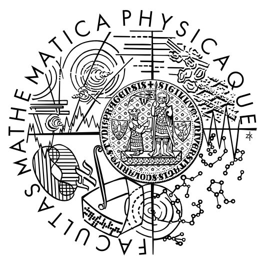


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Rastislav Špakovský

Medieval Adventures:Beyond Unknown – 2D príbehová hra s 3D grafikou

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.
Studijní program: Informatika
Studijní obor: Programování a softwarové systémy

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne ....

Podpis autora

Moje poďakovanie patrí najmä mojej rodine a kamarátom, ktorí mi boli oporou počas ťažkých čias, ktoré trápili posledný rok celú našu planétu. Moja nesmierna vďaka patrí aj môjmu vedúcemu práce, Mgr. Pavlovi Ježkovi, že si na mňa stále našiel čas a posúval ma pri tvorbe práce správnym smerom. Bez týchto ľudí by sa mi nikdy nepodarilo túto prácu dotiahnuť do konca.

Názov práce: Medieval Adventures:Beyond Unknown – 2D príbehová hra s 3D grafikou

Autor: Rastislav Špakovský

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Množstvo hráčov je fanúšikom dobrodružných príbehových hier. V práci takúto hru nielen tvoríme, ale ide nám aj o rozšírenie editora, ktoré by uľahčilo prácu na dizajne a rozširovaní hry. Pracujeme v hernom engine Unity. Samotným výsledkom našej práce je 2D dobrodružná príbehová hra s 3D grafikou, štruktúrovaná do levelov. V hre hráč ovláda hlavného hrdinu a počas hry musí využívať rôzne predmety, zbrane a vybavenie na boj s nepriateľmi a interakciu s rôznymi postavami. Na konci každého levelu sa nachádza špeciálny boss. Príbeh mu je podávaný formou questov, ktoré dostáva od rôznych postáv v leveloch a dialógov, ktoré s nimi vedie. Rozšírili sme editor Unity tak, aby sme zjednodušili vytváranie nových levelov a prácu na nich. Vytvorili sme nástroje na pridávanie nepriateľov, predmetov a ďalších postáv do hry a ich editáciu. Dizajnéri môžu na vytváranie a úpravu questov a dialógov v hre použiť grafové editory.

Kľúčové slová: príbehová hra Unity dobrodružná hra editor

Title: Medieval Adventures:Beyond Unknown – 2D story game with 3D graphics

Author: Rastislav Špakovský

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: A number of players are fans of adventure story games. In our thesis, we are trying to not only create such a game, but we are also focusing on extending the editor to make future work on both the design and the implementation of our game easier. We are using game engine Unity. The result of our work is a 2D adventure story game with 3D graphics, divided into levels. In the game, player controls the main character and throughout the game he must use different items, weapons and equipment to fight enemies and interact with other characters. At the end of each level there is a special boss. The story of the game is told in a form of quests, given to him by characters in the game and in a form of dialogues with those characters. We implemented extensions for the Unity editor to make creation of new levels and the work on them easier. We provided tools for adding enemies, items and other characters to the levels and editing them. Designers can use graph editors for creating and editing quests and dialogues in the game.

Keywords: story game Unity adventure game editor

Obsah

1 Úvod **3**
1.1 Výber žánru . . . 3
1.2 Single-player hra . . . 4
1.3 Editor . . . 5
1.4 Cieľ práce . . . 5

2 Špecifikácia zadania **6**
2.1 Štruktúra hry . . . 6
2.2 Predmety . . . 6
2.3 Nepriatelia . . . 7
2.4 Súboj s nepriateľmi . . . 8
2.5 Questy . . . 8
2.6 Dialógy . . . 9
2.7 Ovládanie . . . 9
2.8 Zvuk . . . 9
2.9 Ukladanie a načítavanie . . . 10
2.10 Editor . . . 10
2.11 Cieľová platforma . . . 10

3 Analýza **11**
3.1 Herný engine . . . 11
3.1.1 Unity . . . 12
3.1.2 Unreal Engine . . . 12
3.1.3 Rozhodnutie a výber herného enginu . . . 12
3.2 Priblíženie Unity . . . 13
3.3 Štruktúra hry . . . 13
3.3.1 Levely . . . 13
3.4 Ovládanie postavy . . . 15
3.5 Predmety . . . 16
3.6 Inventár . . . 18
3.7 Kúzla . . . 20
3.8 Nepriatelia . . . 20
3.9 Systém života a many . . . 22
3.9.1 Udalosti . . . 23
3.10 Questy . . . 23
3.10.1 Štruktúra questov . . . 23
3.10.2 Editor pre questy . . . 26
3.11 Dialógy . . . 27
3.11.1 Štruktúra dialógov . . . 27
3.11.2 Implementácia dialógov . . . 27
3.11.3 Editor . . . 28
3.12 Ukladanie a načítavanie . . . 29
3.12.1 Formát ukladaných dát . . . 29
3.12.2 Výber prístupu ku ukladaniu . . . 32
3.12.3 Implementácia ukladania a načítavania . . . 32

4 Projektová dokumentácia 34
4.1 Štruktúra 34
4.2 MainMenu scéna 34
4.3 Level scéna a Base scéna 35
4.4 Skripty a kód 38
4.4.1 Character skripty 39
4.4.2 Controls 40
4.4.3 Skripty a kód implementujúci dialógový systém 41
4.4.4 Skripty a kód implementujúci questový systém 41
4.4.5 Skripty implementujúce nepriateľov 42
4.4.6 Implementácia systému predmetov 44
4.4.7 Inventárový systém 45
4.4.8 Systém ukladania a načítavania 46
4.5 Rozširovanie editora 47
4.5.1 Grafové editory 48

5 Užívateľská dokumentácia 50
5.1 Dokumentácia pre hráča 50
5.2 Dokumentácia pre dizajnéra 53
5.2.1 Level 53
5.2.2 Mapa 53
5.2.3 Prechody medzi levelami 56
5.3 Predmety 57
5.4 Nepriatelia 58
5.4.1 Úprava skeleton nepriateľa 59
5.5 Dialógy 59
5.5.1 Vytvorenie postavy 59
5.5.2 Vytvorenie dialógu 60
5.5.3 Úprava dialógu 61
5.6 Questy 65
5.6.1 Tvorba questu 65
5.6.2 Úprava questu 65
5.7 Ukladanie a načítavanie 68

Záver 70
5.8 Implementácia hry 70
5.9 Rozšírenie editora 70
5.10 Ohodnotenie práce 71

Seznam použité literatury 72

Zoznam obrázkov 75

A Prílohy 77
A.1 Unity 77
A.1.1 Scéna 77
A.1.2 GameObject 77
A.1.3 Inšpektor 77
A.2 Assety 78

1. Úvod

Počítačové hry sú jedným z najrozšírenejších a najobľúbenejších typov zábavy pre ľudí po celom svete. Hry možno rozdeliť podľa toho, koľko hráčov ju súčasne hraje (single-player, multi-player) a taktiež podľa žánru a zamerania hry (akčné, dobrodružné, strategické, hrané vo virtuálnej realite a mnohé ďalšie). Práve široké možnosti, ktoré tvorba počítačovej hry ponúka boli dôvodom, prečo sa ju autor rozhodol mať ako zameranie svojej práce.

1.1 Výber žánru

Prvý krok, ktorý bol potrebný bol výber žánru. Štýl hry sme si vyberali podľa dvoch hlavných kritérií, ktoré sme si stanovili:

- dôraz na funkcionalitu,
- rozšíriteľnosť.

Pod dôrazom na funkcionalitu si autor predstavoval to, že pri vývoji hry sa viac zameriava na programovaciu časť ako na tú dizajnérsku. Ako príklad hry, ktorý by toto kritérium nesplňoval si môžeme povedať napríklad FPS (First-person shooter) hru s kampaňou. Veľkú časť práce na takejto hre by predstavoval dizajn levelu, pod čím si môžeme predstaviť návrh miestností, budov a prostredia, v ktorých sa bude hráč pohybovať. Autor práce sa však chcel v práci zamerať hlavne na to ako hra funguje, jej implementáciu. Vďaka tomu by sa mohol popri samotnom vývoji hry zamerať aj na pridanie možností potenciálnym dizajnérom, ktoré by im umožnili pracovať na rozširovaní hry, v podobe pridávania a stavby levelov, bez znalosti kódu a programovacích zručností.

Pod rozšíriteľnosťou si autor práce predstavoval najmä to, aby nás žáner hry neobmedzoval v pridávaní rozšírení do hry — pri spomínaných FPS hrách máme možnosti hlavne v pridávaní nových typov zbraní[1], ale napríklad pri vývoji RPG hry nemáme až tak zviazané ruky, keďže ide o univerzálnejší žáner[2].

Po stanovení a zvážení hlavných kritérií sme sa zamerali na tri typy hier, ktorým sa budeme venovať vo vlastných sekciách.

Autochess/autobattles hry

Autochess/autobattles[3] sú strategické, ťahovo založené hry, kde si hráči počas ťahu kupujú jednotky, ktoré môžu následne kombinovať, vylepšovať a rozostavovať po hernom poli. Po ukončení ťahu nasleduje bitka medzi dvojicou hráčov, počas ktorej už ale hráč jednotky neovláda, tie bojujú samé a ich správanie je automatické. Najznámejšie hry tohto typu sú *Teamfight Tactics* od spoločnosti Riot Games či *Hearthstone Battlegrounds* od spoločnosti Blizzard. Nad týmto typom hry sme uvažovali najmä z dôvodu širokého množstva mechaník, ktoré môžu byť v hre prostredníctvom jednotlivých jednotiek implementované.

Strategické hry na mape

Ďalším typom strategickej hry, ktorá je rozdelená do ťahov jednotlivých hráčov sú hry na štýl *Heroes of Might and Magic*. Hráč ovláda istú časť na mape, budovy a jednotky a počas ťahov ich vylepšuje, presúva, bojuje s nepriateľmi a obsadzuje nepriateľské územia. Hra ponúka mnoho oblastí, na ktoré je možné sa počas vývoja zamerať. Rôzne typy jednotiek, budov, spôsobov boja, ale taktiež interakcie s budovami, neutrálnymi stavbami, nepriateľmi a mapou.

Dobrodružná hra

Treťou možnosťou, nad ktorou sme premýšľali bola príbehová dobrodružná hra, s prvkami RPG. Jeden z najobľúbenejších a najrozšírenejších žánrov v oblasti single-player hier, kde hráč prevezme úlohu hlavného hrdinu a počas hry spoznáva svet, nepriateľov, komunikuje s inými postavami a plní rôzne úlohy. Príkladom sú série ako *The Elder Scrolls*, *Fallout* a mnohé ďalšie.

Výber dobrodružnej hry

Po zvážení kritérií sme sa nakoniec rozhodli pokračovať s možnosťou dobrodružnej príbehovej hry. Tento žáner nám prišiel najzaujímavejší, keďže ponúka mnoho možností v tom, aké mechaniky a funkcie v hre implementovať.

1.2 Single-player hra

Ďalší krok, ktorý nás čakal, bolo rozhodnutie, či budeme pripravovať hru pre jedného alebo viacerých hráčov. Tento žáner hier sa väčšinou vyvíja ako single-player a pre tento spôsob sme sa rozhodli aj my. Multi-player hry sú podľa nás vhodnejšie pre žánre hier, kde proti sebe hráči súperia — napríklad vyššie spomínané autobattles hry. Keďže autor pracoval na hre sám, sťažilo by sa mu samotné testovanie hry a v konečnej verzii by bola multi-player verzia hry oproti single-player okresanejšia. Najväčším dôvodom však bolo to, že sa tento žáner jednoducho hodí pre jedného hráča.

Rozhodnutie implementovať hru 2D

Veľkým rozhodnutím, ktoré sme urobili už na začiatku bolo to, že sme na hre pracovali v 2D, čo sa týka hernej logiky a samotnú grafiku hry sme nechali v 3D. Kombinujeme teda obidva spôsoby, čoho výsledkom je hra, ktorá sa niekedy označuje ako 2.5D[4]. Rozhodnutie sme si odôvodnili tým, že ide vlastne presne o to, čo sme od našej hry chceli. Po prvé, presunutie hernej logiky do 2D neznížilo náročnosť programovacej časti. Väčšina vecí v hre by fungovala rovnako aj v čisto 3D verzii — dialógový systém, questový systém alebo systém predmetov by fungovali a boli implementované v obidvoch verziách úplne rovnako. Najvýraznejšie sa nám ovplyvnil pohyb objektov v hre, na čo sa však môžeme pozerať iným spôsob — tým, že sme nemuseli riešiť tretiu dimenziu, mohli sme si niektoré veci implementovať sami. Po druhé, nemuseli sme venovať toľko času vytváraniu prostredia, v ktorom sa hra odohráva. Namiesto toho sme sa rozhodli do projektu

pridať možnosti, pomocou ktorých môže potenciálny dizajnér nadizajnovať hru sám bez znalosti programovania – rozšírili sme editor, v ktorom sme hru pripravovali. Príkladom podobných titulov môže byť napríklad séria *Prince of Persia* alebo mobilná hra *Swordigo* na obrázku 1.1

Obr. 1.1: Swordigo[5]

1.3 Editor

Už sme teda viackrát spomínali to, že nám myšlienka rozšírenia editora pre hru prišla veľmi zaujímavá. Umožnila nám samotný vývoj hry rozdeliť na dve časti, programovaciu a dizajnérsku s tým, že na dizajne hry (príbeh, úlohy) by sa dalo pracovať s čo najmenšou znalosťou kódu. Chceli sme tým teda uľahčiť potenciálnym dizajnérom prácu na tvorbe hry. Nám to taktiež umožnilo sa na dizajn pozerať inak a implementovať tieto rozšírenia editora z kódu.

1.4 Cieľ práce

Hrubým cieľom práce teda bolo pripraviť 2D dobrodružnú príbehovú hru s prvkami RPG, ale taktiež rozšírenie editora s účelom sprístupnenia a zjednodušenia práce na hre ľuďom, ktorý by o internej implementácii potrebovali vedieť čo najmenej. V nasledujúcej kapitole sa nachádza presná špecifikácia — čo presne sme od hry a rozšírenia editora požadovali.

2. Špecifikácia zadania

Po výbere hry sme mohli začať s jej konkretizáciou. V tejto kapitole teda rozoberáme to, čo presne od našej hry očakávame a čo v nej chceme mať.

2.1 Štruktúra hry

Hra bude štruktúrovaná do levelov. Každý level bude jedinečný, po splnení úloh a porazení nepriateľov hráč level dokončí a posunie sa do ďalšieho. Hráčov progres sa bude uchovávať, teda všetko to, čo v získal v predchádzajúcich leveloch bude schopný využívať v novom — levely budú na seba nadväzovať.

V editore bude možné pridať do hry nový level a upravovať ho pomocou mnohých rozšírení. Tie však silno závisia od toho o akom objekte sa rozprávame a z toho dôvodu budeme podrobnejšie o editore rozprávať v jednotlivých sekciách.

V odovzdávanej hre budú k dispozícii aspoň dva úplne dokončené levely a taktiež tutoriál, ktorý hráča naučí základy hry.

2.2 Predmety

V hre bude implementovaný rozšíriteľný systém predmetov. V hre by sme chceli mať aspoň tieto tri typy predmetov:

- statické,
- použiteľné,
- nasaditeľné.

Pod pojmom statické predmety si môžeme predstaviť napríklad kľúče. V hre by sme mohli mať napríklad bránu, ktorá by sa otvorila a hráča pustila ďalej len v takom prípade, že hráč vlastní potrebný kľúč. Použiteľné predmety by mohol hráč spotrebovať za účelom získania nejakého bonusu — napríklad elixír, ktorý by obnovil hráčove zdravie. Nasaditeľné predmety sú zbrane a hráčove „oblečenie“ — prilba, brnenie, topánky. Hráč by si ich mohol nasadiť — zbrane by mu umožnili útočiť a poškodzovať nepriateľov, nasadené brnenie by mu pasívne zvyšovalo odolnosť proti nepriateľom.

Predmety by mal hráč mať možnosť získavať rôznymi spôsobmi — za splnenie úloh, zdvihnutím predmetu zo zeme alebo ako dar od inej postavy v hre. Malo by byť jednoduché do hry pridať predmet nový alebo upraviť existujúci. U predmetov, ktoré sa zdvíhajú, by mal mať dizajnér možnosť ich do levelov pridávať rýchlo, nemalo by byť potrebné nič nastavovať.

Inventár

Hráč bude mať predmety uložené v inventári, ale už z toho, aké predmety by sme chceli v hre mať vyplýva, že budeme inventáre potrebovať aspoň dva - jeden

normálny a jeden, ktorý bude obsahovať hráčove nasadené predmety (hráčove vybavenie). To znamená, že budeme musieť byť schopní obmedziť to, aké predmety bude možné v jednotlivých inventároch nosiť.

Čo sa týka grafického užívateľského rozhrania pre hráča , budeme chcieť dosiahnuť toho, aby mohol hráč s predmetmi v inventári pracovať prirodzeným spôsobom — predmety sa budú zobrazovať v podobe ikoniek, premiestňovať pomocou ťahania, používať (použiteľné predmety, o ktorých sme písali pár riadkov vyššie) pomocou kliknutia na ikonku predmetu a nasadiť pomocou toho, že predmet pretiahne z jedného inventára do druhého. Na obrázku 2.1 môžeme vidieť podobný prístup. Časť 1 a 2 predstavuje vybavenie, časti 3 a 4 „normálny“ inventár.

Obr. 2.1: Inventár v Minecrafte[6]

2.3 Nepriatelia

V hre budeme tiež chcieť mať nepriateľov, ktorých budeme rozdeľovať do dvoch základných skupín a to konkrétne na:

- bežných nepriateľov,
- bossov.

Bežní nepriatelia

Pod pojmom bežní nepriatelia má autor na mysli postavy, ktoré môže hráč stretnúť počas prechádzania levelov a môžu byť súčasťou rôznych úloh, ktoré hráč počas prechádzania levelom dostáva. Z implementačnej časti budeme od bežných nepriateľov požadovať to, aby využívali nejaký všeobecnejší systém — stavový automat, ktorý bude možné využiť aj pri tvorbe budúcich nepriateľov.

Z editora by malo byť možné takýchto nepriateľov do jednotlivých levelov pridávať jednoducho pomocou tlačidla z nejakého menu. Po takomto vložení nepriateľa by mal nepriateľ pri zapnutí hry fungovať a správne reagovať, bez ďalšieho potrebného nastavovania.

Bossovia

Bossovia budú špeciálni nepriatelia, ktorí sa budú nachádzať na konci levelov. Pred súboj s bossom bude *cutscéna* (animácia), boj s bossom bude zaujímavejší, mal by mať viacero fází a správania bossa by sa počas boja malo meniť. Porazením bossa sa hráč presunie do ďalšieho levelu.

2.4 Súboj s nepriateľmi

V hre bude mať hráč dva spôsoby ako bojovať proti nepriateľom:

- zbrane,
- kúzla.

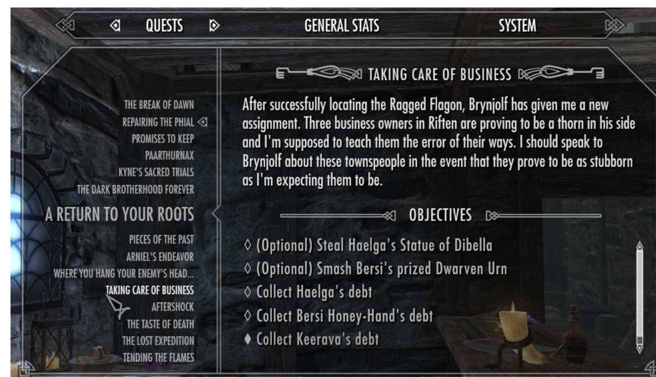
O zbraniach sme už písali v sekcii Predmety a kúzla si môžeme predstaviť ako podobný koncept. Hráč bude mať možnosti sa počas hry naučiť kúzla, ktoré bude potom môcť ďalej využívať. Pod kúzlom si môžeme predstaviť zoslanie ohnivej gule, ktorá poškodí nepriateľov alebo efekt, ktorý nás vylieči od poškodenia, ktoré sme dostali. Naučené kúzla, rovnako ako získané predmety, hráčovi ostanú do ďalších levelov. V hre budeme mať systém, do ktorého bude jednoduché kúzla pridávať.

Taktiež budeme mať v hre, rovnako ako pre predmety, užívateľské rozhranie pre prácu s kúzlami. Hráč si bude schopný zobraziť všetky jeho naučené kúzla a vyberať, ktoré kúzlo bude „nasadené“. Nasadené kúzlo bude to, ktoré sa zošle, keď hráč stlačí klávesu na kúzlenie. Budeme teda chcieť mať špeciálny „inventár“ pre kúzla.

2.5 Questy

Už sme písali o tom, že v hrách tohto žánru sa využíva koncept úloh, ktoré hráč musí plniť. V našej práci ich budeme nazývať questy, ktoré budeme deliť na menšie časti a týmto častiam budeme hovoriť úlohy. Questy budú v hre hrať významnú úlohu, spoločne s dialógmi, o ktorých budeme písať v nasledujúcej kapitole, budú hráčovi približovať príbeh a nahrádzať funkciu rozprávača. U úloh budeme chcieť mať k dispozícii viacero druhov — poraziť nepriateľov, pozbierať predmety, dostať sa na konkrétne miesto alebo sa porozprávať s nejakou postavou, doniesť postave nejaký predmet alebo nejakú postavu presvedčiť. Systém budeme vytvárať s ohľadom na budúcnosť — rozumne by sa mali dať pridávať nové druhy úloh. Jednotlivé úlohy v rámci konkrétneho questu budeme chcieť štruktúrovať a prepájať. Pod tým si môžeme predstaviť quest, v ktorom máme presné poradie v ktorom sa budú úlohy plniť — s postavou B je potrebné hovoriť až po dialógu s postavou A. Taktiež bude v hre vypracované užívateľské rozhranie, v ktorom

bude môcť hráč vidieť jeho questy a bude schopný si o nich vypísať podrobnosti — ktoré úlohy je potrebné práve splniť. Na obrázku 2.2 môžeme vidieť podobný systém v Skyrime, kde sa questy delia na menšie časti, ktoré hráč postupne plní.



Obr. 2.2: Questy v Skyrime[7]

Po splnení úloh sa splní celý quest a hráč získa odmenu. U nich budeme požadovať to, aby sme mohli do hry pridávať nové typy týchto odmien.

Dizajnérom ponúkneme grafový editor na vytváranie a úpravu týchto questov. Keďže ide o jeden z dvoch hlavných spôsobov, pomocou ktorých je v našej hre vytváraný dej, bude dizajnér schopný priamo v editore pridávať úlohy, prepájať ich a vytvárať odmeny.

2.6 Dialógy

Pod dialógmi si predstavujeme to, že hráč by mal možnosť sa porozprávať s inými postavami v hre. Takéto postavy budú statické – nebudú sa po leveli pohybovať, budú stále stáť na jednom mieste. Dialóg sa bude deliť na prehovory týchto postáv s tým, že hráč bude mať možnosť postave odpovedať. Taktiež implementujeme systém *dialógových akcií*. Dialógová akcia bude niečo, čo sa vykoná, ak si danú odpoveď hráč zvolí, napríklad prijatie questu od postavy alebo získanie predmetu od postavy.

Opäť vytvoríme grafový editor, v ktorom bude možné celý dialóg vytvoriť — pridávať prehovory, nastavovať odpovede a priraďovať ku nim dialógové akcie.

2.7 Ovládanie

Taktiež dáme hráčovi možnosť, aby si nastavil ovládanie.

2.8 Zvuk

Autor práce si uvedomuje, že zvuk je veľmi dôležitá súčasť hier. U mnohých hier je práve zvuková stopa to, na čom stavia propagácia hry. V kontexte tejto práce sa však rozhodol do hry zvuk nepridávať. Jeho rozhodnutie ovplyvnilo to, že s prácou so zvukom nemal predchádzajúce skúsenosti a čas, ktorý by musel stráviť získavaním vhodných zvukových stôp by radšej venoval práci na nových funkciách

alebo editore. Ide však asi o najpodstatnejšie rozšírenie v prípade budúcej práce na hre.

2.9 Ukladanie a načítavanie

Hráč bude schopný svoj postup hrou ukladať. Najdôležitejším prvkom tohto systému bude rozšíriteľnosť — jeho návrh bude stáť na tom, aby pridávanie nových herných prvkov a ich integrácia do tohto systému (ich správne ukladanie) bola čo najjednoduchšia.

2.10 Editor

Ako sme teda už spomínali v kapitole Úvod, ale aj v predchádzajúcich sekciách tejto kapitoly, súčasťou odovzdaného projektu bude rozšírenie editora, pomocou ktorého sa budeme snažiť oddeliť dizajnérsku časť od tej programovacej, čím by sa potenciálnym dizajnérom práca na hre uľahčila. V prípadoch, kde to dáva zmysel, (questy a dialógy) im bude ponúknutý grafový editor.

U vytvárania levelu si pod tým môžeme predstaviť, že dizajnérom ponúkneme nejakú základný level, ktorý bude obsahovať všetko potrebné na to, aby level mohol fungovať. Dizajnér potom môže pracovať a nový level stavať z tohto základu. U nepriateľov alebo predmetov by mal mať dizajnér možnosť ich pridávať do levelov prostredníctvom nejakého menu, kde by sa automaticky nastavili všetky referencie tak, aby sme s ním mohli ďalej normálne pracovať – nepriateľ sa bude správne ukladať a budeme ho môcť zaradiť do konkrétnych questových úloh.

Ďalším príkladom by mohla byť pohybujúca sa platforma, u ktorej dáva zmysel vykresliť v editore čiary, ktoré reprezentujú kam až sa platforma bude pohybovať. Podobný princíp by sa hodil napríklad aj u nepriateľov, ktorý hliadkujú nejakú oblasť a pod.

V podstate ide o čo najväčšie uľahčenie práce dizajnérom.

2.11 Cieľová platforma

Posledným dôležitým bodom špecifikácie je určenie platformy, pre ktorú bude hra vyvíjaná. Cieliť budeme pre PC a konkrétne operačný systém Windows.

3. Analýza

Po stanovení toho, čo v práci chceme a čo nechceme mať bolo potrebné sa rozhodnúť, akým spôsobom budeme jednotlivé veci implementovať. V nasledujúcich sekciách bude autor pojednávať hlavne o tom, aké mal možnosti a prečo a ako sa rozhodol jednotlivé časti implementovať daným spôsobom.

3.1 Herný engine

Prvým a jednoznačne najdôležitejším rozhodnutím, ktoré sme museli uskutočniť, bol výber herného enginu, v ktorom budeme hru vyvíjať. Pred touto prácou nemal autor žiadne predchádzajúce skúsenosti s vývojom takýchto počítačových hier. Z toho dôvodu sme sa rozhodli využiť nejaký z väčších a známejších herných enginov, ku ktorým existuje veľké množstvo materiálov — tutoriály, prehľadná dokumentácia, diskusné fóra alebo v danom engine. Taktiež sme sa rozhodli použiť engine, za ktorý by nebolo treba platiť.

Najdôležitejším faktorom bolo to, v akom programovacom jazyku sa v danom engine pracuje. Autor mal pred prácou skúsenosti najmä s tromi jazykmi:

- C#,
- C++,
- Java.

Jazyky sú zoradené zostupne podľa toho, s ktorým by chcel autor pracovať najradšej (preferujeme herný engine, v ktorom by sa pracovalo v C#).

Pred výberom enginu sme si tiež potrebovali určiť, čo všetko sme od engine očakávali. Keďže sme predchádzajúce skúsenosti s vývojom takejto hry nemali, bolo ťažké určiť, čo presne by nám engine mal ponúknuť. Mali sme teda len tri veľmi hrubé požiadavky:

- možnosť vytvorenia 3D hry (je potrebné si uvedomiť to, že aj keď logika hry bude nakoniec len v 2D, hra sa bude sama osebe odohrávať v 3D),
- možnosť práce s užívateľským rozhraním,
- rozšírenie editora.

Každý väčší engine naše kritéria v podstate spĺňal. Z toho dôvodu sme sa pozreli hlavne na dva, azda najpopulárnejšie[8]:

- *Unity*,
- *Unreal Engine*.

3.1.1 Unity

Unity je herný engine, v ktorom je od verzie 2017.1 hlavným programovacím jazykom C#. Taktiež je považovaný za engine, ktorý je vhodnejší pre menšie tímy vývojárov, čo nám vyhovovalo. Na internete sa nachádza veľké množstvo materiálov (tutoriály, fóra) a samotná dokumentácia pre herný engine je veľmi dobrá. Unity ponúka možnosti pre vývoj obrovského množstva rôznych typov hier a má asi najväčšiu komunitu zo všetkých herných enginov. Taktiež dáva vývojárom možnosť využiť *Unity Asset Store*, prostredníctvom ktorého majú vývojári možnosť veľmi jednoducho a rýchlo pridávať do hry rôzne modely, animácie, materiály ale aj kód, ktorý už pripravil niekto iný. Veľké množstvo týchto *assetov* (v jednoduchosti dáta pre hru) je dokonca zadarmo, vďaka čomu by sme mohli využívať pri tvorbe hry už existujúce modely pre hráčovu postavu, nepriateľov, predmety, zbrane a v podstate všetko, čo sa týka grafiky.

3.1.2 Unreal Engine

Druhou alternatívou, ktorá sa nám ponúkala bol *Unreal Engine*, herný engine od spoločnosti Epic Games. V Unreal Engine sa využíva kombinácia vizuálneho skriptovania (programovanie bez písania kódu) v podobe Blueprintov a programovacieho jazyka C++. Vizuálne skriptovanie je spôsob, ktorým môžu vývojári hry definovať triedy a ich funkcionalitu bez písania kódu.

Oproti Unity dáva Unreal Engine väčší dôraz na dizajn a grafické prvky, čo ho robí príťažlivejším pri vývoji takzvaných AAA herných titulov – hier, na vývoj ktorých pracuje veľké tímy s veľkým finančným kapitálom a sú určené pre veľké množstvo zákazníkov.

