EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Algoritmusok és Alkalmazásaik Tanszék

Evolúciós algoritmus fogó-játékban

Veszprémi Anna
mesteroktató

Árvai Balázs
Programtervező informatikus BSc.

Budapest, 2022

Tartalom

I. Bevezetés

Szakdolgozatom a gépi tanulás folyamatát vizsgálja egy egyszerű fogó-játék szimulációjában. A játék során két különböző szerepű játékos versenyez egymással: fogók és menekülők. Az egyes játékosok mozgását egy-egy neurális háló irányítja, melyet egy evolúciós algoritmus alakít a játék fordulói között. A játékhoz több különböző pálya is tartozik, melyek célzottan arra készültek, hogy egy adott feladatra tanítsák meg a játékosokat. Ebben segítenek a különböző értékelési rendszerek, melyek a játék során büntetik illetve jutalmazzák a játékosokat egy-egy tevékenységért. A mesterséges intelligenciát felépítő algoritmusok, illetve a játék megvalósítása is a Unity játékfejlesztői keretrendszerben történik.

I. 1. A gépi tanulás

A mesterséges intelligencia kutatás a számítógép-tudomány egyik legfrissebb ága, melynek lehetőségeit még napjainkban sem térképezték fel teljesen. Az informatika ezen ágazata olyan problémákra keres megoldó módszert, melyekre a probléma komplexitása miatt hagyományos módszerekkel nem lehet, vagy csak nagyon költségesen lehetne kiszámító algoritmust tervezni. E tudományág két rendkívül erős, ám önmagukban korlátolt eszközei az evolúciós algoritmusok és a neurális hálók. A két módszer kombinálása azonban új lehetőségeket nyit meg, és közös használatukkal lehetőség nyílik egy, a témabeli fogójátékot megtanulni képes mesterséges intelligencia megalkotására.

I. 2. A játékról

A szimuláció a jól ismert fogó-játék egyik változatát dolgozza fel. A játék egy elkerített statikus pályán játszódik, melyen belül szerepelhetnek házaknak nevezett biztonságos területek. A dinamikus elemeket a játékosok jelentik, amelyek közül a fogók feladata a menekülők elkapása, míg a menekülőké az, hogy minél tovább játékban maradjanak. Egy fogó elkap egy menekülőt, ha vele érintkezésbe kerül. Ekkor az elkapott menekülő azonnal kiesik a játékból. Egy menekülő úgy lehet minél eredményesebb, ha minél tovább játékban marad, mert egyetlen fogó se érinti meg, vagy ha érintkezésbe lép egy házzal, hiszen onnantól biztonságban van, és ebben az esetben úgy vesszük, a játék végéig senki nem kapta el, de a pályáról az egyszerűség kedvéért eltávolítjuk.

I. 3. A témaválasztás indoklása

A szakdolgozatom elsődlegesen a tanulás megvalósításával és folyamatával foglalkozik egy konkrét, egyszerű példán keresztül. Választásom azért erre a témára esett, mivel régóta foglalkoztat az evolúciós algoritmusok megvalósítása és újabban a neurális hálókban rejlő lehetőségek is. Eleinte mindenképpen evolúciós algoritmust akartam alkotni, azonban nehezen találtam konkrét példát, melyre alkalmazni tudnám és újnak is hatna. A Youtube-on megláttam az „OpenAI" csatorna egy videóját [1], melyben az evolúciós algoritmust egy háromdimenziós bújócska játék keretei között mutatnak be, aminek hatására egyszerű gyerekjátékokat kezdtem vizsgálni. Bár megfordult a fejemben többek között a kő-papír-olló, a „padló lávából van" és a kiütő is, végül a fogó-játék mellett döntöttem, mivel úgy véltem, egy kétdimenziós reprezentációja könnyen megvalósítható, mindamellett a játékban megjelenő különböző szerepek és szabályok érdekes viselkedéseket alakíthatnak ki a mesterséges intelligenciában.

I. 4. A tanulás folyamata

A program mesterséges intelligenciája kombinálja az evolúciós algoritmusok és a neurális hálók technológiáját.

I. 4. 1. A neurális hálózatokról röviden

A neruális hálózat egy biológia ihlette modell, melyet általában gráfokkal reprezentálunk. Programomban egy előrecsatolt neurális hálózatot használok a játékosok döntéshozatalának megtestesítésére. A neurális háló rétegekből épül fel, melynek típusai: bemeneti, közbülső és kimeneti réteg. Minden réteg neuronokat tartalmaz, melyek -1 és 1 közötti értéket vehetnek fel. Két szomszédos réteg esetén az egyik réteg minden neuronja össze van kötve a másik réteg minden neuronjával, ahogyan egy gráf csúcsai között szerepelhetnek élek, ha a neuronokat a csúcsoknak feleltetjük meg. Ezek az élek úgynevezett súlyokat határoznak meg, ami befolyásolja egy neuron értékének kiszámítását. A rétegekhez felveszünk egy-egy konstans neuront is, melynek értéke mindig 1, és a célja, hogy a kiszámítási egyenlet értékkészletét növeljük.

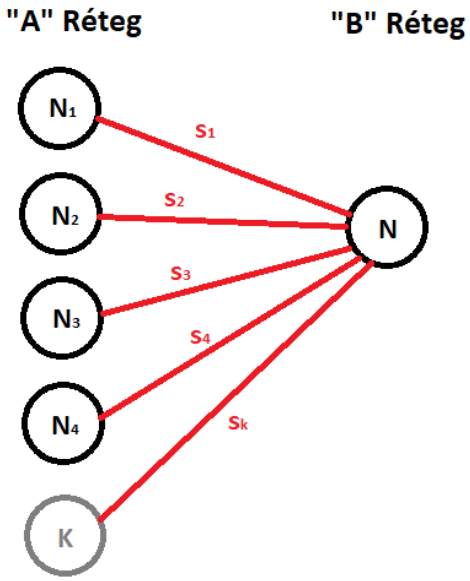
A neurális hálózat első rétege mindig a beviteli réteg, mely értékeit jelen esetben a játékos által „látott" adatok határozzák meg, míg az utolsó réteg mindig a kimeneti réteg, ami ebben a reprezentációban a játékos mozgását befolyásolja. A kettő között tetszőleges számú közbülső réteg szerepelhet. A neurális háló előrecsatolt tulajdonsága arra utal, hogy az információ csakis egy irányba, a beviteli rétegtől a kiviteli réteg felé terjed.

Az 1. ábrán láthatunk egy példát két réteg neuronjai közötti kapcsolat felépítésére.

Ebben az esetben az N_1,N_2,N_3,N_4 az A réteg neuronjait jelölik. K az A réteghez tartozó konstans neuron.N a B réteg egy neuronja. Az s_1,s_2,s_3,s_4,sk az összekötésekhez tartozó súlyok. Ekkor az N neuron értéke:

$$N = f(N_1 s_1 + N_2 s_2 + N_3 s_3 + N_4 s_4 + K s_k),$$

$$ahol\ az\ f(x)\ egy\ olyan\ függvény, ami\ az\ x\ értéket\ a\ transzformálja\ a\ [-1,1]\ tartományra.$$



1. ábra Példa neurális hálózat rétegei közötti kapcsolatra

I. 4. 2. Az evolúciós algoritmusok

Az evolúciós algoritmusok a neurális házatokhoz hasonlóan szintén biológia ihlette modellek. Az algoritmus ciklikusan ismétli ugyanazokat a lépéseket, miközben újabb generációkat hoz létre. Egy generáció egy bizonyos populációból áll, melyen az algoritmus műveleteket hajt végre. Egy populáció egyedek összessége, melyek jelen esetben a játékosokhoz tartozó neurális hálózatok adatai. Az algoritmus célja, hogy generációnként finom változtatásokat eszközöljön ezeken a neurális hálókon, ezzel optimalizálva a játékosok viselkedését. Fontos, hogy a menekülők és a fogók két külön populációt képviselnek a modellben, azonban az algoritmus ugyanazokat a műveleteket hajtja végre mindkét populáción. A legelső generáció egyedeit véletlenszerűen generáljuk, vagy fájlból olvassuk be.

Evolúciós lépések

1. Minden generáció a játék egy fordulójával kezdődik. Ezalatt lehetőségük van a játékosoknak pontokat szerezniük adott idő alatt.
2. A következő lépés az egyedek kiértékelése. Egy neurális háló hatékonysága a hozzá tartozó játékos pontszámától függ. Hatékonyság szerint sorba rendezzük a populációt.
3. Ezután kiválasztjuk a populáció azon egyedeit, akikből a következő generáció populációját létre akarjuk hozni.
4. A keresztezés során a kiválasztott egyedek részeiből hozunk létre új egyedeket.
5. Az új egyedek részeit kis valószínűséggel megváltoztathatjuk. Ennek szerepe, hogy még ha a populáció nagy része ugyanazt a viselkedést követi, akkor is létrejöhessenek egyedek, amelyek teljesen eltérő viselkedést mutatnak.
6. Az új egyedekből létrehozzuk az új generációt és elölről kezdjük a lépéseket.

II. Felhasználói dokumentáció

II. 1. A felhasznált technológiák

A program a Unity játékfejlesztői környezet 2020.3.8-as verziójában fut. Ez a játékmotor többek között olyan beépített funkciókkal segítette a fejlesztést, mint hitboxok használata, raycasting, vagy éppen képkocka-frissítéshez kötött metódusok.

A program nyelve a Unity által legenerált fájlokon kívül C#-ban írodott. Bár a Unity több programnyelvet is támogat, úgy mint C, C++, Rust vagy Python, a C# az alapértelmezett és messze a legjobban támogatott közülük.

II. 2. A telepítés folyamata

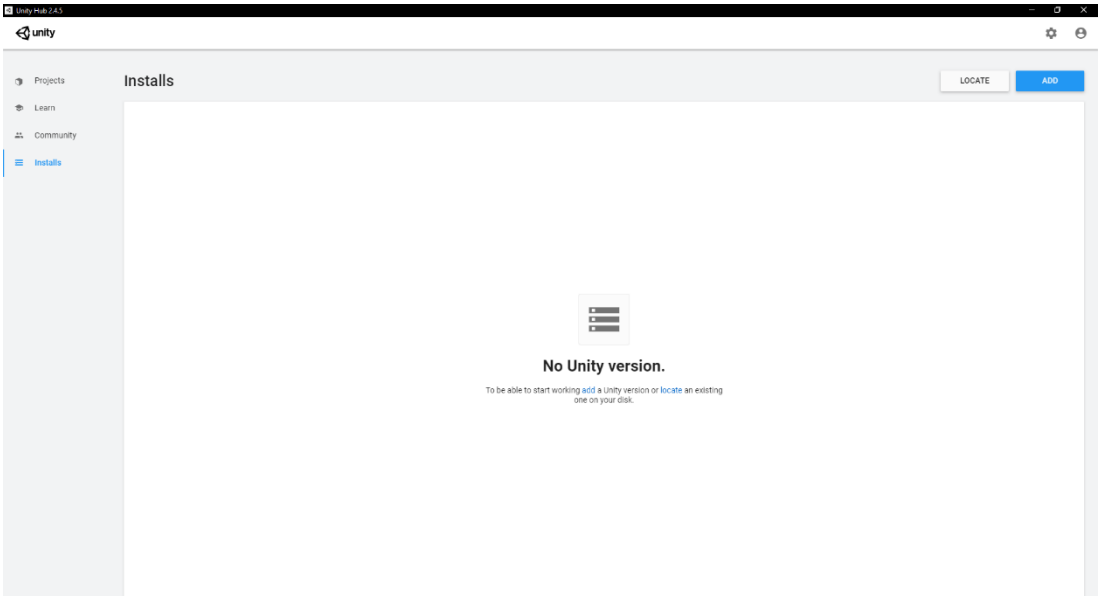
A program tesztelése az alábbi környezeten történt: Intel Core i5 processzor, 16GB DDR4 memória, NVIDIA GeForce GTX 1050 Ti videókártya, Windows 10 Pro operációs rendszer.

A csomag Runable nevű mappája a pályákkal megegyező nevű almappákat tartalmaz, melyek pedig futtatható Tag.exe-ket tartalmaznak. Az almappa elnevezése utal a tartalmazott pályára. Az .exe-t elindítva az adott pálya egy előre beállított konfigurációval teljes képernyős módban betölt és a szimuláció elkezdődik rajta. A program az előre beállított maximum generációs számig fut, utána bezáródik, de közben is kiléphetünk a programból az escape gomb lenyomásával. A lefordított paraméterektől függően a program képes menteni és betölteni adatokat. Bár ezzel a módszerrel könnyen láthatunk konkrét példát a tanulási és játék folyamatra, mégis sok lehetőséget elveszítünk, amit a Unity nyújthat.

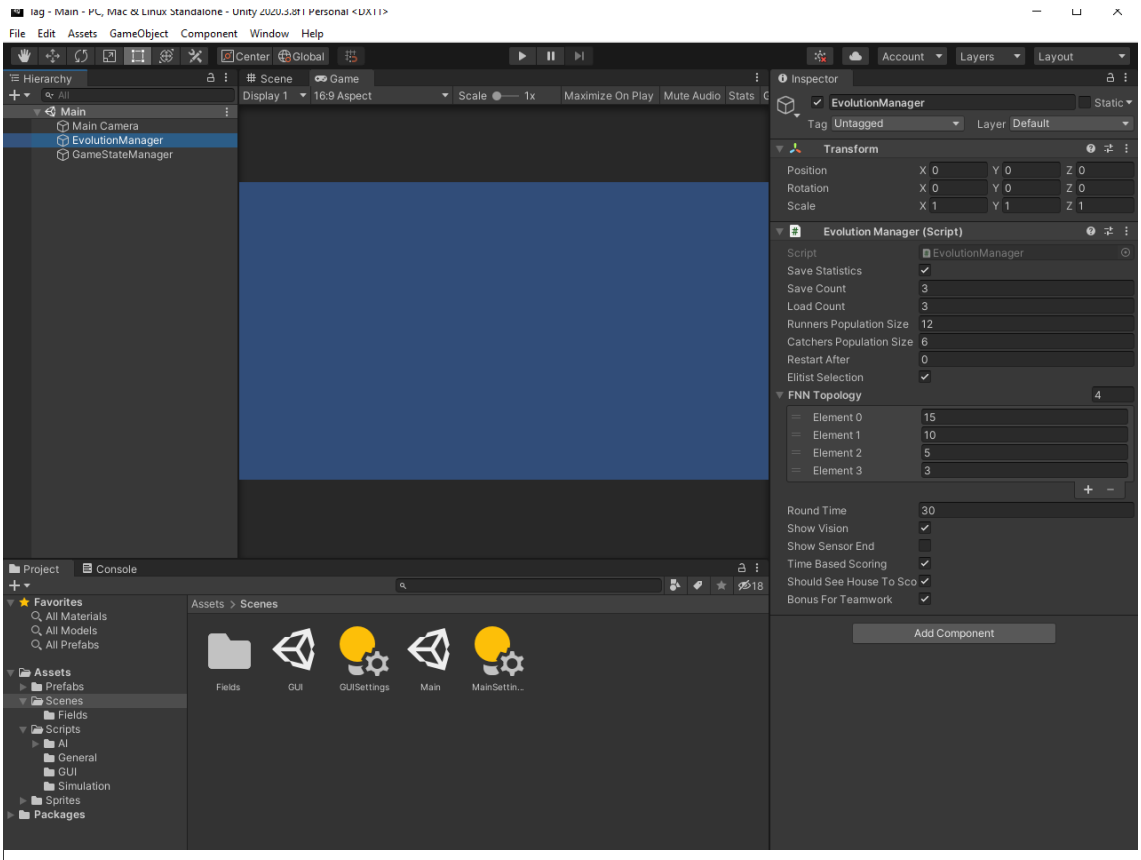
A For Unity mappa tartalmazza a projekt létrehozásához szükséges fájlokat. Azonban előbb szükséges a Unity 2020.3.8-as verziójának telepítése. Bár a program csupán alapvető és ingyenes funkcióit használja a Unitynak, azonban mivel nem biztos, hogy ezek támogatása a jövőben is megmarad, így az említett verzió telepítése ajánlott. Ehhez tartalmaz a csomag egy UnityHubSetup.exe-t, amivel feltelepíthető a 2. ábrán látható Unity Hub, a Unity általános verizókezelője és indítója. Ezután a UnityDownloadAssistant-2020.3.8f1.exe-vel lehet telepíteni a megfelelő verziót, amit utána a hozzá is kell adni Unity Hubban az Installs menüpont alatt.

Így már könnyedén készíthető egy új kétdimenziós, üres, megfelelő verziójú projekt a Projects menüpontban. Ezután elindul a Unity általános kezelőfelület [2], a létrejött projekt könytárába pedig be kell másolni a For Unity mappa tartalmát, felülírva ezzel a

Unity által legenerált projekt fájlokat. Az Assets mappa tartalmaz minden kódot, képet, modellt és jelenetet, míg a ProjectSettings az editor általános beállításáért felelős, mint például a fizikai paraméterek. A Loader mappa üres, erre azonban akkor lesz szükség, ha egy korábbi szimuláció adait szeretnénk betölteni egy új szimuláció indulásakor.



