Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

2.5D hra v Unreal Engine

2024

Bc. Štěpán Šumpík

Vedoucí práce:
RNDr. Martin Trnečka, Ph.D

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Bc. Štěpán Šumpík
Název práce:	2.5D hra v Unreal Engine
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	RNDr. Martin Trnečka, Ph.D
Počet stran:	79
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Štěpán Šumpík
Title:	2.5D Game in Unreal Engine
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	RNDr. Martin Trnečka, Ph.D
Page count:	79
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Diplomová práce je zaměřena na vývoj 2.D arkádové akční hry. Hra umožňuje hráči ovládat postavu a prozkoumávat dynamický svět složený z několika levelů. Vývoj proběhl za použití Unreal Enginu jako hlavní platformy. Implementace jednotlivých částí hry včetně soubojového systému s umělou inteligencí je provedena pomocí skriptovacího jazyka a dalších nástrojů enginu. Na tvorbu některých assetů pro tuto hry byly použity další aplikace jako Audacity, Blender, GIMP a Cascadeur. Hra obsahuje široké spektrum herních prvků včetně skóre, různých typů překážek a pastí, bonusů a vylepšení. Systém lezení umožňuje hráči navigaci levelem a soubojový systém boj s nepřáteli s vlastní umělou inteligencí. Je možnost si v nastavení upravit podle svých preferencí a hardwarových možností.

Synopsis

The thesis focuses on the development of a 2D arcade action game. The game allows players to control a character and explore a dynamic world composed of several levels. The game development was carried out using Unreal Engine as the main platform. Implementation of various parts of the game, including the combat system with artificial intelligence, is done using scripting language and other engine tools. Additional applications such as Audacity, Blender, GIMP, and Cascadeur were used to create some assets for this game. The game features a wide range of gameplay elements including scoring, various types of obstacles and traps, bonuses, and enhancements. The climbing system enables players to navigate through levels, and the combat system involves fighting enemies with their own artificial intelligence. There is an option to customize settings according to personal preferences and hardware capabilities.

Klíčová slova: hra; Unreal Engine, skákačka, akce

Keywords: game; Unreal Engine, platformer, action

Děkuji Bohu, vedoucímu práce, rodině a všem kdo mě po celou dobu studia podporovali. Děkuji také svým kolegům z firmy ConfigAir.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

1 Úvod

Již od malička jsem velkým fanouškem skákacích her, skákaček. Velmi se mi líbila stará hra Prince z Persie. Když jsem si vybíral téma diplomové práce, napadlo mě vytvořit hru, která by se těmito hrami z dětství inspirovala. Neměl jsem s vývojem her nijak velké zkušenosti. Pracuji však ve firmě ConfigAir a již druhým rokem používám Unreal Engine na vývoj konfigurátoru. Výsledkem diplomové práce je dle zadání hra vyvinutá v Unreal Enginu. Hra je vytvořena v rozměrech 2.5D. Ovládání hry a design levelů je dělaný ve stylu pro hry ve dvou rozměrech, ale modely a prostředí jsou ve třech rozměrech. Hra je složená z mnoha komponent, které budu postupně v tomto textu popisovat.

Jednotlivé komponenty, ze kterých je hra složena, jsou naprogramovány tak, aby byly co nejvíce upravitelné, rozšiřitelné a znovu použitelné pro další projekty nebo i pro rozšíření hry. Hra má několik již vytvořených levelů, ale díky těmto komponentám lze vytvořit velké množství nových designově zajímavých a hratelně zábavných levelů. V levelech jsou také nepřátelé s umělou inteligencí, kteří reagují na podněty od hráčovy postavy, Prince. Ten může v jednotlivých úrovních získávat nové schopnosti a vylepšovat své statistiky či používat rozdílné zbraně.

Velká část assetů, které jsou použity pro tuto hru, není mým dílem. Jedná se o textury, materiály, animace, zvuky a modely. Bylo by velmi náročné tyto assety připravovat od začátku, ač jsem to u množství z nich udělal. V tomto textu budou tyto assety vždy popsány. Nejsou nicméně zahrnuty ve výsledné práci, kterou dávám k dispozici, protože je podle smlouvy se společností Epic Games nemohu distribuovat. Odevzdaný projekt bude tedy bez těchto assetů a v souboru `README.txt` je postup, jak projekt zprovoznit v případě, že by to někdo chtěl udělat a dané balíčky měl. Až na tři balíčky byly v jednu dobu všechny na čas zdarma dostupné v Epic Games Marketplace. [1] Tyto assety tedy mohu použít, a to i pro komerční účely. Hra vytvořená z projektu je tedy zabalena i s těmito assety.

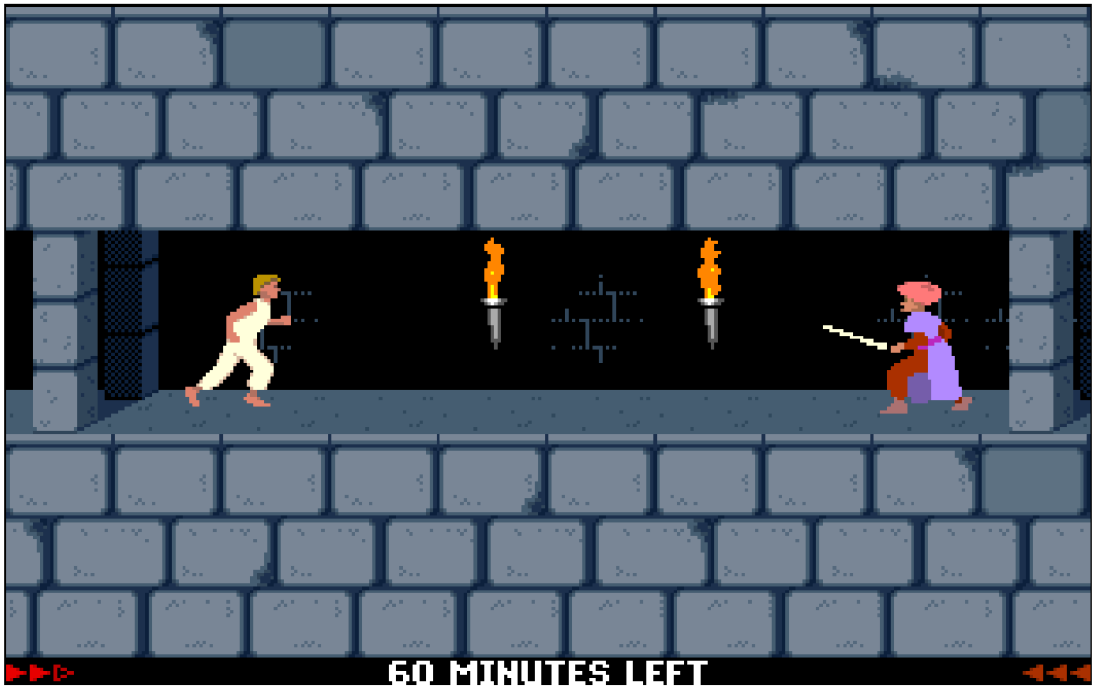
Snažil jsem se také o to, abyh uživatele povzbudil k opětovnému hraní hry. To podporují nové typy zbraní, získatelné až po více průchodech hrou a časovač.

2 Analýza požadavků

Ze zadání práce vyplynulo množství požadavků na herní obsah. Tyto požadavky jsem už popsal v úvodu, zde je pro pořádek shrnu. Vytvořil jsem tedy hratelnou postavu, `Prince`, hráč může tuto postavu ovládat, překonávat překážky pomocí systému lezení, vyhýbat se pastem a bojovat s nepřáteli. V levelech může hráč nacházet vylepšení `Prince`. Hra podporuje a odměňuje znovuhratelnost.

2.1 Inspirace - Prince of Persia

Ještě než začnu popisovat jednotlivé části této práce, odbočím k inspiraci. Nápad a vlastně celá tato práce má počátek ve hře Prince of Persia z roku 1989, 1 která byla napsána Jordanem Mechnerem v nízkoúrovňovém jazyce assembler.[2]



Obrázek 1: Obrázek z DOS verze hry Prince of Persia

Hra okamžitě po vydání zaznamenala obrovský úspěch a napříč platformami překonala milník 2 milionů prodaných kopií. [3] Ovlivnila a nadále ovlivňuje velkou řadu vývojářů [2].

Na začátku vývoje této hry v červnu 2023 Ubisoft nečekaně oznámil nový díl, který se vrací ke kořenům série. Stejně jako tato práce je to 2.5D hra s analogickou hratelností. [1]

[1] Hra vyšla 18. ledna 2024. 2

Obrázek 2: Obrázek ze hry Prince of Persia: The Lost Crown

2.2 Inspirace - Assassin's Creed

Série her Prince of Persia pokračovala s velkým úspěchem a dobrým přijetím jak od kritiků, tak od hráčů. V roce 2007 však vydal Ubisoft novou hru, Assassin's Creed. Ta byla původně koncipována jako další titul série Prince of Persia s pracovním názvem Prince of Persia: Assassins, kde měl hráč ovládat princova strážce z řádu assassínů, ale během vývoje se z této hry stal právě Assassin's Creed.

Měla do té doby téměř nevídaný úspěch a během prvního roku prodala přes 8 milionů kopií. Odstartovala tak herní sérii, která čítá k dnešnímu dni přes 13 hlavních titulů a přes 200 miliónů prodaných kopií. Nepřímo tak tuto obrovskou značku nastartoval jeden člověk svou hrou. Toto však mělo za následek, že série Prince of Persia byla zastíněná úspěchem Assassin's Creed a po nevelikém úspěchu filmové adaptace a poslední hry pojmenované Prince of Persia: Forgotten Sands Ubisoft přestal další hry v této sérii vyvíjet. Některé herní mechaniky v této práci jsou sérií Assassin's Creed také inspirované. Tato série stále sdílí také množství znaků se sérií Prince of Persia. Nejnovější díl Assassin's Creed: Mirage 3 se vrací tématikou zase na Blízký východ, a to do Bagdádu 8. století po Kristu. [3]

Obrázek 3: Obrázek ze hry Assssin's Creed: Mirage

3 O herních enginech

Pro lepší pochopení této práce popíši ještě základy herních enginů obecně a více specificky Unreal Engine. Vývoj her v dnešní době probíhá ve velké většině případů pomocí enginů. Vyvinout hru od základu by nebylo efektivní. Herní engine proces vývoje velmi urychluje a zjednodušuje. Engine si herní studia vyvíjí a upravují dle své potřeby sami, nebo mohou využít dostupných enginů jiných studií. [4]

Herní engine je platformou používanou pro vývoj her. Další použití zahrnují vizualizaci, konfiguraci, virtuální produkci, tvorbu filmů, také najde využití v architektuře. Mezi funkcionality patří znovupoužitelnost na více projektů a široká podpora platforem. Vývojáři mají možnost vyvíjet hry či aplikace na konzole a mobilní zařízení. [5]

Engine umožňuje renderování grafiky, převádí vstupní data na pixely zobrazené ve scéně. Obsahuje také fyzikální engine pro detekci kolizí, dokáže zpracovávat světlo a jeho působení na materiály objektů, obsahuje nástroje na práci s texturami. Engine také obsahuje několik modulů, například modul animací, který umožňuje například pohyb, rotaci či změnu tvaru objektů ve scéně.

Modul umělé inteligence enginu umožňuje definovat pravidla pro chování postav, které nejsou ovládány hráčem. V Unreal Enginu je tato funkcionalita realizována pomocí `BehaviourTrees`, stromů chování.

Síťový modul umožňuje hráčům hry hrát společně ve stejný čas, přičemž jejich zařízení komunikují zpravidla prostřednictvím internetu. Unreal Engine umožňuje, aby se výsledná aplikace chovala i jen jako server, nebo jen jako klient. Stejně tak je možná komunikace peer-to-peer.

Enginy obsahují také nástroje k optimalizaci výsledného programu. Některé volně dostupné enginy, jako Unreal Engine, umožňují vývojáři upravit dokonce zdrojový kód enginu nebo úplně novou část enginu. Z hardwarových funkcí engine usnadňuje a zprostředkovává streamování, paralelizaci procesů či přiřazování paměti. Zpracovává také vstup od uživatele a jeho propagaci. [6]

Možné kandidáty na engine pro diplomovou práci jsem si vybíral z dostupných možností na základě svých zkušeností s enginem, a také podle možností, nabízených funkcí, stylu financování a upravitelnosti enginu.

3.1 Unity

Unity je herní engine vyvíjený společností Unity Technologies. [6] Dnes podporuje vývoj na herních konzolích, a to včetně Nintendo Switch, na všech standardně používaných operačních systémech (Windows, Linux, macOS), stejně jako vývoj aplikací pro virtuální realitu. Pro vývoj se používá programovací jazyk C#. [6]

Používá modulární systém založený na komponentech. Objekty jsou složeny z komponent, které jsou sadou funkcí, což umožňuje používat dědičnost a tím se podobá objektově orientovanému programování. Výhodou je velká flexibilita. Unity je zvlášť mezi mladými a začínajícími programátory velmi populární díky svému designu, komunitě a podpoře. Na podzim roku 2023 se ale dostali jeho vývojáři do problémů, když změnili přístup k financování projektů vytvořených v tomto enginu. Nově měli vývojáři platit poplatek za každé stažení hry, což by citelně zasáhlo zvlášť začátečníky. Z toho důvodu mnoho z nich hledalo alternativu a našlo ji právě v Unreal Enginu. [7] Unreal se oproti Unity v komunitě často vnímá jako silnější engine s více možnostmi. [6]

3.2 Cry Engine

Je herní engine vyvíjený společností Crytek. [6] Taktéž je vhodný a přístupný pro začátečníky, podobně jako Unreal Engine se za jeho použití platí po přesažení určité vydělané částky procento ze zisku. Ve srovnání s Unreal Enginem není tak populářní, tudíž nemá tak velkou komunitu a s ní spojené informace na fórech, a také menší množství tutoriálů. Pro programování v enginu se používají jazyky C# a C++. Navíc se používá grafický systém Flow Graph pro programování logiky hry. [6]

Cry Engine byl použit českými vývojáři na vývoj jedné z nejpopulárnějších a nejúspěšnějších českých her všech dob, Kingdom Come: Deliverance 4 od studia Warhorse. Hra se dočkala velmi dobrého přijetí jak od kritiků, tak hráčů. Velmi byla vyzdvihováno prostředí české přírody, konkrétně okolí Sázavy. Tato grafika byla zprostředkována právě Cry Enginem.

3.3 Unreal Engine

Unreal Engine (i Cry Engine) nabízejí z vývojářekého pohledu mnohem širší spektrum možností toho, co s nimi dokáže vývojář vytvořit. [6] Uvedu zde pár

Obrázek 4: Obrázek ze hry Kingdom Come: Deliverance

příkladů.

3.3.1 Využití v multimédiích

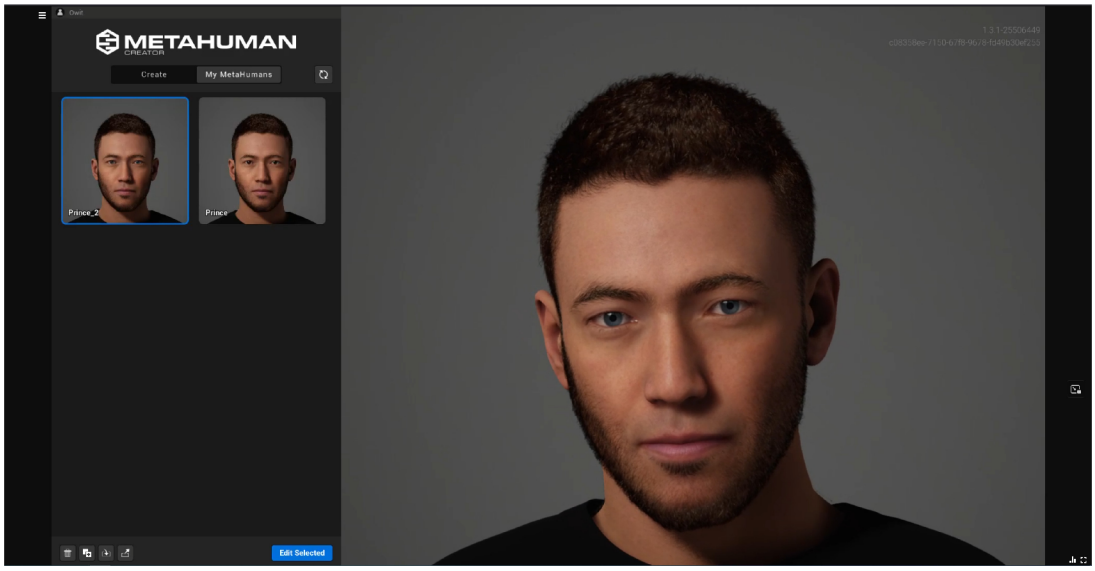
Oproti Unity může Unreal Engine vytvářet až fotorealistické prostředí a dnes se hojně používá při natáčení filmů či seriálů. Z posledních let jsou to seriály ze světa Star Wars produkované společností Disney, například Mandalorian. 5 Dosavadní technologie blue/green screen, kdy se barevně jednolité pozadí vyklíčuje při tvorbě speciálních efektů (VFX), se nahrazuje obrovskými LED monitory, na kterých na pozadí běží aplikace v Unreal Enginu.

Vytváří se v podstatě virtuální set, který se snadno kombinuje s fyzickým setem. Při rotaci a posouvání kamery se scéna v Unreal Enginu taktéž posunuje. Výhodou je například to, že herec na monitoru vidí prostředí, ve kterém se jeho postava nachází, což mu může velmi usnadňovat jeho práci. Další výhodou jsou například odlesky na lesklých površích, které jsou jinak na klíčování a tvorbu velmi náročné. [8]

3.3.2 Metahumans

Společnost Epic Games si pro své potřeby vyvinula vlastní editor postav a tváří, který v posledních letech stále vylepšuje, jmenuje se **Metahumans**. 6 Na jeho použití se vztahují stejné podmínky jako na samotný editor. **Metahumans** zažil svůj debut jako součást propagace filmu Matrix: Resurrection. Editor postav **Metahumans** jsem použil i v této práci, a to pro některé části modelu hratelné postavy, `Prince`.

Obrázek 5: Obrázek z Making of Mandalorian



Obrázek 6: Obrázek z editoru postav Metahumans

Nevýhodou může být to, že importovaná postava má rozsáhlou datovou velikost, protože se importuje velké množství úrovní detailů, `LOD`, mnoho animací tváře, fyzika vlasů a další. V této práci jsem importovaného `Prince` částečně optimalizoval.

3.3.3 O enginu

Unreal Engine začal jako rozšíření pro FPS hru jménem Unreal. [5] Na začátku mohli uživatelé pouze upravovat a přidávat nové elementy do této hry, v pozdějších verzích se ale jeho funkcionalita velmi rozšířila. Podporuje vývoj pro virtuální či rozšířenou realitu, herní konzole či mobilní zařízení nebo HTML. Pro mobilní platformy a HTML však není primárně určen a vývojář proto musí počítat s velkou náročnost výsledné aplikace a omezenými možnostmi, co se týče optimalizace a plynulého běhu.

Kromě jazyku v C++, ve kterém lze naprogramovat v podstatě cokoliv v enginu a který se doporučuje primárně používat pro rozsáhlé projekty lze programovat pomocí blueprintů, vizuálního skriptování. Programování v C++ rovněž usnadňuje synchronizaci vývoje v případech, kdy na jednom projektu pracuje více vývojářů.

Blueprint systém je ale snadněji pochopitelný, než když vývojář programuje celou aplikaci od začátku v C++. Výhodou je, že potom i lidé, kteří s programováním mají malé či žádné zkušenosti, mohou s pomocí vizuálního skriptování skrze `Blueprints` programovat, či kódu alespoň rozumět.

Unreal Engine se skládá z následujících komponent: [6]

- Zvukový engine: **Sound Cue**. **Sound Cue** v této práci společně s **Metasounds** 7 na úpravu zvuků často používám. Slouží mimo jiné k tomu, že pokud se nachází postava hráče dál od zdroje zvuku, je tlumenější a od určité vzdálenosti nejde slyšet vůbec. Zvuky lze totiž přehrát buď 2D (například UI zvuky) nebo přímo v nějaké lokaci ve scéně. Některé zvuky upravuji pomocí komponent **Metasounds**. K výšce zvuku přidávám náhodnou hodnotu v předem určeném rozsahu (například u kroků) a tak je zvuk každého kroku unikátní a neopakuje se. Toto dělám pro velké množství zvuků v projektu, pro zranění, kroky nepřátel i útoky.

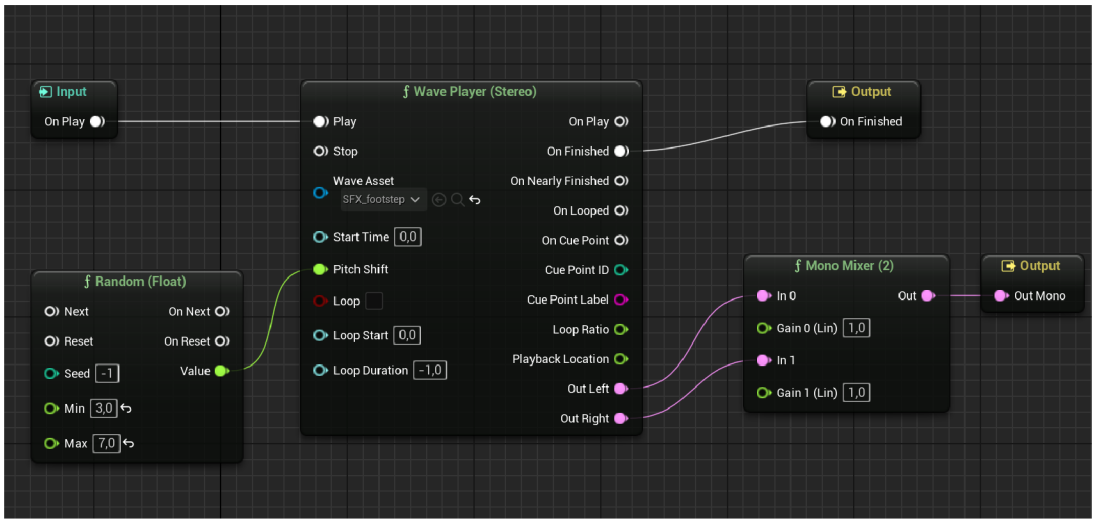
- Fyzikální engine: **PhysX**, Tento engine využívá i Unity, je vyvíjen společností Nvidia. Zprostředkovává mimo jiné chování objektů v prostoru, gravitaci či pohyb objektů.

- Grafický engine podporuje DirextX11/12, OpenGL, Javascript, WebGL knihoven.

- Post-process efekty. Jedná se například o shadery či úpravy světelnosti nebo barev. Například lze nastavit, jak se bude chovat obraz kamery ve scéně při náhlé změně světelnosti, nebo maximální a minimální možné nasvícení scény.

- Základ hry. Funkce kontrolující průběh hry.

- Umělá inteligence. Tímto se myslí umělá inteligence nehratelných postav v levelu. V Unreal Enginu se používají BehaviourTrees.



Obrázek 7: Obrázek – ukázka úpravy zvuku pomocí Metasounds komponenty

- Síťový modul. Velmi dobře implementovaný, a to i v porovnání s ostatními enginy. [6]

Unreal Engine obsahuje mnoho dalších funkcí jako **Lumen, Chaos Engine, Nanite** či snadnou podporu lokalizací. Všechny vyjmenovávat a popisovat není ale předmětem této práce a proto budu posléze zmiňovat jen ty, které budou relevantní.

3.4 Základní třídy v Unreal Enginu

Při popisování mnoha komponent velmi často zmiňuji základní třídy, k správnému pochopení je zde krátce popíši. [9]

3.4.1 UObject a prefixy

Nejzákladnější třídou, ze které dědí velké množství ostatních, je `UObject`. Usnadňuje funkci garbage collectoru a zprostředkovává mimo jiné reflexi. Jako všechny další třídy, které zmíním, je to další úroveň abstrakce nad jazykem C++. Unreal C++ je rozšířením jazyka C++ pro potřeby Unreal Enginu. Tento jazyk úroveň abstrakce provádí rovněž nad mnoha datovými typy a dalšími třídami. Například místo datového typu string používá `FString` či `TEXT`. Tyto datové typy jsou pak velmi dobře kompatibilní s `Blueprints`. Datové typy a struktury, a to i vývojářem vytvořené, se dají snadno poznat podle prefixů. Unreal Engine prefixy při programování v C++ vyžaduje. Je také správnou praxí používat prefixy i u blueprintů, není to však vyžadováno a projekt se zkompiluje i jejich absenci navzdory. [10] V této práci se těmito prefixy řídím, nejčastěji používaný prefix je `BP`, který znamená, že se jedná o blueprint třídu.

3.4.2 AActor

`AActor` má prefix `A` a označuje objekty, které se dají umístit do levelu. Podporují tedy základní transformace lokace, rotace a škálování. Není totožný s `UObject`, ale dědí z něj. Začínající vývojář může tyto třídy plést. `AActor` není každý objekt, který se umístí do levelu. V levelu může být třeba jen pouze `StaticMesh` objekt. (který dědí z `UObject` a znázorňuje 3D model společně s materiálem a kolizí). `AActor` je typický tím, že má už nějakou funkci a může mít jako svou součást komponentu `StaticMesh`, a to i více než jednu.

3.4.3 Pawn, Character, Controller

Dědí z `AActor` a reprezentují objekt, či postavu v případě `Character`. Tuto postavu lze ovládat hráčem, nebo umělou inteligencí. `Character` je třídou dědící z `Pawn` u které se předpokládá, že je humanoidní, chodí po dvou končetinách. Pro ovládání obou tříd je zapotřebí `Controller`, tedy oddělené třídy, která má na starosti zpracování vstupu od hráče. `Controller` je oddělen od třídy `Pawn`, takže lze přepínat mezi více ovladatelnými postavami zároveň. Teoreticky jde například vytvořit situaci, kdy postavu původně ovládanou hráčem převezme umělá inteligence a hráč ovládá místo toho postavu nepřítele. `Controller` se dělí ještě na `PlayerController` a `AIController`.