Podobne ako u Unity, aj o práci s Unreal Enginom sa nachádza na internete množstvo materiálov, nie však až v takom množstve ako pre Unity. Pre prístup k assetom je možné využiť Unreal Marketplace.

3.1.3 Rozhodnutie a výber herného enginu

Pre našu prácu boli teda obdiva enginy vyhovujúce. Vecou, ktorá nakoniec rozhodla bol programovací jazyk a taktiež materiály a informácie na internete. Predstava programovania na rozsiahlej práci bola autorovi bližšia v C#, keďže išlo o jazyk, s ktorým mal najväčšie skúsenosti. To, že sme s vývojom takejto hry nemali žiadne predchádzajúce skúsenosti tiež znamenalo, že pre nás naozaj obrovské množstvo prameňov, z ktorých by sme pri práci v Unity mohli čerpať prišlo veľmi vhod. Unity je obecne obľúbenejší herný engine pri vývoji hier v menších tímoch[9], čo bola aj autorova situácia.

Konečným rozhodnutím bolo teda využiť služby Unity. V práci Unity využívame vo verzii 2019.4f, keďže táto verzia bola v danom čase najstabilnejšia, existovalo k nej najväčšie množstvo aktuálnych materiálov a taktiež bola poslednou LTS (long-term support) verziou.

3.2 Priblíženie Unity

V nasledujúcich sekciách budeme prácu analyzovať v kontexte Unity — aké riešenia nám Unity ponúkalo, ako sa od seba líšili a pre aké sme sa nakoniec rozhodli. Z toho dôvodu sa v prílohe A.1 nachádza krátka sekcia o Unity.

3.3 Štruktúra hry

V prílohe A.1 je opísaná scéna, kde vlastne ide o kontajner `GameObjectov`.

Pri štruktúrovaní hry pomocou scén sme mali v Unity viacero možností ohľadom toho, akým spôsobom rozdeliť hru do scén, kde sme z kapitoly Špecifikácia zadania vedeli, že ju chceme deliť na levely. Ponúkali sa nám tri hlavné možnosti:

- jedna scéna pre jeden level,
- viacero scén pre jeden level,
- jedna scéna pre všetky levely.

Prvá možnosť bola očividná a ide asi o najpoužívanejší prístup pri vývoji hier delených do levelov. Prácu by nám to sprehľadnilo a prinieslo do projektu rozumnú štruktúru — (komponenty na `GameObjectoch` by sa mohli odkazovať len na `GameObjecty` z toho rovnakého levelu. Pre dizajnérsku časť by nám to umožnilo pridať do hry nejakú základnú scénu, ktorú by si dizajnér skopíroval a na kópii mohol pracovať ako na „jeho" leveli.

Využitie toho, že by sa level členil na viacero scén by mohlo dávať zmysel pri naozaj obrovských leveloch, kde by nám mohlo robiť problém celú takú scénu načítať naraz — samotné načítanie by trvalo dlho a stále by sme mali načítané veľké množstvo `GameObjectov`. To by sme potom mohli vyriešiť dynamickým načítavaním levelov. Štruktúra by už potom nemusela byť pekná, keďže by sme mohli mať rôzne množstvo scén pre viacero levelov a komponenty by sa nemohli medzi týmito scénami odkazovať (aspoň nie v editore).

Ďalšou možnosťou by mohlo byť to, že by sa celá hra nachádzala v jednej scéne (poprípade menu by malo vlastnú scénu). Tento prístup by nám zjednodušil mnohé problémy – napríklad prechod medzi dvoma levelmi by sme vyriešili presunutím postavy, ovládanej hráčom. Pri prístupe „1 scéna,1 level" by sme museli riešiť to, ktorá scéna je načítaná, ktorá sa má načítať a čakať kým sa nová scéna načíta. Z kritéria rozšíriteľnosti a prehľadnosti bol však tento prístup neakceptovateľný, keďže práca na dvoch rozdielnych leveloch by znamenala prácu na jednej scéne.

V hre teda budeme používať prístup, kde bude každý level reprezentovaný jednou scénou.

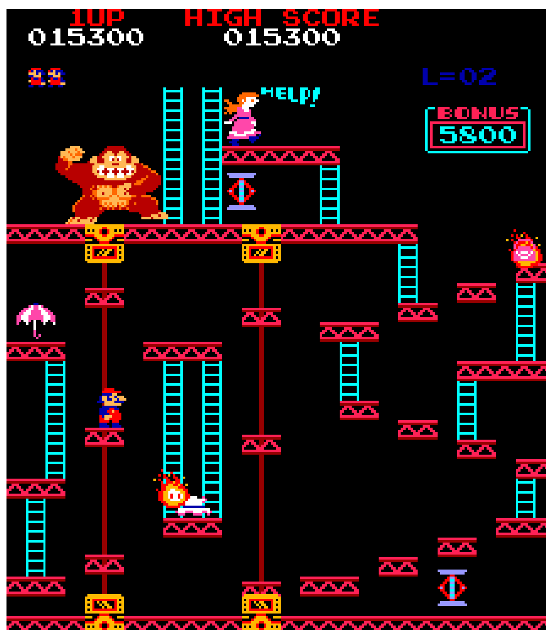
3.3.1 Levely

Pri tvorbe a dizajne jednotlivých levelov sme mali k dispozícii dve hlavné alternatív:

- dizajn levelov pomocou platforiem,
- využitie terénu.

Platformy

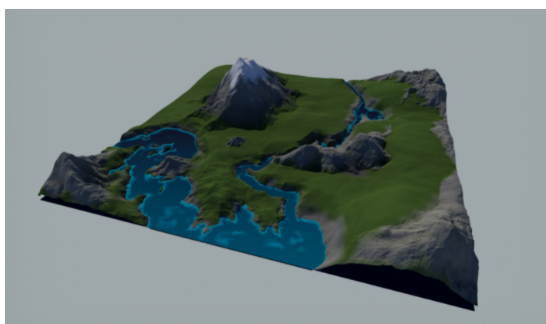
Prvý prístup, ktorý sa nám ponúkal je typický pre 2D hry. Levely v hrách ako *Super Mario*, *Donkey Kong* alebo napríklad Prince of Persia sú založené platformách a plošinách, čo môžeme vidieť na obrázku 3.1. Po levely sa rozmiestnia platformy a hráč sa phybuje primárne po nich.



Obr. 3.1: Donkey Kong[10]

Terén

Ďalší spôsob, ktorý nám Unity ponúka je využitie terénu. Terén možno vidieť na obrázku 3.2. Terén v podstate slúži na modelovanie krajiny. Využíva sa najmä v 3D hrách, na vytvorenie otvoreného sveta.



Obr. 3.2: Terén v Unity[11]

Práca s terénom a terén sám osebe má však niekoľko obmedzení. Tento systém totiž v Unity využíva výškovú mapu – každý bod má len jednu výšku. Naša hra sa však bude logicky odohrávať len v 2D, čo by znamenalo, že použitie iba samotného terénu by hráčov pohyb obmedzovalo až príliš – boli by sme v situácii, kde by sa len posúval zľava doprava po nejakej krivke.

Kombinácia terénu a platforiem

Rozumným ponúkaným riešením bolo tieto prístupy skombinovať. A to tak, že:

- hlavnú časť levelu vytvoríme pomocou terénu,
- zvyšok vhodne doplníme platformami.

Level bude teda stáť na teréne, ktorý nám aj potenciálnym dizajnérom dáva možnosť vytvoriť pekné prostredie s pozadím a na miestach, kde nás terén obmedzuje pridáme platformy, ktoré môžu byť statické alebo pohyblivé.

3.4 Ovládanie postavy

Ďalší bod, ktorý som si musel ujasniť bolo to, akým spôsobom bude v hre riešené ovládanie postavy a fyzika. Situáciu budeme mať trochu uľahčenú, keďže pohyb bude len v 2D, väčšina fyziky však bude rovnaká ako v podobnej hre v 3D.

Skutočný „tvar" `GameObjectov` pre fyzikálny engine v Unity reprezentuje ich `Collider`. Fyzický engine ďalej pracuje s týmito `Collidermi` a kolíziami medzi nimi. Dva najpoužívanejšie prístupy, ktoré sa v Unity využívajú keď potrebujeme ovládať pohyb nejakého `GameObject` sú:

- `Rigidbody[12]`,
- `CharacterController`.

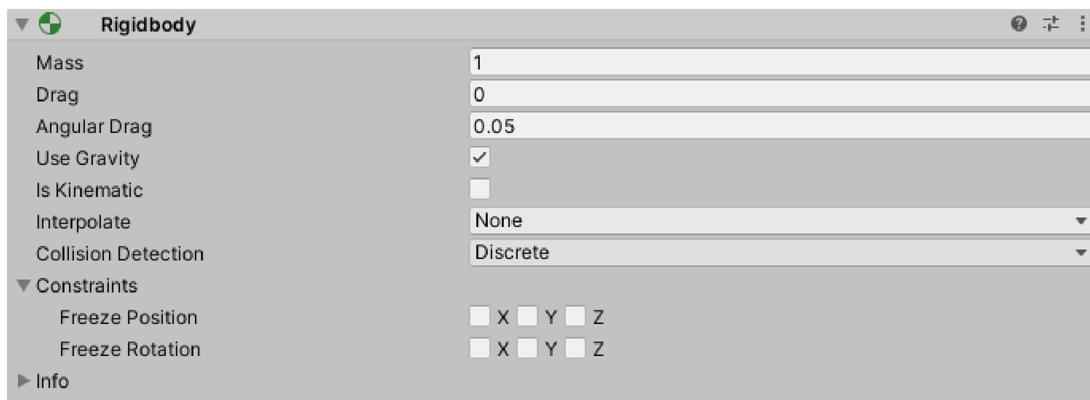
V oboch prípadoch ide o komponenty, ktoré nám ponúkajú rôzne metódy na ovládanie `GameObjectu`.

Rigidbody

`Rigidbody` používame, keď chceme využiť simuláciu fyziky priamo z Unity. Ihneď po pridaní `Rigidbody` na `GameObject` s `Colliderom` bude za nás Unity riešiť väčšinu potrebných vecí — bez akéhokoľvek ďalšieho kódu vieme `GameObjectu` priradiť hmotnosť a fungovať nám budú veci ako gravitácia, sily aplikované pri náraze alebo zrýchľovanie. Pridanie auta, ktoré sa správa realisticky vieme pomocou `Rigidbody` vyriešiť za zlomok času, ktorý by sme tomu museli venovať v prípade, že by sme `Rigidbody` nevyužili. Z kódu potom môžeme na `Rigidbody` aplikovať rôzne sily. Inšpektor pre tento komponent môžeme vidieť na obrázku 3.3

CharacterController

Druhá možnosť, ktorá sa nám v danom kontexte ponúka je využite `CharacterControlleru`. Ako už z jeho názvu vyplýva, využíva sa najmä pri simulovaní pohybu nejakých postáv. To je najmä z toho dôvodu, že pôsobenie fyzikálnych síl ako zrýchľovanie, gravitáciu a nárazy si musíme implementovať sami. To nám však v mnohých prípadoch vyhovuje, keďže živé objekty



Obr. 3.3: Rigidbody komponent v inšpektore

sa správajú inak ako statické, napríklad sa neprevrátia, keď stoja v strede kopca. **`CharacterController`** nám tiež dáva možnosti ako to, že vie prekonávať „schody", takže sa na malej prekážke nezasekne, ale „prekročí ju" — ako postava.

Využitie CharacterControlleru

Obidva prístupy sa využívajú v rozličných situáciách[13]. Na precíznu simuláciu fyziky sa využíva najmä **`Rigidbody`**, kde môžeme veľmi jednoducho a s malým množstvom kódovania nechať Unity nech rieši fyziku za nás. Hodí sa teda najmä pre neživé objekty ako lopta, keďže vieme nastaviť aj to, ako bude reagovať na rozdielne povrchy.

Na druhú stranu pre simuláciu postáv je vhodnejší **`CharacterController`**, keďže necháva logiku na nás a poskytuje nám len základy. Vždy teda vieme, ako bude v situácii postava reagovať, keďže sme si to implementovali sami. Pri stúpaní do kopca predsa nechceme, aby nám postava prevrátila a začala sa gúľať z kopca. Ak by sme to chceli dosiahnuť u **`Rigidbody`**, museli by sme povypínať množstvo funkcií a to, ako fyzika pre **`GameObject`** funguje by mohlo začať byť veľmi neprehľadné.

Rozhodli sme sa teda využiť **`CharacterController`**, čo znamenalo, že sme si väčšinu vecí museli implementovať sami.

3.5 Predmety

Už v kapitole Špecifikácia zadania sme rozoberali to, čo budeme od nášho systému požadovať:

- rôzne typy predmetov,
- možné pridávanie ďalších typov, rozšíriteľnosť,
- rôzne funkcie predmetov – použiteľné, nasaditeľné (zbrane, brnenie), kľúče,
- interakciu s predmetmi v hre – predmety rozmiestnené po leveli, získanie predmetu od inej postavy, za splnenie questu, predmety ako súčasť questových úloh.

Prvé tri body nás smerujú k objektovému návrhu, kde by sme využili dedičnosť. Mohli by sme mať základnú triedu (dávalo by zmysel ju mať aj abstraktnú) pre všetky predmety a jednotlivé typy predmetov by od nej dedili. Prvým riešením by teda mohlo byť využitie obyčajnej C# triedy.

Predmet ako bežná C# trieda

Pre každý typ predmetu by sme mali vlastnú triedu. S týmto prístupom by nám však vzniklo niekoľko nepríjemností. Prvou by bolo to, že ak by sa v jeden moment nachádzalo v leveli viacero predmetov jedného typu a pre každý z nich by sme vytvárali novú inštanciu, plytvali by sme pamäťou. Rôzne predmety toho istého typu by boli vždy rovnaké. To by sme síce mohli vyriešiť pomocou návrhového vzoru *Singleton*, no stále by nám ostávali problémy. Upravovať tieto predmety by sme mohli len priamo v kóde. Ak by sme chceli zmeniť aké poškodenie dáva meč nepriateľom, o koľko percent menej poškodenia dostávame s brnením alebo koľko zdravia sa nám obnoví pri použití elixíru, museli by sme prepisovať kód triedy. Ak by sme chceli mať dva rôzne meče, museli by sme pre nich vytvoriť dve rôzne triedy, aj keď by mali rovnaké premenné, ktoré by sa len líšili v hodnotách.

Úplne by sme teda stratili možnosti, ktoré nám ponúkal editor Unity. Najradšej by sme chceli to, aby sme mohli tieto predmety upravovať pomocou inšpektora. Mohlo by nás napadnúť mať v každom leveli nejaký `GameObject`, ktorý by reprezentoval akúsi databázu predmetu, na ktorom by sme ako komponent mali skript, v ktorom by sme mali pre jednotlivé predmety uložené inštancie, ktoré by sme upravovali z inšpektora. Tento prístup má však veľa mínusov. Po prvé, pre každú scénu by sme museli mať takýto `GameObject` a informácie v nich by sa mohli líšiť. Po druhé, všetky predmety by sa upravovali v jednom spoločnom okne. Po tretie, Unity nepodporuje serializáciu pri dedičnosti. To znamená, že zmeny, ktoré by sme na predmetoch urobili, by sa nám neuložili, keďže by sa serializovali informácie len o základnej triede. Prvý a posledný bod by sa teoreticky dali pri takomto prístupe vyriešiť. Existujú spôsoby, pomocou ktorých by sa `GameObject` pri načítaní scény do hry presunul do špeciálnej scény, ktorá bude vždy načítaná a stačil by nám teda len v prvej scéne a taktiež od Unity 2019.4 existuje atribút `SerializeReference`[14], ktorý Unity hovorí, aby serializovalo celého potomka. Väčšina problémov by však zostávala. Ak by sme chceli pridať nový predmet do databázy, museli by sme opäť prepisovať kód skriptu pre databázu. Ak by sme sa chceli na takéto predmety odkazovať z iných `GameObjectov`, muselo by to byť natvrdo z kódu a znovu by sme nevyužívali v plnej miere inšpektor.

Najlepším riešením by bolo, keby sme predmety mohli ukladať v projekte ako samostatné assety, ktoré nie sú prepojené so žiadnou scénou, môžeme ich upravovať pomocou inšpektora a vieme sa na nich odkazovať z `GameObjectov`. Taktiež by bolo super, keby sme mohli mať pre jeden typ viacero assetov — viacero brnení, ktoré sa líšia hodnotami premenných a ide o samostatné assety.

Predmet ako ScriptableObject

`ScriptableObject`[15] nám všetko toto ponúka. `ScriptableObject` je špeciálna trieda, od ktorej sa „inštancie" vytvárajú v podobe assetov, ktoré sa ukla-

dajú v projekte. Assety nie sú súčasťou žiadnej scény a nemôžu ani obsahovať referencie na `GameObjecty` zo scén, len na iné assety. Unity nám dáva možnosť tieto assety vytvárať priamo v editore, funguje pre nich inšpektor a vieme pri nich využívať dedičnosť s tým, že sa serializujú správne.

Okrem toho, že inak fungujú ich inštancie, ide o normálne triedy. Vieme pomocou nich napríklad spĺňať interface, čo nám pomohlo implementovať použiteľné predmety. V projekte by sme teda mali pripravený interface s metódou `void Use()`, ktorá by sa pri použití predmetu zavolala.

Na jednotlivé assety sa môžeme odkazovať priamo zo skriptov a vieme v inšpektore vyberať, na ktorý asset má ukazovať referencia. `ScriptableObject` sa využívajú hlavne v prípadoch, kde si potrebujeme uchovávať nejaké informácie pre celý projekt.

Ak teda dizajnér potrebuje upraviť nejaký predmet, nájde si konkrétny asset a pomocou inšpektora ho môže upravovať. Ak chce pridať napríklad nový meč, môže vytvoriť nový asset priamo v editore a hneď s ním pracovať.

ScriptableObjecty teda splňujú všetko čo sme od predmetov chceli v hre mať a práca s nimi je v editore jednoduchá a preto sme ich v práci použili.

Predmety v leveli

Taktiež sme hovorili o predmetoch v leveloch, ktoré by mohol hráč zbierať. K tomu sme sa rozhodli použiť normálne `GameObjecty` s `Colliderom`, ktorý je trigger. Tento `GameObject` obsahuje ako komponent skript, ktorý sa odkazuje na asset konkrétneho predmetu a pri tom, ako hráč vojde do triggeru získa daný predmet. Tu môžeme vidieť potrebu inventára, ktorému sa budeme venovať v nasledujúcej sekcii.

3.6 Inventár

Keďže sme už mali systém predmetov, potrebovali sme do hry pridať nejakú formu inventára. Už v predchádzajúcej kapitole Špecifikácia zadania sme písali aj o tom, že by sme potrebovali byť schopný mať viacero takýchto inventárov naraz – normálny inventár a inventár na vybavenie. Ešte pred prácou na inventárnom systéme sme si museli odpovedať na dve otázky:

- Aké objekty v hre budú môcť mať inventár?
- Aké typy inventárov potrebujeme?

V hre sme nepožadovali mať možnosť toho, že inventár by mali mať aj iné postavy ako tá, ktorú ovláda hráč. Ako jedno z kritérií, ktorým sme sa riadili už pri výbere žánru hry v kapitole Úvod sme si však povedali, že budeme klásť dôraz na rozšíriteľnosť. Dávalo teda zmysel rozmýšľať aj nad situáciami, v ktorých by sme chceli pridať inventár aj iným objektom v leveli, napríklad kvôli forme obchodovania s inými postavami..

Taktiež budeme chcieť dosiahnuť to, aby bol inventár čo najflexibilnejší. To nám vyplýva z toho, že u vybavenia chceme obmedziť to, aké typy predmetov do neho môžu vôbec ísť.

Znovu sme teda museli riešiť to, ako bude inventár implementovaný, pomocou:

- bežnej C# triedy,
- `ScriptableObjectu`,
- skriptu.

Inventár ako bežná C# trieda

Na rozdiel od predmetov dáva zmysel aj normálna C# trieda. Každá inštancia inventáru je jedinečná a v podstate by sme sa mohli zaobísť aj bez dedičnosti, kde by nám stačila jedna trieda a mohli by sme teda využiť aj inšpektor aj bez použitia `SerializeReference`. Inventár by sme mali uložený v nejakom skripte a keďže by sme mali len jednu triedu, mohli by sme pre konkrétnu komponentu na konkrétnom `GameObjecte` inventár nastavovať z inšpektora.

Problémom daného prístupu ju však rozšíriteľnosť. Ak by sme do hry chceli pridať nový typ inventára, kde by sa nám hodila už spomínaná dedičnosť, systém by sa nám začal rozbíjať.

Inventár ako ScriptableObject

`ScriptableObject` sa ponúka v prípade, kde je v hre malý počet inventárov, takže nám nevadí ich vyrábať ako assety a zároveň myslíme na rozšíriteľnosť – keďže podporujeme dedičnosť. Ak by sme ale mali mať v hre stovky inventárov, mohlo by začať byť nepríjemné a neprehľadné, že by každý jeden z nich bol samostatný asset.

Inventáre v podobe `ScriptableObjectov` nám ponúkajú ešte jednu výhodu. Keďže nepatria žiadnemu `GameObjectu` ani scéne, pri zmene scén nazaniknú a nezmení sa ich obsah. V prípade obyčajnej C# triedy, by sme museli mať inventár uložený v nejakom komponente. Zmenou scény by nám však `GameObjecty` spoločne s komponentami zanikli.

Inventár ako komponent

Tretím, trochu netradičným riešením tohto problému by mohlo byť implementovanie inventáru ako skriptu, ktorý by sme potom mohli pridať na `GameObject` ako komponent. Mali by sme možnosť jednoducho pridať inventár na akýkoľvek `GameObject` v scéne, podporovali by sme dedičnosť, prácu v inšpektor aj správnu serializáciu. Problém by však mohol nastať, keby sme chceli mať pre jeden `GameObject` viacero inventárov, čo by mohlo viesť k tomu, že na jednom `GameObjecte` by sme mali viacero komponent rovnakého typu, čo by sa mohlo stať neprehľadným.

Rozhodnutie použiť ScriptableObjecty

Nakoniec sme sa rozhodli ísť cestou `ScriptableObjectov`. Možnosť, že by sme chceli do hry pridávať nové typy inventárov, nám vylúčilo prvú možnosť, keďže ináč by sme museli používať atribút `SerializeReference`. V takom prípade však samo Unity odporúča použiť `ScriptableObjecty`[14].

Implementácia inventára ako skriptu by dával zmysel v hrách s veľkým množstvom postáv, ktoré by mali inventáre. Problém by však mohol byť ten, že inventár by bol pevne spojený s nejakým `GameObjectom`. Zdieľanie inventára medzi viacerými `GameObjectami` či scénami by bolo veľmi náročné na implementáciu. Ako lepší spôsob nám prišlo mať inventáre ako `ScriptableObjecty` a prácu s nimi upravovať v skriptoch, ktoré by si na ne držali referenciu.

3.7 Kúzla

Ďalšiu hernú mechaniku, ktorú sme v hre požadovali boli kúzla. V prvom rade sme sa museli pozrieť na to, aké typy kúziel by sme v hre mohli mať, aby sme sa mohli rozprávať o tom, ako by sme ich chceli implementovať:

- kúzla, ktoré len vyvolajú nejaký efekt,
- kúzla, u ktorých nám vznikne nejaký objekt (napríklad ohnivá guľa).

Prvá situácia je jednoduchá, ide len o zavolanie nejakej metódy. V druhej ide vlastne o vytvorenie nejakého `GameObjectu` s nejakým komponentom, ktorý simuluje jeho správanie. K vytváraniu `GameObjectov` za behu môžeme využiť `Prefaby`[16]. `Prefab` je `GameObjectov`, ktorý nepatrí do žiadnej scény a je uložený ako asset v projekte. Môžeme ho využívať napríklad k tomu, aby sme za behu aplikácie vytvárali a pridávali do scény jeho kópie.

Potrebovali sme však kúzlam dať nejakú štruktúru, aby sme ich mohli v hre vytvárať a ukladať. Pre hráča sme taktiež potrebovali špeciálny inventár na kúzla — informáciu o tom, ktoré kúzla je naučený a ktoré kúzlo je práve aktívne.

Uvedomili sme si, že sme sa nachádzali vo veľmi podobnej situácii ako pri práci na predmetoch. Kúzla sa totiž veľmi podobajú predmetom. Ide o niečo čo nesúvisí konkrétne so žiadnou scénou a pri práci na nich by sme chceli využiť možnosti editora. Inšpirovali sme sa teda predmetmi a kúzla sa rozhodli rovnako implementovať ako `ScriptableObjecty`. Pomocou `ScriptableObjectu` sme implementovali aj hráčov inventár pre kúzla.

3.8 Nepriatelia

V kapitole Špecifikácia zadania sme písali o potrebe implementácie nepriateľov a taktiež sme si určili to, že ich budeme rozdeľovať do dvoch skupín na:

- bežných nepriateľov,
- bossov[17].

Bežní nepriatelia

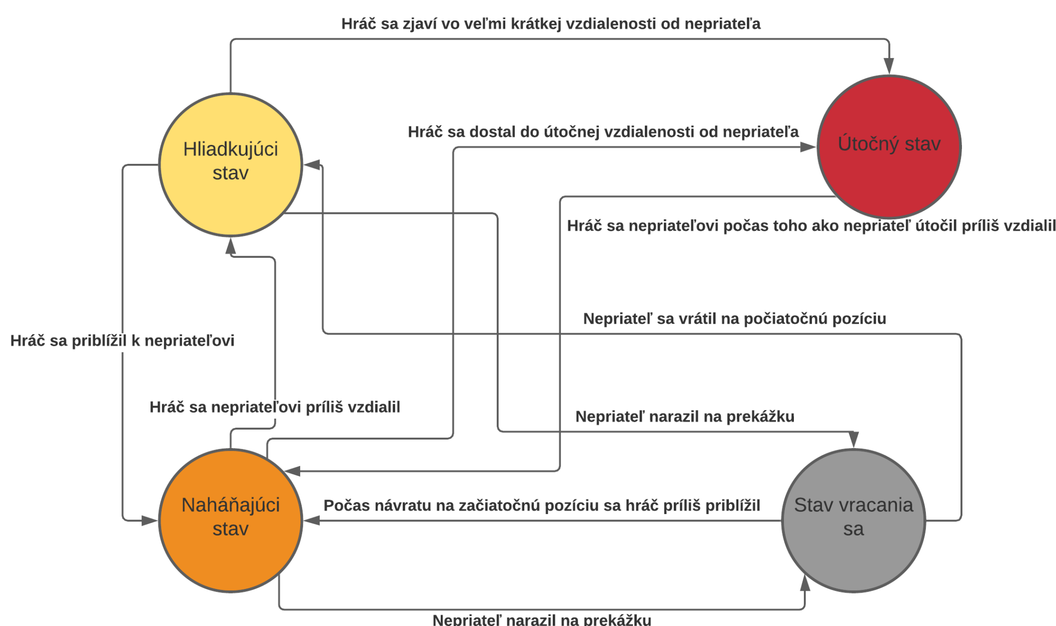
Najdôležitejšou časťou pri implementácii nepriateľov je implementácia umelej inteligencie, ich správania, v skratke *AI* (artificial intelligence). V projekte sme sa snažili prísť s modelom, ktorý by bol čo najvšeobecnejší a dal sa dobre rozširovať.

Stavový automat

Autor práce sa rozhodol pracovať so stavovým automatom. Ide o model, v ktorom pracujeme s množinou stavov a s prechodmi medzi nimi. Automat sa môže v jednom momente nachádzať len v jednom stave a medzi stavmi prechádza, ak sa splnia nejaké podmienky.

Stavový automat je využívaný pri mnohých problémoch, keďže je len na nás, čo si určíme ako stavy. V bankovníctve by mohlo ísť o to, aké množstvo prostriedkov sa nachádza na nejakom účte, v našej počítačovej hre by sme mohli do stavov rozdeliť správanie sa nepriateľov. Na nepriateľských postavách by sme teda mali komponent, ktorý by obsahoval náš stavový automat s nejakými stavmi. Zo skriptu by sa potom volala na stavovom automate metóda, ktorá by podľa stavu, v ktorom sa automat nachádza, zavolala jeho kód na simuláciu správania sa nepriateľskej postavy. Potom by sa skontrolovalo to, či sa nemá prejsť do iného stavu a poprípade do nového stavu prešlo.

Ako by mohol vyzerať graf, ktorý by ilustroval správanie sa takéhoto nepriateľa pomocou rozdelenia na stavy a prechody môžeme vidieť na obrázku 3.4, ktorý reprezentuje Skeleton nepriateľa z našej hry, kde vrcholy sú stavy a hrany predstavujú možné prechody.



Obr. 3.4: Stavový automat pre skeletona v hre

Stavy

Pred implementáciou stavového automatu a jednotlivých stavov sme si však museli určiť pár záležitostí. Bolo potrebné si ujasniť:

- či budú prechody uložené v automate alebo jednotlivých stavoch,
- či obmedzíme aké stavy bude možné v automate využívať,
- kedy sa budú stavy vytvárať,

- či sa pri prechode vytvoria nové stavy alebo si budeme stavy pamätať.

Keďže jeden z dôvodov prečo sme sa rozhodli automat využiť bolo to, že si vieme implementáciu sprehľadniť a rozdeliť, dáva zmysel mať celú logiku (vrátane rozhodovanie sa do akého stavu sa má prejsť) v jednotlivých stavoch. Taktiež sa autorovi páčilo to, aby sa vedelo už na začiatku, v ktorých stavoch sa bude môcť automat nachádzať – čim sa nám vyrieši aj tretí bod a stavy budeme vytvárať pri vzniku automatu. Taktiež sme sa rozhodli si stavy uchovávať, čo nám umožňuje si v stavoch pamätať informácie aj po prejdení do iného stavu – napríklad informáciu kde naposledy hráča nepriateľ videl. Napríklad u skeletona, o ktorom sme písali už trocha vyššie si stav pamätá ak narazil na prekážku a v budúcnosti sa nebude ani snažiť sa za ňu dostať. Tým, že pri prechode nevytvárame nové stavy, vie túto informáciu využiť aj po prechode do iného stavu.

