



**Budapesti Műszaki és Gazdaságtudományi Egyetem**  
Villamosmérnöki és Informatikai Kar  
Automatizálási és Alkalmazott Informatikai Tanszék

Molnár Levente

# UNITY JÁTÉKFEJLESZTÉS

Körökre osztott stratégiai játék készítése Unity 3D  
keretrendszerrel

KONZULENS

Hideg Attila

BUDAPEST, 2023

# Tartalomjegyzék

<b>Összefoglaló .....</b>	<b>5</b>
<b>Abstract.....</b>	<b>6</b>
<b>1 Bevezetés .....</b>	<b>7</b>
1.1 Stratégiai játékok .....	7
1.1.1 Az ötlet forrása.....	8
1.1.2 A projekt célja.....	8
1.1.3 Követelmények elemzése .....	9
<b>2 Használt technológiák.....</b>	<b>10</b>
2.1 Átfogó technológiák .....	10
2.1.1 C#.....	10
2.1.2 .NET keretrendszer .....	10
2.1.3 JSON .....	10
2.2 Alapjáték, naplózó- és visszajátszó rendszer .....	11
2.2.1 Unity keretrendszer.....	11
2.3 Hálózati kommunikáció .....	11
2.3.1 Websocket.....	12
2.4 Mesterséges intelligencia .....	12
2.4.1 Python .....	12
2.5 Automatizáló rendszer .....	13
2.5.1 Docker konténerizáció .....	13
2.6 Weboldal .....	13
2.6.1 ASP.NET Core.....	14
2.6.1.1 Blazor Server .....	14
2.6.1.2 ASP.NET Core Identity .....	15
<b>3 A rendszer felépítése és működése .....</b>	<b>16</b>
<b>4 Komponensek bemutatása .....</b>	<b>18</b>
4.1 A játék.....	18
4.1.1 A játék célja .....	18
4.1.2 Alapszabályok, funkciók .....	18
4.1.2.1 Felhasználói felület .....	19
4.1.2.2 Konfiguráció .....	21

4.1.3 Naplózó rendszer .....	21
4.1.4 Visszajátszó rendszer .....	22
4.1.4.1 Felhasználói felület .....	23
4.2 Automatizáló rendszer .....	24
4.2.1 Működése .....	25
4.2.1.1 Hálózati kommunikáció .....	29
4.2.2 Mesterséges intelligencia fejlesztése .....	30
4.2.2.1 Játék stratégiák .....	30
4.2.2.2 Konténerizálás .....	31
4.2.2.3 Hálózati kommunikáció .....	32
4.3 Weboldal .....	32
4.3.1 Felhasználókezelés .....	33
4.3.2 Kezdőoldal .....	34
4.3.3 Mesterséges intelligencia feltöltése .....	34
4.3.4 Versenyek lebonyolítása .....	35
4.3.4.1 Verseny indító felület .....	35
4.3.4.2 Verseny indító implementációja .....	37
4.3.4.3 Eredmények .....	38
<b>5 Az implementáció és a tesztelés folyamata .....</b>	<b>39</b>
<b>6 Összegzés .....</b>	<b>41</b>
6.1 Projekt értékelése .....	41
6.2 Továbbfejlesztési lehetőségek .....	41
<b>Irodalomjegyzék .....</b>	<b>43</b>

# HALLGATÓI NYILATKOZAT

Alulírott **Molnár Levente**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2023. 12. 13.

.....  
Molnár Levente

# Összefoglaló

A stratégiai játékok napjainkban is nagy népszerűségnek örvendenek a videojátékok terén részletgazdagságuk, és a bennük található számtalan játékfunkciónak köszönhetően. Többek között emiatt is kifejezetten izgalmas az a terület, amely az ilyen típusú játékokhoz készíthető mesterséges intelligenciákkal foglalkozik.

Ezen játékokhoz pont a magas komplexitásuk miatt nagy kihívás a saját mesterséges intelligencia készítése, a fejlesztési és tesztelési folyamatok akkor is lassúak és bonyolultak lehetnek, ha a játékhoz rendelkezésre áll a megfelelő keretrendszer ennek elkészítéséhez.

A projekt ezen hivatott segíteni, ehhez adott egy korábban elkészített, viszonylag egyszerű játék, ezen dokumentum pedig arra szolgál, hogy bemutassa a játékhoz elkészített komponenseket, melyek képesek mesterséges intelligenciák közötti versenyek lebonyolítására, valamint lehetővé teszik a végbement játszmák megtekintését.

A probléma megoldásához készült komponensek között találunk egy, a Unity keretrendszer segítségével készült visszajátszó rendszert, egy konténerizált kliensekkel működő verseny automatizáló logikát, valamint egy webportált, amely lehetővé teszi a mesterséges intelligenciák szerverre való feltöltését, tárolását, kezelését, illetve a versenyek konfigurációját, elindítását, végül az eredmények megtekintését.