3.4.4 Level blueprint

Každý level ve hře má také svůj blueprint, do kterého má vývojář možnost naprogramovat, co se stane při otevření tohoto konkrétního levelu. Tuto funkcionalitu ve své práci dále také používám.

3.4.5 Gamemode a jeho části

Důležitou blueprint třídou v každé aplikaci vytvořené v Unreal Enginu je `Gamemode`. Obsahuje množství tříd, které mají nastavené nějaké defaultní chování, vývojář ale má možnost tyto třídy přepsat, vytvořit nové třídy stejného typu s jiným chováním. První z nich je `PlayerController`. Dalšími jsou `GameSession`, která se používá primárně pro online komunikaci se serverem, dále `GameState` pro uchování dat napříč levely. Přepsání a použití třídy `GameState` jsem pro tuto práci zvažoval, hodila by se pro uchování skóre a hodnoty časovače napříč levely. Nakonec jsem ale místo toho použil `SaveGame`, takže tyto informace jsou ukládány po dohrání levelu.

Dále je možnost použít vlastní `HUD` class, takže se uživatelské rozhraní načte hned při otevření levelu. Já jsem místo toho uživatelské rozhraní realizoval za pomocí vlastní třídy `MenuPawn`. Mnoho z požadovaného kódu a funkcionalit jde totiž napsat do více tříd bez ohledu na to, jestli jsou pro to určeny nebo ne. Není to však dobrá praxe a z hlediska udržitelnosti a pozdějšího vývoje to rozhodně doporučit nemohu. Nicméně to vždy záleží domluvě vývojářského týmu

a například `HUD` jako součást třídy `Pawn` je dle mých zkušeností s vývojem běžná praxe.

4 Architektura aplikace

4.1 Uživatelské rozhraní v MainMenu

V této sekci jsou popsány jednotlivé části rozhraní v hlavním menu. Hlavní menu vypadá po zapnutí hry takto. 8

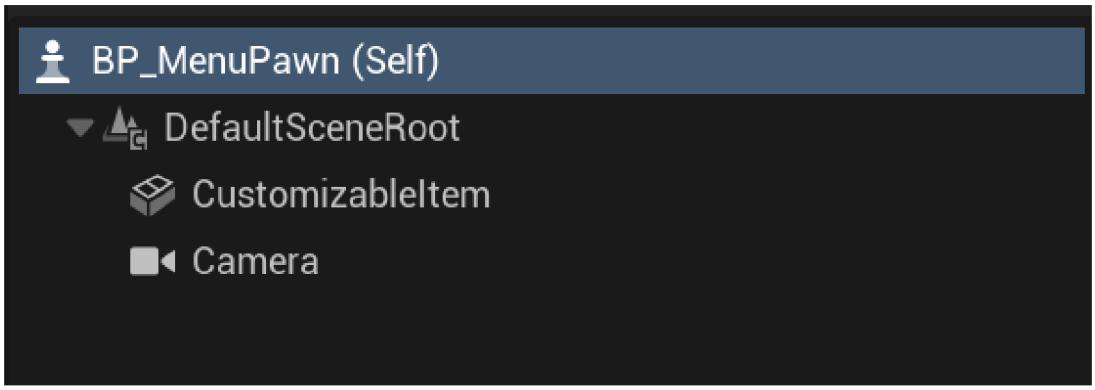
Obrázek 8: Obrázek z hlavního menu

`Gamemode` třída je nastavená tak, že defaultní `Pawn` po zapnutí hry je `BP_MenuPawn`. 9 Tento `Pawn` má statickou kameru, která zprostředkovává náhled do 3D prostoru levelu. Obsahuje také prázdnou `StaticMesh` nazvanou `CustomizableItem`. Tato `StaticMesh` slouží k zobrazování zbraní v části tohoto menu. `MenuPawn` má pouze defaultní `Controller` a vlastně slouží spíše jako `AActor`. Je však `Pawn`, protože obsahuje kameru, skrze kterou se hráč dívá na scénu při zapnutí levelu.

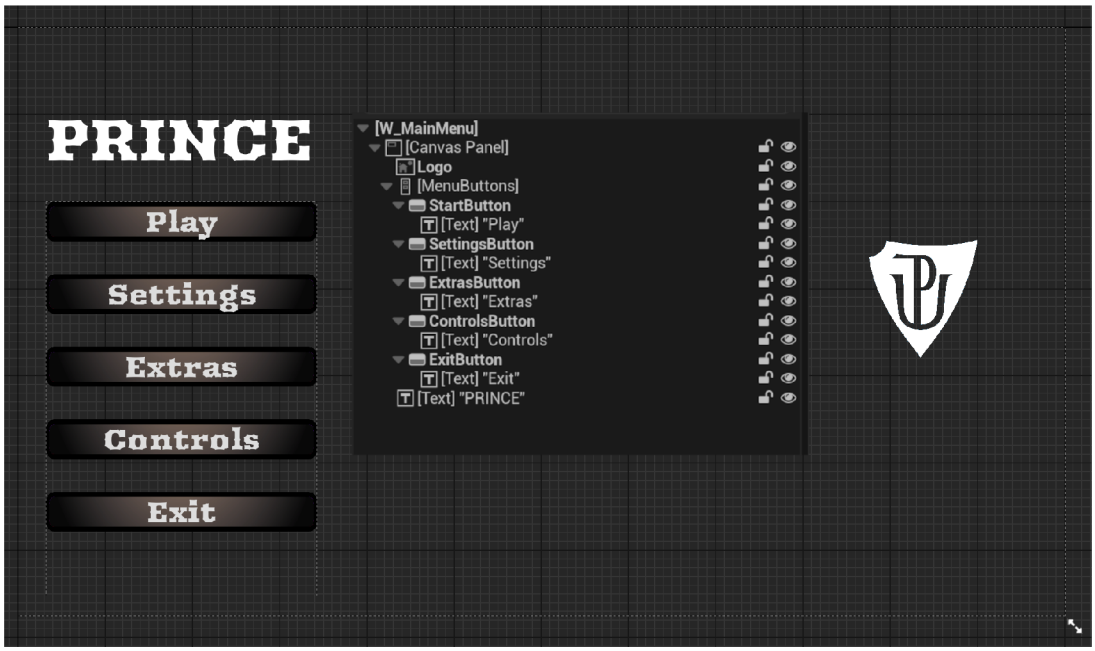
UI v hlavním menu přidávám při spuštění levelu v blueprint třídě levelu do viewportu. Společně s tímto se spustí hudba v hlavním menu a také se načte nastavení. Jak budu ukazovat i později, grafické nastavení ve hře se musí při každém spuštění znovu nahrát, aby se jeho efekt aplikoval. V blueprintu levelu tedy nahrávám nastavení, které bylo naposledy uloženo v `W_SettingsMenu`. (prefix `W` znamená widget).

4.1.1 MainMenu Widget

Uživatelské rozhraní se v aplikacích vytvořených v Unreal Engine realizuje takzvanými widgety. Jejich nastavení a chování není nepodobné leckterým editorům HTML. Místo `UObject` tyto třídy dědí ze třídy `UWidget`. Obrázek ukazující widget třídu `MainMenu`. 10



Obrázek 9: Obrázek z blueprint třídy MenuPawn



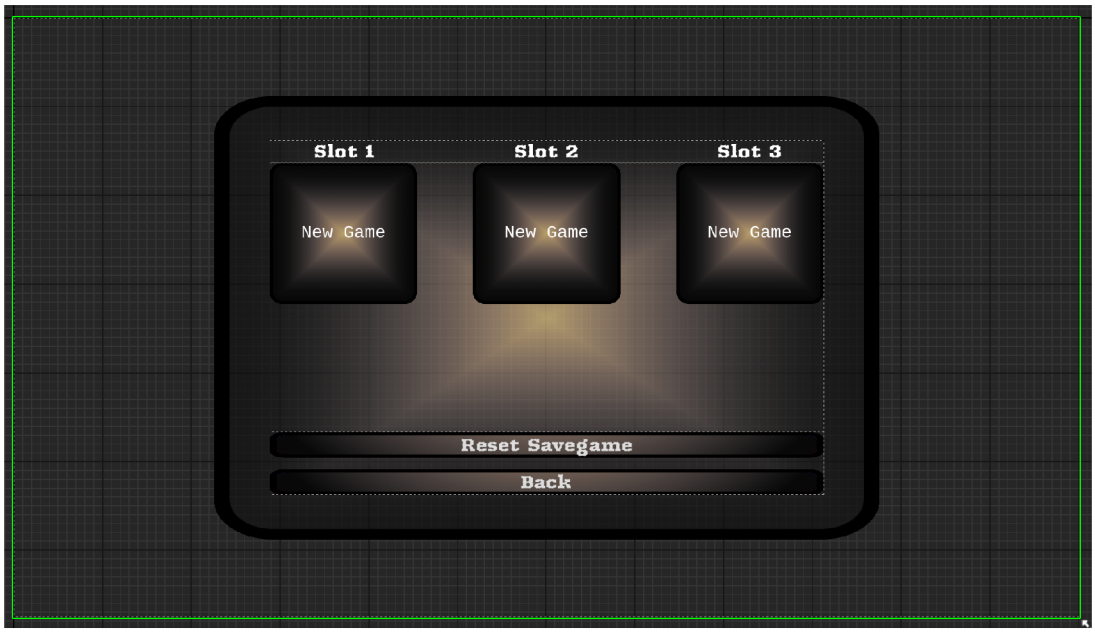
Obrázek 10: Obrázek z blueprint widget třídy MainMenu

Ve všech částech menu používám i vlastní widget, který jsem pojmenoval `MenuButton`, tlačítko, na které jsou navázány textury. Tyto textury vypadají jinak při přesunu kurzoru nad tlačítko a po jeho stisknutí. Jsou mnou vytvořené v grafickém programu **GIMP**. Na toto tlačítko je také navázán zvuk, který se přehraje po kliknutí. Vytvořil jsem i další tlačítka pro `W_SettingsMenu`.

4.1.2 PlayMenu Widget

Po stisknutí tlačítka `Play` v UI se odebere `MainMenu` widget a místo něj se do viewportu přidá`W_PlayMenu`. 11

Tento widget slouží k načtení předtím hrané hry. V případě, že je hra rozehrána, je text New Game nahrazen levelem, ve kterém se hráč zrovna nachází.



Obrázek 11: Obrázek z blueprint widget třídy PlayMenu

Hra se ukládá automaticky při dokončení každého levelu. K načtení používám funkci LoadLevel, část z ní lze pro představu vidět na obrázku. Tato funkce nejprve zkontroluje, jestli ve slotu, na který hráč klikl, již existuje uložená hra. Pokud ano, pak načte level uložený v tomto slotu. Pokud ne, vytvoří novou instanci třídy `BP_SaveGame_Level`, která bude popsána později, a načte první level. V obou případech se do blueprint třídy `BP_GameInstance` uloží reference na slot, na který hráč klikl. Toto slouží k pozdějšímu ukládání hry.

Textury v tomto UI prvku jsou taktéž mnou vytvořené v programu **GIMP**. Tlačítkem `Back` se `PlayMenu` widget odebere a znovu se přidá na viewport `MainMenu` widget.

4.1.3 ResetMenu Widget

Designově velmi podobný `PlayMenu`. Widget slouží k resetování uložených pozic. Nejprve zkontroluje, zda tato uložená pozice existuje. Jestliže ano, tak ji vymaže, jestli ne, nedělá nic.

4.1.4 SettingsMenu Widget

Tento widget 12 slouží k zobrazení a ukládání změn nastavení hry. Lze konfigurovat grafické nastavení, hlasitost zvuku nebo jazyk. Základní vady zraku lze kompenzovat nastavením typu barvosleposti. V Unreal Enginu jsou některé z těchto hodnot ukládány do třídy `UGameUserSettings` a přetrvají ukončení programu. Ne však hlasitost a nastavení barvosleposti. Tento nedostatek jde například obejít tím, že se tyto hodnoty uloží do třídy `SaveGame_Game`. Také jsem si rozšířil třídu nastavení a napsal v C++ vlastní třídu `UGameSettings`, která

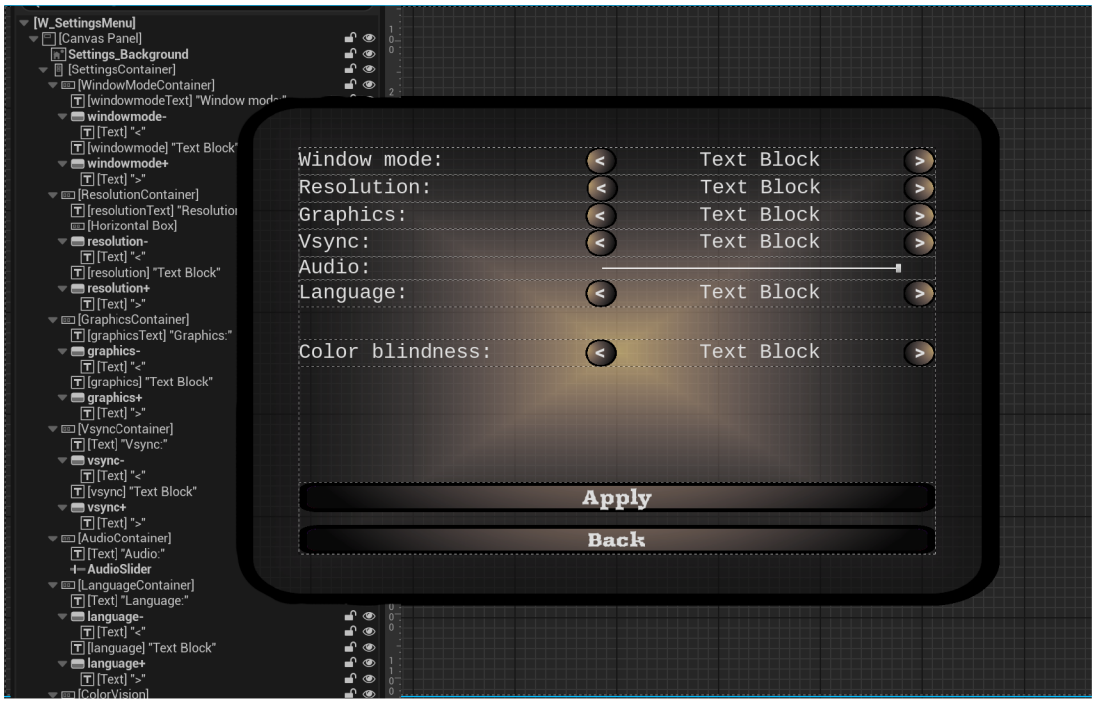
dědí z `UGameUserSettings` a obsahuje navíc tyto hodnoty společně s gettery a settery. Tímto tedy jsem tedy původní třídu rozšířil a přitom zachoval její funkčnost.

```
1  UGameSettings::UGameSettings(const FObjectInitializer&
       ObjectInitializer) : Super(ObjectInitializer)
2  {
3   AudioVolume = 1.0f;
4  }
5
6  void UGameSettings::SetAudioVolume(float audiovolume)
7  {
8   AudioVolume = audiovolume;
9  }
10
11 float UGameSettings::GetAudioVolume() const
12 {
13  return AudioVolume;
14 }
15
16 void UGameSettings::SetColorBlindnessIndex(int colorblindnessindex)
17 {
18  ColorBlindnessIndex = colorblindnessindex;
19 }
20
21 int UGameSettings::GetColorBlindnessIndex() const
22 {
23  return ColorBlindnessIndex;
24 }
25
26 UGameSettings* UGameSettings::GetExpandedGameSettings()
27 {
28  return Cast<UGameSettings>(UGameUserSettings::GetGameUserSettings
        ());
29 }
```

Zdrojový kód 1: C++

Nahrávání hodnot a jejich ukládání se ukázalo být mnohem složitější, než jsem čekal. Při přidání tohoto widgetu do viewportu se musí již zobrazit místo textu Text Block naposledy vybrané hodnoty. Proto se hned nastavení načte, hodnoty se uloží do proměnných a na jejich základě se zobrazí text. Všechna slova v projektu jsem také pomocí nástrojů na lokalizaci [11] přeložil a nastavil tak, že je jazyk možné změnit z angličtiny na češtinu.

Dále je třeba aktualizovat stav všech tlačítek ve scéně podle příslušných hodnot., tlačítko může mít dva stavy, **enabled** a **disabled**. Ve stavu **disabled** je zašedlé a nelze na něj kliknout. Funkce, nastavující stav tlačítek, `SetButtonsStatus`, bere jako vstup index (jaký index má právě zvolená hodnota), dále reference na tlačítka, u kterých status nastavuje, a maximální index,



Obrázek 12: Obrázek z widgetu třídy SettingsMenu

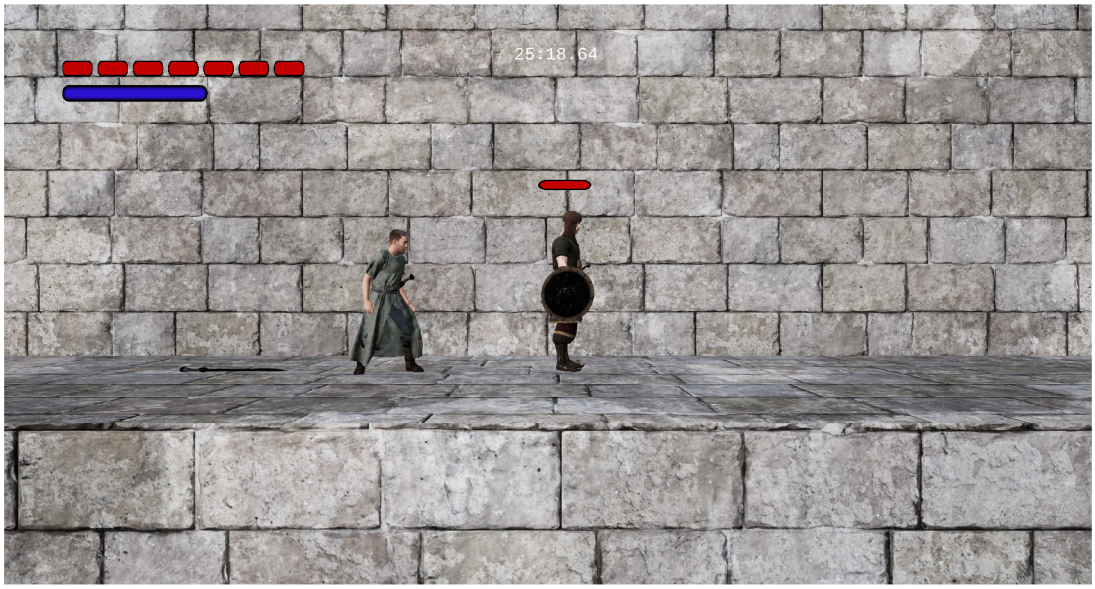
jaký může konkrétní nastavení mít. Při stisknutí levého tlačítka se index sníží a při stisknutí pravého zvýší.

Vybrané nastavení se aplikuje stisknutím tlačítka `Apply`. Pomocí něj se jednak tyto hodnoty uloží do třídy `UGameSettings` a `SaveGame_Game`, jednak se aplikují. Tlačítkem `Back` se hráč vrátí do hlavního menu.

4.1.5 ExtrasMenu Widget

`ExtrasMenu` widget 13 slouží ke změně zbraně, kterou `Prince` v levelu používá. První zbraň, dýka, se odemkne poté, co hráč hru vyhrál. Druhá zbraň, meč, se odemkne poté, co hráč hru vyhrál 10x. Lze si vybrat ze 2 druhů mečů a 2 druhů dýk. Tento výběr nemá vliv na sílu útoku, ani vzdálenost, ze které `Prince` nepřítele může zasáhnout. Velmi snadno bych mohl zbraní přidat více. V jedné fázi práce to tak i bylo, ale do projektu mi neseděly, proto jsem je odebral. V rámci testování jsem měnil u `Prince` i oblečení, 14 u kterého bylo dokonce možné měnit barvy. Šla by naprogramovat funkcionalita, aby tyto faktory ovlivňovaly například to, kolik životů Prince má, nebo jaké poškození způsobuje nepřátelům, do výsledné hry mi to ale nesedělo. `StaticMesh` oblečení není součástí projektu. `StaticMesh` zbraní také není součást projektu, ale pochází z momentálně zadarmo dostupného balíčku z Epic Games Marketplace. Odkazy jsou v `README.txt`. V `ExtrasMenu` je také počítadlo vítězství a povinná copyright informace, která musí být ve všech publikovaných projektech.

Obrázek 13: Obrázek z widget třídy ExtrasMenu



Obrázek 14: Obrázek ukazující Prince v jiném oblečení

Při stisknutí tlačítka `Swords`, respektive `Daggers`, se vytvoří nový widget, `W_Extras_Swords`, respektive `W_Extras_Daggers`. 15 Informace o výběru nové zbraně se uloží do `SaveGame_Game`. Aby ale `Prince` tuto novou zbraň mohl používat, musí ji najít v levelu a zvednout ji. Toto je záměrné.

Obrázek 15: Obrázek ukazující výběr dýky

4.2 Uživatelské rozhraní v levelech

Při startu každého levelu se objeví widget `LevelNameText` zobrazující jméno aktuálního levelu. Při jeho vzniku se přehrává animace `TextFadeOut`, která způsobí, že text po chvíli zmizí.

4.2.1 PauseMenu Widget

Je widget, ve kterém lze hru pozastavit. Když je hra v tomto stavu, časovač neběží a jiné události v levelu jsou také pozastaveny. Tlačítkem `Resume` se hráč vrátí zpět do Levelu. Stisknutím tlačítka `RestartLevel` se restartuje aktuální level. Pomocí `BackToMenu` se hráč vrátí zpět do hlavního menu a `Quit` hru vypíná.

Widget má rozmazané pozadí, z mé zkušenosti často užívané ve hrách v pause menu.

4.2.2 Widgety hráče

Widget `TimeWarpingStamina` má modrou barvu. Jeho vyplnění výraznější modrou a jeho šířka se mění podle `TimeStamina` parametru `Prince`. Jestliže `Prince` zpomaluje čas, výplň postupně ubývá a tak ukazuje hráči, jak dlouho může ještě čas zpomalovat. Pokud `Prince` vypije lektvar na zvýšení hodnoty `TimeStamina`, šířka `ProgressBar` (jeden z prvků ve widgetech) se taktéž zvýší. Tato logika je ale řešena až v `BP_Prince`, a to na začátku při načtení levelu a poté vždy v případech, kdy je `TimeStamina` zvýšena. Všechny textury tohoto widgetu jsem vytvořil v programu **GIMP**, to stejné platí i pro ukazatel

životů Prince. Widget HealthHUD funguje téměř na totožném principu jako TimeWarpingStamina widget.

Widget Timer 16 je reprezentace časovače ukazujícího, kolik času má hráč ještě na to, aby hru dohrál. Jeho text je nastaven v BP_PrinceGameMode. Po vypršení časového limitu se vytvoří widget TimerRanOut a přidá se do scény. Hra se pozastaví, přehraje se zvuk, pozadí se rozmaže a pomocí TimeLine [12] widget komponenty se postupně ztmavuje. Tyto komponenty v projektu hojně používám.

Widget DeathText zobrazí text v případě, že Prince ve hře zemřel. Vzhledově platí to samé jako pro widget TimerRanOut. Level se restartuje.

Obrázek 16: Obrázek z widgetu DeathText

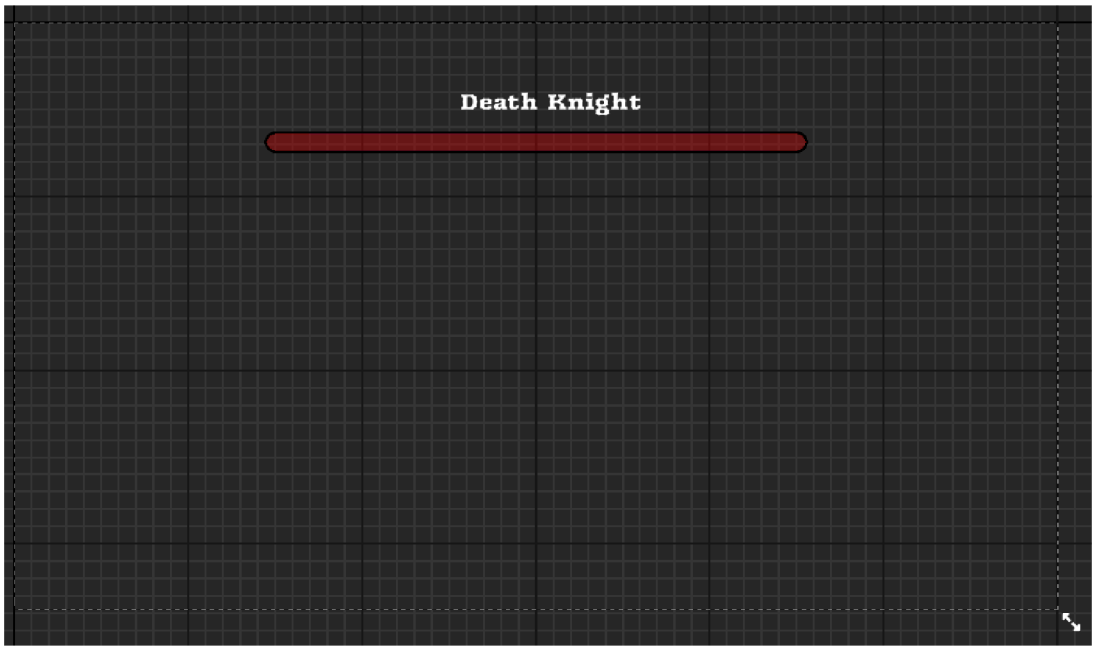
4.2.3 Widgety nepřátel

EnemyHealthBar 17 je widget, který zobrazuje zdraví nepřátel. I tyto textury jsem vytvořil v programu **GIMP**. Tento widget je vázán na pozici nepřítele a zobrazuje se na pozici, kde nepřítel stojí. Widget BossHealthbar 18 zobrazuje zdraví BP_Enemy_Knight. Zobrazuje se přímo na obrazovce a není vázán na lokaci nepřítele.

