České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Čáp Martin

Studijní program: Otevřená informatika
Obor: Softwarové systémy

Název tématu: Porovnání herních platforem Unity a Unreal engine

Pokyny pro vypracování:

Nastudujte možnosti vývojářských platforem pro tvorbu počítačových her Unity a Unreal engine. Porovnejte tyto platformy pomocí realizace stejné hry přiměřené složitosti v obou platformách. Soustřeďte se na dominantní prvky vybraného herního žánru a popište způsob jejich realizace na obou platformách. Srovnání proveďte z pohledu implementační náročnosti v obou systémech (časová náročnost, nutné teoretické základy pro implementaci, atd.) ale i z pohledu finálního výsledku jako je rychlost vykreslování, a obecně hráčův požitek. Rozsah hry a srovnávací kritéria konzultujte s vedoucím práce.

Seznam odborné literatury:

[1] Creating Games with Unity and Maya. Adam Watkins. Focal Press, 2011.
[2] https://unity3d.com
[3] https://www.unrealengine.com

Vedoucí: Ing. David Sedláček, Ph.D.

Platnost zadání do konce letního semestru 2017/2018

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 17.1.2017

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Bakalářská práce

Porovnání herních platforem Unity a Unreal Engine

Martin Čáp

Vedoucí práce: Ing. David Sedláček, Ph.D.

Studijní program: Otevřená informatika

Obor: Softwarové systémy

18. května 2017

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 18. května 2017 ....

Poděkování

Chtěl bych poděkovat vedoucímu práce panu Ing. Davidu Sedláčkovi, Ph.D. za konzultace a cenné rady během její tvorby.

Abstrakt

Předmětem této bakalářské práce je porovnání dvou herních enginů: Unity 3D a Unreal Engine 4. V každém enginu byla vytvořena hra. Porovnání je založeno na nabytých zkušenostech při vývoji obou her. Při tvorbě her byl kladen důraz na jejich škálovatelnost a všeobecnou kvalitu ve zkoumaných kategoriích herního vývoje jako je grafiký vzhled, ovládání hry a funkčnost. Práce slouží k lepšímu pochopení výhod a nevýhod obou enginů pro účel výuky herního vývoje a základních herních algoritmů. Výsledné projekty jsou dvě hratelná dema, která si mohou vyzkoušet budoucí studenti předmětů o počítačových hrách.

Abstract

The purpose of this bachelor thesis is to compare two game engines: Unity 3D and Unreal Engine 4. In each engine, a game project was created. The comparsion is based on the gained experience during the development of these games. The scalability and general quality of researched game components such as graphical fidelity, controls and functionality of the game was emphasized. The thesis will demonstrate pros and cons of both engines when it comes to learning game development and basic game algorithms. The final game projects are playable demos that can be presented to future students who are interested in game development in these particular engines and also in general.

Obsah

1 Úvod **1**

2 Základy enginů, skriptování a herní framework **3**

2.1 Verze enginů . . . 3

2.2 Pojmy a názvosloví . . . 3

2.3 C#, C++, Blueprint editor . . . 4

2.3.1 Unreal Engine API . . . 5

2.3.2 Nevýhody hybridní implementace . . . 6

2.3.3 Nevýhody čisté C++ implementace . . . 7

2.3.4 Nevýhody Blueprint skriptů . . . 7

2.4 Herní framework . . . 9

2.5 Práce s časem a lineární interpolace . . . 10

2.6 Ukládání hry . . . 13

3 Herní svět **14**

3.1 Herní objekty, prefabrikáty a Blueprint třídy . . . 14

3.2 Změna úrovní a persistence dat . . . 16

3.2.1 Unity . . . 17

3.2.2 Unreal Engine . . . 18

3.3 Terrain a Landscape . . . 19

3.4 Foliage . . . 21

3.5 Geometry Brushes v UE . . . 23

4 Uživatel **25**

4.1 Uživatelské rozhraní . . . 25

4.2 Zpracování vstupů . . . 28

4.2.1 Připojení ovladače . . . 30

5 Vzhled a animace **31**
5.1 Materiály a shadery 31
5.1.1 Editace materiálů za běhu hry a statické přepínače 33
5.2 Částicové systémy (Shuriken a Cascade) 35
5.2.1 Typy zdrojů 36
5.3 Post processing 39
5.4 Export a import FBX souborů 41
5.5 Animace (Mecanim a Persona) 42
5.5.1 Animační masky, vrstvy a míchání 42

6 Umělá inteligence **46**
6.1 Rozhodovací stromy v Unity 46
6.2 Behaviorální stromy v Unreal Engine 47

7 Herní mechaniky žánru **49**
7.1 Úkolový systém 49
7.1.1 Unity 49
7.1.2 Unreal Engine 51
7.2 Systém dialogů 52
7.3 Systém předmětů 53
7.3.1 Serializace dat 54

8 Kolize **56**
8.1 Kolizní tělesa 56
8.2 Filtrování kolizí 57
8.3 Detekce zásahů 60

9 Ostatní **61**
9.1 Organizace scény 61
9.2 Závislosti assetů 61
9.3 Kvantitativní testování 62
9.4 Pořizování snímků 63

10 Závěr **64**

11 Použité assety **66**

Literatura **67**

A Snímky z her **73**
A.1 Vytvořené úrovně 73
A.2 Snímky ze hry v Unity 76
A.3 Snímky ze hry v UE 78

B Vytvořené modely **81**

C Seznam použitých zkratek **85**

D Obsah přiloženého DVD **87**

Seznam obrázků

2.1 Rozpojené uzly pole struktur Inventory a funkce Get v UE. 8
2.2 Použití Timeline komponentu v UE. 12
2.3 Coroutine implementace v Unity. 13

3.1 Materiál půdy v UE. 20
3.2 Tráva a kameny v Unity. 22
3.3 Foliage Edit Mode a vzhled flóry v UE. 23

4.1 Uživatelské rozhraní inventáře v Unity. 27
4.2 Uživatelské rozhraní inventáře v UE. 27
4.3 InputSettings v obou enginech 29

5.1 Material Editor v UE a jeden z vytvořených materiálů. 33
5.2 Ukázka materiálu stejného charakteru v obou enginech. 34
5.3 Ukázka použití Post Process Volume v UE. 40
5.4 Animator Controller postavy v Unity. 43
5.5 Animační Blueprint a míchání animací v UE. 45

6.1 Coroutine akce útoku nepřátelské postavy v Unity. 48
6.2 Behaviorální strom v UE. 48

7.1 Dialog s NPC v UE. 53
7.2 Formát uložení dat v UE a Unity. 55

8.1 Kolizní tabulka dvou fyzikálních těles v UE. 59
8.2 Vizualizace oblastí zásahu při útoku nepřátelské postavy v UE. 60

9.1 Strom referencí v UE. 62

A.1 Snímek všech úrovní v UE z dáli. 73
A.2 Detail úrovně interiéru v UE. 74

A.3 Bližší pohled na první úroveň ve hře v UE. 74
A.4 První úroveň vytvořená v Unity. 75
A.5 Druhá úroveň vytvořená v Unity. 75
A.6 Boj s úhlavním nepřítelem v Unity. 76
A.7 Snímek ze hry v Unity. 76
A.8 Interakce s předměty v Unity. 77
A.9 Dialogové okno pro rozhovor s NPC v Unity. 77
A.10 Interakce s předměty v UE. 78
A.11 Truhla s předměty v UE. 78
A.12 Boj s úhlavním nepřítelem v UE. 79
A.13 Snímek ze hry v UE. 79
A.14 Noční scéna v UE. 80
A.15 Částicový systém portálu v UE. 80

B.1 Render vozíku v Blenderu. 81
B.2 Render vesnického domu v Blenderu. 82
B.3 Render domu s věží s testovacím materiálem v Blenderu. 82
B.4 Render domu s věží v Blenderu. 83
B.5 Kompozice většiny vlastních modelů s přiřazenými materiály v UE. . . . 83
B.6 Vlastní modely zbraní a dalších předmětů v UE. 84

1. Úvod

Tato bakalářská práce pojednává o rozdílech mezi dvěma herními enginy: Unity 3D a Unreal Engine 4. Výstupem práce je porovnání jednotlivých částí a funkcionalit obou enginů v rámci vytvořeného projektu. Jedním z hlavních cílů práce je usoudit, zda se předmět Počítačové hry a animace (A7B39PHA) a v následujících semestrech jen Počítačové hry (B4B39HRY) má nadále vyučovat v Unity, nebo naopak, zda přestoupit na odnedávna volně přístupný Unreal Engine (UE).

Porovnání je založeno na konkrétním projektu, který je vypracován v každém enginu zvlášť. Projekt je žánru twin stick shooter s elementy RPG her (Role-Playing Game) [1]. Pro twin stick shooter styl ovládání platí, že jeden joystick ovladače řídí pohyb hráčovy postavy, zatímco druhý ovládá míření zbraně [2]. Hra je z pohledu třetí osoby a používá pseudo-izometrický pohled kamery, tzn. že je kamera umístěna nad postavou, ovšem perspektiva je zachována. Hra je vytvořena ve 3D. Mezi hry podobného charakteru patří například Dead Nation, Helldivers, Alienation a okrajově i Diablo 3 pro konzole. Jak název žánru napovídá, prioritním způsobem ovládání hry je pomocí herního gamepadu, konkrétně Dualshock 4 pro konzoli Playstation 4. Ovládání klávesnicí a myší je také podporováno, ale některé systémy byly koncipovány předně pro použití s gamepadem (např. uživatelské rozhraní, pohyb postavy).

Hlavním cílem projektu je vytvořit hru s kvalitní prezentací demonstrující jednotlivé prvky herního vývoje v obou enginech. Prvky, na které byl kladen největší důraz, jsou následující: tvorba prostředí, herního světa a jeho grafická prezentace (úrovně a jejich terén, statické a skeletální modely, textury a animace postav), zpracování vstupů, kvalitní uživatelské rozhraní, herní mechaniky (úkolový systém, herní předměty a inventář postavy), umělá inteligence a další. Vzhledem k tomu, že se jedná o velmi širokou škálu systémů, bude v práci kladen důraz na ty nejkritičtější pro správnou funkčnost hry a porovnání enginů.

Základ hry v Unity byl vytvořen v rámci předmětu A7B39PHA v týmu čtyř lidí: Martin Čáp, Marek Landa, Michaela Urbanovská a Tomáš Neubauer. V týmu jsem působil jako vedoucí týmu, programátor a grafik. Valná většina skriptů, které nebyly mé vlastní byly ovšem deprekovány a byla implementována vylepšená vlastní verze. Grafický vzhled hry byl vytvořen pouze mnou.

2. Základy enginů, skriptování a herní framework

2.1 Verze enginů

Vývoj hry v Unity byl zahájen ve verzi 5.4.0 (f3) a z důvodu týmové práce byla tato verze zvolena jako výchozí po zbytek vývoje. Pro vývoj hry v UE byla použita verze enginu číslo 4.13.2.

2.2 Pojmy a názvosloví

V dokumentu bude kladen důraz na standardní názvosloví v obou enginech. Pokud se tedy bude text týkat projektu v Unity, budou použity výrazy z Unity. To samé platí pro UE. Zde bych chtěl čtenáře upozornit, že v některých nešťastných případech se totožně pojmenované prvky enginů liší svým významem. Jako hlavní problematickou oblast bych uvedl pojem materiál. Materiál v Unity definuje jak má být povrch vykreslen a jednotlivé parametry shader programu, jež používá. To znamená že v Unity je materiál entita, jež používá shader program [3] (viz kapitolu "Materiály a shadery"). Naopak materiál v UE je samotný HLSL (High-Level Shader Language) program s přednastavenými hodnotami parametrů [4]. Entita používající materiálem definovaný shader v UE se pak nazývá Material Instance. Pro přehled dalších důležitých párů pojmů a názvů viz tabulku 2.1.

Unity	Unreal Engine	Význam
Material	Material Instance	objekt popisující vzhled povrchů ve hře
Shader	Material	shader program
Terrain	Landscape	objekt reprezentující půdu
Update	Tick	funkce volána každý snímek hry
Awake/Start*	BeginPlay	funkce volána na začátku hry
Shuriken	Cascade	editor částicových systémů
Mecanim	Persona	editor animací
Animator Controller	Animation Blueprint	animační ovladače/automaty
RayCast	LineTrace	prostorové dotazy (vrhání paprsků)
GameObject	Actor	herní objekty v úrovni
MonoBehaviour	AActor	základní třída všech herních objektů
Scene	Level/Map**	úrovně her

*Funkce Start je volána pouze při aktivaci objektu.
**Level je obecné pojmenování úrovní, zatímco Map je název souboru úrovně.

Tabulka 2.1: Tabulka často používaných názvů a pojmenování v obou enginech.

2.3 C#, C++, Blueprint editor

Oba enginy nabízí více možností jak hru implementovat. Zde bych stručně popsal jaké jazyky byly zvoleny a proč.

V Unity je projekt napsán v jazyce C#, který je z možné nabídky nejvíce flexibilní, nejrozšířenější a také nejlépe zdokumentovaný. Mimo C# je možné psát skripty v Javascriptu (někdy též nazývaný Unityscript, neboť se od klasického Javascriptu v mnoha ohledech liší) a v minulosti bylo možné psát skripty také v OOP jazyce Boo, jehož podpora byla ukončena v roce 2014 [5].

Unreal Engine poskytuje také více možností k přístupu vývoje. Zde můžeme využít nativní vizuální skriptovací editor, takzvaný Blueprint editor, ve kterém se vytvářejí Blueprint skripty. Ty umožňují tvorbu herní logiky bez znalosti syntaxe C++. Druhou možností je již zmiňované C++ spojené s UE API, jež tento jazyk v mnoha ohledech rozšiřuje. Třetí možností je využít jak C++, tak Blueprint editor a vytvořit tím hybridní systém, což je preferovaná možnost i podle vývojářů UE Epic Games.

Zde je ovšem důležité uvést, že doporučovaný hybrid tvoří převážně C++ třídy. Blueprint třídy by měly být v tomto hybridu použity pro listy objektové hierarchie. Jinými

slovy by Blueprint třídy měly reprezentovat konkrétní objekty použité v úrovních, které ve většině případech rozšiřují C++ třídy. Na začátku vývoje byl ze studijních důvodů zvolen hybrid, aby bylo možné porovnat enginy v plné šíři. Časem byl však systém převeden pouze na Blueprint skripty z důvodů uvedených v kapitolách "Nevýhody hybridní implementace" a "Nevýhody čisté C++ implementace". Předtím bych čtenáři nejdříve krátce představil základy UE API a jeho komunikaci s Blueprint skripty, které jsou pro pochopení volby důležité.

2.3.1 Unreal Engine API

Unreal Engine API (UE API) poskytuje mnoho funkcí a vlastních datových struktur, které je vhodné používat pro tvorbu herní logiky. Většinu UE API je možné použít jak při psaní vlastního C++ kódu, tak při tvorbě Blueprint skriptů.

UE API používá vlastní jmenovací konvence, kde každá třída musí mít speciální prefix podle jejího typu nebo rodičovské třídy. Pokud tyto konvence nejsou dodrženy, UE generuje varování nebo v některých případech i chybná hlášení. Všichni potomci třídy Actor mají prefix A, potomci třídy Object mají U, enum typy mají prefix E, rozhraní I, template třídy T, potomci SWidget (původní systém pro tvorbu UI zvaný Slate) mají prefix S a vše ostatní má prefix F.

UE API poskytuje své vlastní datové typy a struktury. Místo short, int a long jsou zde int8/uint8, 16, 32, 64 pro integer/unsigned integer s přesně daným počtem bitů. Pro práci s textem jsou zde tři základní datové typy: FString, FText a FName. FString je obdoba std::string v C++. Pro jeho tvorbu je nutné použít makro příkaz TEXT(). FText je lokalizovaný text, který můžeme vytvořit příkazy NSLOCTEXT() nebo LOCTEXT(). FName jsou identifikační názvy, které slouží k úspoře paměti a zvýšení rychlosti porovnávání. FName jsou uloženy podle indexů. Tedy pokud dva objekty sdílí stejné jméno, mají přidělen tentýž index. Samotný textový řetězec je uložen ve speciálním listu v paměti, na kterou FName hodnota odkazuje podle svého indexu. Při srovnání dvou FName hodnot jsou porovnány jejich indexy, díky čemuž není nutné porovnávat jednotlivé znaky textového řetězce, čímž je redukován počet operací. Datové struktury má UE API také vlastní. Zde se jedná o obdobu datových struktur C++ s přidanou funkcionalitou. Tři hlavní jsou TArray, TMap a TSet (obdoba std::vector, std::map, std::set) [7].

Oproti klasickému C++ podporuje UE API reflexi, díky které lze použít speciální anotace. Tyto anotace, a k ním příslušné parametry zvané specifiers, lze použít pro ko-

munikaci C++ kódu s editorem nebo Blueprint skripty. Anotace též slouží pro garbage collection, která není ve standardním C++ podporována.

Pro použití anotací je nutné upozornit UHT (Unreal Header Tool), že bude v souboru použita reflexe použitím příkazu #include "Filename.generated.h". Mezi nejčastější takové anotace patří UCLASS, UPROPERTY, UFUNCTION, USTRUCT a UENUM. Každá třída a struct musí navíc obsahovat makro příkaz GENERATED_UCLASS_BODY(), respektive GENERATED_USTRUCT_BODY(). Garbage collector bude tedy dohlížet na všechny ukazatele na objekty označené anotací UPROPERTY a proto není nutné používat příkaz delete pro uvolnění paměti (nebo jeho alternativy) [8].

Jak již bylo řečeno, každá anotace může používat seznam parametrů zvaných specifiers. Specifier parametrů je velké množství a slouží k mnoha účelům (např. často používaný parametr Category pro třídění funkcí a proměnných). Nás zde zajímají především parametry anotace UFUNCTION, které popisují viditelnost funkcí v Blueprint skriptech. Mezi čtyři základní patří BlueprintCallable, BlueprintPure, BlueprintNativeEvent a BlueprintImplementableEvent. BlueprintCallable umožňuje funkci v Blueprint skriptu volat. BlueprintPure též, zde je ovšem důležité, že funkce nijak nezmění hodnoty objektu, na kterém je volána. BlueprintNativeEvent a BlueprintImplementableEvent jsou funkce, které chceme přepsat v Blueprint třídě rozšiřující C++ třídu. BlueprintNativeEvent má oproti BlueprintImplementableEvent výchozí implementaci v C++, která je použita, pokud funkce není přepsána v Blueprint třídě [9].

2.3.2 Nevýhody hybridní implementace

Systém sice umožňuje komunikaci mezi C++ kódem a Blueprint skripty v podobě dědičnosti, kde správně anotované funkce a atributy třídy v C++ jsou k dispozici v Blueprint skriptech, které tyto C++ třídy rozšiřují. Není však možné mezi třídami a Blueprint skripty komunikovat přímo. V C++ třídě tedy není možné vytvořit nebo přetypovat objekt na Blueprint objekt. Jako příklad bych uvedl situaci, která mě přesvědčila o tom, abych předělal hru do Blueprint skriptů. Pro zbraně byla napsána C++ třída Firearm, která se starala o střílení zbraně a informování postižených cílů (sebrání životů, tvorba částicového systému kouře v místě úderu a další). Pro postavy ve hře byla také implementována C++ třída zvaná BaseCharacter. Od BaseCharacter dědily Blueprint třídy EnemyCharacter a HeroCharacter. Při zásahu nepřátelské postavy zbraní (v C++ kódu) ovšem není možnost zasažený objekt přetypovat na EnemyCharacter, neboť o něm C++ kód hovorově řečeno "nic neví". Pro jednoduché akce jako sebrání životů je samozřejmě

možnost postavu přetypovat na BaseCharacter a rozhodnout se podle tag označení, což ovšem přináší do logiky další úroveň. Ta by byla zbytečná, pokud by všechny třídy byly pouze v C++ nebo v Blueprint skriptech. Kdybychom chtěli navíc zavolat konkrétní funkci ve třídě EnemyCharacter, např. že nepřítel po smrti exploduje, tuto specifickou funkci opět neznáme, a zde nám nepomůže ani znalost, že se jedná o nepřátelskou postavu (díky tag označení), neboť neexistuje žádný způsob jak tuto funkci zavolat.

2.3.3 Nevýhody čisté C++ implementace

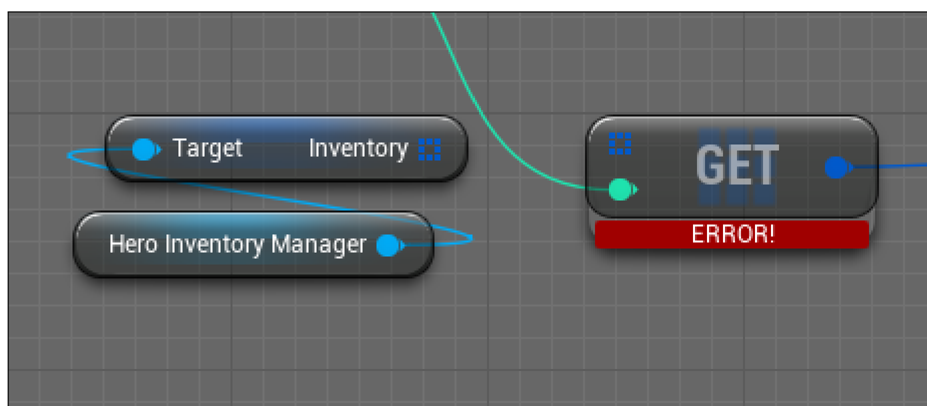
Důvod proč nebylo zvoleno čisté C++ je pragmatický. Při použití C++ v UE se pracuje s enginem jako s celkem (jedním velkým projektem). Epic Games poskytují speciální funkcionalitu zvanou Hot Reload pro výrazné urychlení kompilace [10]. Ta se ovšem i přes specifikace počítače (32GB RAM, SSD disk, i7 6700K Skylake a NVidia GeForce GTX1080) pohybuje kolem 15 sekund po změně jednoho řádku v kódu (např. výpis na konzoli). Velmi pomalé je též doplňování slov IntelliSense, které po změnách a uložení kódu vyžaduje podobnou dobu na aktualizaci. Kompilace Blueprint skriptů je téměř instantní (podobně jako v Unity) a proto byly zvoleny jako výchozí pro tento projekt.

2.3.4 Nevýhody Blueprint skriptů

Při práci s Blueprint skripty jsem také narazil na mnohá omezení, která bych zde rád uvedl. První obecně diskutovanou nevýhodou je jejich horší výkon jak C++ kód. Zde se jedná o velmi těžce porovnatelné téma, neboť v různých testovacích podmínkách dostaneme odlišné výsledky. Konstantou zůstává, že Blueprint skripty budou vždy pomalejší než C++ kód, jen je otázkou, jak moc a za jakých podmínek [11]. Hlavní zpomalení hry nejsou však ve většině případech způsobena herní logikou, a proto bych se zde zaměřil na praktičtější problémy.

Prvním z těchto problémů je vysoká poruchovost Blueprint skriptů v případech, kdy manipulujeme se strukturami (struct), které jsou v nich použity. Jako příklad bych uvedl problém, se kterým jsem se potýkal většinu času při vývoji projektu. Po úpravě ItemData (struct) se vždy po spuštění UE editoru "rozpojí" uzly mezi polem obsahujícím tuto strukturu a jednotlivými funkcemi (např. Get, Length, ForeachLoop) (viz obrázek 2.1). Ve většině případech takový kód nelze zkompilovat a uživatel je nucen chybu opravit. V některých případech se ovšem o chybu, na kterou by nás kompilátor upozornil, jednat nemusí (ForeachLoop). Zde tedy hrozí, že se ve hře objeví chyba, o které se ne-

musíme dozvědět ihned. Řešením problému je smazání pole této struktury, vytvoření nového a oprava všech přerušených spojů. Po této opravě by se ve většině případech problém neměl opakovat. Provedeme-li však opětovnou úpravu struktury (což se děje často např. při tvorbě systému předmětů), musíme tento proces celý opakovat. Proto je dobré si strukturu před její implementací a použitím patřičně rozmyslet.



