Matematikai és Informatikai Intézet

Körökre osztott stratégiai játék készítése MiniMax alfa-béta vágó algoritmus használatával

Készítette
Herbák Marcell
Programtervező Informatikus BSc

Témavezető
Dr. Kovásznai Gergely
Egyetemi docens

Eger, 2025

Tartalomjegyzék

1. fejezet

Bevezetés

Szakdolgozatom témájának kiválasztása nem okozott számomra fejtörést, ugyanis amióta az eszemet tudom, valamilyen formában játszottam már számítógépes játékokkal. Emlékszek, amikor nagyapám 5. születésnapomra egy számítógépet adott ajándékba, még akkor Windows 2000-es operációs rendszerrel. Bár egy ideig utána abbahagytam a számítógépes játékokat, általános iskola 4. osztályában az osztálytársakkal elkezdtünk Minecraft-ozni, és onnantól kezdve töretlenül szeretem valamilyen játékkal elütni az időt.

A szakdolgozatom tervezése során az olyan játékot szerettem volna készíteni, amelynek stílusát ismerem, játszottam már olyannal és szeretek is vele játszani. Ennek az elképzelt játéknak a stílusával és alapszabályaival a 2. fejezetben foglalkozok. Ebben a játékban az ellenfeleket egy mesterséges intelligencia irányítja gráfkereső algoritmus segítségével. Az MI történeti áttekintéséről és játékelméletről a 3. fejezetben olvashat. A játékot Unity játékmotorban képzeltem el az elejétől kezdve, saját készítésű karakterrajzokkal. A 4. fejezetben részletesebben olvashat erről.

A teljes szakdolgozati projektem elérhető a következő linken: `https://github.com/herbakmarcell/thesis`

2. fejezet

A fejlesztendő játék

Mint a bevezetésben említettem, szakdolgozatom programjaként egy olyan játékot szerettem volna készíteni, amely nem csak jól implementálható, hanem amivel jól szórakozok és szabadidőmet is szívesen töltöm. Választásom végül is egy véges, körökre osztott, zérusösszegű stratégiai játék elkészítésére esett. Ötletadónak, az általam kedvelt videójáték szériát, a Persona játékokat választottam. [4]

Az első Persona játék közel 30 éve jelent meg a Shin Megami Tensei szériának spin–off-jaként, így a játékok működésben és történetben bár eltérőek a modern megjelenésektől, jelenleg legfrissebb 2024-ben kiadott (2.1 ábra) Persona 3 Reload-tól, a játék fő mechanikája nem változott: a játékos karakterei csatába kerülnek egy fix számú, hasonló képességű ellenfelekkel szembe. Ezekben a játékokban, különleges képességekkel rendelkező karakterekkel, úgynevezett Personákkal harcolnak a különböző szereplők. [5]

2.1. ábra. Persona 3 Reload

Bár a modern játékokban nem alkalmazzák már, a Revelations: Persona harcrendszere (2.2 ábra) rendelkezett mezőkre felbontott csatatérrel, amelyeknél még a támadásoknak volt egy bizonyos maximális távolsága. Ezek alapján szerettem volna egy olyan játékteret készíteni, ahol a játékosnak nem csak a támadásának a távolságát kell figyelembe vennie, hanem a helyét is a csatatéren. [6, 7]

2.2. ábra. Revelations: Persona

2.1. Játék szabályok

A játék egy fixált méretű, négyzetekből álló, 7x10 nagyságú táblán folyik. A játékot kettő játékos tudja játszani, melyből a szakdolgozatomban az egyik játékos a mesterséges intelligencia lesz. Mindegyik játékos rendelkezik karakterekkel, amelynek kezdő mennyisége a játék indítása előtt kiválasztható. Mindkét játékos rendelkezik minimum 1, maximum pedig 3 karakterrel. A karakterek mennyisége játékosonként eltérő lehet, nem szükséges mindkét játékosnak ugyanazzal a karakter mennyiséggel kezdenie. Az egyik játékos karakterei (több karakter esetén függőlegesen egy mező kihagyással) a 2. oszlopban, a másik játékos karakterei pedig a 9. oszlopban kezdenek. A táblán léteznek akadályozó mezők, amelyekre a játékosok nem léphetnek, illetve nem támadhatják meg.

A játék során a játékosok egymás után jönnek, egy körben az összes karakterükkel végre kell hajtaniuk egy interakciót. Ez a két interakció *lépés* és *támadás* lehet. Lépés során a karakterükkel egy mezőt léphetnek a négy irány közül valamelyik irányba: fel, le, balra vagy jobbra. A játékos karaktere nem léphet olyan mezőre, amelyen már áll egy másik saját karakter, egy ellenfél karakter vagy egy akadály. Támadás során a játékos egy mezőn belül támadhat négy irány közül valamelyik irányba. A játékos

karaktere nem támadhatja meg a saját karakterét, illetve nem támadhat üres vagy akadály mezőt.

A karakterek rendelkeznek életerővel, minden karakter a játék kezdésekor 10 életerőponttal kezd. Támadás során a megtámadott karakter elveszít 1 életerő pontot. Amennyiben a játékos minden karakterére végrehajtott egy interakciót, a játékos átadja a körét a másik játékosnak. A játékosoknak minden körben kötelező valamelyik interakciót végrehajtaniuk mindegyik karakterrel végrehajtani, illetve ha egy karakter interakcióját jóvá hagyták, azt vissza már nem vonhatják.

Egy karakter, amennyiben elveszíti összes életerejét, eltűnik a tábláról, mezője felszabadul, illetve innentől kezdve azzal nem tud a játékos interakciót végrehajtani és nem hozhatja vissza.

A játékos célja, hogy ellenfele összes karakterét eltüntesse a tábláról. A játékot az a játékos nyeri, akinek marad legalább 1 karaktere a táblán, legalább 1 életerővel.

3. fejezet

Mesterséges intelligencia

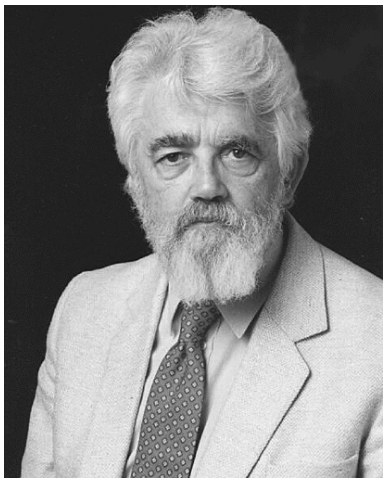
3.1. Története

A mesterséges intelligencia története az 1930-as években vett rohamos lépéstempót Alan Turing és John McCarthy munkásságával.

3.1. ábra. Alan Turing

Alan Turing (1912–1954) az 1930-as évek elején megalkotta a Turing-gépet Kurt Gödel (1906–1978) munkássága alapján, amely a modern számítógépek elméleti alapját képezte. 1950-ben publikálta a "Computing Machinery and Intelligence" című tanulmányát, amelyben felvetette a gépek gondolkodási képességének kérdését, és bevezette a híres Turing-tesztet, amely azt vizsgálja, hogy egy gép képes-e emberihez hasonló intelligens viselkedésre. [8, 9]

John McCarthy (1927—2011) amerikai számítástechnikai és kognitív tudós volt, akit a mesterséges intelligencia egyik alapítójaként tartanak számon. Ő alkotta meg a "mesterséges intelligencia" (artificial intelligence) kifejezést az 1956-os Dartmouth Konferencián, amelyet ő szervezett, és amelyet az MI hivatalos születésnapjaként tartanak számon. McCarthy 1958-ban kifejlesztette a Lisp programozási nyelvet, amely a mesterséges intelligencia-kutatás egyik legfontosabb eszközévé vált. Emellett jelentős hatással volt az ALGOL nyelv tervezésére, népszerűsítette az időosztásos rendszereket, és feltalálta a szemétgyűjtést (garbage collection).



3.2. ábra. John McCarthy

McCarthy munkássága jelentős hatással volt a mesterséges intelligencia fejlődésére. Az 1960-as és 1970-es években az MI-kutatás optimizmusa után nehéz időszak jött, amikor a fejlődés lelassult a korlátozott számítási kapacitás és a finanszírozási problémák miatt. Azonban McCarthy és kortársai kitartása hozzájárult ahhoz, hogy az MI napjainkban az egyik legdinamikusabban fejlődő területté váljon. Az utóbbi évtizedekben a gépi tanulás, a neurális hálózatok és a mélytanulás forradalmasították az MI-t, amely ma már számos területen, például az orvostudományban, az iparban és az önvezető autókban is kulcsszerepet játszik. [10]

3.2. Játékelmélet

A játékelmélet a matematika egyik, tudományágak közé egyértelműen nehezen besorolható, interdiszciplináris ága, mely olyan kérdésekkel foglalkozik, hogy mi az ésszerű (racionális) viselkedés olyan helyzetekben, ahol minden résztvevő döntéseinek eredményét befolyásolja a többi résztvevő lehetséges választásai, röviden a stratégiai problémák elmélete. A játékelmélet alapjait Neumann János fektette le ”Zur Theorie der Gesellschaftsspiele” című 1928-as munkájában [3], majd Oskar Morgenstern neoklasszikus matematikus-közgazdásszal közösen megírta a „Játékelmélet és gazdasági viselkedés”

című (The Theory of Games and Economic Behavior, 1944) művüket. [11, 12] Ezen művek alapján a következő fogalmakat tisztáznunk kell [1]:

- A *játék* a játékosok viselkedését és lényeges körülményeket meghatározó szabálysor által leírt folyamat.
- Az információs halmaz (ismeret) meghatározó szerepű. Ez azt jelenti, hogy az információs halmaz alapján különböző típusokat sorolhatunk fel, például a *tökéletes információs* és *véges* játék, ahol minden résztvevő birtokolja az összes vonatkozó adatot (szabályok, lehetséges és korábbi események).
- Egy játék lehet két- vagy többszemélyes.
- Mikor a játékban a játékosok versengenek egymással, akkor *nem kooperatív* játékról beszélünk.
- Zérusösszegű az a játék, amelyben a játékosok csak az ellenfelük nyereségük csökkentésével növelhetik nyereségüket.
- A játékost győzelemre, de minimum döntetlenre segítő módszere a *stratégia*. Ilyenkor kihasználhatja az ellenfél érzékelt hibáit.