4.3 PrinceGameMode a související třídy

Pro uložení postupu hráče levely a uložení informací o statistikách Prince používám třídy typu SaveGame. Bez uložení by hráč hru začínal hrát pokaždé znovu, bez předchozího postupu.

Obrázek 17: Obrázek – ukázka healthbar nepřítele



Obrázek 18: Obrázek z widgetu BossHealthbar

4.3.1 Třídy pro ukládání dat

Pro zachování dat i po skončení hry používám v projektu primárně dvě třídy: `BP_Savegame_Game`, která má jen jednu instanci a dále `BP_Savegame_Level`, která má tři instance. Každá instance slouží pro jeden slot, který se hráč může

zvolit za pomocí widgetu PlayMenu. Obsahuje data, která jsou důležitá pro status Prince v levelu.

BP_Savegame_Game slouží pouze k uložení informace o tom, jakou zbraň zrovna Prince používá. Obsahuje proměnné PlayerSword a PlayerDagger, které referují na vybranou StaticMesh.

Proměnná v BP_Savegame_Level Score typu integer, zůstala nepoužitá, protože jsem usoudil, že se do tohoto projektu, nehodí. Jako skóre se dá počítat nejlepší čas, za který hráč všemi levely projde. Velmi jednoduše jde ale v projektu funkcionalitu počítání Score povolit.

Další proměnnou je CurrentLevel typu enum (E_Level_Name), která obsahuje informaci o tom, jaký level hráč zrovna hraje. Tento enum jsem vytvořil a obsahuje jména všech levelů ve hře.

LevelTimersInfo je pole struktur typu S_Shadow_Timer, které jsem vytvořil a které se skládá ze čtyř proměnných:

- LevelName typu FString .Slouží k přiřazení instance ShadowTimer k levelu.
- ShadowExists typu boolean. Nese informace o tom, zda v daném levelu již Shadow existuje.
- ShadowPoses typu pole PoseSnapshots. Pózy Shadow uložené v poli.
- ShadowLocations typu Transform (3 vektory, rotace, lokace, velikost)

Nemožnost využití funkcionality Shadow_Timer mě na tomto projektu mrzí nejvíce. Dalo mi to hodně práce a funguje podle mých představ, má však jednu chybu. Instance Shadow_Timer si totiž každý snímek pomocí funkce SnapshotPose ukládá pózu Prince do pole a s tím i jeho lokaci a rotaci. Při druhém průchodu levelem se děje to stejné, a navíc se ještě ukazuje minulý, respektive nejlepší pokus průchodu levelem. Tato logika funguje bezchybně a ani snímky ve hře nejsou ovlivněny, alespoň ne takovým způsobem, aby to šlo zaznamenat, optimalizace je tedy dobrá.

Problém nadešel při testování, kdy dvě minuty používání instance Shadow_Timer, kdy se ukládala Transform aSnapshotPose Prince, měla výsledná SaveGame_Game okolo 500 MB a hra se načítala opravdu dlouho. Snažil jsem se tento problém obejít, ale v rozumném čase jsem řešení nenašel. Rozhodl jsem se tedy tuto funkci do výsledné hry nepřidat. Náznaky jiných možných řešení jsem ale v dokumentaci našel, takže věřím, že je nějakým způsobem implementovatelná. Bylo mi líto Shadow_Timer z projektu úplně smazat, proto jsem ho tu ponechal a logika je k dispozici k nahlédnutí. Věřím, že mít více času, přijdu na jiný způsob, jak Shadow_Timer vytvořit.

Důležitým parametrem třídy BP_Savegame_Level je TimeRemaining typu float, který obsahuje informaci o tom, kolik času ještě zbývá.

A posledním parametrem je struktura typu PlayerInfo skládající se z šesti proměnných:

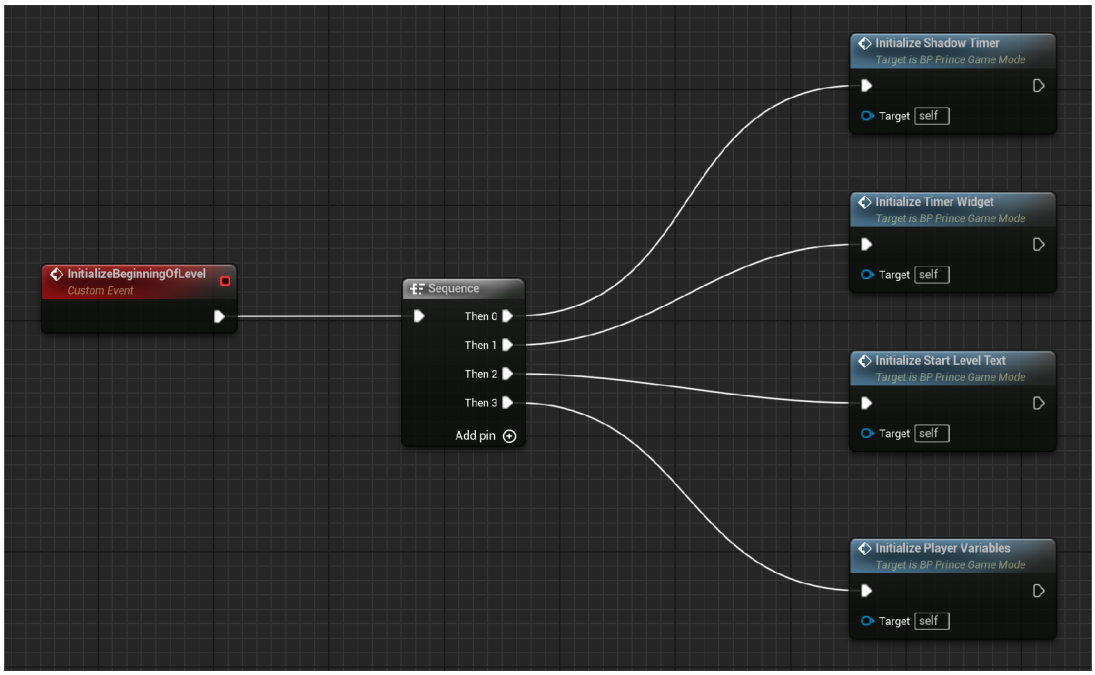
- `PlayerHealth` typu `float`. Údaj o hráčově maximálním zdraví, tato hodnota se může během hraní levelu zvýšit. Je třeba ji ukládat.
- `PlayerStaminaIndex` typu float. Podobně jako hráčovo zdraví je `PlayerStaminaIndex` hodnota určující časový interval, ve kterém může Prince zpomalovat čas. Hodnota se může v levelu zvýšit.
- `SwordStaticMesh` a `DaggerStaticMesh` typu `StaticMesh`. Na začátku, než `Prince` zdvihne meč, jsou prázdné. Poté je to `StaticMesh` získaná z reference z `BP_Sword`.
- `CanEnterAttackState` a `CanEnterTimeWarpState` typu `boolean`. Než `Prince` získá schopnost bojovat/zpomalovat čas, jsou tyto hodnoty **false** a hráč tedy nemůže útočit, respektive zpomalovat čas.

4.4 PrinceGameMode

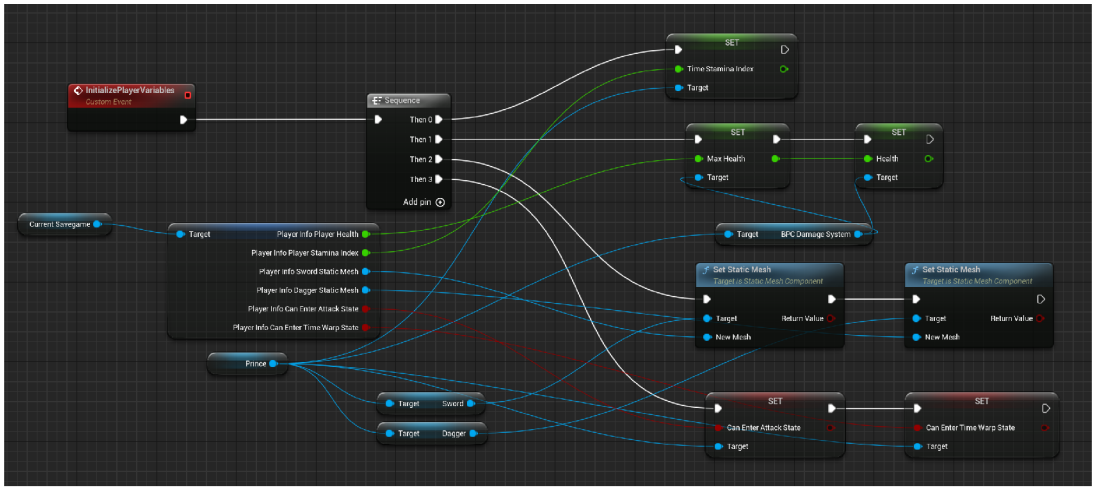
Hlavní menu používá třídu `MenuGameMode`, v ostatních levelech je to `PrinceGameMode`. V této třídě je velké množství funkcionality související s počátečním načtením levelu a také jeho ukončením a uložením informací.

Při vzniku této třídy 19 se zjistí hodnota `CurrentSavegame` ze třídy `GameInstance`. Díky tomu víme, na jaký slot hráč klikl a jaký `SaveGame_Level` se momentálně používá. Také se uloží reference na `BP_Prince`, což je `Pawn` používaný v levelech. Následně se zavolá událost (`Event`, obdoba metody) [13] `InitiliazeBeginningOfLevel`. Ta v sekvenci zavolá další události, které popíši: 20

- `InitializeShadowTimer`. Vytvoří novou instanci třídy `BP_ShadowTimer` a uloží její referenci proměnné. Pomocí `Event Dispatchers`, [14] jedné z funkcionalit Unreal Enginu, zajistí, že se stín pohybuje normální rychlostí, i když `Prince` zpomaluje čas.
- `InitializeTimerWidget`.Vytvoří `Timer` a nastaví jeho počáteční čas na hodnotu `TimeRemaining` získanou z reference na aktuálně používanou `SaveGame`. Poté se `Timer` přidá do scény a uloží se na něj reference do proměnné.
- `InitializeStartLevelText`. Vytvoří se `LevelNameText` widget, přidá se do scény a přehraje se zvuk. Po krátké době se widget odebere, aby nezabíral paměť, není již potřeba.
- `InitializePlayerVariables`. Hodnoty z `BP_Savegame_Game` se předají referenci na `Pawn Prince` a nastaví se.



Obrázek 19: Obrázek metody InitializeBeginningOfLevel



Obrázek 20: Obrázek metody InitializePlayerVariables

4.4.1 Ukončení levelu

Důležité je zmínit událost `LevelEnded`. Tato událost se stane v případě kdy se mění level, hráč došel na konec. Udělá se v podstatě opak toho, co v `InitializePlayerVariables`, hodnoty z `Pawn` se v `PrinceGameMode` předají a uloží do `SaveGame`. Zvýší se také hodnota `enum CurrentLevel` v `SaveGame`.

Dále se také zavolá událost `SaveShadowLocationsToSaveGame`, která nejdříve v podmínce zkontroluje, zda pro level již existuje stín (`Shadow`), hráč

úrovní už prošel. Jestliže existuje, uloží informace posbírané napříč levelem o pozici hráče v časovém úseku do `SaveGame_Level`. `Prince` může na konec úrovně dorazit ale později, než `Shadow`, reference na něj již neexistuje. V tomto případě se posbírané informace neuloží.

4.4.2 EndLevel a EndLevel_Last

Jsou blueprint třídy dědící z `AActor`. Jsou reprezentovány jako kolizní oblast v levelu ve tvaru kvádru. Jakmile `Prince` vstoupí do této oblasti, zavolá se událost `LevelEnded` z `PrinceGameMode`. Pokud se do této oblasti dostal dřív `Shadow`, odstraní se.

`EndLevel_Last` se liší v drobnostech. Nastaví se `CurrentLevel` na první level, resetuje se hodnota `Timer` a otevře se `MainMenu` level.

4.4.3 TriggerVolume

Blueprint třída `TriggerVolume` je taktéž kolizní oblast. Funguje tak, že v případě kolize s `Pawn` v předem nastavené lokaci v levelu vytvoří `AActor` předem nastavené třídy. Pro dynamickou tvorbu nových pastí v levelu je to velmi užitečná komponenta.

4.5 Pickups

Pickupy (jiný název - collectibles) jsou častým prvkem, který lze ve hrách, [15] zvlášť skákačkách získat či sbírat. Většinou dávají hráči nějaké výhody či postihy. Tato hra jich také několik obsahuje.

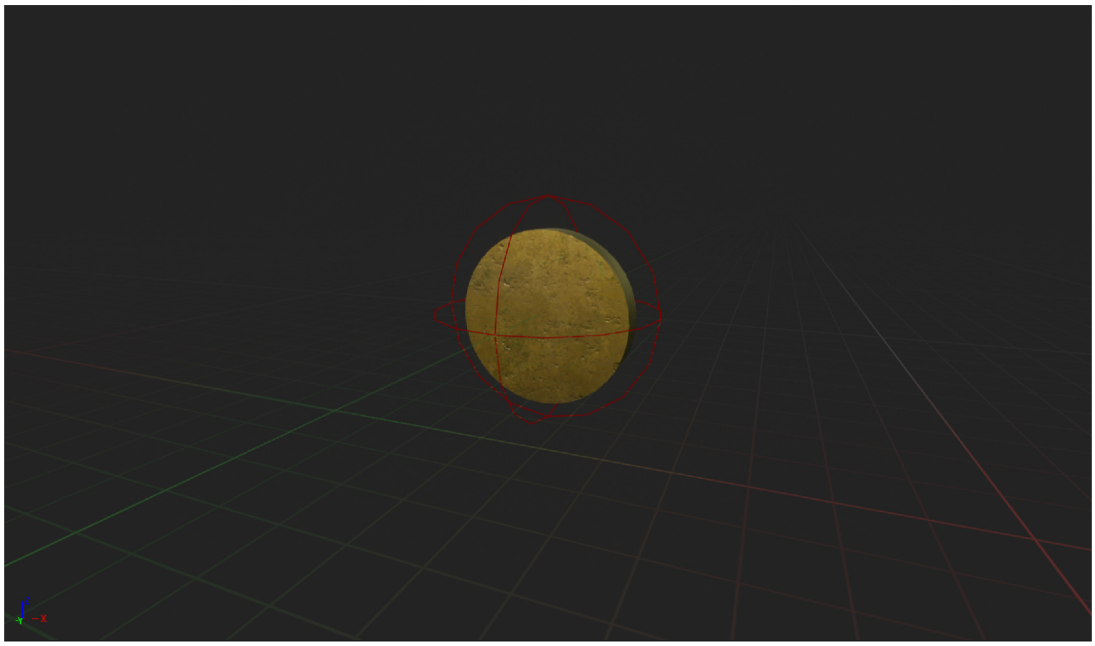
4.5.1 Score

Prvním je blueprint `Score_Base`, který obsahuje nevyužitou funkcionalitu počítání skóre. Ve svém kódu zkontroluje, jestli `AActor` který vstoupil do collision oblasti, je blueprint `Prince` a jestliže ano, zavolá jeho funkci na zvýšení skóre a přehraje zvuk. Tato třída nemá sama o sobě žádnou `StaticMesh` a v levelu by nešla vidět, slouží pouze jako předek.

Blueprint `Coin` 21 dědí z `Score_Base` a již `StaticMesh` obsahuje, stejně jako komponentu `RotatingMovement` způsobující rotaci `AActor` v levelu. V třídě `Coin` stačilo nastavit, kolik bodů získá `Prince`, když vstoupí do červené oblasti na obrázku, a také zvuk, který se přehraje. Zbytek funkcionality je již v `Score_Base`.

4.6 Potion

Na podobném principu funguje i blueprint třída `Potion`. Zkontroluje, jestli `AActor` který vstoupil do kolizní oblasti je blueprint `Prince` a jestliže ano, nastaví mu proměnné `CanDoEAction` a `CurrentOverlappedActorType`



Obrázek 21: Obrázek z blueprint třídy Coin

a `CurrentOverlappedActor` na hodnoty **true**, `Potion` (jedná se o `enum`) a referenci na sebe. V případě opuštění kolizní oblasti se tyto hodnoty vrátí do svého předchozího stavu.

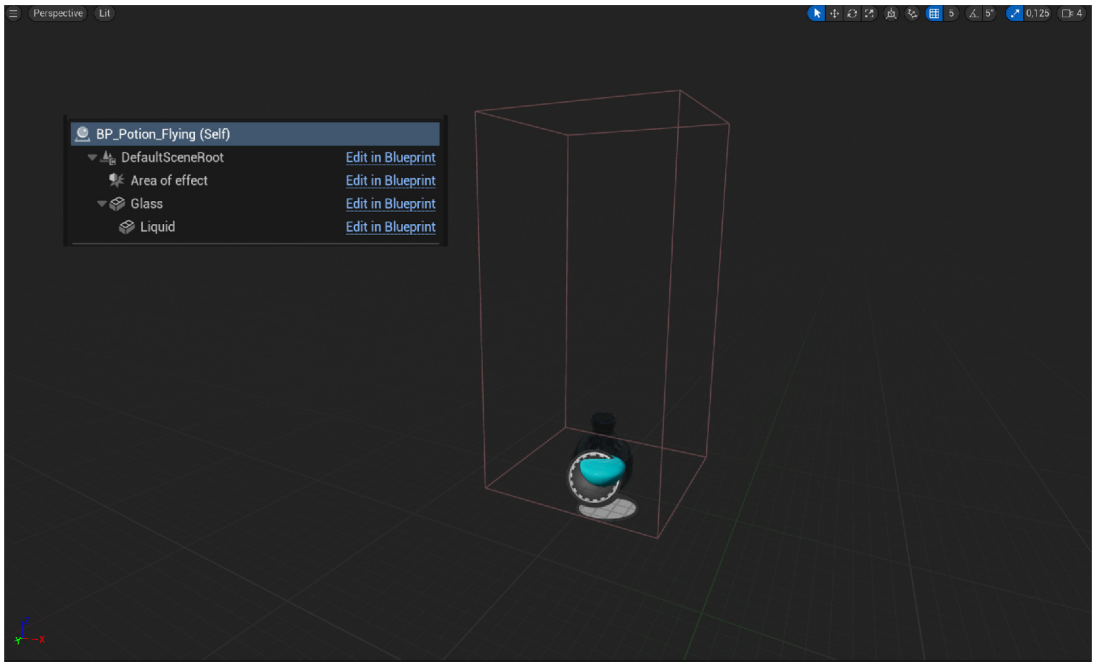
Několik tříd z `Potion` dědí. `Potion_Damage` je třídou, která způsobí, že efekt `Potion` po vypití je zranění hráče. `Potion_Health` dobije hráči život. `Potion_Health_Increase` a `Potion_Time_Stamina_Increase` zvýší u `Prince` maximální množství životů, respektive časové staminy.
`Potion_Flying` 22 zavolá funkci, která na čas sníží hráčovu gravitaci. 3D model lahvičky, včetně kapaliny uvnitř, jsem vymodeloval v programu **Blender**. Materiály jsou vytvořeny za pomoci nástrojů v enginu. [16]

4.7 Sword

Posledním typem `Pickup` objektu je instance blueprint třídy `Sword`. Při vytvoření této třídy, která dědí z `AActor`, se z `BP_Savegame_Game` zjistí jakou `StaticMesh` hráč nastavil, a podle ní se nastaví `StaticMesh` této třídy. Při vstupu `Prince` do kolizní oblasti se jako v minulých případech nastaví hodnoty parametrů této třídy.

4.8 Pasti

Pasti jsou další typickou funkcionalitou patřící ke skákačkám. [15] V této práci jsem jich několik vytvořil.



Obrázek 22: Obrázky z blueprint třídy Potion_Flying

4.8.1 Turret a Arrow

Blueprint třídy dědící z `AActor`, 23 které spolu vzájemně souvisí. Poté co `Character` (může to tedy být i nepřítel) vstoupí do kolizní oblasti této třídy, začne turret střílet šípy. 24 Docílí toho tím, že každých několik vteřin vytvoří instanci třídy `Arrow`. Tuto prodlevu je u každé instance této třídy umístěné do scény možno nastavit individuálně.

Blueprint třída `Arrow` obsahuje komponentu enginu`InterpToMovement`, která zajišťuje, že poté, co instance této třídy vznikne ve scéně, pohybuje se v předem určeném směru po předem určený časový úsek. V tomto projektu jsou tyto hodnoty 5 sekund a vektor.

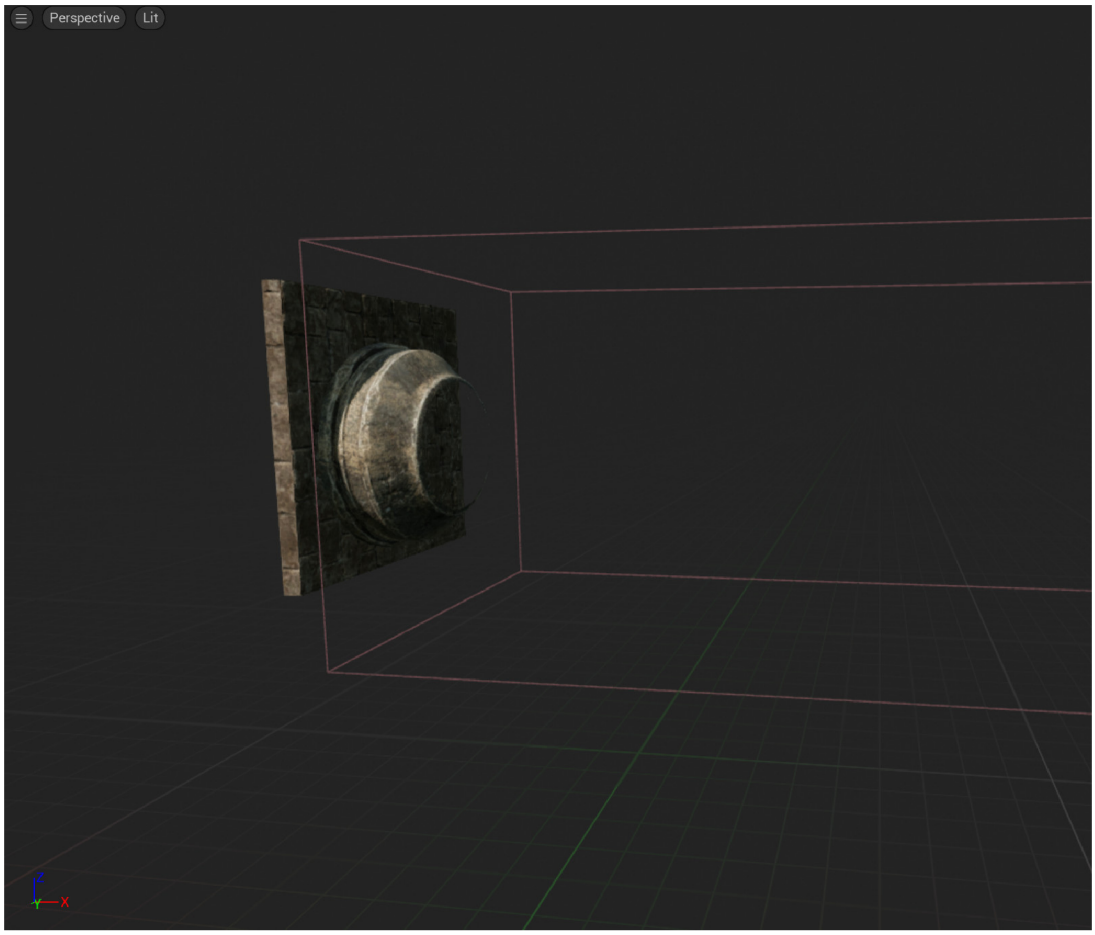
$$v = 2130, 0, 0$$

Velmi jednoduše by ale šly tyto hodnoty nastavit tak, aby byly u každého turretu jiné.

Jestliže se `Arrow` v prostoru střetne s `AActor`, který implementuje rozhraní `Damageable`, tak mu způsobí poškození a poté je zničen. V tomto projektu jsou to všechny typy nepřátel, `Pawn` dědící z `BP_Enemy` i `Pawn Prince`. Jestliže se nestřetne s žádným takovým `AActor` v již zmíněném časovém úseku, zničí se. Model šípu je mnou vytvořený v programu **Blender**. Textury jsou použity z balíčku StarterContent, který je součástí enginu.

