ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Boros-Jékey István Mihály**
2022	Hallgató törzskönyvi száma:	**T/004021/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve:	**Boros-Jékey István Mihály**
Törzskönyvi száma:	T/004021/FI12904/N
Neptun kódja:	[redacted]

A dolgozat címe:

Szerepjáték-fejlesztés korszerű technológiák használatával
Roleplaying game development using modern technologies

Intézményi konzulens:	Simon-Nagy Gabriella
Külső konzulens:	
Beadási határidő:	2022. május 15.
A záróvizsga tárgyai:	Számítógép architektúrák Szoftvorrendszerek fejlesztése specializáció

A feladat

Tervezzen és valósítson meg egy szerepjátékot, amely demonstrálja a retro stílusú játékok modern megvalósításának lehetőségeit. Elemezze a klasszikus szerepjátékok által nyújtott szolgáltatásokat és ezek korlátait, különös tekintettel a játékmenet felépítésére, a játékos és egyéb karakterek közötti párbeszéd, kereskedelem, együttműködés és harc módjaira. Vizsgálja meg a modern játékfejlesztő eszközöket, válassza ki és ismerje meg a feladat megoldásához célszerű technológiákat. Készítsen egy olyan kétdimenziós, felülnézetes szerepjátékot, amely bemutatja a fentebb említett funkciókat.

A dolgozatnak tartalmaznia kell:

- a feladat pontos ismertetését,
- a klasszikus szerepjátékok tipikus játékmenetét és funkcióit,
- a feladatban meghatározott célú hasonló szoftverek bemutatását,
- a feladat megoldásához alkalmazható technológiák tömör bemutatását,
- a választott megoldások indoklását,
- a tervezés és megvalósítás részletes leírását,
- a tesztelés tervét és eredményeit,
- a továbbfejlesztési lehetőségek számba vételét.

Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elővülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022.05.09.

.....
hallgató aláírása

ÓBUDAI EGYETEM
Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: **BOROS-JÉKEY ISTVÁN MIHÁLY** Neptun Kód: [redacted] Tagozat: **NAPPALI**

Telefon: [redacted] Levelezési cím: [redacted]

Szakdolgozat címe magyarul:
SZEREPJÁTÉK-FEJLESZTÉS KORSZERŰ TECHNOLÓGIÁK HASZNÁLATÁVAL

Szakdolgozat címe angolul:
ROLEPLAYING GAME DEVELOPMENT USING MODERN TECHNOLOGIES

Intézményi konzulens: **SIMON-NAGY GABRIELLA** Külső konzulens: -

Alk.	Dátum	Tartalom	Aláírás
1.	2018. 09. 28.	A téma meghatározása	
2.	2018. 10. 16.	A feladatok megbeszélése	
3.	2018. 12. 04.	Az irodalomkutatás összegzése	
4.	2018. 12. 17.	Rendszerspecifikáció elkészítése	

A hallgató a „Szakdolgozat I." tantárgy követelményét teljesitette, beszámolóra bocsátható.

.....
Intézményi konzulens

Budapest, 2018. december 17.

ÓBUDAI EGYETEM
Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: **BOROS-JÉKEY ISTVÁN MIHÁLY** Neptun Kód: [redacted] Tagozat: **NAPPALI**

Telefon: [redacted] Levelezési cím: [redacted]

Szakdolgozat címe magyarul:
SZEREPJÁTÉK-FEJLESZTÉS KORSZERŰ TECHNOLÓGIÁK HASZNÁLATÁVAL

Szakdolgozat címe angolul:
ROLEPLAYING GAME DEVELOPMENT USING MODERN TECHNOLOGIES

Intézményi konzulens: **SIMON-NAGY GABRIELLA**
Külső konzulens: -

Alk.	Dátum	Tartalom	Aláírás
1.	2019. 02. 20.	Megvalósítási lehetőségek áttekintése	
2.	2019. 03. 22.	Fejlesztési fázis ellenőrzése	
3.	2019. 04. 24.	Tesztelés ellenőrzése	
4.	2019. 05. 15.	Eredmények, összegzés	

A hallgató a „Szakdolgozat II." tantárgy követelményét teljesítette, védésre bocsátható.

.....
Intézményi konzulens

Budapest, 2019. május 15.

TARTALOMJEGYZÉK

1. BEVEZETÉS

1.1. Előzmények

Már egészen korán, általános iskola első osztályának idején elkezdtem érdeklődni a videójátékok titokzatos világa iránt. Egy-két évvel később megismertem egy nagyon jó barátomat, aki szüleim akkortájt elképzelhetően hatalmas örömére még tovább kalauzolt a PC-s játékszoftverek terén. Valójában ekkor kezdtem el igazán belekóstolni ebbe az szubkultúrába.

A főleg számítógépes, illetve később a konzolos és okostelefonos videójátékok végig kísérték és meghatározó részét képezték az életemnek mind a mai napig. Remélem, hogy ez a megállapítás a jövőben is helytálló lesz. Persze a legtöbbünknek egyre kevesebb ideje jut akárcsak egyetlen hobbi űzésére is.

Kétlem azonban, hogy tudna bárki is akár egyetlen olyan komolyabb műfajjal előhozakodni, amelyet már valamilyen módon ki ne próbáltam volna ebben a méretes szórakoztatóipari ágazatban. Vitathatatlan viszont, hogy a legtöbb időt szerepjátékok kiélvezésével töltöttem eddig el az ilyen jellegű tevékenységek keretében. A tapasztalatom szerint ugyanis ezekben a művekben lelhető fel a legtöbb történet, párbeszéd, játékrendszer, összefüggés, egy szóval a legtöbb munka.

Szerény véleményem szerint az nyugodt lelkiismerettel kijelenthetem, hogy egyvalami engem valóban megkülönböztet az átlagos videójáték élvezőtől. Ez mégpedig az, hogy már tizenéves korom elejétől kezdve aktívan igyekszem tudomást szerezni ezen alkotások készítésének háttere felől is. Gyakran olvasok cikkeket és nézek meg videóanyagokat az említett témához fűződően. Elsősorban a játékdizájn érdekel amellett, hogy időközben kialakult bennem egy erős érdeklődés maguk a számítógépek működése iránt is, mind szoftveres, mind pedig hardveres szinten.

1.2. Célok

A jelen szakdolgozat legfőbb célja egy „világilag” kétdimenziós felépítésű, felülről nézetes, „retró stílusú” szerep-videójáték elkészítése, illetve a folyamat megfelelő módon történő dokumentálása legalább egy „Proof of Concept [PoC]” („elképzelés bizonyítása”), „prototípus”, vagy „alfa” szint elérésének, azaz egy fajta előrehaladottabb kezdeti stádiumba besorolható játékszoftver létrejöttének pontjáig.

Ez az elkészültségi szint természetesen többféleképpen is értelmezhető, ezért a később tárgyalandó, harmadik fejezetben helyet foglaló specifikációban igyekeztem ezt a fogalmat majd reálisan meghatározni. Mindezt pedig a kiválasztott korszerű technológiák felhasználásával szándékozom véghez vinni. A részletek a további fejezetek során fognak majd természetesen kikristályosodni.

Egy ilyen típusú tevékenység (játékfejlesztés) meglehetősen sok összetevőből, illetve valójában számos különálló szakterület versenyképes felhasználásából áll. Ezen építőkockák közül a meghatározóbbak feltérképezése mentén szeretném majd azt a következőkben körvonalazni. Bízom egyúttal abban is, hogy a már említett „retró" kifejezés is megfelelő értelmet fog nyerni az olvasó számára a tárgyalt munkára nézve.

1.3. Kihívások

A dolgozat legnehezebb részének azt tartom, hogy alapvetően merőben különböző szakmai területeken szükséges majd egymagam személyében a megfelelő szintű munkafolyamatok gyakorlati alkalmazását elsajátítanom.

Egyike az ezzel járó lehetőségeknek, hogy a szóban forgó feladat teljesítése által mélyebbre áshatom magam mind az általános programozási ismeretek, mind a kiválasztott programozási nyelv alkalmazásába, és az ezen esetben legalkalmasabbnak ítélt videójáték-motor megismerésébe, illetve a játékfejlesztés még egyelőre jogosan igen fiatalnak tartott világába.

Az érintett szakterületekről minden alkalommal szándékomban áll majd egy kisebb-nagyobb feltérképezést felhasználva először egy valóságos és modern képet festeni, majd további részletekkel támogatva felvázolni azt is, hogy mik az okai az adott lehetőségek és módszertanok kapcsán végül szükségszerűen meghozott döntéseimnek.

Utolsónak még megemlítenék egy olyan elvárást a végterméktől, amit talán sokan egyáltalán nem tartanának szem előtt ebben az esetben, de én törekedni fogok rá, hogy azt egy ponton túl, ahogy a nevében is benne van, valóban játéknak lehessen majd tekinteni.

2. IRODALOMKUTATÁS

2.1. Motor

Amikor egy videójáték fejlesztése kapcsán a motorról beszélünk, akkor valójában az adott szoftver „magjáról”, a videójáték-motorról van szó. Régebben előfordult ehelyett még a „grafikus motor” kifejezés használata is, viszont mára ez már csak egy része a gyakorlatilag minden - leginkább - technikai alap-rendszert lefedő játékmotoroknak. A teljesség igénye nélkül gondolok itt természetesen a grafikára, fizikára, animációkra, hangra, pályaszerkesztésre, platformkompatibilitásra, a különböző programozható rendszerekre és végül, de közel sem utolsó sorban, magára a szerkesztő alkalmazásra és annak csomagjaira. [1]

Manapság már ritkának számít, ha egy fejlesztőcsapat saját játékmotort hoz létre, ez ugyanis - ahogyan bízom benne, hogy a tisztelt olvasó is ezt megérezte - egy rendkívülien nagy szakértelmet, komoly programozói gárdát és hatalmas költségeket igénylő feladat. Az ilyesmi napjainkban kiváltságnak, illetve egy nagyon részletesen átgondolt és nagy súlyú befektetésnek számít, feltételezve további sikeres, az adott motorra építkező termékek legyártását, illetve annak folyamatos „csiszolását” a jövőben.

Egy új projekt esetében - ahol nincsen házi motor, vagy megvásárolt licensz - a gyakrabban előforduló hozzáállás inkább manapság az, hogy a piacon jelenlévő játékmotorok között körül nézve, a fejlesztő csoport kiválasztja azt az adott programcsomagot, amely az elvárásainak a legjobban megfelel. Ez természetesen szintén nem egy kis költséggel járó befektetés a legtöbb esetben, ám az garantálható, hogy nagyságrendekkel barátibb, mint egy házi mag, illetve az arra még ráépülő rengeteg réteg sajátkezű legyártása. [2] Egy ennyire összetett program nyilvánvalóan nagyon sokrétű. A napjainkban használatosak gyakorlatilag kivétel nélkül tartalmazzák a már fentebb említett különböző fejlesztői komponenseket automatizmussal, vizualizációval, vagyis több, egészen magas szintű absztrakciós eszközzel ellátva, rengeteg redundáns és időigényes munkát szükségtelenné téve.

Gyakori és nagy előnye egy kifejlett motornak még az, hogy általában már előre különböző platformokkal kompatibilis egy ilyen rendszer. Ez nagyjából azt jelenti, hogy amennyiben például az adott mag kompatibilis a „Windows” és a „Mac” operációs rendszerekkel, akkor gombnyomásra vagyunk képesek a fejlesztés bármely pillanatában egy olyan futtatható állományt létrehozni, amely az éppen kiszemelt környezet elvárásainak teljes módon megfelel. Ebben a példában jól demonstrálható, hogy míg elsősorban ez előbbi a „DirectX” [3] addig az utóbbi az „OpenGL” [4] elnevezésű grafikus API-t támogatja.

Sőt, még tovább mehetünk a szoftveres, illetve akár a hardveres környezeti megfelelés kapcsán felvetődő kérdések tekintetében, amikor is - ha már a játékiparról van szó - megemlítjük a videójáték-konzolos és a személyi számítógépes területek közti átjárást, és persze a rivalizálást is. Bár az utóbbi szerencsére csökkenni látszik az elmúlt évek történéseiből kiindulva. Ezen esetben rengeteg többletkérdés tehető még fel. Gondolhatunk itt egyebek között az általános képernyőtávolságra és méretre, az irányítási eszközökre és a bevett szokások közötti számos különbségre. Az előbb említett platformspecifikus információk, illetve az azok alapján megszületők döntések rendkívül érdekes módon mellesleg magára a játékdizájnra is kihatnak. Ez utóbbi olyannyira igaz, hogy sokszor az átjárás csak jelentős játékmenetbeli változtatásokon keresztül biztosítható azesetben legalábbis, ha az adott stúdió egy szerencsésebb színvonalhoz kíván ragaszkodni. [5]

A következőkben be szeretnék mutatni párat a „bérbe vehető”, nevesebb és jelenleg relevánsabb játék-motorok közül. [6]

2.1.1. Unreal

Az „UE” ként is emlegetett játék-motor valószínűleg az első, amely a többségnek a fogalom kapcsán eszébe jut. Ez természetesen nem véletlen, hiszen a már több mint húsz éve létező szoftver a legelterjedtebb licenszelhető játékmotor a „3D” grafikájú konzolos és PC-s játékok körében. A támogatott programozási nyelv itt első sorban a „C++”, amely segítségével kiválóan lehet az adott programot optimalizálni.

Bár a kezdetben elsődlegesen „FPS” (First-Person Shooter, azaz első-személy nézetű lőfegyveres) játékok felépítését támogatta csak, azóta sok más zsánerben is bizonyított már. [7] Az „Epic Games” által fejlesztett környezet mind a mai napig fejlődik, 2015 óta pedig már ingyenesen is elérhető és igen komoly referencia anyagnak örvend. [53][8]

2.1.2. Unity

Saját marketing-kampányuk szerint, de az általános felfogás szerint is: a Unity motor jelenleg az UE legnagyobb vetélytársa. A programcsomag fejlődésének története valóban figyelemreméltó. Támogatja mind a legmodernebb 3D-s technológiákat, illetve a „2D” alapú fejlesztést is, mindezt lassan húsz éves tapasztalattal a cég háta mögött. Bár a motor magja a C++ nyelvet használja, a környezet programozása a Microsoft kezében lévő „C#” nyelven keresztül támogatott. Ezáltal a fejlesztőknek nem kell az alacsonyabb absztrakciós szintekre leereszkedniük, többek között például a programozásból alapvetően fakadó hulladékkezelést illetően. [9] Széles körű bővítő-piaccal rendelkezik, melyen ingyenes és fizetős csomagok is fellelhetőek a különböző textúracsomagoktól kezdve az önálló alrendszerekig. [10]

2.1.3. Godot

Ez a nyílt forráskódú játékmotor egyre nagyobb lendületet nyert az elmúlt években. Óriási erőssége a közösséggel létrejött kapcsolata. Nyíltsága azonban, arra kérem, hogy ne tévessze meg az olvasót: egy komoly és nagyon is alkalmas kettő, illetve 3D-t is magas fokon kezelni képes szoftverről van itt szó. Fejlesztése 2007-ben kezdődött, nyílttá azonban csak hét évre rá, 2014-ben vált. Rengeteg nyelven keresztül lehetséges a kezelése, létezik azonban saját nyelve is, a „GDScript", amely rendkívül magas szinten helyezkedik el absztrakciós szempontból. Eközben viszont szükség esetén akár a C++ adta lehetőségeket is ki lehet használni, sőt ezeket még keverni is lehet igény szerint. Nagy vonalakban gyakorlatilag mindennel rendelkezik, amivel a többi nevesebb és hasonszőrű szoftvercsomag is. [11] A három közül mindenképpen ez a legígéretesebb projekt abból a szempontból, hogy semennyit sem köteles a felhasználónak mindezért fizetnie, akár milliárdos sikerek esetén sem. Véleményem szerint érdemes lesz ezt a fejlesztést is a jövőben is szemmel tartani. [12]

2.2. Grafika

Nagy annak a valószínűsége véleményem szerint, hogy ha egy ilyen programnak a színvonalas elkészítése mentén felmerülő problémákról kérdeznénk meg valakit, akkor annak az személynek az első válasza a grafikus megjelenítéshez kapcsolódna. Az illetőnek ebben az esetben ténylegesen sikerült volna az iparra jellemzően nagy méretű költségvetésekre rátapintania. [13][14]

Ez a komponens persze szó szerint a leg szembe tűnőbb, illetve a technológiai fejlődést is ezen keresztül lehet a legkönnyebben szemléltetni. Az egyik legkönnyebben kiemelhető ilyen jellegű áttörés a kettőről a háromdimenziós megjelenítésre való átváltás volt a játékiparban. Ezt a két egymással szembenálló megjelenítési módot szeretném most érintőlegesen, de szükségszerűen kifejteni. Természetesen mind a kettő nézethez rengeteg alkategória, illetve felhasználási módszertan és megvalósítási megközelítés létezik, én most csupán egy beletekintést szeretnék szolgáltatni az olvasó részére.

Röviden említeném még meg azt, hogy manapság az internetnek hála rengeteg lehetőség létezik arra, hogy mind a két grafikai megjelenítés esetén könnyen és legálisan felhasználható, már előre elkészített építőelemekhez (például textúrákhoz és modellekhez) jussunk hozzá, ráadásul akad ezek között sokszor még olyan is, amelyet kereskedelmi célú szoftverekben is ingyenesen felhasználhatunk. Kiemelném itt a véleményem szerinti két egyik legelterjedtebb ilyen jellegű szolgáltatásokat is nyújtó weboldalt: az „OpenGameArt.Org" -ot és az „itch.io" -t. [15][16]

2.2.1. 2D

A legalapvetőbb formája ennek a grafikai típusnak gyakorlatilag annyiban összefoglalható, hogy adott egy két tengely mentén meghatározott tér, nevezzük ezeket „X" -nek és „Y" -nak, majd ezeknek a mentén különböző területekhez ismétlődő vagy különböző digitális képeket rendelünk hozzá, majd az így módosított teret valamilyen szögben megjelenítjük. Ezt általánosságban úgy szokás megoldani, hogy ezek a képek mind ugyanolyan felbontásúak, teszem azt 32 * 32 pixelméretűek. Ezáltal a „grafikát" mintegy mozaikszerűen lehet kirakni. Ezt, az angol „tile" szót lefordítva, csempe-alapú grafikának hívják. [17]

Ennek a technológiának a segítségével sokféle típusú videójátékot létre lehet hozni a játéktér vizualizációja alapján, ezek közül a három legelterjedtebb a felülről, illetve az oldalról nézetes és az izometrikus. Míg az első kettő úgy gondolom, hogy igencsak egyértelmű, a harmadik egy különleges kategória és szokás még pszeudó 3D, illetve 2.5D-nek is nevezni. Az izometria eléréséhez gyakorlatilag a csempekialakítás matematikai transzformációja szükséges egy „gyémánt alakú" koordinátarendszer elérése érdekében. Ennek következtében a további munkafolyamatokhoz is persze többleterőfeszítés válik szükségessé. A végeredmény azonban többé-kevésbé objektívan esztétikusabb, mint a hagyományosabb megoldások. Talán a legjobb példa erre a grafikai irányzatra a Blizzard Entertainment által fejlesztett és 2000-ben ki is adott, mai napig legendásnak tartott „Diablo 2" akció-videójáték. [18][19]

A már említett digitális, összeollózandó képecskékre több név alatt is ráakadhatunk a szakirodalomban. A kettő leghasználatosabb az angol „Sprite" és „Texture". Mivel egyikre sem igazán találtam magyar megfelelőt, ezért ezeket legfeljebb szprájtnak és textúrának tudom hívni. A kettőt sajnálatos módon sokszor keverik, de az általános konszenzus minden területen a következő szokott lenni: míg a szprájtokat kétdimenziós környezetben értelmezzük leginkább, addig a textúrákat háromdimenziós testek felületére szokás „rávetíteni". [20]

A szprájtokat „ponyvákban", illetve „atlaszokban" szokták elhelyezni annak érdekében, hogy egy ilyen összesített kép-mátrixot a memóriában csak egyszer kelljen eltárolni. Ezután ugyanis az adott ponton elhelyezkedő csempére minden használatkor csak a megfelelő cím alapján kell hivatkozni, nagyban csökkentve a felesleges műveletek számát. Az „atlasz" kifejezést egyébként a kétdimenziós játékok világában is szokás a textúra fogalmával együtt, mint textúra atlasz használni annak ellenére, hogy elviekben a már fent részletezett módon azt valójában a testek kapcsán illene csak emlegetni. Egy ilyen összesített ponyva létrehozása ma már rengeteg módon lehetséges., a modem motorok kivétel nélkül rendelkeznek már saját csomagoló modullal. [21]

2.2.2. 3D

Abból adódóan, hogy a háromdimenziós grafika a kétdimenziós világ elemeire épült, természetes, hogy a kettő közötti kapcsolat szorosnak mondható. A legfőbb, és tulajdonképpen mindent megváltoztató újítás a harmadik „Z" tengely megjelenése, amelynek a segítségével már a valódi testek is szimulálhatóvá váltak. Ez az új megközelítés természetesen sokkal több és bonyolultabb számításhoz vezetett, ami pedig szükségszerűen erősebb, összetettebb és drágább hardverhez.

A kiinduló pont ebben az esetben tehát egy mélységet is tartalmazó tér, amelyet meg kell tölteni modellekkel, 3D-s tárgyakkal, majd ezek felületét a megfelelő textúrákkal a hozzájuk rendelt területeken lefedni, végül pedig a testeket szükségszerűen animálni, azaz mozgatni, és persze megvilágítani. Bízom benne megint csak, hogy érzi az olvasó, hogy ez egy exponenciálisan nagyobb problématér, mint a 2D esetében.

Az effajta vizualizáció már régebb óta létezik például a filmiparban, mint a videójátékokban, ez viszont nem véletlen. Létezik ugyanis egy hatalmas különbség a kettő felhasználási cél között: míg a játékiparban ezeket a kalkulációkat valós időben szükséges egy számítógépnek végrehajtania (ráadásul lehetőleg minél többször), addig a filmek esetében elfogadható, ha egy pár perces ilyen típusú jelenet pár nap alatt mozgóképre fordul. Ezt a folyamatot az angol eredetű „rendering" elnevezés szerint szokás emlegetni. Ebből adódik, hogy az utóbbi megjelenítési forma (CGI: Computer-Generated Imagery) sokkal összetetteb, élethűbb és részletesebb képes lenni.

Az egyik leglényegesebb részlet amire a 3D világában oda kell figyelni, az a poligonszám. Minden modell poligonokból, vagyis sokszögekből áll. Azonban a felhasznált sokszögek típusának meghatározása egyáltalán nem triviális feladat. Általánosságban elmondható, hogy három- és négyszögeket szokás használni. Minden ilyen eljárás esetén ugyanis a legvégső kalkuláció a poligonokat háromszögekre fogja felbontani. Ezért vagy már eredetileg ezokkal dolgoznak a szakemberek, vagy esetleg négyszögekkel, amelyeket még egyszerűen, legfeljebb kétféleképpen lehet háromszögekre felbontani. [22]

Rengeteg részét fel lehetne még térképezni a 3D-s megjelenítés világának, ám jelenleg nem ez a fő célom. Azonban úgy gondolom, hogy egy áttekintés szintjének az eddigiek megfelelnek. Az utolsó pont, amiről itt még röviden beszélni szeretnék, a játékos-perspektíva. Három fő típust szokás ebből a szempontból megkülönböztetni, az első a rögzített 3D-s szemszög. Ez röviden annyit tesz, hogy a kamera egy-egy jelenetben vagy helyiségben mindig ugyanabból a szögből mutatja a játékvilágot. Ez természetesen sok szempontból megkönnyíti az adott szoftver fejlesztését, mivel a háttérben lévő statikus tárgyak és felületek kezelése könnyebb szokott lenni ebben az esetben.

A második kategória a „First-person”, azaz az elsődleges személy szerinti nézet. Ez a személy a játékosra vonatkozik, aki mintegy a játékkarakter bőrébe belebújva látja a világot ezáltal. Ennél a megközelítésnél az az előny, hogy a játkos karaktert magát sok esetben nem szükséges megjeleníteni és animálni, ami bizony igen méretes munkától szabadítja meg a fejlesztőket. Az utolsó a harmadik személy szerinti perspektíva, ahol is a kamera dinamikusan követi a játékos karaktert, általában hátulról. Ez utóbbi testesíti meg a legnagyobb kihívásokat. [23]

2.3. Felhasználói felület

A felhasználói felület a játékos interakciójának szempontjából az egyik legfontosabb audiovizuális játék-komponens. Sokszor egyszerűen csak „UI” néven emlegetik, az angol „User Interface” azonos jelentésű kifejezés nyomán. Ez a rendszer magába foglalja azokat az elemeket, amelyeket a játékos a „játékvilágon kívül” hall és lát, sőt akár például rezgésen keresztül érez a játék adott állapotának tájékoztatása céljául. Ilyenek például a térkép, felszerelés, életerő.

A másik gyakran használatos, ám az előbbivel nem összecserélendő fogalom a felhasználói élmény, avagy a UX (User Experience). Ez inkább az előbbi minőségének leírására szolgál, azaz kifejezi, hogy ezek az interakciók mennyire javítják, vagy akár rontják a játékélményt. Itt elsősorban folyamatokat szokás feltérképezni. [24]

Maga a fizikai eszköz, amivel a játékot irányítjuk természetesen nagyban befolyásolja az előbbi két fogalom viszonyát az éppen adott szoftver esetében, ezt a vetületet viszont úgy döntöttem, hogy egy külön pontban fogom majd tárgyalni.

Ismereteim szerint még nem létezik egységes osztályozási módszertan a UI altípusok vizsgálására, viszont találtam szerencsére egy cikket, amely feldolgoz egy tézist, amelyben két úriember véleményem szerint rendkívül logikus és letisztult módon értekezik eme témakör kapcsán, négy részre bontva azt. [25] A kategorizálást ők a diegetika mentén végezték el.

A szó jelentése a következő: valami diegetikus, ha egy fikcióban azt a benne szereplő lények is észlelik, például a háttérben szóló zeneszám egy filmben, amit a szereplők is hallanak. A következőkben most kifejtem röviden ezt a négy alfajt, felhívom viszont arra a figyelmet, hogy egy játét UI-ja tartalmazhat elemeket akár mind a négy osztályból, nem muszáj tehát az feltétlenül egységes legyen.

2.3.1. Diegetikus

Egy diegetikus elem egy-az-egyben a játékvilág része, helyét abban logikusan foglalja el. Ilyen a „Metro 2033” című alkotásban a hátralévő oxigénmennyiség, illetve a láthatóság leellenőrzése a játékoskarakter órájára pillantva. [26] Az ilyen UI viszont kétélű fegyver: óriási mértékben növelheti az játék „belemerülhetőségét”, angolul az „immersion” erősségét, de másrészt feleslegesen nehézkessé is teheti a kezelést.

2.3.2. Meta

A meta UI-elemek szintén részét képezik a narratíva, azaz az elbeszélésnek, azonban eközben mindenképpen a felhasználói felületsík vetületének a része, kiváló példa erre a „Call of Duty” játéksorozat immáron szerves részévé vált képernyőn megjelenő vérfoltjai, amelyeknek a mennyisége a játékoskarakter egészségének a romlását sugallják. Ezeket a szereplő azonban nyilvánvalóan nem, vagy legalábbis semmiképpen sem ebben a formában lát. [27] Ez a kategória talán sokszor a legnehezebben meghatározható.

