

Die offizielle Technische Dokumentation zu

Project Wallker

- If there is a wall, there is a way -

Projektmitglieder:

Marc Schillke	ms493	35138
Ngoc That Ton	nt023	35025
Fabian Reißer	fr055	35239

Veranstaltung: Gameplay-Programming 113522a
Prof. Dr. Stefan Radicke

Derzeitiger Stand: 14.07.2019

Inhaltsangabe:

0 Vorwort	S. 3
1 Content Browser	S. 3
2 Blueprint Übersicht	S. 4
2.1 Übergeordnete Blueprints	S. 4
2.2 Vektorwand Blueprints	S. 6
2.3 Charakterspezifische Blueprints	S. 8
2.4 Levelobjekt Blueprints	S. 12
2.4.1 Dynamische Hindernisse	S. 12
2.4.2 Kollisionsereignisse	S. 27
2.4.3 Sonstige Levelobjekte	S. 29
3 Logische Komponente	S. 34
3.1 Tastenbelegung	S. 34
3.2 Timer und Zeitrekorde	S. 36
3.3 Todeszonen und Checkpoints	S. 38
4 LevelDesign	S. 39
4.1 Tutorial-Sektion	S. 39
4.2 Levelknotenpunkt (HubWorld)	S. 40
4.3 LevelP1	S. 41
4.4 LevelP2	S. 42
4.5 LevelP3	S. 43

0. Vorwort

Project Wallker ist ein Gameplay Prototyp für die Veranstaltung *Gameplay Programming* an der Hochschule der Medien, welches im Sommersemester 2019 stattfand. Dozent und Betreuer dieses Moduls ist Prof. Dr. Stefan Radicke.

Diese technische Dokumentation des Projektes schildert detailliert die Spielgegebenheiten des Gameplay Prototypen sowie die Funktionalität der selbstgeschriebenen Blueprint-Skripte. Nicht im Fokus dieser technischen Dokumentation sind Skripte, die nicht als Eigentum der Projektmitglieder erachtet werden können und zusätzlich keinen Belang in Bezug auf die Projektvorgabe haben.

Im sprachlichen Sinne können innerhalb dieser Dokumentation englische Begriffe verwendet werden, welche primär auch in der verwendeten Software im Umlauf sind und dessen Nutzung wir auch auf deutscher Sprachebene Gebrauch machen. *Wir* bezieht sich auf alle Projektmitglieder und diese Dokumentation wird sich oft in dieser Perspektive äußern.

Für das Projekt wurde per Vorgabe der Veranstaltung die **Version 4.21.2-4753647** des **Unreal Editors** verwendet. Ebenfalls macht dieses Projekt Gebrauch von den Assets, welche im StarterContent des Unreal Editors mitgeliefert werden.

1. Content Browser

Der Content Browser des Projektes ist folgendermaßen aufgebaut und kategorisiert:

Ordner	Beschreibung
Audio	Hintergrundmusikdateien, welche im Spiel verwendet werden
Instances_Modes	Übergreifende Blueprints wie Instanzen und Modi
LevelMaps	Alle spielbaren Levels
MapObjects	Statische und dynamische Levelobjekte sowie Materialien
PlayerCharacter	Charakterspezifische Assets wie Blueprints, Animationen und Meshes
StarterContent	Von Unreal Editor mitgeliefert
UI	User Interface Dateien wie die HUD und sonstige Bildschirmausgaben

2. Blueprint Übersicht

2.1 Übergeordnete Blueprints

NewGameMode

In der Designphase des Projektes war geplant, dass es zwei GameModi geben soll je nach Levelmodi: PuzzleMode für Puzzle-Levels und SpeedModi für Speed-Levels. Da der jetzige Prototyp nun keine Speed-Levels mehr beinhaltet, wird einheitlich der NewGameMode verwendet.

SaveInstanceState

Diese Instanzdatei erlaubt es dem Spiel levelübergreifend auf Variablen zuzugreifen. Unter anderen sind folgende Variablen hier angelegt:

Name	Typ	Verwendungszweck
LevelP2_unlock	bool	Überprüft ob Levels bereits freigeschaltet wurden.
LevelP3_unlock	bool	Wichtig, um einen linearen Spielablauf zu ermöglichen.
display_vectorHelp	bool	Im Tutorial werden animierte Pfeile unter jeder Vektorwand eingeblendet, welche aber in jedem anderen Level typischerweise ausgeschaltet sind.
RadickeMode	bool	Der [RadickeMode] schaltet alle Spielinhalte frei. Ist dieser aktiviert, ändert sich eine Funktionalität innerhalb des Levels [HubWorld].
CheckPoint_Counter	int	Viele Levels implementieren ein Checkpoint System, welches die Todeszonen beeinflussen.
curMin / curSec	int	Der Levetimer speichert die Zeit in diese Integer
record_T1, ..., P3	int	Die Rekorde aller Levels, individuell gespeichert in Sekunden.
deathCount	int	Die Anzahl der Charaktertode sind hier festgehalten
deathReason	Text	Der Grund des letzten Todes ist hier festgehalten

GameHUD

Angegliedert sind folgende Dateien:

Name	Beschreibung
Boss_UI	Das Widget, welches den Namen und die Gesundheitsanzeige des Bosses in LevelP3 anzeigt.
Custom_UI	Das Widget, welches alle traditionellen HUD-Elemente auf den Bildschirm anzeigt wie gespeicherte Vektorkräfte und den Levetimer.
Help_UI	Das Widget, welche für die Hilfeanzeigen im Tutorial verantwortlich ist.

Ein detaillierter Einblick in Custom_UI:



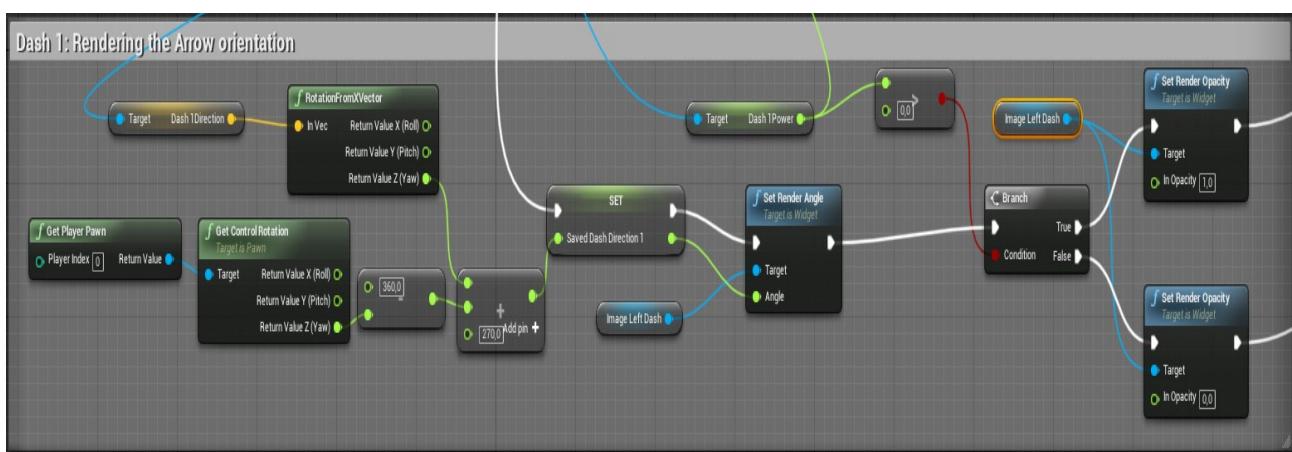
A – Der Vektormeter

Der Vektormeter ist eine Leiste, welche bei einem Wert von 1.0 voll ist und bei 0.0 leer.

Die Funktion erhält von der DashPlayer-Klasse die derzeitig gespeicherte Vektorkraft und das vom Level festgelegte Vektorkraftlimit und teilt die Vektorkraft durch das Limit um einen Wert zwischen 0.0 und 1.0 zu erhalten.

Ebenso prüft die Funktion, ob derzeitig der Vektoraufladeknopf gedrückt ist und falls nicht, dann wird der TLabel weiß dargestellt, ansonsten rot. Zusätzlich wird die Leiste bei Nichtdrücken des Vektoraufladeknopfs halbdurchsichtig dargestellt.

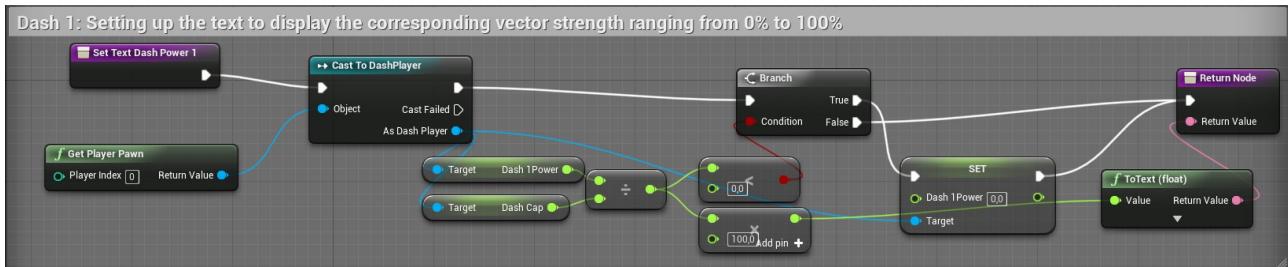
B – Der Vektorpfeil



Der Vektorpfeil wird eingeblendet, wenn die gespeicherte Vektorkraft mehr als 0.0 beträgt. Ansonsten bleibt der Vektorpfeil unsichtbar. Die Orientierung der eigentlichen Dash-Richtung

ist durch den **Dash1Direction** Parameter dargestellt, welches die DashPlayer-Klasse bereitstellt. Da der Vektorpfeil sich auch gegen die Spielerkamera drehen soll um immer in die Richtung des Vektors zu zeigen, muss durch den PlayerPawn die **ControlRotation** miteinbezogen werden. Aber da wir den Vektorpfeil entgegengesetzt von der Kamera-Rotation drehen möchten, ziehen wir von 360° die ControlRotation ab und rechnet damit weiter. Das Ergebnis wird mit Dash1Direction in Bezug auf der Z-Achse zusammen mit 270° addiert, denn das Vektorpfeilbild ist von sich aus falsch orientiert und dieses Manko wird hiermit behoben. Das Ergebnis wird dann via **SetRenderAngle** dargestellt.

C – Die Prozentanzeige des gespeicherten Vektors



Wie bereits bei **A** beschrieben teilen wir die gespeicherte Vektorkraft durch das Vektorkraftlimit. Das Ergebnis multiplizieren wir anschließend mit 100 und erhalten somit die Vektorkraft prozentual zum Vektorkraftlimit.

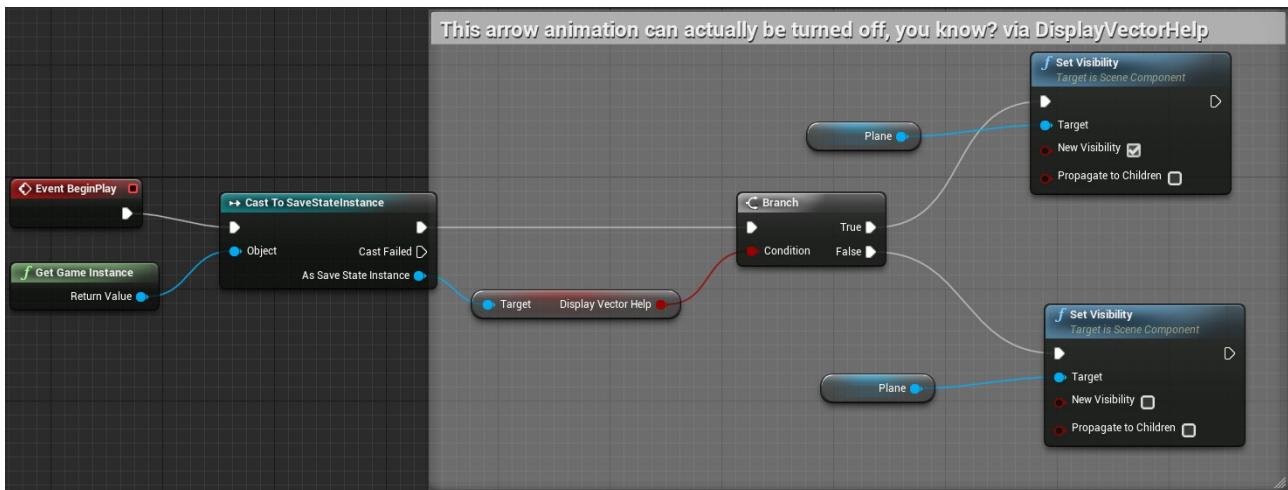
Als Bugs aufgetreten sind wodurch auch negative Prozente angezeigt wurden, musste eine Branch-Node hinzugefügt werden, welches negative Werte herausfiltert. Bei solchen Fällen wird der Text statisch auf 0.0% gesetzt.

D – Zusätzliche Anzeigen von Parametern

Die Informationen für die zusätzlichen Anzeigen bekommt die UI durch die SaveStateInstance. Zwar wird die Timer-Anzeige noch zuvor formatiert um eine Stoppuhr zu simulieren, der Rest wird jedoch ohne Formatierung dargestellt.

2.2 Vektorwand Blueprints

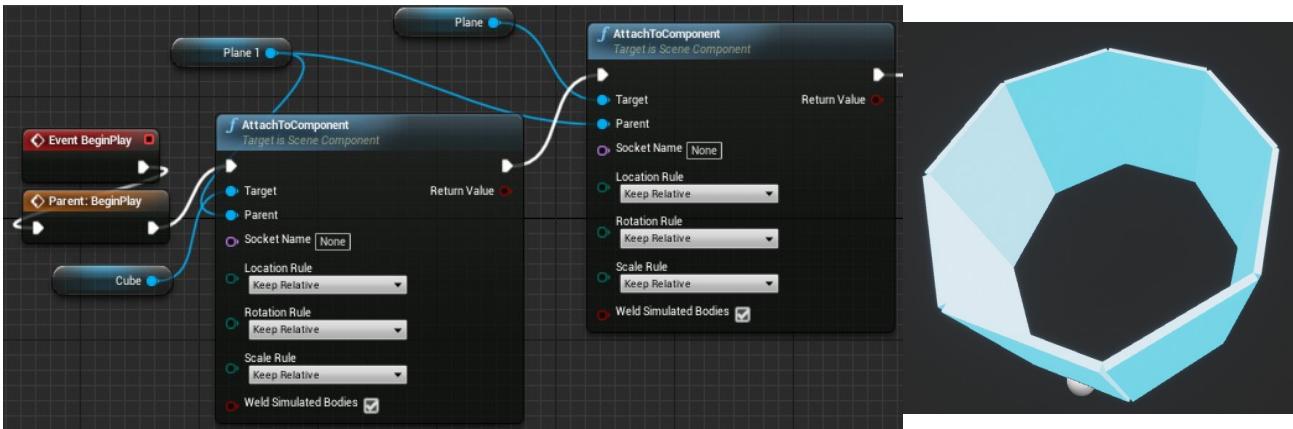
VectorHitBlock



Die VectorHitBlock-Klasse ist die einzige Klasse im Spiel, welches die Vektor-Mechanik ermöglicht. Überraschend zu diesem Fakt ist, dass die Vektor-Mechanik gar nicht in dieser VectorHitBlock-Klasse spezifiziert ist. Lediglich der PowerAdd Parameter ist entscheidend für die Aufladerate des Vektors, welches die Klasse bietet.

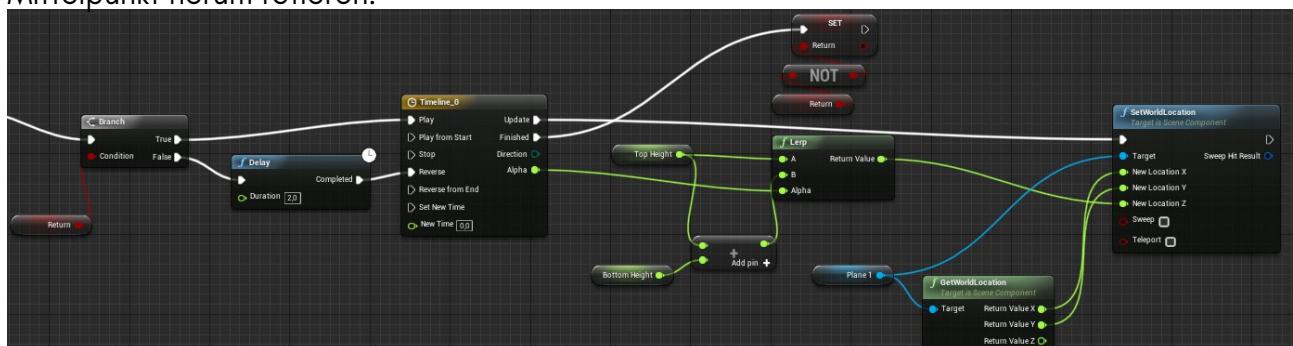
Die kurze Blueprint Sektion der Klasse bezieht sich auf die Animationspfeile unterhalb aller Vektorwände, welche in den Tutorial-Levels dargestellt werden, aber überall sonst ausgeschaltet sind. Die SaveStateInstance gibt Auskunft darüber, ob diese Animationspfeile angezeigt werden sollen oder nicht.

Carousel_naked



Das Vektorkarussel ist eine Kindklasse von VectorHitBlock und ist im Grunde ein zusammenhängendes Oktagon bestehend aus Vektorwänden. Zusätzlich ist eine achteckige, unsichtbare Plane gespannt denn es war aus Leveldesign-technischen Gründen wichtig, dass die Hauptachse dieses Vektorobjektes im Mesh-Mittelpunkt ist.

Die BeginPlay-Node fügt alle Elemente der geerbten Elternklasse zur zentrierten, unsichtbaren Plane hinzu und somit lässt sich das gesamte Vektorobjekt um seinen Mittelpunkt herum rotieren.



Ansonsten existiert im Vektorkarussel noch via EventTick-Node eine statische Rotation und ein periodisches Wippen nach oben und unten. Diese Bewegung ähnelt der von Karussells.

Auf dem obigen Bild wird das periodische Wippen funktional dargestellt. Das Auf und Ab wird durch eine Return-Flag ausgelöst und die Timeline-Node hilft eine geradlinige Bewegung von TopHeight zu BottomHeight zu bewerkstelligen. Die lineare Interpolation macht diese geradlinige Bewegung zudem noch spürbar geschmeidig. Ist die Bewegung abgeschlossen, wird die Flag umgeschalten und die Bewegung wird umgekehrt.

2.3 Charakter Blueprints

Wir haben uns, statt den Standard-Charakter von Unreal zu verwenden, dazu entschlossen den Charakter von null an selber aufzubauen. Dies war eine große Herausforderung da wir viele der Standardfunktionen selber nachbauen mussten. Im Allgemeinen war unser Fokus auf eine möglichst angenehme und natürliche Steuerung und der Implementierung der Dashfunktion gelegt.

MovementBase

MovementBase ist unsere Grundklasse, welche unseren Kontroller-Input auf Funktionen in unserem Charakter mappen. Diese Klasse kümmert sich um die Stick-Inputs und um den Sprung-Input. Es wäre, im Nachhinein, besser gewesen auch unsere anderen Inputs über unsere MovementBase abzubilden, allerdings kam diese Idee erst gegen Ende der Entwicklung und ein kompletter Umbau unserer Klassen zu diesem späten Zeitpunkt schien uns zu knapp um das Projekt fertigzustellen.

Der eigentliche Aufbau von MovementBase ist denkbar simpel, wir greifen aus unserem Kontroller die Eingaben ab und generieren daraus Geschwindigkeit und Richtung unserer Bewegung.

DashPlayer

Unser Charakter ist ein kinematischer Charakter. Diese Entscheidung haben wir getroffen um die maximale Kontrolle über unseren Charakter und allen seiner Werte und Positionen zu besitzen.

DashPlayer ist unser spielbarer Charakter und beinhaltet all seine Funktionalitäten: Die Berechnung der Bewegung als auch die Kollisions-Erkennung mit den Wänden und dem Boden, sowie der Anpassung an Steigungen um über diese laufen zu können. Ebenfalls ist das Aufladen unserer Dashes und das Auslösen dieser hier implementiert.

