**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Numerikus Analízis** **Tanszék**

**Reaktív mesterséges intelligencia megvalósítása videojátékban**

**Témavezető:**

Fábián Gábor

tanársegéd

**Szerző:**

Barton Patrik

Programtervező informatikus BSc.

Budapest, 2020

Tartalom

[1. Bevezetés 4](#_Toc40529817)

[2. Felhasználói Dokumentáció 5](#_Toc40529818)

[2.1 A program által megoldott feladat 5](#_Toc40529819)

[2.2 A játék jellemzői 5](#_Toc40529820)

[2.2.1 A játék története, alaphelyzete 5](#_Toc40529821)

[2.2.2 Karakterek 6](#_Toc40529822)

[2.2.3 Játéktér 7](#_Toc40529823)

[2.3 A program alapvető működése 7](#_Toc40529824)

[2.3 Felhasznált módszerek áttekintése 8](#_Toc40529825)

[2.3.1 Wavefront algoritmus 8](#_Toc40529826)

[2.3.2 Monte Carlo Tree Search algoritmus 9](#_Toc40529827)

[2.4 A program használata 9](#_Toc40529828)

[3. Fejlesztői Dokumentáció 11](#_Toc40529829)

[3.1 Specifikáció, követelmények 11](#_Toc40529830)

[3.2 Használati esetek 11](#_Toc40529831)

[3.3 Algoritmusok 11](#_Toc40529832)

[3.3.1 Wavefront algoritmus 11](#_Toc40529833)

[3.3.2 Monte Carlo Tree Search algoritmus 12](#_Toc40529834)

[3.4 Programozási környezet, eszközök 15](#_Toc40529835)

[3.4.1 Unity (2019.3.6f1) 15](#_Toc40529836)

[3.4.2 Tiled 18](#_Toc40529837)

[3.5 A program felépítése 20](#_Toc40529838)

[3.5.1 Unity, játékobjektumok, játéktér 20](#_Toc40529839)

[3.5.2 C#, osztályok, metódusok 21](#_Toc40529840)

[3.5.2.1 Character 22](#_Toc40529841)

[3.5.2.2 PlayerController 25](#_Toc40529842)

[3.5.2.3 AltarezController 27](#_Toc40529843)

[3.5.2.4 DemonMageController 29](#_Toc40529844)

[3.5.3 Kapcsolatok 30](#_Toc40529845)

[3.6 Tesztek 30](#_Toc40529846)

[3.6.1 Elvárt működés 30](#_Toc40529847)

[3.7 Továbbfejlesztési lehetőségek 30](#_Toc40529848)

[4. Összefoglalás 31](#_Toc40529849)

[5. Fogalomtár 32](#_Toc40529850)

# Bevezetés

A legtöbb mai videojátékban a számítógép által irányított karakterek Mesterséges Intelligencia algoritmusa nem veszi figyelembe, hogy a játékos karakterének milyen lehetőségek állnak rendelkezésére az adott pillanatban ahhoz képest, hogy összesen milyen képességei lehetnek.

Vegyünk például egy általános, online szerepjátékot. A játékos a játék elején tipikusan kiválasztja a karaktere alapvető tulajdonságait, többek közt a kasztját. Ennek függvényében a karaktere lehet egy mágus, íjász, esetleg lovag. A választott kaszttól pedig nagymértékben függ a játék során szerezhető képességek és felszerelések típusa. Mindezek a játékstílust alapvetően változtathatják meg a játékos számára, az ellenfelek viszont tipikusan ezt figyelmen kívül hagyják. Ugyanúgy reagálnak egy közelharcos karakterre egy pajzzsal, aki ellen egy emberi játékos esetleg oldalba támadást alkalmazna a pajzs megkerülése érdekében, és egy távolharcos íjász karakterre, akit egy emberi játékos valószínűleg a lehető leggyorsabban meg akarna közelíteni, hogy ne használhassa hatékonyan a fegyverét.

Ennek a programnak a célja, hogy egy olyan Mesterséges Intelligenciával lássa el a játékos ellenfeleit, amely reagál a játékos által elérhető lehetőségekre. Ha a játékos használja a képességei egyikét, az ellenfelek tudomást szereznek róla, hogy a játékos hozzáfér az adott lehetőséghez. Ennek tudatában az MI által irányított ellenfelek más döntéseket hozhatnak, annak függvényében, hogy a játékos milyen képességeket ér el.

Ezen megoldás előnye, hogy a játékélményt sokkal színesebbé és izgalmasabbá tehetné. A játékosoknak minden eltérő tulajdonságú karakter esetében fel kell készülniük, hogy az ellenfeleik is máshogy fognak viselkedni. Emellett azon is érdemes lehet elgondolkodniuk, hogy egy adott képességet ne használjanak túl korán, így taktikai előnyt szerezve a meglepetés erejével.

Eme program elkészítésével célom volt még az is, hogy gyakoroljam a Unity és a C# használatát, mivel régóta ki szerettem volna próbálni mindkettőt, de eddig nem sikerült sort kerítenem erre.

# Felhasználói Dokumentáció

A következőkben ismertetjük a program által megoldandó feladatot, majd a felhasználó szempontjait szem előtt tartva áttekintjük a megoldás elméleti hátterét. Végül részletesen bemutatjuk a program használatát, a játékos lehetőségeit és a Mesterséges Intelligencia által irányított ellenfelek várt működését.

## 2.1 A program által megoldott feladat

A mai videojátékok nagyrészében a játékos által irányított karakteren kívül szerepelnek „Nem-Játékos Karakterek” is. Ezekre a dolgozat hátralevő részében NPC néven hivatkozunk.

Ezek a karakterek tipikusan Mesterséges Intelligencia algoritmusok alapján cselekednek. Általában beállításoktól, vagy az adott karakter alapvető tulajdonságaitól függ, hogy az MI algoritmus milyen fajta viselkedést imitál. Egy magas védekezési értékekkel rendelkező NPC például lehet, hogy igyekszik elsőnek megütközni az ellenfelével, ezzel megpróbálva magára vonni az ellenség figyelmét a kevésbé strapabíró társai helyett. Jellemzően viszont ennek a karaktertípusnak minden helyzetben ugyanez lesz a stratégiája, attól függetlenül, hogy hány, milyen lehetőséggel bíró ellenfele van.

Ebben a programban egy ismert MI algoritmust, a *Monte Carlo Tree Search* -innentől MCTS - módszert alkalmazzuk. Ezt felhasználva mutatjuk be, hogy ugyanaz az ellenfél másképpen reagálhat annak tudatában, hogy a játékos karaktere milyen képességekkel rendelkezik.

## A játék jellemzői

### A játék története, alaphelyzete

Clestia világát démonok támadták meg! Az alvilági erők a természet megrontói, az emberek mészárosai. Egy eldugott tisztáson a démonok egy portált kezdtek előkészíteni, mellyel még több fajtársukat tudnák az emberek világára szabadítani. Két hősünk Amy és Altarez, és jelenleg csupán ők ketten állnak a világuk és az azt fenyegető gonosz között! Szánt szándékuk, hogy a két Démon Mágust, akik a portál létrehozásához szükségesek, végleg eltegyék láb alól, megakadályozva ezzel hőn szeretett világuk pusztulását.

### Karakterek

A két hős karakter Amy és Altarez, ellenfeleik pedig két Démon Mágus.

Amy egy íjjal rendelkezik. Ennek megfelelően az alap támadása fel, le, jobbra vagy balra egy szimpla nyílvesszőt lő ki, 7 mező távolságig.

Képes még forgószeleket megidézni is. Ez a megadott irányban egy három mező széles, három mező hosszú sávban minden ott talált ellenfélre kilő egy forgószelet.

Ezen kívül maga körül tüskéket tud előhívni a földből, melyek egy 5x5-ös mezőben minden ellenfelet sebeznek.

Altarez fegyvere egy kard, mellyel maga körül mind a nyolc irányba tud támadni. Képes varázsolni is.

Az egyik képessége meggyógyítja a Játékoskaraktert, amennyiben a közelében van, beleesik egy Altarez középpontú 5x5-ös mezőbe.

A másik képessége egy, a közelben lévő ellenségre egy mágikus kígyót idéz, mely lecsap, sebezve a kijelölt ellenfelet. Ehhez az ellenfélnek az Altarez körüli, 7x7-es mezőben kell lennie.

A Démon Mágusok varázslataikkal támadnak. Alapvető támadásuk egy tűzgolyó kilövése, melyet fel, le, jobbra vagy balra tudnak kilőni, öt mező távolságig.

Első képességük egy lángcsóva, melyet egy, hozzájuk képest átlósan maximum öt mező távolságra elhelyezkedő ellenfeleikre varázsolnak.

Második képességük Démoni tüskék idézése a föld alól, mely minden, a körülöttük 5x5-ös mezőben tartózkodó ellenfelüket sebzi.