2.3.3. Spaciális

Magyarul térbeli. A szóban forgó módszer már jelentősen kitör a játék elbeszéléséből, azonban megfelelő módon megoldva esztétikus és egyben lényegre törő tud lenni. A „Splinter Cell Conviction” nevű akciójátékban ezt az eszközt a fejlesztők gyönyörűen alkalmazzák a játékvilágban logikusan megjelenő „vetítések” formájában. [28] Túlzásba vitele persze rendkívül zavaró lehet, illetve a fikció típusától függően a beleélés mértékét is nagyban ronthatja. A szépség és a használhatóság egészséges arányának fenntartását ajánlott folyamatosan szem előtt tartani.

2.3.4. Nem diegetikus

Ez a szemléletmód mondható a legrégebbinek, mivel természetesen ennek kialakítása a legkönnyebb. Ebben az esetben az adott információközlő elemeket úgy hozzuk létre, hogy - persze lehetőleg azért a narratívát szem előtt tartva és annak megfelelően kialakítva az esztétikumot - szinte kizárólagosan arra összpontosítunk, hogy az információközlés minél gördülékenyebben, egyszerűbben és lényegre törőbben valósulhasson meg. A „World of Warcraft” című MMORPG (Massively Multiplayer Online Role-Playing Game) felhasználói felülete például nagy részben ilyen jellegű elemeket tartalmaz. Egy kivétel ez alól a többi játékoskarakter nevének általános megjelenítése, amely spaciális módon, a karakterek feje fölött történik. [29]

2.4. Hang

A fejlesztők szerencsére egyre inkább odafigyelnek a videójátékok eme, sokszor elhanyagoltnak tűnő aspektusának. Az utóbbi években talán kijelenthető, hogy elterjedni látszik az a felfogás mind az ipar szereplői, mind a fogyasztók körében, hogy a színvonalas hang-dizájn nagyban befolyásolja az élvezhetőségét egy ilyen program használata során. Feltűnő az is, hogy sokszor leplezni is lehet vele olyan hiányosságokat, amelyek esetleg a játék más dimenzióiban találhatók meg. Gondolok itt például olyasmikre, mint amikor egy felhasználói felület nem túl díszes, ám az annak használata során megszólaló hangok mégis élvezhetővé teszik a használatát. [30]

Természetesen, egy videojátékban sok helyen megjelenhet az audió, többek között a felhasználói felület, párbeszédek, aláfestés és egyéb hatások esetében is. Ebbe az alfejezetbe nem kívánkoznék túl mélyen beletekinteni, mivel jelenleg ez a vetület nem szerepel a prioritásosak között, viszont úgy éreztem illene valamilyen formában mégis tisztelegni potenciális, sőt sokszor bebizonyított fontossága előtt. Emiatt csak röviden kiemelnék pár ügyes fogást, amelyeket az „amplifon" elnevezésű internetes folyóirat egyik cikkében leltem fel. [31]

Mielőtt a fent említett kategorizálást kifejteném, megemlítem a grafikai bevezetés végéhez hasonlóan itt is azt, hogy szerencsére a zenék és hanghatások kapcsán mára már szintúgy léteznek ilyen adatbázisokkal rendelkező internetes oldalak. Ezeken rengeteg fájl közül választhatjuk ki azokat, amelyekkel dolgozni szeretnénk, amennyiben nem áll módunkban saját magunknak elkészíteni ezeket a projektünk felélénkítése érdekében. A legelterjedtebb ilyen jellegű weboldal a „Freesound". [32]

2.4.1. Pentatónia

Egyik régóta bevált hanghatás-típus a pentatónia, vagyis az ötfokúság. Ez annyit tesz, hogy bizonyos eseményekhez a pentaton-hangskálának megfelelő hangokat rendelünk, amelyek általánosságban boldog, kedves érzésekre emlékeztetik az embert. A legjobb példa erre a szintlépés hanghatása sok játékban.

2.4.2. Nem-lineáris

A következő a nem-lineáris hangok használata. Ezek gyakorlatilag eltorzított hangok, amelyek valójában nem származhatnának az adott hangszerből, hangszálból. Ezeket általában arra használják, hogy a hallgatóságból kellemetlen, nyomasztó vagy ijesztő érzéseket váltsanak ki. Gondolom egyértelmű, hogy ilyesmit leginkább a thriller, horror típusú videójátékokban szokás alkalmazni.

2.4.3. Adaptív

Az adaptív hang, illetve amire itt valójában gondolok az a zene, kijelenthető, hogy elvárt eleme lett a mai videojátékoknak. A fogalom annyit tesz, hogy a különböző helyzetektől és helyszínektől függ, hogy éppen milyen aláfestés szól. Példának okáért, ha egy lopakodós játéban lebukunk, akkor a zene valami halkabb és feszültséggel telibbről álltában egy akciódúsabb, pánikhangulatú bejátszásra vált.

2.5. Input

Mára már rengeteg beviteli eszköz létezik. Vannak olyanok, amik általános használatúak, de elérhetőek bizonyos körű szoftverekre „szakosodottak" is. Ráadásul, az utóbbi kategória esetén a PC-s világban különböző programok segítségével ezeket a hardvereket gyakorlatilag egy kevés konfigurációt követően szintén bármilyen alkalmazásban tudjuk használni. A lehetőségek tárháza azt mondhatjuk tehát, hogy egyértelműen számos. [33] Az okostelefonok esetében nyilvánvalóan egyeduralkodó maga az érintőképernyő. A konzolvilágban ugyan ez a helyzet, csak a kontroller kapcsán. A PC univerzumában adott a billentyűzet és egér páros használata.

Tudva levő, hogy a fenti három számítógép típus a legmeghatározóbb a játékiparban, ezeken belül pedig messze az említettek a legelterjedtebb beviteli eszközök. A legkomolyabb videójátékoknál használatos inputmetodikák ezen írás szerzője szerint a kontroller és a billentyűzet-egér páros. Az érintős világot itt őszintén szólva leginkább mellőzném, többnyire az általam komolytalannak tartott okostelefonos videójátékiparral szembeni ellenérzéseimből kiindulva. Ezt a fajta gondolkodásmódot egyébiránt más is osztja. [34] Függetlenül ettől viszont, a következőkben erről a háromról szándékozok szolgáltatni egy-egy rövid jellemzést.

2.5.1. Érintőképernyő

Míg kívülről ennek az eszköznek a használata intuitívnak tűnhet, a videójátékok világában gyakorlatilag teljesen új műfajok jöttek létre erre a platformra, mivel valójában rendkívül korlátolt a használata.

A legnagyobb problémát az jelenti, hogy a használó keze, illetve ujja, az elsődleges visszacsatoló felületből, magából a kijelzőből nagy területeket kitakar a használat közben. Ezen felül a UI elemeket pedig igen nagyra kell tervezni annak érdekében, hogy könnyen hozzáférhetőek legyenek akár a még nagyobb végtagokkal rendelkező emberek számára is. Szomorú még az a tény, hogy ez ráadásul csak egy kis része az érintőképernyős játékok problémakörének. [35]

2.5.2. Kontroller

Ez az eszköz „gamepad” néven is ismert még. Mára gyakorlatilag a videójátékvilág szimbólumává vált. Napjainkra szerencsére kifejlődött egy olyan fizikai interfész, amelyet gyakorlatilag nem kimondott módon, de már mindenki betart. Ezt láthatjuk a különböző, a fő csapást képviselő játékkonzolok esetében is, gondolok itt a következőkre: Sony Playstation, Microsoft Xbox, Nintendo Switch. [36][37][38] Ennek a beviteli módszernek tulajdonképpen csak egy darab „hibája” van, mégpedig az inputkombinációk egy ponton túli alacsony száma. Külön megemlíteném a két darab analóg módon működő „joystick” elnevezésű eszközt, amelyek ebben az esetben az egész részei. Ezek segítségével rendkívül látványos mozgásokat érhetnek el a játékosok. [39]

2.5.3. Egér és billentyűzet

A három közül ez bizony a legrégebbi versenyző. Vitathatatlan, hogy az egér pontosságával eddig semelyik alternatíva sem tudott versenyezni. A billentyűzet erőssége pedig egyértelműen abban rejlik, hogy rengeteg gomb áll a felhasználó rendelkezésére. Ebből fakad az a tény, hogy a PC-s videójátékok irányítását szinte kivétel nélkül maximálisan testre lehet szabni, míg a kontrollerek esetében ez a funkció sajnálatos módon még mindig ritkának számít, elsősorban amiatt, hogy eleve nincsen sok lehetőség.

Rengeteg sajátos tulajdonsággal rendelkezik ez a páros, mint ahogyan a másik kettő eszköz is, azonban még egyet meg szeretnék említeni, ez pedig a makrók használatának a lehetősége. Ennek előnyeit leginkább talán a többnyire igen komplex MMORPG szoftverekben figyelhetjük meg. A legelterjedtebb példa erre az, amikor is egy sok képesség-aktiválásából álló műveletet egy darab gombnyomásra lehet redukálni. [40][41]

2.6. Perzisztencia

Az állandóság megteremtése egy videójáték fejlesztésének esetében ma már elengedhetetlen. Míg a régi játékok esetében sokszor előfordult, hogy az ember addig játszott velük, amig tudott - ugyanis a játékállapot elmentésére nem volt lehetőség -, addig a mai, illetve főleg a magas költségvetésű videójátékok általában rendkívül összetett „szörnyetegek”, amelyeket igen sok esetben több tíz, száz vagy akár ezer órába is telhet „teljes mértékben” végig játszani. Az ilyen szoftverek esetében hatalmas állapotterekről beszélünk, egy-egy állapot pedig rengeteg információval lehet megtöltve. A „rossz hír” pedig az, hogy ezeknek egy nagy méretű halmaza az adott játékosprofil előrehaladásához kötött.

Ez röviden tehát annyit tesz, hogy ha például a felhasználó leállítja a programot, az előbb említett adathalmazt az ideiglenes, felejtő memóriából (pl.: RAM: Random Access Memory) a nem felejtő háttértárra (pl.: HDD: Hard Disk Drive, SSD: Solid State Drive, SDC: Secure Digital Card) le kell menteni annak érdekében, hogy az visszatértekor ugyanott folytathassa a játékot, ahol azt abbahagyta.

Ennek a folyamatnak rengeteg módja van, ráadásul függ az adott programnyelvtől, illetve a játékmotortól (tárgyalva egy elkövetkező fejezetben) is, ám az elvek mindig ugyanazok. Egy darab, mindenhol emlegetett fogalom megegyezik, ez pedig nem más, mint a szerializáció („sorosítás”), illetve ennek ellentéte, a deszerializáció. Az előbbi lényege a következő: a megjelölt adatok valamilyen formátumba történő konvertálása, majd azoknak egy egységbe való „tömörítése” és valamilyen módon történő lementése.

Fontos kikötés, hogy csak a valamilyen módon előre megjelölt adatokat dolgozzuk ily módon fel. Az pedig, hogy a végeredmény pontosan milyen formátumú, például bináris vagy másmilyen (esetleg „könnyen” olvasható), megint csak implementációfüggő. A végeredmény tárolása kapcsán három lehetséges megoldást szokás felvázolni: fájlban, adatbázisban vagy pedig memóriában. A deszerializáció nyilvánvalóan az előbb felvázolt folyamat fordítottja.

2.6.1. Fájllal

A legáltalánosabb megoldás a sima, fájl-alapú mentés-rendszer implementációja. Ebben az esetben egy mentési állományba „pakolunk” bele mindent, ami a felhasználó előrehaladásához kapcsolódik. Maga a formátum kiválasztása alapvetően a fejlesztők kezében van. Lehet az például „XML”, „JSON” vagy bináris. A legelterjedtebb a legutóbbi felépítés használata, azt ugyanis teljes mértékben visszafejteni elviekben csak a forráskód segítségével lehetséges. Az ilyesfajta zártságra pedig illik törekedni a szoftverfejlesztés világában: mindig csak annyit fedjünk fel a felhasználó előtt mind kód, mind pedig fájl szinten a programunkból, amennyit csak minimálisan szükséges. A mai környezetek sokrétűen támogatják ezt a fajta állandóságot egyébként. [42]

2.6.2. Adatbázissal

Ezt a megközelítést leginkább a webes alapú szoftvereknél szokás implementálni, amikoris a szoftver-kliens alapvetően folyamatosan kommunikál a szerverrel, amely általában összetett adatbázisokat futtat. Ilyenkor létrehozhatunk egy összekapcsolt táblázatrendszert, majd ennek feldolgozásával menthetjük le, illetve állíthatjuk elő a megfelelő információkat. Ez a megközelítés rendkívül előnyös módon együttműködtethető a különböző entitás-rendszerekkel.

2.6.3. Memóriával

Ez a kategória a jelen esetben valójában irrelevánsnak mondható, ám a teljesség igényé miatt megemlítem. Videójátékokban nincsen sok értelme a memóriába való szerializációnak a kutatásom, és ezáltal a tudomásom szerint. Olyan esetben tudnám ezt csak elképzelni, amikor valamilyen adatot még később fel kívánnánk használni, ámde azt kizárólag a memóriában akamánk tartani úgy, hogy az egy egységet alkosson, sorosított formában. Valóban, a példa igencsak erőltetettnek mondható.

2.7. Programstruktúra

Az alkalmazásfejlesztés világában mára már szerintem nyugodt szívvel kijelenthető, hogy az objektum orientált programozási paradigma, illetve annak számos változata (függetlenül attól, hogy azok ezt a rokonságot éppen tagadják, avagy nem), igencsak elterjedt a régi, egyszerűen csak strukturáltnak tekinthető módszertannal szemben. Ez utóbbi nagyon gyorsan a méltán hírhedt „spagetti" szintű kódátláthatatlansághoz vezet, amint az adott projekt komplexitása növekedésnek indul. [43] Napjainkra szerencsére a „goto", a „break", vagy akár a „continue" nemkívánatos utasításoknak számítanak, a monolitikus alkalmazásoknak viszont fontos részeit képezték.

A jelen írás szempontjából nem tartanám szerencsésnek, ha ezen a ponton fejest is ugranék ebbe a nyúlüregbe, viszont, mint ahogy eddig is, megpróbálok az olvasó számára egy, a fontosabbakat összefogó betekintéssel szolgálni.

Az effajta „szabványosításnak" nagyon nagy haszna van. Ezeket meglátásom szerint két fő területre lehet osztani. Az egyik lényege: elősegíteni azt, hogy a programozó olyan kódot tudjon írni, hogy az minél absztraktabb, olvashatóbb és valóságközelibb legyen. A másik fontos szempont pedig az, hogy a megírt szoftver minél optimálisabban működjön, vagyis minél jobban kihasználja az éppen adott hardver lehetőségeit. A szomorú igazság azonban az, hogy ez a két törekvés a valóságban szinte mindig egymással szemben áll: az egyikre való összpontosítás - inkább előbb, mint utóbb - a másik rovására fog menni.

2.7.1. OOP

Igen könnyen belátható, hogy az OOP (Object-Oriented Programming) rendkívül intuitívan ráilleszthető egy videójáték felépítésére. [44][45]

Jó példa erre a következő: adott egy játékos osztály, ebben szerepelnek különböző adattagok (pl.: életerő, sebesség), függvények és eljárások (pl.: sebez, lép). Maguk az adattagok lehetnek objektumok is (egy-egy osztály példánya), például rendelkezhet egy felszerelés objektumtömbbel, illetve a metódusok (eljárás vagy függvény) is dolgozhatnak objektumokkal. A játékos pedig származhat mondjuk a „karakter" osztályból. Majd létrehozhatunk az osztályok között logikus interfész kapcsolatokat, sőt ezeket az objektumcsoportokat aztán szintén különböző tulajdonságok alapján komponensekbe is sorolhatjuk.

Az absztrakció minősége és mennyisége csak a fejlesztő(k) kreativitásától függ. Létrehozhatunk még különböző elkészültségi szinteket maguk az objektumok kapcsán is annak érdekében, hogy sokkal átláthatóbb legyen a fejlesztés. Márpedig egy videójáték elkészítése közel sem egyszerű. [46]

2.7.2. CBA

Röviden érintettem az OOP kapcsán a komponenseket, vagy alkotóelemeket. Ahogy ott is, lényegük többek között a nagyobb logikai vagy funkcionális egészek egybefoglalása, illetve az azok közti szükség esetén fellépő információközlés meghatározása. Nem is kívánnám ezt a vonalat jóval jobban kifejteni, egyedül azt tenném még ehhez hozzá, hogy ezen elemek újrahasznosíthatósága is magas prioritást kap az ilyen jellegű architektúrák esetén.

Sokféle nézet létezik a CBA (Component-Based Architecture) jellemzőit és hovatartozását illetően, mivel a fogalom még csak mostanság kezd szélesebb körben elterjedni, annak ellenére, hogy az már jó ideje létezik. Ezt a fajta felépítést egyébként meg lehet valósítani OOP segítségével vagy anélkül is. [47]

2.7.3. DOP

A Data-Oriented Programming a játékfejlesztés esetében mondhatni egy többé-kevésbé gyakran megemlítésre kerülő fogalom. Vannak azonban olyan vélekedések, hogy szerepe a számítástechnika fejlődésével egyre kevésbé jelentős. Kivétel nélkül az OOP féle gondolkodással kerül szembeállításra, miközben jómagam, személy szerint bátorkodnék amondó lenni, hogy valójában ez sem választható attól teljes mértékben el, de alapvetően persze valóban egy más fajta szemléletmódot ír le. Ebben az esetben arra kell törekedni, hogy adat-alapú felfogással írjuk meg a programkódot. Mindezt úgy, hogy az azonos típusú adatokat egy adott helyen tudjuk kezelni. OOP esetén egy osztály objektumaiban nyilvánvalóan „ugyanazok" az adatok találhatóak meg, csak sokszor más állapotokban. Azonban tegyük fel, hogy egy ilyen osztály egy adattagját az összes objektumban módosítani szeretnénk, ekkor mindegyik objektumot külön kell „lekérdeznünk", majd módosítanunk.

Sokkal előnyösebb lehet tehát, ha ezek az adatok egy adatszerkezetben lennének elérhetőek, az adott művelet szempontjából jelenleg gyakorlatilag felesleges többletinformáció és működés kezelése nélkül. Ez a fajta megközelítés bizonyos esetekben hatalmas méretű sebességnövekedést eredményezhet, ilyenek például a konzolokra fejlesztett videójátékok, ahol mindig az adott platform limitációihoz kell a fejlesztőcsapatoknak alkalmazkodniuk. A DOP elveinek alkalmazását elsősorban a gyorsítótár-tévesztések (Cache miss) kapcsán lehet felfedezni. [48]

2.8. Játékmenet

Ez az a bizonyos építőeleme a videójátékoknak, amely teljes egészében csak arról szól, hogy az adott szoftver kezelése mennyire szórakoztató. A szórakozás definíciója persze a tervezőktől függ. Rengeteg rendszer sorolható ennek a fogalomnak a hatáskörébe, közülük viszont néhányat emelnék majd csak ki.

Mára már a klasszikus videójátékok stílusát felépítő tulajdonságok a retró fogalom alatt gyűjthetők össze. Hogy ezek a jellemzők pontosabban azonban mit is képviselnek, az már egy nehezebben megválaszolható kérdés. Az igazság az, hogy erősen úgy érzem, hogy ez bizony emberfüggő. Alapvetően ilyenkor a „régi" játékokra gondolunk, valójában azonban ki tudja, hogy kinek mi számít réginek? A következőkben megkísérlem a saját meghatározásomat létrehozni azzal a céllal, hogy az minél általánosabb legyen.

Ahogy kicsit is utánanéz az ember, rengeteg összesítő játéklistát talál ebben a kategóriában, akár történelmi lebontásban is. [49] Természetesen íródtak könyvek is a témában, talán a legfigyelemreméltóbb viszont a nonprofit, közösség által előállított (közöttük több nagy volumenű névvel), rendkívül átfogó „The CRPG Book Project", magyarul a „A CRPG (Computer, azaz Számítógépes RPG) könyv projekt". [50]

Jómagam most két játékot fogok a következőkben alapul venni, de ezeken kívül még másokat is meg fogok persze majd említeni, amikor szükséges. A két megvizsgálandó videójáték tehát akkor a Diablo II („D2") és a Star Wars: Knights of the Old Republic II - The Sith Lords („KOTOR2"). [51] Többek között, de elsődlegesen a saját tapasztalataimból fogok kiindulni.

Az első tehát, a már emlegetett 2000-es megjelenésű akció-szerepjáték. Egy akkoriban csodálatosnak számító izometrikus grafikát felhasználva kápráztatta el a játékosokat a procedurálisan generált, külső területeken elhelyezkedő, legjobb felszerelésért folytatott és sosem abbamaradó küzdelemmel. Ezek mellet még számos másik újdonságot is tartogatott a mind a mai napig aktív többjátékos közösségét nem leszámítva.

A 2004 vége környékén napvilágot látott 3D-s Star Wars játék egy személyes kedvencem. Az Obsidian kezei közül kikerült folytatás sok szempontból igencsak más irányt vett, mint a Bioware eredeti szerepjátéka. Taktikus harcrendszere, kitűnően realizált karakterei és felejthetetlen hangulata a mai napig gyakran standardként teszi emlegetetté, mindannak ellenére, hogy a létrehozásának története több szempontból is rámutat a rendkívül nehézkes fejlesztésre, amely az eredetileg kiadott szoftver hiányosságain és hibáin keresztül is észrevehető.

Mind a kettő játék most már nagyjából kijelenthetően húsz éve jelent meg. Úgy gondolom, hogy már az ő halmazukon keresztül is sokszínűen lehetséges a retró RPG-k jellemzése, anélkül, hogy az olvasó elvesztené a fonalat a túlzott mennyiségű cím felemlegetése, illetve a túlságosan régi, ma már nehezen átlátható címek ecsetelése során.

Lassan már nem gondolom, hogy a klasszikusok fogalmát kizárólagosan a 2D-s játékok körére kéne lekorlátozni, ezt a vetületet én ma már inkább csak technikai apróságnak tekinteném.

2.8.1. Felépítés

Ezt az aspektust először a KOTOR2-vel kezdeném, mivel az a tradicionálisabb felépítésű a kettő közül. A játék során különböző bolygók meglátogatására adódik lehetőség, ezáltal egy többnyire nyitott világot kapunk a fontos küldetések alatti zártságtól eltekintve, illetve függetlenül attól, hogy sokszor kell a felhasználónak töltőképernyőket néznie. Bár az utóbbiak már a megjelenés idején se vettek el az embertől túlzottan sok időt.

Többé-kevésbé mindegyik főbb helyszín felépítése hasonló: egy magas népességű város tele feladatokkal és interakciókkal, majd az ezen keresztül elérhető különböző alterületek. A játékmenet lényege a fő küldetés teljesítése, illetve az opcionális mellékszálak felgöngyölítése. Az ezeken keresztül felhalmozható tapasztalati pontok begyűjtésével a játékoskarakter és az egyre számosabb követő-sereg is fejlődik mind a passzív és mind az aktív képességek terén.

A D2 esetén a narratíva (innentől kezdve a kiegészítőt is mindig beleszámítva) öt fejezeten keresztül fedhető fel. Minden fejezet kiindulópontja egy település, majd onnan elindulva érhetőek el a vad, ellenségekkel teli területek. A hangsúly jelen esetben a harcon van: a cél a minél több jó minőségű felszerelés bezsákmányolása. Az aktokon végig kalauzol egy cselekményszál, ám az valójában bizonyos szempontokból elhanyagolható. A többi karakter legfeljebb felbérelhető arra, hogy segítsen nekünk, de komolyabb kapcsolatok nem alakulnak ki ilyen téren. Egy egyszerű, de lényegre törő fejődésrendszer itt is persze megtalálható.

2.8.2. Párbeszéd

A Blizzard alkotásában ez egy rendkívül passzív komponens. Valójában csak annyiból áll, hogy bizonyos barátságos kerektereket megszólítva felkínálkozik pár téma, melynek kapcsán az adott lény véleményét végig hallgathatjuk, illetve elolvashatjuk. A cselekmény előrehaladásával pedig olykor előfordul, hogy újabb monológok megismerésének lehetőségei érhetők el.

A Star Wars játékban ez a rendszer viszont központi helyet foglal el. Rengeteg karakterrel léphetünk interakcióba, és az általános felépítés egy adott beszélgetés esetén egy összetett lehetőség-variáció. Nagyjából mindenre többféle választ adhatunk, ráadásul a különböző történeti szegmenseket ezekkel nagyban be is folyásolhatjuk, az adott beszélgető társ rólunk alkotott véleményét is folyamatosan tovább alakítva.

Az eltérő viselkedésformák követése különböző következményekkel, többek között „jó" és „rossz" pontok elnyerésével, illetve a csapattársaink velünk kapcsolatos állásfoglalásának megváltozásával járhatnak. A kiváló szövegírók és a szinkronszínészek átlagon felüli teljesítménye pedig még magasabb szintekre emeli az élményt.

2.8.3. Kereskedelem

Ebből a szempontból a KOTOR2 jóval egyszerűbbnek mondható. Világában sok helyen megtalálhatóak olyan NPC-k (Non-Player Character, azaz Nem Játékos Karakterek), akiktől különböző tárgyakat tudunk kreditekért felvásárolni. A kínálat leginkább statikus, tehát nem igazán változik egy játék alatt, de egy új játék megkezdésével sem. Ha a felhasználó nem gazdálkodik rendkívül rosszul, akkor érdekes megfigyelni, hogy a játék vége fele a pénz értéke meglehetősen jelentéktelenné válik.

A Diablo-ban minden „városban" több árust is meg lehet lelni, akiknek a készlete folyamatosan, procedurális módon változik, ráadásul illeszkedik is a játékos karakter állapotához. Abban az esetben, amikor is egy vadonban lejárt kör után még mindig nem találtunk jobb felszerelést, akkor is nagy eséllyel tudunk vásárolni ezektől az eladóktól olyasvalamit, ami majd segít a helyzetünkön. Az előbb említett dinamika miatt a pénz (itt arany) hasznossága is jobban alakul, mint a kreditek esetében.

2.8.4. Együttműködés

Amennyiben nem interneten játszunk a D2-vel, akkor a lehetőségek igencsak korlátozottak e tekintetben. Csak a már korábban említett zsoldosokat lehetséges felbérelni, hogy azok az oldalunkon harcoljanak. Ezeket a karaktereket viszont legalább módunkba áll fel is szerelni.

Mindazonáltal, amikor online játszunk megnyílik a lehetőség a más emberekkel való kooperálásra, ami általánosságban véve nagyban függ az adott társaságtól, de legtöbbször azért igen szórakoztató szokott lenni. Kiemelném, hogy az effajta multiplayer, azaz többjátékos alrendszer előnyös módon a játék egészére kiterjed. Természetesen jelen van még az a lehetőség, amikor is a játékosok egymás ellen is kipróbálhatják a képességeiket és karakterüket.

A másik szoftverben viszont szinte mindig a saját karakterünket, illetve még két másikat irányítunk egyszerre (a különlegesebb „szóló" küldetéseket leszámítva). Ezek között ráadásul gyakori az interakció is, mivel a játékos tetteit és szavait a követők időről-időre véleményezni is fogják. A harcok folyamán is csapatban kell gondolkodni.

