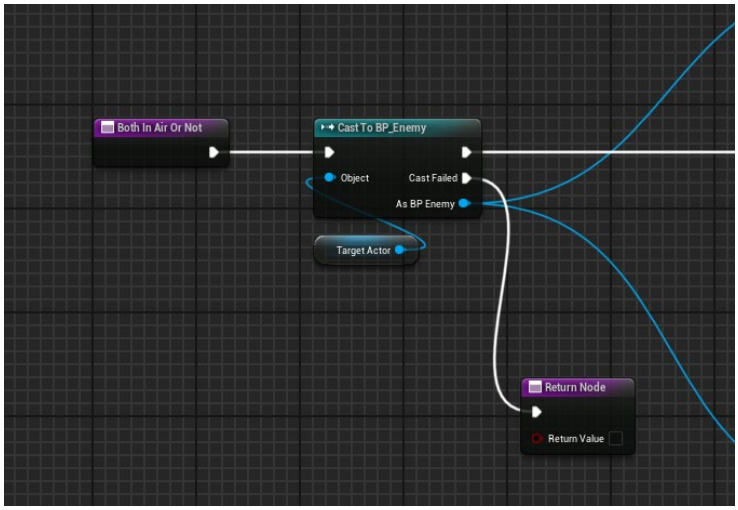
Immer wenn ein Gegner tot war, wurde das Target Lock trotzdem noch aktiv gehalten. Auch das SoftLock führte dazu, dass sich der Player beim Schlagen in Richtung eines toten Gegners ausrichtete. Dieses Event im BP_ThirdPersonCharacter wird per Tick aufgerufen und überprüft, ob das Target tot ist. Falls ja, wird das Target gelöscht und der LockOn-Modus ausgeschaltet.

Mehrfach Verteilter Schaden im durch BP_ThirdPersonCharacter



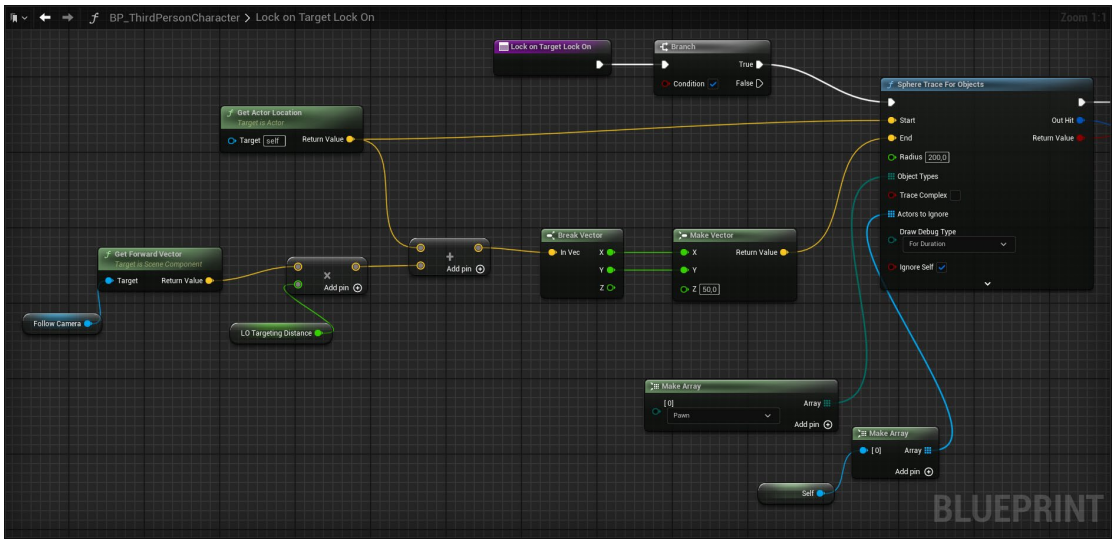
Der Schaden wurde nach der Umstrukturierung des Projekts zur Dokumentation mehrfach und damit nicht mehr korrekt ausgeteilt. Das Problem lag daran, dass die Verbindung des HitActors zu ADDUNIQUE AlreadyHitActors nicht mehr vorhanden war. Nachdem diese Verbindung wiederhergestellt wurde, funktionierte der Schaden wieder wie vorgesehen.

BothInAirOrNot im BP_ThirdPersonCharacter



Aufgrund dessen das der Target Actor auf Null gesetzt wird, wenn ein Gegner Tod ist, gab es einen Error bei der Pure Funktion, weil diese nicht testete ob der Cast erfolgreich war. Wurde behoben, indem die Funktion False ausgibt, wenn der kein Aktives Target existiert.