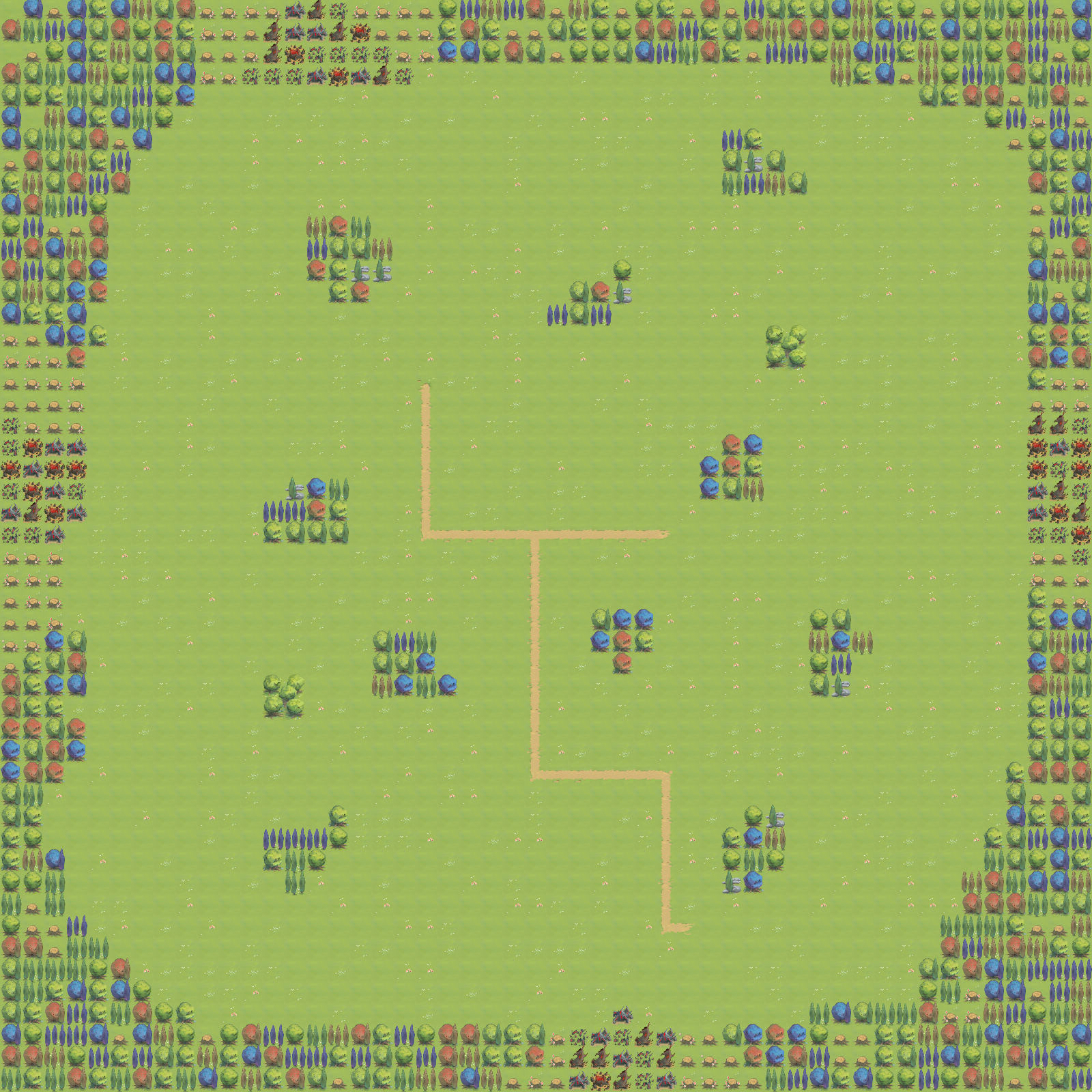
<sok sok ábra>



A játékbeli karakterek. Balról jobbra: Amy, Altarez, Démon Mágus.

### Játéktér

A játék helyszíne egy tisztás egy erdő közepén. Ennek megfelelően fák veszik körbe a játékmezőt, illetve néhány facsoport a tisztáson is található. A fák és tönkök akadályt képeznek, melyekre a karakterek nem léphetnek rá.



A játékteret alkotó pálya

## A program alapvető működése

A játék célja, hogy a játékos elpusztítsa az összes ellenfelet. A program során a játékos a képernyő közepén látható karaktert irányítja. Őt egy segédkarakter támogatja, akit közvetett módon irányíthat. Ellenfele két ellenséges karakter, melyeknek célja a játékos elpusztítása.

A Q és E gombokkal változtatható meg a kiválasztott akció. A játékos számára elérhető akciók listája: Mozgás; Alaptámadás, Képesség1, Képesség2 (együtt: Támadás); Segédkarakter irányítása (Parancs). A Q gombbal a listában balra, az E gombbal jobbra léphetünk.

A kiválasztott akció végrehajtása a WASD iránygombokkal történik. Ezekkel a kiválasztott akció végrehajtásának irányát adhatja meg a játékos. Felhasználásuk: W–fel, A–balra, S–le, D–jobbra.

Amennyiben a Mozgás akció van kiválasztva, a játékos a WASD iránygombok használatával a karakterét mozgatni képes. Az adott gomb megnyomásakor a karakter a megfelelő irányba mozog pontosan egy mezőnyit.

Amennyiben az Alap támadás, Képesség1 vagy Képesség2 (Támadás) akció van kiválasztva, a játékos a WASD iránygombok használatával támadni képes. A játékos karaktere az iránygomboknak megfelelő irányba a kiválasztott akciónak megfelelő támadást végez.

Amennyiben a Segédkarakter irányítása (Parancs) akció van kiválasztva, a játékos az A gombbal a saját karakterének közelébe hívhatja a Segédkaraktert, míg a D gombbal az ellenfelek közelébe küldheti őt, hogy megtámadja őket. A W gomb megnyomásának következtében a Segédkarakter véletlenszerű akciót hajt végre, az S gomb megnyomásának következtében pedig egyhelyben marad.

A játék körökből áll. Minden kör elején a Játékos karaktere cselekszik először. Két mozgás akciót, egy támadás jellegű akciót, és egy segédkaraktert irányító akciót hajt végre. Ezután a Segédkarakter következik, aki a Játékos utasításainak megfelelően cselekszik. Ezt követően az ellenséges karakterek következnek. Az ellenséges karakterek végrehajtandó akcióját a MCTS algoritmus határozza meg, erről lentebb írunk bővebben.

## 2.3 Felhasznált módszerek áttekintése

Most, hogy bemutattuk a vizsgált feladatot és a program alapvető működését, áttérünk a feladat megoldására. Ehhez mindenekelőtt a programban felhasznált algoritmusok, módszerek bemutatására kerítünk sort, felhasználói szempontok szerint, informálisan. Ezalatt a fő (MCTS) algoritmust, valamint a játék működtetéséhez felhasznált útkereső algoritmust értjük. Az algoritmusok részletes, formális leírására a [Fejlesztői Dokumentációban](#_Fejlesztői_Dokumentáció) kerül sor.

### 2.3.1 Wavefront algoritmus

A Wavefront algoritmus használatával képesek a karakterek mozogni a játéktéren, és kikerülni az akadályokat, illetve egymást. Lényege, hogy a karakter ***Kezdőpozíció*** mezője és egy kiválasztott ***Célpozíció*** mező közötti legrövidebb utat adja meg. Működési elve a következő:

A *Kezdőpozíció* mezőből kiindulva az összes elérhető mezőhöz egy egész szám értéket rendel, ez a szám pedig egyre nő, minél többedik szomszédot vizsgáljuk. Ez az érték reprezentálja, hogy az adott mező hány lépésből érhető el a *Kezdőpozíció* mezőjéről.

Miután a mezőkhöz értékeket rendeltünk, megkeressük az értékekkel ellátott játéktérben a *Célpozíció* mezőt. Innen lépésről-lépésre mindig a legkisebb szomszédot választjuk, amíg a *Kezdőpozíció* mezőt meg nem találjuk, és minden érintett mezőt megjegyzünk közben.

Amikor a *Kezdőpozíció* mezőt elértük, az érintett mezők listája egy mezőről-mezőre lépő utat jelöl ki, melynek az elején a *Célpozíció*, a végén pedig a *Kezdőpozíció* található. Ezt a listát használjuk a karakter mozgatására.

### 2.3.2 Monte Carlo Tree Search algoritmus

Az MCTS algoritmus határozza meg, hogy egy adott körben az ellenfelek milyen lépést tegyenek meg. Ez az algoritmus négy lépésből áll, melyek folyamata ciklikusan ismétlődik, egészen addig, amíg a megadott időkeret le nem jár. Ez a négy lépés a *Kiválasztás* (Selection), *Kibővítés* (Expansion), *Szimuláció* (Simulation/Rollout) és *Visszaterjesztés* (Backpropogation).

Az algoritmus alapja a Szimuláció. Az algoritmus véletlenszerű akciókat hajtat végre az eredeti játéktér egy másolatával. Ezen akciók hatására a karakterek mozognak és támadják egymást. Ha a Játékos- és Segédkarakternek kevesebb élete marad egy adott Szimuláció lépés végén, mint a két Démon Mágus karakternek, akkor azt a szimulációt, illetve annak kezdő lépését kedvezőnek tekinti az algoritmus. Miután elég sok Szimuláció fut le, a többi három lépés segítségével a két ellenséges karakter képes egy olyan akciót kiválasztani, amely viszonylag nagy valószínűséggel segíti a Játékos- és Segédkarakter elpusztítását.

## A program használata

A játékos a játék során az ellenfelek elpusztítására törekszik, az ellenfelek pedig a játékos- és segédkarakter elpusztítására. A játék körökre van osztva, és minden kört a játékos kezd. Az akciók között szabadon válthat, végrehajtani azonban korlátozott számban tudja őket. Kettő Mozgás, egy Támadás, és egy Parancs akciót adhat ki, mielőtt számára véget ér a kör.

Ezután a Segédkarakter lépése történik. Az alapján, hogy a játékos milyen parancsot adott ki a Segédkarakter számára, a Segédkarakter vagy megközelíti a legközelebbi ellenfelet, vagy megközelíti a játékos karakterét, vagy egyhelyben marad, vagy véletlenszerű akciót hajt végre.

Utána a két ellenséges karakter következik, melyek a MCTS algoritmus alapján választanak lépést. Ennek során figyelembe veszik, hogy a játékos milyen képességeket használt már. Amikor az algoritmus befejeződik, a két ellenséges karakter egyszerre hajtja végre akcióit.

Miután a két ellenséges karakter is cselekedett, új kör kezdődik, mely elején ismét a játékos cselekedhet. Egészen addig ismétlődnek a körök, ameddig vagy a két játékos karakter, vagy a két ellenséges karakter meg nem hal. Ebben az esetben "Győzelem!", vagy "Vereség!" képernyő jelenik meg, és kiléphetünk a játékból.

# Fejlesztői Dokumentáció

## 3.1 Specifikáció, követelmények

A program feladata a játékos parancsainak követése, ennek megfelelően a játékos- és a segédkarakter mozgatása és működtetése, valamint az ellenséges karakterek működtetése. Az ellenfelek akcióinak megadásához a Monte Carlo Tree Search algoritmust használja.

Az MCTS algoritmus szimulációval működik, erről bővebben lentebb ejtünk szót. A szimulációnak figyelembe kell vennie, hogy a Játékos karakter milyen képességeket használt már. Az ellenséges karakterek által választott akcióknak is többé-kevésbé tükrözniük kell ezt.

Minden karakternek képes kell lennie mozognia, de csak a szabad mezőkön, és támadnia képességekkel és alaptámadással. A játékos a saját karakterét közvetlenül irányítja, míg a segédkaraktert közvetetten.

## Használati esetek

* Játékosként mozogni szeretnénk a játéktéren.
* Játékosként át szeretnénk váltani az éppen aktív akciót.
* Játékosként meg szeretnénk támadni egy ellenfelet.
* Játékosként szeretnénk használni egy képességet.
* Játékosként szeretnénk irányítani a segédkaraktert.