A barátságos NPC-ket viszont csak a játék előrehaladtával fedezzük fel, ráadásul némely esetben még ebbe a játékmechanikába is sikerült a fejlesztőknek egy kis „nonlinearitást" belecsempészniük. Megfigyelhető ez abból a szempontból legalábbis, hogy éppen melyik szereplő csatlakozik majd hozzánk. Itt egyébként a játékos karakter nemének kiválasztása általi következményekre gondolok első sorban. Akit ez viszont mélyebben érdekelne, annak csak azt javasolnám, hogy próbálja ki a játékot minél előbb, ha ezt eddig még nem tette volna meg. Kár kihagyni.

2.8.5. Harc

A Diablo-sorozat akció-szerepjátékokból áll. Ez a megkülönböztetés természetesen elsődlegesen a harcrendszer felépítésére vonatkozik. A küzdelmek itt valósidejűek, és merem kijelenteni, hogy tizennyolc év távlatából is rendkívüli módon élvezetesek. Az egér bal és jobb gombjához képességeket rendelhetünk hozzá, amelyeket a szintlépések során szerzett pontok segítségével szerezhetünk meg a megfelelő képességfák kiszemelt helyeire befektetett pontok kiosztásával. Ezek mellett minden ilyen alkalommal kapunk még az alap, passzív statisztikákra elkölthető pontokból is, amelyek mind a képességeket, mind pedig a felszerelésünket is befolyásolják.

Az előbb említett zsákmány magunkra aggatása értelemszerűen nagy mértékben hozzájárul a harcbéli lehetőségeinkhez, de módunkban áll elhasználható tárgyakat is, mint például életerő-elixíreket felhasználni.

A „Régi Köztársaság Lovagjai" esetében szintén rendelkezhetünk e kétfajta ponttal a szintlépéseken keresztül, illetve a hasonló elveken alapuló aktív és passzív képességek megszerzése kapcsán is mi dönthetünk, sőt ebben az esetben is megtalálhatók az egyszer-használatos tárgyak. Ezeket azonban az összes hozzánk tartozó karakter esetében kezelnünk kell, bár játék lehetővé teszi az automatikus szintlépést is.

Ennél a programnál viszont körökre osztottak a küzdelmek, és egy sokkal inkább taktikus hangvételt vesznek fel. Mindenki minden körben csak egyvalamit csinálhat. Bármikor szüneteltethetjük az időt is, hogy tudjunk még gondolkozni, illetve az adott helyzetet megfelelően felmérni. Felszerelésünk persze a hagyományokhoz hűen rendkívül meghatározó ebben az esetben is, már csak azt is figyelembe véve, hogy itt a társainkról is nekünk kell gondoskodnunk minden szempontból, amiből is az következik, hogy ez az alrendszer is még külön „mikromenedzsmentet" igényel.

3. SPECIFIKÁCIÓ

3.1. Alapötlet

A kiindulópont egy bizonyos szempontokból „retró stílusú”, első körben angol nyelvezetű (hiszen ez az ipari standard is) szerepjáték prototípusának létrehozása volt. A „Tervezés” fejezetben mélyebben elmerítem majd az olvasót a részletekben, de most természetesen kiemelek pár fontosabb részletet, amelyek egészét a specifikációnak tekintem. El szeretném érni, hogy felépítsek egy olyan videójáték „demó-t" (bemutató példányt), amely játszható, rendelkezik a megfelelő interfészekkel és dokumentációval, illetve egy olyan felépítésű szoftverarchitektúrával, amely segítségével logikusan és letisztult módon tovább építhető a program a jövőben is.

A játékélmény szempontjából arra fogok törekedni, hogy egy egyszerűen kezelhető, egyszerű rendszerekkel rendelkező szerepjátékot hozzak létre. Rendelkezni fog a megfelelő interakciókkal és remélhetően elegendő bájjal annak érdekében, hogy amikor valaki elkezdi használni, akkor lehetőleg ne akarja a tevékenységet azonnal abbahagyni. A játékos rögzített mértékű és gyakoriságú lépésekkel mozoghat majd a csempealapú, kétdimenziós pályákon, amelyeken többek között felfedezhet, feladatokat teljesíthet, párbeszédekbe elegyedhet és akár harcolhat is.

A felhasználó a különböző aktivitásokon keresztül fog majd tapasztalati pontokat begyűjteni. Ezek segítségével majd a karakter fejlődni fog. Minden egyes szintlépés után egyre több pontra lesz majd szükség a következőhöz, de ezt persze ellensúlyozza majd a karakterünk egyre növekvő harci rátermettsége. Minden ilyen jellegű ugrásnál módunkban áll majd különböző aspektusokra pontokat elosztani. Az aspektusok és a szint változásával különböző aktív és passzív képességek válnak majd elérhetővé.

A felszerelések terén is lesz majd választási lehetőségünk. A játékrendszer támogatja majd például a különböző fegyverek, páncélzatok fellelését, megvételét, hordását és eladását. A játékfejlesztőnek nem lesz nehéz különböző ráhatású és kinézetű tárgyakat gyártania és elhelyeznie a világban, az ő kreativitása szab majd csak határt a játékmenet változatossági potenciálja terén.

Mindezt a modern fejlesztőeszközök felhasználásával fogom kivitelezni, amelyek legfőbb erőssége az, hogy ne kelljen minden ilyen jellegű projekthez mindent teljesen felesleges módon a legalsó szintekről újraépíteni, ezáltal időt nyerve azokra a tevékenységekre, amelyek valóban értékteremtők. Ilyenek eszközök egyébként például a motor, annak szerkesztője és csempekezelője, illetve több kitűnő integrált fejlesztési környezet vagy egy egyszerűbb, de megfelelően okos szövegszerkesztő.

3.2. Elvárások

A következőket várom majd el tehát az előbbiekben felvázolt, általam elkészítendő, már korábban említett állapotú és osztályú videójátéktól; tartalmazzon:

- egy játékos karaktert.
- egy kezdeti „világcsonkot”, melynek részét egy település, illetve egy „külső”, vadabb terület képezi.
- (míg az előbbi) legalább egy barátságos karaktert,
- (addig az utóbbi pedig) ellenségeket.
- egy adott rendszerű, „viszonylagos” módon alapszintűnek megítélt harcrendszert.
- egy általános párbeszéd-alrendszert.
- egy egyszerű felszerelés-szisztémát.
- karakterstatisztikákat, és az ehhez kapcsolódó fejlődésrendszer alapjait.
- legalább egy kezdeti történetet.
- kereskedési lehetőséget legalább egy NPC-vel.

Ezt a felsorolást vettem tehát az egész projekt kiindulópontjául. Véleményem szerint azonban az csak természetes, hogy egy alkotói tevékenység során mindig szem előtt tartandó egy bizonyos fokú rugalmasság megőrzése, amely sokszor kisegítheti az illetőt a munkájában. Emellett persze vitathatatlan, hogy egy sikeres munkafolyamat véghezvitelének szempontjából elengedhetetlen a már minél korábban lefektetett problémamegfogalmazás. Ezért is tartottam fontosnak, hogy az irodalomkutatás tevékenységét minél sikeresebben végezzem, ezáltal még mélyebben belelátva a játékipari nehézségekbe.

Megemlíteném még utoljára, bár egyéltalán nem utolsó sorban, a platform kérdését. Példának okáért, az érintőképernyő természetesen elsősorban az okostelefonok, illetve tabletek világában jellemző, azonban egy ideje egyre számosabban bukkannak fel például olyan laptopok, amelyek az emlegetett technológiával is rendelkeznek. Elsősorban én a „klasszikusabb” PC platformra fogok összpontosítani, de továbbfejlesztési lehetőségként megjelenhetnek majd más opciók is.

Bízom benne, hogy a tervezési tevékenység alatt a fentebb felvázolt célok mögött meghúzódó elképzelések majd remélhetőleg fokozatosan kikristályosodnak az olvasó számára is.

4. TERVEZÉS

4.1. Motor

A három felsorolt közül végül úgy döntöttem, hogy a Unity-t fogom választani. Ennek egyik oka, hogy a modern játékipart tekintve az a legelterjedtebb, és fejlődése, illetve népszerűségének növekedése nem látszik megtorpanni. Fontosnak tartottam még azt is, hogy a környezet már egy jó ideje kizárólagosan a C#-ot használja, amely programozási nyelv erősen a szívemhez nőtt egyetemi éveim alatt. Ebből kiindulva megfelelő mennyiségű és minőségű tapasztalatra is már szert sikerült tennem benne. Ezek mellett egy teljesen felkészült játékmotorról van szó, amely rengeteg beépített alrendszerrel támogatja mind a 2- és 3D-s projekteket. Ellátás rendszere elméletileg mindent tartalmaz, amire nekem szükségem lesz a munkafolyamatok kivitelezése során.

4.2. Grafika

A két fő ábrázolási lehetőség közül ezúttal a kétdimenziós, csempe alapú alternatívát választottam sima, felülről nézetes kivitelezésben, ahogy ez szerintem már a specifikációból is kisejlik. A fenti döntésnek egyik oka az, hogy remélhetőleg így majd valamivel egyszerűbb lesz a munkám ebben a tekintetben, ugyanis a grafikai megjelenítés modern kivitelezése merem állítani, hogy nagyságrendekkel több rendelkezésre álló időt feltételezne. Objektívan igaz az, hogy alapszintű grafikát a választott módon még mindig jelentősen könnyebb egy amatőrnek létrehozni és kezelni, mint a másik már emlegetett utat járva. Létezik azonban a választásnak egy másik aspektusa is: ily módon sokkal inkább tudok más részeire összpontosítani a dolgozatomnak, miközben a grafikát egy letisztult, könnyen értelmezhető, kezelhető és bővíthető szinten tartom. A Unity-nek van egy mára már egészen kiforrottnak mondható, beépített csempekezelő rendszere, amellyel saját szprájt vásznakat lehet létrehozni, illetve magukat a csempéket is sokféleképpen lehet javítani, újra szabni és még sok más módon kezelni. Szerencsére, ahogyan ezt már korábban is említettem, több helyről is lehetséges ingyenes textúra atlaszokat beszerezni különböző témákban, felbontásokban. Ez már valóban csak a játékkészítő ízlésére és leleményességére bízza az alapvető kinézet hangulatát. Jómagam ebben az esetben az „OpenGameArt.Org” -on található „Dungeon Crawl” 32 * 32 bites atlaszt („supplemental kiadás”) fogom felhasználni, mivel ez egy közösség által karbantartott, egységes és rendkívül méretes gyűjtemény.

4.3. Felhasználói felület

Ebben a modulban elsődlegesen a nem diegetikus hozzáállást választom. Terveim szerint lesz majd egy állandó része a GUI-nak, amely folyamatosan jelen lesz és lesznek majd olyanok is, amelyeket majd közvetett módon kell előhívnia a játékosnak. Elsődleges

szempont a könnyű és átlátható kezelés. Alapértelmezetten egyszerűbb gombokat, listákat, és az ezeket tartalmazó tárolókat szeretnék majd létrehozni, illetve felhasználni. Kifejezetten nagy hangsúlyt tervezek a skálázhatóságra és a méltán híres és hírhedt „reszponzivitásra", azaz a különböző felbontások és képarányok támogatására fektetni.

4.4. Bevitel

A bevitelt leginkább a billentyűzet felhasználásával akarom kialakítani, a játék alapvető egyszerűsége miatt. Terveim szerint egy ilyen rendszerhez a felhasználók gyorsan hozzá tudnak szokni, és miután ezt a rutint elérték, objektívan kijelenthető, hogy ez a leggyorsabb módszer. Nagy előnye még ennek a választásnak, hogy rendkívül könnyen „portolható" kontrolleres vezérlése egészen addig, amíg nem túl magas a minimálisan szükséges inputok száma. Elképzelhető még bővítési lehetőségként majd az egér-, illetve az érintőképernyős funkcionalitást is. Amennyiben tehát a billentyűzetnél maradunk, a tervezett kiosztás a következőképpen elképzelendő. A mozgást a méltán híres „WASD" billentyűkkel, míg a támadás irányát a nyilakkal tudjuk majd befolyásolni. Magát a támadás cselekedetét a kijelölt pontra az „E" billentyűvel vitelezhetjük ki. Az egyéb interakciókat pedig az „F" és „ENTER" billentyűk segítségével kezdeményezhetjük. Az „ESC" gomb lenyomásával pedig a különböző állapotokból léphetünk majd ki például.

4.5. Perzisztencia

A szerializáció fájl alapú lesz, háttértáras tárolással. A fájl formátumát tekintve a binárist fogom választani, mivel nem szükséges, hogy bárki is könnyen olvassa azt. Technikailag a beépített, bináris formázót fogom alkalmazni, a Unity-nak megfelelő konfigurációval. A rendszer bemutatásához alapvetően elegendő lesz egy gyorsmentési és betöltési rendszer. Ez könnyen skálázható egy bonyolultabb és összetettebb felállás irányába, mivel a funkció legnehezebb része már ezzel is készen lesz. A szerializálandó elemek a megfelelő attribútumokkal meg lesznek jelölve a kódban. Lesznek persze majd olyanok, amelyeket nem szükséges lementenünk, de adódnak természetesen majd olyanok is, amiket viszont igen. Gondolhatunk itt többek között a karakterünk statisztikájira, tárgyaira. Az előbbiekhez hozzáteszem itt most azt is, hogy az adatok tárolását a „Programstruktúra" alfejezetben tárgyalt szkriptekkel, illetve a már említett sorosítással fogom megoldani. Az adatbázisok használata ugyanis a videójátékok világában sokszor a szerver oldali kommunikációt is tartalmazó, vagy a nagyon bonyolult egyjátékos játékok esetében szokásos.

4.6. Programstruktúra

Ennek kapcsán a jóval több információt a tervezés „Architektúra” pontja tartalmazza majd, de szeretnék azért már itt is kitérni röviden csak pár részletre. Az OOP elveit fogom használni, többek között mivel a Unity is elsősorban ezt a gondolkodásmódot támogatja, például különböző típusú Unity játékvilági elemekhez különböző kódokat rendelhetünk hozzá gyakorlatilag a kompozíciót megvalósítva. Mindenképpen törekedni fogok az átláthatóságra és a tiszta kód alapelveinek fenntartására. Mindezt természetesen egy (elsősorban) manuális tesztközpontú fejlesztés-szemlélettel kötöm majd egybe. Röviden említem csak meg a láthatóság kapcsán fennálló hozzáállásomat. Klasszikus módon törekszem arra, hogy mindig a legminimálisabb láthatóságot adjam a mezőimnek, tulajdonságaimnak, metódusaimnak, osztályaimnak stb. Mindenütt kezelem a névtereket is, pedig egyébként a Unity-ben ez nem mindig szokás. A „public” hozzáférést kerülöm, ha szükséges egy nyíltabb pont, akkor az „internal” elérhetőséget használom. Bizonyos esetekben egyébként a Unity miatt például muszáj a public-ot használnom. Természetesen tudom azt, hogy mondjuk reflexióval sok mindent meg lehet kerülni, de magának a kód minőségének is jót tesz, ha valamilyen láthatósági elvekhez tartja magát az ember. Ahol csak megtehetem string-ek helyett felsorolásokat fogok alkalmazni olyankor, amikor valami egyedit szeretnék azonosítani. A sztringek ilyennemű alkalmazásának legnagyobb hátránya az, hogy irtózatosan könnyű őket elírni. Ráadásul ugyanazt az értéket több helyen, több féleképpen is módosíthatjuk úgy, ahogy azt esetleg valójában nem is szerettük volna. Még a legfejlettebb fejlesztői eszközök se nagyon fognak emiatt szólni, pontosan azért, mert csak szövegekről van szó. Az enum-ok viszont t önmagukban típusok. Ebből kifolyólag abban a pillanatban, hogy elírjuk őket, a kód nem fordul le. Ez igen szembetűnő és ezáltal sokkal biztonságosabb. A hátránya ennek a megközelítésnek persze az, hogy emiatt viszont akár egész sok fajta és nagyságú enum-ot karban kell tartani. Egy másik odafigyelni való még az, hogy amikor egy enum-ot módosítunk, akkor az enum-ok mögött rejlő egész számos azonosítók sorrendje felbomolhat. Ez alapesetben implicit, azonban a fejlesztő megteheti, hogy egy enum-on belül minden értékhez kézzel rendeli hozzá a szükséges integer-t. A felsorolásokhoz hasonlóan ez is egy olyan döntés, ahol is törekedni fogok egy eszköz használatára más alternatívákkal szemben. A C#-ban ezeket „Dictionary” -nek hívják. Ezek is természetesen kulcs-érték párokból állnak. A nagy előnyük az, hogy nagyjából O(1)-es gyorsasággal lehet keresni bennük, vagyis mivel a kulcsok eredetiek, könnyű őket az értékeknek megfeleltetni. A hátrány persze a keresés, ha nagyon sokszor akarunk egy struktúrában nézelődni, lehet jobban járunk egy más felépítésűvel. Pont a felsorolások miatt viszont én sokszor tudni fogom, hogy pontosan mi az, amit vissza szeretnék kapni. Ebből kifolyólag az ilyen esetekben például igyekezni fogok szótárakat használni. Mi több, terveim szerint az is elő fog majd fordulni, hogy szótár-értékekben is fogok majd szótárakat használni.

4.7. Játékmenet

A játékvilági felépítés alapvetően két részből fog állni, amelyek együtt egyébként megfelelhetnek az elemzett játékok egy-egy narratív komponensének: egy város és egy veszélyesebb terület. Jelen lesz a lehetőség a párbeszédek lefolytatására is, amelyeket majd természetesen jómagam fogok megírni. Elsődlegesen a KOTOR2-ben megemlített hozzáállás megvalósíthatóságát kívánnám majd bemutatni. A kereskedelem legalább egy NPC-vel lehetséges lesz, egy egységes pénzformát felhasználva. Az együttműködés mentén legalább egy-egy példa nyomán a feladatrendszert említeném, ahol is fő és mellékküldetések kipróbálására is lesz mód. A harcrendszer egy speciális, körökre osztott, ámde „globális" alrendszer alapján fog működni. Ahogyan azt már korábban említettem, a világ a kiszemelt csempe-atlasz 32 * 32-es kirakósdarabjaiból fog majd felépülni. Úgy gondolom, hogy nagyságrendileg 100 * 100 -as pályaméretnél semmiképpen sem lesz szükség nagyobbra, de erősen úgy érzem, hogy egy-egy tér ennél valójában jóval kisebb lesz. Mindegyik „map" több szintből fog állni a megjelenítés szempontjából, lesznek majd csempék, amelyeket másokra rajzolunk rá, így érve el a komplexebb felépítést. Mindegyiken megtalálhatóak majd olyan területek, amelyekre a játékos nem tud rálépni, ilyenek például a falak, zárt ajtók, díszítőelemek vagy más karakterek. Ezt a motor ütközésérzékelő alrendszerének felhasználásával fogom majd megoldani, így kijelölve azokat a szekciókat, amelyek a játékos karakter pozíciójának szempontjából tiltottnak tekinthetők. A karakterünk rendelkezni fog majd egy egyszerűbb fizikai alrendszerrel, amely eseménykezelés segítségével tájékoztat arról, hogy ilyen helyre kerültünk. Lesznek majd természetesen olyan elemek, amelyekkel interakcióba léphetünk, például zsákmányolás, párbeszéd és harc. Annyit tennék még ehhez viszont hozzá, hogy terveim szerint minden pályához külön „scene" -t, azaz jelenetet fogok hozzárendelni a megfelelő felépítés elérése érdekében. A motor ezt a rendszert adja azt segítve, hogy egységeket hozhassunk létre a pályáink alapján. A lépések csempéről-csempére történnek mindegyik irányba, az átlósakat leszámítva. A karakterek alapértelmezetten egyszerre csak egyet léphetnek. Lesz majd egy időlimit, ami azt méri, hogy a játékos karakter mikor léphet megint, persze itt azért egy másodpercnek egy töredékére kell csak gondolni alapértelmezetten, ám lesznek majd olyan tényezők, amelyek ezen változtatni is tudnak. A jelenetben a többi lépni szándékozó karakter is csak akkor léphet, amikor a játékos karakter már tett valamit a körben. Ennek a rendszernek a működtetésére létre fogok hozni egy „okos" osztálycsoportot a kód újra felhasználás érdekében, majd az adott mozgatói szkriptet minden olyan játékobjektumhoz hozzá fogom csatolni, amelynél ez szükségesnek ítéltetik. A harcrendszer alapvetően a közelharci fegyverekre fog összpontosítani, de továbbfejlesztési lehetőségként adott a távolsági lehetőségek implementálása is. Az ellenségek a játékost követni is tudni fogják (útkeresés), amennyiben észrevették azt. Egyedül a közvetlen és nem átlós szomszédokkal tudunk ilyen kapcsolatba lépni, kivéve a területi sebzést kiváltó képességgel. Hasonlóan a mozgáshoz, itt is egymást váltva folynak az interakciók,

például a játékos karakter üt egyet a tőle jobbra lévő ellenségre, a sebzést a program kiszámítja, majd ezt fordítottan is elvégzi a megfelelő értékeket figyelembe véve. Csak ezután kapjuk megint vissza az irányítást, hogy a következő lépésünket kitalálhassuk. A harcbéli teljesítményünket befolyásolják majd az ellenfél értékei, a sajátjaink és a felszerelésünk is. Természetesen, mint sok minden másnál is, itt is kísérhetik a különböző tetteket hozzáadott audiovizuális hatások, például ütés-hang, vérfolt és az elszenvedett sebzés felszálló értéke a megfelelő karakter fölött. Jelen lesz ennek kapcsán is egy olyan osztály-kapcsolati stratégia, amivel biztosítani fogom azt, hogy a harcolni képes kreatúrák mind rendelkezzenek az alapvetően ebből a szempontból szükséges változókkal és metódusokkal, ilyenek például az életerő, a sebzés értékei és a célpontra történő ütés eljárása. Lesz pár alapvető lehetőségünk arra, hogy a karakterünket jobban rászabhassuk az általunk elképzelt harcos képére. Ilyen lesz a felszerelésrendszer is. Biztosított majd a lehetőség az általunk kedvelt fegyver hordására, páncélzat felöltésére, illetve kiegészítők, mint például egy amulett viselésére. Ezen tárgyak információit a háttérben kis méretűnek nevezhető objektumokban fogom majd leírni egy egységes módon kezelhető osztályhierarchia alapján. Ez architekturálisan majd független és valóban egyenként mérsékelt fájlméretű szkriptekből álló gyűjteményeket jelent majd. Minden ilyen tárgy rendelkezik majd egy enumerációban rögzített típussal és névvel, illetve persze a szükséges játékmenetmódosító tulajdonságokkal és egy leírással. Természetesen egyes felszerelés darabok esetében bizonyos előfeltételeknek is teljesülniük kell. A játékban aspektusokként fellelhetőek a karakter két fő és fejleszthető statisztikája. Talán a vetület szó e legközelibb fordítás jelen esetben. Ez a kettő a test és az elme. Emelésük különböző alstatisztikákat növel, például a test az életerő pontok számát gyarapítja. Ebbe a kategóriába sorolom bele mind az aktív és a passzív képességeket is. Ezek legfőképpen a harcot fogják besorolni, de később elképzelhetőnek tartom az azon kívüli alkalmazásukat is. Elsősorban egyszerűségre fogok itt is törekedni, de miután felépítettem egy ilyen rendszer alapjait, nem lesz nehéz a bővítés. A passzív képességek esetében gondolni lehet a kritikus sebzési esély növelésére, vagy pedig a gyógyítás erősítésére. Az aktívak közül első maga z előbb említett gyógyítás, a második pedig egy a játékos karakter körüli sebzés egy körön belüli kiosztása. Ezek is bizonyos feltételek elérése után válhatnak elérhetővé, ezeket befolyásolhatjuk például az aspektusokba fektetett pontjaink különféle elosztásával. Ezeket a játékos megfelelő alszkriptjében fogom külön-külön objektumokban tárolni. Azt pedig, hogy a felhasználó melyeket sajátította el, illetve, hogy milyen szintig, egy egyszerű szótár felépítésű adatszerkezetben fogom nyomon követni, amely adat majd persze a mentési fájlokba is belekerül. A világ és a mondavilág kialakítása talán inkább irodalmi részlet, ezért ezekről a mellékletek 11.1 és 11.2-es pontjaiban talál a kedves olvasó több részletet.

4.8. Architektúra

Ebben a pontban szándékozom összegyűjteni azokat a tervezési információkat, amelyek már szorosabban kapcsolódnak a fejlesztés előkészítéséhez és aztán persze a megvalósítás megértéséhez. Bizonyos elemek már a tervezés „Programstruktúra" alfejezetében is megemlíttettek, azonban itt azért annál még egy kicsit mélyebbre is szeretnék leásni. Nagyon érdemes kiemelni, hogy maga a fejlesztés mindig két fő vonalon halad egy Unity-s projekt esetében- Értem ezek alatt a játékmotoros szerkesztőben fellelhető jelenet-hierarchiák felépítése és az abban definiált játék-objektumok állapotát, illetve a szigorúbban vett kódbázist.

4.8.1. GameObject

Amikor létrehozunk egy új Unity-s projektet, az valóban nagyjából teljesen üres. Ezután lehetőségünk van üres jeleneteket létrehozni, majd azokba GO (Game Object) hierarchiákat telepíteni. Egy game object kezdetben csak egy transzformációs komponenssel („Transform") rendelkezik, amely beállítja annak a méreteit, forgatását és skáláját. Ezt követően egy ilyen objektumra rákapcsolhatunk Unity-s, harmadik féltől származó, vagy akár saját komponenseket. A komponenseket egyébiránt az határozza meg, hogy a „MonoBehaviour" ősosztályból származnak le. Ezen a kapcsolaton keresztül öröklik meg a rejtett (például a „komponensiséget") és a gyakorta használt képességeket is, például az életciklusi metódusokhoz való hozzáférést. A komponensek nyilvánvalóan a GO-k legfontosabb részei, ezekkel a következő pont még fog is foglalkozni. Azonban megemlítenék itt még pár dolgot, ami egyedi. Egyrészt jelölhető egy játék objektumon, hogy statikus. Ez annyit tesz, hogy illik ezt az olyanoknál megtenni, amelyek esetében tudjuk, hogy transzformációs szempontból nem fognak változni majd a jelenetek alatt. Ezzel a Unity képes optimalizációkat végrehajtani, például kötegelve feldolgozni az ilyen egyszerűbb elemeket. Kiválasztható még egy réteg is. Ez is elsősorban hasonló, segítő, csoport-elkülönítő funkciókat lát el. Én nem nagyon fogom alkalmazni, egyedül a UI elemeknél fogom az előre megadott UI réteget majd kiválasztani. Fontos viszont megemlíteni a „Tag" funkciót. Ez gyakorlatilag egy szöveges meta információ, amivel megkülönböztethetjük az objektumainkat. Itt is elérhetők előre megadottak. de használhatunk sajátokat is. Ezt mindenképpen fel fogom majd használni bizonyos funkciók megvalósításához, például a játékos GO-jának megtalálásához. A GameObject osztály rendelkezik pár statikus metódussal is, így például olyanokkal is, ahol különböző információk alapján megpróbál visszaadni a hívónak jelenetben lévő objektumokat. Egy ilyen a „FindWithTag" metódus, ahol is értelemszerűen egy címkét kell szolgáltatnunk. Egy GO egyébként jelen lehet már egy jelenetben alapból is, de példányosítani is lehet a beépített „Instantiate" függvénnyel. Ennek különböző túltöltései is vannak persze, de általában az objektumot kell megadni, illetve mondjuk egy szülőt és az esetleges eltolást stb. Természetesen megsemmisíteni is lehet ezeket, mégpedig a „Destroy" metódussal. Fontos megemlíteni azt, hogy a pusztítás nem történik meg azonnal, hanem csak a

megadott életciklusi pont után. További, és nagyon fontos GO-s fogalom a „prefab” (prefabricated) is természetesen egy Unity-s fogalom, egy általam is majd sokat használt funkcióról van szó. Lényege, hogy összeállítunk egy tetszés szerinti bonyolultságú játék objektumot, amelyet majd a jövőben is szeretnénk használni, vagy pedig könnyen sokszorosítani, illetve központosítottan (egy helyen) változtatni. Ezt lehetőségünk van a projektben a jelenettől függetlenül lementeni és rá hivatkozni. Módosítani jelenet függően és függetlenül is tudjuk. Amennyiben az utóbbit választjuk, lehetőségünk van rá, hogy a módosítás minden jelenet minden példányában megjelenjen. Tökéletes példa tehát erre a játékos objektum-csoportja, mivel annak a részeit nagyjából mindenütt szeretnénk használni, viszont mindig csak minimálisan egy helyen - a prefabban - módosítani. Erre az egységre majd rengeteg fejezetben lesz is egyébként utalás. Ez idővel egy rendkívül összetett, sok komponensből és sok játék objektumból álló prefab lesz. Vannak ugyanis olyan rendszerek, amik mindig csak akkor vannak jelen, ha a játékos is, és vannak függetlenek. Jó példa erre az előbbire a játékos körüli fényforrás.