Obrázek 2.1: Rozpojené uzly pole Inventory a funkce Get.

Jednou z dalších velkých slabin UE Blueprint skriptování je rozlišení funkcí a událostí (Events) a implementace rozhraní, které je předepisuje. Blueprint editor odlišuje funkce a události tak, že funkce mají návratovou hodnotu, zatímco události ne (obdoba procedur v jiných jazycích). Lze ovšem standardně vytvořit funkci, která nemá návratovou hodnotu. Zde je na uživateli, zda použije funkci nebo událost. Blueprint editor se rozděluje na Graphs (Event Graph), Functions, Interfaces, Macros, Variables a Event Dispatchers. V Event Graph můžeme přidávat údálosti, implementovat jejich funkcionalitu, ovšem, jak již bylo řečeno, tyto události nemohou mít návratové hodnoty. Složka Functions slouží pro tvorbu a přepsání (override) funkcí. Ty sice mohou mít návratové hodnoty, ale nepodporují manipulaci s časem (musí tedy být vykonány okamžitě v rámci jednoho snímku hry). To znamená, že uzly jako Delay, Timeline a další nejsou ve funkcích povoleny (pro více informací viz kapitolu “Práce s časem a lineární interpolace”). Pokud takové uzly potřebujeme použít, musíme vytvořit událost, kterou z funkce zavoláme. Velká komplikace však nastává při implementaci různých rozhraní. Pokud rozhraní definuje funkci, pak ve třídě, která rozhraní implementuje, je tuto funkci možné implementovat ve složce Interfaces, což je naprosto v pořádku. Pokud na druhou stranu definujeme v rozhraní funkci, jež nemá návratovou hodnotu, není taková funkce ve třídě, která ji implementuje, předepsána. Pokud funkci neimplementujeme, nejedná se o kompilační chybu! Funkce je implicitně vytvořena editorem (bez žádné funkcionality). Uživatel tak vůbec neví, zda

je správně implementováno rozhraní a tento fakt nelze zjistit jinak, než každou funkci manuálně najít v Event Graph a tím si ověřit, že je rozhraní úplně implementováno, což pokořuje základní princip jeho použití. Někteří vývojáři doporučují v rozhraní definovat pouze funkce s návratovými hodnotami s tím, že pokud tato hodnota není potřebná, zvolíme libovolnou nepodstatnou hodnotu jako je například boolean s názvem executed, abychom byli donuceni rozhraní patřičně implementovat [12].

Další nepříjemností Blueprint skriptů je, že chybí podpora pro přetížení (overload) funkcí, neboť volání funkcí v Blueprint editoru se řídí jejich názvy [13]. To samé platí i pro funkce v C++, které mají anotaci UFUNCTION. To vede k velkému množství funkcí s dlouhými názvy a nepřehlednému kódu. Tento problém je nejvíce znatelný při práci s daty jako je například v QuestManager třídě.

2.4 Herní framework

Důležitou částí UE je jeho předpřipravený herní framework, který je možné využít při tvorbě vlastní hry [14]. Stěžejní třídou v Unity je třída Monobehaviour, kterou musí rozšiřovat každý skript, který chceme umístit na objekt ve scéně hry. Mimo to Unity žádný nativní herní framework nenabízí, je zde ale možné použít skripty pro pohyb postavy, ovládání kamery a další nabízené v balíčku Standard Assets. Tyto skripty je však nutné do vlastní hry správně zakomponovat. UE framework nabízí všechny základní prvky a mechanismy hry, jež je vhodné použít při tvorbě hry vlastní. Pokud si to uživatel nepřeje, není nucen framework použít až na pár základních tříd jako je Controller umístěný na hráčově postavě, který je vyžadován pro naslouchání ke vstupům a k funkčnosti kamery ve hře.

Herní framework UE obsahuje třídy řídící průběh hry jako GameMode a GameState, tedy pravidla hry a její současný stav. Obodobou MonoBehaviour je třída Actor, ta reprezentuje každý objekt, který může být do herního světa vložen. Samozřejmě ne vše je do herního světa vkládáno, proto je Actor potomkem třídy Object, která reprezentuje každý objekt v UE.

Kromě zmiňovaných patří mezi stěžejní třídy herního frameworku UE navíc:

- Pawn - agenti ve hře, kteří mohou být ovládáni kontrolerem (Controller). Pawn nemusí být humanoidní postava
- Character - humanoidní Pawn třída. Obsahuje CapsuleComponent a CharacterMovementComponent
- Controller - ovladače, které řídí pohyb postav třídy Pawn
  - PlayerController - pro zpracování vstupů hráče
  - AIController - ovladač postav s umělou inteligencí, většinou využívá behaviorální strom

Mezi další potomky třídy Actor např. patří: CameraActor, DecalActor, Emitter, Light, SkeletalMeshActor, StaticMeshActor a další.

Podobně jako Unity, nabízí UE komponenty chování pro jednotlivé objekty ve hře. Zde se jedná o ActorComponent a SceneComponent, který má navíc vlastní transform, tedy souřadnice, rotaci a škálu.

UE také poskytuje vlastní SaveGame systém, kde je serializace SaveGame objektu interně zařízena. Dále jsou již implementovány základní herní mechanismy jako je pohyb postav (komponent CharacterMovement), projektily (komponent ProjectileMovement) a další (např. třída CameraShake). V Unity můžeme podobné skript komponenty nalézt ve Standard Assets.

2.5 Práce s časem a lineární interpolace

Nedílnou součástí práce s enginy je práce s časem. Jedním z častých příkladů je lineární interpolace, díky které můžeme animovat změnu stavu objektů v čase (posun, barevný přechod, zmizení a další). Lineární interpolace se v projektu vyskytuje na mnoha místech, je použita pro postupné mizení nepřátel, naváděné předměty, nebo například pro změnu polohy nepřátelské postavy během jejího skoku.

Krátce bych zde tedy rozebral jednotlivé pomůcky a postupy pro práci s časem v obou enginech. Jak v Unity, tak v UE máme funkci, jež je volána každý snímek (frame) hry a tvoří tak její základní smyčku. V Unity se jedná o funkci Update, v UE o funkci Tick. Pro každou známe délku trvání předešlého snímku zvanou deltaTime. Unity navíc nabízí dvě specializované funkce FixedUpdate a LateUpdate. FixedUpdate zaručuje konstantní délku deltaTime zvanou fixedDeltaTime a LateUpdate je volána každý snímek

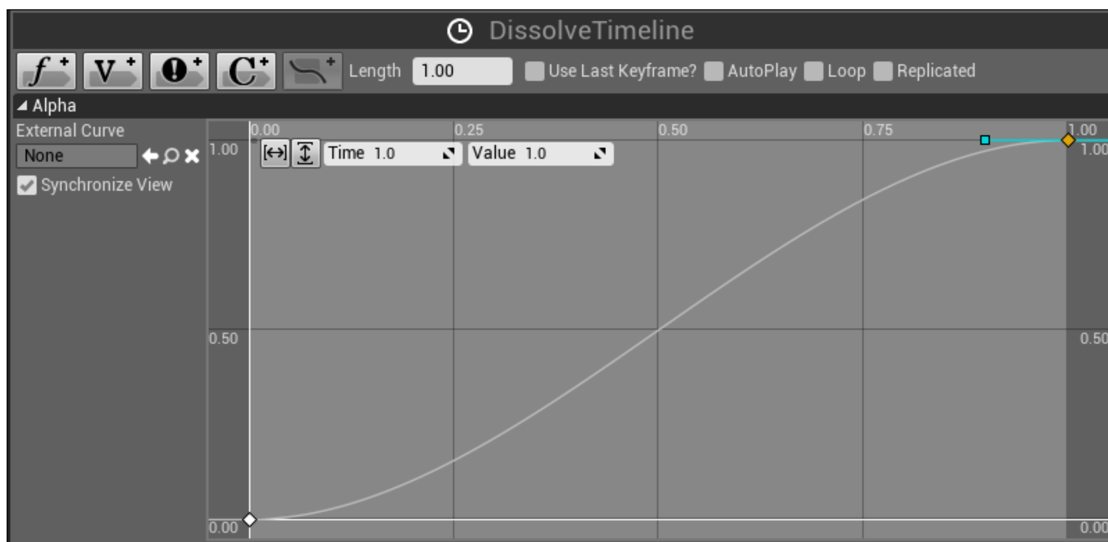
po provedení Update funkcí na všech objektech ve scéně. FixedUpdate je vhodné použít např. při fyzikální simulaci, kde je pevně daný časový krok potřebný. UE nenabízí žádné funkce pro zajištění konstantního časového kroku.

Pokud bychom chtěli lineárně interpolovat v těchto funkcích, musíme použít časovače pro ukončení interpolace. Naštěstí oba enginy nabízí jednoduchá řešení, jak se použití časovačů vyhnout.

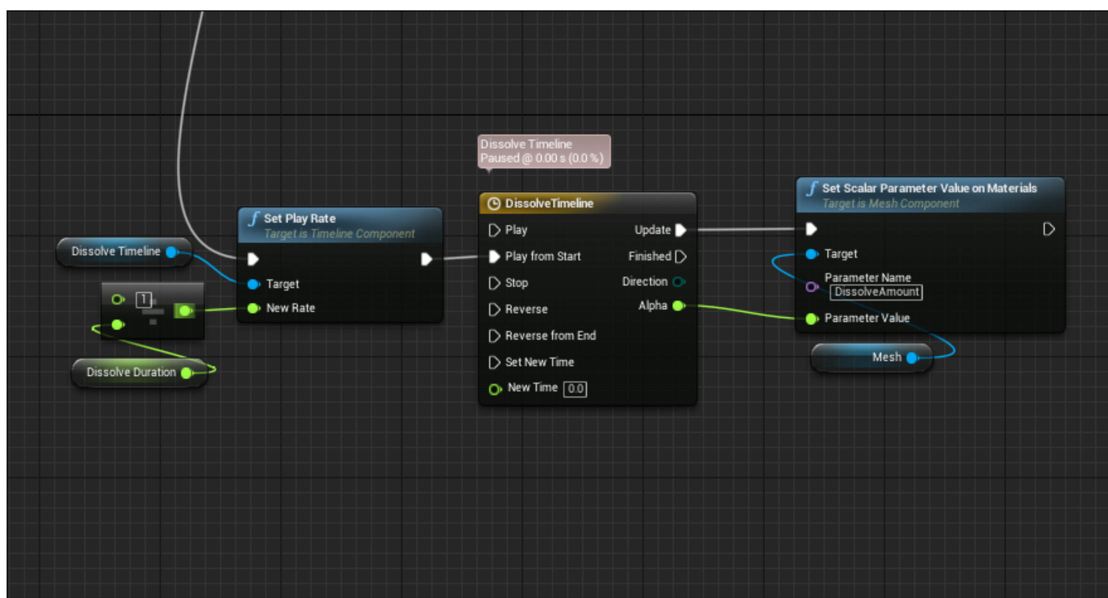
V Unity můžeme použít tzv. coroutines. Coroutines jsou stejně jako Update volány v každém snímku, přičemž coroutine bude zavolána vždy po Update v daném snímku. Vykonání coroutines je možné pozastavit pomocí příkazu yield. Příkaz yield může být použit pro pozastavení coroutine na specifikovaný počet sekund (yield return new WaitForSeconds), čekání do určitého úseku nebo konce snímku (yield return null, respektive yield return new WaitForEndOfFrame), pro úplné ukončení coroutine (yield break) nebo například pro spuštění jiné coroutine a pozastavení, dokud nebude vykonána (yield return StartCoroutine) (viz obrázek 2.3).

Další dvě funkce, které je možné použít, jsou Invoke pro zavolání funkce po určitém časovém intervalu a InvokeRepeating pro opakované volání funkce s pevným časovým intervalem mezi voláními.

V UE je možné pozastavit vykonávání na určitou dobu času pomocí uzlu Delay. Dále UE poskytuje nativní časovače, které fungují podobně jako Invoke a InvokeRepeating v Unity. Pro interpolaci je v UE specializovaný nástroj zvaný Timeline. Jedná se o objekt, jež umožňuje definovat křivky závislé na čase, jejichž hodnoty lze využít pro animaci objektů a interpolaci mezi různými hodnotami (viz obrázek 2.2). Timelines byly specificky navrženy pro použití u jednoduchých animací. Pro složité animace je v UE vhodné použít filmový editor zvaný Sequencer (následník systému Matinee). V projektu byl pouze použit pro jednoduchý fade in/out obrazovky při přechodech mezi úrovněmi, je však dobré ho zmínit, neboť Unity v současné době žádný filmový editor nenabízí. Nicméně je pro budoucí verze Unity plánováno vydání filmového editoru zvaného Timeline (obdoba Sequencer UE). Předpokládaný datum vydání je červen 2017 pro verzi 2017.1.0 (beta) [15].



(a) DissolveTimeline pro postupné zmizení nepřítel.



(b) Použití DissolveTimeline pro nastavení float parametru materiálů během časového intervalu DissolveDuration.

Obrázek 2.2: Použití Timeline komponentu v UE.

```
private IEnumerator Dissolve(float time) {
  for (float t = 0f; t < 1.0f; t += Time.deltaTime / time) {
    foreach (Material mat in mats) {
      mat.SetFloat ("_DissolveAmount", t);
    }
    yield return null;
  }
}
```

Obrázek 2.3: Použití coroutine pro postupné zmizení nepřátel v Unity. Všimněte si použití Time.deltaTime pro hladký průběh postupného zmizení.

2.6 Ukládání hry

Systém ukládání hry je v obou enginech implementován velmi podobně. V UE je nám poskytnuta třída SaveGame.

SaveGame nám nabízí rozhraní pro serializaci dat. Pro uložení libovolných dat je možné vytvořit potomka této třídy a v něm vytvořit proměnné, které chceme serializovat. Serializovaná data jsou uložena v adresáři `ProjectFolder\Saved\SaveGames` pokud hrajeme hru v editoru nebo v adresáři `C:\Users\Username\AppData\Local\ProjectName\Saved\SaveGames` pokud se jedná o standalone hru.

V Unity je přístup obdobný. Pro ukládání dat lze použít abstraktní serializovatelnou třídu SaveGame a její potomky, kteří musí být také serializovatelní. Do těchto potomků vkládáme data, která chceme uložit. Pro ukládání byla vytvořena pomocná třída SaveManager, která poskytuje rozhraní pro tvorbu složky, uložení SaveGame a nahrání SaveGame podle jména souboru. Uložená data standalone hry můžeme pak většinou nalézt v adresáři `C:\Users\Username\AppData\LocalLow\CompanyName\ProjectName\saves`.

3. Herní svět

3.1 Herní objekty, prefabrikáty a Blueprint třídy

Každá herní scéna je uspořádána do stromu objektů a jejich transformací. V Unity se herní objekty nazývají GameObject a v UE se nazývají Actor. V textu však nadále bude používán český překlad - herní objekt - pro oba enginy. Důležité je zmínit, že herní objekt v Unity je jakýsi nositel komponentů a nemá žádné vlastní speciální vlastnosti [6]. Hlavní komponent v Unity je tzv. Transform komponent, jenž udává polohu, rotaci a škálu objektu. Některé skupiny objektů se budou ve hře opakovat, a proto je vhodné z nich vytvořit znovupoužitelné objekty v Unity, tzv. prefabrikáty. V UE je každý herní objekt přímo C++ nebo Blueprint třída, jež rozšiřuje třídu Actor (AActor s prefixem) a je také znovupoužitelná. Filozofie obou přístupů je podobná, je však dobré zmínit některé odlišnosti mezi oběma systémy. Jak prefabrikáty, tak Blueprint třídy reprezentují jednoduché i složité entity, které mohou být do herního světa vkládány.

V Unity vytváříme prefabrikáty z již existujících objektů nebo stromů objektů a jejich komponentů ve scéně. Pro tvorbu prefabrikátu je nutné manuálně přemístit objekt ze scény do adresáře hry. Ve scéně poté používáme instance těchto prefabrikátů. U každé instance lze provádět tři možné operace: Select, Revert a Apply. Select zvolí prefabrikát, podle kterého byla instance vytvořena v adresáři. Revert navrátí zpět všechny nově nastavené hodnoty instance na výchozí hodnoty prefabrikátu a Apply přepíše hodnoty prefabrikátu podle hodnot zvolené instance [6].

Při práci s instancí máme přístup ke všem potomkům prefabrikátu a jeho veřejným nebo privátním proměnným, pakliže jsou označeny anotací [SerializeField]. Veřejné C# properties třídy nejsou v Inspectoru Unity bohužel dostupné. Totéž platí obecně pro herní objekty ve scéně. Při práci s prefabrikátem v adresáři můžeme manipulovat pouze s jeho potomky v hloubce 1 (tedy potomci potomků již nejsou přístupní).

V UE je též možné vytvořit třídu z již existujícího objektu v úrovni pomocí příkazu Blueprints → Convert Selected Actor to Blueprint Class. Zde si může čtenář všimnout, že to samé nebylo řečeno pro skupiny objektů, neboť Blueprint třída nemůže obsahovat více herních objektů (může obsahovat pouze více komponent - Blueprints → Convert Selected Components to Blueprint Class based on Actor that contains the Component). UE nabízí způsob, jak vytvářet znovupoužitelnou hierarchii objektů v podobě Child Actor komponentu. Zde ovšem můžeme narazit na velký problém. Všechny editovatelné proměnné (veřejné pro editor) tohoto potomka nelze nastavit v jeho rodičovské Blueprint třídě ani v editoru samotném. Pokud navíc chceme s potomkem pracovat, je nutné ho nejdříve přetypovat a uložit ho do proměnné třídy. Zde se jedná o problém ve verzi UE 4.13.2, která byla použita pro tvorbu projektu. Tento problém byl údajně vyřešen ve verzi 4.14 [16].

Možným řešením, jak problém obejít, je vytvořit komponent, který má stejnou funkcionalitu jako Child Actor třída náležející Child Actor komponentu. V případě, kdy Child Actor třída měla své vlastní komponenty, jako je například Box Collision, je nutné je vytvořit v rodičovské třídě a náhradnímu komponentu předat reference na nové komponenty, aby je mohl používat.

Jako příklad bych uvedl ItemSpawner a ItemSpawnerComponent, jež mají na starost vytvářet nové předměty z databáze předmětů v úrovni. ItemSpawner je Blueprint třída dědící od třídy Actor. Má vlastní komponent Box Collision (kolizní těleso kvádru), který slouží pro generaci náhodných lokací, ve kterých budou předměty vytvářeny. Pokud bychom chtěli ItemSpawner použít v rámci jiné třídy, jako je například truhla s předměty (třída LootChest), musíme pro tuto truhlu vytvořit duplicitní proměnné, které používá ItemSpawner, jako jsou například počet předmětů, jejich rarita a další. Hodnoty těchto proměnných na začátku hry překopírujeme do ItemSpawner. Pokud naopak použijeme ItemSpawnerComponent, nemusíme vytvářet žádné duplicitní proměnné. Musíme však ve třídě LootChest přidat komponent Box Collision, jehož referenci předáme ItemSpawnerComponent komponentu, který jej bude používat pro generaci náhodných lokací předmětů.

3.2 Změna úrovní a persistence dat

Jedním z důležitých témat při vývoji hry námi vybraného žánru je přechod a persistence dat mezi jednotlivými úrovněmi/scénami. Zde je ovšem důležité nejdříve odlišit o jaký přechod mezi úrovněmi se jedná. První možností je přechod, ve kterém je nová úroveň načítána na hlavním herním vlákně, čímž se blokuje jakákoliv další aktivita ve hře. Druhá možnost je asynchronní načítání úrovně, které probíhá ve vlastním vlákně, a při kterém není nutné hru pozastavit.

Na úvod bych uvedl, že koncept a design hry se v průběhu vývoje měnil, proto se mohou mezi oběma verzemi vyskytovat určité rozdíly. Nicméně je v obou enginech použito asynchronní nahrávání úrovní. V Unity se jedná o LoadLevelAsync [17], zatímco v UE je tento proces nazýván jako Level Streaming (funkce LoadStreamLevel) [18].

Důležité je podotknout, že UE poskytuje nativní řešení problému v podobě Level Streaming Volumes, v Unity je takový systém nutné implementovat. Level Streaming Volumes jsou oblasti, které zajišťují nahrání a mazání úrovní podle současné pozice kamery (pozor! - ne hráčovy postavy). V každém snímku hry (frame) je pro každou úroveň ověřeno, zda se kamera nevyskytuje v Level Streaming Volume asociovaném s touto úrovní. Pakliže ano a úroveň není v daný moment načtena v paměti, dojde k jejímu načtení, a naopak, pokud hráč není v Level Streaming Volume úrovně a zároveň úroveň načtena je, pak je asynchronně smazána z paměti [19].

V UE existuje tzv. persistentní úroveň (Persistent Level), která je persistentní pro celou hru (pokud není použit pevný přechod mezi úrovněmi). Level Streaming Volumes musí být umístěny v této úrovni, aby fungovaly. Důležité je také podotknout, že každá úroveň může mít jednu i více Level Streaming Volumes a každé Level Streaming Volume může být asociováno s několika úrovněmi (vztah m:n).

Persistentní úroveň může být právě jedna a slouží k managementu tzv. aditivních úrovní. Jak persistentní úroveň, tak úrovně aditivní jsou běžné mapy (asset úrovně v terminologii UE). V editoru v tabulce World Outline (obdoba Hierarchy v Unity) můžeme vidět všechny objekty ze všech viditelných úrovní. V tabulce Levels můžeme tvořit, mazat, vkládat a volit úrovně, nastavit jejich viditelnost ve hře, přemisťovat mezi nimi objekty (pravé tlačítko myši → Move Selected Actors) a také jím nastavit Level Streaming Volumes. Nastavení Level Streaming Volumes je nešikovně schováno ve vlastním menu nazvaném Level Details, které lze otevřít kliknutím na lupu s otazníkem v tabulce Levels.

V obou enginech je vlastní systém založen na interakci s objektem, který vyvolá přechod a změní pozici hráčovy postavy. Systém v UE je založen na aditivním nahrávání nových úrovní k úrovni persistentní. Zde se implementace v enginech liší v tom, že systém v Unity nevyužívá aditivních scén, neboť se koncept hry v průběhu vývoje změnil. Aditivní scény jsou ovšem v Unity také podporovány.

Aditivní Level Streaming v UE má nevýhodu ve zpracování navigačních dat (NavMesh). Navigační data nelze nahrát společně s aditivní úrovní a je tedy nutné je při otevření nové úrovně buď dynamicky vytvořit nebo umístit NavMeshVolume (oblast, ve které chceme navigační data vytvořit) aditivní úrovně do úrovně persistentní. Pro použití dynamické generace navigačních dat je nutné nastavit hodnotu automaticky vytvořeného objektu RecastNavMesh - Runtime Generation na Dynamic. V Details panelu UE navíc není možné přiřazovat reference na objekty z odlišných úrovní, proto byly všechny interaktivní objekty pro přechod mezi úrovněmi (dveře, portal) umístěny do persistentní úrovně, aby na sebe mohly navzájem odkazovat, což je použito pro nastavení pozic hráče.

3.2.1 Unity

V Unity je, jak již bylo zmiňováno, použito asynchronní nahrávání úrovní pomocí statické funkce LoadLevelAsync. Celý tento systém je spravován vlastní singleton třídou LevelManager, která se navíc stará o zobrazení obrazovky načítání (loading screen).

LevelManager v sobě drží informace o současné a předešlé scéně, kterou hráč navštívil. Scéna je reprezentována svým číselným indexem (buildIndex), který je možné nastavit v Build Settings projektu. LevelManager má dále na starost načítání souřadnic hráče pro přechod mezi scénami ze souboru s vlastním textovým formátem. Formát je následující: start > destination: x y z.