4. fejezet

Implementáció

A 2.1. szekcióban meghatározott játékszabályok és a 3.2. szekcióban megfogalmazott játékelméleti fogalmak alapján megállapíthatjuk, hogy az implementálandó játék egy zérusösszegű, teljes információjú, kétszemélyes játék lesz. Teljes információt kielégíti, hiszen mindkét játékos ismeri a lehetséges következő lépéseket. A két játszó fél közül az egyik a felhasználó, másik játékos szerepét a mesterséges intelligencia kapja.

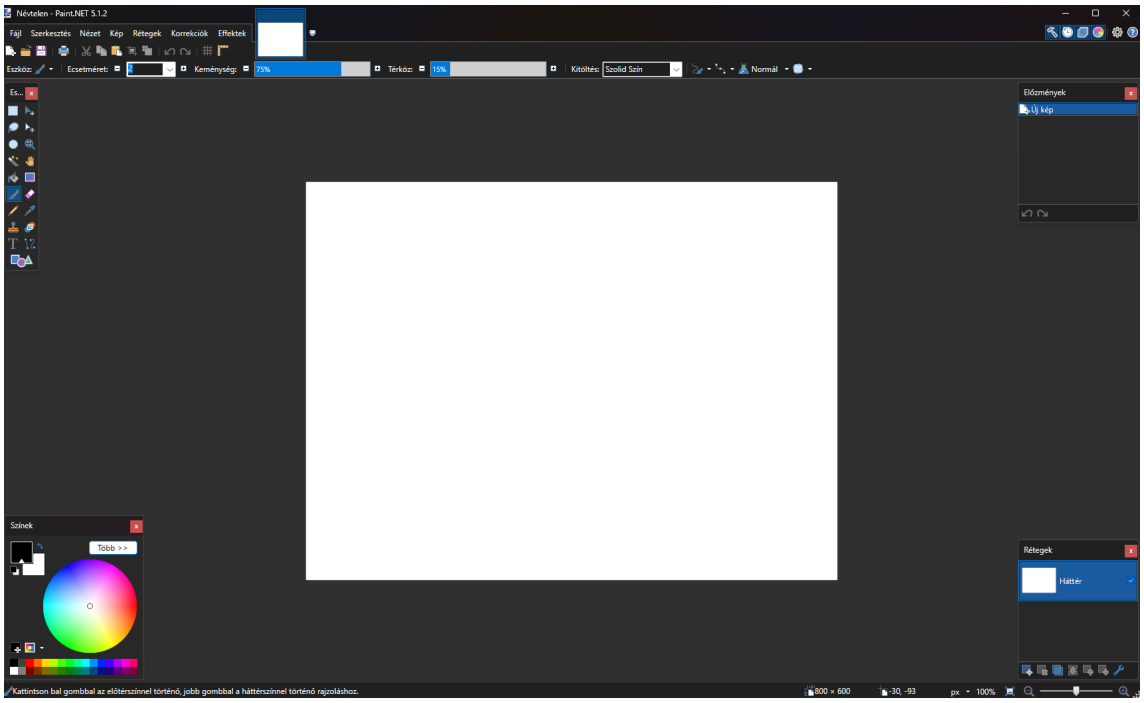
4.1. Technológiák

4.1.1. Játékmotor

A szakdolgozatom megvalósításához a Unity-t (korábban Unity3D) használom. A Unity egy világszerte ismert és használt videójáték-motor, amelyet a Unity Technologies fejleszt 2005 óta. A motor támogat több különböző platformot, például PC, videójáték konzolok és okostelefonok. Különösen kedvelik a kezdő játékfejlesztők a letisztult felülete és egyszerű használata miatt. Választásom azért esett a Unity-re, mert a szkriptekhez natívan támogatja a C# nyelvet. A szakdolgozatomban a Unity-nek a 2022.3.32f1-es verzióját használom.

4.1.2. Grafikus szerkesztő

A szakdolgozatomat a Unity-be beépített színeken és objektumokon kívül, általam készített pixelábrákat használok a karakterek képeként, amelyekhez a Paint.NET szoftvert használtam. A Paint.NET egy szabad licenszű, rasztergrafikus alkalmazás, amelyet a dotPDN fejleszt. A szakdolgozatom készítésekor 5.1.2 verzióját használtam a szoftvernek.



4.1. ábra. Paint.NET felülete

4.2. Megjelenítés

A játék készítésekor törekedtem a minimalista és könnyen vezérelhető kezelőfelületre. A Unity-ben úgynevezett *scene*-ekre lehet osztani a játékot, legközelebbi példaként a *WinForms* alkalmazásokban egy új *form* létrehozásával lehet párhuzamba hozni ezt a megoldást. A WinForms, másik nevén Windows Forms egy Microsoft által kiadott, nyílt forráskódú szoftver, amely elsődleges céljának kliens alkalmazások fejlesztésére adtak ki Windows rendszerekre. [15]

A játék három scene-ből épül fel: *MenuScene*, *GameScene* és *EndScene*.

4.2.1. MenuScene

A *MenuScene* az alap *Camera* objektumon kívül tartalmaz egy *Canvas* 2D-s objektumot. A *MenuScene* felépítése a 4.2 ábrán látható. Ezen a felületen találhatóak a különböző funkciókat betöltő objektumok. A kép bal alsó sarkában található az *ExitButton* nevű gomb objektum, amely névéhez illően kilép az alkalmazásból, felette található a *PlayButton* gomb, amely a felhasználó által kiválasztott beállítások alapján elindítja a játékot. A képernyő jobb oldalán lehet találni a beállításokat módosító gombokat és karakterek számát kiválasztó *Dropdown*[1] listákat. A gomb megnyomásával kiválaszthatja a játékos, hogy saját maga szeretné kezdeni a játékot vagy átadja a kezdés jogát a gépnek. A listákban a játékos ki tudja választani, hogy hány karakterrel induljon

[1] Legördülő lista

a játék, amely a korábban a 2.1 szekcióban említett szabályok alapján ez akár négy karakterig is terjedhet.

4.2. ábra. A játék menüjének felépítése

4.2.2. GameScene

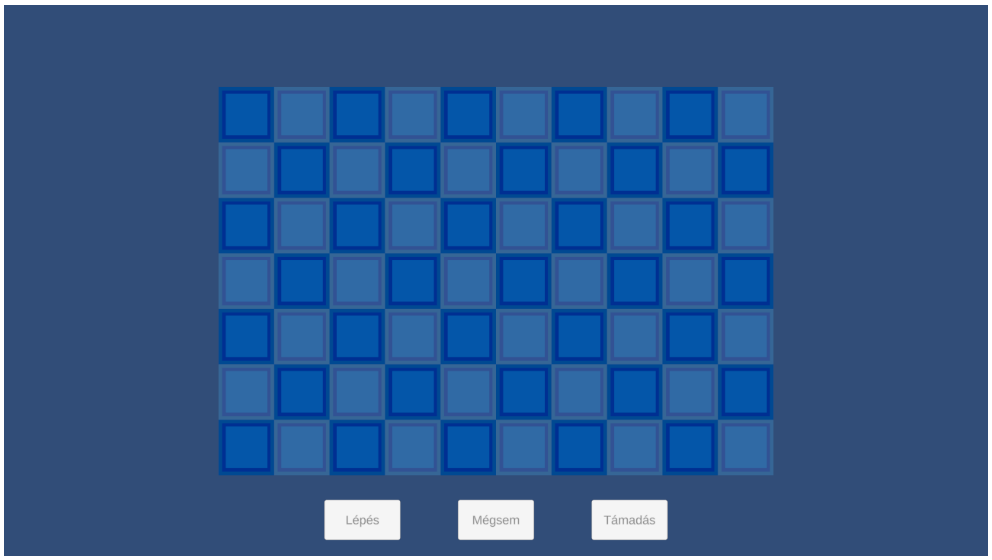
A *GameScene* az alkalmazás felületének leglényegibb része, ahol a játék folyik. A *scene* kettő objektumot tartalmaz: *GridObjectPrefab* nevű *prefab*-et és egy canvas-t.

A *prefab* akár több *GameObject*[2] egyvelege, egy *prefab*-ben a programozó tárolhat konfigurációt, mező értékeket illetve gyermek *GameObject*-eket. Természetesen ezeket újra fel lehet használni, akár kódból példányosítani, illetve törölni is lehet őket a *scene*-ből. [16]

Ebben a *prefab*-ben található a későbbi 4.3 szekcióban tárgyalt állapottér megjelenítése, amelynek reprezentációs mátrixát egy *Grid* objektummal implementálom. A *Grid* objektum magában még nem a megjelenítést oldja meg, csupán a mátrix értékeit adom át a komponensnek átalakítva, ehhez még szükséges úgynevezett *Tileset* objektumot létrehoznom. Ez a *Tileset* objektum *Tile* objektumokból áll, ami pedig kétféle négyzet alakú, 2D-s *sprite* felületet vehet fel. Ez a felület függ az objektum koordinátájától, így sakktábla (4.3 ábra) hatást keltve jelenik meg.

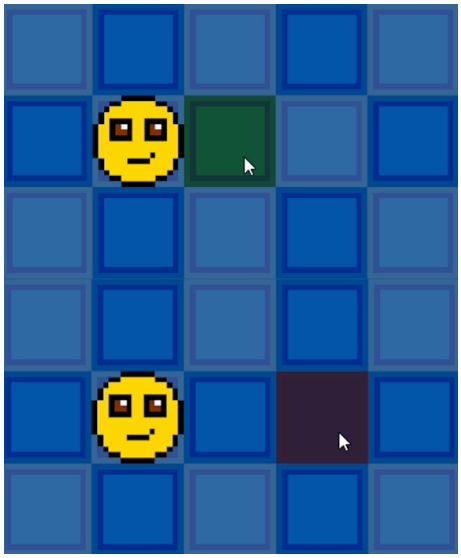
A játék indulása során az állapottérből lekérdezett pozíciók alapján ezen prefab gyermekobjektumaként példányosítom a játékos és MI karakterek prefab-jeit. Ezek a prefab-ek tárolják magukban az állapottérben is tárolt adatokat (név, életerő, pozíció), illetve a karakter tulajdonosától függő sprite-ot.