## Algoritmusok

### Wavefront algoritmus

A karakterek a játéktéren mozogni képesek. Nem minden mezőre léphetnek viszont. Ha egy mezőn terepakadály, pl. egy fa van, vagy egy másik karakter áll az adott mezőn, a karakter nem léphet oda. Ahhoz, hogy ezeket az akadályokat figyelembe véve mozogjanak a karakterek, a Wavefront algoritmust használjuk. Ez az algoritmus egy ***kiinduló pozícióból*** egy ***cél pozícióba*** keresi meg a legrövidebb elérhető utat.

Ennek megvalósításához rekurzívan bejárja a ***kiinduló pozícióból*** elérhető játékteret, és a rekurzió minden egyes lépésével egyel nagyobb értéket ad az adott mezőknek. Ez a rekurzió a ***kiinduló pozícióból*** indul, az annak megfelelő mezőnek pedig az *1* értéket adja. Ezután a közvetlen azzal szomszédos mezőnek egyel nagyobb, *2* értéket ad, és így tovább, amíg több szomszédos mezőt már nem lehet felfedezni a játéktéren.

Miután a bejárható mezőt feltöltötte, a ***cél pozíciónak*** megfelelő mezőből kiindulva visszaköveti és elmenti az utat a ***kiinduló pozícióig***. Ezt úgy éri el, hogy a szomszédos mezők közül mindig a legkisebb értékű mezőt választja az út következő elemének. Ez a módszer végül megtalálja az összes közül a legkisebb értékű mezőt, ami pont a ***kezdő pozíciónak*** felel meg, mivel ennek értékét az algoritmus elején egyre állítottuk, és az összes többi mező értékét ehhez képest növeltük.

##### Miért a Wavefront?

Azért a Wavefront algoritmust választottam az útkeresés problémájának megoldására, mert a játék későbbi továbbfejlesztésében előfordulhat, hogy a karakterek mozgása nem mezőről mezőre történik, hanem az egy körben elérhető mezők közül egyet választva a karakter oda mozog. Ebben az esetben hasznos, hogyha megjeleníthető, hogy egy adott karakter egy adott körben mely mezőkre léphet, ebben pedig a Wavefront algoritmus által létrehozott, számokkal ellátott játéktér nagy segítség lehet.

### Monte Carlo Tree Search algoritmus

Az MCTS algoritmus határozza meg, hogy egy adott körben az ellenfelek milyen lépést tegyenek meg. Ez az algoritmus négy lépésből áll, melyek folyamata ciklikusan ismétlődik, egészen addig, amíg a megadott időkeret le nem jár.

Ez a négy lépés a *Kiválasztás* (Selection), *Kibővítés* (Expansion), *Szimuláció* (Simulation/Rollout) és *Visszaterjesztés* (Backpropogation). Az algoritmus egy csúcsokból álló fa gráfot használ, mely kezdetben egy gyökér csúcsból áll, és működése közben az algoritmus egyre több csúcsot ad hozzá a fához.

##### A csúcsok tulajdonságai.

Minden egyes csúcs egy körön belül vagy a játékos- és a segédkarakter, vagy az ellenfél karakterek lépéseit határozza meg. Ennek megfelelően minden csúcs tartalmaz egy *akciót*, ami megfeleltethető az egyik fél karaktereinek lépéseinek; és egy logikai változót, amely megadja, hogy az adott csúcs az ellenfél egységeinek akcióját adja-e meg, vagy a játékos- és segédkarakterét. Ez a két érték minden csúcs létrejöttekor inicializálódik, és később nem változik.

Minden csúcs tartalmaz továbbá két nem negatív egész számot, a *Győzelmek* (Wins) és *Látogatások* (Visits) értékeket. Ez a két érték minden újonnan létrejött csúcsban alapvetően nullával egyenlő, és a negyedik lépésben, a *Visszaterjesztésben* változik.

A csúcsok tartalmaznak még mutatókat, mind a szülőjükre, mind a gyermekeikre. A szülő csúcs nem változik az algoritmus futása közben, a csúcs létrejöttekor inicializálódik. A gyerek csúcsok egy listában tárolódnak, amely kezdetben üres, az algoritmus futása során pedig bővül.

Mindezeken kívül a csúcsok még egy logikai változóval rendelkeznek, ez azt adja meg, hogy a szimulációban létrejövő csúcsok-e, vagy sem. Ennek jelentősége az algoritmus harmadik lépésében, a Szimulációban lesz. Ennek a változónak az értékét is a csúccsal együtt inicializáljuk.

Minden esetben, ha egy csúcs szülője egy másiknak, az egyik csúcs az ellenfelek, a másik pedig a játékos- és segédkarakter akcióit adja meg. Két szülő-gyermek kapcsolatban álló csúcs tehát sosem adhatja meg ugyanannak a csapatnak az akcióit.

Kiválasztás (Selection).

*Ebben a lépésben a fának egy csúcsa választódik ki.* Ez a kiválasztás lépésenként egy képlet segítségével történik, melynek jelentősége, hogy a legígéretesebb csúcsot válassza ki a lehetségesek közül, de a túl régen kiválasztott csúcsoknak is adjon esélyt, ezzel csökkentve az esélyt, hogy a legjobb megoldást csupán azért elkerülje az algoritmus, mert eleinte rossz értékeket mutatott, és ezért nem vizsgálta tovább. A *Kiválasztás* első lépése mindig a gyökér csúcsból indul. A *Kiválasztás* minden lépésében kiválaszt egy csúcsot az adott csúcs és a gyermekei közül.

Ez a választás a következő képleten alapszik: , ahol: *w* az adott csúcs *Győzelmek* értéke; az *n* az adott csúcs *Látogatások* értéke; az *N* az adott csúcs szülőjének *Látogatások* értéke; a *C* pedig egy adott konstans, ami a mi esetünkben .

Ezt a képletet minden megvizsgált csúcsra kiszámoljuk: az adott csúcsra és a gyermekeire is. Amelyik csúcsnak a legnagyobb a képlet által kiszámolt értéke, azt a csúcsot fogja a *Kiválasztás* lépés választani.

Miután kiválasztott egy gyermek csúcsot, a választott csúcsra ismét végrehajtja a kiválasztást, amennyiben a csúcs nem levél csúcs. Ha azonban nem gyermek csúcsot választ ki hanem önmagát, vagy levél csúcsot, akkor a kiválasztott csúccsal tér vissza, és a *Kiválasztás* lépés befejeződik.

##### **Kibővítés (Expansion).**

*Ebben a lépésben a Kiválasztás lépés során kiválasztott csúcs gyermekeként generálunk egy új csúcsot.* Ennek a csúcsnak az *akció* változója véletlenszerűen generálódik. Ez tehát egy véletlenszerű lépésnek felel meg. Ebből a csúcsból fogjuk indítani a *Szimuláció* lépést.

##### Szimuláció (Simulation/Rollout).

*Ez a lépés a Kiválasztás lépésben véletlenszerűen létrehozott akciójú csúcsból indul ki.* Minden szimulációs lépésben létrehoz egy csúcsot, véletlenszerű akcióval, és az előzőleg szimulált csúcs gyermekévé teszi. (Nyilván a legelső csúcs, amit létrehoz, a *Kibővítés* során létrehozott csúcs gyermeke lesz.) Ezzel gyakorlatilag egy véletlenszerű kimenetelt szimulál, amelyben a két fél tetszőleges lépéseket tesz.

A *Szimuláció* a MCTS algoritmusokban tipikusan addig fut, amíg az egyik fél egyértelműen meg nem nyeri a játszmát, így egyértelműen eldöntve a győztes felet. Mivel a mi környezetünkben a terminális végkimenetel nagyon sok véletlenszerű lépést jelenthetne, ezzel jelentősen megnövelve az algoritmus erőforrásigényét, ezért a *Szimuláció* ebben a programban csak bizonyos számú lépést tehet meg. Emiatt a *Szimuláció* végén nem feltétlen lesz terminális végkimenetel bármelyik fél számára; ebben az esetben azt a felet tekintjük győztesnek, amelyiknek a karakterei együttesen több életponttal rendelkeznek.

##### Visszaterjesztés (Backpropogation).

A *Visszaterjesztés* a *Szimuláció* eredményére alapoz. A legutolsó, *Szimuláció* során létrehozott csúcsban elindítunk egy rekurzív folyamatot, amely mindig a csúcs szülőjében folytatódik, egészen a gyökér csúcsig. Ez a folyamat minden érintett csúcsnak egyel növeli a *Látogatások* értékét, és amennyiben a *Szimuláció* végén az ellenfelet tekintettük győztes félnek, szintén egyel növeli az érintett csúcsok *Győzelmek* értékét.

##### Miért a Monte Carlo Tree Search Algoritmus?

Azért a MCTS algoritmust választottam az ellenfelek Mesterséges Intelligencia algoritmusának, hogy bemutathassam, hogy viszonylag kevés változtatással is adaptívvá (vagyis a Játékos karakter rendelkezésére álló lehetőségek függvényében cselekvő) tehető egy játék.

Jelen esetben az adaptívvá tételhez arra volt szükség, hogy az MCTS Szimulációja során a Játékos karakternek megengedjük, hogy használja azokat a képességeket, amiket már használt, de azokat ne, amiket még nem. Így a megfelelő algoritmusból kiindulva, viszonylag alacsony befektetéssel, sokkal reszponzívabb játékélmény teremthető.

## Programozási környezet, eszközök

### Unity (2019.3.6f1)

A Unity tökéletes alapot biztosít egy videójáték programozásának. Könnyen átlátható panelek, egyszerű Drag&Drop lehetőségek és sok egyéb támogató funkció található benne. Ezek közül kiemelendő néhány, melyek jelentősen befolyásolták, illetve megkönnyítették a program elkészítését.

A **Scene** panel segítségével tekinthetjük meg, illetve módosíthatjuk a létrehozandó játékteret. Segítségével kiválaszthatunk, mozgathatunk, áthelyezhetünk, és egyéb formában módosíthatunk minden játékbeli objektumot, ugyanúgy például a karaktereket, mint a kamerát.

A **Game** panel segítségével szimulálhatjuk a játék futását. A játéktéren elhelyezett kamerákat használja, hogy megjelenítse a programot futás közben. A Unity felületén felül található egy eszköztár, mely egy „Lejátszás” és egy „Megállítás” gombot is tartalmaz. A Lejátszás gombot megnyomva aktiválódik a Game nézet, és elkezd futni a Unity-n belül a program, minden olyan tulajdonsággal együtt, amely a futtatott programban is meg fog jelenni. Ekkor a Scene panelről automatikusan a Game panelre vált a szerkesztő.