Bossovia

Správanie sa bossov je špecifické tým, že je často silno sekvenčné a cyklické. Môžeme si predstaviť bossa, ktorý na hráča útočí sériou nejakých útokov a po tom, ako ho hráč dostane pod polovicu života sa rozzúri a jeho útoky začnú hráča viac poškodzovať, začne využívať nové útoky alebo sa napríklad stane odolnejším. Keďže sa herná logika v našej hre bude odohrávať len v 2D, dáva zmysel aby bol boss statický — nebude sa teda zo svojho miesta hýbať. Taktiež sme plánovali mať hru implementovanú tak, že súbojom s bossom sa práve hraný level ukončí. Hráč by teda nemal mať možnosť od takéhoto bossa utiecť, ale po začatí súboju s bossom boj aj dokončiť. Preto sa autor rozhodol neobmedzovať v tom, ako k implementácii takýchto bossov pristupovať. Bossovia sa nebudú v hre využívať viackrát, budú špecifickí pre ich level. V tomto prípade teda nedáva veľmi zmysel sa nejakým spôsobom snažiť vymyslieť štruktúru, ktorú by mali bossovia dodržovať, keďže rôzni bossovia môžu fungovať úplne inak. Samozrejme, implementácia bossov pomocou stavového automatu nemusí byť zlá, no nesnažili sme sa ku nej smerovať.

V hre sme sa rozhodli využiť *Animation Eventy*[18]. Animation Events nám umožňujú zavolať kód v presný čas animácie. Na bossovom `GameObjecte` sa bude teda nachádzať komponent, ktorý bude kontrolovať priebeh boja (zníženie života pod kritickú hodnotu, priblíženie sa hráča, bossovu smrť) a podľa toho nastavovať animačné parametre. O volanie potrebných metód (kontrola toho, či boss nezasiahol hráča, vytvorenie `GameObjectu` kúzla) počas animácií sa budú starať *Animation Eventy*.

3.9 Systém života a many

Už v sekcii Kúzla sme písali o tom, že používanie kúziel by malo hráča niečo stáť. V hrách sa používa „mana“ – zdroj podobný životu, kde si môžeme predstaviť, že hráč platí životom za poškodenie, ktoré mu utŕžia nepriatelia a manou platí za používanie kúziel.

Je teda vidno, že tieto dva systémy toho majú veľa spoločného. Chceli sme pripraviť systémy, ktoré by boli rozšíriteľné a prehľadné. Je dôležité si uvedomiť, že systém života je potrebné implementovať aj u nepriateľov. Cieľom bolo taktiež

to, aby sme tento systém mohli použiť aj u bossov, ktorý môžu na poškodenie od hráča reagovať veľmi špecificky. Ako vhodné riešenie nám prišli *udalosti*.

3.9.1 Udalosti

Udalosti[19] nám dávajú možnosť toho, že trieda alebo objekt môže oznámiť iným triedam alebo objektom, keď nastane niečo dôležité. Táto trieda sa nazýva *poskytovateľ*, triedy ktoré sú oboznámené sú *odoberatelia*. Udalosti nám prinášajú prehľadnejšiu a bezpečnejšiu prácu s delegátmi – k udalostiam sa môžu odoberatelia len registrovať a registráciu odstrániť, nedá sa celá udalosť prepísať. Udalosť sa taktiež môže zavolať len z kódu triedy, do ktorej patrí.

Udalosti nám teda ponúkli to, čo sme požadovali. V základných triedach teda využívame sadu udalostí, cez ktoré môžu rôzne skripty, registrovať na tieto udalosti rôzne metódy — u hráča potrebujeme pri zmene zdravia prekresliť užívateľské rozhranie, pri smrti ukončiť hru, pri zvýšení zdravia by sme mohli chcieť zmeniť farbu hráčovho zdravia. U bossa by sme mohli chcieť pri určitej hodnote prejsť do ďalšej fáze boja, pri jeho smrti ukončiť level.

```
public class HealthSystem
{
    public event Action OnDamage;
    public event Action OnHeal;
    public event Action OnHealthChange;
    public event Action OnMaxHealthChange;
    public event Action OnDeath;

    private float health;
    private float maxHealth;
```

Obr. 3.5: HealthSystem využívajúci eventy

3.10 Questy

Questy plnia v hre môžu úlohu rozprávača, hráča posúvajú v deji, sú spestrením príbehu a dávajú mu cieľ. Z toho dôvodu sme chceli nielen vytvoriť nielen samotný systém questov, ale taktiež rozšíriť editor v Unity tak, aby mohol tieto questy vytvárať dizajnér pri vytváraní levelu. V tejto sekcii sa budeme venovať tomu, na aké problémy sme pri plnení týchto cieľov narazili a akým spôsobom sme ich nakoniec vyriešili.

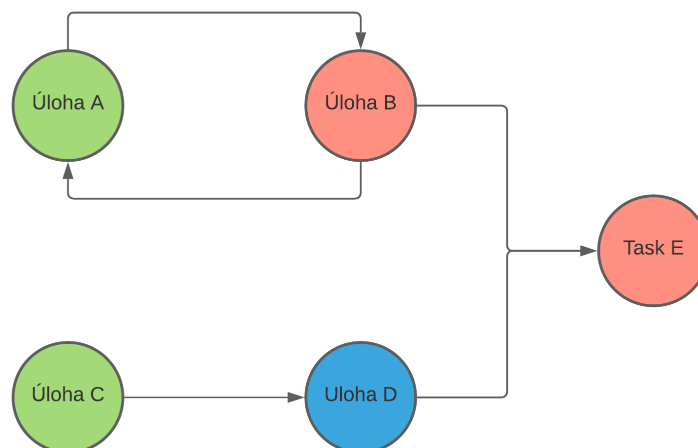
3.10.1 Štruktúra questov

Prvú vec, ktorú sme potrebovali vyriešiť bolo to, akým spôsobom budeme questy štruktúrovať. V kapitole Špecifikácia zadania sme sa rozhodli questy ďalej

rozdeľovať – na úlohy. Na splnenie samotného questu je potom potrebné splniť všetky podúlohy. Quest by mali tvoriť aj odmeny, ktoré by hráč získal za jeho splnenie. Pri práci na questoch autor rozmýšľal aj nad nepovinnými úlohami — úlohami, ktoré nie je potrebné dokončiť ku splneniu questu. V odovzdávanej verzii hry takéto úlohy nie sú k dispozícii, jedná sa však o jedno z potenciálnych, zaujímavých rozšírení tohto systému.

Taktiež sme chceli mať možnosť jednotlivé úlohy v queste prepájať. To znamená, že by sme úlohám v queste dávali poradie, v ktorom by sa museli vykonávať. To nám opäť rozširuje možnosti pri tvorbe questov.

Odhliadnuc od konkrétnej implementácie úloh, odmien a celkovo questov, quest si môžeme predstaviť ako orientovaný graf, v ktorom vrcholy predstavujú jednotlivé úlohy a hrany medzi nimi predstavujú spomínané závislosti. Z toho tiež dostávame, že tento graf musí byť orientovane acyklický. Ak by totiž takýto graf obsahoval orientované cykly, znamenalo by to, že by bol nesplniteľný. Túto situáciu môžeme vidieť na obrázku 3.6. Orientovaná hrana reprezentuje závislosť, pri ktorej je potrebné splniť prvú úlohu, z ktorej hrana vychádza. Teda v tomto obrázku je potrebné splniť úlohu C predtým, ako splníme úlohu D. Problém nám robia úlohy B a A. Keďže existuje orientovaná cesta z A do B, ale zároveň aj z B do A, nikdy nezačneme žiadnu z týchto úloh plniť, keďže budeme čakať na to, kým bude splnená druhá.



Obr. 3.6: Nesplniteľný quest

Toto bol jeden z hlavných dôvodov, prečo sme v kapitol Špecifikácia zadania rozprávali o editore vo forme grafu. Editor je určený na to, aby dizajnérom uľahčoval prácu na tvorbe hry. Graf reprezentuje štruktúru questu najdôveryhodnejšie, vďaka čomu dizajnér najlepšie a najprehľadnejšie uvidí o čo v quest ide. Z hrany vedúcej medzi dvomi vrcholmi je hneď zrejmé, medzi ktorými úlohami sa v queste nachádza závislosť, čo sa nedá povedať o prípade, keby s touto závislosť pracoval dizajnér priamo ako s nejakým indexom do poľa prehovorov. Vznik cyklov, ktoré sme spomínali by sme mohli kontrolovať za neho v oboch prípadoch a nedovoliť mu takýto quest vytvoriť, ale v prípade grafu by volo oveľa prehľadnejšie tento problém spozorovať a teda aj opraviť.

Pred prácou na editore sme však ešte museli vyriešiť otázky týkajúce sa implementácie questov.

Implementácia questov

Určite sme pri questoch, konkrétne úlohách a odmenách chceli využiť dedičnosť. Tú sme už riešili viackrát a dospeli k tomu, že sme mali opäť dve možnosti:

- `SerializeReference`,
- `ScriptableObject`.

Spomínali sme, že `SerializeReference` neodporúča používať samotné Unity, čo by teda znamenalo, že by sme opäť použili `ScriptableObjecty` a jednotlivé úlohy, odmeny a aj samotné questy ukladali ako assety v projekte. Keďže však `ScriptableObjecty` nie sú súčasťou žiadnej scény, nemôžu sa do scén priamo odkazovať. Ak by sme teda chceli mať úlohu, ku splneniu ktorej by bolo treba poraziť nejakých nepriateľov, nevedeli by sme sa na týchto nepriateľov zo `ScriptableObjectu` odkazovať. Ak by sme predsa len použili `SerializeReference`, nemuseli by sme tento problém riešiť. Každý level by obsahoval `GameObject`, v ktorom by boli uložené všetky questy pre daný level a questy by sa mohli do scény voľne odkazovať. To by nám ale vylúčilo v budúcnosti pridávať zložitejšie questy, ktoré by boli určené pre viacero scén naraz, keďže quest by musel byť natvrdo prepojený s jednou scénou. Tento prístup sa autorovi nepáčil, aj keď by bol na implementáciu jednoduchší ako využitie `ScriptableObjectov`. Ak sme ich však chceli použiť, potrebovali sme vyriešiť, akým spôsobom sa budeme z úloh a odmien odkazovať do scén.

Manažéry

Potrebovali sme teda vymyslieť model, v ktorom by sme sa vedeli na `GameObjecty` v scénach odkazovať nepriamo.

Autor práce sa rozhodol využiť koncept *manažérov*. Manažér je `GameObject`, ktorý sa nachádza v scéne a obsahuje skript, v ktorom si drží indexované referencie na konkrétny typ `GameObjectov` v scéne — manažér pre nepriateľov by si držal referencie na všetkých nepriateľov v scéne, manažér pre predmety by si držal odkazy na všetky `GameObjecty` reprezentujúce predmet, ktorý môže hráč pozbierať.

Ak sa teda úloha alebo odmena potrebuje odkazovať na nepriateľa, namiesto priamej referencie si uloží jeho index v manažéri.

Questy ako assety

Využitím manažérov a `ScriptableObjectov` sme dosiahli pár vecí:

- vieme sa z úloh a odmien „odkazovať“ do scén,
- je jednoduchšie v budúcnosti pridať do hry questy, ktoré sa plnia vo viacerých leveloch,
- questy znamenajú v iných scénach niečo úplne iné,
- a pri pridávaní nového typu odmeny alebo úlohy do hry bude možno potrebné pridať do scén nový typ manažéra.

Keďže máme questy uložené ako samostatné assety, ktoré nepatria žiadnej scéne, je možné quest použiť v ľubovoľnom leveli. Na jednotlivé **`GameObjecty`** sa odkazujeme nepriamo, pomocou indexov do manažérov. V každej scéne znamená index niečo iné, preto je potrebné si dávať pozor na to, aby sme questy používali len v leveloch, pre ktoré boli určené. Ak chceme manažéry využívať, musia sa v jednotlivých scénach nachádzať. Ak by sme teda chceli pridať nový druh úlohy, ktorý by sa potreboval odkazovať na niečo, k čomu nemáme manažér, museli by sme tento manažér do scény pridať. Nevidíme v tom však problém, keďže manažéry sú veľmi jednoduchý koncept a pridanie nového manažéra zaberie len chvíľu.

Quest manažér

Quest manažér je špeciálny **`GameObject`** v scéne, ktorý v sebe drží referencie na všetky questy, ktoré sú pripravované pre danú scénu. Keďže questy znamenajú v každej scéne niečo iné, považovali sme za vhodné mať nejaký systém, pomocou ktorého by sme vedeli pri akej scéne quest upravovať. Ak teda chceme pristupovať v editore ku questom, malo by to vždy byť prostredníctvom quest manažéra, aby sme si boli istí, že pracujeme v dobrej scéne.

3.10.2 Editor pre questy

Rozšírenie editora bolo pre nás teda nutnosťou. Predstava toho, že by sme od dizajnéra chceli, aby pracoval s indexmi v manažéroch bola nemysliteľná. Budeme sa teda venovať tomu, aké možnosti nám v tomto smere Unity ponúklo a ako sme sa nakoniec rozhodli.

Rozšírenie inšpektora

Mohli by sme skúsiť rozšíriť inšpektor quest manažéra a vytvárať a upravovať questy priamo v ňom. Potom by sme mali zaručené, že budeme pracovať s questom v správnej scéne. Problém je však v tom, že v inšpektore nemáme nemáme možnosti na kreslenie a teda sme tento prístup použiť nemohli.

EditorWindow

`EditorWindow`[20] je okno, ktoré môžeme nazvať ako „nezávislé“. Inšpektor je vždy spojený s nejakou konkrétnou entitou v projekte, či už ide o asset, Prefab alebo **`GameObject`**. **`EditorWindow`** však stojí samostatne a prepojenie si musíme vytvoriť sami, pomocou parametrov pri vytváraní okna. **`EditorWindow`** nám dáva možnosť kreslenia, ktorú sme v inšpektore nemali. To, čo vieme pripraviť v inšpektore vieme pripraviť aj v **`EditorWindow`**, opačný smer však neplatí.

GraphViewEditorWindow

`GraphViewEditorWindow`[21] je rozšírenie **`EditorWindow`**, prostredníctvom ktorého nám Unity dáva možnosť jednoduchšie pracovať s grafmi a prináša nám základné stavebné jednotky – vrcholy (uzly) a hrany. Jediný problém je v tom, že ide o *experimentálne API*. Tým pádom k nemu existuje minimum dokumentácie

a materiálov na internete. Z tých pár materiálov[22], prišlo toto riešenie autorovi práce ako najvhodnejšie s tým, že nám umožní pripraviť dizajnérovi najlepšie podmienky na tvorbu questov a preto sme sa rozhodli ho v práci využiť. Od Unity sme dostali naozaj základy — vytváranie a vykresľovanie hrán a vrcholov, presúvanie sa po grafe a približovanie a oddiaľovanie okna. Logiku okna, čo hrany a vrcholy znamenajú, ako má vnútro vrcholu vyzerať, čo hrany reprezentujú a všetko ostatné sme si museli implementovať sami.

Základná trieda, s ktorou sa pracuje je `Node`. `Node` je v podstate kontajner vizuálnych elementov, pomocou ktorých do `Nodu` kreslíme. Ide teda o také mini okno. Vďaka tomu sme mohli pripraviť rozdielne vyzerajúce `Nody` pre rôzne druhy úloh. S hranami pracujeme pomocou portov a základnej triedy `Edge`. Do `Nodov` vieme pridávať porty, medzi ktorými potom ťaháme hrany. Môžeme obmedzovať to, medzi akými portmi môže viesť hrana, čím môžeme vyriešiť problém s cyklami.

Dôležitým bodom bolo to, akým spôsobom by prebiehal preklad indexov. Dizajnér by sa totiž nemal zaoberať tým (vôbec by to ani nemal vedieť) ako sú questy reprezentované. Rovnako sme už písali o probléme toho, že v rôznych scénach quest reprezentuje niečo iné. Riešime to tak, že pri kreslení grafu komunikuje okno s práve načítanou scénou v editore a pozerá sa do jej manažérov. V jednotlivých uzloch sa teda zobrazujú už priamo `GameObjecty` zo scény, ktoré sa získali prekladom indexov. Využívame to aj pri úprave úloh a odmien, kde do jednotlivých políčok vo vrchole môže dizajnér rovno vkladať `GameObjecty` s tým, že rovno prebieha preklad a zisťuje sa jeho index `GameObjectu` v príslušnom manažéri. Pri uložení questu sa uložia už len indexy.

3.11 Dialógy

Dialógy by, podobne ako questy, mali v našej hre tvoriť jej dej a príbeh. Preto sme chceli do projektu pridať aj editor, v ktorom by sme vytváranie a úpravu dialógov uľahčili. V tejto kapitole teda budeme riešiť to, ako sme postupovali pri implementácii nielen samotných dialógov, ale aj spomínaného editora.

3.11.1 Štruktúra dialógov

V kapitole Špecifikácia zadania sme hovorili o tom, že dialógy by sme chceli členiť na prehovory. K tým by patrili hráčove odpovede, ktoré by reprezentovali prechod medzi prehovormi. A aby dialógy mohli ovplyvňovať dianie v leveli, ku odpovediam sme sa rozhodli pridať dialógové akcie. Pod dialógovou akciou si môžeme predstaviť získanie predmetu, ktorý je potrebný na prechod levelu, prijatie questu a pod.

3.11.2 Implementácia dialógov

V mnohých aspektoch sa sa dialógy podobajú questom. Delíme ich na menšie časti, ktoré môžeme navzájom prepájať (u questov poradie úloh, u dialógov prechody medzi prehovormi). Dialógy aj questy sa vďaka tomu veľmi dobre reprezentujú pomocou orientovaných grafov.

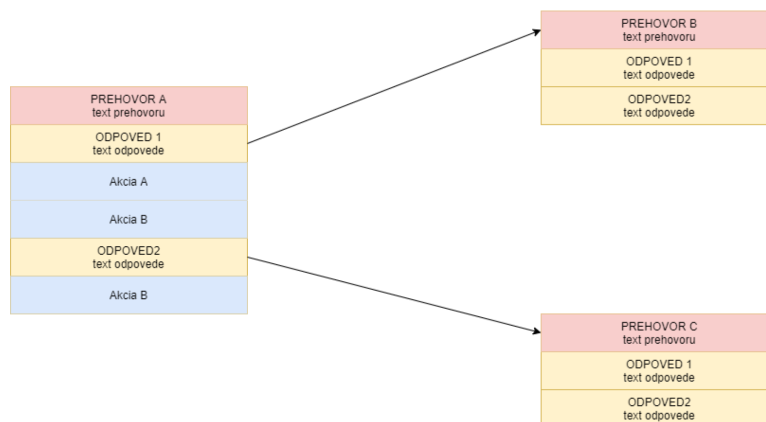
Z toho dôvodu sme sa pri implementácii opäť rozhodli využiť `ScriptableObjecty` a dialógy ukladať ako samostatné assety. Čo sa týka

prehovorov a odpovedí, rozhodli sme sa u nich využiť normálne C# triedy. Prehovorov je vlastne len text a list odpovedí a samotná odpoveď je tiež reprezentovaná textom a listom dialógových akcií. Tie sme už ale museli implementovať pomocou `ScriptableObjectov`, keďže sme chceli využiť dedičnosť a o `SerializeReference` sme si hovorili už viackrát, že ju využívať v projekte nechceme. Takéto rozdelenie dáva zmysel aj v prípadnom rozširovaní dialógového systému. Napríklad v prípade, že by sme chceli odpovede podmieňovať (odpoveď by sa nám zobrazovala len v prípade, že sme splnili nejaké podmienky), by sme mohli do odpovedí pridať list dialógových podmienok, ktoré by sme implementovali ako `ScriptableObjecty`. Autor sa tiež rozhodol obmedziť počet možných odpovedí pre jeden prehovor na dva z dôvodu, ako chcel riešiť vykresľovanie užívateľského rozhrania pre dialógy.

3.11.3 Editor

V podstate všetko čo sme povedali v predchádzajúcej sekcii Questy platí aj tu. Mali sme v podstate jedinú možnosť — `GraphViewEditorWindow`. O fungovaní a práci s `GraphViewEditorWindow` sme písali už pri questoch. Poslednou vecou, ktorú sme potrebovali vyriešiť bolo rozhodnutie čo budú reprezentovať vrcholy a hrany. Význam hrán závisí od toho, aké typy vrcholov sa v grafe nachádzajú. Mali sme teda dve možnosti:

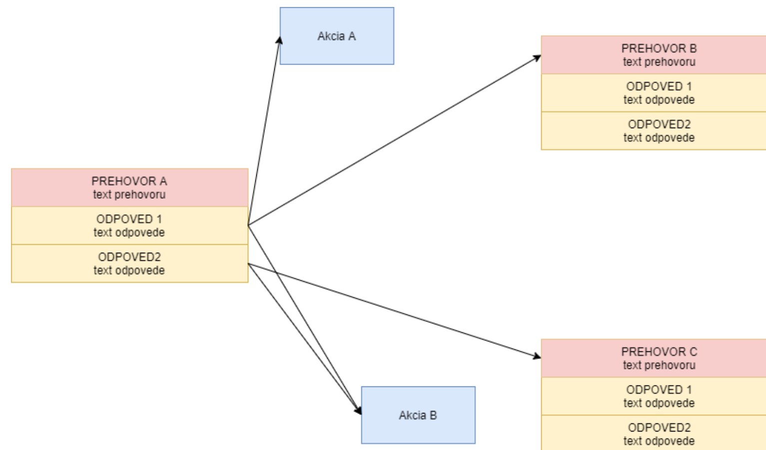
- vrcholy len pre celé prehovory (obrázok 3.7) ,
- vrcholy pre prehovory a akcie (obrázok 3.8).



Obr. 3.7: Vrcholy len pre celé prehovory

Tu sme sa rozhodli pre prvú možnosť a to hlavne z dôvodu prehľadnosti. Aj keď by mohlo dávať zmysel mať vlastné uzly pre jednotlivé akcie a ku jednotlivým odpovediam a prehovorom ich pripájať pomocou hrán (čo by nám napríklad umožnilo zdieľať jednu akciu pre viacero odpovedí), podľa autorovho názoru by tým mohlo vznikať až prílišné množstvo uzlov, hrany by sme museli rozdeľovať podľa toho, či vedú z vrcholu pre prehovor alebo akciu a zvýšilo by to neprehľadnosť pre dizajnéra — chceme dosiahnuť pravý opak. Akcie a odpovede sú teda súčasťou vrcholu pre prehovor, ku ktorému patria.

U grafu dialógu nemusíme riešiť to, či je graf orientovane acyklický — niekedy môže dávať zmysel vrátiť sa na nejaký prehovor.



Obr. 3.8: Vrcholy pre prehovory a akcie

3.12 Ukladanie a načítavanie

Ďalším požiadavkom bolo vytvoriť a pripraviť systém, vďaka ktorému by si hráč mohol svoj progres v hre ukladať. V kapitole Špecifikácia zadania sme písali aj o tom, že cieľom bolo taktiež to, aby rozširovanie tohto systému bolo čo najjednoduchšie. Pod tým si autor predstavoval to, že pri pridávaní nových vecí do hry nebude robiť problém ich začleniť do ukladacieho systému. V tejto kapitole sa venujeme tomu, ako sme k tomuto cieľu pristupovali a aké riešenia sme si zvolili.

3.12.1 Formát ukladaných dát

Prvý krok, ktorý nás čakal, bolo rozhodnutie, kam sa budú naše dáta ukladať. V Unity sme mali dve možnosti:

- **`PlayerPrefs`**,
- ukladanie do súborov.

PlayerPrefs

`PlayerPrefs`[23] je trieda od Unity, ktorá hráčovi umožňuje si ukladať dvojice kľúč a hodnota, ktoré pretrvávajú aj po tom ako hru vypneme a zapneme. Hodnoty, ktoré takto môžeme ukladať môžu byť typov string, int alebo float. Aj keď by sa tento spôsob teoreticky mohol použiť na ukladanie hier, práca s ním by bola neuveriteľne neprehľadná, pomalá a z hľadiska rozšíriteľnosti neakceptovateľná. Ide však o jediný spôsob, ktorý nám ponúka priamo Unity.

Ukladanie do súborov

Jediným riešením , ktoré nám ostávalo bolo využitie save súborov[1]. Vytváranie súborov a ukladanie dát do nich je spôsob, ktorý sa využíva vo väčšine hier. Najprv sme však museli vyriešiť to, kam by sme tieto súbory ukladali. Tu nám Unity ponúklo jednoduché riešenie – využitie **`ApplicationPersistentDataPath`**, ktorý obsahuje cestu k priečinku, ktorá sa nemení medzi rôznymi behmi aplikácie. Túto

[1]save súbor — súbor, v ktorom si ukladáme nejaké dáta

cesta sme sa rozhodli využiť ako základ pri ukladaní save súborov. Pri ukladaní hry sa v tomto priečinku ešte vytvorí priečinok Saves, do ktorého sa už ukladajú jednotlivé save súbory.

Ďalej sme sa museli rozhodnúť v akom formáte budeme informácie ukladať (ako budeme dáta serializovať). Mali sme výber tri hlavné možnosti:

- JSON,
- XML,
- BinaryFormatter.

JSON aj XML sú si podobné v tom ako dáta reprezentujú. Obdiva formáty sú hierarchické, „human readable"/„self describing" (človek ich vie čítať, keďže sa ukladá text), a oba ukladajú dáta ako text. To nám dáva možnosť si takéto súbory otvárať a upravovať priamo v nejakom textovom editore. To môžeme brať ako plus, ale aj mínus. Ak by sa totiž hráč dostal priamo k takýmto súborom mohol by ich prepísať a zmeniť to, čo v nich bolo uložené. Autorovi to skôr prišlo ako niečo nevhodné. Taktiež to, že pracujeme s textom znamená, že v mnohých prípadoch ukladáme dáta neefektívne, keďže všetko sa ukladá ako text. Z týchto dvoch prístupov by sme preferovali JSON, keďže je kratší(nepoužíva uzatváracie tagy), ľahšie sa parsuje (deserializuje) a hlavne podporuje polia. Bez tých by sme hru ukladať nevedeli, takže XML sme vylúčili.

BinaryFormatter[24] na rozdiel od JSON serializuje dáta binárne, neukladá ich teda ako text, čím vo väčšine prípadov šetríme pamäť. Na druhú stranu však JSON ponúka spätnú kompatibilitu — ak by sme upravili triedy, ktoré sme ukladali, stále by nám išlo načítať staré save súbory. To ale autorovi to prišlo nevhodné — ak sme zmenili triedy, pomocou ktorých sme ukladali dáta, znamenalo by to, že už existujúce save súbory by boli zlé, nedržali by v sebe správne informácie. V takejto situácii by sme naopak nechceli byť schopní tieto dáta načítať. Ďalší problém s JSON bol ten, že nepodporuje dedičnosť, keďže mu musíme priamo pri deserializácii povedať, na aký typ sa má deserializovať s tým, že stratí všetky informácie o potomkovi. BinaryFormatter dedičnosť podporuje. Dedičnosť sme však potrebovali z dôvodu rozšíriteľnosti. Ak sme chceli novým objektom v hre dať jednoduchú možnosť si o seba ukladať a načítať nejaké dáta, nemohli sme im vnútiť presný formát týchto dát. Prístup, ktorý autora napadol bol taký, že by nám stačilo to, keby nám tieto objekty v hre vracali informácie o sebe uložené v premennej typu System.Object. Tú by sme potom serializovali a pri načítavaní hry deserializovali a danému objektu vrátili. Ten by vedel inštancia akého presného typu tam bola predtým a mohol teda dáta správne načítať.

Základná trieda, ktorú nám ponúka Unity na prácu s JSON je **`JsonUtility`** a pomocou nej dedičnosť v uložených dátach nevyriešime. Na Unity Asset Store sa síce nachádza asset *JSON .NET For Unity*, ktorý sa už s dedičnosťou vie vysporiadať, avšak z dôvodu toho, že BinaryFormatter nám ponúkal všetko to, čo sme potrebovali, sa autor rozhodol pre serializáciu dát využiť BinaryFormatter.

Posledný bod, ktorý bolo potrebné vyriešiť bolo to, ako hráčovi ukladanie sprístupnime. Mali sme na výber medzi:

- „voľným" ukladaním,
- „obmedzeným" ukladaním.

Jednotlivé prístupy si popíšeme v nasledujúcich častiach.

Voľné ukladanie

Pod voľným ukladaním máme na mysli systém, v ktorom by mal hráč možnosť si hru uložiť v akomkoľvek okamihu. Pre hráča najlepšia a najpríjemnejšia možnosť, pre vývojára jednoznačne najnáročnejšia na implementáciu. Na to, aby takýto systém fungoval správne je potom potrebné vedieť v akejkoľvek situácii uložiť dostatok informácii tak, aby sa hra mohla správne načítať späť. Ak by chcel vývojár takýto systém implementovať v hre s veľkým množstvom druhov nepriateľov, musel by vedieť verne zachytiť, uložiť a načítať všetky potrebné informácie o ktoromkoľvek z nepriateľov. To by mohlo obsahovať:

- nepriateľov život,
- level,
- dostatok informácii o animácii nepriateľa,
- celý stavový automat.

V takejto situácii by hráčovi určite vadilo, keby sa nepriateľ správal po načítaní inak, nachádzal sa v inej časti animácie a pod. Na druhú stranu, ak by sme u takejto hry zakázali ukladanie počas boja, mnoho vecí by sa nám zjednodušilo – napríklad by mohlo nastať to, že u nepriateľa by sme si informácie o animácii a stavovom automate ukladať nemuseli vôbec, keďže by sme mohli pracovať s informáciou, že určite nebojujú s hráčom. Hráčovi by potom určite menej prekážalo to, že nepriateľ je po načítaní v trochu inej časti animácie, keďže vie, že ho to v podstate nijako neovplyvní – čo sa u boja povedať nedá.