Target LockOn Sphere Trace im BP_ThrirdPersonCharacter



Der Sphere Trace hatte sich nicht mehr so wie gewünscht verhalten und wurde nicht in die Blickrichtung ausgeführt. Das lag daran das die Location des Players mit dem Forward Vector der Camera mal der Distanz multipliziert anstatt addiert wurde, dadurch wurde der Endpunkt des Trace nicht richtig gesetzt. Zusätzlich wurde die Z-Achse fest definiert, um das Targeten zu verbessern da in dem Projekt sowieso keine Gegner während si in der Luft sind neu erfasst werden müssen.

SmoothRotationToTarget im BP_ThirdPersonCharacter

Der Spieler zitterte komisch, wenn er beim Ausrichten durch den Lerp genau zwischen 180 und -180 grad stand. Das Problem war bereits aus Unity Projekten bekannt, normalerweise müsste hierfür eine neue Funktion erstellt werden, die dir kürzeste Rotationsdistanz errechnet und die Inputs des Lerps entsprechend anpasst. In Unreal ist das Problem durch das setzten des Hakens Shortest Path für den Lerp gelöst.

CanLandClasses im BP_Enemy

Im Tutorial Level funktionierte die Abfrage beimEventOnLanded nicht mehr. Das lag daran das der Boden des Landscapes kein StaticMeshActor ist, sondern eine LandscapeStreamingProxy Class. Diese musste hier lediglich ergänzt werden, um das Problem zu beheben.

BreakHitResultCheckForWeaponCollision im BP_ThirdPersonController

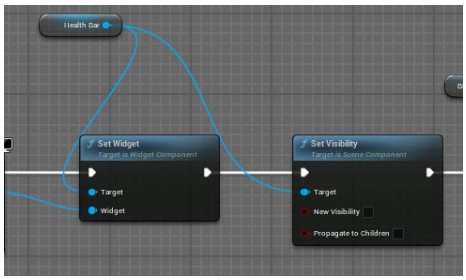
Es wurde Vergessen alle Gegnertypen in den Check mit einzubeziehen also konnten einige Gegner keinen Schaden vom Player bekommen. Das wurde gefixt, indem sie hier zum Check hinzugefügt wurden.

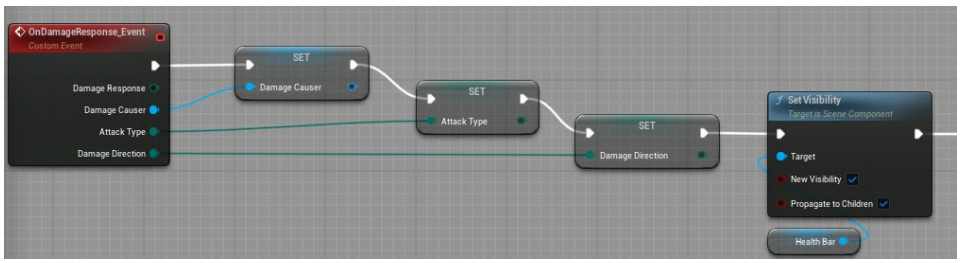
Altes BuiltIn DamageSystem

Da nichts mehr aus dem Alten DamageSystem das mit Unreal Engine kommt gebraucht wird werden die Überreste des alten Systems aus dem BP_ThirdPersonCharacter gelöscht und gegeben falls durch das neue ersetzt, wo es noch nicht der Fall war.

HealthBar Widget im BP_Enemy

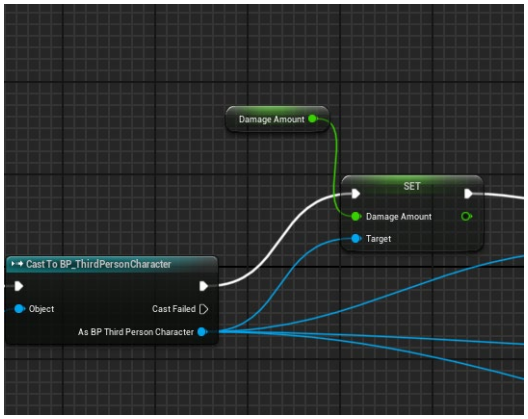
Die Health Bar der Enemies wurde angezeigt egal wo sie sich befinden und ob Sie Schaden bekommen haben oder nicht.