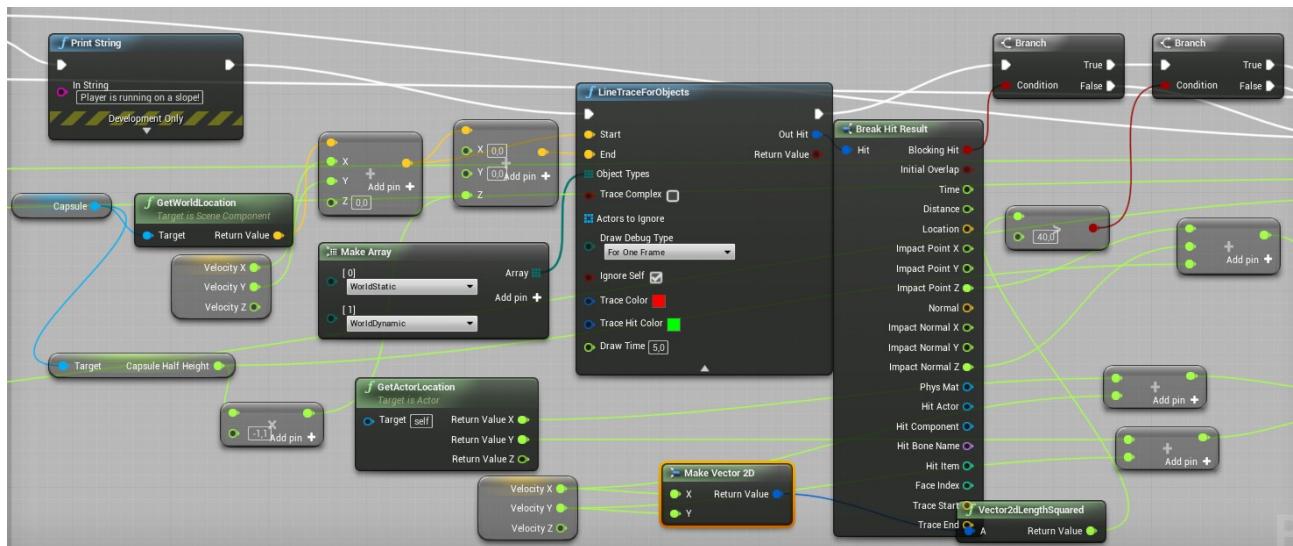
Name	Typ	Verwendungszweck
Gravity	float	Künstliche Gravitation
Velocity	vector	Unsere Geschwindigkeit in jede Richtung um unsere Position anzupassen
SweepPosition	vector	Die durch Geschwindigkeit und Kollisionstest errechnete neue Position unseres Charakters
BaseTurnRate	float	Die Geschwindigkeit mit welcher der Spieler die Kamera horizontal bewegen kann
BaseLookUpRate	float	Die Geschwindigkeit mit welcher der Spieler die Kamera vertikal bewegen kann
Dash1Direction	vector	Die Richtung des ersten Dashes. Ist ein normalisierter Vektor
Dash1Power	float	Die Kraft des aufgeladenen ersten Dashes
Dash2Direction	vector	Die Richtung des zweiten Dashes. Ist ein normalisierter Vektor

Dash2Power	float	Die Kraft des aufgeladenen zweiten Dashes
Dash1Active	bool	Ein Boolean um zu testen welcher Dash aktiv ist, und dazu da ist zu bestimmen, welcher gelöscht wird
IsDashing	bool	Ein Boolean um einen Dashenden-Zustand darzustellen und Movement und Geschwindigkeit zu setzen.
DashCap	float	Der Maximalwert der Aufladung der Dashes
Dashing1	bool	Zeigt welcher Dash gerade aktiviert ist und dadurch ausgelöst wird.
JumpPower	float	Die Kraft mit welcher der Charakter in die Luft befördert wird
allow_changeDash	bool	Workaround Boolean um im Tutorial den zweiten Dash zu deaktivieren
Held_Dash1	bool	Boolean um zu bestimmen ob Dash1 zum Aufladen oder Löschen ausgewählt wurde
Held_Dash2	bool	Boolean um zu bestimmen ob Dash2 zum Aufladen oder Löschen ausgewählt wurde
DoubleDash	bool	Boolean um zu Prüfen ob beide Dashes simultan ausgelöst wurden um eine andere Art der Berechnung zu aktivieren

Wie die Kollisionserkennung implementiert wurde

Die Kollisionserkennung ist tatsächlich der größte und komplexeste Teil unseres DashPlayer Blueprints, durch viele Nodes und Verbindungen wurde dieser immer unübersichtlicher. Unser Kollisionserkennung unterteilt sich grob in 4 Phasen, die Berechnung der Gravitation auf unseren Player, die Erkennung von Kollisionen mit dem Boden, die Kollision mit Wänden und die Prüfung auf Steigungen.

Die größte Herausforderung in diesem Bereich war das Laufen auf Steigungen, bis kurz vor Ende des Projekts war es kaum möglich Steigungen zu erklimmen.



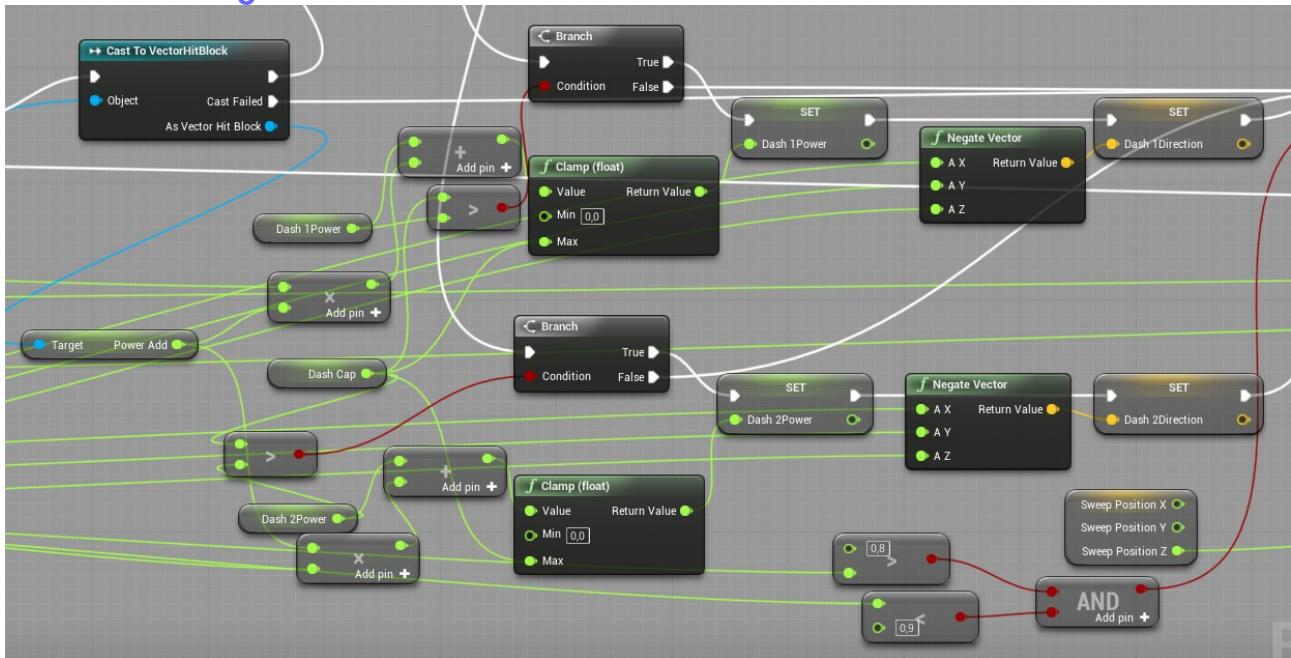
Nach der Prüfung ob wir uns auf einer Erhöhung befinden wird ein Raycast ausgeführt, dieser beginnt auf Höhe des Mittelpunktes unseres Charakters und der X- und Y-Achse des nächsten Frames, diese errechnet sich aus unserer aktuellen Position addiert mit unserer Geschwindigkeit(Velocity), und endet auf einige Einheiten unter unserer Fußhöhe.

Durch den Treffer den wir dadurch erhalten, können wir die nächste Z-Position erfahren und legen diese an unsere Z-Koordinate in SweepPosition an. Zusätzlich wird auf geringe Geschwindigkeit getestet, falls die Bewegungsgeschwindigkeit zu niedrig wird, wird dieser Code nicht ausgelöst, da ansonsten ein stottern in der Bewegung entsteht.

Die Dash Mechanik

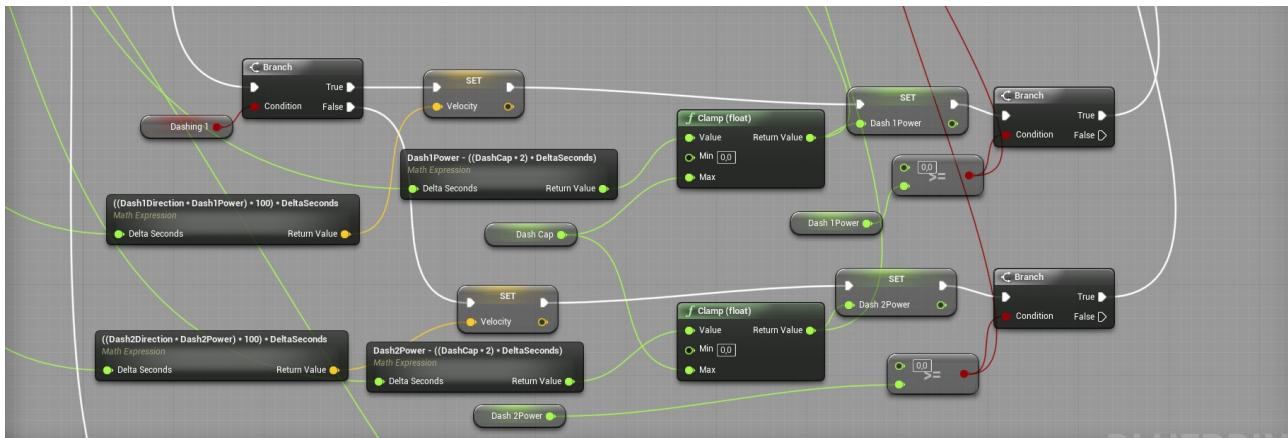
Das Dashen mithilfe von Vektorkräften ist das zentrale Spielelement von Projekt Wallker und basiert auf der Überlegung einen Dash an anderen Objekten aufzuladen wobei die Richtung der Kollision gleich der Richtung des aufgeladenen Dashes ist.

Vektoraufladung



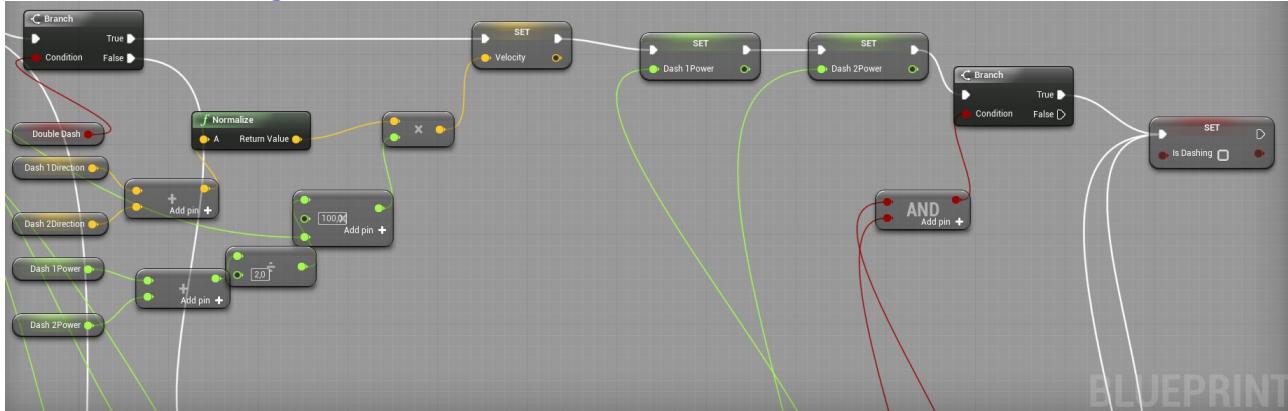
Bei Kollision Tests mit Wänden wird geprüft ob es sich bei diesen um eine besondere Art von Wand handelt, einem VectorHitBlock. VectorHitBlock besitzt einen Floatwert namens PowerAdd, dieser beschreibt die Aufladung der VectorPower pro Sekunde, je nach dem welcher ShoulderButton gedrückt wird während man gegen diese Wände läuft, lädt man die Power des ausgewählten Dashes auf. Ebenfalls wird die ImpactNormal negiert und dieser Vector anschließend als Richtung des Dashes in DashDirection festgelegt.

Vektorverwendung: Dash



Der Dash wird errechnet in dem man die Richtung des Dashes(DashDirection) mit der aufgeladenen Power(DashPower), einer Konstante um auf die kinetische Geschwindigkeit zu kommen, und den DeltaSekunden multipliziert. Diese Rechnung ergibt einen Vektor der unsere Dashgeschwindigkeit im 3Dimensionalen Raum abbildet, diesen setzen wir als Velocity unseres Charakters fest. Anschließend verrechnet man die Abnahme der DashPower. Diese ist so fest- gelegt, dass ein voll aufgeladener Dash genau 2 Sekunden anhält, solange er nicht unterbrochen wird, von einer Wand zum Beispiel.

Vektorverwendung: Combine Dash



Beim CombineDash werden die beiden aufgeladenen Dashes kombiniert um eine andere Richtung des Dashes zu erhalten. Hierzu werden die beiden Richtungsvektoren miteinander addiert und anschließend normalisiert. Dieser neue Richtungsvektor wird nun mit dem Durchschnitt der beiden aufgeladenen Vektorkräfte multipliziert. Das Ergebnis ist nun unsere kombinierte Geschwindigkeit und wird auf die Velocity gelegt.

Wenn ein Dash aktiviert wurde, kann der Spieler ihn nicht mehr aus eigener Kraft heraus beenden, ein Dash endet nur wenn eine Wand getroffen wurde oder die DashPower des Dashes auf null sinkt.

Vektorzurücksetzung

Aufgrund unserer Playtests und dem Feedback, welches wir für diese erhalten haben, wurde das Zurücksetzen unserer Vektoren nochmal überarbeitet. Zuvor bestimmte die letzte Kollision mit einer Vektorwand, in welche Richtung der Vektor zeigte. Dies machte es jedoch einfach,

aussersehen einen speziellen Vektor zu überschreiben und dadurch ein Rätsel nicht mehr lösbar zu machen, was in einem Reset endete. Da dies durch eine einfache Berührung einer Wand passierte, konnte dies sehr leicht vorkommen. Noch dazu kam, hatte man einen Vektor falsch aufgeladen, so musste man diesen mühsam wieder Verbrauchen, was besonders in Bereichen mit wenig Freiraum problematisch werden konnte. Als Lösung für diese Probleme haben wir folgende Ansätze implementiert: Erstens wird nun, wenn ein Vektor zu 100% aufgeladen ist, die Richtung des Vektors nicht mehr überschrieben, selbst wenn man eine Vektorwand berühren sollte. Zweitens können Vektoren nur noch aufgeladen und überschrieben werden, wenn eine extra Taste dafür gedrückt wird. Es ist möglich jeweils einen einzelnen zu Überschreiben, oder aber auch beide gleichzeitig. Als letzte große Neuerung an den Vektormechaniken resultierend aus unseren Playtests, ist ein designerter Button, mit welchen man den/die ausgewählten Vektor/en auf Knopfdruck komplett leeren kann. Dies erfolgt sofortig und komplett nach dem Knopfdruck, es ist nicht möglich mithilfe dieser Funktion nur einen Teil der Vektorkraft abzulassen, es setzt diesen Vektor komplett auf 0.

2.4 Levelobjekt Blueprints

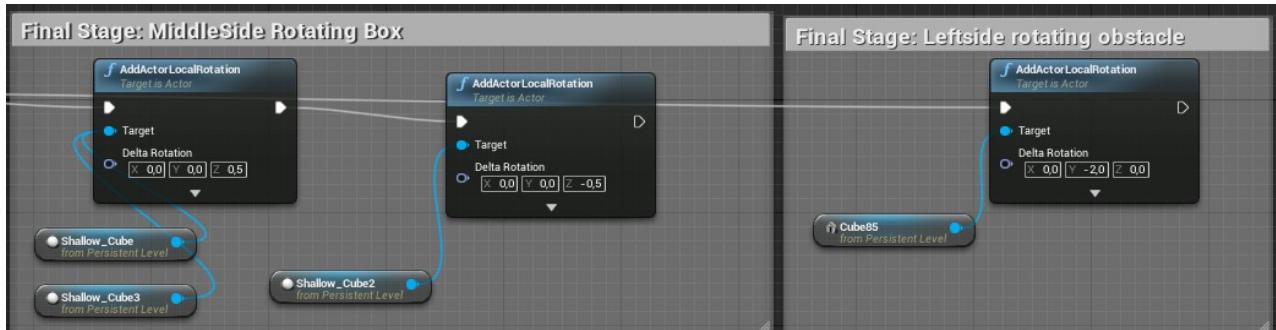
2.4.1 Dynamische Hindernisse

Als dynamische Hindernisse verstehen wir in Abhängigkeit zum Projekt die vom Spieler visuell wahrnehmbaren Hürden, welche der Spieler überwinden muss. Dynamisch verwenden wir hier als Abgrenzung zu statischen Hindernissen, welche keine besonderen Blueprint-Funktionalitäten aufweisen und somit nicht im Fokus dieser technischen Dokumentation stehen. Nichtsdestotrotz existieren auch in der Kategorie der dynamischen Hindernisse Abgrenzungen, denn im Laufe der Projektbearbeitung wurden interne Strukturen innerhalb der Blueprints überarbeitet und somit sind auch verschiedene, sogenannte Generationen entstanden worden.

Dynamische Hindernisse: Erste Generation

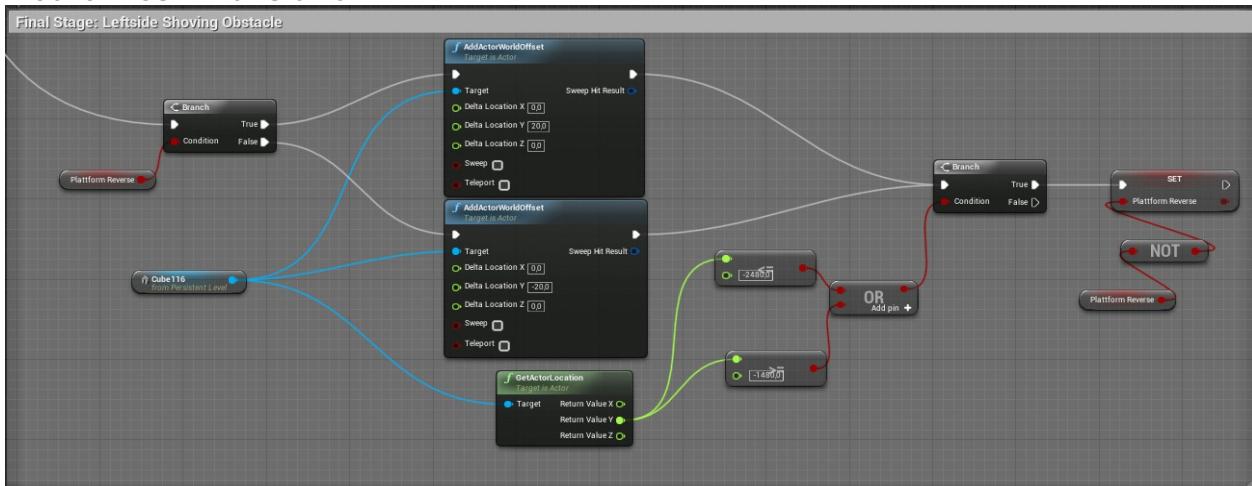
Auf diese Hindernisse trifft der Spieler nur im ersten Tutorial-Level und vereinzelt in LevelP1. Für die automatisierte, rhythmische Bewegung bedient sich die erste Generation am LevelBlueprint. Somit sind es statische Objektmeshes, welche durch den LevelBlueprint aus externe Quelle heraus bewegt werden. Aufgrund dieser Abhängigkeit ist der Aufwandsanteil innerhalb des LevelBlueprints relativ hoch und kann mit Ansteigen der Levelkomplexität auch unnötig die allgemeine Lesbarkeit erschweren. Bewegungen dieser Art können in zwei Rubriken aufgeteilt werden: Lineare Mesh-Translation und Lineare Mesh-Rotation.

Lineare Mesh-Rotation



Auf dem obigen Beispiel sind die beiden einzigen Anwendungsbeispiele gezeigt, eine *Rotation um die Z-Achse* und eine *Rotation um die Y-Achse*. Erstes ist eine horizontal rotierende Plattform worauf der Spieler stehen kann. Die Rotationsgeschwindigkeit ist aus diesem Grund auch vergleichsweise geringer als Mesh-Rotationen um die X- oder Y-Achse. Auf der rechten Bildseite ist solches zu Bestaunen. Dieses Hindernis ist vertikal rotierend und versperrt dem Spieler den Weg. Je höher die Rotationsrate, desto geringer ist das Timing, mit welchen der Spieler dieses Hindernis überwinden kann. Wichtig bei beiden Anwendungsbeispielen ist die Tatsache, dass der Blueprintaufwand hierbei unter der *EventTick-Node* zufinden ist.

Lineare Mesh-Translation



Lineare Mesh-Translationen beschreiben eine lineare Bewegung des Objektmeshes in eine bestimmte Richtung bis zu einem statisch festgelegten Bereich. Ist dieser Bereich erreicht, wird ein Boolean umgeschalten und der Objektmesh bewegt sich anschließend in die entgegengesetzte Richtung bis zu einem statisch festgelegten Bereich. Dieser Vorgang wiederholt sich ununterbrochen und auf unbestimmte Zeit.

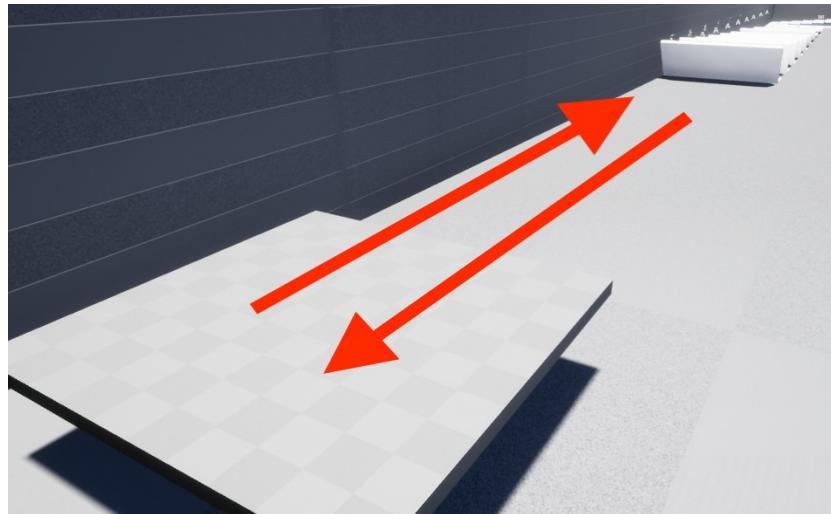
Für dynamische Objekte der ersten Generation ist es unabdingbar, dass der LevelBlueprint auf diese Objektmeshes zugreifen und referenzieren kann, denn sonst ist diese Bewegungszuweisung unmöglich. Da der gesamte Aufwand im LevelBlueprint zu finden ist, muss unter Umständen dafür gesorgt werden, dass diese angesprochenen Objektmeshes auch namentlich erkennbar gemacht werden, denn sonst hätte man nur schwer kontextuellen Bezug zu diesen herkömmlichen Objektmeshes. Im Großen und Ganzen ist diese Art der Bewegungszuweisung funktional vollständig, birgt aber für komplexere Levels ein viel zu hohes Risiko.

Dynamische Hindernisse: Zweite Generation

Auf diese Hindernisse trifft der Spieler beginnend ab LevelP1 und vorwiegend in LevelP2. Eine große Veränderung im Vergleich zur vorherigen Generation ist die Tatsache, dass alle Funktionalitäten nun im ObjektBlueprint zu finden sind. Zusätzlich dazu kann der Leveledesigner noch flexibel die Bewegungseigenschaften parametrisieren und somit sind diese Hindernisse auch nach Platzierung in das Level noch anpassbar. Zum Teil ist aber eine neue Schwierigkeit aufgetreten um frei parametrisierte Vorgänge zu erlauben, welche wir im Folgendem erläutern werden.