Druhým príkladom môžu byť kúzla. Tie sme opisovali už v sekcii Kúzla. Tie môžu fungovať napríklad tak, že nám v scéne vytvoria nový `GameObject`. Pri systéme voľného ukladania by sme potom pri načítavaní museli vedieť pre všetky takéto kúzla zo save súboru kúzlo presne zreplikovať. Opäť by sme mohli povedať to, že jednoducho neumožníme hru uložiť ak sú nejaké kúzla aktívne.

Voľné ukladanie je najnáročnejšie nielen na implementáciu, ale taktiež na to, koľko informácií je potrebné si uložiť.

Obmedzené ukladanie

Obmedzené ukladanie je prístup, pri ktorom hráčovi jednoducho obmedzíme to, v akých situáciách je možné si hru uložiť.

Potrebné je si tiež uvedomiť, že nie vždy je tento princíp použitý za účelom zjednodušenie si práce alebo zmenšenia veľkosti vytvoreného save súboru. Napríklad by sme v hre mohli implementovať z veľkej časti voľné ukladanie, no na určitých miestach ukladanie zakázať a to nie z dôvodu, že by sme toho neboli schopní. V hre, v ktorej by išlo o reakcie, by sme nechceli to, aby si mohol hráč hru pozastvaiť a už vôbec nie uložiť. Ďalším príkladom by mohol byť súboj s bossom v našej hre. Ten by mal ukončovať level a mal by byť náročnejší ako zvyšok levelu. Keby si mohol hráč hru ukladať ľubovoľne počas tohto súboja, stratil by tento súboj pointu.

U voľného ukladania je celkom zjavné čo chceme. U obmedzeného ukladania však môžeme mať viacero prístupov, ktoré sa delia do dvoch skupín podľa toho, či:

- hovoríme o tom, kde a kedy *nie je* možné hru uložiť,
- hovoríme o tom, kde *je* možné hru uložiť.

Prvý zo spomínaných prístupov je celkom priamočiary – v istých situáciách jednoducho hru nedovolíme uložiť – počas boja, rozhovoru, počas toho ako nie sme na zemi a pod.

Druhý spôsob môžeme nazvať ako systém *save pointov*.

Save pointy

Save pointy sú špecifické miesta, na ktorých je možné hru uložiť. Tento spôsob môže byť kombinovaný napríklad s tým, že hráč sa už nemôže po uložení ani vrátiť späť. Dizajnér vďaka tomu môže level v hre rozdeľovať na celky, kde je vlastne umožnené hráčovi hru uložiť až po tom, ako prejde celú časť.

3.12.2 Výber prístupu ku ukladaniu

Všetky prístupy dávali autorovi práce v jeho hre zmysel. Nakoniec sa rozhodol využiť voľné ukladanie s tým, že ukladanie bude zakázané jedine pri súbojoch na konci levelu. Rozhodol sa tak najmä z dôvodu toho, aby sa potenciálny dizajnér ničím okolo ukladania nemusel zaujímať. Save pointy by mohli byť v budúcnosti zaujímavým rozšírením pre hru.

3.12.3 Implementácia ukladania a načítavania

Ku implementovaniu ukladania a načítavania sme použili už spomínaný systém manažérov. V hre už sme niekoľko typov manažérov mali – na predmety v leveli, nepriateľov, ale aj postavy, s ktorými hráč vie viesť rozhovor. V leveloch sa však nachádza veľké množstvo `GameObjectov` a vytvárať manažér pre každý typ by nedávalo veľmi zmysel. Vymysleli sme teda všeobecný manažér, ktorý drží referencie na také `GameObjecty`, ktoré si potrebujú ukladať nejaké informácie. Aby nám to fungovalo, pripravili sme do hry jednoduchý interface, ktorý vidno na obrázku 3.9.

Má dve metódy `object Save()` a `void Load(object data)`. Na manažéri sa nachádza skript, ktorý drží referencie na komponenty `GameObjectov`, ktoré splňujú tento interface. Pri ukladaní sa všetky komponenty prejdú, získajú sa od nich dáta a tie sa serializujú. Pri načítavaní sa potom dáta vrátia komponentom, aby ich mohli spracovať.

Ukladanie GameObjectov, ktoré vznikajú počas levelu

Postup, ktorý sme spomínali vyššie funguje len pre `GameObjecty`, ktoré sa nachádzajú v scéne pri jej načítaní do hry. Napríklad kúzla takto uložiť nedokážeme. Pri kúzle totiž môže vznikne v scéne `GameObject`. Aj keby sme si informácie o ňom uložili, pri načítaní hry sa žiadny takýto `GameObject` v scéne

```
public interface ISavableLoadable
{
    4 references
    object Save();
    4 references
    void Load(object data);

}
```

Obr. 3.9: Interface

nenachádza, takže nemáme komu načítané dáta predať. Ani to by však nebol až taký problém. Vytvorili by sme nový interface, ktorý by bol úplne rovnaký ako ten predchádzajúci, obsahoval by však naviac vlastnosť, ktorá by vracala cestu k potrebnému **`Prefabu`**. Z **`Prefabu`** by sa vytvoril **`GameObject`** a tomu by sa predali dáta. Najäčší problém, ktorý sme potrebovali vyriešiť bolo to, ako sa o takýchto **`GameObjectoch dozvedieť`**. Mali sme dve možnosti:

- pri ukladaní hry prejsť všetky **`GameObjecty`** v scéne a zistiť to tak, že sa na každom spýtame, či neobsahuje komponentu, ktorá spĺňa daný interface,
- pridať do scény manažér a nechať na **`GameObjectoch`**, aby sa do tohto manažéra pridali.

Prvý spôsob sa autorovi nepáčil. Je síce najjednoduchší na implementovanie a jednotlivé **`GameObjecty`** v ňom nemusia nič riešiť, je ale veľmi pomalý, keďže musí prejsť všetky **`GameObjecty`** v scéne a zistiť, ktoré obsahujú správnu komponentu.

Vybrali sme si teda druhý prístup. V scéne sa nachádza manažér, na ktorom sa nachádza skript s listom na takéto **`GameObjecty`**, do ktorého je povinný ten, kto vytvára **`GameObject`**, o ktorom je potrebné si niečo ukladať, tento vytvorený **`GameObject`** do manažéra pridať.

4. Projektová dokumentácia

V tejto kapitole rozoberieme štruktúru celého projektu a kódu.

Pred prácou na projekte je potrebné si posťahovať a naimportovať všetky assety a závislosti, popísané je to v príloha A.2

4.1 Štruktúra

V kapitole Analýza sme sa rozhodli našu hru členiť do scén. V rovnakej kapitole sme sa tiež rozhodli v našej hre mať jednu scénu pre jeden level. Rovnako používame jednu scénu aj pre hlavné menu a tutoriál. Práve tieto scény sa používajú pri buildovaní hry (tvorbe spustiteľnej aplikácie). Poslednou scénou, ktorá sa v projekte nachádza je BaseScene, ktorá slúži na to, aby sme pri vytváraní nového levelu nemuseli všetky potrebné objekty do scény pridávať, ale stačí nám pri tvorbe nového level túto scénu využiť. V projekte máme teda štyri typy scén:

- MainMenu scéna,
- level scéna,
- tutoriál scéna,
- base scéna.

V nasledujúcich sekciách si tieto scény rozoberieme a pozrieme sa na to aké `GameObjecty` sa v nich nachádzajú a aké sú ich úlohy.

4.2 MainMenu scéna

Na obrázku 4.1 môžeme vidieť štruktúru scény hlavného menu. MainMenu je

Obr. 4.1: MainMenu scéna

tzv. východisková scéna, ktorá sa načíta po zapnutí hry a hráč sa môže rozhodnúť

začať novú hru, načítať uloženú alebo spustiť tutoriál. Taktiež sa jedine v MainMenu dá nastavovať ovládanie a vymazávať uložené hry. Hlavné `GameObjecty`, ktoré nás v v tejto scéne zaujímajú sú *Canvas*, *LoadManager* a *InputManager*.

MainMenu Canvas

Canvas je plátno, na ktorom sa nachádzajú jednotlivé tlačidlá hlavného menu. Nachádza sa na ňom taktiež skript `MainMenu.cs`, v ktorom sa nachádzajú metódy, ktoré sa volajú pri stláčání jednotlivých tlačidiel. Na to, aby sme mohli používať tlačidlá je potrebné mať v scéne `EventSystem`, ktorý však sám osebe nie je zaujímavý a preto tu o ňom písať nebudeme.

LoadManager

LoadManager je `GameObject`, ktorý sa stará o väčšinu práce, ktorá je spojená s načítavaním scén a zmenou aktívnych scén. Nachádza sa na ňom `SaveManager.cs` skript, v ktorom je tiež implementovaná väčšina logiky spojená s načítavaním a ukladaním hry, prechodmi medzi scénami a pod.

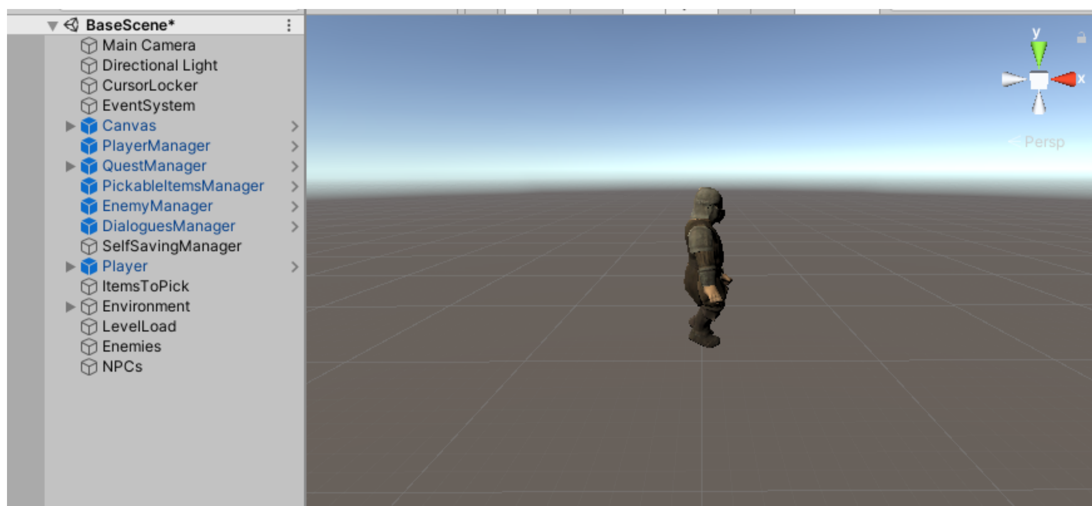
Z toho nám taktiež vyplývalo, že sme informácie z LoadManager potrebovali uchovávať medzi scénami. Napríklad pri načítaní scény s prvým levelom sme potrebovali vedieť, či sme level načítali ako novú hru alebo ako uloženú hru. Z toho dôvodu pri načítaní LoadManager zavoláme na ňom funkciu *DontDestroyOnLoad()*, ktorá nám zaručí to, že `GameObject` pretrvá aj zmenu scény – v skutočnosti sa vytvorí nová scéna, v ktorej sa všetky takéto `GameObjecty` nachádzajú a táto scéna bude stále aktívna. Tu nám vznikol problém, že pri viacnásobnom načítaní hlavného menu, by sa nám LoadManager začal duplikovať – vyriešili sme to aplikovaním návrhového vzoru *Singleton*.

InputManager

V kapitole Analýza sme sa rozhodli implementovať vlastný input systém, kde sme mali `ScriptableObject`, v ktorom sme držali informácie o tom, aká klávesy patria k ovládacím prvkom. Hráčovi sme však chceli dať možnosť si tieto nastavenia meniť. V builde hry (v spustiteľnej aplikácii) sa však zmeny `ScriptableObjectov` neukladajú medzi zapnutiami aplikácie. Pri nastavovaní ovládania je teda potrebné si tieto informácie ukladať do súboru a pri zapnutí hry ich načítať. Presne o toto sa stará InputManager. Podobne ako s LoadManagerom by sa nám hodilo mať tento `GameObject` uchovaný pri zmene scén, aby sme nemuseli stále čítať zo súboru. Aplikovali sme teda rovnaký prístup – DontDestroyOnLoad() a vzor Singleton.

4.3 Level scéna a Base scéna

Keďže scéna, v ktorej je pripravený konkrétny level je v podstate len zaplnená BaseScene scéna, budeme o nich písať v jednej sekcii. Štruktúru BaseScene môžeme vidieť na obrázku 4.2 Tu je dôležitých `GameObjectov` viacero, tak si o nich niečo napíšeme.



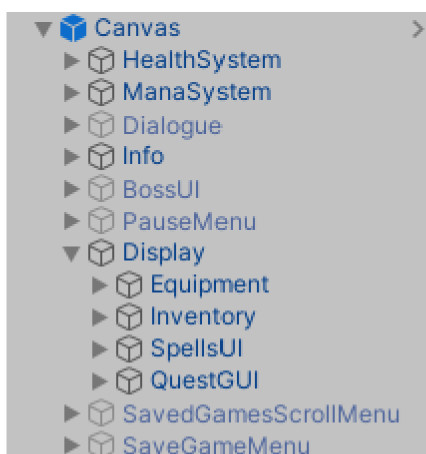
Obr. 4.2: BaseScene scéna

Main Camera

Main Camera je **`GameObject`** s kamerou v scéne. Nachádza sa na nej jednoduchý skript, ktorý v podstate nasleduje hráča. V jednotlivých leveloch sa na nej nachádza taktiež skript, ktorého úlohou je kameru zafixovať pri súboji s bossom na konci levelu.

Canvas

Canvas je plátno, na ktorom sa vykresľuje väčšina užívateľského rozhrania. Z obrázka 4.3 môžeme vidieť, že sa jedná o informácie o hráčovom živote a mane, vykresľovanie dialógov, rozhrania s questami, kúzlami, vykresľovanie inventára a vybavenia a taktiež napríklad PauseMenu.



Obr. 4.3: BaseScene Canvas

Manažéry

Ďalšími veľmi dôležitými **`GameObjectami`** v scéne levelu sú manažéry. Ako sme už v kapitole Analýza písali, využívame ich najmä pri ukladaní, pri questoch

a dialógoch. Konkrétne ide o:

- *PlayerManager* – najjednoduchší z manažérov, jediné čo obsahuje je referencia na `GameObject` postavy, ktorú hráč ovláda,
- *QuestManager* – obsahuje odkazy na všetky questy, ktoré patria do danej scény. Keďže questy sú `ScriptableObjecty`, môžeme okno na ich editáciu otvoriť nad akoukoľvek scénou, avšak význam questu sa potom mení podľa scény. QuestManager teda slúži hlavne na to, aby sa questy otvárali nad správnymi scénami,
- *PickableItemsManager* – skript, ktorý je na ňom drži indexované referencie na všetky `GameObjecty` v scéne, ktoré reprezentujú predmet, ktorý môže hráč „zdvihnúť“. Využíva sa pri questoch a ukladaní,
- *EnemyManager* – manažér pre nepriateľov v leveli. Na to, aby mohol byť nepriateľ pridaný do manažéra, musí obsahovať skript, ktorý rozširuje interface *EnemyHealth*,
- *DialoguesManager* — nejde priamo o manažér dialógov, ale `GameObjectov`, ktoré na sebe majú `DialogueController.cs` skript. Odkaz na dialógy sa nachádza v tomto skripte. Taktiež sa využíva najmä pri questoch a ukladaní hry,
- *SelfSavingManager* – ako sme spomínali v kapitole Analýza, chceli sme mať možnosť pridávať do hry nové typy `GameObjectov` a v prípade potreby si o nich nejaké informácie ukladať pri ukladaní hry. Keby sme však pre každý typ `GameObjectu` mali mať vlastný manažér, mohlo by nastať to, že by bola práca s nimi príliš neprehľadná. Tento manažér teda obsahuje list referencií na `GameObjecty`, ktoré majú na sebe skript, ktorý splňuje interface *ISavableLoadable*. SelfSavingManager sa potom pri ukladaní a načítavaní stará o to, aby sa dáta správne ukladali a správne pri načítavaní predávali `GameObjectom`,
- *SelfSavingRuntimeManager* – síce znie podobne ako predchádzajúci manažér, plní však úplne inú úlohu. Tento manažér sa stará o ukladanie a načítanie takých `GameObjectov`, ktoré sa nenachádzajú v scéne pri jej načítaní.

Player

`GameObject`, ktorý reprezentuje postavu ovládanú hráčom. Nachádza sa na ňom množstvo skriptov, ktoré riešia rozličné veci – hráčov pohyb, kúzla, život, questy a pod. O jednotlivých skriptoch píšeme podrobnejšie v sekci Character skripty. Taktiež máme v každom leveli jeden takýto `GameObject` – avšak pri prejdení levelu a postupe do ďalšieho levelu nechceme nejaké informácie stratiť (inventár, naučené kúzla, zvýšený život), o čo sa stará vyššie spomínaný LoadManager.

LevelLoad

LevelLoad sa stará o animácie pri dokončovaní levelov. Obsahuje skript, ktorý sa odkazuje na dve animácie. LoadManager o tomto LevelLoad `GameObjecte` vie a dané animácie pri zmene levelov prehrá.

Kontajnery

`GameObjecyt` ako ItemsToPick, Environment, Enemies a NPCs sú v podstate len kontajnery, do ktorých vkladáme ďalšie `GameObjecty`. Všetci nepriatelia by mali byť potomkom Enemies, postavy, s ktorými je možné viesť rozhovor by sa mali nachádzať v NPCs. Ide však len o snahu o sprehľadnenie práce – na prístup k takýmto `GameObjectom` v prípade potreby využívame skôr manažéry.

4.4 Skripty a kód

Štruktúru a to ako je projekt rozdelený do scén sme si teda popísali. V Unity sa ale herná logika implementuje najmä v skriptoch, ktoré dávame „na" `GameObject` v jednotlivých scénach. Skripty sú triedy, ktoré dedia od `MonoBehaviour` a vieme ich v podstate pridávať na jednotlivé `GameObject` ako takzvané komponenty.

Všetok kód, ktorý súvisí s implementáciou hry sa nachádza v priečinku Assets/Scripts.

Ten je ďalej rozdelený do priečinkov:

- *Camera Scripts* – v tomto priečinku sa nachádzajú skripty, ktoré ovplyvňujú správanie sa kamery v scéne,
- *Character Scripts* – tu sa nachádza spomínané skripty,v ktorých je ovládanie hráčovej postavy
- *Controls* – v hre sme museli implementovať vlastný input systém — v tomto priečinku sa nachádza všetok potrebný kód,
- *Dialogue System Scripts* – implementácia dialógového systému,
- *Quest Scripts* – implementácia questov,
- *Enemy Scripts* – v tomto priečinku sa nachádzajú skripty, pomocou ktorých sú implementovaní nepriatelia — bežní, ale aj bossovia. Tu sa taktiež nachádza implementácia stavového automatu,
- *Item Scripts* – implementácia systému predmetov,
- *Inventory Scripts* – kód súvisiaci s implementovaním inventáru,
- *Spell Scripts* – implementácia kúziel v hre,
- *Save Load Scripts* – v tomto priečinku sa nachádza kód, pomocou ktorého je implementované ukladanie a načítavanie v hre,
- *Other* — všetok ostatný kód – napríklad implementovanie pohybujúcich sa platforiem, brán, blokátorov a pod.

Taktiež sme v práci intenzívne rozširovali editor. Všetok kód, ktorý k tomu slúži musí byť v priečinku Editor. V našej práci konkrétne Assets/Editor.

4.4.1 Character skripty

V tejto sekcii budeme písať o skriptoch, ktoré súvisia s ovládaním postavy a nachádzajú sa len na `GameObject` hráčom ovládanej postavy. Nachádza sa tu 7 skriptov:

- **`CharacterMovementController.cs`**,
- **`CharacterInventoryController.cs`**,
- **`CharacterEquipmentController.cs`**,
- **`CharacterAttackController.cs`**,
- **`CharacterSpellController.cs`**,
- **`CharacterHealthController.cs`**,
- **`CharacterQuestController.cs`**.

CharacterMovementController

`CharacterMovementController` pracuje v dvoch fázach:

- práca so vstupom od hráča,
- vypočítanie pohybu a posunutie **`GameObjectu`** postavy.

`CharacterMovementController` sa teda stará o všetko to, čo sa týka pohybu postavy. Zvyšné komponenty s ním komunikujú vtedy, keď potrebujú informáciu o tom, či sa postava nachádza na zemi (nie je vo vzduchu) a taktiež komunikuje s plátnom za účelom prekreslenia UI (užívateľského rozhrania)..

CharacterInventoryController

`CharacterInventoryController` je zodpovedný za správu hráčovho inventára. Prostredníctvom **`CharacterInventoryController`** pristupujú k hráčovmu inventáru questy, NPC a dialógy a taktiež jednotlivé predmety v leveli, ktoré môže hráč zodvihnúť. Rôzne objekty môže zaujímať to, či sa v hráčovom inventári nachádzajú nejaké predmety, čo sa rieši práve prostredníctvom neho. O implementácii inventára budeme písať viac v sekcii ....

CharacterEquipmentController

`CharacterEquipmentController` pracuje s hráčovým vybavením. Samotná logika presúvania predmetov a toho, či predmet je možné do vybavenia vložiť si rieši asset, ktorý reprezentuje inventár hráčovho vybavenia sám. **`CharacterEquipmentController`** pomocou udalostí reaguje na tieto zmeny vo vybavení. Ostatné Character skripty s ním komunikujú vtedy, keď potrebujú získať informáciu o nasadených predmetov (pri útočení, utŕžení poškodenia od nepriateľov). Taktiež je zodpovedný za zobrazovanie nasadenej zbrane.

CharacterAttackController

Pomocou `CharacterAttackControlle`r postava útočí a zodpovedá za poškodenie nepriateľov. Komunikuje hlavne s `CharacterEquipmentControllerom` z dôvodu toho, aby mohol hráč útočiť len vtedy keď má nasadenú zbraň. Taktiež komunikuje s `CharacterMovementControllerom` a `CharacteromSpellControllerom`, aby zistil, či sa hráč nachádza na zemi a nezosiela práve kúzla.

CharacterSpellController

V `CharacterSpellControllere` je implementované hráčove používanie kúziel. Drží v sebe referenciu na špeciálny „inventár kúziel", v ktorom sú uložené informácie o tom, ktoré kúzla sa hráč naučil, ktoré kúzla ma zvolené a aktívne. Reaguje na hráčov vstup a zodpovedá za samotné používanie kúziel a taktiež prekresľovanie UI. Komunikuje najmä s `CharacterAttackControllerom` a `CharacterMovementControllerom`.

CharacterHealthController

`CharacterHealthController` implementuje hráčov systém života rieši poškodenie od nepriateľov a správne prekresľovanie UI. Z ostatných Character skriptov komunikuje najmä s `CharacterEquipmentController` za účelom získania informácií o hráčových nasadených predmetoch. Zodpovedá za ukončenie hry pri hráčovej smrti prostredníctvom LoadManager. Využívať ho môžu rôzne predmety v hre, ktoré nejakým spôsobom pracujú s hráčovým životom – obnova, nastavenie maximálneho života.

CharacterQuestController

`CharacterQuestController` rieši väčšinu vecí spojených s questami v hre. Prostredníctvom neho hráč prijíma questy a pri prijatí questu ho „setupne". O questoch budeme písať podrobnejšie v sekcii Skripty a kód implementujúci questový systém Taktiež s ním komunikuje LoadManager pri ukladaní informácií o questoch pri ukladaní hry. Taktiež je zodpovedný za vykresľovanie UI pre prácu s questami.

4.4.2 Controls

V hre sme si potrebovali vytvoriť vlastný systém ovládania z dôvodu obmedzení od Unity. Prvou časťou tohto systému je `ScriptableObject`, uložený v podobe assetu. Ten v sebe obsahuje list dvojíc – meno a klávesa.

Druhou časťou je `GameObject` InputManager, ktorý sme už rozoberali pri štruktúre levelov. Nachádza sa na ňom skript `PlayerInputManager.cs`, ktorý spomínaný asset sprístupňuje `GameObjectom` v scéne. Skripty, ktoré sa potrebujú pýtať na hráčov využívajú InputManager, kde sa môžu pýtať na vstup podľa jeho mena. InputManager ho potom „preloží" na konkrétnu klávesu. Ide najmä o Charakter skripty (`CharacterMovementController`, `CharacterAttackController`), ale aj napríklad aj skripty, ktoré riešia otváranie rôznych menu (Pause Menu) či skripty, ktoré riešia zobrazovanie dialógu.

Všetok hráčov vstup teda prechádza cez InputManager.

4.4.3 Skripty a kód implementujúci dialógový systém

O tom ako by sme chceli mať dialógový systém implementovaný sme písali intenzívne už v kapitole Analýza. Samotný dialóg je `ScriptableObject`, teda uložený v podobe assetu. Je tvorený listom prehovorov, kde prehovor je normálna C# trieda, obsahujúca text prehovoru a maximálne dva prechody. Prechod podobne obsahuje svoj text a k tomu ešte index prehovoru, do ktorého by sa malo prejsť a list dialógových akcií.

DialogueAction

Dialógové akcie sú najzaujímavejšou časťou celého dialógu. Taktiež sme sa u nich rozhodli využiť `ScriptableObejcty` a ide teda vlastne rovnako o assety, ktoré nie sú prepojené so žiadnou scénou. Z toho dôvodu využívajú dostupné manažéry, ak si potrebujú ukladať odkazy na `GameObjecty` v scéne vo forme ich indexov v manažéroch.

Základná trieda pre dialógové akcie sa nachádza v `DialogueAction.cs` Ďalej sa v projekte nachádza enum `DialogueActionType`, kde je pre každý druh dialógovej akcie (teda pre každú podtriedu) jedna konštanta. Tento enum sa využíva pri tvorbe dialógov pomocou grafového editora, o ktorom budeme písať neskôr v sekcii

DialogueController

Dialógy sú do jednotlivých scén prinášané pomocou skriptu `DialogueController.cs`. Tento skript v sebe drží referencie na asset dialógu a všetku logiku už rieši on – vykresľovanie dialógu na plátno, vypisovanie textu prehovorov, ponúkanie možných prechodov hráčovi, vykonávanie jednotlivých akcií a progres dialógom. Všetky potrebné informácie (na ktorom prehovore sa hráč nachádza, ktorý prechod asi hráč vybral) sa nachádzajú jedine v DialogueControlleri. Samotný dialóg slúži len ako kontajner informácií.

DialogueManager

Keďže DialogueController je normálny skript, ktorý je ako komponent na nejakom `GameObjecte`, potrebujú `ScriptableObjecty` – úlohy, odmeny v questoch ale aj napríklad samotné dialógové akcie mať spôsob sa na ne odkazovať. K tomu slúži vyššie spomínaný DialoguesManager. Ten na sebe obsahuje jednoduchý `DialogueManager.cs` skript, ktorý obsahuje indexované referencie na DialogueControllery v scéne.

4.4.4 Skripty a kód implementujúci questový systém

Podobne ako u dialógov, otázky implementácie questového systému sme rozoberali už v kapitole Analýza. Questy sme sa nakoniec rozhodli používať ako

`ScriptableObjecty` a je implementovaný v triede `QuestInfo`, takže sa v projekte ukladajú ako samostatné assety. Skladajú sa z úloh a odmien. Úlohy a odmeny sú rovnako implementované ako `ScriptableObjecty`. Questy hráč prijíma pomocou `CharacterQuestControlleru`.

Odmeny za questy

V prípade odmien je štruktúra jednoduchá — po dokončení všetkých úloh sa zavolá príslušná metóda odmeny, ktorá v prípade potreby komunikovať s `Game Objectami` v scéne využije potrebné manažéry. Základnou triedou je `QuestReward` a volá sa metóda `void Reward()`. Taktiež sa podobne ako u dialógových akcií nachádza v projekte enum `QuestRewardType`, kde je pre každý typ odmeny jedna konštanta — rovnako využívaná pri tvorbe questu v grafovom editore.

Úlohy v queste

Pre úlohy je pripravená základná trieda `QuestTask` a enum `QuestTaskType`. Zo všetkých `ScriptableObjectov` sú práve questové úlohy tie, ktoré využívajú manažéry a komunikujú so scénou najviac. Pri dialógovom systéme sme písali o tom, že samotný dialóg je len kontajner informácií a pristupujeme k nemu cez skripty. Questový systém je však implementovaný inak. Pri prijatí questu je `CharacterQuestController` zodpovedný jedine o naštartovanie „prípravy“ questu. Pod prípravou si môžeme predstaviť všetky kroky potrebné k tomu, aby quest správne fungoval. Po príprave (nastavení questu) už komunikujú `GameObjecty` v scéne priamo s assetmi jednotlivých úloh. Príprava questu taktiež nie je nič iné ako príprava jednotlivých úloh. V obidvoch prípadoch využívame udalosti, konkrétne udalosť `OnTaskFinish`, ktorá sa vyvolá pri dokončení úlohy. Túto udalosť využíva nielen samotný quest k tomu, aby vedel, kedy sa splnia všetky úlohy, ale taktiež aj jednotlivé úlohy a `CharacterQuestController`. Úloha môže mať v rámci questu prerekvizity (úlohy, ktoré musia byť splnené predtým ako sa začne plniť ona sama) a príprava takejto úlohy je možná až po splnení všetkých jej prerekvizít. Úloha sa teda pri každej prerekvizite pozrie na to, či už bola dokončená a ak nie, tak sa zaregistruje na jej udalosti a pri jej splnení o tom dostane informáciu. `CharacterQuestController` sa na udalosti registruje z dôvodu toho, že je zodpovedný za prekresľovanie UI.

Udalosti sú taktiež primárnym spôsobom, ktorým komunikujú úlohy s `GameObjectami`. Napríklad taký `DialogueController` ponúka až 3 rôzne udalosti, pomocou ktorých môže úloha dostávať informácie o tom ako ju hráč plní.

Progres v questoch sa takiež ukladá, LoadManager k nim pristupujem prostredníctvom `CharacterQuestControlleru`, aby ukladal informácie len o aktívnych questoch.

4.4.5 Skripty implementujúce nepriateľov

Všetky skripty a kód, ktoré riešia implementáciu nepriateľov sa nachádzajú v priečinku Assets/Scripts/EnemyScripts.

