ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Dorogi Tamás**
2021	Hallgató törzskönyvi száma:	**T/005605/FI12904/N**

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Dorogi Tamás ..... [redacted] Nappali .....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka címe magyarul:

Neurális hálózatok kihasználhatóságának vizsgálata a játékiparban, egy Unity motorban létrehozott programmal demonstrálva .....

Szakdolgozat / Diplomamunka címe angolul:

The possibility of using Neural networks in the game industry, demonstrated by a program created in Unity game engine .....

Intézményi konzulens: Külső konzulens:

Lovas István..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2020.02.18	Témaválasztási konzultáció	
2.	2020.03.03	Szakirodalom áttekintése	
3.	2020.04.14	Lehetséges megvalósítás áttekintése	
4.	2020.05.12	Tartalmi ellenőrzés	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplo-mamunka 2 / Diplomamunka 3 / Diplomamunka 4 tantárgy követelményét teljesítette, beszámolóra / védésre bocsátható.

Budapest, 2020. 05. 19.

.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Dorogi Tamás ..... [redacted] Nappali .....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat / Diplomamunka címe magyarul:

Neurális hálózatok kihasználhatóságának vizsgálata a játékiparban, egy Unity motorban létrehozott programmal demonsrálva .....

Szakdolgozat / Diplomamunka címe angolul:

The possibility of using neural networks in the game industry, demonstrated by a program created in Unity game engine .....

Intézményi konzulens: Külső konzulens:

Lovas István..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2021.02.10	Specifikáció áttekintése	
2.	2021.03.13	Rendszerterv áttekintése	
3.	2021.04.13	Fejlesztés, tesztelés ellenőrzés	
4.	2021.05.11	Tartalmi, formai ellenőrzés	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. / Szakdolgozat II. (BSc) vagy Diplomamunka 1 / Diplomamunka 2 / Diplomamunka 3 / Diplomamunka 4 tantárgy követelményét teljesítette, beszámolóra / védésre bocsátható.

Budapest, 2021. 05. 12.

.....
intézményi konzulens

Óbudai Egyetem
Neumann János Informatikai Kar
Kiberfizikai Rendszerek Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Dorogi Tamás**
Törzskönyvi száma: T/005605/EI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Neurális hálózatok kihasználhatóságának vizsgálata a játékiparban, egy Unity motorban létrehozott programmal demonstrálva
The possibility of using Neural networks in the game industry, demonstrated by a program created in Unity game engine

Intézményi konzulens: Lovas István
Külső konzulens:

Beadási határidő: 2020. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
IoT,beágyazott rendszerek és robotika specializáció

A feladat

Tervezzen és valósítson meg egy játék alkalmazást, melyben az ellenfelet egy mesterséges neurális hálózat irányítja. A cél, hogy a játékos a mesterséges intelligenciát elkerülve, vagy kijátszva megnyerje a játékot. A programnak rendelkeznie kell egy irányítható karakterrel, egy előre definiált 3D játéktérrel, illetve egy külön, az ellenfelet irányító mesterséges intelligenciával. Ezen technológia segítségével próbáljon egy lehetséges megoldást találni, az előre megírt események használatával elért kiszámíthatóságra, monotonitásra.

A dolgozatnak tartalmaznia kell:

- a megoldandó feladat pontos leírását és részletes elemzését,
- a meglévő alkalmazások vizsgálatának elemzését,
- az alkalmazni kívánt technológiák elemzését,
- a rendszertervet és döntések indoklásait,
- a megvalósítás leírását,
- a tesztelési tervet és a tesztek eredményeit.

Ph.

.....
Dr. habil. Lovas Róbert
intézetigazgató

A szakdolgozat elévülésének határideje: **2022. december 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

HALLGATÓI NYILATKOZAT

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

Budapest, 2020.05.15

.....

hallgató aláírása

Tartalomjegyzék

Absztrakt

Dolgozatomban egy Unity 3D játékmotorban fejlesztett példán keresztül mutatom be, hogy a mai játékok egy csoportjában hogyan lehetne felhasználni a tanulás alapú mesterséges intelligenciákat. Az alapvető problémafelvetésem, hogy egy idő után az előre definiált események monotonná, vagy kiszámíthatóvá tehetik a játékmenetet. Erre szeretnék egy megoldást találni, egy olyan mesterséges intelligenciával kiegészített játékkal, mely minden egyes időpillanatban hoz egy döntést a játékos tettére, vagy más változóra alapozva. A játék maga egy First Person „sétaszimulátor" lenne egy előre behatárolt és felépített környezetben, melyben egy célt keresve kell végig mennünk, különböző puzzle feladatokat megoldva. Ezt probálja az említett, neurális hálóval megvalósított AI megakadályozni, vagy nehezebbé tenni. Maga a környezet, a belső tere egy elhagyatott űrállomásnak, melyet a játék során bebarangolhatunk. A dolgozatban kitérek a Unity platform előnyeire és hátrányaira, illetve, hogy miért erre esett a választásom. Megvizsgálom a mesterséges neurális hálózatok játékiparra vetett lehetséges hatásait, illetve elterjedésük hiányának okait. Kitérek az ezen dolgozatban használt gépi tanulási módszerekre, illetve könyvtárakra. Továbbá meg kell felelni bizonyos a Gamedesign területén lefektetett alapoknak is.

Cél meghatározása

Szakdolgozatom célja egy olyan játék létrehozása, mely - eltérően a konvencionálisan megalkotott programoktól - olyan mesterséges intelligenciával van felszerelve, amelynek alapját egy mesterséges neurális hálózat adja. Ezzel segítve a minden újra játszásnál eltérő élményt, és játékosra szabott eseményeket. Ez az úgynevezett „scriptelt" történéseket helyettesítené, amelyek egy bizonyos idő elteltével kiszámíthatóvá teszik a játékokat.

Az elkészítés közben kitérek a Unity játékmotor tulajdonságaira és szeretnék egy képet alkotni arról. Azon belül is, annak 3D részegységéről. Ismertetem majd az általam készített, illetve felhasznált játékmechanikai elemeket, melyek a játék irányításáért és a játékszabályokért felelnek.

A kész program egy First Person – belsőnézetes – kalandjáték, melyben a játékos különböző interakciókat hajthat végre egy előre felépített környezetben, miközben próbálja elkerülni az ellene tevékenykedő neurális hálózattal ellátott mesterséges intelligenciát.

1. Bevezetés

Egész életem során érdekeltek a játékok. Ezeken nőttem fel, és végsősoron nekik köszönhető az érdeklődésem a számítástechnika iránt is. A velük való játék mellett mindig is érdekelt, hogyan készülnek, és szerettem volna készíteni egyet. Azonban mindig feltűnt, hogy a többségükben a mesterséges intelligencia nem valódi. Néha csak kisebb megmozdulásokból, mint pl.hogy ugyanazt a mozgást végzi mindig, vagy hogy szélsőséges esetben nem veszi észre, ha előtte állok. Ezekre a dolgokra a mai játékokban már nagyon figyelnek, ugyanakkor még mindig látható, hogy csak imitálják az emberi viselkedést. A NeocoreGames egyik game designere azt mondta nekem, hogy "akkor életszerű egy AI, ha félti az életét". Ez persze csak bizonyos típusú játékokra vonatkozik, de valóban igazibbnak érzünk egy ellenfelet, ami fedezék mögé bújik veszély esetén ahelyett, hogy hozzánk futna. Ez valóban igaz, és tettenérhető az újabb programoknál. Sőt! Az utóbbi 15 év fejlesztéseinél már egyre okosabbak a játékbeli intelligenciák.

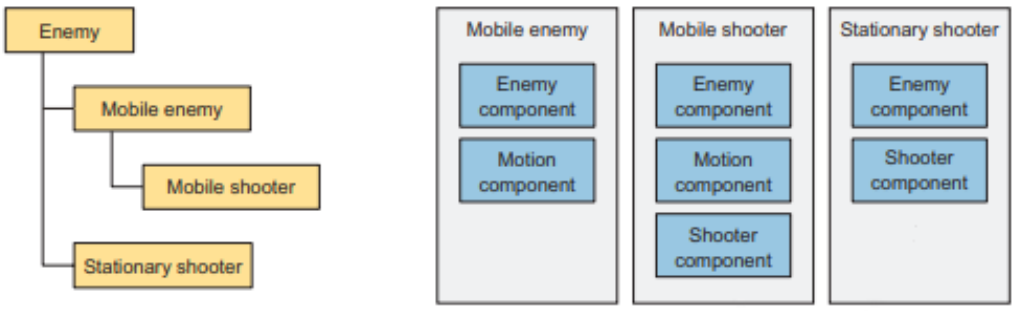
Látható azonban, hogy még mindig nem eléggé emberi, hiszen legtöbb esetben ugyanazt csinálják újra és újra. Részben ennek is köszönhető, hogy egy ilyen szoftver sokak számára monotonná, kiszámíthatóvá válik. Egy olyan AI, ami nem cselekszik előre megírt szabályok szerint, megoldás lehetne erre a problémára. Ez a fajta megközelítés - mai tudomásunk szerint - a gépi tanulás módszerével érhető el, amely egy régebbi technológia, de nemrég kezdték felfedezni a benne rejlő lehetőségeket. Egyre okosabb AI-k születnek, melyek sok területen jobbak az embernél az adott, akár igen komplex területeken.

A videojátékok piacára azonban, még nem tört be ez a megközelítés. Dolgozatomban azt vizsgálom, hogy hol lehetne ezeket felhasználni ebben az iparágban, és egy játékot készítek el, mely olyan mesterséges intelligenciával rendelkezik, amelyet egy gépi tanulási algoritmus is hajt. Ezen kívül szeretnék egy rálátást kapni a játékfejlesztésben használt technikákra, megérteni a játék egyes részei hogyan működnek, készülnek el. Ebben a segítségemre egy általam előre kijelölt játékmotor lesz, melynek használatáról szintén szeretnék egy általános képet kapni.

2. Irodalomkutatás

2.1. A Unity Engine

A Unity[1] egy videojáték motor, melyet a Unity Technologies fejleszt. Ez a világ egyik legelterjedtebb professzionális motorja, melyet nap mint nap rengeteg programozó használ. Elterjedése többek között köszönhető a magas minőségű multiplatform támogatásnak és a letisztult, vizuális megközelítésű, könnyen kezelhető fejlesztői környezetének. Ahhoz, hogy tisztában legyünk a későbbiekben felhasznált tudásanyaggal, értenünk kell, miért fontos valamilyen motorban fejlesztenünk a játékunkat ahelyett, hogy a nulláról építenénk fel. Egy adott Game engine-ben létrehozott program, örökli annak speciális vonásait, emellett hozzáadhatunk saját modelleket (asseteket) és specifikus kódokat, algoritmusokat a játékunkhoz. A teljesség igénye nélkül, a Unity rendelkezik saját fizikai szimulációkkal, „normal map" leképezéssel, SSAO-val (screen space ambient occlusion), dinamikus árnyékokkal, és részecske effektekkel, melyet a benne készített játékok felhasználhatnak. Ez jelentősen meggyorsítja a fejlesztést, kvázi keretrendszerként működik. Amikor specifikusan Unityről beszélünk, meg kell említenünk a többi megoldástól eltérő moduláris komponens rendszert, mely a játékobjektumok felépítésében segít. A komponens rendszer egy, az objektum orientált programozásban használt megközelítés, ahol az objektumok komponensekből vannak felépítve, az általánosságban használt öröklés helyett.[2]

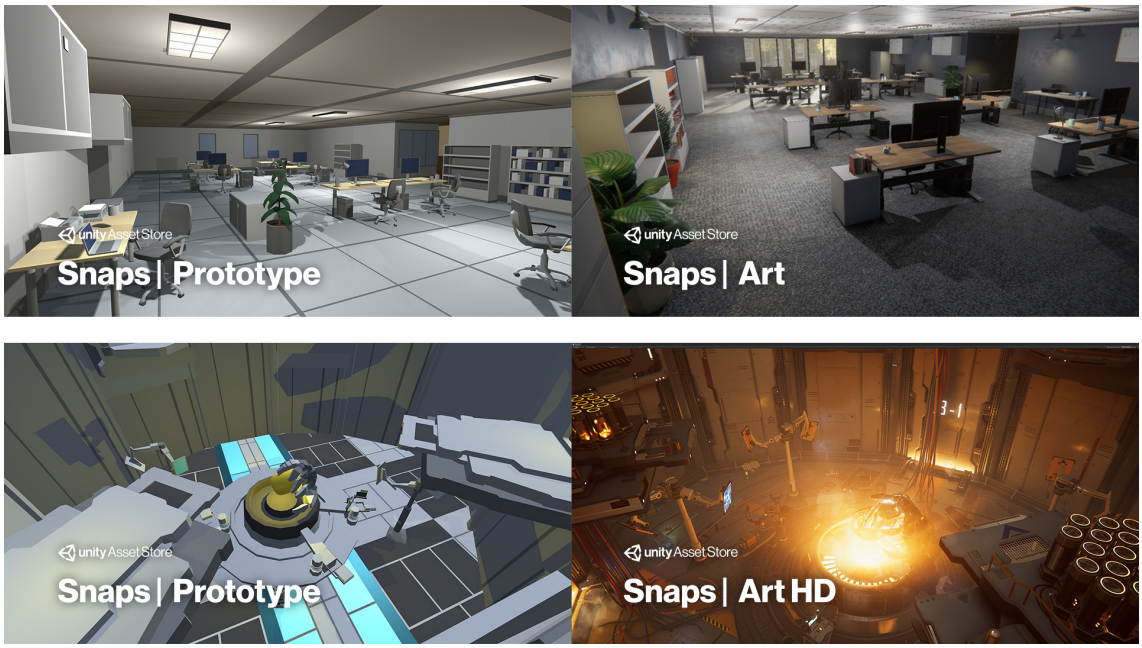


2.1. ábra. Az öröklés(bal) és a komponens(jobb) rendszerek összehasonlítása

Ez a megoldás a játékfejlesztésben, flexibilisebb megközelítést enged meg. Elsősorban a gyors prototípusok készítését segíti, mivel a különböző komponenseket át lehet rakni egyik objektumból a másikba. Itt meg kell azonban említeni, hogy öröklést is lehet választani a kódjainkban, ezzel elérve, hogy használhassuk azokat a legjobbnak ítélt design mintákat, melyek ilyen úton oldják meg a problémát.

Fontos megjegyezni, hogy a Unity rendelkezik egy ingyenes disztribúcióval, mely lehetővé teszi használatát teljesen kezdő, vagy tanulni vágyó játékfejlesztőknek is. Hasonló kategóriában legnagyobb versenytársa az Unreal Engine 4, mely szintén ingyenesen használható, tanulható. Fejlesztői szempontból az egyik legnagyobb különbség, hogy míg a Unity a C# programozási nyelvvel való fejlesztést teszi lehetővé, addig az Unreal motor a C++ nyelvvel működik együtt. Így sok felhasználó, köztük magam is, a nyelvi preferencia miatt választja a Unity-t.

Vannak a Unitynek ezekkel szemben hátrányai is. Ezek közül is számomra a legfontosabb, a külső könyvtárak importálásának hiánya. A probléma megoldható azzal, ha a teljes library-t bemásoljuk a projektünkbe, de ezeket nem tehetjük elérhetővé egy külső hivatkozáson keresztül.[2]



2.2. ábra. A Snaps Prototype és Snaps Art összehasonlítása

2.1.1. Assetek

Egy számítógépes játék elkészülte rengeteg időt igényel, még nagyobb csapatoknak is. Természetesen ez függ az adott játék nagyságától és komplexitásától. Ezért az egyik első feladat egy mindössze egy emberes fejlesztésnél, kiszámítani mekkora bonyolultságú játékot vagyunk képesek adott idő alatt elkészíteni. Az esetemben adott volt, hogy 3D-ben dolgozom és belső nézetből. Az ilyen típusú programokban egy nagy szeletét a fejlesztésre szánt időnek a 3D modellek elkészítése adja. Mivel a modellezés egy külön szakma, így sok fejlesztő nem rendelkezik az ehhez szükséges

tudással, vagy idővel. A Unity-ben erre az Asset store[3] adhat megoldást, mely egy online piactér. Itt tematizált modellcsomagok, effektek, hangok ezreit vehetjük meg, tölthetjük le azokat, felhasználandó saját programunkban. Ezzel a megoldással a töredékére csökkenthető a fejlesztési ciklus, az asset-eket jól kombinálva pedig, egyedülálló játékokat hozhatunk létre.

Számomra is ez a megoldás volt az egyetlen járható út, így olyan professzionális, moduláris asset csomagokat kellett találnom, ami elégséges az általam kitalált pálya felépítésére. Ezek közül is kiemelkedik a Unity saját csapata által készített Snaps projekt. Ez egy több csomagból álló asset család, melynek célja, hogy megkönnyítse a prototípusok készítését a Unityban. Ennek lényege, hogy egy témában két csomag áll rendelkezésre. Az egyik, a prototípusok készítéséhez kiváló "Prototype", melyben alacsony poligon számmal rendelkező "Mesh"-ek találhatóak, míg a másik, a magas felbontással rendelkező "Art" csomag (2.2 ábra). Ezzel mégjobban megkönnyítve a gyors pályaépítést, a prototípusok készítését, illetve azok egy kattintásra való átváltását a végleges formára. Másik különlegessége, hogy a csomagok modulárisak, és az egyes modulok tökéletesen összeilleszthetők.[4] Ehhez a ProGrids nevű bővítményt kell használnunk, mely teljesen pontossá, és könnyen kezelhetővé teszi az Editorban végzett összeillesztéseket. Maguk a modellek a szintén Unity bővítmények, ProBuilder-ben készültek, ezzel teljesen Unity-re szabva a modelleket, kizárva az inkompatibilitásokat és könnyen elérhetővé téve a saját magunk által végzett átalakításokat a ProBuilderben.

A ProBuilder egy ingyenesen felhasználható bővítmény, amely tulajdonképpen egy leegyszerűsített modellező program a Unity Editor rendszerét felhasználva. Gyorsan és egyszerűen készíthetőek benne modellek, melyeket azonnal felhasználhatunk. A textúrák készítése és az UV térképezés ugyanúgy megoldható benne. Előbbi a Unity Editor, utóbbi pedig a ProBuilder sajátja. Amennyiben textúrákat akarunk készíteni saját, ProBuilderben létrehozott modelljeinkhez, úgy az exportálás és importálás is egy egyszerű folyamat. Ezek összessége a Unity egyik legnagyobb előnyéhez vezet, amit "Rapid Prototyping-nak" nevezünk.[5]

2.1.2. Játékfejlesztési Technikák

A következőkben a kikerülhetetlen játékfejlesztésben használatos technikákat, motorokat ismertetem, melyek az én programom esetén is nélkülözhetetlenek.

Fények: A játékfejlesztésben a fényeknek két típusa van. Ezek a "Baked" és a valós idejű (Real-time) fények. Az elsőt általában statikus objektumoknál szokták használni, melyek a játék során nem változnak. Ezek előre kalkulált fények, változások a textúrán. Megadhatunk néhány paramétert a számítás

előtt, és a motor azokat felhasználva fogja a fénytérképet felrajzolni. Ennek ellentéte a Real time megvilágítás. A Unityben ennek több fajtája van. A Spot light, Directional light, Point light és az Emission Light. Ezek többnyire valamilyen valós fényt szimulálnak. Mivel a valós idejű fény-árnyék hatások nagy erőforrásokat mozgatnak meg, célszerű minél kevesebb, jól elhelyezett fényt használni, és statikus fényekkel kiegészíteni azokat.