Indem die Visibility der Health Bar Nach der Erstellung auf False und anschließend so bald im Enemy das OnDamageResponse Event ausgelöst wird wieder sichtbar gemacht wird. Wird die Health Bar immer erst angezeigt, wenn ein Gegner schaden bekommen hat.

ANS_WeaponCollision



Anstatt das der von dem Character ausgeteilte Schaden bei jedem Schlag der gleich ist wird der Anim Notify ANS_Weapon Collision und der BP_ThirdPersonCharacter so ergänzt das der Schaden für jede WeaponCollision separat bestimmt werden kann. Somit kann der Schaden jeder einzelnen Attacke spezifisch angepasst werden.

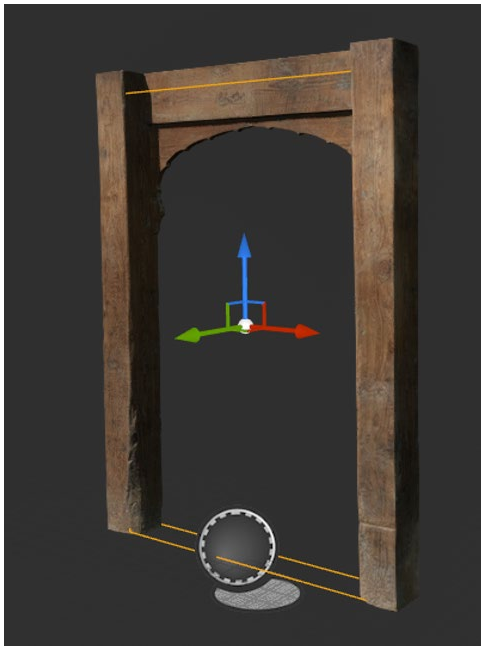
Tweaking

Das Tweaking wurde hauptsächlich durch die Anpassung der Werte der gesetzten Variablen, der Animationen und der Timings der Anim Notifys vorgenommen. Dies geschah im Verlauf des Projekts mehrfach, da viele Systeme voneinander abhängig sind und kontinuierlich angepasst werden mussten. Final wurden die zentralen Werte wie das Leben des Players und der NPCs sowie der Schaden, den die jeweiligen Attacken austeilen, angepasst.

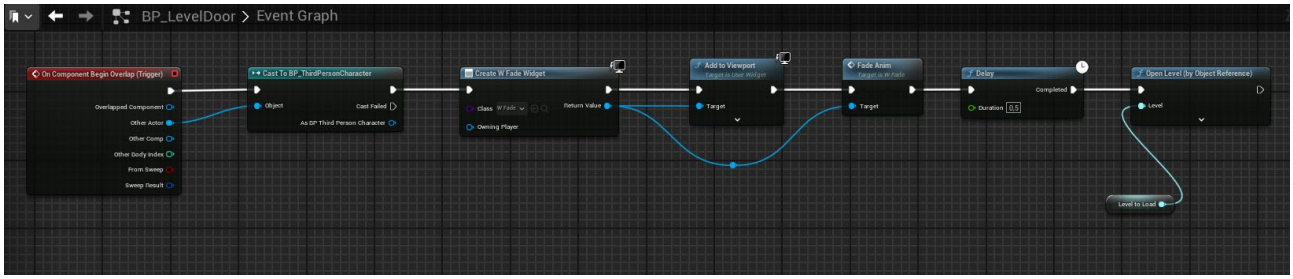
Build

Zum Abschluss wurde das Projekt gebuildet. Ursprünglich sollte das Boss-Level nach dem Tutorial-Level kommen. Da jedoch kein Boss im Projekt enthalten ist, wird das Build im Boss-Level gestartet. Der Spieler läuft durch das Level, geht durch eine Tür und gelangt dann zum Tutorial-Level.