4.8.2. MonoBehaviour

Egy fenti módon létrehozott osztálynak lehetősége van tehát a szerkesztőben könnyen látható és kezelhető mezőket definiálnia. Elképzelhetünk itt például egy publikus százalékot jelölő integert, amit egy annotáció segítségével be tudunk korlátozni a, tegyük fel ötven és kilencvenötös értékek közé. Egy másik kiemelendő és talán még fontosabb eszköz ezeknél a szkripteknél, hogy ki tudnak használni különböző jellegű eseményeket, gyakorlatilag ezekre fel tudnak iratkozni. Ilyen például amikor „Awake” -ről beszélünk, ami is a játék objektum létrejöttekor hívódik fel több másikkal együtt. Ezt a fajta motorral való kommunikációt a Unity-s életciklusnak hívjuk, szerencsére egy nagyrészt részletesen dokumentált rendszerről van szó. A már emlegetett Unity-s életciklus kritikus fontosságú a fejlesztés szempontjából. [52] Minden MB életében csak egyszer, az elején fog lefutni a három inicializáló metódus: a már említett Awake, a „Start”, illetve az „OnEnable”. Alapértelmezetten az ember minél előbb végezteti el az ilyen jellegű feladatokat, többek között pontosan azért, hogy ha szükséges, legyen rá még mód, hogy az ilyesmit lépcsőzetesen közelítsük meg. Kezdjük el tehát a fő, belső iteráció megtekintését. Az első lépés a „FixedUpdate” esemény-metódus. Több „Update” -típusú, azaz frissítéses metódus is a rendelkezésünkre áll, ezek közül én a kettő legfontosabbat használom csak: az FixedUpdate és az Update-et. A ciklusban a Fixed jön előbb és ennek az a sajátossága, hogy a fejlesztő maga megadhatja, hogy ez másodpercenként hányszor fusson le. Az én projektemben ez a szám a hatvan. Ennek a beállításnak az érvényesítésére a Unity minden erejével törekedni fog. Persze, előállhat az az eset amikor már erre sincs erőforrás, kijelenthető ekkor azonban az, hogy az adott számítógépen a szoftver „már régen” nem futtatható. Ez a rögzítettség kiemelten fontos, mivel vannak olyan kódsorok, amiket nem akarunk kiszámíthatatlan időközönként futtatni. Híresen ilyenek például a fizikai rendszereket érintők. Ennek a részletnek az érme-szerű másik oldala az Update.

Ennek az események a rendszeressége egy-az-egyben a képfrissítésé. Ezt szokták általánosabb célra használni, például az inputok lekérdezésére. Az Update után találhatóak meg a „Coroutine” -ok „yield” -jei. Az MB-kben elindíthatunk különböző felépítésű korutinokat. Ezek gyakorlatilag könnyű megoldások arra, ha egy komponensben szeretnénk független vagy párhuzamos munkákat is elvégezni. Megemlítem azért még az „OnDisable” és az „OnDestroy” -t. Ezekben lehet a sokszor szükséges „feltakarítási” munkákat elvégezni. Lehetőségünk lesz arra, hogy GO-kat vagy MB-ket kikapcsoljunk, ne pedig elpusztítsuk. Értelemszerűen ilyenkor eltűnnek az adott funkcionalitások, viszont ezeket aztán megint bekapcsolhatjuk újra példányosítás nélkül. Ezzel sok erőforrást meg lehet takarítani. Az ilyen „inactive” vagy „disabled” állapot alatt az életciklusi működések nem funkcionálnak az érintett objektumokban, az adat viszont nem veszik el. Egy GO kikapcsolásával egyébként az összes gyerek is együtt állítódik.

4.8.3. ScriptableObject

A GO-MB páros mellett a másik igen hasznos építőelem számunkra a szintén Unity-s „ScriptableObject” (későbbiekben majd mint SO szerepelhet) ősosztály. [54] Rendelkezik pár az előbbitől különböző tulajdonsággal, de a legfontosabb az, hogy nem szűnik meg a jelenet váltások alatt. Cserébe viszont, mint projekt elem van jelen, nem pedig mint játék objektum, ráadásul igazából egy típusból csak egy kéne létezzen. Ennek megfelelően tehát bármilyen MB (Mono Behaviour) és SO számára regisztrálhatjuk az adott szkriptelhető objektum referenciáját, jelenettől, illetve akár annak hiányától (lásd: Előregyártás) függetlenül. Ez egy óriási előny. Az életciklusa ennek természetesen más, mint egy játék objektumé, pontosabban az azon lévők komponenseké. A dokumentáltsága ennek az iterációnak kínosan gyenge, majdnem nem létező. Az én szememben az egyetlen felhasználható SO-s életciklus-pár az OnEnable és OnDisable. Ezek gyakorlatilag a szoftver elindulása, illetve leállása esetén futnak le, a GO-s megfelelőik előtt, illetve után. Amennyiben az szükséges, a komponensi életciklusokhoz való szinkronizáció kézzel kell majd megtörténjen. A szkriptelhető objektumok alkalmazása a saját és mások filozófiája alapján leginkább egyébiránt akkor indokolt, amikor is egy olyan egységet akarunk létrehozni, amely nem közvetlenül a játékvilágot manipulálja, nem igényel játékvilági pozíciót, tehát valamilyen menedzserképpen kategorizálható feladatot lát el. Ezt sokan „Singleton” GO-kkal oldják meg, miközben ez csak egy nagyon ritkán indokolt és alapvetően piszkos megoldás. Én a menedzser rendszereken kívül is használom majd ezt a megoldást, ezek lesznek majd a dinamikusan váltogatandó, szprájtokat tartalmazó tárolók, vagy pedig a központilag beállítandó és elérhető értékeket kezelő konfigurációk.

4.8.4. Megjelenítés

A Unity jelenleg négy darab fő megjelenítési „mag” -ot ajánl fel. [56] Az első a „legacy”. Ez volt sokáig az alapértelmezett. A másik három már jobban

összekapcsolódik. Tekintsük először a „Scriptable Render Pipeline” -t (SRP). Ez a legelőrehaladottabb, komoly grafikai programozási ismeretekre van hozzá szükség, de persze ezáltal nagyon sok lehetőséget is biztosít. A következő legyen a HDRP, ami a „magas felbontású” RP. Ez az SRP-re alapozott, Unity által biztosított, egyszerűbben felhasználható, de nagyon jó megjelenítésű rendszer. A komolyabb 3D-s kinézetű játékokhoz ajánlott. Az utolsó az URP, vagyis az univerzális RP. Ez a HDRP egyszerűbb testvére, sokkal modernebb és személyre szabhatóbb, mint a legacy, de a használata nagyon egyszerű. Ezt az utolsó választottam én a jelenlegi projekthez. A játék bizonyos részeit ellátom egyébként animációkkal. Erre kiváló példa a gyógyulás képesség effektusa lesz majd. A Unity-s „animation” ablakban állíthatjuk össze ezeket. Valójában nagyon egyszerű dologról van szó: megadhatunk egy idő intervallumot, majd azon referálhatjuk a játék objektumon lévő komponensek állapotait és annak változásait. Az ilyen módon összerakott animációkat animátorokban használhatjuk fel. Ezek gyakorlatilag állapotgépek, ahol is állapot változások gráfját építhetjük fel. Az átmeneteket természetesen elsősorban kódból kezeljük.

4.8.5. Bevitel kezelés

A Unity-ben több mód is van rá, hogy az ilyesmit kezeljük. Én végül a legújabb és egyébként ajánlott megoldást választottam: az „Input System” nevezetű csomagot. [55] Kezdésnek majd kapunk egy eszközt, amivel egy GUI-n keresztül különböző kiosztásokat tudunk létrehozni - jelen esetben kettőt (játékos, felhasználó) -, majd ezekhez akciókat rendelhetünk hozzá, amelyeken belül több szempontból is specifikálhatjuk, hogy mire gondolunk. Itt első sorban egy névre, beviteli típusra és egy beviteli értékre gondoljunk. Például az interakciós bevitel neve az angol „Interaction”, típusa a gomb, értéke pedig az „F” billentyű. Miután testre szabtuk az összes szükséges lehetőséget, a Unity legenerál egy ez alapján vett osztályt, amint mentjük a kiosztásainkat. Ezen keresztül tudunk majd a későbbiekben kód szinten erre a viselkedésre hivatkozni.

4.8.6. UI/UX

A felhasználói felület kialakításához a Unity UI elnevezésű hivatalosan rendelkezésre álló csomagot fogom felhasználni. A mély részletezésébe semmiképp sem megyek bele, de érintőlegesen fel szeretném hívni pár funkcióra a figyelmet azok közül, amelyeket mindenképpen ki fogok majd használni. [57] Amennyiben bizonyos, a csomagból származó komponenseket hozzácsatoljuk egy GO-hoz, annak Transform komponense átváltozik „RectTransform” -má. Az ilyen GO-k elhelyezkedése ugyanis más szabályok alapján működik a játékvilágban, illetve megjelenítésben, hiszen nyilván a UI elemeit kicsit másképpen kell kezelni. Ennek a komponensnek inkább relatív elhelyezkedési beállításai vannak, például X, illetve Y koordináta szerinti eltolódás és a hossz-magasság páros. Itt már azt hiszem érezhető, hogy nagyon fontos szempont ebben

a témakörben az úgynevezett reszponzivitás. Minden UI GO-hierarchia tetején egy „Canvas” komponens van. Ebben adhatjuk meg, hogy az adott felépítést milyen módon kell a rendszer skálázza és rangsorolja például. Én ezt majd a legtöbbször a képernyő szintjén fogom kezelni, hiszen nem az adott játékvilág koordinátájához akarom majd kötni a legtöbb ilyennemű elemet. A rangsorolás persze nagyon magas lesz, vagyis nem igen lesz majd az lehetséges, hogy egy UI elemet más elem eltakarhasson. Egy másik segéd komponens a vászon skálázó. Itt meg fogok adni egy referencia felbontást, mégpedig a teljes HD-jét: 1920 * 1080. Ezzel érem el azt, hogy a UI minden felbontásnál ugyanakkora maradjon. Ellenkező esetben nagyon magas felbontásokon például nehezen láthatóvá válna a felület. Megjelenítés szervezése kapcsán megemlíteném még az olyan elemeket, mint a „Vertical- és HorizonalLayoutGroup” komponenseket. Ezekkel értelemszerű módon lehet elosztásokat - és az azok között lévő területeket - létrehozni, illetve a különböző szülő-gyerek kapcsolatokat beállítani. Fontos még a „LayoutElement” komponens. Ezzel lehet felülírni a fentiekben helyet foglaló elemek öröklött beállításait, például a különböző mérettel kapcsolatos viselkedéseket. Vannak még más érdekes MB-k is, de ezek a legfontosabbak. A lényegi elemek maguk vagy szöveges mezők, képek, vagy valamilyen típusú interakciós elemek, például gombok, kapcsolók, vagy görgetősávok. A szöveges tartalmak megjelenítésére a már a Unity-hez tartozó „TextMeshPro” (TMP) nevű csomagot fogom felhasználni. [58] Ezzel lehet könnyen, minőségi, skálázott és jól látható, illetve könnyen kezelhető szövegekkel dolgozni. Itt említem meg a UI kezelés egyik örök problémakörét: dinamikus tartalmak esetén a fókuszt is nekünk kell majd állítanunk, hiszen az mindig valahol aktív kell legyen, mivel csak gombokat tud a játékos használni, nincsen kurzor. Ezeket a lehetőségeket fogom tehát majd kihasználni annak érdekében, hogy egyszerűen átlátható és kezelhető felületeket hozzak létre a játékos képernyőjének méretétől, arányától és felbontásától függetlenül, amennyire csak jelen esetben szerintem ez elvárható. Az irányítás egyébként a fenti elemek esetében sokszor egy más, nem a bevitel kezelési pontban részletezett, általam implementált, hanem egy beépített rendszeren keresztül történik. A legtöbb esetben ez a rendszer automatikusan és helyesen felismeri az interaktív elemek közti interakciós útvonalakat, ami után is bizonyos inputok maguktól és értelemszerűen használhatók. Ilyenek például a klaviatúrán a nyilak.

5. MEGVALÓSÍTÁS

5.1. Megjelenítés

Kezdjük tehát első körben a megjelenítés felépítésével. Először is ki kellett válogassam a számomra szükséges csempéket, majd ezeket különböző atlaszokban eltárolni és elrendezni. Ezután felhasználtam a beépített és láthatatlan „grid" komponenst egy ős játék objektumban, amelyben meghatároztam, hogy minden cella egyszer egyes legyen, majd ezt a játékvilági origóra alapoztam. Ezután definiáltam három fő megjelenítési réteget: talaj, akadályok, tető. Ezeknek leginkább egy rangsoroló funkciója van: melyik csempe takarhatja melyiket? A definiálás itt annyit jelent, hogy egy esetében létrehozok megint egy új és üres játék objektumot, majd arra ráaggatom a már elérhető és konfigurálható „Tilemap" és a „TilemapRenderer" komponenseket, illetve ezeket megfelelően be is állítom. A beállítás kapcsán itt egyet emelnék ki, létre is kell hozni ezeket a rétegeket és persze használni is kell őket. A rétegeken belül is lehet egyébként még sorrendeket kialakítani egy számérték segítségével. Az akadály réteg itt a legérdekesebb, ugyanis ez értelemszerűen rendelkezik egy fizikai alrendszerrel. Ez annyit tesz, hogy ez még megkapta a „Rigidbody2D", a „TilemapCollider2D" és a „CompositeCollider2D" egységeket. Az első az alapja minden Unity-s fizikai rendszerrel való kommunikációnak, jelen esetben statikusra állítottam, hiszen ez a GO nem fog mozogni. A második szolgál arra, hogy a lefektetett csempék síkját, mint ütközési terület kezelje a rendszer. A harmadik pedig egy segítő komponens, amely a második által kialakított sokszögeket minél kevesebbe optimalizálja. Ehhez a szinthez tartoznak a fák, falak és egyéb akadályok. Említésre méltó még a tető réteg, amihez is gyártottam egy egyszerű komponenst, amely az „OnTriggerEnter2D" és „Exit" eseményeket figyelve lekezeli, ha az adott területre egy játékos címkéjű GO be- és kilép. A belépésnél kikapcsolja az elemen lévő másik komponenst, a tilemap renderer-t, amikor is ennek a rétegnek a megjelenítése megszűnik. A kilépésnél persze ennek csak a fordítottja történik. Felhívom itt a figyelmet a címke vizsgálatára: az ilyen típusú logikáknál általában érdemes a játékos karakterre korlátozni azt. Működése megfigyelhető Cain kriptájában, ahol is belépve egyből látjuk a tető alatti teret, majd miután kilépünk, a játék megint elrejti azt. Rávilágítanék itt még arra, hogy ebben az esetben egy trigger eseményről beszélünk, nem pedig ütközésesről. Ez azért fontos, mert az ilyen fajta ütközőket ilyenkor erre a funkcióra kell beállítani, ily módon ezek nem fognak ütközést kiváltani. Az atlaszokból a különböző rétegeket igény szerint feltöltjük és formáljuk. Több „festési" módszer is rendelkezésünkre áll. A grid-nek hála egyenként is könnyen módosíthatjuk a koordinátákat, de lehetőségünk van kitöltésre és még véletlenszerűsítésre is. Nagyon sokat segíthet, ha a festést úgy végezzük, hogy a csempéket véletlenszerűsítve váltjuk föl minden lehelyezésnél, ezzel elkerülve az erősen szembetűnő ismétlődési mintákat. Egy

általánosabb Unity tervezői képernyőképet a mellékletek 11.6. pontjában talál meg az olvasó.

5.2. Jelenetváltás

Jelenleg összesen három jelenet létezik a projektben: a fő menü, a veszélyes terület és a menedék. A Unity-ben megtudjuk adni a jelenetek sorrendjét, ez alapján pedig a program mindig az így megadott elsőbe fog először betölteni, ez természetesen az említett fő menü. A fő menü egy nagyon egyszerű jelenet, nem is kívánom azt részletezni, ám ez egy fontos szempont. Ez ugyanis nagyban megkönnyíti a háttérben futó folyamatok felépítését, mivel még csak nagyon kevés rendszerre van szükség. Ezen a ponton sajnos be kell ismernem a következőt: jobb híján kénytelen voltam a logika felső, absztrakciós szintjét a minden jelenet elején és végén, az általam implementált, de automatikusan lefutó be- és kilépési animációhoz kötni („kivilágosodás és besötétedés"). Az animációt a „LoadingScreen" prefab tartalmazza és kezeli. Ez önállóan jelen van az első jelenetben, de része a játékos objektumnak is. A két animáció elejére és végére rákötöttem egy trigger-t, ily módon készítettem négy darab „eseményt". Az animáció-s komponens ezeket pedig egy menedzserhez delegálja. Itt következik a „SceneChangeGeneralManager" SO munkája. Az OnEnable-jében csak pár nagyon egyszerű alapértelmezést találunk, ami viszont már első körben is érdekesebb, az a fent említett négy kezelési pont: „PreInitializeSceneBeginning", „PostInitializeSceneBeginning", „PreInitializeSceneEnding", „PostInitializeSceneEnding". Ezzel a rendszerrel tudom azt elérni, hogy egy jelenet csak akkor legyen látható és irányítható, ha már biztos vagyok benne, hogy minden szükséges rendszer megfelelően beállíttatott. Pár példa az ilyenekre: zsákmány, felszerelés, feladat és bevitel kezelés. Része még ennek a menedzsernek egyébként a kívülről szintén elérhető betöltés és kilépés is.

5.3. VFX

Az URP-nek hála rendelkezésre áll több minden is annak érdekében, hogy egy ilyen egyszerű kinézetű játékot is könnyen fel tudjunk ruházni a vizuális hatások csemegéjével. Ez egy nagyon tág fogalom, itt első sorban arra értem, hogy a sima, információ közlő megjelenítésen túl még valamit hozzáadunk a képi világhoz. Az URP több beépített hatást is felajánl, én ezek közül a következőket élesítettem: bloom, kromatikus aberráció, film szemcsék, vignette. A bloom gyakorlatilag a fények virágzása. A fényesebb pixel-csoportok körül létrehoz egy fajta fény-gyűrűt. A kromatikus aberráció a kép szélein megjelenő, esztétikusnak mondható torzulásokot váltja ki. Tulajdonképpen az elromlott kamera lencsék hatását szimulálja. Nagyon sokan ki nem állhatják, másrészt viszont kicsit segíteni szokott a laposabb képhatás elkerülése irányában. A film szemcsézettség jelenléte szintén adhat egy fajta kellemes zajt a túl tiszta vagy unalmas

megjelenéshez. A vignette a kép sarkaiban megjelenő, enyhe sötétítés. Finoman szimulálja az organikusabb látásmódot. Fontos itt kiemelni, hogy a rendszer szépen, automatikusan megoldja azt, hogy ezek az effektek csakis a játékvilági megjelenést befolyásolják, a UI rétegét nem. Ez utóbbi ugyanis nagyon zavaró tudna lenni. Az utolsó pont ehhez kapcsolódóan az fény és árnyékrendszer. Rendkívül hasznos része ez az URP-nek, másrészt viszont még nem teljesen kiforrott a 2D-s támogatást illetően (a HDRP tudtommal még ennyire se szereti ezt a megjelenítési formát). A különböző game object struktúrákhoz hozzáadhatunk „ShadowCaster2D" -ket. Ezeknek a formáját egyelőre sajnos kézzel kell kialakítanunk, állítólag tervezett, hogy az ütközők alapján ezt a rendszer magától is ki tudja majd számítani. Ezután már csak globális és lokális fényforrásokat kellett definiálnom és személyre szabnom, és máris része ez is a játéknak. Sokat változtatott a kinézetén.

5.4. SFX

Az „AudioManager" kivételesen egy MB típusú menedzser, de ezt majd egy kicsivel később részletezem. Inicializáláskor végig megy egy dictionary-n. A szótár kulcsai egy általam készített „Audio" elnevezésű felsorolás elemei, minden kliphez tartozik egy ilyen, tehát egyedi azonosító. Az értékek pedig egy szintén általam definiált „Soundtrack" példányai. Minden ilyenben eltárolom az adott klip referenciáját, egy „playOnAwake" boolean mezőt, egy „loop" értéket és a „volume" mennyiségét. A ciklusban tehát végig megyünk ezeken a Unity-s szerkesztőben beállított értékeken, majd mindegyiknél hozzáadunk egy fentieknek megfelelően konfigurált „AudioSource" komponenst a GO-hoz. Amennyiben az ébredéses lejátszás zászló igaz, el is indítjuk a hangsávot, így indul például el az aláfestő zene a játék elején. A menedzser másik fontos része egy kiajánlott „Play" eljárás. Ennek csak egy audio kulcsot kell megadni, majd az ehhez rendelt bejegyzést a rendszer lejátssza. Ezt használja minden hanghatás-kiváltó rendszer, például a lépés, az ütés és a betöltés hangok esetében is. A következő három pont kötődik a hang menedzserhez és kitűnő alkalom itt őket ennek kapcsán röviden kitárgyalni.

5.4.1. FindableManager

Ez egy absztrakt menedzser osztály, amelyből többen is leszármaznak, például a hang és útkereső menedzserek. Ahogy remélem ez látszik, a hang menedzser azért nem lehet SO típusú, mert szükséges, hogy mint objektum jelen legyen a jelenetekben, és hogy lehessenek komponensei. Ebből kifolyólag viszont felmerülhet például a prefab-ok létrehozásánál, hogy hogyan referáljunk - elegánsan - egy ilyen jellegű objektumot: menedzser, csak egy van belőle, viszont MB eredetű. Ennek orvosolására fejlesztettem ki tehát a „kereshető menedzser" osztályt. Működése nagyon egyszerű, aki tőle származik, automatikusan kap egy statikus metódust. Ennek csak egy címke karakter láncot kell

megadni, és a rendszer megkísérli visszaadni a jelenetben lévő első ily módon jelölt GO-t a FindWithTag-gel, majd magát a leszármaztatott példányát abból kikeresni a GameObject statikus „GetComponent" metódusa segítségével az elérhető legelegánsabb módon: a típust mint generikus metódus-információ adjuk át. Ilyen módon tehát, amennyiben valaki egy ilyen fajta menedzser jelenet-referenciáját keresi, csak az osztályt kell ismerje, a fenti függvény segítségével meg fogja találni azt, amennyiben létezik.

5.4.2. TagConfiguration

Mint említettem, igyekszem elkerülni a „beégetett" szövegek azonosító célú alkalmazását, azonban a címkékkel - mint ahogy az az előző pontból is látszik, ez néha szükséges - ezt hogyan oldjuk meg? Az én válaszom erre itt a „TagConfiguration" SO ötlete volt, amit akkor most itt egy kicsit ki is fejtenék. A kivitelezés nagyon egyszerű. Minden szükséges címkéhez létrehoztam egy teljesen üres prefab-ot, amiben egyedül a címke van kézzel beállítva. Ezeket felvettem referált mezőkként a címke konfigurációba, hiszen jelenet-független előregyártott GO-król van csak szó. Ezt követően bárki számára, aki lekérdezi ezt a konfigurációt, elérhetőek ezek a könnyen és támogatottan referálható objektumok, amelyek viselik a különböző címkéket. Ezzel nagyjából annyit nyertem, hogy csak egy helyen kell regisztrálnom magát a címkét, az adott objektum nevének esetleges átnevezése meg már intelligensen támogatott lesz az IDE által. Így kerülöm el, hogy a kódban címkék forduljanak elő, mint egyszerű string-ek.

5.4.3. SingletonInstanceManager

Szintén az audio menedzser kapcsán kell megemlítenem a „SingletonInstanceManager" -t. Bár korábban már kitértem arra, hogy a singleton szerintem egy programtervezési ellen-minta, alkalmazása sajnos nem mindig kerülhető el. A már említettek miatt ez az eset is az utóbbiak közé sorolható. Az ilyenek segítésére alkottam tehát ezt a komponenst. Lényege a következő. Létre kell hozni egy GO-t az első jelenetben, amikor is már létrehozzuk, beállítjuk és alárendeljük neki (gyermek GO) a szükséges MB menedzsereket. Ez a gyökér GO persze meg kell kapja ezt a komponenst. Az eképpen definiált menedzserek inicializáláskor le kell kérdezzék a szülő komponenst az „IsRegistered" függvény használatával. Amennyiben ez hamissal tér vissza, hozzá kell magukat adják a „RegisteredGameObjects" listához. Az eképpen létrehozott „SessionScopedManagers" GO majd pedig megoldja azt, hogy a jelenetek között életben maradjon, ezáltal pedig a gyerekei is. Ezt a Unity „DontDestroyOnLoad" elnevezésű eljárásával tesszük meg. A rendszernek része még egy példány figyelő logika is. Amennyiben már létezik egy ilyen singleton manager, az adott új és ezáltal felesleges egyed megsemmisíti magát a gyermekeivel együtt. A felhasználó persze a megsemmisítésekből nem fog semmit se észrevenni, hiszen mindez még a jelenet

inicializációs fázisaiban lezajlik. Így lehetséges tehát, hogy a zene például töretlen a jelenetváltások között is.