[2] A GameObject lényegében egy objektum, ami létezhet egy scene-n belül [17]



4.3. ábra. A játéktábla karakterek nélkül

A játék ideje alatt, ahogyan a karakterekkel lépnénk vagy támadnánk, a szabályoknak megfelelő módon a végrehajtható művelet céljának négyzete zöld színt kap, ha ez lehetséges, amennyiben nem, piros színt kap árnyalatnak (4.4 ábra). Amikor a játékos a felületen a kurzort mozgatja anélkül, hogy műveletet választana, akkor a négyzet, amelyen a kurzor található, szürke árnyalatot kap.

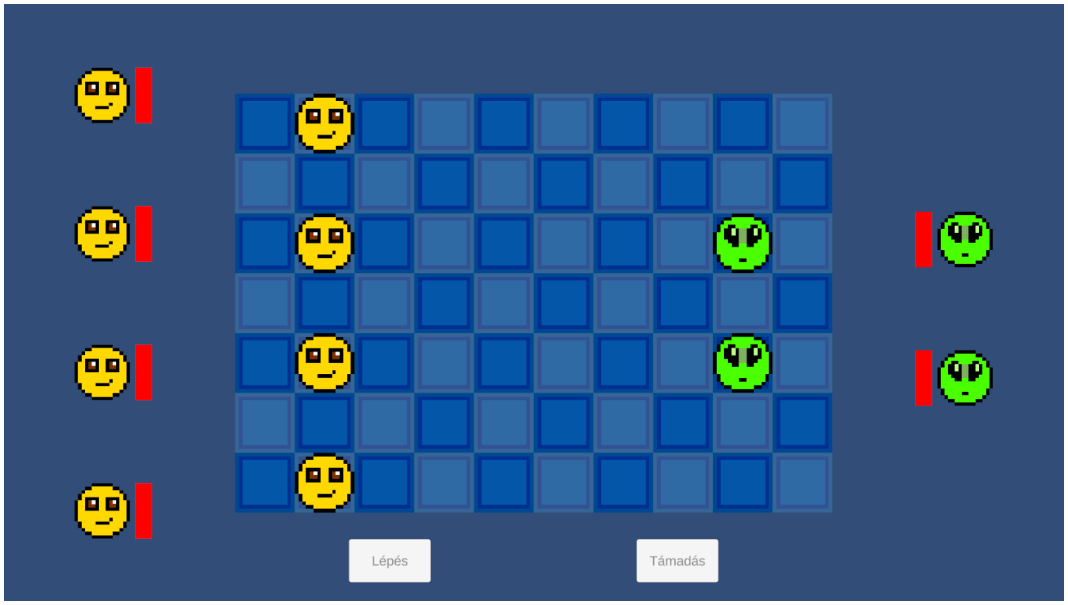


4.4. ábra. Példa helyes és helytelen lépésre

A scene másik összetevője egy canvas, amin találhatóak az életerőket tartalmazó vertikális csoportok és a vezérlő gombok. Három gomb közül kettő a játékteret befolyásoló erővel bír, a középső gomb arra szolgál, hogyha a játékos az akció végrehajtása előtt meggondolja magát, tudjon változtatni a műveleten, azonban amint kattintással jóváhagyta a játékos, ez a lehetőség megszűnik, tehát lépést nem lehet vissza vonni.

A vertikális csoport tartalma a karakterek számától függően változik, tartalmaz-

za a játékos vagy MI karaktereinek képét és két téglalap alakú 2D-s objektummal összerakott életerő csíkot, amely a karakterek életerejét mutatja. Ahogy a játék halad, amennyiben a játékos vagy MI karakterei életerőt vesztenek, a piros téglalap arányosan egyre kevesebb részt tölt ki az csíkból. Amennyiben a karakter elveszíti az összes életerejét, a prefab-je eltűnik a sávból.



4.5. ábra. Játék egyik kezdőállapota

4.2.3. EndScene

A játék befejezése után, miután az egyik játékos nyert, a kijelzőn megjelenik, hogy a nem MI játékos nyert vagy veszített. Ezen kívül egy gomb található még a képernyőn, amely kilépteti a felhasználót az alkalmazásból.

4.3. Állapottér

Az állapotterét kidolgozását a *StateRepresentation* nevű osztályban oldottam meg. Ez az osztály tartalmaz egy *FieldObject* típusú, hét sorból és tíz oszlopból álló mátrixot. A *FieldObject* osztály az ősosztálya minden táblán szereplő entitásnak, ez alatt értendők az üres mezők, karakterek vagy akadályok. Az osztály példányai rendelkeznek egy azonosítóval és egy *Vector2*[3] típusú pozícióval. Az osztály ezen kívül implementálja az *ICloneable* interfészt, ezáltal mély klónozással a 4.5 szekcióban tárgyalt minimax algoritmus fel tudja használni. Ebből az osztályból származik a *PlayerObject* osztály, amely már a játékos karakterek reprezentációját képezik a táblán. Ez az osztály az ősosztálybéli mezőkkel együtt tartalmazza még a karakterek életerejét, támadási erejét

[3] A Vector2 a Unity-n belüli megvalósítása a 2D-s vektoroknak és pontkoordinátáknak. [18]

és egy logikai változót, amely a könnyű beazonosíthatóság érdekében eltárolja, hogy azt a karaktert az MI irányítja vagy sem.

Az állapottér osztálya ezen kívül tartalmaz még a működéshez szükséges metódusokat. A *ChangeTurn()* eljárás az állapottér aktuális körét átállítja meghívás után a másik játékoséra. A *GetStatus() Status* felsorolás elemmel visszatérő függvény a játékosok karaktereinek száma alapján visszaad egy aktuális állapotot[4], amely szerint a játék jelenleg melyik fázisban tart. Lehetséges, hogy a *játékos* vagy *MI* nyert, valamint még folyamatban van a játék.

A működést befolyásoló metódusok közül az egyik legfontosabb a *GetHeruistrics()* függvény, amely visszatér a paraméterben megadott játékos állapotának heurisztikai értékével. A függvény működéséről részletesebben a 4.7 szekcióban olvashat. Ezeken kívül található több, a táblán különböző objektumok lekérdezésére szolgáló, PlayerObject és FieldObject listával visszatérő metódusok, amelyek a többi osztály számára elengedhetetlenek.

Az állapottér példányosítása során a konstruktor elindít több eljárást is, ezek végzik a tábla és akadályok generálását, a játékos és MI karakterek hozzáadását a táblához, illetve a kezdő játékos beállítását. Ezek az információk elérhetőek az *Options* statikus osztály mezőinek lekérésével. Az Options osztály értékeit a 4.2.1 alszekcióban kifejtett, a beviteli mezőkkel és gombbal tudja a felhasználó a játék indítása előtt befolyásolni.

4.4. Operátor

Az operátorokat a *TurnOperator* osztályban valósítottam meg, ami implementálja az *Operator* interfészt, melyben kettő metódus, egy logikai változóval visszatérő *IsApplicable()* és egy állapottal visszatérő *Apply()* függvény található. Mindkét metódus paraméterként egy állapotot kér be.

A játékszabály a két interakció típust tartalmaz, a lépést és támadást egyértelműen be tudjuk sorolni az operátorok közé. Mivel a szabályok alapján interakciót nem lehet kihagyni, ezért "üres" operátort nem használhatunk a játék során. Elsősorban az *IsApplicable()* függvény megvizsgálja, hogy a paraméterként adott állapot valódi állapot-e, azaz hogy a típusa StateRepresentation, amennyiben igen, típuskényszerítéssel eltároljuk, így későbbiekben könnyebben felhasználhatjuk. Ezután az összes *PlayerAction* listaelemet – amelynek felépítésről és működéséről a 4.4.1 alszekcióban részletesebben kitérek – megvizsgálva ellenőrizzük, hogy létező karakterre próbálunk alkalmazni egy operátort. Miután ebből az osztályból kiderült, hogy melyik interakcióról volt szó, meghívja az annak megfelelő segédfüggvényét, amely lépés esetén az *IsApplicableMove()*, támadás esetén az *IsApplicableAttack()* logikai függvény lesz.

[4] Nem összekeverendő az állapottér reprezentáció és minimax algoritmus által visszaadott állapottal.

Az *IsApplicableMove()* függvény a lehetséges lépéseket vizsgálja, hogy azokat lehet alkalmazni később operátorként. Ez a metódus a lépés leendő pozícióját vizsgálja, azaz azt, hogy az új pozíció továbbra is a játéktéren belül maradjon. Megvizsgálja, hogy megfelelő karakterrel lép-e, nehogy véletlenül másnak a karakterét, esetleg másik saját karaktert rakna arrébb az operátor. Megnézi, hogy az adott cella, amelyre szeretnénk lépni nem tartalmaz másik, nem "*EMPTY*" azonosítójú *FieldObject* objektumot. Végezetül eldönti, hogy az adott játékosnak a köre van vagy sem. Amennyiben ez a függvény hamis értékkel tér vissza, az *IsApplicable()* metódusban tárolt *illegalAction* változót igazra állítja, ebben az esetben a teljes függvény kiértékelése hamis lesz. Amennyiben hamis értéket ad az *illegalAction*, tehát a lépések megfelelnek a szabályoknak, attól még nincs vége az ellenőrzésnek. Megvizsgáljuk, hogy esetleg ezekben a *PlayerAction* objektumokban találhatóak azonos koordináták, amennyiben igen, az *IsApplicableMove()* kiértékelése hamis lesz. Erről szintén részletesebben a 4.4.1 alszekcióban olvashat.