Post-processing: A Post processing egy képi hatásokat feljavító technika, kvázi a kameránkon lévő szűrőként viselkedik. Ezzel elérhetünk többek között még valóságosabbnak ható fény-árnyék hatásokat, vagy különböző effekteket használhatunk, melyek a látott kép valamilyen formában történő megváltoztatását szolgálják. A Post-processing elérhető a Unityben letölthető csomagként, Post-Processing Stack néven.

Fizikai motor: A fizikai motor valós fizikai szimulációkat végez el, élethűbbé téve az élményt. Ezt a Unityben beépítették a motorba, így az Editor grafikus felületén elérhető. Amennyiben egy adott objektumot ellátunk egy úgynevezett Collider komponenssel, úgy alkalmazni tudjuk rá az általunk megírt fizikát. Egy collider az adott objektum alakját definiálja, mely az ütközések számításánál segít. 3D programokban ezek általában Mesh Colliderek.[6]

2.1.3. Unity ML-Agents Toolkit, PyTorch és TensorBoard

A Unity Machine Learning Agents Toolkit egy nyílt forráskódú Unity bővítmény, mely lehetővé teszi, hogy játékok és szimulációk szolgáljanak környezetként intelligens ágensek tanításában. Az ágensek használhatnak megerősítéses tanulást, imitációs tanulást, neuroevolúciót, és más gépi tanulási módszereket egy Python API-n keresztül. Ez lehetővé teszi, hogy nem játékos karaktereket tanítsunk az általunk megírt program használatára. Az ML-Agents a legkézenfekvőbb módja a gépi tanulás használatára Unity játékokban, mert jelentősen megrövidíti az AI kifejlesztését és betanítását.[7] Mivel ezen toolkit segítségével szeretném implementálni a mesterséges intelligenciát, bővebben is részletezem ezt az eszközt.

Az ML-Agents PyTorch alapú, így ennek rövid bemutatása is szükséges: A PyTorch egy open-source python bővítmény, mely képes számításokra adatfolyam gráfok használatával. Ezek tulajdonképpen a deep learning modellek valódi reprezentációi. Ez elősegíti a használt eszköz CPU, illetve GPU-ján való tanulást. Az ML-Agents Toolkit használatakor, a viselkedés tanítás után a kimenet egy modell (.onnx file), amelyet az ágenshez kapcsolhatunk.

Szót kell ejteni még a TensorBoard-ról, mely nagyban segíti, az AI tanítási folyamatát, azaz emberek számára átláthatóbbá teszi azt. A PyTorchban való tanítás

egyik komponense bizonyos modell tulajdonságok értékeinek bellítása. Ezeket nevezzük hiperparamétereknek. A saját esetünkre a hiperparaméterek optimális értékének megtalálása időigényes feladat lehet. Ennek megkönnyítésére használhatjuk a TensorBoardot, mely egy olyan vizualizációs eszköz, ami megjeleníti bizonyos hiperparaméterek értékeit a tanítás folyamata alatt, így könnyen nyomonkövethetjük a beállt változásokat. Ez elősegítheti a jó irányba való változtatásokat és megkönnyíti az optimális értékek beállítását.

2.2. Mesterséges intelligencia a játékokban

Ahhoz, hogy a mesterséges intelligenciáról beszéljünk, előbb fel kell tennünk a kérdést, hogy mi is az az intelligencia. Tudósok és filozófusok már több ezer éve próbálják meghatározni, körülírni ezt. A szótári definíció szerint „az intelligencia a képesség a gondolkodásra, érvelésre és megértésre ahelyett, hogy ok nélkül, ösztönből cselekszünk" [Collins]. Ez a leírás megfeleltethető az emberi intelligenciának, azonban a számítógépes játékok szempontjából pontatlan. Ebben az esetben inkább viselkedésre, bizonyos eseményekre való reakciókra gondolunk, nem pedig belső gondolatokra. Ezek legtöbbször egy látható változáshoz vezetnek, az adott számítógép irányította entitás állapotában. Megjegyezném azonban, hogy a tervezés, melyhez bizonyos szintű előtudás szükséges az adott témával kapcsolatban, egy fokig belső gondolkodást igényel, amely sokszor a mesterséges intelligenciák sajátja. Ezt a szempontot is hozzáadva mit nevezhetünk a számítástechnikában, de még inkább a videojátékokban intelligens viselkedésnek?[8] Egy válasz lehet, amely a következő: olyan viselkedés, ami életszerű. Formálisabban olyan intézkedés, mely az adott szituáció kontextusában odaillő. Ez a leírás ránézésre tükrözi az intelligenciát, anélkül, hogy valójában intelligensnek tekinthető entitásról beszélnénk. Dolgozatom szempontjából ez a definíció elegendő, így nem térnék ki azokra a kutatásokra, amelyek az emberi szintű mesterséges intelligencia határait feszegetik.[9]

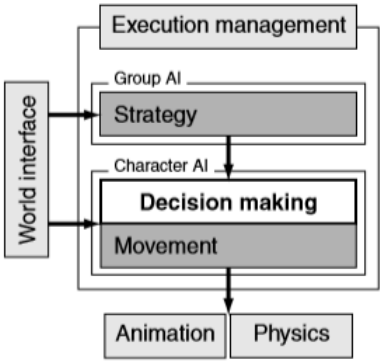
A fenti leírásból következik, hogy a legtöbb videojátékban fellelhető AI, valójában nem a tudományos értelemben vett mesterséges intelligencia, hanem inkább egy összessége bizonyos technikáknak, melyeknek az a feladatuk, hogy egy hihető illúzióját keltsék az intelligenciának. Ezekben az esetekben az adott AI-nak többnyire annyi a feladata, hogy elemezze és kiértékelje a beérkező adatokat saját környezetéből. Legtöbbször ezek szenzoradatok, vagy kommunikáció a többi NPC-vel (Non-player character). Így lebonthatjuk egy adott NPC döntéshozatali ciklusát három lépésre:

1. észlelés (információ fogadása a környezetből – szenzor információ)

2. gondolkodás (a kapott információ feldolgozása és ezek szerinti tervezés)

3. cselekvés (az eltervezett akció végrehajtása)

Ez egy meglehetősen leegyszerűsített felsorolás, azonban az esetek legnagyobb százalékában elegendő egy játékban. Az AI-nak egy szórakoztatóipari termékben nem kell jobbnak vagy kompetensebbnek lennie a játékosnál, ami ellehetetlenítené a továbbhaladását. Ezzel szemben az lenne a feladata, hogy az AI engedje a játékost nyerni, méghozzá egy izgalmas és kihívást jelentő úton. Ez persze kizárólag az ellenséges NPC-kre vonatkozik. A játékok fejlődésével teret kaptak a neutrális jellemű, vagy baráti AI-k is, melyeknek többnyire az immerzió elősegítése, vagy a játékosok asszisztálása a feladata. A számítási teljesítmény növekedésével egyre grandiózusabb játékok születtek, és mára már eljutottunk abba a fázisba mikor fontos tényezővé vált egy játék megítélésében, hogy magához a játékhoz sokat hozzá nem tevő háttér NPC mennyire realisztikusan viselkedik. Ez persze nagy számítási kapacitást igényelhet.[10]



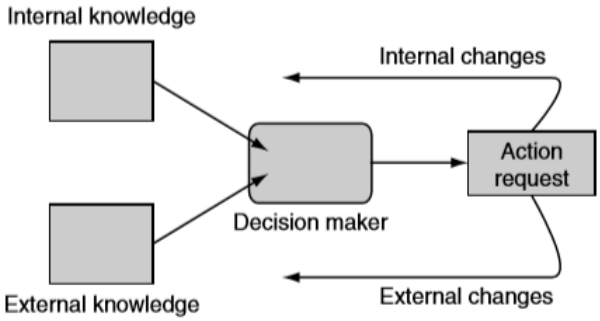
2.3. ábra. Az AI általános modellje

2.2.1. Nem tanulás alapú döntéshozási módszerek

A döntéshozás csak egy része a videojátékokban használt AI-nak, azonban dolgozatom szempontjából ez a legfontosabb. Ez dönti el, hogy milyen eseményre, mit tegyen az adott karakter. Az AI-nak ugyan része az animáció vagy a mozgás és azok kivitelezése, de a burkot ezek köré a döntéshozás adja, mely folyamatban eldől mikor, és hogyan tegye meg azokat. A legtöbb mai játék viszonylag egyszerűbbnek mondható technikákat használ úgy mint állapotgépeket (state machine), vagy viselkedés fákat. Az utóbbi években egyre nagyobb érdeklődés mutatkozott a kifinomultabb döntéshozó technikák iránt is, úgy mint a neurális hálók, vagy a fuzzy logika, de a jól bevált stratégia elhagyása lassú folyamat, és mivel sokszor nehéz jól működőre alkotni ezeket, a fejlesztők nem sietnek.

Beszélhetünk többféle döntéshozó modellről, habár mindegyik hasonlóképp cselekszik. A karakter feldolgoz valamennyi beérkező adatot, és ezekből generál egy akciót, melyet végre szeretne hajtani. Az inputot a döntéshozó rendszerbe, a karakter által birtokolt információk jelentik. A kimenet pedig egy akció kérés. Ezek tobább bonthatók belső és külső ismeretanyagra. Külső ismeretanyag, az adatok összessége, melyet az entitás tud a környezetéről, míg belső ismeretanyaga az, amit saját magáról tud. Az akcióknak két komponense lehet: kérhetik, hogy a változtassák meg az NPC külső állapotát (pl. mozgás egy szobába), vagy változtassák meg a belső állapotát (pl. véleményváltozás a játékosról)(lásd 3. ábra).[11]

A következőkben nem gépi tanulás által vezérelt AI technikákat mutatok be, amelyek betekintést nyújtanak a még ma is használt gyakorlatba. Elengedhetetlen tudni ezek mibenlétét, előnyeit és hátrányait ahhoz, hogy összehasonlítsam a gépi tanulás alapú megközelítésekkel. Bizonyos részeit az utóbbiban is fel kell használnom.



2.4. ábra. Vázlat a döntéshozatalról

Decision tree: A döntési fák könnyen implementálhatóak, és egyszerű a megértésük. Ezek számítanak a legegyszerűbbnek a taglaltak közül, de kibővítve az algoritmusokat szofisztikáltabb megoldások születhetnek. Főleg a játékbeli karakterek irányítására szolgálnak, vagy animációk kontrollálásánál használják őket. Előnyük, hogy modulárisak és könnyen elkészíthetők. Lényege, hogy egy bizonyos mennyiségű információból generálnunk kell egy ahhoz kapcsolódó műveletet előre definiált lehetséges cselekvésekből. Az input és output közötti belső út igen komplex lehet. Sok különböző input adatokra lehet elérni ugyanazt a cselekvést, de egy kis változás a bemenő adatban is elmozdíthatja a görbét odaillő, és nem odaillő cselekvés között. Erre a megoldás egy metódus, amely egy cselekvés közben csoportosítja az input adatokat, és azokat a bemeneti adatokat engedélyezi, melyek szignifikánsak a kimenet megadásához.

A döntési fák összekapcsolt döntési pontokból állnak. Minden ág végén egy döntés áll, melyek közül lehet több ugyanaz, így, többfajta folyamat is vezethet

ugyanahhoz a döntéshez. [11]

Finite State Machine: A Finite State Machine-k (FSM) a máig egyik legggyakrabban használt AI technika a játékokban. Egy FSM-ben az NPC-k viselkedése lebontható logikai állapotokra - egy állapot, egy lehetséges viselkedéshez - melyek közül egyszerre csak egy lehet aktív. Egy állapotot egy boolean értékként definiálunk, amely aktív, vagy inaktív. Amikor egy adott állapotot meg kell változtatni, pl. védésből támadásba kell kapcsolni, az FSM állapotot fog váltani. Egy FSM leprogramozása viszonylag egyszerűnek tekinthető, mely nem lesz túl kifinomult, képes lesz az elvárható stabilitásra, és megteszi a kellő minimumot. Ezen technika hátulütője, hogy az FSM gyorsan túlkomplikálttá és nehezen karbantarthatóvá válhat. Azonban, ha túl egyszerű, akkor könnyen kiszámíthatóvá válhat a viselkedése. Ezt a tulajdonságot, még mai videojátékokban is megfigyelhetjük, és játékosként rendszeresen építünk erre. Ennek kiküszöbölésére használnak hierarchikus FSM-eket, melyekben minden állapot maga is egy-egy FSM. Bár használják még, de más technikák már kezdik kiszorítani a piacon; köszönhetően annak, hogy a mai játékokban hatalmasra nőhet a száma az állapotoknak, és lehetetlenné válik az átláthatóságuk. Ugyanakkor még mindig előfordulnak FSM-mel ellátott játékok akár a AAA piacon is. Példaként megemlíteném az id Software DOOM 2016-ját, mely a hierarchikus FSM technikát alkalmazta.[10]

Fuzzy State Machine: A Fuzzy State Machine (FuSM) egy kiterjesztése az FSM-nek, mely boolean állapotok helyett fuzzy logikát alkalmaz. Ennek eredményeképp a FuSM nincs bekorlátozva aktív és inaktív állapotokra, de felvehet egy közbenső értéket azok között. Ez azt jelenti, hogy egy időpontban több állapot is lehet aktív. (Pl. egyszerre lehet mozogni és lőni.) Bár némiképp komplikáltabb a felépítése, mint egy átlagos FSM-nek, általánosságban véve kiszámíthatatlanabbá teszi az AI-t. Előnye még, hogy nagyban csökkentheti a state machine komplexitását, mert egy szélesebb körű viselkedés leírása valósítható meg, kevesebb állapotleírással.[10]

Meg kell jegyezni, hogy a FuSM nem minden esetben egyenlő a fuzzy logikával. Legtöbbször FuSM-ként hivatkoznak bármilyen állapotgépre, melyben a fuzzy logika bármilyen része megtalálható. A fentebb leírt state machine-t ugyan nevezhetjük FuSM-nek, de nem feltétlenül található meg benne minden fuzzy logika tulajdonság.

Behavior Tree: A Behavior Tree lett napjaink egyik legnépszerűbb eszköze AI karakterek készítésére. Egyik első példája a Bungie Software 2004-es játékában a Halo 2-ben került implementálása, melyhez részletekbe menő leírás is ké-

szült. Ezután rengeteg játék követte a példáját. A Behavior Tree egyfajta halmaza különböző AI technológiáknak. Erőssége, hogy ötvözi az összes ilyen technika erősségét és könnyű megérteni, akár programozással nem foglalkozó egyéneknek is, köszönhetően vizualitásának. Hasznos technika mint programozási eszköz, de legtöbbször párosul hozzájuk egy grafikus interfész (GUI), hogy szerkeszthessük a fát. Meg kell azonban jegyezni, hogy vannak olyan szituációk, mikor nehéz vele jó megoldást tálalni, és nem mindig a legjobb megoldás a döntéshozatalra.

Sok közös van a Behavior Tree-kben és a hierarchikus állapotgépekben, de az alap építőelemet nem egy állapot adja, hanem egy feladat (task). A taskok alfákba (sub-tree) tömörülnek, így komplexebb tevékenységeket is megvalósítva. Viszont ezek a komplex akciók ugyancsak lebonthatók magasabb szintű viselkedésekre. Ez a felbonthatóság adja az erősségét a Behavior Tree-nek. Az összes task közös interfészen helyezkedik el, és függetlenül egymástól képesek működni, így felépülhet egy hierarchia közöttük, és képesek egymást meghívni, anélkül, hogy tudnák hogyan implementálták a másikat.[11]

Szabály alapú rendszerek: A szabály alapú rendszerek ugyan egy viszonylag réginek mondható technika (1970-es évek), mindmáig nem terjedt el széleskörű használata. Ez annak köszönhető, hogy sok esetben nem elég hatékonyak és nehezen implementálhatók. Főleg azokban az esetekben a legjobb, mikor a karakternek a világot maga körül meg kell értenie és azok szerint cselekednie.

Ezek a rendszerek két részre bonthatók: tartalmaznak egy adatbázist, amely azzal a tudással szolgál, amely az AI számára elérhető, és használnak elágazásokat, mely alapján megtörténnek a cselekvések. Ez első ránézésre hasonlít az állapotgépekhez, de itt egy adatbázis tartalma alapján kerülnek ki a döntések.[11]

Cél orientált viselkedés: A cél orientált viselkedés nem egy újabb technológia, inkább egy irány, mely célokat és szükségleteket vesz alapul. Ez a megközelítés még mindig ritkán használt mai játékokban is, így nem jelenthetünk ki leggyakrabban használt technikát erre az esetre, viszont mivel maga a megközelítés másféle megoldást igényel meg kell említenem.

Az eddigiekben bemutatott módszerek mindegyike reakciókra alapú technika volt: bemenő adatokat biztosítottunk a karakternek, ami aztán azok alapján végrehajtott egy cselekvést. Egy célt, vagy vágyat nem lehetséges implementálni. Lehetséges azonban azt elérni, hogy úgy tűnjön rendelkezik ezekkel. Ezt elérhetjük akármelyik egyszerű döntéshozási móddal. Implementálni tudjuk, hogy ha az AI meglátja a játékost lőjjön, ezzel elérve, hogy úgy tűnjön mint-

ha ez lenne a célja. Ugyanakkor egy olyan játékban mint a The Sims [Maxis Software] ez túl nagy elágazásokhoz vezene, mert ott egy karakter akár több száz döntés közül is választhat. (Az említett játék egy életszimulátor, ami próbálja minél valóságosabban szimulálni a benne lévő karakterek igényeit és vágyait minden pillanatban.) Egy jobb megközelítés lenne, ha a karakternek lehetősége lenne egy bizonyos mennyiségű akció közül választani, és azt kellene megadnia, amelyik legjobban illeszkedik a pillanatny szükségletéhez. Ez a cél orientált viselkedés, mely kifejezetten a karakter belső céljait próbálja kielégíteni.[11]

2.2.2. Tanulás alapú módszerek

A machine learning felhasználása játékokban egy még máig kiaknázatlan terület, mivel a technológia maga is csak az utóbbi években kezdett fellendülni. A precíz felhasználásával sokak szerint ez lehet a játékfejlesztés jövője is, de még nem járunk annál a pontnál, hogy mindent tudunk a lehetőségekről. Elméletben a tanulás alapú mesterséges intelligencia lehet a kulcsa a hihetőbb, sokkal életszerűbb karaktereknek, melyek a környezetük megfigyelésével emberekre szabott élményt és kihívást érhetnek el. Ez lehet a megoldás az újra felhasználható AI-kra is a játékiparban, köszönhetően az utóbbinak. Dolgozatomban Ian Millington és John Funge Artificial Intelligence for games című könyvét vettem alapul a mesterséges intelligenciák tulajdonságainak csoportosításaihoz. Úgy vélem, hogy logikusan bontja csoportokra a technológia részegységeit, különös figyelmet fordítva a videojátékokban sarkalatos pontokra.