2. ábra Unity Hub



3. ábra A Unity általános kezelőfelülete

II. 3. A Unity általános kezelése

A projekt automatikusan érzékeli a bemásolt fájlokat és a 3. ábrán látható módon frissíti az általános kezelőfelületen jelzett információkat.

Alul a Project ablakban a projekt könyvtárán belül navigálhatunk. Közvetlenül a Projekt-fül mellett található a Console-fül, melyre kattintva átválthatunk a Console ablakra, amiben a program által kiírt log-okat és hibaüzeneteket tekinthetjük meg.

Középen a Unity két fő ablaka helyezkedik el hasonlóképpen, a Scene és a Game. A Scene-ben egy jelenet felépítését tekinthetjük meg és módosíthatjuk, míg a Game ablakban a játékprogram futása során a kamera által megjelenített képet láthatjuk.

E két ablak felett közvetlenül található három gomb, melyek balról jobbra a Play, Pause és Step gombok. A play gombbal elindíthatjuk a program futását, illetve ha már fut, akkor teljesen leállíthatjuk. A Pause gombbal szüneteltethetjük vagy folytathatjuk a program futását. A Step gombbal pedig a szüneteltetett programot léptethetjük képkockánként, ami lehetőséget kínál alaposabb vizsgálatra.

A Scene ablaktól balra található a Hierarchy ablak, amiben az aktuális betöltött jelenet vagy jelenetek játékelemeit és struktúráját láthatjuk. Ezek vizsgálatában segít a Scene ablakától jobbra található Inspector ablak, amiben a Hierarchy ablakban kiválasztott játékelem komponenseit és azok paramétereit vizsgálhatjuk meg és szabályozhatjuk. Ez a tulajdonság lehetővé teszi, hogy a szimuláció tulajdonságait könnyedén alakítsuk, akár közvetlenül indítás előtt.

II. 4. A játékelemek és alapvető szabályok

A játék számos jelenetből épül fel, amelyeket a szimuláció megkezdésének legelején a GameStateManager játékelem GameStateManager kódja tölt be. Kivételt képez a Main jelenet, amelyben a GameStateManager játékelem is található, mivel azt eleve be kell tölteni, hogy megkezdhessük a szimulációt.

II. 4. 1. A Main jelenet

Ez a jelenet adja a szimuláció alapvető vázát. Bár több másik jelenetet is betölt a futtatása során, rendelkezik saját játékelemekkel is:

Main Camera

Ez a játékelem egy egyszerű kameraelem, mely nem rendelkezik saját kód komponenssel, szerepe csupán az, hogy megjelenítse a szimuláció futása során a Game ablakban a játék aktuális állását. Fontos, hogy ez az elem úgy van beállítva a jelenet

háromdimenziós terében, hogy a később betöltött pálya jelenet elemei benne legyenek a kamera látószögében.

EvolutionManager

Ez a játékelem rendelkezik az EvolutionManager kód komponenssel, ami a szimuláció második legnagyobb irányító egysége. A komponens paramétereivel szabályozhatjuk a szimuláció tulajdonságait.

GameStateManager

Egy rendkívül rövid GameStateManager kód komponenst tartalmazó játékelem, ami azonban a program legmagasabb szintű irányító egységét rejti. Egyetlen paraméterével a betölteni kívánt pálya adható meg.

II. 4. 2. A UI jelenet

Ez a jelenet jeleníti meg a szimuláció során a Game ablakban a játék aktuális statisztikáit. Fő játékeleme:

UI

Ez alá a játékelem alá több játékelem is tartozik, melyek a statisztikai szöveg megjelenítéséért felelősek. Közvetlen rendelkezik a UIController nevű komponenssel, mely felelős a statisztikák folyamatos frissítéséért a szimuláció során.

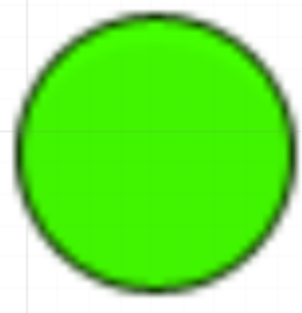
II. 4. 3. A pálya jelenetek

A pályák megjelenése változó, azonban az alapvető felépítésük minden esetben megegyezik. Az alapvető játékelemek:

FieldManager

Ez a játékelem a pálya jelenet irányító egysége. Tartalmazza a FieldManager kód komponenst, aminek segítségével meghatározhatjuk, hogy az adott pályán maximálisan hány játékos lehessen a különböző játékostípusokból, illetve, hogy az egyes játékosok pontosan hol és hogyan helyezkedjenek el a pályán egy forduló megkezdésekor.

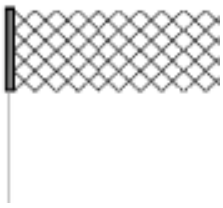
Ez a játékelem magába foglalja az összes House játékelemet is (ha vannak a pályán). Ezek a játékelemek jelentik a menekülők számára a biztonságos területet és rendelkeznek egy House kód komponenssel, ami a házakba beérkezett menekülőket számlálja, illetve egy CircleCollider2D [3] komponenst, ami a Unity egy beépített komponense, és ami a különböző ütközések érzékeléséért felelős. Vizuális reprezentációjuk egy egyszerű zöld körlap, ahogyan az a 4. ábrán látható.



4. ábra Egy ház reprezentációja

Environtment

Ez a játékelem semmilyen szinten nem rendelkezik kód komponenssel, azonban magába foglalja a pálya falait. Ezek a falak Wall játékelemként vannak feltüntetve és fontos, hogy rendelkeznek egy BoxCollider2D [4] komponenssel, aminek szerepe megegyezik a már korábban említett CircleCollider2D komponenssel. A 5. ábrán láthatunk példát egy függőleges és egy vízszintes falszakaszra is. A falak kerítés jellegének és a falak találkozásánál szereplő oszlopoknak csupán esztétikai szerepe van.



5. ábra Falak reprezentációja

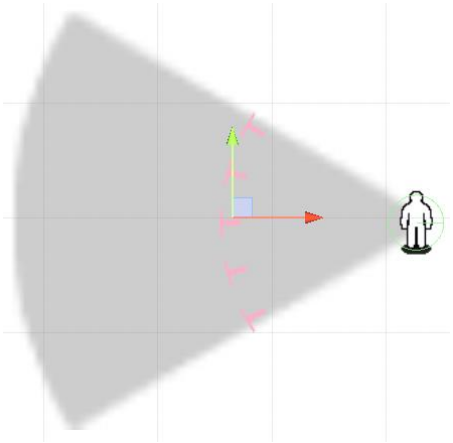
Játékosok

A pálya tartalmaz számos játékost. Valójában ezek a játékelemek nem aktívak a szimuláció során, a FieldManager kód csupán minden forduló elején ezeket a játékelemeket veszi mintának, hogy a játékosokat hova és milyen beállításokkal hozza létre. Egy játékos két kód komponenssel is rendelkezik, a PlayerController és a PlayerMovemenet komponensekkel. Előbbi a játékos általános viselkedéséért felelős, míg utóbbi pusztán a mozgással és érzékeléssel kapcsolatos logikát tartalmazza. A játékos rendelkezik továbbá egy CircleCollider2D komponenssel is.

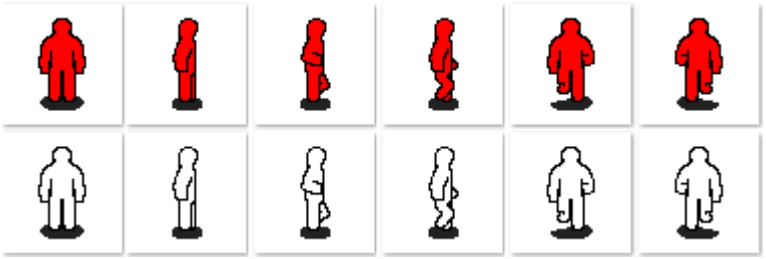
A játékos további játékelemeket is magába foglal, többek között 5 sensor és 1 vision játékelemet. A sensorok mind rendelkeznek egy-egy Sensor kód komponenssel, melyek az érzékelésért felelősek, míg a vision egy Vision kód komponenssel rendelkezik, ami

pusztán a látómező megjelenítésének szabályozásáért felelős. A 6. ábrán láthatjuk egy játékos vizuális reprezentációját: az ember modellje jelöli a játékos testét, a szürke terület a látómezőt, míg a rózsaszín „T” jelek az egyes szenzorok végeit, melyek távolságát a játékos testétől a Sensor programja futási időben szabályozza.

A játékosokat típusuk alapján különböző színnel jelöljük a játék során, a fogókat pirossal, míg a menekülőket fehérrel, más vizuális különbség azonban nincs a két játékosfajta között. A mozgás típusát animációval is indikáljuk, amelyet a játékos képének gyors és egyenletes változtatásával hozunk létre. Az eltérő játékos képek megfigyelhetőek a 7. ábrán.



6. ábra Egy játékos reprezentációja



7. ábra Játékos animációk

II. 4. 4. A játék szabályai

Általános szabályok

A pályák különböző felépítése a tanulási folyamatot kívánja befolyásolni, azonban bizonyos szabályok általánosan igazak minden pályán minden beállítás mellett.

A játékosoknak két típusa van: menekülő és fogó. Egyes pályákon tanítási célból nem szerepel fogó játékos, menekülő játékosnak azonban minden pályán szerepelnie kell. A fogó játékosok célja, hogy elkapjanak menekülő játékosokat, amit úgy tehetnek meg, hogy ütköznek velük. A menekülő játékosok feladata, hogy elérjék a házak jelentette

biztonságos területet, vagy minél tovább életben maradjanak. Előbbit úgy tehetik meg, hogy ütköznek valamelyik ház játékelemmel, míg utóbbit úgy, hogy nem ütköznek fogóval vagy fallal. Ha akár fogó, akár menekülő játékos ütközik egy fallal, akkor úgy vesszük, hogy ki akart lépni a játékterületről és a játék adott fordulójából azonnal kiejtjük. Ez a szabály azért fontos, mivel a játékosok mozgási és érzékelési tulajdonságaik miatt könnyedén beleakadhatnak a falakba.

Specifikus szabályok

Bizonyos szabályok paraméterekkel állíthatóak, ezzel befolyásolva a játékot.

- Időalapú pontozás

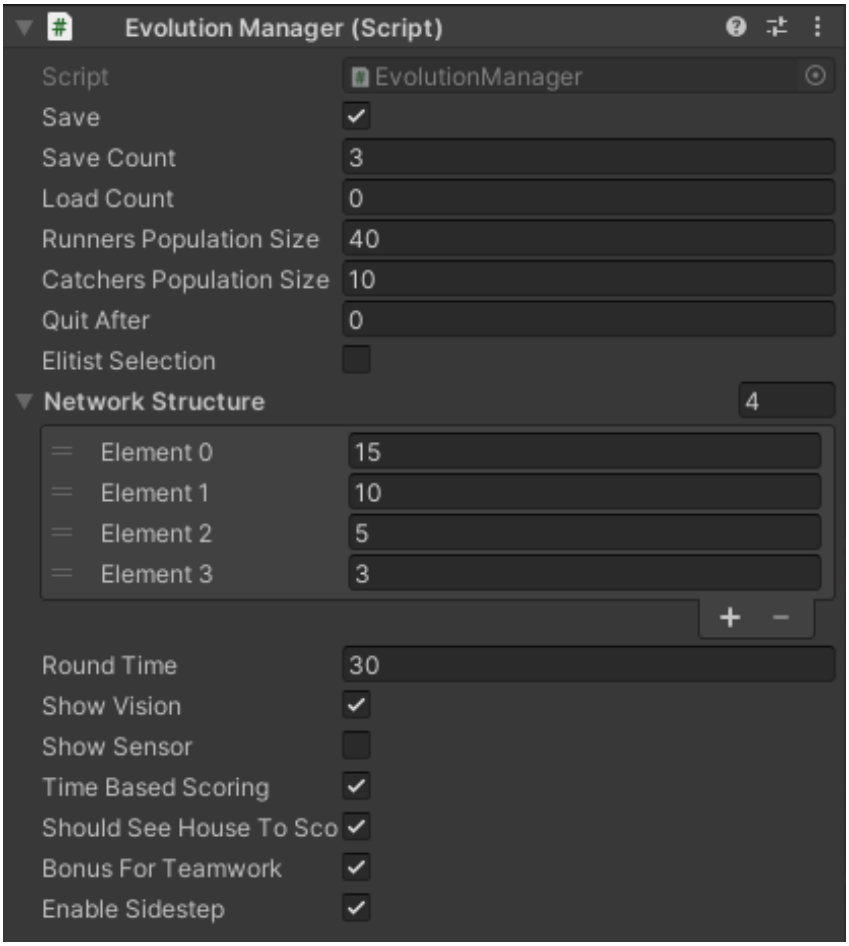
  Az időalapú pontozás során a menekülő játékosok bónuszpontot kapnak a szerint, hogy az adott fordulóban mikor estek ki. Ha egy falnak vagy menekülőnek ütköznek, a kapott pont egyenlő a fordulóból eltelt idővel, míg ha elérnek egy házat, akkor a bónuszpont a fordulóból eltelt idővel fordítottan arányos. Ha ez a szabály nincs bekapcsolva, a menekülők csakis akkor szerezhetnek pontot, ha elérnek egy házat.

- Ház láthatóság szabálya

  A menekülő játékosok csakis akkor kapnak pontot egy ház eléréséért, ha az érintkezés pillanatában valamelyik szenzorjuk érzékelte a házat.

- Csapatmunka szabálya

  A fogó játékosok bónuszpontot kapnak, ha látnak egy menekülő játékost az elkapás pillanatában. Ennek célja, hogy a fogókat akkor is díjazzuk, ha nem sikerült elkapniuk egy menekülőt, de sikerült egy másik fogóhoz odacsalniuk.



8. ábra EvolutionManager

II. 5. Programspecifikus beállítások

II. 5. 1. Evolúciós paraméterek

Az evulúció paraméterei az EvolutionManager játékelem megegyező nevű kód komponense alatt állíthatóak, ahogyan az a 8. ábrán látható. A paraméterek az alábbiak:

Save

Ezt a checkboxot bepipálva a program lementi a statisztikákat, beleértve a Save Count értékével megegyező legjobb menekülő és fogó játékosok neurális hálójának súlyait, melyet generációnként frissít. A mentésekhez létrehoz a projekt alatt (vagy a futtatható .exe mellett) közvetlenül egy ”Simulation_”+{{Pálya neve}}+” ”+{{jelenlegi dátum}} nevű mappát. Létrejön továbbá egy megegyező nevű .txt fájl is ide, ami tartalmazza a szimuláció főbb paramétereit, illetve bővül minden generáció kielemzése után a legjobban teljesítő menekülő és fogó játékosok pontszámával, illetve a házba beérkezett menekülők számával.

Save Count

Ez a szám meghatározza, hogy egy generáció kiértékelése végeztével a legjobban teljesítő játékosok közül hány felépítését mentsük el, illetve felelős a szimuláció általános paramétereinek mentésért is. A mentés csak akkor történik meg, ha a Save paraméter igaz, és külön történik a fogókra és a menekülőkre, valamint tartalmazza a neurális hálójuk súlyait. Az egyes játékosokhoz tartozó adatok külön szövegfájlokba kerülnek, melynek elnevezései az alábbi módon épül fel: ”gene_”+{{játékos típusa, Runner/Catcher}}+”_”+{{játékos indexe}}+”.txt”.

Ha a Save Count nagyobb mint a valamelyik játékostípus populációjának mérete, akkor a program az összes adott típusú játékoshoz tartozó neurális hálók adatait lementi. Minden generációról külön mentés készül, ami a ”Generation_”}}+”_”+{{generáció száma}}. Az összes ilyen mappa megtalálható a statisztika által létrehozott mappában.

Load Count

Ez a szám meghatározza, hogy a legelső generáció hány játékosát kívánjuk fájlból betölteni. A fájlok felépítésének és elnevezésének egyeznie kell a mentés által létrehozottakkal, azonban a “Loader” elnevezésű mappán belül kell szerepelniük. Ha kevesebb fájl szerepel valamelyik játékostípushoz mint a Load Count által meghatározott mennyiség, akkor a program véletlenszerűen legenerált neurális hálót használ helyette. Ha ezt a mennyiséget 0-ra állítjuk, akkor a program nem használ betöltött adatokat, és az első generációt teljesen véletlenszerűen hozza létre.

Runners Population Size

A menekülők populációjának mérete, fontos azonban, hogy ez a szám ne legyen nagyobb, mint a betölteni kívánt pálya menekülő kezdő helyeinek száma, különben a program hibát jelez. Negatív szám esetén szintén hibát jelez, azonban 0 értéket felvehet, hiszen egy pályán nem kötelező menekülőknek is tartózkodni.