Az *IsApplicableAttack()* hasonlóan a lépéshez megvizsgálja segédfüggvényekkel, hogy a célzott mező a tábla területén belül található, a végrehajtó karakter megegyezik a *PlayerAction* objektumban tárolt karakter azonosítójával, megfelelő körben szeretnék ezt az operátort végrehajtani, azonban különbség, hogy megvizsgálja a cellán található karaktert, hogy az ellenfél vagy sem, így kiszűrve, hogy bármelyik játékos a saját karakterét, üres mezőt vagy akadályt támadjon meg. A teljes függvény végkiértékelése megegyezik a lépésével, azaz amennyiben ez a függvény hamis értéket ad vissza, az *illegalAction* mező igaz lesz, tehát az *IsApplicable()* függvény visszaadott értéke hamis lesz. Bár ugyanúgy lefut a *HasSamePosition()* függvény, benne egy elágazás védi, hogy támadás esetén ne szűrje az ugyanarra mezőre indított támadásokat.

4.4.1. Operátorok felépítése

A játékszabályok megkötik, hogy egy játékos körében az összes karakterével végre kell hajtania valamilyen interakciót, emiatt a klasszikus „egyszer Te, egyszer Én" felosztás kibővül a karakterek által, egyenként végrehajtható akciókkal, amely a *PlayerAction* osztályban foglal helyet. A *PlayerAction* osztály rendelkezik egy játékos azonosító mezővel, egy *ActionType* és egy *ActionDirection* felsorolás mezővel. Az *ActionType* mező tartalma lehet *MOVE*, azaz lépés vagy *ATTACK*, azaz támadás. Az *ActionType* pedig felsorolja a négy irányt, amely a fel, le, balra és jobbra. Amikor egy *PlayerAction* típusú objektumot létrehozunk, akkor a következőképpen tesszük: megadjuk a karakter azonosítóját, amivel szeretnénk az akciót végrehajtani, az akció típusát és az akció irányát.

Ezáltal, hogy az interakciókat listaszerűen felsoroljuk az operátorokban, erőforrást spórolunk meg, ugyanis így nem szükséges minden karakter körében a játékfát újrage-

nerálnunk, hanem elég csupán a végrehajtott operátor után újraszámítanunk, hiszen összességében, ha minden karakterre külön alkalmaznánk az operátorokat, ugyanarra az állapotra jutnánk, mint az egybe alkalmazott akciókkal.

Ezzel a megoldással azonban feljöttek hibalehetőségek, mégpedig, hogy a 4.5 szekcióban kifejtett minimax algoritmus azt vélte a legjobb lépésnek, hogyha két karakterét is ugyanarra a koordinátára helyezi. Ezzel viszont adatvesztést ért el az állapottérben, hiszen az egyik karakter felül lett írva a később a mezőjét elfoglaló karakter által, így szükség volt egy ellenőrző függvényre, amely megvizsgálja, hogy az operátorokban lépés esetén ne lehessen két vagy több azonos végkoordináta, ennek segítségül szolgál a *HasSamePosition()* függvény. A függvény az aktuális játékos körétől függően kilistázza az operátor által elmozdított karaktereket. Ezeket klónozással elmozdítja az operátorban foglalt irányba és ezután ezen karakterek koordinátáját összehasonlítja. Amennyiben vannak egyező koordináták, az adatvesztést jelent, hiszen akkor egyik karakter felülírta a másikat, így ezt jelzi igaz értékkel. Amennyiben nincs egyezés, hamis értékkel tér vissza.

4.4.2. Operátor generálás

A 4.4 szekcióban és a 4.4.1 alszekcióban tárgyalt operátorokat elérhetővé kell tenni a megoldó algoritmusunk számára. Emiatt szükség van az operátorok generálására, amelyet a *TurnOperatorGenerator* osztályban oldottam meg. Az osztály egy *OperatorGenerator* interfészt implementál, amely egy *Operator* lista csak olvasható mezőt tartalmaz, ebben tároljuk el az összes operátorunkat. A generálás kódját megtekintheti a 4.1 kódrészletben.

Az operátorok legelőször a játék indulása előtt készülnek el, amikor a játékos a menüből rákattintva a ”Játék” gombra kattint. Ekkor az *Options* osztályból kinyert adatok alapján létrehozza az összes operátort a játékosnak és az MI-nek. Teszi mindezt úgy, hogy annyiszor ismétli a folyamatot, ahány játékos karakterünk van. Ezeket egy *playerActionI* nevű listában eltároljuk, ezen objektumok tartalmazzák a karakterre vonatkozó összes interakciót. Ezeket a listákat eltároljuk egy *actions* nevű másik listába. Ezen listaelemek Descartes-szorzatával létrehozzuk az összes lehetséges operátort, melyeket az *operators* listában tárolunk el. Az MI operátorainak generálása szintén így történik, ebben az esetben az *Options* osztály *enemyCount* statikus változóját veszi figyelembe.

4.1. Kód. Operátorok létrehozása a játék elején

```
1 void GeneratePlayerOperators()
2 {
3     // Összes karakter összes operátorát tartalmazó lista
4     List<List<PlayerAction>> actions = new List<List<PlayerAction>>();
5     for (int i = 0; i < Options.friendlyCount; i++)
```

```
6       {
7           // Minden karakterre legeneráljuk az összes interakciót
8           // Itt még nem operátorról beszélünk
9           List<PlayerAction> playerActionI = new List<PlayerAction>
10          {
11              new PlayerAction("PLAYER" + (i + 1), ActionType.MOVE,
                    ActionDirection.UP),
12              new PlayerAction("PLAYER" + (i + 1), ActionType.MOVE,
                    ActionDirection.DOWN),
13              new PlayerAction("PLAYER" + (i + 1), ActionType.MOVE,
                    ActionDirection.LEFT),
14              new PlayerAction("PLAYER" + (i + 1), ActionType.MOVE,
                    ActionDirection.RIGHT),
15              new PlayerAction("PLAYER" + (i + 1), ActionType.ATTACK,
                    ActionDirection.UP),
16              new PlayerAction("PLAYER" + (i + 1), ActionType.ATTACK,
                    ActionDirection.DOWN),
17              new PlayerAction("PLAYER" + (i + 1), ActionType.ATTACK,
                    ActionDirection.LEFT),
18              new PlayerAction("PLAYER" + (i + 1), ActionType.ATTACK,
                    ActionDirection.RIGHT)
19          };
20          // A karakterhez tartozó lehetséges lépéseket egy listába gyüjtjük,
                amit szintén egy listába
21          actions.Add(playerActionI);
22      }
23
24
25      // Egyszerűség kedvéért kigyújtük az egyes listák hosszát
26      List<int> lenghts = actions.Select(x => x.Count).ToList();
27      // Mivel nem fix karakterszámmal játszódik a játék, ezért kell egy
            dinamikusan generálandó indexlista
28      List<int> indexes = actions.Select(x => 0).ToList();
29      // Akkor végzünk a szorzással, amikor az első listán végig értünk
30      while (indexes[0] < lenghts[0])
31      {
32          // A lista, amit majd az operátor tartalmazni fog
33          List<PlayerAction> playerActions = new List<PlayerAction>();
34          for (int i = 0; i < actions.Count; i++)
35          {
36              // Minden al-listából kivesszük azt az elemet, ahol a lista
                    indexénél tartunk
37              playerActions.Add(actions[i][indexes[i]]);
38          }
39          // Létrehozzuk az új operátort
40          Operators.Add(new TurnOperator(playerActions));
41          // Lépünk a következő párosításra
```

```
42          for (int i = indexes.Count - 1; i >= 0; i--)
43          {
44              // Először az utolsó lista indexét növeljük meg
45              indexes[i]++;
46              // Ha még nem értünk végig a listán, vagy már az első listán
                    vagyunk, akkor a következő iterációra lépünk
47              if (indexes[i] < lenghts[i] || i == 0) break;
48              // Ellenkező esetben a lista végére értünk, ezért vissza állunk a
                     lista elejére az indexszel.
49              // A ciklus a következő iterációban megnöveli az előtte lévő
                    lista indexét eggyel
50              // (illetve ha annak a végére értünk, akkor azt is nullára
                    állítja, és növeli az előtte lévőt, ezt folytatva az első
                    listáig), ezzel biztosítva, hogy minden lehetséges kombináció
                     szisztematikusan létrehozásra kerül
51              indexes[i] = 0;
52          }
53      }
54  }
```

Játék közben megtörténhet, hogy az egyik karakter elveszíti az összes életerejét, ekkor el kell távolítani a tábláról. Az operátoraink kiértékelésének innentől nem lesz optimális, hiszen minden operátort meg kell vizsgálnia, amelyben az eltávolított karakter is szerepel. Ez annyit jelent, hogyha bár két karakter maradt a táblán a háromból, az operátor továbbra is várja a már nem létező operátorra a műveletet. Ehhez nyújt segítséget a körönként ellenőrzött karakterszám vizsgálat. Erről bővebben a 4.8.1 szekcióban olvashat. Ebben az esetben, ha a kör végén a karaktereknek számában csökkenést tapasztalunk, akkor a megmaradt karakterekre új operátorokat generálunk. Ez lényegében az operátor generálás metódusának túltöltése, ebben az esetben csak az állapottérben tárolt karakterekre generáljuk le újra az operátorokat, ezzel megoldva a szükséges interakciók számát is.

4.5. Minimax

A minimax elv olyan alkalmazott döntési szabály a játékelméletben, ami szerint azt a lehetőséget kell választani, ami minimalizálja a maximális veszteséget. Ezt az elvet felhasználhatjuk a kétfős zérusösszegű játékoknál, ami magába foglalja a két játékos szimultán döntéseit és a felváltva tett lépéseit is. [13]

Formális definíció szerint a következőképpen tudjuk felírni a számítást: [13]

$$\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i})$$

A minimax érték azt fejezi ki, hogy egy játékos legrosszabb esetben mekkora értéket

érhet el, ha a többi játékos a számára legkedvezőtlenebb stratégiát követi. Másképpen megfogalmazva, ez az a legnagyobb érték, amelyet a játékos garantáltan megszerezhet, ha ismeri a többi játékos lépéseit.