Offline és Online tanulás

A tanulási módszereknél megkülönböztetünk offline és online tanulást. Az utóbbinál a tanulás folyamatosan történik, miközben a játékos játszik. Ez az a módszer, amivel az AI dinamikusan a játékoshoz szoktatja magát, így elérve a személyre szabott élményt, kihívást. Ahogy a játékos játszik, úgy a mesterséges intelligencia is egyre jobban tudja majd megjósolni a tetteket, vagy a stratégiát. Ez első hallásra jól hangzik, de ugyanakkor problémákat is felvethet. Fejlesztői szempontból nagyon nehézkessé teheti a játék tesztelését, hiszen nem lesz két ugyanolyan szcenárió. Ez köszönhető annak, hogy minden játékos máshogy fog játszani, de még egyazon tesztelő sem feltétlenül játszik ugyanúgy több alkalommal. Így nem feltétlenül találják meg a rendszer hibáit, ha nem térnek arra az útra, ahol az esetleg előjöhet. [11][12]

A legtöbb esetben, ha a tanulási módszert választják, az offline történik. Ez azt jelenti, hogy a tanulás még a játék kiadása előtt a fejlesztőknél történik, vagy más esetben a pályák között. Ez másfajta input adatokat jelent, hiszen egy teljes játék-

eseményből nyerik ki azokat, így kiszámíthatatlanabb, jobb stratégiákat kifejlesztve. Egyik új példája az offline tanulásnak az Nvidia által kifejlesztett DLSS (Deep Learning Super Sampling) technológia. Ez egy képfeljavítási technika, amely az új RTX GPU-kra jelent meg. Ez egy speciális AI, amit "convolutional autoencodernek" neveztek el. Mivel lényegében gyengébb minőségű képből gyárt jobb minőségűt a számítógépünkön történő render folyamatok nélkül, így kisebb számítási kapacitással játszhatunk szebb képélménnyel. A technológia még jobban működik a 2.0 változatban, ahol már ultra nagy felbontású képeken tanították az algoritmust. Az offline tanulás hátrányai hasonlók az online tanuláséhoz, így a szituáció dönti el melyiket használjuk.[13]

Az implementálásnál hasonlóan kell eljárni, a különbség főleg a bemenő adatokban lesz. Online esetben az adatok a játék által generált valós időben átadottak lesznek, míg offline esetben az adatok eltárolüdnak és később egyszerre kerülnek felhasználásra.

Intra-Behaviour Learning és Inter-Behaviour Learning

Az Intra-Behavior Learning azt jelenti, hogy a tanuló algoritmus egy kis, elemi részét irányítja a karakter viselkedésének. Ezek nem változtatják meg a teljes viselkedést, csak egy pontján befolyásolják azt. Ilyen lehet a pontos lövés megtanulása egy íjjal, ahol a nyíl útja egy fizikai törvények által leírt görbét rajzol. Amennyiben jobb megoldásra vezetne egy teljesen eltérő viselkedés azt már nem tudja megtenni.

Az előző eset ellentéte az Inter-Behaviour Learning. Ebben az esetben az AI képes lenne teljesen másfajta cselekvéseket bevetni a cél érdekében. Pl. ha el akarna kapni egy játékost, és kitalálná, hogy a legjobb módja ennek elbújni egy helyen, ahol biztosan elsétál, meglepheti őt. Ezek a karakterek már szinte teljesen emberiek lennénkek, de sajnos még alig létezik ilyesmi. Egy ilyen megoldásnál a fejlesztőknek csak az atomi viselkedés mintákat kellene implementálni és az AI idővel megtanulna összetetteket, valamint azok helyes használatát. Még kérdéses, hogy ez megérné-e a befektetést egy játékban erőforrások szempontjából. Szemben azzal, hogy tanulás nélkül kódoljuk azokat a cselekvéseket, melyeket előre ismerünk. A mostanában bevett szokás az ezzel való kísérletezésben, hogy az előzőekben említett nem tanulás alapú döntéshozó módszereket lecserélik tanulás alapúakra. Az AI többi része megmarad egyértelműen leprogramozott részekként. Így limitálva a gépi tanulással ellátott részeket, egy hibrid mesterséges intelligencia fog születni, mely megfelelő az adott játék szempontjából. A tesztelés azonban még itt is nehézkes, hiszen ha felvetődik egy probléma nem lehetünk biztosak benne, melyik résztől származik. Ez nemiképp kiküszöbölhető egy moduláris teszteléssel, melyben egyenként tesztelhetjük az AI képességeit, és összeépítés után megvizsgálhatjuk, hogyan működnek

együtt.[11]

2.3. A gépi tanulásról és a mesterséges neurális hálózatokról általánosan

Az áttekinthetőség érdekében röviden taglalnék néhány létfontosságú definíciót a neurális hálózatokról, melyek az általános összképhez szükségesek: Neurális hálózatnak nevezzük azt a hardver vagy szoftver megvalósítású párhuzamos, elosztott működésre képes információfeldolgozó eszközt, amely:

- azonos, vagy hasonló típusú – általában nagyszámú – lokális feldolgozást végző műveleti elem, neuron (processing element, neuron) többnyire rendezett topológiájú, nagymértékben összekapcsolt rendszeréből áll,
- rendelkezik tanulási algoritmussal (learning algorithm), mely általában minta alapján való tanulást jelent, és amely az információfeldolgozás módját határozza meg,
- rendelkezik a megtanult információ felhasználását lehetővé tevő információ előhívási, vagy röviden előhívási algoritmussal[14]

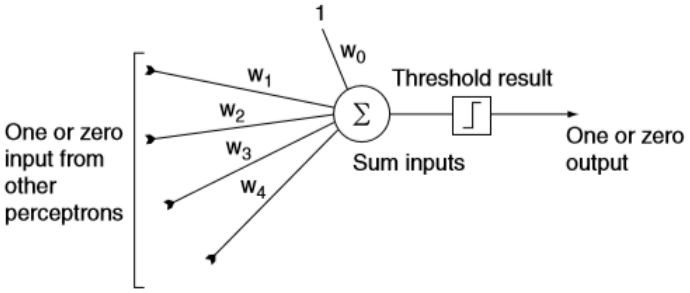
A neurális hálózatoknak, azon belül is a „deep learning” argoritmusoknak több típusát megkülönböztetjük. Ezeket eredeti nevükön említeném, mivel a legtöbb szakirodalomban is így hivatkoznak rájuk. Ezek a teljesség igénye nélkül a "Convolutional Neural Network" (CNN), és a "Recurrent Neural Network" (RNN). Megemlíteném itt a megerősítéses tanulást (Reinforcement Learning) is, mely a szűkebb értelenben véve nem egy fajtája a neurális hálózatoknak, de felhasználását tekintve egy olyan gépi tanulási módszer, mely merőben eltér az előzőektől, és gyakran párosítják neurális hálózatokkal, így létrehozva egy mély megerősítéses tanulási algoritmust.

A CNN-t általában képfelismeréssel kapcsolatos problémáknál használják. Jó eredménnyel implementálják pl. az arcfelismerésnél. Az RNN típust alkalmazzák pl. beszéd, illetve kézírás felismerésénél. Ennek fő jellemzője, hogy visszacsatolás található benne, azaz a hálózatnak memóriája van. Ezeknek jellemzője, hogy nagy adathalmaz szükséges a tanításukhoz. A harmadik említett típus a megerősítéses tanulás. Ennek lényege, hogy nincs szükségünk nagy tanítási adatbázisra, hanem jóság szerint tanítjuk az algoritmust. Ennek részét képezi egy ágens, mely futáskor a lehetséges utat keresi a célhoz; valamint az előre meghatározott jóságok, melyek aszerint változnak, hogy épp a jó, vagy a rossz irányba tart az ágens. Így sokszori lefutásra megtanulja, melyik a jó irány, azaz melyik irányban érheti el a legtöbb jósággal rendelkező pontokat. Felhasználása sokrétű lehet, pl. út keresés.

2.3.1. Mesterséges neurális hálózatok a videojátékokban

A fentebb említett lista csak egy kis része a neurális hálók típusainak, ebből kifolyólag csak azokkal az esetekkel foglalkoznék, amelyeket szokás felhasználni játékoknál. Egy gyűjtőfogalomról lévén szó megállapíthatjuk, hogy csak egy elenyésző részük alkalmas játékbeli mesterséges intelligenciák megteremtésére. Sokan kísérleteztek már számtalan módszerrel, de a legjobban elterjedt alfaja, mely működőképes eredménnyel szolgált játékoknál, az úgynevezett többrétegű perceptron (multi-layer perceptron).

A neurális hálózatok alapját a nagyszámú és egyszerűnek mondható csomópontok adják, melyek mindegyike ugyanazt az algoritmust futtatja. Ezeket mesterséges neuronoknak hívjuk.Minden egyes neuron egy bizonyos mennyiségű neuronnal kommunikál, melynek száma a hálózat topológiájától függ.[15]



2.5. ábra. A perceptron működése

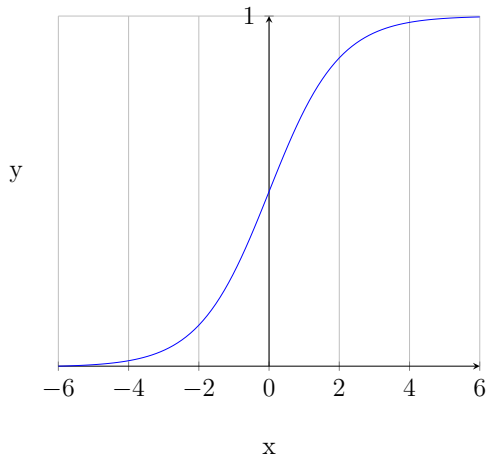
A perceptronok réteges formát alkotnak, melynek minden neuronja kapcsolódik az előző és következő rétegben elhelyezkedő összes neuronhoz. Egy többrétegű perceptron az előrecsatolt (feedforward) hálózatok közé tartozik, mert a neuronjai az azt megelőző rétegből nyernek adatokat és továbbadják a következő réteg neuronjainak. A legbaloldalibb réteg az input réteg, melyet manuálisan kell beállítanunk, és a legjobboldalibb réteg az output réteg, mely a már általunk felhasználható adatokat szolgáltatja. A neuronok mindegyike rendelkezik egy állapottal, melyet egy algoritmus által számít ki a hálózat. Egy többrétegű perceptronnál ez az állapot mindig átadódik a következő rétegnek. Minden egyes input adathoz tartozik egy súly és minden réteghez egy "bias" súly. A bias egy bemenetnek tekinthető, melynek bemeneti értéke állandó. A teljes réteget szummázva átadjuk egy aktivációs függvénynek, mely átalakítja azt. Amennyiben az így kapott érték kisebb mint nulla, úgy a neuron kikapcsolt állapotba kerül (értéke 0 lesz), ha nagyobb mint nulla, akkor bekapcsolt, azaz értéke egy lesz.[11][16]

Aktivációs függvények

Az aktivációs függvény egy adott neuron bemeneti értéket kimeneti értékké változtatja.Többnyire ez azt jelenti, hogy a hálózat által feldolgozható formába alakítja. Ezekből többfélét használhatunk, melyek mindegyike különböző előnyökkel és hátrányokkal rendelkezhet az adott feladattól függően. Érdemes megemlíteni közülük a szigmoid és ReLU (Rectified Linear Unit) függvényt, melyek a leggyakrabban használtak közülük. Ezek a függvények árnyalják a kimenetet. A szigmoid függvény(1), ha bemenetnek egy nulla közeli számot kap, árnyaltabb kimenetet kap nulla és egy között, míg ha ezektől távol áll, úgy működik mint a fent leírt. A függvény egyenlete

$$f(x) = \frac{1}{1 + e^{-hx}}, \tag{1}$$

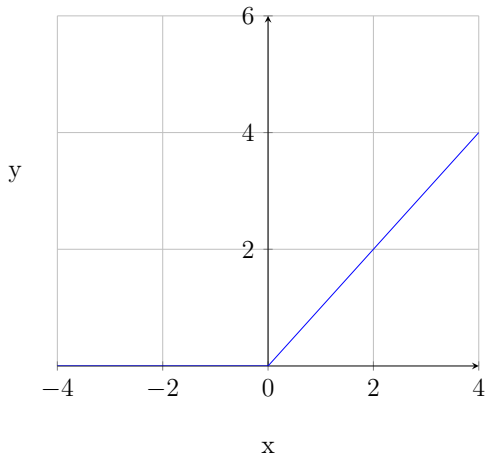
ahol h egy változtatható paraméter, mellyel a függvény alakját változtathatjuk. Minél nagyobb a h, annál meredekebb a függvény.[11]



2.6. ábra. A szigmoid függvény

A másik leggyakrabban használt aktivációs függvény a ReLU(2), melynek egyenlete:

$$f(x) = \begin{cases} 0 & \text{, ha } x < 0 \\ x & \text{, ha } x \geq 0 \end{cases} \tag{2}$$



2.7. ábra. A ReLU függvény

Ez a függvény nullát ad vissza, ha az eredmény kisebb mint nulla, és a bemenet értékét, ha nagyobb. Újabban sokszor inkább a ReLU-t választják a szigmoid függvény helyett neurális hálózatok fejlesztésekor, többek között azért is, mert felgyorsítja annak tanulását. Ez annak köszönhető, hogy számítása egyszerű a számítógépek számára.

Érdemes még megemlíteni ennek egy változatát, a "Leaky ReLU" függvényt. Ez hasonló a ReLU-hoz, viszont kiküszöböli azt, hogy nullától kisebb számokra nullát ad vissza. Ebben az esetben, a negatív számokat elosztja egy nagy számmal, ami így közel kerül a nullához, de nem lesz az.

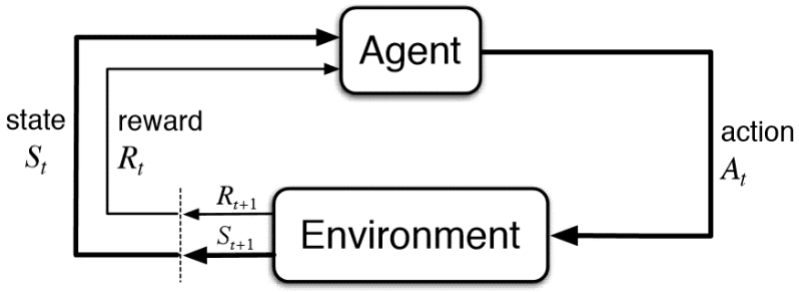
Visszaterjesztés

A tanulás elvégzésének leggyakoribb módja a visszaterjesztés (Backpropagation). Ekkor az utolsó réteg kimenetén lévő eredményt megvizsgálva, egy hibafüggvény (Loss function) használatával megtudhatjuk a hiba mértékét. Ennek eredményeit felhasználva visszafelé is végighaladunk a rétegeken változtatva a súlyokat. Visszaterjesztésnél ez tulajdonképpen függvényminimumok, vagy maximumok keresését jelenti, melyet az aktivációs függvények deriváltjaival tudunk keresni. Ez a függvény meredekségét adja vissza adott pontban, így megjelölve melyik irányba haladjunk a jobb eredmény elérése érdekében, ezzel tanítva az AI-t. Ez egyben azt is jelenti, hogy egy neurális hálózat nem ad biztosan optimális megoldást, a függvény lokális minimumai és maximumai miatt. Léteznek technikák ezen eset elkerülésére, de biztosak nem lehetünk abban, hogy a kimenet a keresett globális szélsőérték.

2.4. Megerősítéses tanulás a játékokban

A megerősítéses tanulás tulajdonképpen tapasztalat alapú tanuló algoritmus. Ennek okán bizakodva tekintenek rá lehetséges módszerként a játékiparban. Mivel nem szükséges hozzá az a nagymennyiségű tanuló adat, ezért könnyebben felhasználható játékos intelligenciák készítéséhez. Az ilyen algoritmusok három részre oszthatók:

- egy felfedező stratégia, mely különböző akciókat hajt végre a játékban,
- egy megerősítési funkció, mely visszajelzést ad arról, mekkora mértékben jó az adott akció,
- és egy tanulási szabály, mely összeköti az előző kettőt.



2.8. ábra. A megerősítéses tanulás módszerei

A következőkben a megerősítéses tanulás módszereiről lesz szó. A legalapvetőbb Q-tanulással kezdenék, mert minden további módszernek ez adja alapkövét. Az általam használt algoritmus ugyan, egy mai szemmel nézve is sokat használt és fejlett módszer, azonban, szükségesnek éreztem, hogy képet kapjak az elvekről, amely alapján felépítették azt.

2.4.1. Q-tanulás

A Q-tanulás (quality) a legegyszerűbb megerősítéses tanulási módszer, mely egy állapotgépként tekint a világra. Az algoritmus minden egyes időpillanatban egy bizonyos állapotban van, mely állapotban kódolva van az összes releváns adat az ágensről és annak környezetéről. Minden egyes állapothoz és cselekvéshez tartozik egy jutalom (reward), melyet a karakter megkaphat lépésenként, vagy a cselekvés végén. A Q-tanulás olyan eredményt próbál elérni, amiben a jutalmakat maximalizálja. Az optimális megoldás elérése érdekében az algoritmusnak tennie kell "explore" lépéseket, melyek véletlenszerűek, és "exploit" lépéseket, melyek a megtanult mintát követik. Ezek aránya feladatonként eltérő lehet.

A Q-tanulás nem egy modell alapú módszer. Ez azt jelenti, hogy nem próbálja felépíteni a világot maga körül egy bizonyos modellben, hanem minden lehetséges változóhoz egy állapotot rendel. Egy játékban hatása lehet a játékosoknak, vagy más karakterekeknek az állapotokra.Ez azt jelenti, hogy azonos állapotoknál nem feltétlenül ugyanaz lesz a döntés a következő állapotra vonatkozóan. Ez a fajta rugalmasság az egyik előnye a megerősítéses tanulási algoritmusoknak.[17]

A tanulás folyamata

A tanulás folyamatában megszerzett tudást egy úgynevezett Q-táblában tároljuk. Az algoritmus minden egyes állapot-akció párra, amit véghezvitt, tárol egy Q-értéket. Ez azt reprezentálja mennyire jó volt az adott döntés, az adott állapotban. A kezdő állapotot, a megtett akciót, a megerősítési értéket és az eredmény állapotot együttesen az "experience tuple-nak" nevezik (s, a, r, s'). Ez két részre bontható: a kezdő állapot és a megtett akció megkeresi a hozzájuk tartozó Q-értéket a táblában. A megerősítési érték és az eredmény állapot pedig frissíti a Q-értéket, az alapján, hogy mennyire volt jó az adott akció és mennyire lesz jó a következő állapotban. Ezt frissítést a Q-tanulási szabály végzi:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma * max(Q(s', a'))), \tag{3}$$

ahol α az úgynevezett "learning rate", és γ a "discount rate". Ezek az algoritmus paramétereiként megadhatók. Az előbbivel az állítható, hogy mennyire gyakorol erős hatást az érkező adat az eltárolt adatra, míg az utóbbi kontrollálja, hogy a végrehajtott akció Q-értéke mennyire fontos az eredmény állapot elérésének Q-értékéhez képest. Azaz, minél magasabb ez a szám, annál jobb az algoritmus a hossabb távú stratégiák kifejlesztésében, azonban annál több időbe és erőforrásba kerül.

Mély Q-tanulás

A mély Q-tanulásnál a Q-értéket egy táblázat helyett egy neurális háló tanulja. Ez a megoldás sokkal kevesebb memóriát emészthet fel, és nagy előnye, hogy a sima Q-tanulással ellentétben a neurális hálók képesek általánosítani. Míg egy Q-tanulási algoritmus nem képes megmondani, hogy egy adott szituációban is elfogadható egy másik nagyban hasonló esetben megtanult minta, úgy erre egy neurális háló már képes lehet.