4.8.2 Ball

`Ball` je blueprint třídou, která není ovlivněna gravitací, ve scéně model ve vzdu-



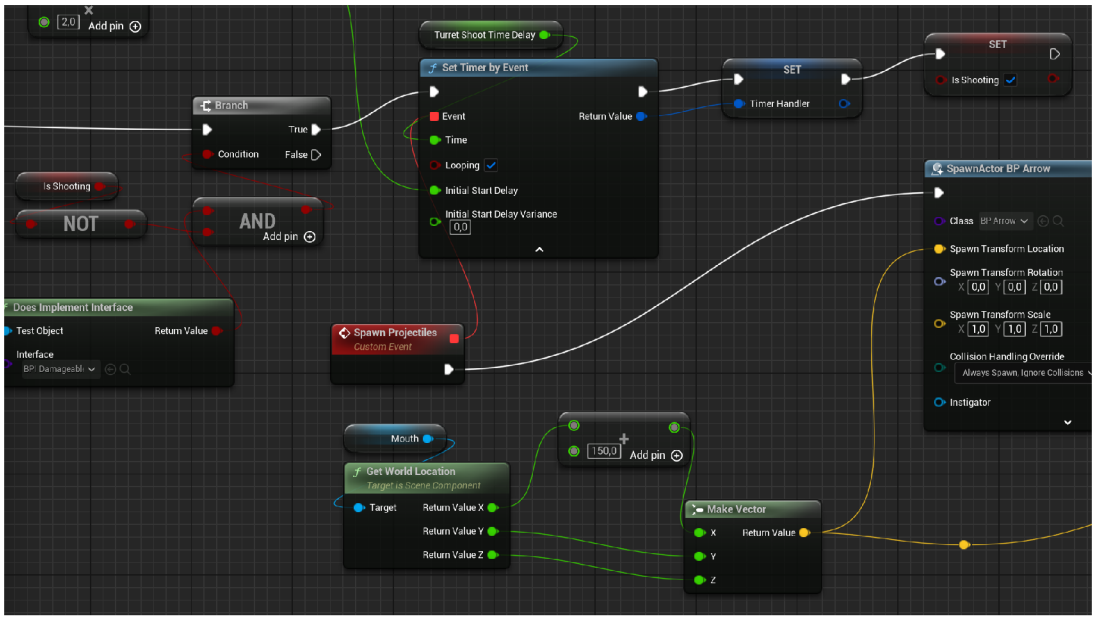
Obrázek 23: Obrázky z blueprint třídy Turret

chu nepadá, dokud se nezavolá událost `OnTriggered`, kterou tento `AActor` implementuje díky rozhraní `Triggerable`. Poté, co se událost zavolá, se gravitace a kolize povolí a objekt je po časové prodlevě zničen, tato prodleva je nastavitelná. Textura i model v této třídě jsou z StarterContent, který je součástí enginu.

Když instance této třídy ve scéně koliduje s jiným `AActor`, který implementuje rozhraní `Damageable`, způsobí mu `Ball` zranění. Lze prodlevu, po které je tento objekt zničen při zavolání `OnTriggered` a také jeho lokaci ve scéně.

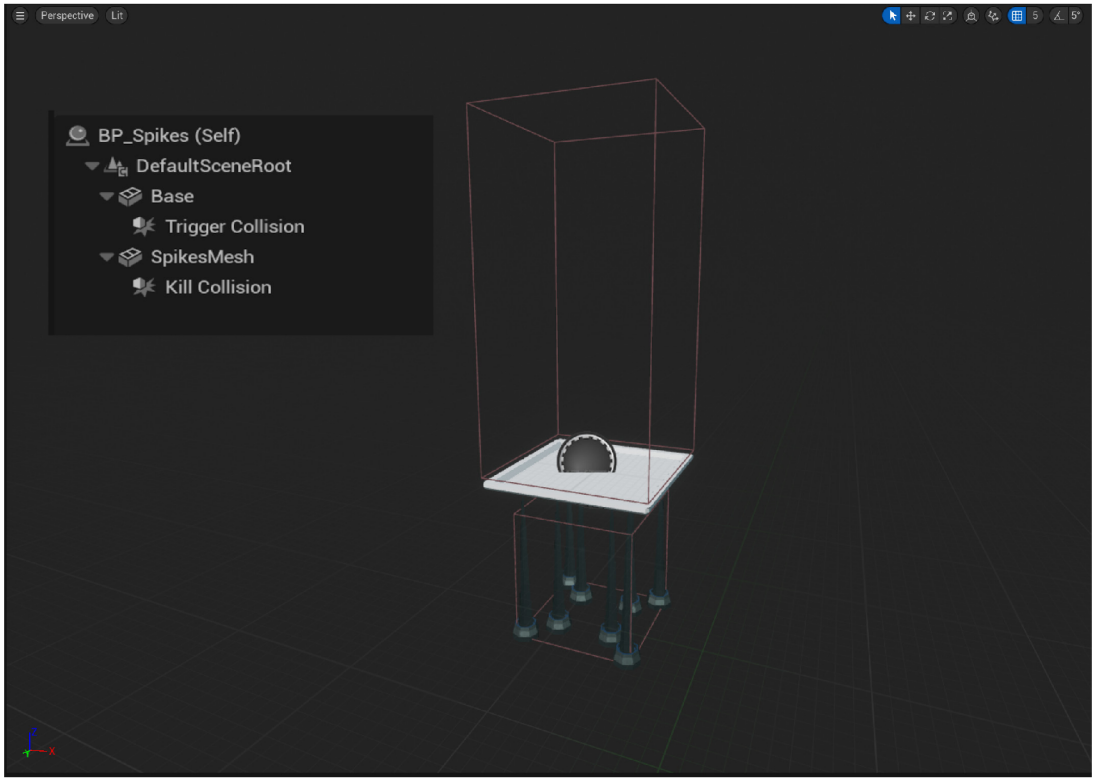
4.8.3 Spikes

`Spikes`, 26 bodláková past, je další blueprint třídou. Obsahuje dvě kolizní oblasti. Jednu nazvanou `TriggerCollision`, do které když vstoupí `Pawn` (i nepřítel), vysunou se bodláky. Když poté `Pawn` koliduje i s druhou oblastí `KillCollision` a zároveň implementuje rozhraní `Damageable`, bodláky mu způsobí zranění. V případě ale, že tímto `AActor` je `Princ`, který zpomaluje čas, (`IsWarpingTime` = **true**), nebo pomocí klávesy Shift chodí pomaleji, bodláky

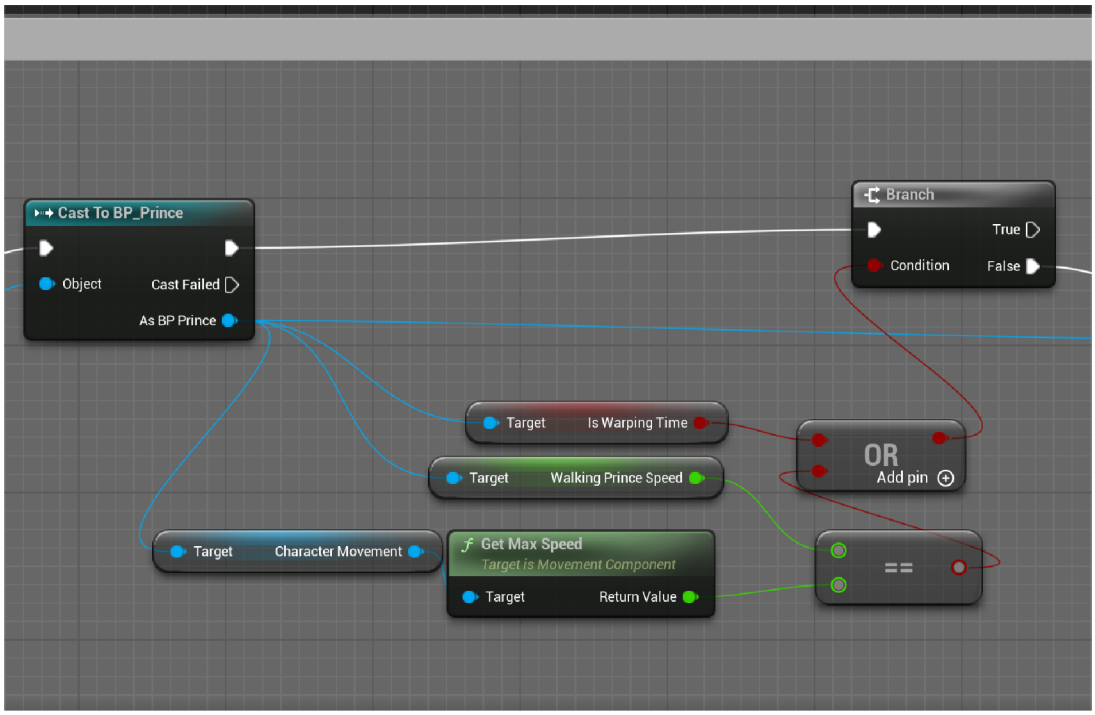


Obrázek 24: Obrázky z části funkce použité v blueprint třídě Turret

mu zranění nezpůsobí. 3D model bodlákové pasti jsem vytvořil v programu **Blender**, textury jsou použité ze StarterContent enginu.



Obrázek 25: Obrázky z blueprint třídy Spikes



Obrázek 26: Obrázky podmínky v blueprint třídě Spikes

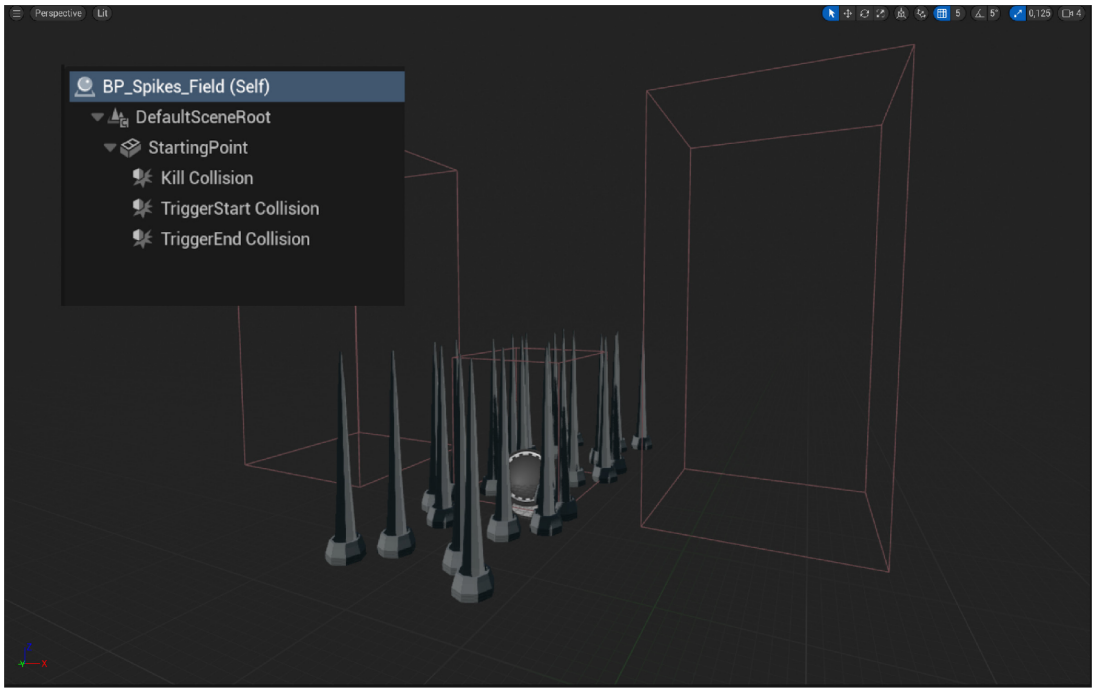
4.8.4 Spikes_Field

Blueprint třída `Spikes_Field` 27 se od třídy `Spikes` velmi liší. Tato třída je navržena tak, aby mohla být v levelu umístěná v libovolném úhlu na zdi, stropu či podlaze. S tím souvisí i dynamická délka `StaticMesh`. Používá sice stejný 3D model a také obsahuje `KillCollision`, avšak má dvě `TriggerCollision`: `TriggerStartCollision` a TriggerEndCollision. Navíc ještě obsahuje `StaticMesh StartingPoint`.

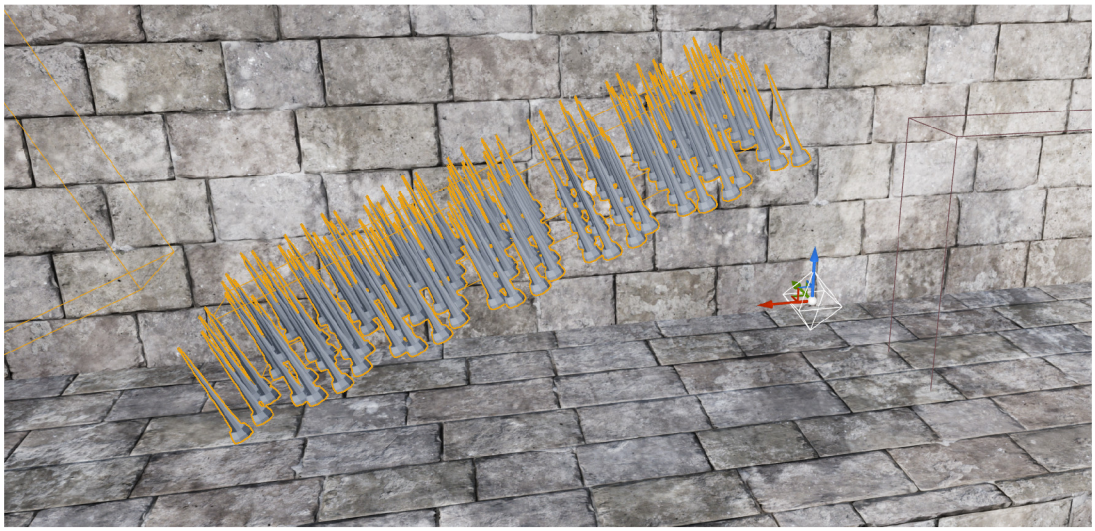
Vektor `EndPoint` je také velmi důležitou proměnnou. Poté co se vývojář umístí instanci této třídy do scény a posune tento vektor, 28 spočítá se vzdálenost mezi `StartingPoint` a `EndPoint`, pak se podle předem hardcodované proměnné `MeshWidth` spočítá, kolik `StaticMesh` (neboli 3D modelů bodláků) se přidá ke komponentě `StartingPoint`. Toto se provádí ve for cyklu pro každou takovou `StaticMesh`, protože se musí ještě nastavit její lokace (relativní k `StartingPoint`) a stejně tak se musí rozšířit i `KillCollision`. V cyklu se také nové `StaticMesh` nastaví materiál.

`TriggerStartCollision` způsobí to, že se bodláky začnou vysunovat, předtím se v levelu nevysouvají. Lokace a velikost `TriggerStartCollision` jsou libovolně nastavitelné u každého instance této třídy ve scéně. Stejné platí pro `TriggerEndCollision`, která způsobí to, že se bodláky vysouvat přestanou.

Zmíním ještě proměnné `SpikesUpDuration` a `SpikesUpSpeed`, které určují to, jakou rychlostí se budou bodláky vysouvat a jak dlouho zůstanou nahoře. Opačně to platí pro proměnné `SpikesDownDuration` a `SpikesDownSpeed`.



Obrázek 27: Obrázky z blueprint třídy Spikes_Field



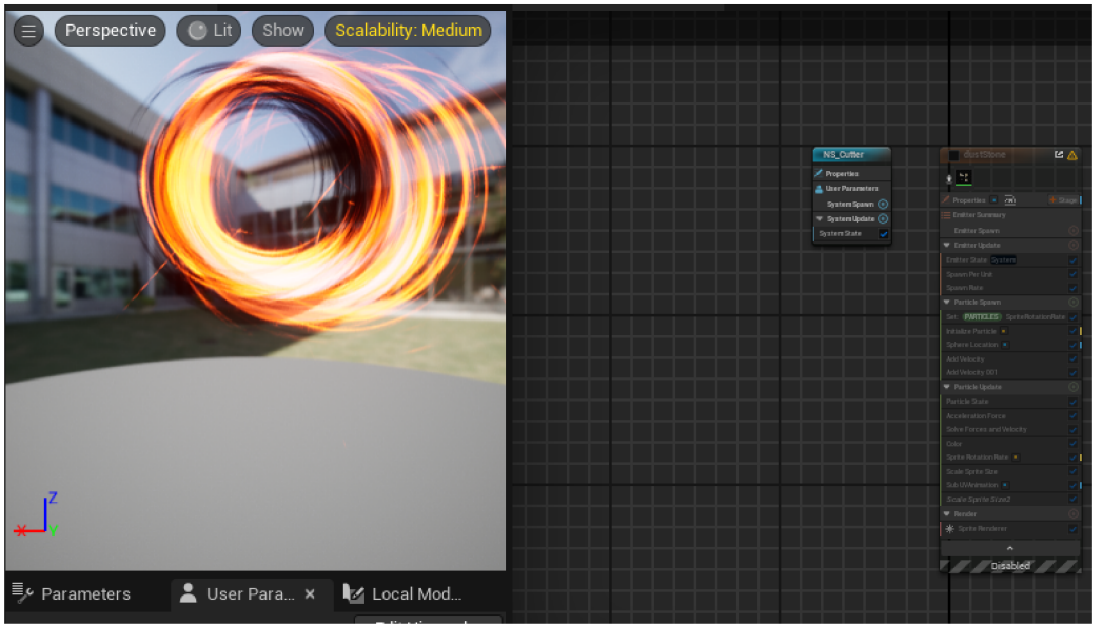
Obrázek 28: Obrázky - ukázka vektoru EndPoint patřící třídě Spikes_Field

I tyto proměnné jsou libovolně nastavitelné pro každou instanci této třídy.

4.8.5 Cutter

`Cutter` je blueprint třídou, která obsahuje proměnnou komponentu **Niagara System**, 29 což je částicový systém používaný v Unreal Engine. [17] Až na vizuály je jeho funkcionalita totožná s třídou `BP_Arrow`. `Pawn` je ale zraněn

o více životů. Vizuální efekt pochází z Epic Game Marketplace.



Obrázek 29: Obrázky Niagara System používaného v blueprint třídě Cutter

4.8.6 Další Niagara System třídy

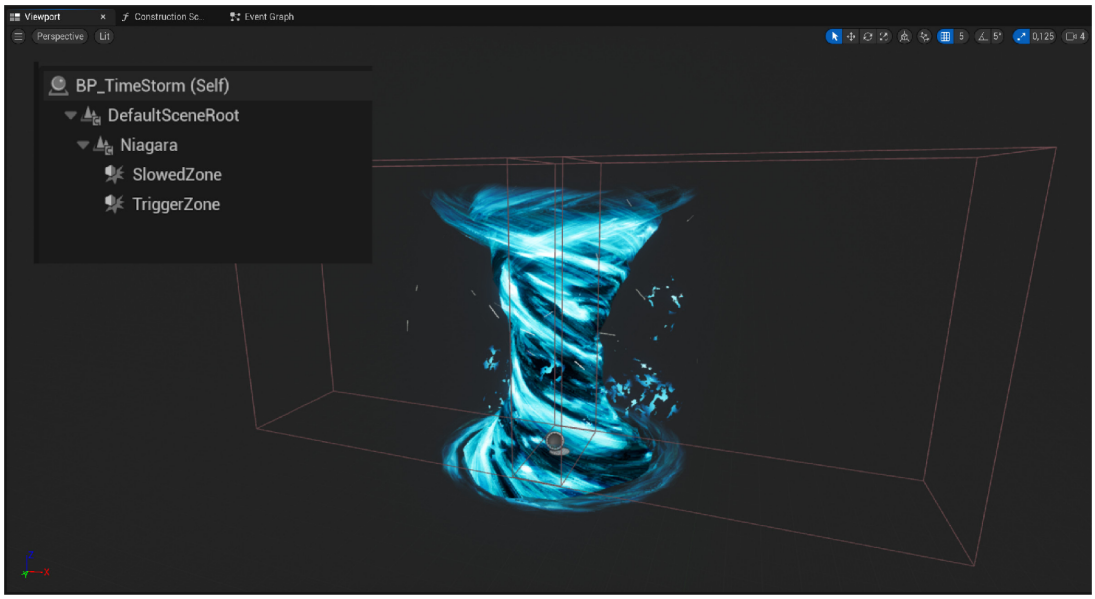
Popíšu nyní ještě další třídy, které v projektu využívají Unreal Engine **Niagara System**. Blueprint třída `TimeStorm` 30 znázorňuje modrý vír, skládá se z **Niagara System** instance a dvou kolizních oblastí, SlowedZone a TriggerZone. Poté, co Prince vstoupí do oblasti `SlowedZone`, hra se zpomalí a přehrává se zvuk. Jakmile tuto oblast opustí, vrátí se rychlost hry do normálu a zvuk se zastaví. Pokud již `Prince` může čas zpomalovat (hráč hraje hru podruhé, nebo do oblasti podruhé vstoupil), v této oblasti schopnost fungovat nebude.

Pouze když `Prince` vstoupí poprvé do oblasti `TriggerZone`, znamená to, že získá schopnost zpomalovat čas. V této oblasti nemůže chvíli nic dělat a přehraje se také zvuk. Poté získá schopnost zpomalovat čas. Může ji použít, jakmile opustí oblast `SlowedZone`. **Niagara System** efekt pochází z Epic Games Marketplace.

Blueprint třída `TimeWarpEffect` 31 je třídou slouženou pouze z upraveného **Niagara System** efektu z Epic Games Marketplace. `Pawn Prince` (BP_Prince) má v komponěntě na instanci této třídy referenci. Je to samostatná třída, protože to zmenšuje složitost a zlepšuje přehlednost komponent.

4.9 Buttons, Elevators, Platforms

Tlačítka jsou další položkou hojně používanou v plošinových hrách. [15] Tlačítka v tomto projektu a `ButtonDoor` implementují `ButtonInterface`.



Obrázek 30: Obrázky z blueprint třídy TimeStorm

Obrázek 31: Obrázky z Niagara System Timewarp

`BP_Bouncer` je v projektu nevyužitá, ale funkční třída, která vystřelí `Prince` do vzduchu.

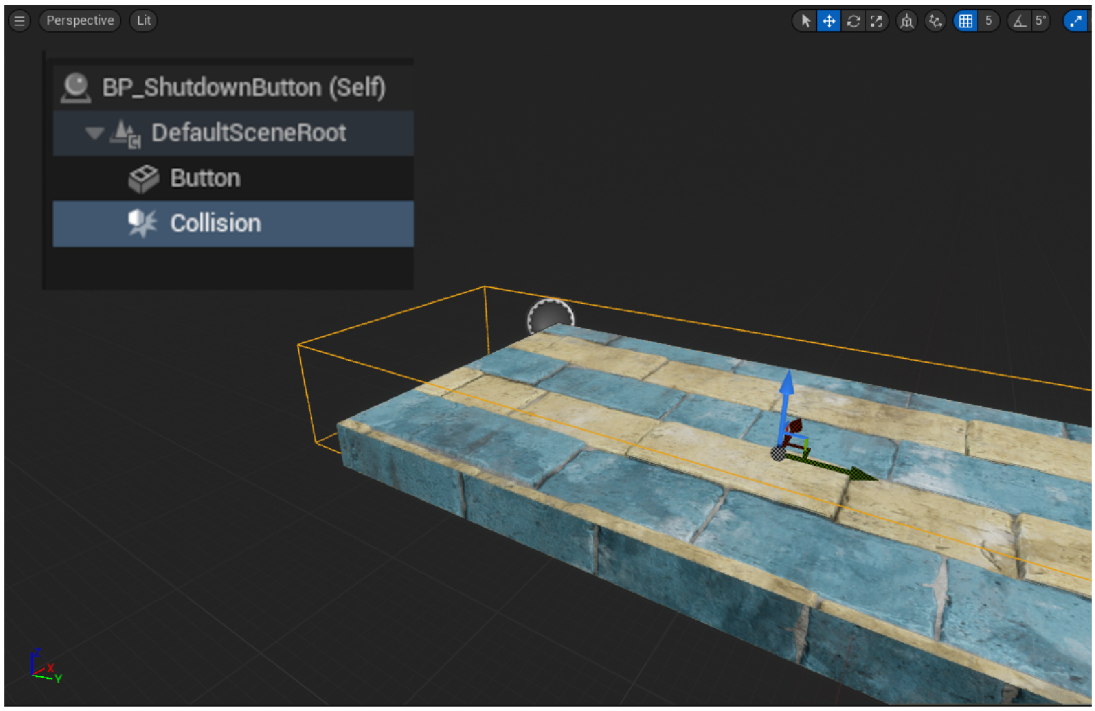
4.9.1 PressingButton a ShutdownButton

Při vstupu třídy `Pawn` (`Prince` i nepřátel) do kolizní oblasti tlačítka se stlačí dolů, přehraje se zvuk a také se pošle zpráva `OnButtonClicked` všem `AActors`

v proměnné `AffectedActors`, což je pole referencí na další `AActors` ve scéně. Reference do `AffectedActors` se přidají až k instanci této třídy ve scéně, takže jedno `PressingButton` může poslat zprávu `OnButtonClicked` několika `AActors`. Poté, co `Pawn` opustí kolizní oblast, tlačítko se vrátí do své původní lokace a pošle zprávu `OnButtonLeft` všem `AffectedActors`.

`ShutdownButton` 32 funguje téměř identicky jako `PressingButton`, je ale používáno na více účelů. Posílá se buď zpráva `OnButtonSinglePressed`, nebo `OnTriggered`.

3D model těchto tlačítek je použit z StarterContent enginu a textury jsou použity z Epic Games Marketplace.