Catchers Population Size

Működése megegyezik a Runners Population Size-zal, de ez a fogók populációját szabályozza.

Quit After

Ezzel a paraméterrel megadható a program futási ideje fordulókban mérve. Ha a generációs számláló meghaladja ezt a számot, a program automatikusan bezáródik.

Elitist Selection

Ezt az opciót kiválasztva elitista kiválasztást és kombinációt állíthatunk be. Ha a checkboxot üresen hagyjuk, akkor teljesítményarányos kiválasztással és véletlenszerű kombinációval fog futni az evolúciós algoritmus.

Network Structure

Ez a paraméter egy tömböt takar, amelynek először a méretét adhatjuk meg. A méret beállításával szabályozzuk a játékosok által használt neurális háló rétegeinek számát. Ezután megadhatjuk sorban a tömb értékeit, melyek a neurális háló egyes rétegeinek neuron számát szabályozza. Ennek segítségével állítható a neurális háló felépítése. Azonban fontos, hogy a tömb legelső és legutolsó eleme mindig illeszkedjen a játékosok által elvárt beviteli és kiviteli paraméterekhez. Jelen beállítások szerint a beviteli neuronok száma 15, míg a kiviteli neuronok száma 3 kell, hogy legyen. Fontos megjegyezni, hogy a tömbben meghatározott számok nem foglalják magukba a rétegekbe beépülő konstans neuront.

Round Time

Ez a szám meghatározza, hogy a játék egy fordulója hány másodpercig tart, vagyis egy generációnak mekkora a maximális élettartama.

Show Vision

Ezt a checkboxot kipipálva láthatóvá válik a játék során a játékosok látómezeje. Ezáltal pontosabban szemlélteti, hogy a játékosok mit látnak a játék aktuális állásában.

Show Sensor End

Ezt az opciót kiválasztva láthatóvá válik a játék során a játékosok szenzorainak vége, ezzel a legpontosabb adatot szolgáltatva arról, hogy egy játékos éppen milyen másik játékelemeket érzékel szenzoraival. Azonban már kevés játékos esetén is megnehezítheti a játék jelenlegi állásának átláthatóságát.

Time Based Scoring

Ezt az opciót kiválasztva az „Időalapú pontozás” lesz érvényes a menekülő játékosokra.

Should See House To Score

Ezt az opciót kiválasztva a „Ház láthatóság szabálya” lesz érvényben.

Bonus for Teamwork

Ezt az opciót kiválasztva a „Csapatmunka szabálya” teljesül.

Enable Sidestep

Ha ez a checkbox nincs kipipálva, akkor a játékos csak előre vagy hátra tud mozogni a relatív helyzetéhez képest.

II. 5. 2. Pálya paraméterei

Egy pálya paraméterei a FieldManager játékelem megegyező nevű kód komponense alatt állíthatóak. Ahhoz, hogy ezt elérjük, először be kell tölteni az adott pálya jelenetét. A paraméterek az alábbiak: **Starting Runners** és **Starting Catchers**.

Mindkét paraméter egy-egy tömböt takar, amelyeknek először a méretét adhatjuk meg. A pálya létrehozásakor játékos játékelemeket hozunk létre a pályán, a tömbök az ezekhez tartozó PlayerController kód komponenseket tárolják. A játék futása során ez alapján fogja felismerni a FieldManager, hogy milyen helyekre és paraméterekkel kell egy kör kezdetekor a játékosokat elhelyezni. Értelemszerűen a Starting Runners a menekülők, míg a Starting Catchers a fogók adatait tárolja. A FieldManager nem feltétlen helyez minden megadott helyre játékost, ezt a Runners Population Size és a Catchers Population Size paraméterek befolyásolják.

II. 5. 3. Játék általános paramétere

A játék egyetlen általános paramétere a GameStateManager játékelem megegyező nevű kód komponense alatt állítható be:

Field Name

Itt szövegesen megadhatjuk a betölteni kívánt pálya nevét. A pályához tartozó jelenet fájlját a Scenes/Fields mappában keresi a program. A szimuláció végig a kiválasztott pályát fogja használni.

II. 6. Statisztikák és futási adatok

II. 6. 1. UI elemek

A játék futása során a Game ablakban megjelenítünk kiemelt adatokat a játék pillanatnyi állásáról.

Best Runner Score

A játék adott fordulójában a legjobban teljesítő menekülő játékos pontszáma. Csak lepontozott játékosokat vizsgál, így ha be van állítva az időalapú pontozás, akkor lehet olyan menekülő, akinek több pontja lenne az adott pillanatban, de még gyűjti a pontokat, így nem számítjuk.

Best Catcher Score

A játék adott fordulójában a legjobban teljesítő fogó játékos pontszáma.

Remaining Runners

Az aktuálisan még játékban lévő menekülő játékosok száma.

Safe Runners

Azoknak a menekülőknek a létszáma, akik megérkeztek bármelyik házba és pontot is szereztek ezzel, figyelembe véve a Ház láthatósága szabályt, ha az be van állítva.

Remaining Time

A játék adott fordulójából még hátralévő idő másodpercekben.

Generation

Az aktuális generáció hányadik a játék elindítása óta.

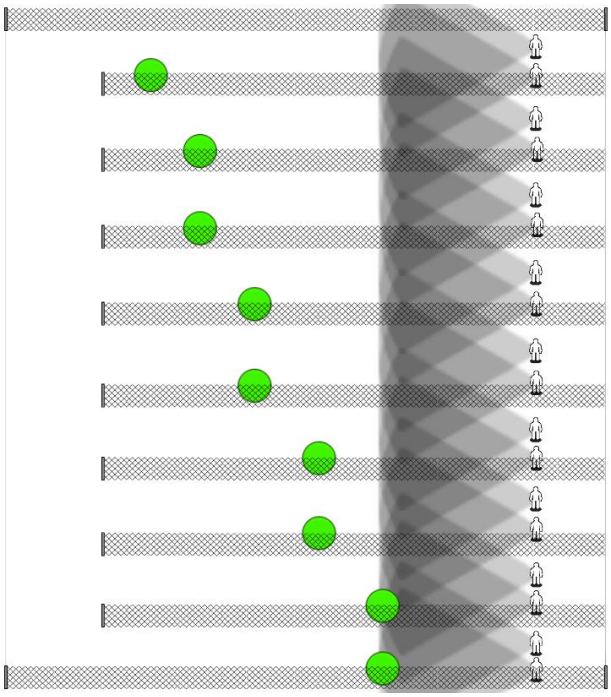
II. 6. 2. Mentési adatok

Ha a Save opciót választjuk az EvolutionManager paraméterei között, akkor a program futása közben több fájlt is létrehoz, amiket a szimuláció adataival tölt fel. Legelsőként a létrehozott ”Simulation_”+{{Pálya neve}}+{{aktuális dátum és idő}}+”.txt” fájlba leírja a két populáció méretét, a pálya nevét, az egyes típusokból előre betölteni kívánt egyedek számát, a fordulóra megszabott időkeretet, a pontozási szabályokat és azt, hogy az elitista opció van-e kiválasztva. Ez magába foglalja a szimuláció legfontosabb beállításait, de ezen felül minden forduló végeztével ebbe a fájlba kerülnek a legjobb játékosok pontszámai, illetve az, hogy az adott fordulóban hány menekülő szerzett házba érkezéssel pontot. Ezekből az összegyűjtött adatokból ránézésre könnyen láthatjuk a legsikeresebb generációkat.

Az első forduló végeztével létrejön egy, a fent említett szöveges fájllal megegyező nevű mappa. Ezután minden generáció kiértékelése végeztével ide létrejön egy új mappa, amiben a generáció legjobb játékosainak adatait mentjük le a beállított Save Count paraméter alapján. Ezekből az adatokból képesek vagyunk a játékos neurális hálóját újraalkotni.

A program tehát képes korábbi játékosok betöltésére is, ám ehhez szükséges a Load Count megfelelő beállítása. A játékosok adait a Loader nevű mappába kell helyezni indítás előtt, és a szövegfájloknak követniük kell a mentési konvenciókat. A játék legelső generációjába tölti be ezeket az adatokat a program, de ha nem talál a Load Count-nek megfelelő számú adatot, akkor véletlenszerű paraméterekkel létrehozott játékosokkal tölti fel a hiányzó helyeket a populációkban.

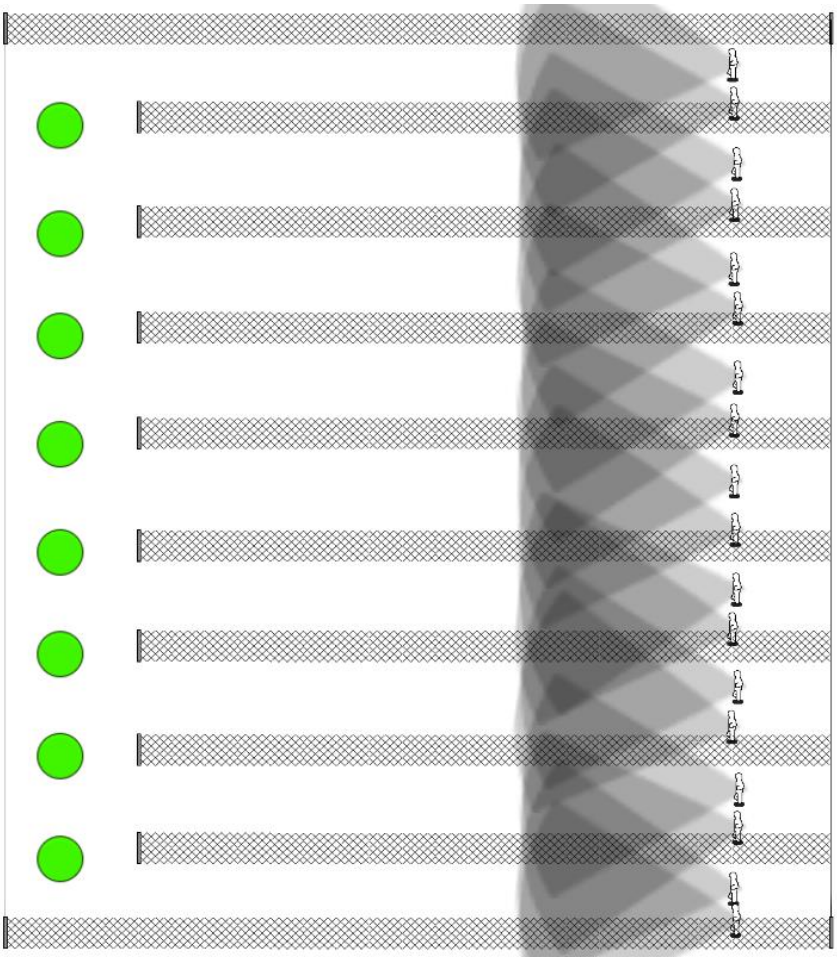
II. 7. Előre létrehozott pályák



9. ábra Field1

II. 7. 1. Field1

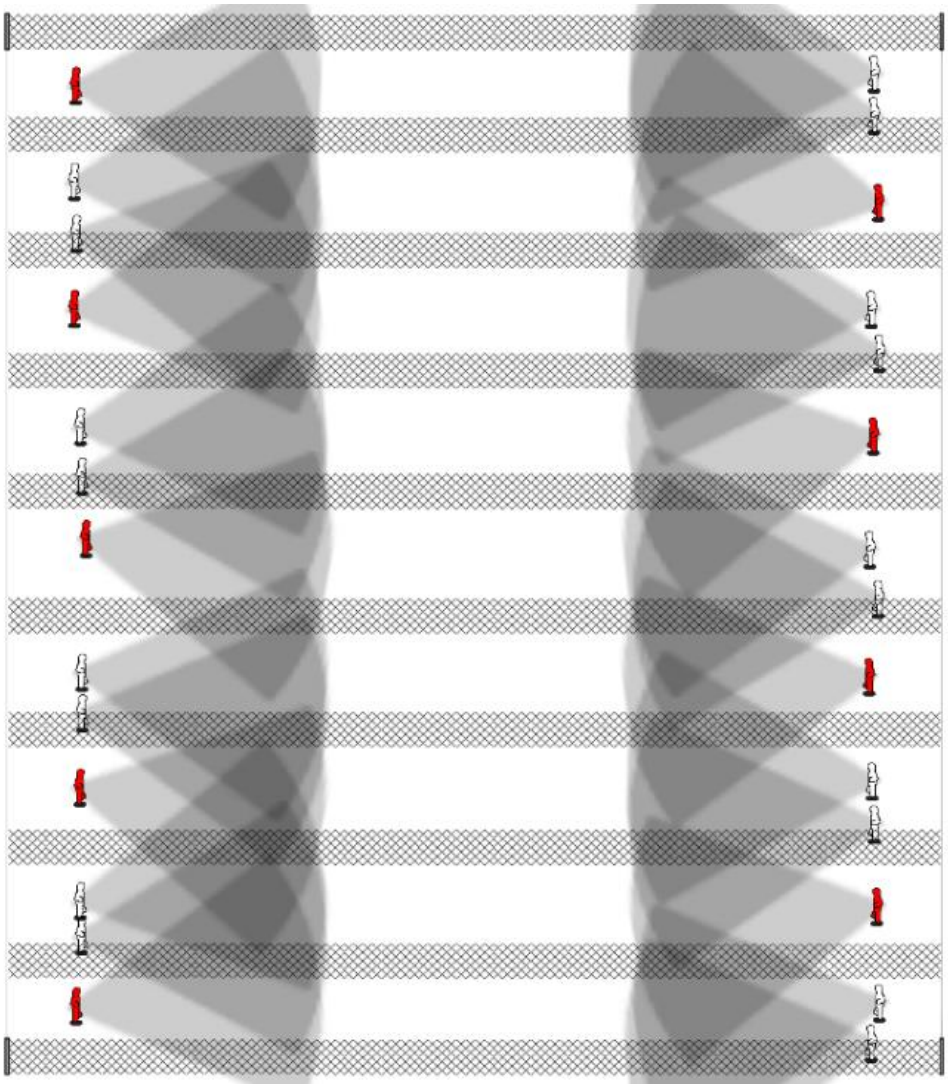
A 9. ábrán látható pálya célja rendkívül egyszerű: a menekülők betanítása a biztonságos házakba való befutásra. Ezért a pályán a maximálisan elhelyezhető 18 menekülő mellett nincs hely fogók számára. A menekülőket rövid folyosókra osztjuk, melyeken tőlük különböző távolságokra házakat helyeztünk el. Mivel a távolságok nem egyenlőek, így az egyes pozíciókban kezdő menekülők által szerezhető maximális pontszám között lényeges eltérés lehet. Ennek ellenére a pálya ezen gyengesége egyben az erőssége is. A legközelebbi házakba már véletlenszerűen is nagy eséllyel beleszaladhatnak a játékosok, és mivel az egyedekhez tartozó kezdőpozíciók minden fordulóban véletlenszerűek, így a tanult tulajdonság könnyebben terjed minden menekülő között. Ezt erősíti az is, hogy a legközelebbi házakat már a kezdőhelyükből is azonnal látják a menekülők.



10. ábra Field2

II. 7. 2. Field2

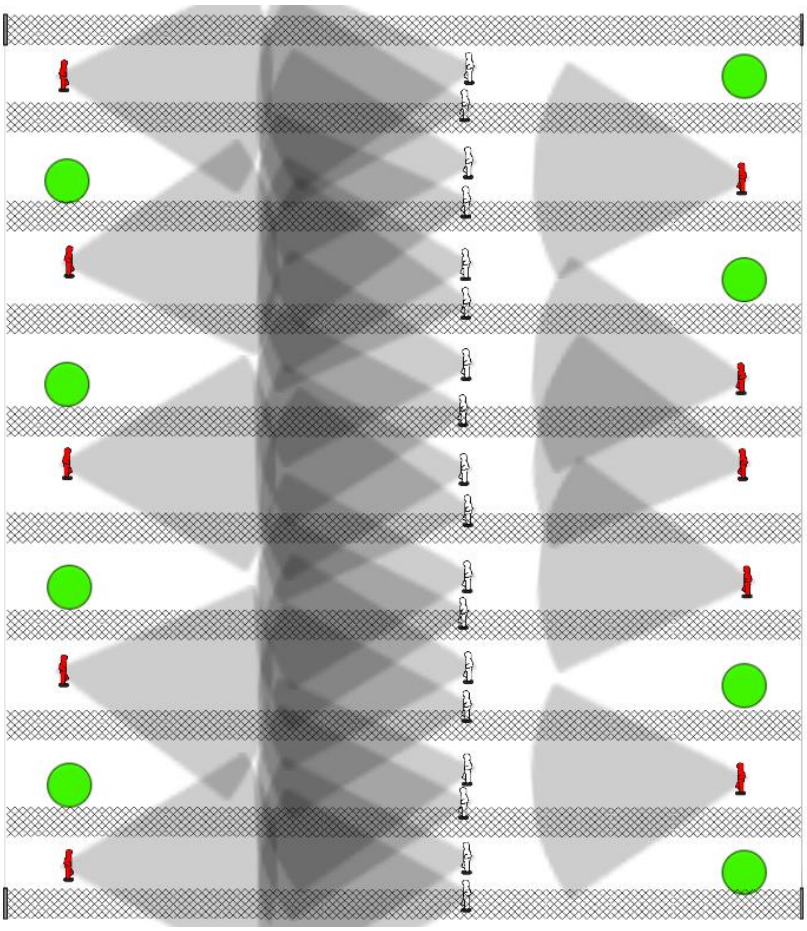
A 10. ábrán látható pálya az előzőhez képest elrendezésében letisztult, céljában pedig rendkívül hasonló. Ez a pálya is 18 menekülőt és 0 fogót képes befogadni. A házakat ezúttal egyenlő távolságra helyeztük el a menekülőktől, hogy kezdőpozíciótól függetlenül közel ugyanakkora legyen a lehetséges maximális pontszámuk. A vékony folyósok és a távoli házak komoly kihívást jelentenek, hiszen a játékosoknak precízen kell manőverezniük, hogy elérjék a biztonságos zónákat. Mivel a házak két folyó között felúton helyezkednek el, így ha csak teljesen egyenes vonalban mozog egy menekülő, az még nem jelenti biztosan azt, hogy el is ér egy házat.