Egy kétszemélyes játékban az egyik játékos a maximalizáló, aki a saját pontszámának maximalizálására törekszik, míg a másik a minimalizáló, aki a maximalizáló pontszámának minimalizálására törekszik. Az algoritmus úgy működik, hogy kiértékeli az összes lehetséges lépést mindkét játékos számára, előrejelzi az ellenfél válaszait, és a heurisztika alapján kiválasztja az optimális lépést annak érdekében, hogy a lehető legjobb eredményt biztosítsa. [14]

A MiniMax megoldó algoritmus implementációját a *MiniMaxBase* osztályba rendeztem, amely gyermekosztálya a *Solver* absztrakt osztálynak. Ez a *Solver* osztály tartalmaz egy operátorokat tartalmazó listát és egy absztrakt, állapot paramétert bekérő *NextMove()* metódust. A *NextMove()* állapottal visszatérő függvény fog a későbbiekben a megoldó algoritmusnak megfelelően egy állapotot a játékfának kiértékelése után visszaadni.

4.2. Kód. Következő lépés logikája alfa-béta vágásnál

```
1 public override State NextMove(State state)
2 {
3     Node currentNode = new Node(state);
4     ExtendNode(currentNode, int.MinValue, int.MaxValue, currentNode.State.
          CurrentTurn);
5     return currentNode.Children[0].State;
6 }
```

Esetünkben, a MiniMax algoritmus alapján a paraméterként megadott állapotot eltároljuk egy *Node* osztályba. A *Node* osztály egy példánya lényegében a játékfának egy csomópontja, amely tartalmaz egy állapotot, a saját mélységét, a szülőcsomópontját és gyermekcsomópontok listáját, amennyiben azok léteznek. Továbbá ez a *Node* osztály az állapottérből le tudja kérdezi a játék állását. A *GetHeuristics()* metódus, amennyiben a csomópontnak nincs gyermeke, a saját állapotteréből, egyébként meg a legelső gyermekcsomópont *GetHeruistics()* függvényéből kérdezi le az állapottér heurisztikáját.

A legkisebb vagy legnagyobb csomópont kiválasztásához szintenként rendezem a fát növekvő és csökkenő sorrendbe a *SortChildrenMiniMax()* eljárás segítségével. A paraméterben megadott kör, és a másodlagos opcionális paraméter alapján eldönti, hogy a nyereséget maximalizálni vagy az ellenfél nyereségét szeretnénk minimalizálni, mindezt rekurzívan meghívva minden egyes csomópontra, amíg van gyermekcsomópontjuk.

4.3. Kód. Kiértékelt fa rendezésének logikája

```
1 public void SortChildrenMiniMax(Turn currentTurn, bool isCurrentPlayer =
      true)
2 {
```

```
3       foreach (Node node in Children)
4       {
5           node.SortChildrenMiniMax(currentTurn, !isCurrentPlayer);
6       }
7       if (isCurrentPlayer)
8       {
9           Children.Sort((x, y) => y.GetHeuristics(currentTurn).CompareTo(x.
                GetHeuristics(currentTurn)));
10      }
11      else
12      {
13          Children.Sort((x, y) => x.GetHeuristics(currentTurn).CompareTo(y.
                GetHeuristics(currentTurn)));
14      }
15  }
```

Ezek alapján a felülírt *NextMove()* metódus egy csomópontban eltárolja a paraméterben megadott állapotot, majd ezt kiterjeszti az *ExtendNode()* eljárás. Ez megnézi a csomópontban, hogy a játék még folyamatban van-e vagy elérte-e már a maximális mélységet, amennyiben igaz, nem terjesszük ki tovább. Ezután minden operátornál megnézzük, hogy alkalmazhatóak-e, amennyiben igen, úgy alkalmazzuk, és létrehozzuk az új állapotot. Ezt eltároljuk egy új csomópontba, és felvesszük az eredeti csomópont gyermekelemei közé. A létrehozott csomópontot végezetül rekurzívan tovább kiterjesztjük. Ezek után tudjuk felhasználni a fentebb említett *SortChildrenMiniMax()* függvényt.

4.6. Minimax alfa-béta vágással

Az MiniMax alfa-béta vágásos verzióját a *MiniMaxAlphaBetaPruning* osztályban valósítottam meg. A kiterjesztés logikáját a 4.4 kódrészletben találja. A lényeges eltérés az *ExtendNode()* metódus törzsének változtatása okozza, illetve a rekurzió használata miatt innentől egy egész értékkel tér vissza. Az alap MiniMax algoritmushoz hasonlóan a függvény ellenőrzi a játék folyamatban van-e vagy a mélység eléri-e a kívánt mélységet, ha valamelyik igaz, akkor visszaadja a csomópont heurisztikai értékét és leállítja a kiterjesztést. Továbbra is minden operátort megvizsgálunk, amennyiben alkalmazható az állapotra, az operátor alkalmazása után az új állapotot eltároljuk egy új csomópontként, ezt szintén felvesszük az eredeti csomópont gyermekelemei közé és azokat rendszerezzük.

A gyermekelemek kiterjesztése pedig alkalmazza az alfa-béta vágás módszerét, amely egy-egy csúcs kiterjesztését szakíthatja félbe. Ez azért ajánlatos, mert nagy lehet a játékfa és exponenciálisan felrobbanhat, így a kiértékelés rengeteg erőforrást és időt vehet igénybe. A módszer lényegében az egyes részfák kiértékelése után meg tudja állapítani,

hogy a következő részfák esetleg nem fognak optimális megoldásra jutni, így azokat „levágjuk” a játékfáról.

4.4. Kód. Csomópont kiterjesztése alfa-béta vágással

```
1  private int ExtendNode(Node node, int alpha, int beta, Turn currentTurn,
       bool currentPlayer = true)
2  {
3      if (node.State.GetStatus() != Status.PLAYING || node.Depth >= Depth)
           return node.GetHeuristics(currentTurn);
4
5      int v = currentPlayer ? int.MinValue : int.MaxValue;
6
7      foreach (Operator op in Operators)
8      {
9          if (op.IsApplicable(node.State))
10         {
11             State newState = op.Apply(node.State);
12             Node newNode = new Node(newState, node);
13             node.Children.Add(newNode);
14             node.SortChildrenMiniMax(currentTurn, currentPlayer);
15             if (currentPlayer)
16             {
17                 v = Mathf.Max(v, ExtendNode(newNode, alpha, beta, currentTurn,
                       !currentPlayer));
18                 if (v > beta) return v;
19                 alpha = Mathf.Max(alpha, v);
20             }
21             else
22             {
23                 v = Mathf.Min(v, ExtendNode(newNode, alpha, beta, currentTurn,
                       !currentPlayer));
24                 if (v < alpha) return v;
25                 beta = Mathf.Min(beta, v);
26             }
27
28         }
29     }
30
31     return v;
32 }
```

Attól függően, hogy az aktuális csomópontunk minimalizáló vagy maximalizáló körben van, összehasonlítjuk az ebből kiterjesztett részfa legmélyebb csomópontjának értékével. *Alfa-vágás* esetén a részfa és az aktuális csomópont értéke közül kiválasztjuk a minimum értéket. A következő részfa indulásakor szintén kiválasztjuk az aktuális és a részfa minimumát, azonban a további részfák kiterjesztését félbeszakítjuk, ha a

csúcsban eddig kiszámolt minimum kisebbé válik, mint a szülőben eddig kiszámolt maximum. Ebben az esetben a többi részfa kiterjesztése irreleváns, hiszen a szülő maximumának legkisebb értéke nagyobb, mint az aktuális csúcs minimumának legnagyobb értéke. [2, 5.6. fejezet]

4.7. Heurisztika

A heurisztika talán a másik legnehezebben implementálható komponense a játéknak, hiszen nem csak tisztában kell lennünk a szabályokkal, és az azzal előnyt vagy hátrányt jelentő tényezőivel, hanem tudnunk kell, hogy az ilyen típusú játékoknál mely pozíciók, szituációk, akciók okoznak nagyobb fölényt vagy visszaesést az adott játékosnak.

A heurisztikai értéket a *StateRepresentation* osztályból a *GetHeuristics()* egész értékkel visszatérő függvényből lehet elérni. Ez a függvény látja el a MiniMaxhoz szükséges heurisztikai értékkel a megoldó algoritmust. Ez a metódus több kisebb függvényekből áll, amelyek módosítják a benne lokális, *result* néven tárolt változót, amely maga a heurisztika értéke lesz.

A függvény törzse elágazásokkal kezdődik, amely vizsgálja, hogy az állapottér státusza alapján valamelyik játékos megnyerte-e a játékot. Nyilván, ez a játékfa kiterjesztése esetén hasznos, ugyanis az előre vetített lépéseknél megtörténhet, hogy egy játékos a következő lépéssel megnyeri a játékot. Mivel a játékosnak a legfontosabb nyerni, ezért mindennél több pontot kap a nyertes és veszít a másik játékos. Amennyiben a játék még folyamatban van, úgy a további kisebb számítási függvények futnak le. Attól függően, hogy éppen a minimax algoritmus saját magára vagy az ellenfelére számolja a heurisztikát, az elágazás azon ága fut le.

Ezek a függvények az állapottér helyzetéből, azaz a játékosok pozíciójából, életerejéből és lehetséges lépéseiből számítja ki a heurisztikát a következőképpen:

- *HealthBonus*: A játékos minden karakterének életerejének száma után kap 1 pontot, illetve minden ellenfél karakterének életerejének száma után veszít 1 pontot
- *AttackBonus*: A játékost jutalmazzuk, amiért próbálja az ellenfelének karaktereit eltüntetni, azonban figyelembe kell venni azt, hogy akkor ne támadjon, ha leütnék utána, ezzel hosszútávon vesztésre állna, ezért a negatív ponttal nyomatékosítjuk ilyenkor. Minden más esetben mérlegelje a támadásnak kockázatát. Ha több életereje van, mint ahány ellenfél karakter mellette, akkor nem tudnák leütni egyből, ha az ellenfelek között maradna sem. Abban az esetben lesz negatív az érték, ha olyan karakterrel szeretnénk támadni, ami veszélyesen közel van, hogy leüssék, azaz annyi életereje van, vagy kevesebb, ahány ellenfél éppen körbeveszi.
- *PlayerCount*: Megvizsgálja, hogy a játékosnak mennyi karaktere van az ellenfeléhez képest. Legtöbb esetben több karakterrel könnyebb megnyerni a játékot,

hiszen több lehetőség áll rendelkezésre, ezért a több karakterrel rendelkező játékost jutalmazzuk az ellenfelével szemben.