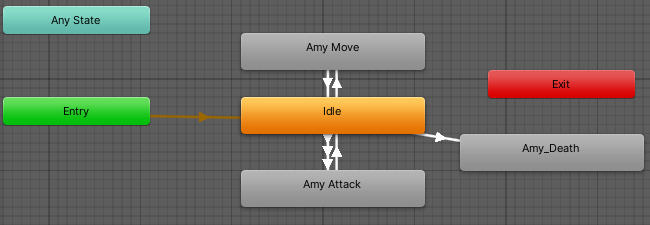
Az összes Unity által generált objektum a **GameObject** osztály leszármazottja, ez pedig biztosít néhány hasznos, minden objektumban elérhető tulajdonságot, funkciót. A Unity által generált C# fájlban alapvetően két függvény található, a Start és az FixedUpdate.

A **Start** függvény az objektum létrejöttekor automatikusan hajtódik végre, így például változók értékének inicializálásához ez hasznos.

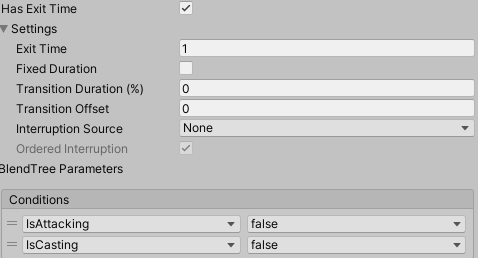
Az **FixedUpdate** függvény minden képkockában meghívódik egyszer. Ennek a függvénynek számtalan hasznosítási lehetősége van, és ez adja gyakorlatilag a program alapját. Felhasználásával minden képkockában például ellenőrizni tudunk gombnyomást, vagy épp apró távolságokkal mozgatni a karaktereket.

A **Hierarchy** panel tartalmazza az összes játéktéren szereplő objektumot. Az objektumok egymás között szülő-gyermek kapcsolatot is kialakíthatnak, ennek hatása többek közt, hogy a gyermek objektum a szülővel együtt mozog. Ilyen módszerrel van például az életerőt jelző sáv minden egyes karakterhez rendelve.

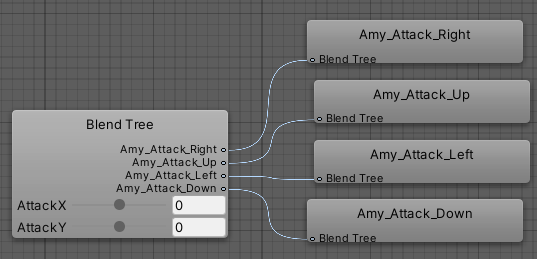
Az **Animation**, panel segítségével lehet képkockák felhasználásával animációkat megvalósítani, az **Animator** panel segítségével pedig az animációkat struktúráltan elrendezni, biztosítva, hogy a megfelelő animáció a megfelelő helyzetben játszódjon le. Az animációs állapotok között átmeneteket adhatunk meg, animációs állapotokon belül pedig feltételtől függően másfajta animációkat játszhatunk le, ha úgynevezett „Blend Tree”-t használunk.



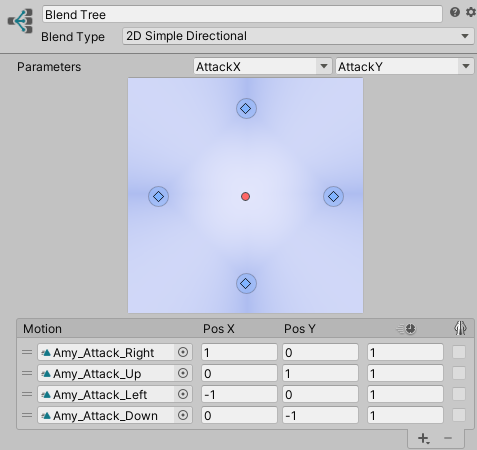
Amy Animator panelje, mely ábrázolja az átmeneteket az állapotok között.



Az *Amy Attack* állapotból az *Idle* állapotba irányuló átmenet tulajdonságai.

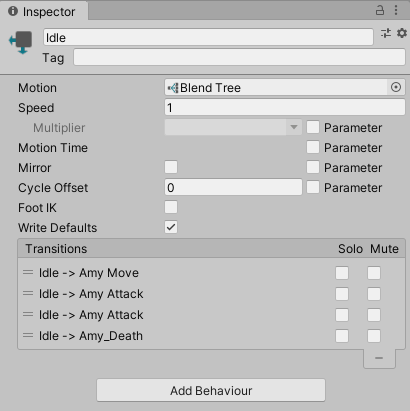
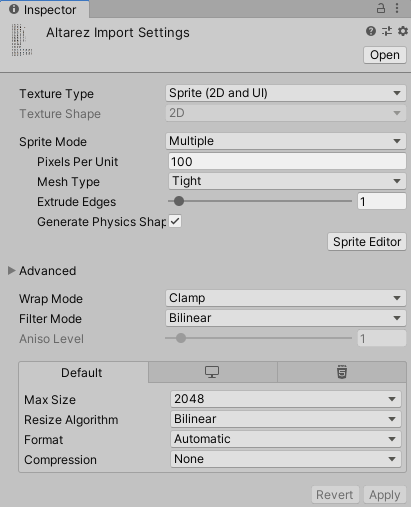
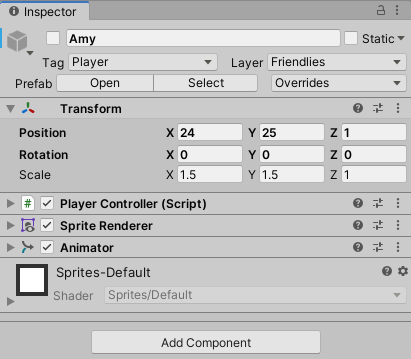


Az *Amy Attack* állapot, mely egyben egy Blend Tree.



A feltételek, melyek alapján az animációk változnak az *Amy Attack* állapoton belül.

Az **Inspector** panel az összes többi panelen belül kiválasztott objektum adataihoz biztosít Read/Write hozzáférést. Egy Animator állapot adataihoz ugyanúgy hozzáférünk itt, mint egy felhasználandó képnek, vagy egy játéktéren szereplő karakternek. Itt tudunk publikus adattagoknak kezdőértéket beállítani egy osztályhoz is.

Az Inspector panel tartalma néhány más panelben kiválasztott objektum esetén.

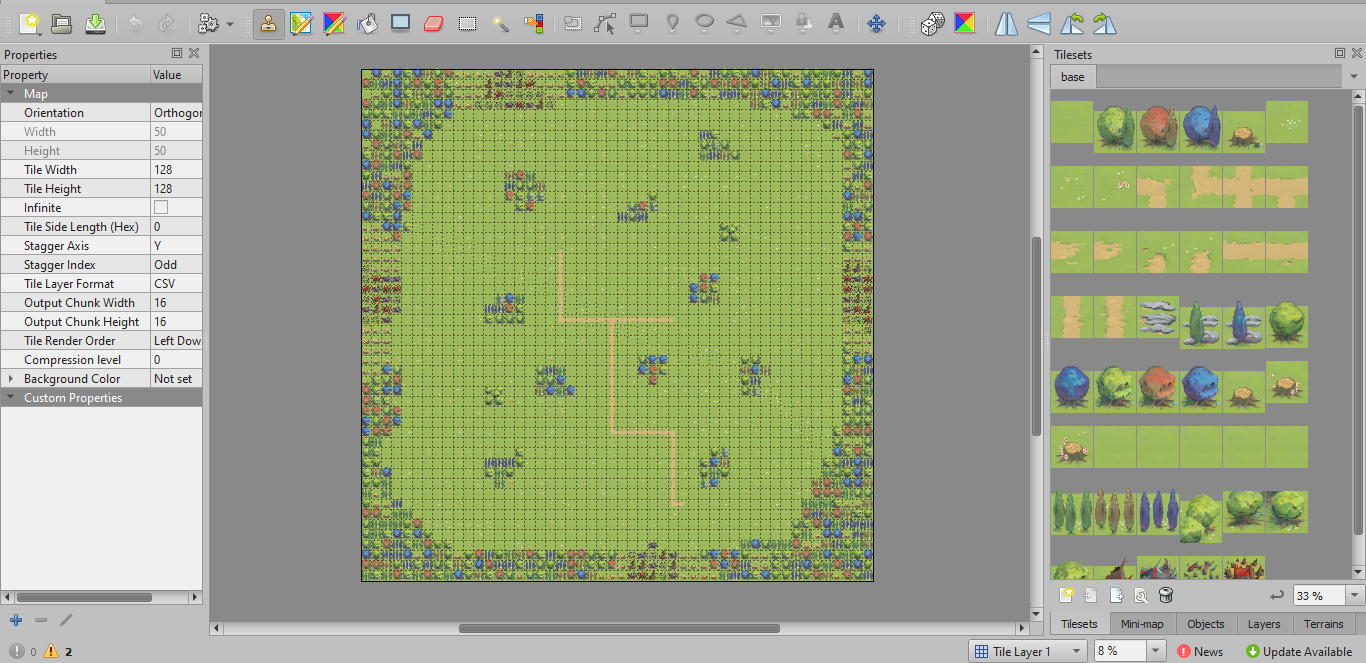
A **Project** panel a játék fájlrendszerét tartalmazza. A hagyományos fájlböngészőn kívül fontos tulajdonsága, hogy a kiválasztott fájlok tulajdonságai az Inspector panel által hozzáférhetőek.

A **Console** panel hibákat, figyelmeztetéseket, és a programozó által feladott üzeneteket jelenít meg. A programírás folyamata közben természetesen nagy segítség, ha ki tudjuk írni a kívánt adatokat az adott helyzetekben.

Fontos tulajdonsága még a Unity-nek az úgynevezett **Prefab** objektumok használata. Egy kész GameObject objektumot a Hierarchy panelből a Project panelbe „húzva” (Drag&Drop) könnyedén létrehozhatunk egy Prefab objektumot. Egy Prefab tulajdonságait ugyanúgy módosíthatjuk, mint egy egyedi, a játéktéren található objektumnak. Prefab objektumokat azonban könnyedén példányosíthatunk C# használatával. Előnye ennek a módszernek például, hogy szükségtelen minden egyes példányosított objektum tulajdonságait egyesével beállítanunk, elég, ha egy Prefab tulajdonságait alakítjuk át, és ezt a Prefab-ot példányosítjuk.