# **Abstract**

Strategy games continue to enjoy great popularity in the field of video games, thanks to their intricacy and the numerous gameplay features they offer. This is one of the reasons why the field of artificial intelligence for this type of games is particularly exciting.

Building your own AI for these games is a significant challenge because of their high complexity, the development and testing processes can be slow and intricate even when the game has the appropriate framework for implementing it.

The project is intended to help with this by providing a relatively simple game that has been created previously. This document serves to showcase the components created for the game, which are capable of organizing and running competitions between artificial intelligences and allowing to view the replay of completed matches.

The components designed to solve the problem include a replay system built using the Unity framework, a tournament automation logic operating with containerized clients, and a web portal that facilitates the upload, storage and management of artificial intelligences on the server, as well as the configuration and initiation of tournaments, then viewing the results.

# 1 Bevezetés

Ezen fejezet bemutatja, hogy honnan ered a projekt ötlete, valamint, hogy milyen szempontokat, követelményeket fogalmaztunk meg a projekttel kapcsolatban a fejlesztést megelőzően.

## 1.1 Stratégiai játékok

A stratégiai játék kategória a játékokon, jelen esetben a videojátékokon belül egy olyan műfajt képvisel, amelybe többnyire magas komplexitással rendelkező, rengeteg funkciót, játékmechanikát, apróbb játékelemet tartalmazó játékok tartoznak. Ezek a játékok legtöbbször nagyon változatos, részletgazdag, ugyanakkor sok tervezést, gondolkodást – ideértve az előregondolkodást is – igénylő játékelményt garantálnak a játékosok számára. Az ilyen játékokban minden apró döntés számít, nagyon fontos az erőforrások, egységek jó menedzsmentje, emiatt általában sok időt is vesz igénybe a játék minden részletének elsajátítása, viszont pont ezért lesz érdekes és izgalmas, mert akkor lehet eredményesen játszani, ha minden ilyen részletet elsajátít a játékos.[1]

A stratégiai játékoknak több típusa létezik, ilyenek például a valós idejű, vagy a körökre osztott stratégiai játékok, de ide tartoznak a 4X játékok, a városépítős szimulációs játékok, valamint különböző tábla- és kártyajáték adaptációk is.

Ahogy a típusuk, úgy a témájuk is rengeteg féle lehet. Léteznek történelmi, háborús, tudományos-fantasztikus, különböző fantázia világban játszódó, valamint valósághű témájú játékok is, melyek dizájnja, kinézete sokat hozzátesz a játékelményhez. A dizájn lehet valósághű, táblajáték kinézetű, absztrakt minimalista, rajzfilmes, de pixel art is.

A stratégiai játékok egyik típusába tartoznak a körökre osztott stratégiai játékok. Ezek lényege, hogy a játékosok egymás után következnek, akár csak egy valódi társasjáték során. A játékosoknak meg van határozva, hogy egy körön belül milyen cselekvéseket, illetve, hogy a különböző cselekvésekből hányat hajthatnak végre, a lépések végrehajtása után tovább kell adniuk a kört a következő játékosnak. Ez a körökre bontás, a játékbeli idő diszkrét egységekre bontása lehetővé teszi, hogy a játékosoknak legyen idejük átgondolni a lépéseiket, megtervezniük a győzelemhez szükséges stratégiát,

emellett ez elősegíti azt is, hogy egy új játékos könnyebben megtanulhassa a játék szabályait, mit hogyan kell, vagy hogyan érdemes csinálni.

### **1.1.1 Az ötlet forrása**

A projekt ötlete is ilyen, az előzőekben említett kategóriába tartozó játékokon alapul, ezek közül külön kiemelendő a Civilization VI nevű játék. A Civilization VI egy körökre osztott stratégiai játék, amelyben egy olyan birodalmat kell építeni, amely kiállja az idők próbáját. A játék a kőkorszakban kezdődik, majd a birodalom folyamatos fejlesztése, különböző technológiák megszerzése révén juthat el a játékos egészen napjainkig, valamint azon túlra is. A játékosnak irányítania kell a birodalom gazdaságát, iparát, mindeközben társadalmi, kulturális és technológiai előrelépéseket tenni, illetve diplomáciai kapcsolatot kialakítani a többi játékos által irányított birodalmakkal is, melyek háborúhoz, vagy éppen szövetségekhez vezethetnek. A játékot többféleképpen meg lehet nyerni, ilyen lehet a hadászati, kulturális, technológiai vagy diplomáciai fölény, csak a játékoson múlik, hogy melyikre mennyi hangsúlyt fektet, közben viszont meg is kell tartsa az ezek közötti egyensúlyt ahhoz, hogy sikeres legyen.[2]