5.5. Irányítás

A tervezési fázisban említettek után tehát itt következik az ügy érdekesebb része, a kódolás. Az input rendszerrel többféleképpen is együtt működhetünk: feliratkozhatunk rá és le is kérdezhetjük. Mivel a legkezdetibb próbálkozásaim alkalmával a Unity most már legacy-nak számító rendszerét használtam, és mivel abban tudtommal csak az utóbbira volt lehetőség, ezért az átállás után meghagytam ezt a logikát. Ennek még az is az előnye egyébként, hogy pontosan tudom, hogy az életciklus mely pontjában kezelem ezt a problémakört. Itt említem meg röviden, hogy a projekthez készített „README" fájlban megtalálható pár alap információ. Ilyenek többek között az irányítás és a felhasznált kellékek. Készítettem négy, egymástól független, „Controller" komponenst, feladatkörileg a harc, mozgás, célzás és a kezelőfelületre felosztva. A közös mindegyikben a következő: az „Update" életciklusi metódusukban fellelhető egy vagy több lekérdezés az „InputManager" elnevezésű SO-nk irányába. Ott találhatók ugyanis meg az InputSystem és a többi szükséges ellenőrzés releváns metódus-csoportjai. Ilyen például a „CheckForMovementInput" metódus. Amennyiben egy ilyen egy „pozitív" válasszal tér vissza, a kontroller osztályok belevezérelhetnek a további kapcsolódó teendőkbe. Ebben az esetben ez a tényleges fizikai mozgatás lenne, amire majd egy másik alfejezetben térünk viszont csak ki. Az InputManager-ben először is a külön ezt kezelő állapot beviteli menedzser segítségével végzünk el egy ellenőrzést. Ezután megvizsgáljuk, hogy a beviteli típus és a paraméterként beérkező komponens-referencia alapján egy hasítótáblából kiolvasott legutóbbi lekérdezés időpontja óta, eltelt-e már a megfelelő mennyiségű idő. Egy input típust ugyanis több helyről, több céllal is lekérdezhet a játék, ezáltal biztosítva egy nagyobb fokú flexibilitást a fejlesztőnek. A továbbiakban megvizsgáltathatik még, hogy minden entitás és maga a játékos várakozó állapotban van-e, hogy folyik-e harc jelenleg, illetve nem legutolsó sorban, hogy maga a bevitel értéke is eléri-e a megfelelő küszöböt (a gomboknál ez egy bináris érték). Amennyiben minden szükséges lekérdezésnek megfeleltünk - ez persze bevitelek között eltérés forrása lehet -, aktualizáljuk az időbélyegeket, és szükség esetén beviteli állapotot váltunk. Maga a visszatérési érték több fajta is lehet, például előfordul mint „boolean", vagy akár mint irány (fel-le, jobbra-balra). Létezik a projekt kód szintjén egy „Utilities" elnevezésű névtér, amelyben se nem MB-ket, se nem pedig SO-kat nem találunk. Ide rendeztem a tradicionálisabb C# típusaimat. Az egyik alnévtere ennek a felsorolások, ahol is fellelhető a „GameState" enumeráció. Ilyenek például a fő, a kereskedési képernyő, a dialógus állapot stb. Ennek tárolását, kiértékelését és módosítását kezeli a StateInputManager menedzser. Itt eltárolunk különböző bevitelekhez rögzített lehetséges játékállapot kombinációkat, ugyanis nem mindegyiket fogadjuk el minden játék

állapotban. Amennyiben viszont igen, az osztály biztosítja a lehetőséget magára az állapot váltására is.

5.6. Mozgás

Mivel az egyik legfontosabb ütközési halmaz a játékosé, természetesen ennek a gyökérobjektumnak is van egy merev test komponense, ez azonban kinematikus beállítású, tehát elfogadott az is, ha önmaga mozog. Létezik még egyébként dinamikus típusú beállítás is, de ebben a projektben ezt nem használom, mivel az már csak a bonyolultabb szimulációkhoz szükségeltethet. A mozgás kezelését nagyrészt egy „MovementControl" elnevezésű „házi" komponensben végzem el. Az entitások is nagyjából ezt a logikát követik, emiatt tehát őrájuk ebben vonatkozásban nem szándékozom majd külön kitérni. Az Awake-ben beszerezzük az összes szükséges referenciát, mint például a RigidBody testvér komponensét. Ezeket a beszerzéseket nagyon ajánlott egy játék objektum életében csak egyszer elvégezni, mivel a különböző lehívási stratégiáktól függően igen költséges műveletek szoktak lenni. Itt állítom még be a többi mező kezdeti értékét is, annak érdekében, hogy ezek egyértelműek legyenek. Átadjuk még itt a „MovementControlManager" számára a játékos RB-jét is, ez a menedzser levesz a komponensünk „válláról" pár terhet. Kijelenthetem, hogy mindig törekedek arra, hogy a komponenseim minél könnyedebbek legyenek, és hogy a menedzser SO-imban legyen lehetőleg inkább a logika nagy része. Ez több szempontból is előnyös: a menedzsereket gyakran csak egyszer - jelenettől függetlenül - kell inicializálni, és átláthatóbb lesz a kódbázis. Az osztályban az első vizsgálatok „időrendi sorrendben" FU-ban (FixedUpdate) lévőek. Itt három egymástól független elágazást találunk. Megvizsgáljuk, hogy a mozgásnak már véget-kell-e vetni, eldöntjük, hogy szükséges-e egy új pozíció kiszámítása, le ellenőrizzük, hogy szükséges-e a mozgást a jelen állapotban még folytatni. Egy külön, statikus osztályba kiemelt metódus az, hogy megvizsgáljuk, hogy egy merev test elfoglal-e egy adott pozíciót. Ez ugyanis egy sokszor előkerülő probléma. A Unity-ben a pozíciók legtöbbször egy saját, „Vector3" elnevezésű osztállyal kezeltetnek. Ez magába foglalja a három koordináta „float" -típusú lebegőpontos értékeit. Én sokszor ezt egyébként igyekszem átalakítani egy Vector2-es típussá, hiszen a játékben nincsen pozícionális mélység. Mivel ennek a vektomak azonban nagyon „finom" értékei lehetnek, a mozgások idejét pedig nem lehet teljes mértékben megjósolni, ezért szükséges ezt az ellenőrzést egy fajta toleranciával elvégezni. Ebben a projektben ez az érték egy század. Amennyiben tehát a karakter a szükséges pontra megérkezett, többek között beállítjuk a szükséges pozíciót a pont az előbbiekben részletezett tolerancia kijavítása végett. Ebből a játékos semmit se fog látni, ugyanis nagyon kicsi értékekről van csak itt szó. Azonban én szeretném az objektumokat a pontos helyükön tudni. Emellett még több rendszernek is jelzünk a játékos karakter mozgásának végéről, ilyenek például az útkeresés, input kezelés és az entitás menedzsment. A középső vizsgálat esetében (szükséges-e egy új célpozíció kiszámítása)

több változót is leellenőrzünk, illetve megnézzük, hogy az (Update-ben) kapott pozícióra egyáltalán ráléphet-e az objektum. Ez utóbbit egyébként szintén kiszerveztem egy statikus metódusba. Ebben végig megyünk több kollíziós rétegen is, és megvizsgáljuk, hogy az adott pozícióra találunk e „nem-triggeres" ütközőket. Amennyiben a kritériumoknak megfelelünk, beállítjuk a szükséges változókat a megfelelő értékekre, például az új abszolút pozíciót. A harmadik elágazás csak egy zászlót vizsgál meg, amennyiben még mindig (tehát ebben az FU iterációban is) szükséges a haladás, újra meghívjuk a mozgatás parancsát. Ez a parancs a Unity-be beépített RB-s „MovePosition". Ő megpróbálja az objektumot ténylegesen mozgatni, nem pedig áthelyezni. A számára bejövő pozíciót a következőképpen számoljuk ki: jelenlegi koordináta + irány * gyorsaság * „Time.fixedDeltaTime". A legutolsó érték az utolsó biztosítékunk arra az esetre, hogy az esetleges lassulások vagy átpriorizálások alkalmával is kalkulálhassunk azzal az idővel, ami az előző FU óta eltelt. Az Update-ben ezekhez képest már csak kevés dolgot végzünk el. Kettő, egymástól megint csak független elágazást találhatunk meg itt. Megvizsgáljuk, hogy fennáll-e egy újonnan megkezdett mozgás, illetve hogy jelenleg nyugalmi helyzetben van-e az objektum. Amennyiben az első fennáll, megint szólunk több alrendszernek is, például a hang és a kör kezelésnek. A másodiknál megkíséreljük, hogy az input menedzsertől kapunk-e felhasználható választ, tehát a játékos jelez-e éppen a rendszernek. Amikor is az irány beállíttatik, a többi metódus ebből és más mezők értékéből tudni fogja, hogy időszerű egy új mozgás kivitelezésének megkezdése.

5.7. Célzás

A játékos elsődleges interakciós rendszerének alapja a célzó rendszer. A játékos GO hierarchiájában megtalálható a kurzor. Ennek részei egy sprite renderer, egy dobozos ütköző és a célzó kontroller szkriptünk. A kurzor ki és bekapcsolt állapotban is lehet, az előbbi az alapértelmezett. Ezen kívül pedig a játékossal szomszédos négy fő irányban is elhelyezhetjük azt, természetesen ilyenkor be is kapcsolódik - hang, megjelenítés és logikailag. A rendszer rendelkezik még egy szín-kódolásos visszacsatolással is: a kijelölő színe alapértelmezetten sárga, barátságos interakciók felett zöld, ellenségeseknél pedig piros. Az átmozgatás esetében hanghatások is kísérik azt, más hang fájlt lejátszva a kontextustól függően. A tevékenység maga a háttérben különböző interakciós komponenseket figyel és ideiglenesen le is ment annak érdekében, hogy az adott bevitelek alapján ezeket egyből fel is lehessen használni. Egy Awake-ben itt is megtaláljuk az inicializációt. A FixedUpdate metódusában leellenőrizzük, hogy a kívánt és a tényleges kurzor pozíció megegyezik-e, amennyiben igen, kilépünk az eljárásból. A következő egy független elágazás, ahol is megvizsgáljuk, hogy melyik állapotban kéne lennie. Amennyiben a játékos olyan irányt ad meg a kurzornak, ami ugyanaz, mint a jelenlegi, abban az esetben a kurzort vissza kell irányítani a relatív origójába, és ki kell kapcsolni a megjelenítését és az ütközőjét is. A szóban forgó ütköző természetesen szintúgy trigger típusú, hiszen nem akarunk fizikai ütközéseket kiváltani, csak érzékelést. Ennek és az

életciklusnak megfelelően implementáljuk a fenti után az OnTriggerEnter2D és a társ Exit metódusokat. Ezekben értékeljük ki a lehetséges állapotokat, és állítjuk be a megfelelő értékeket. Természetesen azt is le kell kezelnünk, amikor a változást nem közvetlenül a játékos idézi elő, hanem az entitások. Az Update-ben lekezeljük az előbb említett eset végét, amikor a játékos ráhatása nélkül történt változás, illetve megkíséreljük a szokásos módon egy új input parancs kiolvasását, felhasználását és egy újabb állapot változtatási folyamat elindítását. Ehhez a rendszerhez is készítettem egy menedzser objektumot, ahol is ténylegesen kiértékeljük a megtalált GO típusát, és ahol lementjük a más rendszerek részére fontos referenciákat. Mivel szerencsére megkapjuk az másik fél referenciáját az ütközési eseményekben, abból ki tudjuk olvasni a ráaggatott címkét, majd annak alapján megpróbálhatunk belőle komponenseket is lekérni, akár bonyolult hierarchiák gyerekeiből is. Ilyenek MB-k például az esetleg fellelt ellenséges életerő vagy pedig a barátságos interakció komponensek. Eltároljuk még ezen a központi helyen magát a célzási állapotot is egy erre elkészített felsorolásban.

5.8. Támadás

A „CombatController” nagyon egyszerű, mivel felhasználja az input és a küzdelem menedzsereket. Három egymástól független elágazást találunk itt az Update metódusban. Mind a három lekérdez a bevitelitől, majd siker esetén felhívja a CombatControlManager -t. Ez a három a kör átugrás, a támadás és a kiválasztott aktív képesség aktiválása. Ez sem túl összetett, mivel az elsődleges szerepe az, hogy csak őt kelljen az előbb említett osztálynak felhívnia. A kiválasztott képesség aktiválása szintén nagyon egyszerű ezen az oldalon. Van egy kívülről szabályozott tulajdonság, amit le kell ellenőrizni, majd egy öröklött metódus segítségével megpróbálhatjuk használni azt. A képesség-rendszerről mélyebben később lesz majd szó. A sebzés kicsivel érdekesebb. Elkérjük a célzó menedzsertől az aktuálisan - és persze opcionálisan - kiválasztott entitás élet-interakciós komponensét, majd kiszámítjuk a következő pontban tárgyalt menedzser és egy statikus „hasznosságos” metódus segítségével a játékos által kiváltott sebzés mennyiségét a variancia és a kritikus sebzési esély figyelembevételével. Miután rendelkezésre állnak a szükséges értékek, meghívjuk a sebzési metódust az interakciós komponensen. A harc lefolyására a mellékletek 11.6. pontjában talál képernyőképes példát is az olvasó.

5.8.1. CharacterGeneralManager

Itt megragadom az alkalmat arra, hogy szót ejtsek erről az erősen központi menedzserről. Először is, ebben találhatóak meg a játékos karakter statisztikái, például a tapasztalati pont mennyiség, a jelenlegi és maximális életerőpontszám, az alapvető sebzés, az aspektusok értékei stb. Azt hiszem, hogy ezzel könnyen bemutatható egy előbbi állításom, miszerint megéri az ilyesmit nem MB -kben tárolni. Ugyanis, ha ilyen

adatok azokban lennének, minden pályaváltást követően elvesznének. Így viszont nem kell ettől tartani. Mezőkön és tulajdonságokon kívül fellelhető még itt számos eljárás, amelyekkel úgy tudunk módosítani értékeket kívülről, hogy az ezáltal keltett hatásokat központilag lekezeljük és továbbítjuk minden szükséges rendszerbe. A későbbiekben még lesz párról szó bővebben. Ezek között található maga a tényleges kör-lépés kezelése. Itt csak egy „long" típusú változót növelünk meg eggyel, illetve az „EventGeneralManager" (lásd később) segítségével kiváltjuk a „TurnHasAdvanced" eseményt. A long típus azért indokolt, mert egy futás alatt nincsen arra igazán okunk, hogy ezt újra-inicializáljuk, ebből kifolyólag magas kell legyen a maximális értéke. Az esemény pedig sejthető, hogy sok mindenre lesz kihatással.

5.8.2. HealthInteraction

Az interakciók absztrakt névterébe raktam azokat a komponenseket, amelyek arra szolgálnak, hogy játék objektumok egymásról lekérjék őket, majd ezeken keresztül kommunikáljanak, ha szükséges. Egy külön alstruktúrát képez az életinterakciós rendszer. Értelemszerűen, illetve ahogy ezt már érintettem is, elsődlegesen ezen keresztül folynak le a sebzési és gyógyítási folyamatok. Az első pont az absztrakt „HealthInteraction" osztályt illeti. Ez már tartalmazza az életerőt reprezentáló „csík-szprájtok" számítását és kezelését, illetve az alapvető sebzési és gyógyítási logikákat. Tartalmazza még a különböző animációk és hanghatások lejátszásának levezérlését és a halál kinyilvánítását is. A két gyermekosztály a „Player" és „Enemy" variánsok. Bizonyos esetekben mind a kettő kicsit máshogy csinál egy-két dolgot. Halál esetén például a játékos interakciónak le kell kezelni mindenféle rendszert, míg az ellenségnél meg kell próbáljon a rendszer zsákmányt előteremteni. Az utóbbi esetében implementálnom kellett egy külön logikát, ami először is le ellenőrzi, hogy az adott pozíción létezik-e már zsákmány, hiszen ez nincsen kizárva. Utána pedig amennyiben igen, végig kell annak a tartalmán menni, majd minden cikknél el kell dönteni, hogy hozzáadhatunk-e egy értéket hozzá, vagy példányosítani kell az adott GO-t.

5.9. Ellenségek

Minden ilyen prefab rendelkezik az „EnemyDeciderAI" komponenssel a gyökérobjektumán. Az AI fogalom jelenléte előfordulhat, hogy egyes definíciók szerint tudományosan itt nem helytálló, azonban az iparban így szokás elnevezni az NPC vezérlő rendszereket. Ennek egyik legfontosabb része az, hogy megvalósítja az „IReactionDecider" interfészt, amely előír egy „TryToReact" nevű függvényt. Ezen az absztrakción keresztül biztosított, hogy a játékos cselekedeteire válaszoló rendszereket többféleképpen is meg lehessen valósítani úgy, hogy a reakció élesítése viszont minden entitásnál egységes módon történhessen meg. Az ellenséges logika ezen a szinten egyszerű. Amennyiben a felelet lekérdeztetik, az első teendő az entitás és a játékos közti

távolság kiszámítása. A játékos objektum referenciáját ez az osztály már az Awake-ben a TagConfiguration segítségével megszerzi a statikus, Game Object osztályban fellelhető FindWithTag metódussal. Amennyiben a GO már rendelkezésre áll, felhívhatjuk a GetComponent függvényét. Ebben az esetben pedig ez maga a játékos merevteste. Használhatnánk egyébként a transzformációs komponenst is a pozíció megszerzésére, de ez egy árnyalatnyival piszkosabb megoldásnak számít, mivel az objektum a fizikai rendszer része. Ebből kifolyólag ugyanis ilyenkor csak az RB-t illik kezelni, miután az majd maga beállítja a transform-ot is. Egy ilyen távolsági lekérdezést én a következőképpen oldok itt meg. A Vector2 kiajánl egy statikus „Distance" nevű függvényt. Ennek megadhatunk kettő példányt, majd egy float formájában pedig megkapjuk az eredményt. A Unity egyébként mindig float-okat és nem double-öket használ. Ezt az eredményt majd pedig például a „Math.Round" függvény segítségével egy tizedes jegyre kerekítem. A kapott távolságot kétféleképpen használom fel. Egy elágazásban először megvizsgálom, hogy az ellenség a játékos irányába konfigurált legnagyobb támadási távolságon belül helyezkedik-e el. Amennyiben igen, meghívja az „AttackingEnemyReactionAI" nevű testvérkomponensén fellelhető „React" metódust. A másik ellenőrzés a következő: ha az előkonfigurált játékos-követési távolságon belül helyezkedünk el, kövessük őt egy lépés erejéig. Ebbe az ágba futunk bele akkor is, ha már a követés fenn áll. Maga a TryToReact egyébként visszaad egy bool változót, amivel azt jelzi, hogy interaktál-e a játékossal, ebből adódik a Try elnevezés. A támadási logika gyakorlatilag megegyezik a játékoséval, csak persze fordított módon. Kiszámoljuk a megfelelő értékeket, majd felhívjuk a játékos életerő-interakciós komponensén a sebzési metódust. Ez a komponens pedig majd tovább számol, illetve animációkat és hanghatásokat indít el. A követés viszont összetett.

5.10. Útkeresés

Az MI döntés szintjéről letekintve nagyjából a következő történik. Megkérdezzük az útkereső rendszert, hogy tud-e szolgáltatni egy lehetséges útvonalat a játékoshoz és amennyiben igen, megtesszük azon az első lépést fizikailag, hanghatásilag stb. Az A* algoritmus az ipar legelterjedtebb gráfbejáró/útkereséses algoritmusa. A „Dijkstra algoritmus" kiterjesztésének tekinthető. A megvalósításom részletei a mellékletek 11.3. pontja alatt találhatóak meg, szöveges pszeudokód formájában. Mint már viszont említettem, nagyon erősek azok az érvek, amelyek a menedzser osztályok SO-kénti implementálása mellett sorakoznak fel. Akadnak azonban ez alól is kivételek, ilyen pedig a „PathfindingManager". Ennek oka egyszerű: működése erősen kötődik az adott jelenethez. A Unity-s szerkesztőben több módon is konfigurálhatóvá tettem ezt a rendszert. Természetesen azonban először is a jelenet részévé kell tenni abban az esetben, ha abban szeretnénk ezt az alrendszert működtetni. A menedékben például ez nincsen jelen, hiszen jelenleg nincsen ott rá szükség. Mondanom sem kell, hogy ez is egy összetett prefab. Térjünk hát vissza, a beállítás főbb vonalára. A játék felépítése biztosít minket

afelől, hogy annak világa rács alapú. Ez persze reprezentálható egy kétdimenziós mátrixban. Az előbb említett konfigurációs GUI-n tehát a következőket kell megadnunk. Mekkora az a legkisebb rács (ne dolgozzunk feleslegesen nagy adatterekkel), amely lefedi a teljes pályát, illetve milyen irányba és mennyivel legyen annak eltolva az origója, annak érdekében, hogy helyesen tudjunk a különböző pozíciós számítási módok között átalakítani. Az Awake-ben a fentiek alapján hozzuk tehát létre a gráfunkat, amelyben is a pozíciók a csomópontok, és minden csomópontból a négy szomszédosba vezet egy kétirányú, súlyozatlan él. A csúcsok tartalmazzák természetesen azt az információt is, hogy foglaltak, vagy nem. Egy ponton, egy időben csak egy entitás tartózkodhat. A foglaltságot az inicializáláskor minden csomóra automatikusan ellenőrizzük, méghozzá ugyanazzal a módszerrel, amit a mozgásnál is használunk: a „CollisionCalculator" osztály statikus „IsAreaOccupied" metódusával. A menedzser rendelkezik rengeteg funkcióval: ki- és bekapcsolható „debug" rendszerekkel, illetve a mátrix külvilág irányába kiajánlott különböző módú, centralizált, garantáltan jól működő módosítási lehetőségeivel. Az rendszer egyébként több módon is igyekszik minél optimálisabban működni, például törekszik megtartani és újra felhasználni a már kiszámított útvonalakat az előző körökből, ha lehetséges. Ehhez persze figyelembe kell vegye a játékos és a többi teremtény helyzetváltozásait. Ehhez többek között egy MB referencia-útvonal párokat tartalmazó szótárt használok fel, amivel vissza tudom keresni a már az entitás által használt régebbi útvonalat. A már említett kör menedzsment részeként minden játékos eredetű - ilyen szempontból figyelembe veendő - cselekedet után, a jelenetben lévő összes MI-s entitás hozhat egy döntést. Ennek a központi levezényelését az „EntityGeneralManager" SO végzi. Ez tartja számon a lényeket, hogy érvényesül-e a „harci mód", illetve ő tartalmaz több ilyen irányultságú segítő funkciót is.

5.11. Felhasználói felület

A „UIControl" nevű osztály persze megint csak egy kiindulási pont a folyamatban, hiszen valahol egy MB-ben kell elkezdjük az iteratív vizsgálatokat. Az első említésre méltó dolog itt az, hogy nem az Awake-ben kezdjük az inicializálást, hanem a Start-ban. Jelen esetben pont azért kell itt ezt használni, mert egy-két itt szükséges objektum még az Awake-ben állítódik be, ilyen módon pedig garantálhatjuk, hogy eddigre már ezek is készen állnak. Mivel a UI rendszerem sok részből áll, ezért nagyon sok mindent kell itt összegyűjteni. Emiatt készítettem egy kommunikációs komponenst, a „UIManagerDependencyContainer" -t. Ebbe pakoljuk bele az összes referenciát, amelyekre majd a „UIControlManager" szüksége lesz. Itt adjuk még át egyébként a „FloatingValueUIManager" -nek a kamera transform-ját is. Az átadottakra egyébként szolgáltatok itt pár példát: képernyő hierarchia gyökér GO-k (ilyenkor a virtuális képernyőkre gondolok, vagyis például a képesség képernyőre), konténerek, alapértelmezett „Selectable" -ök. Az Update-ben találjuk meg a számos beviteli ellenőrzést. Persze itt is ezt mindig a beviteli menedzserrel kezdjük, majd a pozitív válasz

esetében továbbítjuk a kérést a megfelelő menedzsernek. Megemlítem még, hogy lekezelem itt a két-részes UI eseteket is.

5.11.1. UIControlManager

A menedzser a kontrollertől kapott függőségi konténert elég sokáig dolgozza fel az osztály elején. Lementi a szükséges referenciákat, illetve elvégez még pár, már azok alapján elvégezhető, lokális inicializálást is. Több szótárat is használok itt, hiszen majd, amikor például beérkezik egy képernyő gyökér referenciája, egy jelenet alatt az mindig kiváló kulcs lesz. Az SO nagy részét a képernyő váltó, vagy váltani próbáló metódusok teszik ki, amiket az irányító hív. Ezekben sokszor más műveleteket kell kinyitás, illetve becsukás esetén elvégezni. Ezen a ponton tekintsük most meg a „ToggleScreen" metódust. Ez az utolsó lépés, ami közös, ezt hívja a többi váltó metódus is. Bementként megkapja az adott képernyő GO referenciáját, illetve opcionális paraméterként egy boolean-t, ami megmondja, hogy a metódus kezelheti-e a fókuszt. Ez utóbbi egyébként alapértelmezetten igaz értékű. A metódus alapvetően két részre oszlik: ki- vagy bekapcsoljuk a rendszert? Ha a GO már aktív, akkor ki, és persze fordítva. Bekapcsolásnál először is kikapcsolunk minden képernyőt, ami aktív, majd megkísérelünk az alapértelmezetten kiválasztott elemét egy hasító táblából kikeresni. Ez nem minden képernyőnél létezik. Ezután aktiváljuk a játék objektumot a „SetActive(true)" metódusának jelzett értékű („true") felhívásával. Amennyiben engedélyeztetett a fókusz kezelés, kibocsátunk egy ehhez kapcsolódó eseményt (lásd: EventGeneralManager). A képernyő kikapcsolásánál nagyjából ezek ellentettje történik. A „FallbackLayers" rendszer a következő miatt szükséges. Előfordul, hogy több UI képernyőt is megnyitunk egymás után, ezek a szintek. Erre példa, amikor először a dialógust látjuk, abból aztán a kereskedési és végül a szám beviteli képernyőt. Úgy érzem el lehet képzelni, hogy az ilyen nemű rendszerek nyitogatása és csukogatása a háttérben elég sok, különféle fajta és komplexitású rendszer inicializálásait takarja. Tehát a hibák elkerülése, és a tiszta elkezdési és befejezési folyamatok megtartása érdekében, illetve az iteratív fejlesztési tapasztalatom alapján úgy döntöttem, hogy amennyiben egy szintet visszalépünk, ezt a rendszer a következőképpen teszi meg. Először meghívja az összes nyitott állapotú képernyő bezárását, majd aztán sorban, a nulladik szintről megint felnyitogatja a rendszereket. A fenti példa mentén, amikor is bezárjuk a beviteli mezőt, mindent bezárunk, majd először megint megnyitjuk a dialógust és aztán a kereskedést. Persze ebből a játékos a szokásos módon semmit se fog látni, az egész a képfrissítések között, az adott Update ciklusban folyik majd le. Visszatérve tehát, a visszalépés rétegelő rendszer pont ez a helyzetet szolgálja ki, ha nem is egyszerű, de skálázható módon. Bizonyos képernyő típusok megnyitása esetén tehát fellelhetünk a kódban ilyen nemű kiértékeléseket: mi a jelenlegi (már elavult) állapot, eszerint pedig beállítjuk a legújabb visszatéréses szintet annak a validátorára, illetve a képernyő kapcsolójára, így „rakódnak egymásra" ezek a szintek pár mátrix segítségével. Ezt egy viszonylag bonyolult rendszer

tartja karban, például a „SetFallbackLayers” és a „OpenFallbackScreensIfPresent” metódusok. Az utóbbi metódus hívódik meg a kontrollerben abban az esetben, ha ez indokolt, például amikor kileptünk egy képernyőből, és értelmezhető a visszaeséses rendszer adattere, ez ugyanis mindig törlésre kerül. Amennyiben tehát igen, végig futunk a felgyűlt rétegeken. Ennek a módszernek talán a legérdekesebb része, hogy a több dimenziós tömbök lehetőségének segítségével két részre osztatott ez a logika. Le kell ugynais azt is kezelni, pontosabban meg kell azt különböztetni, amikor csak azért lépegetünk át képernyőkön, mert éppen „visszaesésben vagyunk”. Ilyenkor ugyanis szintén el kell mentenünk ezt a „nyomot”, azonban nem szabad az éppen tartó visszalépést megzavarni. Emiatt ennek a végén a tartalékiból átmásoljuk a rétegeket a fő rendszerbe.