Samotný přechod probíhá následovně. Nejdříve je hráči zobrazena obrazovka načítání, poté je spuštěno asynchronní nahrávání (zde si o něm udržujeme informace v podobě AsyncOperation objektu). Pro úplnou kontrolu nad načítáním nastavíme hodnotu allowSceneActivation na false, čímž zaručíme, že nebude přechod proveden, dokud nedokončíme všechny námi požadované akce. Postup načítání můžeme sledovat v procentuální hodnotě progress. Protože není povolena automatická aktivace nově načtené scény (allowSceneActivation), hodnota progress se zastaví na magickém čísle 0.9f (90%). Posledních 10% je samotná operace aktivace scény (jedná se o náhodně vybraný poměr, není nijak popisný co se týče počtu/náročnosti operací nahrávání) [20]. Jakmile tedy dosáhne progress 90%, můžeme manipulovat s novou scénou, pro kterou nastavíme potřebné hodnoty (například

změníme pozici hráčovy postavy), nastavíme allowSceneActivation na true a deaktivujeme obrazovku načítání.

Pokud si přejeme provést různé akce na objektech ve scéně při přechodu mezi scénami, můžeme použít delegáty pro události SceneManager.sceneLoaded/sceneUnloaded. Tyto delegáty jsou ve většině případech využity na načtení nebo uložení dat hry jako jsou splněné úkoly nebo inventář hráče.

3.2.2 Unreal Engine

V UE je spojen vlastní systém, který replikuje chování v Unity, se systémem poskytnutým od Epic Games (Level Streaming Volumes). V persistentní úrovni je též singleton objekt zvaný LevelManager, který poskytuje rozhraní pro asynchronní nahrání a smazání úrovně společně se zobrazením obrazovky načítání, který funguje podobně jako LevelManager v Unity.

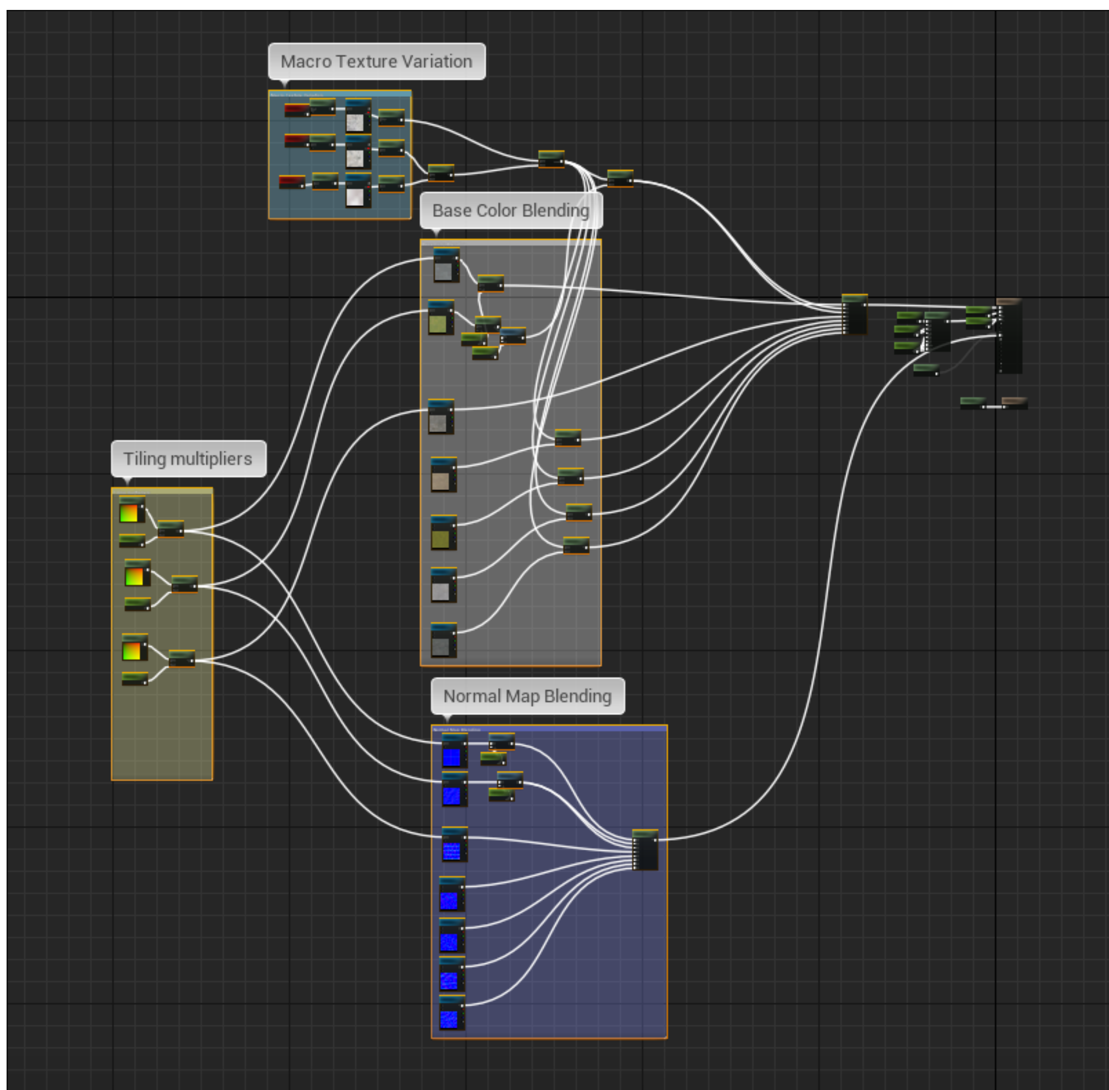
Co se týče událostí vyvolaných při změně úrovně je zde ovšem aplikován více manuální přístup, neboť události OnLevelLoaded a OnLevelUnloaded jsou zde volány pro každou úroveň zvlášť. Proto byl vytvořen dispečer (Event Dispatcher) LevelLoaded a LevelUnloaded v LevelManager třídě. Pokud je úroveň nahrána nebo smazána, je zavolána dispečerova funkce LevelLoaded, respektive LevelUnloaded, která není na úrovni závislá. OnLevelLoaded a OnLevelUnloaded jsou navíc zavolány pouze v případě, že jsou použity nativní Level Streaming Volumes. Pokud je použit vlastní přechod pomocí funkce LoadStreamLevel nebo UnloadStreamLevel, je nutné funkce zavolat manuálně. Tento systém vznikl především pro notifikaci úkolového systému při změně libovolné úrovně, který vyhledá všechny objekty v úrovni, jež jsou součástí nějakého úkolu a aktualizuje jejich stav. Uživatel může manuálně nastavit, u kterých úrovní bude dispečink použit. Důležité je zmínit, že Level Streaming Volumes nepovolují klasické Overlap události (nelze naslouchat, zda do nich někdo vstoupil, aniž bychom neporušili jejich originální funkcionalitu).

3.3 Terrain a Landscape

V této kapitole bych se zaměřil na rozdíly systémů pro tvorbu půdy a jejího vzhledu v obou enginech. Pro hru námi zvoleného žánru se jedná o kritické téma, už jen z toho důvodu, že hráč díky umístění kamery vidí půdu většinu času stráveného hraním. Půda by tedy neměla být monotónní a měla by dotvářet atmosféru hry. Důležité je ovšem podotknout, že hra svým charakterem vyžaduje spíše plochý hrací terén. Jak Unity, tak UE nabízí svůj vlastní systém pro tvorbu půdy. V Unity se takové půdě říká Terrain, v UE se nazývá Landscape.

Jedním z hlavních rozdílů, se kterými se vývojář setká v implementaci obou systémů, je dostupnost základního shaderu pro rychlou tvorbu v Unity, která poskytuje na výběr ze tří možností. První z nich je Built-in Standard shader. To je PBR (Physically-Based Rendering) shader, který se v Unity vyskytuje od verze 5.x. Tento materiál nám dovoluje texturovat plochu následovně: Pro každý "splat" (vrstvu textur) můžeme definovat jeho albedo a smoothness v jedné textuře, kde smoothness se řídí jejím alfa kanálem, normálovou mapu v druhé textuře a metalness jako skalární parametr. Zbylé dva shadery jsou Built-in Legacy Diffuse a Built-in Legacy Specular, které jsou z verze 4.x (nejsou PBR) a jak název napovídá, není již vhodné je používat v nových projektech. Je také možné použít vlastní shader. Zde je velkým nedostatkem Unity chybějící podpora pro rychlou tvorbu shaderů v podobě vizuálního editoru. Pokud by si vývojář přál vytvořit svůj vlastní, je buď nutné celý napsat, nebo jak je doporučeno v dokumentaci Unity, rozšířit Built-in Standard shader [21].

UE žádné základní materiály, které by podporovaly vrstvení textur, na Landscape neposkytuje, ovšem jeho tvorba pomocí vestavěného Material editoru není problematická. Základ takového materiálu, který nabízí stejné možnosti jako Build-in Standard shader v Unity a více, lze vytvořit podle postupu uvedeného v dokumentaci [22].



Obrázek 3.1: Vytvořený materiál půdy v UE. Můžeme zde vidět dvě hlavní části: Base Color Blending a Normal Map Blending, kde mícháme jednotlivé albedo textury a normal mapy pomocí uzlů Layer Blend. Dále si můžeme všimnout levé sekce nazvané Tiling multipliers. Tiling multipliers používá Landscape Coordinates uzly pro parametrizaci velikosti jednotlivých vrstev (splats).

Důležité je také zmínit, že Material editor poskytuje funkce přímo pro tvorbu materiálů pro Landscape objekty v UE. To znamená, že je možné jednodušše vytvořit vlastní míchání textur podle alfa hodnoty nebo podle výškové mapy (height map), masku neviditelných částí terénu a dokonce i teselaci a displacement. Tyto funkcionality Unity Built-in Standard shader nenabízí a je nutné napsat vlastní shader pro jejich použití.

Další značnou odlišností je rozdělení terénu v UE na části, které slouží k výpočtu viditelnosti, LOD (Level of Detail) a kolizí. UE Landscape je rozdělen na části zvané Components, Sections a Quads. Každý Component je tvořen Sections a každá Section se skládá z Quads. Pro výpočty jsou použity Components. HeightData (výšková mapa) je pro každý Component uložena zvlášť v jedné textuře, jejíž rozlišení by mělo být ve tvaru $2^n \times 2^n$ (pro možné použití Mip Map). Volba velikosti a počtu Components má velký vliv na výkon. Volbou menší velikosti a většího počtu dosáhneme hladších přechodů za cenu ztráty výkonu. Volbou větší velikosti a menšího počtu získáme lepší výkon. V Unity má každý Terrain objekt svou vlastní výškovou mapu (angl. height map - HeightData v UE). Pokud bychom tedy chtěli pospojovat více různých výškových map, je nutné vytvořit více Terrain objektů, zatímco v UE je možné použít zmíněné Components. Unity Terrain nabízí svá nastavení pro LOD. Je zde možné určit zda budou stromy, tráva a detailní modely vykresleny (Draw), jejich maximální vzdálenost viditelnosti od kamery (Detail Distance, Tree Distance), hustota trávy a detailních modelů (Detail Density) a poté ještě další nastavení pro stromy jako je přechod jejich modelu na billboard texturu a rychlost takového přechodu. V UE jsou tato nastavení vázána na jeho dekompozici do částí a na samotné objekty jako je tráva, flóra a další.

3.4 Foliage

Velkou částí půdy hry je její flóra a detailní modely jako jsou například kameny. V Unity je možné na terén přidat trávu reprezentovanou 2D texturou. Trávě je možné nastavit její minimální a maximální šířku a výšku, její náhodné uspořádání do skupin, zdravou a suchou barvu (tint) a zda se jedná o billboard texturu (vždy rotována vůči kameře). Druhou možností je přidat tzv. detailní modely (Detail Mesh). Ty mají podobná nastavení, pouze je zde místo billboard parametru nastavení vykreslovacího módu (Render Mode). Vykreslovací mód může mít buď hodnotu Grass, jež model zploští do 2D textury, která se bude následně chovat jako klasická tráva (včetně simulace větru), nebo hodnotu Vertex Lit, jež vykreslí objekt takový jaký je. Vertex Lit detailní objekty ovšem nepodporují vrhání stínů [23].

Obrázek 3.2: Tráva a kameny v Unity. Můžeme si zde všimnout zajímavého zabarvení trávy, jež lze upravit v nastavení Terrain v tabulce Wind Settings for Grass > Grass Tint. Dále je zde možné pozorovat, že kameny nemají vlastní stíny.

UE nabízí speciální editační mód pro tvorbu trávy, flóry a obecně detailních modelů pro povrch půdy zvaný Foliage Edit Mode [24]. Pro přidání trávy/rostlin je nutné vytvořit objekt typu Foliage Type, přiřadit mu statický model, jež chceme pro rostliny použít a nastavit mu jednotlivé parametry. Pro každý Foliage Type je možné vymezit nadmořskou výšku (od počátku kartézské soustavy souřadnic úrovně), ve které se může nacházet. Foliage v UE navíc podporuje LOD nastavení statického modelu, kolize a vrhání statických/dynamických stínů (jako u běžného statického modelu). Každý model Foliage Type může mít vlastní kolizní matici a také fyzický materiál. Kolizní obalová tělesa těchto modelů pak mohou být zahrnuta do výpočtu navigačních map. Zde nám tedy systém v UE nabízí mnohem větší volnost a kontrolu. Pokud bychom si přáli do úrovně přidat velké modely jako jsou např. kameny/skály o velikosti hráčovy postavy, můžeme pro rychlou tvorbu použít tento editační mód. V Unity u detailních modelů nejsou kolize podporovány, a proto je nutné velké objekty po úrovních rozmístit manuálně.

Editační mód pro tvorbu a editaci flóry a detailních modelů v UE nám umožňuje určit hustotu, rozestupy a škálování (se zamčením různých os) modelů při jejich tvorbě. Navíc je podporována možnost přidávat více druhů Foliage Types do úrovně zároveň. Editor nás informuje o současném počtu jejich instancí. Další velkou výhodou editoru je jeho Lasso nástroj pro rychlou a jednoduchou selekci více druhů Foliage Type modelů zároveň. Navíc je možné selektovat všechny instance určitého druhu rostlin tak, že

klikneme pravým tlačítkem myši na rostlinný typ a zvolíme možnost Select All Instances.

Obrázek 3.3: Foliage Edit Mode a vzhled flóry v UE. Můžeme si zde všimnout dynamických stínů u trávy.

Systém tvorby flóry je tedy obecně mnohem robustnější v UE, neboť nabízí širokou škálu možností, jak rostliny přidávat a editovat. Podporuje kolize, stíny a zahrnutí jeho modelů do výpočtu navigační plochy. Díky tomu je snadné rychle vytvářet detailní prostředí plné rostlin a dalších objektů jako jsou kameny. Systém Unity je více přímočarý. Vytvořit půdu s trávou je otázka pár vteřin, dokonce bude u trávy simulován její pohyb podle větru, což je např. v UE nutné implementovat v materiálu trávy manuálně, ovšem pro pozdější stádia vývoje se systém jeví jako poněkud omezený.

3.5 Geometry Brushes v UE

Při vývoji každé hry je dobré vytvářet prototypy úrovní, podle kterých je v pozdějších fázích vývoje vytvořena jejich finální verze. UE poskytuje speciální pomůcku zvanou Geometry Brushes pro zrychlení prototypování. Jedná se o jeden z editačních módů UE editoru, který umožňuje vytvářet a editovat geometrická primitiva jako jsou krychle, kužel, válec, koule a speciálně schody. Tento mód umožňuje manipulaci s uzly, hranami a stěnami těchto primitiv. Kromě jednoduché manipulace a základních operací jako je extruze je možné využít principů tzv. Constructive Solid Geometry (CSG). CSG

nám umožňuje provádět boolean operace mezi jednotlivými primitivy [37]. UE podporuje pouze aditivní a subtraktivní štětce, jež přidají nebo smažou geometrii z geometrie stávající. Jednotlivým stěnám vytvořených modelů lze přiřadit materiály a navíc je možné editovat jejich UV mapy (rotace, škála). Model pak lze konvertovat buď na statický model nebo na speciální herní objekt jako je např. BlockingVolume, jež tvoří neviditelné zdi úrovní.

V Unity žádná taková technologie není. Je jí možné zakoupit na Asset Store. Nejpopulárnější volbou je ProBuilder, jehož cena se v době vypracování práce pohybovala kolem 100$. Je ovšem možné použít ProBuilder Basic, který nabízí základní funkcionalitu a je zdarma.

Je nutné připomenout, že Geometry Brushes jsou převážně vhodné pro prototypování úrovní. Jejich zastoupení ve výsledných úrovních je mizivé, neboť jsou výkonnostně mnohem náročnější než statické modely.

4. Uživatel

4.1 Uživatelské rozhraní

Co se týče tvorby uživatelského rozhraní UE má systém zvaný Unreal Motion Graphics (UMG), zatímco systém v Unity nemá žádný oficiální název, v praxi a v dokumentaci se jednodušše označuje jako Unity UI.

Jedním z hlavních rozdílů je zde přítomnost samostatného editoru v UE, který poskytuje jednoduché nástroje pro editaci uživatelského rozhraní. Editace uživatelského rozhraní v Unity probíhá v samotné scéně hry. Podobně jako vše v UE jsou jednotlivé grafické komponenty (nazývány widgets) samostatnými Blueprint třídy speciálně uzpůsobenými pro tvorbu uživatelského rozhraní. Díky tomu má každý widget svůj vlastní Event Graph, ve kterém je možné vykonávat herní logiku a přednastavit potřebné hodnoty, které bude widget využívat. Dynamicky modifikovatelné komponenty jako je text, velikost písma, barva pozadí, pozice posuvníku a další mohou být vázány (bind) k funkcím a proměnným, které dohlížejí na aktualizaci zobrazených hodnot [25].

V Unity je obecný přístup odlišný. Samozřejmě by bylo možné k uživatelskému rozhraní vytvořit skript, který se neustále dotazuje na stav svých zobrazených proměnných. Mnohem častější přístup je nastavování hodnot v uživatelském rozhraní ze tříd, které na něj mají vliv. Pokud se jedná o manipulaci uživatele s rozhraním pomocí vstupů jako je myš nebo klávesnice, je vhodné použít EventSystem objekt (ten se automaticky vytváří v hierarchii scény).

Jako příklad zde uvedu, jak byl vytvořen ukazatel zdraví hráče (health bar) v obou enginech. Vše ostatní funguje na stejných principech. Hráčův ukazatel zdraví je v obou enginech jednoduchý posuvník, jehož vyplnění reprezentuje poměr životů a maxima životů hráčovy postavy. V UE je hodnota vyplnění posuvníku vázána (bind) na funkci, která vrací poměr životů a maxima životů hráče, kde ukazatel na hráčovu postavu byl získán

na začátku hry v Event BeginPlay. Zatímco v Unity má naopak hráčova postava (PlayerCharacter skript) od začátku hry uloženou referenci na posuvník reprezentující jeho zdraví a po změně stavu hráčova zdraví (funkce IncreaseHealth, DecreaseHealth) je posuvník aktualizován.

Jak již bylo řečeno, UMG podporuje hierarchickou strukturu svých grafických komponent, což zlepšuje jejich znovupoužitelnost. Podobného efektu lze v Unity dosáhnout pomocí prefabrikátů. Jak v Unity, tak v UMG je hlavní grafickou komponentou canvas, který slouží pro organizaci velikostí a pozic jednotlivých elementů uživatelského rozhraní. U každého potomka canvas komponentu lze nastavit jeho ukotvení, posun, zarovnání a další parametry.

V UMG má každý grafický komponent pevně definováno kolik může mít potomků. Klasickým příkladem jsou grafické komponenty Image a Border, kde Image nemůže mít žádné potomky, zatímco Border může mít právě jednoho. Přes odlišnost jejich názvu a počtu potomků se ale jedná o komponenty funkčně totožné. Velkou výhodou UMG je jednoduchost tvorby listů a dalších organizačních šablon. Jako příklad bych uvedl tvorbu uživatelského rozhraní inventáře, který má podobu posuvného listu položek, kde každá položka reprezentuje předmět v inventáři. V Unity je pro tvorbu takového listu nutné použít tzv. ScrollRect komponent, který se stará o posouvání listu. ScrollRect je nutné přiřadit tzv. VerticalLayoutGroup, která uspořádá jednotlivé položky pod sebe do listu. Protože ScrollRect se stará o posun a VerticalLayoutGroup je celý list položek, je nutné schovat položky, které přetečou z oblasti inventáře pomocí komponentu Mask, který je umístěn na stejném objektu jako ScrollRect komponent. Každá položka listu musí být jeho potomkem v hierarchii a navíc musí mít komponent LayoutElement, aby byla registrována VerticalLayoutGroup. Protože si přejeme, aby byly položky uniformní svou velikostí/vzhledem, musí každá z nich mít nastavenou stejnou šířku a výšku v RectTransform komponentu (speciální transform komponent objektů uživatelského rozhraní) a navíc je nutné na rodičovský objekt s VerticalLayoutGroup přiřadit komponent ContentSizeFitter, jež se postará o dodržení požadavků na velikost položek listu.

V UE lze podobného efektu dosáhnout pomocí použití grafického komponentu ScrollBox, do kterého vložíme položky. Položky tzv. obalíme (wrap - dáme jim rodiče) komponentem SizeBox, který má stejnou funkci jako ContentSizeFitter. Tím jsme hotovi, neboť ScrollBox již nativně uspořádá položky do listu (jak vertikálního, tak horizontálního podle naší volby) a postará se o maskování přetečeného obsahu. V UMG je navíc možné použít mnoho dalších speciálních komponent, jež nemají v Unity alternativu. Mezi ty patří např. ScaleBox, který dokáže vynutit škálování komponentu bez použití canvas

objektu, Spacer, který vkládá mezery mezi komponenty a především WidgetSwitcher, jež podle indexu přepíná viditelnost skupin komponentů.

Obrázek 4.1: Uživatelské rozhraní inventáře v Unity (hráno na gamepadu).

Obrázek 4.2: Uživatelské rozhraní inventáře v UE. Zde byla přidána navíc funkcionalita pro zabarvení políčka inventáře podle rarity předmětu. V políčku zbraní lze navíc přepínat mezi vlastnosmi zbraně a jejím popisem.

Obecně lze říci, že UMG poskytuje přehlednější systém, který lze také snadněji udržovat při tvorbě větších projektů. Práce s prefabrikáty v rámci uživatelského rozhraní se může jevit jako těžkopádná, především pokud existuje mnoho odkazů mezi UI komponenty, které jsou nastaveny v Inspectoru, a jakákoliv neopatrná manipulace s takovým prefabrikátem, jako je jeho rozdělení na části, může odkazy rozbít.

4.2 Zpracování vstupů

Jedním z dalších velkých témat je zpracování vstupů ve hře. Každý engine nabízí své řešení tohoto problému. Zde je systém v Unity z mnoha důvodů poněkud problematický. V Unity je nastavení vstupů uloženo v Input Settings. Vstupy jsou uspořádány do pole, kde je každý identifikován svým jménem. K naslouchání vstupních událostí je možné použít statické funkce třídy Input: GetButton, GetButtonDown, GetButtonUp, GetKey, GetKeyDown, GetKeyUp. Funkce obsahující Button v názvu naslouchají tlačítkům definovaným v Input Settings, zatímco funkce obsahující Key v názvu slouží k naslouchání na konkrétních klávesách klávesnice. GetButton vrací true pro každý frame hry, kdy je tlačítko stisknuto. GetButtonDown vrací true, právě když bylo stisknuto a GetButtonUp vrací true, když uživatel stisk tlačítka ukončil. K naslouchání a zpracování vstupů tedy dochází v herní smyčce ve funkci Update.