Die Bewegungseigenschaften sind nach wie vor erhalten geblieben: Linear Mesh-Rotation und Linear Mesh-Translation. Bei der Linear Mesh-Rotation gibt es im Vergleich zur ersten Generation keine Unterschiede. Die Rotationsrate ist auch frei anpassbar und es bedarf keinen zusätzlichen Aufwand zur Implementierung dieser Flexibilität. Bei der Linear Mesh-Translation jedoch mussten viele Änderungen vorgenommen werden.

Lineare Mesh-Translation: Zweite Generation



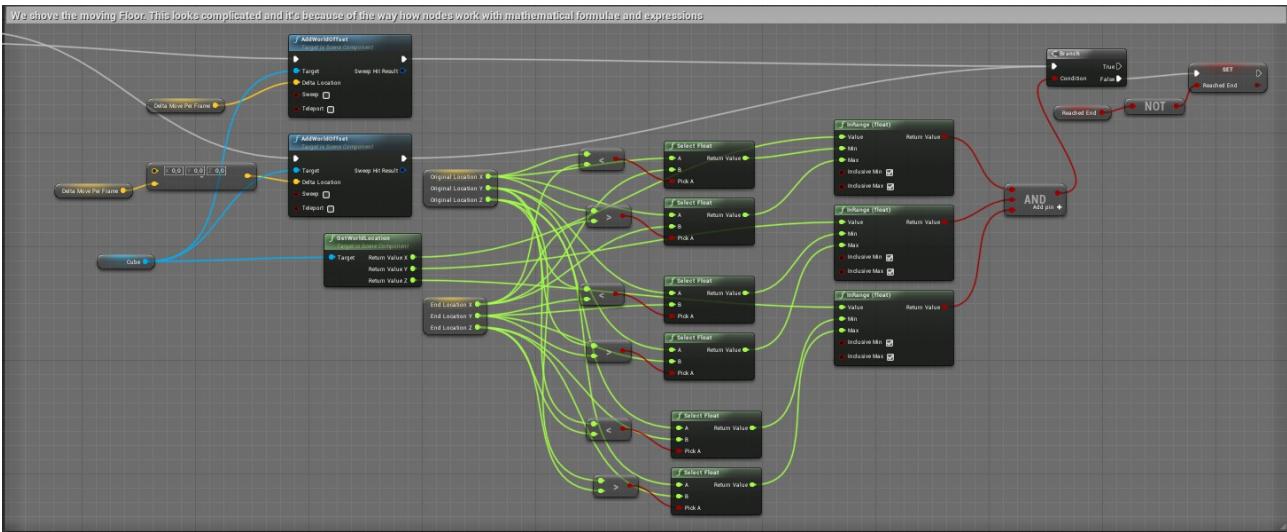
Das im Bild gezeigte Levelobjekt ist der LinearMovingFloor, ein dynamisches Hindernis der zweiten Generation, welches in LevelP2 Gebrauch findet. Folgende Variablen finden sich innerhalb dessen ObjektBlueprint:

Name	Typ	Öffentlich?	Verwendungszweck
OriginalLocation	vector	Nein	Ausgangsposition des Objektmeshes
EndLocation	vector	Nein	Zielposition der linearen Bewegung
DeltaMovePerFrame	vector	Ja	Bewegungsgeschwindigkeit pro Frame
LocationDelta	vector	Ja	Distanz von Ausgangsposition zum Ziel
reachedEnd	bool	Nein	Flag für die umgekehrte Bewegung

Im Editor kann der Leveldesigner via DeltaMovePerFrame festlegen, wie schnell der Objektmesh sich in eine bestimmte Richtung bewegt. Die LocationDelta gibt die Distanz an, welche das Objektmesh zurücklegen soll ehe es zurückbewegt wird.

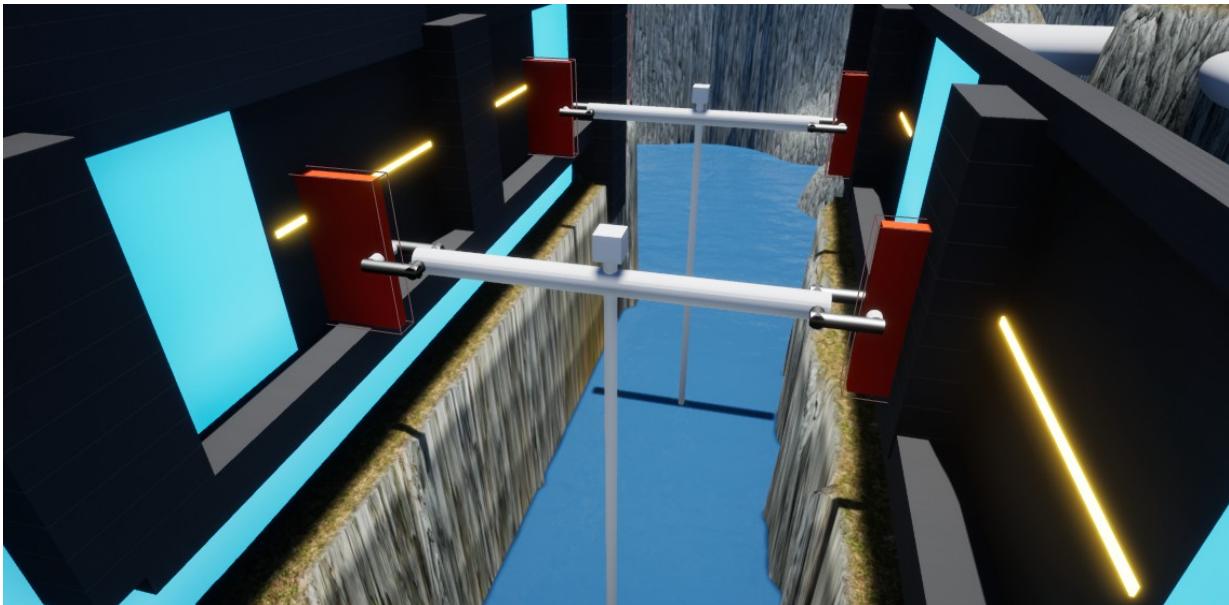
In der BeginPlay-Node wird die Ausgangsposition in OriginalLocation abgespeichert und die EndLocation ergibt sich aus der Addition von OriginalLocation und der vom Leveldesigner angegebenen LocationDelta. Die wahre Hürde im Blueprint ist es herauszufinden, wann der reachedEnd Boolean umgeschaltet werden soll. Das Bild auf der nächsten Seite zeigt den Ausschnitt dieses Unterfangens. Via InRange-Node wird geprüft ob die Plattform in X-, Y- und Z-Position sich noch inmitten zweier Werte befindet, und wenn bei einem der drei Koordinaten diese Kondition nicht mehr zutrifft, dann wird der Boolean umgeschaltet.

Die Hauptschwierigkeit war es eine flexible Blueprint-Sektion zu gestalten, welches auch mit negativen Koordinaten arbeiten kann. Der SelectFloat-Node hilft herauszufinden, welches der jeweiligen Koordinaten von OriginalLocation und EndLocation kleiner, beziehungsweise größer ist, um für die InRange-Node das Minimum und Maximum festzulegen.



Die zweite Generation der dynamischen Hindernisse bietet zudem noch zwei neue Levelobjekte, welche für den Spieler auch aktiv Konsequenzen bieten.

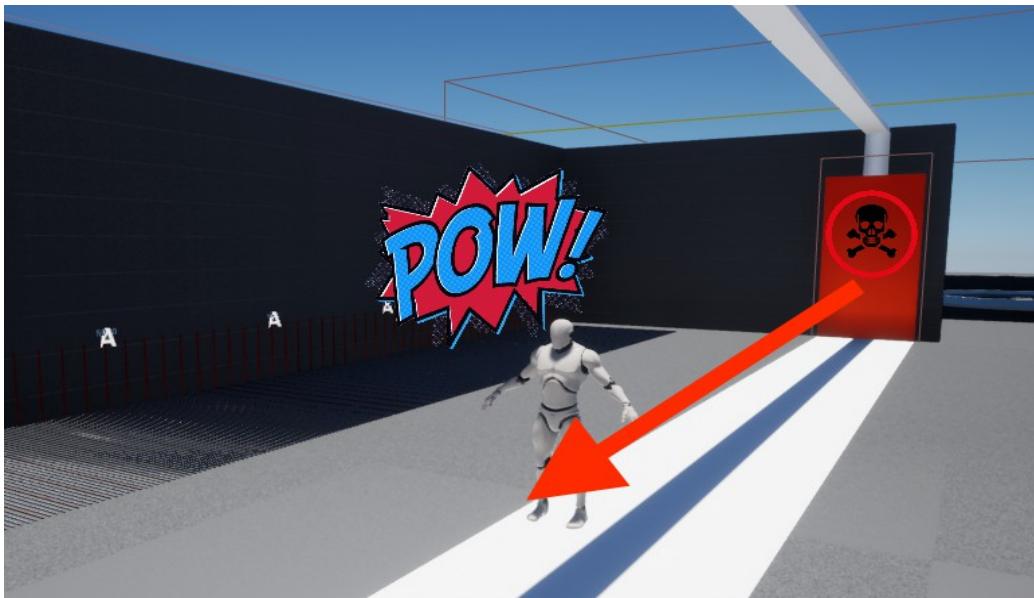
Paddle-Rotation: Zweite Generation



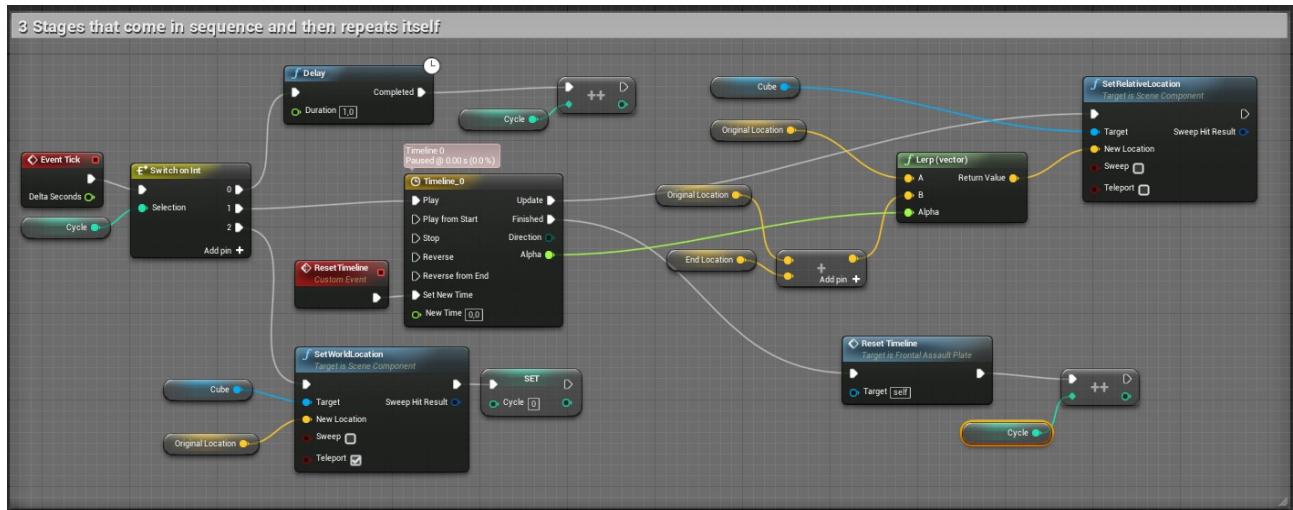
Die Paddle-Rotation ist ein sich horizontal rotierendes Levelobjekt mit zwei roten Platten an jeder Seite. Der Spieler trifft auf dieses Hindernis nur in LevelPl. Zusätzlich zur gleichmäßigen Rotation mit frei anpassbarer Rotationsrate verursachen die beiden roten Platten bei Kollision mit dem Spieler eine Todesteleportation in eine intern festgelegte Position.

Ein großes physikalisch-basiertes Problem ist es herauszufinden, mit welcher Art von Kollisionserkennung die Paddle-Rotation arbeitet, somit ist die Paddle-Rotation nur mäßig funktional und wird auch nirgendwo anders im Spiel Gebrauch finden, bis die Lösung des Problems gefunden und implementiert wurde.

FrontalAssault-Plate: Zweite Generation



Die FrontalAssault-Plate ist ein dynamisches Hindernis, welches bei Berührung mit dem Spieler eine Todesteleportation verursacht. Die Position der Todesteleportation ist im Vergleich zur Paddle-Rotation frei anpassbar. Ebenso gibt der Leveldesigner nach Platzieren dieses Hindernisses noch die Distanz der linearen Bewegung an. Der frei anpassbare Parameter ist ein Vektor, jedoch wird nur die positive Bewegung auf der X-Achse gebraucht.



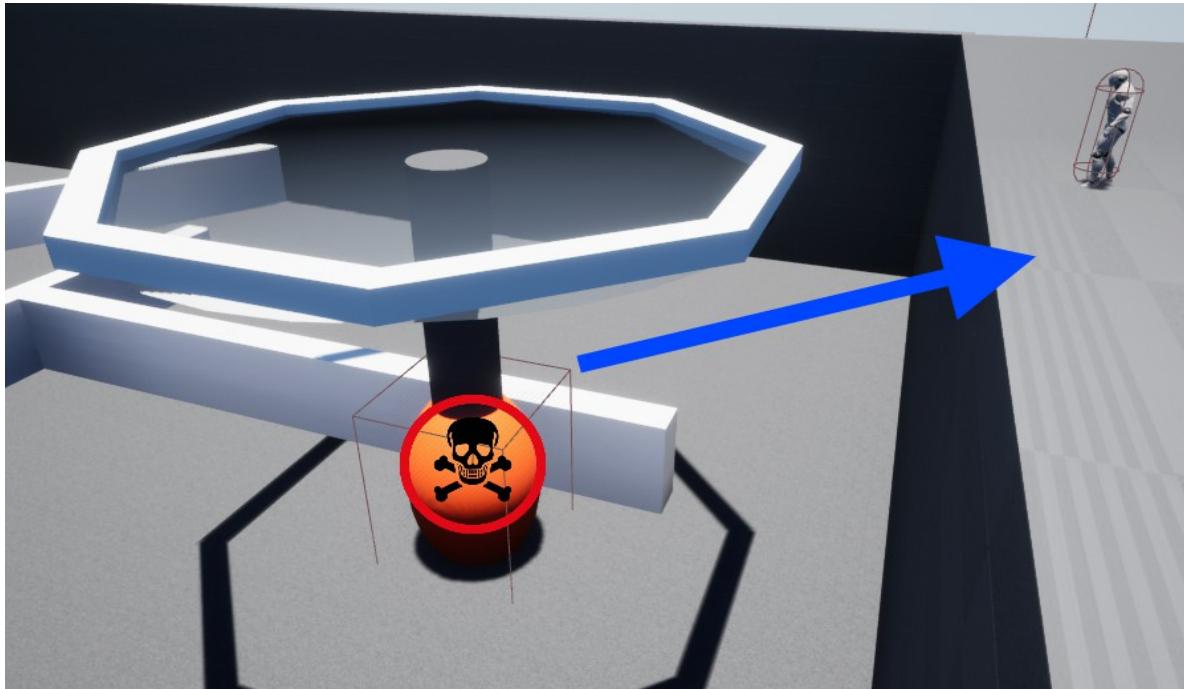
Funktional wird innerhalb des Blueprint mit einer Timeline gearbeitet. Ebenso folgt die FrontalAssault-Plate einem sich immer wiederholenden Zyklus von drei Etappen. Bei der ersten Etappe wartet die rote Platte für eine Sekunde. Die zweite Etappe beschreibt die rasend schnelle Vorwärtsbewegung. Nach der Beendigung dieser Bewegung wird die Timeline zurückgesetzt (weil wir keine Rückbewegung haben möchten) und die letzte Etappe teleportiert die rote Platte schließlich zurück zur Ausgangsposition und startet den Zyklus neu.

Dynamische Hindernisse: Dritte Generation (Namentlich: Gestapo Serie)

Die dynamischen Hindernissen der dritten Generation haben in sich ein geschlossenes System dessen Bewegungseigenschaften spielerabhängig sind. Diese sind in der Lage den Spieler zu orten und sich in Richtung oder entgegen der Richtung des Spielers zu bewegen. Für diese

Hindernisse existieren jeweils Untersysteme in Form von Blueprints, welche unabdingbar für das Funktionieren dieser sogenannte Gestapo-Hindernisse sind. Den Walker findet der Spieler in einer kurzen Tutorial-Sektion in Level02 und in LevelP2 bei der eigentlichen Anwendung wieder.

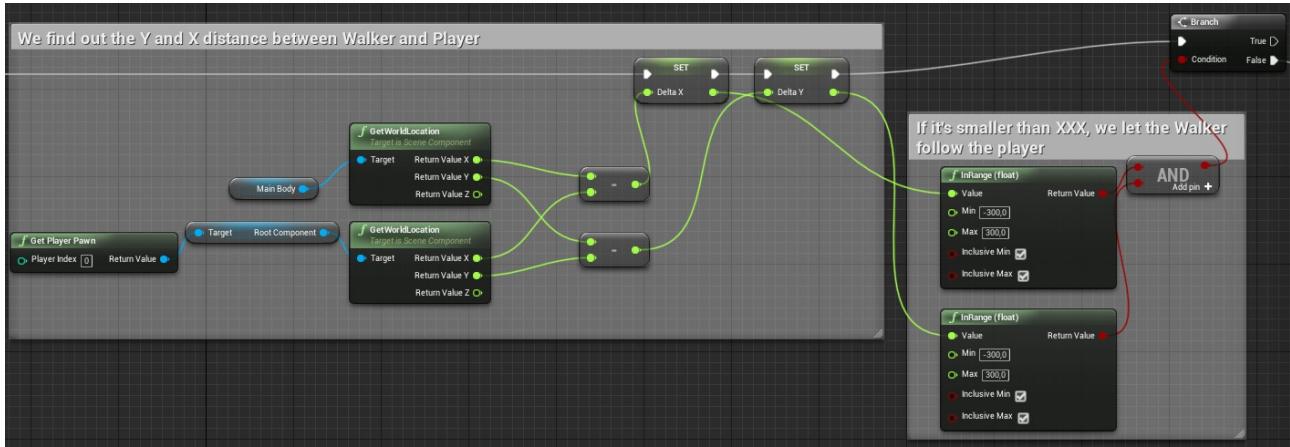
Gestapo-Walker: Dritte Generation



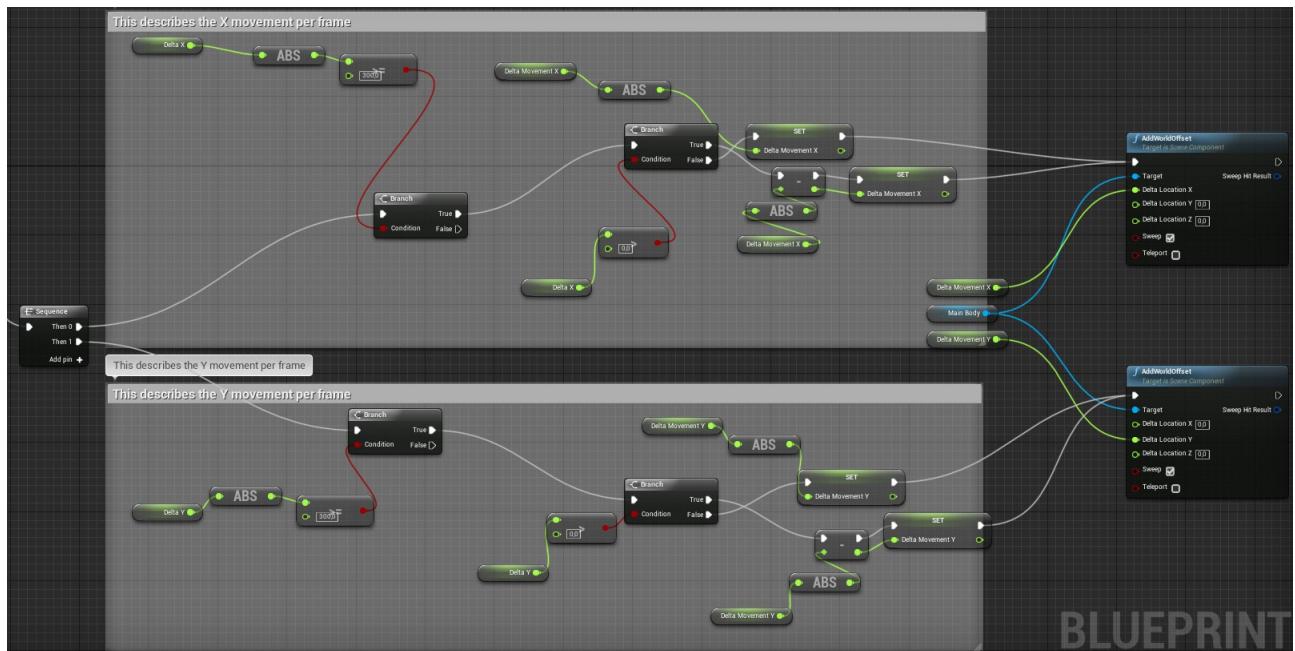
Der Gestapo-Walker besteht funktional aus zwei Komponenten: Die Glasplattform und der rote Bewegungskörper. Bei Berührung mit dem roten Bewegungskörper wird eine Todesteleportation verursacht, somit sollte der Spieler diesen Körper meiden. Der Bewegungskörper ist so ausgelegt, dass dieser dem Spieler verfolgt, aber der Walker ist gleichzeitig so programmiert, dass er unmittelbar vor dem Spieler anhält. Auf der oberen Glasplattform kann der Spieler stehen, somit bietet der Gestapo-Walker auch eine attraktive Möglichkeit zur Spielerbeförderung um gegebenfalls Hindernisse zu überwinden. Die Plattform ist auch mit einem weißen Rahmen versehen, somit sollte der Spieler nicht versehentlich runterfallen können.