11. ábra Field3

II. 7. 3. Field3

Ez a pálya a fogók alapvető betanítására lett létrehozva. Maximum 9 fogó és 18 menekülő fér el egyszerre rajta. Az előző pályához nagyon hasonlít felépítésében, azonban ezúttal, ahogyan az a 11. ábrán is látható, lezártuk a folyosók végeit, hogy ne legyen köztük átjárás, és a házakat is kivettük, hogy a menekülők ne érhessenek el hozzájuk. Minden fogónak lehetősége van legfeljebb két menekülő elkapására, azonban mivel a kezdő pozíciójából nem látja őket, így először közelebb kell kerülnie hozzájuk. A cél ebben az esetben az, hogy ne érjenek se a falakhoz, se a fogókhoz.

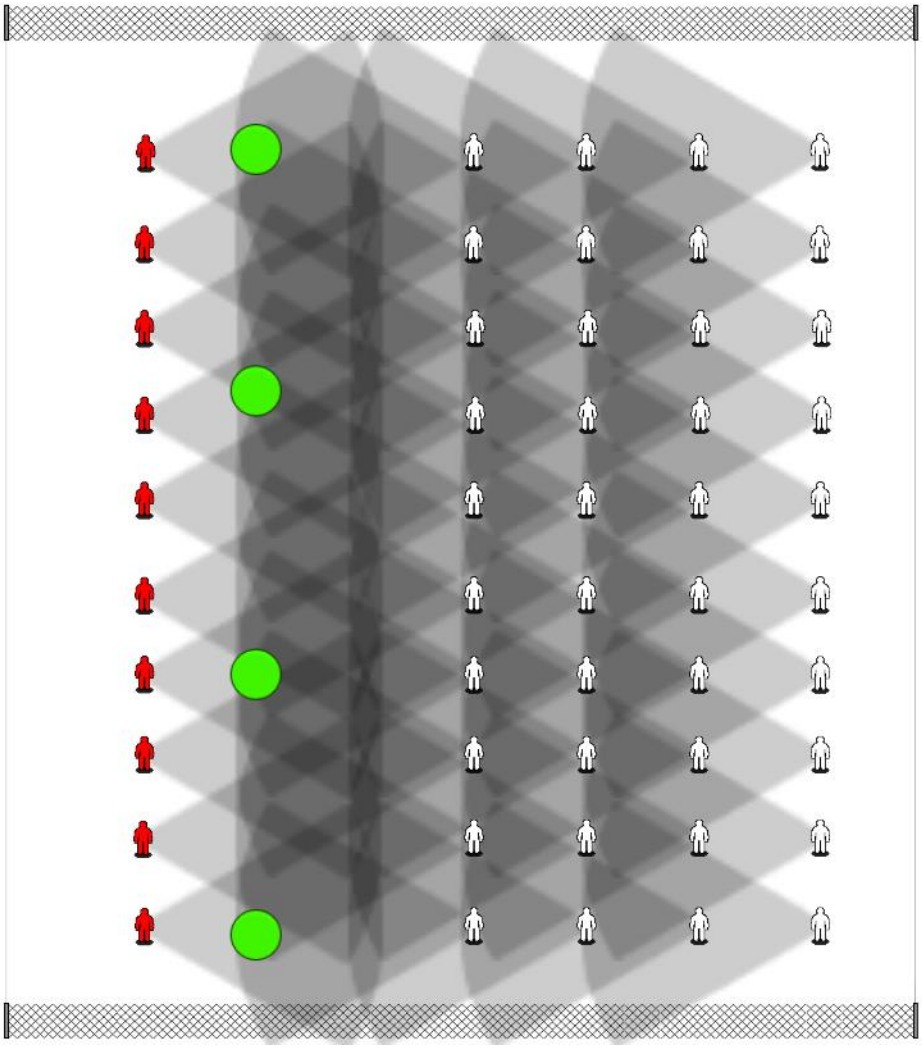


12. ábra Field4

II. 7. 4. Field4

A 12. ábrán látható pálya már egészen komplex. Az előzőhöz hasonlóan most is elválasztjuk a folyosókat, azonban a menekülőket a folyosók közepére helyeztük el. A pálya maximális kapacitása is közel megegyezik az előző pályáéval, de felvettünk 1 extra fogó helyet, ezzel 10 fogóra és 18 menekülőre növelve a számot.

A 9 folyosó elrendezése ezúttal változatos, ez a változatosság fontos célt szolgál. A menekülők minden esetben a folyosó közepére kerültek, 4 esetben a ház előttük, míg 4 esetben a ház mögöttük helyezkedik el a folyosón, míg a fogók pont az ellenkező oldalon. Ezáltal ha a fogók a ház irányába indulnak el, akkor hamar biztonságba juthatnak, ha a fogó irányába, akkor vissza kell fordulniuk mielőtt a fogó elkapná őket. Ez a szimmetria és a véletlenszerű irányválasztás a reakció alapú viselkedés erősítésére lett létrehozva. A középső folyosó nem követi ezt a sémát, itt mindkét irányban fogó helyezkedik el. Ez a fogók elől való kitérés tulajdonságát hivatott erősíteni.



13. ábra Field5

II. 7. 5. Field5

A 13. ábrán látható pálya elrendezésében a legegyszerűbb, ám játékososok számában a legnagyobb. Maximálisan képes 10 fogó és 40 menekülő befogadására. A pálya nem tartalmaz extra, csak határoló falakat, így amíg a játékosok nem mennek túl közel a pálya széléhez, csupán egymásra kell figyelniük. A 4 ház elhelyezése rendkívül tudatos, mert sok menekülő már a kezdőpozíciójából is látja és a fogók épp csak annyival vannak közelebb hozzá, hogy kellő ügyességel akár a forduló kezdésekor rögtön a házak felé rohanó menekülőket is elkaphassák. A menekülőknek tehát feltétlen ki kell játszani a fogókat, ha biztonságban akarnak lenni, azonban magas létszámfölényüknek köszönhetően a fogóknak még így is nehéz minden menekülőt elkapniuk.

II. 8. Hibaüzenetek

A program számos beállítást és paramétert ellenőriz, melyek során az alábbi hibaüzeneteket írhatja ki a Unity Console ablakába:

"More than one EvolutionManager in the Scene."

A jelenet nem egy EvolutionManager kód komponenssel rendelkező játékelemet tartalmaz.

"A Scene should only contain one FieldManager."

A jelenet nem egy FieldManager kód komponenssel rendelkező játékelemet tartalmaz.

"Should be exactly one GameStateManagers in a Scene."

A jelenet nem egy GameStateManager kód komponenssel rendelkező játékelemet tartalmaz.

"Player count does not match population size."

A pályán szereplő játékosok száma nem egyezik meg a populáció méretével valamelyik játékostípus esetén.

"Parameter not matches weight count."

A Brain létrehozásakor a megadott Gene súlyszáma nem egyezik a megadott hálózati struktúra kapcsolatszámával.

"Range should be positive."

A megadott tartomány, amennyivel egy kapcsolat súlya eltérhet a nullától nem pozitív.

"Loadable file has incorrect format."

A betölteni kívánt fájl tartalma nem megfelelő, így nem rekonstruálható belőle játékos.

"Input count not matches connection count!"

Egy Neuron nem a megfelelő számú kapcsolattal rendelkezik, vagy nem megfelelő számú súly értéket kapott az érték kalkulációhoz.

"Population size should be 0 or more than 2."

A megadott Runners Population Size vagy Catchers Population Size nem megfelelő.

III. Fejlesztői dokumentáció

III. 1. A feladat felépítése és lebontása

A feladat kidolgozásának legelején már tisztán elkülönültek a program legfontosabb területei. A szimuláció létrehozásához mindenképpen szükséges a játék megvalósítása, a tanulási folyamat pedig igényel egy mesterséges intelligencia modellt. Ezek még további részfeladatokra bonthatóak a betölteni kívánt szerepek szerint:

Fogó-játék szimuláció

- Játékos megvalósítása
- Pályaelemek megvalósítása
- Játékelemek kezelése

Mesterséges intelligencia

- Neurális hálózat megvalósítása
- Evolúciós algoritmus megvalósítása

Szükséges továbbá egy vezérlő, ami a globális irányításért felelős, és összehangolja a két nagy egység munkáját.

III. 2. Unity osztályok

A Unity rendelkezik egy beépített alaposztállyal, aminek a neve MonoBehaviour [5]. A legtöbb C# osztályt ebből származtatjuk, mivel lehetőséget kínál speciális funkciók használatára. Ezek közül a programban is szerepelnek:

- Start() [6]

  Ez a függvény abban a képkocka-frissítésben hívódik, amikor a komponens aktívvá válik. Megelőzi bármelyik Update() függvényhívást.
- Awake() [7]

  A Start() függvényhez nagyban hasonlít, de ez mindig megelőzi a Start() hívását, valamint vele ellentétben az Awake() akkor is meghívódik, ha a kód komponens inaktív.
- Update() [8]

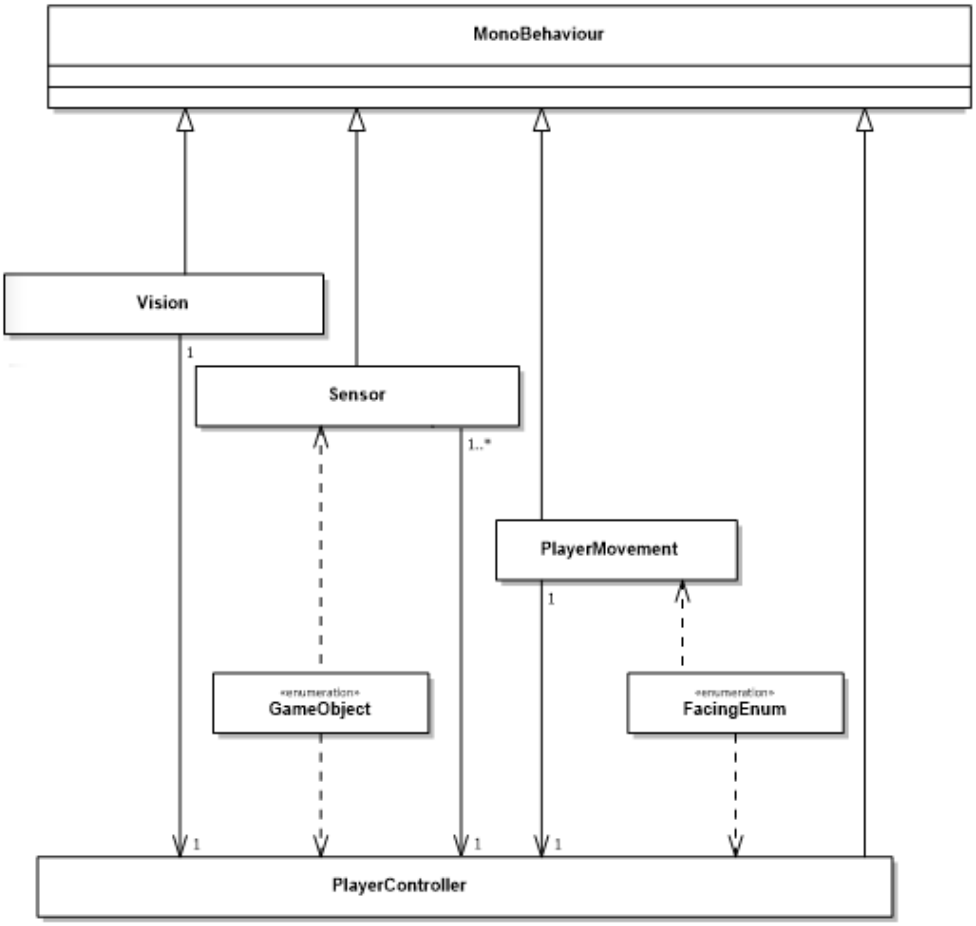
  Ez a függvény minden képkocka-frissítés alkalmával meghívódik. A programban többnyire a mozgási- és játékadatok frissítését hívja meg.

- FixedUpdate() [9]

  Képkocka-frissítéstől függetlenül egyenletes, állítható időközönként meghívódik, amit általában fizikai kalkulációkhoz használunk.

- OnCollisionEnter2D(Collision2D collidingObject) [10]

  A megfelelő működéshez szükséges, hogy a játékelem a kódkomponens mellett egy collider komponenssel is rendelkezzen, úgy mint CircleCollider2D vagy BoxCollider2D. A függvény akkor hívódik meg, amikor a játékelem saját collider-e érintkezik egy másik játékelem collider-ével és paraméterként kapja az ütközés adatait.



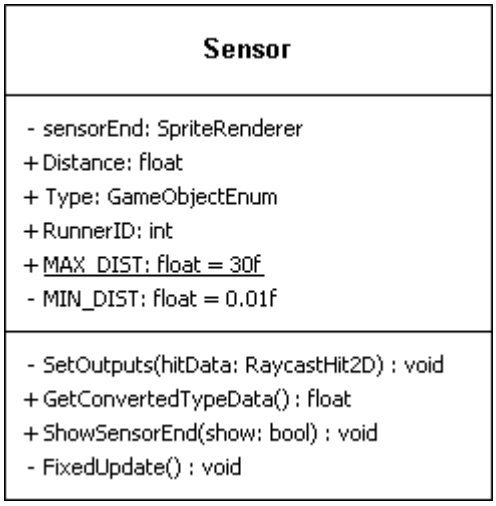
14. ábra A játékos osztályainak viszonya

III. 3. Játékos felépítése

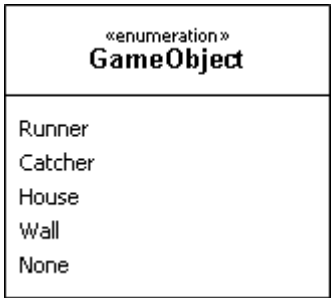
A játékos felépítése során az volt a cél, hogy egy jól kezelhető, de egységes modellt lehessen létrehozni a szimulációban mindkét játékostípushoz. Szerencsére a fogók és a menekülők megegyezhetnek mozgásukban és abban, hogy hogyan érzékelik

környezetüket, így elég a különbségeket összefoglaló logikát egy, a kontrollért felelős osztályba szervezni.

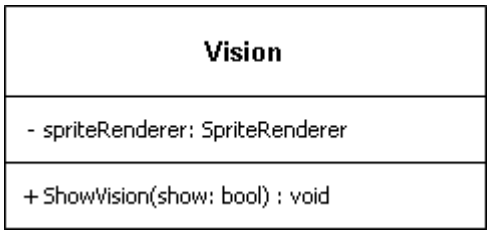
Ez alapján épültek fel a 14. ábrán látható osztályok és a köztük lévő viszonyok.