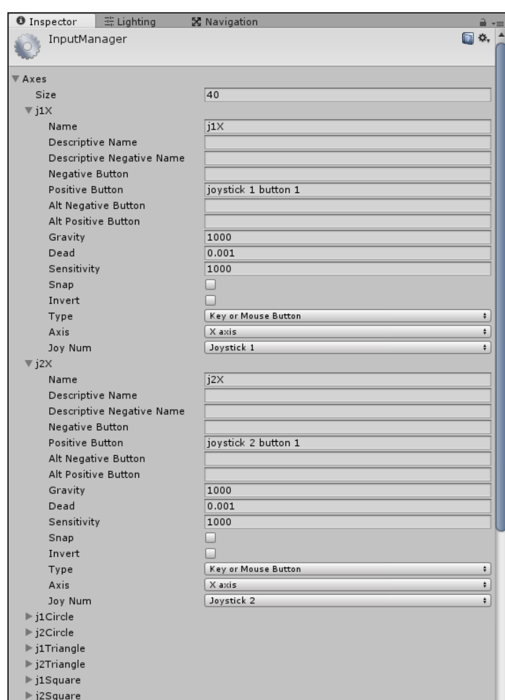
Problém nastává, pokud chceme do hry zakomponovat lokální multiplayer (hru pro více hráčů na jednom PC/konzoli) s více ovladači. Výsledný projekt sice lokální multiplayer nepodporuje, ale v počátcích vývoje byly vytvořeny prototypy, kde byl podporován. Problém je následující: V systému Unity nejsme schopni definovat univerzální ovládání, které by bylo sdíleno více ovladači. Jsme nuceni pro každý ovladač vytvořit duplikátní nastavení (viz obrázek 4.3). Navíc nám Unity neposkytuje informace o čísle hráče a jeho ovladači, které si musíme manuálně udržovat podle pořadí připojení. Tato hodnota je v UE poskytována v podobě čísla hráče v jeho Controller třídě.

Například pokud máme hru o čtyřech hráčích a každý má tlačítko pro střelbu, musíme v Input Settings vytvořit čtyři stejná tlačítka (až na odlišnou kolonku čísla ovladače), kterým dáme různá pojmenování (např. “c1Fire”, “c2Fire”, atd.). Ve hře je pak při naslouchání nutné konkatenovat string čísla ovladače (“c1”) se stringem tlačítka (“Fire”).

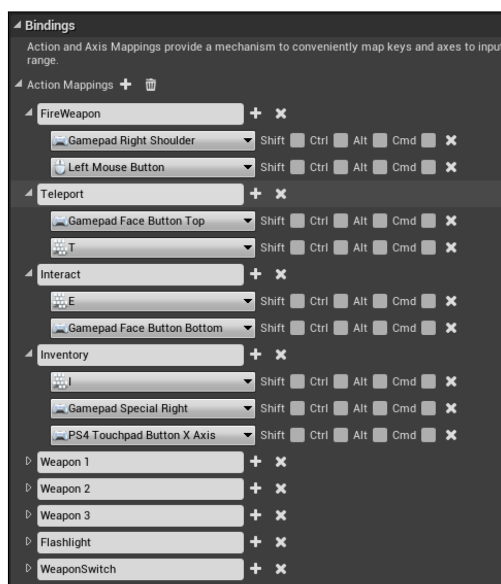
Pokud bychom chtěli modifikovat nějaké nastavení pro střelbu jako je např. sensitivita tlačítka, je nutné tuto hodnotu změnit pro každého hráče zvlášť. Takový systém může být poněkud náchylný na chyby a vyžaduje mnohem pracnější údržbu. Problém lze řešit využitím softwaru třetích stran (jako jsou populární systémy Rewired [26] nebo

InControl [27] na Asset Store). Další možnost představuje vytvoření vlastního systému, což je však časově náročné.

Unreal Engine takové problémy nemá. Místo konkrétního zpracování jednotlivých vstupů se zde vytváří tzv. Action Mappings a Axis Mappings, kde vytváříme “akce” a “osy”, kterým přiřazujeme jednotlivé vstupy [28]. Navíc UE nezajímá, od kterého ovladače vstup přišel, neboť to můžeme dynamicky odlišit podle čísla hráče ve hře (za podmínky, že používáme herní framework od Epic Games). Například vytvoříme Action Mapping “Fire”, přiřadíme mu levé tlačítko myši a pravé horní tlačítko ovladače. Ve hře je pak nasloucháno události InputAction Fire, kde si můžeme zjistit pořadové číslo hráče, který událost vyvolal, a podle něho se můžeme dále rozhodnout, jak vstup zpracovat (který hráč vystřelí).



(a) Unity - ukázka z jiného projektu, ve kterém byl podporován lokální multiplayer pro dva ovladače (prefix j1 a j2).



(b) UE Action Mappings. Můžeme si zde všimnout možnosti kombinace tlačítek společně s běžně používanými klávesami Shift, Ctrl, Alt a Cmd (Mac OSX).

Obrázek 4.3: Ukázka InputSettings v obou enginech. V Unity můžeme vidět duplicitní tlačítka pro podporu více ovladačů. V UE takový problém nevzniká, neboť definujeme obecná nastavení pro všechny hráče.

4.2.1 Připojení ovladače

Hra byla vyvinuta jak pro klávesnici a myš, tak i pro ovladač herní konzole Playstation 4 Dualshock 4, přičemž návrh UI byl spíše koncipován pro použití s ovladačem. Unity je schopna ovladač rozpoznat automaticky, ovšem jeho nastavení se liší podle platformy (hodnoty v InputSettings odpovídají odlišným tlačítkům v závislosti na operačním systému počítače). Pro použití ovladače s UE je nutné nainstalovat jemu příslušný driver. UE má oproti Unity univerzální zpracování vstupů ovladače. To znamená, že odlišuje tlačítka podle jejich polohy na ovladači (např. shoulder buttons, face buttons - top, bottom, left, right). UE navíc podporuje ovládání kamery v editoru pomocí připojeného ovladače. Pro připojení ovladače byl použit volně dostupný driver DS4Windows [29].

5. Vzhled a animace

Tvorba kvalitní grafické prezentace hry je jedním z klíčových prvků herního vývoje. Grafická tvorba her lze rozdělit do mnoha oborů jako je tvorba konceptuálních ilustrací a obecně vymyšlení uměleckého stylu hry, tvorba modelů, textur, animací a speciálních efektů, režie a střih cinematických videí a další. V této kapitole nás bude především zajímat tvorba materiálů, nastavení animací, speciální efekty v podobě částicových systémů a post processing. Pro tvorbu herního prostředí jako je půda a její flóra viz kapitolu “Terrain a Landscape”.

5.1 Materiály a shadery

Materiály a shadery jsou jedny z nejdůležitějších grafických komponent každé hry. Materiály jsou obecně objekty, které popisují jak vypadá povrch modelu, na který jsou aplikovány. Materiály definují, jestli je povrch modelu hrubý, hladký, lesklý, průhledný, kovový, jaké barvy nebo textury používá, a mnoho dalších parametrů. Shader je program, který při kontaktu světla s modelem vypočítá, jak bude světlo interagovat s jeho povrchem. Obecně každý materiál používá shader program pro tento výpočet.

Na úvod bych připomenul rozdílné názvosloví v Unity a UE. V Unity odlišujeme shadery a materiály, kde materiály používají shadery a definují použité parametry jako jsou textury, barvy a další hodnoty. V UE odlišujeme materiály a jejich instance. Materiál je v UE shader program, jenž nese navíc některé informace důležité pro engine. Není to tedy doslova čistý shader. Pro naše účely lze ovšem říci, že je materiál v UE ekvivalentní shaderu v Unity. Jak Unity shadery, tak materiály v UE jsou převážně napsány/generovány v jazyce HLSL (High Level Shading Language) [30] [31]. Instance materiálu v UE je pak obdobou materiálu v Unity.

Jedním z kritických rozdílů, se kterým se při práci v obou enginech vývojář může sektat, je nepřítomnost shader editoru v Unity. Unity zakládá svůj systém materiálů na

takzvaném Standard shaderu. Standard shader je PBR shader, který poskytuje rozsáhlou funkcionalitu pro tvorbu realistických materiálů. Standard shader má čtyři vykreslovací módy: opaque, transparent, cutout a fade. Opaque mód je výchozím nastavením a je použit pro neprůhledné objekty. Transparent mód je vhodný pro průhledné předměty jako jsou skla nebo plasty. Průhlednost je zde určena alfa kanálem Albedo textury. Transparent mód zachovává odrazy. Cutout mód, jak název napovídá, je použitý v případě, kdy část materiálu je neprůhledná a druhá část je 100% průhledná (nic mezi). Cutout mód je vhodný pro tvorbu flóry jako jsou listy stromů nebo tráva. Fade mód je podobný Transparent módu, zde ovšem nejsou zachovány žádné odrazy (je např. vhodný pro postupné mizení objektů) [32].

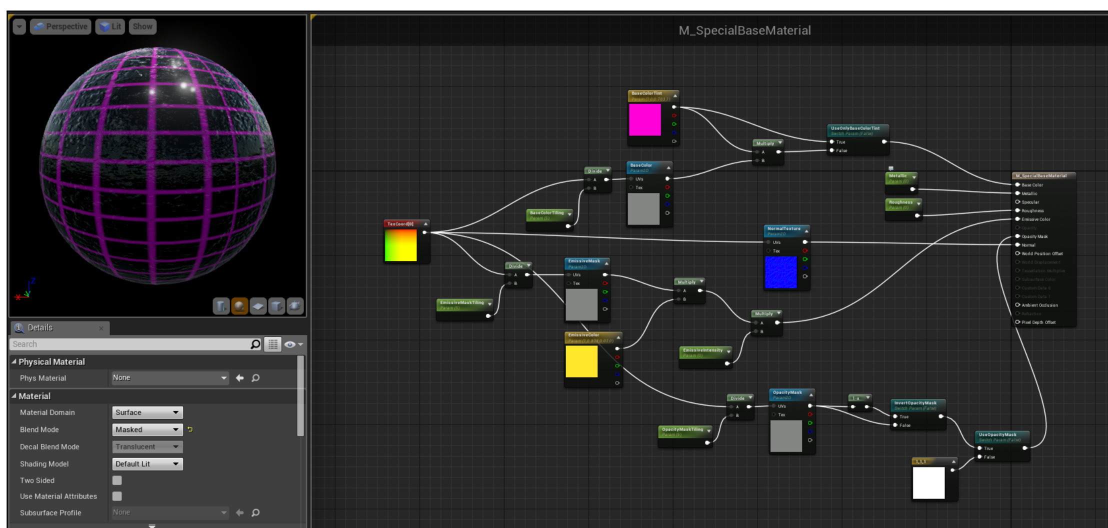
Standard shader obsahuje funkcionalitu pro použití Albedo, Metallic, Smoothness textur, Normal, Height, Occlusion, Emission map a dalších jako je detailní maska pro situace, kdy je kamera blízko povrchu materiálu. Společně se Standard shaderem jsou k dispozici ještě shadery pro particle systémy (aditivní, multiplikativní a jejich variace), unlit shadery (nejsou ovlivněny světlem), FX (glass, water), Toon, Projector, Skybox, Sprites a další. Tato nabídka pokryje téměř všechny základní materiály, ovšem pokud bychom si přáli vytvořit nějaký více specifický shader jako je noční vidění, X-ray, nebo např. shader měnící své parametry podle času, je nutné ho napsat, nebo se opět obrátit na software vytvořený třetí stranou. V mém případě jsem se ze studijních důvodů (hra v Unity byla vyvíjena před hrou v UE) rozhodl investovat peníze do ShaderForge, které poskytuje vizuální rozhraní pro tvorbu a editaci shaderů. ShaderForge má však omezení, že pokud v něm shader nebyl vytvořen, není ho v něm možné otevřít/editovat.

V UE je situace výrazně odlišná. UE nabízí sérii materiálů (pozor - myšleno shaderů, viz kapitolu “Pojmy a názvosloví”) ve Starter Content. Zde se ovšem nejedná o univerzální materiály, ale pouze o ukázky, jak takové materiály tvořit. Nabídka navíc není vhodná pro použití ve finálním produktu, neboť je to sbírka materiálů, jejichž funkcionalita je obdobná (tedy shadery, které jsou podobné, jen mají odlišné parametry). Jejich použití je nevhodné z důvodů optimalizace hry, kde vykreslení pěti odlišných materiálů bude náročnější než vykreslení jednoho materiálu v podobě pěti jeho instancí (tedy je použit jeden zkompilovaný shader, který materiálové instance sdílejí, instance se pouze liší svými parametry).

UE ovšem poskytuje velmi robustní Material editor, ve kterém je možné tvořit širokou škálu materiálů. Velkou výhodou tohoto editoru je jeho provázanost s ostatními systémy UE. Ve výběru uzlů jsou dostupné uzly poskytující funkce pro manipulaci různých vlastností částicových systémů, Landscapes, Foliage, Grass, Decals a dalších odvětví enginu.

Jednou z velmi užitečných funkcí Material editoru v UE je možnost ukázky (preview) jednotlivých mezivýsledků. Samotné instance materiálů lze pak náležitě editovat podle parametrů rodičovského materiálu ve vlastním editoru. Pro popsání celého systému materiálů by byla potřeba samostatná bakalářská práce.

Na druhou stranu UE nepodporuje vkládání vlastních shader programů do projektu, aniž by byl modifikován zdrojový kód enginu. Material editor však umožňuje vkládat úryvky HLSL kódu v podobě Custom Material Expression uzlu [33].



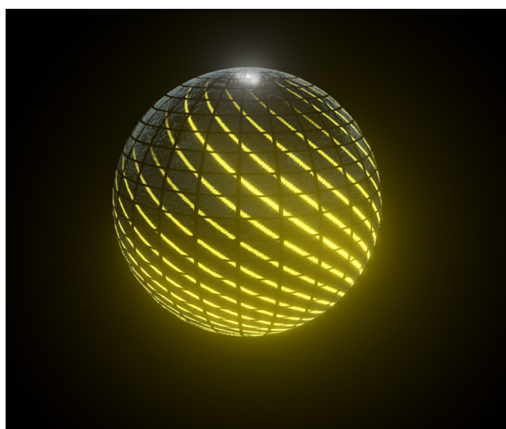
Obrázek 5.1: Material Editor v UE a jeden z vytvořených materiálů, jež byl použit na různé zbraně.

5.1.1 Editace materiálů za běhu hry a statické přepínače

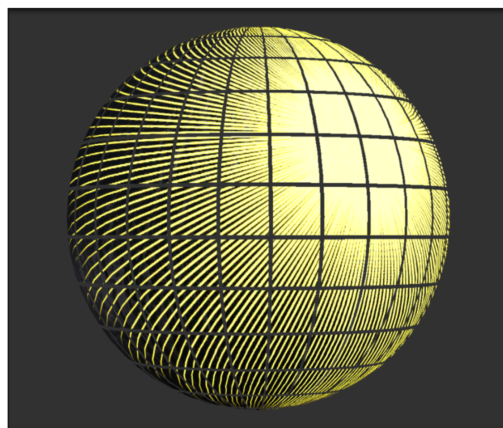
Je důležité poznamenat, že přístup k editaci materiálů během hry v obou enginech je také odlišný. V Unity je možné nastavovat parametry materiálu přímo ve skriptu pomocí funkcí SetFloat, SetColor, SetTexture, SetInt atd.. Při práci se Standard shaderem Unity je nutné dávat pozor na jeho odlišné módy, jež lze přepínat nastavením speciálních parametrů. Standard shader je optimalizován tak, že všechny jeho nepoužité parametry a módy jsou ignorovány. Pokud plánujeme přepínat mezi módy Standard shaderu, je nutné jeho jednotlivé varianty zahrnout v buildu hry, tzn. že materiál používající Standard shader v požadovaných nastaveních, na která chceme během hry přepnout, musí být buď použit v nějaké scéně, nebo přítomen ve složce Resources [34].

V UE odlišujeme dva druhy instancí materiálu: Material Instance Constant a Material Instance Dynamic (MID). Material Instance Constant je spočítán před spuštěním hry a je při běhu hry neměnný. MID je počítán v průběhu hry, tzn. že můžeme během běhu hry libovolně editovat jeho parametry. MID je nutné na začátku hry vytvořit ve skriptu pomocí uzlu Create Dynamic Material Instance z materiálu nebo konstatní instance materiálu. Po vytvoření je též nutné MID přidělit modelu pomocí uzlu Set Material. MID bývá ve většině případech vytvářen v Construction Script, především pokud si přejeme měnit jeho parametry v editoru úrovně [35].

Nakonec bych uvedl, že v UE Material editoru lze též používat tzv. statické přepínače, jež nám umožňují vybírat jeden ze vstupů při tvorbě materiálu. To je vhodné použít při tvorbě dvou velmi podobných materiálů, jež se liší jednou vlastností. Statické přepínače lze editovat pouze ve statických instancích materiálu (Material Instance Constant) a jsou použity během kompilace. To znamená, že vytváříme v podstatě dva materiály, jež se liší pouze jednou hodnotou (místo toho, abychom celý materiál kopírovali). Zde je důležité čtenáře varovat, že musí být zkompilovány všechny permutace materiálu podle jeho přepínačů, a proto není vhodné jich používat pro jeden materiál mnoho [36]. Statické přepínače byly použity při tvorbě speciálního materiálů, jež podporuje masku průhlednosti a její invertování (operace 1-x pro invertaci barvy) (viz 5.1).



(a) Instance materiálu z obrázku 5.1, která používá invertovanou masku průhlednosti a též masku pro zářivé paprsky (Emissive Color).



(b) Podobný materiál v Unity, jež je založen na bázi Standard Shaderu v módu Cutout. Zde musela být vytvořena textura invertované mřížky.

Obrázek 5.2: Ukázka materiálu stejného charakteru v obou enginech. Zde je nutné čtenáře upozornit, že okno s ukázkou materiálu v Unity nepoužívá některé post process metody jako je bloom a další.

5.2 Částicové systémy (Shuriken a Cascade)

Částicové systémy (particle systems) slouží pro reprezentaci objektů, jejichž tvar (členitost, neostrá hranice) nebo chování (náhodný pohyb, rotace, odrazy) nelze jednoduše vyjádřit statickými či animovanými modely. Ve většině případů jsou to velmi chaotické systémy velkého množství textur, modelů nebo jiných grafických objektů, které vytvářejí jeden celek. Takový celek většinou reprezentuje přírodní fenomena jako jsou oheň, kouř, pára, kapky vody, jiskry, sníh, déšť, prach v atmosféře a další. Jsou také vhodné pro tvorbu různých grafických efektů jako jsou kouzla, aura, stříkající krev, dráha za pohybujícím se objektem a další [37].

Jednou z hlavních částí částicového systému je jeho zdroj (emitter). Zdroj může být model (tedy nějaké 3D těleso - buď jeho objem nebo povrch), nebo to může být pouze bod v prostoru (všechny částice vytvořeny ve stejné počáteční pozici).

Zde můžeme narazit na první rozdíl mezi Unity a UE a jejich editory pro částicové systémy, Shuriken a Cascade. Shuriken (Unity) nabízí zvolit za zdroj libovolný 3D model. Toto nastavení můžeme najít v Shape modulu částicového systému. Částice mohou být vytvářeny buď v uzlech (vertices), na hranách (edges), nebo na povrchu trojúhelníků (triangles) modelu [38]. Toto nastavení v Cascade (UE) chybí. Jsou zde, podobně jako v Unity, na výběr základní tvary jako válec, koule a trojúhelník. Cascade místo použití libovolného 3D modelu poskytuje možnost mít jako zdroj jednotlivé kosti/sockets skeletálního modelu za použití modulu Skel Vert/Surf Location. Je tedy možné importovat statický model jako model skeletální. Tento přístup je ovšem poněkud komplikovaný pro běžné a opakované použití.

Částice mohou být v prostoru reprezentovány jako body, textury (také animace vytvořená z částí textury - tzv. Texture Sheet Animation modul v Unity) nebo dokonce jako 3D geometrie. Zde je ovšem důležité čtenáře upozornit, že náročnost statických 3D modelů pro použití v částicových systémech může být v závislosti na situaci velmi vysoká. Co se týče reprezentace částic můžeme narazit na nedostatek v Shuriken systému, kde částice v podobě 3D geometrie nejsou podporovány.

Standardní moduly jako je rychlost, barva, velikost, pozice a další jsou dostupné v obou enginech. Pokud se vlastnost mění v závislosti na čase nebo jiném parametru, jako je například rychlost, je možné v obou enginech také editovat samotnou křivku reprezentující tyto závislosti.

Hlavním rozdílem mezi systémy v Unity a UE, Shuriken, respektive Cascade je, že Cascade má opět svůj vlastní editor, zatímco Shuriken je poskytován jako komponent, který lze umístit na herní objekty ve scéně. Pro tvorbu hierarchie částicového systému jako je například oheň (což obnáší textury ohně, kouř, jiskry, světelný zdroj a uhlíky) je v Unity nutné vytvořit hierarchii prázdných objektů, kde je každému přiřazen příslušný komponent částicového systému. V Cascade můžeme editovat více zdrojů a jejich částic zároveň. Zde je důležité především kvalitní uživatelské rozhraní a okno s grafickou ukázkou částicového systému v izolovaném prostředí (v Unity lze pouze pozorovat částicový systém ve scéně). Cascade navíc poskytuje možnost pracovat právě na jednom ze zdrojů (izolovat ho - single out), aniž bychom museli manuálně deaktivovat všechny ostatní jako v Unity.

5.2.1 Typy zdrojů

Cascade rozlišuje více typů zdrojů (nazýváno TypeData). Jedná se o typy AnimTrail, Beam, GPU, Mesh, Ribbon a Standard Sprite (výchozí hodnota). Jsou reprezentovány speciálními moduly, přičemž každý zdroj může mít právě jeden z nich (pokud je políčko tohoto modulu prázdné, je zvolena výchozí hodnota Standard Sprite). Nyní budou rozebrány jednotlivé typy zdrojů a nakonec budou uvedeny jejich alternativy v Unity, pokud existují.

Typ zdroje AnimTrail je použit společně se skeletálním modelem a jeho animacemi pro tvorbu efektů doprovázejících pohyb postav a jiných objektů. Tento typ zdroje nebyl v rámci projektu použit.

Typ zdroje Beam Data byl použit pro tvorbu efektů střelby zbraní (dráha kulky). Je vhodný pro částice, které se během svého života pohybují po úsečce/křivce. Odlišujeme dva typy tohoto zdroje - Distance a Target (podle parametru Beam Method). Distance vyžaduje počáteční bod a vzdálenost, kterou mají částice urazit, zatímco Target vyžaduje počáteční i koncový bod. Jak již bylo řečeno, částice mohou putovat i po křivce, kterou lze definovat tečnými vektory. Pro naše účely byl použit typ Target. Pro nastavení počátečního a koncového bodu je nutné přidat speciální moduly Source a Target. Dále je zde velmi důležitá hodnota rychlosti částic (Speed), která, pokud je nastavena na hodnotu 0, z částic vytvoří instantní paprsek po celé dráze křivky. Vzhledem k tomu, že je ve hře použito vysílání paprsků (line tracing) pro určení zásahů zbraní, je volba takového částicového systému pro reprezentaci střelby ideální. Podobného efektu je v Unity pomocí částicového systému relativně obtížné docílit. Unity proto poskytuje sa-

mostatný komponent zvaný Line Renderer, který též slouží pro vykreslení paprsku po úsečce (zde nelze vytvářet křivky).

V obou případech je při výstřelu ve hře zobrazen částicový systém, respektive Line Renderer, jehož konečná pozice je nastavena podle zasaženého bodu, případně možnému dosažitelnému bodu střely, pokud nebylo nic zasaženo. Pro částicový systém v Cascade je také nutné nastavit počet jeho opakování (Loop) na 1 a počet vytvořených částic (Spawn) na 0 a použít tzv. Burst Mode. Burst Mode vytvoří námi definovaný počet částic, v případě dráhy střel nám stačí jedna. V Unity, neboť pracujeme s LineRenderer komponentem, je přístup odlišný. Zde je nutné při výstřelu zobrazit Line Renderer a počkat dobu výstřelu (např. 0.1 sekundy) a následovně Line Renderer schovat.