### **1.1.2 A projekt célja**

Kezdetben, az önálló laboratórium keretein belül, csapatban elkészített játék során egy, az előzőekben leírtakhoz mind témában, mind az elkészített funkciókban hasonló, de természetesen leegyszerűsített alapjátékot valósítottunk meg, amelyben 2-4 fő játszhat egyszerre egymás ellen, egy véletlenszerűen generált pályán. Ehhez készítettünk még egy naplózó rendszer is, amely elmentette a végbement játékokban történt lépéseket, valamint egy kezdetleges, hálózati kommunikációt megvalósító interfészt is.

A szakdolgozatban, immáron egyedül végzett fejlesztéseim során elkészített komponensek célja az volt, hogy ezt a projektet továbbfejlesztve kiegészítsék újabb elemekkel, amelyek végül lehetővé teszik, hogy könnyedén be lehessen konfigurálni, elindítani és végigkövetni egy, az indítást követően már automatikusan végbemenő, mesterséges intelligenciák által játszott, több játszámából álló versenyt, majd ennek összesített eredményét meg lehessen tekinteni, valamint a verseny során végbement mérkőzéseket a naplófájlaik alapján vissza lehessen játszani lépésről-lépésre egy grafikus visszajátszó rendszer segítségével.



### 1.1.3 Követelmények elemzése

A visszajátszó rendszer feladata, hogy egy végbement játék során készült naplófájl beolvasson, a játék logikai komponensét inicializálja a beolvasott adatokkal, majd megjelenítse a kiinduló állapotot a felhasználó számára. Ezt követően a felhasználó a grafikus felületen tudja léptetni a játék állapotát, minden lépésnél a visszajátszó rendszer a naplófájl következő bejegyzése alapján frissíti a megjelenített játékállást, ezáltal végig lehet nézni az egész lejátszott mérkőzést.

A projekt során a mesterséges intelligencia elkészítésének célja, hogy a többi komponenst, többek között az automatizációs-, valamint a visszajátszó rendszert, ezek működését tesztelni lehessen vele. Ennek feladata, hogy csatlakozzon a játékot futtató szerverre, majd a játék egy adott pillanatnyi állapota alapján eldöntse, mi legyen a következő lépés, valamint ezt a lépést továbbítsa a játék felé a közöttük kialakított kommunikációs csatornán keresztül.

A játékban egymással versengő mesterséges intelligenciák egy-egy konténerben futnak, ezek létrehozását és kezelését, valamint a játék és a konténerek közötti kommunikációs csatorna konfigurálását az automatizáló rendszer végzi. A mesterséges intelligenciák konténerekben való futtatása egységessé, ezáltal egyszerűbbé teszi a kezelésüket, a mesterséges intelligencia bármilyen programozási nyelven készülhet, a konténerizáció elfedi a közöttük lévő implementációs különbségeket.

A weboldal feladata, hogy leegyszerűsítse az alkalmazás használatát az alábbi funkciók megvalósításával. A weboldalon a játékosoknak lehetőségük van mesterséges intelligenciát tartalmazó fájlok feltöltésére, kezelésére, a szervezőknek pedig versenyek konfigurálására, elindítására, az eredmények megtekintésére. A konfiguráció során lehetőség van kiválasztani a versenyben résztvevő mesterséges intelligenciákat, a pályát, amin a verseny zajlik, valamint beállítani a verseny további paramétereit.

Az automatizáló rendszer feladata, hogy a weboldalon beállított konfigurációs paraméterekkel, valamint a kiválasztott fájlokkal lebonyolítsa a versenyt. Minden mérkőzéshez indítson egy játékot, hozza létre a konténereket, futtassa bennük a kiválasztott mesterséges intelligenciákat, kösse össze a hálózaton keresztül a játékot a konténerekkel, majd a játék befejeztével bontsa fel a kapcsolatot, szabadítsa fel az erőforrásokat, valamint értesítse a weboldalt az eredményről.

## 2 Használt technológiák

Ebben a fejezetben a projekt fejlesztése során használt technológiák kerülnek bemutatásra, kiemelve az adott komponenst, komponenseket, melyek elkészítése során használtam őket, valamint azt is, hogy miért ezekre esett a választás.

### 2.1 Átfogó technológiák

Ezen technológiák a projekt szinte minden komponensének fejlesztése során elengedhetetlenek voltak, ezek képezték a projekt alapját.