15. ábra Sensor osztály



16. ábra GameObject enumerátor



17. ábra Vision osztály

III. 3. 1. A játékos szemei

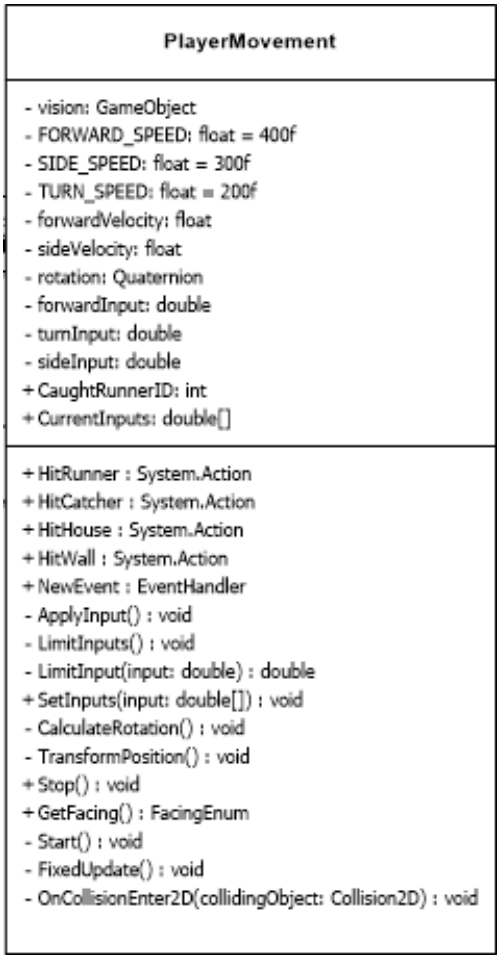
A játékosok környezetének érzékelésért felelős a 15. ábrán látható Sensor osztály. Az osztály feladata, hogy egy meghatározott irányban a beállított maximum és minimum távolság között érzékelje bármelyik másik játékelemet, ami rendelkezik egy collider komponenssel, amik csakis a játékosok és a pálya elemei.

Ezt a Unity-be beépített raycast technológiával viszi véghez, ami egy sugarat bocsájt ki egy megadott vektor mentén, és visszatér az első játékelemhez tartozó ütközési adatokkal. Az adatok alapján a Sensor (SetOutputs()) feladata meghatározni, hogy a játékelem mekkora távolságra van és milyen típusú. Mivel szükséges, hogy a játékosok meg tudják különböztetni, hogy éppen egy falat, házat, egy fogót vagy egy menekülőt látnak, így bevezettük a 16. ábrán látható GameObject enumerátort, amit ezután általánosan használunk a különböző típusú játékelemek meghatározására. A játék különböző elemei mellé felvettünk egy None értéket is az enumerátorba, hogy könnyedén kezeljük, ha egy szenzor épp nem érzékel semmilyen játékelemet.

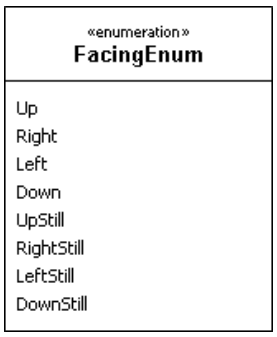
Minden szenzor végén szerepel egy vizuális indikátor, aminek távolságát a játékostól dinamikusan változtatunk attól függően, milyen távolságra érzékelünk egy játékelemet. Ez rendkívül pontos képet ad egy játékos pillanatnyi érzékeléséről, azonban mivel minden játékos rendelkezik 5 szenzor játékelemmel, így már kevés játékos esetén is

nehezen kivehetővé teszi a játék folyamatát. Emiatt a szenzor végének láthatósága állítható az EvolutionManager paraméterei között.

Létrehoztunk továbbá az átláthatóság kedvéért egy Vision osztályt, ami a 17. ábrán látható, és egy hozzá tartozó vision játékelemet, aminek megjelenítését a Vision osztály ShowVision(bool show) metódusával szabályozhatjuk. Minden játékos rendelkezik pontosan egy vision játékelemmel. Ennek mérete a szenzorok maximális érzékelési hosszához van kalkulálva, és nem változik dinamikusan. Így bár nem ad pontos képet arról, hogy éppen mit érzékel egy játékos, de arról igen, mi van érzékelési távolságon belül. A láthatóság állítása azért lett egy lehetőség, mert a szenzorokhoz hasonlóan sok játékos esetén megnehezítik az átláthatóságot.



18. ábra PlayerMovement osztály



19. ábra FacingEnum enumerátor

III. 3. 2. A játékos lábai

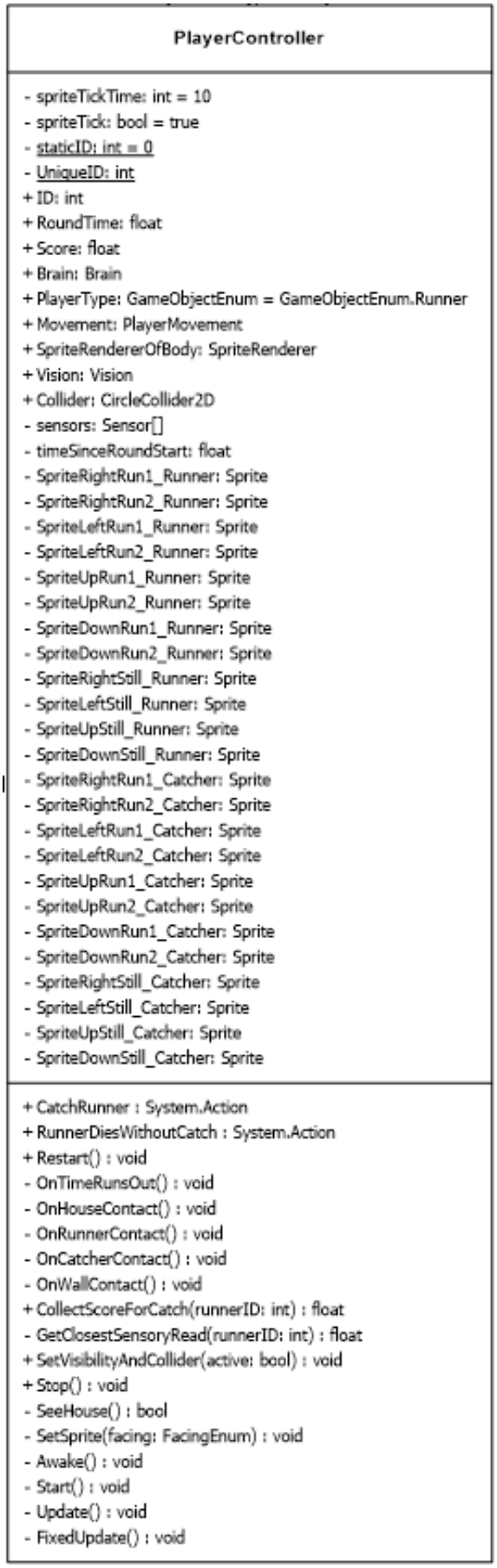
A játékos mozgásáért a 18. ábrán látható PlayerMovement osztály felel. A játékos képes bármilyen irányba mozdulni és a fordulni is, ezeket csupán 3 szám szabályozza, melyet az osztály a PlayerController osztálytól kap meg. Ez a 3 szám mindegyike -1 és 1 közé esik (LimitInputs()), és azt határozzák meg, hogy az előre megadott maximális előre-hátra

mozgás, jobbra-balra mozgás és jobbra-balra fordulás sebességeinek hány százalékával mozduljon el a játékos. A 0 és 1 közötti szám egy adott irányt jelöl, míg egy -1 és 0 közötti a vele ellentétes irányt.

Mivel a játékos tényleges elmozdulása az előre-hátra illetve jobbra-balra lévő mozgás által meghatározott vektorok összessége, így egy játékos képes helyét bármilyen irányban megváltoztatni a kétdimenziós térben.

A mozgás komplexitását növeli a forgás lehetősége, mivel ez változtatja, hogy a játékos mit tekint a különböző irányoknak. A fordulat mindig bal vagy jobb irányba történik, és a játékos csak ezután mozdul el az új helyzetéhez relatívan (CalculateRotation()). Valójában a játékos játékeleme nem fordul ilyenkor, csupán az alá tartozó, látótérhez és érzékeléshez tartozó játékelemek. Ezáltal a játékos képe nem fordul el, csak a látótere és a szenzorai. Mivel a játékosokat animáljuk is a játék során, így fontos meghatározni milyen irányban néznek a globális irányokhoz képest. Ehhez a PlayMovement osztály a játékos játékelemének irányait veti össze a látómező irányával, és a kettejük bezárt szöge alapján visszaadja a 19. ábrán látható FacingEnum valamely értékét.

Abban az esetben, ha az EvolutionManager paraméterei között nem állítjuk be a Sidestep lehetőségét, akkor a játékos oldalirányú mozgását mindig nulla input értékhez kalkuláljuk, vagyis nullvekor lesz. Tehát ebben az esetben a játékos csak a relatív helyzetéhez képest előre vagy hátra tud haladni. A fordulási képesség változatlan.



20. ábra PlayerController osztály

III. 3. 3. A játékos teste

Ha a szenzorok a játékos szemei és a PlayerMovement a lábai, akkor a 20. ábrán látható PlayerController osztályt jogosan hívhatjuk a játékos testének. Feladata, hogy összekösse a játékos minden részét, és továbbítsa a szükséges adatokat közöttük. Tagváltozóként tartalmazza a szenzorok listáját és a PlayerMovement egy pédányát. Kapcsolatban áll a játékos agyának számító neurális hálózatot megvalósító osztállyal is.

A PlayerController tartalmaz minden játékoshoz kötött logikát. A szemei által generált adatokat képes továbbítani az agy felé, majd az onnan érkezett válasz alapján továbbítani a mozgásra vonatkozó paramétereket a lábhoz. Az érzékelt ütközések mellett figyelemmel tartja a játék adott fordulójából hátralévő időt, és ezek alapján pontozza önmagát (OnTimeRunsOut()).

Az osztály létrehozásakor alkotott egyedi azonosító szerepe, hogy a játék során akár beazonosíthassuk a konkrét játékost akivel egy játékos ütközik, vagy akit egy szenzor érzékel. Ez szükséges, hogy ki tudjuk osztani az extra pontokat, ha a csapatmunka szabálya van érvényben, bár a bónuszpontok kiosztásáért már a FieldManager lesz felelős (DealCatchScores()).

A PlayerControll sajátos feladata a játékoshoz tartozó minden megjelenés kezelése, ezáltal az animációé is. Az animációhoz tartozó képek külön adattagokként vannak felvéve, melyek között ciklikusan vált adott időközönként, figyelembe véve melyik irányba néz az adott pillanatban, és végez-e mozgást (SetSprite()).

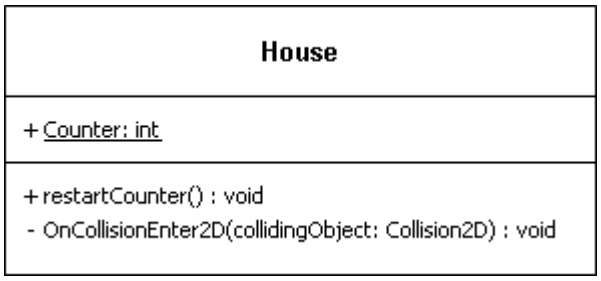
III. 4. Pályakezelő felépítése

Már elkészítettük a pályát benépesítő játékosokat, azonban a statikus elemekre és egy, a pályát általánosan vezérlő osztályra még szükség van. A falak természetükből adódóan nem igényelnek saját kódot, a velük való ütközést a PlayerController teljesen kezeli. Szükséges azonban egy House osztály bevezetése, aminek felépítése a 21. ábrán látható, és amivel minden ház játékelem rendelkezik és globálisan számolja a házakba beérkezett játékosok számát. Ez az adat a játék során kiírt statisztikákhoz szükséges.

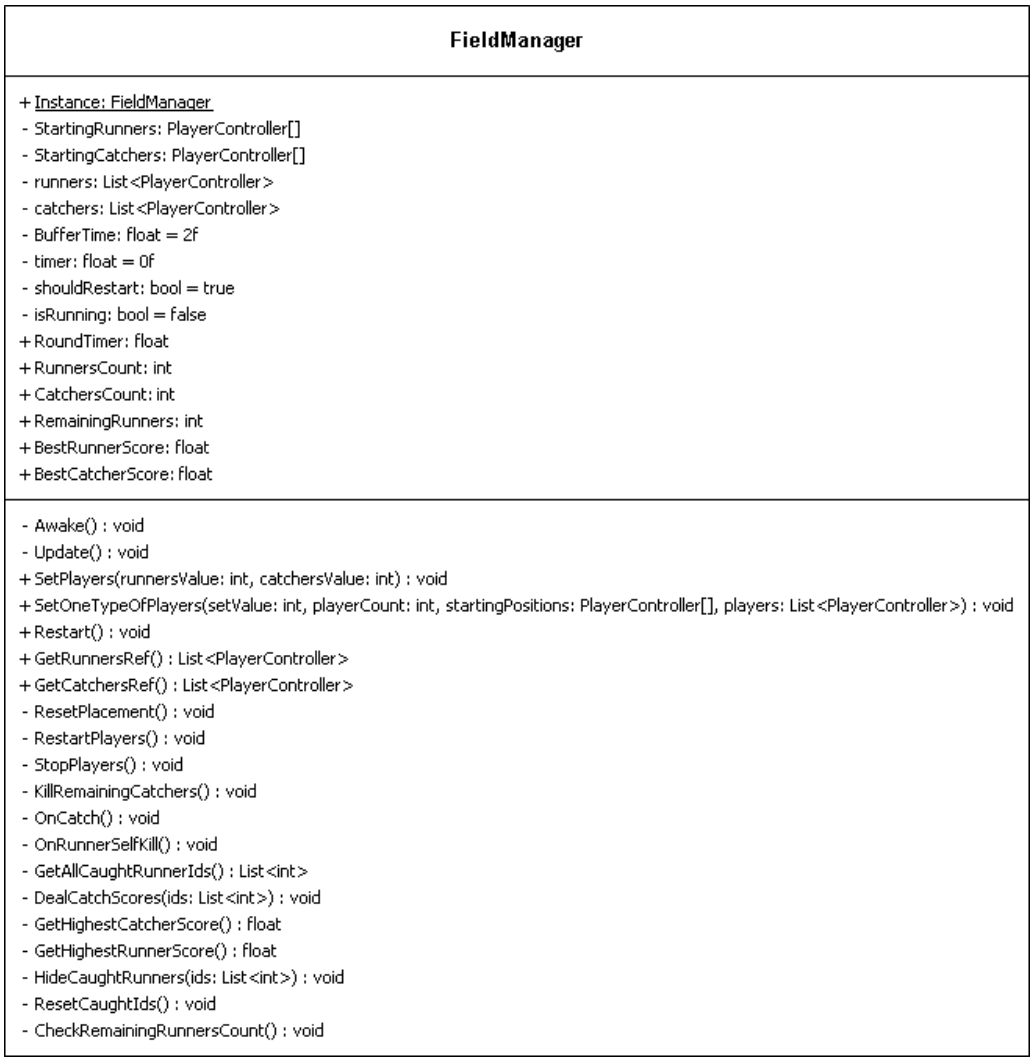
A pálya teljes szabályozásához a 22. ábrán látható FieldManager osztály felelős. A játék kezdetén létrehozza a játékosokat (SetPlayers()), a fordulók elején a megfelelő pozícióba mozgatja őket (ResetPlacement()), és végig figyeli őket. Felelős továbbá olyan műveletek végrehajtásáért, amiért a játékosok összehangolása szükséges, például a csapatmunkáért való bónuszpontok kiosztása, vagy a legmagasabb pontszámok kigyűjtése és továbbítása a UI felé. A pályához tartozó visszaszámláló óra is itt valósul meg, de rendelkezik egy

másik időzítővel is, ami azért felelős, hogy minden forduló kezdetén várjon egy rövid időtartamot a játékosok elindításával az átláthatóság kedvéért.

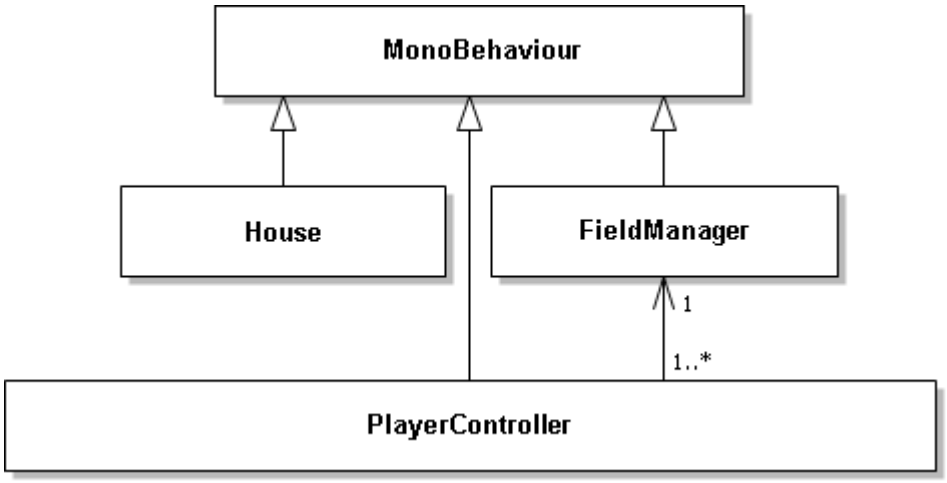
Az új osztályok egymással és a már meglévő PlayerControllerrel való viszonyát a 23. ábra mutatja.