Typ zdroje zvaný GPU Sprite Emitter (nadále zkráceně GPU částice) umožňuje vykreslení výrazně většího počtu částic díky svému výkonu. Všechny výpočty pro tento zdroj jsou (jak název napovídá) provedeny na grafické kartě. Samozřejmě to nese svá omezení. GPU částice nemohou mít v Cascade modul, který na jednotlivých částicích vytváří světelný zdroj (Light). Pro GPU částice je také nutné nastavit tzv. Bounding Box. Jedná se o oblast, která určuje, zda se mají částice vykreslovat. Částice budou vykresleny v případě, že aktivní kamera ve hře vidí aspoň část této oblasti. GPU částice dokonce podporují kolize (modul Collision (Scene Depth)), které jsou zde vypočítány podle Z-Bufferu (v UE dokumentaci nazvaný jako Z-Depth Buffer). Pro tyto kolize je možné nastavit odrazivost (bounciness) a tření (friction). Jeho výhodou a možnou limitací, pokud chceme dosáhnout nějakého specifického cíle je, že nemusíme vytvářet kolizní geometrii pro odrazy těchto částic, neboť jsou vypočítány podle hloubky viditelných objektů od kamery. To znamená, že jakákoliv viditelná geometrie bude kolidovat s částicemi [39]. Podle dokumentace a výukových materiálů od Epic Games může počet částic pro GPU particle systémy dosáhnout až statisícových hodnot [40].

Unity výpočet částicových systémů na grafické kartě bohužel v současných verzích nepodporuje. Existují opět řešení na Asset Store, zde ovšem není zaručená podpora v nových verzích Unity a obecná funkčnost systému. Např. TC Particles za 85 Euro [41].

Další z typů zdrojů je tzv. Mesh Emitter. Zde se někdy plete terminologie v Unity a UE. V Unity je často označován libovolný 3D zdroj (Shape → Mesh) jako Mesh Emitter, zatímco v UE se jedná o zdroj, jehož částice mají podobu 3D modelu. Tato funkcionalita v Unity chybí, ovšem její použití není tak široké jako v případě GPU částic.

Poslední typ zdroje/částic jsou tzv. Ribbon částice, které slouží k vytvoření dráhy za částicemi nebo objekty. Pro určení za jakým objektem mají být částice vytvořeny slouží modul Trail → Source, ve kterém nastavíme, zda mají následovat částice ze stejného částicového systému ale jiného zdroje (podle jeho jména) nebo herní objekt. Ribbon částice byly použity pro tvorbu teleportačního efektu za hráčovou postavou.

Podobnou funkcionalitu, kterou poskytují zdroje typu AnimTrail a Ribbon částice můžeme v Unity implementovat pomocí tzv. Trail Renderer komponentu, který slouží pro tvorbu částic následujících různé objekty. Pokud chceme ovšem použít Trail Renderer pro vytvoření dráhy za jednotlivými částicemi námi vybraného částicového systému, je nutné manuálně instanciovat a přiřadit každé vytvořené částici její vlastní Trail Renderer. Trail Renderer lze také využít pro animované postavy a další objekty ve hře.

Trail Renderer nabízí mnoho vlastních nastavení. Podobně jako u každého částicového systému mu lze nastavit materiál, který by měl používat částicový shader. Trail Renderer může vrhat stíny a také na něj mohou být vrhány. Má svou životnost, šířku a barvu během života. Důležitým parametrem je tzv. minimum vertex separation, který určuje jak často může být vytvořena nová sekce dráhy, kde nižší hodnota znamená více sekcí, tím i vyšší členitost a hladší průběh dráhy [42].

Navíc je opět dobré připomenout, že systém Cascade je v UE provázaný s Material editorem, kde lze vytvořit speciální parametry a uzly jako je ParticleColor. Ty lze použít pro rychlou tvorbu vlastních materiálů pro částicové systémy.

Obecně lze říci, že systém Cascade je výkonější, neboť zahrnuje více možných variant tvorby částicových systémů a poskytuje na to vlastní přehledný editor. Největším nedostatkem Unity jsou chybějící GPU částicové systémy, které mají své uplatnění při tvorbě moderních efektů s vysokým počtem částic a jsou integrálním prvkem současných her. Jediným nedostatkem UE systému Cascade je chybějící možnost vytvářet částice na povrchu nebo v objemu libovolného modelu.

Pro ukázky částicových systémů ve hře viz obrázky A.8 (mlha a déšť v Unity), A.13 (střelba zbraně a střílna v UE) a A.15 (portál v UE).

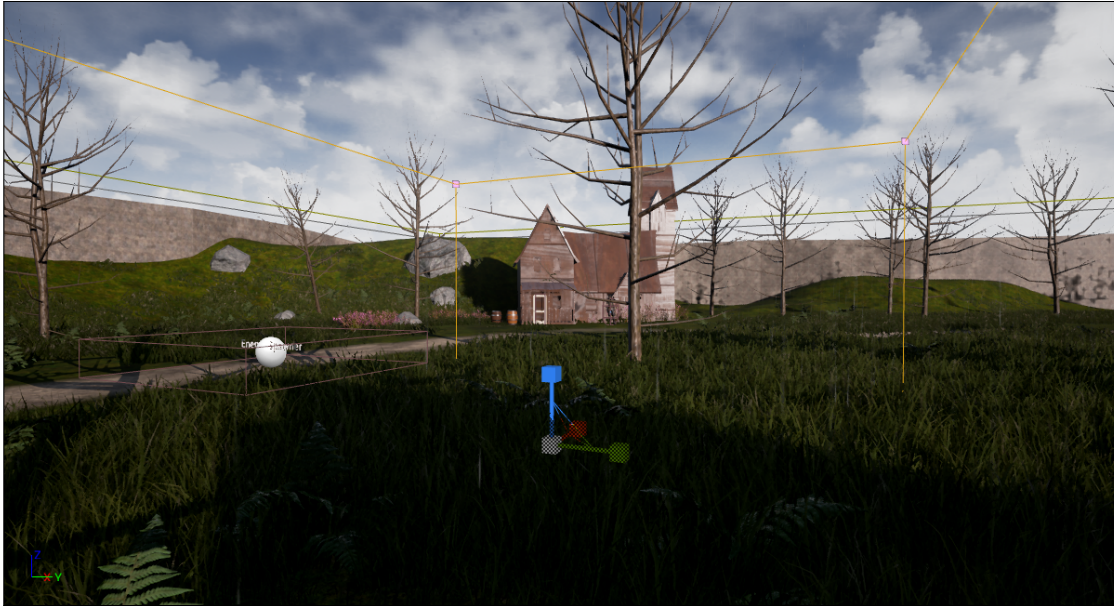
5.3 Post processing

Vzhled hry je mimo jiné závislý na post processing. Post processing nám umožňuje provést korekturu barev, aplikovat efekty jako bloom, vignette and chromatic abberation, multisampling (antialiasing), ambient occlusion, depth of field, motion blur a další.

V Unity jsou tyto efekty nativně poskytovány ve Standard Assets balíčku (Effects/ImageEffects) jako komponenty, které lze umístit na kameru. Důležité je podotknout, že efekty jsou aplikovány v pořadí, ve kterém jsou na kameře umístěny.

Unreal Engine poskytuje také celou řadu efektů. Zde je ovšem důležitý rozdíl v tom, že je automaticky aplikuje i v editoru (tedy ne pouze ve hře jako Unity). Základní nastavení post process efektů můžeme najít v Project Settings pod Rendering (především v tabulce Default Postprocessing Settings). V editoru navíc můžeme vypnout a zapnout jednotlivé efekty přes menu Show → Post processing. Pro samotné efekty v úrovni je použito tzv. Post Process Volume, ve kterém můžeme aktivovat a deaktivovat jednotlivé efekty a nastavit jejich hodnoty. Všechny efekty budou aplikovány pouze v případě, že je kamera uvnitř oblasti kvádru Post Process Volume. Pokud bychom chtěli tyto efekty aplikovat pro celou úroveň ve hře, stačí zaškrtnout políčko Unbound, které zruší omezení kvádru (jinými slovy kamera bude ovlivněna tímto Post Process Volume kdekoliv). Post Process Volume funguje i v editoru, takže je možné jeho rozsah editovat a ihned testovat. Pokud bychom chtěli určit oblasti s různými efekty v Unity, museli bychom vytvořit vlastní systém, který podle pozice hráče (například podle kolizního tělesa) vypne, respektive zapne jednotlivé efekty na kameře, nebo přepne kameru na jinou.

Post Process Volume také nabízí možnost pro aplikaci post process shaderu/materiálu na jednotlivé statické modely ve scéně pomocí atributu Custom Depth. Nejdříve je ovšem nutné vytvořit příslušný materiál jehož parametr Material Domain je nastaven na hodnotu Post Process. Tento materiál nebo jeho libovolnou instanci přiřadíme do položky Blendables v Post Process Volume. Pro použití tohoto materiálu stačí aktivovat parametr Render CustomDepth Pass na statickém nebo skeletálním modelu [43]. V Unity bylo označení objektů vytvořeno pomocí přepínání mezi dvěma shadery.



(a) Vně Post Process Volume.

(b) Uvnitř Post Process Volume.

Obrázek 5.3: Na obrázcích můžeme vidět jak Post Process Volume v UE ovlivňuje vzhled hry. Zde bylo aplikováno mnoho efektů jako úprava barev a Depth of Field, které simulují vizi zraněné postavy. Můžeme si také všimnout, že snímek byl zachycen uvnitř editoru.

5.4 Export a import FBX souborů

Důležitou částí tvorby hry je import a management souborů 3D modelů. Nejčastějším datovým formátem, se kterým se při práci s oběma enginy setkáme je FBX (Filmbox). Pro hru byly vytvořeny modely ve 3D modelovací aplikaci Blender. Kromě vlastních modelů byly použity modely postav od společnosti Mixamo a modely poskytované s enginy (Standard Assets v Unity, Starter Content, Content Examples a Kite Demo v UE).

Při práci s FBX soubory v obou enginech se můžeme setkat s určitými problémy. V Unity je hlavní problém rotace a škála importovaných modelů, jež byly exportovány z aplikace Blender verze 2.78a do FBX. Důležité je zde zmínit, že Unity a UE používají jinou základní jednotku délky. V Unity je základní jednotkou metr, zatímco v UE je to centimetr.

Při importování modelů do Unity se nám může stát, že jsou sice zobrazeny správně, ovšem jejich škála je nastavena na hodnotu 100 a jejich rotace podle osy x je -90°. Pro importaci objektu bez aplikované rotace je nutné model v Blenderu nejdříve rotovat podle osy x o -90°, aplikovat rotaci (ctrl + a → Apply Rotation), model rotovat zpět o 90° a novou rotaci neaplikovat. Co se týče škály, je nutné při exportu do FBX zrušit výchozí nastavení v Blenderu zvané “Scale all data according to current Blender size, to match default FBX unit”. Pokud tak neučiníme, importovaný model má sice správnou velikost, ovšem jeho škála v Transform komponentu je nastavena na hodnotu 100 pro každou osu, což může způsobit velké problémy pro fyzikální výpočty.

V kombinaci UE a Blenderu tyto problémy nenastaly. Nastal zde ovšem problém při exportu/importu skeletálního modelu. Zde jsou špatně importovány velikosti kostí modelu, které jsou 100 násobně menší než na modelu originálním. Díky tomu není vytvořen takzvaný Physics Asset, jenž popisuje fyzikální vlastnosti skeletálního modelu. Physics Asset poskytuje vlastní robustní editor pro nastavení fyzikálního chování skeletálního modelu a jeho částí. Physics Asset nám umožňuje definovat jednotlivá kolizní tělesa popisující skeletální model, nastavit pro ně různá pohybová omezení (například zamčení rotace podle os) a simulovat chování modelu v editačním okně pro kontrolu správnosti jeho nastavení. Řešení problému špatně importovaných kostí je podobné jako řešení problému s rotací v Unity. V Blenderu musíme nejdříve zvětšit rig modelu na 100 násobek požadované velikosti, aplikovat škálu (ctrl + a → Apply Scale) a zmenšit ho zpět na originální velikost.

5.5 Animace (Mecanim a Persona)

Animace je nedílnou součástí každé hry. Protože je skeletální animace velmi časově náročná, především pro dosažení přesvědčivých výsledků, byly použity animace a modely postav od firmy Adobe ze stránek Mixamo.com [61]. Animace je ovšem nutné zakomponovat do hry, na což nám poskytují oba enginy svá rozhraní. Rozhraní v Unity se nazývá Mecanim, v UE se jedná o systém zvaný Persona. Obě rozhraní nabízí širokou škálu možností tvorby a úpravy animací.

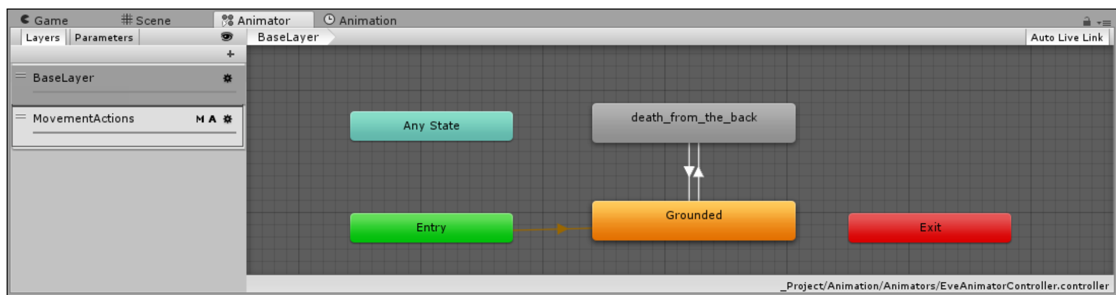
Animace ve hře jsou implementovány pomocí takzvaného Animator Controller v Unity a Animation Blueprint v UE. V obou případech se jedná o stavový automat, kde každý stav reprezentuje určitou animaci (nebo mix více animací) a jednotlivé přechody míchají animace pro dojem nepřerušeného pohybu. Základní rozdíl ve filozofii obou rozhraní je v tom, že Animator Controller nabízí svému okolí proměnné, které jej ovládají (například boolean hodnota, která vyvolá přechod mezi stavy). Tyto proměnné lze poté nastavovat externě ve vlastních skriptech, například pomocí funkce Animator.SetBool pro boolean hodnoty. Naopak v UE se Animation Blueprint může starat o svůj stav a stav svých proměnných samostatně pomocí dotazování (polling) ve funkci Blueprint Animation Update. Proměnné lze samozřejmě nastavit též externě.

5.5.1 Animační masky, vrstvy a míchání

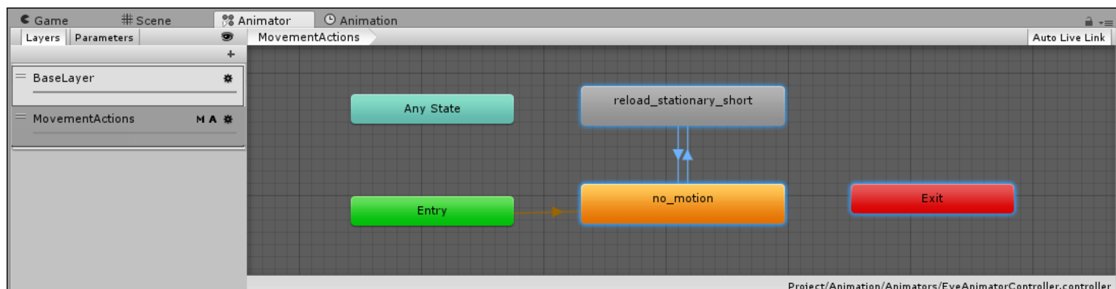
V obou enginech je možné animace míchat. Míchání animací je velmi důležité pro hladký a plynulý pohyb postav. Můžeme je vidět při přechodu z jedné animace do druhé (např. Animation.CrossFade v Unity, nebo častěji již zmiňovaný přechod ve stavovém automatu), při použití více animací současně (pohyb postavy v různých směrech) nebo při použití dvou animací pro jiné části skeletálního modelu.

Všechny jmenované způsoby byly při vývoji použity. První z nich můžeme vidět u přechodů mezi stavy v animačním stavovém automatu. Tyto přechody jsou v obou enginech podobné. Míchání více animací pro pohyb postavy lze realizovat v obou enginech pomocí tzv. Blend Trees v Unity a Blend Spaces v UE, kde podle dvou proměnných (pokud se bavíme o 2D Blend Tree/Space) mícháme skupinu animací jako je běh dopředu, do stran a dozadu, čímž dosáhneme plynulého pohybu postavy. Zde také nejsou žádné podstatné rozdíly ve funkcionalitě.

Třetí je míchání dvou animací podle jednotlivých kostí modelu. Zde se můžeme setkat s trochu odlišným přístupem k věci. V Unity máme tzv. vrstvy animace (Layers), jež nám povolují míchat více stavových automatů animací. Míchání bylo v projektu použito pro animaci přebití zbraně, která funguje společně s pohybem postavy (Blend Tree pohybu). V Unity jsou navíc používány tzv. masky (Avatar Mask) [44], které určují, jaké kosti budou v animační vrstvě ovlivněny. Unity poskytuje jednoduché rozhraní pro editaci humanoidních masek (tedy pokud je skeleton typu humanoid rig), která ulehčuje pochopení pro začínající vývojáře a umožňuje rychlou a jednoduchou editaci. V našem případě však nebyl použit humanoid rig, neboť došlo ke komplikacím při importování některých animací. Z tohoto důvodu nebylo možné použít masku pro lidské postavy (Humanoid Mask), ale specifikovat jednotlivé kosti, které budou animací ovlivněny. Samotná animační vrstva pak používá tuto masku. Animační vrstva může mít dvě nastavení míchání (Blending Type) - Additive a Override. Nastavení Override ignoruje animace ve vyšších vrstvách (tím je tedy přepisuje). Nastavení Additive animace ve vrstvě přičítá k animacím z vyšších vrstev. Pro animaci přebíjení byla použita vrstva MovementActions, jež používá Additive míchání.



(a) Hlavní vrstva stavového automatu postavy.

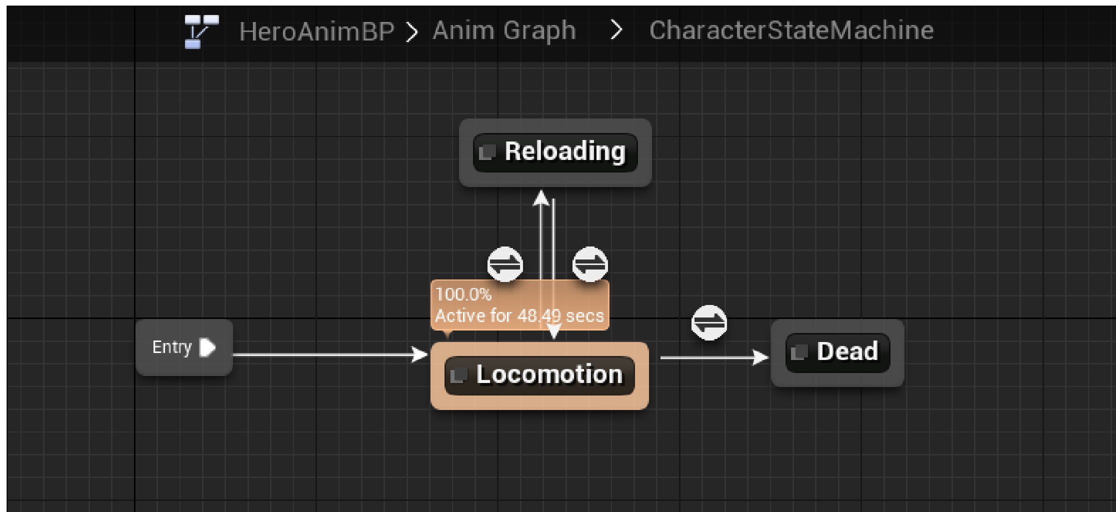


(b) Aditivní vrstva MovementActions používající masku (A a M vedle názvu vrstvy).

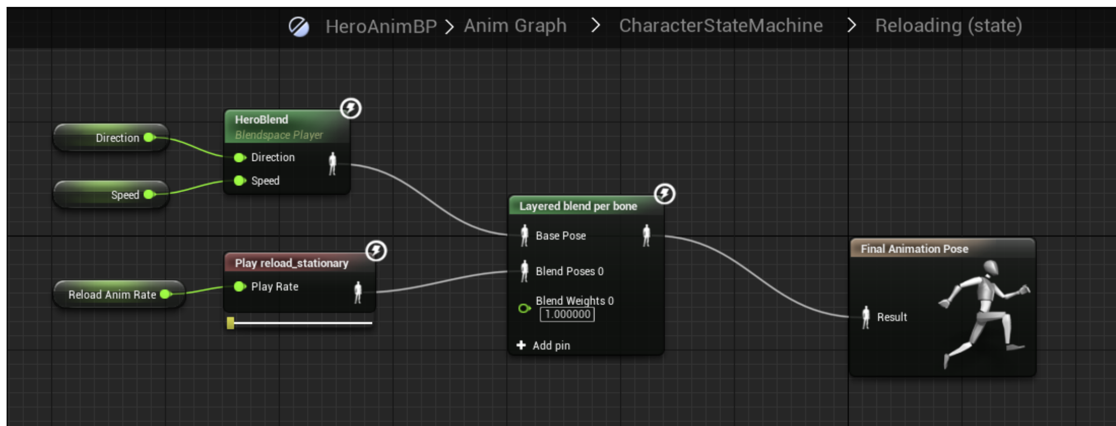
Obrázek 5.4: Animator Controller postavy v Unity.

Unity navíc poskytuje možnost synchronizovat jednotlivé vrstvy. Stavové automaty jsou v synchronizovaných vrstvách stejné a editací jednoho editujeme i ten druhý. Tuto funkcionalitu lze použít například pro různé fyzické stavy postavy, jejíž chování je stejné (struktura stavového automatu včetně jeho přechodů je stejná - postava například běhá, skáče, může umřít, střílí a leze). Je-li například postava zraněná, je stále schopná dělat vše, co postava zdravá, jenom v každém stavu bude používat odlišné animace. Synchronizace nebyla v projektu potřebná, ale jedná se o velmi šikovnou pomůcku, kterou je vhodné zmínit, a jejíž zastoupení v UE nenajdeme [45].

V UE je přístup odlišný. Animační vrstvy sice existují, ale mají jiný význam. Zde lze vrstvit jednotlivé animační montáže, které jsou použity jako výsledné animace ve stavovém automatu. Samotný automat vrstvit nelze. Pro animaci přebíjení je použit uzel Layered Blend per Bone [46]. Ten nám umožňuje míchat dvě a více animací opět s použitím animační masky. V UE ovšem není možné vytvořit znovupoužitelnou masku (a podpora zjednodušenné masky pro humanoidní postavy také není). U každého míchacího uzlu je nutné specifikovat, které kosti (rekurzivně do hloubky) animace ovlivní. Například v případě animace přebíjení se jedná o kosti LeftShoulder a RightShoulder a všechny jejich potomky, neboť chceme, aby animace přebíjení ovlivnila pouze ruce postavy. V UE byly také využity tzv. animační notifikace, které jsou volány při změně stavu ve stavovém automatu (změna stavu animace nebo přechod mezi animacemi).



(a) Animační Blueprint postavy a jeho stavový automat.



(b) Použití Layered Blend per Bone uzlu uvnitř stavu Reloading. Zde je míchána animace přebíjení (reload_stationary) a Blend Space postavy (HeroBlend).

Obrázek 5.5: Animační Blueprint a míchání animací v UE pomocí Layered Blend per Bone uzlu.