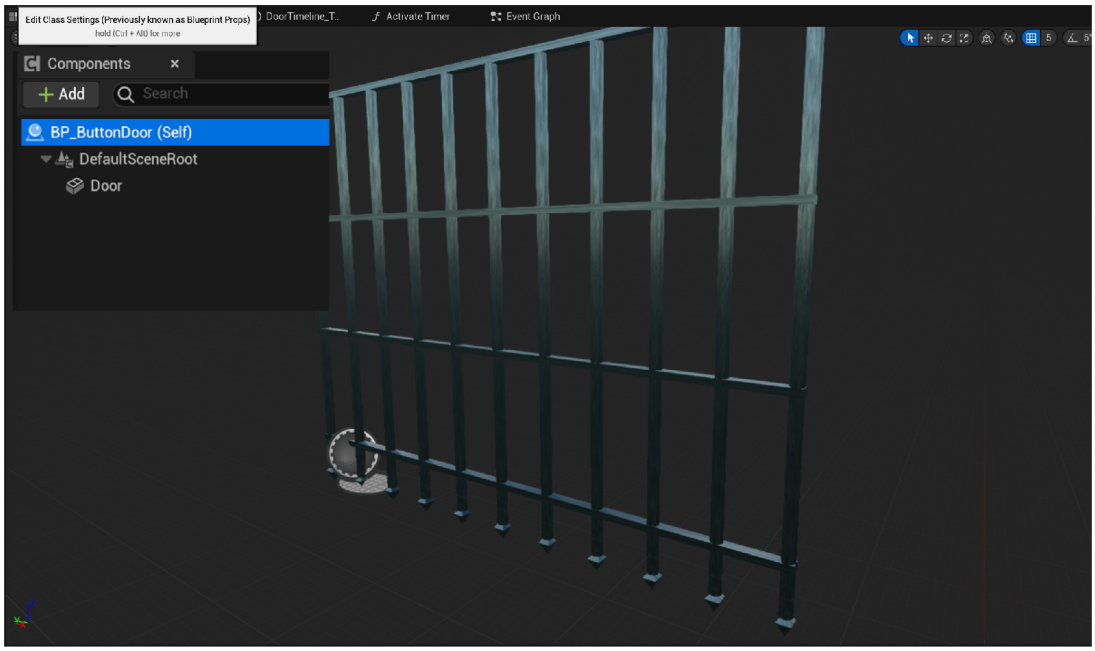


Obrázek 32: Obrázky z blueprint třídy ShutdownButton

4.9.2 ButtonDoor

ButtonDoor 33 je blueprint třída se `StaticMesh` mříží, které jsem vytvořil v programu **Blender**. Materiál je z StarterContent. Tato třída zpracovává zprávy, které jí zasílají tlačítka. Podobně jako u `Spikes_Field` lze i v tomto případě nastavit rychlost, s jakou vyjede, jak dlouho zůstane nahoře a rychlost s jakou pojede dolů, a to u každé instance této třídy ve scéně zvlášť.

Při zprávě `OnButtonClicked` se mříž pomocí `Timeline` [12] posune v čase nahoru. `OnButtonLeft` se mříž po prodlevě spustí dolů a při zprávě `OnButtonSinglePressed` se přeruší všechny zvuky, které se přehrávají, spustí se zvuk spuštění mříže dolů a mříž se spustí dolů hned z místa, kde se momentálně nachází.



Obrázek 33: Obrázky z blueprint třídy ButtonDoor

4.10 Elevator a Elevator_Falling

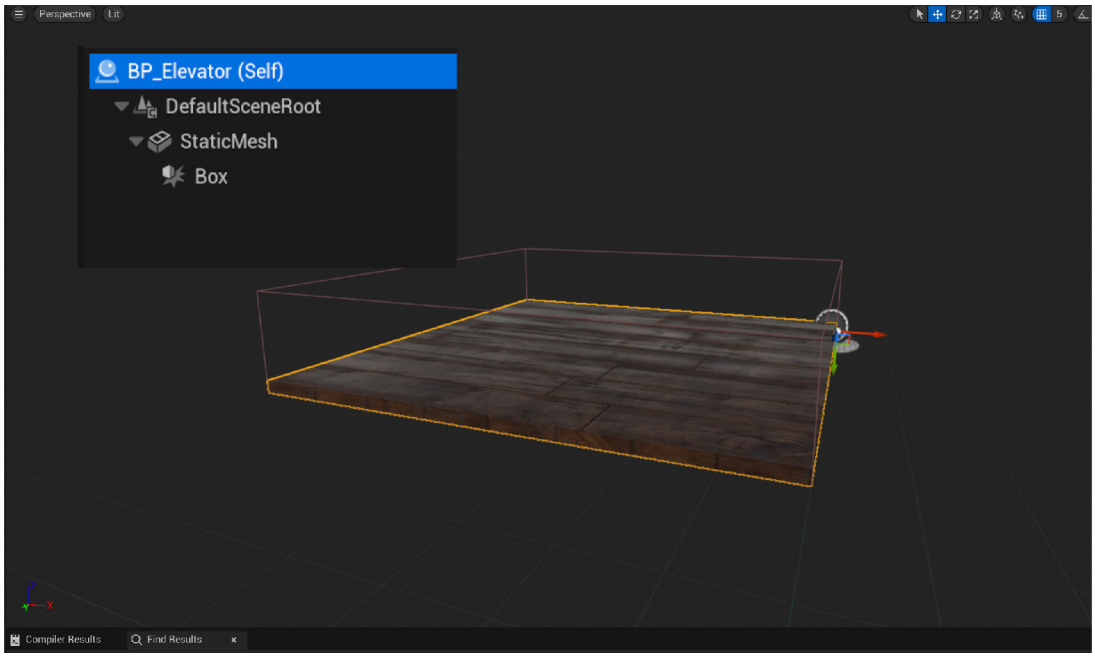
Blueprint třída `Elevator` slouží k přemístění `AActors` ve směru a vzdálenosti určitého vektoru za daný čas. Poté, co nastane průnik `AActor` s kolizní oblastí, `Elevator` se začne posunovat a nepřestane, dokud se nedostane do konečné lokace určené startovní pozicí, k níž je přičten vektor `VectorToMove`. Vektor a čas jsou pro každou instanci této třídy ve scéně nastavitelné. Tato plošina může tedy cestovat v libovolném směru i úhlu. Poté, co se `Elevator` dostane do konečné lokace, vrátí se do původní lokace. `StaticMesh` je ze StarterContent a materiál taktéž.

Funkčnost `BP_Elevator_Falling` je téměř identická s `BP_Elevator`, 34 s tím rozdílem, že pokud `AActor` opustí kolizní oblast, `Elevator_Falling` se začne okamžitě vracet do startovní lokace bez ohledu na to, jestli dorazil do konečné destinace, či nikoli. V tomto projektu je instance této třídy použita pro cestu prostorem směrem dolů, proto název třídy `Elevator_Falling`, může být však použit pro libovolný směr. `StaticMesh` je ze StarterContent a materiál pochází z balíčku z Epic Games Marketplace.

4.10.1 Platform

Blueprint třída `Platform_Dissappearing` je třídou, která zůstala v projektu nevyužitá. Její funkcionalita spočívá v tom v tom, že při kolizi `AActor` s kolizní oblastí instance této třídy po časovém úseku zmizí. Ve vývoji této práce byla nahrazena `BP_Platform_Trigger`.

Blueprint třída `Platform_Trigger` 35 obsahuje `StaticMesh` ze Star-



Obrázek 34: Obrázky z blueprint třídy Elevator

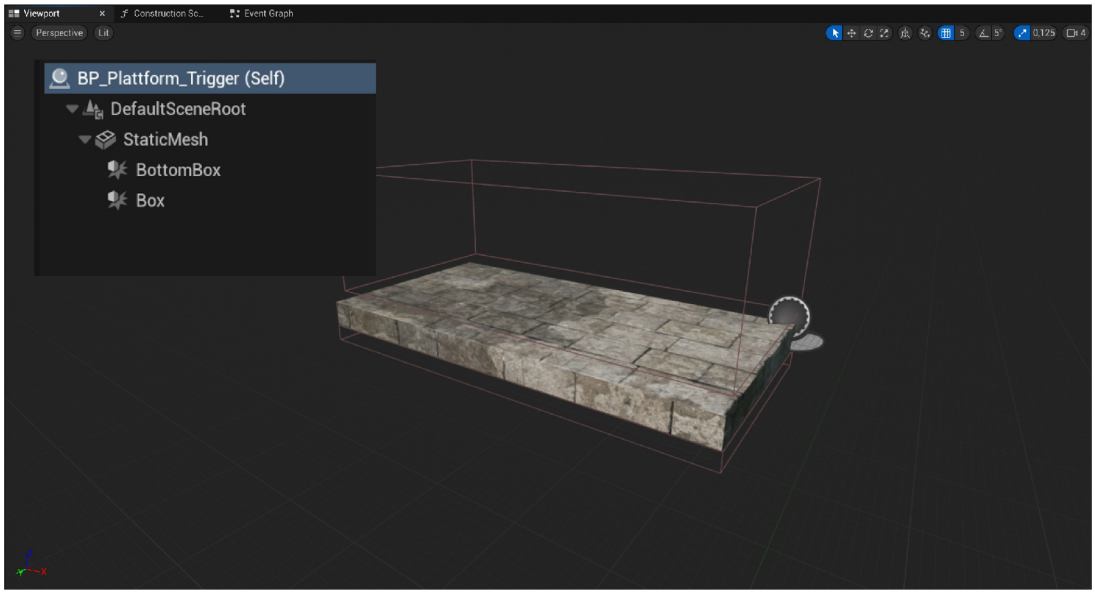
terContent a dvě kolizní oblasti, `Box` a `BottomBox`. Při kolizi `AActor` s oblastí `Box` se po prodlevě `DurationBeforeFall` vytvoří nová instance třídy `Platform_Destroyed` a zavolá se událost `FallToGround`, poté se instance `Platform_Trigger` zničí. V případě kolize `AActor` s oblastí `BottomBox` je chování totožné, akorát se nově vytvořené instanci třídy `Platform_Destroyed` nastaví boolean proměnná `FromBellow` na hodnotu **true**.

Blueprint třída `Platform_Destroyed` nemá žádnou `StaticMesh`, místo ní obsahuje `GeometryCollectionComponent`. 36 Tato komponenta využívá systém **Chaos Destruction**, [18] který má v sobě další komponentu, a to `Platform_GeometryCollection`, jež má nastavené chování tak, aby se platforma při kolizi s `AActor` rozbila. U této třídy je také pozměněna kolize, aby `Prince` nestoupal na zbytky instance této třídy, ale aby `CollisionBox Prince` tyto roztříštěné zbytky odstrkoval do stran.

Událost `FallToGround` se zavolá `BP_Platform_Destroyed` při svém vzniku. Nejprve zkontroluje v podmínce, jestli je její parametr `FromBelow` **true** nebo **false**. Pokud je **true**, po časové prodlevě se zavolá funkce `SphereTraceForObjects`, která na základě přepadaných parametrů zkontroluje, jestli se pod plošinou nachází `AActor`, který implementuje rozhraní `Damageable`. Pokud ano, způsobí mu zranění. Docílí toho tak, že vytvoří kolizní plochu ve tvare koule. Tato kolizní plocha platí jen pro frame, kdy je vytvořena. `SphereTraceForObjects` je funkce, kterou v tomto projektu velmi často používám, zejména v systému lezení.

V obou případech, když je `FromBelow` **true** i **false**, se po časové prodlevě rozbité kusy `GeometryCollection` pomocí `Timeline` postupně zmenší a ná-

sledně se celý tento `AActor` zničí. V samotné hře to tedy vypadá, že padající plošina je jeden `AActor`, ačkoliv ve hře se původní `Platform_Trigger` zničí a v tom samém místě se vytvoří `Platform_Destroyed`.



Obrázek 35: Obrázky z blueprint třídy Platform_Trigger

Obrázek 36: Obrázky - ukázka rozbitého GeometryCollectionComponentu

4.11 Umělá inteligence nepřátel

K tomu, aby se nepřátelé v levelu pohybovali a reagovali na hráče, je potřeba, aby měli nějakou svou vlastní umělou inteligenci. Umělou inteligenci v tomto projektu nepřátelům poskytují dvě `Controller` třídy: `Enemy` a `Enemy_Boss` pro nepřítele v posledním levelu. `AIC_Enemy` je třídou, která je defaultně nastavená v předkovi všech nepřátel, blueprint třídě `Enemy`.

Nepřítel v průběhy hry reaguje na aktuální situaci podle předem naprogramovaného stromu. Tento strom je v projektu pojmenován `Enemy (BT_Enemy)`. K správnému fungování stromu je zapotřebí proměnných. Je nutno si udržovat přehled o tom, v jakém stavu nepřítel je, a podle hodnot těchto proměnných pak strom vyhodnocuje která větev chování stromu je zrovna vykonávána. Tyto proměnné nemohou být přímo součástí stromu, jsou součástí další Unreal Engine třídy nazvané blackboard (v tomto projektu `BB_Enemy`). Tato třída je se stromem spojena. Jeden blackboard může používat vícero tříd `BehaviourTree`. Proměnné v `BB_Enemy` jsou nastavovány pomocí `AIC_Enemy`, který může být ovlivněn `BP_Enemy`, a jsou pak používány v `BT_Enemy`. Jinak řečeno, Blackboard proměnné, které jsou využívány ve stromech událostí, jsou nastavovány pomocí `AI_Controller`, který je přiřazen k root třídě všech typů nepřátel.

`BB_Enemy` má následující proměnné:

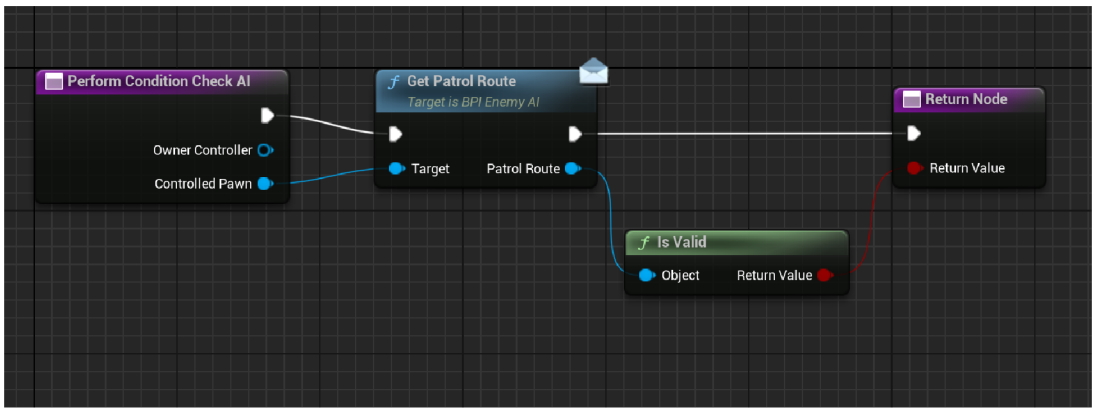
- `SelfActor` Reference na `AActor`, který má `AIC_Enemy`.
- `AttackTarget` Reference na `AActor`, na který `BP_Enemy` útočí.
- `State` typu `enum Enum` všech typů stavů, ve kterých se může `BP_Enemy` nacházet, jsou to:
  - `Passive`
    Nepřítel je ve stavu, kdy buď nedělá nic, nebo chodí po předem určené trase, instance blueprint třídy `PatrolRoute`.
  - `Attacking`
    Nepřítel útočí na hráče.
  - `Staggered`
    Nepřítel byl zraněn hráčem.
  - `Investigating`
    Nepřítel hráče zaslechl.
  - `DeadState`
    Nepřítel je mrtvý.
  - `Seeking`
    Nepřítel ztratil hráče z dohledu, pátrá po něm.
- `PointOfInterest`
  Lokace ve světě, kam chce nepřítel jít.

Tento projekt obsahuje 3 základní stromy chování: `Enemy_Persian`, `Enemy_Skeleton` a `Enemy_Knight`, a také množství podstromů. Na pořadí uzlů ve stromě chování záleží, prioritně se při vykonávání větve dívá zleva doprava, přičemž projde až k nejspodnější větvi. Uzly jsou v enginu očíslované, aby bylo jasné, jakou mají prioritu. `BehaviourTrees` jsou jakousi abstrakcí toho, kterou akci má nepřítel vykonávat. Teoreticky při programování umělé inteligence nepřátel není ani potřeba tuto třídu používat a vše lze psát do `Character` třídy, které se to týká. Výsledný kód by byl ale velmi složitý, těžko přehledný.

Dva základní uzly ve stromech jsou `Sequence` a `Selector`. `Sequence` oznamuje, že se budou vykonávat `Tasks`, a to v pořadí zleva doprava. `Selector` je vlastně if/switch podmínkou. Vybírá se mezi uzly podvětev, která bude vykonávána. V případě, že by se více podvětví stromu vyhodnotilo na true, vykoná se nejlevější podvětev.

4.11.1 Decorators, Tasks, Services

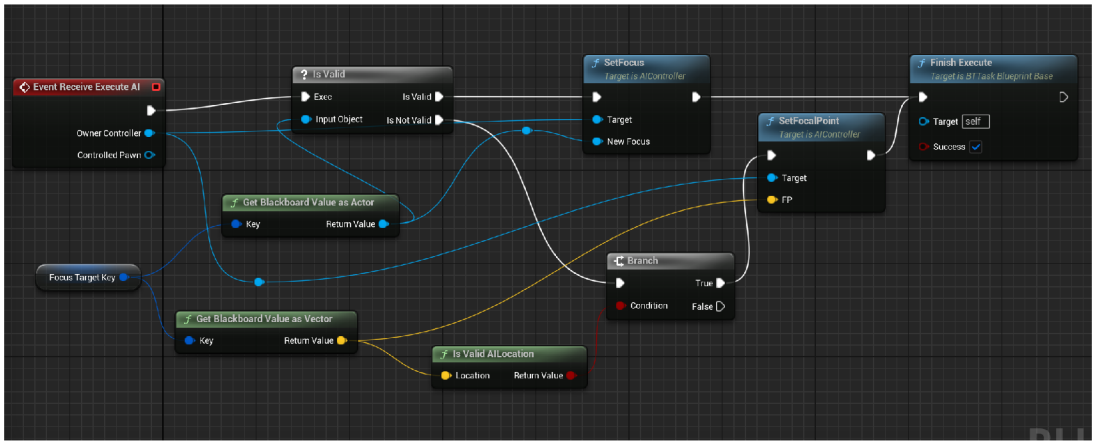
`Decorators` mají vícero funkcí, ale v tomto projektu jsou používány jako if podmínky kvůli proměnným v `BP_Enemy`. Proměnné v `BB_Enemy (Blackboard)` jsou společné pro všechny instance `BP_Enemy`. Pomocí `Decorator HasPatrolRoute` se zjistí, zda konkrétní instance třídy `BP_Enemy`, `Pawn` ovládaný `AIC_Enemy`, který implementuje rozhraní `Enemy_AI`, má validní referenci na blueprint třídu `PatrolRoute` ve scéně.37



Obrázek 37: Obrázek decoratoru HasPatrolRoute

`Tasks` 38 jsou všechny koncové uzly ve stromu. Buď jsou to přímo funkce, které jsou k dispozici z `BehaviourTrees` tříd, nebo jsou to funkce, které voláme stejným způsobem jako u `Decorators`, ať už se přitom odkazujeme na rozhraní `Pawn`, nebo na `Controller`. Aby se `Tasks` vykonávaly v sekvenci, musí předchozí `Task` vrátit hodnotu `FinishExecute` **true**. Může se stát, že strom v rámci optimalizace se bude snažit pomocí vláken vykonávat některé tasky dopředu. V praxi to pak může vypadat tak, že první `Task` se nedokončí celý a už začne druhý. V tomto projektu se mi to v některých případech také stalo. Nepřítel například útočil na hráče, aniž by předtím vytasil meč. Zvuk

i animace vytasení meče proběhly, avšak nepřítel v ruce nic nedržel, funkce `AttachComponentToComponent` se neprovedla. Vyřešil jsem tento problém pomocí `Event Dispatcher`. Tento problém jsem řešil v projektu několikrát.



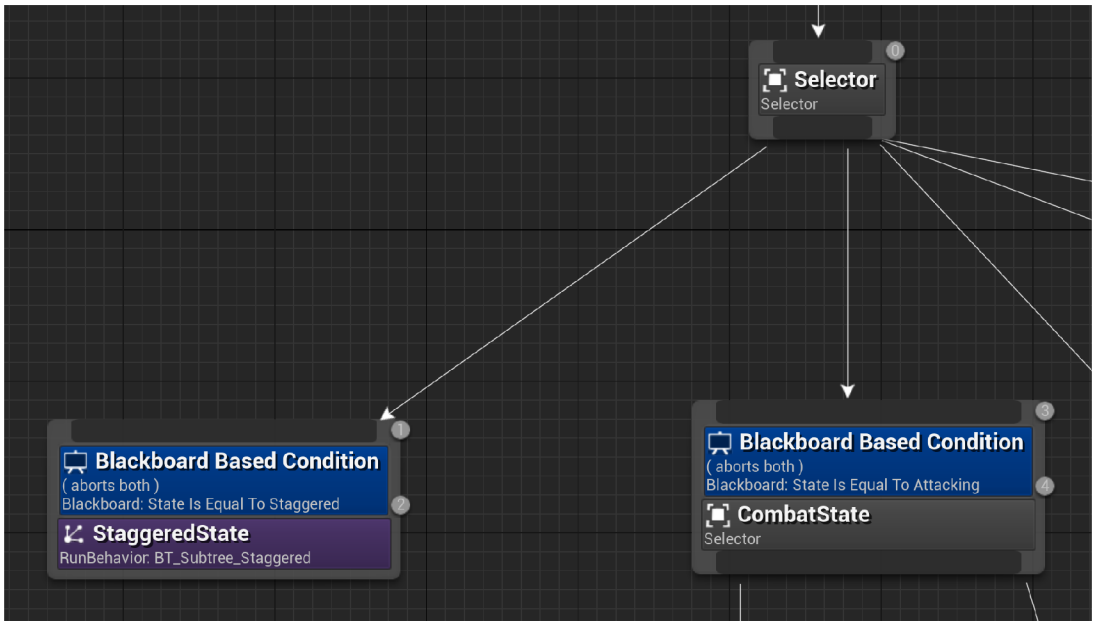
Obrázek 38: Obrázek tasku Focus

U `Services` lze nastavit funkcionalitu, která se periodicky vykonává, až když je v projektu jedna taková třída vytvořena, v projektu nebyla nikde použita.

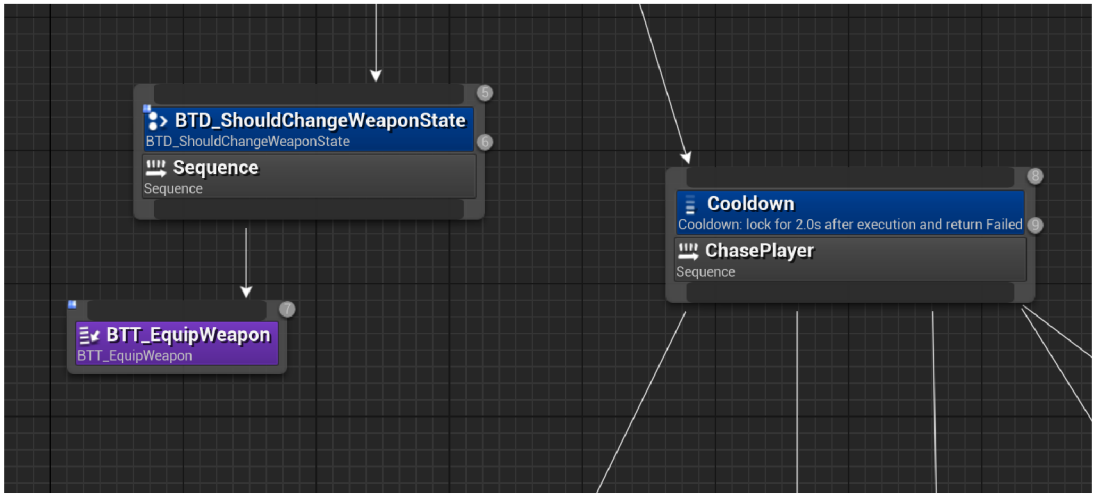
Popíši zde základní chování v `BT_Enemy_Persian`. Každý typ nepřítele má vlastní, trochu upravený strom chování, detailně ale popíši jen tento.

Z kořenového uzlu vede `execution flow` do `Selector`, to znamená že se bude vybírat uzel nejvíce vlevo, který vrátí hodnotu **true**. 39 Největší prioritu v tomto stromě má tedy `StaggeredState`, hráč byl zasažen. Jestliže zasažen byl, zavolá se podstrom `Staggered`. Tento podstrom obsahuje `Tasks ClearFocus` a `Wait`. Pokud je aktuální `State` jiný, vyhodnocování pokračuje dál. Na obrázku si lze všimnout vlastností: `(aborts both)`, `(aborts lower priority)` a `(aborts self)`. Znamená to, že v případě, že když se stav změní, zatímco se vykonává jiný podstrom na stejné úrovni v hierarchii, přeruší se vykonávání stromu podle tohoto parametru. `(aborts both)` přerušuje vykonávání podstromů, a to i těch s vyšší prioritou. `(aborts lower)` přeruší vykonávání těch stromů s menší prioritou a `(aborts self)` ukončí vykonávání jen sebe.

`State = Attacking` ? 40 Pokud ano, následuje další `Selector`, kde se v podmínce ptáme, zda má `Enemy_Persian` vytasit zbraň. V druhé větvi je pak dvousekundový `Cooldown`, to proto, aby nepřítel pořád ve smyčce neútočil a dal i `Prince` šanci zaútočit. Jiné typy nepřátel mají tento `Cooldown` variabilní, například dvě sekundy jsou pevná čekací doba a potom se přičte nebo odečte interval od 0 do 1 sekundy, takže jsou útoky více nepředvídatelnější. Dokud dvousekundový `Cooldown` nevyprší, nemůže se tato větev vykonat znovu. V této větvi je pak několik `Tasks`:



Obrázek 39: Obrázky z BT_Enemy_Persian



Obrázek 40: Obrázky z BT_Enemy_Persian

- `BTT_Focus`

  Nepřítel se zaměří na `AttackTarget`. Může to být kdokoliv, ne jen `Prince`, jako je to nyní.

- `BTT_SetMovementSpeed`

  Nastavení rychlosti s jakou se nepřítel bude pohybovat.

- `MoveToTarget`

  Nepřítel se bude pohybovat (rychlostí nastavenou v `BTT_SetMovementSpeed`) k `AttackTarget`. Je ještě důležité zmínit

AcceptableRadius, určuje, v jaké vzdálenosti od AttackTarget se má nepřítel zastavit.