### Tiled

A játékban felhasznált alap térkép elkészítéséhez a **Tiled** nevű programot használtam. Ez a program egy négyzetrácson elhelyezett ábrahalmazt (úgynevezett Spritesheet-et) megfelelő beállítások alapján mezőkre „szabdal”. Ezek a mezők fogják a térkép alapját alkotni.

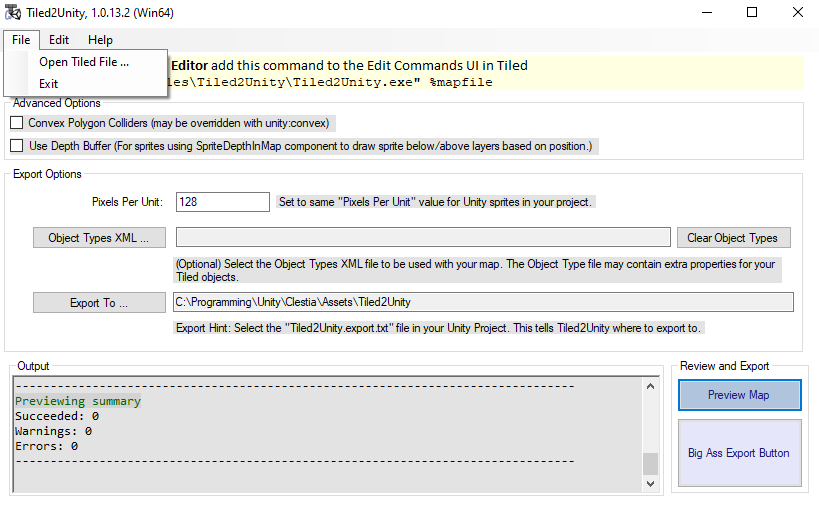


A játékmező a Tiled programban. Középen látható a négyzetrács, melyre a mezőket helyezhetjük, jobb oldalt pedig a mezők palettája.

Miután a mezőket létrehoztuk, egy négyzetrácsra elhelyezhetjük a mezőket, ezzel kialakítva a játékteret. Sok egyéb képességgel is rendelkezik a program, de a mostani projekthez nem volt szükség bonyolultabb funkciókra, így ezekről nem ejtünk szót.

##### Tiled2Unity

A Tiled programban létrehozott térkép fájlt be is kellett importálni a Unity-ba, ehhez a **Tiled2Unity** programot használtuk. Ez a program kifejezetten erre a célra van tervezve. Mindenek előtt természetesen telepítenünk kell a programot, majd megnyitnunk.



A megnyitott Tiled2Unity program.

A program telepítése és megnyitása után meg kell nyitnunk a Unity-ban a projektünket. Az *Export To..* gombra kattintva beállítunk egy mappát a projektünk struktúráján belül, ahová szeretnénk, ha a program exportálja a feldolgozott térképet. Ezután a menüsorban a *File*, majd az *Open Tiled File…* gombra kattintunk. Ekkor megjelenik egy böngésző ablak, melyben ki kell választanunk a Tiled programban létrehozott térkép fájlt. Az *Output* panelben nyomon követhetjük, hogy mikor készül el a program a térkép fájl feldolgozásával. Amikor elkészült, a *Big Ass Export Button* gomb megnyomásával exportálhatjuk a térképet Unity-ba.

Ekkor a projektünkben a kiválasztott mappában létrejön egy Tiled2Unity mappa, melyen belül a program által importált térkép működtetéséhez szükséges erőforrások találhatóak. Ezután a Drag&Drop funkcionalitást használva a játéktérre húzzuk a kész térképet a Unity-n belül található Tiled2Unity mappából.

## A program felépítése

### Unity, játékobjektumok, játéktér

Alapvetően a kamera objektumon kívül a játékmező és a C# scriptek központjául szolgáló GridModel objektum található, valamint a játékmező és a GridModel másolata a MCTS szimulációja számára.

A **Kamera** a CameraController C# fájl által van vezérelve. Két publikus adattagja, a moveSpeed avagy mozgási sebesség, illetve a followTarget avagy a követendő karakter. Ezeket csak más C# scriptben módosítjuk, kezdőértékük Unity-ban nincs meg adva, ezért itt nem számolunk be róla részletesen.

A **Játékmező** a Tiled és Tiled2Unity programok által létrehozott és beimportált térképet használja. Ennek paramétereit a Tiled2Unity állítja be, ezért mozgatáson kívül nem módosítjuk.

#### GridModel

A GridModel képzi a C# scriptek csomópontját, ebben fut a fő program, és ezzel kommunikál a többi osztály. Itt már több alapvető beállítást is végrehajtottunk a Unity-n belül. A beállított változók használatára később térünk ki, itt a beállított változók értékeit írjuk le.

A **WhatStopsMovement** változó típusa a Unity-ben játékobjektumoknak beállítható Layer (avagy Réteg). A Tiled programban elkészített térkép által tartalmazott ütközési mezők <írjunk róla Tiled téma alatt> Layer értékét beállítjuk egységesen a CollideLayer rétegre, majd a GridModel WhatStopsMovement változójának ezt a CollideLayer réteget adjuk értékül. Így a GridModel osztályon belül később ellenőrizhetjük, hogy adott pozíciókban van-e a Tiled program által létrehozott ütközési mező, hogy megtilthassuk a karaktereknek az ilyen mezőkre való lépést.

A **DemonMage**, **Amy** és **Altarez** változó típusa a Character saját osztály. Ennek a három változónak a nevüknek megfelelő Prefab objektumot adjuk értékül. Így a GridModel osztályban képesek leszünk példányosítani őket.

Az **Arrow**- és **Fireball** Left, Right, Up és Down változók típusa a Projectile saját osztály. Ezeknek a változóknak a nevüknek megfelelő Prefab objektumokat adjuk értékül. Ezek példányosítására a mozgó lövedéket igénylő támadások esetében lesz szükségünk a GridModel osztályban.

Az **earthspikes, icespikes, fireblast, icetacle, torrentacle, tornado, lightning, shield, snake** változók GameObject típusúak. Ezeknek szintén a nevüknek megfelelő Prefabot adjuk értékül. Ezeket a többi, mozdulatlan képesség esetén példányosítjuk a GridModel osztályban, a megfelelő esetekben.

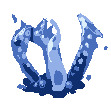
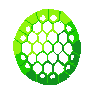
A **Simulation** logikai változót a második, szimulációra használt GridModel esetében igazra állítjuk, mivel ezt sok esetben fontos ily módon elkülönítenünk a GridModel osztályban.

#### Prefab objektumok

A Karakterek példányosítását lehetővé tevő Prefab-ek az **Amy**, **Altarez** és **DemonMage** Prefab. Mindhárom Prefab-hez a neki megfelelő Controller osztály van rendelve (sorrendben: PlayerController, AltarezController, DemonMageController). Ezek segítségével irányíthatóak a GridModel osztály által a példányosított Karakter Prefabek.

A lövedékek példányosítását az **Arrow**- és **Fireball**- **Down/Right/Up/Left** Prefab-ok teszik lehetővé. Ezekhez a Projectile osztály van rendelve a Unity-ban, és ez a lövedékek mozgásáért felel.

A többi képesség példányosítását a következő Prefab-ok teszik lehetővé: **earthspikes, icespikes, fireblast, icetacle, torrentacle, tornado, lightning, shield, snake.** Ezek egyszerű GameObject típusúak, ezért hozzájuk a Unity-ban nem rendelünk semmit.

A felsorolt képességek Prefab-jeinek egy-egy képkockája, a fenti felsorolás sorrendjében.

### C#, osztályok, metódusok

A projekthez létrehozott osztályokat írjuk le. A karaktereket három osztály irányítja: **PlayerController, AltarezController, DemonMageController**. Az első a Játékoskaraktert, a második a Segédkaraktert, a harmadik a Démon Mágus karaktereket irányítja. Ezek a **Character** osztályból származnak. A kamerát a **CameraController** irányítja. A lövedékeket (nyílvessző, tűzgolyó) a **Projectile** osztály irányítja. A játék, valamint az összes többi osztály kapcsolatának alapját a **GridModel** osztály alkotja. A továbbiakban ezeket az osztályokat írjuk le részletesen, az adattagjaikat, metódusaikat és kapcsolataikat.

### Character

Ebből az osztályból származik az összes többi karaktert irányító Controller osztály.

* Adattagok:

| Deklaráció | Leírás |
| --- | --- |
| public int health | Ez az érték adja meg a karakter aktuális értékét. |
| protected float moveSpeed | Ez az érték adja meg a karakter mozgási sebességét a játékmezőn. |
| public bool canMove | Ez az érték adja meg, hogy a karakter képes-e éppen mozogni. |
| protected Vector2 lastMove | Ez az érték adja meg, hogy a karakter melyik irányba néz a legutóbbi lépése alapján. |
| private Vector3 targetTile | Ez az érték adja meg a karakter mozgásának célját egy közvetlen egyenes vonalon belül. |
| public Vector3 moveTo | Ez az érték adja meg a karakter mozgásának célját a térképen akárhol. |
| public Vector3[] path | Ez a tömb tartalmazza a karakter jelenlegi pozíciójától a moveTo Vector3 változó pozíciójáig vezető utat a Wavefront algoritmus alapján. |
| public int readyAction | Ez az érték adja meg, milyen akciót hajtson végre a karakter, amint végzett a mozgással. Elsősorban a szimuláció során használjuk. |
| public int selectedAction | Ez az érték adja meg, hogy a játékos karaktere éppen melyik végrehajtható akcióját választotta ki. |
| public int[] actions | Ez a tömb adja meg a karakterek által egy körben megtehető két lépés irányát, valamint a végrehajtandó akcióját. |
| public bool moving | Ez az érték adja meg, hogy a karakter éppen mozog-e. |
| public bool attacking | Ez az érték adja meg, hogy a karakter éppen támad-e. |
| public bool casting | Ez az érték adja meg, hogy a karakter éppen varázsol-e. |
| public bool proceed | Ez az érték adja meg a szimuláció során, hogy a karakter végzett-e a lépéseivel. |
| public bool simChar | Ez az érték adja meg, hogy az adott karakter a szimuláció része-e, vagy sem. |
| public GridModel script | Ez az érték adja meg, hogy melyik GridModel objektum tartozik a karakterhez. |
| protected Animator anim | Ez az érték adja meg, hogy a karakterhez melyik Unity Animator objektum tartozik. |
| public int layer | Ez az érték adja meg, hogy a karakter melyik Unity réteghez van hozzárendelve. |
| public List<GameObject> spells | Ez a lista adja meg, hogy a karakternek milyen képesség objektumai vannak éppen létrehozva. |
| public List<Projectile> projectiles | Ez a lista adja meg, hogy a karakternek milyen lövedékei vannak éppen létrehozva. |