6. Umělá inteligence

Umělá inteligence (AI) je rozsáhlý obor, jež popisuje chování jednotlivých agentů v prostředí. Zde se budeme především bavit o umělé inteligenci nepřátelských postav, jež se pohybují po herním světě a útočí na hráčovu postavu. Unity poskytuje vlastní komponent pro tvorbu agentů zvaný NavMeshAgent, který slouží pro jejich pohyb po scéně. UE v tomto ohledu nabízí kromě navigačního systému navíc vlastní rozhraní pro tvorbu behaviorálních stromů postav. Nyní bych rád krátce představil oba systémy.

6.1 Rozhodovací stromy v Unity

V Unity je systém vytvořen v podobě rozhodovacích stromů. Tento systém je implementován v AI třídach nepřátel. Jednotlivé akce, jež nepřátelská postava může vykonat, jsou implementovány v coroutines. Rozhodování, kterou akci zvolit, je ve funkci Update. Unity podporuje filozofii komponentů, jež popisují jednotlivé prvky chování postavy. Proto rozhodování o pohybu/navigaci, jako je pronásledování nebo únik, jsou umístěny v samostatných komponentech. V UE je pohyb postav zahrnut v behaviorálních stromech.

Pro rozhodování v Unity jsou použity různé informace z herního světa, atributy energie postav a časovače. Ve funkci Update je tedy rozhodnuto o akci nepřítele, která, pokud je akce dostupná, obnoví časovače, ubere postavě energii a zavolá coroutine, jež akci obstará. Pro ovládání vstupů do coroutines jsou implementovány bool mutex (mutual exclusion) zámky, které nepovolují vykonání další akce, pokud právě nějakou akci nepřítel provádí. Tento zámek také může ukončit vykonávání funkce Update (není nutné se rozhodovat, pokud právě provádíme akci, jedině pokud bychom chtěli AI, která uvažuje o akcích předem).

Provedení akce je obecně následující: zamkneme zámek, nastavíme animaci akce pomocí změny bool parametru v Animátoru (tedy samotný přechod za nás provede animační stavový automat), podle animace počkáme (například při úderu pěstí chceme, aby byly hráči sebrány životy až po úderu, ne při napřažení), provedeme akci (sebrání životů, skok postavy, atd.), přepneme stav animace a případně obnovíme pohyb postavy (viz obrázek 6.1).

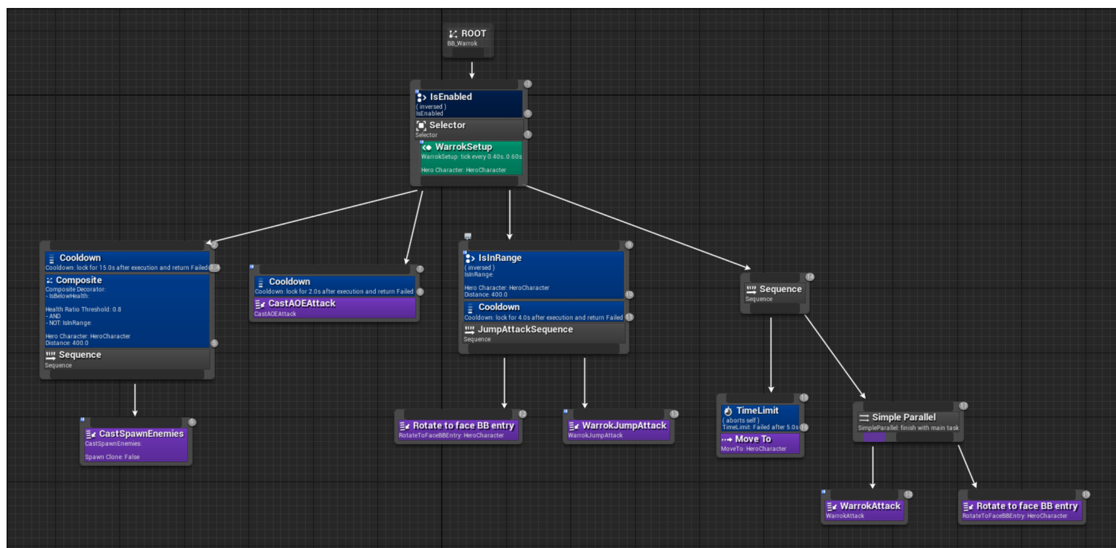
6.2 Behaviorální stromy v Unreal Engine

V UE byl použit nativní systém, jež umožňuje vytvářet behaviorální stromy ve vlastním grafickém rozhraní. Každý behaviorální strom má svůj tzv. Blackboard, jež udržuje hodnoty, se kterými strom může pracovat. Jinými slovy je to seznam proměnných stromu. Blackboards podporují dědičnost a také synchronizaci proměnných pro všechny AI ovladače, jež tento strom používají. Stromy se v UE standardně skládájí ze tří hlavních uzlů, což jsou kompozitní uzly Selector, Sequence a Simple Parallel. Selector postupně spouští potomky zleva, přičemž ukončí svou činnost, pokud jeden z nich uspěje a vrací nahoru úspěch. Pokud žádný neuspěje, vrací neúspěch. Sequence též spouští potomky zleva. Pokud žádný z potomků neselže, vrací úspěch, a obráceně, pokud jeden selže, ukončí vykonávání a vrací neúspěch. Simple Parallel je speciální varianta paralelního uzlu v UE, jež má v levém listu jednu akci (Main Task) a v pravém podstrom, který má být s touto akcí vykonáván paralelně. Pří skončení hlavní akce (levého listu), je podle atributu Finish Mode buď vykonávání pravého podstromu ukončeno, nebo se čeká na jeho dokončení.

Kromě kompozitních jsou zde 3 další kategorie uzlů: Services, Tasks a Decorators. Service uzly mohou být umístěny na uzlech kompozitních. Slouží k aktualizaci proměnných na Blackboard. U každého Service uzlu lze též nastavit jak často bude vykonáván. Task uzly jsou samotné akce, které AI postava provádí. Mezi Task uzly patří například přemístění postavy, přehrání animace, útok postavy, vydání zvuku, čekání a další. Decorator uzly lze umístit buď na uzly kompozitní nebo na Service uzly. Popisují podmínky, které je nutné splnit pro vykonání jejich podstromu. V UE lze vybírat z předpřipravených uzlů jako jsou Cooldown, Cone Check, Does Path Exist, Time Limit a další. Každý z těchto druhů uzlů je možné implementovat pro vlastní potřeby [47].

```
protected virtual IEnumerator TaskAttack() {
    canAct = false;
    movementComponent.StopMovement();
    anim.SetBool("attacking", true);
    yield return new WaitForSeconds (attackDelay);
    if (AttackHitCheck(playerCharacter)) {
        playerCharacter.DecreaseHealth(damage);
    }
    yield return new WaitForSeconds(postAttackDelay);
    anim.SetBool("attacking", false);
    movementComponent.ResumeMovement();
    canAct = true;
}
```

Obrázek 6.1: Základní coroutine pro vykonání akce útoku nepřátelské postavy v Unity.



Obrázek 6.2: Behaviorální strom nepřítele v UE. Šedé uzly jsou kompozitní, modré jsou Decorator, zelený je Service a fialové jsou Task uzly.

7. Herní mechaniky žánru

7.1 Úkolový systém

Pro hru byl v obou enginech vytvořen robustní úkolový systém. Při tvorbě byl kladen důraz na škálovatelnost systému pro jeho budoucí rozšíření. Nyní představíme jak byl systém původně implementován a jak se změnil během vývoje. Dále budou stručně popsány možnosti systému a jak jsou úkolová data spravována.

7.1.1 Unity

V rámci předmětu Počítačové hry a animace byl úkolový systém v Unity pevně vázán na herní objekty ve scéně. Tento systém byl ovšem náročný pro tvorbu úkolů mezi více scénami, neboť vyžadoval manuální nastavení parametrů každého úkolu zvlášť. To znamená, že pro změnu vlastností úkolu bylo potřeba vyhledat příslušný objekt ve scéně. Protože bylo nutné u každého úkolu uvést jeho pořadové číslo (i napříč scénami), systém vyžadoval neustálé přepínání mezi objekty a scénami, a opakované přepisování pořadových čísel, pokud byl úkol do systému přidán nebo z něj vyjmut.

Nově navržený systém v Unity je založen na zrušení závislosti (decoupling) informací o úkolu od samotného objektu ve scéně. Informace o úkolu byly přesunuty do externího JSON souboru, zatímco ve hře má k seznamu úkolů přístup singleton třída QuestManager. QuestManager na začátku hry nahraje databázi z JSON souboru a následně seznam úkolů spravuje. Každému objektu ve scéně může být přiřazen buď QuestComponent, nebo MultiQuestComponent, jež drží seznam ID čísel úkolů pro tento objekt. Obě tyto komponenty notifikují QuestManager o splnění úkolů. Systém umožňuje mnoho různých přístupů k návrhu postupu ve hře. Bylo též automatizováno nastavení a výběr dialogů pro jednotlivé NPC (Non-Playable Character) (viz kapitolu “Systém dialogů”).

Nový systém umožňuje splnit více úkolů zároveň (není to ovšem nutné). Jako dobrý příklad lze uvést Trigger Area úkol, pro který nastavíme, že při vstupu i výstupu z jím definované oblasti lze splnit úkol. Pokud na tuto oblast navážeme dva úkoly s ID 0 a 1, a povolíme splnění více úkolů současně, budou oba úkoly splněny, jakmile vejdeme do této oblasti (pokud jsou označeny jako dostupné - jinými slovy jsou ve stavu Available). Pokud bychom takovému chování chtěli zamezit, máme v systému dvě možnosti jak toho dosáhnout.

První možnost je v JSON databázi úkolu s ID 1 nastavit prerekvizitu, že je před jeho splněním nutné splnit úkol s ID 0. Při prvním vstupu do oblasti bude splněn opravdu jen úkol s ID 0, čímž se úkol s ID 1 odemkne. Při výstupu z oblasti tedy splníme úkol s ID 1. Samozřejmě to platí i obráceně, ovšem je dobré dodržovat konvenci, že dříve bude splněn úkol s nižším ID číslem.

Druhá možnost je u MultiQuestComponent nastavit parametr AllowParallelCompletion na false. V tomto případě je splněn pouze jeden z nich (preference je dána úkolu s nižším ID), přestože jsou oba dostupné (state == Available). Výběr postupu je na rozhodnutí vývojáře. Systémy lze libovolně kombinovat.

Po dokončení úkolu je hráči zobrazeno dialogové okno o splnění úkolu. Zde je použita fronta pro případy, kdy je splněno více úkolů zároveň nebo hráč před splněním dalšího úkolu nezrušil dialogové okno. Při zobrazení jsou z fronty postupně vybírány všechny dokončené úkoly, o kterých hráč nebyl doposud informován.

Dále je v systému zahrnuta podpora pro notifikaci jednotlivých QuestComponent ve scéně při úspěšném splnění všech potřebných prerekvizit pro jejich odemknutí. To může být například použito, pokud chceme do hry vložit nepřítele nebo předmět až poté, co splníme požadovaný úkol ve hře. Zamezí se tím též případům, ve kterých by hráč zabil nepřítele, jehož úkol nebyl doposud odemčen, a tím by byl znemožněn postup ve hře, dokud nebude restartována hra nebo úroveň (čímž je nepřítel obnoven).

Je zde též možné přidělit každému předmětu třídy Item úkol. Předměty mohou být vyžadovány pro interakci s jinými objekty ve hře (třídy Interactible). Sebraný úkolový předmět nelze z inventáře vyřadit, dokud není splněn jemu příslušný úkol (to neplatí v UE, kde je možné úkolové předměty libovolně vyřadit). Otázkou je, jak zachovat informaci o použitých objektech jako jsou například klíče od dveří. Každý klíč otevírá určité dveře, ukládat však pro každé dveře, zda byly odemčeny, je zbytečně složité, zejména pokud bychom měli takových interaktivních párů více. Byl proto vytvořen systém ar-

chivace předmětů. ID předmětu, který byl použit v rámci úkolu, je uloženo do archivu. Pokud se při dalším herním sezení pokusíme tytéž dveře otevřít, je ověřena přítomnost klíče v archivu nebo v inventáři.

7.1.2 Unreal Engine

Systém v UE je založen na stejném principu. Chronologicky byl systém v UE vytvořen před nově upraveným systémem v Unity. Zde jsou úkoly načítány z tzv. Datatable. Datatable je interní reprezentace excelových tabulek v UE. Pro vytvoření Datatable je nejdříve nutné vytvořit korespondující strukturu (struct), která popisuje jednotlivé sloupce tabulky. Datatable lze navíc importovat v podobě CSV nebo JSON souborů. Jejich editace je ovšem možná i v enginu (není podporováno kopírování řádků/sloupců tabulky).

Podobně jako v Unity, má každý úkol své ID a i zde singleton QuestManager spravuje hráčův postup ve hře. V UE musí každý objekt, jenž má na sebe vázaný úkol, implementovat rozhraní IQuestObject a mít svůj vlastní MultiQuestComponent. MultiQuestComponent zde navíc udržuje informace o stavech jednotlivých úkolů. Zde došlo k nálezu jednoho důležitého nedostatku Blueprint editoru, a to, že nejsou podporovány datové typy slovníků (dictionary). Proto je seznam ID a stavů oddělen do dvou polí, ve kterých je velmi důležitá stejná indexace. Dále je v Blueprint třídě nutné dopředu odlišit, jakým způsobem mohou být úkoly splněny. Jako příklad uvedu třídu Item, kde mohou být úkoly splněny buď sebráním předmětu nebo také jeho použitím. Proto byly vytvořeny dva MultiQuestComponent komponenty (UseMultiQuestComponent a PickupMultiQuestComponent). Přiřazení komponentů ve scéně je v UE též podporováno. Oba systémy se nijak jinak neliší a struktura databáze úkolů je téměř totožná.

7.2 Systém dialogů

S úkolovým systémem je úzce spjatý také systém dialogů, tedy systém rozhovorů s NPC ve hře. NPC mohou hráči zadat v dialogu úkoly nebo ho informovat o současném stavu úkolu.

Dialogový systém je založený na načítání a parsování textových souborů ve vlastním formátu. Formát je následující: [číslo mluvčího]:[text]. Při parsování jsou odstraněny veškeré neplatné hodnoty (neplatné číslo mluvčího je ve výchozím nastavení převedeno na číslo NPC). Text navíc umožňuje vkládat komentáře a prázdné řádky. Každá NPC může mít celou řadu dialogů, které jsou členěny do tří kategorií.

První kategorií jsou dialogy, při nichž zadává NPC hráči nějaký úkol. Takové dialogy jsou aktivovány v případě, kdy hráč interaguje s NPC, na které je dostupný libovolný úkol. Jak již bylo zmíněno v kapitole o úkolovém systému, každý objekt ve hře může být asociován (až na výjimky, kde by to nedávalo smysl) s více úkoly, to samé platí pro NPC a jejich dialogy. Systém je ovšem robustní v tom ohledu, že pokud uživatel v úkolové databázi nenastaví prerekvizity mezi dialogy, jež na sebe navazují, a tyto dialogy probíhají mezi hráčem a stejnou NPC, budou postupně vykonány podle jejich ID čísla, kde úkol s nižším ID má vyšší prioritu.

Druhou kategorií je dialog doplňkový k současnému úkolu. Ten je použit v případě, kdy NPC na sobě v současné době nemá žádné úkoly. Každá NPC může mít vlastní dialogy ke všem úkolům, jež jsou rozpoznány podle jména postavy a ID úkolu.

Třetí možnost je výchozí rozhovor, který je zvolen, pokud ani jedna z předešlých možností není dostupná. Ve většině případů se jedná o informování hráče, že tato postava nemá žádné důležité sdělení.

Pokud při interakci s NPC není dostupná ani jedna možnost (tedy soubory k těmto dialogům nebyly nalezeny), s postavou nelze interagovat.

Při tvorbě systému byl učiněn důležitý nález. Blueprint editor UE nepodporuje regulární výrazy (nejsou poskytovány žádné funkce nebo datové typy pro práci s regulárními výrazy). Pro jejich použití je potřebná externí knihovna (třeba i vlastní). V Unity jsou regulární výrazy použity pro výměnu speciálních symbolů za jména NPC a hráčovy postavy v dialogu (symboly $$playerName$$ a $$npcName$$).

Obrázek 7.1: Dialog s NPC v UE.

7.3 Systém předmětů

Jak již bylo zmíněno, výsledná hra obsahuje také prvky RPG žánru. Jeden z takových stěžejních prvků je systém předmětů a inventáře, který byl pro hru implementován v obou enginech. Nyní krátce rozebereme jak funguje a kde byly zvoleny při implementaci mezi enginy odlišné postupy.

Hlavním stavebním kamenem systému je předmět samotný, který je v obou případech reprezentován třídou Item a rozhraním IItem. Protože každý objekt musí být interaktivní a může být také součástí úkolu, je též nutné implementovat rozhraní IInteractible a IQuestObject. Původně byla třída Item potomkem třídy Interactible, ovšem z důvodů nárůstu odlišností během vývoje bylo nakonec zvoleno udělat je nezávislé.

Všechny předměty ve hře sdílí své základní atributy, které jsou popsány v proměnné ItemData. ItemData je serializovatelná třída v Unity a struct v UE. ItemData obsahují informace o třídě předmětu (použito při jeho dynamické tvorbě během běhu hry), reference na samotný předmět ve scéně, název, ikonu, model, materiály, raritu a další. Důležitá hodnota v ItemData je ClassSpecificRow, která nás odkazuje na doplňující informace o předmětu.

Informace o předmětech jsou uloženy v JSON souboru v případě Unity a v Datatable v případě UE. Při tvorbě systému byl kladen důraz na jeho znovupoužitelnost pro různé situace ve hře. Místo toho, abychom vytvářeli různé prefabrikáty nebo Blueprint třídy, lze použít objekt obecné třídy Item, kterému je možné jak v editoru, tak během hry, nastavit model a materiály. Pro konstrukci a dekonstrukci předmětů byla použita třída ItemManager. ItemManager je schopný vytvořit libovolný předmět z databáze a vložit ho do hry. Pro nastavení proměnných u předmětů, které nejsou třídy Item, ale jejím potomkem, je navíc nutné použít reflexi. Podle jména třídy předmětu jsou načtena doplňující data z příslušné databáze. Například pokud chceme vytvořit zbraň, je podle atributu ClassSpecificRow vyhledána položka v databázi zbraní, kde jsou uloženy informace jako velikost zásobníku, rychlost střelby, atd.

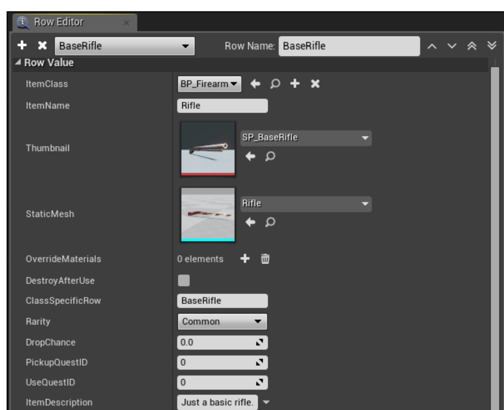
Pro vkládání předmětů do hry jsou dále vytvořeny dvě pomocné třídy, ItemSpawn a ItemSpawner/ItemSpawnerComponent. ItemSpawn slouží pro vkládání předmětů v editoru. ItemSpawn požaduje identifikační název předmětu, který má být do hry vložen. ItemSpawner pak slouží pro vkládání skupin objektů během hry, například při otevření truhly nebo pokoření nepřítele. Pro každý ItemSpawner lze nastavit rozmezí počtu předmětů a atributy, jež musí předměty splňovat, jako je například jejich třída, rarita nebo samotný název. Pro správu inventáře je ve hře třída InventoryManager. InventoryManager v sobě udržuje informace o všech předmětech hráče a také se stará o jejich patřičné ukládání a nahrávání.

7.3.1 Serializace dat

Zde bych krátce zmínil pár odlišností mezi JSON a Datatable serializací v Unity a UE. První důležitou odlišností je nativní podpora pro tabulky typu Datatable v UE v podobě vlastního editoru. Editor nám umožňuje nastavovat proměnné podle výběru dostupného v adresáři projektu. To znamená, že pro výběr statického modelu nebo např. enum hodnoty stačí vybrat z listu existujících položek. V UE tak máme zaručenou správnost dat v tabulce. V Unity, kde pracujeme s JSON souborem, je nutné na některé objekty, jako jsou modely a další, odkazovat pomocí jejich absolutní nebo relativní cesty (což za nás UE Datatable dělá sama - exportovaný JSON nebo CSV soubor z Datatable obsahuje relativní cesty k assetům a jeho zpětnou importací do enginu by měly být reference opět funkční). Nejvhodnějším přístupem v Unity je všechny použité objekty uložit do adresáře Resources a načítat je přes relativní cestu funkcí Resources.Load(relativeFilepath). Zde je důležité čtenáře upozornit, že relativeFilepath nesmí obsahovat příponu souboru (např. json, txt, fbx), aby mohlo dojít k jeho úspešnému načtení. Podle nahraného typu sou-

boru můžeme objekt přetypovat pro použití v enginu. Co se týče enum proměnných, JsonUtility pro serializaci a deserializaci v Unity nepodporuje implicitní přetypování string proměnných na enum typy. Pokud bychom toho chtělo docílit, musíme string hodnotu přetypovat manuálně ((EnumType)Enum.Parse(typeof(EnumType), enumString)). Proto je nutné v JSON souboru enum typy ukládat jako celá čísla, což může být pro databázi předmětů poněkud nepřehledné.

Důležité je ještě zmínit, že JsonUtility nepodporuje serializaci/deserializaci samostatného pole. Tento problém lze obejít tak, že vložíme pole do JSON objektu a vytvoříme serializovatelnou C# třídu, která obsahuje též obsahuje pouze toto pole. Tuto třídu nazveme Wrapper class.



(a) Nastavení hodnot jedné položky Datatable v UE.

```
{
  "itemID": 4,
  "itemClass": "Firearm",
  "itemName": "Base Rifle",
  "itemDescription": "A common rifle.",
  "iconResource": "SP_BaseRifle",
  "meshResource": "Rifle",
  "actionText": "Equip Rifle",
  "destroyAfterUse": false,
  "classSpecificRow": 1,
  "additionalClasses": [ ],
  "canRotate": true,
  "questID": -1
},
```

(b) Jedna položka databáze předmětů v JSON souboru použitého v Unity.

Obrázek 7.2: Formát uložení dat v UE a Unity.

8. Kolize

Kolize jsou další důležitou částí vývoje hry. Kolize mohou být použity pro fyzikální simulaci, herní logiku a také uživatelské rozhraní. Základem kolizního systému ve hře jsou tzv. kolizní obalová tělesa. Jak Unity, tak UE používají fyzikální engine PhysX 3.3 [48] [49]. Protože námi vytvořený projekt nevyužívá žádnou složitou fyzikální simulaci, zaměříme se v této kapitole především na použití kolizních těles pro herní logiku, zejména pak na filtrování kolizí.

8.1 Kolizní tělesa

Kolizní obalová tělesa popisují tvary objektů/modelů pro jejich fyzickou simulaci. Nejméně náročná tělesa jsou primitiva jako je koule (Sphere), tažená koule (Capsule) nebo kvádr (OBB - Oriented Bounding Box). Kromě standardních primitiv lze též použít tzv. k-DOP tělesa (k-dimensional Discrete Oriented Polytope). Ve zkratce jsou k-DOP obalová tělesa orientované mnohostěny, jež jsou definovány k/2 jednotkovými vektory. Čím vyšší je hodnota k, tím přesnější a zároveň výpočetně náročnější je takové obalové těleso. Nakonec můžeme jako obalové těleso použít konvexní obálku modelu [50].

Kolizní tělesa jsou v Unity komponenty, které lze přiřadit herním objektům ve hře. Je zde možné použít základní kolizní tělesa a také tzv. Mesh Collider, který buď kopíruje povrch modelu, nebo je jeho konvexní obálkou, pokud ho označíme jako Convex v Inspector panelu. Unity bohužel nenabízí tvorbu k-DOP obalových těles.