– *CheckPossibleMoves*: Minden lehetséges lépés további pontokat ér, amennyiben egy vagy több irányba nem tud lépni, az negatív pontot ér, mivel a játékos korlátozza karakterét. Legrosszabb lehetőség, ha valamelyik sarokba visszük a karakterünket.

– *MaxCoverageBonus*: A karakterek mindig maguk mellett 1 egység távolságra tudnak támadni. Ha a karakterek egymástól csak 2 egységre vannak, akkor két karakter tud támadni egy négyzetet, amely támadás esetén hasznos, viszont pozicionálás szempontjából megadja a lehetőséget az ellenfélnek menekülésre. Ezért érdemes inkább 3 négyzet távolságra lenni egymástól x és y koordináta szerint, hiszen ilyenkor már 2 négyzetet fedünk le minimum a támadásunkkal. Amennyiben átlósan 1 négyzet választ el, és a két karakter között egy ellenfél megállna, akkor a legoptimálisabb lépés esetén is egy megtámadható pozícióba kerülne.

– *BadPosition*: Amennyiben a játékos karaktere mellett közvetlenül található ellenfél, az ellenfelek számától függően negatív pont. Amennyiben saját karakter is található, az további negatív pont, hiszen korlátozza mindkét karakter mozgását ezzel.

– *CentralControl*: Ez az előnyt a sakkban is szokták alkalmazni, ugyanis aki a tábla közepén tartózkodik a karakterével, annak van a legtöbb lehetősége taktikázni, azaz agresszívabban támadhat, esetleg védekezhet, visszavonulhat, átrendezheti a karaktereit.

– *ConfrontationMovement*: A játék előrehaladása érdekében jutalmazzuk a játékost, ha próbál harcot kezdeményezni, illetve távolságot minimalizálni az ellenfele között. Figyelembe kell azonban venni, hogy nem mindig maradjunk közel ellenfelünkhöz, ezért kevesebb pontot adunk, mint az *AttackBonus*-ért.

4.8. További fontosabb osztályok

4.8.1. GameManager

A *GameManager* osztály a Singleton tervezési mintát használja fel alapul. Ennek az osztálynak egyetlen példánya van futási időben és ennek változóit az *Instance* mezőn keresztül lehet lekérdezni, amely a Singleton tervezési mintának legfőbb jellegzetessége. A *GameManager* osztály tartalmazza a *canvas*-on megjelenő akadály és karakter

GameObject-ek listáit. Ezen kívül működteti a UI[5] elemeket, mi esetünkben az életerősávot manipulálja az állapottérnek megfelelően, tartalmaz egy példányt az állapottérről és a megoldó algoritmusról.

Tartalmaz pár szükséges metódust is, mint például a *CheckStatus()* logikai függvényt, amely az állapottérből lekérdezi az aktuális állapotot, amennyiben valamelyik játékos győzelmet aratott, igaz értékkel tér vissza, ellenkező esetben pedig hamis értékkel, amikor még folyamatban van a játék.

Az előbbi függvényt használja a *ProgressGame()* eljárás. Lényegében ez a metódus intézi a körök felosztását, az interakciók kezelését és ezek eljuttatását az állapottérbe az operátor *Apply()* metódus használatával. Az eljárás magjában szereplő *PlayerAction* példányt létrehozza a játékos által választott interakció és kurzor pozíció alapján, amelynek működése a 4.8.2 alszekcióban össze van foglalva. Ez a metódus egy körben annyiszor fut le, ahány karaktere van a játékosnak. Mivel operátornak egy kör összes interakciója egybevéve felel meg, ezért a játékos lépéseinél vizsgálni kell, hogy az aktuális interakciójával megnyeri a játékot vagy sem. Emiatt minden *PlayerAction* létrehozása után megvizsgáljuk a *GameManager*-ben tárolt *enemies GameObject* listának a hosszát, amennyiben ez nulla, a játékos nyert, és átirányítjuk a képernyő képét az *EndScene*-re. Amennyiben nem nyert a játékos, a *RedrawTable()* metódus átalakítja az életerősávot és pozíciót az interakcióknak megfelelően és az ismétlések után létrehozzuk az operátort szintén az interakciók alapján. Ezt alkalmazzuk az állapottéren, a játékos köre pedig a lokális változók alaphelyzetbe állítása után véget ér.

Ezután vizsgáljuk a 4.4.2 alszekcióban taglalt operátorok újragenerálásának szükségességét. Amennyiben az eljárásunk előtti karakterszám eltér az eljárás utáni állapottól, a *solver* példányunkat kicseréljük egy új példánnyal, melyben az operátorokat már a módosult karakterkészlet alapján hozzuk újra létre. Ez után érkezik az MI köre, ahol szimplán meghívjuk a megoldó algoritmus *NextMove()* metódusát. Amennyiben az állapottér státuszából kiderül, hogy nyert az MI, átirányítjuk a képernyő képét az *EndScene*-re, ellenkező esetben ismételten felülvizsgáljuk az operátorok újragenerálásának igényét. Mindezek után végrehajtjuk a módosításokat a *canvas*-on és átadjuk a játékosnak a kört. Ezzel a folyamat kezdődik elölről.

A *RedrawTable()* eljárás összehasonlítja a *GameManager* karakterlistáit az állapottérben található karakterlistákkal, amennyiben a korábbiban olyan karakter van, ami az állapottérben már nem létezik, azt eltávolítja a *canvas* táblájáról és példányát az életerősávból is. Szintén ez a metódus kezeli, ha valamely karakternek változik az életereje vagy pozíciója.

[5] User Interface, magyarul felhasználó felület, esetünkben a játék felülete

4.8.2. TileSelector

Ez az osztály végzi az interakciókhoz tartozó koordináták kiválasztását. A *SelectTile()* metódus minden képkockán ellenőrzi a bal egér gombjának lenyomását, ezzel elindítva egy folyamatot: a kurzor pozícióját felhasználva, a *scene*-n belüli globális pozícióját kiszámolva meg tudjuk állapítani, hogy melyik *Tile* található ezen a pozíción, így ezt eltárolva felhasználjuk. A *GameManager*-ben található *actionSelected* változó értéke alapján eldöntjük, hogy lépés vagy támadás lesz az interakció, a játékos karaktere és a kiválasztott cella pozíciója alapján pedig az interakció irányát tudjuk meghatározni. Ezután a karakter pozícióját a komponensében és pozíciójában is átmozgatom, és meghívom a *ProgressGame()* metódust. Támadás esetén a különbség annyi, hogy a kijelölt cellán található karakter életerejét csökkentem, amennyiben elfogyna, azaz nulla lenne, eltávolítom a listából és kitörlöm a *GameObject* példányát. Végül engedélyezem újra az interakció kiválasztó gombokat.

Természetesen, mint az operátoroknál, a felületen is vizsgálni kell, hogy a felhasználó legális lépéseket vagy támadásokat akar-e végrehajtani, így kizárólag csak azokat a műveleteket engedélyezem interakcióként elfogadni, amelyek a játék szabályainak megfelelnek.

4.5. Kód. Cella kiválasztása és továbbítása

```
1  public void SelectTile()
2  {
3      if (Mouse.current.leftButton.wasPressedThisFrame)
4      {
5          Vector2 mousePos = Mouse.current.position.ReadValue();
6          Vector3 worldPos = cam.ScreenToWorldPoint(new Vector3(mousePos.x,
               mousePos.y, cam.nearClipPlane));
7          worldPos.z = 0;
8
9          Vector3Int currentCell = tilemap.WorldToCell(worldPos);
10         Vector3Int playerCell = tilemap.WorldToCell(GameManager.Instance.
               friendlies[GameManager.Instance.activePlayer].transform.position)
               ;
11         if (GameManager.Instance.actionSelected == ActionSelected.MOVE)
12         {
13             if (IsValidMove(currentCell, playerCell))
14             {
15                 GameManager.Instance.actionDirection = GetDirection(
                       currentCell, playerCell);
16                 GameManager.Instance.friendlies[GameManager.Instance.
                       activePlayer].GetComponent<EntityStat>().position =
                       SetNewObjectPosition(GameManager.Instance.actionDirection)
                       ;
17                 GameManager.Instance.friendlies[GameManager.Instance.
```

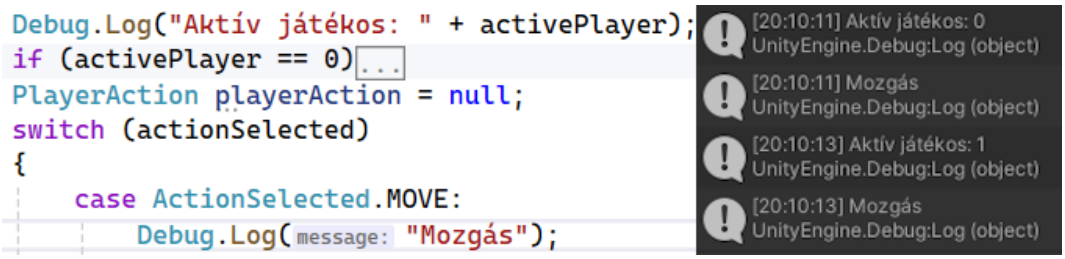
```
                activePlayer].transform.position = tilemap.
                    GetCellCenterWorld(currentCell);
18              GameManager.Instance.ProgressGame();
19              GameObject.Find("Canvas").GetComponent<GameButtonScript>().
                    EnableTurnButtons();
20          }
21      }
22      else if (GameManager.Instance.actionSelected == ActionSelected.ATTACK
            )
23      {
24          if (IsValidAttack(currentCell, playerCell))
25          {
26              GameObject enemy = SelectEnemy(currentCell);
27              GameManager.Instance.actionDirection = GetDirection(
                    currentCell, playerCell);
28              enemy.GetComponent<EntityStat>().health -= GameManager.
                    Instance.friendlies[GameManager.Instance.activePlayer].
                    GetComponent<EntityStat>().attack;
29              if (enemy.GetComponent<EntityStat>().health <= 0)
30              {
31                  GameManager.Instance.enemies.Remove(enemy);
32                  Destroy(enemy);
33              }
34              GameManager.Instance.ProgressGame();
35              GameObject.Find("Canvas").GetComponent<GameButtonScript>().
                    EnableTurnButtons();
36          }
37      }
38  }
39  }
```

5. fejezet

Tesztelés

Egy alkalmazás fejlesztése során elengedhetetlen a különböző funkciók, algoritmusok és kódrészletek tesztelése. A tesztelés tökéletes módja, hogy ellenőrizzük a programunk működését, azt, hogy az üzemi kód eredményei a követelményekben megfogalmazott értékeket adják, valamint javítási lehetőségek találására. Általában a legegyszerűbb módja a tesztelésnek a manuális tesztelés. Ebben az esetben automatizált folyamatok nélkül használjuk az alkalmazásunkat, vagy adjuk ki tesztelőknek és tesszük próbára különböző szituációkban.