21. ábra House



22. ábra FieldManager



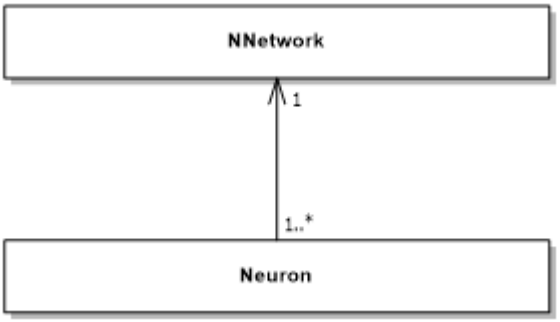
23. ábra A pályakezelő osztályainak viszonya

III. 5. Neurális hálózat felépítése

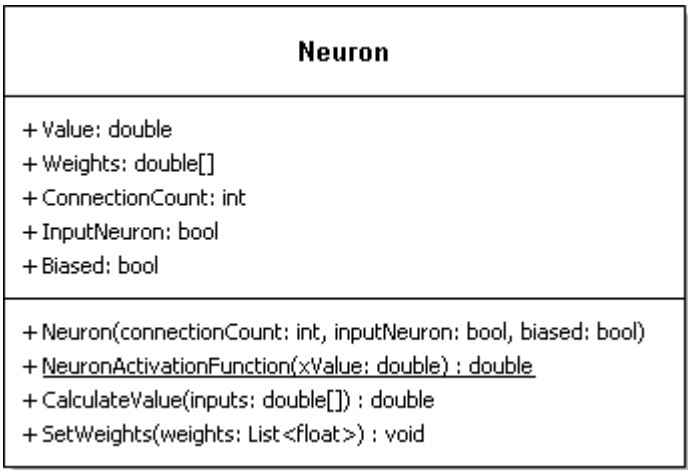
A neurális hálózat megvalósítását két részre bontottuk. Az NNetwork osztály a hálózat felépítéséért és általános tulajdonságainak kezeléséért felelős, míg a Neuron osztály a kisebb építőelem, amit az NNetwork felhasznál. Viszonyuk a 24. ábrán látható.

A 25. ábrán szereplő Neuron osztály nevével ellentétben többet foglal magába, mint csupán egyetlen neuront. Azért, hogy ki tudjuk számolni a neuron értékét az osztályon belül, muszáj volt idevennünk a neuronhoz tartozó kapcsolatokat. Egy neuron létrehozásakor megadjuk azt is, hogy ha beviteli neuronról vagy ha biased neuronról van szó. A beviteli neuron nem rendelkezik kapcsolatokkal, így értéke mindig az lesz, amire közvetlenül állítjuk. Ezeket az értékeket gyűjti össze a PlayerController a szenzorok adatai alapján. A biased neuron esetén sem kell az érték kiszámolásával bajlódnunk, mivel azok értéke mindig 1 lesz. Minden más esetben a neuron értékét a kapcsolatok adta súlyok és egy, az előző réteg értékeit tartalmazó paraméter segítségével számoljuk ki (CalculateValue(double[] inputs)). A kalkulált értéket az aktiválási függvény alapján véglegesítjük (NeuronActivationFunction(double xValue)).

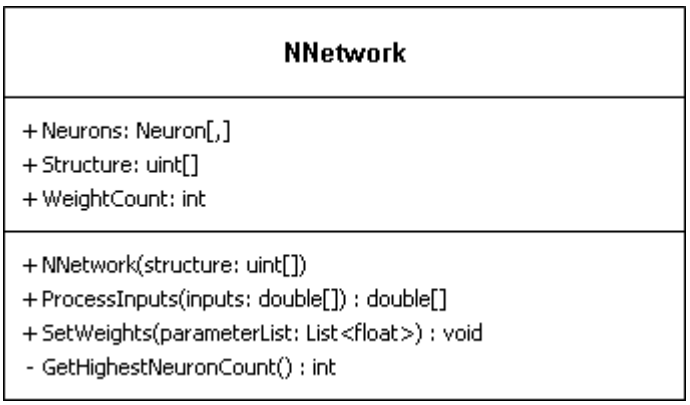
Az 26.ábrán látható NNetwork az EvolutionManagerben megadott struktúra alapján létrehozza a neuronokat és azokat rétegekbe foglalja, ezáltal kialakítva a hálózatot. Az osztály továbbá képes a kapcsolatok súlyait beállítani minden neuron esetén, illetve azok értékeit is kiszámolni (ProcessInputs(double[] inputs)). Mivel előrecsatolt neurális hálózatot használunk, így az adatok mindig csak a beviteli rétegtől a kiviteli réteg felé terjednek. Elég tehát a neuronok kiértékelését is rétegenként ugyanebben a sorrendben végrehajtani.



24. ábra Neurális osztályok viszonya



25. ábra Neuron



26. ábra NNetwork

III. 6. Evolúciós folyamatok felépítése

Az evolúciós algoritmus megvalósításához először is szükségünk van egy osztályra, ami a populáció tagjait fogja reprezentálni. Az NNetwork erre majdnem alkalmas, hiszen ami változik az öröklődés során, az a neurális hálózat, de mivel annak felépítése állandó, így elég csupán a kapcsolatok súlyait használnunk. A 27. ábrán látható Gene osztály fő

feladata ezen súlyok kezelése, de rendelkezik Evaluation és Fitness paraméterekkel is, amik az egyedek kiértékeléséhez szükségesek.

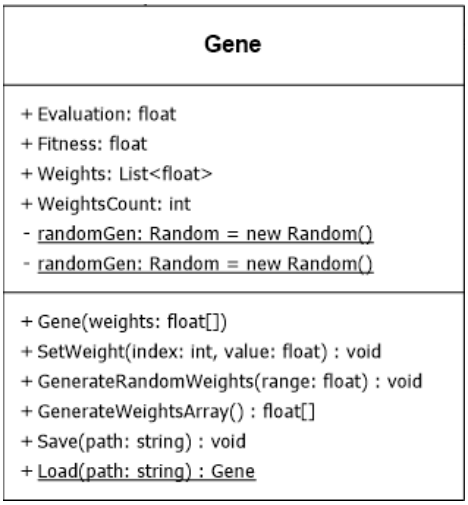
A Gene által reprezentált egyedeket a 28. ábrán szereplő GeneticAlg osztály foglalja evolúciós folyamatba. Ez az osztály egy általános evolúciós folyamatot valósít meg, végighaladva az ismert kiértékelés, kiválasztás, keresztezés és mutáció műveleteken. Képes elitista és teljesítmény arányos kiválasztásra, illetve keresztezésre is. Egy evolúciós ciklus végeztével az egyedeket véletlenszerűen sorba rendezi, hogy a pályán az egyes kezdő pozíciókba osztott játékosok ne feltétlenül mindig ugyanazt a Gene-t kapják.

A 30. ábrán lévő EvolutionManager osztály feladata, hogy a megadott paraméterekkel létrehozza az evolúciós folyamatot és a játék szimulációt (StartEvolution()), és ezek működését összehangolja. Kezeli a szimulációs adatok mentését is egy generáció kiértékelésének végeztével. Működéséhez azonban még egy osztályt igényel.

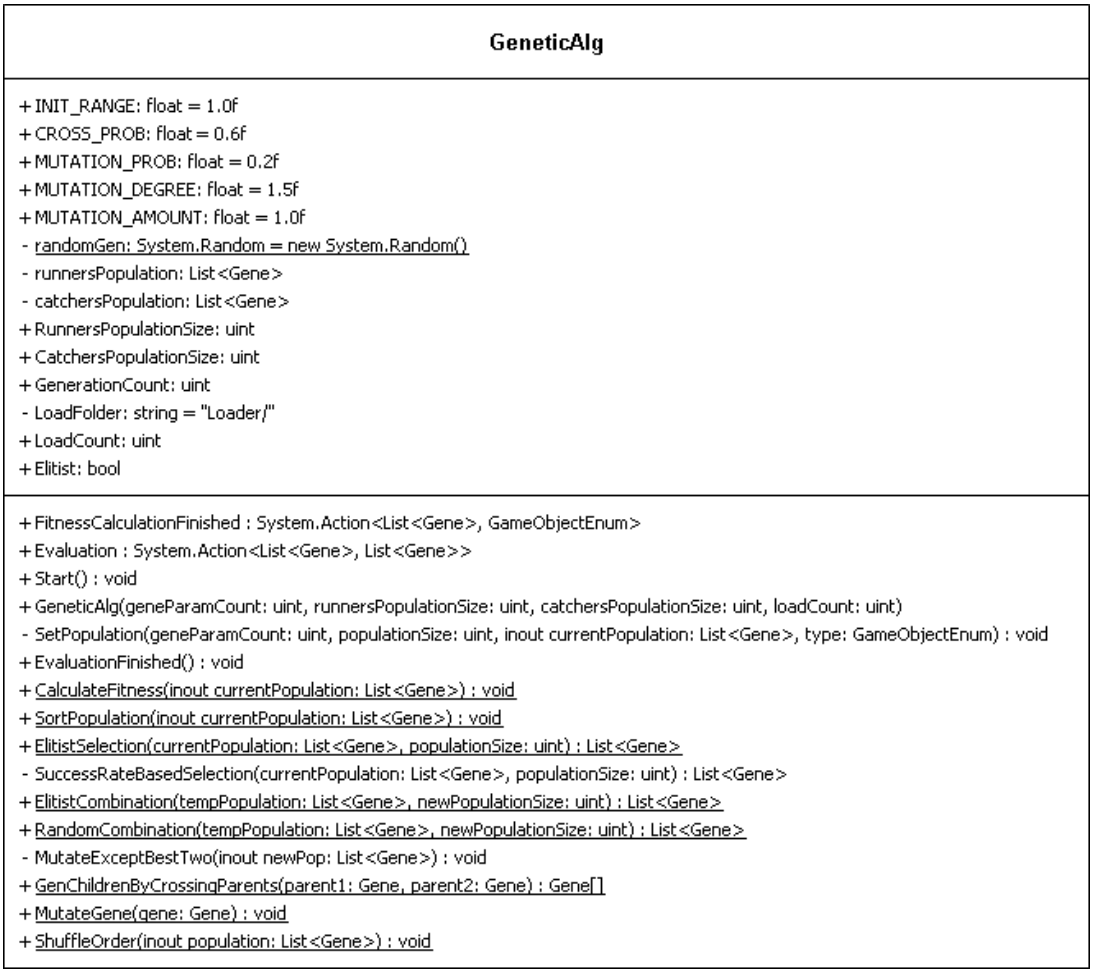
A 29. ábra Brain osztályának feladata, hogy a Gene adattagja alapján felépítsen egy neurális hálózatot és, hogy megvalósítsa a kommunikációt a PlayerController és a kialakult hálózat között. Ahogyan a neve is utal rá, valóban a játékos agyáról van szó, nem csupán hálózat szempontjából, de tartalmazza azt is, milyen specifikus szabályokat állítottunk be az EvolutionManager segítségével, mint például az időalapú pontozás.

Az EvolutionManager felel tehát a FieldManager és a GeneticAlg megfelelő ciklikus futásáért, aminek szabályozásához szükséges adatoka a szimulációban résztvevő játékosok agyából nyeri ki (OnBrainDied()).

Ezen osztályok kapcsolata a 31. ábrán látható.

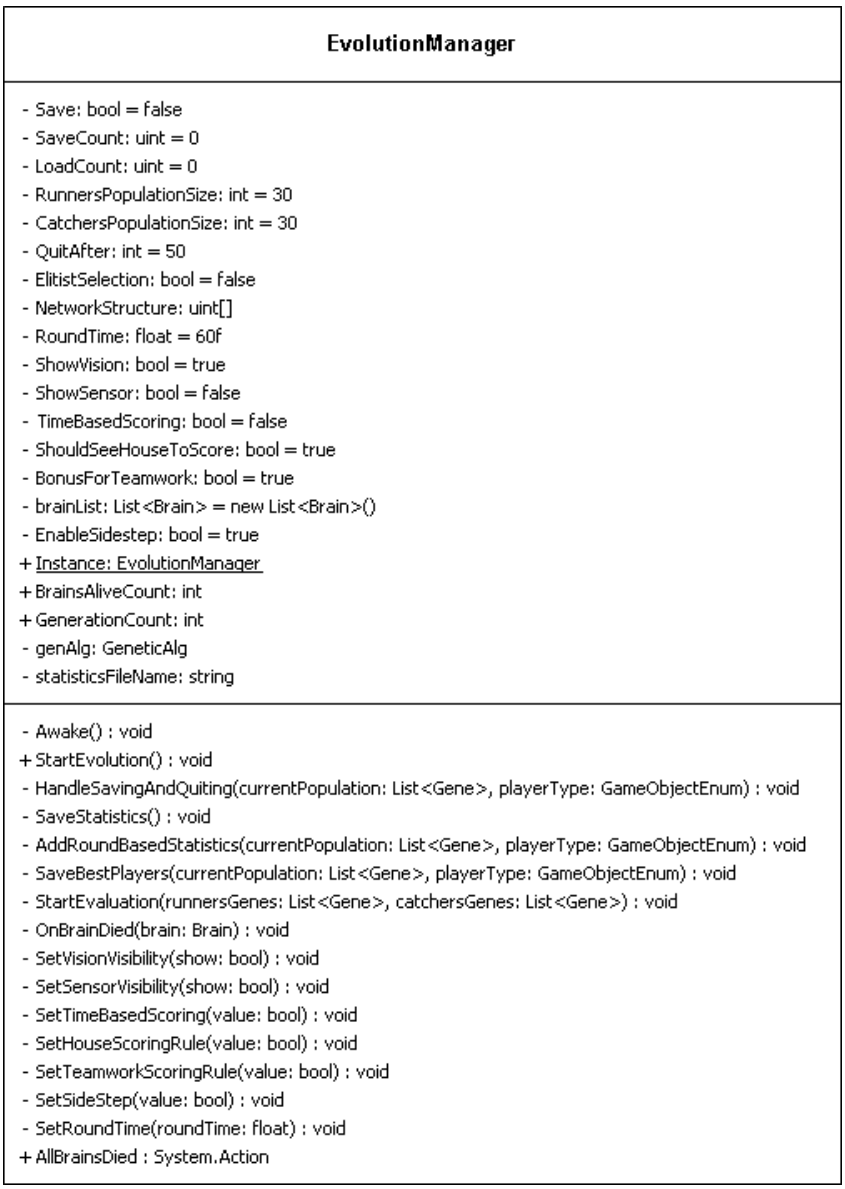


27. ábra Gene

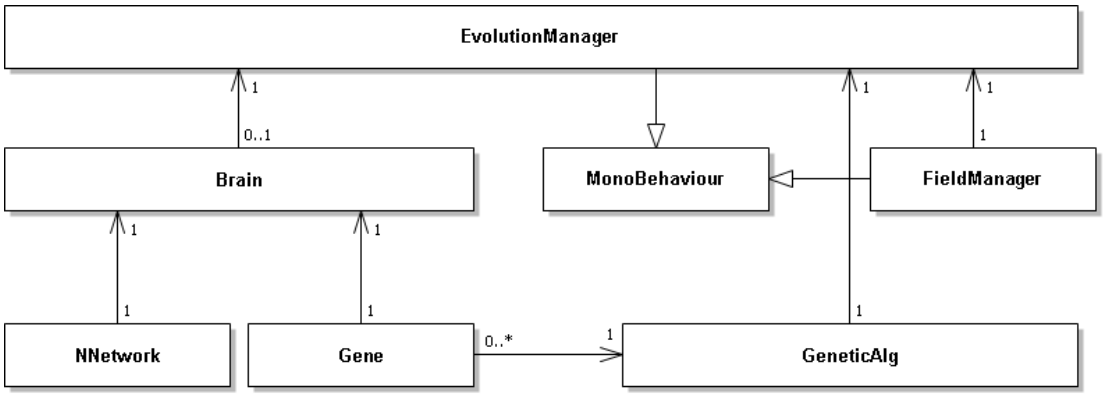


28. ábra GeneticAlg

29. ábra Brain



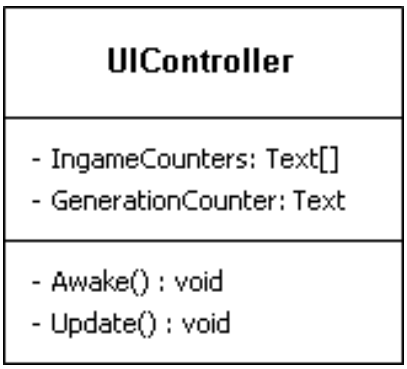
30. ábra EvolutionManager



31. ábra Az evolúciós osztályok viszonya

III. 7. Felhasználói felület

A játék során fontos adatokat jelenítünk meg a képernyőn a felhasználó számára. Ezen adatok összegyűjtéséért és frissítéséért felelős a 32. ábrán látható UIController osztály. Jól behatárolható feladata tömör felépítést tesz lehetővé. A megjelenített adatokat két csoportba osztja: a játék egy fordulója közben változó statisztikák, illetve a forduló végén változó generációs számláló. A legfrissebb adatok megjelenítésében a Unity Update osztálya segít. A megjelenített adatokat a FieldManager, House és EvolutionManager osztályoktól kapja.