Nem várhatjuk azonban ezektől a megoldásoktól, hogy ugyanazzal a Q-értékkel térjenek vissza, mint ami a bemenetük volt. A hálózat tanulása során ezek az értékek folyamatosan változni fognak, így lehetséges, hogy a karakter viselkedése nem lesz optimálisnak mondható.[11]

2.4.2. Proximal Policy Optimization (PPO)

A PPO az OpenAI által biztosított deep reinforcement learning metódus. A tesztek azt mutatták ki, hogy hasonló, vagy jobb eredményt hozott, mint az addigi legfejlettebb ilyen algoritmusok, míg implementálni és finomhangolni sokkal egyszerűbb. Többek között ezen okok miatt az egyik legelterjedtebb megerősítéses tanulási mód lett, kiszorítva a fentebb taglalt mély Q-tanulást. Számomra azért bír nagy jelentőséggel, mert az általam is használt Unity ML-Agents-ben a választható beépített módszerek közül ez az elsődleges. Ugyan magára az algoritmusra a használatakor tekinthetünk fekete dobozként, fontos lesz ismerni bizonyos paramétereket, anelyek a hatékony használatához szükségesek.

2.4.3. Soft Actor Critic (SAC)

Amikor a deep reinforcement learning algoritmusokról beszélünk, meg kell említenünk a SAC-ot is. Az algoritmust a UC Berkeley és a Google fejlesztette ki és főleg robotokkal való kísérletezéshez használják. Megemlítése az én szempontomból azért szükséges, mert ez a másik használható módszer az ML-agents Toolkitben. Mivel tulajdonságaiban a másik módszert találtam közelebbinek az általam definiált feladathoz, így ezt tovább nem taglalnám.

2.5. A Unity ML-Agentsről bővebben

A mesterséges intelligenciát a föntebb leírt módon a Unity ML-Agents Toolkitben szeretném elkészíteni, így ebben a fejezetben szeretném azt bővebben bemutatni. Ennek megértéséhez szükséges volt az előzőekben ismertetett módszerek és általános szabályok megismerése, melyeknek egy része az ML-Agents alapjául szolgál és annak hatékony használatához elengedhetetlen.

Az ML-Agents segítségével NPC-k (azaz nem játékos karakterek) viselkedését tanulás útján állítjuk be. Ehhez alapvetően három entitást kell definiálnunk a játék minden egyes időpillanatában:

Megfigyelések (Observations): Az első entitás, az amit az ágens érzékel a környezetéről. Ezek lehetnek numerikus vagy vizuális megfigyelések. Numerikus esetben ezek az ágens szempontjából megfigyelhető környezeti értékek. Ezzel ellentétben a vizuális megfigyelések azokra a képekre vonatkoznak amik az ágensre kapcsolt kamerákból generálódnak az adott időpillanatban, és a látást reprezentálják. Itt fontos kiemelni, hogy ez nem ugyanaz mint maga a game state, azaz a teljes játék állapota. Ezen megfigyelések alatt csak az ágens által érzékelhető környezetet értjük, azaz, az AI nem érzékelheti pl. a mellette lévő szobában álló játékost.

Akciók (Actions): Azok a cselekvések, melyeket az ágens tenni képes. Ez lehet folytonos vagy diszkrét. Azt, hogy melyikre van szükségünk az ágens, és a környezete felépítése dönti el. Pl. egy sakktáblán diszkrét akciókat tudunk végrehajtani, míg egy bonyolultabb pályán, ahol az NPC folytonosan tud mozogni, használhatunk két folytonos cselekvés mintát: egy az iránynak, és egy a sebességnek.

Jutalom (Reward signals): Egy skalár érték, amely azt reprezentálja milyen jól teljesít az ágens. Ez a megerősítéses tanulás egyik legfontosabb aspektusa. Amennyiben az ágens a kívánt módon cselekszik pozitív jutalmat adunk neki, amennyiben azzal ellentétesen akkor negatív jutalmat osztunk ki.

Ezen három entitás definiálása után elkezdhetjük tanítani az ágenst. Az ML-Agentsben ez a szcenáriók sokszori leszimulálásával történik, ahol az ágens ideális esetben elér egy olyan viselkedési mintát, ahol minden egyes megfigyelésre megtanulja mi az optimális cselekvés úgy, hogy a maximális jutalmat próbálja elérni.

Ezeken felül az ML-Agents Toolkitben megkülönböztetünk öt kulcs komponenst:

Tanulási környezet (Learning Environment): Tartalmazza a Unity scene-t és az összes karaktert. A scene adja azt a környezetet, amit az ágensek megfigyelhetnek, tanulhatnak, illetve amivel interaktálhatnak. Ezt mi építhetjük fel, hogy azt szolgálja, amit célul tűztünk ki az ágenseknek. Az ML-Agents Toolkit tartalmaz egy ML-Agents Unity SDK-t, amely lehetővé teszi, hogy egy szimpla Unity scene-t tanulási környezetté alakítsunk úgy, hogy definiáljuk az ágenseket és a viselkedési mintákat.

Python Low-Level API: Az alacsonyszintű Python interfészt biztosítja, amely segítségével interaktálhatunk és manipulálhatjuk a tanulási környezetet. A Python API nem része a Unity motornak, hanem azon kívül létezik és a Communicator-on keresztül kommunikál azzal.

Külső Kommunikátor (External Communicator): A fentebb említett kommunikátor, amely hídként szolgál a tanulási környezet és a Python Low-Level API között.

Python Trainer-ek: Tartalmazzák azokat a gépi tanulási algoritmusokat, amelyek lehetővé teszik, hogy megkezdjük az ágens tanítását. Az algoritmusok Python nyelven íródtak és az "mlagents" bővítmény részei.

Gym Wrapper: Egy, az OpenAI által biztosított út, amellyel kapcsolatba léphetünk a szimulációs környezettel a Pythonon belül, végsősoron akkor szükséges,

ha saját algoritmusokat szeretnénk készíteni. Használata elkerülhető volt, mivel a beépített és használható algoritmusok a lehető legfejlettebbek és használatukkal optimálisabb eredményt érhetek el és időt is megtakarítok. Ezek miatt mélyebben nem

A tanítási környezet két olyan Unity komponenst biztosít, mely segítségével rendezhetjük a scene-ünket. Ezek a következők:

Agents: Akkor alakíthatjuk a Unity GameObject-et ágenssé, ha Agent-é tesszük. Ezúton kezelhetjük a megfigyeléseket, cselekvéseket és jutalmazást. Minden Agent egy Behavior-höz, azaz egy viselkedési mintához van kötve.

Behavior: Ezzel definiálhatjuk az Agent specifikus viselkedésmintáit. Pl. hogy hány akciót engedünk meg neki. Minden Behavior egy egyedülálló névvel van ellátva, amellyel azonosíthatjuk. Működése hasonló egy függvényéhez amely bemenetként megfigyeléseket és jutalmakat kap és cselekvésekkel tér vissza. Három fajtája lehet, ezek a tanulási (Learning), heurisztikus (Heuristic) és következtető (Inference). A tanulási viselkedésminta az, amelyet még nem definiáltunk és tanítottunk, de fogunk. A heurisztikus egy olyan szabályrendszer, amit előre kódban definiáltunk. A következtető minta pedig, egy olyan Behavior, amely tartalmaz egy, már tanított neurális hálót. (Azután, hogy a Tanulási minta tanítása megtörtént, már következtető viselkedési mintaként hivatkozunk rá.)

2.6. Létező példák

2.6.1. AI-k, melyek megtanultak játszani

A gépi tanulás vegyítve a videojátékokkal máig nem gyakori példa. Erre magyarázatként szolgálnak az előzőekben említett hátrányai, nevezetesen, hogy nem jól tesztelhető, nem feltétlenül ad optimális megoldást, és erőforrás igényes. Mondhatjuk, hogy a cégeknek nem éri meg kísérletezni ezzel, és inkább maradnak a jól bevált és működő módszereknél. Ahol azonban előfordulnak kísérletek, az nem annyira a felhasználói élmény javítására való törekvések miatt történik, mint inkább tudományos fejlődési érdekekből. Nevezetesen inkább fejlesztenek olyan mesterséges intelligenciákat, melyek megtanulnak egy játékot játszani, és esetleg bevetik azt az adott játékban profik ellen, minthogy a játékos piac számára valószerűbb élvezetesebb élményt nyújtsanak. Ezeket csak röviden megemlíteném dolgozatomban, mert eltérnek az általam lefektetett céloktól.

Ezek általában komplex stratégiai játékokat tanulnak meg játszani, és igen jó eredménnyel szerepeltek. Tudni kell róluk, hogy hatalmas mennyiségű tanulás áll mögöttük, több szuperszámítógéppel adatok millióihoz jutottak hozzá, viszonylag rövid

idő alatt. Egyik leghíresebb ilyen próbálkozás a Go nevű táblajáték megtanulása, és profi szintre emelése a Google, DeepMind nevű fejlesztése által. Mindössze Lee Sedol, professzionális játékos volt képes legyőzni egyszer az öt meccsükből, ezzel Ő lett az egyetlen aki valaha legyőzte azt. Amit érdemes megjegyezni, hogy program, tanulása egyik fázisában saját maga ellen játszott, megerősítéses tanulási technikával. Ugyanígy a DeepMind profi játékosokat győzött le az ismert stratégiai játékban, a StarCraft 2-ben.[18]

Meg kell említeni az OpenAI mesterséges intelligenciát, mely a világon elsőként világbajnok csapatot győzött le többször a DOTA 2-ben Ez egy komplex MOBA (Multiplayer Online Battle Arena) játék melyben két ötfős csapat mérkőzik meg egymással.[19][20]

2.9. ábra. Screenshot a Black and White 2 című játékból

2.6.2. Black and White és Black and White 2

A Black and White egy, a Lionhead stúdió által fejlesztett, és az Electronic Arts által kiadott stratégiai játék, mely 2001-ben jelent meg. A játék alapja, hogy a játékos egy

istent játszik. Egy szigeten, melyet különböző kultúrák népesítenek be, a játékosnak először választania kell egy "Creature"-t, melyet a játék elején felügyelni, nevelnie kell. A játék lényege, hogy minél több szigetlakó tisztelje istenként a játékost, ami több módon elérhető. A szigeten van azonban egy másik, ellenséges isten is, mely arra törekszik, hogy mindent elpusztítson amit a játékos felépített. A tény, ami különlegessé teszi a játékot, az az AI, ami a Creature-t irányítja. A cél az volt, hogy egy hasznos teremtményt hozzanak létre, aminek emberibb a viselkedése. A létrejöttéhez három féle metódust vegyítettek:a szabály alapú rendszert, döntésfát és ami a legfontosabb egy neurális hálózatot, pontosabban egy perceptront. Ezek mindegyike a lény egyes tulajdonságaiért, vagy képességeiért felel. A neurális háló segíti hozzá az emberibb viselkedéshez, ez felel a vágyaiért. Ezt egy megerősítéses tanulási módszerrel ötvözték, mely jutalom és büntetés alapú. A Creature a játékos megfigyelésével, online módon tanul. Ezáltal a játék folyamán egyre inkább úgy teszi majd a dolgát, ahogy azt a játékos megkívánná manuálisan. Azaz pl, ha a játékos egy bizonyos fajta falusit támad meg általában először, úgy egy idő után a Creature útmutatás nélkül is azt fogja először támadni. Ez egy nagy lépés volt, viszont hátránynak felvethető, hogy a számítási kapacitás limitáltságának köszönhetően, néhány másik tulajdonságából visszább kellett venni.A Creature képtelen volt a hosszabb távú tervezésre, mivel egy egyszerű táblázatot használt a játék. Így a lény csak az azonnali vágyaira tudta megkeresni a helyes viselkedési mintát. A második részben hasonlóan jártak el, szintén gépi tanulást is alkalmazva a játékban.[21]

2.7. Összegzés

A játékomat Unity Game Engine segítségével fogom létrehozni, mely egy nagyobb, mindenki számára elérhető fejlesztőkörnyezet. Az előre beépített funkciói nagyban meggyorsítják és megkönnyítik a fejlesztést, így lehetővé téve, hogy egy személyben is lehetséges egy 3D videojátékot létrehozni belátható időn belül. Ebben nagy segítségemre lesznek az elérhető assetek, melyek közül is fontos kiemelnem a grafikai csomagokat. Ezekben modellek találhatóak, melyek moduláris formában érhetők el. Közülük is, többek közt a cégen belül legyártott Snaps csomagokat fogom alkalmazni. Ezen túl, a ProBuilder és ProGrids kiegészítések segítségével könnyen tudok majd prototípusokat készíteni, melyekben kialakíthatom a pályastruktúrát és tesztelhetem a meglévő funkciókat.

A jobb grafikai hatások érdekében használom az előre beépített technikákat, úgy mint a fények és Post-Processing Stack. A Unity egy magas szintű verzióját biztosítja ezen szimulációknak, így a fényhatások egy magas minőségű változata érhető el könnyedén.

A mesterséges intelligencia megalkotásánal figyelembe kellett vennem több szempon-

tot, úgy mint a tanulási adatok hiányát. Ez ellehetetleníti az olyan neurális hálók használatát, melyek minden tudásukat előre megadott adatokból nyerik. Számomra a legjobb megoldás a megerősítéses tanulás egy formája. Ezek közül az ML-Agents által is biztosított PPO algoritmust fogom használni, mely nagy segítségemre lesz az ágens elkészítésében és tanításában. A csomag Python alapú, így a második választott nyelv amiben a kódokat készítenem kell ez utóbbi lesz. Ugyan magas szintű Python tudásra nincs szükség az említett csomag használatakor, jó ha nagyvonalakban értjük, hogyan van megírva, illetve kisebb változtatásoknál szükséges lehet ennek megfelelő ismerete.

A neurális háló elkészítésénél nincsen előre lefektetett útvonal, mit kellene egy adott esetben használni. Ennek tudatában az eredmény fügvményében változtatnom kell a paramétereit, hogy a lehető legoptimálisabb megoldást hozhassam ki belőle. A videojátékokban nem cél, hogy legyőzhetetlen ellenfelet hozzunk létre, így a tanulási algoritmusok szélesebb körű kihasználását ebben az iparágban, az intuitív, illetve vágyakkal összefüggő szituációkban lehet keresni. Ebbe beletartozik a játékélmény irányítása is, mikor az AI nem legyőzni akarja a játékost, hanem az előre lefektetett szabályok szerint "jobbá" tenni az élményt. Ezt észben tartva egy olyan AI-t szeretnék előállítani, mely inkább a felsorolt kategóriákban jeleskedik, mint a játékos vesztét akarná.

3. Rendszerterv

3.1. Game Design

A játék egy belső nézetes kalandjáték, melyben a játékos egyetlen karaktert irányít, és célja, hogy tárgyak felvétele után eljusson egy bizonyos pontba és megnyerje a játékot. Ezt különböző akadályok nehezítik, melyek hiányzó elemek képében reprezentálódnak. A játék közben a győzelem elérését egy ellenfél próbálja megakadályozni. Az ellenfélnek különböző lehetőségei vannak, hogy elérje a célját. Tulajdonképpen meg kell keresnie a játékost.

A pályát elhagyatott helyek adják. A játékosnak lehetősége van objektumok felhasználásával elbújni, mely elrejtheti az ellenfél szeme elől, továbbá felvenni a kellő elemeket.

3.1.1. Modulok

A játék különböző nagyobb részeit modulokra bontottam, melyek a fejlesztési ciklusban elkülönülnek:

UI: A játéknak rendelkeznie kell egy főmenüvel, ahonnak elérhető az indítás, és a kilépés. A játékon belüli user interface - egy egyszerű letisztult verzió, mely nem zavarja a játékost - a képernyő tetején reprezentálódik.

Játékos: Belső nézetes, azaz egy objektum melyre a kamera rá van erősítve. Az egér mozgatásával a játékos körülnézhet mind a két tengelyen. A harmadik tengelyen lévő mozgás a "w,a,s,d" kombinációval használható alapbeállításon.

Ellenfél: Egy ML-Agent mely a pályán mozog és sajat céljait próbálja elérni. Ezek prioritások szerint vannak beosztva, és egy stratégiát kell kialakítania, hogy megtalálja a játékost. A cél egy olyan AI, mely képes kialakítani egy stratégiát, és kellő kihívásként szolgál a játékosnak.

Pálya: Elhagyatott belső területek, melyek a játék folyamán szabadon bejárhatóak. Előfordulnak rajta nem mozgatható objektumok, melyek a pálya részét képezik.

Objektumok: Ide sorolnám a mozgatható objektumokat és a tárgyakat, melyek felvehetők.

3.2. Opcionális lehetőségek:

A fejlesztési időtől függően a következőket opcionálisnak jelöltem. Ezek jobbá tehetnék a játékélményt, de nem feltétlenül szükségesek hozzá:

- Elemlámpa és felszedhető elemek, melyek a sötét részeknél bekapcsolhatók
- Futás implementáció
- Mentés/Betöltés

3.3. A program fejlesztése, időre lebontva

Tevékenység	1.	2.	3.	4.	5.	6.
Design						
Pályatervezés						
Karakter						
Objektumok						
Ellenfél						
Hangok						
Tesztelés						

3.1. ábra. A játékfejlesztés hónapjainak beosztása

4. Implementáció

4.1. Történet és kontextus

Mielőtt a tényleges implementációt elkezdem fontos, hogy azt amit elkészítek, kontextusba helyezzem. Egy tényleges játékszoftver elkészítéséhez ugyan több ember és sok esetben évek szükségesek, de fontosnak tartottam, hogy én is bevezessek egy egyszerű történetet, mely után a kész program egy egységes egészet alkothat. Ezen kívül fontos szempont számomra, hogy a dolgozat keretein belül és az AI fejlesztésén kívül, minél többet felhasználhassak a játékfejlesztés aspektusaiból.

A dolgozat egyik alapvető célja, hogy bemutasson egyet azon lehetőségek közül, melyekkel egy játék, az AI révén kiszámíthatatlanabbá válhat. Emiatt egy olyan műfajt kellett választanom, ahol a hagyományos programoknál az egyik legszembetűnőbb tendencia, hogy monotánná, kiismerhetővé válik. A műfaj kiválasztásánál a másik fontos tényező az idő és emberi erőforrások korlátoltságából adódott. Egy olyan típusú játékot kellett készítenem, ahol annak ellére, hogy 3D-s szoftverről van szó, mégis elkészül időre, mert relatíve kevés benne az intrakció és a játéktechnikai elem. Azaz, egyetlen ember számára is abszolválható. Ezen okoknál fogva egy túlélő-horror kategóriájú játékra esett a választásom, ahol mindössze - ahogy a neve is mutatja - a túlélés a feladat. Ezeket a típusú játékokat visszavezethetjük egy különböző történettel és hozzáadott elemekkel bővített bújocskára.

Most, hogy megállapítottuk a műfajt beszélnék egy kicsit a kontextusról. Ezen játékok többnyire a bezártságra és elhagyatottságra alapoznak, így olyan környezetekbe helyezik őket, mely szinte üres, csöndes és klausztrofób. Az én esetemben a választás egy jövőbeli, elhagyatott űrállomásra esett. Ennek az egyik oka, hogy a lehetőségeim

korlátozottak a grafikai asset-ek miatt. Ezek ugyan sokrétűek, minőségben és árban nagyon eltérőek. Mindezeket figyelembe véve egy olyan csomagot választottam, mely moduláris blokkokból áll, és sci-fi témájú. Lehetővé téve ezzel, hogy a kész pálya egy űrállomásra hasonlítson. Ez megfelel a fentebb említett kritériumoknak.