Die einzige öffentlich zugängliche Variable ist der SendPlayerToLocation-Vektor mit welchem der Leveledesigner die Weltposition des Spielers für die Todesteleportation festlegen kann. Die Bewegungsgeschwindigkeit des Walkers auf der X- und Y-Achse sind statisch festgelegt und sollte immer geringer sein als die höchste Bewegungsgeschwindigkeit des Spielers. Somit ist gewährleistet, dass der Spieler immer vor dem Walker wegrennen kann.

Aufgrund der Komplexität des Blueprints wurde dieser in zwei Teile aufgespalten. Die Bewegungskondition und Bewegungsabwicklung.



Bewegungskondition: Die Bewegungskondition erklärt, wann wir dem Walker gewähren den Spieler zu verfolgen. Die Kondition ist, dass der Walker nur bis zu einem bestimmten Radius um den Spieler herum sich nähern darf. Für diesen Zweck wird vorher die Distanz zum Spieler ausgerechnet und als DeltaX-Float und DeltaY-Float abgespeichert und diese Distanz wird ständig neu berechnet. Befinden sich die Distanzwerte in X- und Y-Richtung außerhalb des Radius, wird der Walker sich dem Spieler nähern.



Bewegungsabwicklung: Die Bewegungsabwicklung ist aufgeteilt in zwei aufeinanderfolgende Sequenzen. Die Erste beschreibt die Annäherung in X-Richtung, die Zweite beschreibt die Annäherung in Y-Richtung. Der wichtige Aspekt ist hierbei herauszufinden, ob die Distanz negativ oder positiv ist, damit wir ausgehend davon eine positive, beziehungsweise negative Bewegung ermöglichen. Die funktionale Bewegung wird dadurch motiviert die Distanz gegen den Nullpunkt streben zu lassen. In der Praxis haben wir eine quasi-diagonale Bewegung geschaffen.

Im Nachhinein gesehen könnte man sich überlegen die beiden Sequenzen zu vereinen um korrekte, diagonale Bewegungen zu bewerkstelligen. Diese Überlegung ist jedoch für spätere Projekte zu implementieren und in diesem Stadium des Projekts nicht mehr bewerkstelligbar.

UtilityBlueprints zum Gestapo-Walker:

Um die Funktionalität des Gestapo-Walkers innerhalb des Levels zu garantieren, müssen UtilityBlueprints als Akteure ins Level gesetzt werden. Nachfolgend wird über den Gestapo-Trigger und den Gestapo-Teleporter berichtet.

Gestapo-Trigger: Eine Triggerbox wird über einen Bereich innerhalb des Levels gespannt und aktiviert den Walker sobald der Spieler sich innerhalb der Triggerbox befindet. Der Existenzgrund dieses UtilityBlueprints ist, dass wir verhindern möchten das der Walker auch ohne Spieleranwesenheit ständig Bewegungsinputs bekommt und sich möglicherweise in eine ungünstige Position manövriert, ohne das der Spieler seinen Einfluss auf den Walker spürt.

Gestapo-Teleporter: Eine schmale Triggerbox welche den Walker zu einer bestimmten Position innerhalb des Levels teleportiert. Als das Level zu einem früheren Zeitpunkt mehrere Walkers zuließ, passierten viele unvorhergesehene Probleme in Bezug auf Aktor-Referenzierung. Um dieses Problem zu umgehen wird in Levels nur noch mit einem einzigen Walker gearbeitet und somit war die Existenz von einem Gestapo-Teleporter vonnöten.

Gestapo-Crawler: Dritte Generation



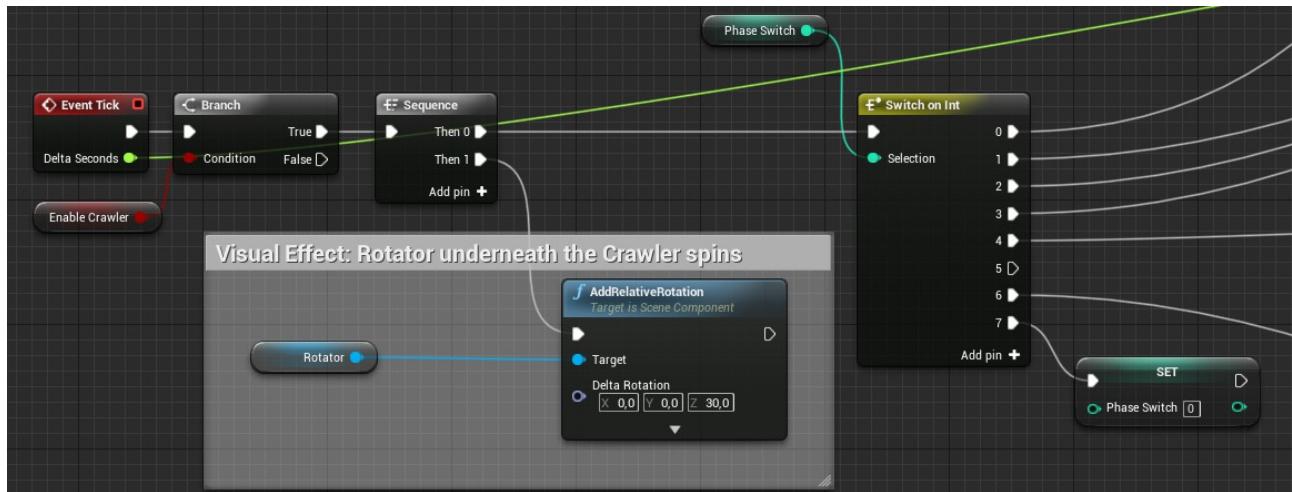
Viele übersetzen den Crawler fälschlicherweise als Krabbler oder Kriecher, aber eigentlich soll der Crawler als Kraulschwimmer bezeichnet werden, welcher unermüdlich seine Strecken mit Sprints absolviert. Der Crawler ist ein Boss und der Spieler findet ihn in LevelP3.

Die vordere Seite des Crawlers ist für den Spieler tödlich und löst eine Todesteleportation aus. Der Rücken des Crawlers ist seine Schwachstelle und diese ist visuell auch auffällig mit einem X markiert. Eine bloße Berührung reicht aber nicht aus um den Crawler zu schaden, der Spieler muss aktiv gegen die Schwachstelle dashen um Schaden zuzufügen. Der Schaden hierbei ist auf genau 20 Lebenspunkte pro Dash gesetzt.

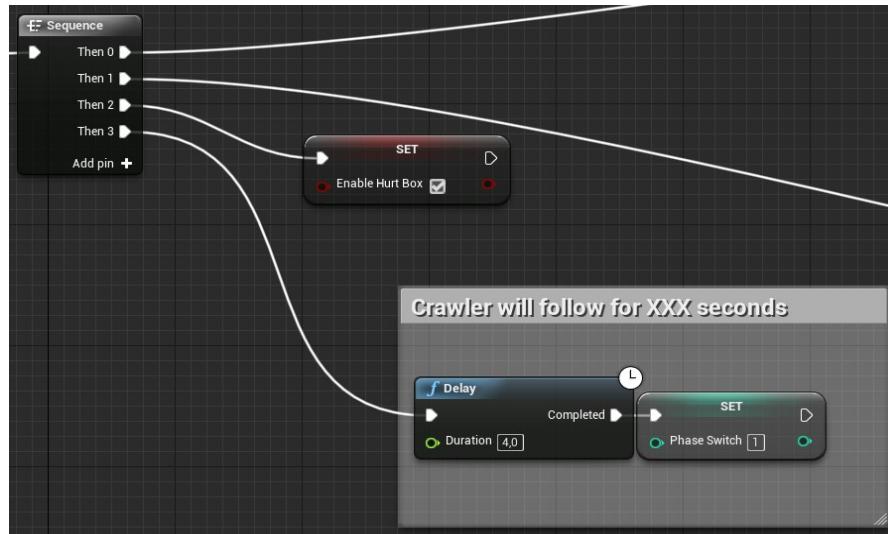
Alle Verhaltensmuster des Crawlers sind statisch in seinen Blueprint verfasst worden und der Leveldesigner hat lediglich einen SendPlayerToLocation-Vektor um die Weltposition des Spielers zu bestimmten, nachdem dieser eine Todesteleportation erlitten hatte. Nachfolgend ist eine Liste aller Variablen aufgezählt, welche im Gestapo-Crawler Blueprint Gebrauch finden:

Name	Typ	Verwendungszweck
SendPlayerToLocation	vector	Position des Spielers nach Todesteleportation
BossOriginalPosition	vector	Ausgangsposition des Bosses im Falle von Glitches
BoomLocation	vector	Der Ort, an dem der Boss gegen eine Wand gesprintet ist
enableCrawler	bool	Schaltet den Crawler an und aus, z.B. wenn seine Lebenspunkte auf 0 fallen
enableHurtBox	bool	Die Fähigkeit des Bosses den Spieler zu töten
boss_HP	int	Aktuelle HP des Crawlers, kann die boss_maxHP nicht übersteigen.
boss_maxHP	int	Festgelegtes Maximum an Lebenspunkten des Bosses
PhaseSwitch	int	Gibt an, in welcher Bossphase sich der Crawler befindet, resultierend in verschiedenen Verhaltensmustern
CurrentLocationToFacePlayer	float	Rotationswert, welcher zum Spieler zeigt
deltaX	float	Distanz von Boss zum Spieler in X-Richtung
deltaY	float	Distanz von Boss zum Spieler in Y-Richtung
deltaMovementX	float	Bewegungsgeschwindigkeit in X-Richtung
deltaMovementY	float	Bewegungsgeschwindigkeit in Y-Richtung

Aufgrund der Komplexität und Zusammenspiel von Crawler mit seinen UtilityBlueprints werden die Einzelheiten seiner inneren Struktur schrittweise erläutert.



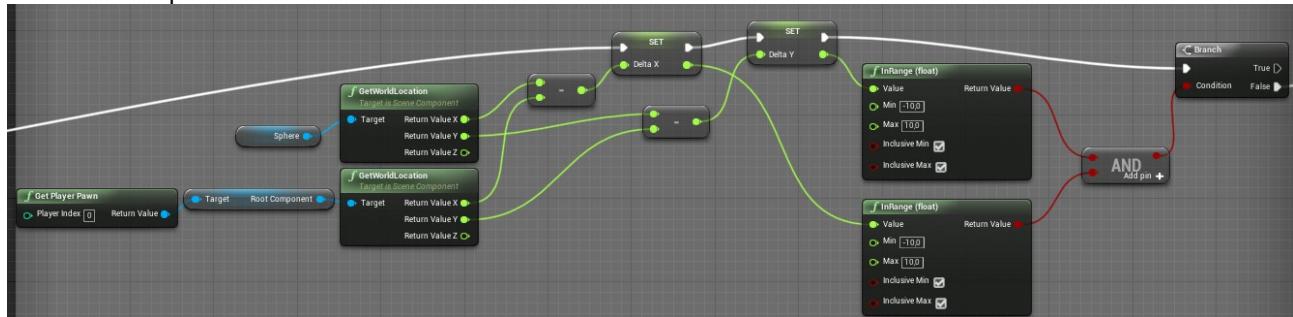
Genauso wie die bereits erklärten Etappen der FrontalAssault-Plate, durchläuft der Crawler auch einen bestimmten Zyklus ab (dargestellt mit dem Integer *Phase-Switch*). Ob diese Phasen überhaupt durchlaufen werden, wird mit dem Boolean enableCrawler aktiviert. Das bedeutet, dass der Crawler und allgemein der gesamte Kampfabschnitt extern aktiviert werden kann, was hier der Fall ist. In der Tat aktiviert das LevelBlueprint den Boss und näheres dazu wird dann in **Kapitel 4.5 LevelP3** geschildert.



Die allererste Etappe (PhaseSwitch=0) wird in 4 Sequenzen abgehandelt. Die erste Teilsequenz ist verantwortlich für die Bewegung. Die zweite Teilsequenz ist verantwortlich für die Rotation des Crawlers, ausgerichtet in Richtung der derzeitigen Spielerposition. Die dritte Teilsequenz schaltet die HurtBox des Crawlers ein, d.h.: die Vorderseite des Crawlers erzeugt bei Berührung mit dem Spieler eine Todesteleportation. Die letzte Sequenz beschreibt die Dauer dieser Etappe (auf dem Bild sind es 4 Sekunden).

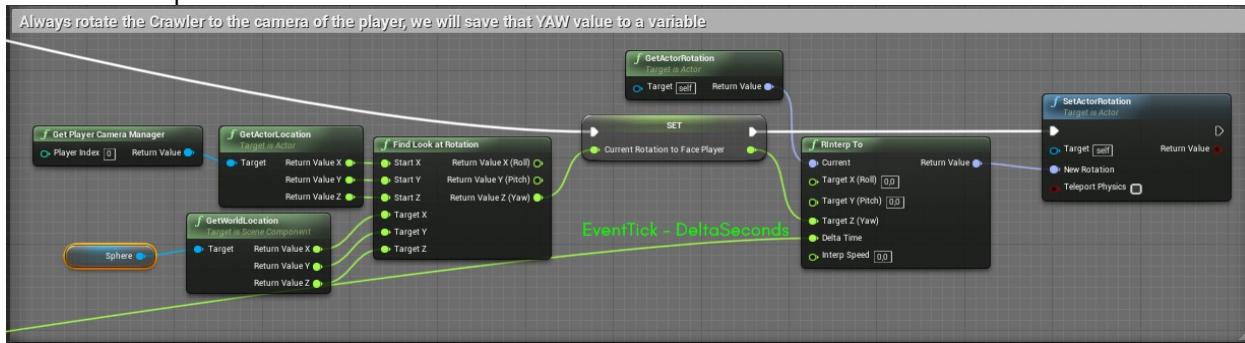
Nachfolgend werden die ersten beiden Teilsequenz erläutert, wobei die erste Teilsequenz im wahrsten Sinne bloß eine angepasste Kopie des Bewegungsverhaltens des Gestapo-Walkers ist.

Erste Teilsequenz:

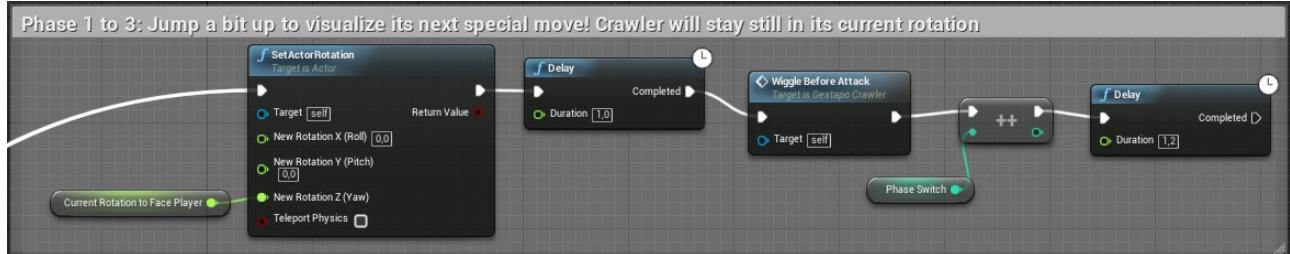


Wie zuvor beim Gestapo-Walker prüft der Crawler ob die Distanz zwischen ihm und dem Spieler innerhalb des akzeptierten Spektrums liegt. Im Falle des Crawlers betrifft diese Distanz ein Spektrum zwischen -10 und 10, was auf jedem Fall eine Todesteleportation motivieren wird. Die Bewegungsabwicklung des Crawlers mithilfe von DeltaMovementX und DeltaMovementY ist bereits beim Gestapo-Walker erläutert worden und bedarf hierbei keine erneute Wiederholung. Jedoch ist anzumerken, dass im Vergleich zum Gestapo-Walker die Bewegungsgeschwindigkeit des Crawlers auf einem deutlich höheren Niveau ist. Der Spieler ist in aktiver Gefahr vom Crawler eingeholt zu werden.

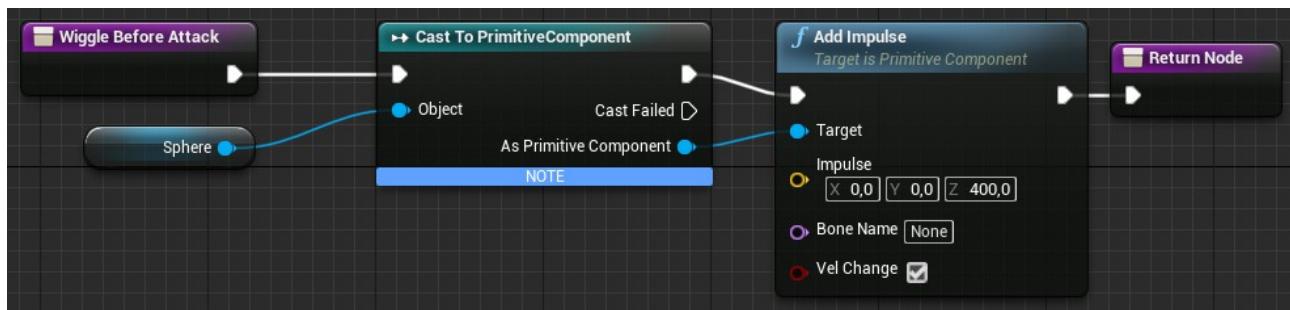
Zweite Teilsequenz:



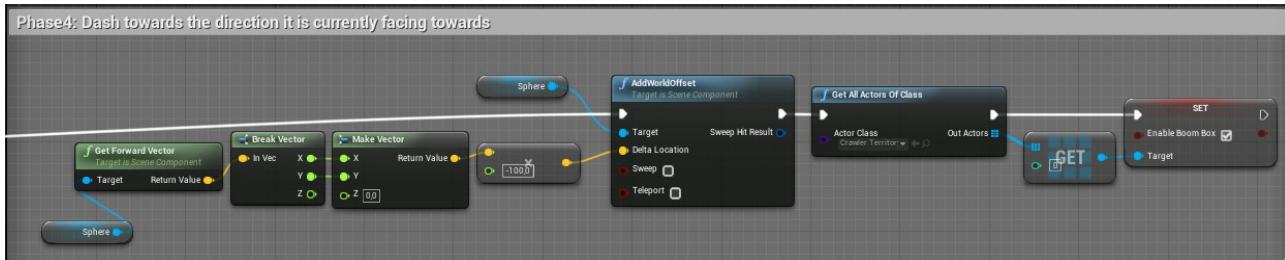
Die zweite Teilsequenz befasst sich mit der natürlichen Z-Rotation des Crawlers, um immer in Richtung des Spielers zu schauen. Wichtig hierbei ist es herauszufinden, wohin der Crawlermesh und auch der Spielermesh derzeit orientiert sind. Die FindLookAtRotation-Node hilft dabei den gewünschten Z-Rotationswert zu erhalten indem wir als StartObjekt die Werte des Crawlers eintragen und beim ZielObjekt die Werte des Spielers. Diesen Z-Rotationswert speichern wir in den currentRotationToFacePlayer-Float ab, denn von dieser Information machen wir noch später Gebrauch. Die RotationInterpolationTo-Node ist die Kernumwandlung von unserem Rotationswert (derzeit als Float) zum gewünschten Rotationswert als funktionalen Rotationswert. Die DeltaTime als Voraussetzung für die sanfte Rotationsbewegung erhalten wir von unserer EventTick-Node.



Die zweite, dritte und vierte Etappe (PhaseSwitch=1, 2, 3) ist eine Wiederholung einer Sprungsimulation des Crawlers. Diese Sprungsimulation soll vom Spieler als visueller Hinweis wahrgenommen werden, damit der Spieler weiß, dass bald eine womöglich gefährliche Attacke des Crawlers bevorsteht. Im Grunde behält der Crawler die gespeicherte Orientierungsrotation während dieser die WiggleBeforeAttack-Funktion auruft.



Die WiggleBeforeAttack-Funktion gibt dem gesamten Körper des Crawlers einen Bewegungsimpuls in positiver Z-Richtung und wird somit durch einen Sprung angetrieben. Durch die eingefügten Delay-Nodes vor und nach dem Aufruf dieser Funktion werden die simulierten Sprünge abgewartet.

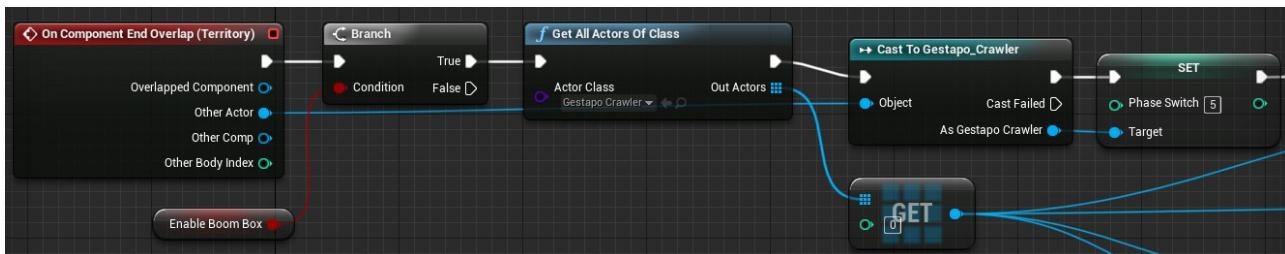


Die fünfte Etappe (PhaseSwitch=4) ist eine rasend schnelle Vorwärtsbewegung in die Richtung, in welcher der Crawler derzeit schaut. Gleichzeitig wird der enableBoomBox Boolean der Crawler-Territory aktiviert, ein UtilityBlueprint des Crawlers. Diese wird noch anschließend erklärt. Wichtig ist, dass die Bewegungsgeschwindigkeit ungemein hoch ist damit es so wirkt, als wäre der Crawler wie auch der Spieler in der Lage einen Dash auszulösen.