* Metódusok:

| Szignatúra | Leírás |
| --- | --- |
| protected virtual void Start () | Beállítja az adattagjainak kezdőértékeit. |
| protected virtual void FixedUpdate () | Részletes leírás a többi metódus leírása alatt. |
| public void doActions (int[] action) | Végrehajtatja a paraméterként megadott számoknak megfelelő akciókat. |
| public bool canProceed () | A proceed adattag értékével tér vissza, ha a karakter animációi lejátszódtak (vagyis „Idle” állapotban van), és éppen nem létezik a játéktéren általa létrehozott lövedék vagy képesség. |
| public void Move (float x, float y) | Az x és y paraméterek által előállított nullvektor irányába mozgatja a karaktert vagy az x, vagy az y tengely egyik irányába. |
| public void addHP (int x) | A karakter health adattagjának értékéhez adja a paraméterül kapott értéket. |
| public void setAttrTo  (Character ch) | A szimulációs ciklus elején használt függvény a szimulációs karakterek életének és pozíciójának beállításához, az eredeti játéktéren tartózkodó karakterek értékei alapján. |
| public void addMoveTo  (float x, float y, float z) | A paraméterként kapott értékek alapján képzett Vector3 objektumot hozzáadja a karakter objektum moveTo adattagjának értékéhez. |
| public virtual bool attackDir  (float h, float v) | Az x és y paraméterek által előállított nullvektor irányába támad. Ellenőrzi, hogy az adott irányban három mezőn belül van-e ellenséges karakter, és ha igen, megtámadja az Alaptámadását használva. |
| public virtual bool spell1Dir  (float h, float v) | Az x és y paraméterek által előállított nullvektor irányába támad. Ellenőrzi, hogy az adott irányban három mezőn belül van-e ellenséges karakter, és ha igen, megtámadja a Spell1 képességét használva. |
| public virtual bool spell2Dir  (float h, float v) | Az x és y paraméterek által előállított nullvektor irányába támad. Ellenőrzi, hogy az adott irányban három mezőn belül van-e ellenséges karakter, és ha igen, megtámadja a Spell2 képességét használva. |
| public virtual void tryAttacking () | Az AttackDir függvényt használva megpróbál mind a négy irányba ellenfelet támadni. Amennyiben talál egy megtámadható ellenfelet, a függvény véget ér. |

* FixedUpdate

Az FixedUpdate függvény minden képkockában meghívódik. Eltávolítja a karakter objektum spells és projectiles listájából azokat az elemeket, melyek már megsemmisültek. Lekerekíti a moveTo adattag x és y értékét egész számra. Hamisra állítja az attacking és casting adattagok értékét, valamint igazra a canMove adattag értékét. A továbbiakat csak akkor hajtja végre, ha létezik az osztályhoz tartozó Unity objektum.

Ha a karakter közel áll a targetTile adattagban megadott mezőhöz, a karakter pozícióját megfelelteti a targetTile értékével, így a karakter pozíciója biztosan egyenlővé tud válni a targetTile értékkel.

Ha a karakter targetTile értéke megváltozik, és ezért nem egyenlő a pozíciójával, a karakter a Move funkció használatával a targetTile adattagban megadott mező felé mozog.

Ha a karakter moveTo értéke megváltozik, és ezért nem egyenlő a pozíciójával, a karakter utat keres a moveTo adattag által megadott mezőhöz. Ehhez a GridModel típusú script adattagjának pathFinding metódusát használja. A metódus visszatérési értékét a path adattagjának értékül adja. Amennyiben az FixedUpdate későbbi iterációja történik, melyben a path adattag már megkapta az értéket a pathFinding metódus alapján, akkor a path adattagban található Vector3 értékeket a karakter targetTile adattagjának adja értékül egyesével, haladva a path adattagnak a moveTo értékével megfelelő eleméig.

Ha a karakter nem mozog, és a readyAction adattagjának értéke nem nulla, a TryAttacking függvény meghívásával támadni próbál.

Végül, az Animator típusú anim adattagjának értékül adja a megfelelő adattagjainak értékét, ezzel beállítva a lejátszódó animációját.

### PlayerController

Ez az osztály irányítja a Játékoskaraktert.

* Adattagok

A Character osztály által örökölt adattagok.

* Metódusok

A Character osztály által örökölt metódusok. Továbbá:

| Szignatúra | Leírás |
| --- | --- |
| protected override void Start() | A Character Start() metódusa. |
| protected override void FixedUpdate() | Részletes leírás a többi metódus leírása alatt. |
| public override bool spell1Dir(float h, float v) | Részletes leírás lejjebb. |
| public override bool spell2Dir(float h, float v) | Részletes leírás lejjebb. |
| public override bool attackDir(float h, float v) | Részletes leírás lejjebb. |
| public override void tryAttacking() | Megpróbálja az első sikeres találatig a Spell1Dir, Spell2Dir és AttackDir felhasználásával megtámadni az elérhető ellenfeleket. |

* *FixedUpdate*

A Character osztály által örökölt FixedUpdate függvényt meghívja, ezen felül több funkcionalitása is van, ezeket írjuk le most.

A saját *health* adattagjának értékét a hozzá tartozó Animator típusú *anim* változónak a *health* nevű változójának adja.

Ezután leellenőrzi, hogy a karakter szimulációs karakter vagy sem, és ha nem, de nem a karakter köre jön, vagy a szimuláció még fut, megszakítja a FixedUpdate függvényt. Hogy a karakter köre jön-e, azt a GridModel típusú *script* nevű adattagjának *whoseTurn* nevű változója adja meg, melyet emiatt a PlayerController lekér.

Ha az előző ellenőrzésen tovább jutott, lekéri a beérkező billentyűlenyomásokat, a W-A-S-D gombokét az akció irányának megadásához, és a Q-E gombokat az akció váltásához.

Ha megnyomódik a Q, illetve az E gomb, a *selectedAction* nevű adattagját egyel csökkenti, illetve növeli. Ezzel adja meg, hogy a karakter a W-A-S-D gombok lenyomásával megadott irányba milyen akciót hajtson végre (mozgás, támadás, stb.). A *selectedAction* változó értékét nullára állítja, ha öttel egyenlő, és négyre, ha mínusz egyel egyenlő. Ezzel biztosítja, hogy ez az adattag mindig a [0-4] tartományból vesz fel értéket.

Ezután a W-A-S-D billentyűk megnyomása által megadott irányban, a *selectedAction* adattag értékétől függően használja az egyik akcióját.

Végül beállítja az anim adattagjának értékeit, hogy az animáció lezajlódhasson.

* *AttackDir*

A h és v paraméterek alapján megadott irányban leellenőrzi, hogy a következő hat mezőn áll-e ellenfél. Ezt a *script* adattagja *checkEnemyInPosition* metódusával teszi meg, melyet mind a hat mezőre meghív. Amint megtalál egy karaktert, nem vizsgál több mezőt.

Amennyiben talál ellenfelet, létrehoz egy Projectile típusú objektumot a *script* adattagja *makeProjectile* metódusával. Paraméterként megadja, hogy egy nyilat („arrow”) kell létrehozzon, mely a karaktertől a célpont karakterig mozog, 20-as sebzéssel, 30-as sebességgel. Az animációját támadás állapotba állítja, a *h* és *v* paramétereknek megfelelő irányba.

* *Spell1Dir*

S h és v paraméterek által megadott irányban egy három mező széles, három mező hosszú sávban megkeres minden ellenfelet. Ezt a *script* adattag *checkEnemyInPosition* metódusával teszi meg, melyet mind a kilenc mezőre meghív.

Amennyiben talált legalább egy ellenfelet, létrehoz egy Projectile típusú objektumot a *script* adattag *makeProjectile* metódusával. Paraméterként megadja, hogy egy tornádót („tornado”) kell létrehozzon, mely a karaktertől a célpont karakterig/karakterekig mozog, 20-as sebzéssel, 30-as sebességgel. Az animációját támadás állapotba állítja, a *h* és *v* paramétereknek megfelelő irányba.

* *Spell2Dir*

A megadott paraméterektől függetlenül ellenőrzi, hogy a karakter körüli 5x5-ös mezőben tartózkodik-e ellenfél. Ezt a *script* adattagja *checkEnemyInPosition* metódusával teszi meg, melyet mind a huszonnégy mezőre meghív.

Amennyiben tartózkodik a területen ellenfél, a terület mind a 24 mezőjén létrehoz egy objektumot a *script* adattagja *makeSpell* metódusával. Paraméterként megadja a létrehozandó objektumok pozícióit, valamint, hogy „earthspike” típusú objektumot hozzon létre. Ezután megtámadja a megtalált ellenfelet/ellenfeleket a *script* adattagja *attackEnemy* metódusával. Az animációját varázslás állapotba állítja, mely során lefelé néz.

### AltarezController

Ez az osztály irányítja a Segédkaraktert.

* Adattagok

A Character osztály által örökölt adattagok.

* Metódusok

A Character osztály által örökölt metódusok. Továbbá:

| Szignatúra | Leírás |
| --- | --- |
| protected override void Start() | A Character Start() metódusa. |
| protected override void FixedUpdate() | Részletes leírás a többi metódus leírása alatt. |
| public override bool spell1Dir(float h, float v) | Részletes leírás a többi metódus leírása alatt. |
| public override bool spell2Dir(float h, float v) | Részletes leírás a többi metódus leírása alatt. |
| public override bool attackDir(float h, float v) | Részletes leírás a többi metódus leírása alatt. |
| public override void tryAttacking() | Megpróbálja az első sikeres találatig a Spell1Dir, Spell2Dir és AttackDir segítségével megtámadni az elérhető ellenfeleket. |

* FixedUpdate

A saját *health* adattagjának értékét a hozzá tartozó Animator típusú *anim* változónak a *health* nevű változójának adja. A Character osztály által örökölt FixedUpdate függvényt meghívja. Ezután megvizsgálja, hogy a karakter szimulációs karakter vagy sem, és ha nem, de nem a karakter köre jön, megszakítja a FixedUpdate függvényt. Hogy a karakter köre jön-e, azt a GridModel típusú *script* nevű adattagjának *whoseTurn* nevű változója adja meg, melyet emiatt az AltarezController lekér.