BP_LevelDoor

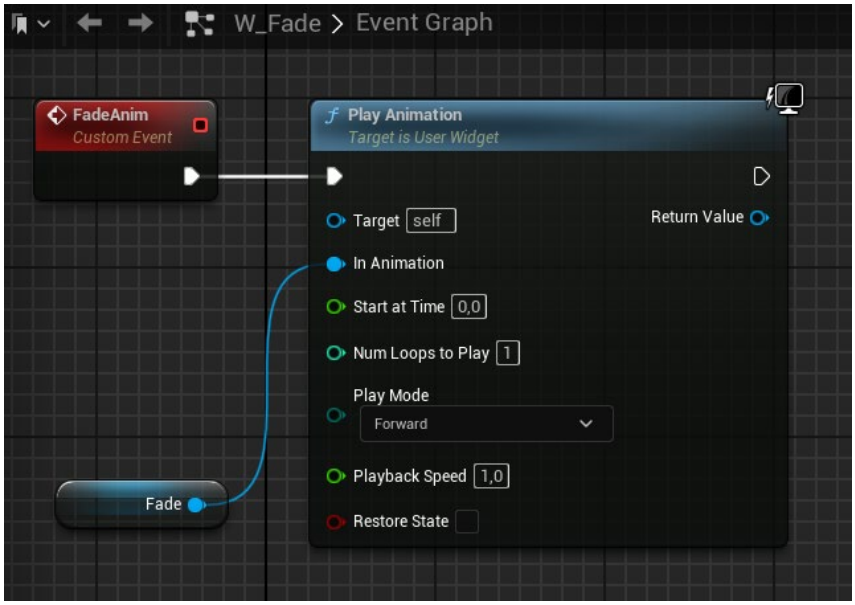


Hierzu wurde ein Blueprint Actor erstellt, um den Spieler beim Durchlaufen von einer Szene zur nächsten zu bringen. Dieser Blueprint Actor enthält das Mesh und einen Trigger. Ohne diesen Actor wäre eine Verbindung zwischen den Szenen im Build nicht möglich.



Im Event Graph des Actors wird das W_Fade Widget erzeugt und die Fade-Animation ausgelöst, sobald der Player mit dem Trigger überlappt. Nach einem Delay von 5 Sekunden, was der Animationszeit des Fades entspricht, wird das neue Level geladen.

W_Fade



Das W_Fade Widget ist ein Widget Blueprint, das den Bildschirm schwarz werden lässt, wenn das Level endet. Es besitzt eine Fade-Animation, die die Transparenz steuert und den Bildschirm dadurch schwarz werden lässt. Im Event Graph des Widgets wird einfach die Animation mit einem Custom Event gestartet.

Project Settings

Vor dem Builden des Projekts müssen einige Einstellungen vorgenommen werden. Die Build Configuration wird auf "Shipping" gesetzt. "Full Rebuild" wird aktiviert. Ein Thumbnail wird festgelegt und die Start Map wird definiert. Windows wird als einzige unterstützte Plattform ausgewählt.

Errors

Leider gab es mehrere Errors, bevor das packen des Builds klappte.

Windows SDK

Musste aktualisiert werden, um das Build zu packen.

Building compressed audio format BINKA

Konnte behoben, indem Visuall Studio geupdated wurde. Normalerweis nutze ich Raider und hatte alle notwendige für C++ installiert. Um den Error zu beheben, musste ich visuell Studio einmal komplett mit entsprechenden Packages neu installieren.

AutoSpline

Eine Komponente mit dem Namen AutoSpline machte Problem. Leider ergab die Suche im Internet nichts und ich musst mich manuell mit JetBrainsRaider Auf die Suche in den Files des Projektes machen, wo diese Komponente sitzt und was den Fehler verursacht. Darüber wurde herausgefunden das es sich bei dem AutoSpline um eine Komponente aus dem UltimateLevel Editor Tool Plugin handelt.

Zwar verbesserte das Entfernen des Plugins und der Reste davon auch erheblich die Performance im Boss Level. Doch leider zugunsten dessen das die Objekte, die

mithilfe des Plugins in das Level platziert worden sind verloren gingen und aus dem Level verschwanden.

CryptoKeys Cache not exist

Dieses Problem war nach vielen Ansätzen und versuchen nur lösbar, indem das gesamte Projekt einmal in ein anders Verzeichnis verschoben wurde, anscheinend lag es daran das der Pfad des Projektes ein ungültiges Zeichen enthielt, welches das Build-Tool der Unreal Engine nicht erfassen konnte.

Finales Build

Nach dem Fixen des „CryptoKeys Cache not exist" Problems konnte das Build final erstellt werden.