Természetesen alapesetben a történet hamarabb megszületik, mint a pályák, és azokat rendelik alá a történetnek. Az én esetemben kompromisszumokat kellett kötnöm az említett korlátok miatt, így a sorrend megfordult. Megszületett először a pálya ötlete, mely az előre elkészített űrállomás, majd ezután tudtam azt kontextusba helyezni. A történet célt ad a játéknak, még ha egyszerű is. Emiatt mindössze annyit kell tudni róla, hogy a karakter felébred egy üresnek tűnő űrállomáson, ahol nem sokára kiderül, hogy mégsincs egyedül és ki van szolgáltatva valaminek, ami ott kószál. Ezt a lényt elkerülve el kell hagynia az állomást egy mentőkapszulában, melynek indítórendszere azonban elromlott. Karakterünkkel meg kell próbálnunk összeszedni a szükséges alkatrészeket, amelyek az állomáson szét vannak szórva. Ezek összegyűjtése után megjavíthatjuk a kioldószerkezetet, hogy el tudjunk menekülni.

4.2. Az AI létrehozása

Ebben a fejezetben a mesterséges intelligencia létrehozásának folyamatát szeretném ismertetni. Kitérek a tervekre, az adódó nehézségekre, és magára a létrehozás folyamatára a kezdetektől.

4.2.1. Az AI tervezése

Elsőként specifikálnom kellett az AI-t. Készítettem egy alapszintű követelményt, amit elvárhatunk a kész verziótól. Azonban a gépi tanulás egyik sajátossága, hogy sok paraméter változhat az idő során, mert nem tudjuk pontosan, hogyan fog reagálni bizonyos helyzetekre, vagy milyen megoldásokra jut. Ezek a megoldások lehetnek jók, de egy játékprogramnál figyelembe kell venni azt is, hogy ez az adott optimális megoldás jól áll-e egy szórakoztató szoftvernek.

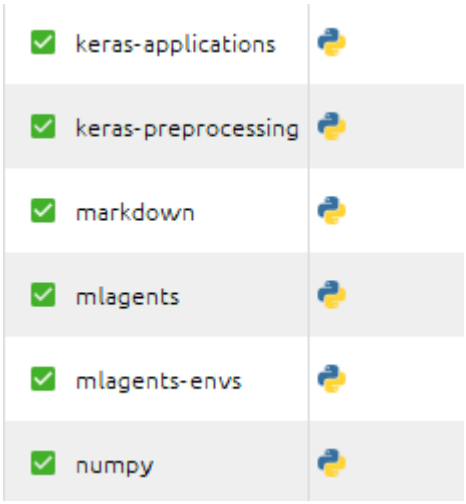
Nagyvonalakban a tanított AI-tól a következőket vártam el:

- keresse a játékost az egész játéktérben, és ha megtalálja próbálja meg elkapni
- nagyjából "emberien" mozogjon

Ez ugyan elsőre nem tűnik soknak, azonban a végrehajtásból kiderül, hogy a számítógép elsősorban nem emberi módszerre törekszik, így ki kell kényszeríteni azt.

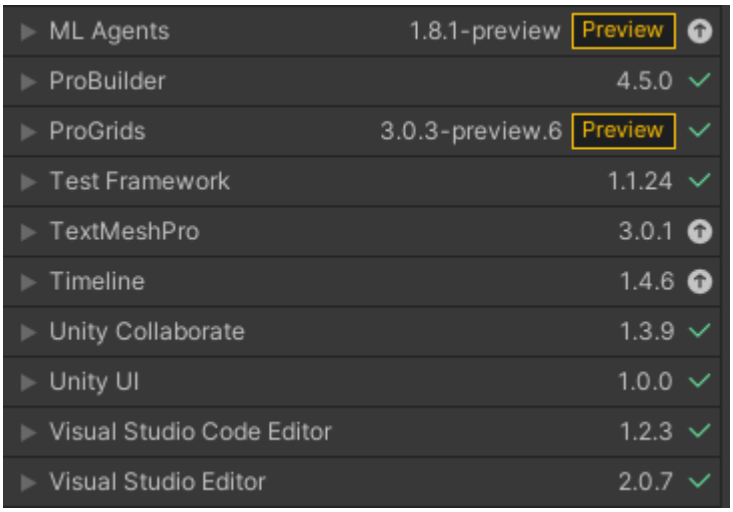
4.2.2. A Unity ML-Agents felhasználása

Fentebb már ismertettem az ML-Agents plugint, itt csak annak saját programomban való használatát emelném ki. A szoftveremben a PPO algoritmussal dolgoztam. A bővítmény használatához szükség van még egy Python környezetre, amiben futtathatjuk azt. Az én választásom az Anaconda disztribúcióra esett, mert ezzel könnyen kezelhető virtuális környezeteket teremthetünk a Pythonnak, és ezeket egymástól teljesen elkülönítve használhatjuk. Amire nekem szükségem volt, az mindössze egy olyan virtuális környezet, mely azokat a python csomagokat tartalmazza, amelyeket ehhez a programhoz használok. Ugyan összesen hatvan darab csomagot kellett importálnom, de ezek többsége a dependenciák miatt történt.



4.1. ábra. Fontosabb Python csomagok

A következő lépés ezt összekötni a tanulási környezettel, ami jelen esetben a Unity scene volt. Ezt egy ML-Agents bővítmény telepítésével érhetjük el a Unity-n belül.

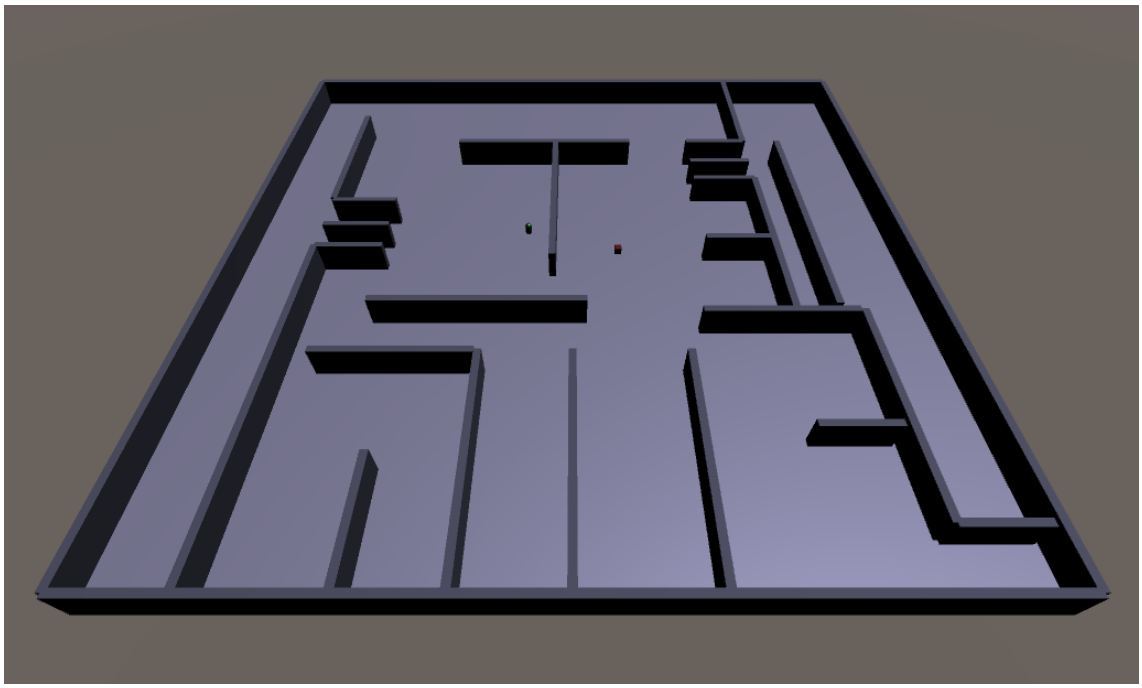


4.2. ábra. Telepített Unity package-ek

A PPO algoritmus paramétereit .yaml fájlokból nyeri, így ezeket kézzel kell beállítani. Ez a folyamat rendkívül iteratív és időigényes, hiszem nem tudhatjuk előre az adott paraméterekkel milyen eredményt kapunk. Ráadásul azokat többször változtatni szükséges, hogy egy optimumot érhessünk el. A tanítás egy ML-Agentsbe beépített mlagents-learn paranccsal indítható el, és ezután a konzol ablakban, illetve a tensorboardban is figyelhetjük, merre halad.

4.2.3. Tanulási folyamatok és paraméterek

Az ágens tanulását két részre osztottam. Először azt kellett kideríteném, hogy az ötletem működőképes-e, azaz képes megkeresni a játékost. Ezt tesztelendő építettem egy kisebb tanítási teret, amelyben csak az alap funkcióit kell ellátnia az ágensnek. Nevezetesen, egy véletlenszerű helyen álló kapszulát megkeres, és elkap. Ebben az esetben a kapszula reprezentálta a játékost, és egy mozgó kocka az ellenfelet. A játéktér egy négyzet alapú kisebb "labirintus" volt.



4.3. ábra. A kezdeti tanítás tér

Miután a tanulópálya elkészült, el kellett készítenem az ágens script-et. Unityben ezt egy olyan komponens ágens objektumhoz csatolásával érhetjük el, mely az Agent osztálytól származik. Természetesen erre a kellő ML-Agents bővítmény beállítása és importálása után van lehetőség. Az AI paramétereinek beállításán túl ez a másik legfontosabb fejlesztés egy jól működő ágens létrehozásában, mivel itt készíthetjük el a szabályrendszert.

Ezen a kisebb scripten ismertetném az Agent osztályból leszármazott osztály használatát.

```
public override void OnEpisodeBegin()
{
    this.rb.angularVelocity = Vector3.zero;
    this.rb.velocity = Vector3.zero;

    target.localPosition = new Vector3(Random.value * 80, 5,Random.value * 80);
    Debug.Log(floor.position);

    Vector3 pointToCentre = new Vector3(-target.localPosition.x,-transform.localPosition.y, -transform.localPosition.z);
    Debug.DrawLine(target.localPosition, pointToCentre,Color.red);
    targetsrb.AddForce(pointToCentre * 20);
}
```

4.4. ábra. Az OnEpisodeBegin() metódus

A tanítás folyamatát epizódokra bontjuk. Ezeknek kezdeti szabályait az OnEpisodeBegin() metódusban definiálhatjuk, és az EndEpisode() metódust meghívva fejezhetjük be. Esetemben itt állítom be az ágens és a célpont kezdeti értékeit, minden egyes epizód elején. Több variánst próbáltam az epizódok szabályaira vo-

natkozóan, és ebben a kezdeti fázisban sikeresnek bizonyult az a módszer, hogy addig tart egy epizód, amíg az AI el nem kapja a kapszulát. Ezzel a módszerrel az epizódok hossza egyre csökkent, azaz egyre gyorsabban elkapta a játékost. Mivel a végeredmény nem hasonlított emberi mozgásra (faltól falig rohant az ágens, és rengeteg ütközés történt), így bevezettem egy büntetést, amennyiben fallal ütközik. Ezt követően, miután elérte az optimumot és kiértékelhettem, a mozgása lassú és kimért lett, viszont amikor meglátott egy kapszulát azonnal rávetette magát. Ez az amit szerettem volna ebben a fázisban elérni, így továbbléphettem a végleges tér megalkotásához, és az abban való tanításhoz.

A következő függvény, a CollectObservations(). Ebben ahogy a neve is mutatja, azokat a megfigyeléseket definiáljuk, amelyet szeretnénk, ha az ágensünk feldolgozna. Ezek alatt nem a vizuális típusú megfigyeléseket értjük, hanem azokat, amelyek számszerűsíthetők. Pl. lehetnek ezek az ágens sebessége, vagy orientáltsága. A jobb teljesítmény eléréséhez nagy szüksége van erre a tudásra, így rendkívüli figyelmet igényel, hogy felvegyünk minden releváns információt a megfigyelések közé.

Az ágens az OnActionReceived() metódussal hajtja végre az ActionBufferekben eltárolt akciókat. Ez a függvény képkockánként egyszer hívódik, kvázi Update() függvényként működik. Ebben történik pl. az ágens mozgatása is. Amiatt, hogy mindig meghívódik használhatjuk konstans jutalmazásra, vagy büntetésre is.

A MoveAgent()-ben a mozgás szabályait definiáljuk, és pontosan úgy működik, mintha egy játékosnak írnánk azt. Azzal a különbséggel, hogy elindítás után ezzel mozgatja a számítógép az adott objektumot. Az én programomban először egy erő alapú mozgásformát választottam, mivel jól kezelve hasonlít a valós mozgáshoz. Ez azt jelenti, hogy miután definiáltuk merre akarunk haladni egy erőhatás éri a testet, így meglökve azt. Azonban ez a módszer inkább testek külső behatástól történő mozgatásához hasonlít, így később lecseréltem ezt egy transzláció alapú mozgásra, mely jobban megfelelt a követelményeknek.

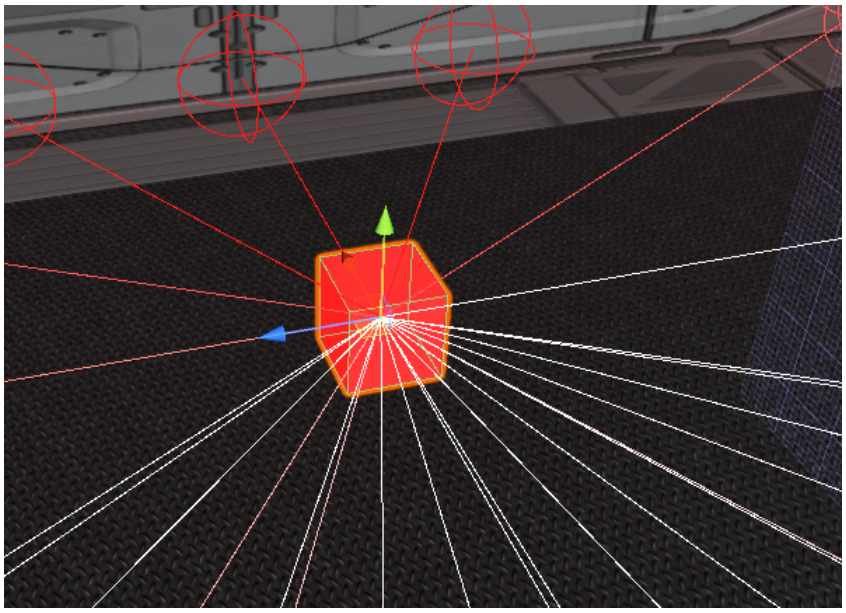
```
public void MoveAgent(ActionSegment<int> act)
{
    var dirToGo = Vector3.zero;
    var rotateDir = Vector3.zero;

    var action = act[0];
    switch (action)
    {
        case 1:
            dirToGo = transform.forward * 1f;
            break;
        case 2:
            dirToGo = transform.forward * -1f;
            break;
        case 3:
            rotateDir = transform.up * 1f;
            rotationCounter+=1f;
            break;
        case 4:
            rotateDir = transform.up * -1f;
            rotationCounter+=1f;
            break;
    }
    transform.Rotate(rotateDir, Time.deltaTime * 200f);
    rb.AddForce(dirToGo * 2f, ForceMode.Impulse);
```

4.5. ábra. A MoveAgent() metódus

Utolsóként megemlíteném még a Heuristic() metódust, mely teszteléskor jön jól. Ezzel átvehetjük az ágens fölött az irányítást és kézzel működtethetjük, mintha egy játékos karakter lenne.

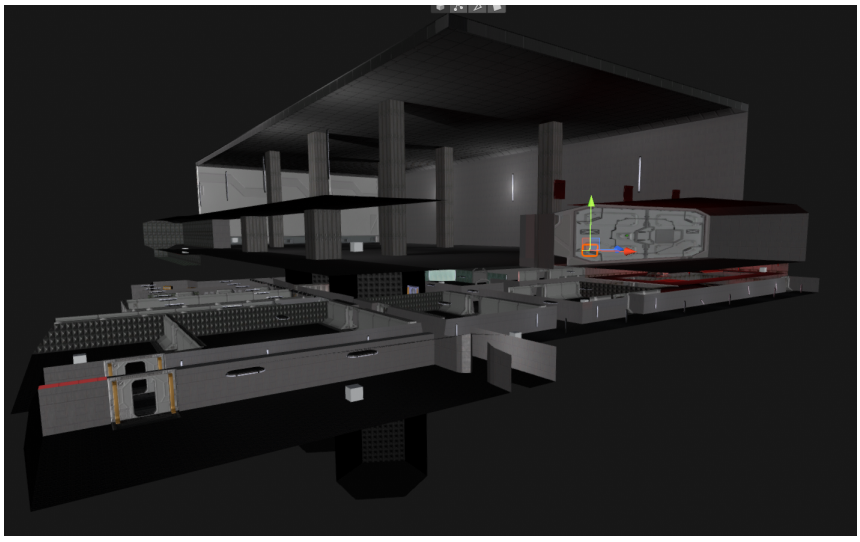
Végül szót kell még ejteni arról, hogy az ágens, hogyan lát. Fentebb kifejtettem, hogy az ML-Agents erre két módot is kínál: a vizuális módot, melyet kamerával kell elérni, és a Raycast-ot. Én ez utóbbit választottam, mert Unity tagekkel hasonló eredmény érhető el, mint látással, és nem volt szükségem olyan mértékű részletességre, melyet a vizuális megközelítés adhat. Így tehát elláttam a falakat "Wall" címkével, a játékost "Player" címkével és hozzáadtam egy RayPerceptionSensor3D komponenst az ágens objektumhoz. Ez utóbbival beállíthatjuk a folyamatos sugár cast-olást, és megszabhatjuk a Raycast-ok bármely tulajdonságát. Így elérhettem az AI szinte láthassa a körülötte lévő elemeket. Ennek segítségével felfedezhette található-e valahol játékos, illetve érzékelte a falakat is.