V Unity odlišujeme dva druhy kolizních těles podle toho, zda herní objekt, na kterém jsou umístěny, má komponent tuhého tělesa (Rigidbody). Pokud vlastnící objekt nemá tuhé těleso, nazýváme kolizní těleso jako statické a v opačném případě jako dynamické. Statická kolizní tělesa jsou použita pro nehybné předměty jako jsou zdi, země a další. Pokud plánujeme během hry s polohou tělesa jakkoliv manipulovat, je nutné mu přidělit

tuhé těleso, které označíme jako kinematické (IsKinematic nastaveno na true). Kinematická tělesa nijak nereagují na vnější síly a nerespektují fyzikální pravidla jako klasická tuhá tělesa, ale je možné s nimi manipulovat ve skriptech. Na rozdíl od objektu bez komponentu tuhého tělesa však vytváří tření a probouzí ostatní tuhá tělesa, se kterými ve scéně při svém pohybu interagují. V Unity je tedy obecně důležité odlišovat tato tři možná kolizní tělesa: statické, dynamické-kinematické a dynamické [51].

Každému objektu je v Unity možné přiřadit neomezený počet kolizních těles, čímž vytvoříme tzv. kompozitní kolizní tělesa (compound colliders). Kompozice kolizních těles je vhodná pro snížení výpočetní náročnosti a dobrou estimaci tvaru modelu. Druhý způsob, jak vytvořit kompozitní kolizní tělesa, je pomocí tvorby hierarchie herních objektů s kolizními tělesy. V takovém případě je důležité dodržet pravidlo, že kořenový objekt má jako jediný komponent tuhého tělesa v této hierarchii.

V UE máme k dispozici samostatný editor pro statické modely a jejich nastavení včetně jejich kolizních těles. Kolizní tělesa pro skeletální modely jsou spravována v již zmiňovaném Physics Asset editoru. V editoru statických modelů je možné pro model vytvořit hierarchii kolizních těles. Zde jsou na výběr kromě primitivních těles a konvexních obálek (Auto Convex Collision) též k-DOP tělesa. U tvorby konvexní obálky je též možné nastavit její přesnost a omezit její maximální počet uzlů. Pro k-DOP tělesa je na výběr z 10DOP-X, 10DOP-Y, 10DOP-Z, 18DOP a 26DOP [52].

8.2 Filtrování kolizí

Filtrování kolizí (collision filtering) slouží k určení, které objekty se budou navzájem překrývat nebo mezi sebou budou kolidovat podle jejich rozdělení do kategorií a vrstev.

Při kolizi objektů v Unity je jejich výsledná interakce závislá na konfiguraci (nebo přítomnosti) tuhého tělesa na každém z nich. Pro kolizní matice v závislosti na tuhých tělesech a nastavení trigger kolizí viz dokumentaci: [51]. V Unity můžeme navíc objekty rozdělit do tzv. vrstev (Layers). Výpočet kolize pak bude proveden mezi objekty podle kolizní matice, kterou můžeme najít v Edit → Project Settings → Physics. Kolizní matice jednoduše popisuje, jaké vrstvy mezi sebou mohou kolidovat [53]. Vrstvy je možné vytvářet v Project Settings → Tags and Layers. Vhodným použitím vrstev a kolizní matice lze snížit výpočetní náročnost.

Co se týče vrhání paprsků (Physics.Raycast) je v Unity možné specifikovat, které vrstvy budou zasaženy a které ignorovány pomocí LayerMask objektu. LayerMask je bitová maska, jejíž jednotlivé bity popisují, zda může být vrstva zasažena (1 ano, 0 ne). Pro její tvorbu můžeme použít základní bitové operace jako je levý bit shift, boolean or a invertor. LayerMask lze též nastavit v Inspectoru, pokud je to veřejná proměnná třídy. Dále lze u funkce Physics.Raycast nastavit, zda budou vyvolány trigger kolize. Globální nastavení tohoto parametru můžeme opět najít v nastavení Physics pod názvem Queries Hit Triggers.

V UE je systém principielně podobný, ale v praktickém použití se můžeme setkat s mnoho odlišnostmi. Zde není filtrování kolizí nijak závislé na vlastnostech fyzikálního tělesa. Pokud má tedy model nastaveno Simulate Physics na true (má tuhé těleso) není důležité při detekci kolizí. To samé platí o jeho mobilitě. Detekce kolizí je stejná jak pro tělesa statická, stacionární, tak mobilní.

Každý statický model má v Details panelu nastavení Collision, kde můžeme specifikovat mnoho jeho vlastností. U každého tělesa je možné určit, zda vyvolá tzv. Event Hit (událost zásahu) a zda generuje Overlap Events (překryv, podobně jako Trigger v Unity), přepsat jeho fyzikální materiál a nastavit tzv. Collision Presets.

Collision Presets nám umožňují vybrat jedno z předpřipravených kolizních nastavení, nebo vytvořit pro zvolený objekt nastavení unikátní (Custom). Lze zde nastavit, zda těleso reaguje na kolize fyzikální, zda reaguje na prostorové dotazy (queries jako je linetrace a sweep), anebo zda nereaguje na kolize vůbec. Nadále je v nastavení možné zvolit Object Type a Object Type a Trace Channel odezvy. Object Type a Trace Channel jsou obdobou vrstev v Unity.

UE odlišuje tyto dva typy vrstev podle toho, která strana (objekt nebo dotaz) vybírá, co s čím bude kolidovat. Každý objekt musí mít určen svůj Object Type a také odezvy na ostatní Object Types, zatímco pro Trace Channel může mít pouze nastaveny odezvy. Filozofie je, že pokud vysíláme dotaz jako je vrhání paprsků (linetracing), pak můžeme určit Trace Channel, který vyhledáváme. Jestliže paprsek narazí na objekty, které mají odezvu na námi vybraný Trace Channel nastavenou na block, bude objekt zasažen. Dobrý příklad je odlišení dvou Trace channel: Visibility a Weapon. Neprůstřelné sklo bude blokovat Weapon Channel a bude ignorovat Visibility Channel. Jinými slovy kulka zbraně jím neproletí (způsobí zásah) a pohled postavy jím projde. Paprsky lze vrhat i podle Object Type, kde určíme jaké Object Types má paprsek zasáhnout (obdoba použití LayerMask). Pro kolize mezi objekty je nutné použít Object Types, kde výsledek

		Object A		
		Ignore	Overlap	Block
Object B	Ignore	Ignore	Ignore	Ignore
	Overlap	Ignore	Overlap	Overlap
	Block	Ignore	Overlap	Block

Obrázek 8.1: Kolizní tabulka dvou fyzikálních těles v UE. [54]

kolize je závislý na nastavení obou objektů. Zde je vždy vybráno to "méně blokující", tzn. že pokud se srazí dva objekty, jež mají vztah block-overlap, pak se překryjí, zatímco pokud mají vztah block-block, tak se srazí (viz tabulku 8.1) [54]. Důležité je zmínit, že pro úspešné vyvolání Overlap Event v objektech je nutné, aby k sobě měli oba vztah overlap (nebo jeden z nich block a druhý overlap) a oba měli nastaveno Generate Overlap Events na true [55]. Nové Object Types, Trace Channels a Collision Presets můžeme definovat v nastavení projektu.

Tento systém je možné rozebrat do mnohem větší hloubky, ovšem v rámci této práce na to není prostor. Hlavním cílem zde bylo nastínit, jak je tento systém v každém enginu navržen. UE zde nabízí velmi dynamické řešení, které není vázané na fyzikální simulaci objektu. Největší výhodou UE je přehledná tabulka popisující kolizní chování u každého objektu. V obou enginech musí uživatel dávat velký pozor, aby byla interakce nastavena správně u obou objektů, neboť může jednoduše dojít k chybám jako je například negenerování Overlap událostí v UE, pokud nemají oba objekty nastavený atribut Generate Overlap Events na true. V Unity je důležité pamatovat si kolizní matice podle nastavení jejích tuhých těles. V UE je vyhledávání pomocí dotazů/paprsků ulehčeno odlišením vrstev Object Type a Trace Channels, zatímco v Unity je nutné používat LayerMask objekty.

(a) Trojúhelník vypočítaný podle úhlu a vzdálenosti.

(b) Dva paprsky společně s taženými koulemi (spherecast) podle polohy kostí nepřátelské postavy.

Obrázek 8.2: Vizualizace oblastí zásahu při útoku nepřátelské postavy v UE.

8.3 Detekce zásahů

Ve hře je také důležité patřičně detekovat zásahy hráče nepřáteli. Zde existuje přístupů mnoho. Výběr způsobu detekce je silně vázaný na žánr hry a požadovanou přesnost. Například v bojové hře je požadovaná přesnost zásahů velmi vysoká, naopak v boji proti hordě nepřátel nemusí být přesnost exaktní, stačí estimovat údery pomocí kolizních těles nebo jiných nástrojů. Systém v Unity a v UE je implementován podle detekce přítomnosti hráče v kolizním trojúhelníku, jemuž lze nastavit vzdálenost od postavy a úhel rozsahu. V UE byla také implementována detekce pomocí vrhání paprsků nebo koulí (linetracing/spheretracing), kde počátkem paprsku byl ramenní kloub nepřátelské postavy a koncem jeho ruka (vektor lze samozřejmě prodloužit pro umělé navýšení dosahu útoku nepřítele). Tento systém byl opuštěn, neboť pro dobré výsledky je nutné vysílat paprsků více v určitých časových intervalech útoku, což je pro takový typ hry zbytečně výpočetně náročné a výsledky jsou téměř stejné jako při estimaci pomocí trojúhelníku.

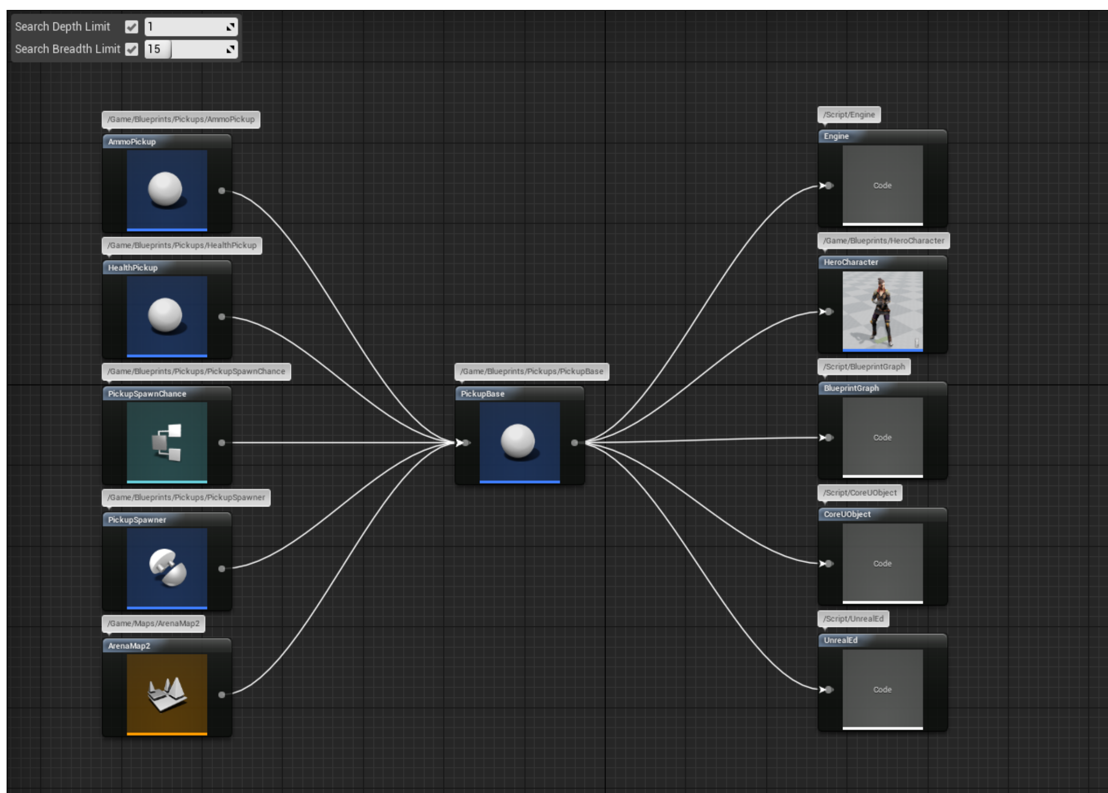
9. Ostatní

9.1 Organizace scény

Co se týče organizace scény, Unity nabízí jednoduché řešení v podobě použití prázdných herních objektů pro tvorbu hierarchie scény. Problém nastává ve chvíli, kdy máme skupiny singleton objektů, jež chceme zachovat mezi scénami (DontDestroyOnLoad). Pokud bychom takovou skupinu vložili do "složky" (tedy jako potomky prázdného herního objektu), tento objekt musí být také singleton, který je zachován mezi scénami, jinak budou z důvodu smazání svého rodiče všichni potomci také smazáni, přestože jsou označeni jako DontDestroyOnLoad. Musíme být tedy opatrní při použití prázdných objektů jako složek pro organizaci scény, neboť jsou vázány na své potomky svou polohou a vlastnostmi (pokud posuneme "složku", posuneme všechny potomky). UE poskytuje v editoru klasické složky, které nejsou nijak vázány na hierarchii transformací objektů ani na logiku hry.

9.2 Závislosti assetů

Při tvorbě rozsáhlého projektu je potřeba dobře spravovat jednotlivé assety a jejich závislosti. UE nabízí tzv. Reference Tree, který ve vizuální podobě představuje všechny závislosti jednotlivého assetu. V Unity je bohužel snadné zjistit pouze jednosměrný vztah, a to, které assety námi zvolený objekt používá (např. textury, které námi zvolený materiál používá; pravé tlačítko myši → Select Dependencies). Opačný vztah bohužel nelze pro libovolný předmět jednoduše zjistit (např. které objekty ve hře používají námi zvolený materiál). Jediné co zjistit lze, je zastoupení námi zvoleného prefabrikátu v současné scéně (pravé tlačítko myši → Find References In Scene). Tento nedostatek je nejvíce znatelný při čištění projektu od nepoužitých assetů jako jsou materiály, textury a další.



Obrázek 9.1: Strom referencí v UE.

9.3 Kvantitativní testování

Z důvodů komplexnosti obou enginů a jejich implementace jsem se rozhodl, že kvantitativní porovnání enginů v rámci jejich výkonu, rychlosti a dalších parametrů nelze objektivně učinit. Porovnání výkonu obou enginů je poněkud komplexní téma a jednoduchá měření rychlosti vykreslování a dalších parametrů by nebyla vypovídající o kvalitě enginu. Navíc hra v Unity používá forward rendering, podle kterého bylo navrženo osvětlení (použití lightmap místo dynamických světel), zatímco hra v UE používá deferred rendering, neboť forward rendering je v současné verzi experimentální [57] [58]. V potaz by musela být i brána komplexnost shaderů, post processing, globální iluminace, a další.

9.4 Pořizování snímků

Při tvorbě této práce jsem narazil na jednu velmi užitečnou pomůcku UE. Jedná se o její nativní nástroj pro pořizování snímků hry ve všech editačních oknech ve velmi vysokém rozlišení, které lze použít pro reklamní a promoční účely. Pro pořízení běžného snímku (rozlišení okna editoru) lze v UE použít klávesu F9. Pro pořízení snímku ve vysokém rozlišení je nutné použít tzv. High Resolution Screenshot Tool (HRSST). Pomůcka nám umožňuje specifikovat kvalitu a oblast pořízeného snímku, exportovat jednotlivé vizualizace G-Bufferu (geometrický buffer) a též exportovat do formátu OpenEXR, který podporuje HDR (High Dynamic Range) obrázky. V HRSST lze navíc použít masku, jež bude ignorovat všechny objekty až na ty, které mají nastaveno Render Custom Depth na true [59]. Pořízení snímků je možné ve všech editačních oknech UE jako je např. okno pro editaci statického modelu nebo pro tvorbu animací postav. V Unity je pro pořizování snímků poskytnuta funkce Application.CaptureScreenshot, jež nám také povoluje pořídit snímky ve vysokém rozlišení podle parametru superSize, ovšem zbylá funkcionalita zde chybí [60]. Pro použití v editoru byl vytvořen skript ScreenshotUtils, jenž rozšiřuje editor pomocí použití anotace [MenuItem] společně se statickými funkcemi.

10. Závěr

K závěru práce bych přistoupil ze dvou úhlů pohledu. Prvním z nich je pohled začínajícího vývojáře nebo studia, které chce vytvořit a vydat plnohodnotnou 3D hru. Druhým je hodnocení enginů v rámci jejich vhodnosti při výuce základních herních konceptů a algoritmů.

Z pohledu vývojáře, který chce vytvořit 3D hru je UE jednoznačná volba. UE dovoluje vytvářet rychlé prototypy, poskytuje pomůcky a editory pro všechny důležité části vývoje hry, které v Unity chybí, zejména Material editor, Sequencer a Behaviour Tree editor. Nástroje pro tvorbu terénu a flóry prostředí jsou mnohem komplexnější a umožňují dosáhnout přesvědčivých výsledků velmi rychle. Samozřejmě tím není vyloučeno, že podobných výsledků není možné dosáhnout v Unity, ale zde to, podle mého názoru, vyžaduje mnohem více úsilí a zkušeností vývojáře. UE je navíc velmi přátelský pro tvorbu v týmech, neboť poskytuje dvě možnosti skriptování, kde jedna vychází vstříc i méně zkušeným programátorům. Tím může být skriptovací práce rozložena mezi více členů týmu. Například designer je sám schopen si pomocí Blueprint editoru vytvořit logiku pro základní objekty jako jsou lampy, dveře, herní události a další. Na závěr bych zde podotknul, že snad největší výhodou UE oproti Unity v rámci projektu, jeho škálovatelnosti a budoucí údržby a rozšiřování je dostupnost zdrojového kódu. Pokud tedy vývojáři něco v enginu nevyhovuje, nebo se setkal s chybou, může zdrojový kód upravit/opravit a pokračovat dále ve vývoji.

Z pohledu výuky předmětu PHA je situace ovšem složitější. Jedním z hlavních cílu předmětu je výuka základních herních algoritmů. Zde je Unity vhodnější volbou, neboť nemá žádný vlastní herní framework a všechny (i základní) algoritmy je nutné implementovat (některé skripty jako ovladače postav a kamery můžeme najít ve Standard Assets). UE poskytuje ve svém herním frameworku řešení pro většinu základních herních mechanismů, což může být vnímáno v rámci předmětu poněkud negativně.

Dobrým příkladem pro úplného začátečníka je tvorba projektilů. V Unity je student nucen seznámit se se strukturou herních objektů a komponentů. Dále musí objektu přidat komponent tuhého tělesa (RigidBody) a svůj vlastní skript pro projektily. Ve skriptu zjistí, jak mezi sebou komponenty mohou komunikovat a jak získat jejich referenci pomocí funkce GetComponent. Navíc by si měl student uvědomit, že GetComponent je náročná operace, a proto je ji dobré provést jen jednou v metodě Start nebo Awake místo Update, čímž se blíže seznámí s herní smyčkou Unity. Tuhému tělesu nastaví počáteční rychlost, a pokud chce projektil po nějaké době zničit, musí buď vytvořit vlastní časovač, nebo použít metodu Invoke. V UE pro tvorbu projektilu stačí vytvořit Blueprint třídu, vybrat statický model projektilu, zaškrnout položku Simulate Physics a přidat ProjectileMovement komponent, kde je vše implementováno. Pak stačí pouze nastavit jeho parametry a je hotov. Totéž platí i pro složitější systémy jako je pohyb postav, který je zakomponován do herního frameworku UE. V Unity je však také možné použít různé skripty pro herní mechaniky (jako je pohyb postavy), které se nachází ve Standard Assets. Zakomponovat je do projektu však musí programátor sám.

Na druhou stranu se student v UE odprostí od implementace jednoduchých mechanismů jako uvedený projektil a může si vyzkoušet práci s pokročilými pomůckami pro tvorbu materiálů a umělé inteligence. Co se týče umělé inteligence se nedá od studentů očekávat (především v druhém ročníku, ve kterém se bude předmět vyučovat), že implementují vlastní behaviorální stromy. Místo implementace jednoduchých systémů v Unity si mohou vyzkoušet jak fungují behaviorální stromy a jak je použít pro dosažení specifických cílů. Co se týče materiálů, studenti se mohou dobře seznámit s jejich tvorbou, používáním parametrů pro jejich jednotlivé instance, vytvářením zajímavých vizuálních efektů s pomocí různých matematických funkcí a jejich dalšími vlastnostmi.

Obecně lze říci, že i přes některé velké nedostatky UE jako je poruchovost Blueprint skriptů (která se týká i programátorů, neboť UI a další systémy je v nich nutné vytvářet) nabízí tento engine mnoho pečlivě vytvořených nástrojů, které v Unity chybí. Pro vlastní tvorbu bych mu v případě tvorby 3D her dal vždy osobně přednost. Jediný problém, co se týče výuky počítačových her, je jeho rozsáhlý herní framework, který může studentům až příliš usnadnit práci při jejich tvorbě, především v začátcích. Osobně bych doporučil použití UE, případně jeho větší začlenění do výuky, neboť je s ním možné mnohem rychleji a efektivněji vytvořit kvalitní produkt a uplatnit tak nabyté teoretické znalosti na zajímavých herních konceptech.

11. Použité assety

V projektu byly použity volně dostupné modely a animace postav ze stránky Mixamo.com [61]. Dále byly v projektu použity volně dostupné textury ze stránek Textures.com [64]. Pro tvorbu některých shaderů v Unity byl použit vizuální shader editor ShaderForge od Joachima Holméra (zakoupen na Unity AssetStore) [62]. Hudba pochází ze sbírky Ultimate Game Music Collection od Johna Leonarda Frenche (zakoupena na Unity AssetStore) [63]. Zbylé assety pocházejí buď ze základní nabídky enginů - Standard Assets v Unity, Starter Content + Content Examples + Kite Demo v UE, nebo jsou vlastní tvorbou - statické modely budov, zbraní včetně ručně kreslených textur, stromů, kamení, animované truhly, barelů a dalších předmětů.

Literatura

[1] *D*efinition-What does Role-Playing Game mean?, [online], [cit. 15.5.2017], Dostupné z: https://www.techopedia.com/definition/27052/role-playing-game-rpg

[2] *Twin Stick Control*, [online], [cit. 18.2.2017]
Dostupné z: https://www.giantbomb.com/twin-stick-control/3015-3933

[3] *Materials, Shaders & Textures*, [online dokumentace], Copyright 2017 Unity Technologies, Publication 5.6-001G 29.3.2017 [cit. 2.4.2017]
Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/Shaders.html

[4] *Materials*, [online dokumentace], [cit. 2.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/

[5] Aleksandr, *Documentation, Unity scripting languages and you*, [online blog], 3.9.2014 [cit. 15.4.2017], Dostupné z: https://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/

[6] WATKINS, Adam, *Creating Games with Unity and Maya: How to Develop Fun and Marketable 3D Games*, Burlington: Elsevier, 2011. ISBN: 978-0-240-81881-8.