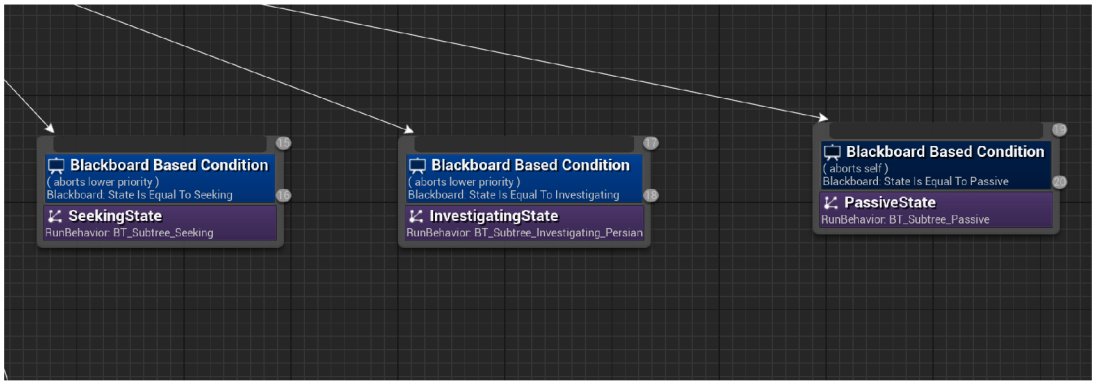
- BTT_DefaultAttack

  Nepřítel zaútočí, zavolá funkci Attack v BP_Enemy.

- Wait

  Nepřítel počká 1 sekundu.

Pokud není State ani Attacking, bude v jednom z 3 zbývajících stavů v této třídě, Seeking , Investigating , nebo Passive . Podle toho bude vykonávání pokračovat v jednom ze tří podstromů. 41



Obrázek 41: Obrázky z BT_Enemy_Persian

- BT_Subtree_Seeking

  Persian ztratil Prince z dohledu, nastaví se rychlost, focus a poté půjde k lokaci, kde Prince naposledy viděl. Jakmile se do ní dostane, počká 5 sekund +- 1 sekundu a potom se vrátí do stavu Passive .

- BT_Subtree_Investigating_Persian

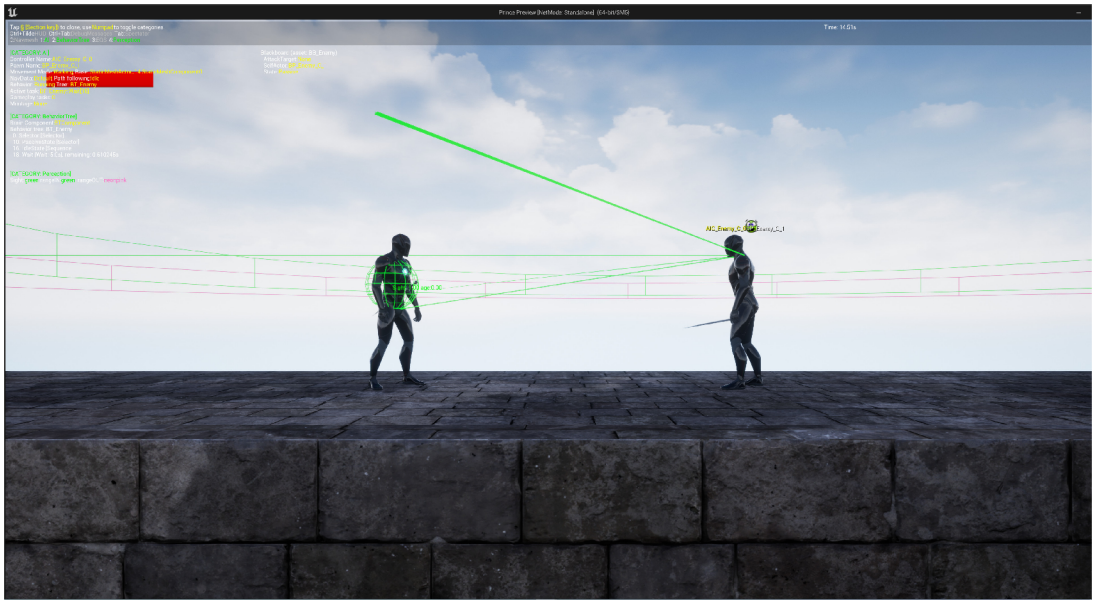
  Persian Prince zaslechl, nejprve počká, potom se přehraje zvuk, pak počká ještě jednou, nastaví se rychlost a poté půjde k lokaci, kde Prince slyšel. Tam počká 3 sekundy a potom se vrátí do stavu Passive .

- BT_Subtree_Passive

  Defaultní třída, nic se neděje. Pokud má Enemy_Persian vytasenou zbraň, schová ji. Pokud má PatrolRoute, nastaví se rychlost a chodí po ní. Pokud ani to ne, čeká na místě a nic nedělá.

4.11.2 Další funkce AI Controlleru

`AIController` obsahuje další funkční komponenty využívané v tomto projektu. Prvním z nich je `AIPerception`, 42 které umožňuje nepřátelům vnímat podněty (nejen) od hráče za pomocí smyslů. Tyto smysly pak určují to, v jaké vzdálenosti nepřítel hráče uvidí, uslyší jeho podněty apod. `AIC_Enemy_Boss` dědí z třídy používané u všech ostatních typů nepřátel, tedy `AIC_Enemy`. Liší se právě jen tím, v jaké vzdálenosti `BP_Enemy_Knight` hráče dokáže zahlédnout.



Obrázek 42: Obrázek – ukázka vizualizace AIPerception

I když hráč, `Prince`, nějakým způsobem ovlivní nějaký ze smyslů, musí se doprogramovat to, jak na to nepřítel zareaguje. V tomto projektu na to většinou nepřátelé zareagují změnou stavu, ať už na `Investigating` , kdy nepřítel hráče hledá podle polohy, kterou získá když se smysl aktivuje, nebo rovnou na `Attacking`, kdy začne na `Prince` útočit.

Specificky zmíním ještě funkci `HandleSensedSight`, kdy se v tomto případě vytvoří časovač, který periodicky spouští událost `CheckIfForgottenSeenActor`. Ta kontroluje, jestli se `Prince` neztratil nepříteli z dohledu. Pokud ano, vytvoří na místě, kde si `Controller` pamatuje, že hráč naposledy byl, instanci třídy `Shadow`. `BP_Shadow` 43 je třídou s `PoseableMesh`, nastavitelnou `Mesh` podle pózy `Character`, který má stejný `Skeleton`, jako `PoseableMesh`. V praxi to znamená, že se při vytvoření instance třídy `Shadow` vytvoří průhledná `StaticMesh` s pózou, která je zkopírovaná od `Prince` v ten moment, kdy jej nepřítel ztratil z dohledu. Po prodlevě, kdy `BP_Enemy` zapomene, že hráče viděl, a ze stavu `Seeking` přejde do stavu `Passive` , se `Shadow` zničí.

Obrázek 43: Obrázek – ukázka instance třídy BP_Shadow

4.11.3 PatrolRoute

Další komponentou, kterou `AIC_Enemy` obsahuje, je `ActionsComponent`. Ta je zodpovědná za různé akce, které může `Pawn` vykonávat a také PathFollowingComponent, který umožňuje `Pawn` pohybovat se po předem určené cestě. Na pohyb po předem určené cestě používám blueprint třídu `PatrolRoute`. 44 Ta má jako svou komponentu `Spline`, což je geometrická křivka, která má dva a více bodů. Aby si mohl vývojář vytvořit cestu podle svých představ, přidá instanci `PatrolRoute` do scény a pak nastaví tvar křivky. Projekt je vytvořen k pohybu pouze ve dvou rozměrech, tato třída však při testování v pořádku funguje i pro tři rozměry. Instance `PatrolRoute` se pak referencuje v instanci `BP_Enemy`. K jedné `PatrolRoute` lze přiřadit i víc nepřátel.

4.12 Soubojový systém a animace

Ve hře je několik typů nepřátel, kteří se liší svými útoky na hráčovu postavu `Prince`, dále také chováním, rychlostí animací, hodnotami v proměnných a dalšími vlastnostmi, které v této části popíši. Každý z nepřátel má vlastní třídu `Pawn`, k ní přiřazenou třídu animací, `SkeletalMesh` a s ní související `Skeleton`.

4.12.1 Animace

Třídy `Skeleton` obsahují popis kostí, jež určují, jak budou animace vypadat. Animace ve hrách jsou často udělané tak, že se nejprve zachytí v reálném světě technologií **Motion Capture**, 45 kdy se nasnímá pohyb pomocí lokačních bodů.

Obrázek 44: Obrázek – ukázka vizualizace PatrolRoute

[19] `Skeleton` v Unreal Enginu pak kopíruje lokaci snímaných bodů pomocí `Bone` (kostí). 46 Kosti umožňují to, že výsledný `Skeleton` může mít jiné rozměry, než nasnímaná animace.

Obrázek 45: Obrázek – ukázka technologie Motion Capture

Ke všem animacím použitým v tomto projektu jsem načasoval a přidal `Notify` události. Například přehrání zvuku při každém kroku nebo zvuky při útoku apod.

Obrázek 46: Obrázek – ukázka kosti lowerarm_l z třídy SK_Persian_Skeleton

Kosti jsou v `SkeletalMesh` třídě ve stromové hierarchii. Je to z praktických důvodů, když animace pohne kostí `lowerarm_ll`, chceme aby se zároveň s ní pohnuly kosti levé ruky včetně všech kostí prstů. Velmi často se stává, že animace jsou vytvořené s jiným počtem kostí, než má `SkeletalMesh`, na které se bude animace používat. Unreal Engine na to má řešení, retargeting. Retargeting lze použít ve dvou případech. V prvním má `Skeleton` jiné rozměry, animace se tedy přizpůsobí pro tento `Skeleton` se stejnými kostmi. Druhým případem je, když `Skeleton`, na kterém se má animace přehrát, má jiný počet kostí. V tomto případě se kosti sloučí. Kupříkladu mnoho animací neřeší konkrétní animace každého z prstů na ruce. Pomocí Retargetingu se tedy animace prstů na ruce sloučí s animací ruky. Díky stromové hierarchii, ve které kosti jsou, pak rozdíl nelze ani postřehnout.

4.12.2 HitDetection

S animacemi a soubojovým systémem souvisí také blueprint třída `HitDetection`. V animaci útoku se zavolá událost `NotifyTick` v této třídě a potom podle toho, jestli se jedná o `Prince`, nebo instanci `BP_Enemy`, se zavolá funkce `DetectHit` v těchto třídách. `NotifyTick` je tedy vázaná na každý frame animace a v animaci se tedy zavolá každý frame, kdy je přítomná. V obou případech, jak u `Prince`, tak `Enemy`, se u `StaticMesh` zbraně od socketu `BladeStart` do `BladeEnd` vytvoří vektor. Tyto sockety jsem musel manuálně přidat do každé `StaticMesh` zbraně, která je používaná, tedy kromě dýk, kde zásah zjišťuji jiným způsobem. Poté se zavolá funkce

SphereTraceForObjects, kde délka lokace je přímo čepel zbraně. Pokud kolizní oblast vytvořená touto funkcí zasáhne AActor, který implementuje blueprint rozhraní Damageable, způsobí se mu zranění. Nízká kvalita obrázku 47 je způsobena technologií **MotionBlur** a také tím, že jsem testoval v debugovacím módu v editoru.

Obrázek 47: Obrázek – ukázka funkce DetectHit ve scéně

4.12.3 DamageSystem

Jak hráč, tak nepřátelé mají jako součást svých tříd blueprint komponentu DamageSystem. Ta vyhodnocuje, zda byl hráč zraněn či nikoli, kolik má životů, kolik může mít maximálně životů, jestli se momentálně brání a podobně. Tuto komponentu sdílí jak Prince, tak všechny typy nepřátel. Má parametry Health a MaxHealth, které ukazují, kolik má Pawn životů aktuálně a maximálně. Poté obsahuje množství boolean proměnných, které popisují stav, v jakém se nachází útočící/bránící Pawn. Jsou to:

- IsInvinsible

  V projektu není použita, určuje, jestli je možné, aby byl Pawn zraněn.

- IsDead

  Informace o tom, jestli je Pawn mrtvý, ve stavu, kdy neútočí, ani se nepřehrávají žádné animace. Typicky se Pawn po prodlevě zničí, reference se odebere ze scény.

- `IsInterruptible`

  Informace o tom, jestli je útok, který je na `Pawn` veden, přerušitelný. Například pokud proti sobě dva `Pawn` vedou útoky souběžně, přehraje animace zranění typicky ten, který je zasažen první. Pokud by ale `IsInterruptible` bylo v tomto případě **false**, dokončí svůj útok a dva `Pawn` jsou zasaženi.

- `IsBlocking`

  Informace o tom, jestli je `Pawn` ve stavu, kdy útok blokuje.

- `IsParrying`

  Informace o tom, jestli `Pawn` útok odráží.

Funkce `Heal` se vstupem `Amount` typu `float` je funkce, která hráči zvýší počet životů o hodnotu `Amount` v případě, že `IsDead` = **false**. Funkce `DoDamage` je hlavní funkcí této komponenty, jako vstup má referenci na `AActor` `DamageCause`, který zranění způsobil. Tato funkce je tudíž volána, když `Pawn` zranění dostal, ne způsobil. Dalším vstupem je struktura `S_DamageInfo`, která obsahuje:

- `Amount` typu `float`

  Množství zranění, které může být `Pawn` způsobeno.

- `DamageType` typu `enum`

  Typ útoku, který je na hráče veden. Mohl by to být například útok zdálky, pasti, útok zblízka. Hodnota by se dala použít ve statistikách na konci hry, nebo například k tomu, že různé typy oblečení by poskytovaly `Prince` procentuální ochrany před nějakým typem útoku. V projektu však tato funkcionalita není použita.

- `DamageResponse` typu `enum`

  `Enum`, popisující reakci hráče na útok. Silnější útoky by například mohly způsobit jiný typ zranění, z kterého by se hráč vzpamatovával delší dobu. V projektu není použito.

- `CanBeBlocked` typu `boolean`

  Informace o tom, zda může být útok zablokován.

- `CanBeParried` typu `boolean`

  Informace o tom, zda může být útok odražen.

- `ShouldForceInterrupt` typu `boolean`

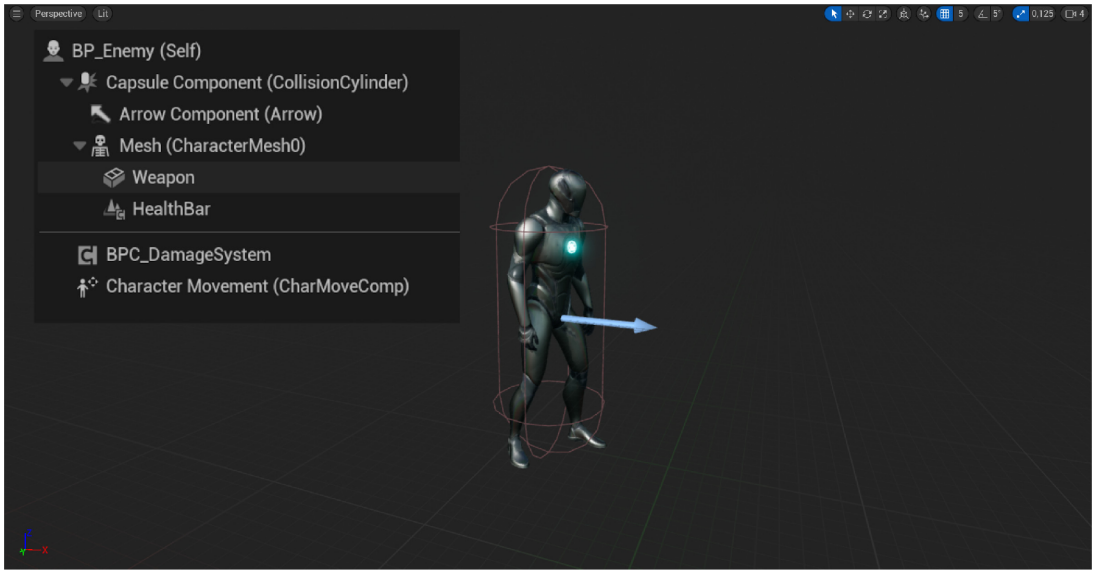
  Informace o tom, jestli vedený útok přeruší animaci `Pawn`.

V reakci na útok se zavolá pomocí `EventDispatcher` událost. Umožňuje zavolat funkci, která je nabindovaná v tomto případě ve třídě, [14] která má `DamageSystem` jako svou komponentu. Logika této funkce není tedy naprogramovaná v `DamageSystem`, ale každý `Pawn` ji má vlastní.

Pokud se `Pawn` brání, `IsBlocking` je **true**, zavolá se `OnBlocked` a funkce `DoDamage` vrátí `WasDamaged` **false** a `AttackParried` **false**. Pokud je `IsParrying` **true**, vrátí se `WasDamaged` **false** a `AttackParried` **true**. V ostatní případech je `Pawn` zraněn, proměnná `Health` se sníží o hodnotu `Amount`. Jestliže hodnota `Health` je nižší nebo rovná nule, zavolá se `OnDeath`, jestliže ne, tak `OnDamageResponse`. V obou případech potom bude `WasDamaged` **true** a `AttackParried` **false**. Logika `IsParrying` je rozdílná oproti například `IsBlocking`. Je to z praktických důvodů, protože je použit pouze ve třídě `Prince`. Logika by ale šla přepsat tak, aby byla stejná jako u `IsBlocking`.

4.13 Typy nepřátel

Všechny typy nepřátel dědí z blueprint třídy `Enemy`. 48 V `Enemy` se na začátku vytvoří `HealthBar`, nastaví se jako widget a přidá se `Controller`. V `Enemy` je naprogramována základní logika chování pro všechny třídy z ní dědící, takže v těchto třídách pak stačí změnit pouze hodnoty proměnných. Tyto proměnné jsou použity u implementací událostí navázaných na komponentu `DamageSystem`.



Obrázek 48: Obrázek z BP_Enemy

Proměnné v `BP_Enemy`:

- `IsWieldingWeapon` typu `boolean`
  Informace o tom, zda nepřítel drží zbraň.

- `DrawAnimation` a `SheatheAnimation` typu `AnimMontage`

  Animace vytasení a schování zbraně.

- `ExecutedAnimation` typu `AnimMontage`

  Animace popravy nepřítele.

- `HitAnimations` a `DeathAnimations` typu pole `AnimMontage`

  Animace zranění, respektive smrti. Můžou mít libovolnou délku, podle této délky se potom přehraje animace náhodně vybraná z pole.

- `DeathSound` typu `SoundBase`

  Zvuk přehraný při smrti `Enemy`, zničení instance třídy.

- `HitSounds`, `HitSoundsEnemy` a `AttackSounds` typu pole `SoundBase`

  Podobně jako u animací se náhodně jeden vybere z pole a přehraje se. `HitSounds` jsou míněné jako zvuky, které nepřítel vydá při zasažení a `HitSoundsEnemy` jsou zvuky, které jsou přehrány přímo fyzickým zasažením. Například instance `BP_Enemy_Skeleton`, kostlivec, nevydá žádný zvuk, ale zvuk zasažení kostí mečem je přehrán. `AttackSounds` jsou zvuky, které vydává nepřítel, když útočí.

- `BlockChance` typu `float`

  Je šance na zablokování útoku, hodnota je od 0 do 1, přičemž 1 znamená, že odrazí všechny útoky a 0 že žádný útok neodrazí.

- `WasDamaged` typu `boolean`

  `Enemy` byl zraněn.

Událost `Death` nejprve nastaví `State` v `AIC_Enemy` jako `Dead`, potom přehraje animaci, vypne kolizi `StaticMesh` a zastaví logiku vykonávání ve stromu, přehraje zvuk a po prodlevě instanci třídy zničí. Událost `DamageResponse` přehraje náhodné zvuky z `HitSounds` a `HitSoundsEnemy` a poté nastaví `State` jako `Attacking`. Když proběhne změna stavu ve všech případech, změní se i vykonávání podvětve stromu.

`DrawWeapon` událost přehraje `DrawAnimation` a poté pomocí `Notify` v určitý snímek animace připojí `StaticMesh` `Weapon` do socketu (kosti) `sword_draw` ve `SkeletalMesh`. Tento socket jsem v každé `SkeletalMesh` vytvořil, je navázán na pravou, respektive levou ruku. Animace útočení tedy hýbou pouze rukou, ale protože je ve stromové hierarchii socket `sword_draw` pod hand_l, tak se pohybuje i zbraň, poté přehraje zvuk. `SheatheWeapon` událost udělá to stejné, s rozdílem jiných proměnných a socketu `sword_sheath`. Pokud `Prince` útok odrazí, zavolá se událost `OpponentParried`, pustí se animace a nastaví se stav jako `Staggered`.

Událost `Action` je událost z blueprint interface `Actionable`. Popisuje chování nepřítele při popravení hráčem. Je nazvaná `Action`, protože hráč tuto událost spustí pomocí klávesy E, na kterou jsou navázány i jiné akce jako zvedání meče či lahviček, tyto třídy taktéž implementují rozhraní `Actionable`. Nejprve se nastaví stav nepřítele jako `Dead`, vypne se jeho kolize, zastaví logika vykonávání, přehraje se animace, zvuk a po prodlevě se instance `AActor` zničí. Dále tato třída implementuje další množství funkcí implementovaných z rozhraní, které nebudu vypisovat.

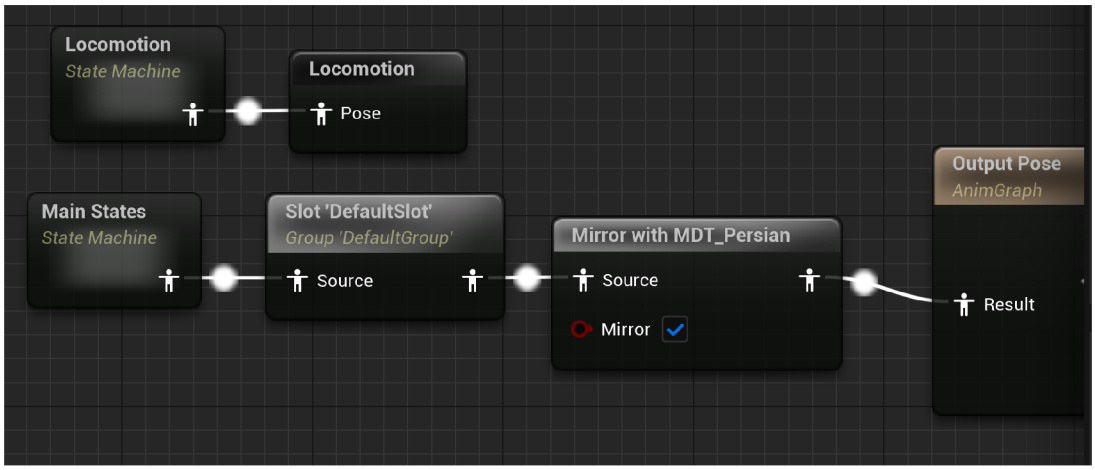
Kromě `DamageSystem` má třída a všechny z ní dědící ještě `CharacterMovementComponent` zajišťující pohyb, a velmi důležitý `CapsuleComponent`, který představuje skutečné rozměry `StaticMesh Pawn` ve scéně. Všechny kolize instance této třídy, včetně soubojového systému, jsou tedy počítány s tímto `CapsuleComponent`. Ve hře jsou jeho hranice skryty. Pro třídy dědící z `BP_Enemy` jsou tyto rozměry pozměněny. Mít kolizi jiného tvaru, než je `StaticMesh`, je běžnou praxí ve hrách, [20] neboť pokud je `Mesh` méně komplexního tvaru, snadněji se kolize počítají a velmi to vylepšuje optimalizaci.

4.13.1 Persian

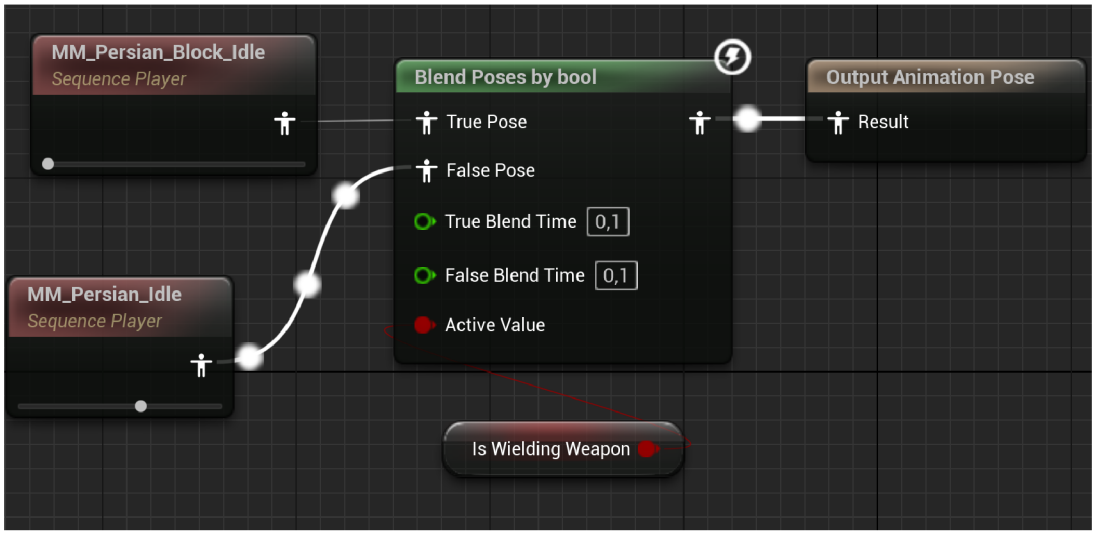
`Enemy_Persian` dědí z `Enemy`, má navíc `StaticMesh Shield` se štítem. Při vytvoření instance této třídy se nastaví maximální počet životů a proměnné zvuků a animací. V tomto projektu jsou použity dvě verze blokování útoků nepřátel. První, používaná `Enemy_Persian` funguje díky implementaci funkce `DoDamage` z rozhraní `Damageable` ve třídě. Předtím, než se zavolá z blueprint komponenty `DamageSystem`, tak se zavolá funkce `TryToBlockAttack` (jen v případě, že útok může být blokován). Zde se vygeneruje náhodná hodnota typu `float` v intervalu 0 až 1 a jestliže je proměnná `BlockChance` menší než tato hodnota, `Enemy_Persian` byl zraněn. Jestliže je větší, zavolá se událost `StartBlock`. V této události se přehraje animace, zvuk, útok se zablokuje.