4.6. ábra. Raycastok az ágens objektumon

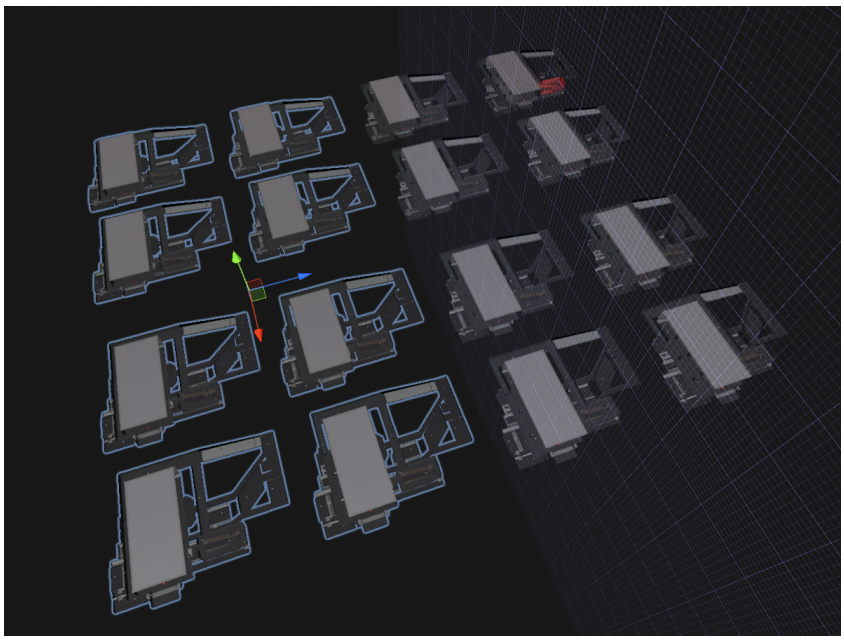
4.2.4. Tanítás a végleges térben

Miután a tesztpályán megfelelően teljesített, áttérhettem a végleges pálya felépítésére. Egy elég nagy, de nem túl méretes nyílt területet képzeltem el, melynek a játékos és az AI is szabadon bejárhat. A modellezés rendkívül sok időt igényelt, azonban figyeltem arra, hogy a végeredményben egy olyan játszótér keletkezzen, amelyen kevés a zsákutca, és tulajdonságaiban jó táptalajt adhat egy fogócska szerű játékmenetnek. Mint a 4.7-es ábrán látható a pálya többszintes és igen kiterjedt. Az első



4.7. ábra. A játéktér

próbálkozásokra beigazolódott, hogy ami működik kis léptékben, ebben a térben már egyáltalán nem. Problémák sora ütközött ki, nevezetesen az AI nem fedezte fel a rendelkesésre álló területet, kiszökött a játékterületről, vagy csak nem tudta megkeresni a játékost, így képtelen volt tanulni. Egy új szabályrendszert kellett készítenem, amelyben figyelembe veszem az ágens kezdeti korlátozott képességeit és a pálya nagyságát. Készítettem több "Player"-nek tagelt objektumot melyet lehelyeztem a játéktér véletlenszerű pontjain. Mindössze öt darab volt egyszerre a pályán bizonyos ideig, majd másik ötöt jelentettem meg. Ez így ment az egész tanulási folyamat alatt. Ennek oka az volt, hogy az AI kevesebb valószínűséggel tanulja be a játékos helyét, mivel a kész változatban csak egy lesz, és az pedig mozog. Ezután megvalósítottam egy checkpoint rendszert tovább ösztönözve arra, hogy mozogjon és fedezze fel a pályát. A játékos és checkpoint objektumok megtalálásáért jutalmat biztosítottam, ugyanakkor büntettem a falnak ütközésért és a túl sok forgásért. Ezzel a rendszerrel a háta mögött már elkezdett mozogni, többször játékosokat is megtalált, azonban stratégiát nem alakított ki. Még ennél a pontnál is több büntetést szedett össze, mint jutalmat, így a legjobb eredménynél is a mean reward (összesített jutalom) épphogy elérte a pozitív intervallumot. (Kb -40, -60 körül kezdett.) Ebben az állapotában a játék nem szórakoztató, azonban látszik, hogy egy jól működő szabályrendszerrel igen jó eredményeket érhetnénk el, azonban ezt megtalálni rendkívüli hozzáértést és időt igényel, hiszen előre nem tudjuk meghatározni, hogyan fog viselkedni.



4.8. ábra. A tanulási tér

A tanítás egy igen lassú folyamat és egy ennyire grafika igényes programnál ren-

geteg erőforrást igényel. A módszer melyet használtam az volt, hogy létrehoztam több tréning arénát, amelyeket a Unity scene-ben lehelyeztem egymás mellett. Az, hogy mennyivel tudunk egyszerre dolgozni nagyban múlik a számítógépünk teljesítményén. Egy ekkora játéktér esetén az én esetemben a maximumot kb. húsz darab pályában állapítottam meg. Így a tanítás sebessége sokszorosa annak, mintha egyszerre csak egy pályán tennénk ugyanezt.

4.3. A 3D pálya létrehozása

A következőkben a játéktér létrehozásának lépéseit és miként tjeit ismertetem a saját példámra összpontosítva. A pályatervezést megvizsgálom abból a szemszögből is, hogy milyen nehézségeket, illetve eltéréseket mutathat a hagyományos fejlesztéstől amiatt, hogy egy gépi tanulással ellátott ágens is részt vesz a folyamatban.

4.3.1. Prototyping és ennek hatása az AI-ra

A játékfejlesztés egyik korai folyamata az úgynevezett „prototyping". Ez, ahogy a neve is utal rá prototípusok készítését jelenti. Szokás többek között a pálya megtervezésekor és a játékfunciók készítésekor is ezt használni, számos előnye miatt. Ezek egyike az időfaktor. Egy pálya prototípusát nagyon rövid idő alatt el lehet készíteni, ami lehetővé teszi, hogy ebben teszteljük le a különböző játékelemeket. Ugyanakkor, ha egy terület felépítését szeretnénk megtervezni, akkor is jó szolgálatot tesz. A prototípusok önmagukban nagyon leegyszerűsített másai a végső pályának vagy általános esetben játékelemnek. A méretek megegyeznek, viszont alapvető objektumokkal dolgozunk bennük, amelyek gyorsan elkészíthetőek és nem igényelnek nagy számítási kapacitást. (Azaz többnyire nagyon kevés polygonból állnak.) Amikor egy játékmechanikát szeretnénk tesztelni is célszerű először prototípusokat használni, hiszen akkor csak az adott elemre kell fókuszálni és elkülönítve a többi tényezőtől fejleszthetjük kizárólag azt. Az én esetemben figyelembe véve az időfaktort, egy hibrid megoldást választottam. Elkerülhetetlen volt, hogy prototípusokat készítsek, hiszen a pálya paraméterei folyamatosan változtak a fejlesztés folyamán, ami ilyen módon könnyen kezelhető volt. Azonban a sok tervezés miatt nem maradt volna elég idő a kész letisztult pálya ujjáépítésére, melyet szintén célul tűztem ki. Mielőtt kitérek a módszerre, amit alkalmaztam, meg kell említenem az ebben a részben felhasznált ProBuilder Unity bővítményt.[5]

Ez utóbbi is nagyban hozzátesz ahhoz, hogy a Unity-ben olyan egyszerű és gyors a prototípusok létrehozása. A bővítményt kezelhetjük egy egyszerű és letisztult 3D modellező szoftverként, amelyben lehetőségünk van a Unity környezetén belül 3D objektumokat alkotni. Természetesen sokkal kevesebb funkció kapott helyet benne, mint egy teljes értékű szoftvernél (pl. Blender, Maya), de prototípusok készítésére

kiváló és gyors alternatívát ad. Képesek lehetünk benne kifaragni bármilyen 3D modellt, és azt azonnal felhasználhatjuk a Unity programunkban. Annak ellenére, hogy töredékére képes, mint a nagyobb 3D szoftverek, így is számtalan lehetőséget kínál. Az én megoldásom tehát a fentebb említett hibrid módszer volt, amelyben úgy alkalmaztam a ProBuildert, mint valós 3D szoftvert. Azaz a végleges pálya prototípusának elkészítése és felhasználása után, a kész prototípust fejlesztettem tovább végleges pályává. Ennek részeit textúráztam és építettem hozzá grafikai elemeket. Ez azt jelenti, hogy ahol úgy láttam, hogy maga a prototípus kinézete ellátja a feladatát, vagy csak egy kis változtatásra szorul, ott meghagytam azt és esetleg rátettem néhány dekorációs elemet. Vannak olyan helyek a pályán ahol a moduláris asseteket helyeztem rá a ProBuilderrel készített objektumra, így eltüntetve azt, tartva azonban a méreteket. Amikor megnézzük a kész bejárható pályát akkor elmondhatjuk, hogy mögötte láthatjuk a prototípust is, amin az AI tanult, azonban bizonyos helyeken teljesen el lettek fedve. Egyik előnye ennek a megközelítésnek, hogy az AI tesztelésénél és tanításánál is biztosak lehetünk benne, hogy a végső, kész területen ugyanolyan teljesítményt produkál majd, mint a prototípusnál.

Itt felvetődik a kérdés, mi lenne, ha a fejlesztési idő nem lenne behatárolva és egy teljesírtékű kommerciális szoftver készülne, ahol a prototípus és a végleges pálya készítése teljesen elkülönül. Ebből pedig következik, hogy esetlegesen a prototípus eltér a végleges változattól. Erre megvizsgáltam a választ, egy egyszerű térben. A tesztfolyamatban két lehetőséget próbáltam ki: az első, hogy minden méretet meghagyok olyannak amilyen, csak a felhasznált objektumokat változtatom meg. Ez a vártnak megfelelően ugyanazt az eredményt hozta, mint az eredeti prototípus környezet. Ez köszönhető annak, hogy a raycast-al ellátott AI sugarai ugyanolyan címkékkel ellátott objektumokba ütköztek mindkét esetben, csak a szemünknek voltak eltérőek a falak, számszerűsítve pontosan megegyeznek. A második eset mikor szignifikánsan megváltoztatom a környezetet. Ebben az esetben ahogy várható egy túltanított AI nem teljesít majd jól, ellenben egy optimálisan képzett AI tudja kezelni a helyzetet. Itt megállapíthatjuk, hogy egy játék szempontjából meglehet, hogy ez nem annyira hatékony mint egy túlképzett egyed.(Természetesen lehetne ugyanannyira hatékony, ha azonban a számítási kapacitás véges, akkor el kell gondolkodnunk egy középúton.) Az én esetemben az ágens nem tudta kezelni a helyzetet, de ez nem jelenti, hogy egy olyan AI, amelyet arra képeztek ki, hogy mindenféle környezetben helyt álljon, ne tudná megoldani azt. Azt a következtetést vontam le, hogy előzetesen meg kell állapítanunk, milyen feladatra szánjuk a mesterséges intelligenciánkat. Amennyiben nem lesz más dolga, mint egy előre legyártott környezetben tevékenykedni, úgy a játékok esetében nem gond, ha megtanulja a környezetét, hiszen így biztosak lehetünk benne, hogy jól cselekszik majd az ottani adott helyzetben, ellenben azzal, ha pl. procedurálisan generált pályákon kellene helyt állnia. Abban az esetben vagy szoli-

dabb teljesítményt várhatunk, vagy nagyobb számítási kapacitásra lesz szükségünk és már a tanítás során szükséges kezelni, hogy többféle területen tanulhasson.

4.3.2. Assetek és textúrázás

Ahogy azt fentebb is említettem a programban keverednek a saját magam által készített 3D objektumok és a külső forrásból szerzett assetek. A Unity Asset Store[3] lehetővé teszi, hogy moduláris modelleket szerezzünk, melyekből, mint az építőkockákból felépíthessük a saját magunk által megálmodott játékterünket. Így én is ezt a megoldást választottam, egy Sci-fi modulokból álló asset csomag felhasználásával, kiegészítve saját építményemet. A munka meglehetősen egyszerű az ilyen típusú assetekkel, hiszen előre elkészített (méretezett és textúrázott, shaderrel rendelkező) un. prefabokat is kapunk, melyeket drag and drop módon lehelyezhetünk a Unity környezetében. Ezek rendelkeznek Colliderekkel, ami lehetővé teszi, hogy valós pályaelemként viselkedjenek, azonban teljes szabadságot is kapunk felhasználásukban, hiszen teljesen személyre szabhatók, nem kötelezően kell az előre elkészített beállításokat használni. Élve ezzel a szabadsággal, én magam is hasznát vettem a kész verzióknak és személyre is szabtam néhányat. Hátrányuk azonban, hogy nagyon verzióérzékenyek. Mivel a készítőktől függ ezen modellek frissítése, így az sokszor elmaradhat és egy „up-to-date" Unity verzióban, az assetben felhasznált technológiák (pl. shaderek) már legacy-nak számíthatnak.

A textúrázás ugyan időigényes, de egyszerű a ProBuilderben. Számomra többnyire fém felületek kellettek, így ilyen textúrákat kerestem. Ezeket a ProBuilderben lehetséges objektum face szinten elhelyezni, és beállítani, majd a megfelelő helyekre letenni azokat.

4.3.3. Fények és pályaközi effektusok

A fényeket a játékokban általában két részre bonthatjuk: valós idejű (real time) és un. baked fényekre. Az első típusba azok a fényforrások tartoznak, melyeknek valamilyen módon változnia kell a futási időben. Ez azt jelenti, hogy folyamatosan számítani kell az értékeket, miközben a program fut. Alapvető tehát, hogy ezeket a játékokban a minimumra szorítjuk, azonban sokszor szükségesek. Egy saját példával élve, a karakter kezében található fényforrás mindig valós idejű, hiszen folyamatosan magunkkal visszük.

A baked típusú fények ezzel szemben a játék fejlesztési szakaszában kerülnek kiszámításra. Készítés közben tekinthetjük valós idejű fényeknek, és miután lehelyeztük azokat egy bake paranccsal a Unity motor elvégzi a kellő számításokat, így valójában a játéktér részeivé válnak.

Ugyan nem teljesen fény típus, de meg kell említenem még a Post-processing stacket is. Ez tulajdonképpen tekinthető utómunkának is, hiszen a kameránkra teszünk vele effekteket. Sok fajtája van, amelynek én csak töredékét használtam. Ennek az egyik oka, hogy minél több ilyen effektet használunk annál számításigényesebb lesz a játék. Az én esetemben szerettem volna ezt a minimumon tartani. Használtam úgynevezett bloom effektust, mely egyfajta fluoreszkálást ad bizonyos tárgyaknak, illetve egy vignettet, mely szűkíti kissé a látóteret, így valódibbnak tüntetve fel a sötétséget.

4.3.4. Animációk

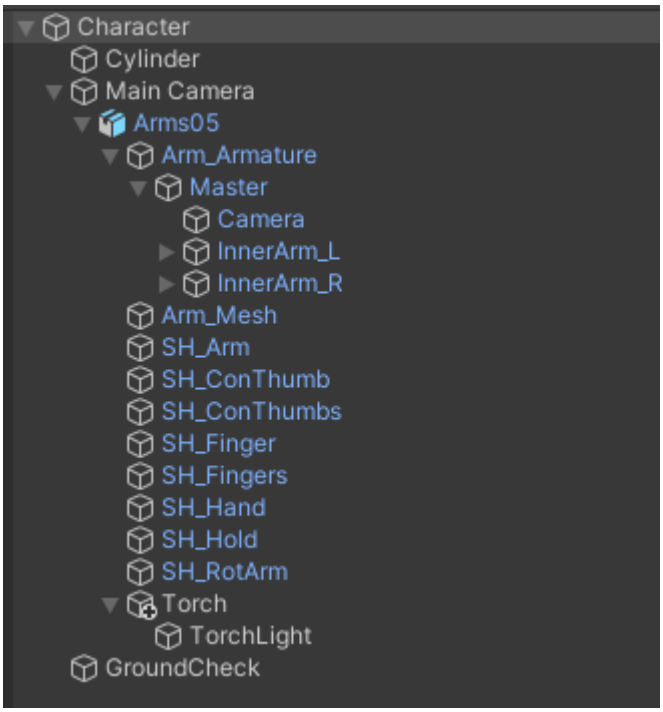
Animációk terén előre legyártott animációkat használtam, hiszen újakat készíteni rengeteg idő lett volna, és akkor érdemes, ha saját magunk által készített modellek is rendelkezésünkre állnak. A pályán szétszórva fellelhetünk kisebb animációkat, ilyen pl. a karakterünk kezének mozgása. A Unityben lehetőségünk van egy animációt elkészíteni, személyre szabni a beépített Animátorral. Ezzel hozhatunk létre magát ismétlő ciklusokat, és külön helyzetre, külön animációt előhívó szabályokat.

4.4. Játéktechnikai elemek

Ebben a részben a játékos perspektívájából ismertetem a mechanikákat és azok technikai hátterét. Ezek többnyire csak a játék olyan részeit fedik, amik nem kapcsolódnak szorosan az AI-hoz, azonban szükségesek, hogy létrejöhessen egy teljesértékű játékszoftver. Kitérek a karakter és egyéb pályaelemek programozás és motorbeli hátterére is - melyek egy egyszerű FP (First Person) programban szinte kötelező jelleggel előfordulnak - és azok általam való implementálására.

4.4.1. Karakter

Lévén egy "first person" típusú játékról beszélünk implementálnom kellett egy ilyen típusú karaktert, melyet játékos irányít és mindent ezen szemszögből lát. A Unity motor nagyban segít ezt gyorsan és egyszerűen implementálni. Létezik egy "Character Controller" nevű típus, melyet bármilyen GameObjecten felhasználva az adott objektum karakternek tekinthető. Ez beállítja az alap kellékeket, amivel egy irányítható karakternek rendelkeznie kell, azonban a szabályokat nekünk kell scriptekben definiálnunk. Az én esetemben, mivel egy túlélő séta szimulátorról beszélünk, elsőként egy minél realisztikusabb mozgásra van szükség. Ez azt jelenti, hogy képes előre-hátra mozogni, egérrel szétnézni és ugrani. Ezen kívül egy interakció gomb is szükséges. Ezeket a tulajdonságokat akkor érhetjük el, ha a karakterünk egy fizikai testként jelenik meg. Unityben ez a Rigidbody komponenst jelenti, ami tulajdonképpen a karakter testén (mesh) helyezkedik el, és rákényszeríti az alapvető fizikai



4.9. ábra. A játékos karakter hierarchia

törvényeket. Így képesek vagyunk az ütközéseket (collision) detektálni, hathat rá súlyerő és gravitáció, azaz elérhetjük, hogy a szabályaink szerint és csak a pályán belül mozoghasson. Mint a 4.9-es ábrán is látható a karakter rendelkezik egy testel, ami tulajdonképpen egy mesh, és egy kamerával ami az egerünk mozgatásával irányítható és a karakter fejét szimbolizálja - ezen keresztül látja a világot a játékos.

Az inputok a következők:

- w, a, s, d - előre mozgás, balra mozgás, hátra mozgás, jobbra mozgás
- kamera mozgatás az egérrel
- space - ugrás

4.4.2. Az ellenfél

A pályán egyetlen ellenfél van, amely kutat a játékos után. Amennyiben elkapja, a játék véget ér. Ezt az ellenfelet elkerülve kell teljesítenünk a feladatunkat, és túlélni.

4.4.3. Tennivalók a pályán

A játék célja, hogy összegyűjtsük a kellő mennyiségű alkatrészt a pályán szétszórva, és elhagyjuk az űrállomást egy mentőkabinnal, mielőtt az ellenfél ránk talál és elkap. Ezt nehezíti, hogy az alkatrészek a pályán szétszórva találhatók meg, és a keresés

közben nem szabad engedni, hogy az ellenfél elkapjon.

Implementáció szempontjából most is arra törekedtem, hogy a tennivalók sora később bővíthető legyen. Ezért készítettem egy Collectable osztályt, és abból származtattam az alkatrész osztályt. Így, amennyiben később beraknék még másfajta játéktechnikai elemet, melyet képes a játékos felvenni, úgy a kódban csak egy másik származtatott osztályt szükséges létrehozni, és az adott felvehető objektumot ellátni a "Collectable" címkével. A játék ezen állapotában, ha megkerestük és felvettük az összes szükséges alkatrészt egy felirat jelenik meg a képernyő tetején mely arra figyelmeztet, hogy mehetünk a cél felé és megnyerhetjük a játékot. Egy egyszerű inventory rendszer pedig segít követni, hogy épp hány alkatrészt vettünk föl. Ezt a képernyő bal felső sarkában láthatjuk.