Resümee

Die Engine

Die Strukturierung von Content innerhalb eines Projektes in der Unreal Engine stellte sich als äußerst schwierig heraus. Assets vom Marketplace werden importiert, wie es vom Content bestimmt wurde, und die Art und Weise, wie die Registry in Unreal funktioniert, erlaubt zwar ein Verschieben des Contents, aber postum ist es nicht möglich, alte ungenutzte Ordner zu löschen. Dies führt oft zu zerstörten Referenzen und unerwarteten Fehlern, die behoben werden müssen, sei es bei Materialien, Texturen oder ganzen Blueprints, deren Funktion gestört ist und die neu erstellt werden müssen. Diese Problematik scheint schon seit der Unreal Engine 4 zu bestehen, wird jedoch langsam von Version zu Version der Unreal Engine 5 verbessert.

Die von mir gewählte Unreal Engine 5.3.2 erwies sich als ziemlich unzuverlässig, mit zahlreichen Abstürzen und einem komplett zerstörten BP_ThirdPersonControllers aufgrund eines aktuellen Fehlers meiner Version. Solche Probleme hatte ich bei meinen Projekten mit Unity nicht. Trotzdem muss ich gestehen, dass das Sammelsurium an Tools und Möglichkeiten, die die Unreal Engine von Haus aus mitbringt, beeindruckend und vielseitig ist. Die Modeling und Sculpting Tools ermöglichen einen ganz anderen Umgang mit der Engine und der Erstellung von Szenen. Die Tools, um Animationen anzupassen, und die Vielseitigkeit der vordefinierten Blueprint Classes bieten einen Workflow, der fast komplett innerhalb der Unreal Engine ohne externe Tools auskommt.

Genau das versuchte ich innerhalb des Projektes auch zu erreichen: Möglichst ohne externe Programme durch das Projekt zu kommen. Trotz der genannten Schwierigkeiten zeigt die Unreal Engine ihre Stärke in der integrierten Vielfalt und den umfassenden Möglichkeiten, die sie bietet.

Endzustand des Projektes

Im Großen und Ganzen bin ich sehr zufrieden mit dem Projekt. Zu Beginn versuchte ich vor allem, mit den mir aus C# bekannten Grundlagen zu arbeiten. Allerdings führten diese Ansätze häufig zu Logikfehlern, die das Projekt ausbremsten. Das erste Drittel des Projektes wurde fast ausschließlich auf den BP_ThirdPersonCharacter und das Kennenlernen der Engine verwendet. Der Character Controller wurde insgesamt dreimal komplett neu erstellt und einmal stark überarbeitet, um neu erlernte Systeme der Unreal Engine richtig zu implementieren.

Das zweite Drittel des Projektes widmete ich den Gegnern. In der ersten Version des Gegners war die komplette Logik innerhalb des BP_Enemy integriert, und es gab keine Behavior Trees. Die letzte Hälfte des Projektes wurde auf Sound und Levelbuilding verwendet. Dabei wurden immer wieder Änderungen am Character Controller und am Enemy Controller vorgenommen. Nach und nach konnte ich das Wissen, das ich aus Vorprojekten in Unity hatte, transferieren und die gewünschten Ergebnisse erzielen.

Im Endergebnis des Projektes wollte ich ein möglichst breit gefächertes Toolset der Möglichkeiten der Unreal Engine und der Blueprints verwendet haben und im finalen Zustand des Projektes zeigen können. Es ging mir vor allem darum, die Grenzen und Möglichkeiten der Engine zu erproben. Dabei konnte ich die Möglichkeiten der Animationsverarbeitung der Unreal Engine kennenlernen. Die vielseitige Verwendung von Anim Notifys, die Nutzungsmöglichkeiten der Animation Blueprint Class, und die Verwendung eines Character Blueprints zur Erstellung eines Character Controllers mithilfe von Enumerators, Arrays, Finite States, der Movement Component und des Built-In Damage Systems sowie eines eigenen Damage Systems.

Ich habe ein eigenes Damage System mithilfe von Interfaces, Structures und Enumerators erstellt. Die Funktionalität von Behavior Trees, inklusive Custom Environment Queries, Decorators, Services und Tasks, wurde erschlossen und erprobt.

Vor allem bin ich mit der Summe des während des Projektes erlernten Wissens sehr zufrieden. Das Endresultat der Bachelorarbeit stellt mich ebenfalls zufrieden.

Vermerk zur Dokumentation

Während der Dokumentation wurde ein Daily Log geführt. Zunächst als Word-Dokument und später, ab der Verwendung von GitHub, über die Kommentare der Commits. Der Umfang der Dokumentation hätte, wenn alle Versionen des Projektes detailliert aufgefasst worden wären, jeglichen Rahmen gesprengt und die Dokumentation auf über 300 Seiten gebracht. Selbst ohne diesen Teil hatte die Dokumentation, bevor sie von mir so gut es ging, mehrfach gekürzt worden ist, um die 250 Seiten. Einige Stellen der Dokumentation sind bewusst länger geschrieben, um die gedanklichen Schritte, die ich bei der Erstellung einiger Funktionen hatte, nachvollziehbarer zu machen. Ein Beispiel dafür ist der Abschnitt zum Damage System.