#### 2.1.1 C#

A C# a Microsoft által fejlesztett, rugalmas, objektum-orientált programozási nyelv, melynek sok különböző felhasználási területe van, ide tartoznak a webes-, mobil-, és az asztali alkalmazások, valamint a játékok is, ahogy korábban is említettem, a Unity keretrendszerben írt játékok forráskódjai is ezen a nyelven készülnek.[3]

A projekt elkészítése során tehát elengedhetetlen technológia volt, a játék logikája és a visszajátszó rendszer mellett a weboldal, valamint az automatizáló rendszer fejlesztése során is ezt a programozási nyelvet használtam, ezáltal kényelmesen meg tudtam valósítani a komponensek közötti kommunikációt.

#### 2.1.2 .NET keretrendszer

A .NET egy, a Microsoft által fejlesztett, nyílt forráskódú alkalmazásfejlesztési környezet, amely támogatja – többek között – a C# nyelvben való alkalmazások készítését. Fordítót, futtatókörnyezetet, valamint rengeteg beépített és külső könyvtárat is biztosít a fejlesztők számára, ide tartozik a projektben felhasznált, Unity alapú játékfejlesztést támogató könyvtára is.[4]

#### 2.1.3 JSON

A JavaScript Object Notation, vagyis JSON egy könnyűsúlyú adatsereformátum, amelyben emberek által könnyen írható és olvasható, viszont gépek által is könnyen feldolgozható és generálható szerkezetben lehet adatot tárolni kevés szintaktikai adatfelesleggel. A JSON univerzális, tehát bármilyen adatszerkezet leírására alkalmas,

emellett szinte minden modern programozási nyelv támogatja a használatát, beleértve a projektben használt C# nyelvet is.[5]

A JSON adatstruktúrát a projektben a naplózó rendszer által készített naplófájlok, a játék különböző paramétereit beállító konfigurációs fájlok, a játék által generált pályák, valamint a játék lebonyolítása során a szerver és a kliensek közötti kommunikációra használtuk fel.

Azért döntöttünk a JSON formátum mellett, szemben például a szintén népszerű és széles körben használt Extensible Markup Language (XML) [6] formátummal, mivel a hálózati kommunikáció során fontos, hogy minél kevesebb adat kerüljön elküldésre, ezáltal is csökkentve a hálózati forgalmat, valamint a késleltetést. A JSON tömörebb formátuma emiatt előnyösebbnek bizonyult, emellett a .NET keretrendszerhez készült Json.NET kiegészítő komponenssel az adatok feldolgozása is egyszerű volt.

## **2.2 Alapjáték, naplózó- és visszajátszó rendszer**

A játékhoz, illetve a visszajátszó rendszerhez tartozó grafikus megjelenítésekben közös, hogy ugyanazon technológia, a Unity keretrendszer segítségével készítettem el őket.

### **2.2.1 Unity keretrendszer**

A Unity egy játékfejlesztő motor, amely platformfüggetlenségével támogatja a fejlesztőket abban, hogy játékokat készítsenek számtalan platformra, beleértve a mobil eszközöket, számítógépeket, játékkonzolokat, valamint a kiterjesztett- és virtuális valóságot támogató eszközöket. A keretrendszer fejlesztőbarát felhasználói interfésszel, hatalmas eszköztárral, valamint magasszintű kódírási támogatással rendelkezik az által, hogy a játékokhoz használt szkripteket C# nyelven lehet implementálni, ami nagyban megkönnyíti a fejlesztést. A Unity széleskörben használt a játékfejlesztői iparban, de ezen kívül sok más területen is alkalmazzák, ide tartozik többek között a szimuláció, az építészet, valamint a filmipar is.

## **2.3 Hálózati kommunikáció**

A játékot futtató szerver, valamint a kliensek, jelen esetben a konténerekben futó mesterséges intelligenciák között a játék során szükség van adatok átküldésére. A szerver elküldi a klienseknek a játék állapotát, az éppen soron következő kliens pedig válaszként

elküldi a szervernek az általa választott lépést. Ez az összeköttetés websocket alapú kommunikáción keresztül történik.

### **2.3.1 Websocket**

A websocket egy kommunikációs technológia, amely interaktív kétirányú kommunikációt tesz lehetővé a két fél között egy darab, hosszú élettartamú kapcsolat segítségével. A kapcsolat ideje alatt a két fél tetszőleges számú üzenetet küldhet és fogadhat egymás között.[7]

A játék elindítása előtt minden kliens egy külön websocket segítségével kapcsolódik a szerverhez, majd a játék során ezeken a csatornákon keresztül történik az adatátvitel.