Animace pomalé, či rychlé chůze, padání, skoku se nepřehrávají tím, že by se manuálně spustily přímo z `BP_Enemy_Persian`, ale tím, že každá `Character` třída (dědící z `Pawn`) má vlastní animation blueprint (`ABP`). 49 Je to zase jakási forma abstrakce, veškerá funkcionalita je totiž ovládána pomocí proměnných. Teoreticky by šla tedy napsat přímo do `BP_Enemy_Persian` do události `EventTick`, která se volá každý snímek, a tam zase pomocí proměnných jednotlivé animace přehrávat. Nicméně je dobrou praxí `ABP` třídy používat a leccos to usnadňuje. Pokud se však jedná o animace, které se přehrají jen jednou, a to v určitý okamžik, je dle mého názoru lepší je přehrávat přímo v `BP` třídě místo `ABP`, a to i v tomto projektu praktikuji. 50

Všechny animace v této třídě také zrcadlím, protože `Prince` většinou ve hře chodí zleva doprava a nepřítel vypadá lépe, když drží meč v levé ruce. Část kódu v `ABP_Persian` jsem použil již naprogramovanou od Epic Games. Animace, textury nepřítele, i 3D modely jsou upravené z Epic Games Marketplace. Logika však zůstává a lze nahradit všechny tyto assety vlastními se zachováním funkcionality (funkcionalita v animacích ovšem zachována není).



Obrázek 49: Obrázek z ABP_Persian

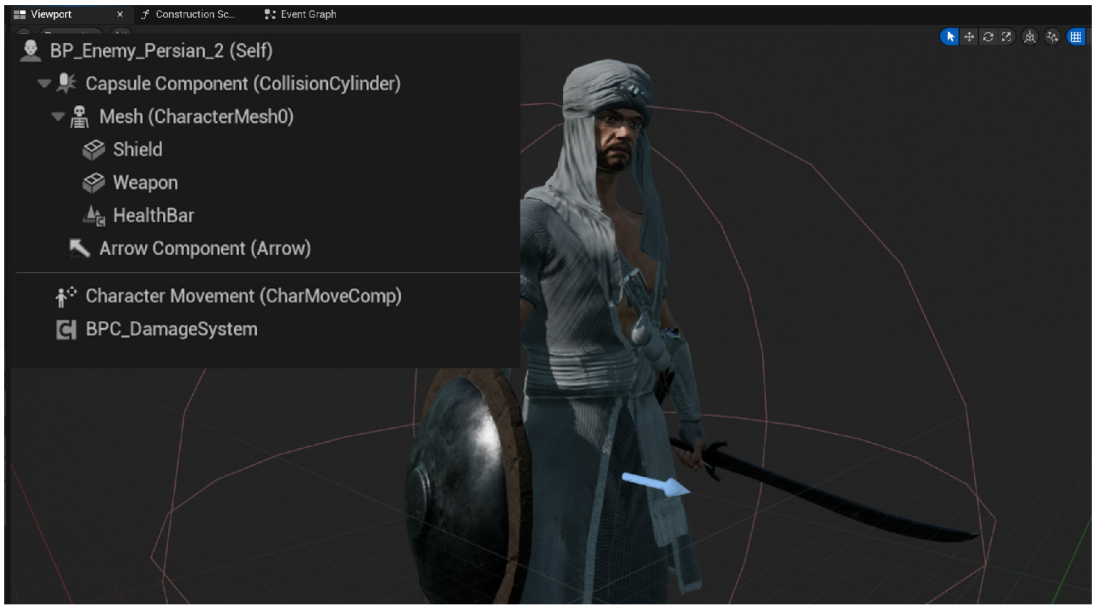


Obrázek 50: Obrázek z ABP_Persian 2

Z `BP_Enemy_Persian` dědí ještě další typy nepřátel `Persian`. 51 Liší se drobnostmi, úpravou proměnných, barvou upravených materiálů, počtem životů, šancí na zablokování útoku a rychlostí útoku.

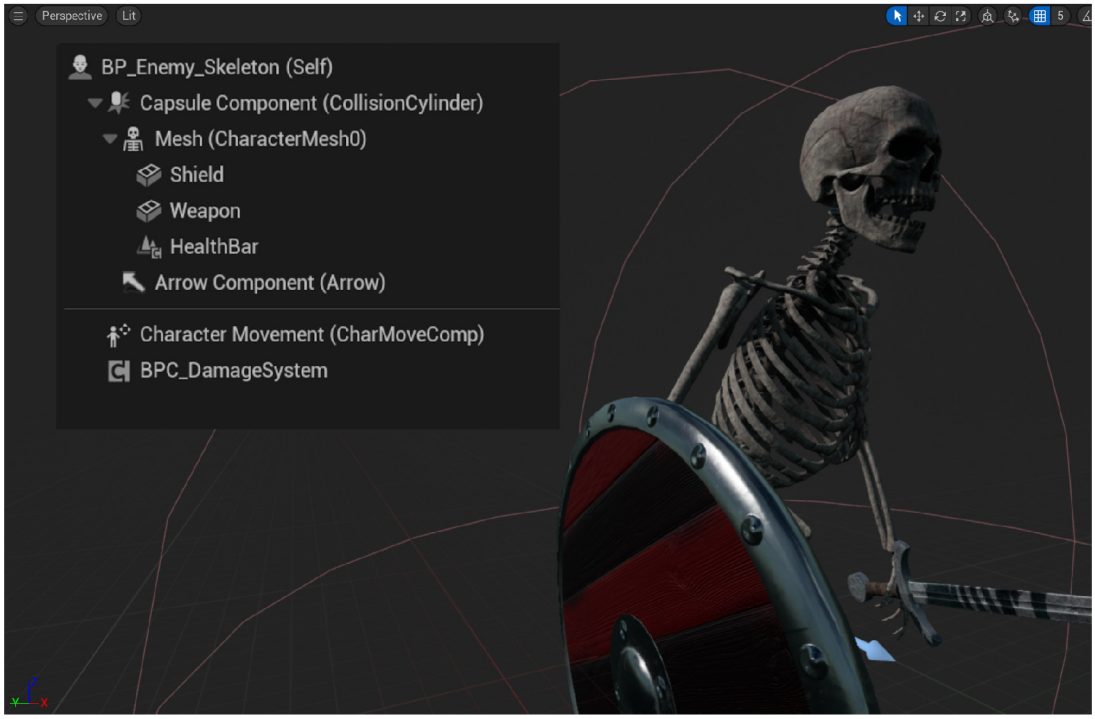
4.14 Skeleton

`Enemy_Skeleton` 52 je velmi podobný `Enemy_Persian` , v určité fázi vývoje z něj i přímo dědil. Stejně jako u `Enemy_Persian` pochází jeho 3D modely z Epic Game Marketplace, animace jsou přetargetované z jiného balíčku, nebo upravené. Některé jsou defaultně z enginu. Ač jsem tyto animace nevytvářel, dalo mi velkou práci pochopit, jak retargeting funguje a jak jej správně používat v tomto projektu. Strávil jsem na tom mnoho hodin. `Skeleton` byl totiž můj vzorový `Enemy` typ, na kterém jsem všechno testoval a debugoval. Stejně jakou



Obrázek 51: Obrázek z blueprint třídy Enemy_Persian_2

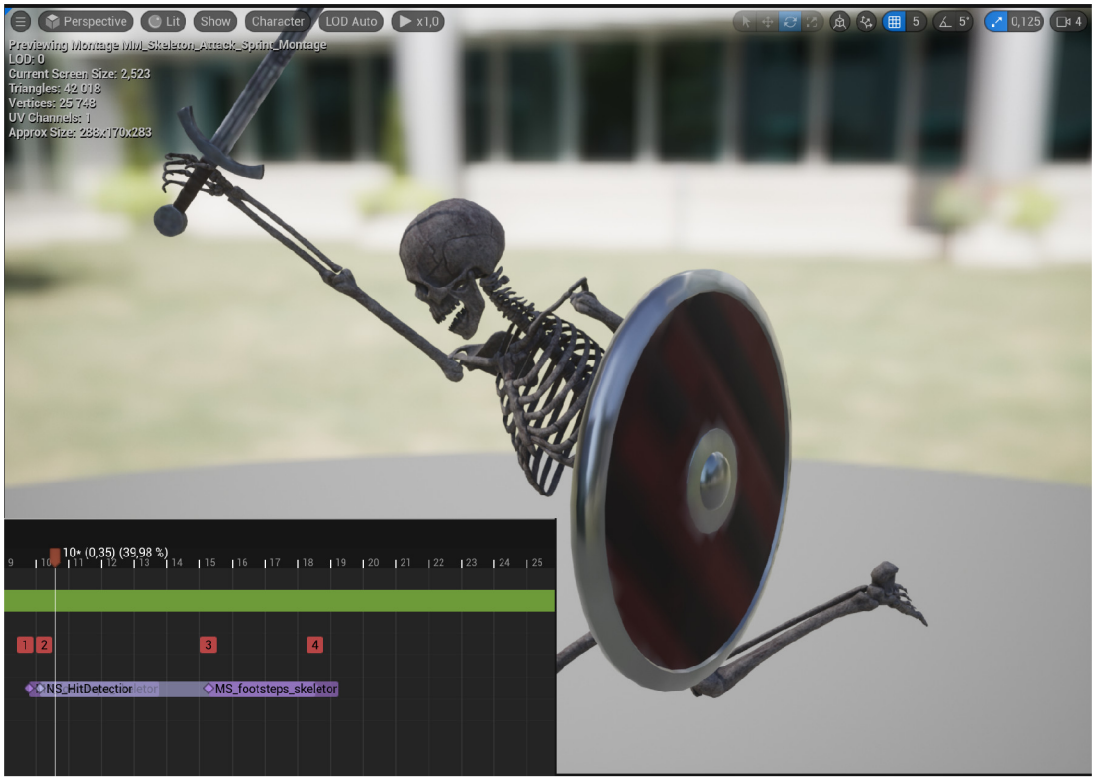
`Enemy_Persian` má vlastní `ABP` třídu, `SkeletalMesh` i strom a podstrom. Některé podstromy sdílí s `BP_Enemy_Persian`.



Obrázek 52: Obrázek z blueprint třídy Enemy_Skeleton

Hlavním rozdílem je pak to, jakým způsobem může zablokovat útok nepřítele. Ve stromu se v podmínce zjišťuje, zda `Prince` útočí. Pokud ano, `Skeleton`

se brání. Bránit se může však jen každé 2 sekundy, takže větší množství útoků kostlivce zasáhne, jeden za delší časovou dobu však ne.



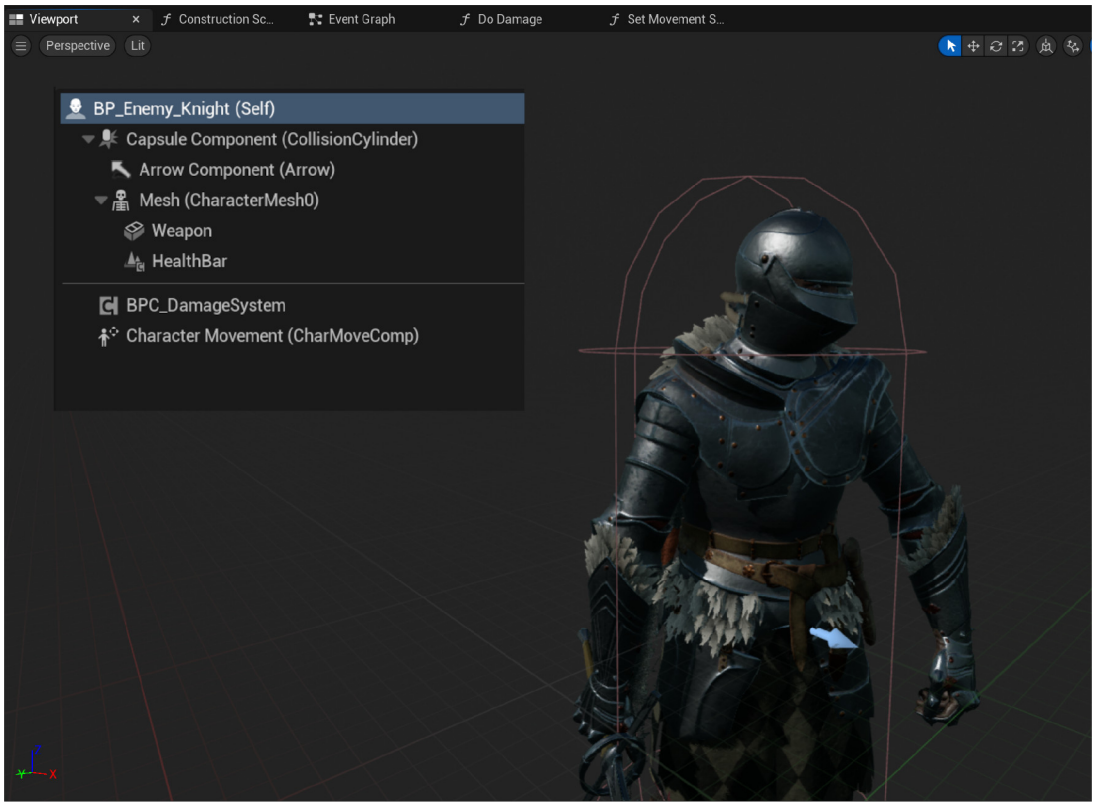
Obrázek 53: Obrázek – ukázka speciálního útoku kostlivce

Má také vlastní speciální typ útoku. 53 Když je `Prince` ve větší vzdálenosti, zaútočí na něj skokem.

$$|x| > 150$$

4.14.1 Knight

I assety používané v `BP_Enemy_Knight` 54 nejsou součástí tohoto projektu. 3D modely a materiály jsou z Epic Games Marketplace, animace jsou přetargetované z jiného balíčku. Také dědí z `BP_Enemy`, liší se pak od ostatních typů nepřátel propracovanějším systémem útoků a také tím, že má vlastní `HealthBar` widget odlišný od ostatních. Pokud instanci této třídy klesne hranice životů pod 50 %, rychlost a typ jeho útoků se zrychlí a rychleji se vzpamatovává ze zranění. V jeho `ABP` třídě se v tomto případě také změní animace stání a rychlost přehrávání všech animací se zvýší.

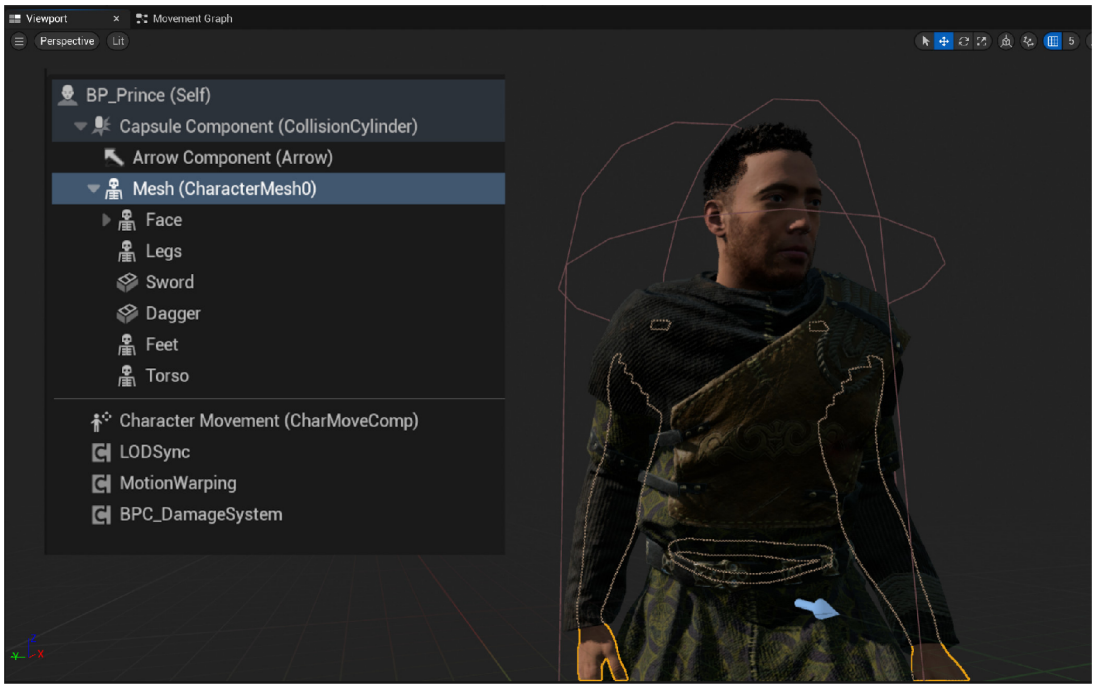


Obrázek 54: Obrázek z blueprint třídy Knight

4.15 Prince

`Prince` 55 je hratelný `Pawn`, který je ovládán hráčem ve hře. Veškerou naprogramovanou funkcionalitu jsem rozčlenil v této třídě (`BP_Prince`) do několika grafů. `Prince` obsahuje root `Mesh` komponentu. Nastavit tuto komponentu byla jedna z nejtěžších výzev v tomto projektu. Model a materiál hlavy `Prince`, ruce a část noh jsou totiž z **Metahumans** od Epic Games. Oblečení je z Epic Games Marketplace. Zkombinovat tyto části bylo značně obtížné. Nakonec to funguje tím způsobem, že **Metahumans** je vyexportovaný a nastavený tak, aby výškově a tělesnou konstitucí odpovídal co nejvíce defaultní `Mannequin`, který je používaný v enginu. U všech `SkeletalMesh` (`SkeletalMesh`, `StaticMesh` `Character`, které mají nastavené kosti) jsem pak změnil `Skeleton` z používaných na ten používaný defaultně v enginu, `SK_Mannequin`. Výsledek funguje velmi dobře i pro různé typy oblečení. Původně bylo totiž oblečení včetně textur v tomto projektu možné měnit, nebyl jsem ale spokojený s výsledkem a proto jsem tuto funkcionalitu odstranil.

`CharacterMovementComponent` umožňuje `Prince` pohybovat se v levelu. Stejně jako u nepřátel, Princův pohyb je omezen jen na osu x a z.
`LODSync` je součástí **Metahumans**, automaticky přepíná level detailů tváře a těla. Nastavil jsem override na fixní hodnotu, protože se kamera nachází ve fixní vzdálenosti od `Prince`. `MotionWarping` souvisí se systémem lezení, bude



Obrázek 55: Obrázek z blueprint třídy Prince

popsána později. Poslední komponentou je `DamageSystem`.

4.15.1 UIGraph a MovementGraph

`UIGraph` slouží primárně k inicializaci počátečních hodnot a nastavení. Nejprve se vytvoří nový `AActor`, blueprint komponenta typu `CameraSystem`, aktivuje se a přiřadí do proměnné ve třídě. Poté se vytvoří `TimeWarpingStaminaWidget` a `HealthWidget` a nastaví se jejich hodnoty. Jak bylo zmíněno, je to dynamické, takže se rychlost vyčerpání `TimeStamina` a velikost widgetů nastaví podle aktuálních maximálních hodnot. Tyto hodnoty jsou `Prince` přiřazeny v `PrinceGameMode`. Aktualizace velikosti widgetu, což většinou znamená výměnu textury, se provádí při každém zvýšení maximální hodnoty jedné z proměnných `TimeStamina` a `MaxHealth`. Připomenu, že proměnná `MaxHealth` je součástí blueprint komponenty `DamageSystem`, kterou `Prince` obsahuje. Dále se přidá (**Niagara System**) jako vizuální ukazatel toho, že `Prince` zpomaluje čas, aktivuje se jen když `Prince` čas zpomaluje. Nakonec se inicializuje `EnhancedInputSubsystem` pro přijetí vstupu pohybu od hráče.

Blueprint třída `CameraSystem` obsahuje kameru a SpringArm, je nastavená tak, aby jako cíl sledovala `Pawn Prince`. Kamera jej však nesleduje přesně, ale se zpožděním, takže když se `Prince` pohne, kamera jeho pohyb následuje až po chvíli. Také má offset, takže `Pawn` není úplně ve středu zorného pole. Kamera se aktualizuje každý snímek ve hře. V tomto grafu se ještě nachází bindování klávesy Escape na vytvoření `PauseMenuWidget`.

MovementGraph obsahuje pouze jednu událost a to EnhancedInputAction Move, kde je nastaven pohyb Prince doleva a doprava. Pohyb je omezen proměnnými, aby se Prince nehýbal, když např. zvedá meč. Pohyb je ještě upraven, když je Prince v takzvaném CombatMode, protože má vytasen meč a nemůže se otáčet.

4.15.2 DebugGraph a ActionsGraph

DebugGraph sloužil při vývoji testování nových funkcionalit. Obsahuje bindování, nejčastěji na numerické klávesy, léčení, zvýšení životů, změnu stavu nepřátel a dalších. V packagované hře jsou tyto funkcionality odbindované. ActionsGraph obsahuje navázání klávesy H na hvízdání, vytvoření hluku. Prince nemůže hvízdat, když zpomaluje čas.

4.15.3 CombatGraph

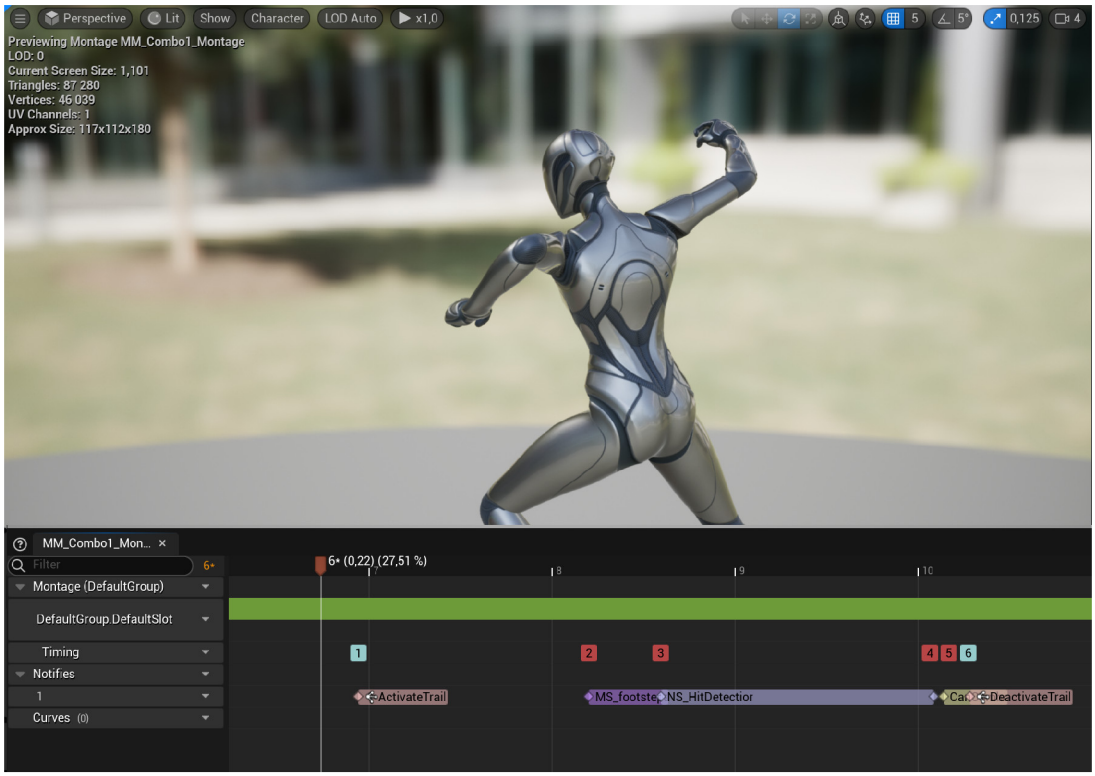
Soubojový systém využívá již popsanou komponentu HitDetection. Mimo to Prince na rozdíl od nepřátel může útoky navazovat a propojovat. V každé animaci je krátký časový úsek, během něhož může hráč znovu stisknout levé tlačítko myši, na které je útok navázán, a Prince pokračuje v útoku. 56 V případě, že je Prince uprostřed sekvence útoků zasažen nepřítelem, resetuje se kombo na začátek, animace útoku se přeruší a přehraje se animace zranění společně se zvuky zasažení.

S útoky souvisí i SwordWeaponTrail a DaggerWeaponTrail, jsou to bílé efekty ve vzduchu, které zůstanou na zlomek sekundy v místě, kde Prince zbraní útočil. Tyto efekty jsou také **Niagara System** a vytvořil jsem je sám. U každé animace se na krátký časový úsek aktivují a následně deaktivují. Všechny animace útoku a obrany jsou z Epic Games Marketplace.

Pomocí klávesy V hráč přepíná tzv. CombatStance. Je to k tomu, aby měl možnost se i během boje otáčet. Dosahuje se toho změnou orientace v CharacterMovementComponent komponentu a úpravou animací v ABP třídě.