Stavový automat

V súbore StateMachine.cs sa nachádza implementácia stavového automatu, pomocou ktorého v hre simulujeme správanie bežných nepriateľov — tých, ktorých môžeme stretnúť počas prechádzania levelu. Stavový automat obsahuje Dictionary<Type, IState>. IState je interface pre všetky stavy a obsahuje jednu metódu Type Tick(). V nej je naimplementovaná logika stavu a taktiež nám hovorí tom, do akého stavu sa má prejsť — v prípade, že sa stav meniť nemá mala by metóda vratiť null. Na stavovom automate je najdôležitejšia funkcia void Update(), ktorá zavolá na aktívnom stave (stave, v ktorom sa automat nachádza) Type Tick() a podľa návratovej hodnoty zmení aktuálny stav. V prípade, že pre návratovú hodnotu sa nenachádza vo vyššie spomínanom Dictionary stav, automat sa prepne do DoNothingState, kde už z názvu vyplýva, že tento stav nič nerobí a nikdy do žiadneho iného stavu neprechádza. Mohli by sme síce tento problém riešiť aj inak, napríklad by sme ostali v stave alebo sa presunuli do nejakého náhodného stavu, ale prišlo nám, že ide o celkom zásadnú chybu, ktorú vývojár hneď spozoruje, keď sa, v našom prípade, nepriateľ zasekne a prestane reagovať.

Stavový automat pre nepriateľa

Na GameObjecte nepriateľa budeme mať teda skript, ktorý bude takýto automat obsahovať, pri načítaní levelu nainicializuje stavy a automat spustí a bude na ňom opakovane volať metódu Update(). Najdôležitejšou úlohou nepriateľa v hre je útočiť po hráčovi. Skript s automatom si teda bude držať referencie na GameObject hráča a pri vytváraní stavov im ho teda predá pomocou parametra v konštruktore stavu. Útok na hráča potom predstavuje to, že sa počas útočnej animácie skontroluje, či nepriateľ hráča nezasiahol a v prípade, že áno, zavolá na CharacterHealthControllere metódu TakeDamage(float dmg).

U nepriateľov však potrebujeme mať možnosť na nich útočiť. Z tohto dôvodu je v projekte jednoduchý interface IDamagable s jedinou metódou TakeDamage(float dmg). Aj samotný CharacterHealthController tento interface splňuje. V Unity sa môžeme pýtať na to, či sa na GameObjecte nachádza skript splňajúci nejaký konkrétny interface a potom na ňom jednotlivé metódy z interfacu volať. Nepriatelia teda na sebe majú skript, ktorý tento interface splňuje a pri útočení môže pomocou neho hráč nepriateľa zasiahnuť.

Poslednou vecou, o ktorej by sme chceli písať je to, akým spôsobom spolu komunikujú questové úlohy a nepriatelia. V kapitole Analýza sme písali tom, že sme chceli mať v hre pripravený systém života, ktorý by mohli využívať rôzne entity v hre — samotná hráčova postava, nepriatelia, ale aj bossovia. Tento systém sa nachádza v HealthSystem.cs a využíva veľké množstvo udalostí, vďaka čomu si ho môžu rôzne skripty prispôsobovať — CharacterHealthController potrebuje pri zmene zdravia prekresliť UI, pri zvýšení zdravia o tom hráča informovať zmenou farby a pri smrti ukončiť hru. U nepriateľov nám potom využitie tohto systému umožní to, že sa questová úloha môže registrovať na udalosť OnDeath a dostať tak informáciu o tom, že sme nepriateľa porazili.

Bossovia

Bossov, ktorí ukončujú level sme implementovali trochu inak. Ak potrebujeme v Unity nejaký objekt animovať, pridáme na jeho `GameObject` komponent `Animator`. V ňom potom môžeme mať poprepájané rôzne animácie a z kódu potom rôznymi spôsobmi nastavovať to, ktoré animácia sa má prehrávať. často potom potrebujeme to, aby nám zavolanie nejakého kódu sedelo s práve hranou animáciou — už spomínané útočenie nepriateľov, ale aj hráča, kde by sme chceli stále v rovnaký moment animácie zavolať kód, ktorý by riešil napríklad to, či sme niekoho zasiahli.

`Animation Event` je presne to. Prostredníctvom neho dosiahneme to, že vieme zavolať metódu zo skriptu na `GameObjecte`, na ktorom je samotný `Animator` v presne stanovenom bode v animácii a nemusíme si to z kódu nejak počítať podľa toho, koľko času už prešlo od začiatku animácie. V našej hre sú bossovia statický, súboj s nimi sa vždy odohráva v presne stanovenom priestore a v skripte na bossovi nám stačí len kontrolovať to, či by sa nemala zmeniť animácia (napríklad mohol hráč prísť príliš blízko alebo mohlo zdravie bossa spadnúť pod nejakú hodnotu) a kód, ktorý zosiela kúzla alebo kontroluje, či nebol zasiahnutý hráč je volaný automaticky s animáciou.

4.4.6 Implementácia systému predmetov

V kapitole Analýza sme si odôvodnili implementovanie predmetov ako `ScriptableObjecty`. Základnou triedou je `ItemObject`, ktorého štruktúru môžeme vidieť na obrázku 4.4.

```
public class ItemObject : ScriptableObject
{
    public Sprite uiSprite;
    public bool isStackable;
    public int id;
    public ItemType type;
}
```

Obr. 4.4: ItemObject

`ItemObject` teda obsahuje naozaj len to najzákladnejšie, aby sme mohli s predmetmi pracovať v inventári. `ItemType` je enum, korý využívame, keď si hráč predmety nasadzuje. Id je číslo, ktoré sa využíva pri ukladaní a načítavaní hry. Na to, aby ho mali všetky predmety jedinečné sa využíva `ItemDatabaseObject`, asset, ktorý slúži ako databáza pre predmety. Po vytvorení nového predmetu je potrebné ho vložiť do databázy, ktorá mu id priradí.

Ďalej sa v hre nachádzajú základné triedy pre jednotlivé typy vybavenia, keďže od nich požadujeme extra informácie — u meča (zbrane) to, aké ma poškodenie, dosah a taktiež to aký `GameObject` sa má zobraziť, keď si zbraň nasadíme. V hre využívame ten spôsob, že všetky takéto zbrane sa už v scéne nachádzajú a pri nasadzovaní meča sa rozhoduje o tom, ktorý z nich sa bude

zobrazovať a vykresľovať. O prácu s vybavením sa stará vyššie spomínaný **CharacterEquipmentController**.

S predmetmi pracuje taktiež questový a dialógový systém, kde u dialógov môžeme mať napríklad akciu, ktorá nám do inventára pridá nejaký predmet a u questov môžeme mať úlohu doniesť postave istý predmet.

Ďalšou časťou tohto systému sú predmety „v scénach“, kde však ide o obyčajné **GameObjecty**, ktorá na sebe majú skript **GroundItem** a **Collider** nastavený na trigger. Tento skript si drží referencie na nejaký **ItemObject** a keď hráčova postava do triggeru vôjde, tento predmet sa hráčovi pridá do inventára. **PickupTask** je questová úloha, ktorej cieľom je takéto predmety pozbierať. Preto sa v scénach levelov nachádza PickableItemsManager, pomocou ktorého sa na ne úlohy odkazujú. Samotný **GroundItem** potom obsahuje udalosť **OnAdd**, na ktorú sa môže úloha registrovať a dostať informáciu o tom, že hráč predmet získal.

4.4.7 Inventárový systém

O inventárovom systéme si napíšeme v dvoch častiach:

- o implementácii logiky inventárov,
- o implementácii užívateľského rozhrania.

Implementácia inventára

Najzákladnejšou časťou nášho inventárového systému je trieda **InventorySlot**, ktorá obsahuje informáciu o tom, aký predmet sa v nej nachádza, aké množstvo tohto predmetu a taktiež to, aké typy predmetov sa v nej môžu nachádzať (využívame enum **ItemType**) spomínaný pri implementácii predmetov.

Ďalej sa jedná o **InventoryObject**, **ScriptableObject**, ktorý obsahuje pole týchto slotov a poskytuje metódy na pridávanie predmetov, zisťovanie toho, či sa predmet v inventári nachádza a odoberanie predmetov. V našej hre sa nachádzajú dva:

- pre náš „batoh“ (21 slotov, môžu do nich ísť ľubovoľné predmety),
- pre naše vybavenie (4 sloty, do každého môže ísť len jeden typ predmetu).

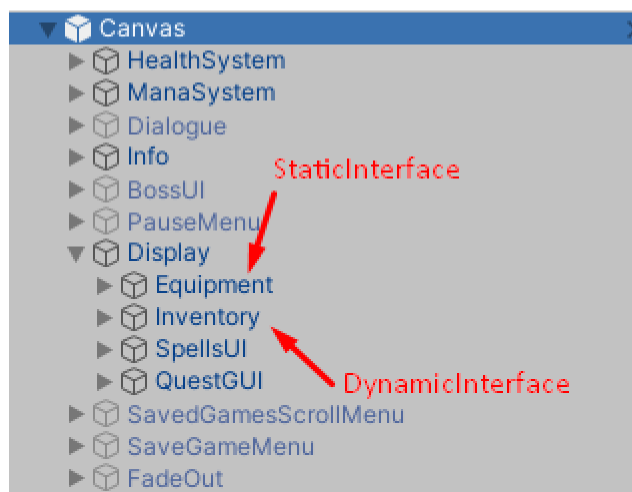
Ak objekty v hre potrebujú komunikovať s hráčovými inventármi, pristupujú k nim prostredníctvom **CharacterInventoryControllera**, ktorý si na ne drží referencie. Pri Charakter skriptoch sme tiež písali tom, že **CharacterEquipmentController** si tieto referencie nedrží. Z tohto dôvodu sú na triede **InventorySlot** k dispozícii dve udalosti:

- **BeforeUpdate**,
- **AfterUpdate**.

Na tieto udalosti sa potom môžu objekty registrovať a dostávať informácie o zmene slotu.

Užívateľské rozhranie pre inventáre

V hre je implementované užívateľské rozhranie, kde je možné predmety preťahovať na rôzne pozície v inventári, medzi inventármi a používať ich. Základom je abstraktný skript `UserInterface`, ktorý má v sebe referenciu na `InventoryObject`, ktorý má vykresľovať. Od neho dedia ďalšie dva skripty — `DynamicInterface` a `StaticInterface`. Tieto skripty sa nachádzajú na `GameObjectoch` na Canvase zobrazené na obrázku 4.5.



Obr. 4.5: Hierarchia Canvasu

Tieto skripty majú pre každý slot referenciu na `GameObject`, na ktorom sa vykresľuje. Dynamic preto, lebo ich skript vytvára a Static preto, lebo v tomto prípade sú už `GameObjecty` vytvorené. Dynamic je pre náš „batoh", Static pre naše vybavenie — u vybavenie chceme mať pre prázdny slot obrázok, ktorý reprezentuje predmet, ktorý do neho môže byť vložený. Tieto skripty sa potom registrujú na udalostiach daných slotov a pri zmene grafiku pre konkrétny slot prekreslia. Na `GameObjecty` samotných interfacov ako aj na tie, ktoré reprezentujú sloty pridáme `EventTriggery`, vďaka ktorým môžeme volať rôzne metódy pri tom, keď hráč s myškou príde nad inventár, klikne na slot, začne slot ťahať. Pri presúvaní predmetov to potom máme veľmi jednoduché — pri začiatku ťahania si uložíme do statickej premennej to, z ktorého slotu sme začali ťahať a vytvoríme na Canvase nový obrázok s predmetom. Pri ťahaní potom tento obrázok presúvame spoločne s myškou, čo dáva hráčovi ilúziu, že predmet ťahá. Pri skončení ťahania o tom dostanem informáciu a ak sme skončili nad nejakým slotom, môžeme sa pokúsiť predmety vymeniť.

4.4.8 Systém ukladania a načítavania

Systém ukladania a načítavania hry (save systém) je časť hry, ktorá potrebuje komunikovať s takmer všetkými ostatnými časťami hry. Pri ukladaní hry si od nich potrebuje nejaké informácie zistiť, pri načítavaní tieto informácie spracovať a hru navrátiť do stavu, pri ktorom sa ukladala. Logika toho, ako sa hra ukladá, načítava a ako sa prechádza medzi levelmi sa nachádza v súbore `SaveManager.cs`. `GameObject`, ktorý tento skript obsahuje sa volá LoadManager a nachádza sa v

scéne hlavného menu. LoadManager sa nezničí pri zmene scén a využíva návrhový vzor singleton.

Ukladanie a načítavanie hry

Priblížime si ako LoadManager ukladanie hry rieši a ako pri tom komunikuje s ostatnými časťami hry. Pri uložení hry sa ukladajú informácie o:

- *predmetoch v leveli* — informácie o nich získava pomocou ich manažéra — PickableItemsManager,
- *dialógoch s postavami* — ukladajú sa informácie o **`DialogueController`** komponentách na postavách v leveli. Samotné dialógy sú assety, ktoré sa nikdy nemenia. K **`DialogueControllerom`** pristupuje LoadManager cez ich manažéra — DialoguesManager,
- *nepriateľoch v leveli* — pristupuje k nim cez EnemyManagera,
- *hráčovi* — informácie o inventári, živote, kúzlach, vybavení, o pohybe hráča a taktiež sa uloží hráčov progres v plnení questov,
- *ostatných objektoch v leveli* — pomocou SelfSavingManagera (pre **`GameObjecty`**), ktoré sa načítavajú so scénou a taktiež využíva SelfSavingRuntimeManagera, ktorý je zodpovedný za ukladanie dynamicky vytváraných **`GameObjectov`**

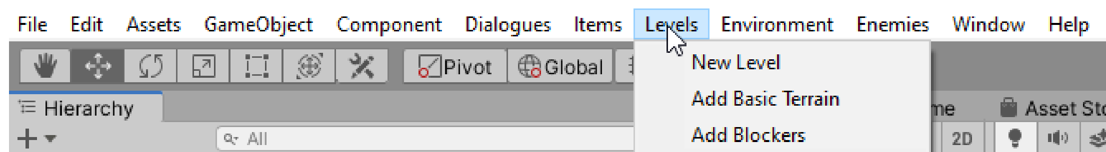
Pri načítavaní je LoadManager zodpovedný za prechody medzi scénami, za načítanie scén a za predanie informácií správnym **`GameObjectom`**. Všetko ide cez jednotlivé manažéry.

Prechod medzi levelmi

V tomto prípade stačí LoadManageru preniesť do ďalšej scény len informácie o hráčovej postave — o inventári, vybavení, naučených kúzlach a vylepšenom zdraví.

4.5 Rozširovanie editora

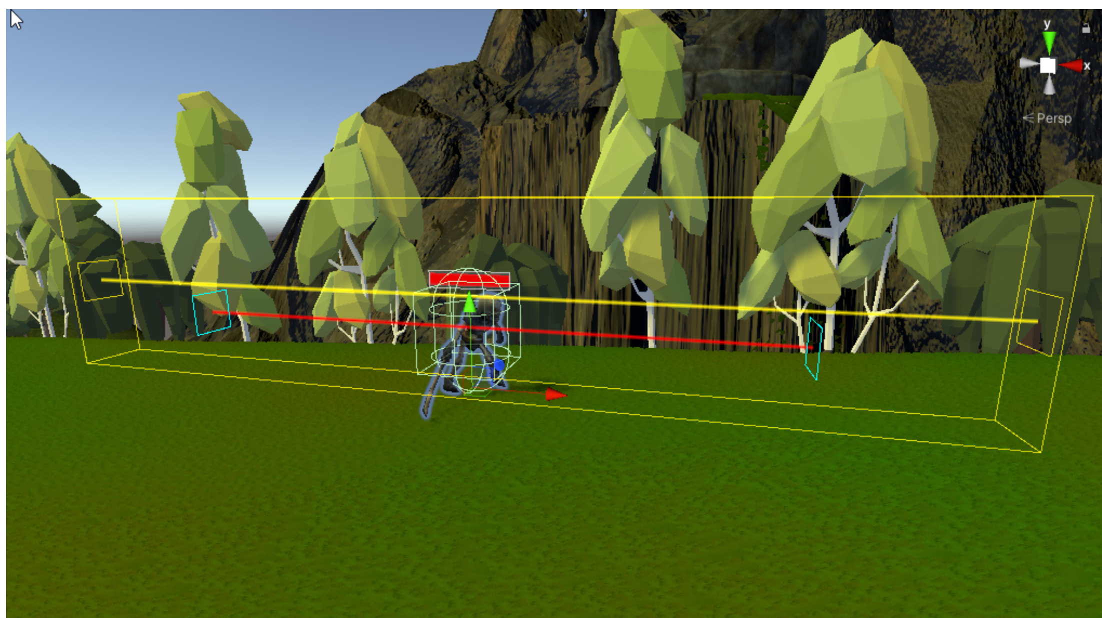
Na konci projektovej dokumentácie sa budeme venovať tomu, ako sme v Unity rozširovali editor. Všetok tento kód sa nachádza v priečinku Assets/Editor. Prvý spôsob, ktorý sme používali bol ten, že sme pridávali možnosti do hlavného menu Unity, obrázok 4.6.



Obr. 4.6: Rozšírené menu

Do tohto menu sa pridáva pomocou **MenuItem** atribútu, ktorý môže stáť len pred statickými metódami. Tento spôsob sme teda pripravili najmä na pridávanie **GameObjectov** do levelov — nepriatelia, predmety, postavy, ale taktiež na vytvorenie nových levelov alebo pridanie komponenty na **GameObject**, ktorý je v scéne práve zvolený. Vďaka tomu, že sme pracovali z kódu, mohli sme hneď takto pridávané **GameObjecty** vkladať do ich príslušných manažérov a nastavovať komponentom premenné.

Ako ďalšie rozšírenie sme do projektu pridali *custom editory* [25]. Vlastné editory nám umožňujú meniť vzhľad inšpektora pre prezeraní objektu, ale taktiež napríklad vykresľovať do scény interaktívne objekty, pomocou ktorých môže dizajnér jednoducho nastavovať rôzne komponenty **GameObjectu**. Príklad môžeme vidieť na obrázku 4.7, kde červené čiary ohraničujú oblasť, ktorú nepriateľ stráži a žltý kváder reprezentuje kam až nepriateľ dovidí.



Obr. 4.7: Pomocné vykresľovanie

Dizajnér môže tieto čiary po scéne ťahať a tým rovno nastavovať nepriateľove vlastnosti.

4.5.1 Grafové editory

Najzaujímavejším rozšírením editora sú grafové editory, cez ktoré je možné vytvárať a do hry pridávať questy a dialógy. Na ich vytvorenie sme použili **GraphViewEditorWindow**, ktoré nám umožnilo jednoduchšie pracovať s hranami a vrcholmi. Všetok kód, v ktorom sa implementuje logika hrán a vrcholov sa nachádza v Assets/Editor/Quests, respektíve Assets/Editor/Dialogues.

Manažéry

Pri vytváraní grafov a dialógov aktívne komunikujeme s manažérmi v práve otvorenej scéne. To robíme z toho dôvodu, aby mal dizajnér ilúziu, že môže do questov a dialógov vkladať **GameObjecty** zo scény. Teda pre dizajnéra sa tvárime, že

je možné do políčka predať priamo odkaz na **`GameObject`** zo scény. V skutočnosti sa však vrchol spýta manažéra, či na tento **`GameObject`** neobsahuje indexovaný odkaz. Ak áno, tak sa bude vrchol tváriť, že sa na danom mieste nachádza na neho referencia, aj keď vnútorne si pamätá len jeho index. Manažéry sa snažia byť múdre a preto, ak sa daný **`GameObject`** v manažéri nenachádza, no mohol by sa (obsahuje potrebnú komponentu) nachádzať, manažér si ho automaticky pridá.

5. Užívateľská dokumentácia

5.1 Dokumentácia pre hráča

Spustenie hry

Hra sa spúšťa kliknutím na aplikáciu MedievalAdventures_BeyondUnknown. Hre je určená pre Windows 10, 32- aj 64-bitové verzie operačného systému. Je potrebné mať aspoň 4GB RAM a 2-jadrový procesor. Grafická karta by mala podporovať DirectX11.

Hlavné menu

Po zapnutí hry hry sa hráč nachádza v hlavnom menu, ktoré môžeme vidieť na obrázku 5.1

Obr. 5.1: Hlavné menu

V hlavnom menu má hráč k dispozícii viacero tlačidiel. Prvým z nich je NEW GAME, pomocou ktorého sa spustí prvý level.

Druhým tlačítkom je LOAD GAME, ktoré umožňuje hráčovi otvoriť menu s uloženými hrami. Toto menu môže byť v dvoch módoch:

- načítavací,
- vymazávací

Načítavací mód môžeme vidieť na obrázku 5.2, vymazávací na obrázku 5.3

V načítavacom móde sa po kliknutí na tlačítko s menom uloženej hry hra načíta, vo vymazávacom sa tým hra vymaže.

Medzi módmi hráč prepína pomocou tlačítka X, menu s uloženými hrami sa zatvára pomocou CLOSE.

Pomocou tlačidla TUTORIAL sa hráč dostane do tutoriálu, v ktorom mu budú vysvetlené herné mechaniky.

Štvrtou možnosťou je tlačítko OPTIONS, stlačením ktorého sa otvorí menu možností. Menu vidíme na obrázku 5.4. V menu si vie hráč nastaviť ovládanie.

Obr. 5.2: Načítavací mód

Obr. 5.3: Vymazávací mód

V tomto menu môže hráč vidieť všetky možné nastaviteľné ovládania.

Obr. 5.4: Options menu

Pre zmenu prvku je potrebné stlačiť tlačítka ktoré sa nachádza vpravo do mena prvku. Po stlačení tlačidla potom stačí stlačiť klávesu, ktorú chceme nastaviť.

Posledným tlačítkom je QUIT GAME, pomocou ktorého sa aplikácia ukončuje.

Pause Menu

Počas hrania levelu má hráč k dispozícii Pause Menu. To otvorí pomocou ovládacieho prvku PAUSE a vidieť ho môžeme na obrázku fig:pausm.

Obr. 5.5: Pause Menu

Stlačením RESUME alebo pomocou ovládacieho prvku PAUSE sa Pause Menu zatvorí.

Ďalším tlačítkom je MAIN MENU, ktorým sa hráč vráti do hlavného menu. Stratí tým však všetok svoj neuložený progres.

Hra sa ukladá pomocou tlačítka SAVE GAME, ktoré otvára ukladacie menu. Prácu s ukladacím menu môžeme vidieť na obrázku 5.6.

Obr. 5.6: Menu na ukladanie hry

Do poľa ENTER SAVE NAME hráč napíše meno, ktoré chce aby reprezentovalo uloženú hru. Ak už uložená hra s takýmto menom existuje, starý save súbor sa prepíše.

Hra sa konečne ukladá kliknutím na tlačítko SAVE. Ak si hráč praje ukladacie menu ukončiť a vrátiť sa do Pause Menu, použije na to tlačítko BACK.

V niektorých situáciách nie je možné hru uložiť, napr. počas boss fightu. V takomto prípade bude tlačidlo deaktivované a zosivené.

Hráč môže z Pause Menu hry rovno aj načítavať. Pomocou LOAD GAME sa mu otvorí menu s uloženými hrami, vidieť na obrázku 5.7. Po kliknutí na tlačidlo s uloženou hrou sa aktuálny level ukončí a načíta sa daný save. Neuložený progres hráč stratí.

Obr. 5.7: Uložené hry z Pause Menu

Poslednou časťou Pause Menu je QUIT GAME, ktoré hru ukončí. Podobne ako pri ostatných akciách hráč svoj neuložený progres stratí.

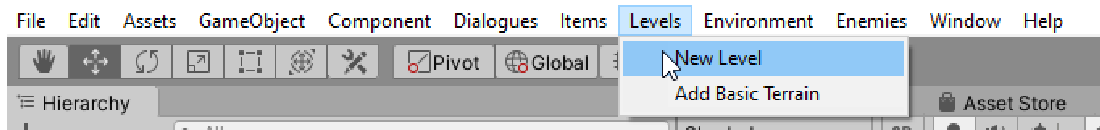
5.2 Dokumentácia pre dizajnéra

5.2.1 Level

Prvým krokom je príprava nového levelu. Keďže v hre sú jednotlivé levely rozdelené do scén je potrebné vytvoriť scénu. Veľa vecí a objektov je však už dopredu daných - napríklad manažéry, hráč, kamera a plátno. Na obrázku 5.8 môžeme vidieť ako sa takýto level do hry pridá. Z horného menu editora klikneme na Levels/NewLevel. Chvíľku počkáme, kým sa scéna pripraví a automaticky zobrazí v editore.

5.2.2 Mapa

Z dôvodu toho, že hra je logicky v 2D sú k dispozícii dve možnosti ako vytvárať mapu. Prvou je terén, ktorý by mal slúžiť ako základ. Pri použití výhradne terénu

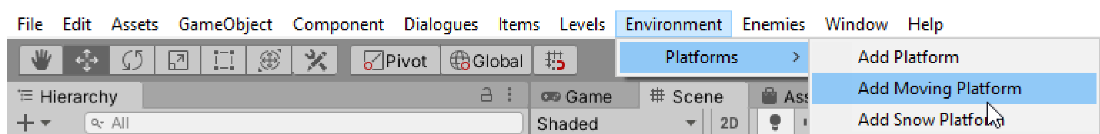


Obr. 5.8: Pridanie nového levelu

by sme však stratili možnosť pohybu hore a dole, členitosť mapy. To môžeme dosiahnuť využitím platforiem.

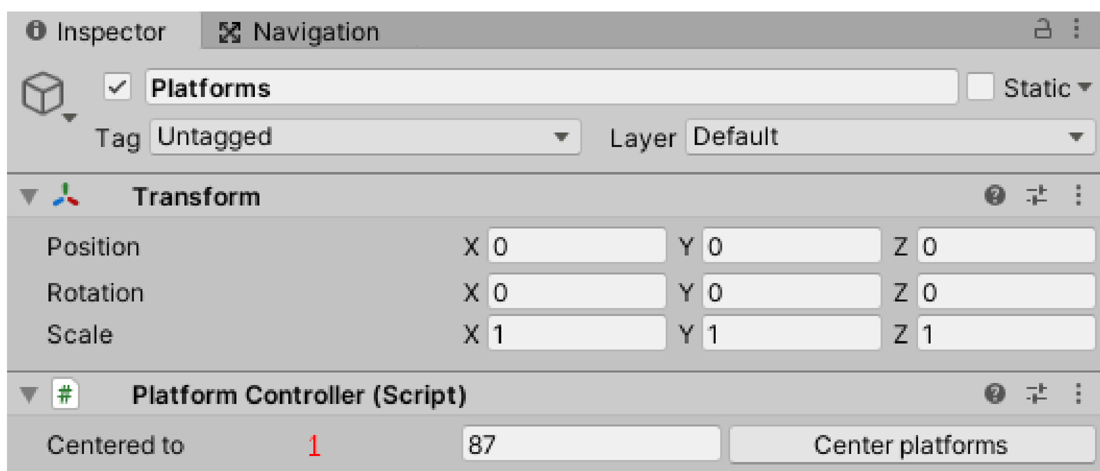
Platformy

K dispozícii sú dopredu vytvorené viaceré platformy. Do scény sa vkladajú cez menu Environment->Platforms->Konkrétna Platforma



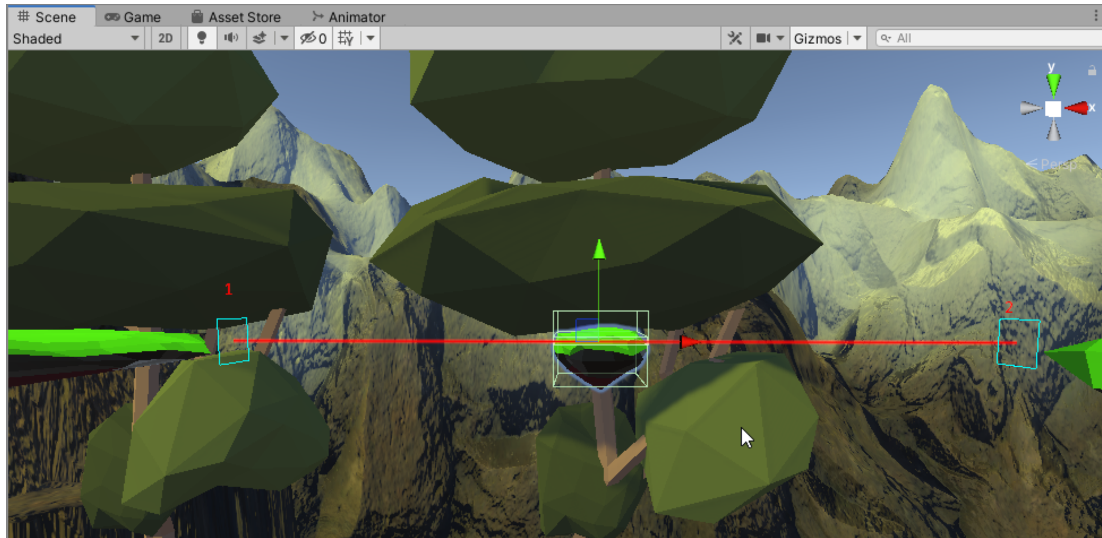
Obr. 5.9: Pridanie platformy

V predpripravenej scéne je **`GameObject`** Environment. Jeden z jeho potomkov sa nazýva Platforms. Všetky platformy pridané do hry by mali byť v hierarchii scény potomkom Platforms. Platformy pridané z menu sú takto nastavené automaticky. Keďže sa hráčov pohyb odohráva len v 2D, treba zaručiť, aby boli všetky platformy v rovnakej línii ako hráč. Na obrázku 5.10 môžeme vidieť inšpektor Platforms. Pomocou neho môžeme nastaviť, do akej línie sa vkladané platformy nastavujú a môžeme tiež všetky platformy vycentrovať (1).