A projektben azért döntöttünk – még az önálló laboratórium tárgy keretében történő fejlesztések során – a websocket alapú kommunikáció használata mellett, mert ezáltal egyszerűen meg tudtuk valósítani a kapcsolatok élettartamát. Fontosnak tartottuk, hogy a játék során végig aktív kapcsolat legyen a kliensek és a szerver között, viszont a játék befejeztével a csatornákat be lehessen zárni, ezáltal felbontani a megnyitott kapcsolatokat. A websocket egyik előnye, hogy nem kell minden adatátvitelhez külön kapcsolatot kiépíteni, majd azt lebontani, ezáltal csökkenthettük a késleltetést. Emellett a websocket előnyös tulajdonságai közé tartoznak, hogy az átküldött plusz adat mennyisége kicsi a lényeges tartalomhoz képest, valamint, hogy mindkét fél tud adatot küldeni és fogadni is, ami az általunk megvalósított projektben elengedhetetlen volt.

## **2.4 Mesterséges intelligencia**

A játékban egymással versengő klienseket, vagyis a játékosokat mesterséges intelligenciák adják. A projekt során implementáltam egy kezdetleges, viszont a teszteléshez tökéletesen megfelelő mesterséges intelligencia logikát, ehhez a Python nyelvet használtam.

### **2.4.1 Python**

A Python egy általános célú, interpretált, objektum-orientált, magasszintű programozási nyelv, amely magasszintű beépített adatstruktúrákat tartalmaz, ezt ötvözi a dinamikus típusfelismeréssel és dinamikus típuskezeléssel, ezáltal nagyon népszerű a gyors alkalmazásfejlesztési módszertant alkalmazók körében.[8]

A mesterséges intelligencia fejlesztése során azért választottam a Python nyelvet, mert ebben könnyedén meg tudtam valósítani a JSON formátumban kapott adatok feldolgozását, valamint a szintén JSON formátumú válasz előállítását. Az adatfeldolgozás mellett pedig a websocket kommunikáció megvalósítása is egyszerű volt, ehhez a websocket [9], valamint a websocket-client [10] nevű könyvtárakat használtam.

## **2.5 Automatizáló rendszer**

Az automatizáló rendszer elkészítéséhez szükség volt konténerizációs technológia használatára, amely lehetővé tette, hogy a különböző programozási nyelveken, különböző technológiákat használó mesterséges intelligenciák implementációjuktól függetlenül, egységesen kezelhetők legyenek.

### **2.5.1 Docker konténerizáció**

A Docker egy szoftver platform, melynek segítségével gyorsan lehet alkalmazásokat fordítani, tesztelni, és kitelepíteni. A Docker standardizált egységekbe, úgynevezett konténerekbe csomagolja a szoftvert. A konténer tartalmaz mindent, ami a szoftver futásához szükséges, beleértve a futtatókörnyezetet, a szükséges könyvtárakat, operációs rendszerbeli eszközöket. Ezen függőségeket a Dockerfile nevű fájlban kell megadni, az ebben található információk alapján fog elkészülni a szoftver image (memóriakép) fájlja, amiből később tetszőleges konténert lehet példányosítani.[11]

A mesterséges intelligenciák esetében a játék előtt létre kell hozni a konténereket, ezekben a játék ideje alatt egymástól függetlenül tudnak futni, csatlakozni a játék szerveréhez, majd a játék végeztével csak a létrehozott konténereket kell megsemmisíteni.

Azért választottam a Docker technológiát, mert manapság ez az egyik legnépszerűbb konténerizációs megoldás, ezáltal rengeteg segédlet, dokumentáció létezik hozzá, emellett van grafikus felhasználói interfésze is Docker Desktop [12] néven, valamint a .NET környezetben a Docker.DotNet [13] csomaggal tudtam forráskódból kezelni a konténereket.

## **2.6 Weboldal**

A projekt minden eddigi komponense a .NET keretrendszeren, valamint a C# programozási nyelven alapul, ezért amikor a weboldal megvalósítása előtt technológiát

kellett választanom, kézenfekvőnek találtam, hogy ezt is a már jól megszokott környezetben implementáljam, ezzel is egyszerűsítve a többi komponenssel való összeköttetés megvalósítását.

## **2.6.1 ASP.NET Core**

Az ASP.NET Core egy .NET alapú, platformfüggetlen, nagy-teljesítményű, nyílt forráskódú keretrendszer modern, internethez kötött alkalmazások fejlesztéséhez. A keretrendszer támogatja mind a felhasználói felület, az üzleti logika, mind az adatbázis-elérési réteg fejlesztését, így nem szükséges ezeket külön technológiákkal megvalósítani.[14]

### **2.6.1.1 Blazor Server**

A Blazor egy webes keretrendszer a felhasználói interfész komponenseinek implementálásához. Minden felhasználói felületi komponens egy-egy Razor [15] komponens, amelyekben a weboldalakon található dinamikusan változó tartalom is implementálható. Ezek a komponensek egymásba ágyazhatóak, újrafelhasználhatóak, ezzel is növelve a modularitást.[16]