5.11.2. EventGeneralManager

Ezen a ponton használnám ki röviden a lehetőséget az EventGeneralManager áttekintésére. A fejlesztés alatt az eseménykezelésre először az összetettebb UI rendszerek építésénél került sor. Itt ugyanis sokszor egymástól elég „távol” lévő rendszereknek is kommunikálniuk kell. Az ilyenek közötti kapcsolatok közvetlen kezelése pedig már nagyon esetlen szokott lenni. Ilyen esetekben jut általában eszébe a fejlesztőnek, hogy létezzen tehát egy központi egység, ahol is különböző eseményekre bárki könnyen fel tud iratkozni, illetve akár azokat ki is tudja váltani. Idővel viszont, mint ezt ahogy a neve is mutatja, más rendszerek is elkezdtek eseményeket használni. Mivel azonban nincsen olyan sok belőlük, hogy azokat mindenképpen indokolt lenne szétválasztani, létrejött ez az általánosabb működésű SO. Lényegi felépítése a következő: kivételesen mezők formájában kiajánl „Action” típusú „event” -eket. Ezzel gyakorlatilag azt mondjuk, hogy ezeket eljárásokkal figyelhetjük, bizonyosaknál paramétereket is kaphatunk. Őrájuk lehet fel- és leiratkozni. Itt is fontos, hogy a fejlesztő mindig a fel- és leiratkozás párjában gondolkodjon! Elviekben nem illik simán mezőket láthatóvá tenni, azonban az események esetében nincsen szerintem szebb megoldás. Az OnEnable-jében biztosítjuk azt, hogy minden mező az alapértelmezett értékét vegye fel, illetve szólunk még pár másik SO-nak, hogy az eseménykezelő készen áll. A „TriggerSceneLoadRelatedEvents” metódust a már említett jelenetkezelő hívja meg, mivel erre többen is számítanak az inicializálás helyes levezényléséhez. Ezután jönnek az egyszerűbb metódusok. Megtaláljuk itt a kézi fókusz kezelést is: lekérhetjük és beállíthatjuk a kiválasztott GO-t a Unity-s „EventSystem” statikus osztályával. Ez egy beépített és központi komponens, amivel ezt a UI állapotot felül tudjuk írni. Ezek után már csak az esemény kiváltók maradtak hátra. Jöjjön az eseményekre itt pedig pár példa: jelent betöltés, nyelv változás, statisztika változás, harci mód állítás, kiválasztott fegyver változás, feladat frissülés, NPC elhalálozás stb. Egy átlátható, de rendkívül hasznos központi menedzserről volt tehát itt szó.

5.11.3. Skálázás

Az „AutoScrollingUI” komponens egy szerintem érdekes és szokatlanabb a legtöbbhöz képest. UI prefab-okban használom és lényege a következő. Egy olyan GO-ra kell rácsatlakoztatni, amelynek természetesen RectTransform-ja van. Ezután pedig be kell neki állítani, az egyel alatta lévő szöveges GO-t, illetve opcionálisan az arra vonatkozó Selectable-t. Amennyiben megadtunk egy interakciós komponenst, a gördülés csak akkor fog érvénybe lépni, ha azt épp a játékos fókuszálja. Ennek hiányában pedig akkor, ha szükséges. Feltevődik tehát a kérdés, hogy mikor szükséges? Az MB folyamatosan figyeli a szöveget tartalmazó tároló méreteit és azt, hogy a szöveg elfér-e, vagy sem. Amikor is nem fér el, elkezdi a szöveget balra görgetni. Az interakciós működés esetében a gördülés mindig visszaállítódik a kiinduló pontba, amint az elvesztette a kijelölést. Kiemelem, hogy ez a rendszer automatikusan lekezeli az „elő” képarány változásokat is például. A trükk pedig az, hogy ez a komponens leklónozza a szövegest, és hozzákapcsolja a jobb széléhez. Amikor a gördülés alatt a hozzákapcsolt GO eléri azt a pozíciót, amit az eredeti a kezdetben elfoglal, az egészet megint visszatoljuk jobbra. Ebből a felhasználó megint csak nem vesz észre semmit, majd ezután folytathatjuk a balra eltolást. Így érhető el az „örökké gördülés” hatása. A mozgatás maga nem túl érdekes, a RectTransform manipulálásával érhető el értelemszerűen. Az egyetlen talán fontosabb pont itt, hogy felhasználom a beépítetten elérhető „Time.DeltaTime” értéket annak érdekében, hogy a mozgatás gyorsasága ne függjön az FPS-től. A figyelmes olvasó talán felhúzza erre a szemöldökét és felteszi azt a kérdést, hogy ezt miért nem a FixedUpdate-ben végezzük el? Mivel azonban ez az egész a UI rendszert érinti csak, fizikai vonatkozása nincsen ezeknek a GO-knak, megtehetjük, hogy az Update-ben kezeljük le mindezt.t Az utolsó érdekesebb részlet még az, hogy a görgetéssel először egy kicsit mindig várunk azért, hogy a játékosnak legyen elég ideje a tartalom elolvasására. Ezt egy korutinnal oldjuk meg, vagyis elindítunk egy független szálat, ami egy kicsivel később fogja majd megkezdeni a tényleges munkát. A függőleges skálázásért a „VerticallyScrollableUI” (VSU) komponens felel. Ez már összetettebb az előbbinél. Feladata, hogy a hozzákapcsolt konténer GO alatt lévő Selectable gyerekeket és az azokat körülvevő teret, illetve a görgetősávot és magát a görgetést kezelje. Ez is egyébként automatikus, folyamatosan szemmel tartja az ehhez szükséges paramétereket. Ilyenek, hogy jelenleg mekkora a konténer, mekkorák a gyerekek a listában, elférnek-e, aktívak-e. Amennyiben nem férnek el, láthatóvá válik a görgetősáv és a görgő. A navigálással a lista elejére és végére tudunk ugrani, a görgetés pedig automatikusan megvalósul akkor, ha a fókuszált elem nem látható. Természetesen a tartalom is változhat (pl.: kevesebb, több, aktiválás, inaktiválás), emiatt szükséges a navigálási beállítások kézzel kezelése. A sok állapot váltás miatt rendelkezik számos segítő kalkulációval és kívülről elérhető hívásokkal is.

5.11.4. Generálás

A „GeneratableUI” osztály általi generálhatóság biztosítása egy nagyon fontos feladat. Vannak esetek ugyanis, amikor még nem tudjuk előre, hogy futás időben milyen tételeke kell például a játékos felszerelése gyanánt kilistázzunk, ez csak az előrehaladástól függ. Szükséges volt tehát egy rendszer kifejlesztése, aminek a segítségével nem kellett efelől aggódnom. Ennek az első része ez az MB. Ez egy absztrakt komponens, ami nem kifejezetten bonyolult. Valójában nagyrészt olyan, mint egy szerződés, amivel biztosítjuk, hogy az ebből leszármazottak mindenképpen rendelkezzenek egyes referenciákkal. Ilyen például pár menedzser, a UI állapot, a cél VSU, egy ikon, a szöveg stb. Sajnálatos módon itt egy kicsit összemosódik a nézet és a modell, de ez nem mindig elkerülhető. Ebben az esetben ez azt jelenti, hogy majd több játékmeneti osztály is ebből fog leszármazni azért, hogy egységesen tudjam kezelni a generálást. Az előbbit kiegészítő másik alkotóelem az „GeneratorUI”. Ennek kell jelen lennie a UI-októl egyébként független GO-kon, amik alól összegyűjti a GeneratableUI komponenseket, majd ezeket legenerálja egy bereferált UI konténer alá. Azt, hogy milyen elemeket generáljon egy referenciával állíthatjuk be, jelenleg a kettő támogatott a lista elem gomb és a kapcsoló. Mindkettő egy-egy általam összeállított prefab. A komponens bizonyos állapotoknál ébredéskor generál, de meghívom ezt a funkciót kívülről is, így állítódik be egyébként a GeneratableUI-okon a legtöbb referencia is. Említésre méltó még, hogy amennyiben már vannak a konténerben elemek, azokat előbb elpusztítja. Bizonyos modellekhez elhelyezhetünk itt egyébként inicializálási felhívásokat is, jelenleg csak az „Item” használja ezt ki.

5.11.5. Lebegtetés

A „Floating Value” rendszer jelenléte itt lehet, hogy luxusnak tekinthető, de azért ezt csak nem hagyhattam ki. A sebzési és gyógyítási értékek változó nagyságúak, pozíciójúak, színűek és még el is szállnak, sőt az elszállás végén még fokozatosan is tűnnek el. Ezt kezeli a „FloatingValueUIManager” SO a „FloatingValueUI” MB segítségével. A játékos pozíciójáról a rendszer tud, így tehát az elhelyezést csak ellenségek esetén kell megadni. Ezen kívül persze magára az értékre mindig szükség van, illetve választható módon megadhatjuk még azt is, hogy kritikus értékről van-e szó. A felhívásra különböző eljárások állnak a rendelkezésre, amelyek alapján a rendszer eldönti még a színeket, típust, irányt stb. Nem csak számokat támogat, hanem lehetőség van a játékosnak értesítéseket is megjeleníteni, ilyen például az, amit szintlépés esetén látunk. A felhívást követően az SO hozza létre és állítja be az újonnan létrehozott és felépített GO-t és vele a komponenst. A komponens különböző építő tervezési mintás függvények elérését biztosítja, ezekkel szépen és egyszerűen lehet megadni az igényeket. A lebegtetés lefolyásába véletlenszerűség is beépíttetett annak érdekében, hogy ez a játékmeneti elem élvezetesebbnek és organikusabbnak hasson. A példányosított GO hierarchia persze a már megszokott módon egy saját, bereferált prefab. A menedzserben egyébként külön

büszke vagyok a várósoros megoldásomra. Mivel őt akár konkurensen is fel lehet hívni, ezért számoltam azzal az üzenetek tekintetében, hogy azok mindig rendesen láthatóak legyenek, vagyis ne takarják egymást. Ezt úgy oldottam meg, hogy bevezettem egy konkurens sort, amelybe is elhelyezem a már legyártott, de még nem „elengedett" komponenseket. Ennek a rendszernek az adminisztrálását maguk az MB-k végzik, ők állapítják meg, hogy egyből elkezdhetik-e a munkát, vagy pedig egy korutin segítségével majd csak később. A nyilvántartás viszont a menedzserben található természetesen.

5.11.6. LocalizationUIManager

Ez az SO felel azért, hogy egy központi helyről tudja mindenki elérni az adott időpontban beállított nyelvhez tartozó szövegeket. Jelenleg csak az angol nyelv elérhető a játékban, de igyekeztem úgy felépíteni, hogy más nyelvek bevezetése se legyen túl nehézkes, magyarán itt is törekedtem a skálázhatóságra. Három fő mezője van: a nyelvi beállítás enum-ja („Language" típus), egy összetett „szótárak szótára" (nyelv kulcsok alapján elérhető „AssetKey" / string párok) és egy AssetKey/dialógus szótár. Az esemény menedzser inicializációja után felhívódik ennek az „elindítása", emiatt ez egy ülés alatt csak egyszer történik meg. Így történik meg a fentieknek a beállítása. Jelenleg csak a dialógus rendszer szerveztetett innen ki, mivel az egy jóval bonyolultabb témakör, de szükség esetén persze a többi szöveget is át lehetne mozgatni külön definiált helyekre. Maguk az adatok lekérése egyszerű. Alapvetően a dialógus gráfok és a szövegek lekérésére is elég pusztán az egyedi azonosító, a „kiválasztott" nyelvet maga az SO tárolja. A sima szövegnél módunkban áll még több, változó mennyiségű paramétert is megadni, hiszen egyes szövegek támogatják a sztring interpolációt is. Kiemelem azt a tényt még itt, hogy ténylegesen minden a játékban fellelhető szöveg ezen keresztül hívódik le. Tehát, ha létezne még más nyelvi beállítás - amit, megint csak, a struktúra támogat -, a nyelv változó átállításával és a nyelvváltozás esemény kiváltásának segítségével mindenütt lecserélődnének a szöveges tartalmak.

5.12. Fejlődés

A játékos karakter a különböző tevékenységekért tapasztalati pontokat kap. Ilyenek az ellenfelek megsemmisítése és a feladatok teljesítése. Bizonyos határok átlépése szintlépésnek felel meg. Minden ilyen változásnál tehát le kell ezt ellenőrizzük. Az egyenletek, amelyek ezeket számítják a mellékletek 11.4. pontja alatt találhatóak meg. A szintlépésnek több előnye is van. Automatikusan növekedik a kettő aspektus érték, illetve az „UnusedAspectPoints" érték is. Az utóbbi segítségével tudja a játékos jobban személyre szabni a karakterét. Ennek hatására válhatnak például egyes képességek elérhetővé. Ezek két kategóriába sorolhatóak be: az aktív vagy a passzívéba. Egy aktív képességet mindig ki tud a játékos jelölni és aktiválni, amennyiben persze létezik olyan, aminek a feltételeit teljesítette. A passzívak értelemszerűen inkább statisztikai bónuszokat

adnak. Mind ehhez létrehoztam egy absztrakt „Skill” osztályt. Ez a szintén absztrakt és már tárgyalt GeneratableUI-ból származik, hiszen szükségünk lesz a UI-on való megjelenítésére is a képességeknek. Ez a képesség osztály elintéz pár egyszerűbb dolgot a Start, illetve az OnDestroy metódusaiban, illetve meghatároz három fontos absztrakt metódust: „IsUnlocked”, „RefreshAvailability”, „UpdateDescription”. Az utolsó kettőt fel, illetve le is iratkoztatjuk a fentebb említett életciklusokban az esemény menedzserünk „statisticsHaveChanged” eseményére. Talán egyedül az első absztrakt metódus szorul csak egy kis magyarázatra: ezzel lehet majd leellenőrizni, hogy felhasználhatja-e a játékos az adott képességet. A Skill-ből még két absztrakt osztály származik le: az „ActiveSkill” és a „PassiveSkill”. Mindkettő rendelkezik valamennyire hasonló és persze teljesen eltérő jellegzetességekkel is. Elég sok mindent lekezelnek ezek már. Kezdjük először az aktív érdekesebb részeivel. Kapott még pár függőséget, amikre a legtöbb leszármazottnak majd szüksége lehet, ilyenek a címke konfiguráció és a hang menedzser. Rendelkezik még a „Cooldown” kezelésével is. Én a játékban ezt a körök számolásával oldom meg. Minden aktív képességnek tehát van egy ilyen értéke, és még egy olyan is, amely megmondja, hogy melyik az a kör, ahol majd megint lehet őt használni. Ezeknek a kezelését a védett metódusokon keresztül a gyermekek is el tudják érni. Még egy absztrakt eljárást kapnak a leszármazottak ebben az ágban, mégpedig a „TryToUse”. Jelen van még pár UI-hoz kapcsolódó funkció is, ebből kiemelem azt, hogy az aktív képességek kapcsolót, nem pedig gombot használnak. A passzív képesség osztálya érthető okokból egyszerűbb. UI-ilag itt csak gombokól van szó, amelyekre rá lehet lépni, de nem lehet aktiválni. Ennek is van egy absztrakt metódusa, a „Toggle”. Ennek az a szerepe, hogy szabályozza a hatást, amennyiben már bekapcsolható a képesség. Ezután már csak a tényleges implementációs osztályok következnek, ez jelenleg négy darab, kategóriaként kettő-kettő. Itt igazából az adott osztály mindig implementálja a megkövetelt metódusokat, majd mindegyik elvégzi a kontextustól függően a szükséges teendőket, legyen az sebzés, gyógyítás, statisztika változás stb. A UI-os alfejezetnek köszönhetően, ezen a téren már alapvetően nem kell sok mindent megemlítenem. A képességek természetesen GO-kon, a játékos prefab-ban vannak jelen egy alhierarchiában. A gerjesztő rendszer segítségével meg innen generálódnak le a szintén a játékos prefab-ban megtalálható UI képernyők hierarchiájának megfelelő konténerei alá. Így lesz ez majd a felszerelések esetében is például. Röviden tehát: a képesség képernyő két listából áll, az aktív és a passzív képességek halmazaiból. Felhasználtam itt is az eddig még nem említett „Switching” rendszeremet, amivel is itt például a két lista között a tabulátor billentyű segítségével lehet választani. A fókuszálható jelöltek dinamikus kiajánlását a VSU kezeli. A képességeknél elolvasható több információ is, többek között az is, hogy mikor válnak használhatóvá. Az aktív képességeknél egyet ki is tudunk választani az „enter” billentyű segítségével. Ilyenkor egy sárga keret jelöli a jelenleg regisztrált képességet. Újbóli megnyomásra, vagy másik kiválasztásával változtathatunk ezen az állapoton.

5.13. Zsákmány

A fenti témához először a felszerelés alapjainak elemzését kell végrehajtanunk. A rendszer a felső absztrakciós szinten a képességekhez hasonlóan működik. Itt is megtalálhatjuk ezeket egy GO elrendezésben a játékoson belül, és itt is a generátor rendszer segítségével jelenítjük meg ezeket, viszont csak egy listában. Hasonlóan visszatér a sárga kijelölő rendszer is stb. Egy lényegesebb különbség viszont, hogy itt nem ideiglenes a megjelenés, hanem persze folyamatos: ha felszereli magát a játékos egy karddal, akkor az a kezében meg is jelenik. Ezt a játékos origójával könnyen le tudom kezelni, mivel a szprájtok, amiket itt felhasználok erre lettek tervezve, vagyis a kard a játékos szprájtjának kezébe beleilleszkedik. A modell struktúra is hasonló a képességekéhez, viszont ez azért egy kicsit árnyaltabb. Az ehhez kapcsolódó UML-es osztály diagramot a mellékletek 11.5. pontjában helyeztem el. Megtaláljuk itt is először az „Item" fő, absztrakt ősosztályát, ami persze a generálhatóból származik. Ez rendelkezik az alapvető függőségekkel, az inicializálás és az elpusztulás lekezelésével. Már itt megtalálható a mennyiségváltozások rendszere, ami összefűzetett a UI reprezentáció karbantartásával is, tehát például a leírás beszerzése, a darabszám megjelenítése vagy akár az elem elrejtése. Ezek a fajta elemek folyamatos kapcsolatban vannak a hozzájuk kapcsolt VSU-val, ez a programban gyakran előfordul, nem igazán elkerülhető. Az aktiválás esetében már itt megkülönböztetjük, hogy egy felszerelési, egy zsákmányi, vagy pedig egy kereskedelmi elemről van szó. Ennek megfelelően fel tudja hívni a szükség esetén az „InputScreen" rendszerét, ha valamiből több darab is van, és venni vagy zsákmányolni szeretnénk. Erre a képernyőre sajnos nem tudok kitérni, de megemlítek itt annyit, hogy szabályozni lehet az értékeket, és csak számokat fogad el, sőt még a szorzott árat is meg tudja jeleníteni. Ahogy említettem lekezel még mindenféle kereskedelmi funkciót is ez az osztály, a leszármazottaknak jóval kevesebb dologgal kell foglalkozniuk. Négy darab absztrakt metódus található meg benne: a játékos általi eladás árának lekérdezése, aktiválható-e, aktiválás, „felszedés". Belőle még három, szintén absztrakt osztály származik le: a fogyaszthatók, a hordhatók és az egyebek. Ezek mind nagyjából csak megvalósítják az értelemszerűen szükséges metódusokat. Ezekből származnak aztán le a tényleges tárgyak. Ilyenek például a fogyaszthatóknál az életerőt adó varázsital, a hordhatóknál a mellvért és az egyebeknél az arany pénzérme. Van amelyik tehát a játékos egészségi interakciójával tud kommunikálni, mások a statisztikákat emelik, az arany például egyszerűen csak fizetéshez kell. A tárgyak osztályai sok menedzserrel és rendszerrel kommunikálhatnak, de már a legfelső osztály is bereferálja az „ItemGeneralManager". Ez egy igen terebélyes SO, amely mindenféle tárgy-karbantartási és számontartási munkákat elvégez. Ez a modell kiinduló pontja a játékos számára. Itt vannak regisztrálva szótárakban a különböző mennyiségek, jelenlegi tárgy GO-k, miket hord éppen a játékos, zsákmén rendszerek stb. Sokszor ez az az egység, amin keresztül a UI reprezentációk is frissülnek. A későbbiekben még említésre kerül. Visszatérve tehát hozzá, a „loot" nagyon fontos része az RPG-knek. Amikor elpusztítunk

ellenségeket esély van arra, hogy az „dobjon” tárgyakat. Már említettem, hogy az ellenségek élet interakciójában lehetséges a fejlesztőnek megadni a Unity felületén keresztül egy listában a lehetséges zsákmányokat. Itt be tudjuk mindegyiknél állítani, hogy mi legyen rá az esély, illetve a lehetséges mennyiségi intervallumot. A halál lekezelésénél ezek és a véletlenszám generátor segítségével aztán feltöltődhetnek a már említett módon a zsákmány zsákok az adott pozíción. A játékosnak ezután az interakciós bevitel segítségével lehetősége van a zsákmány képernyő megnyitására. Innen aztán eldöntheti, hogy mit visz magával. A dinamikus zsákmányok szintváltás után eltűnnek, a statikusok nem, erre még a perzisztenciánál térek majd vissza. Amennyiben a játékos enter-t nyom egy zsákmányra, és abból több van jelen, mint egy, akkor a már említett beviteli képernyőre megyünk át. A különböző tárgy csoportok aztán a tétel menedzser különböző funkcióival érik el a megfelelő folyamatok lezajlását. Magát a zsákmány GO példányosítását és annak tartalmának kezelését is egyébként ezen keresztül lehet könnyen lefolytatni.

5.14. Párbeszéd

A célzó rendszer segítségével lehetőségünk van egy NPC „DialogInteraction” komponensével kölcsönhatásba kerülnünk. A dialógusok központi magja pedig a „DialogNode” osztály lesz. Ezek segítségével lehet aztán többek között felépíteni ezeket a beszélgetéseket és persze meg is jeleníteni azokat. A második pályán tudjuk ezt a rendszert működés közben megfigyelni, a Cain karakter esetében. Amikor elkezdünk egy ilyen interakciót, felnyitjuk a párbeszéd képernyőt. Ezt nagyjából a képernyő alsó harmadában láthatjuk. Fentről lefelé haladva áll egy megnevezésből (ki beszél éppen), utána pedig vagy egy darab - szöveget tartalmazó - gombból, vagy pedig egy szintén szöveges gomb listából. Mindegyik elem reszponzív: a titulus automatikusan gördülhet, a lista szükség esetén görgethető, a nem-listás gomb pedig „több oldalassá” változik, ha indokolt. Elkezdjük a beszélgetést, ez először mindig az egy-gombos variánssal kezdődik. Ilyenkor az enter-rel tudjuk előre mozgatni az eszmecserét, vagy esetleg a monológot. Egy ponton a képernyő átválthat a listás variánsra, ilyenkor választhatunk. Amennyiben aztán választunk, értelemszerűen az eddigiek további kombinációi várhatnak ránk. Természetesen mindig el lehet aztán jutni egy olyan választási lehetőségig, ahol is elhagyhatjuk a beszélgetést. A dialógus maga több módon is változhat a játékos döntései alapján. Először is a kilépésre visszatérve: a rendszer figyeli, hogy milyen csomókat érintettünk. Ennek alapján elmentődhetnek olyan pontok, amikről a kilépés esetén a beszélgetést újra el kell kezdeni, hiszen ugye nem akarjuk mindig mindenképpen ugyanazokat a beszélgetéseket lefolytatni. Ez persze egy interakció során mindig csak egy érték lehet. Egy másik fontos tulajdonság, hogy vannak olyan választási opciók, amiket, ha már egyszer aktiváltunk, többet nem lesznek elérhetőek. Ezek ugye az egyedibb, nem visszatérő beszélgetések. Ezzel ellentétben vannak olyan lehetőségek, amik először nem elérhetőek és nem láthatóak, azonban bizonyos feltételek teljesülése

esetén használhatóvá válhatnak. Ez persze az eseménykezelés segítségével történik. Az azt hiszem érezhető, hogy minden előbb említett képernyő állapot valójában egy-egy csomópont reprezentációja. A mellékletek 11.6. pontjában talál képernyőképes példát is az olvasó.

5.14.1. DialogNode

Ennek az osztálynak több funkcióját is érintettem már az előzőekben. A csomópont elnevezés egyébiránt az osztály a nevében egyáltalán nem véletlen. Az egész rendszer ugyanis gráf alapú, ezekből a csomókból áll, az élek pedig mint referenciák vanna jelen. Minden csomó rendelkezik egy „Next" elnevezésű DialogNode referencia tömbbel. Ez alapján dönti el a rendszer a következőket: csak „tovább nyomható" dialógus csomón állunk, választhatunk-e folytatási pontot, vége van-e a dialógusnak? Megkönnyítve ennek a tömbnek a különböző kontextusokban való lekérdezését indexelhetővé is tettem egyébként ezt az osztályt. A könnyebb dialógus létrehozás végett elláttam még az osztályt különböző célú, statikus építő metódusokkal is. Mielőtt egy dialógus gráf felépítettnek tekintetett, szükséges lefuttatni a statikus „FinalizeNode" metódust. Ez a metódus rekurzívan végigmegy az összes csomón attól a ponttól kezdve, ahol meghívják. Emiatt - mint számos más funkció esetében is - érdemes ezt a gyökér csomón meghívni. Minden csomónál beállít egy egyre növekedő szám azonosítót, amit referencia szerint kezel az eljárás. Ezen kívül még fel is iratkozik a „SubscribeTryToShow" akció segítségével, ha az erre bevezetett zászló ezt indokolja. Természetesen ezt az akciót minden csomó maga kell definiálja, amikor már része egy párbeszéd struktúrának. Az utolsó érdekesség a „SearchWithin" statikus függvény. Ez gyakorlatilag egy rekurzív, majdnem logaritmikus keresés, amit pont azért tudunk megvalósítani, mert az előző segítségével már legyártottuk egy mélységi bejárás alapján az ID-kat. Ebből kifolyólag a keresés ki tudja használni azt, hogy a csomók azonosítója alapján merre érdemes tovább keresni, hiszen az egy szinten lévő azonosítók mentén meg lehet mondani, hogy mélyebben mely értékek találhatóak még meg. Ez a későbbiekben még nagyon hasznos lesz. Jelenleg a „CainDialogEnglish" az egyetlen dialógus gráf, de persze ennek alapján lehetne létrehozni még másakat is. Mivel ez azért egy jóval bonyolultabb struktúra, mint a sima szövegek felépítése, ezért az ilyenek külön osztályokat kapnak. Nagyrészt igazából az említett építő metódusokat használjuk itt fel az adott NPC kommunikációs gráfja legyártásához. A lokalizációval kapcsolatban már egy korábbi fejezetben leírtam a szükségeseket, de azért megemlítem, hogy itt is beépíttetett a viszonylag jól kezelhető lehetőség a többnyelvűség támogatására. Az építőkön kívül implementáltam még számos más Node metódust azért, hogy itt könnyebben lehessen ezeket összefűzni. A végeredmény a lokalizációs menedzserben kerül eltárolásra.