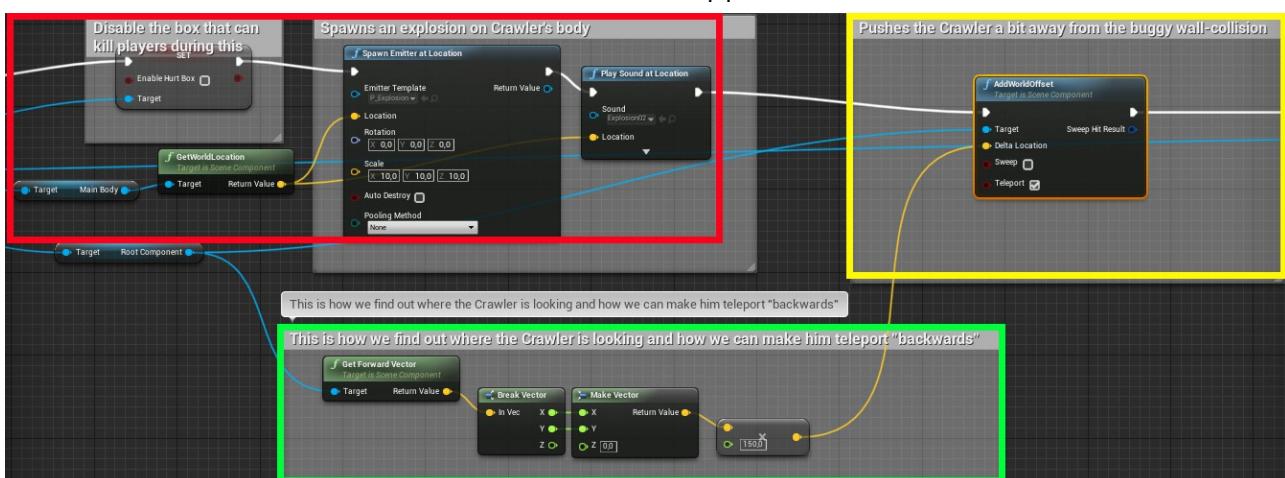
Durch die GetForwardVector-Funktion bekommen wir den Vorwärtsvektor, welchen wir mit Ausnahme des Z-Vektors statisch um einen Wert erhöhen. Somit bewegt sich der Crawler in die Blickrichtung ohne sich überhaupt Gedanken um den Rotatorwert zu machen.

Wichtig ist, dass der PhaseSwitch nicht inkrementiert wird. Faktisch bedeutet es, dass der Crawler unendlich lang in diese Richtung sprinten wird wenn wir nichts dagegen unternehmen. Aus diesem Grund führen wir nun den UtilityBlueprint Crawler-Territory ein.

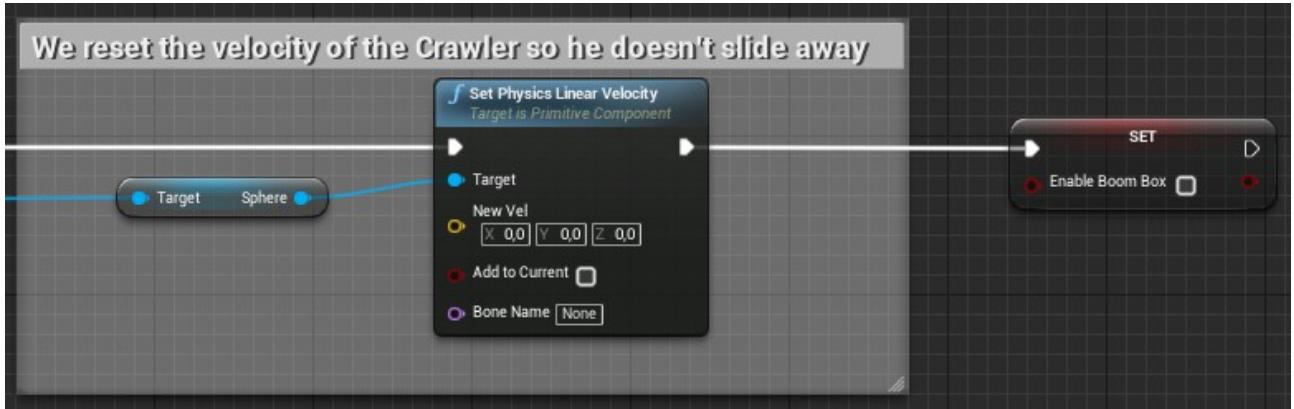
UtilityBlueprint zum Gestapo-Crawler: Crawler-Territory



Die Crawler-Territory ist eine Triggerbox, welche um die Kampfarena gespannt ist und idealerweise am Rande der Arena endet. Ein EndOverlap-Event kümmert sich um den Fall, wenn der Crawler den Rand der Arena berührt (hier: durch seinen Sprint). Das Kollisionseignis wird zudem auch mit seinem Sprint aktiviert (siehe: enableBoomBox) und der PhaseSwitch wird inkrementiert um die nächste Etappe einzuleiten.



Um das Verständnis zu erleichtern, wird die obige Blueprint-Sektion in Abschnitte unterteilt. Der **rote Kasten** wird zuerst die HurtBox des Crawlers deaktivieren (somit kann der Spieler gefahrlos den roten Körper des Crawlers berühren ohne eine Todesteleportation zu erleiden) und danach wird eine Explosionsanimation mit passendem Soundeffekt am Körper des Crawlers abgespielt. Der **grüne Kasten** berechnet die Blickrichtung des Crawlers und addiert zum Wert noch statisch eine Zahl um eine gewisse Position hinter dem Crawler als Vektor zu speichern. Der **gelbe Kasten** nutzt diesen Wert um eine Rückwärtstranslation einzuleiten und teleportiert den Crawler genau da hin. Sinn und Zweck ist, dass wir nicht wollen, dass der Crawler eventuell durch seinen Sprint in der Wand stecken bleibt.



Der letzte Abschnitt in Crawler-Territory setzt sämtliche Bewegungsvektoren auf null um ein eventuelles Schlittern nach der Kollision zu verhindern und deaktiviert die BoomBox. Im Nachhinein hätte man vielleicht mit einer DoOnce-Node arbeiten können, funktional sind aber beide Varianten gleichwertig.

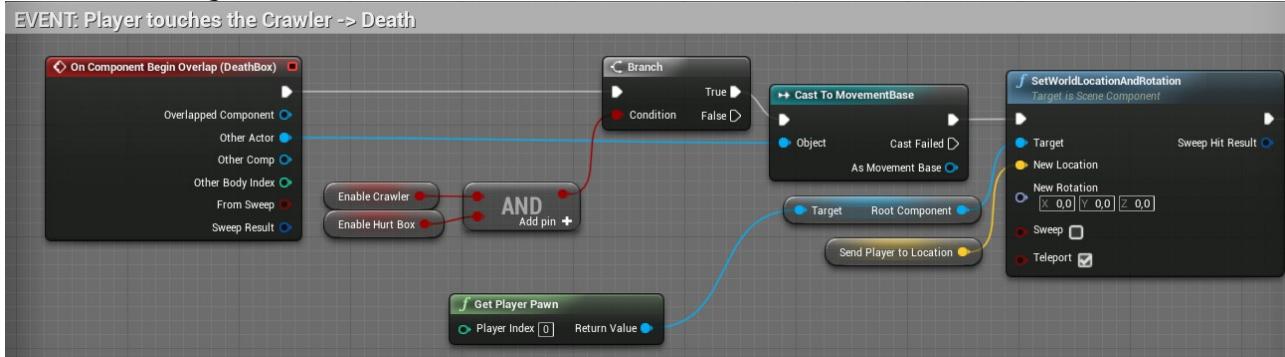


Die sechste Etappe (PhaseSwitch=5) ist zugleich auch die letzte Etappe ehe der Zyklus wiederholt wird. Da aufgrund der Kollision des Crawlers mit der Außenwand vieles passieren kann, muss via SetActorRotation-Node dafür gesorgt werden, dass der Crawler aufrecht zur Wand ausgerichtet ist in welcher er gerade gesprintet ist, egal was bei der Kollision auch geschah. In dieser Position und Rotation verweilt der Crawler für ganze 5 Sekunden.

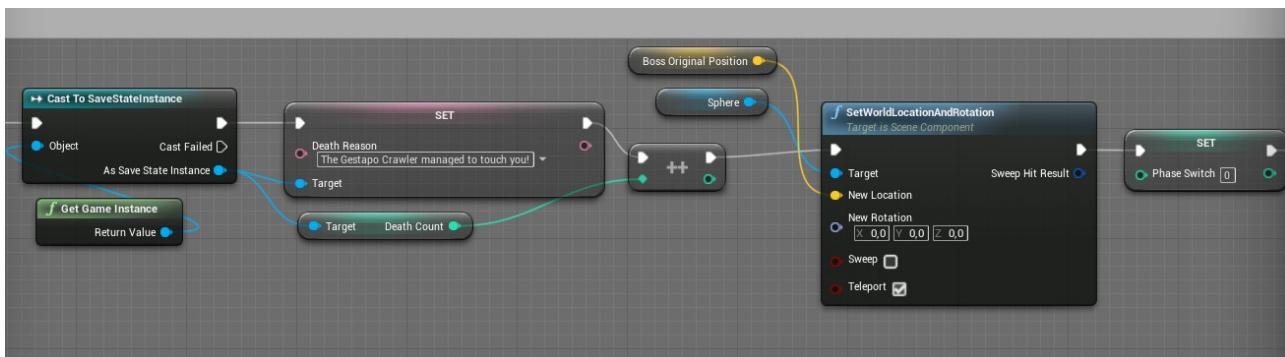
Was der Spieler im Grunde vor sich hat ist ein bewegungsunfähiger Crawler mit entblößter Schwachstelle. Die fünf Sekunden sollten genug sein, um eine Vektorkraft des Spielers genügend aufzuladen und gegen die Schwachstelle einzusetzen.

Kollisionereignis mit der HurtBox des Crawlers:

EVENT: Player touches the Crawler -> Death



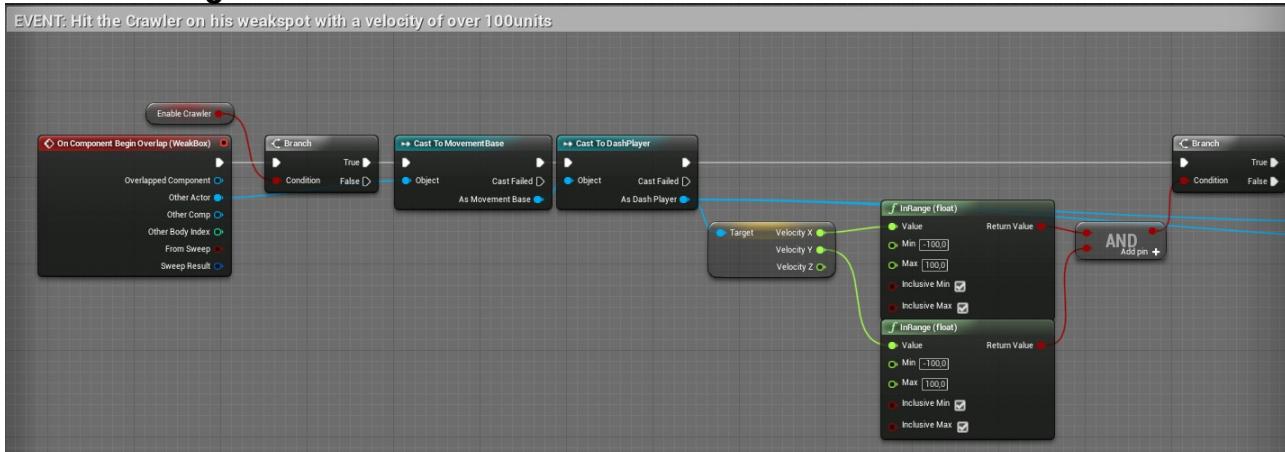
Das Kollisionereignis ist in zwei Abschnitte unterteilt. Das obige Bild zeigt den ersten Abschnitt. Diese Kollision wird aktiviert wenn sowohl der Crawler selbst aktiviert ist (enableCrawler) und auch die HurtBox (enableHurtBox). Die daraufhin erfolgte Teleportation befördert den Spieler zu einer vom Leveldesigner ausgewählten Position auf der Welt. Darunter folgt der zweite Abschnitt.



Die Todesursache und der Todeszähler wird durch dieses Ereignis bereichert und der Boss teleportiert sich selbst zurück zu seiner Ausgangsposition mithilfe des BossOriginalPosition-Vektors. Zu guter Letzt wird der PhaseSwitch auf null gesetzt um das Verhalten zurückzusetzen.

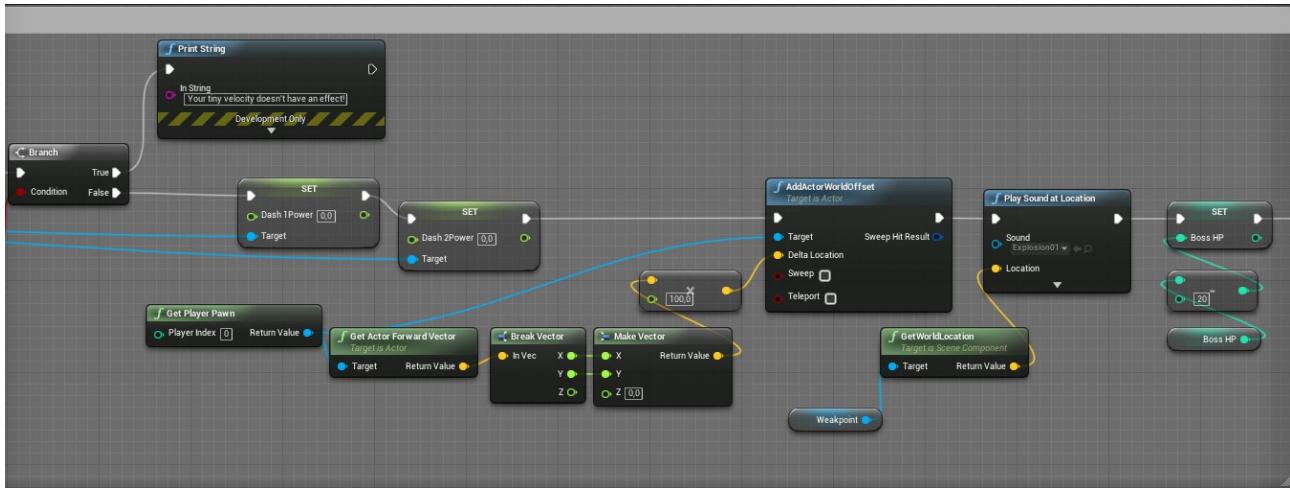
Kollisionereignis mit der Schwachstelle des Crawlers:

EVENT: Hit the Crawler on his weakspot with a velocity of over 100units



Das Kollisionereignis der Schwachstelle ist unterteilt in 3 Abschnitte. Im ersten Abschnitt kann man sehen, dass das WeakBox-Kollisionsevent auslösbar ist sobald der Crawler aktiviert ist. Dann wird geprüft ob die Geschwindigkeit des Spielers in einem bestimmten Rahmen ist.

Das ist immer der Fall bei normaler Laufgeschwindigkeit oder die Nutzung von geringer Vektorkraft. Wenn dies nämlich nicht zutrifft, dann bedeutet es, dass der Spieler gerade seine Vektorkraft verwendet hatte und genau solch einen Fall filtern wir heraus.

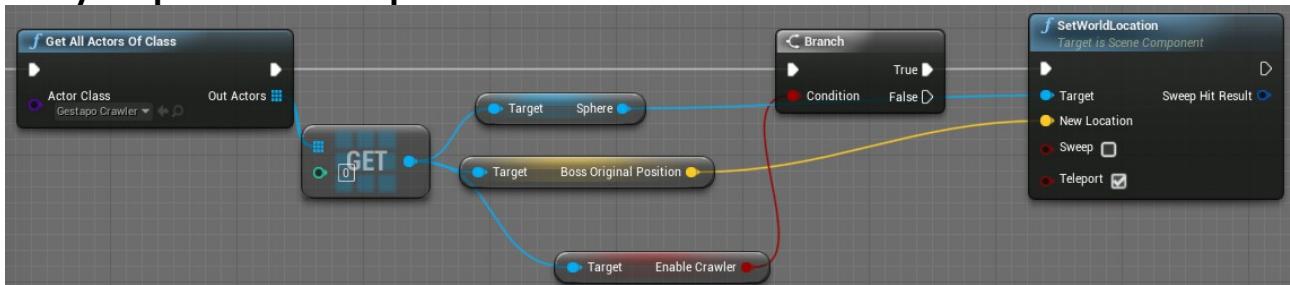


Wenn die Branch-Node den False-Strang entlangläuft, dann haben wir eine Kollision, welche durch Verwendung einer Vektorkraft zustande gekommen ist. Die Vektorkräfte werden beide auf null gesetzt, damit der Spieler nicht motiviert wird, beide Vektorkräfte einzusetzen für einen verkürzten Bosskampf. Danach setzen wir den Spieler für eine bestimmte Distanz hinter sich damit die Kollision keine Nebenwirkungen hervorruft. Es wird ein Explosions-Soundeffekt gespielt und die Lebenspunkte des Bosses werden um 20 reduziert.



Der letzte Abschnitt prüft ob die Kollision den Tod des Crawlers verursacht hat. Falls ja, dann wird eine weitere Explosion ohne Soundeffekt abgespielt und der Crawler wird deaktiviert. Somit erhalten wir einen Bossmesh ohne Befehlsimpetus. Durch die fehlende Restabilisierung fällt der Boss auch leicht um und ist bewegungsunfähig. Der Checkpoint Counter wird erhöht damit das Level weitergeht und Arenabarrieren verschwinden. Mehr dazu im Leveldesign-Kapitel.

UtilityBlueprint zum Gestapo-Crawler: Crawler-AntiOoB-Protection



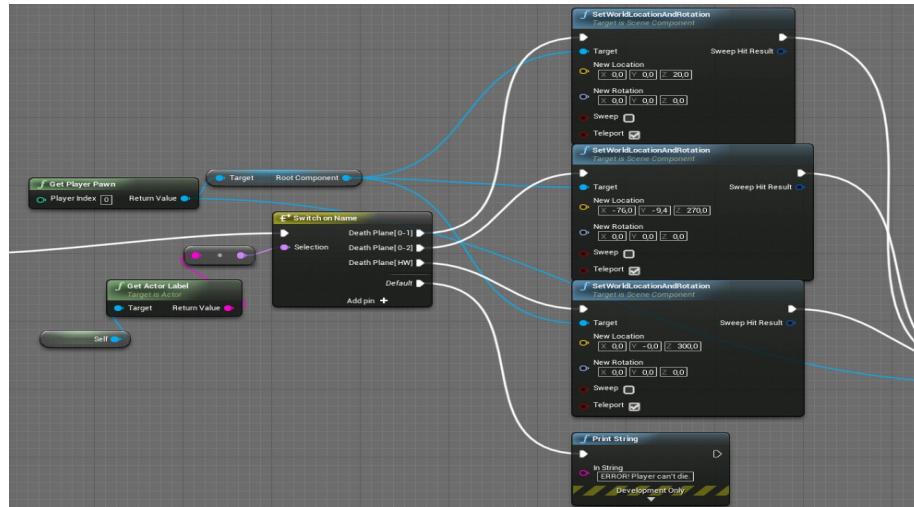
Das obige Bild zeigt die Funktionalität eines weiteren UtilityBlueprints des Gestapo-Crawlers, welches darauf ausgelegt ist, dass der Crawler sich nicht irgendwie aus seiner Arena herausbewegen kann. Solche sogenannten Out-of-Bound-Problematiken sind sehr üblich sobald man separat agierende Akteure im Level hat. Wäre der Crawler nämlich aus irgendeinem Grund außer Reichweite des Spielers, wäre das Level unspielbar und aus diesem Grund fatal für dieses Projekt. Wie auch die Crawler-Territory, wird dieses obige Event nur ausgelöst, sobald der Crawler eine bestimmte Triggerbox verlässt. Diese Triggerbox ist zudem größer als die der Crawler-Territory um einen gewissen Spielraum zu gewähren.

Die BossOriginalPosition-Variable des Crawlers wird verwendet, um eine Referenz der Ausgangsposition des Crawlers zu erhalten, zu welchem dieser nun teleportiert wird.

2.4.2 Kollisionsereignisse

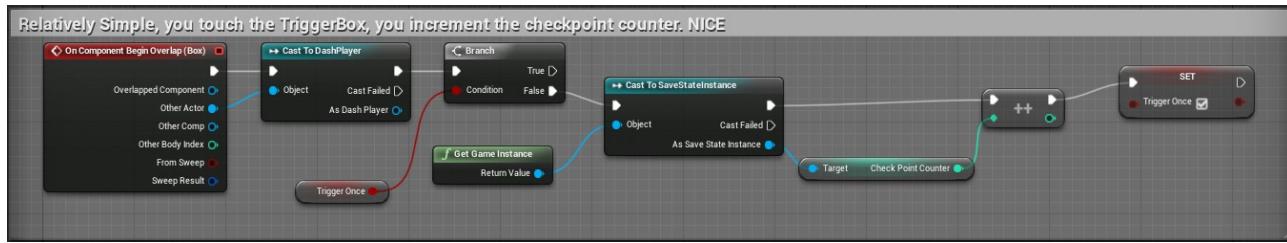
Im Laufe dieser technischen Dokumentation wurden schon einige Kollisionsereignisse aufgezählt, darunter die UtilityBlueprints von Gestapo-Walker und Gestapo-Crawler aber nun werden wir auch noch andere sehr wichtige Blueprint erläutern, welche Kollisionsereignisse induzieren.