A weboldal tervezésekor két megközelítés közül kellett választanom, az egyik a Blazor Server, a másik pedig a Blazor WebAssembly [17] volt. A választás azért esett a Blazor Server modellre, mert szemben a Blazor WebAssembly megközelítéssel a Blazor Server modell alkalmazása esetén a komponensek feldolgozása szerveroldalon, az ASP.NET Core applikáción belül történik, ezáltal kliens oldalon kevesebb a letöltött adat mérete, valamint mivel már minden komponens feldolgozva érkezik meg, ezért gyorsabban is tölt be az alkalmazás.

Ez a megközelítés a vékonykliens alkalmazásokat támogatja, tehát az adatok feldolgozását a szerveren végzi, a kliensen csak a parancsok kiadása történik, nagyobb számításokat nem végez. Mivel az általam készített weboldal feladata a versenyek elindítása, kezelése, ezen parancsok feldolgozása pedig szerveroldalon kell történnie, ezért is találtam jobb választásnak ezt a megközelítést.

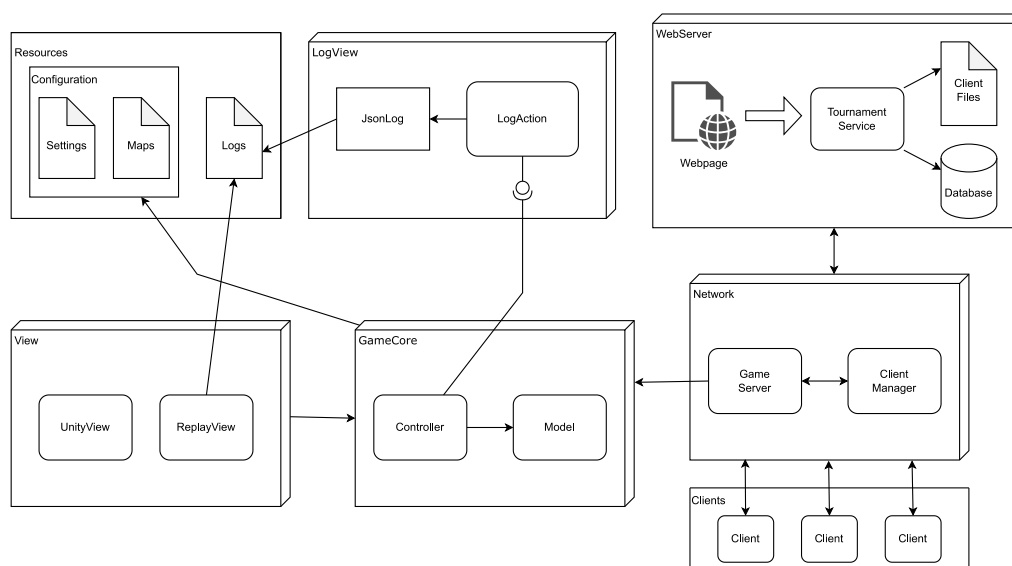
A használat mellett a Blazor Server fejlesztést támogató előnyökkel is rendelkezik, mivel az alkalmazás szerveroldalon fut, ezáltal a megszokott módon működik minden .NET keretrendszerbeli eszköz, többek között például a hibakeresés funkció is.

### **2.6.1.2 ASP.NET Core Identity**

Az ASP.NET Core Identity egy olyan API (Application Programming Interface, magyarul: alkalmazásprogramozási interfész), amely lehetővé teszi az ASP.NET alapú alkalmazások számára a felhasználókezelést. Ez magába foglalja a felhasználók menedzsmentjét, a jelszavak, felhasználói adatok, jogosultságok stb. adatbázisban való tárolását. Az API továbbá biztosít alap sablon weboldalakat a különböző felhasználói interakciókhoz, mint például a regisztráció, bejelentkezés, email cím megerősítés vagy a profil szerkesztése nézet. Ezekhez mind a megjelenített oldal, mint a mögötte található logika automatikusan generálódik, ezt követően mindent testre lehet szabni.[18]

A projekthez elkészítése során kézenfekvő megoldásnak találtam ennek a technológiának a használatát, mivel kifejezetten az ASP.NET projektekhez készült, így szorosan együtt tud működni az általam implementált weboldallal, tökéletesen megvalósítva a szükséges minimális, de ugyanakkor biztonságos felhasználókezelést.