Fazit

Das Projekt hat mich auf eine gute Art und Weise unglaublich gefordert. Ich neige dazu, mir in meinen Projekten immer zu viel vorzunehmen, und genau das war hier auch wieder der Fall. Der Aufwand, den man abseits des Scripting und der Funktionalitäten hat, wurde von mir stark unterschätzt. Allein das Heraussuchen der Animationen, das Abstimmen der Timings der Anim Notifys und das Sammeln und Anpassen der Sounds dauerte mehrere Tage. Die Nebenarbeiten, die ich neben dem Erstellen der Funktionen hatte, verschlangen insgesamt viel mehr Zeit als ursprünglich eingeplant.

Somit kam ich zum Ende des Projektes in Zeitnot und konnte leider keinen Bossgegner mehr erstellen. Auch das visuelle Design des Tutorial Levels musste darunter leiden. Ebenso die Performance des BossLevels, dass ich zwar so optimieren konnte, dass es auf einem leistungsstarken PC spielbar ist, aber eigentlich noch einmal rebuildet werden müsste, um eine bessere Performance auf schlechteren Geräten zu ermöglichen. Insgesamt bin ich allerdings zufrieden mit dem Projekt.

Ich konnte die Engine in einem Ausmaß kennenlernen, das es mir ermöglicht, mich bei ausgeschriebenen Stellen, die Vorkenntnisse der Unreal Engine fordern, selbstbewusst zu bewerben und aufzutreten.

Die Videospielindustrie erwartet, dass man nie stehen bleibt, sondern ständig weiterlernt und sein Wissen erweitert und verbessert. Man muss immer auf dem neuesten Stand bleiben, und nachdem die Unity Engine vor kurzem so negative Schlagzeilen gemacht hatte und immer mehr Entwickler, die keine hauseigene Engine verwenden, die Unreal Engine bevorzugen, erschien mir dieses Projekt als sehr passend für meine Bachelorarbeit und das im Studium Erlernte.

Es geht dabei nicht nur um das Wissen aus den Kursen, sondern auch um das Skillset, mit neuen Herausforderungen klarzukommen, erlerntes Wissen auf neue Situationen und Umgebungen anwenden zu können und sich stetig eigenständig weiterzuentwickeln. Dieses Projekt hat mir die Möglichkeit gegeben, genau das zu tun.

Verwendete Assets und Plugins

Das Modell des Players

https://www.unrealengine.com/marketplace/en-US/product/cyberpunk-streetboy

Animationen

https://www.unrealengine.com/marketplace/en-US/product/grruzam-powerful-sword-pack

https://www.unrealengine.com/marketplace/en-US/product/sword-animation-pack-01

https://www.mixamo.com/#/

Schwerter

https://www.unrealengine.com/marketplace/en-US/product/sci-fi-swords-pack

Assault Rifle für den Ranged Enemy

https://cosmos.leartesstudios.com/products/cyberpunk-assault-rifles

Muzzle Effekt für die Assault Rifle und HitImpact

https://www.unrealengine.com/marketplace/en-US/product/paragon-sparrow

Sounds und SFX

https://soundscrate.com/

https://leartesstudios.gumroad.com/l/SFXsoulslike

Fooley Library der HS

https://mixkit.co/free-sound-effects/sword/

Ambient Sound erstellt von Evgeniy Ussach

Landscape und Levelbau

https://quixel.com/

https://www.unrealengine.com/marketplace/en-US/product/megascans-trees-european-hornbeam-early-access

https://cosmos.leartesstudios.com/products/fighting-stage

https://unf-vault.teachable.com/p/modeling-a-japanese-torii-in-unreal-engine-5

verwendete externe Tools (Plugins)

https://cosmos.leartesstudios.com/products/ultimate-level-art-tool

https://cosmos.leartesstudios.com/products/material-assignment-tool

https://cosmos.leartesstudios.com/products/object-distribution-tool

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sowie Inhalte des Projektes die nicht in Eigenarbeit entstanden und verwendet worden sind, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Unterhaching, 20.06.2024

_____

Julius Berndl