5.14.2. DialogInteraction

Az emlegetett MB teszi tehát lehetővé azt, hogy játékmeneti szinten mindezt viszont megtapasztalhassuk a Unity segítségével. Számos menedzsert kér számon, és az egyedi enum-mal pedig be kell állítanunk, hogy melyik NPC dialógusát hívja majd be. Ez az osztály tartalmazza majd a megjelenítéshez is szükséges referenciákat is. Ezeken keresztül szabályozódik, hogy melyik nézet van érvényben, mi legyen a tartalom stb. A lista itt is egy VSU. A dialógus modelljével kapcsolatban három értéket tárol el: a gyökér csomót, a jelenlegit és a visszatérhető azonosítóját. Ez alapján fogja kikeresni azt a csomópontot a rendszer, ahonnan a legutóbbi beszélgetés óta a következő megnyitásánál ki kell indulnia. Ez például egy nem látható választási útvonal jelenleg, amin csak az első beszélgetés utáni dialógus kezdetek esetében megyünk végig.

5.15. Kereskedés

A kereskedés témaköre is kötődik az előzőhöz, ugyanis ezt a funkciót csak a párbeszéd-rendszeren keresztül érhetjük el. Az egyes csomókon be lehet állítani egy bool értékkel, hogy kereskedelmi node-ról van-e szó. Amennyiben igen, a dialógus interakció felhívja a UIControlManager-t, aki megkísérli a becélzott GO-ról lehívni a „TradingInteraction” komponenst. Ezek után már ez végül is egy unalmasnak tekinthető komponens. Az egyetlen, amit kiemelnék, hogy itt is a Unity szerkesztőjén keresztül tudjuk az osztály implementálásának hála megadni a kereskedő kezdeti áru halmazát. A struktúra maga egy szótár, ahol az egyedi azonosító enum-ok a kulcsok, majd ehhez rendelünk egy szám értéket. Amennyiben egy mennyiség a negatív egyesre állítatott, a tárgyat végtelenszer megvehetőnek veszi a rendszer. A betöltést magát az ItemGeneralManager-en keresztül végzi, mivel annak már a rendelkezésére is áll az össze tárgy jelenleg érvényes GO/MB reprezentációja a jelenetben. A „TradingScreen” egy két darab oszlopból álló UI, a már megszokott funkciókkal rendelkezik. A játékos alatt külön megtalálható kettő kereskedelmi gyökér-GO, amelyek alá mind a játékos és a kereskedő aktuális objektumai is legeneráltatnak. Ezt a generálást a menedzser a jelenetváltás felszólítására minden betöltésnél elvégzi. Az a megközelítés, hogy ezeket a GO-kat függetlenül kezeljük a játékos tényleges tárgyaitól. több szempontból is segít. Nem utolsó sorban azért is előnyös, hogy bárhogyan tudjuk manipulálni őket. A képernyő megnyitásánál viszont már csak annyit kell tennie a rendszernek, hogy végig megy mind a kettő oldal tárgyain, és aktualizálja a mennyiségeket. A tényleges adás-vétel pedig a már emlegetett UI rendszerek felhasználásával könnyen elképzelhető. A tárgyak ilyenkor megkapják a leírásukban a mennyiséget, az árat, illetve látjuk a játékos vagyonát is.

5.16. Küldetések

A küldetés rendszert az RPG-kben „questing” szokás hívni. Az általános felépítés a következő szokott lenni, van egy fő küldetés-sorozat, amit az igazi előrehaladás

érdekében előbb-utóbb muszáj elvégezni. Ezen kívül pedig létezhetnek még mellék küldetések. Mindezt pedig sokszor egy fajta küldetés naplóban tudjuk számontartani. Így működik ez az egész nagyjából ebben a játékban is. Kezdjük először a modell első alappillérével, a „Quest" osztályával. Két azonosítót is tartalmaz: magát a feladatét és az opcionális feladat-láncét. A feladat lánc csak a lekezelés bizonyos részeinek jelzi, hogy az adott quest-ek összefüggenek, vagy nem. Ezeken kívül tartalmaz egy referenciát egy „QuestStage" és a következő küldetésre. A megjelenítés irányába pedig kiajánl két nyelv kulcsú szótárt, a cím és a leírás tárolóit. Jelen van még egy privát konstruktor is, őt egy szintén jelenlévő statikus, építő metódus használja majd fel kifelé. Ezeken kívül már csak pár később felhasznált, építő jellegű segítő metódus van jelen. A QuestStage osztály a valódi feladat tartalmazó. Egy feladatnak több fokozata is lehet. A fokozatokhoz is mindig társul egy egyedi azonosító. Tartalmaz még egy opcionális mutatót a következőre, és a szülő küldetés kötelező referenciáját. Egy fontos metódus referencia trió a következő: „CanBeCompleted", „TryToComplete", „OnReward". Azt hiszem ezek elég egyértelműek: nagyrészt kívülről megadható logikákról van szó, feladat függőek. Megadható még kettő feliratkozó akció is, a TryToComplete fel- leiratkozása. Az egész úgy építtetett meg, hogy a különböző építő és segítő metódusokkal egyetemben az osztály elég általános, viszont a tényleges feladat összeszerelésnél kívülről nagyon sok fajta quest logikát találhassunk ki. A feladat fokozatoknak is van leírása egyébként. Tartalmaz még az osztály egy számláló tömböt is, aminek a segítségével dinamikusan számolni is lehet a feltétel teljesüléseket. A feladat figyelésre még visszatérve: a TryToComplete privát, a másik kettő kívülről jön. A konstruktorban építjük össze a TryToComplete-et úgy, hogy abba bármilyen paraméter beérkezhet, benne pedig csak egy elágazást találunk: ha CanBeCompleted, akkor kiváltjuk az eseménykezelő „feladat fokozat befejeződött" eseményét. Ezt persze a megfelelő helyeken figyelni kell, majd aktiválni a jutalmazó akciót. A fentebb említett kettő, fel- és leiratkozókat pedig az építő metódus állítja össze. Ezek az esemény kezelések itt első körben csak eltárolódnak, majd aztán más rendszerek kezelik ezeket.

5.16.1. QuestBuilderContainer

Hasonlóan a dialógusokhoz, a feladat felépítések is kaptak saját osztályokat, ahol a különböző metódusokon keresztül, sokszor egymáshoz csatolva tudjuk kialakítani a feladat láncainkat. Erre példák a „MainQuestChain" és a „CainSideQuest0Chain" statikus osztályaim. A küldetés sorozat építőimet egyébként egy statikus, konténer osztályban, a QuestBuilderContainer-ben tartom számon egy szótárban azért, hogy az egyedi azonosítók segítségével egy helyről lehessen ezeket elérni, így létrehozva egy eltakaró, absztrakciós réteget az építő osztályok fölé.

5.16.2. QuestGeneralManager

Az „általános küldetés menedzser” fogja természetesen össze ezt az egész rendszert, illetve ő teszi lehetővé a többi rendszer felé az információk elérését is, például a UI-nak. Ő egy szótárban tárolja a jelenleg aktív quest-ek referenciáját, a kulcs az azonosító. Mivel a küldetés rendszer is összeláncolt struktúrákból áll, ezért megtehetjük, hogy mindig csak az adott pontokat „fogjuk meg” egy-egy referencia segítségével, a gráfok attól még a memóriában jelen vannak. Az említésre méltó inicializálást itt is az eseménykezelő hívja fel, ugyanis rendkívül erősen támaszkodunk itt őrá. A menedzser számára legfontosabb eseményekre egyből fel is iratkozunk: feladat kezdet és vég, fokozat kezdet és vég, nyelvváltozás. A négy feladat esemény figyelésével ez a menedzser kezeli tehát a referenciák tovább léptetését, az új feliratkozások megtételét. a jutalmazások felhívását és persze a határesetek lekezelését. Ahogy említettem a naplót is ez az SO kezeli. A UI baloldalt egy feladat listából, jobb oldalt pedig egy felső nagyobb és egy alsó kisebb szövegdobozból áll. Itt van arra lehetőségünk, hogy több információt szerezzünk ezekről. A még meg nem kezdett és a már befejezett küldetések nem láthatók.

5.17. Perzisztencia

Végül is úgy döntöttem, hogy a kiterjedés kezelése miatt egyelőre csak egy mentési fájlt fogok támogatni, a gyorsmentésit. Először is szó kell essen a fő menüről. UI-ilag kezdhetünk itt új játékot, betölthetünk egy meglévőt és ki is léphetünk. A meglévő folytatása miatt szükséges volt egy apró logikát is implementálnom, amivel is eldönttetik, hogy látszódhat-e a folytatás gomb vagy nem. Kevés gyökér UI szintű MB létezik egyébként a projektben, pont azért, mert a legtöbb mindent általánosan és dinamikusan sikerült megoldanom. Ez azonban egy kivétel. A gomb csak akkor látszódhat, ha a nulladik jelenetben tartózkodunk és létezik már mentési fájl. Mindezt a Start-ban végezzük el. Erre azért is volt szükség, hogy játékmenet közben is fel tudjam használni ezt a menüt aztán. A fő menü említésével ugyanis alapértelmezetten magáról az előkészítő szerepű, nulladik jelenetről is beszélek, nem csak a felületről. Sok rendszer kihasználja ugyanis azt, hogy először mindig ebbe tölt be a játék.

5.17.1. PersistenceGeneralManager

Visszatérve viszont a „tartósítás fenntartó” rendszerünkre, természetesen ez is kapott egy menedzsert. Rengeteg másik SO referenciájára is szükség van itt, azonban pár más dolgot is meg kell adni a Unity tervezőjében. Ilyen a nulladik, és az első játékmeneti jelenetek információja, illetve a kezdési pozíció. Természetesen megtaláljuk itt is a szokásos inicializációkat. Az OnEnable-ben hozzuk létre többek között a formázó objektumot, és itt állítjuk be a mentési útvonalat is - ezt egyébként nagyrészt a Unity biztosítja. Megtalálható metódusok még az új vagy folyamatban lévő játék elkezdése, a fő menübe való visszatérés és a kilépés kezdeti pontja. Ezeket hívja fel a felhasználói

felület. Termesztésen itt van a mentési és betöltési logika is. A folytatásra röviden még visszatérve, felkészültem a jövőbeni több mentési fájl jelenlétére is. Emiatt a folytatás mindig átnézi a beállított könyvtárat, és a legfrissebb fájlt adja vissza a metódus. Az egész mentési fájl alapja a gyökér perzisztencia konténer és az abban lévő alkonténerek. Ezt az objektumot mentem le és töltöm be. Rengetek SO-ban szerepelnek ezek a különböző tárolók, mentésnél ezeket gyűjtöm össze, és csomagolom bele a gyökér objektumba, majd szerializálom. Deszerializálásnál pedig persze ennek a fordítottját teszem. Amikor a menedzserrel új játékot kezdünk, első lépés gyanánt már igazából itt is a fentebb említett konténer struktúrát állítjuk össze. Pár példa a gyökér konténer, tehát a mentési fájl tartalmára: jelenet útvonal, pozíció; statisztika, tárgy, ellenség, feladat konténerek, és így tovább. Az új játéknál ezek legtöbbje üres vagy pedig valamilyen iniciális értéket tartalmaz. Meglévőnél persze ezeket mind az adott menedzserektől kérjük el. Ezekben általában tulajdonságok formájában figyeltetik, hogy épp történik-e mentés vagy betöltés. Amennyiben pedig igen, sokszor ilyenkor állítják össze vagy frissítik az adott konténereket. Amennyiben tehát összeállt egy ilyen konténer csomag (vagy betöltés, vagy pedig új játék megkezdésével), lementjük azt ebben a menedzserben, majd pedig felhívjuk a jelenetkezelő SO betöltési metódusát. Ennek csak a jelenet referenciáját és a kezdési pozíciót adjuk meg. Ezután az elvégzi a jelenetváltást, majd pedig a betöltés után (a játékos ezt még mindig mint „töltés” éli meg) visszaszól a perzisztencia menedzsernek, hogy érvényesítheti a konténerét. Ilyenkor történik meg a már emlegetett tulajdonságokon keresztüli, tényleges információ átadás. Amint ez sikeres, valójában csak ezután szól a jelenetváltó a többi rendszernek, hogy elkezdhetik a jelenetre történő felkészülést. Ezen a ponton már mindenkinél ott lesz az új vagy perzisztált adat. A játék persze egy bool zászlóval figyeli azt egyébként, hogy a jelenetváltás betöltés miatt történt-e, vagy már a játékmenet része.

5.17.2. Konténerek

Ahogy megígértem tehát, térjünk a következőkben ki pár érdekesebb alkonténer működésre, különösebb sorrend nélkül. A legtöbb csak különböző objektumokat tárol, írásuk, olvasásuk szokványos, a konstruktoraikban pedig általában az új játék megkezdéséhez szükséges beállítások szerepelnek. Ilyenre példa a statisztika konténer is. Az adott jelenetekben lévő és különböző állapotú ellenfelek tárolása sem túl bonyolult. A zsákmány konténerben el kell tárolnunk a statikus és a dinamikus zsákmányokat is. A statikusokban igazából jelenet azonosítónként kell eltárolnunk a pozíció párok alapján a tárgy azonosító-mennyiség párokat. Ritka eset ez egy háromszoros szótárra a projektben. Fontos még kiemelni, hogy itt egyébként azt kell eltárolnunk, hogy melyikből, mit és mennyit vettünk ki! Ezek ugyanis mint kezdeti GO csoport mindig jelen vannak a jelenetekben és el kell ilyenkor döntenie a rendszernek, hogy kivesz-e belőlük tárgyakat, vagy pedig teljesen meg is semmisíti-e a zsákokat. Dinamikusnak számít az, amikor egy ellenség a halála után dob valamiket. Ezeket csak akkor jegyezzük meg, ha még az adott

jelenetben tartózkodunk. A zsákmány mentés és visszaállítás mag-logikája egyébként az item menedzserben található, és nem teljesen triviális. Vannak olyan perzisztens alrendszerek, amik nem is csak egy, hanem akár kettő részből is állnak a bonyolultságuk miatt. Erre példa a dialógusok ilyen nemű kezelése. A dialógus gráfokat is részben le kell mentenünk, gondoljunk csak vissza a különböző dinamikusan változó állapotokra. Ezek az osztályok viszont olyan bonyolultságúak, hogy szükséges volt egy külön, leegyszerűsített, perzisztens dialógus csomó reprezentációs osztály létrehozására („DialogNodePersistenceContainer"). A bonyolultság egyébként elsősorban itt az eseménykezelés általi átszőttséget jelentette. Ennek felépítésére létrehoztam a csomóban egy metódust, amit a szokásos módon a gyökéren érdemes felhívni („BuildPersistenceGraph"), persze ennek az ellentettjére is építenem kellett egy hasonszőrű alrendszert („ExtractPersistenceGraph") a csomóban. Ezt az adat csoportot egyébként az EntityGeneralManager kezeli. Az utolsó említésre méltó a feladatkezelés perzisztálása. A dialógushoz hasonlóan ez is két részből áll: a feladat és a fokozatéból. A már említett referencia alapúság miatt valójában csak egy azonosító-fokozat párokat tartalmazó szótárt tartunk számon, azonban a küldetés konténerben elvégzünk pár konverziót. A mentésnél a fokozatokba már beépített konténereket olvassuk ki. A betöltésnél azonban újra fel kell építeni a szükséges feladat-láncokat, majd meg kell keresni bennük azt az állapotot, ahol a játékos tartott. Amennyiben ez sikerült, igazából valójában már csak a számlálókat kell visszaírni, minden más már felhasználható. Az utolsó lépés csak az, hogy az aktuális küldetés fokozatok kapcsán újra ki kell váltanunk a „küldetés elkezdődött" eseményt. Ennek alapján fogja a többi rendszer is beállítani magát.

6. TESZTELÉS

6.1. Teszterv

Egy videójáték tesztelése talán elmondható, hogy még nehézkesebb feladat, mint az általánosabb felhasználású szoftverek esetében. A következőkben a projektemről eme téma szemszögéből fogok értekezni. Alapvetően abból indultam ki a tervezési fázisban, hogy a szokásos munkafolyamatokhoz hűen elsődlegesen az egység és a rendszertesztek vonalán fogok mozogni. Ennek megfelelően emiatt is igyekeztem minél modulárisabb és jól felosztott architektúrát létrehozni. A teszterv tehát tudatosan azt jelentette, hogy nagy mértékben szükséges lesz majd a folyamatos manuális tesztelésre, de persze az egységtesztek is megjelennek az egyszerűbb részeknek megfelelően.

6.2. Egységtesztelés

A következőkben a kisebb egységek tesztelés által finomított működését vizsgáltam, illetve azokat az eseteket, amelyek szélsőségesnek vagy kritikusnak mondhatók. Azt hiszem látszódik a megvalósításból, hogy egy ilyen projektre automatizált (jelen esetben unit) teszteket írni rendkívül nehéz. Már csak azért is, mert a kód nagyon nagy része nem futtatható önmagában: a legtöbb metódusnak különböző GO, MB és SO-s elvárásai vannak. Léteznek erre egyébként megoldások, meg is kíséreltem kipróbálni, de rendkívül kiforratlan és kérdésesen használható.

Bizonyos részekre azonban tudtam egység teszteket írni. A Unity ilyenkor azt várja el, hogy a projekt fejlesztői szemmel kiinduló könyvtárában (Assets) lévő, kitüntetett szerepű „Editor” mappában legyenek a teszt osztályok. Ezeknek a felépítése már a megszokott C#-osnak, tehát „NUnit” alapú. Ezután pedig a Unity szerkesztőben lévő megfelelő helyen ezek megjelennek, és futtathatjuk, kiértékelhetjük a tesztjeinket.

Egyrészt készítettem két teszt osztályt kettő darab, sokat használt, ámbátor nem túl bonyolult kalkulátor segítő osztályomhoz: a százalék és harc kalkulátorokhoz. Odafigyeltem arra, hogy a konvenciókhoz tartsam magam. Egyrészt a metódus neveket úgy alakítottam ki, hogy pontosan tudja az, aki rájuk néz, hogy mit tesztelünk és hogy mi az elvárás. Másrészt tartottam magam az AAA, vagyis az „Arrange Act Assert” teszttervezési mintához.

6.2.1. PathfindingTester

Külön említem azért az útkeresés ellenőrzőmet. Itt már szükséges egy „OneTimeSetup” jelenléte is. Szükséges ugyanis, hogy a gráfot például megfelelően

kialakítsam. Itt megpróbáltam egy olyan, nem túl nagy, tízszer tízes pályát létre hozni, aminek a kialakítása megfelelő komplexitású az akadályokat tekintve.

Ezekben a tesztekben folyamatosan variálom a kezdeti és a végcélt is. A helyességet egyébként úgy vizsgálom, hogy az út hosszát értékelem ki. Ezt azért tehetem meg, mert tudom, hogy az algoritmusom determinisztikus. Több esetben egyébként azt várom, hogy a rendszer kivételt dobjon, hiszen a helyes kivételkezelés is fontos része bármilyen szoftvernek. Erre itt a példa az, amikor kiindexelünk a tömbből.

6.3. Rendszertesztelés

A különböző rendszerek, amelyek alapján a szoftvert felépítettem (például célzás, interakciók mozgatás, UI-ok, dialógus stb.) egy összetett komponenskapcsolatot alkotnak. Ezeknek a rendszereknek a leellenőrzését a videójátékiparban nagyrészt külön szakosodott tesztelők szokták végezni manuálisan, a kiszabott részleteket újra és újra kipróbálgatva, mindezt a program fejlesztésének iterációit lekövetve. Jómagam is alkalmaztam ezt az eljárást.

Amennyiben változtattam bármin is, ami mondjuk a játékoskarakter helyváltoztatásával kapcsolatos, a játékot elindítva végig kell próbálgatnom a lehetséges különböző alrendszereket, amelyek kommunikálni képesek az adott funkcióval.

Ilyen eset volt például az, amikor a célkeresztet implementáltam. Ez ugye egy kollíziós területet irányít valójában, amely az alatta lévő elemeket kérdezi le. Azonban egy időben ez a célkereszt képes volt a pályaváltó trigger-rel is kapcsolatba lépni, ami persze nem kívánt jelenség volt. Nem értettem, hogy ez miért történik, a pályaváltó ugyanis lekérdezte az interaktáló játékobjektum címkéjét is természetesen, pontosan az ilyen és ehhez hasonló esetek elkerülése végett. Az célkereszt gyermeke volt a játékos objektumtársulásnak, amely tetején a fő játékosobjektum valóban rendelkezett a „Player” tag-gel, az érzékelő rész viszont jelöletlen volt. Én pedig a trigger-ben a kollíziót kapom meg, amely a helyes objektumhoz volt hozzákötve. Hosszas próbálgatás, debug-golás és utána olvasás után pedig kiderült, hogy a „collision.CompareTag()” nevű függvény a szülőobjektummal kezdi a vizsgálatot, míg a „collision.tag” az adott collízióhoz tartozó közvetlen játékobjektum vizsgálatával kezdi a lekérdezést. Az utóbbit kellett tehát használnom.

A későbbi, sokkal bonyolultabb rendszereknél rengetegszer végigjátszottam különböző módokon a játékot. Erre példa a küldetésrendszer-dialógus-pályaváltás-perzisztencia rendszerek összefüggése, mindez átfűzve az eseménykezeléssel. Az ilyen és ehhez hasonló kapcsolatokat szerintem csak minőségi kódra való törekedéssel, a határesetek észben tartásával és azok folyamatos ellenőrzésével lehet tesztelni.

7. TOVÁBBFEJLESZTÉS

Rengeteg pont van további hanghatások bevezetésére, illetve a meglévőek frissítésére. Sokszor például a modern játékokban nagyra értékelem azt, amilyen hangokat a UI kezelése kiad. Ez nagyon nagyban tudja növelni a különböző menedzsmentek (pl.: tárgy) élvezetét. Animációk terén is akad még bőven teendő: több és jobb bevezetése, esetleg a fegyverekkel való bánás mozgatása stb. Bizonyos csempék is mozoghatnának; ha lenne például víz a játékban, már most meg tudnám oldani valószínűleg, hogy váltakozzanak ezek a szprájtok. Egy másik, RPG-kben megszokott tulajdonság még a dialógus szövegek fokozatos animálása, és apró hangokkal való kisérése. Kicsit technikaibb jellegű, de talán idesorolható még az árnyékok generálása.

A technikalitások halmazába sorolom az egyből talán kevésbé észrevehető implementációs ötleteket. Pár helyen még javítható lenne a különböző UI kiválasztások megjegyzése, lekezelése. A dialógusok ID rendszerének hála be lehetne vezetni egy fajta validációt is. A bevitel kezelés esetleg át lehetne állítani esemény szintűre. Lehetne használni lokalizációs csomagokat, amelyek segítségével talán komolyabban lehetne ezt a részt kiépíteni. Ehhez kapcsolódóan valószínűleg szebb lenne a szövegeket valamilyen fájlrendszerbe rendezni. A hangrendszert át kéne szervezni úgy, hogy tényleg az adott GO-khoz kapcsolódjanak a források. Pár helyen valószínűleg jól lehetne alkalmazni a már beépített vizuális szkriptelést. Lehet megérné egy fajta absztrakt MB elkészítése, ami optimálisabb módon kezeli a kikapcsolást és a pusztulást, esetleg összevonásokkal. Felmerülhet, hogy optimálisabb lenne még pár rendszert párhuzamosítani. A legutolsó már talán a következő pontba is tartozhatna, de a jelentősebb technikai kihívása miatt még ide soroltam: egy jóval összetettebb és okosabb NPC viselkedési rendszer implementálása. Itt ilyenekre lehet gondolni, mint a követés feladása, visszatérés az eredeti pozícióba, őrjáratozás stb.

A funkciók szempontjából megtekinthetjük ténylegesen új játékmeneti elemeket és az egyértelműen nagyobb feladatokat. Az első az, hogy lenne egy rendszer. amivel a képesség menü megnyitása nélkül lehetne a kiválasztottat megváltoztatni. Modell szinten ez még hagyján, de szükséges lenne ehhez valamilyen UI reprezentáció is. Lehet kellemes lenne a kamerát is mozgatni tudni. Egy ponton túl mindenképpen jelen kéne legyen egy olyan funkció, amivel UI elemeknek a részleteit lehetne megtekinteni, itt például a tárgyak finomabb leírásának megtekintésére gondolok. Ahogy már említettem, ki lehetne bővíteni a perzisztencia rendszert is egy több mentési fájlosra. Lehetnének profilok statisztikákkal, mint még egy absztrakciós szint a mentések fölé. Be lehetne vezetni nehézségi fokozatokat is. Része lehetne a játéknak a játékos mellett harcoló NPC-k is. Abszolút megoldható lenne a különböző beviteli rendszerek támogatása is. Végül, de nem utolsó sorban pedig meg lehetne oldani, hogy bizonyos rendszerek fájl szinten hozzáférhetők legyenek a játékot módosítani akarók számára.

8. ÖSSZEFOGLALÁS

Belekezdeni, ráadásul véghez vinni ezt az ötletet, nagyon nagy lépés volt számomra. Megtapasztalni ezeknek a rendszereknek a sajátkezű tervezését és kiépítését, majd iterálni azokon és a tanulságokat levonni, ezek mind nagyon sok tapasztalathoz juttattak hozzá. Az irodalomkutatásban végig gondolhattam azt a sokrétűséget, amit egy videójáték jelent, főleg egy RPG. Számos szempontból utána kellett járnom annak, hogy régen, hogy néztek ki az adott rendszerek, jelenleg mi a használatos, és meg kellett azt is fontolnom, hogy a jövő szempontjából mit érdemes közelebbről megtekinteni. Megismertem olyan trendeket, amelyekről még addig nem igazán hallottam, a DOP-t.

A specifikációban meg kellett fogalmaznom azokat a fontosabb pontokat, amelyeket az irodalomkutatás kapcsán beazonosítottam, amiken keresztül később a tervezésben összetudtam fűzni az elméletet a gyakorlattal. A tervezés persze már egy megszokottabb folyamat volt. Itt az új és a már meglévő információk segítségével le tudtam fektetni azokat az alapokat, amelyek alapján aztán az implementációt fel tudtam építeni. Külön említésre méltónak találom itt az „Architektúra” alfejezetet, ahol is össze tudtam foglalni egy speciális technikai alapot, ami nélkül nem lettek volna meg a kiindulási pontjaim az ezt követő fejezetben.

Természetesen a megvalósítás maga azért mindig a legizgalmasabb rész egy szoftver mérnöknek. Mint említettem, nagyon sok mindenről lehetett volna még szót ejteni, emiatt ennek a fejezetnek a strukturálása volt messze a legnehezebb számomra. A végeredménnyel azonban, azt kell hogy írjam, nagyrészt elégedett vagyok a korlátokhoz képest. Megfelelően sok rendszer működését tudtam érinteni kisebb-nagyobb részletességgel. Nem bírom hangsúlyozni, hogy milyen sok réteggel jár egy ilyen alkalmazás. Ebbe persze bele tartoznak talán művészibb vetületek is, de azért a súlypont a programozáson maradt. Az egyik legszórakoztatóbb rész egyébként az útkeresés volt nekem, az egész projekt implementálásában pedig azt volt érdekes megfigyelni, ahogy az egyre összetettebb rendszerek valóban „automatikusan” megmutatkoztak aztán a játékélményen is. A legnagyobb kihívást egyébként talán a UI vetületek kialakítása, illetve a határesetek lekezelése jelentette. Szakmailag a dialógus rendszer a legösszetettebb talán, de van még bőven mit említeni.