### 3 A rendszer felépítése és működése



1. ábra: A rendszer felépítése

A projekt hat nagyobb alkalmazáskomponensből áll, ezek együttesen valósítják meg a projekt fő célját, hogy mesterséges intelligenciák között játszó versenyeket bonyolíthassunk le, valamint újranezrhessük a lejátszott mérkőzéseket. A rendszer felépítésének szemléltetésére szolgál az 1. ábra.

Az első komponens a webservert, amely a felhasználói felületből, vagyis a weboldalakból, valamint a versenyeket kezelő szolgáltatásból áll. A weboldal segítségével könnyedén konfigurálhatjuk az elindítani kívánt versenyt, az elindítást követően a szolgáltatás pedig kapcsolatba lép a játék hálózati kommunikációs interfészét megvalósító komponenssel. A szolgáltatás emellett kezeli a kliensek fájljait a fájlrendszeren, valamint rendelkezik egy adatbázissal is, amelyben a felhasználók adatai, illetve a versenyek eredményei vannak eltárolva.

A második komponens a játékot futtató szervert valósítja meg, amelyhez tartozik egy hálózati kommunikációs interfész is, ezen keresztül tudnak csatlakozni a kliensek, valamint kommunikálni a szerverrel a játék időtartama alatt. A komponens a webservert indítja el egy verseny elindításakor. Minden, a verseny során lejátszódó mérkőzéshez új játék szerver indul. Indulását követően a szerver létrehoz a webszervertől kapott adatok alapján egy játékot, amely várja a kliensek csatlakozását. Ezt követően a klienskezelő egység létrehozza a fájlok alapján a klienseket, majd csatlakoztatja őket az elindított szerverhez. Ha minden csatlakozás sikeres volt, akkor megkezdődhet a játék. A játékot



futtató szerver biztosít egy interfészt a kliensek számára, melyen keresztül lekérdezhetik a játék állapotát, illetve elküldhetik a végrehajtani kívánt lépéseiket.

A játék lebonyolítását a játéklógika komponens végzi. A játéklógika tárolja a játékosokhoz tartozó különböző tulajdonságokat, játékbeli értékeiket, valamint minden végrehajtandó lépésnél ellenőrzi a cselekvés helyességét, szabálytalan lépést nem enged végrehajtani. A különböző nézetek főként a játéklógika kontroller egységén keresztül érik el a modellben található tényleges logikát, csak a kontrolleren keresztül tudják módosítani a játék állapotát. Ennek lényege, hogy minél nehezebb legyen érvénytelen állapotba hozni a játékot, valamint egyértelműek legyenek a megvalósított funkciók. A játéklógika külön komponensbe való szervezése függetlenné teszi a logikát megjelenítéstől, ennek is köszönhetően lehetett több, egymástól eltérő felhasználói interfészt készíteni hozzá.

A játék folyamán a játéklógika kontroller egysége folyamatosan értesíti a naplózó komponenst a végrehajtott lépésekről, amely a lépések adatait fájlbejegyzéssé alakítja, ezt követően pedig a játék elején elkészített naplófájlba menti. Minden végbement játékról egy külön naplófájl jön létre.

A játékhoz tartozik két grafikus nézet is, ebből az egyik a visszajátszó rendszer. Ez a komponens megkeresi a megadott fájlnev alapján a kiválasztott naplófájlt a projekthez tartozó fájlrendszeren, majd beolvassa azt. A tartalma alapján inicializál egy játékot, majd megjeleníti a felhasználónak. A visszajátszás segítségével tekinthetjük meg újra a korábban végbement játékokban történt cselekvéseket, akár lépésről-lépésre is. Eközben a visszajátszó rendszer különböző segédinformációkat is megjelenít a játékkal, valamint az aktuális lépéssel kapcsolatban.

A játékhoz tartozó másik grafikus felhasználói nézet a játék nézet. Ennek segítségével tudnak emberi játékosok játszani a játékkal, amely a külön komponensbe szervezett játéklógikának köszönhetően teljesen ugyanúgy működik, mint amikor a mesterséges intelligenciák játszanak egymás ellen. Eltérés viszont, hogy ezen nézet használatakor a játékosok csak lokálisan, egy eszközön tudnak egymás ellen játszani, hálózaton keresztül nem.

Az alkalmazáshoz tartozik az említett komponenseken kívül még az erőforrásokat tároló fájlrendszer is, ahol elérhetőek a játékhoz tartozó konfigurációs fájlok, a korábbi játszmákhoz generált pályák, valamint a végbement játékokról készült naplófájlok is. Ezen fájlokat a komponensek felhasználhatják működésük során.

## 4 Komponensek bemutatása

Ebben a fejezetben bemutatom az alkalmazás különböző komponenseit, ezek feladatát, működését, fontosabb implementációs lépéseit.

### 4.1 A játék

Kezdsnek először a magát a játékot, valamint a hozzá közvetlenül kapcsolódó komponenseket szeretném bemutatni, hiszen ezek képezik a projekt alapját.