Obr. 5.10: Inšpektor Platforms

K dispozícii sú aj pohyblivé platformy. U pohyblivých platforiem je možné rovno v editore nastavovať body, po ktoré sa platforma posúva. Na obrázku 5.11 vidíme, čo sa nám vykreslí, keď si takúto platformu zvolíme. Pomocou modrých obdĺžnikov (1,2) potom môžeme pohyb platformy priamo upravovať.

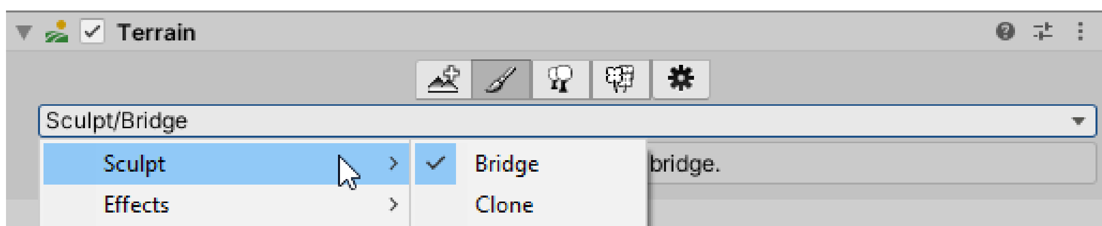


Obr. 5.11: Pohybujúca sa platforma v editore

Terén

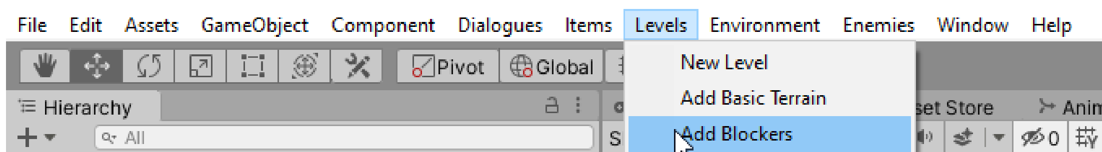
Práca s terénom je trochu zaujímavejšia. Treba si totiž uvedomiť to, že jeho nesprávne použitie môže pokaziť a znefunkčniť celý level. Nevhodne vytvorený a nevyrovnaný terén by totiž mohol spôsobiť to, že postava sa zosunie z hlavnej línie. Potom by mohlo byť nemožné napríklad trafiť nepriateľov, využiť pripravené platformy a v podstate dokončiť level. U platforiem tento problém nenastáva.

Pri tvorbe terénu teda musíme dávať pozor na to, aby bol terén v hlavnej hernej línii (línia, v ktorej sa pohybuje hráč, nachádzajú sa platformy a nepriatelia) správne vyrovnaný. K tomu je možné použiť napríklad *Bridge* (nájdeme ho v inšpektore terénu, obrázok 5.12), ktorý nám umožní vyrovnať terén medzi dvoma miestami.



Obr. 5.12: Vyrovnanie pomocou Bridge

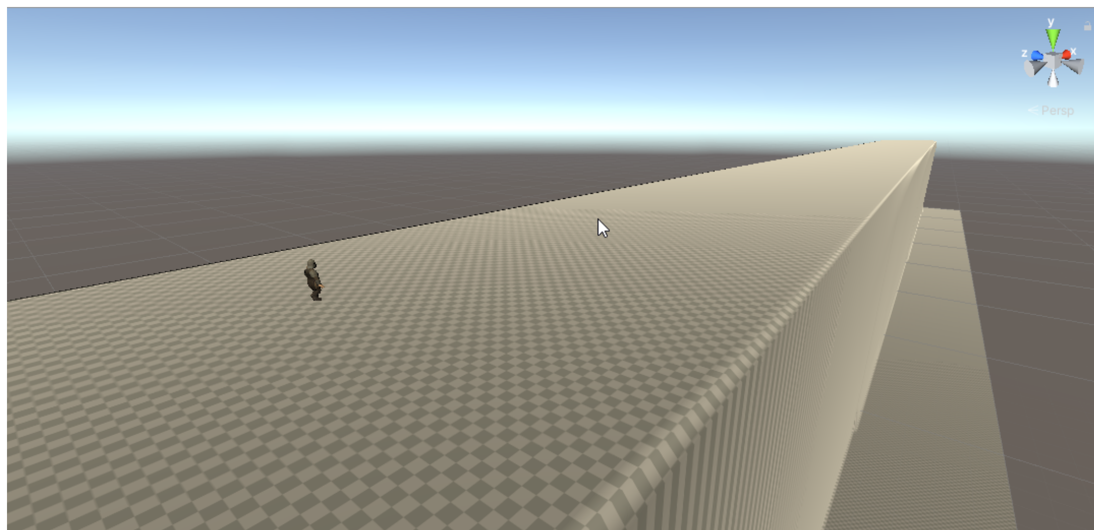
Ak by nám tento spôsob nevyhovoval, je možné využiť *blokátory*. Blokátory lemujú hlavnú hernú líniu a teda neumožnia hráčovi z nej vybočiť. Blokátory je možné do scény pridať pomocou menu cez Level/Add Blockers, obrazok 5.13.



Obr. 5.13: Pridanie blokátorov do levela

Pridanie terénu

V scéne, ktorú sme si vytvorili sa terén nenachádza. Môžeme ho tam pridať z menu cez Levels/Add Basic Terrain. Tento terén už má v sebe vytvorenú základnú cestu pre hráča, obrázok 5.14.



Obr. 5.14: Scéna po pridaní terénu z menu

Ak chceme začať s úplne novým terénom, vieme ho pridať z menu otvorením okna pre Terrain Toolbox — Window/Terrain/Terrain Toolbox.

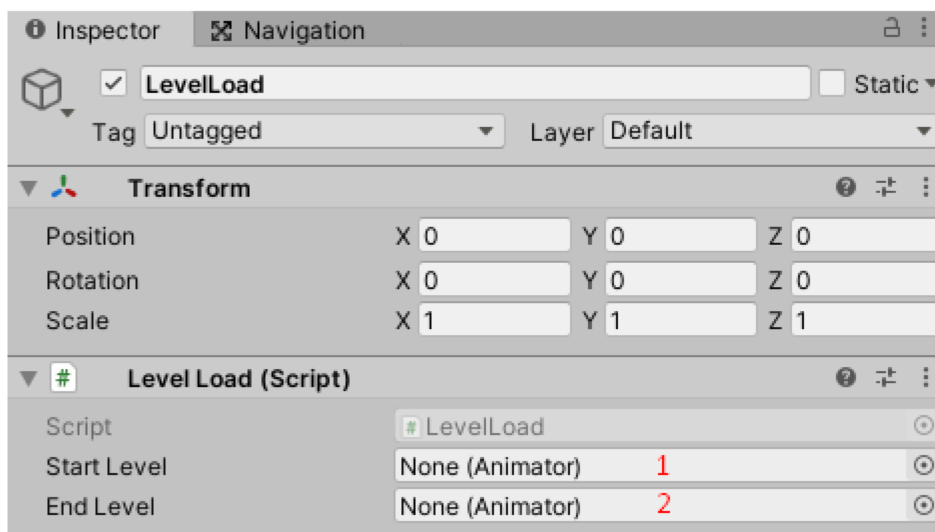
Úpravu terénu potom riešime cez inšpektor terénu.

5.2.3 Prechody medzi levelami

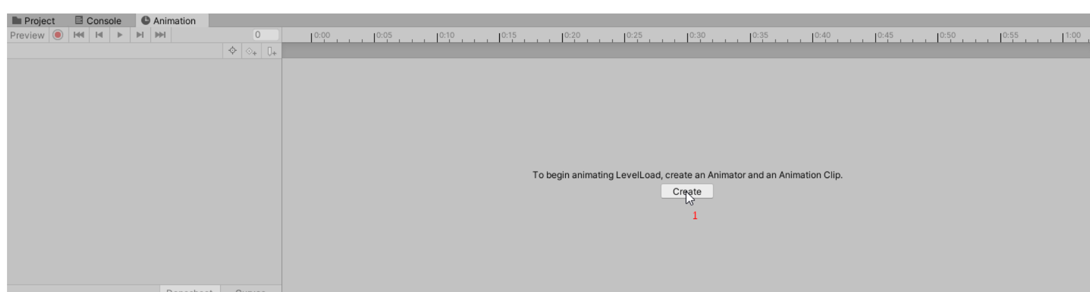
S levelmi ešte súvisia animačné klipy pri konci a začiatku levelu. Ich prehranie je automatické a stará sa o ne systém, ktorý rieši prechody medzi levelmi. V scéne sa nachádza **`GameObject`** LevelLoad. V jeho inšpektore (obrázok 5.15), v časti Level Load môžeme vidieť dva odkazy na **`GameObjecty`** s animátorom. Start level (1) je pre animátora, ktorý obsahuje klip pre začiatok levelu, End Level (2) je pre koniec levelu. V inšpektore **`GameObjectu`** je v časti Animator potrebné nastaviť Update mode na Unscaled Time.

Ak chceme na **`GameObjecte`** pripraviť animátora, zvolíme v hierarchii scény daný **`GameObject`**, a pomocou okna Animation (vieme ho otvoriť z menu cez Window/Animation/Animation) začneme animovať (obrázok 5.16), pomocou Create (1).

K tomu, aby to celé fungovalo je ešte do animátora pridať trigger Start pre animátora začiatku levelu a End pre animátora konca levelu. Systém, ktorý sa stará o spustenie animácií používa práve tieto triggery. Môžeme teda v animátorovi vytvoriť prázdny stav (v okne s klipmi kliknutie pravého tlačidla myši, potom výber Create State/Empty), ten nastaviť ako začiatočný (zvolenie prázdneho stavu, kliknutie pravým tlačidlom myši a Set as Layer Default State) a vytvoriť prechod medzi ním a našou animáciou pri nastavení triggeru (zvolenie prázdneho stavu, kliknutie pravým tlačidlom myši, výber Make Transition a potiahnutie prechodu do nášho stavu). Výsledok môžeme vidieť na obrázku 5.17 —

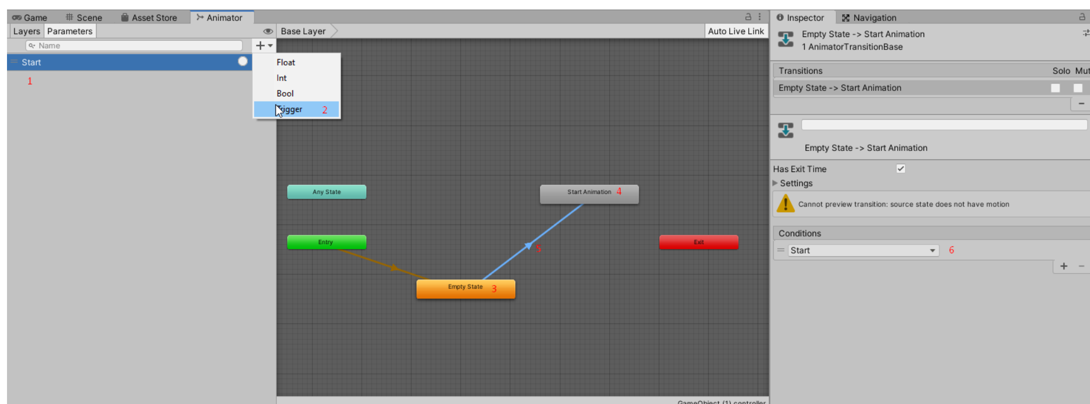


Obr. 5.15: Level Load



Obr. 5.16: Animation okno

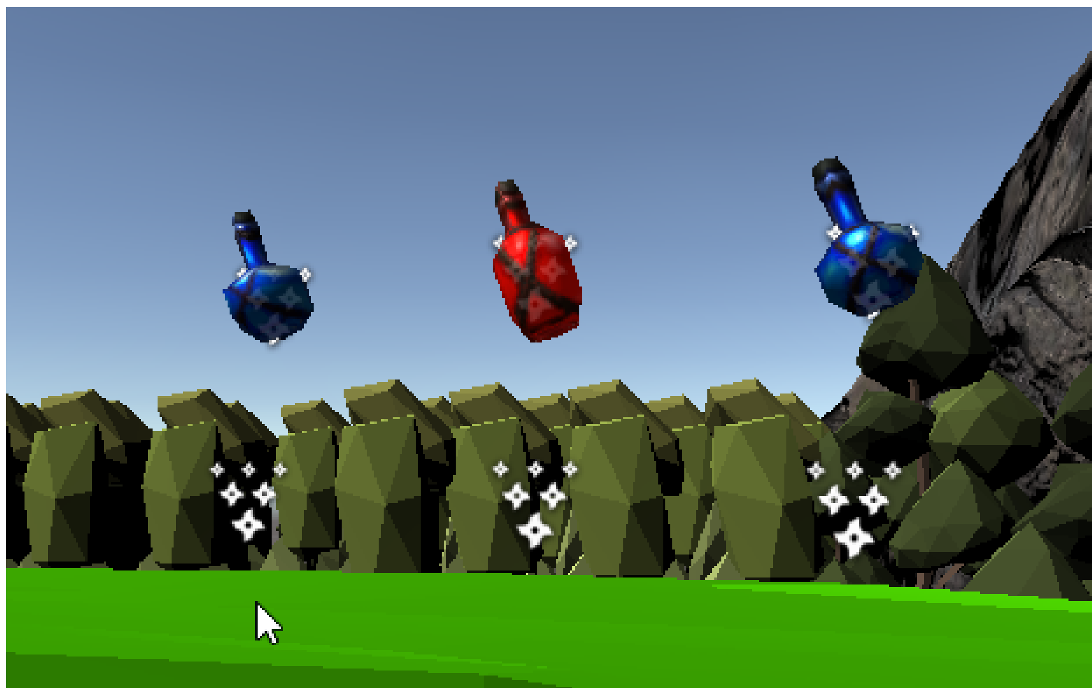
trigger Start (1), pridanie nového triggeru (2), prázdny stav (3), naša animácia (4), prechod medzi stavmi (5) a podmienky zvoleného prechodu (6).



Obr. 5.17: Príklad výsledného animátora pre začiatok levelu

5.3 Predmety

V leveli sa môžu nachádzať predmety, ktoré môže hráč pozbierať, obrázok 5.18.

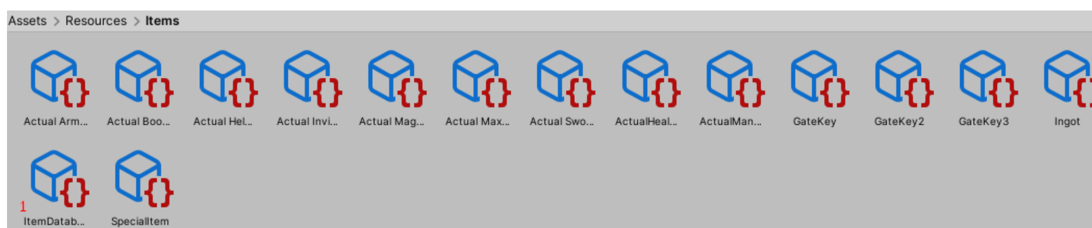


Obr. 5.18: Predmety v leveli

Takéto predmety môžeme do levelu pridávať rovno z menu, pomocou Items/Add/Konkrétny predmet. Vytvorený **`GameObject`** sa bude nachádzať v hlavnej hernej línii. To, ktorý predmet hráč získa môžeme zvoliť v inšpektore **`GameObjectu`**, v časti GroundItem, v položke Item.

Ak chceme z nášho **`GameObjectu`** vytvoriť zdvihnuteľný predmet, označíme náš **`GameObject`** v hierarchii a v menu klikneme na items/Make a pickable item. V inšpektore potom nastavíme zvyšok v časti Ground Item.

Doteraz sme hovorili len o predmetoch v leveli. Skutočné predmety sa nachádzajú v priečinku Assets/Resources/Items (obrázok 5.19). V priečinku sa nachádza aj databáza predmetov (1), ktorá predmetom priraďuje indexy. Ak chceme vytvoriť nový predmet, v menu zvolíme Assets/Create/Inventory System/Items/Typ predmetu. Predmet je po vytvorení pridaný do databázy automaticky. Predmety môžeme upravovať pomocou inšpektora.



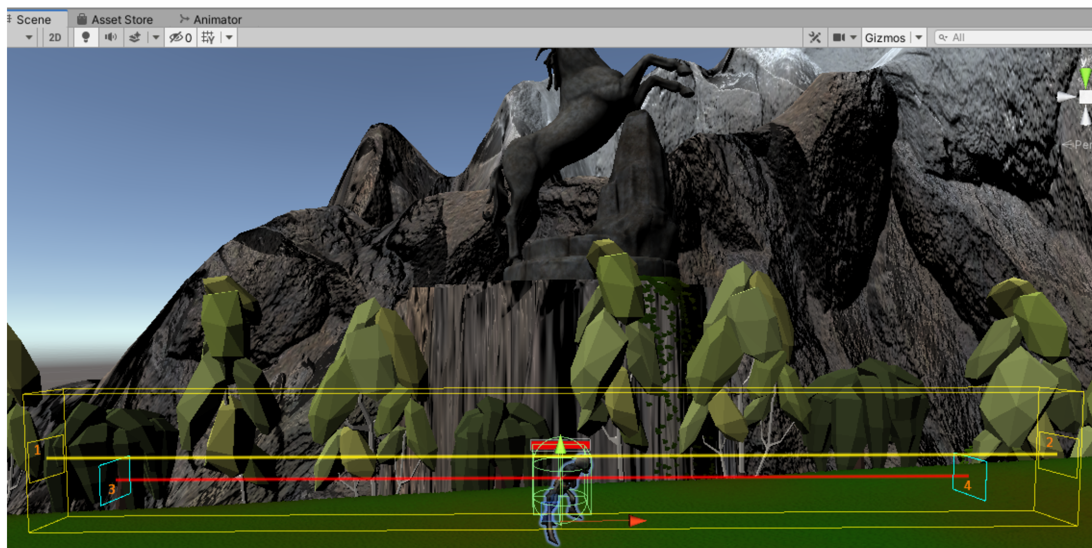
Obr. 5.19: Priečinok s predmetmi

5.4 Nepriatelia

Nepriatelia sa pridávajú do hry cez menu, pomocou Enemies/Add/Typ nepriateľa. Nepriatelia sa taktiež pridávajú do hlavnej hernej línie.

5.4.1 Úprava skeleton nepriateľa

V tejto časti si ukážeme, ako môžeme v editore upravovať skeleton nepriateľa. Na obrázku 5.20 môžeme vidieť ako vyzerá editor keď si zvolíme v scéne nepriateľa.



Obr. 5.20: Zvolený skeleton v scéne

Skeleton hliadkuje oblasť. Červené čiary reprezentujú túto oblasť. Ľavý (3) aj pravý (4) okraj tejto oblasti môžeme nastaviť pomocou presunutia modrých štvorcov. Pravý okraj musí byť vždy napravo od skeletona, ľavý okraj musí byť vždy naľavo.

Žltý kváder reprezentuje oblasť, v ktorej skeleton vidí. To znamená, že ak hráč vojde do tejto oblasti skeleton ho zazrie a začne ho naháňať. Tento kváder sa teda pohybuje so skeletonom.

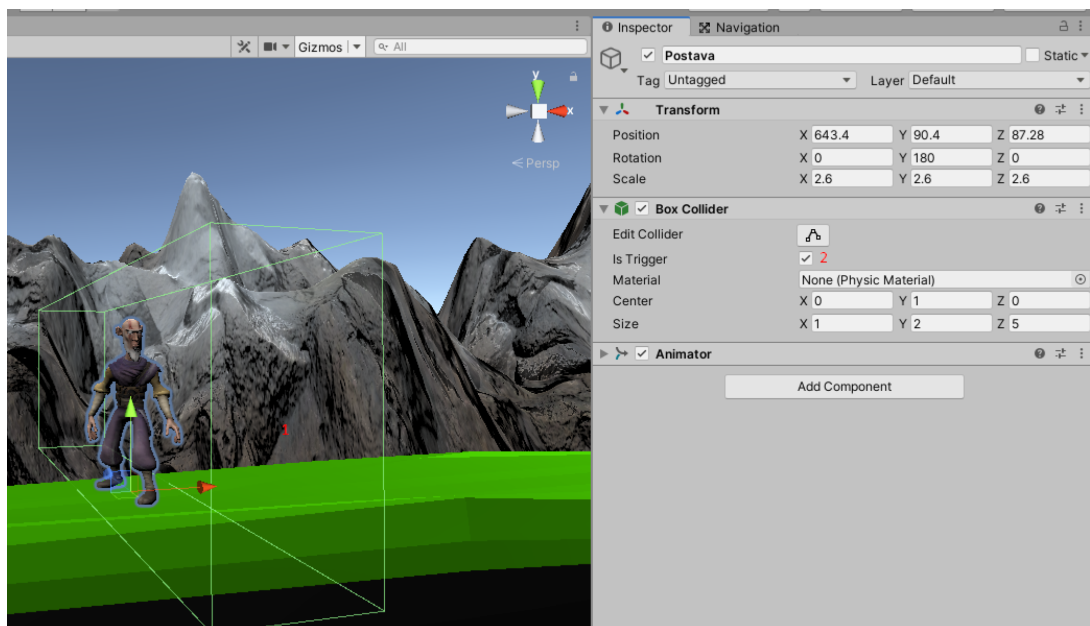
5.5 Dialógy

Súčasťou hry sú dialógy, ktoré môže hráč viesť s ostatnými postavami. V tejto časti si popíšeme ako sa dialógy vytvárajú a prácu s editorom.

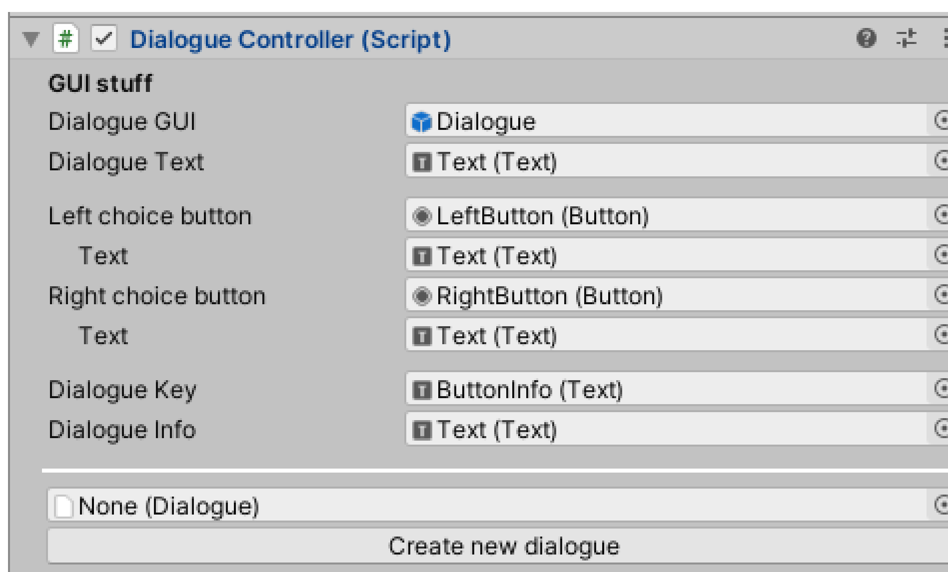
5.5.1 Vytvorenie postavy

Prvý krok, ktorý musíme urobiť je pridanie postavy do hry, s ktorou môže mať hráč rozhovor. K tomu môžeme použiť vlastný **`GameObject`** alebo použiť niektorú z predpripravených postáv. Ak chceme použiť vlastný **`GameObject`**, musí sa na ňom nachádzať **`Collider`**, ktorý je trigger. Ten reprezentuje oblasť, v ktorej môže hráč viesť s postavou rozhovor. Ako by to malo vyzerať môžeme vidieť na obrázku 5.21. Môžeme vidieť **`Collider`** (1) a taktiež to, že je nastavený na trigger (2).

Potom je potrebné pridať postave možno viesť s hráčom rozhovor. V scéne si zvolíme správne nastavený **`GameObject`** a v menu použijeme Dialogues/Add Dialogue Controller. V inšpektore potom uvidíme, že sa nám v inšpektore ukázala časť Dialogue Controller (obrázok 5.22).



Obr. 5.21: Pripravený GameObject

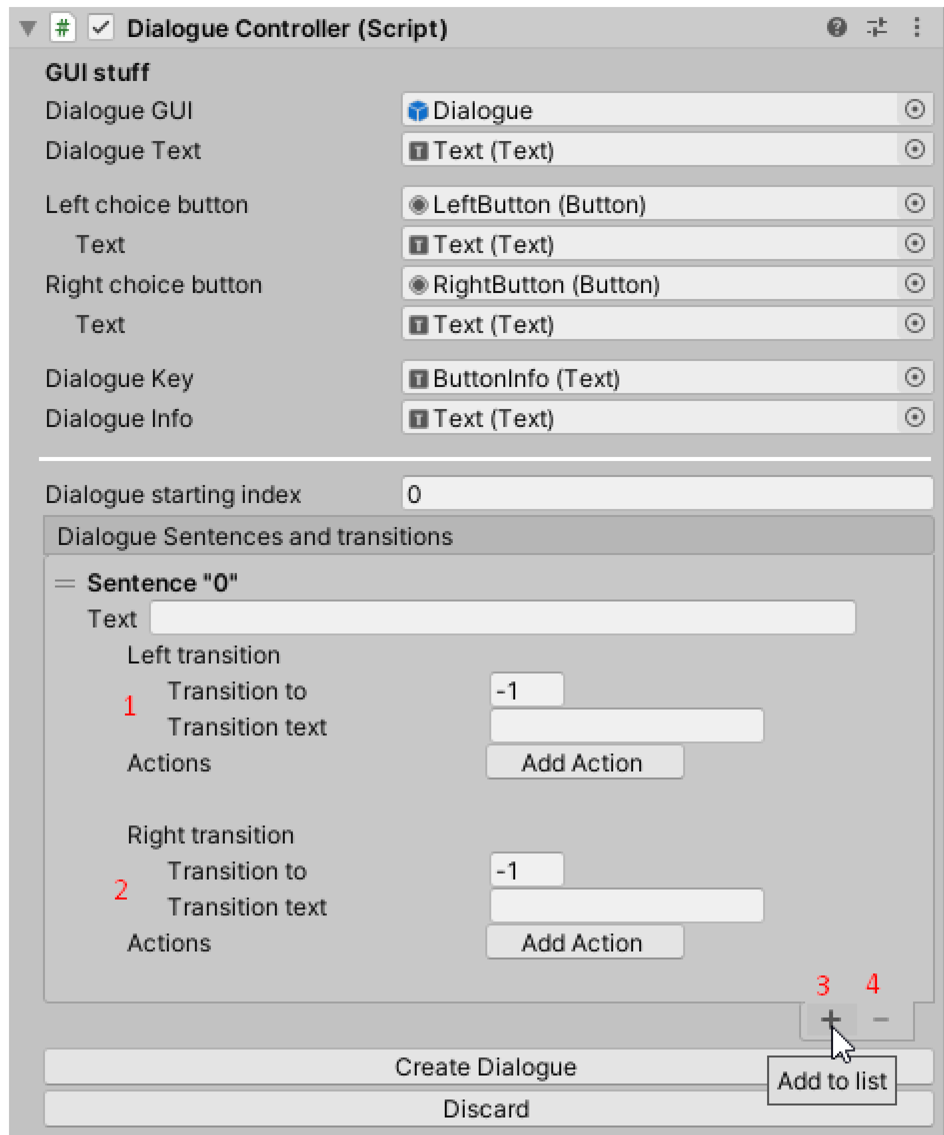


Obr. 5.22: Inšpektor postavy

Pripravené postavy pridáme do scény z menu pomocou Dialogues/Add/Konkrétna postava.

5.5.2 Vytvorenie dialógu

V inšpektore postavy, v časti Dialogue Controller vieme vyberať, ktorý dialóg bude postava obsahovať. Môžeme si vybrať už existujúci alebo vytvoriť nový. Nový dialóg sa otvorí pomocou tlačidla Create New Dialogue. Tým sa nám inšpektor rozšíri a dostaneme sa do režimu pre tvorbu dialógu. Na obrázku 5.23



Obr. 5.23: Režim inšpektora pre vytváranie dialógu

5.5.3 Úprava dialógu

Úprava dialógu cez inšpektor sa však silno neodporúča a dialóg by sa mal cez inšpektor len vytvoriť. Pre dokončenie vytvárania dialógu sa použije tlačidlo Create Dialogue. Odporúčaná sekvencia akcií teda vyzerá takto:

- kliknutie na tlačidlo Create New Dialogue,
- kliknutie na tlačidlo Create Dialogue.

Takto vytvorený dialóg sa hneď nastaví ako dialóg danej postavy.

Výrazne sa teda neodporúča používať inšpektor na úpravu vytváraného dialógu z dôvodu, že pri budúcom rozširovaní dialógového systému sa už nebude pripravovať inšpektor. Taktiež je takáto práca s dialógom neprehľadná. Popíšeme si aj to, ako sa dialóg upravuje ešte v inšpektore, ale nikdy by nemalo byť potrebné to využiť. Používa sa *reordable list*, teda list, v ktorom je možné prehadzovať jednotlivé prvky — v tomto prípade prehovory. Každý prehovor obsahuje text a

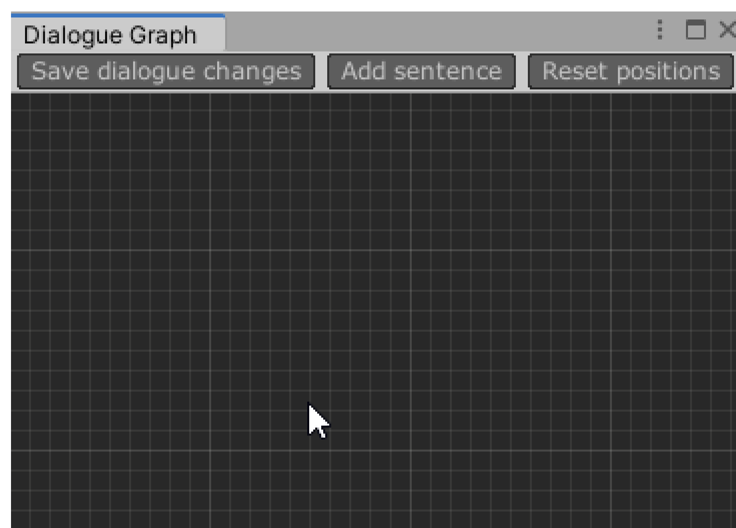
taktiež dva prechody — ľavý (1) a pravý (2). Každý prechod potom obsahuje index prehovoru na ktorý sa ním prechádza (Transition Index). Index rovný -1 znamená, že tento prechod sa nepoužíva. Taktiež obsahuje text (Transition Text) a dialógové akcie, ktoré sa vykonajú pri prechode. Akcie sa pridávajú pomocou tlačidla Add Action. Prehovory sa pridávajú pomocou tlačidla + na konci list (3), zvolený prehovor sa odstráni pomocou - (4). Úpravu dokončíme a dialóg vytvoríme pomocou tlačidla Create Dialogue. Tento dialóg sa nastaví ako dialóg postavy. Ak chceme vytváraný dialóg zahodiť, stlačíme tlačidlo Discard, čím sa ukonči vytvárací režim inšpektora.