[7] *Introduction to C++ Programming in UE4*, [online dokumentace], [cit. 3.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Programming/Introduction/index.html

[8] NOLAND, Michael, *Unreal Property System (Reflection)*, [online], publikováno 27.3.2014 [cit. 15.4.2017], Dostupné z: https://www.unrealengine.com/blog/unreal-property-system-reflection

[9] *Exposing Gameplay Elements to Blueprints*, [online dokumentace], [cit. 15.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/TechnicalGuide/ExtendingBlueprints/#executableandoverridablefunctions

[10] *Hot Reload*, [online dokumentace], Copyright 2017 Epic Games [cit. 16.4.2017], Dostupné z: https://wiki.unrealengine.com/Hot_Reload

[11] *Blueprint vs C++ performance*, [online fórum], 17.8.2016, příspěvek č. 17, [cit. 16.4.2017], Dostupné z: https://forums.unrealengine.com/showthread.php?3791-Blueprint-vs-C-performance

[12] *Implementing Blueprint Interfaces*, [online dokumentace], [cit. 3.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/Interface/UsingInterfaces/

[13] uživatel gmpreussner (Epic Games staff), *Call a blueprint function that is overloaded?*, [online fórum], 10.4.2016 [cit.16.4.2017], Dostupné z: https://answers.unrealengine.com/questions/133918/call-a-blueprint-function-that-is-overloaded.html

[14] *Gameplay Framework*, [online dokumentace], [cit. 9.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/index.html

[15] *Unity Roadmap*, [online], Dostupné z: https://unity3d.com/unity/roadmap

[16] *Request to be able to edit the values of a ChildActor's variables in the details panel of a blueprint*, [online], [cit. 15.4.2017], Dostupné z: https://issues.unrealengine.com/issue/UE-16474

[17] *LoadLevelAsync*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 4.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/ScriptReference/Application.LoadLevelAsync.html

[18] *Level Streaming Overview*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/LevelStreaming/Overview/index.html

[19] *LevelStreamingVolumes*, [online dokumentace], [cit. 4.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/LevelStreaming/StreamingVolumes/

[20] *AsyncOperation.allowSceneActivation*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 19.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/ScriptReference/AsyncOperation-allowSceneActivation.html

[21] *Terrain settings*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 7.4.2017], Dostupné z: https://docs.unity3d.com/Manual/terrain-OtherSettings.html

[22] *Landscape Materials*, [online dokumentace], [cit. 7.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Landscape/Materials/

[23] *Grass and Other Details*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 11.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/terrain-Grass.html

[24] *Foliage Instanced Meshes*, [online dokumentace], [cit. 11.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Foliage/

[25] *Property Binding*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/PropertyBinding/

[26] *Rewired*, [online obchod], Guavaman Enterprises, [cit. 19.4.2017], Dostupné z: https://www.assetstore.unity3d.com/en/#!/content/21676

[27] *InControl*, [online obchod], Gallant Games, [cit. 19.4.2017], Dostupné z: https://www.assetstore.unity3d.com/en/#!/content/14695

[28] *Input*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Gameplay/Input/

[29] *http://ds4windows.com*, [online] [cit. 9.5.2017], Dostupné z: http://ds4windows.com

[30] *Shading Language used in Unity*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 19.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/SL-ShadingLanguage.html

[31] *Essential Material Concepts*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/IntroductionToMaterials/index.html

[32] *Rendering Mode*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 19.4.2017], Dostupné z: https://docs.unity3d.com/Manual/StandardShaderMaterialParameterRenderingMode.html

[33] *Custom Expressions*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/ExpressionReference/Custom/

[34] *Accessing and Modifying Material parameters via script*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/MaterialsAccessingViaScript.html

[35] *Instanced Materials*, [online dokumentace], [cit. 19.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/MaterialInstances/

[36] *Parameter Expressions*, [online dokumentace], [cit. 11.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/ExpressionReference/Parameters/#staticswitchparameter

[37] ČMOLÍK, Ladislav, *Reprezentace 3D těles*, [online prezentace], ČVUT, DCGI, [cit. 13.4.2017], s. 38-41, Dostupné z: https://leyfi.felk.cvut.cz/courses/mga/lectures/05_reprezentace_3d_teles.pdf

[38] *Shape module*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 22.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/PartSysShapeModule.html

[39] *GPU Particles with Scene Depth Collision*, [online dokumentace], [cit. 13.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/EffectsGallery/1_E/

[40] *CPU and GPU Sprite Particles Comparison*, [online dokumentace], [cit. 13.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/EffectsGallery/1_A/

[41] *TC Particles*, [online obchod], [cit. 13.4.2017], Dostupné z: https://www.assetstore.unity3d.com/en/#!/content/7324

[42] *Trail Renderer*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 22.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/class-TrailRenderer.html

[43] FARREL, John, *Creating an Outline with Post Process*, [online video tutoriál], 15.7.2015 [cit. 25.4.2017], Dostupné z: https://www.youtube.com/watch?v=IEHxjw6VEMQ

[44] *Avatar Mask*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 12.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/class-AvatarMask.html

[45] *Animation Layers*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 12.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/AnimationLayers.html

[46] *Blend Nodes*, [online dokumentace], [cit. 12.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Animation/NodeReference/Blend/

[47] *Behavior Trees Nodes Reference*, [online dokumentace], [cit. 15.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/NodeReference/index.html

[48] Anthony, *High-performance physics in Unity 5*, [online blog], 8.6.2014 [cit. 22.4.2017], Dostupné z: https://blogs.unity3d.com/2014/07/08/high-performance-physics-in-unity-5/

[49] *Physics Simulation*, [online dokumentace], [cit. 22.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Physics/

[50] BITTNER, Jiří, *Detekce kolizí*, [online prezentace], [cit. 22.4.2017], Dostupné z: https://cent.felk.cvut.cz/predmety/39PHA/data/prednasky/04-kolize.pdf

[51] *Colliders*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 23.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/CollidersOverview.html

[52] *Setting Up Collisions With Static Meshes*, [online dokumentace], [cit. 23.4.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/HowTo/SettingCollision/index.html

[53] *Layer-based collision detection*, [online dokumentace], Publication 5.6-001G 29.3.2017 [cit. 22.4.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/Manual/LayerBasedCollision.html

[54] GOLDING, James, *Collision Filtering*, [online blog], 17.4.2014 [cit. 8.5.2017], Dostupné z: https://www.unrealengine.com/blog/collision-filtering

[55] *Collision Overview*, [online dokumentace], [cit. 8.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Physics/Collision/Overview/index.html

[56] *Vertex color*, [online], naposledy upraveno 7.12.2016 [cit. 17.5.2017], Dostupné z: http://wiki.polycount.com/wiki/Vertex_color

[57] OWENS, Brent, *Forward Rendering vs. Deferred Rendering*, [online], 28.10.2013 [cit. 17.5.2017], Dostupné z: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

[58] *Forward Shading Renderer for VR*, [online dokumentace], [cit. 17.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Performance/ForwardRenderer/

[59] *Taking Screenshots*, [online dokumentace], [cit. 6.5.2017], Dostupné z: https://docs.unrealengine.com/latest/INT/Engine/Basics/Screenshots/

[60] *Application.CaptureScreenshot*, [online dokumentace], Publication: 5.6-001J 5.5.2017 [cit. 6.5.2017], Dostupné z: https://docs.unity3d.com/560/Documentation/ScriptReference/Application.CaptureScreenshot.html

[61] *Mixamo.com*, [online], [cit. 9.5.2017], Dostupné z: https://www.mixamo.com/

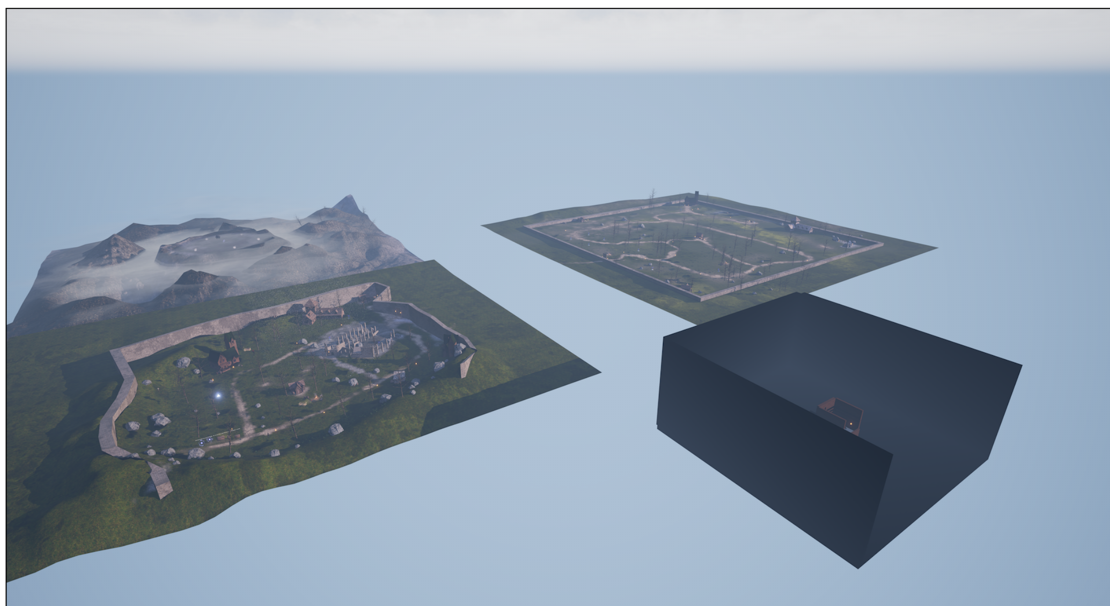
[62] *ShaderForge*, [online], [cit. 9.5.2017], Dostupné z: http://acegikmo.com/shaderforge/

[63] *Ultimate Game Music Collection*, [online], [cit. 9.5.2017], Dostupné z: http://www.johnleonardfrench.co.uk/ultimate-game-music-collection/

[64] *Textures.com*, [online], [cit. 9.5.2017], Dostupné z: https://www.textures.com/

A. Snímky z her

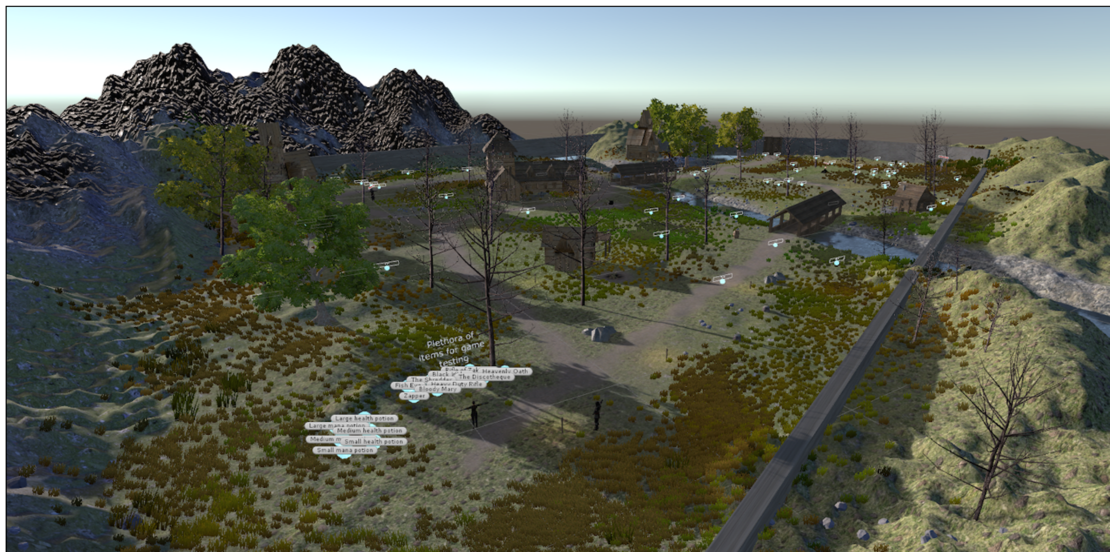
A.1 Vytvořené úrovně



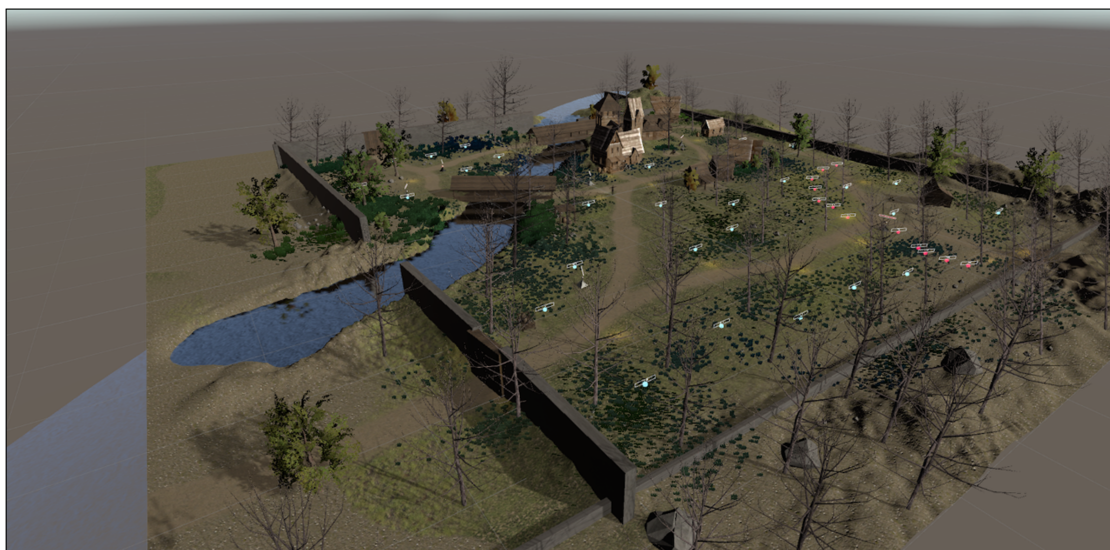
Obrázek A.1: Všechny úrovně v UE z dáli. Vpravo dole si můžete všimnout černého mnohostěnu, jež tvoří pozadí interiérní scény.

Obrázek A.2: Detail úrovně interiéru. Zde je zajímavé použití neviditelných zdí, které vrhají stíny během hry (Hidden Shadow). Zdi jsou použity z důvodu vzdálenosti kamery od hráče, která by vystupovala z místnosti.

Obrázek A.3: Bližší pohled na první úroveň ve hře v UE.



Obrázek A.4: První úroveň vytvořená v Unity. Poznámka: Vzdálenost vykreslování stromů a trávy byla zvýšena pro tvorbu fotografie. Vzdálenost je ve finální hře mnohem nižší, neboť stromy a trávu není možné vidět díky umístění kamery.



Obrázek A.5: Druhá úroveň vytvořená v Unity.

A.2 Snímky ze hry v Unity

Obrázek A.6: Boj s úhlavním nepřítelem v temné scéně.

Obrázek A.7: Snímek ze hry v Unity.

Obrázek A.8: Možnost hráče sebrat klíč, který vypadl z příšery. Mlha v této oblasti je vytvořena pomocí částicového systému.

Obrázek A.9: Dialogové okno pro rozhovor s NPC.

A.3 Snímky ze hry v UE

Obrázek A.10: Temný interiér. Hráč zde může sebrat knihu ze stolu.

Obrázek A.11: Truhla s předměty.

Obrázek A.12: Souboj s úhlavním nepřítelem hry a jeho speciální útok.

Obrázek A.13: Snímek ze hry demonstrující částicové systémy střelby zbraně a magické střílny.

Obrázek A.14: Noční venkovní scéna v UE.

(a) Neaktivní portál.

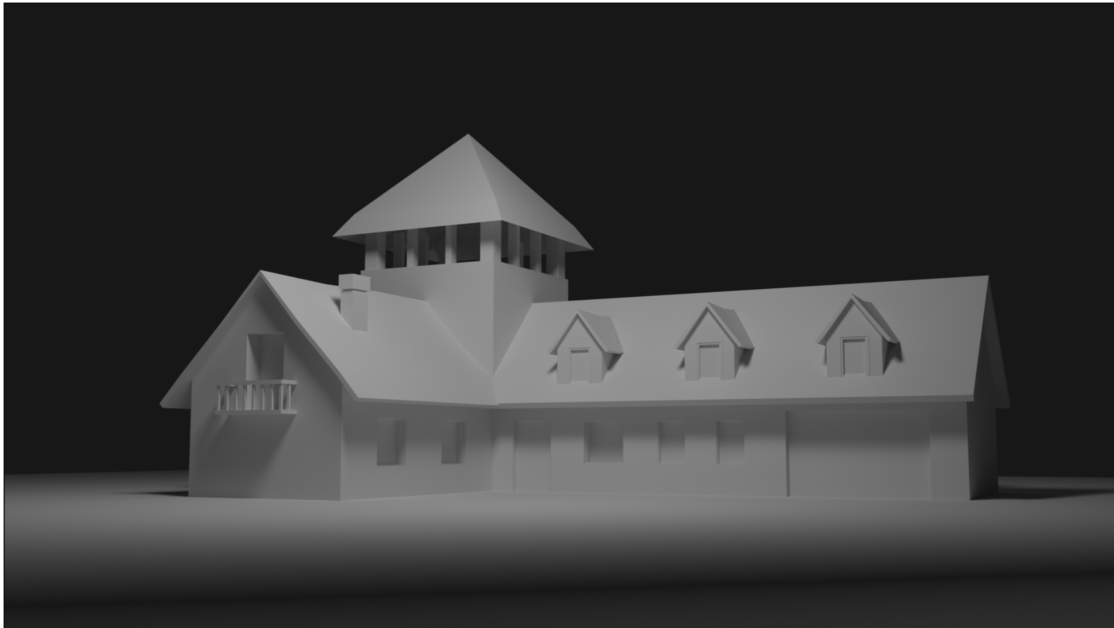
(b) Aktivní portál, jehož intenzita modulu světla v částicovém systému byla zdvojnásobena pomocí dynamického parametru.

Obrázek A.15: Částicový systém portálu v UE, jež používá dynamické parametry pro změnu vzhledu během hry.

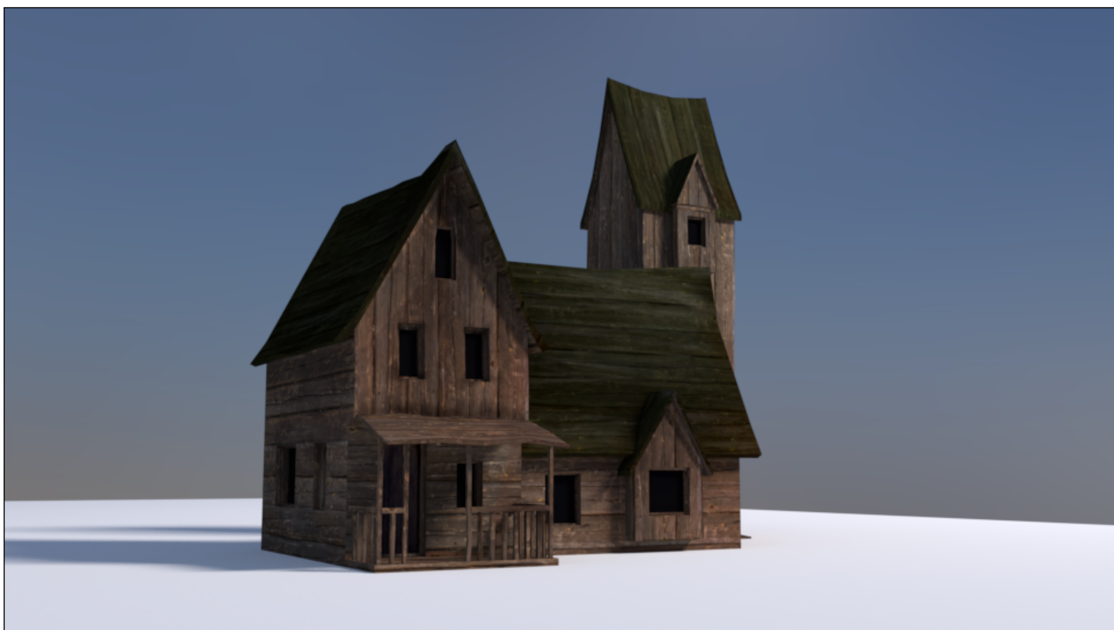
B. Vytvořené modely

Ukázka pár modelů vytvořených pro hru. Modely byly vytvořeny v Blenderu, materiály a textury však byly přiděleny v enginech, neboť má každý vlastní materiálový systém a materiály z Blenderu nejsou přenositelné. Rendery modelů v Blenderu jsou ve většině případech bez textur.

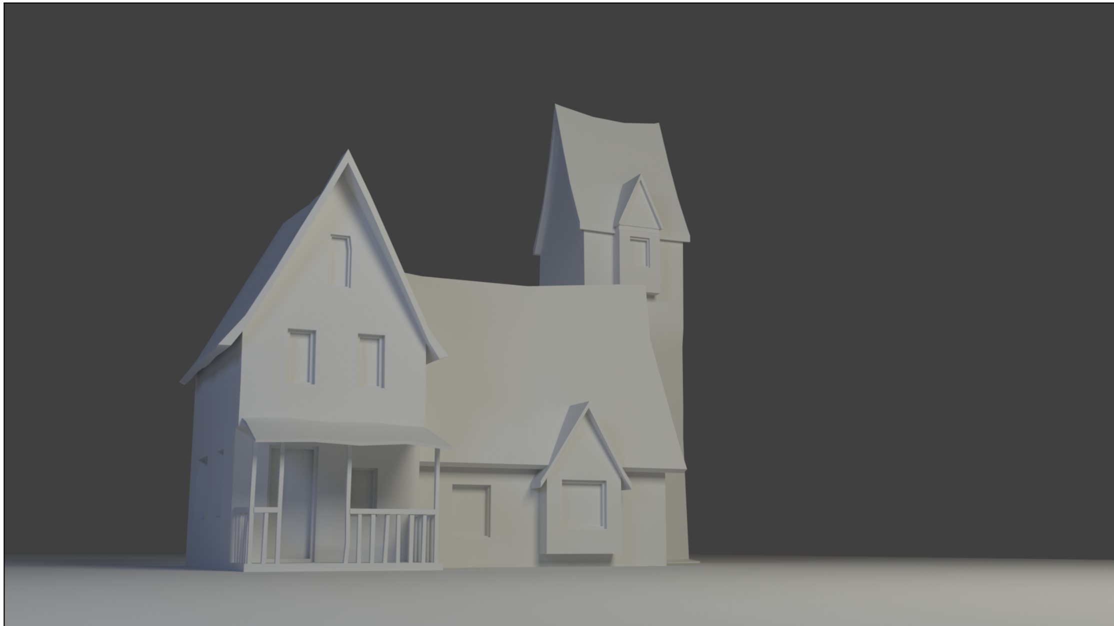
Obrázek B.1: Render vozíku v Blenderu.



Obrázek B.2: Render vesnického domu v Blenderu.



Obrázek B.3: Render domu s věží s testovacím materiálem v Blenderu.



Obrázek B.4: Render domu s věží v Blenderu.

Obrázek B.5: Kompozice většiny vlastních modelů s přiřazenými materiály v UE.

Obrázek B.6: Vlastní modely zbraní a dalších předmětů v UE. Všimněte si použitých materiálů různého charakteru (viz obrázek 5.1 pro implementaci jednoho z nich).

C. Seznam použitých zkratek

UE Unreal Engine

RPG Role-Playing Game

PHA Počítačové hry a animace

HLSL High-Level Shader Language

OOP Object-Oriented Programming

API Application Programming Interface

UHT Unreal Header Tool

RAM Random-Access Memory

SSD Solid-State Drive

PBR Physically-Based Rendering

LOD Level of Detail

CSG Constructive Solid Geometry

UI User Interface

UMG Unreal Motion Graphics

MIC Material Instance Constant

MID Material Instance Dynamic

GPU Graphics Processing Unit

FBX Filmbox

AI Artificial Intelligence

ID Identifier

JSON JavaScript Object Notation

CSV Comma-Separated Values

NPC Non-Playable Character

OBB Oriented Bounding Box

k-DOP k-dimensional Discrete Oriented Polytope

HRSST High Resolution Screenshot Tool

HDR High Dynamic Range

D. Obsah přiloženého DVD

Obsahem přiloženého disku je:

- ProjectCloak_Build_UE - adresář spustitelné hry v UE
- ProjectCloak_Build_Unity - adresář spustitelné hry v Unity
- ProjectCloak_Source_UE - adresář zdrojových souborů hry v UE
- ProjectCloak_Source_Unity - adresář zdrojových souborů hry v Unity
- Controls.pdf - PDF soubor popisující ovládání her
- Thesis_Source - adresář obsahující zdrojové soubory práce v LaTeXu
- Thesis_Cap_Martin.pdf - text práce v PDF