DeathPlane



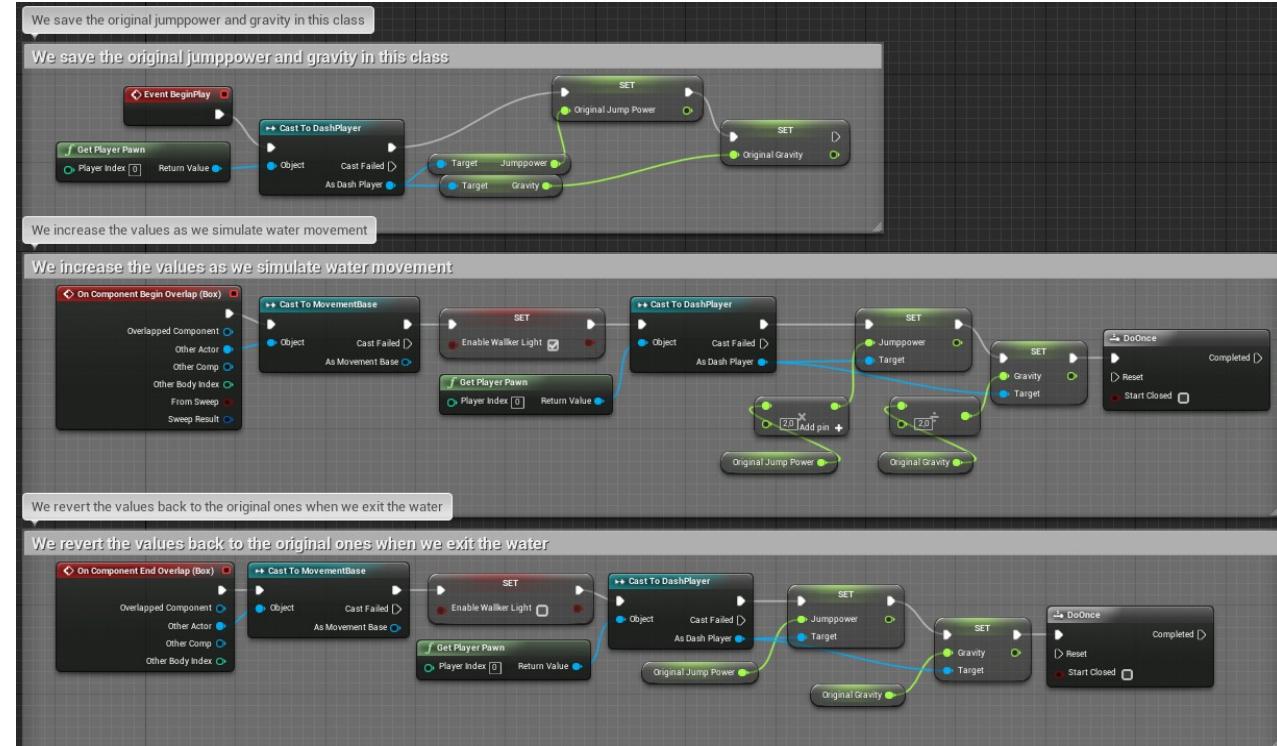
Die DeathPlane wird in allen Tutorial-Levels aber auch in der HubWorld eingesetzt und ist im Grunde eine Fail-Safe-Mechanik falls der Spieler es schafft sich durch das Level zu glitchen. Die Triggerbox ist aus diesem Grund weit unterhalb der Levelstruktur zu finden. Die Besonderheit an dieser DeathPlane ist die Tatsache, dass dieselbe Blueprint für alle drei Levels eingesetzt wird und individuelle Teleportpunkte anbieten. Diese Unterscheidung bildet sich durch den Labelnamen der DeathPlane innerhalb des Levels. In der Blueprint fragt sich das Objekt nämlich wie es heißt und durch diesen SwitchCase-Node können wir dann die möglichen Ergebnisse in separate Spalten. Eine weitere Möglichkeit wäre durch eine Funktion herauszufinden wie das Level heißt auf welchem sich der Spieler derzeit befindet aber diese Art der Lösung wäre eventuell um einiges schwieriger herauszufinden.

Checkpoint



Die Checkpoint-Kollision ist ziemlich simpel. Sie lässt sich nur einmal aktivieren und inkrementiert den CheckpointCounter, welches sich in der SaveStateInstance befindet.

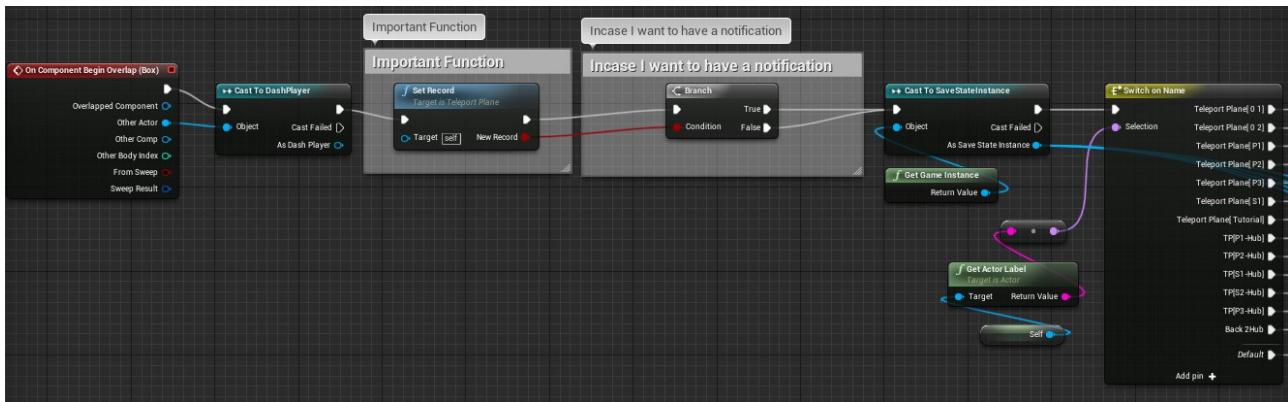
WallkerLight



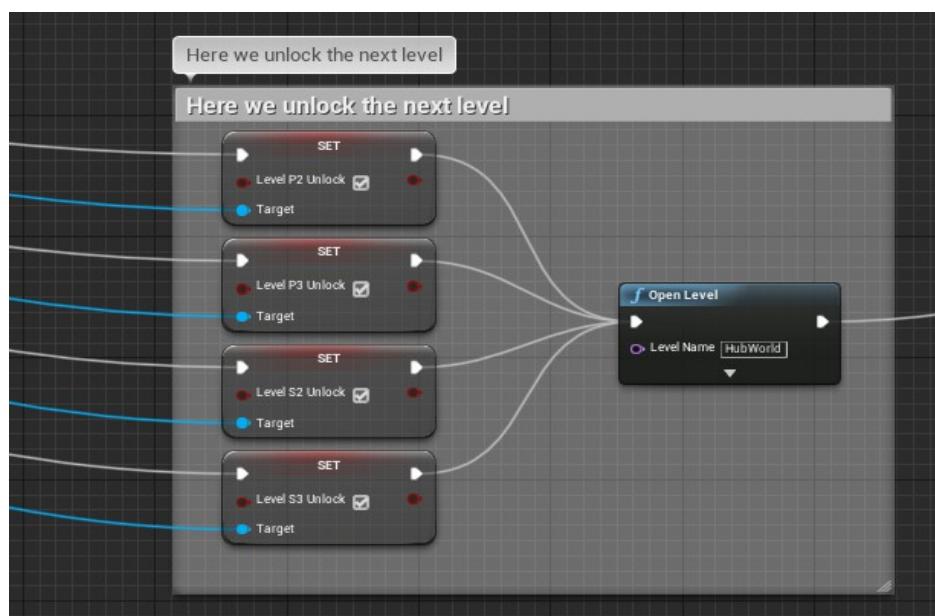
WallkerLight ist eine große Triggerbox, welche der Spieler in LevelP3 wiederfindet. Sie soll die Charakterwerte so anpassen, sodass der Spieler höher und länger springen kann und simuliert dadurch die Wasserphysik. Ebenso schaltet es den WallkerLight ein, was die am Kopf befestigte Scheinwerfer aktiviert und die Stelle vor dem Spieler erleuchtet.

In der Initialisierung speichern wir die Ausgangswerte wie Sprunghöhe und Gravitation ab. Falls der Spieler ins Ozean springt, wird durch Kollisionereignis seine Sprunghöhe und Gravitation angepasst und bei Verlassen der Gewässer werden diese Werte zurückgesetzt.

TeleportPlane



Die TeleportPlane ist sehr wichtig für unser Projekt denn dieses Kollisionsereignis sorgt für den Levelwechsel. Die TeleportPlane wird in jedem Level verwendet und hat individuelles Verhalten basierend auf den Namen der TeleportPlane innerhalb des Levels (also genauso wie die DeathPlane). Die SetRecord-Funktion vergleicht den jetzigen Zeitrekord mit der derzeitig aufgenommenen Zeit. Wie der Timer in diesem Projekt implementiert wurde, wird im **Kapitel 3.4 Timer und Zeitrekorde** näher erläutert.

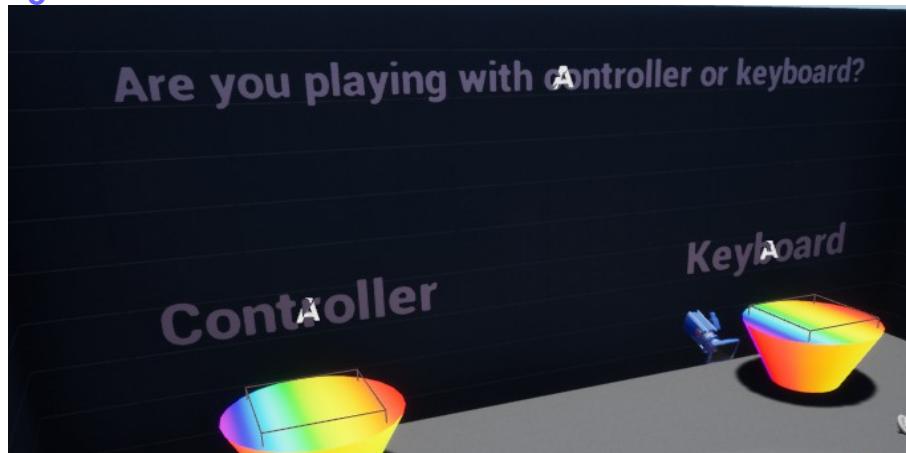


Wichtig für die platzierten TeleportPlanes sind die, welche am Ende des Levels platziert wurden. Das Auslösen dieser Teleportation aktiviert bestimmte Booleans innerhalb der SaveStateInstance und somit spielt der Spieler die nachfolgenden Levels frei. Ein Level-Progression-System wurde geschaffen.

2.4.3 Sonstige Levelobjekte

Sonstige Levelobjekte sind welche, die im Level platziert wurden, jedoch spielunterstützend arbeiten. Solche Levelobjekte geben dem Spieler Output, bewegen den Spieler oder können Einstellungen ändern.

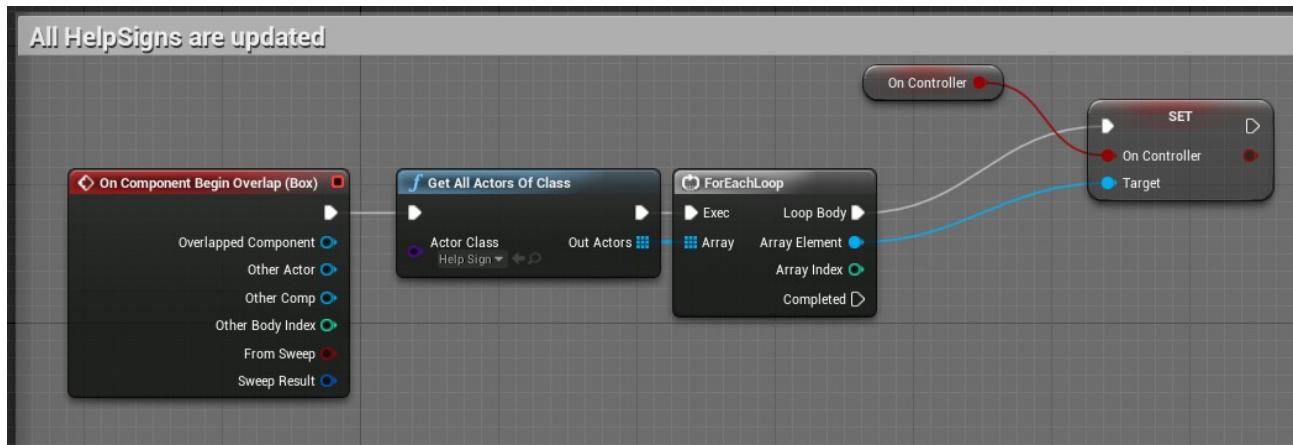
ChooseHelpSignControls



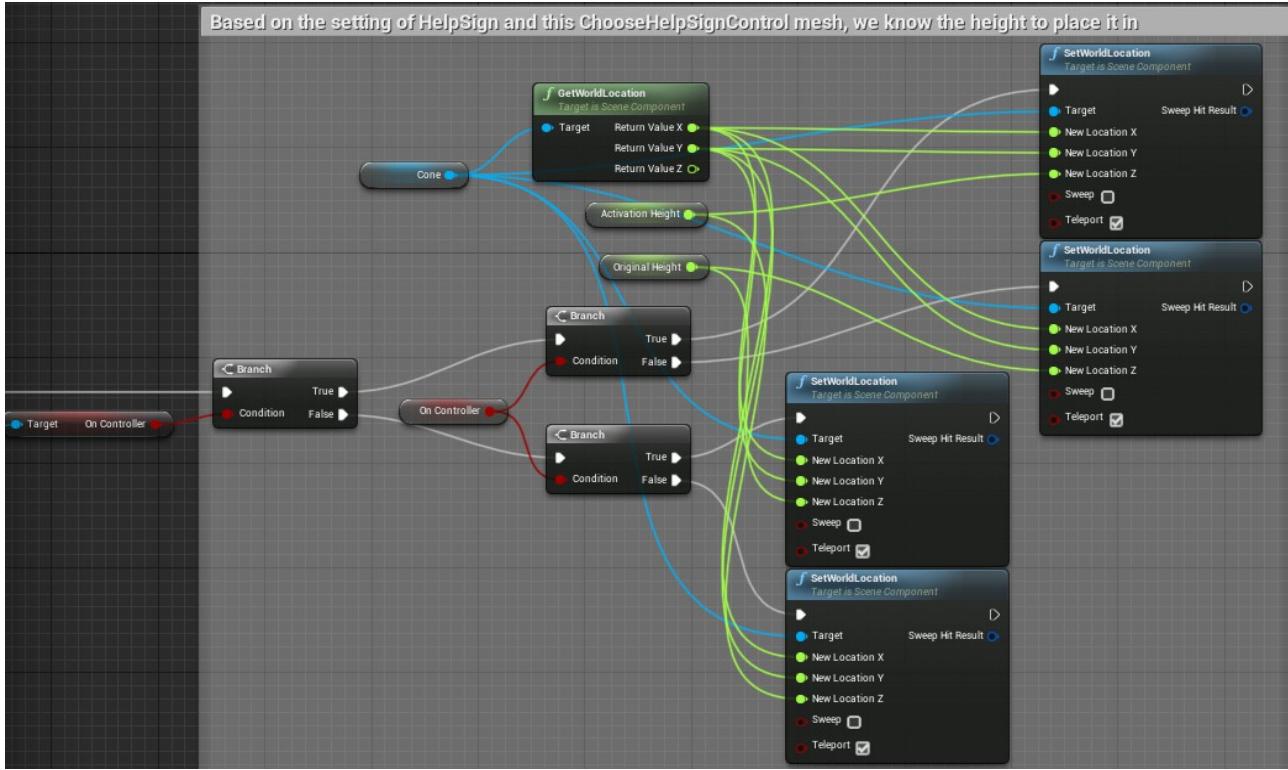
Diese umgedrehten Kegel in Regenbogenfarbe schalten per Kollisionsereignis einen Schalter um. Dieser Schalter regelt die Hilfanzeigen zwischen Kontroller und Tastatur. Dieser Schalter ist eine Boolean Variable im HelpSign-Blueprint. Nachfolgend sind die Variablen innerhalb der ChooseHelpSignControls aufgelistet:

Name	Typ	Öffentlich?	Verwendungszweck
OnController	bool	Ja	Entscheidet ob das Kollisionsereignis den Schalter auf Kontroller schaltet oder nicht (AKA: Tastatur)
OriginalHeight	float	Ja	Die oberen Höhenposition bei Deaktivierung
ActivationHeight	float	Ja	Die untere Höhenposition bei Aktivierung

Die Funktionsweise ist simpel: Man nehme zwei dieser Kegel und setzt bei einem der beiden Kegel den öffentlichen OnController Boolean auf true. Wenn true, dann wird die Höhe des Kegels auf die ActivationHeight Höhe gesetzt. Die Aktivierung des anderen Kegels wird den ersteren Kegel wieder deaktivieren.



Hier sieht man wie jeder HelpSign Aktor im Level gleichzeitig umgeschalten wird falls das Kollisionsereignis eines Kegels ausgelöst wurde. Die HelpSign hat ebenfalls einen OnController Boolean.

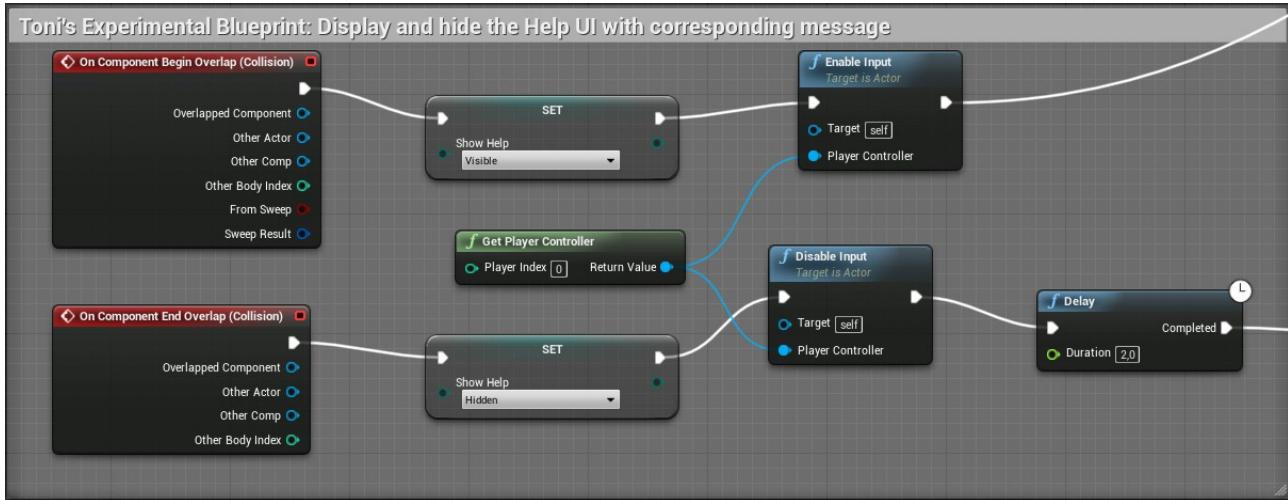


Im EventTick-Node prüft der Kegel den derzeitigen Schalter in einem beliebigen HelpSign und dann seine eigene Schalter-Polarisierung. Durch ein Geflecht aus drei Branch-Nodes können wir jede Situation abdecken und erhalten vier verschiedene Situationen und können anhand dessen bestimmten ob der jeweilige Schalter nun aktiviert sein soll (also dessen Höhe der ActivationHeight entsprechen soll) oder deaktiviert (daher: die Höhe von OriginalHeight).

HelpSign

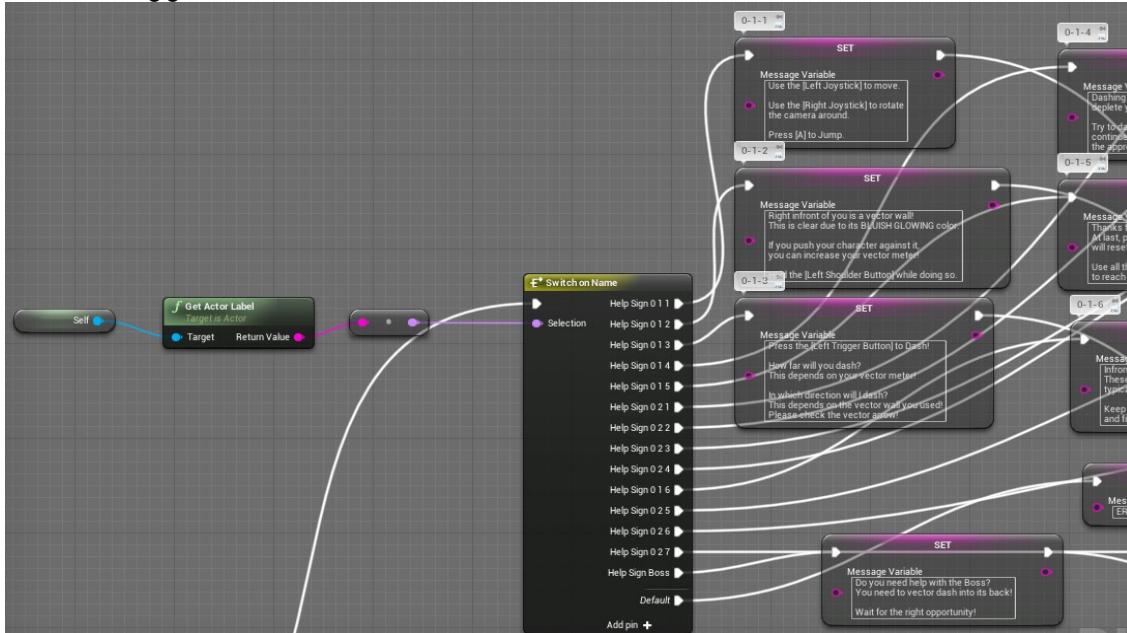


Der HelpSign ist ein Mesh, welches um sich herum eine Triggerbox besitzt. Dringt der Spieler in diese Triggerbox hinein, so wird ein UI Fenster in der Mitte des Bildschirms mitsamt eines Hilfetextes angezeigt. Was der Inhalt dieses Hilfetextes ist, wird durch zwei Faktoren bestimmt. Der erste Faktor ist der Name des Aktors innerhalb des Levels. Es nutzt den Namenslabel als Variable in einer SwitchCase. Der zweite Faktor ist der Schalter den man mittels der regebogenfarbenen Kegel umlegt: Kontroller oder Tastatur.