Grafový editor dialógu

Ak chceme upravovať dialóg, odporúča sa k dialógu pristupovať len cez postavu. To je z toho dôvodu, že dialógové akcie môžu v iných scénach znamenať niečo iné, je teda dôležité, aby sa dialógy stále upravovali len nad tou istou scénou. Najprv si teda v inšpektore postavy dialóg rozklikneme (obrázok 5.24).

Obr. 5.24: Rozkliknutie dialógu

Tým sa nám zobrazí inšpektor pre dialóg. Pre otvorenie grafového editora klikneme na tlačidlo Show Dialogue. Ako vyzerá editor pre prázdny dialóg môžeme vidieť na obrázku 5.25.



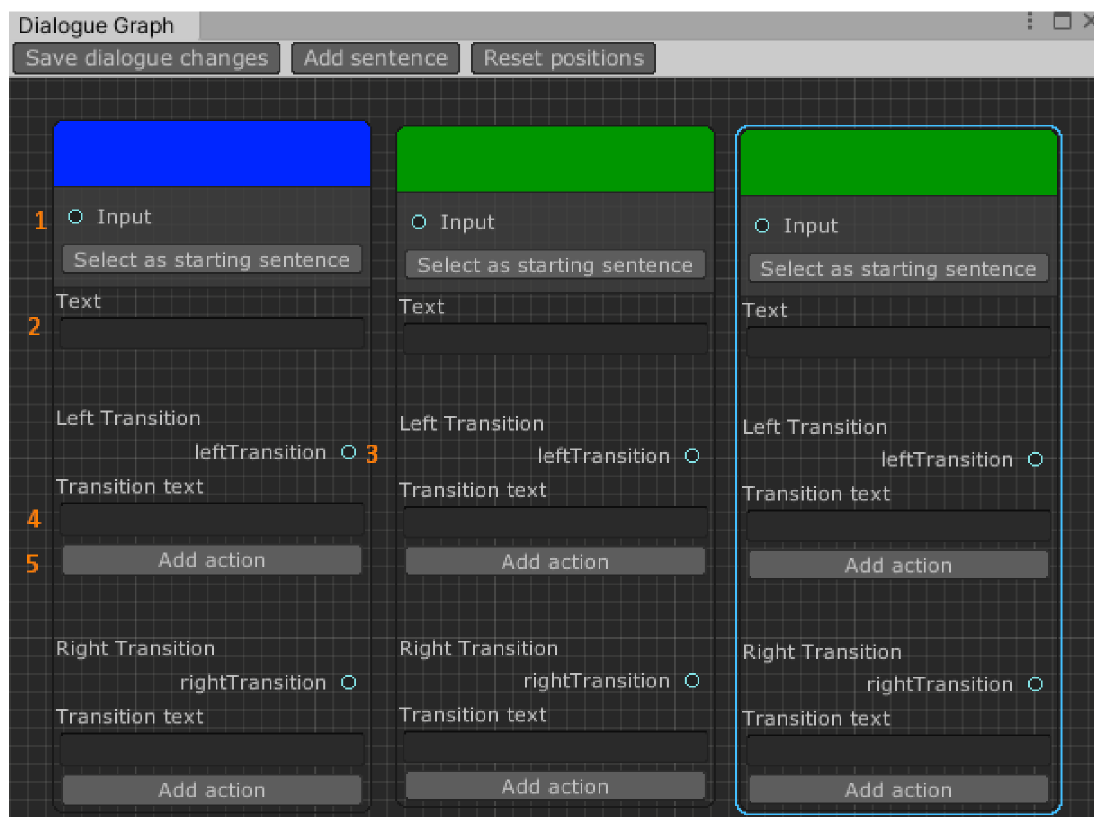
Obr. 5.25: Vzhľad editoru pre prázdny dialóg

V hornom boxe môžeme vidieť tri tlačidlá:

- *Save Dialogue Changes* — uloží zmeny vykonané na dialógu,
- *Add sentence* — pridá do grafu nový prehovor,

- *Reset Positions* — zresetuje pozície jednotlivých uzlov grafu. Môžeme ho využiť vtedy, keď sa začíname strácať v rozložení grafu. Graf sa bude snažiť nájsť čo najlepšie usporiadanie uzlov podľa ich hĺbky v grafe.

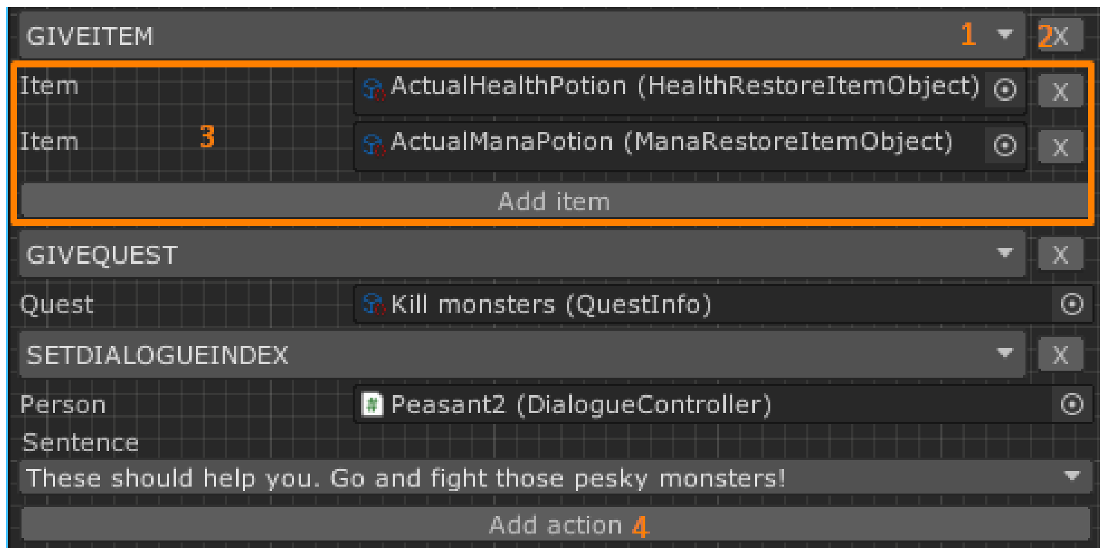
Teraz si ukážeme príklad práce s grafom. Začneme s tým, že do grafu vložíme niekoľko prehovorov. Na obrázku 5.26 môžeme vidieť tri prehovory. Prehovor označený modrou farbou je začiatočný — ide teda o prvý prehovor, ktorý postava hráčovi povie. Prehovor nastavíme na začiatočný tak, že v ňom klikneme na tlačidle Select as starting sentence.



Obr. 5.26: Editor s pár prehovormi

Prvou časťou tela uzlu je vstupný port (1). Do vstupného portu smerujú hrany z prechodov. Hrana, ktorá vedie z nejakého prechodu do vstupného portu nášho uzlu znamená, že z prehovoru , ktorého súčasťou je daný prechod sa jeho vybratím prejde do nášho prehovoru. Každý prehovor má svoj text (2). Potom uzol tvoria ešte dva prechody (Left Transition a Right Transition). Opíšeme si ľavý prechod (Left Transition), pravý (Right Transition) funguje úplne rovnako. Každý prechod má svoj výstupný port (3), z ktorého môžeme viesť hranu do ľubovoľného vstupného portu (aj do vstupného portu toho istého uzlu). Ak hrana z portu nevedie, bude po uložení dialógu tento prechod prázdny (aj keby sme vyplnili ostatné časti prechodu). Ďalej obsahuje prechod text (4) a dialógové akcie (5).

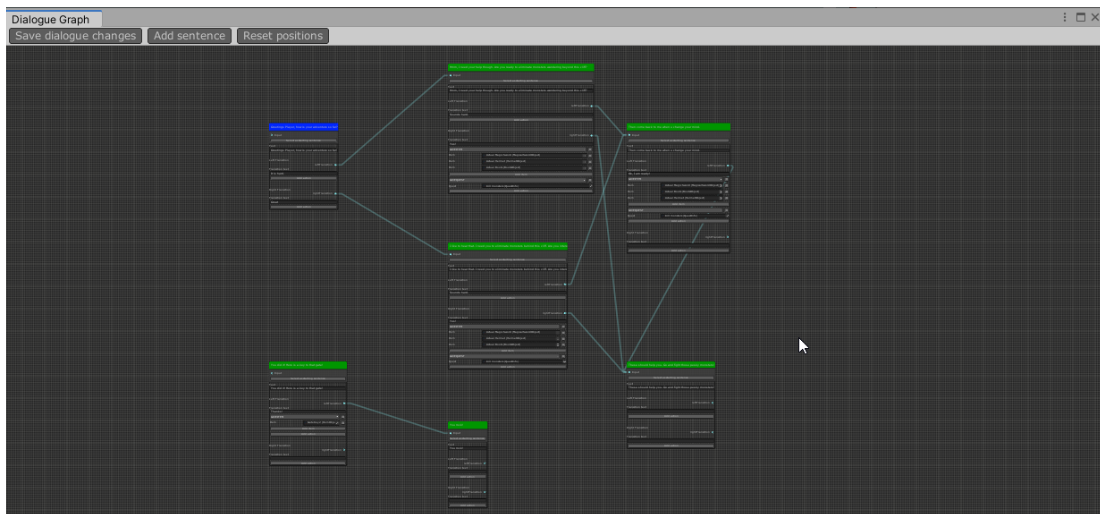
Prácu s akciami môžeme vidieť na obrázku 5.27. U každej akcie si vieme vybrať jej typ (1). Akciu vieme taktiež odstrániť pomocou tlačidla X (2). Ďalej nasleduje telo akcie, ktoré je špecifické pred daný typ akcie. Taktiež môžeme stále pridávať ďalšie akcie (4). Po uložení zmien sa dialóg uloží do priečinka Assets/Quests, dialógové akcie sa budú nachádzať v podpriečinku Actions (obrázok 5.28).



Obr. 5.27: Práca s dialógovými akciami

Obr. 5.28: Uložené dialógy

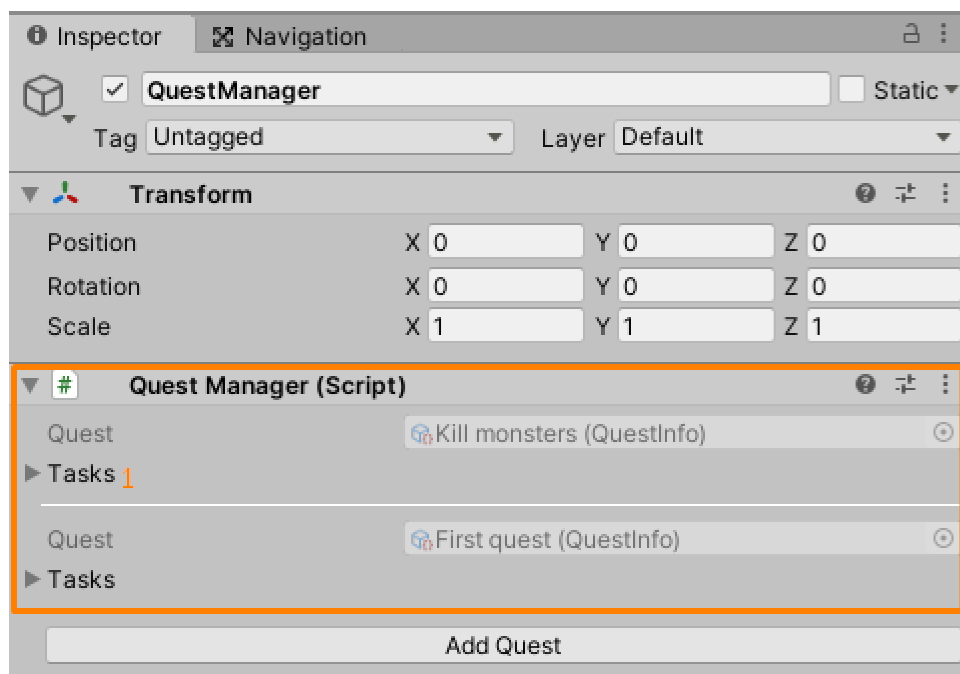
Opísali sme teda ako sa dialógy, vytvárajú, upravujú a ako sa pracuje s grafovým editorom. Príklad dialógu z hry, vytvoreného pomocou tohto editora môžeme vidieť na obrázku 5.29. Z obrázka môžeme vidieť aj to, ako funguje zresetovanie pozícii uzlov. Editor si pamätá naše rozloženie — pri ďalšom editovaní ostanú uzly v pozícii, v ktorej boli, keď sme graf uložili.



Obr. 5.29: Dialóg v hre

5.6 Questy

V tejto časti si vysvetlíme ako sa vytvárajú a upravujú questy. Prvou vecou, ktorú si musíme uvedomiť je to, že podobne ako dialógy, aj questy sú samostatné assety v projekte. Nie sú teda zviazané zo žiadnou scénou a v každej scéne znamenajú niečo iné. Je teda potrebné ich upravovať nad scénou, pre ktorú boli určené. Z toho dôvodu sa v scéne nachádza `GameObject` QuestManager. Ak si ho zvolíme v hierarchii scény uvidíme jeho inšpektor (obrázok 5.30). V ňom sa v časti Quest Manager nachádza zoznam questov, vytvorených pre túto scénu. U každého questu si môžeme rozkliknúť zoznam úloh. Tento zoznam obsahuje iba mená jednotlivých úloh.



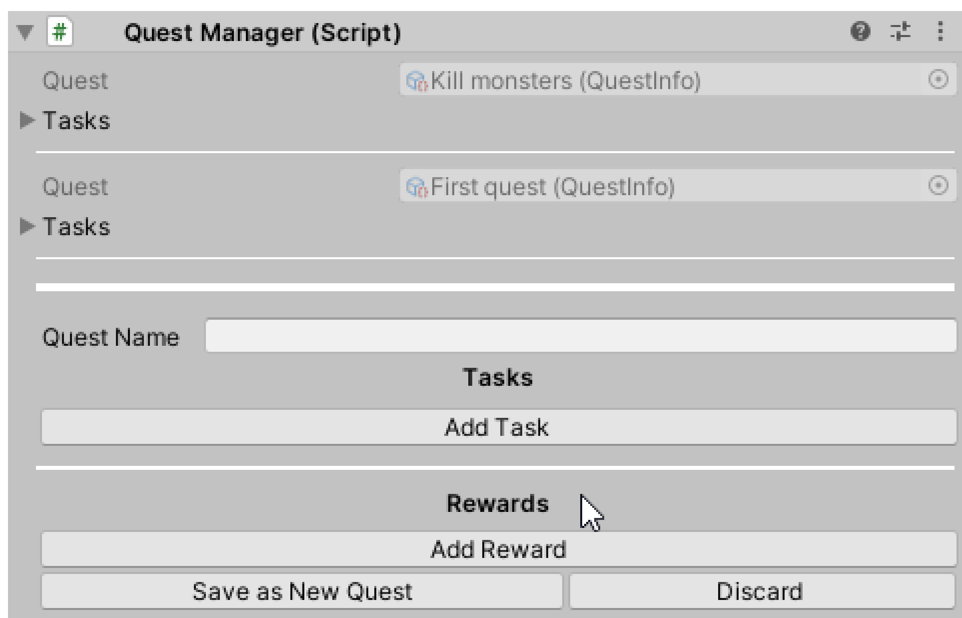
Obr. 5.30: Inšpektor Quest Managera

5.6.1 Tvorba questu

Ak chceme vytvoriť quest použijeme v inšpektore Quest Managera tlačidlo Add Quest. Tým sa, podobne ako u dialógov, dostaneme do vytváracieho režimu inšpektora (obrázok 5.31). Každý quest musí mať meno a dva questy, ktoré sú určené pre jednu scénu musia mať rozdielne mená. Meno questu nastavíme v poli Quest Name. Tu by mala naša práca s inšpektorom skončiť a quest by sme mali vytvoriť pomocou tlačidla Add as New Quest. To nám vytvorí prázdny quest, ktorý potom môžeme upraviť pomocou grafového editora.

5.6.2 Úprava questu

Opäť sa silno neodporúča vytváraný quest upravovať priamo v inšpektore pri jeho vytváraní. Postup toho, ako sa quest upravuje v inšpektore si popíšeme, ale výrazne neodporúčame — úprava questu priamo v inšpektore je neprehľadná a



Obr. 5.31: Vytvárací režim inšpektora Quest Managera

horšie sa v nej robia chyby. Taktiež sa pri úprave questu v inšpektoru nekontrolujú cykly, ktoré spôsobujú nefunkčnosť questu.

Na obrázku 5.31 môžeme vidieť časť pre pridávanie úloh (Tasks) a časť pre pridávanie odmien (Rewards). Úlohy pridávame pomocou Add Task, odmeny pomocou Add Reward. Na obrázku 5.32 môžeme vidieť prácu na úlohách a odmenách.

Každá úloha má meno, ktoré jej nastavujeme pomocou políčka Task name. U každej úlohy vyberáme jej typ, ktorý ovplyvňuje to, čo sa nachádza v tele úlohy (3). Pri úlohách využívame v hre systém prerekvizít, vďaka ktorému vieme určiť poradie, v ktorom sa musia úlohy plniť. U každej úlohe sa teda nachádza časť s prerekvizitami (2,4). Pre každú inú úlohu sa tu nachádza zaškrtávacie políčko, pomocou ktorého označime zaškrtnutú úlohu ako prerekvizitu našej úlohy. Teda v tomto prípade je uloha1 prerekvizitou uloha2 (4), čo znamená, že uloha1 musí byť splnená pred tým, ako môžeme začať plniť úlohu uloha2. Tu môžeme vidieť nepraktickosť inšpektora — ak by sme mali v queste viacero úloh, začali by časti s prerekvizitami byť veľké, práca s nimi by sa zneprehľadnila, ľahko by sme z nepozornosti mohli začať vytvárať cykly a tieto chyby by sa detekovali ťažko.

U odmien, to máme jednoduchšie. Pre každú odmenu vyberáme jej typ (5) a od toho závisí telo odmeny (6).

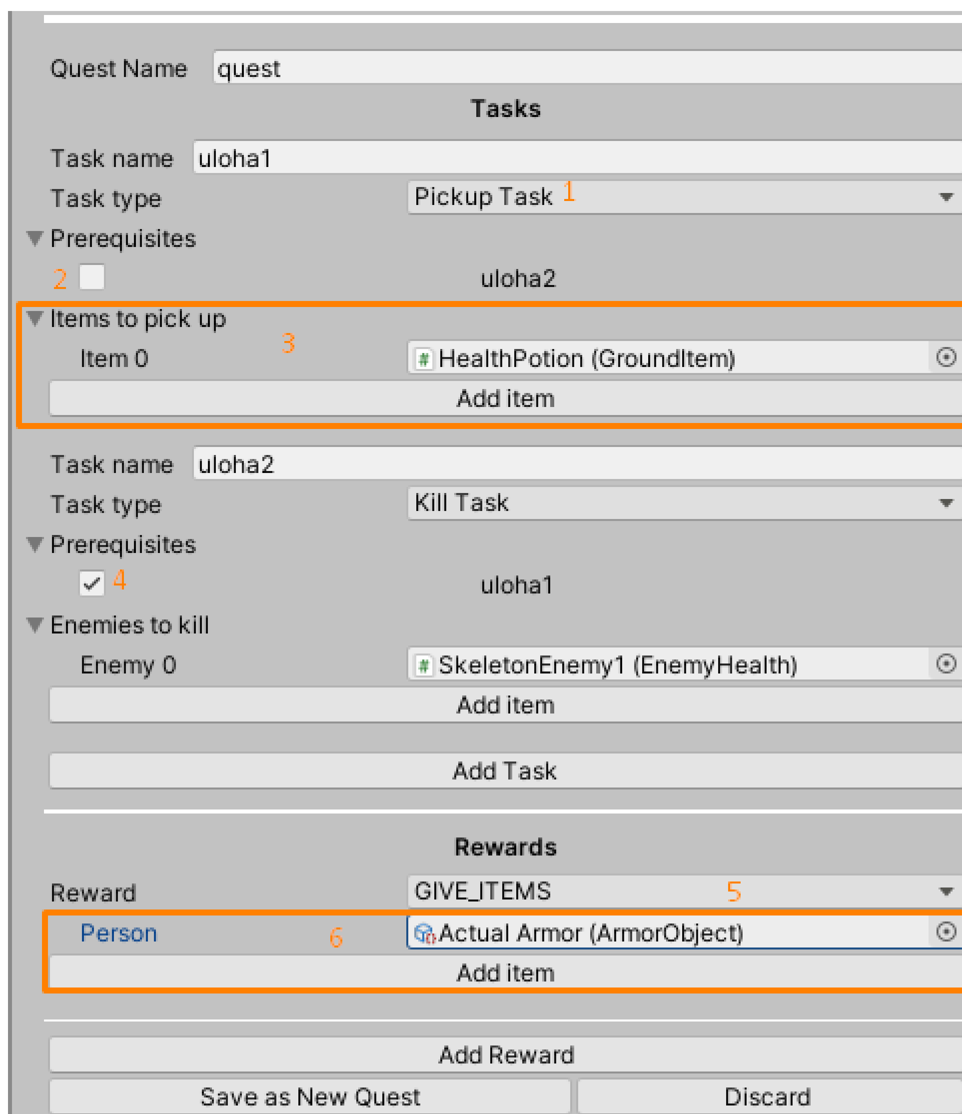
Po dokončení práce na queste môžeme quest uložiť pomocou Save as New Quest, poprípade ho zahodiť tlačidlom Discard.

Grafový editor questu

Silno sa však odporúča na úpravu questu využiť grafový editor. Opäť sa k nemu dostaneme cez inšpektor questu pomocou tlačidla Show Quest. K inšpektoru questu sa dostaneme cez Quest Manager pomocou dvojitého kliknutia na quest 5.33.

Vzhľad editora pre prázdny quest môžeme vidieť na obrázku 5.34.

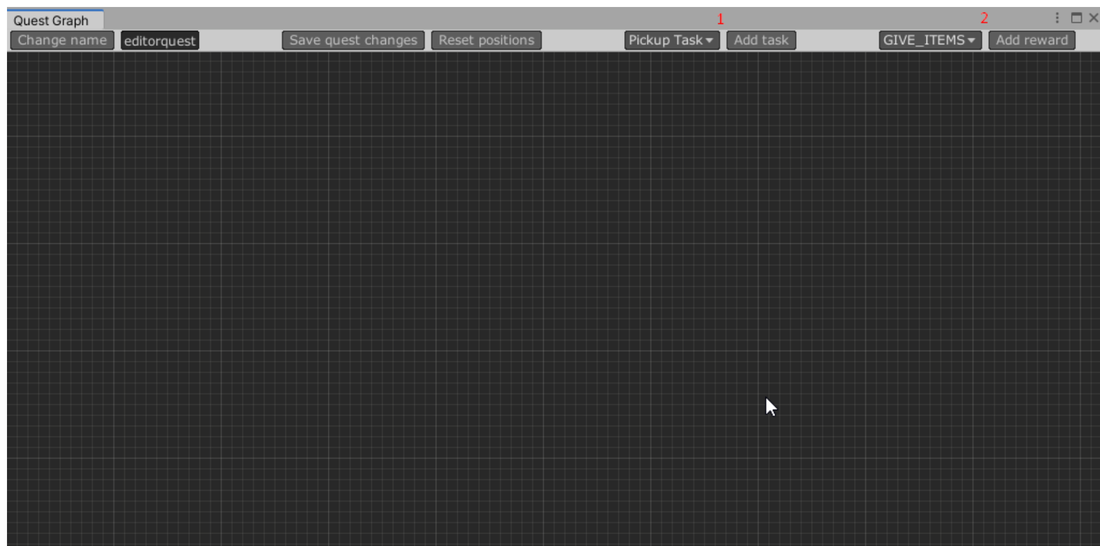
V hornom boxe môžeme vidieť niekoľko tlačidiel:



Obr. 5.32: Úprava questu cez inšpektor

Obr. 5.33: Prístup ku questu

- *Change name* — zmeníme meno questu,
- *Save quest changes* — uložíme prácu na queste,
- *Reset positions* — zresetujeme pozície jednotlivých uzlov. Keďže vieme, že náš graf je ackylický, existuje pre neho topologické usporiadanie. Toto tlačidlo zresetuje pozície uzlov podľa neho,
- *Add Task* — pridá vrchol pre typ úlohy, ktorý si zvolíme (1),
- *Add Reward* — pridá vrchol pre typ odmeny, ktorý si zvolíme (2).



Obr. 5.34: Prístup ku questu

Pre začiatok pridáme do questu niekoľko úloh a odmien. Na obrázku 5.35 môžeme vidieť editor po pridaní štyroch úloh a jednej odmeny. Každý vrchol reprezentujúci úlohu obsahuje dva porty — vstupný (1) a výstupný (2). Hrana môže viesť len medzi vstupným a výstupným portom, medzi dvoma rozdielnymi vrcholmi. Taktiež nie je možné pridať do grafu hranu, ktorá by vytvorila cyklus. Každá úloha má svoje meno, ktoré nastavujeme pomocou políčka Task name.

Hrana medzi dvoma vrcholmi reprezentuje to, že vrchol, do ktorého je hrana pripojená výstupným portom je prerekvizitou pre vrchol, do ktorého je hrana pripojená pomocou vstupného portu. Na obrázku je teda napríklad Reach Task prerekvizitou pre Pickup Task — Reach Task musí byť v queste splnený pred tým, ako začneme plniť Pickup Task.

Po portoch a mene úlohy nasleduje telo vrcholu, ktoré je špecifické pre každý typ úlohy. Každá úloha má vlastnú farbu vrcholu, aby ich išlo rýchlo rozlíšiť.

Odmeny sú vždy červenej farby a obsahujú len telo, špecifické pre každý typ odmeny.

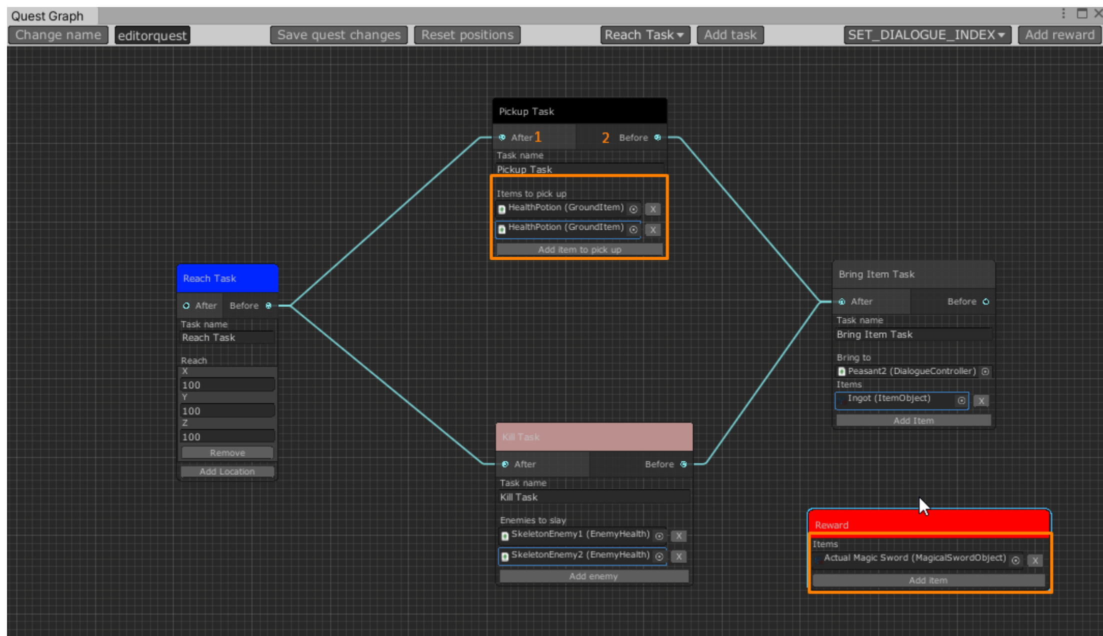
Po uložení sa quest uloží do priečinka Assets/Quests/Meno questu. Jednotlivé úlohy sa nachádzajú v podpriečinku Tasks, odmeny v podpriečinku Rewards (obrázok 5.36).

Ukázali sme si, ako sa vytvárajú a upravujú questy a ako vyzerá práca s grafovým editorom. Na obrázku 5.37 môžeme vidieť quest použitý v hre.

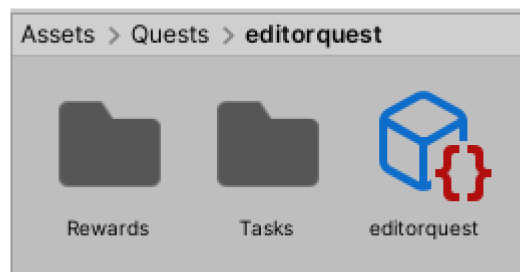
Je ešte potrebné vedieť akým spôsobom je možné quest dostať ku hráčovi. U dialógov, ktoré sme opisovali v časti Dialógy existuje dialógová akcia, ktorá hráčovi pridá quest. Presne týmto spôsobom je tento problém v hre riešený.