A manuális tesztelés, mint már említettem, folyamatos volt. Pár összefüggésre így is jöttem rá. Az egységtesztelést őszintén szólva nem tartom rendkívül sokra, de hasznos tud azért lenni. Az útkeresésnél nagyon jó volt látni, hogy milyen szépen szeparáltak a rendszerek. A továbbfejlesztési lehetőségeket tudatosan az egész projekt alatt gyűjtöttem, ezért is van belőlük ennyi. A feladat egészére azt kell hogy mondjam, büszke vagyok azokra az alapokra, amik ezen a ponton a játékot képezik. Nem lennék szégyenlős másoknak is mutatni belőle.

9. SUMMARY

It was a very big step for me to get started with, and to implement this idea. Experiencing the designing and construction of these systems on your own and then iterating and drawing lessons from them have all contributed to a great deal of experience. Throughout the literature search, I could think about the versatility that a video game means, especially an RPG. In many ways, I had to find out what the systems used to look like in the past, and I also had to consider what to look at for the future. I have also leamed about trends I have not really heard of before, like DOP.

In the specification, I had to articulate the main points I identified in connection with the literature search, through which I was later able to connect the theory with practice in the design. Of course, planning was already a more common process to me. Here, with the help of new and existing information, I was able to lay the foundations upon the basis of which I could then later build the implementation. I find the sub-chapter 'Architecture' worth mentioning here, where I was able to summarize a special technical basis, without which I would not have had my starting points in the following chapter.

Of course, the implementation itself is always the most exciting part for a software engineer. As I mentioned, there was a lot more to be said, which is why structuring this chapter was by far the most difficult for me. However, I must note that with the result, I am largely satisfied with, considering the limitations. I was able to touch on the operation of many systems in varying detail. I cannot stress the fact that just how many layers such an application entails. Of course, this also includes some artistic sides, but the focus has remained on programming. One of the most fun parts was the pathfinding, and in implementing the whole project, it was interesting to see how the increasingly complex systems really showed up 'automatically' in the gaming experience. By the way, the biggest challenge was to create the UI systems and to handle borderline cases. Professionally, the dialogue system is perhaps the most complex one, but there is still plenty to mention.

Manual testing, as I mentioned, was continuous. I even uncovered a few useful connections through this process. Honestly, I do not think unit testing is enormously useful, but it can be helpful, nonetheless. When writing tests for the pathfinding, it was great to see how nicely the systems were separated. I have consciously collected opportunities for further development throughout the project, which is why there are so many of them. For the whole task, I have to say I am proud of the foundations that make up the game at this point. I would not be shy to show it to others.

10. IRODALOMJEGYZÉK

[1] Ward, Jeff: What is a Game Engine? *Game Career Guide*, 2008, (http://www.gamecareerguide.com/features/529/what_is_a_game_.php?print=1), utoljára megtekintve: 2018.12.05.

[2] Hauser, Dietmar: License an engine or create your own? *Gamasutra*, 2015, (https://www.gamasutra.com/blogs/DietmarHauser/20151007/255394/License_an_engine_or_create_your_own.php), utoljára megtekintve: 2018.12.05.

[3] Microsoft: Graphics APIs in Windows, (https://docs.microsoft.com/en-us/windows/desktop/direct3darticles/graphics-apis-in-windows-vista), utoljára megtekintve: 2018.11.27.

[4] Apple: OpenGL Programming Guide for Mac, (https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_pg_concepts/opengl_pg_concepts.html#//apple_ref/doc/uid/TP40001987-CH208-SW1), utoljára megtekintve: 2018.11.27.

[5] Nelson Jr., Xalavier: Why porting games to PC is hard. *PC Gamer*, 2017, (https://www.pcgamer.com/why-porting-games-to-pc-is-hard/), utoljára megtekintve: 2018.12.05.

[6] Instabug Blog: Top Game Engines In 2018. 2017, (https://instabug.com/blog/game-engines/), utoljára megtekintve: 2018.12.11.

[7] Sutorcen: This is Unreal. *Hotgates*, 2016, (http://hotgates.eu/this-is-unreal/), utoljára megtekintve: 2018.12.11.

[8] Epic Games: What is Unreal Engine 4? (https://www.unrealengine.com/en-US/what-is-unreal-engine-4), utoljára megtekintve: 2018.12.11.

[9] Murphy, Kevin: Unity Game Engine Review. *GameSparks*, 2016+, (https://www.gamesparks.com/blog/unity-game-engine-review/), utoljára megtekintve: 2018.12.11.

[10] Unity Technologies: The world's leading real-time engine. (https://unity3d.com/unity), utoljára megtekintve: 2018.12.11.

[11] Rock Milk: Why we choose Godot Engine. *Medium*, 2017, (https://medium.com/rock-milk/why-godot-engine-e0d4736d6eb0), utoljára megtekintve: 2018.12.11.

[12] Godot: Features. (https://godotengine.org/features), utoljára megtekintve: 2018.12.11.

[13] Shreier, Jason: Why Video Games Cost So Much To Make. *Kotaku*, 2017, (https://kotaku.com/why-video-games-cost-so-much-to-make-1818508211), utoljára megtekintve: 2018.11.26.

[14] T.C.: Why video games are so expensive to develop. *The Economist*, 2014, (https://www.economist.com/the-economist-explains/2014/09/24/why-video-games-are-so-expensive-to-develop), utoljára megtekintve: 2018.11.26.

[15] OpenGameArt.Org, (https://opengameart.org/), utoljára megtekintve: 2019.02.02.

[16] itch.io, (https://itch.io/), utoljára megtekintve: 2019.02.02.

[17] Goerdt, Jeremiah: An Introduction to Creating a Tile Map Engine. *envatotuts+*, 2013, (https://gamedevelopment.tutsplus.com/tutorials/an-introduction-to-creating-a-tile-map-engine--gamedev-10900), utoljára megtekintve: 2018.12.06.

[18] Bose, Juwal: Creating Isometric Worlds: A Primer for Game Developers. *envatotuts+*, 2013, (https://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-game-developers--gamedev-6511), utoljára megtekintve: 2018.12.06.

[19] Blizzard Entertainment: Diablo 2, (http://eu.blizzard.com/en-gb/games/d2/), utoljára megtekintve: 2018.12.06.

[20] Shekhar, Gyanendu: Difference between Texture and Sprite in Unity. *Gyanendu Shekhar's Blog*, 2013, (http://gyanendushekhar.com/2018/04/14/difference-between-texture-and-sprite-unity/), utoljára megtekintve: 2018.12.06.

[21] Sciutteri, Matteo: Using a Texture Atlas to Optimize Your Game. *envatotuts+*, 2016, (https://gamedevelopment.tutsplus.com/articles/using-texture-atlas-in-order-to-optimize-your-game--cms-26783), utoljára megtekintve: 2018.12.08.

[22] Silverman, David: 3D Primer for Game Developers: An Overview of 3D Modeling in Games. *envatotuts+*, 2013, (https://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704), utoljára megtekintve: 2018.12.08.

[23] Dollar, Brandon: Types of Game Perspectives. *Mozzastryl*, 2013, (https://mozzastryl.wordpress.com/2013/01/20/types-of-game-perspectives/), utoljára megtekintve: 2018.12.08.

[24] Quintans, Desi: Game UI By Example: A Crash Course in the Good and the Bad. *envatotuts+*, 2013, (https://gamedevelopment.tutsplus.com/tutorials/game-ui-by-example-a-crash-course-in-the-good-and-the-bad--gamedev-3943), utoljára megtekintve: 2018.12.08.

[25] Stonehouse, Anthony: User interface design in video games. *Gamasutra*, 2014, (https://www.gamasutra.com/blogs/AnthonyStonehouse/20140227/211823/User_interface_design_in_video_games.php), utoljára megtekintve: 2018.12.08.

[26] 4A Games: Metro 2033, (http://www.4a-games.com/metro-2033.html), utoljára megtekintve: 2018.12.08.

[27] Activision Publishing: Call of Duty, (https://www.callofduty.com/), utoljára megtekintve: 2018.12.08.

[28] Ubisoft Entertainment: Splinter Cell Conviction, (https://www.ubisoft.com/en-us/game/splinter-cell-blacklist/), utoljára megtekintve: 2018.12.08.

[29] Blizzard Entertainment: World of Warcraft, (https://worldofwarcraft.com/en-us/), utoljára megtekintve: 2018.12.08.

[30] Zimarev, Roman: UI Sounds: From Zero To Hero. *Icons8*, 2017, (https://icons8.com/articles/ui-sounds/), utoljára megtekintve: 2018.12.10.

[31] amplifon: The psychology of sound in video games. 2017, (http://www.amplifon.ie/resources/playing-with-your-mind/), utoljára megtekintve: 2018.12.10.

[32] Freesound, (https://freesound.org/), utoljára megtekintve: 2019.02.02.

[33] Pirillo, Chris: Five alternative input devices used in PC gaming. *Chris Pirillo*, 2011, (https://chris.pirillo.com/2011/05/22/five-alternative-input-devices-used-in-pc-gaming/), utoljára megtekintve: 2018.12.10.

[34] Epand, Victor: The Input Devices Of Modern Games, *Streetdirectory.com*, (https://www.streetdirectory.com/travel_guide/104230/gaming/the_input_devices_of_modern_games.html), utoljára megtekintve: 2018.12.10.

[35] Barnwell, Aliya: Here's why you need the hands of a surgeon to beat Flappy Birds on hard. *Digital Trends*, 2016, (https://www.digitaltrends.com/gaming/touchscreen-game-annoyance-explained-via-new-study/), utoljára megtekintve: 2018.12.10.

[36] Sony Playstation, (https://www.playstation.com/en-us/), utoljára megtekintve: 2018.12.10.

[37] Microsoft Xbox, (https://www.xbox.com/en-us), utoljára megtekintve: 2018.12.10.

[38] Nintendo Switch, (https://www.nintendo.com/switch/), utoljára megtekintve: 2018.12.10.

[39] Boots-Faubert, Chris: Gaming Examined: In Praise of the Gamepad P1. *Game On: Cape Cod Gaming Blog*, 2016, (http://blogs.capecodonline.com/cape-cod-gaming/2016/12/28/gaming-examined-in-praise-of-the-gamepad-p1/), utoljára megtekintve: 2018.12.10.

[40] supercobra: Why a Keyboard and Mouse Provide a Better Gaming Experience. *Das Keyboard Blog*, 2012, (https://www.daskeyboard.com/blog/why-a-keyboard-and-mouse-provides-a-better-gaming-experience/), utoljára megtekintve: 2018.12.10.

[41] Hamilton, Kirk: Long Live The Mouse And Keyboard, A Great Way To Control Video Games. *Kotaku*, 2016, (https://kotaku.com/long-live-the-mouse-and-keyboard-1787990911), utoljára megtekintve: 2018.12.10.

[42] Uccello, Anthony: How to Save and Load a Game in Unity. *raywenderlich.com*, 2017, (https://www.raywenderlich.com/418-how-to-save-and-load-a-game-in-unity), utoljára megtekintve: 2018.12.10.

[43] Obbayi, S. R.: Compare Structured and Object-Oriented Programming: What Are the Real Differences? *Bright Hub*, (https://www.brighthub.com/internet/web-development/articles/82024.aspx), utoljára megtekintve: 2018.12.10.

[44] Hardy, Bertram: Video Games & Object Oriented Programming. ~2015, (https://slideplayer.com/slide/8928459/), utoljára megtekintve: 2018.12.10.

[45] Lambert, Steven: Intro to Object-Oriented Programming for Game Development. *envatotuts+*, 2012, (https://gamedevelopment.tutsplus.com/tutorials/quick-tip-intro-to-object-oriented-programming-for-game-development--gamedev-1805), utoljára megtekintve: 2018.12.10.

[46] Gold, Julian: Object-Oriented Game Development: Iterative Development Techniques. *Gamasutra*, 2005, (http://www.gamasutra.com/view/feature/130765/objectoriented_game_development_.php), utoljára megtekintve: 2018.12.10.

[47] Tutorials Point: Component-Based Architecture. (https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm), utoljára megtekintve: 2018.12.10.

[48] Mines, Jonathan: Data-Oriented vs Object-Oriented Design. *Medium*, (https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056), utoljára megtekintve: 2018.12.10.

[49] Cobbett, Richard: The history of RPGs. *PC Gamer*, 2017, (https://www.pcgamer.com/the-complete-history-of-rpgs/), utoljára megtekintve: 2018.12.11.

[50] The CRPG Book Project: Sharing the History of Computer Role-Playing Games. (https://crpgbook.wordpress.com/about-the-project/), utoljára megtekintve: 2018.12.11.

[51] Obsidian Entertainment: Star Wars: Knights of the Old Republic II: The Sith Lords. (https://www.obsidian.net/games/kotor2), utoljára megtekintve: 2018.12.11.

[52] Unity Technologies: Order of Execution for Event Functions, (https://docs.unity3d.com/Manual/ExecutionOrder.html), utoljára megtekintve: 2019.04.12.

[53] Sweeney, Tim: If You Love Something, Set It Free, (https://www.unrealengine.com/en-US/blog/ue4-is-free), utoljára megtekintve: 2022.03.06.

[54] Unity Technologies: ScriptableObject, (https://docs.unity3d.com/2022.1/Documentation/Manual/class-ScriptableObject.html), utoljára megtekintve: 2022.04.03.

[55] Unity Technologies: Input System, (https://docs.unity3d.com/Packages/com.unity.inputsystem@1.3/manual/index.html), utoljára megtekintve: 2022.04.17.

[56] Unity Technologies: Render pipelines, (https://docs.unity3d.com/Manual/render-pipelines.html), utoljára megtekintve: 2022.04.17.

[57] Unity Technologies: Unity UI: Unity User Interface, (https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html), utoljára megtekintve: 2022.04.23.

[58] Unity Technologies: TextMeshPro, (https://docs.unity3d.com/Manual/com.unity.textmeshpro.html), utoljára megtekintve: 2022.04.23.

11. MELLÉKLETEK

11.1. Mondavilág

Terveim szerint a játék előrehaladtával derül majd egyre inkább fény az összefüggésekre. A prototípusi szinten az előbbiből kifolyólag majd csak korlátozott mennyiségű információ lesz a játékos számára elérhető. A leíró és az interakciós szövegeket (például párbeszéd) ezekre kifejlesztett és könnyen kezelhető osztályok és segédosztályok segítségével fogom majd alakítani, illetve ezeket olyon módon implementálni, hogy elméleti síkon a jövőbeli munka is elfogadható legyen velük. Egy nagy projektnél valószínűleg erre a részre egy külön lefejlesztett írói szoftvercsomagot lenne szükséges kialakítani és karbantartani, amivel is maguk a lokalizációs dokumentumok pedig elkülönítve, speciálisabb fájlformátumokban kerülnének tárolásra. Természetesen ezt is tekinthetjük egy továbbfejlesztési lehetőségnek, ráadásul egy igen jelentős egységnek. A saját karakterünk természetesen és persze váratlan módon emlékezetkieséssel kezdi el a szerepét. Csak pár részletre emlékszik majd, ezeket pedig a kezdeti naplóbejegyzésben találhatjuk meg. A játékot majdnem pucéran kezdi el, egy romos állapotú szobor-csonk lábánál. A játékos kódbeli felépítését a game object-hez hozzárendelt szkriptek fogják együttesen meghatározni, ilyenek lesznek például a már említett módon öröklött harcmechanikák, az irányítás, a mozgás és a felszerelés kezelése. Rendelkezésünkre állnak majd más típusú, úgynevezett menedzser objektumok is, de erről majd később kívánok több szót ejteni. Ahogyan az már a specifikációból kiderül, lesznek ellenséges és legalább egy barátságos karakter is, de míg az előbbiekre nem lesz jellemző, hogy el tudjunk velük beszélgetni (hiszen itt minden ellenséget, még akár egy dühös patkányt is beleveszek a fogalomba), addig a barátságosakkal természetesen lehetséges lesz az ilyen jellegű interakció kiváltása. A második pályán (a főmenüt a „nulladiknak” számolva) találkozunk majd Cain-nal. Ez egy rendkívül titokzatos karakter egy igen vészjósló kinézettel: egy nagy kaszával és véres ruhákkal. Ő fog minket valamennyire útbaigazítani, illetve további feladatokat adni. Vele tudunk majd kereskedni is. Az ellenséges karakterek közül kettővel fogunk tudni találkozni. Az egyik a kevesebb veszélyt jelentő patkány, a másik pedig a lebegő szemgolyók, amelyeket inkább el kell kerülni. Architekturális szempontból a felhasználóén kívüli karakterek hasonló módon, a sokat emlegetett kódrészletek segítségével lesznek felépítve. Mindegyik csak azokkal az interfészekkel és az azokat teljesítő egységekkel fog rendelkezni, amelyekre neki adott esetben szüksége van. Erősen törekedni fogok arra, hogy ugyanazt a rendszert több helyen ne implementáljam.

11.2. Területek

A játékvilágot alapvetően a sötétség és a ridegség fogja meghatározni. „Feltételezhetően” egy olyan területcsoportól van szó, amely a föld alatt van, ugyanis

mindent falak vesznek körül, és nem figyelhető meg egy általános fényforrás sem. A történetnek ez része is lehet, mármint, hogy rájöjjünk mivel is áll a játékos szemben. A Vadonokban kezdődik igazán a játék. Ez egy nagyrészt gyér hatású terület, amit alapvetően egy erdős vadon dominál. Itt-ott fellelhetünk civilizációs maradványokat, egy házat és zárt kapukat is. Nem nehéz elképzelni, hogy számtalan veszélyes kreatúra várakozik itt az erre tévedt és óvatlan kalandorokra. A Menedéket tekinthetjük a játékosunk otthonának. A már említett baráti karakter is itt található meg. Nagy részét egy régi és igencsak elhagyatottnak tűnő, de különös kripta teszi ki. Lehetséges, hogy többet rejt, mint ahogy azt első pillantásra gondolná a játékos…

11.3. A* algoritmus

Az „Astar" osztályban fellelhető „FindPath" függvény két általam definiált csomópontot vár, vissza pedig egy, ezekből álló enumerációt ad. Az osztály példánya, ami a már említett menedzserben található, rendelkezik a szintén általam felépített gráf referenciájával. Egy útkereséses „Node" osztálynak jellemezném most az eddig nem tisztázott, az algoritmushoz kapcsolódó tulajdonságait: ezek a „G", „H", „F" és „PreviousNode". A G jelzi azt, hogy a jelenlegi csúcsra hány ugrás árán jutottunk el a kiinduló pontból, látni fogjuk, hogy ez változhat, illetve felülíródhat. A H egy kötött érték, aminél is valamilyen módon minden érintett pontra meg kell azt mondjuk, hogy amennyiben például nem létezne semmilyen akadály, mennyire lenne messze a cél a jelenlegi pozícióból. Ezt tetszés szerint lehet finomítani és alakítani, én nagyon egyszerűen csak a sima távolsággal számolok. Az F pedig dinamikus módon, a kettő összege. Az előző csúcs referenciája egyrészt pont a G változékonysága miatt fontos: melyik volt az a csúcs, ahonnan idelépve ez a pontszám megszületett? Másrészt, maga az útvisszafejtés számára is elengedhetetlen egy ilyen jellegű referencia. Első lépésként visszaállítjuk az összes csúcs két értékét: a G-t és a PreviousNode-ot. Míg az előzőt a pozitív végtelenre (int.MaxValue), addig a másodikat pedig az alapértelmezettre (default, ebben az esetben „null"). Alapvetően ugye egyik pontra se léptünk még rá egyrészt, másrészt pont emiatt a pontok fellelhetőségének díja kifizethetetlen, vagyis végtelen. Az utóbbit a C#-ban csak a fenti módon tudjuk jelezni. Eztán a kezdeti pont G-jét lenullázzuk, H-ját beállítjuk. Az algoritmus két segéd adatszerkezettel dolgozik: a „processable" és „processed" Nodes halmazokkal (HashSet). A feldolgozható ezen kezdeti csúccsal rendelkezik csak, a feldolgozott pedig üres. Végre elkezdődhet maga a fő ciklus. A külső bejárás elől tesztelő feltétele: a feldolgozható csúcsok száma nagyobb, mint nulla, és ez a szám nem haladja meg a maximálisan feldolgozhatók mennyiségét. A másodikat azért kellett bevezetnem, mert ha éppen nincsen út a játékos felé, egy empirikus határon túl már nem érdemes a függvénynek keresnie. Egy fejlesztési ponton jelentős teljesítmény romlást jelentett ennek a feltételnek a hiánya bizonyos határesetekben. A cikluson belül aztán kiválasztjuk a jelenleg vizsgálandó csúcsként azt a feldolgozandók közül, amelynek a legkisebb az F értéke. Amennyiben ez a célcsúcs,

visszafejtjük az ehhez vezető útvonalat az előző referenciák segítségével, majd a struktúrát visszaadjuk, kilépünk a metódusból. Abban az esetben, amikor is nem voltunk ennyire szerencsések: kivesszük a jelenlegit a feldolgozandók közül és berakjuk a feldolgozottakhoz. Ezen a ponton kezdődik a belső ciklus. A belső hurok „foreach" típusú, a kiválasztott csúcs statikus akadályoktól szabad és létező szomszédjain megy végig. Ez az információ a gráf példányban tároltatik el inicializáláskor, egy listákból álló, kétdimenziós tömb formájában, minden csomóponthoz viszonyítva. Az itt kiválasztott pontot nevezzük tehát szomszédos vagy szomszéd csúcsnak. Először megvizsgáljuk, hogy feldolgozottnak minősül-e már a csúcs, ha igen, „continue" utasítással tovább iterálunk. A következő vizsgálat hasonló: ha a pont esetleg dinamikusan foglalt (pl. játékos vagy ellenség áll rajta), hozzáadjuk a feldolgozottakhoz és megint csak tovább iterálunk. Ez egyébként a node-on az „IsOccupied" lekérdezhető. Ezután kiszámítunk egy ideális szomszédos „G határt": fogjuk a jelenlegi csúcs G-jét a külső ciklusból és hozzáadjuk a jelenlegi és a szomszédos távolságát. Amennyiben ez nagyobb vagy egyenlő a szomszédos már beállított G-jével (tehát azt már érintettük), szükségtelen ennek a további vizsgálata, vagyis megint csak tovább iterálunk. Ám, ha ez a szomszédos vizsgálatilag még egy valóban hasznosnak tekinthető csúcs: beállítjuk a G-jét az előbb említett-re, a H-ját is kiszámoljuk, illetve beállítjuk az előző referenciáját a jelenlegi csúcséra. Utoljára, még a belső ciklus végén, megnézzük, hogy a vizsgált szomszédos csomópont része-e a feldolgozhatók halmazának, ha igen, befejeztük a vizsgálatot, ha viszont nem, akkor hozzáadjuk. Ez azért fontos, mert ahogy ezt már említettem, a csúcsok előfordulhatnak többször is, csak különböző, és egyre optimálisabb megközelítésekkel.

11.4. Szintlépés

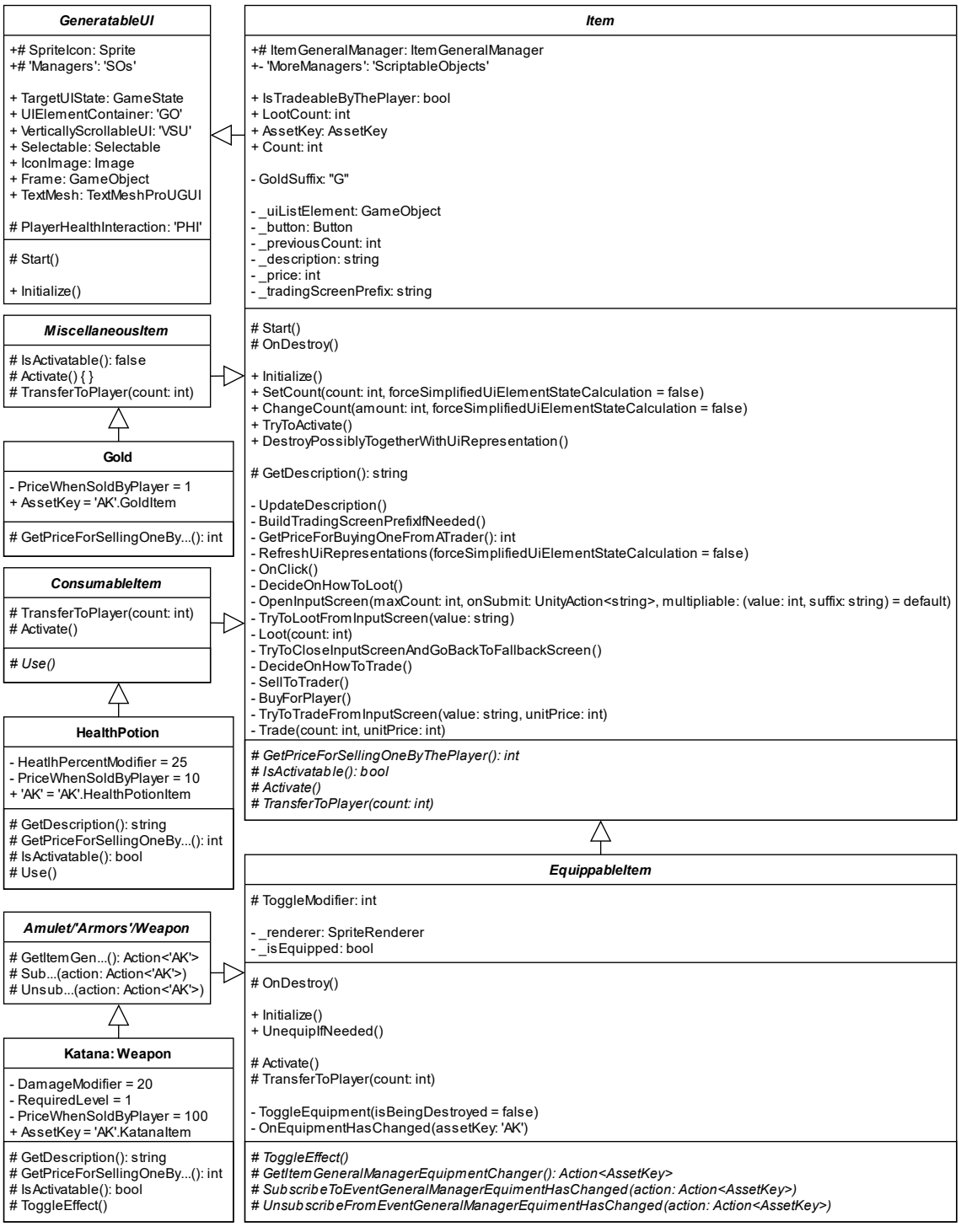
A következőkben olvasható majd a kettő általam implementált kiszámítási módja, azonban ehhez először be szeretnék vezetni pár rövidítést: F („FirstLevelUpExperienceRequirement"), M („NextLevelExperienceMultiplier"), E („Experience"). Alább látható tehát a két képlet, miszerint ezt a kettőt a rendszer figyeli és kezeli.

$$Level = 1 + \frac{\ln((F + (M - 1) * E)/F)}{\ln M}$$

$$NextLevelExperienceRequirement = \mathrm{F} * \frac{1 - M^{Level}}{1 - M}$$

Szerepelnek tehát a fenti függvényekben konstansok. Ezeket a szerkesztőben könnyen, az SO-n keresztül tudja módosítani a fejlesztő. A különböző teszteléseim után ezek jelenleg a következők: F = 100, M = 1.5. A fejlődési függvény egy nagyon sokat vitatott témakör, elsősorban játékmenet tervezők szokták ezt különböző filozófiák mentén kialakítani.

11.5. Tételhierarchia



11.6. Képernyőképek