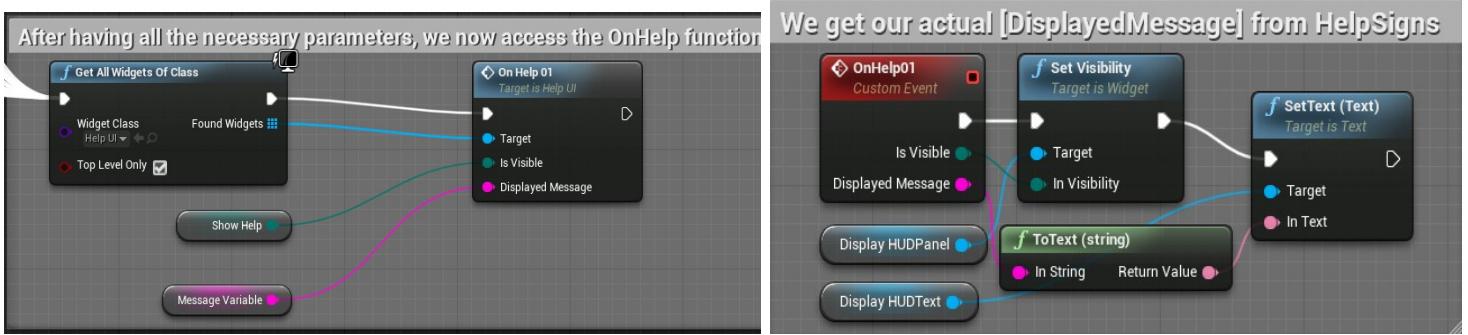


Name	Typ	Verwendungszweck
OnController	bool	Entscheidet den Hilfetext (ob Kontroller oder Tastatur)
showHelp	Eslate Visibility	Variable um Hilfetext anzuzeigen und auszublenden
messageVariable	String	Der Hilfetext zum Ausgeben

Im Grunde wird der Hilfetext angezeigt sobald der Spieler die Triggerbox betritt. Nach dem Verlassen der Triggerbox bleibt der Hilfetext noch für zwei Sekunden bevor es verschwindet.



Es ist nicht wichtig die gesamte Sektion anschaubar zu haben, denn ein Blick genügt um die Funktionsweise zu erkennen. Nach einer Branch-Node um Kontroller und Tastatur in eigene Stränge zu teilen, wird hier auf den Namenslabel geprüft und die messageVariable String wird dementsprechend gesetzt.



Links sehen wir, wie wir das selbsterstellte OnHelp01-Event generieren und den Sichtbarkeitsparameter und Hilfetext mitgeben. Das generierte Event findet sich im Help_UI und liefert den schon vorprogrammierten Outputkasten um den Text auszulesen und anzuzeigen.

3. Logische Komponente

3.1 Tastenbelegung

3.1.1 Kontrollschema Kontroller (XBOX One Wireless Black)



Input	Aktion	Beschreibung
Linker Analog-Stick	Bewegung	Ermöglicht die klassische Bewegungsmöglichkeit.
Rechter Analog-Stick	Kamera	Bewegt die Kamera um den Spieler herum.
[A]-Knopf	Springen	Der klassische Sprungknopf für Jump-and-Runs oder ähnliche Spiele.
[B]-Knopf	Gespeicherte Vektorkraft zurücksetzen	Durch drücken von [B] wird der aktuell ausgewählte Vektor gelöscht. Sollten beide Vektoren ausgewählt sein, werden beide gelöscht, ist keiner ausgewählt, dann wird keiner gelöscht.
[X]-Knopf	Nichts	Keine Funktion
[Y]-Knopf	Nichts	Keine Funktion
[LB]	Alpha Dash selektieren	Vektor 1 wird ausgewählt. Nur während der Vektor ausgewählt ist, kann er aufgeladen oder gelöscht werden. Sollte der Vektor voll sein, kann dieser seine Richtung nicht mehr verändern, bis er geleert oder gelöscht wird.
[LT]	Alpha Dash nutzen	Die Vektorkraft von Vektor 1 wird freigesetzt, und somit wird der Spielcharakter mit der angesammelten Stärke in die Richtung des Vektors geschleudert.

[RB]	Beta Dash selektieren	Vektor 2 wird ausgewählt. Nur während der Vektor ausgewählt ist, kann er aufgeladen oder gelöscht werden. Sollte der Vektor voll sein, kann dieser seine Richtung nicht mehr verändern, bis er geleert oder gelöscht wird.
[RT]	Beta Dash nutzen	Die Vektorkraft von Vektor 2 wird freigesetzt, und somit wird der Spielcharakter mit der angesammelten Stärke in die Richtung des Vektors geschleudert.
[LT] + [RT]	Combine Dash nutzen	Die beiden Vektoren werden kombiniert, also werden die Richtungen miteinander verrechnet, und sofort freigesetzt. Zeigen die Vektoren in entgegengesetzte Richtungen, so steht man auf der Stelle bzw. Steht man kurzzeitig in der Luft, bis die Kraft aufgebraucht ist.
[LB] + [RB]	Beide Dashes selektieren	Hält man sowohl [LB] als auch [RB], so kann man beide Vektoren gleichzeitig auswählen. Dies ermöglicht simultanes Aufladen beider Vektoren sowie simultanes Löschen.

3.1.2 Kontrollschema Tastatur (RAZER Cynosa Chroma & Abyssus Gaming Mouse)

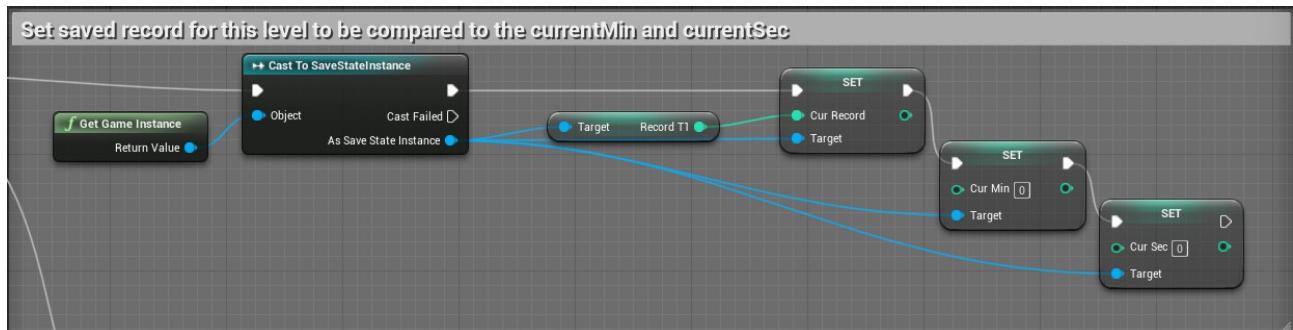


Input	Aktion	Beschreibung
[WASD]-Tasten	Bewegung	Ermöglicht die klassische Bewegungsmöglichkeit.
Maus-Bewegung	Kamera	Bewegt die Kamera um den Spieler herum.
Leertaste	Springen	Der klassische Sprungknopf für Jump-and-Runs oder ähnliche Spiele.
[R]-Taste	Gespeicherte Vektorkraft zurücksetzen	Durch das Drücken der Tasturtaste [R] wird der aktuell ausgewählte Vektor gelöscht. Sollten beide Vektoren ausgewählt sein, werden beide gelöscht, ist keiner ausgewählt, dann wird keiner gelöscht.

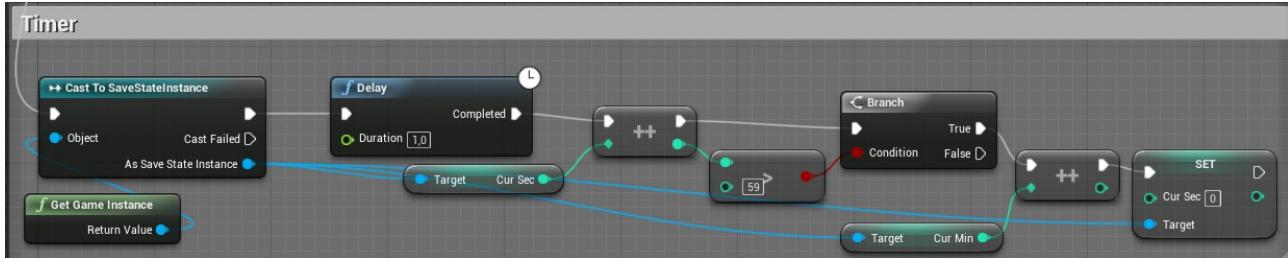
[1]-Taste	Alpha Dash selektieren	Vektor 1 wird ausgewählt. Nur während der Vektor ausgewählt ist, kann er aufgeladen oder gelöscht werden. Sollte der Vektor voll sein, kann dieser seine Richtung nicht mehr verändern, bis er geleert oder gelöscht wird.
Linke Maustaste	Alpha Dash nutzen	Die Vektorkraft von Vektor 1 wird freigesetzt, und somit wird der Spielcharakter mit der angesammelten Stärke in die Richtung des Vektors geschleudert.
[2]-Taste	Beta Dash selektieren	Vektor 2 wird ausgewählt. Nur während der Vektor ausgewählt ist, kann er aufgeladen oder gelöscht werden. Sollte der Vektor voll sein, kann dieser seine Richtung nicht mehr verändern, bis er geleert oder gelöscht wird.
Rechte Maustaste	Beta Dash nutzen	Die Vektorkraft von Vektor 2 wird freigesetzt, und somit wird der Spielcharakter mit der angesammelten Stärke in die Richtung des Vektors geschleudert.
Linke und rechte Maustaste gleichzeitig	Combine Dash nutzen	Die beiden Vektoren werden kombiniert, also werden die Richtungen miteinander verrechnet, und sofort freigesetzt. Zeigen die Vektoren in entgegengesetzte Richtungen, so steht man auf der Stelle bzw. Steht man kurzzeitig in der Luft, bis die Kraft aufgebraucht ist.
[1] + [2]	Beide Dashes selektieren	Hält man sowohl [1] als auch [2], so kann man beide Vektoren gleichzeitig auswählen. Dies ermöglicht simultanes Aufladen beider Vektoren sowie simultanes Löschen.

3.2 Timer und Zeitrekorde

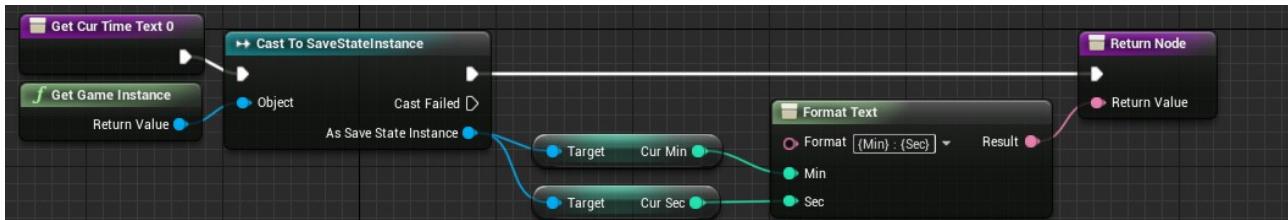
Für die Implementierung eines Leveltimers und das Festhalten von Levelrekorden bedarf es einen relativ großen Aufwand im LevelBlueprint und eine Anzahl an Variablen im SaveStateInstance. Für den Rekordvergleich sorgt eine Funktion innerhalb der TeleportPlane denn das Kollisionereignis beendet zeitlich auch das Level womit es kontextuell schon stimmig ist das der Timer dort stoppt und verglichen wird.



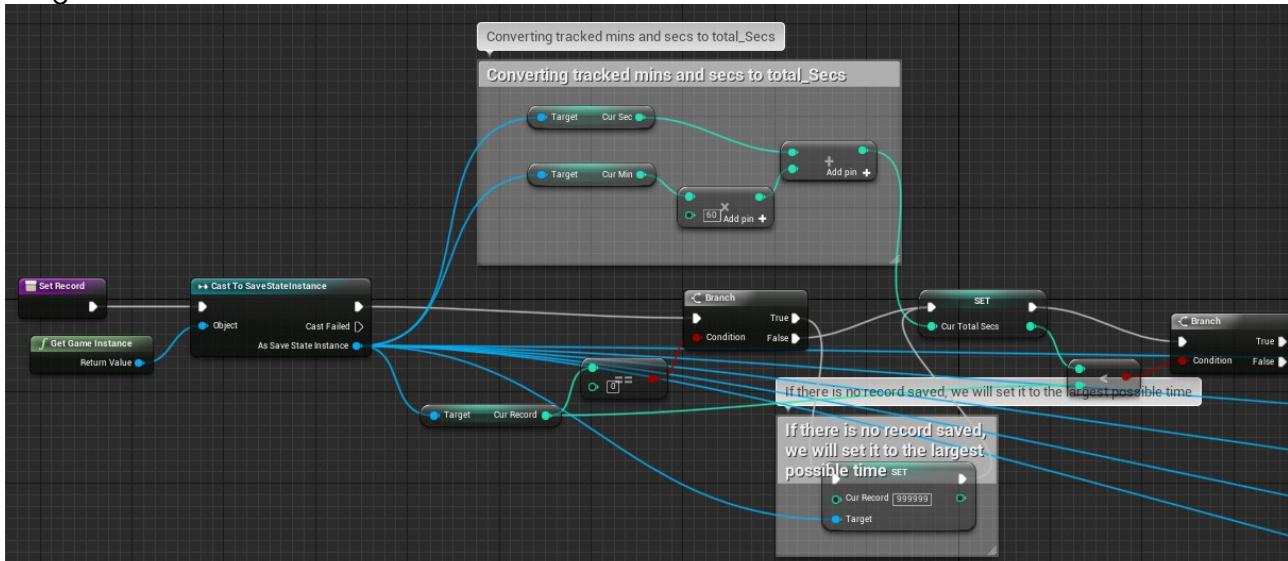
Für jeden LevelBlueprint, welches den Timer implementiert, ist ein zweiteiliger Aufwand nötig. Der erstere Aufwand ist im BeginPlay-Node zu finden und wählt den derzeitigen Levelrekord (hier: RecordT1) als zu vergleichenden Rekord (hier: curRecord) aus. Dann werden die Timer zurückgesetzt. Das Level kann somit beginnen!



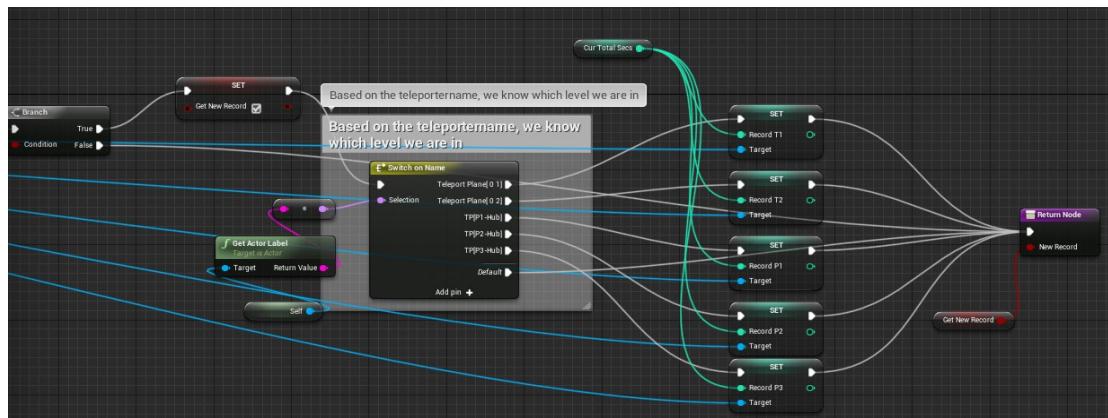
Der zweite Aufwand findet sich im EventTick-Node. Dieser Block inkrementiert sekundenweise den curSec Integer und im Minutenrhythmus den curMin Integer.



Innerhalb der Custom_UI ist eine Funktion, welche die beiden Integer Werte abruft und visuell dargestellt.



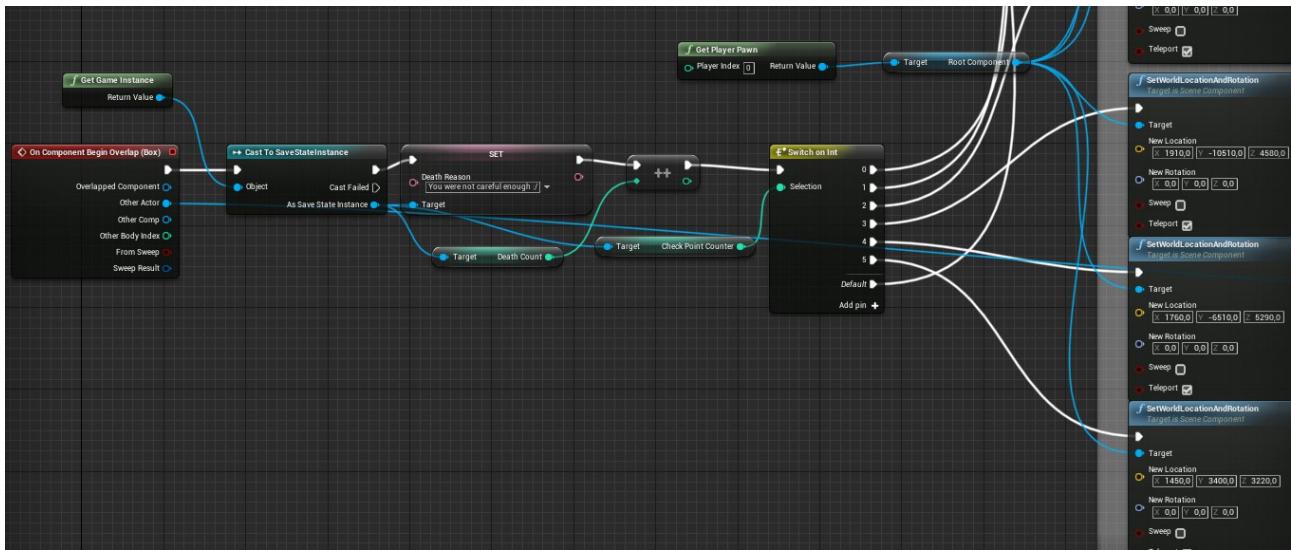
Oben sieht man die SetRecord-Funktion innerhalb der TeleportPlane. Diese Funktion wandelt die curSec und curMin in totalSec um (also zurück in Gesamtsekunden) und schaut erstmal ob es überhaupt einen Rekord zum vergleichen gibt. Falls nein (also wenn curRecord=0) dann wird dieser Rekord auf einen unglaublich hohen Wert gebracht. Danach vergleicht man die aufgenommene Zeit und die Rekordzeit.



Falls die aufgenommene Zeit kleiner ist als der derzeitige Rekord, wird der getNewRecord-Boolean auf true gesetzt und basierend auf den Namen der TeleportPlane wird der vorherige Rekord überschrieben. Falls nein, dann wird nichts überschrieben.

3.3 Todeszonen und Checkpoints

Im Spiel gibt es viele tückische Wege eine Todesteleportation des Spielers hervorzurufen. Aber beispielsweise in LevelP1 und LevelP2 wird überwiegend mit Checkpoints gearbeitet um dem Spiel zu sagen zu welchem Punkt der Spieler gekommen ist um einen geeigneten Ort für die Todesteleportation zu gewähren. Für diese Checkpoint Funktionalität wird ein Integer innerhalb des SaveStateInstance benötigt, welches bei jedem Levelstart zurückgesetzt wird. Der Checkpoint-Blueprint inkrementiert als Kollisionereignis diesen Checkpoint Zähler. Unten wird erläutert, wie die DeathP1 beispielsweise diesen Checkpoint Zähler verwendet, um die geeignete Position für die Todesteleportation zu erhalten.

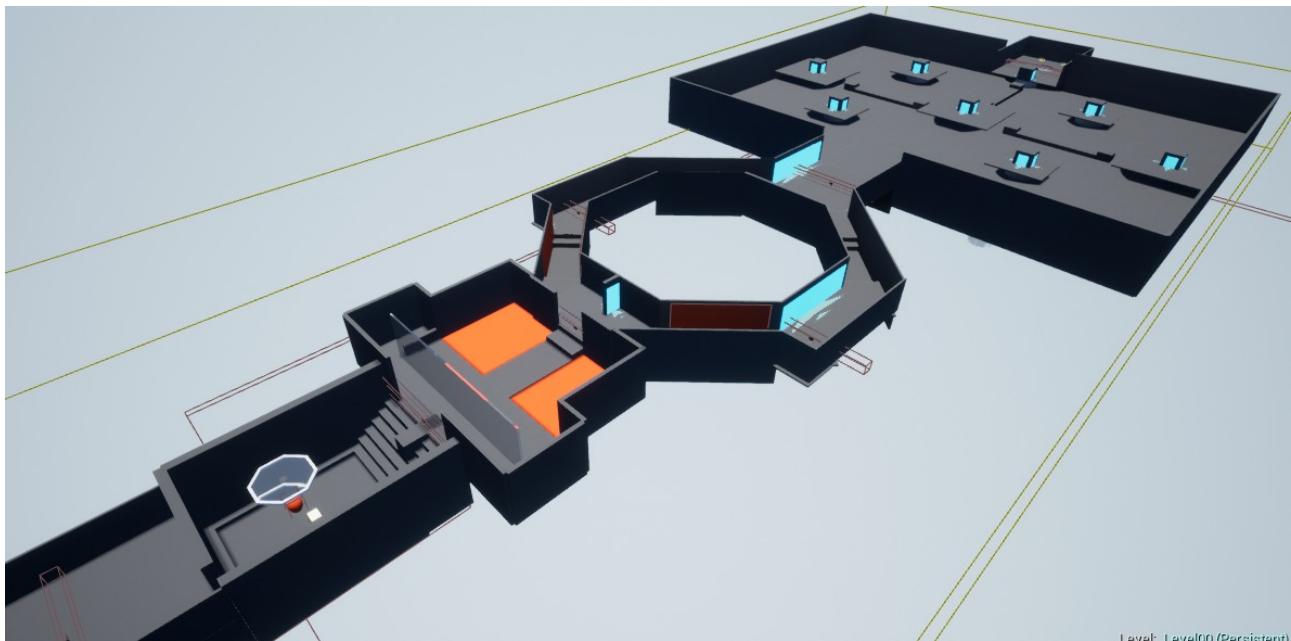
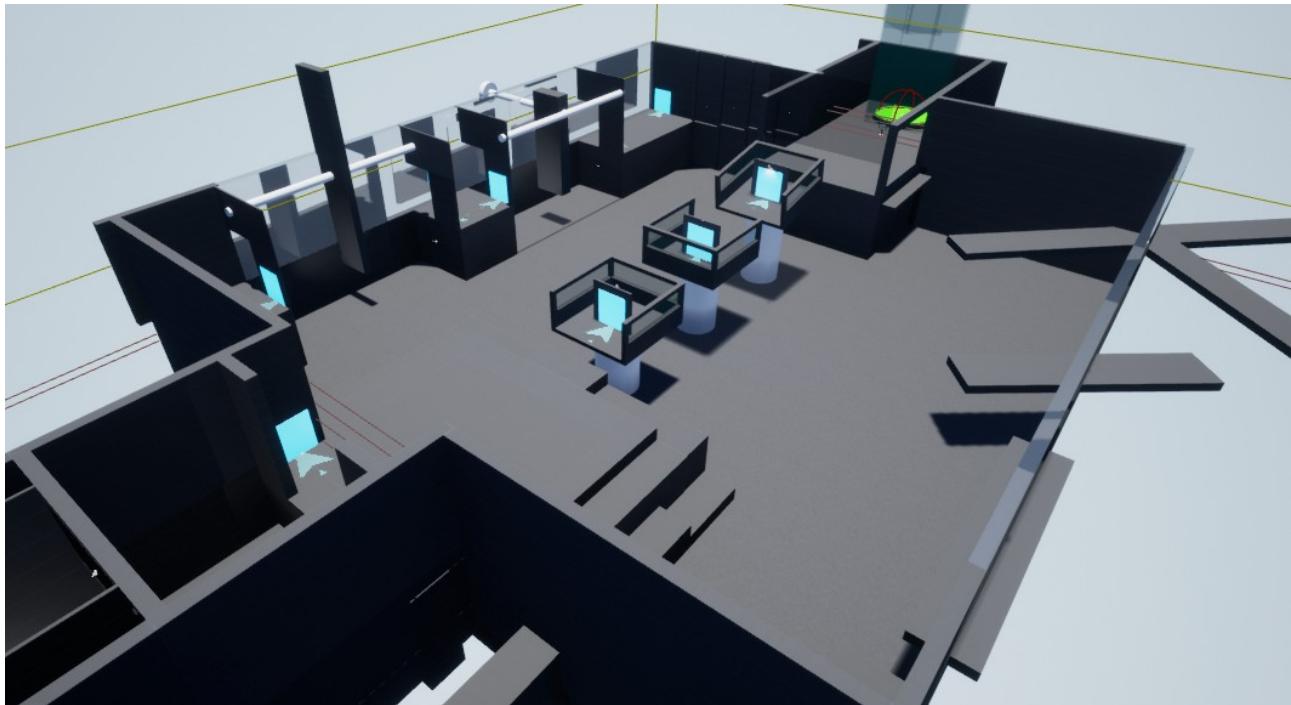


Die Kollision mit der Todeszone wird den Todeszähler und Todeursache anpassen und prüft danach nachdem Checkpoint Zähler in einer SwitchCase-Node. Die Orte sind hier bereits alle statisch angegeben.