A manuális tesztelésnek is több fajtáját használjuk. *Feketedobozos* tesztnek nevezzük, amikor a specifikációnak megfelelő bevitelre, a meghatározott eredményt kapjuk-e, eközben pedig nem ismerjük a forráskódot, illetve hogy az futás után milyen eredményt és milyen lépéseket végzett el. *Fehérdobozos* tesztnek nevezzük, mikor a forráskódot ismerjük, ezért az esetleg problémás kódrészletet tudjuk ellenőrizni, futási időben pedig működését vizsgáljuk. *Szürkedobozos* tesztek az előző kettő technikáját vegyíti. Szakdolgozatom írása során fehérdobozos tesztet, a közösségi tesztelők pedig feketedobozos teszt technikáját alkalmazták. [19, 20, 21]



5.1. ábra. Debug loggolás használata

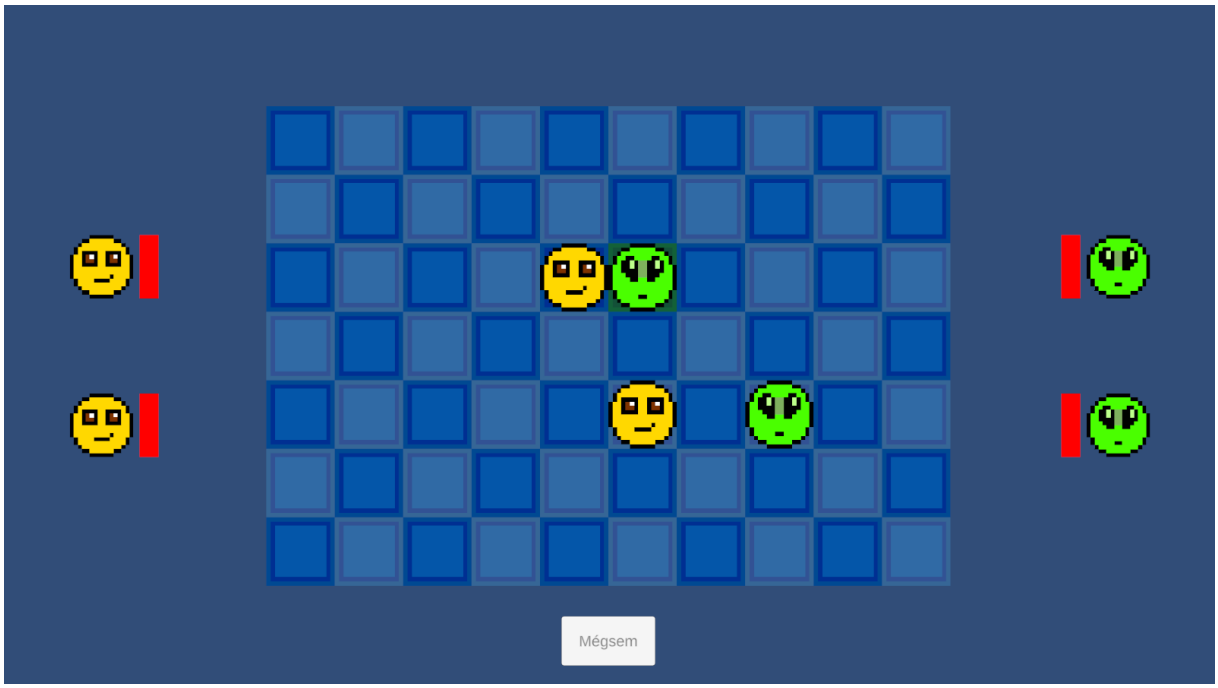
A Unity keretein belül a különböző metódusok és szkriptek tesztelésére az egységtesztek alkalmazása körülményes, ezért az alkalmazásom tesztelése javarészt manuálisan, próba-hiba módszerrel történt. Amikor megírtam egy metódust, megnéztem hogyan reagál a játék, ha valami elromlott, akkor a Unity-be beépített *Debug* osztály *Log()* metódusát alkalmaztam, amelyet a konzolra írja a függvény paraméterét. [22]

A tesztelés során több különböző szempont alapján vizsgáltam a játékomat. Elsődleges prioritást kapott, hogy játék során ne legyenek kivételek, illetve ne álljon le váratlanul a program. A játék kezdeti fejlesztésében a játékfolyamat készítésekor előfordult, hogy végtelen ciklusba került, ezért kényszeríteni kellett leállításra. Örülök, hogy az ilyesfajta problémákat sikerült kiszűrni a szakdolgozati verzióban.

Szintén hasonlóan fontos volt a különböző funkciók és felhasználói felület elemek működése, hiszen ezek nélkül a játék szinte játszhatatlanra romolhat. Ezek közül a legfontosabb az interakciógombok és játéktábla cellaválasztó osztályok működése volt. Fejlesztés során előfordult, hogy az interakció gombok nem jelentek meg, illetve nem tűntek el a körnek megfelelően. Ezek a problémák nincsenek jelen ebben a verzióban.

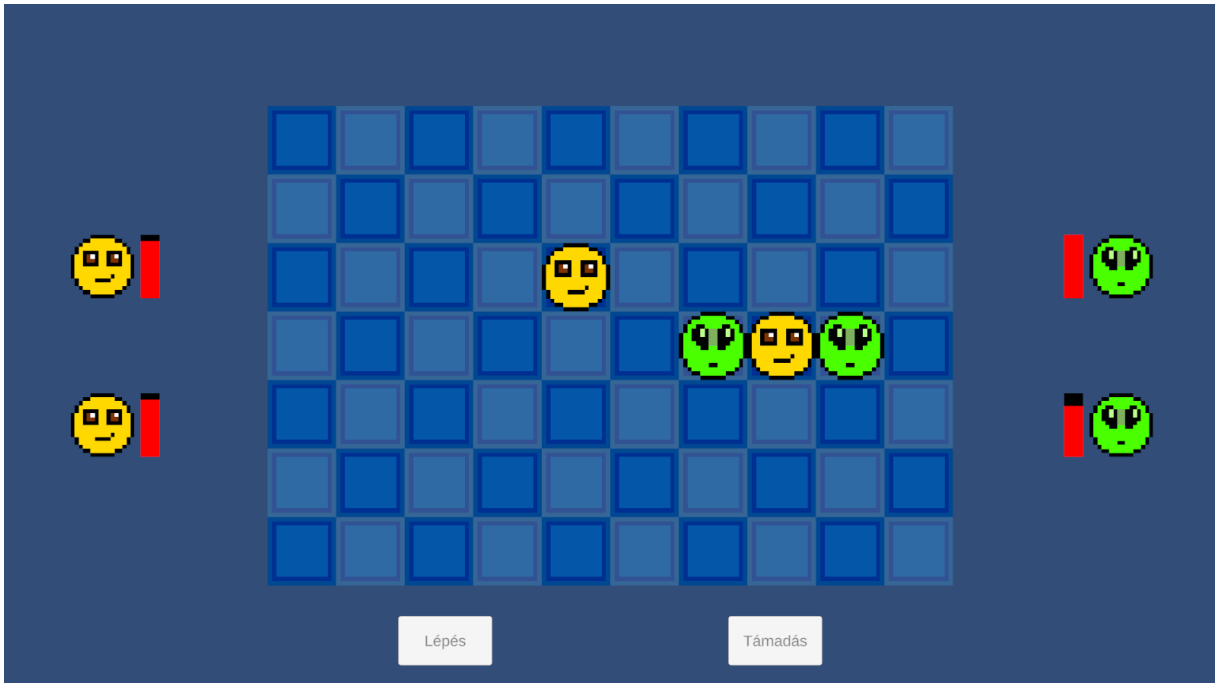
5.2. ábra. Unity pánik

A játékszabályok alkalmazása és ellenőrzésének tesztelése a fejlesztés folyamán folyamatosan szükségesnek bizonyult, ugyanis egy nem megfelelő operátor alkalmazás a szabályok megszegéséhez, rosszabb esetben kivételhez vezettek. Ezek memória allokációs problémákat okoztak, amely a Unity játékmotort teljes pánikba hozták, amelyet az 5.2. ábra szemléltet. Természetesen manuális teszteléssel csak sok idő után lehetünk biztosak, hogy a programunk betartja a specifikációkat, de a hiba lehetősége mindig fenn állhat emberi figyelmetlenség miatt. A tesztelések során ilyen hibával nem találkoztam.