4.5. Hangok

Hangok terén a Unity szintén megkönnyítette a dolgomat, ugyanakkor szerettem volna egy univerzális módot a hangok lejátszására, amely később könnyen bővíthető. Ezért készítettem egy komponenst, amely lehetővé teszi, hogy egyszerűen adhassak hozzá, vagy vonhassak vissza audio fájlokat, bizonyos helyzetekben. Ezen komponens használatával a programkódban bárhol egy sorral behívhatom a kellő hangot. Így valósítottam meg a lábdobogást és a háttérzenét is.

4.6. További lehetőségek

A játékot a későbbiekben szeretném továbbfejleszteni több szempontbol is:

- Első és legfontosabb lehetőség az AI további fejlesztése és tesztelése, egy kihívást jelentő ellenféllé tétele.
- Egy magam elkészítette animált 3D karakterré alakítani az ellenfelet, mely különböző mozgásformákat használ különböző szituációkban
- Bővíteni a felvehető tárgyak listáját és bevezetni új kritériumokat a nyeréshez
- Több pálya integrálása, és ehhez egy nagyobb szabású történet elkészítése
- Több animáció behelyezése, pl. nyitható ajtók formájában
- Véletlenszerű események használata az atmoszféra fenntartása érdekében
- Részletesség növelése több lehelyezett pályaelemmel
- Hallás bevezetése az ellenfélnél, különböző hanghatások produkálása a karakterrel

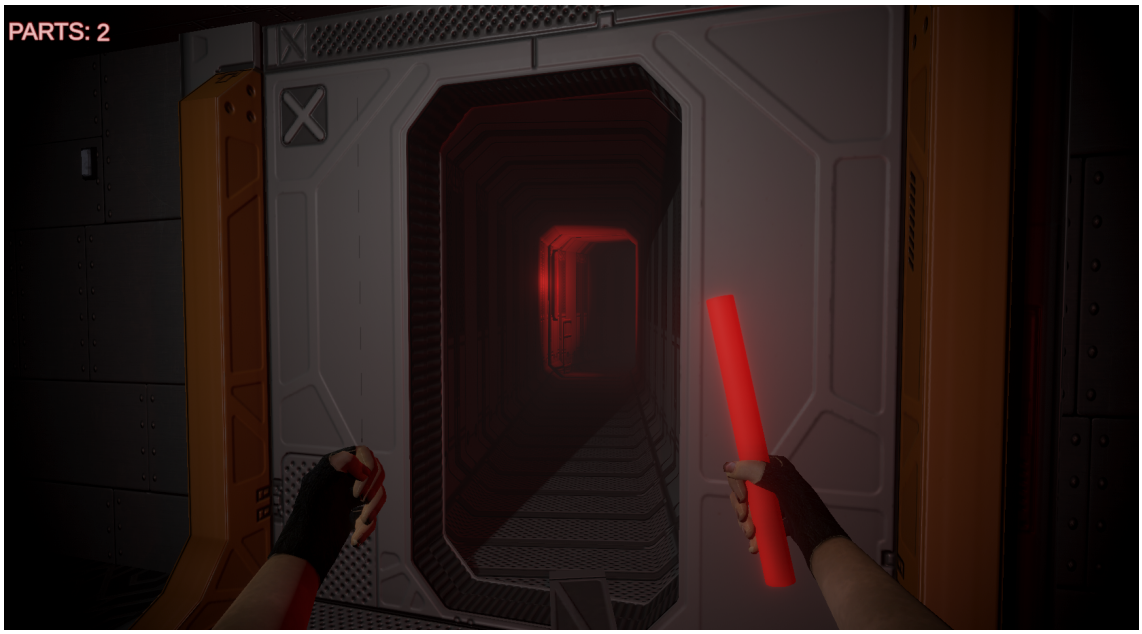
4.7. Összegzés

Ezzel a dolgozattal célom volt közelebb kerülni a játékfejlesztéshez és megtanulni az abban használt módszereket, ugyanakkor egy új szemléletmódot is megvizsgálni azon belül. Mindenképp egy háromdimenziós programot akartam létrehozni, mert ebben több lehetőség rejlik, illetve szélesebb spektrumú tudás elsajátítása szükséges hozzá. Egy ilyen arena megtervezése és létrehozása nagyfokú modellezési tudást is igényel, melybe ily módon betekintést nyerhettem. Létrehoztam egy viszonylag nagy területű játszóteret, melyet elláttam fényekkel és néhány berendezéssel. Ugyanakkor ezen túlmenően szükség volt néhány elemre, melyek interaktívvá teszik a játékélményt. Ilyenek az animációk, mely szintén egy új terület volt számomra. Ezeken túlmenően használnom kellett hangokat, melyekre a játékfejlesztésben szintén sajátos módok vannak. Szükség volt még a materialok és textúrák használatára is, hogy némelyest valósághű környezetben mozoghassunk. Ezen felül a különböző elemeket komponens scriptekkel kellett ellátnom, hogy viselkedésük megfeleljen az általam elvártnak. Ugyan nem építettem be nagy mennyiséget ezekből a technikákból a játékba, de szükségesnek tartottam, hogy képviseljék magukat egyes helyeken, így szimulálva a kereskedelmi szoftverek elkészítésének különböző területeit.

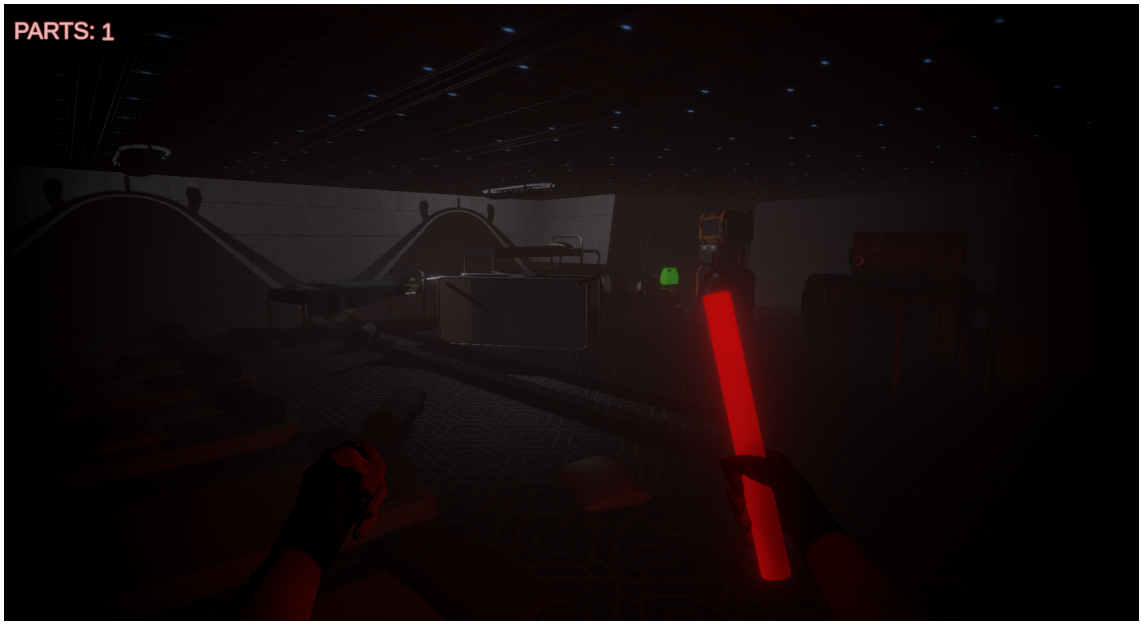
A játékfejlesztés témaköreinek miként jein túl, megvizsgáltam a gépi tanulással ellátott mesterséges intelligencia relevanciáját ebben az iparágban. A játékban ezért szükségem volt egy olyan ellenfél ágensre amelyet ezzel működtetek. Ennek megalkotása több lépésben készült el, melyekhez szükséges volt némi tudást összegyűjtenem az AI területéről is. Nagy segítségemre volt az ML-Agents bővítmény, mellyel az ágens létrehozásához szükséges idő lerövidült. Megépítettem többféle tesztkörnyezetet, melyekben egyre bonyolultabb lett elérni a célt. Az előzetes sikerek ellenére megállapíthattam: minél nagyobb a tér, az AI annál nehezebben boldogul, így egyre több támpont felállítása szükséges. Ugyanakkor nagy hangsúlyt kellett fektetnem arra is, hogy a tanítás folyamán figyelembe vegye, hogy a lényeg nem csak az effektivitás, hanem az emberi mozgás is. Ezért több olyan tényezőt kellett bevezetnem, ami visszafogja. Ezek figyelembevételével kezdtem el a tanítást, amely eleinte, kis környezetben gyorsan haladt, viszont ahogy növeltem a teret egyre lassabb lett. Az így elkészült AI sajnos jelen állapotában nem képes tökéletesen ellátni a feladatát, azonban bízom benne, hogy további próbákkal és tanítási ciklusokkal végül kompetens ellenféllé válik.

A fejlesztés végén elkészítettem bizonyos elemeket, melyek nélkül a játék nem lenne teljes. Ezek a különböző két dimenziós felületek a 3D téren. Ilyen a főmenü, a HUD, vagy a különböző üzenetek a játék közben. A Unity motornál ezek a két dimenziós játékfejlesztés módszereivel történnek, így alkalmam volt bepillantást nyerni ebbe a területbe is.

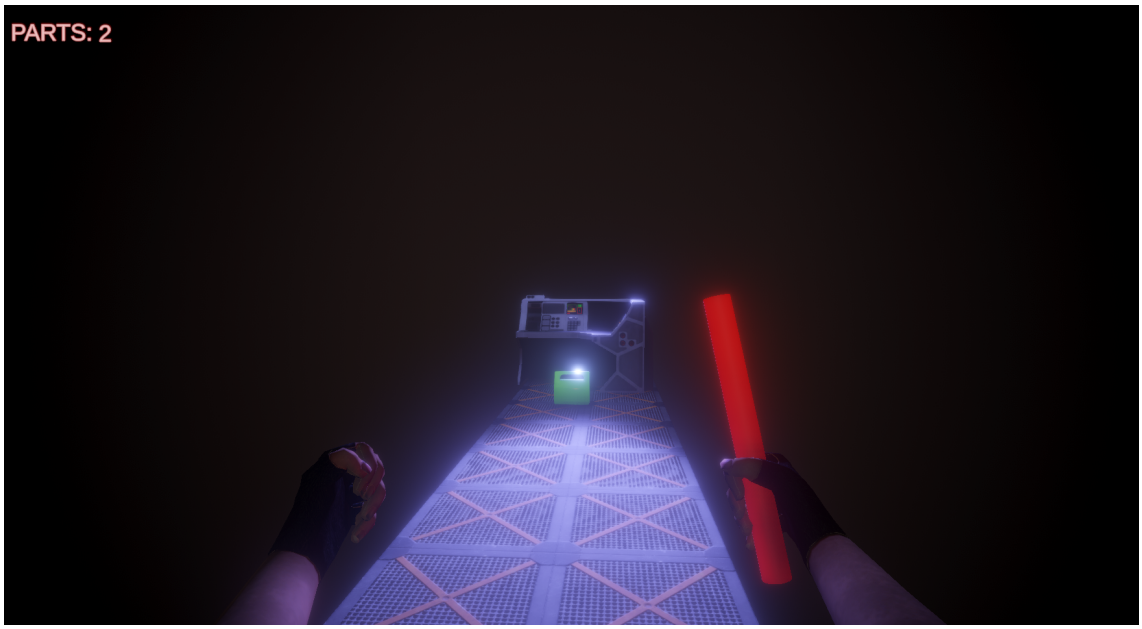
Végezetül álljon itt néhány kép a játékból:



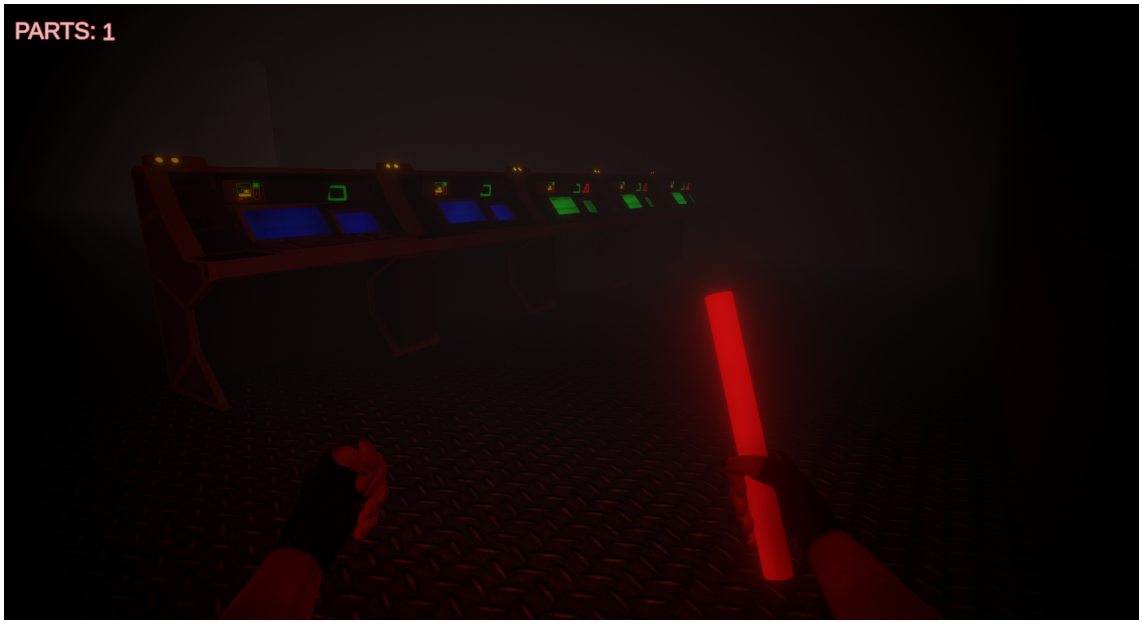
4.10. ábra. Egy folyosó



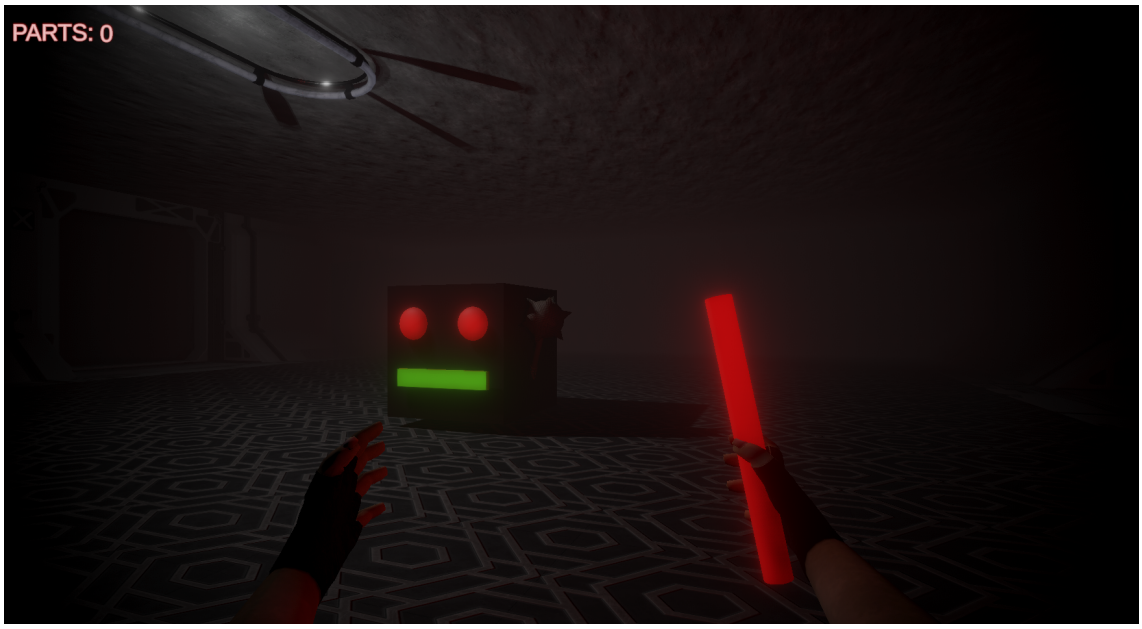
4.11. ábra. Egy berendezett szoba



4.12. ábra. A felvehető tárgyak egyike



4.13. ábra. Az aktiválható számítógépek



4.14. ábra. Az ellenfél

5. Tesztelés

5.1. A karakter tesztelése

A karakter öt kritériumnak kellett, hogy megfeleljen. Ezek a mozgások, az interakció, a környezettel való ütközések hatása, illetve a "game over" és nyerési paraméterek.

- A w, a, s, d - vel a script megírása után lehetővé vált az irányítás. Ezt úgy érhettem el, ha az y tengely scriptjét a kamerára tettem, az x és z tengelyt pedig a karakter testére. A megoldás egy transzláció alapú mozgásforma lett. Az ugrás a space-szel történik, mely eleinte, túl lassú és magas volt. Ezt is egy scriptből kellett megvalósítanom ahol definiáltam a gravitációt, melyet a kellő hatás eléréséhez növelnem kellett. Így többszöröse a valóságnak, de a játékban igazibb hatást kelt. Kiderült a tesztek folyamán, hogy a levegőben is tudunk ugrani, ezt elkerülendő bevezettem egy "groundcheck" változót, mely csak akkor engedélyezi az ugrást, ha a karakter a földön tartózkodik. Ezt a játék folyamán folyamatosan az Update() metódusban ellenőrzöm.

- Az interakciók kimerülnek abban, hogy fel tudunk venni bizonyos tárgyakat, majd azokkal aktiválni az öt számítógépet. A felvétel szimpla áthaladással történik. Ez tecnikailag elpusztítja a tárgyat, viszont a karakternél növelődik egy számláló, hogy hányat szerzett meg. Ezt jelzendő egy számláló található a

HUD-on, mely a nálunk lévő darabokat jelzi. Amennyiben felvettük mindegyiket megjelenik egy felirat, mely tájékoztat róla, hogy aktiválhatjuk az összes számítógépet. Az aktiválás szintén ütközés detektálással történik, melyre válaszul a számítógépek képernyői bekapcsolnak, azaz materialt váltanak. Mivel a számítógép asset számos részből épül fel, azt kellett elérnem, hogy csak a képernyő változzon. Ennek tesztelésénél kiderült, hogy az egész gépre rákerül a material, így ezt kijavítandó a gameobject GetComponent() helyett annak gyermek elemeire kellett a vonatkoztatnom, ami kiválasztva a képernyő elemet megoldotta a problémát.

- Fontos szempont volt, hogy a falakon át ne tudjunk közlekedni. Mivel a pálya maga is egy Mesh és az azon található letett objektumok is rendelkeznek "mesh collider"-rel, azaz képesek detektálni az ütközéseket, ameddig ez fennáll nem tud a karakter átmenni falakon. Természetesen kell, hogy a játékos karaktere rendelkezzen egy ún. "Rigidbody"-val, vagy mint esetemben egy "Character Controller"-rel. Ez utóbbit választottam, mert a funkciói kifejezetten játékos karakterek irányítására vannak specializálva, és az én esetemben az implementált irányítás ezt a komponenst követelte meg. Mivel az ütközések lassan történnek, elegendőnek ítéltem egy diszkrét ütközés vizsgálatot beállítani, amely lassabban vizsgálja, hogy ütköztünk-e, viszont kevésbé erőforrásigényes.