I Prince implementuje rozhraní Damageable, a proto v DamageResponse se taktéž přehraje animace zranění a je přehrán zvuk. Je tu také implementována událost Death, kdy se zastaví pohyb, hráči se zakáže veškerý input, přehraje se zvuk, vytvoří se W_DeathText a po prodlevě se level restartuje.

Pomocí pravého tlačítka myši může hráč buď odrážet útok, nebo zpomalovat čas. Aby spolu různé funkcionality v tomto projektu nesoupeřily, je všechno zase ošetřeno množstvím proměnných, většinou typu boolean. Při odrážení útoku se přehraje animace, nastaví se IsParrying v DamageSystem na **true** a v animaci se taktéž aktivuje **Niagara System** DaggerWeaponTrail. Prince nemůže útoky blokovat, jen odrážet (IsBlocking vs IsParrying). Zpomalování času je ošetřeno tak, aby ho hráč nemohl opakovaně aktivovat krátkými, rychlými stisky pravého tlačítky myši. Nastavil jsem fixní časovou prodlevu, která změní hodnotu proměnné CanTimeWarp a vytvoří se časovač, který ji po uplynutí doby



Obrázek 56: Obrázek – ukázka z animace útoku využívané Prince

změní zpátky na **true**. Při zpomalování času je aktivován další **Niagara System** a je přehráván zvuk. Poté, co hráč přestane čas zpomalovat, se po krátké prodlevě začne doplňovat `TimeStamina`. Znovu je to realizováno pomocí `Timer` (časovače) a Timeline. Pro `Prince` je čas také zpomalen, ale ve výrazně menší míře, než pro okolní svět.

Klávesou F může `Prince`, v případě že neprovádí nějakou jinou akci, vytasit/skrýt svoje zbraně. Funguje to na stejném principu jako v `BP_Enemy_Persian`, v určitý snímek animace se `StaticMesh Sword` a `Dagger` připojí k socketům vytvořených na `Skeleton`, funkce `AttachComponentToComponent`. Dále se přehraje zvuk, změní se `Prince` rychlost chůze a změní se stavy `boolean` proměnných. 57

Klávesa E slouží k vykonání různých akcí v levelu. Pokud má hráč vytasený meč a nachází se poblíž nepřítele, který je není v `Attacking State`, může jej pomocí této klávesy zavraždit. K zjištění, jestli je poblíž nepřítele, se používá již zmíněná funkce `SphereTraceForObjects`. I v tomto případě se aktivuje **Niagara System** `SwordWeaponTrail` a přehrají zvuky. V ostatních případech hráč něco zvedá, pokud je hodnota proměnné `enum CurrentOverlappedActorType Sword`, zavolá se makro `PickUpSword`. Přehraje se několik animací, které jsem pospojoval, a nastaví se proměnné `StaticMesh Sword` a `Dagger`. Důležité je, že se nastaví proměnná `CanEnterAttackState` na **true**, takže poté, co `Prince` dokončí level, může

Obrázek 57: Obrázek – ukázka z animace skrytí zbraně

v dalších levelech již bojovat, protože se tato informace na konci levelu uloží.

V případě, že `CurrentOverlappedActorType` je `Potion`, zavolá se `DrinkPotion` makro. Podobně jako při vytasení meče se `StaticMesh` lahvičky přidá do socketu `Skeleton` v určitý frame animace. Přehrají se dva zvuky. Na `CurrentOverlappedActor` referenci se zavolá funkce `Action`. Tato funkce určuje konkrétní efekt, který `Potion` má. Je z rozhraní `Actionable`.

Animace zvedání věcí a pití lahvičky jsem vytvořil v programu **Cascadeur**. Animace pozvednutí meče vzhůru je z Epic Games Marketplace a není součástí projektu.

4.15.4 ClimbingGraph

Vytvořit systém lezení bylo na celém projektu asi to nejtěžší a strávil jsem na ním nejvíce času. 58 Podobně jako s bojovým systémem jsem i tady měl na začátku pomoc ve formě tutoriálu. Ale jako v případě bojového systému, tak i tady jsem narazil na to, že moje využití je velmi specifické. Proto tutoriál pokryl jen zlomek času vývoje celého systému, postačil jako základ na pochopení principů a zbytek jsem si doprogramoval sám pro své potřeby. Odkazy na použité tutoriály v projektu jsou v `README.txt`.

Klávesou mezerník hráč skáče, ale nejen to, provádí také úhyb, chytí se plošiny nebo uskočí dozadu. Co se právě stane, závisí na několika proměnných. Pokud

Obrázek 58: Obrázek – ukázka složitost blueprint funkcí systému lezení

je `Prince` v `CombatMode`, hodnota `IsInCombatMode` je **true**, má vytasený meč, `Prince` uskočí dozadu. Stejně jako v případě zpomalování času nemůže hráč klávesu stisknout znovu, dokud se `ResetTimer` neobnoví.

V případě, že proměnné `IsOnLedge` a `LedgeHasSurface` jsou **true**, znamená to, že se `Prince` odrazí od zdi a skočí směrem dozadu, zavolá se událost `BackEject`. V ní se zavolá funkce `LaunchCharacter` z Unreal Enginu. Ve funkci se také mění kolize, `MovementMode` a použije se funkce `MoveComponentTo`. Když `Prince` letí vzduchem, může se v ten moment zachytit římsy. Ač výsledek vypadá jednoduše, nastavení správného chování `Prince` v této události je složité. Narazil jsem tu nejspíš na bug enginu, který už je dlouho nevyřešen, alespoň tedy podle odpovědi vývojáře na fóru. [21] Problémy s enginem jsem řešil v projektu na více místech, nejčastěji to byly problémy se synchronizací událostí, s paralelními vlákny. V případě že `IsOnLedge` = **false** && `IsInCombatMode` = **false**, znamená to, že `Prince` skáče, buď aby se chytil římsy, nebo jen tak do vzduchu. k zjištění, jestli je poblíž nějaká římsa, se používá funkce `LedgeTrace`.

4.15.4.1 LedgeTrace

`LedgeTrace` hojně využívá již zmíněnou `SphereTraceByChannel`. Má tři vstupy. `TraceForward` typu `float` určuje, do jaké vzdálenosti v ose x se římsa hledá. `TraceVertical` určuje, v jaké výšce se hledá. `InitialZOffset` určuje výšku, od které se římsa začne hledat. Není totiž dobré, aby se `Prince` zachytil římsy, která je vzhledem k němu například 20 cm nad zemí. V této funkci je for cyklus, který provede `LedgeTrace` 10×, pokaždé s upravenou `TraceVertical` výškou. Dobře to ilustruje obrázek. 59

V případě, že `SphereTraceByChannel` zasáhne římsu, zjistíme pomocí dalšího `SphereTraceByChannel` vrchol římsy, a pokud i ten vrátí **true**, for

Obrázek 59: Obrázek – ukázka vizualizace funkce LedgeTrace

cyklus se přeruší a vrátí se `LedgeDetected` jako **true** a dva vektory, `LedgeLocation` a `LedgeNormal`.

4.15.4.2 CheckTraceRoof

V případě, že byla římsa nalezena, je potřeba ještě zkontrolovat, zda není nad `Prince` plocha. 60 `LedgeTrace` totiž zjistí, že se tam nachází nějaká hrana, ale ta hrana klidně může být součást plochy, takže by `Prince` vylezl skrz objekt, což se mi i při testování stalo. `CheckTraceRoof` tedy udělá tentokrát `LineTraceByChannel` přímo nad `Prince`. Jestliže zasáhne nějakou plochu, `Prince` v lezení nepokračuje a místo toho skočí na místě. V opačném případě se zavolá funkce `HangingOnLedge` s parametry `LedgeLocation` a `LedgeNormal` získanými z `LedgeTrace`.

4.15.4.3 HangingOnLedge

Než vysvětlím, jak `HangingOnLedge` funguje, je nutné pochopit jeden důležitý aspekt animací v Unreal Engine [22]. Když je jakákoliv animace v projektu přehrávána, neznamená to automaticky, že se pohybuje celá instance `Pawn`. Pokud autor v animaci nenastavil i pohyb `RootMotion`, v animaci se hýbe jen `StaticMesh`. Kolize, zprostředkovaná `CapsuleComponent` (což je pro připomenutí koule, která reaguje na útoky či pasti, fyzická reprezentace `Prince` v levelu), se nehýbe. Po skončení animace se tedy `StaticMesh` `Prince` vrátí na původní lokaci. Například, pokud by animace chůze neměla `RootMotion`, `Pawn` by v animaci šel dopředu a když by animace skončila (animace chůze bývají ve smyčce), vracel by se stále dokola na původní místo. Nikam by se tedy neposouval.

Proč je to důležité? Chtěl jsem, aby `Prince` mohl vylézt na římsu v libovolné výšce. Animace lezení (z Mixamo, odkaz v `README.txt`) používají `RootMotion`. Tyto animace mají ale v základu nastavenou nějakou fixní výšku, do které `Prince` vyleze. Římsy v projektu by mohly tedy být jen v jedné výšce. Výslednou implementace systému komplikoval i fakt, že pro rozdílné vi-

Obrázek 60: Obrázek – ukázka vizualizace funkce CheckTraceRoof

sící polohy je rozdílná i výška v jaké `Prince` visí při animaci. `Prince` vyskočil v animaci na římsu ve fixní výšce, aby animace skoku seděla, ale navazující animace visení na římse `Prince` stejně posunula, takže to vypadalo, že visí ve vzduchu. Na scénu tedy přichází již zmíněný `MotionWarping` komponent, použitý z pluginu Unreal Enginu. Pomocí programu přímo z webu Mixama, **Mixamo_Converter**, který podporuje i `Mannequin` Unreal Enginu, jsem přetargetoval využívané animace lození a nastavil `RootMotion` do středu `StaticMesh`. Poté jsem jim zakázal pohyb v jakémkoli směru. Animace tak zůstávají na místě a `Prince` se nikam nepohybuje, pokud bych tedy nepoužíval `MotionWarping`. Při lezení se vytvoří `MotionWarpingtarget` s konečnou lokací a potom se při přehrávání animace `Prince` pohybuje směrem k `MotionWarpingtarget`. Na konci animace k němu dorazí.
`HangingOnLedge` funguje na tomto principu. 61

Dále jsem řešil problém, kdy visící animace nenavazovaly na animace lezení, byly v jiné výšce. Mixamo má k dispozici animace pro skok na římsu a poté i pro visení na římse. Tyto animace na sebe ale výškově nenavazují. Proto jsem skoro ve všech případech udělal to, že animace, kdy `Prince` skočí a zachytil se římsy, je předčasně ukončena. Zbytek animace se přehrává zpomaleně ve smyčce, a to tak, že nejprve ve směru pohybu a poté opačně. `Prince` vypadá, že se houpe na římse, ale ve skutečnosti je to pořád neukončená animace skoku na římsu, která se přehrává ve smyčce. Animace se ukončí v moment, kdy `Prince` seskočí z římsy, nebo na ni vyleze.

Celá logika je ještě o něco složitější. Je při ni použita ještě funkce `CheckIfLedgeHasSurface`, ve které se volá `LineTraceByChannel`, zjistí se, jestli na římse, kde `Prince` skáče, je místo na nohy a pokud ano, vrátí **true**.

Obrázek 61: Obrázek – ukázka visícího Prince

Podle toho se potom přehrají různé animace.

`HangingOnLedge` funkce je ale použita i v případě, že `Prince` padá a zachytí se římsy. V tento moment je vstupní parametr `IsFalling` **false** a animace a hodnoty jsou ještě jiné. Zase se rozlišují dva případy animací podle toho, zda římsa má prostor na nohy, či ne. Pro funkce `MakeMotionWarpingTarget` a `MoveComponentTo` je použito množství hodnot, které jsou v tomto blueprintu natvrdo stanovené, a jelikož jsou čtyři možné scénáře, kdy `Prince` visí (respektive šest, ale o tom později), tyto hodnoty se liší a pro každý scénář jsou jiné. Nastavil jsem je po hodinách testování.

Pokud má `Prince` místo na nohy, změní se také `CameraOffSet` v `CameraSystem` komponentě, aby šlo vidět, že `Prince` visí.

4.15.4.4 Spuštění se z římsy

Pokud `Prince` stojí na římse a chce se z ní spustit, například aby se vyvaroval zranění, které by mu jinak způsobil pád z výšky, může hráč tak učinit, pokud stiskne klávesu S.

Nejprve se zavolá funkce `CheckDroppingFromLedge`, kde se využijí funkce `CapsuleTraceByChannel` a `LedgeTrace`, aby se zjistilo, zda `Prince` má možnost se spustit. Hodnoty jsou opět fixní. Pokud se spustit může, zavolá se ještě funkce `CheckIfLedgeHasSurface`, která zjistí, jestli má římsa, ze které se `Prince` spouští, místo na nohy. Poté se zavolá funkce `DropFromLedge`, která je velmi podobná `HangingOnLedge`, ale je samostatně kvůli lepší čitelnosti blueprintu. Znovu je tedy použit `MotionWarpingtarget` a zase jsem vytvořil visící animace, přehrávající se ve smyčce.

4.15.4.5 Puštění se římsy a vylézání na římsu

`Prince` se může z římsy dostat dvěma či třemi způsoby. Vylézt nahoru, pustit se, nebo se odrazit a skočit (funkce `BackEject`) v případě, že římsa má místo na nohy. Pokud je tedy `Prince` na římse (`IsOnLedge` = **true**), vrátí se všechny proměnné na své defaultní hodnoty, kdy je `Prince` na zemi, a přehraje se animace. `CameraOffSet` se vrátí na původní hodnotu. Pokud `Prince` padá z velké výšky, v události poskytnuté enginem `EventOnLanded` se zjistí jeho rychlost (proměnná `Velocity`) a podle ní bude buď zraněn za jeden život, nebo rovnou zemře.

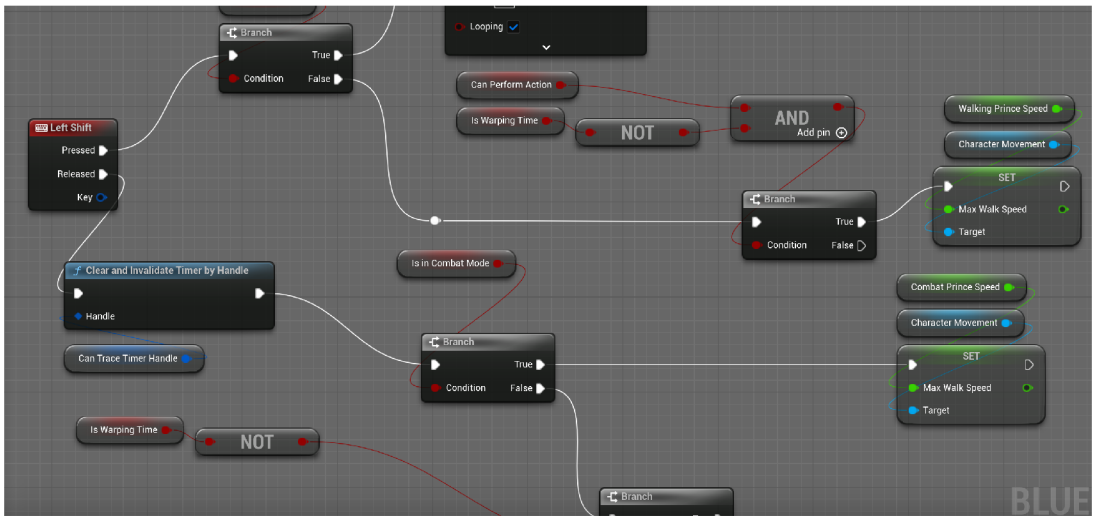
Klávesou W `Prince` vyleze z římsy zpátky nahoru. Tato funkce je co se komplexity týče skoro stejně složitá jako `HangingOnLedge`, protože hodnoty v ní používané (na vytvoření `MotionWarpingtarget` a pro funkci `MoveComponentTo`) jsou rozdílné pro každý z šesti případů, kdy `Prince` visí na římse. Metodou pokusu a omylu jsem našel hodnoty, kdy mi animace připadaly nejpřirozenější.

4.15.4.6 Zachycení se římsy

Pomocí klávesy Shift t může `Prince` chodit pomalu. Činí tak ale pouze v případě, kdy nezpomaluje, nebojuje, nebo není ve vzduchu. Pokud je ve vzduchu a hráč stiskne klávesu Shift, tak po dobu, kdy jej drží, se vytvoří `Timer`, který každých 0,1 sekund volá funkci `TraceForLedge`. 62 `TraceForLedge` volá již popsanou funkci `LedgeTrace` s jinými hodnotami, než když `Prince` stojí na zemi. Když `Prince` padá, chová se chycení římsy trochu jinak. (`InitialZOffset` musí být negativní, `Prince` římsu nehledá nad, ale pod sebou). Pokud `LedgeTrace` vrátí **true**, tak se musí tentokrát zjistit, jestli je blízko podlaha, (ne strop) aby se nezachytil kousek nad zemí. Toho se dosáhne pomocí funkce `LineTraceByChannel`. Pokud je výsledek negativní, zavolá se funkce `HangingOnLedge`.

4.15.5 Prince AnimationBlueprint

I `Prince` má vlastní animation blueprint. Animace chůze se liší v případech, kdy je jedna z proměnných `IsInCombatMode`, `IsInCombatStance`, nebo `IsWarpingTime` **true**. Připomínám, že rychlost chůze se nastavuje v `BP_Prince`, animace pak tuto rychlost reflektují a plynule přechody mezi nimi slučují. Animace kdy `Prince` zpomaluje čas, jsem vytvořil sám přímo v Unreal Enginu. Jako základ jsem použil defaultní animaci chůze. Ostatní animace v této třídě jsou buď ze StarterContentu, Mixama, nebo z Epic Games Marketplace.



Obrázek 62: Obrázek – ukázka z logiky při stisknutí klávesy Shift

Závěr

Tvorba 2.5D hry v Unreal Enginu mě naučila mnoha znalostem a dovednostem. Lépe jsem si uvědomil jak důležité je při vývoji software mít vývojářský tým, který si jednotlivé části práce rozdělí. Znovu jsem si připomenul, že mít konkrétní návrh a představu je při vývoji software velmi důležité. Tento projekt jsem dělal po většinu doby s velkým nadšením a jsem na něj hrdý. Nakonec jsem si tak uvědomil i to, kolik toho člověk zvládne.

Conclusions

Creating a 2.5D game in Unreal Engine taught me much in terms of knowledge and skills. I became more aware of how important it is to have a development team that divides the individual parts of the work. Once again, I reminded myself that having a specific design and vision during software development is very important. I worked on this project for most of the time with great enthusiasm, and I'm proud of it. So, in the end, I also realized how much one can accomplish.

A Ovládání, testování

Tato diplomová práce se skládá ze 2 částí, projektu, který byl použit na vývoj a výsledné hry. Zprovoznění obou aplikací je popsáno v `README.txt`. Hra byla testována mými kolegy z firmy ConfigAir. A také mými přáteli, z nichž jeden pracuje jako herní testér a jeden jako herní designér. Všem se podařilo hru úspěšně spustit bez mých dalších instrukcí. Mám od nich mnoho pozitivních ohlasů i nápadů na zlepšení. Díky jejich testování jsem odhalil a opravil několik bugů.

Ovládání:

- W Vyšplhání nahoru na římsu.
- S Puštění se římsy. Spuštění se z římsy.
- A, D Pohyb do stran.
- Levé tlačítko myši Útok mečem. (meč musí být vytasený)
- Pravé tlačítko myši Odražení útoku nepřítele. (meč musí být vytasený) Zpomalení času. (meč nesmí být vytasený)
- E Zvednout předmět. (meč nesmí být vytasený) Popravení nepřítele. (meč musí být vytasený)
- F Vytasit, schovat zbraně.
- V Změna bitevního postoje. (meč musí být vytasený)
- H Zahvízdání.
- Shift Pomalá chůze. Zachycení se římsy ve vzduchu.
- Mezerník Skok na římsu. Skok přes překážku. Skok dozadu. (na římse s prostorem pro nohy) Úhyb. (meč musí být vytasený)
- Escape Pauza.

B Obsah elektronických dat

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím stylu KI PřF UP v Olomouci včetně všech obrázků a dalších příloh.

`README.txt`
: Instrukce, jak spustit projekt ve kterém byla hra vyvíjena a také jak spustit výslednou hru. Obsahuje také další odkazy na materiály použité v diplomové práci.

`Prince/assets`
: Adresář obsahující mnou vytvořené assety v programech Blender, Audacity, GIMP a Cascadeur včetně zdrojových souborů.

`Prince/development`
: Vizuální ukázky některých bugů. Obrázky a videa z vývoje.

`Prince/packaging/Prince`
: Adresář se zabalenou hrou.

`Prince/source/Prince`
: Projekt ve kterém se hra vyvíjela.

`Prince/videos`
: Videa popisující hru a ukazující průchod hrou. Lze je použít jako návod, jak hru vyhrát.

Literatura

[1] EPIC GAMES, INC. *Epic Games Marketplace* [online]. 2024 [cit. 2024-5-6]. Dostupný z: ⟨https://www.unrealengine.com/marketplace/en-US/store⟩.

[2] Mechner, Jordan. *The Making of Prince of Persia: Journals 1985-1993.* Illustrated Edition. 2020. 336 s. ISBN 978-0578627311.

[3] Shoam, Amir. *35 Years of Prince of Persia* [online]. [cit. 2024-3-26]. Dostupný z: ⟨https://www.techspot.com/article/2788-prince-of-persia/⟩.

[4] Cowant Brent, Kapralos Bill. A Survey of Frameworks and Game Engines for Serious Game Development. *2014 IEEE 14th International Conference on Advanced Learning Technologies.* 2014. Dostupný také z: ⟨http://dx.doi.org/10.17083/ijsg.v4i4.194⟩.

[5] Christopoulou, Eleftheria; Xinogalos, Stelios. Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices. *International Journal of Serious Games.* 2017, vol. 4, s. 21–36. Dostupný také z: ⟨http://dx.doi.org/10.17083/ijsg.v4i4.194⟩.

[6] Barczak Andrej, Woźniak Hubert. Comparative Study on Game Engines. *Studia Informatica : systems and information technology.* 2019, roč. 1-2(23), s. 5–24. Dostupný také z: ⟨http://dx.doi.org/10.34739/si.2019.23.01⟩.

[7] Ash, Parish. *Unity unites the indie game industry against its new pricing model* [online]. 2024 [cit. 2024-3-27]. Dostupný z: ⟨https://www.theverge.com/23873852/unity-new-pricing-model-news-updates⟩.

[8] Farris, Jeff. *Forging new paths for filmmakers on "The Mandalorian* [online]. 2020 [cit. 2024-3-28]. Dostupný z: ⟨https://www.unrealengine.com/fr/blog/forging-new-paths-for-filmmakers-on-the-mandalorian⟩.

[9] EPIC GAMES, INC. *Unreal Engine Documentation* [online]. 2024 [cit. 2024-4-8]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine⟩.

[10] EPIC GAMES, INC. *Unreal Engine Prefixes* [online]. 2024 [cit. 2024-4-8]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/recommended-asset-naming-conventions-in-unreal-engine-projects?application_version=5.0⟩.

[11] EPIC GAMES, INC. *Unreal Engine Localization Tools* [online]. 2024 [cit. 2024-4-26]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/localization-tools-in-unreal-engine?application_version=5.0⟩.

[12] EPIC GAMES, INC. *Unreal Engine Timelines* [online]. 2024 [cit. 2024-4-26]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/timelines-in-unreal-engine?application_version=5.0⟩.

[13] EPIC GAMES, INC. *Unreal Engine Events* [online]. 2024 [cit. 2024-4-26]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/events-in-unreal-engine?application_version=5.0⟩.

[14] EPIC GAMES, INC. *Unreal Engine Event Dispatchers* [online]. 2024 [cit. 2024-4-27]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/event-dispatchers-in-unreal-engine?application_version=5.0⟩.

[15] Smith Gilian, Whitehead Jim. A framework for analysis of 2D platformer levels. 2008. Dostupný také z: ⟨http://dx.doi.org/10.1145/1401843.1401858⟩.

[16] EPIC GAMES, INC. *Unreal Engine Materials* [online]. 2024 [cit. 2024-4-26]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-materials?application_version=5.0⟩.

[17] EPIC GAMES, INC. *Unreal Engine Niagara* [online]. 2024 [cit. 2024-4-26]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-niagara-effects-for-unreal-engine?application_version=5.0⟩.

[18] EPIC GAMES, INC. *Unreal Engine Physics* [online]. 2024 [cit. 2024-4-26]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/physics-in-unreal-engine?application_version=5.0⟩.

[19] Sharma, Shubham; Verma, Shubhankar; Kumar, Mohit; Sharma, Lavanya. Use of Motion Capture in 3D Animation: Motion Capture Systems, Challenges, and Recent Trends. In. *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. 2019, s. 289–294. Dostupný také z: ⟨http://dx.doi.org/10.1109/COMITCon.2019.8862448⟩.

[20] Lazaridis, Lazaros; Papatsimouli, Maria; Kollias, Konstantinos-Filippos; Sarigiannidis, Panagiotis; Fragulis, George F.
Hitboxes: A Survey About Collision Detection in Video Games.
In Fang, Xiaowen (ed.).
HCI in Games: Experience Design and Game Mechanics.
Cham: Springer International Publishing, 2021, s. 314–326.
ISBN 978-3-030-77277-2.

[21] EPIC GAMES, INC. *Unreal Engine Forum Bug Discussion* [online].
2024 [cit. 2024-4-27].
Dostupný z: ⟨https://forums.unrealengine.com/t/is-the-launch-character-node-interrupted-or-overridden-by-move-to-commands/285249/18⟩.

[22] EPIC GAMES, INC. *Unreal Engine Root Motion* [online]. 2024 [cit. 2024-4-27].
Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/root-motion-in-unreal-engine?application_version=5.0⟩.