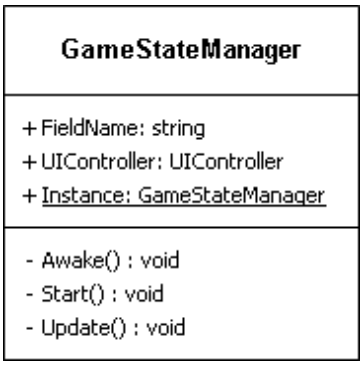


32. ábra UIController

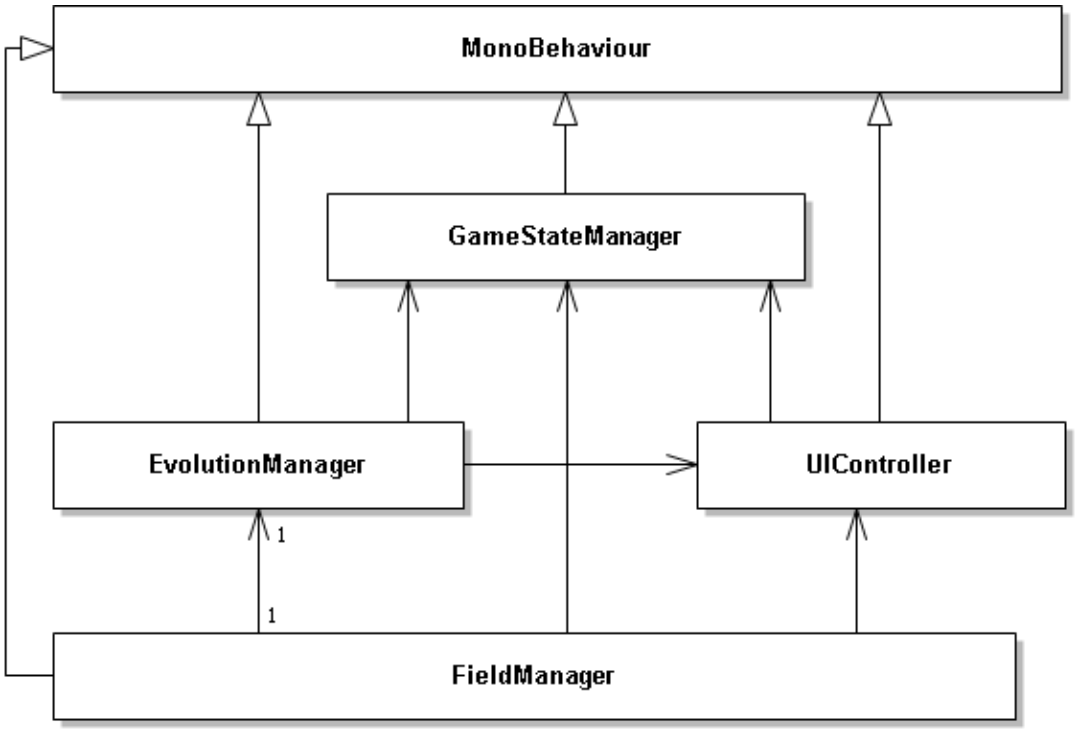
III. 8. Játékkezelő felépítése

Ez a legmagasabb szintű irányító egység, azonban meglepően kevés feladat jut számára. A korábban felépített játékszimuláció és mesterséges intelligencia blokkok önállóan már teljesen működnek, így a 33. ábrán látható GameStateManager osztály köti ezeket össze. Indításkor betölti a megadott pályát és a UI-t tartalamzó jeleneteket, majd kiadja az EvolutionManager-nek, hogy kezdje el a szimulációs folyamatot.

A 34. ábrán látható a GameStateManager és UIController osztályok viszonya egymáshoz, és a már korábban bevezetett osztályokhoz képest.



33. ábra GameStateManager



34. ábra Játékkezelő és UI osztályok viszonya

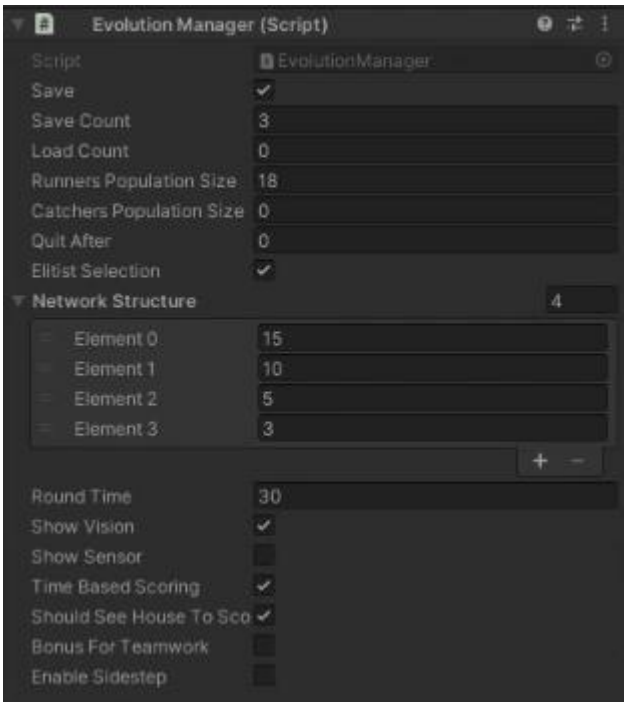
III. 10. A program tesztelése

A program tesztelését sok, a Unity által biztosított eszköz segíti. Az egyes játékelemek segítségével beállított paraméterek helyességét a program futás közben ellenőrzi és hiba esetén hibaüzenetet ír ki a Unity Console ablakára. Egy ilyen hibaüzenet lehet például a „Population size should be 0 or more than 2.”, ami arra utal, hogy vagy a Runners Population Size, vagy a Catchers Population Size hibásan lett megadva. Hasonlóan ellenőrizzük a jelenetek helyes felépítését és a mentés illetve betöltés folyamatokat is. A Unity segítségével futási időben láthatjuk, hogy pontosan mikor és a kód melyik pontjáról érkezett a hibaüzenet.

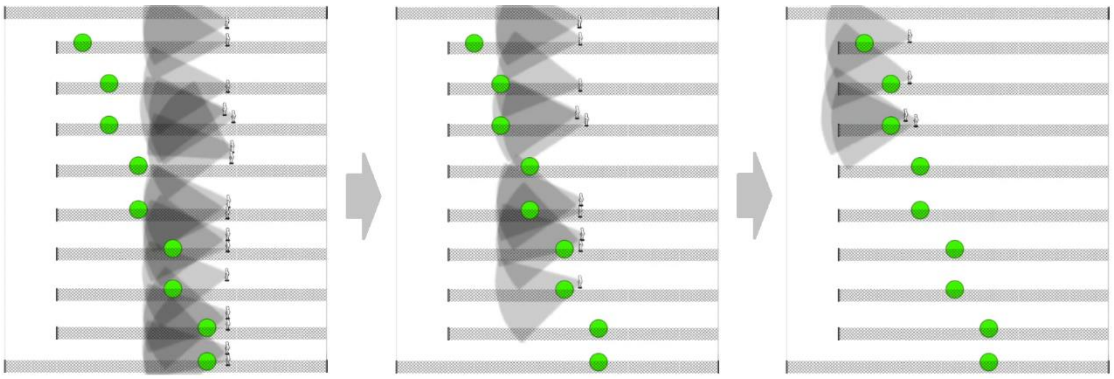
Ez azonban még nem elég a játék szimuláció teszteléséhez. Ebben segít a Unity Pause és Step gombjai. Előbbi segítségével bármikor megállíthatjuk a játék folyamatát, és a Scene ablak segítségével megváltoztathatjuk a játékelemek elhelyezkedését és adatait. Ezáltal könnyű előidézni olyan játékeseteket, mint például amikor egy játékos falhoz ér. A Step gomb segítségével pedig képkocka-frissítésenként léptethetjük a programot, ami lehetőséget ad komplex helyzetek pontos vizsgálatához is, például ha több menekülő játékost egyszerre kapnak el a fogók.

Tehát könnyedén ellenőrzihetjük a program helyes működését, ám a tanulás folyamata a paraméterek precíz hangolásán múlik, amit sokkal nehezebb automatikusan ellenőrizni. A különböző pályák kialakítása pontosan ezért egy-egy specifikus tulajdonság megtanulására próbál hatással lenni.

III. 11. Amit tanultunk



35. ábra Field1 szimuláció beállításai

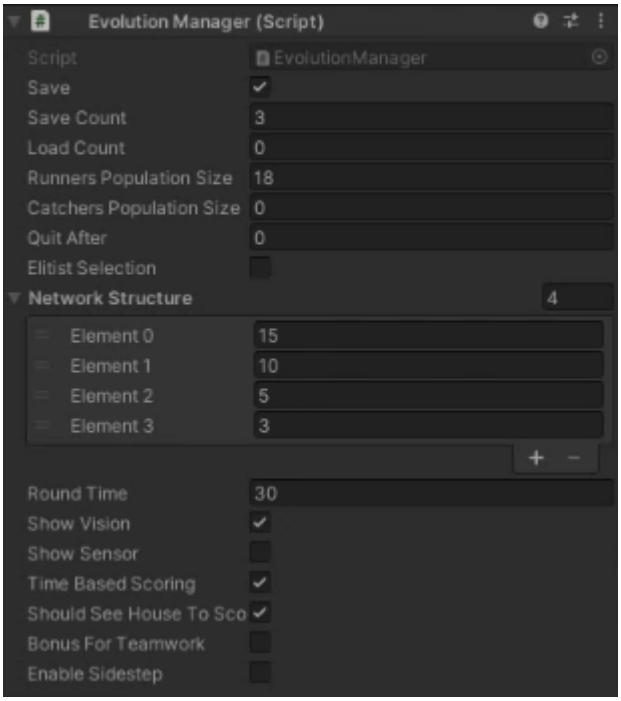


36. ábra Játékosok a 26. generációban

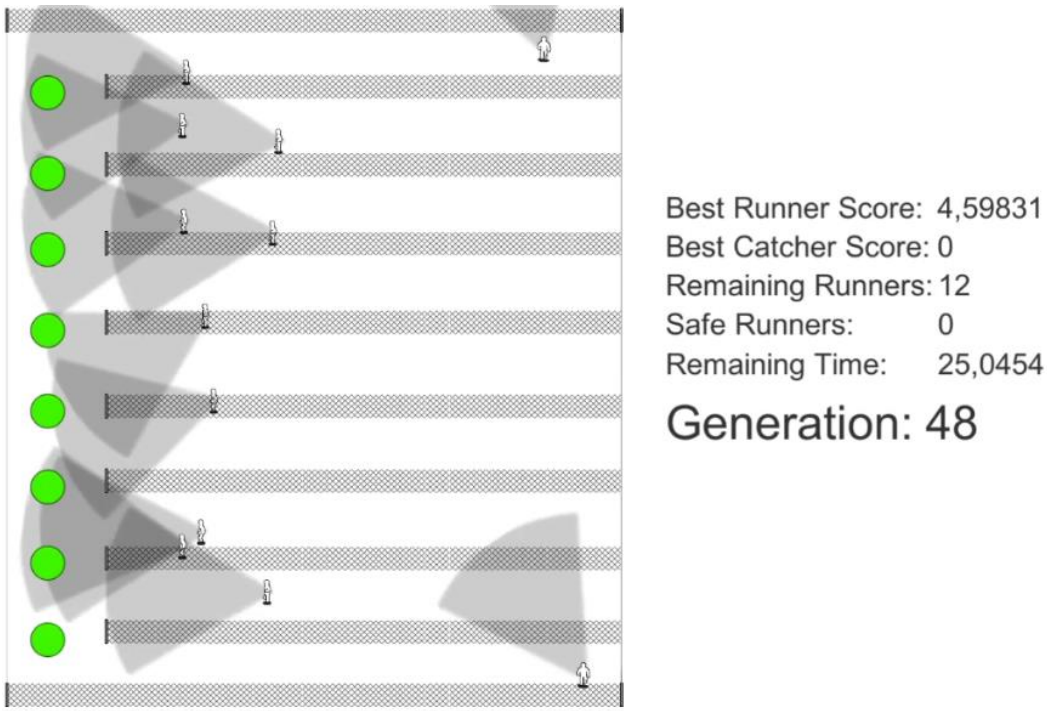
III. 11. 1. Field1

A Field1 pályán a 35. ábrán látható paramétereket használtuk. A pálya elrendezésének köszönhetően már a legelső generációban volt egy játékos, amelyik véletlenül belement egy házba. Az elitista kiválasztás miatt ezután már minden generációban volt játékos, amelyik elért egy biztonságos zónát. A mutáció miatt a házakba érkező játékosok száma

nem biztos, hogy nő generációról generációra, azonban a populáció sikerességének átlaga növekszik. A 26. generációra már 13 játékos elérte a házakat, ami kiugróan magas, ha figyelembe vesszük a keskeny folyosókat, amiken a menőverezés még oldalsó mozgás nélkül is nehéz. A mozgásukról pár pillanatképet láthatunk a 36. ábrán.



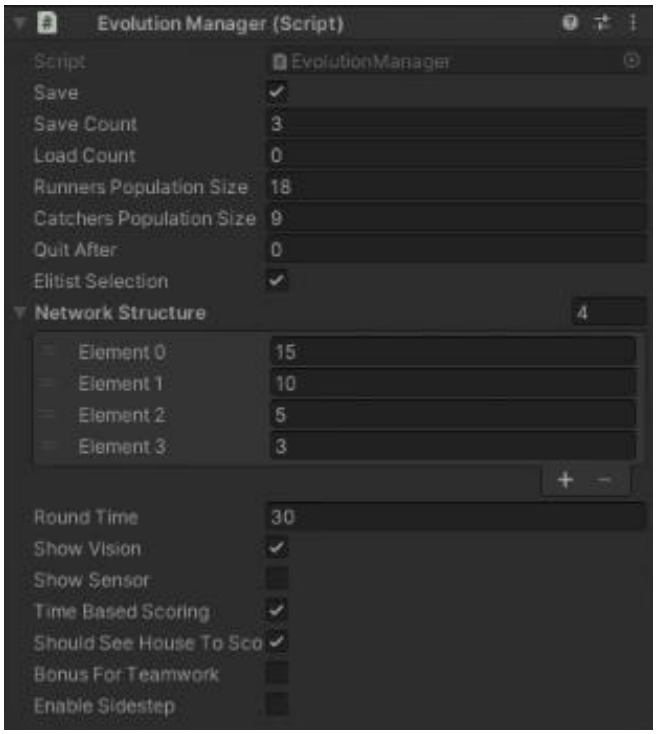
37. ábra Field2 szimuláció beállításai



38. ábra Pillanatkép a 48. generációról

III. 11. 2. Field2

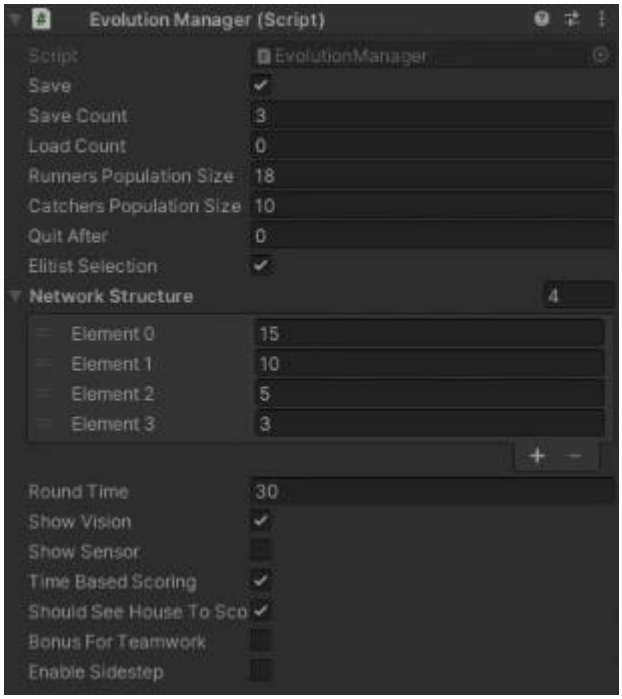
A Field2 pályán először elitista beállításokkal próbálkoztunk, azonban mivel a házak a folyosók legvégén helyezkednek el, a menekülőknek nehezére esett elérniük. A játékosok inkább csak keveset mozogtak, hogy ne menjenek falnak, ezzel szerezve minél több pontot. Ezután váltottam a 37. ábrán látható beállításokra. A játékosok ezúttal változatosabban viselkedtek eleinte, így több lehetőség volt arra, hogy valamelyikük elérjen egy házat. Az ötödik generációban ez meg is történt, ami után a jelentős különbség a szerzett pontszámok között a házba befutó játékos tulajdonságainak öröklődését erősítette. A további generációkban lassan nőtt a biztonságos zónákba érkező játékosok száma és a játékosok átlagos élettartama is stabilan növekedett, ahogyan az a 38. ábrán is megfigyelhető.