4. Level Design

4.1 Tutorial-Sektion

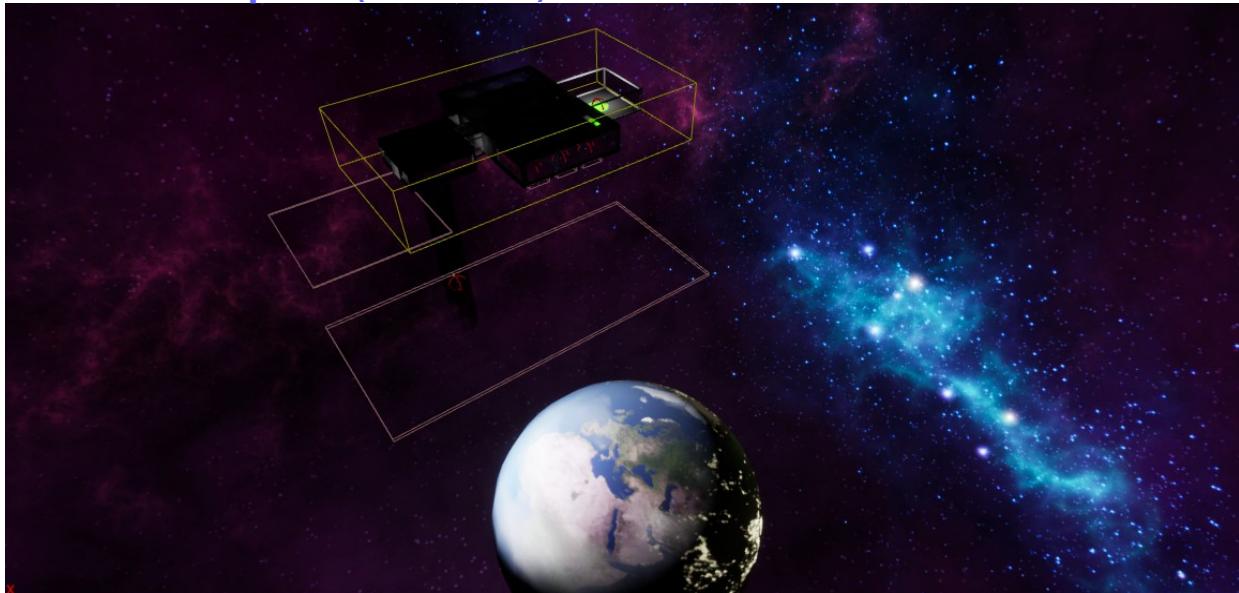
Die Tutorial-Sektion erstreckt sich über Level00_1 und Level00_2 und erklärt dem Spieler die Spielinhalte. Bei der Erstellung dieser Tutorial-Sektion musste darauf geachtet werden den Spieler nicht allzu sehr unter Druck zu setzen. Das schließt mit ein: Timing-basierte Barrieren, schnelles Rätselverständnis, Geschick und frustrierende Bereiche. Die visuelle Darstellung ist schlicht und die Anzahl der Levelobjekte übersichtlich. Zusätzlich befinden sich am Spielstart ein Schalter um die Hilfeanzeigen für Kontroller und Tastatur auszuwählen.



In Level00_1 wurde zudem der Beta Dash deaktiviert, somit war der Spieler in der Art der Problemlösung eingeschränkt und ließ ihn lernen, was man mit einer Vektorkraft alles bewerkstelligen kann. Im letzten Abschnitt des Levels wurden dem Spieler drei Lösungswege präsentiert. Der linke Weg benötigt die aktive Verwendung des AlphaDash um die sich bewegenden Barrieren zu überwinden. Bei dem mittleren Weg bedarf es lediglich einem passendem Timing, und die Barriere rotierte sich um den Spieler herum. Rechts benötigte man keine Vektorkraft und dort lernt man auch, dass in diesem Spiel die rotierenden Plattformen den Spieler nicht mitbewegen.

In Level00_2 wurden beide Vektorkräfte erlaubt, was den Spieler vielleicht verwirren mag. Die Anfangssektion ist daher sehr offen gehalten, was den Spieler animieren soll ein bisschen mit den Kräften rumzuspielen. Zur freien Verwendung stehen vier verschiedene polarisierte Vektorwände zur Verfügung. Eine wichtige Vektorverwendung ist anschließend der CombineDash, bei welchen sich die beiden geladenen Vektoren zusammensetzen um einen Durchschnittsvektor zu erhalten. Diagonale Dashes sind nun möglich. Im zweitletzten Abschnitt wird dem Spieler klargemacht, dass in Bezug auf Distanz bei einer getrennt ausgeführten Folge von AlphaDash und BetaDash eine größere Strecke zurücklegt wird, größer als ein CombineDash alleine erreichen kann. Im allerletzten Abschnitt trifft der Spieler zum ersten Mal auf einen Gestapo-Walker und muss eine kurze Sektion mithilfe des Walkers überwinden.

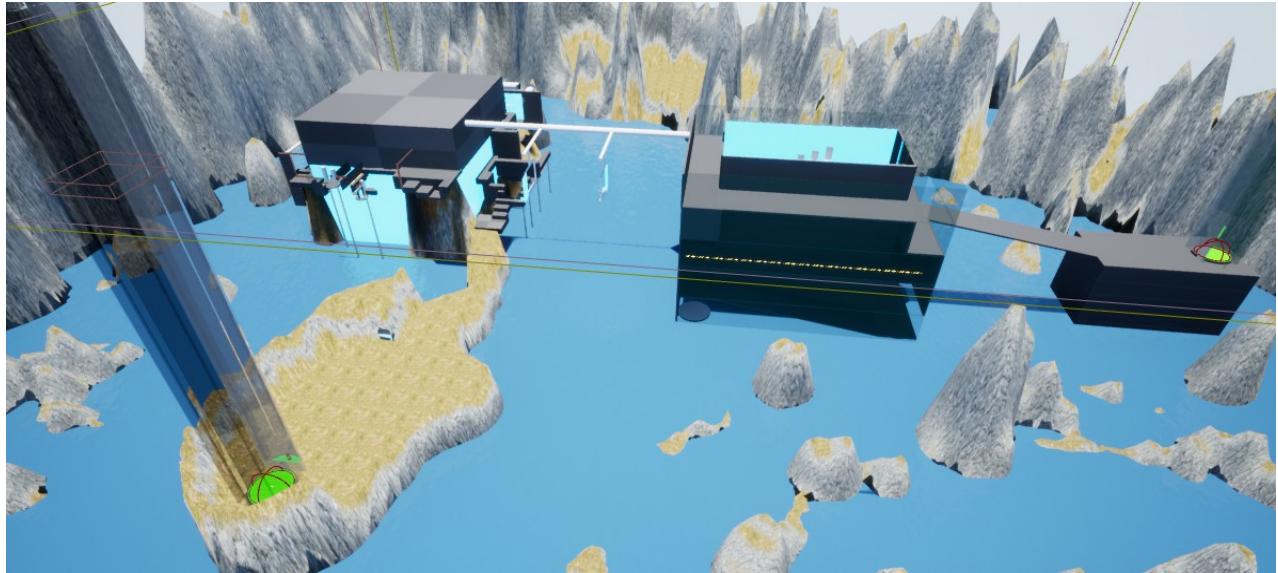
4.2 Levelknotenpunkt (HubWorld)



Die HubWorld bietet Zugang zu allen Levels, jedoch wird anfangs geprüft ob diese via SaveStateInstance bereits freigeschaltet sind. Der Player wird via Aufzug ins Hinterdeck transportiert. Dort findet er den Notausgang mit dem Titel "Radicke Mode" und eine Tafel mit allen Level Rekorden. Nutzt der Spieler den Notausgang wird der Radicke Modus aktiviert und der Spieler schaltet alle Levels frei. Das Aktivieren des Radicke-Modus verändert das TextLabel zu "Quit Game" und das erneute Nutzen des Notausgangs beendet das Spiel. Das Hinterdeck ist mit dem Hauptgang verbunden. Dort hat der Spieler Zugang zum LevelP1, LevelP2 und LevelP3. Weiter vorne findet der Spieler einen Zugang zum Tutorial.

Eine Besonderheit des Levels ist die Tatsache, dass das Levelmesh ungemein groß ist. Aus diesem Grund hat der Leveledesigner entschieden den Spieler für dieses Level allein größer zu skalieren. Dazu gehört ein dreifach verlängerter Kameraarm, höhere Sprungkraft und verdoppelte Spielerskalierung. Die Timervariablen werden auch zurückgesetzt, aber kein Timer ist implementiert. Als Premisse wurde der Weltraum gewählt mit einer sich drehenden Erde und einem sonnenähnlichen Sphärenmesh.

4.3 LevelP1



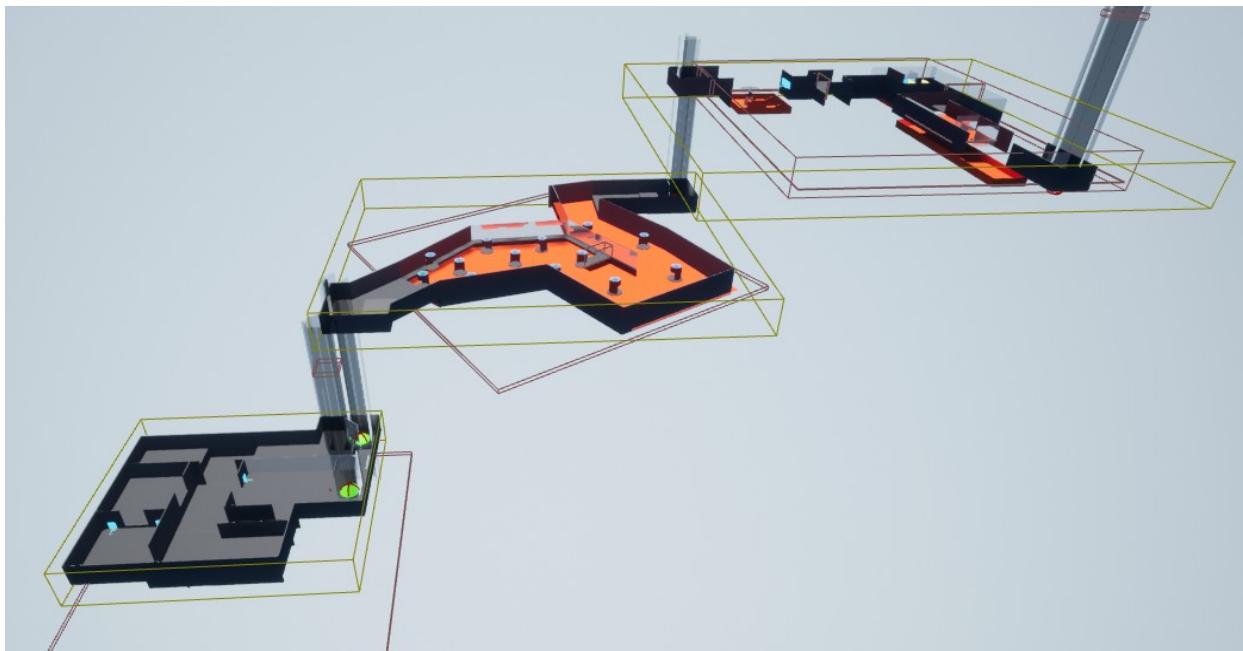
LevelP1 ist ein sehr offenes, aber auch ein sehr langes Level. Es hat an sich simplistische Hürden, aber könnte den Spieler womöglich desorientieren. Solange der Spieler das Ziel erkennt und merkt, dass sich die ersten Hürden um einen Würfel bilden, erkennt er die grobe Richtung seines Zielpunktes. Nach der Würfelsektion trifft der Spieler auf die hängenden Vektorwände. Nach jeder Sektion ist ein Checkpoint gesetzt, damit Frustration möglichst minimiert wird. Die Länge des Levels sollte hiermit kein Problem darstellen.

Auf dem gebäudeartigen Gebilde angelangt, wird er verführt einen doppelten Dash in eine Richtung zu vollführen, wodurch er gegen eine Glasscheibe stößt. Im Vektorkerker muss er mit beiden Vektorkräften hin und her jonglieren um nach oben zu gelangen und das Levelende ausfindig zu machen.

Zusätzlich dazu gibt es noch einen geheimen Weg den man nach der allerersten Hürde erreichen kann. Entweder man hält Ausschau nach Glaswegen oder man nutzt einen doppelten Dash in dieselbe Richtung um diesen Ort zu erreichen. Sie bietet keine Checkpoints aber einen leichteren Weg zum Ziel.

Für das Leveldesign war es wichtig eine natürlich aussehende Erdfarbe für die Landschaftsgestaltung zu simulieren und möglichst keine Unregelmäßigkeiten in der Landschaftshöhe zu hinterlassen. Glatte Oberflächen waren von enormer Wichtigkeit für das Spiel aufgrund von mehreren physikalischen Problemen (welche nun im jetzigen Stand des Projektes beseitigt worden sind). Ebenso wurde an authentisch aussehenden Wasserflächen gearbeitet, welches das Level noch farbiger und lebendiger erscheinen lassen sollen.

4.4 LevelP2



LevelP2 hat zwar einen sehr simplistischen Look, welches sehr an die Tutorial-Sektion erinnert. Jedoch sind die Rätsel kniffliger und die Anforderungen an den Spieler höher als jemals zuvor. Visuell getrennt kann man das Level in drei Abschnitte aufteilen. Der erste Abschnitt ist ein Vektorpuzzle und der Spieler muss mithilfe von Vektoren das Ziel erreichen. Beim Spielstart findet der Spieler den Teleporter bereits hinter einer Glaswand, sodass er sich keine Sorgen um sein Ziel machen muss. Das Level setzt voraus, dass der Spieler entweder eine separierte Folge beider Dashes einsetzt oder den CombineDash. Wie er welche Vektoren aufladen muss, ist Aufgabe des Spielers.

Im zweiten Abschnitt erwarten den Spieler die Vektorkarussells. Mithilfe dieser sich drehenden Vektorwänden darf er sich die Vektorrichtung selber zurechtbasteln. Aber das Geschick ist gefragt sein Ziel auch wirklich mithilfe der aufgeladenen Vektoren zu treffen. Der erste Teil dieses Abschnittes ist zum Teil mit Plattformen umrahmt um dem Spieler die Schwierigkeit zu nehmen den ersten Teilcheckpoint zu erreichen. Den restlichen Weg muss der Spieler aber von sich aus erarbeiten.

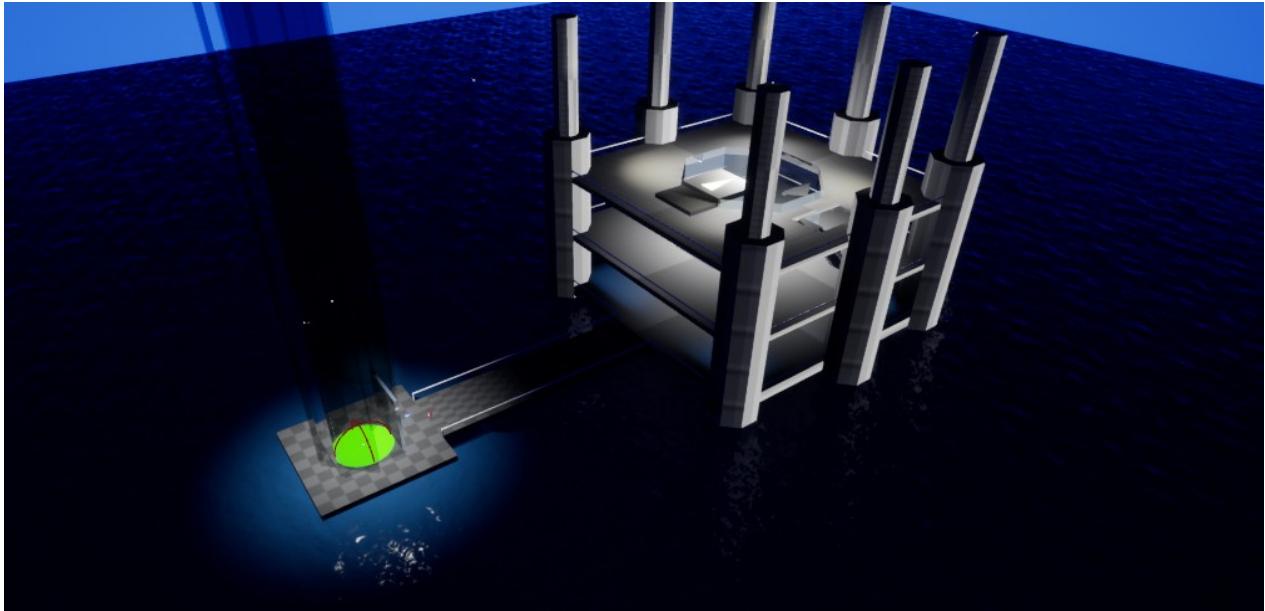
Der letzte Abschnitt verlangt vom Spieler die Zusammenarbeit mit dem Gestapo-Walker und ist in drei Teilaufgaben aufgetrennt. Nun muss er nicht nur auf seine Position acht geben sondern auch die Position des Walkers, denn trifft dieser auf Barrieren, dann kommt der Spieler auch nicht weiter. Mithilfe des Walkers manövriert sich der Spieler in Richtung der Vektorwände und nutzt diese in geeigneten Orten um weiterzukommen. Im zweiten Teilaufgaben muss der Spieler mehr auf seinen Walker acht geben und teilweise über Barrieren springen damit der Walker ihm folgt. Sein Ziel sind zwei Vektoren mit welchen er mithilfe einer separierten Nutzung zum letzten Teilaufgaben gelangt.

Eine rote Barriere erstreckt sich vor dem Spieler. Der Walker scheint problemlos sich rüberbewegen zu können, löst aber beim Spieler eine Todesteleportation aus. Eine leuchtende Plattform kann dem Spieler auf die Sprünge helfen. Er muss mit einem Dash zu einer der bewegenden Plattformen springen. Die Plattformen bewegen sich hin und her unter

einer Glaswand, sodass der Spieler gezwungen ist mit Sprünge auf die andere Seite der Glaswand zu gelangen. Danach findet er sich auf einem stabilen Boden wieder und kann mithilfe seines Walkers an das Levelende gelangen.

In diesem Level wurden viele Spielmechaniken auf einmal eingesetzt und der Spieler wird ständig gefordert. Der erste Abschnitt fordert sein Vektorverständnis, der zweite Abschnitt fordert sein Geschick und der letzte Abschnitt fordert seine Anpassbarkeit an das jeweilige Problem und das Finden von Lösungswegen.

4.5 LevelP3



LevelP3 ist simpel. Den Spieler erwarten hier keine Rätsel und kein Geschick während er sich durch das ölbohrplattform-artige Gebilde manevriert um zum Dach zu gelangen. Man springt in das Loch hinein und gelangt somit in den Ozean. Wo anders im Wasser würde eine Todesteleportation verursacht werden. Am Ozeanboden angelangt findet der Spieler Pfeile, welche ihn almählich zu einem Unterwassergebäude führen: die BossArena. Dort trifft er auf den Boss, den Gestapo-Crawler. Der Crawler hat 200 Lebenspunkte und der Spieler muss wiederholt mit dem Rücken des Crawlers mithilfe von Vektorkräften kollidieren, um den Crawler Schaden zuzufügen. Ist dieser Crawler besiegt, eröffnet sich ein Weg zum Ziel. Wird der Spieler jedoch besiegt, so teleportiert er sich einfach über die Arena und der Crawler wird um 20 Lebenspunkte geheilt.

Im Vergleich zu allen anderen Levels ist dieses Level sehr dunkel gehalten. Es wird mit Lichteffekten gespielt, um den Spieler bestmöglichst zu führen. Ebenfalls verfügt der Spieler in diesem Level über einen Scheinwerfer, der stetig vor ihm gestrahlt wird. Dieser Scheinwerfer verschwindet erst nach Erreichen der Bossarena. Die Wassertextur wurde von LevelP1 übernommen, jedoch hier auch doppelseitig weil der Spieler auch unterhalb des Ozeans die obere Wasserstruktur bestaunen möchte.