* AttackDir

A karakter pozíciójától számítva a *h* és *v* paraméterek által megadott irányban lévő mezőn ellenőrzi, hogy áll-e ellenfél. Ezt a *script* adattag *checkEnemyInPosition* metódusával teszi meg, melyet az adott mezőre hív meg.

Amennyiben talál ellenfelet, a *script* adattag *attackEnemy* metódusával megtámadja azt, 50 sebzést okozva. Az animációját támadás állapotba állítja, a *h* és *v* paramétereknek megfelelő irányba.

* Spell1Dir

A Játékoskaraktert próbálja gyógyítani tíz ponttal. Amennyiben a Játékoskarakter jelenlegi *health* értéke 90 fölött van, a metódus cselekvés nélkül megáll. Máskülönben ellenőrzi, hogy a Játékoskarakter a szomszédos mezők egyikén áll-e. Hogyha igen, a *script* adattag *attackEnemy* metódusának -10 értéked adva paraméterül, gyógyítja a Játékoskaraktert. Az animációját varázslás állapotba állítja, mely során lefelé néz.

* Spell2Dir

Egy ellenfelet próbál megsebezni 20 ponttal. Megvizsgálja, hogy a karakter középpontú 7x7-es környezetben található-e ellenfél. Ezt a *script* adattag *checkEnemyInPosition* metódusával teszi meg, melyet az adott 7x7-es terület minden mezőjére meghív. Amennyiben talál egy ellenfelet, nem keres tovább.

Ha talált ellenfelet, az ellenfél pozíciójában létrehoz egy objektumot a *script* adattagja *makeSpell* metódusával. Paraméterként megadja a létrehozandó objektum pozícióját, valamint, hogy „snakebite” típusú objektumot hozzon létre. Ezután megtámadja a megtalált ellenfelet a *script* adattagja *attackEnemy* metódusával. Az animációját varázslás állapotba állítja, mely során lefelé néz.

### DemonMageController

Ez az osztály irányítja a két Démon Mágus karaktert.

* Adattagok

A Character osztály által örökölt adattagok.

* Metódusok

A Character osztály által örökölt metódusok.

| Szignatúra | Leírás |
| --- | --- |
| protected override void Start() | A Character Start() metódusa. |
| protected override void FixedUpdate() | A saját *health* adattagjának értékét a hozzá tartozó Animator típusú *anim* változónak a *health* nevű változójának adja, majd meghívja a Character FixedUpdate metódusát. |
| public override bool spell1Dir(float h, float v) | Részletes leírás a többi metódus leírása alatt. |
| public override bool spell2Dir(float h, float v) | Részletes leírás a többi metódus leírása alatt. |
| public override bool attackDir(float h, float v) | Részletes leírás a többi metódus leírása alatt. |
| public override void tryAttacking() | Megpróbálja az első sikeres találatig a Spell1Dir, Spell2Dir és AttackDir segítségével megtámadni az elérhető ellenfeleket. |

* AttackDir

A h és v paraméterek alapján megadott irányban leellenőrzi, hogy a következő négy mezőn áll-e ellenfél. Ezt a *script* adattagja *checkEnemyInPosition* metódusával teszi meg, melyet mind a négy mezőre meghív. Amint megtalál egy karaktert, nem vizsgál több mezőt.

Amennyiben talál ellenfelet, létrehoz egy Projectile típusú objektumot a *script* adattagja *makeProjectile* metódusával. Paraméterként megadja, hogy egy tűzgolyót („fireball”) kell létrehozzon, mely a karaktertől a célpont karakterig mozog, 25-ös sebzéssel, 10-es sebességgel. Az animációját támadás állapotba állítja, a *h* és *v* paramétereknek megfelelő irányba.

* Spell1Dir

Egy ellenfelet próbál megsebezni 40 ponttal. Megvizsgálja, hogy a karakter pozíciójához képest átlós irányban akármelyik mezőn van-e ellenfél 5 mezőn belül. Ezt a *script* adattag *checkEnemyInPosition* metódusával teszi meg, melyet az adott terület minden mezőjére meghív. Amennyiben talál egy ellenfelet, nem keres tovább.

Ha talált ellenfelet, az ellenfél pozíciójában létrehoz egy objektumot a *script* adattagja *makeSpell* metódusával. Paraméterként megadja a létrehozandó objektum pozícióját, valamint, hogy „fireblast” típusú objektumot hozzon létre. Ezután megtámadja a megtalált ellenfelet a *script* adattagja *attackEnemy* metódusával. Az animációját varázslás állapotba állítja, mely során lefelé néz.

* Spell2Dir

A megadott *h* és *v* paraméterektől függetlenül ellenőrzi, hogy a karakter körüli 5x5-ös mezőben tartózkodik-e ellenfél. Ezt a *script* adattagja *checkEnemyInPosition* metódusával teszi meg, melyet mind a huszonnégy mezőre meghív.

Amennyiben tartózkodik a területen ellenfél, a terület mind a 24 mezőjén létrehoz egy objektumot a *script* adattagja *makeSpell* metódusával. Paraméterként megadja a létrehozandó objektumok pozícióit, valamint, hogy „demonspike” típusú objektumot hozzon létre. Ezután megtámadja a megtalált ellenfelet/ellenfeleket a *script* adattagja *attackEnemy* metódusával. Az animációját varázslás állapotba állítja, mely során lefelé néz.

### Projectile

* Adattagok

|  |  |
| --- | --- |
| **Deklaráció** | **Leírás** |
| public float moveSpeed | Megadja a lövedék mozgásának sebességét. |
| private Vector3 target | Megadja a lövedék mozgásának célpozícióját. |
| public float wait | Megadja, mennyit várjon a lövedék a létrehozásakor, mielőtt elindul. |
| public Character targetChar | Megadja, melyik karakter a lövedék célpontja. |
| public int damage | Megadja, mennyit sebezzen a lövedék a célpontra, amikor eléri. |

* Metódusok

|  |  |
| --- | --- |
| **Szignatúra** | **Leírás** |
| void Start() | A *wait* adattagok 0.5-re állítja. |
| void FixedUpdate() | Részletes leírás a többi metódus leírása alatt. |
| public void changeTargetChar (Character targ) | A *targetChar* adattagnak értékül adja a paramétert, a *target* adattagnak pedig a paramétere pozícióját. |
| public void changeSpeed(float s) | A *speed* adattagnak értékül adja a paramétert. |
| public void changeDmg(int i) | A *damage* adattagnak értékül adja a paramétert. |

* FixedUpdate

Amíg a *wait* adattag értéke nullánál nagyobb, csökkenti az adattag értékét, majd megszakítja a metódust. Amikor az adattag értéke eléri a nullát, a függvény továbbhaladhat.

A saját pozícióját összeveti a célpont pozíciójával, és ennek megfelelően abba az irányba módosítja a saját pozícióját, hogy a célpontjához közelebb kerüljön. Amennyiben (*0.1 \* Time.timeScale2*) távolságon belülre ér a célpontjához, meghívja a *GameObject* osztály *attackEnemy* függvényét, melynek a *damage* és *targetChar* adattagjait adja paraméterül, megadva ezzel, hogy melyik karakternek mennyi sebzést okozzon. Ezután megsemmisíti a hozzá tartozó objektumot és saját magát.

### CameraController

* Adattagok

|  |  |
| --- | --- |
| **Deklaráció** | **Leírás** |
| public Character FollowTarget | Megadja, melyik karaktert kövesse a kamera. |
| public float MoveSpeed | Megadja, milyen sebességgel mozogjon a kamera. |

* Update metódus

Az osztály egyetlen metódusa. A FollowTarget adattag pozíciója felé mozog a MoveSpeed adattag értéke szerint.

### GridModel

* Adattagok

|  |  |
| --- | --- |
| **Deklaráció** | **Leírás** |
| public bool[,] tileWalkable | Egy 50x50-es tömb, melyben a GridModel számon tartja, hogy az adott koordinátán lévő mezőre lehet-e lépni. |
| public Character[] charList | Ebben tartja számon a GridModel az általa létrehozott karaktereket. |
| public Character DemonMage | Ennek az adattagnak a Unity felületén adtunk értéket. Segítségével hozunk létre DemonMage karaktereket. |
| public Character Amy | Ennek az adattagnak a Unity felületén adtunk értéket. Segítségével hozunk létre Amy karaktereket. |
| public Character Altarez | Ennek az adattagnak a Unity felületén adtunk értéket. Segítségével hozunk létre Altarez karaktereket. |
| public GridModel OtherGrid | Ennek az adattagnak a Unity felületén adtunk értéket. A két, játékmezőn lévő GridModel objektum ezen adattagjainak egymást adjuk értékül. |
| public new CameraController camera | Ennek az adattagnak a Unity felületén adtunk értéket. A játékmezőn alapból szereplő Camera objektumot kapja értékül. |
| private MyNode root | A MCTS gráfjának gyökércsúcsa. |
| private MyNode currNode | Az MCTS algoritmus éppen vizsgált csúcsa. |
| public float timer | A Time.timeScale változót módosítjuk vele. |
| public bool simulation, doSimul, simulating | A szimulációra vonatkozó igaz/hamis adattagok. A szimulációs GridModel simulation értéke igaz, a másiknak hamis, ezt a Unity felületén állítjuk be. A doSimul változó igazra változása esetén kezdi el a szimulációt a szimulációs GridModel. Amíg tart a szimuláció, a simulating értéke igaz, egyébként hamis. |
| private bool doingSelection, selectionEnded, begin | Ezen adattagok segítségével tartja számon és változtatja a szimuláció az éppen aktuális lépését. |
| public float timeSpeed | Ennek segítségével módosítjuk a Time.timeScale értéket, mellyel felgyorsítható a játék sebessége. |
| public Projectile tornado | Ennek az adattagnak a Unity felületén adtunk értéket. Segítségével példányosítjuk a Tornado Prefab-et. |
| public Projectile arrowDown, arrowLeft, arrowUp, arrowRight | Ezeknek az adattagoknak a Unity felületén adtunk értéket. Segítségükkel példányosítjuk az arrow- Down, Left, Up, Right Prefab-eket. |
| public Projectile fireBallDown, fireBallLeft, fireBallUp, fireBallRight | Ezeknek az adattagoknak a Unity felületén adtunk értéket. Segítségükkel példányosítjuk a fireball- Down, Left, Up, Right Prefab-eket. |
| public GameObject earthspikes, icespikes, demonspikes, fireblast, icetacle, torrentacle, lightning, shield, snake | Ezeknek az adattagoknak a Unity felületén adtunk értéket. Segítségükkel példányosítjuk a változó nevének megfelelő nevű Prefab-eket. |
| public GameObject dmgTxt | Ennek az adattagnak a Unity felületén adtunk értéket. Segítségével példányosítjuk a DamageText Prefab-et. |
| public bool spell1Used, spell2Used | Ezen adattagok segítségével tartjuk számon, hogy a játékos használta-e már a képességeit. |
| public Animator actionAnim | Ennek az adattagnak a Unity felületén adtunk értéket. Ennek segítségével változtatjuk a megjelenítendő akció ikont. |
| public int whoseTurn | Ezen adattag segítségével tartjuk számon, hogy mely karakter(ek) köre tart éppen. A 0 érték a Játékoskarakter, az 1 érték a Segédkarakter, a 2 érték pedig az ellenséges karakterek körét jelöli. |
| public int stepsLeft, actionLeft, commandLeft | Ezen adattagok segítségével tartjuk számon, hogy a játékos körén belül hány akciót használt már el. |
| private int spedUp | Ezen adattag értéke szerint változtatjuk a timeSpeed adattagot a szimuláció kezdetén. |