5.7 Ukladanie a načítavanie

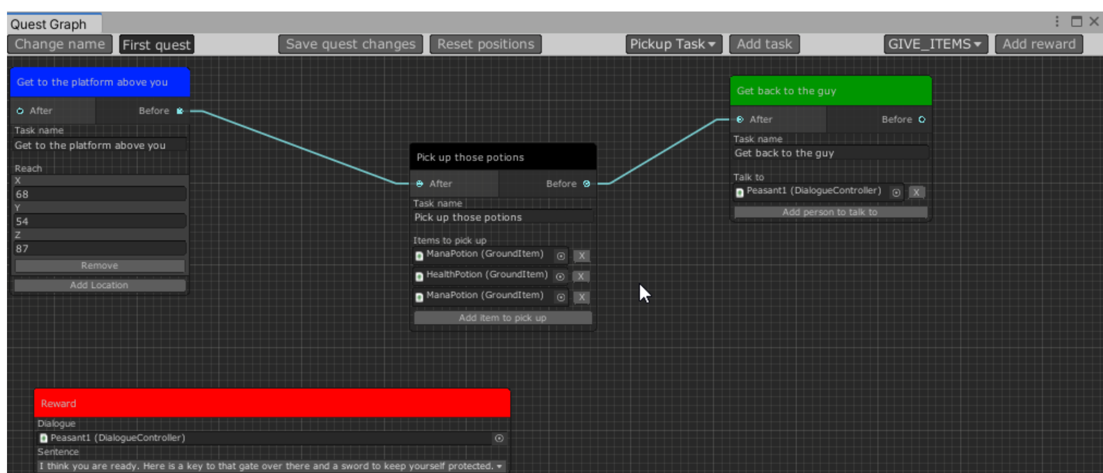
Ak chceme do scény pridať **`GameObject`**, ktorý sa má správne ukladať, musí na ňom byť komponenta spĺňajúca interface ISavableLoadable. Bez toho nie je možné tento **`GameObject`** uložiť. V prípade, že nám programátori tento komponent pridali, môžeme použiť menu. **`GameObject`** si označíme a v menu použijeme



Obr. 5.35: Editácia questu



Obr. 5.36: Uložené questy



Obr. 5.37: Quest v hre

tlačidlo Saving/Add to saving. Pomocou toho zaručíme, že sa nám informácie budú ukladať.

Záver

V tejto kapitole budeme rozoberať výsledky našej práce, či sa nám jej ciele podarilo splniť a prácu celkovo ohodnotíme. Zaoberali sme sa dvoma hlavnými cieľmi:

- samotná implementácia, príprava a vývoj hry,
- rozšírenie editora.

Tieto dva hlavné ciele si rozoberieme samostatne v nasledujúcich sekciách.

5.8 Implementácia hry

Implementácia príbehovej dobrodružnej hry bola prvým cieľom našej práce. Chceli sme teda pripraviť spustiteľnú hru, ktorú by si mohol hráč zahrať a používať v nej herné prvky a mechaniky, o ktorýh sme rozprávali hlavne v kapitole Špecifikácia zadania. Odovzdali sme hru, v ktorej sa nachádzajú dva plne hrateľné levely a taktiež tutoriál, ktorý slúži na oboznámenie hráča s najdôležitejšími časťami hry. Pri implementácii jednotlivých herných prvkov (predmety, questy, dialógy, nepriatelia, ukladanie a načítavanie atď.) sa autor snažil programovať s ohľadom na možné budúce rozšírenia hry. Z tohto ohľadu sme spokojní s tým, ako sme konkrétne ciele vyriešili a implementáciu samotnej hry považujeme za splnenú. Musíme však uznať aj to, že v práci, v ktorej by išlo čisto o danú hru by sme pravdepodobne chceli pripraviť tak tri levely, avšak s ohľadom na to, že je v hre prítomný aj tutoriál a taktiež je súčasťou práce rozšírenie editora, ktoré uľahčuje a urýchľuje prácu na možných budúcich leveloch považujeme odovzdávanú hru za dostatočnú.

5.9 Rozšírenie editora

Druhou časťou práce bolo rozšírenie editora, v ktorom sme hru vyvíjali takým spôsobom, aby bola príprava a vytváranie ďalších levelov čo najjednoduchšia a najprístupnejšia. Snažili sme sa editor pripraviť tak, aby sa práca pre dizajnérov a programátorov dala od seba čo najviac rozlíšiť, kde sme chceli dosiahnuť hlavne to, aby dizajnér nemusel o kóde hry nič vedieť.

K tomu sme využívali možnosti, ktoré nám ponúklo Unity, v ktorom bola celá hra vyvíjaná – pridávanie tlačidiel do hlavného menu editora, pomocou ktorých môže dizajnér pridávať do levelu nepriateľov, predmety, platformy alebo napríklad pridávať možnosť dialógu s postavou. Našim cieľom bolo to, aby nemusel dizajnér riešiť nič čo sa týku toho ako interne veci fungujú. Pridanie nového levelu by malo byť tiež jednoduché, dizajnér by nemal riešiť to, kde sú jednotlive scény uložené, kde je uložený terén, v scéne by už malo byť nastavené všetko potrebné takým spôsobom, aby sa mohol dizajnér venovať len dizajnu. V špeciálnych prípadoch sme do editoru pridali aj grafový editor, kde má dizajnér možnosť na úpravu dialógov a questov, pomocou ktorých sa stavia príbeh hry použiť jednoduché a príjemné prostredie.

Opäť, ciele považujeme za splnené, spokojní sme najmä s grafovými editormi, ktoré sme pre dizajnéra predstavili

5.10 Ohodnotenie práce

Prácu hodnotíme pozitívne, keďže sa nám podarilo splniť ciele, ktoré si autor stanovil v kapitolách Úvod a Špecifikácia zadania. Počas práce na hre sme sa stretli s viacerými problémami, ktoré sa nám napokon podarilo rozumne vyriešiť. Spokojní sme najmä s grafovými editormi, ktoré sme pre potenciálnych dizajnérov pripravili a s tým, že sme hru implementovali na solídnych základoch, kde sme dbali na rozšíriteľnosť a budúcu prácu na hre.

Budúce rozširovanie práce

V práci sa ponúka veľa spôsobov, pomocou ktorých by ju bolo možné v budúcnosti rozširovať. Čo sa týka samotnej hry, mohlo by ísť o:

- pridanie zvuku do hry,
- nové levely,
- pridanie nových predmetov, nepriateľov, kúziel,
- rozšírenie questového systému – globálne questy (cez viacero levelov, achievementy), nové typy úloh, odmien, pridanie možnosti nepovinných úloh,
- rozšírenie dialógového systému – nové typy akcií, podmienené odpovede
- systém levelovania, pridanie výberu náročnosti do hry.

Čo sa týka rozširovania editoru, mohlo by ísť napríklad o pridanie grafového editoru aj pre vytváranie nepriateľov.

Seznam použité literatury

[1] Brandon McIntyre. In 2019, shooters were more boring than ever. https://www.thegamer.com/2019-shooters-more-boring-than-ever/, 2019. Platnosť odkazu: 16.07.2021.

[2] Techopedia. Role-playing game (rpg). https://www.techopedia.com/definition/27052/role-playing-game-rpg. Platnosť odkazu: 16.07.2021.

[3] Gökhan Çakır Bhernardo Viana. What is an autobattler? https://dotesports.com/news/what-is-an-autobattler, 2020. Platnosť odkazu: 16.07.2021.

[4] Janardana. 2.5d. https://www.giantbomb.com/25d/3015-660/, 2019. Platnosť odkazu: 16.07.2021.

[5] Michael Crider. Swordigo review: A solid action-platformer for fans of retro rpgs. https://www.androidpolice.com/2014/02/28/swordigo-review-a-solid-action-platformer-for-fans-of-retro-rpgs/, 2014. Platnosť odkazu: 16.07.2021.

[6] Pure Adrenallen. Inventory. https://minecraft.fandom.com/wiki/Inventory, 2010. Platnosť odkazu: 16.07.2021.

[7] Liberal Ideas. Quests (skyrim). https://elderscrolls.fandom.com/wiki/Quests_(Skyrim), 2011. Platnosť odkazu: 16.07.2021.

[8] Nannings Games. 5 best game engines for beginner indie developers). https://hackernoon.com/5-best-game-engines-for-beginner-indie-developers-y4893ush, 2020. Platnosť odkazu: 16.07.2021.

[9] Creative Blog. Unity vs unreal engine: which game engine is for you?). https://www.creativebloq.com/advice/unity-vs-unreal-engine-which-game-engine-is-for-you, 2019. Platnosť odkazu: 16.07.2021.

[10] Unity. Platform game. https://en.wikipedia.org/wiki/Platform_game, 2019. Platnosť odkazu: 16.07.2021.

[11] Unity. 2018.3 terrain update: Getting started. https://en.wikipedia.org/wiki/Platform_game, 2018. Platnosť odkazu: 16.07.2021.

[12] Unity. Rigidbody. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/Rigidbody.html, 2019. Platnosť odkazu: 16.07.2021.

[13] IronEqual. Unity: Character controller vs rigidbody. https://medium.com/ironequal/unity-character-controller-vs-rigidbody-a1e243591483, 2017. Platnosť odkazu: 16.07.2021.

[14] Unity. Serializereference. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/SerializeReference.html, 2019. Platnosť odkazu: 16.07.2021.

[15] Unity. Scriptableobject. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/ScriptableObject.html, 2019. Platnosť odkazu: 16.07.2021.

[16] Unity. Prefabs. https://docs.unity3d.com/2019.4/Documentation/Manual/Prefabs.html, 2019. Platnosť odkazu: 16.07.2021.

[17] Wikipedia. Boss (video games. https://en.wikipedia.org/wiki/Boss_(video_games), 2014. Platnosť odkazu: 16.07.2021.

[18] Unity. Using animation events. https://docs.unity3d.com/2019.4/Documentation/Manual/script-AnimationWindowEvent.html, 2019. Platnosť odkazu: 16.07.2021.

[19] Microsoft. Handling and raising events. https://docs.microsoft.com/en-us/dotnet/standard/events/, 2017. Platnosť odkazu: 16.07.2021.

[20] Unity. Editorwindow. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/EditorWindow.html, 2019. Platnosť odkazu: 16.07.2021.

[21] Unity. Graphvieweditorwindow. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/Experimental.GraphView.GraphViewEditorWindow.html, 2019. Platnosť odkazu: 16.07.2021.

[22] Mert Kirimgeri. Node graph tutorials. https://www.youtube.com/watch?v=7KHGH0fPL84&list=PLF3U0rzFKlTGzz-AUFacf9_OKiW_hGYIR&index=3, 2019. Platnosť odkazu: 16.07.2021.

[23] Unity. Playerprefs. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/PlayerPrefs.html, 2019. Platnosť odkazu: 16.07.2021.

[24] Microsoft. Binaryformatter class. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=net-5.0, 2019. Platnosť odkazu: 16.07.2021.

[25] Unity. Custom editors. https://docs.unity3d.com/2019.4/Documentation/Manual/editor-CustomEditors.html, 2019. Platnosť odkazu: 16.07.2021.

[26] Unity. The inspector window. https://docs.unity3d.com/Manual/UsingTheInspector.html, 2019. Platnosť odkazu: 16.07.2021.

[27] Unity. Scenes. https://docs.unity3d.com/2019.4/Documentation/Manual/CreatingScenes.html, 2019. Platnosť odkazu: 16.07.2021.

[28] Unity. Gameobjects. https://docs.unity3d.com/2019.4/Documentation/Manual/GameObjects.html, 2019. Platnosť odkazu: 16.07.2021.

[29] Unity. Transform. https:https://docs.unity3d.com/2019.4/Documentation/Manual/class-Transform.html, 2019. Platnosť odkazu: 16.07.2021.

[30] Unity. Colliders. https://docs.unity3d.com/2019.4/Documentation/Manual/CollidersOverview.html, 2019. Platnosť odkazu: 16.07.2021.

[31] Unity. Scripting. https://docs.unity3d.com/2019.4/Documentation/Manual/ScriptingSection.html, 2019. Platnosť odkazu: 16.07.2021.

[32] Unity. Monobehaviour. https://docs.unity3d.com/2019.4/Documentation/ScriptReference/MonoBehaviour.html, 2019. Platnosť odkazu: 16.07.2021.

Zoznam obrázkov

1.1 Swordigo[5] . . . 5

2.1 Inventár v Minecrafte[6] . . . 7
2.2 Questy v Skyrime[7] . . . 9

3.1 Donkey Kong[10] . . . 14
3.2 Terén v Unity[11] . . . 14
3.3 Rigidbody komponent v inšpektore . . . 16
3.4 Stavový automat pre skeletona v hre . . . 21
3.5 HealthSystem využívajúci eventy . . . 23
3.6 Nesplniteľný quest . . . 24
3.7 Vrcholy len pre celé prehovory . . . 28
3.8 Vrcholy pre prehovory a akcie . . . 29
3.9 Interface . . . 33

4.1 MainMenu scéna . . . 34
4.2 BaseScene scéna . . . 36
4.3 BaseScene Canvas . . . 36
4.4 ItemObject . . . 44
4.5 Hierarchia Canvasu . . . 46
4.6 Rozšírené menu . . . 47
4.7 Pomocné vykresľovanie . . . 48

5.1 Hlavné menu . . . 50
5.2 Načítavací mód . . . 51
5.3 Vymazávací mód . . . 51
5.4 Options menu . . . 51
5.5 Pause Menu . . . 52
5.6 Menu na ukladanie hry . . . 52
5.7 Uložené hry z Pause Menu . . . 53
5.8 Pridanie nového levelu . . . 54
5.9 Pridanie platformy . . . 54
5.10 Inšpektor Platforms . . . 54
5.11 Pohybujúca sa platforma v editore . . . 55
5.12 Vyrovnanie pomocou Bridge . . . 55
5.13 Pridanie blokátorov do levela . . . 55
5.14 Scéna po pridaní terénu z menu . . . 56
5.15 Level Load . . . 57
5.16 Animation okno . . . 57
5.17 Príklad výsledného animátora pre začiatok levelu . . . 57
5.18 Predmety v leveli . . . 58
5.19 Priečinok s predmetmi . . . 58
5.20 Zvolený skeleton v scéne . . . 59
5.21 Pripravený GameObject . . . 60
5.22 Inšpektor postavy . . . 60
5.23 Režim inšpektora pre vytváranie dialógu . . . 61

5.24 Rozkliknutie dialógu 62
5.25 Vzhľad editoru pre prázdny dialóg 62
5.26 Editor s pár prehovormi 63
5.27 Práca s dialógovými akciami 64
5.28 Uložené dialógy 64
5.29 Dialóg v hre 64
5.30 Inšpektor Quest Managera 65
5.31 Vytvárací režim inšpektora Quest Managera 66
5.32 Úprava questu cez inšpektor 67
5.33 Prístup ku questu 67
5.34 Prístup ku questu 68
5.35 Editácia questu 69
5.36 Uložené questy 69
5.37 Quest v hre 69

A.1 Inšpektor[26] 78
A.2 Dodatočná práca 79
A.3 Úprava skeletona 80

A. Prílohy

A.1 Unity

A.1.1 Scéna

Jedným zo základných prvkov Unity sú scény[27]. Jednotlivé scény obsahujú časti našej aplikácie, hry. Sú to teda v podstate logické kontajnery, v ktorých staviame našu hru. Pri práci v editore pracujeme skoro vždy v kontexte nejakej scény. Pri hraní hry máme načítané nejaké scény a v hre sú potom načítané len objekty z týchto scén. Jednotlivé objekty v scéne si na seba môžu držať referencie.

Na rôznych fórach, diskusných miestnostiach sa môže človek dočítať, že jedna scéna = jeden level. Nie je to však pravda. V spustenej hre môžeme mať aktívnych a načítaných viacero scén — v hre sú potom načítané všetky objekty, ktoré do týchto scén patria. Jeden level teda môžeme rozdeliť do viacerých scén, ale napríklad aj viacero levelov môžeme postaviť v jednej scéne.

A.1.2 GameObject

V predchádzajúcej časti sme viackrát spomenuli objekty v scénach. Ak hovoríme o objekte v scéne, máme tým na mysli **`GameObject`**[28]. Všetko to, čo sa nachádza v scéne je **`GameObject`** — či už ide o kameru, hráčovu postavu, plátno, na ktoré sa vykresľuje užívateľské rozhranie či malý obrázok na takomto plátne.

`GameObject` je tvorený z komponentov. Komponent, ktorý je na každom GameObjecte a nejde ho odstrániť je komponent **`Transform`**[29] — obsahuje informácie o pozícii, rotácii a škále **`GameObjectu`**. Komponenty sú to, čo ovplyvňuje správanie a výzor **`GameObjectu`**. Niektoré nám Unity ponúka — komponent, ktorý vykresľuje, komponent, ktorý obsahuje text, obrázky, ale napríklad aj komponent, ktorý nastavuje tak, ako **`GameObject`** v kontexte fyziky (**`Collider`**[30]) a pre potrebu kolízií.

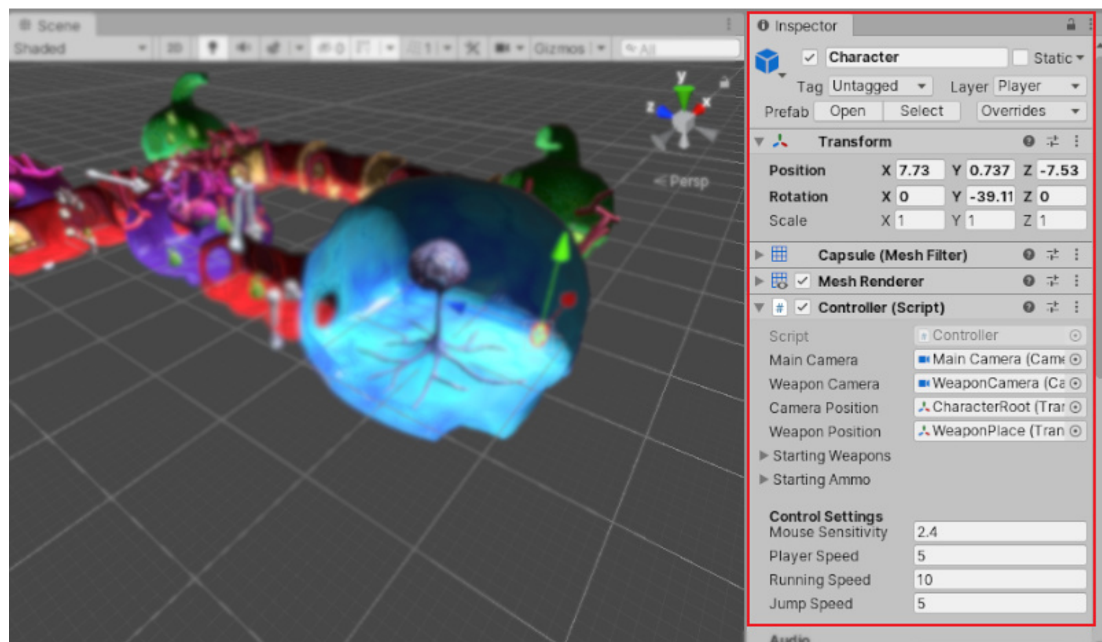
Na to, aby sme teda mohli v Unity pracovať, potrebujeme si vedieť takéto komponenty vytvárať. To sa robí pomocou *skriptov*. Skripty sa píšu pomocou C#. Trieda v skripte musí dediť od **`MonoBehaviour`**. Takýto skript potom môžeme pridávať ako komponent. Pridanie skriptu na **`GameObject`** vytvorí novú inštanciu tejto triedy a prepojí ju s **`GameObjectom`**, na ktorý bol skript pridaný. Komponenty môžeme povoľovať a vypínať.

V skripte potom môžeme implementovať špeciálne metódy ako **`Awake()`** (zavolá sa pri načítaní inštancie skriptu) , **`Start()`** (zavolá sa počas prvej snímky, keď je skript povolený), **`Update()`** (je volaný pri každej snímke, ak je skript povolený), **`FixedUpdate()`** (volá sa vždy vo fixnej frekvencii, súvisí s fyzikálnym enginom) a mnohé ďalšie. O podrobnostiach skriptovania je možné si prečítať v dokumentácii[31, 32].

A.1.3 Inšpektor

Keď v editore Unity zvolíme nejaký **`GameObject`** alebo asset, zobrazí sa nám pre neho *inšpektor*. Pomocou inšpektora môžeme pracovať s komponentami bez

toho, aby sme používali kód — môžeme komponenty pridávať, odoberať, povoľovať alebo napríklad meniť hodnoty premenných v komponente. Inšpektor môžeme vidieť na obrázku A.1



Obr. A.1: Inšpektor[26]

A.2 Assety

Pri tvorbe hry sme používali assety z Asset Store od Unity a z Mixama. Niektoré z nich treba z licenčných dôvodov posťahovať.

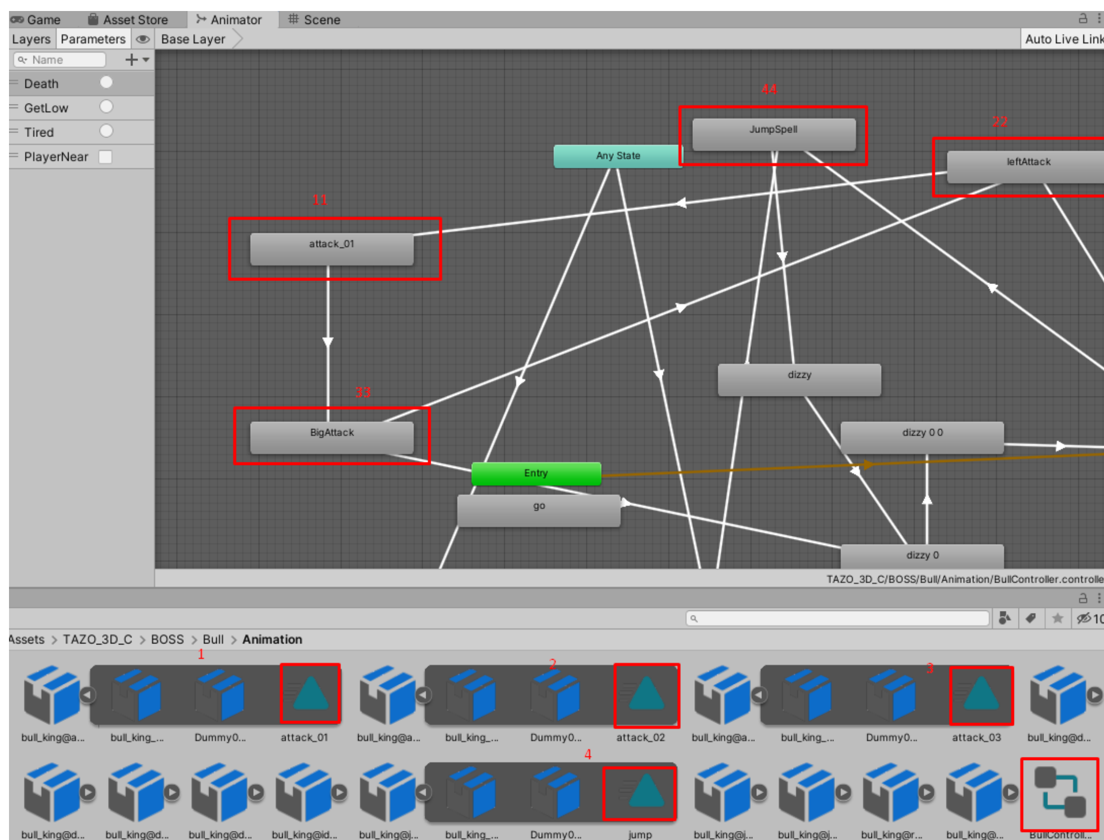
Asset Store

Tieto assety je potrebné z Asset Storu stiahnuť a naimportovať do projektu:

- `https://assetstore.unity.com/packages/3d/environments/fantasy/horse-statue-52025`
- `https://assetstore.unity.com/packages/2d/gui/icons/rpg-inventory-icons-56687`
- `https://assetstore.unity.com/packages/3d/props/handpainted-turntable-platforms-66599`
- `https://assetstore.unity.com/packages/3d/environments/fantasy/polarized-platform-pack-37774`
- `https://assetstore.unity.com/packages/3d/characters/humanoids/boss-class-bull-115521`

  Na to, aby sme sa vrátili do bodu, pri ktorom sme vyvíjali hru ešte musíme vykonať pár krokov. Najprv, **`GameObjectu`** Bull nastavíme v inšpektore v

časti Animator za Controller BullController. Zvyšok si opíšeme pomocou obrázka A.2



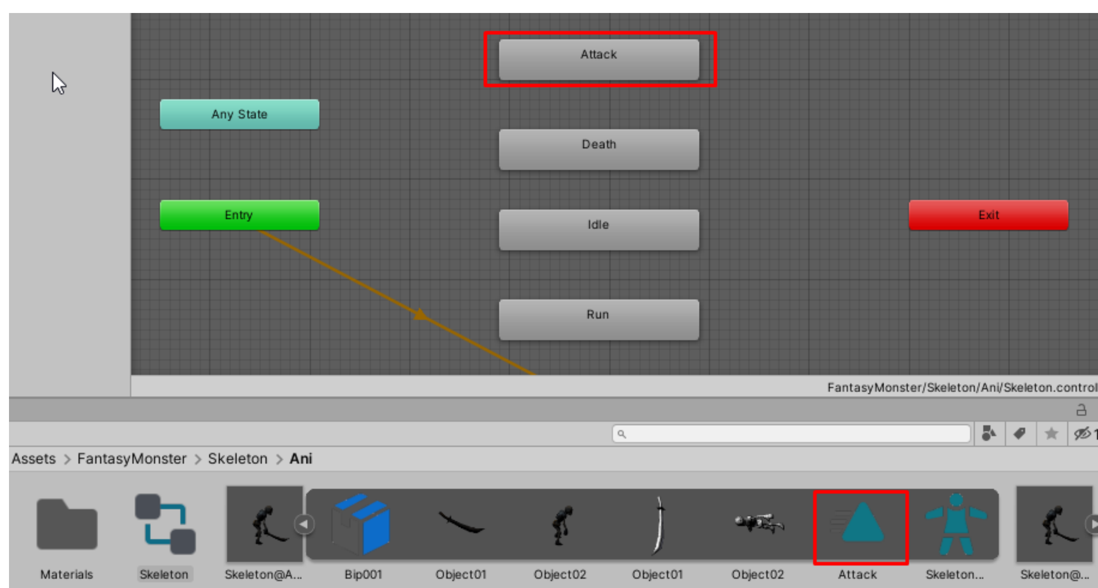
Obr. A.2: Dodatočná práca

Najprv si otvoríme priečinok Animation v importovaných súboroch. 4 konkrétne FBX súbory si rozklikneme, rovnako ako na obrázku. Pre každý jeden z nich potom duplikujeme animáciu (trojuholník) pomocou označenia a stlačenia Ctrl+D. Tieto animácie potom nastavíme ako animácie pre 4 označené stavy v animátorovi (1 pre 11, 2 pre 22, 3 pre 33, 4 pre 44) . To sme potrebovali urobiť z toho dôvodu, že u nich využívame Animation Eventy. Animation Eventy môžeme na animácie pridať cez okno Animation. Otvoríme si teda postupne všetky 4 animácie v okne Animation a v hierarchii scény si označíme `GameObject` Bull. Pre attack_01 a attack_02 pridáme animačný event v osemnástom snímku s metódou Attack, s parametrom, pre attack_03 v dvadsiatom a pre Jump v dvadsiatom druhom. Pre attack_01 a attack_02 zavoláme metódu Attack s parametrom 20, pri attack_03 s parametrom 40 a pre jump budeme volať SpellAtack().

- `https://assetstore.unity.com/packages/2d/textures-materials/water/stylize-snow-texture-153579`
- `https://assetstore.unity.com/packages/3d/props/weapons/low-poly-fantasy-sword-65120`
- `https://assetstore.unity.com/packages/3d/environments/urban/survival-old-house-55315#content`

- https://assetstore.unity.com/packages/3d/environments/fantasy/glowing-forest-79686
- https://assetstore.unity.com/packages/3d/environments/hand-painted-nature-kit-lite-69220
- https://assetstore.unity.com/packages/3d/vegetation/trees/snowy-low-poly-trees-76796
- https://assetstore.unity.com/packages/3d/props/weapons/magic-swords-97694
- https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy-monster-skeleton-35635

  Podobne ako u bull bossa, aj u skeletona si potrebujeme pomôcť s Animation Eventami, aj keď v tomto prípade to bude o dosť jednoduchšie, obrázok A.3. Zduplikujeme si animáciu Attack, vložíme ju do stavu Attack a pridáme Animation Event presne na šesťdesiatu snímku. Volať sa bude metóda Attack bez parametrov.



Obr. A.3: Úprava skeletona

- https://assetstore.unity.com/packages/3d/vegetation/trees/low-poly-tree-pack-57866
- https://assetstore.unity.com/packages/2d/textures-materials/sky/fantasy-skybox-free-18353
- https://assetstore.unity.com/packages/2d/textures-materials/floors/hand-painted-grass-texture-7855
- https://assetstore.unity.com/packages/2d/textures-materials/nature/terrain-tools-sample-asset-pack-145808

Taktiež sme používali postavy a animáce z Mixama. Pre postavy v leveli, s ktorými hráč vedie dialóg používame postavy Abe, Peasant Man a Peasant Girl. U nich nám stačí si ich postavy stiahnuť s nejakou základnou idle animáciou.

Čo sa týka bossa druhého levelu, používame postavu MAW J LAYGO a 6 animácií vo formáte fbx — magic spell casting, mutant punch dvakrát (zrkadlovo), falling back death, idle a mutant flexing muscles. V priečinku Asset/MixamoBoss sa nachádza Animator Controller. Potrebujeme obidva mutant punche vybrať z fbx, rovnako aj magic spell casting, aby sme na nich mohli pridať Animation Events. U úderov sme použili ôsmu snímku s metódou Attack, u kúzla sme použili udalosti až dve. Je nakoniec potrebné doplniť Animator Controller.