5.3. ábra. Találkozás a centrumban az ellenféllel

Miután a szabályok alkalmazása sikeres volt, a funkciók működtek, és nem kaptam sehol kivételt, elkezdtem a heurisztikát és az MI-t tesztelni. Ezt úgy viteleztem ki, hogy megpróbáltam különbözőképpen, más stílussal játszani, például folyamatosan az ellenfél felé lépek, támadom minden lehetőségen, menekülök előle. Ebben az esetben már vitatható eredményt kaptam. A heurisztika, mint ahogyan a 4.7. szekcióban is említettem, a legnehezebben implementálható egysége a játéknak, ezért nehéz is jó heurisztikát írni. Jelen esetben azt a következtetést vonhatom le, hogy az MI ezzel a heurisztikai modellel passzívan játszik, sokszor látszólag előnyös helyzetben sem támadja meg a karaktereimet. Ez természetesen javítható, amennyiben a heurisztikát finomítjuk, illetve a támadási pontszorzókat arányosan növeljük a többi szemponthoz képest. Ha túl sokat adnánk, akkor lehetséges, hogy túl agresszívvá válna és olyan helyzetekben is támadna, ami számára már kedvezőtlen szituációkhoz vezetne. Tehát a heurisztika nem tökéletes, azonban játszható annyira, hogy egy kezdő és közepesen játszó játékos lépéseire képes.



5.4. ábra. Az ellenfél közrefogja a karakteremet

Sajnos kiemelendő, hogy amennyiben mindkét játékos több mint 2 karakterrel rendelkezik, a számítási idő elszáll, ezt az állapottér exponenciális robbanása okozza. Ilyenkor már a karakterekre az adott mélységen már annyi számítást kell végeznie a mesterséges intelligenciának, hogy a játék megáll ideiglenesen, ilyenkor olyan érzést kelt, mintha ”kifagyott” volna. Azonban ilyenkor csak éppen értékeli ki a játékfát, pár másodperc után visszatér az új állapottal.

Néhány embert megkértem, hogy teszteljék a játékomat, amelyről véleményüket egy *Google Forms* kérdőív kérdésein keresztül leírhatták. Fontos megjegyezni, hogy a

tesztelők nem ismerték a forráskódot, illetve a játék működését, azaz feketedobozos tesztnek számítotható, csupán elküldtem nekik az alkalmazást, hogy játszanak vele, a kérdőívet amikor befejezték a játékot. A kérdések a kérdőíven a következők voltak:

1. *Játszott valaha körökre osztott stratégiai játékkal?* - igen vagy nem válasz lehetőség

2. *Mennyire könnyen kezelhető Önnek a játék felülete?* - 1-től 5-ig terjedő skálán értékelhette a játék felületét

3. *Mi a véleménye az ellenfél viselkedéséről?* - Itt tájékoztattam a tesztelőt, hogy az ellenfél MI által van vezérelve, és hosszú szöveges válaszban értékelnie kell a működését.

4. *Találkozott bármilyen hibával a játék során? Ha igen, kérem írja le a hibát!* - Itt amennyiben valamilyen hiba elkerülte a figyelmemet, a tesztelők le tudták írni.

A szakdolgozat írásakor a válaszadók többségének feltűnt az MI korábban említett passzivitása, amely betudható a kevésbé agresszív heurisztikának. Ezen kívül kellemetlen tényezőnek gondolták három a három ellen feletti várakozási időt, azonban további szélsőséges hibákkal nem találkoztak.

A közösségi tesztelés nyers, táblázatos eredményei elérhetőek a következő linken: `https://drive.google.com/drive/folders/1C-2cGemq0t4ECZmj0p5VrQ7pdp0V6QhS?usp=sharing`

Összegzés

A szakdolgozatomban megvalósítottam az általam elképzelt játékot, amely egy remek kihívás volt, az eddig autodidakta módon tanult Unity tudásom teszteléséhez. Bár nem a legmélyebb komponensekbe nyúltam bele, így is tökéletesen mutatja, hogy hogy a játékfejlesztésbe ezzel a játékmotorral akár kellő ambícióval is bele tudunk vágni, nem szükséges hozzá több évnyi tanulás, hogy alkotni tudjunk vele.

A játékról összességében az emberek a közösségi teszt alatt jól vélekedtek, legtöbbet említett hibának a számítási időt említették, ugyanis ilyenkor azt hitték, befagyott a játék, ezt mindenképpen szeretném a jövőben javítani.

A játék készítése során pár váratlan problémába is ütköztem, de ezek a problémák rávilágítottak a szakmai hiányosságaimra, illetve a problémamegoldó képességemet tették próbára, így a dolgozat készítése során ezen téren is fejlődni tudtam. Jó érzés volt, hogy bár sok fajta probléma merült fel a fejlesztés során, ezeket sikerült leküzdeni, ezzel is hozzá járulva ahhoz, hogy egy jobb produktum készülhessen el.

Ez azonban nem azt jelenti, hogy hibátlan a játék. A jövőben tervezem a heurisztikát finomítani, tökéletesíteni, hogy az MI interakciói a lehető legjobbak legyenek, valamint a kódban már előkészített, különböző típusú ellenfelek implementálását is, amelyet eltérő *sprite*-juk alapján lehet megkülönböztetni, ezek különböző erőponttal, életerővel és támadási távolsággal rendelkeznének.

Köszönöm szépen az Eszterházy Károly Katolikus Egyetemnek összes oktatójának, akik tudásukat a lehető legjobban átadták számomra a tanulmányaim során, az egyetem hallgatóinak, akik segítettek tanulmányi és hétköznapi ügyekben. Külön köszönöm a szakdolgozatom készítése során konzulensem, Dr. Kovásznai Gergely szakmai támogatását, aki nélkül ez a szakdolgozat nem jöhetett volna létre ebben a formában.

Irodalomjegyzék

[1] Neumann János, Oskar Morgenstern: *Theory of Games and Economic Behavior*, 1944. https://archive.org/details/in.ernet.dli.2015.215284/page/n5/mode/2up, Megtekintés dátuma: 2025.03.19.

[2] Dr. Kovásznai Gergely és Dr. Kusper Gábor: *A MESTERSÉGES INTELLIGENCIA KÉRDÉSEI A KÖZÉPISKOLAI OKTATÁSBAN* https://aries.ektf.hu/~gkusper/mesterseges_intelligencia.v.1.0.4.pdf, Megtekintés dátuma: 2025.03.28.

[3] Neumann János: *On the theory of games of strategy*, Sonnya Bargmann fordítása, 1928. https://cs.uwaterloo.ca/%7Ey328yu/classics/vonNeumann.pdf, Megtekintés dátuma: 2025.03.19.

[4] Wikipédia: *Persona (series)* https://en.wikipedia.org/wiki/Persona_(series), Megtekintés dátuma: 2025.03.19.

[5] Wikipédia: *Persona 3 Reload* https://en.wikipedia.org/wiki/Persona_3_Reload, Megtekintés dátuma: 2025.03.19.

[6] Wikipédia: *Revelations: Persona* https://en.wikipedia.org/wiki/Revelations:_Persona, Megtekintés dátuma: 2025.03.19.

[7] YouTube: *Persona: Mastering the Formation System* https://www.youtube.com/watch?v=t_FPK84jcbE, Megtekintés dátuma: 2025.03.19.

[8] Wikipédia: *Alan Turing* https://en.wikipedia.org/wiki/Alan_Turing, Megtekintés dátuma: 2025.03.19.

[9] Wikipédia: *Computing Machinery and Intelligence* https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence, Megtekintés dátuma: 2025.03.19.

[10] Wikipédia: *John McCarthy (computer scientist)* https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist), Megtekintés dátuma: 2025.03.19.

[11] Wikipédia: *Játékelmélet* https://hu.wikipedia.org/wiki/J%C3%A1t%C3%A9kelm%C3%A9let, Megtekintés dátuma: 2025.03.19.

[12] Wikipédia: *Game theory* https://en.wikipedia.org/wiki/Game_theory, Megtekintés dátuma: 2025.03.19.

[13] Wikipédia: *Minimax* https://en.wikipedia.org/wiki/Minimax, Megtekintés dátuma: 2025.03.19.

[14] GeeksforGeeks: *Mini-Max Algorithm in Artificial Intelligence* https://www.geeksforgeeks.org/mini-max-algorithm-in-artificial-intelligence/, Megtekintés dátuma: 2025.03.19.

[15] Wikipédia: *Windows Forms* https://en.wikipedia.org/wiki/Windows_Forms, Megtekintés dátuma: 2025.03.27.

[16] Unity Documentation: *Prefabs* https://docs.unity3d.com/2022.3/Documentation/Manual/Prefabs.html/, Megtekintés dátuma: 2025.03.26.

[17] Unity Documentation: *GameObject* https://docs.unity3d.com/2022.3/Documentation/Manual/class-GameObject.html, Megtekintés dátuma: 2025.03.27.

[18] Unity Documentation: *Vector2* https://docs.unity3d.com/2022.3/Documentation/ScriptReference/Vector2.html, Megtekintés dátuma: 2025.03.27.

[19] Wikipédia: *Black-box testing* https://en.wikipedia.org/wiki/Black-box_testing, Megtekintés dátuma: 2025.03.29.

[20] Wikipédia: *White-box testing* https://en.wikipedia.org/wiki/White-box_testing, Megtekintés dátuma: 2025.03.29.

[21] Wikipédia: *Gray-box testing* https://en.wikipedia.org/wiki/Gray-box_testing, Megtekintés dátuma: 2025.03.29.

[22] Unity Documentation: *Debug* https://docs.unity3d.com/2022.3/Documentation/Manual/class-Debug.html, Megtekintés dátuma: 2025.03.29.

Nyilatkozat

Alulírott *Herbák Marcell*, büntetőjogi felelősségem tudatában kijelentem, hogy az általam benyújtott, *Körökre osztott stratégiai játék készítése MiniMax alfa-béta vágó algoritmus használatával* című szakdolgozat önálló szellemi termékem. Amennyiben mások munkáját felhasználtam, azokra megfelelően hivatkozom, beleértve a nyomtatott és az internetes forrásokat is.

Tudomásul veszem, hogy a szakdolgozat elektronikus példánya a védés után az Eszterházy Károly Katolikus Egyetem könyvtárába kerül elhelyezésre, ahol a könyvtár olvasói hozzájuthatnak.

Eger, 2025. április 13.

Herbák Marcell

aláírás