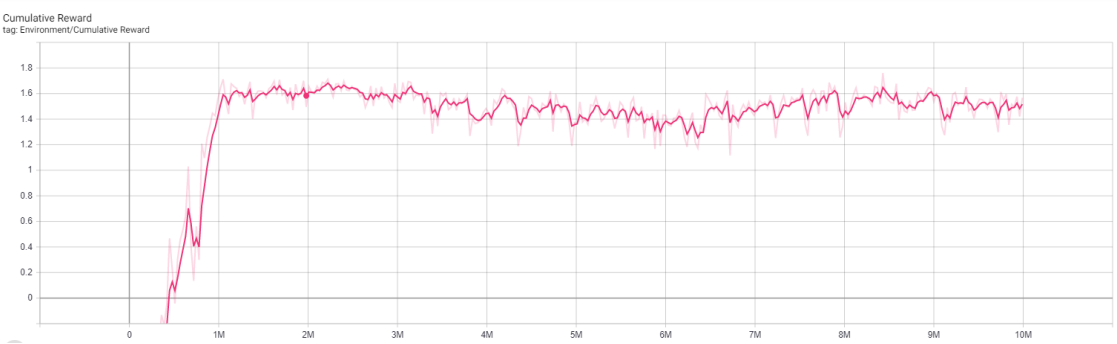
- A nyerés és vesztés letesztelése egyszerű boolean változók lévén kimerült abban, hogy felvettem a megszerezhető tárgyakat, aktiváltam a számítógépeket, és bementem a kinyíló ajtón. Miután a "You won" ablak feljött, ezt a funkciót működőképesnek találtam. Ugyanez a helyzet a vesztes játékkal is. Két módunk van elveszíteni a játékot: az első, hogy ütközünk az ellenféllel, a második, hogy leesünk a pályán elhelyezett mély helyekre. Ezek mindegyike aktiválta a "You died" ablakot, így a funkciót működőképesnek ítéltem.

5.2. Az AI tesztelése

Az AI tesztelését két részre osztottam. Az első a paraméterek állítgatása során beállt változás vizsgálata, mely maga a tanulási fázis. A második részét a kész pályán történő alkalmazásnál vizsgáltam.

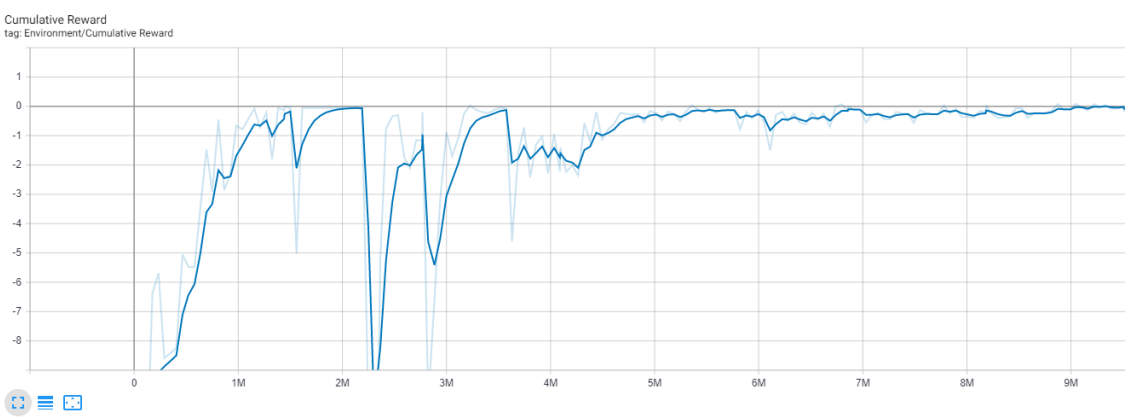
- A kisebb pályákon történő tanítás abból állt, hogy véletlenszerű kapszulákat kell megtalálnia, és mindig új epizód kezdődik, amikor elkapott egyet. Mivel ez meglepően jó eredményekkel szolgált, a teszt fázisban behelyeztem egy játékos karaktert az említett kis térbe, és elindítottam a programot. Mivel a teszt kedvéért az AI-t gyorsabbnak állítottam be, mint a karaktert, így képes volt elkapni. Ez körülbelül 60% hatékonysággal sikerült neki. A többi esetben is

találkoztunk, azonban köszönhetően a pálya labirintusszerű elrendezésének, el tudtam szökni előle. Természetesen, ha az AI-t hatalmas előnyhöz juttatom azzal, hogy nagyságrendileg gyorsabbá teszem a karakternél, akkor előbb utóbb minden esetben elkapott volna.



5.1. ábra. A teszt térben tanított AI kummulált jutalma

- Ugyanez volt a metódus a kész pályán, viszont itt már sokkal kisebb sikerrel végzett az ellenfél. A tesztelést megnehezítette a folyamatos újratanítás, melyre minden paraméter megváltoztatásakor szükség volt, illetve az erőforrások korlátozottsága. Ezért a tesztelés rendkívül időigényes folyamat egy ekkora játéktérben. Míg a kisebb tanítási térben a mean reward nehézség nélkül elérte a kettőt, itt sikeresnek számított, ha átbillent a pozitív oldalra. Ez természetesen köszönhető annak, hogy a jutalmazási rendszerben - a pálya bonyolultsága miatt - sokkal több negatív jutalmat kellett kiosztanom. Célom volt, hogy minél nagyobb területet bejárjon az ellenfél, ezzel szimulálva a játékos keresését. Ennek elérése érdekében több dummy játékost tettem le, melyek megtalálásáért pozitív jutalomban részesítettem. Azonban komoly akadálynak bizonyult az AI véletlenszerű mozgása, így sokszor nem jutott el odáig, hogy megtanulja, hogy nem célravezető egy helyiségben maradnia. A paraméterek változtatása és a dummy objektumok véletlenszerű megjelenítése arra az eredményre vezetett, hogy képes megtalálni a játékost, amennyiben az közel van hozzá, viszont nem jött létre szisztematikus stratégia a pálya feltérképezésére. A gyakorlatban ez azt jelenti, hogy szükségtelenül sok időt tölt egy helyen, és amíg nem lát jutalmat jelentő játékost, a mozgása véletlenszerű marad. Ezen túl a célpontot könnyen elveszíti. A tesztek azt mutatják, hogy ebben az állapotában nem biztosítja azt a kihívást, amit el kellett volna érnie. Azonban ezt hosszú távon kiküszöbölhetőnek találtam további jutalmi rendszerek hozzáadásával és tesztek elvégzésével. A Tensorboard grafikonok nagyon jól mutatják, hogy az AI képes a tanulásra, ekkora térben is, hiszen sokáig meredek növekedés látható a kummulált jutalomban.



5.2. ábra. A tanítás első négy órájának kummulált jutalma

5.3. Az események tesztelése

A programban eseménynek tekintettem, azokat a dinamikus változásokat, melyek a játékos valamely tettére megváltoztatták a játék valamely paraméterét. Ide sorolnám a kellő UI felületek kellő időben való felbukkanását is, azonban a nyerés és vesztés ablakait össze tudtam vonni a játékos tesztjeivel.

- A játékban jelenleg két scene található és ezek váltogatása felhasználói interakcióra történik. Ez többnyire a UI tesztelésének is nevezhető. A Unity-ben ezek beállítása a scene manager-ben és scriptekből történnek. A tesztet sikeresnek ítéltem amennyiben a Start game, Quit és Resume gombok jól működtek, azaz a kellő scene-ekre vezettek át. Ez minden eshetőségre megtörtént.

- A másik fontos esemény a játék megnyerésével áll kapcsolatban. Amennyiben a játékos összegyűjtötte a lehelyezett tárgyakat és aktiválta a számítógépeket, az beindít egy láncolatot. Ennek hatására a pálya végi ajtón beindul egy animáció és ez kinyitja az ajtót. Utána amennyiben belép az ajtón, feltűnik a nyerés UI felülete. A teszt többször is sikeresen végződött egy teljes, leszimulált játékmenetben, ahol az öt felvehető tárgy után aktiváltam a számítógépeket, ez kinyitotta az ajtót - az animáció segítségével -, majd beléptem a nyerést aktiváló játék objektumba. Az ablak megjelent, így kijelenthetem, hogy maga a játékmenet tesztje is sikeres.

5.4. Rendszerkövetelmények

A kész játék viszonylag nagy erőforrásigényű lehetett volna, köszönhetően a több Post processing effect-nek. Azonban a fejlesztés folyamán figyeltem arra, hogy a használt és készített objektumok is kis számú polygonból álljanak, így kipróbálható gyengébb konfigurációkon is. A teszteket a következő összeállításokon teszteltem:

- 1. konfiguráció
  - Operációs rendszer: Windows 10 Education
  - CPU: Intel Core i5 9600k
  - GPU: NVIDIA Geforce RTX 2070s 8 Gb
  - RAM: 16 Gb
- 2. konfiguráció
  - Operációs rendszer: Windows 10 Enterprise
  - CPU: Intel Core i7 7600U
  - GPU: Intel HD Graphics 620
  - RAM: 16 Gb
- 3. konfiguráció
  - Operációs rendszer: Windows 10 Education
  - CPU: Intel Core i7 7700HQ
  - GPU: NVIDIA Geforce GTX 1060 6 Gb
  - RAM: 16 Gb

Az 1. és 3. konfiguráción a játékélmény folyamatos volt, teljesen megfelelt az elvárásoknak. Ezen esetekben az FPS szám majdnem folyamatos 60 volt. A 2. konfiguráción az FPS leesett akár 10-15-re, köszönhetően az integrált GPU-nak.

6. Konklúzió

A szakdolgozatom célja tehát egy olyan játék létrehozása volt, mely egy újfajta megközelítésű mesterséges intelligenciát használ, ezzel azt kutatva, hogy ez a technológia mennyire lehet ésszerű döntés a mai sztenderdekhez mérve. Ahhoz, hogy ezt megvalósíthassam választanom kellett egy játékmotort, melyet behatóbban meg kellett ismernem, hogy dolgozhassak majd benne. Átnézve a lehetséges alternatívákat a Unity tűnt a legjobb választásnak, így emellett döntöttem. Ez után meg kellett ismernem a múltban és a jelenben használatos AI technikákat, mert elengedhetetlen volt, hogy átlássam a mai bevett rendszereket, és meg tudjam fogalmazni a választ arra, hogy miért ezeket használják. Továbbá ki kellett igazodnom a gépi tanulási módszerekben, hogy leszűkíthessem egy olyan kis rétegre azokat, amelyek a játékfejlesztésben is hasznosnak bizonyulhatnak. Lévén ez egy hatalmas témakör, szakdolgozatomban csak a felszínt karcolgatom, célom mindössze annyi volt, hogy

kiválaszthassam azt az egy módszert amit majd behatóbban tanulmányozva elkészíthetem saját verziómat.

A játék készítése során rengeteget megtanultam a Unity motorról, az AI technológiákról és rá kellett jönnöm, hogy rendkívül alábecsültem a munkát, melyet egy jól működő 3D játékszoftver igényel. Az időm egy jelentős részét a játéktér létrehozása vitte el, és azon esztétikai elemek lehelyezése, melyek elengedhetetlenek annak kihasználásához. Rengeteg olyan komponenst kellett ezen kívül elkészítenem, melyek azért felelnek, hogy a játék játszható legyen. Legnagyobb részt azonban az AI tanulmányozása és próbálgatása vitte el. Ezt azért fontos megemlítenem, mert ugyan az ezen kívüli részek hosszadalmas folyamatok, de egytől egyig egzaktak és pontosan tisztában lehetünk azzal, hogy hogyan fog működni. Ezzel szemben a gépi tanulásnál a helyzet más. Nem tudjuk elkerülni a próbálgatást, mert nem tudhatjuk pontosan hogyan fog reagálni bizonyos eseményekre. Adhatunk a kezébe egy szabályrendszert, melyet követ, de a stratégiát nem mi alakítjuk ki. Szakdolgozatom elején pontosan ebben láttam a gépi tanulás előnyét, azonben rá kellett jönnöm, hogy ez a legnagyobb hátránya is. Amennyiben sikerül egy olyan megoldást találni, amely maradéktalanul kielégíti a követelményeinket, úgy szinte minden helyzetben jobbnak bizonyul ez a módszer. Azonban ezt elérni rendkívül körülményes, és akkor sem biztos, hogy minden eshetőségre jól reagál. Ezen kívül egy nagyobb játékhoz igen behatóan ismernünk kell ezt a területet. Az én példámban egy egyszerű mechanikákkal rendelkező szoftvert készítettem el, de még így is sok nehézségbe ütköztem és jelen állapotában még lenne hová fejlődnie az AI-nak. Így érthető miért vonakodnak átállni a játékfejlesztő cégek erre a módszerre. Az iparosítás korát éljük a játékfejlesztésben is és a cégek inkább haladnak a kitaposott, könnyebb úton, mint hogy új dolgokkal kísérletezzenek, ami egyáltalán nem biztos, hogy sikerhez vezet. Ezen kívül szakdolgozatomból az is kiderült számomra, hogy teljesen eltérő megközelítést igényel egy ilyen játék elkészítése, mint egy hagyományos szoftveré. Talán még a szakemberek szükséges képzettsége is eltérő lehet a hagyományos fejlesztőkétől. Összegezve, egy rendkívül nagy átszervezést igényelne az ilyen típusú játékok fejlesztése, és az eredmény akkor sem garantált. Ennek ellenére a kísérletezőbb indie szférában elképzelhető egy ilyen csapat és szoftver elkészítése, amely ha sikeres, többen követhetik a példát. Ugyan teljesen kiszorítani a közeljövőben nem tudja a hagyományos módszert, de néhány műfajnál elképzelhetőnek tartom, hogy mint alternatíva, jelen lesz. Az egyik legkedveltebb indie műfaj a túlélő horror, mely játékok kiváltképp alkalmasak lennének az ezzel való kísérletezésre.

7. Summary

My initial idea was to examine and find a solution to that constant problem in games, that enemy AIs can turn to be predictable, sometimes boring opponents. Also I wanted to get to know more about some of the game development technics and best practices. These are the reasons I chose to work in Unity game engine, which gives me the opportunity to develop an actual 3D game fastly and also makes it easier to experiment with maching learning technics thanks to the Unity ML-Agents toolkit. In my thesis I developed a 3D first person walking simulator game where I, as the character must collect five items in order to escape from an abandoned space station. Meanwhile an enemy, controlled by deep reinforcement learning tries to catch me.

Working on the game I used technics such as 3D modelling, lighting, animating and post-processing. I learnt a lot about AI development, and what are the advantages and disadvantages of it. Developing AI is time consuming at best. Doing it we can't escape from its iterative development cycle and countless unknown variables. I had to tweak little parameters which were followed by hours of traing to see some changes in the result. Thanks to that my AI in its current state is not a chellenging opponent. On the other hand its visible that with retraining and more invested time it has the possibility to become a better one. This is why I think it has a chance to be used in games in the future although to achieve this, it's much more complicated than I initially thought. It can have an effect with the smaller yet more experimental game developers. Since it's so circumstancial to build one working AI companies stick to the best practices in this field. In today's game industry it became more important to build huge environments and visually stunning games with less innovation, than experiment with brand new processes or ideas.

In conclusion it is not certain that machine learning will ever replace regular AI in games, but in some of the genres I see a possibilty to make good use of them.

Hivatkozások

[1] J. K. Haas, „A history of the unity game engine”, 2014.

[2] J. Hocking, *Unity in Action Second Edition*. Manning, 2018.

[3] U. Technologies, *Unity - Manual: Using the Asset Store*. cím: (https://docs.unity3d.com/2018.3/Documentation/Manual/AssetStore.html) (elérés dátuma 2021. 05. 10.).

[4] *Snaps Unity*. cím: https://unity.com/products/snaps (elérés dátuma 2020. 04. 29.).

[5] *Home - ProBuilder Documentation*. cím: https://unity-technologies.github.io/procore-legacy-docs/probuilder/probuilder2-gh-pages/ (elérés dátuma 2020. 04. 28.).

[6] *Unity - Manual: Unity User Manual (2019.3)*, en, Library Catalog: docs.unity3d.com. cím: https://docs.unity3d.com/Manual/index.html (elérés dátuma 2020. 04. 28.).

[7] *Unity-Technologies/ml-agents*, original-date: 2017-09-08T21:09:04Z, 2020. ápr. cím: https://github.com/Unity-Technologies/ml-agents (elérés dátuma 2020. 04. 28.).

[8] G. N. Yannakakis, „AI in Computer Games: Generating Interesting Interactive Opponents by the use of Evolutionary Computation”, en, 257. old.,

[9] C. Bauckhage, C. Thurau és G. Sagerer, „Learning Human-Like Opponent Behavior for Interactive Computer Games”, *Pattern Recognition*, G. Goos, J. Hartmanis, J. van Leeuwen, B. Michaelis és G. Krell, szerk., Series Title: Lecture Notes in Computer Science, 2781. köt., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, 148–155. old.

[10] E. F. Anderson, „playing smart – artificial intelligence in computer games”, en, 15. old.,

[11] I. Millington és J. D. Funge, *Artificial intelligence for games*, en, 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2009, OCLC: ocn319064669, ISBN: 978-0-12-374731-0.

[12] S. Schaal, A. J. Ijspeert, A. Billard, S. Vijayakumar és J. Hallam, szerk., *From Animals to Animats 8: Proceedings of the Eighth International Conference on the Simulation of Adaptive Behavior*, en. The MIT Press, 2004, ISBN: 978-0-262-29144-6. DOI: 10.7551/mitpress/3122.001.0001. cím: https://direct.mit.edu/books/book/4496/From-Animals-to-Animats-8Proceedings-of-the-Eighth (elérés dátuma 2020. 04. 29.).

[13] N. Corporation, *NVIDIA DLSS*, en, Library Catalog: developer.nvidia.com, 2018. szept. cím: https://developer.nvidia.com/dlss (elérés dátuma 2020. 04. 29.).

[14] *Neurális hálózatok*. cím: https://gyires.inf.unideb.hu/GyBITT/19/ (elérés dátuma 2020. 04. 29.).

[15] S. Charles és D. McGlinchey, „The Past, Present and Future of Artificial Neural Networks in Digital Games", 7. old.,

[16] I. Fazekas, *3. fejezet - A többrétegű perceptron*. cím: https://gyires.inf.unideb.hu/GyBITT/19/ch03.html (elérés dátuma 2021. 05. 12.).

[17] L. Tóth, *Megerősítéses tanulás, mély Q-hálók*, 2017. cím: https://www.inf.u-szeged.hu/~tothl/ann/10.%20Megerositeses%20tanulas.pdf (elérés dátuma 2021. 02. 11.).

[18] *DeepMind - What if solving one problem could unlock solutions to thousands more?*, ALL, Library Catalog: deepmind.com. cím: https://deepmind.com/research (elérés dátuma 2020. 04. 30.).

[19] *OpenAI*, en, Library Catalog: openai.com. cím: https://openai.com/ (elérés dátuma 2020. 04. 30.).

[20] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang és W. Zaremba, „Openai gym", *arXiv preprint arXiv:1606.01540*, 2016.

[21] J. Wexler, „Artificial Intelligence in Games:" en, 19. old.,

Ábrák jegyzéke