* Metódusok

|  |  |
| --- | --- |
| **Szignatúra** | **Leírás** |
| void Start() |  |
| void Update() |  |
| private bool canProceed() | Igazat ad vissza, ha a *charList* összes elemének *proceed* változójának értéke igaz, egyébként hamisat. |
| private void createStartCharacters() | Létrehozza a kezdő karaktereket, a megadott pozíciókban, és hozzáadja azokat a *charList* adattagjához. |
| public Character checkEnemyInPosition(int layer, Vector2 position) | Amennyiben talál a *position* pozícióban olyan karakter objektumot, amely eltérő rétegen van, mint a megadott *layer* paraméter, visszatér vele, egyébként null értéket ad vissza. |
| public void attackEnemy(int damage, Character target) | A *target* karakter életét a *damage* értékével csökkenti, és létrehoz a karakter felett egy *damageText* objektumot a *dmgTxt* adattag használatával. |
| private void fillFieldValues(int[,] field, int x, int y) | A *field* kétdimenziós tömböt feltölti az *x,y* paraméterek által kijelölt kezdőpozícióból (melynek az értéke 1). Ehhez megvizsgálja a szomszédos mezőket, és amelyik értéke még nulla, egyel nagyobb értéket ad neki, mint a jelenlegi mezőnek. Ezután ugyanezt az összes feltöltött szomszédos mezőre is végrehajtja. |
| private Vector3 findMin(Vector2 curr, int[,] field) | A *curr* paraméter *x* és *y* adattagja alapján megadott mezőjét vizsgálja a *field* paraméternek. Ha az adott mezőnek bármelyik szomszédos mezőjén kisebb értéket talál, mint az adott mező értéke, akkor visszatér azzal a szomszédos mezővel. |
| public Vector3[] pathFinding(Character recievedCharacter, Vector2 Target) | A *recievedCharacter* pozíciójából keres utat a *Target* pozícióba. Ehhez létrehoz egy *field* változót, mely egy 50x50-es kétdimenziós tömb. Ebben az akadályok értékét -1-re állítja, a vizsgált karakter pozíciójának megfelelő indexű értéket egyre, majd meghívja az adott pozícióra a *fillFieldValues* függvényt. Ezután a *Target* pozícióból kiindulva a *findMin* függvény használatával visszakeresi az utat a vizsgált karakter pozíciójához, ezt az utat menet közben elmenti, majd a függvény végén visszaküldi. |
| private MyNode nextStep(MyNode node) | Az MCTS algoritmus lépéseiért felelős. Megvizsgálja, hogy éppen melyik lépésnél tart az algoritmus, és annak megfelelően meghívja az MCTS függvényeit (selectChild, makeRandomChild, enemyWins, backPropogate actionToCharacters), valamint karbantartja az aktuális lépést jelző változókat. |
| private bool enemyWins() | Igazat ad vissza, ha a két ellenséges karakter együttes életpontjai meghaladják a Játékos- és Segédkarakter együttes életpontjait. |
| private void backPropogate(MyNode node, bool win) | A *node* csúcs *visited* paraméterét növeli, a *wins* paraméterét pedig akkor, ha *win* igaz. Ezt rekurzívan meghívja a csúcs szülőjére (ha létezik) ugyanazzal a *win* értékkel, majd törli a csúcsot, ha annak *simNode* értéke igaz, vagyis a Szimuláció lépés során keletkezett. |
| private MyNode bestChild(MyNode node) | Visszaadja a *node* csúcs gyerekei közül azt, amelyiknek a legnagyobb a visited értéke. Ha nincs gyereke, magát a csúcsot adja vissza. |
| private MyNode selectChild(MyNode node) | A Szimuláció *Selection/Kiválasztás* lépésében tárgyalt képlet alapján kiválasztja a *node* csúcs egyik gyermekét vagy önmagát. |
| private void copyOriginal() | Lemásolja a szimulációs karakterek adatait a nem-szimulációs karakterek adatai szerint a *Character setAttrTo* függvényével. |
| private MyNode makeRandomNode(MyNode node) | A *randomAction* függvény segítségével létrehoz egy véletlenszerű akcióval ellátott csúcsot, a *node* csúcs *enemyAction* paramétere alapján. |
| private MyNode makeRandomChild (MyNode node, bool expansion) | Létrehoz a *makeRandomNode* függvénnyel egy csúcsot, melynek a *node* függvényt adja paraméterül. Ezután a létrehozott csúcsot a *node* gyermekének teszi meg. |
| private string convertActionToString(int[] action) | Az *action* paraméter elemeinek értéke alapján visszaad egy szöveget. A számértékeket egytől négyig irányoknak felelnek meg, öttől hétig pedig akcióknak. |
| private int[] convertStringToAction(string action) | Az *action* paraméter alapján visszaad egy int típusú tömböt. Lényegében a *convertActionToString* ellenkezője. |
| private int[] randomAction (Character ch) | A ch karakter számára létrehoz egy tetszőleges akciót. Véletlen számokat generál, ezeket alakítja akcióvá, majd ellenőrzi, hogy az adott akció a karakter pozíciójából megléphető-e. |
| private int[] rndToAction(int move) | Egy számot alakít át egy int típusú tömbbé. Ez két lépésnek és egy akciónak feleltethető meg. |
| private bool checkAction (Character ch, int[] arr) | A *ch* karakter pozíciója alapján megvizsgálja, hogy az *arr* paraméter értékei által reprezentált akciók helyesek-e. Amennyiben foglalt mezőre (fa, más karakter), vagy más karakter *moveTo* által meghatározott pozíciójára lép, akkor hamisat ad vissza. Ha a karakter a többi karakterek közül legalább egytől legalább hét mezőre áll legalább az egyik irányban, akkor megvizsgálja, hogy a karakter mozgása a többi karakterhez közelebb vinné-e, és ha nem, szintén hamisat ad vissza. Máskülönben igazat ad vissza. |
| public void actionToCharacters (MyNode node) | A *node* paraméter *nodeAction* adattagjának értékét a *convertStringToAction* függvénnyel átalakítja, majd a *node* *enemyAction* adattagja alapján a két ellenséges, vagy a Játékos- és Segédkarakter *doAction* függvényével az átalakított értéket a karaktereknek átadja. |
| public GameObject makeSpell (String str, Vector3 pos) | A *str* paraméter alapján megadott típusú Prefab objektumot hoz létre a *pos* paraméter által megadott pozícióban. |
| public Projectile makeProjectile (Character origin, Character target, String type, int dmg, float speed) | Létrehoz egy *Projectile* osztályú objektumot, melyhez a Prefab-et a *type* paraméter adja meg, kezdő pozíciónak az *origin* karakter pozícióját állítja be, a *Projectile* osztály *target*, *dmg*, *speed* adattagjait pedig a megegyező nevű paraméterek értékeivel teszi egyenlővé. |
| public void controlAltarez (float h, float v) | A segédkaraktert irányítja az iránygombokkal, a Felhasználói Dokumentációban leírtaknak megfelelően. Ehhez használja a *getClosestEnemy* és a *moveAltarezTowardsCharacter* függvényeket. |
| private Character getClosestEnemy() | Visszatér a Segédkarakterhez legközelebb álló ellenséggel. |
| private void moveAltarezTowardsCharacter (Character ch) | A *ch* karakter pozíciója szerint, ahhoz közelítve mozgatja a Segédkaraktert. |

## Tesztek

### Elvárt működés

## Továbbfejlesztési lehetőségek

# Összefoglalás

# Fogalomtár

Játékoskarakter – A karakter, amit a játékos irányít.

Segédkarakter – A játékossal szövetséges karakter.

Démon Mágus – A játékossal ellenséges karakter.

Monte Carlo Tree Search (MCTS) algoritmus – Az ellenséges karakterek akcióit irányító algoritmus.

Amy – A Játékoskarakter neve.

Altarez – A Segédkarakter neve.

Játéktér – A játék színtere, ezen mozognak a karakterek.

Akció – A karakterek rendelkezésére álló cselekvések csoportja.

Mozgás – A karakterek egyik akciója, mely során egy másik mezőre mozognak.

Alaptámadás – A karakterek egyik akciója, mely során megtámadnak egy ellenséget. Mindegyik karakter esetében más.

Képesség1 – A karakterek egyik akciója. Mindegyik karakter esetében más.

Képesség2 – A karakterek egyik akciója. Mindegyik karakter esetében más.

Segédkarakter irányítása – A Játékos egyik akciója. Ezzel tudja irányítani a Segédkarakter mozgását.

Parancs akció – A „Segédkarakter irányítása” akció másik neve.

Támadás akció – Az alaptámadás, és az ellenséget sebző Képességek összefoglaló neve, karakterenként.

Mező – A játékteret alkotó négyzet alakú tér. Ezeken mozognak a Karakterek.

Kör – A játék egy időbeli egysége, melyben először a Játékos választ akciókat, majd a Segédkarakter és a két Démonkarakter cselekszik.

Szimuláció – Az MCTS egyik lépése. Véletlenszerű akciókat játszat le a karakterekkel.

Wavefront algoritmus – A játék útkereső algoritmusa.

Earthspikes

Icespikes

Fireblast

Icetacle

Torrentacle

Tornado

Lightning

Shield

Snake

Arrow

Fireball