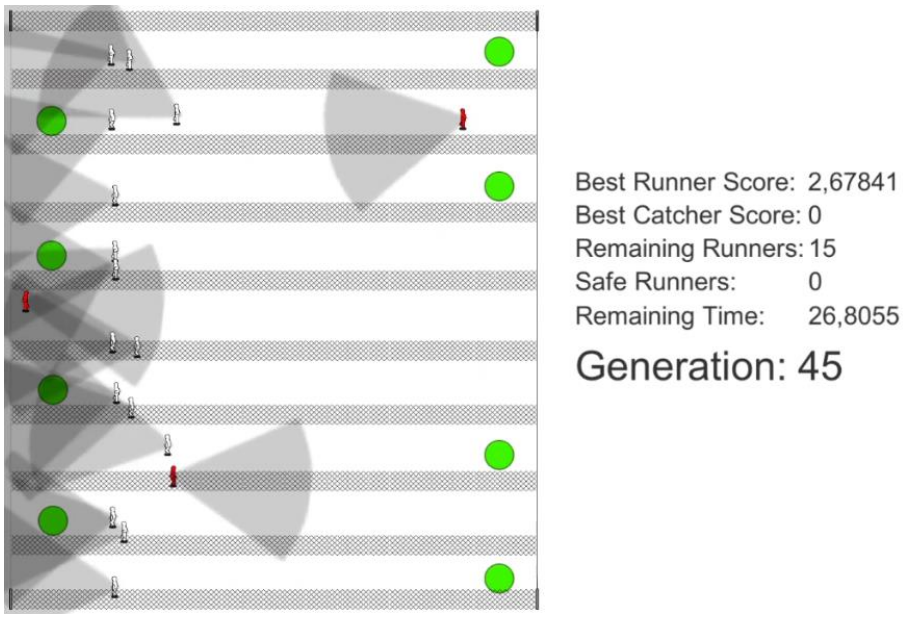
39. ábra Field3 szimuláció beállításai

III. 11. 3. Field3

A Field3 pályán használt beállítások a 39. ábrán láthatóak. Ez a pálya sajnos nem volt túl sikeres. A fogók és a menekülők is gyakran falakba ütköztek, így a játékosok környezete teljesen eltérő lett. Egy jó fogó akár 0 pontot szerezhetett, ha a folyosóján álló menekülők hamar falnak ütköztek. Hasonlóan egy ügyetlenebb menekülő is sokáig túlélhette, ha nem ment falnak és a folyosóján lévő fogó sosem jutott a közelébe.



40. ábra Field4 szimuláció beállításai

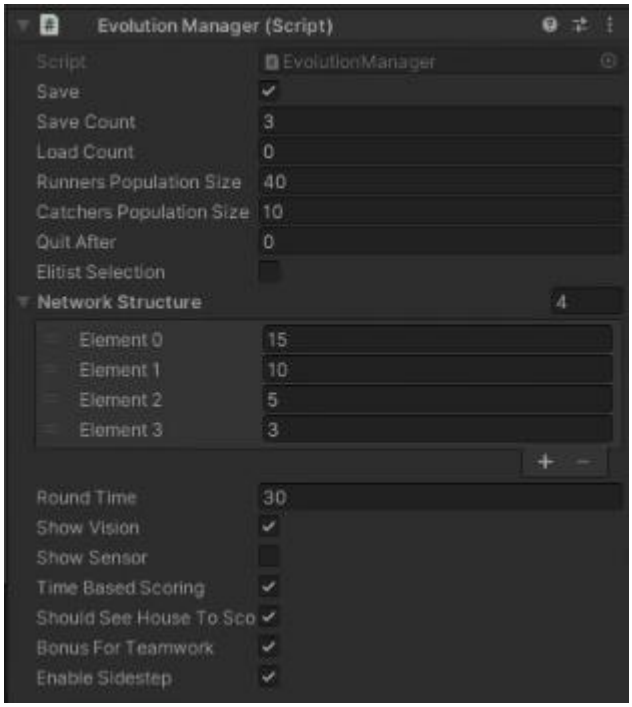


41. ábra Pillanatkép a 45. generációról

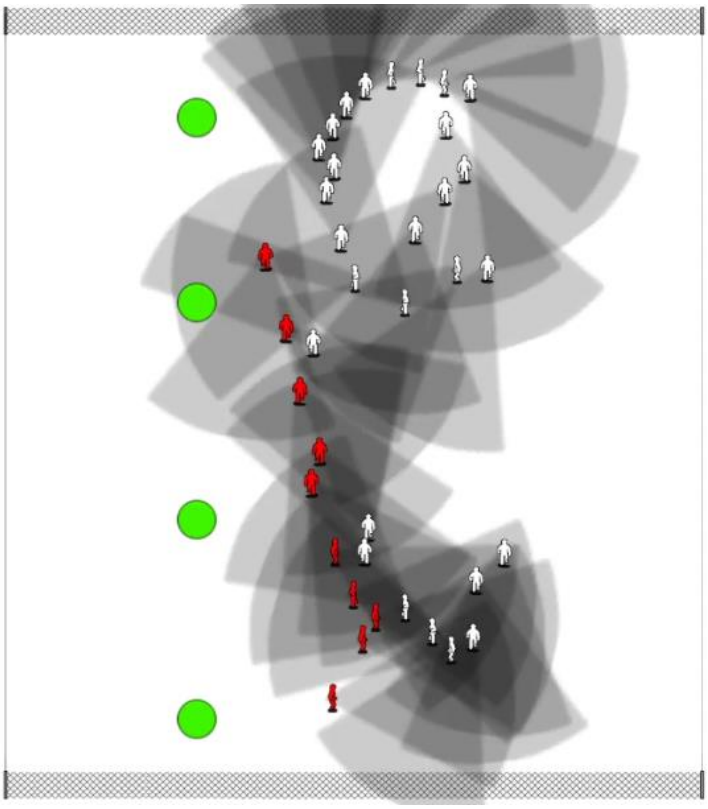
III. 11. 4. Field4

A Field4 pályán a hozzáadott házak jelentősen változtattak a játékosok viselkedésén. A 40. ábrán látható paramétereket használva a menekülők között sikeresen elterjed a tulajdonság, hogy érdemes egyenesen berohanni a házba, azonban a házak váltott oldali elhelyezése ellenére a menekülők többnyire ugyanabban az irányban kezdték el keresni a házakat, és a szűk folyosón ritkán voltak képesek megfordulni, pláne anélkül, hogy egy fogó elkapta volna őket. Erre látható példa a 41. ábrán. A fogók között az az egyszerű

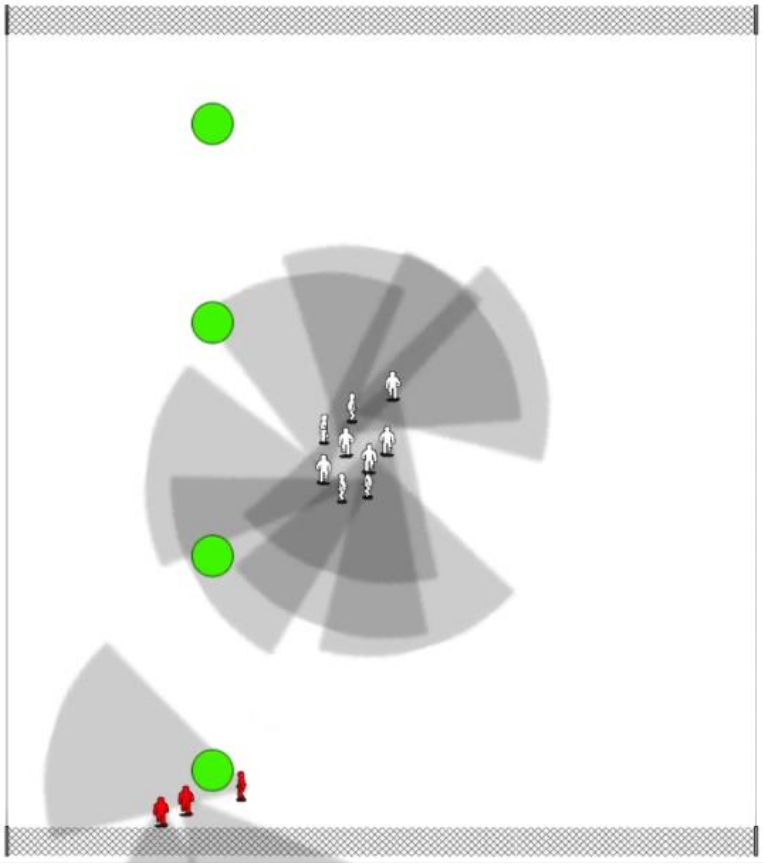
viselkedés alakult ki, hogy egyenesen előre mozogjanak bármi történik, mivel gyakran így is menekülőbe ütköztek, ezzel is pontot szerezve. Hiába volt a folyosó elég széles ahhoz, hogy elférjen két játékos egymás mellett, a játékosok ritkán voltak képesek ilyen precíz manőverezésre.



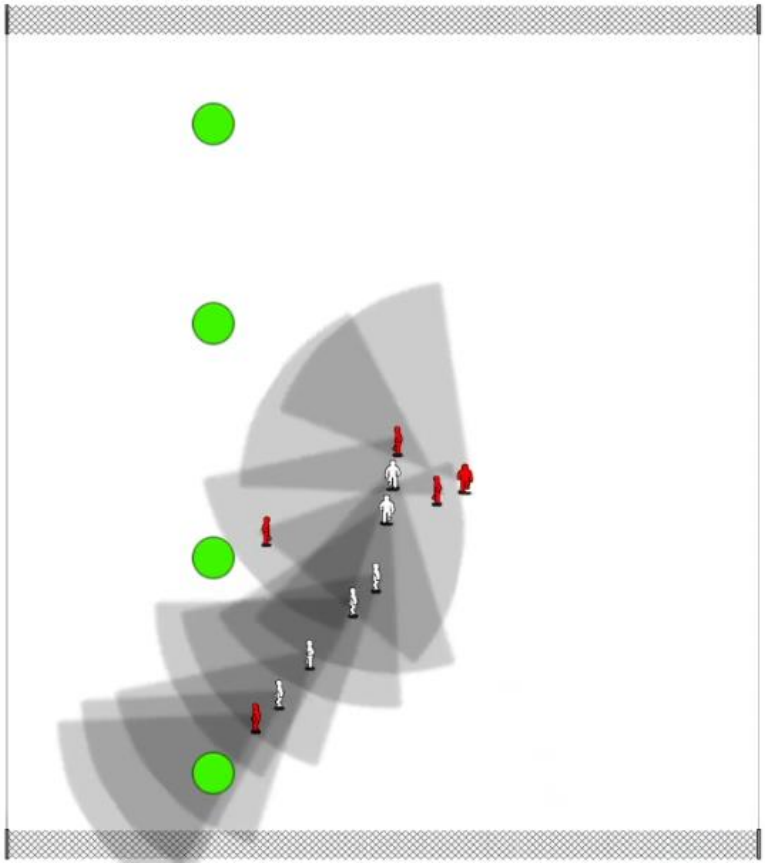
42. ábra Field5 szimuláció beállításai



43. ábra Pillanatkép az egymást követő menekülő játékosokról



44. ábra Pillanatkép a kört alkotó menekülő játékosokról



45. ábra Pillanatkép egy fogót követő menekülő játékosokról

III. 11. 1. Field5

A Field5 pályán voltak megfigyelhetőek a legérdekesebb viselkedési formák. A fogók és a menekülők a nagy létszám ellenére is gyakran csak a 60-70. forduló után kezdtek igazán hatékonnyá válni. A 42. ábrán látható, hogy a fogóknak ezúttal bónusz pontokat osztottunk ki a csapatmunkáért. Ennek következtében a fogók gyakran kisebb csoportokba gyűltek, de lehetőleg a csoport minden tagja próbált úgy helyezkedni, hogy legyen rálátása arra, ha az élen lévő fogó elkap egy menekülőt, ezzel biztosítva a plusz pont megszerzését. A csoport élén gyakran egy rendkívül hatékony fogó haladt, amelyhez a forduló során a legtöbb elkapott menekülő volt társítható. A csoportból olykor leváltak fogók, amelyekhez túl közel haladt el egy menekülő. Ekkor mozgásukkal kört leírva elkapták a játékost, majd visszatértek a csoporthoz. Azonban ha a menekülő megpróbált kitérni ezalatt, a fogó, bár így is elkapta, utána nehezen vagy egyáltalán nem talált vissza a csoporthoz, amit idő közben szem elől tévesztett, és egyedül folytatta útját.

A csoportos mozgás megfigyelhető volt a menekülőknél is, itt azonban gyakran inkább sorban haladtak egymás mögött, ahogyan az a 43. ábrán látható. Az élen álló játékoson kívül a többi nem sokat látott, mégis nagyon közelről követték a vezetőt, amely gyakran ügyesen navigált a menekülők között. A csoport nagy része így gyakran sokáig játékban maradt, csak egy-egy menekülőt sikerült oldalról elkapniuk a fogóknak. Ha az élen álló menekülő megtalálta a házat, az egész csapatot belevezette.

Egy másik megfigyelhető csapatos mozgás volt a 44. ábrán látható formáció kialakítása. A menekülők ilyenkor többnyire egy adott pont körül köröztek többen, látómezejüket a körből kifele fordítva. Ez az alakzat hatékony volt a fogók észrevételében bármilyen irányból is jöttek. Ha egy valakit sikerült elkapniuk a fogóknak, a menekülők áldozatává váltak a hátulról érkező fogóknak, akik a kör mentén haladva elkaptak mindenkit.

A 45. ábrán láthatjuk, ahogyan a menekülők által kialakított sor egy fogót követ. Ez a viselkedés gyakran megjelent, mivel a fogók ritkán mozogtak hátrafelé hirtelen. A menekülők ezért gyakran pont egy fogót szorosan követve voltak a legnagyobb biztonságban. Általában csak egyedül mozgó fogót követett bármelyik menekülő játékos, mivel egy csapatot nehezebb észrevétlenül megközelíteni.

IV. Összefoglalás

Sikerült megvalósítani egy kétdimenziós fogójátékot, illetve egy neurális hálózatot használó döntéshozatalon alapuló evolúciós algoritmust. Bár nem minden pálya váltotta be a hozzá fűzött reményeimet, az ötödik pályán megfigyelhető volt néhány érdekes viselkedési forma.

A fejlesztés során rengeteg ötletem támadt, hogy milyen irányba lehetne fejleszteni a programot. Eleinte gondolkoztam egy menürendszer létrehozásán, hogy a program paraméterei a futtatható .exe-ben is beállíthatóak legyenek, ne kelljen a Unityt használni hozzá. De ahogy haladtam előre, felfedeztem, hogy mennyi funkciója van a Unitynek, ami megkönnyíti a szimulációt, és amiket újra kéne alkotni ahhoz, hogy a program ne vesszítsen funkcióiból egy kész build esetén se.

Természetesen tovább lehet kísérletezni a paraméterekkel, hogy jobb, vagy változatosabb stratégiákat produkáljunk. A program kialakítása új pályák létrehozását és hozzáadását is rendkívül könnyűvé teszi. Legizgalmasabb talán mégis az, hogy a megvalósított logikák közül sok másik játékokra is érvényes. Elég csupán azt átírni, hogy miért kaphat egy játékos pontot, vagy miért esik ki a játékból, és máris teljesen más játékot szimulálunk. A tény, hogy egy és két játékostípust is tud egyaránt kezelni a program, csak még jobban növeli a lehetőségek számát.

Úgy vélem, ez a projekt izgalmas és kihívást jelentő volt. Bízom benne, hogy a program olyan szilárd alapot tud jelenteni akár más mesterséges intelligencia projektek számára is, mint amilyen szilárdan megalapozta a neurális hálózatokról, az evolúciós algoritmusokról és a Unity használatról való tudásomat.

VI. Irodalmi jegyzék

Az alábbi hivatkozott linkeket 2021.december 14-én elértem.

[1] Az OpenAI csatorna Multi-Agent Hide and Seek videója. (online)
https://www.youtube.com/watch?v=kopoLzvh5jY

[2] A Unity átalános kezelőfelülete a Unity Documentation alapján. (online)
https://docs.unity3d.com/Manual/UsingTheEditor.html

[3] A CircleCollider2D osztály a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/CircleCollider2D.html

[4] A BoxCollider2D osztály a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/BoxCollider2D.html

[5] A Monobehaviour osztály a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[6] A Monobehaviour Start() metódusa a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html

[7] A Monobehaviour Awake() metódusa a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html

[8] A Monobehaviour Update() metódusa a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[9] A Monobehaviour FixedUpdate() metódusa a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html

[10] A Monobehaviour OnCollisionEnter2D(Collision2D) metódusa a Unity Documentation alapján. (online)
https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnCollisionEnter2D.html