#### 4.1.1 A játék célja

A játék célja, hogy a játékosok a többi játékos összes városát elpusztítsák, ha egy játékos összes városa megsemmisül, akkor kiesik a játékból. Az utolsó, még várossal rendelkező játékos nyeri a játékot. Annak érdekében, hogy egy játék ne tartson túlságosan sokáig, megadható egy maximális kör szám is. Ilyenkor, ha ezt a számot eléri a játékosok, automatikusan véget ér a játék, a győztes pedig az lesz, akinek a legtöbb a felhalmozott fizetőeszközeinek az összege.

#### 4.1.2 Alapszabályok, funkciók

A játék körökre osztott, egy játékos egy körben a következő cselekvéseket hajthatja végre: egység létrehozása, épület építése, egységgel való lépés, támadás, technológia megtanulása. Szárazföldi egységeket a városokban, vízi egységeket a kikötőkben lehet létrehozni, egy körben egy épület csak egy egységet hozhat létre. Egy egységgel egy körben legfeljebb egyszer lehet lépni és támadni, valamint a passzív egységeket egyszer lehet használni: a telepes egy város létrehozására képes, az építész pedig egy épületet – ami nem város – tud építeni, ezt követően megszűnnek. A technológiák megtanulására nincsen körönkénti limit.

A játékos három különböző fizetőeszköz típussal rendelkezik, ezek a pénz, a nyersanyag és az élelem. Minden olyan cselekvés, amelyek új dolgot hoznak létre – vagyis az egységek kiképzése, az épületek építése, valamint a technológiák megtanulása – fizetőeszközbe kerülnek, általában mindegyik típusúba, de különböző mennyiségben. A fizetőeszközök termelését az épületek végzik.

Egy mezőn egy időben legfeljebb csak egy egység tartózkodhat, minden egység rendelkezik egy távolság értékkel, ami megadja, hogy egy körben a lépés során legfeljebb

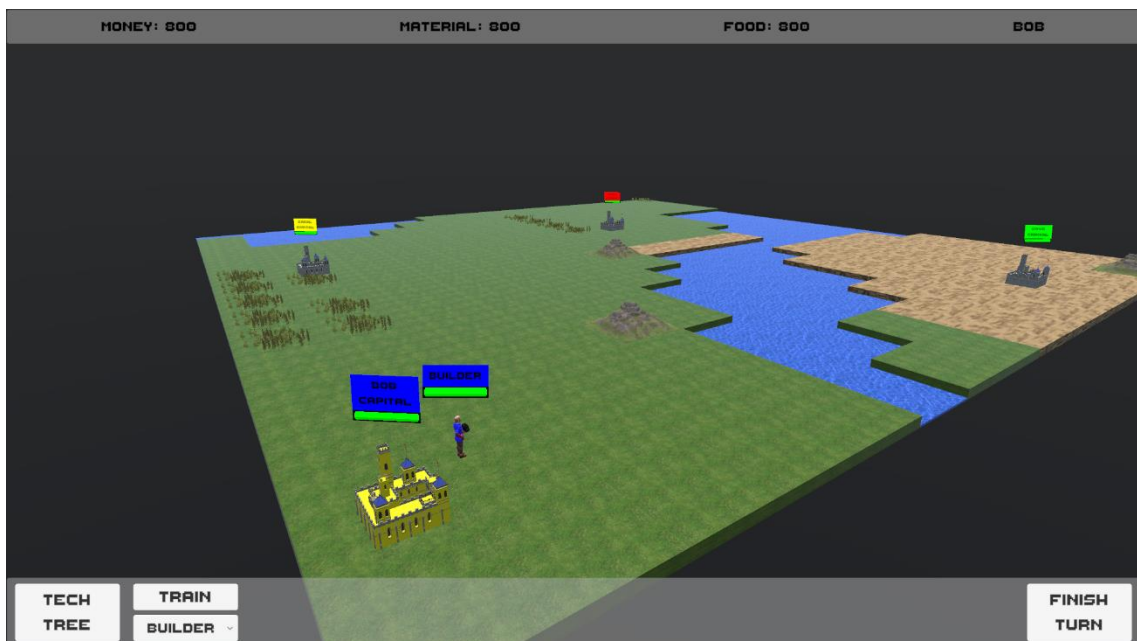
hány mezőnyi távolságot tud megtenni, valamint életpontjuk is van. A harcos egységek emellett rendelkeznek egy sebzés értékkel, és egy támadási távolság értékkel, ami a maximális támadási távolságot adja meg.

Egy mezőn legfeljebb egy épület állhat, az építész által épített épületeket csak a városok körzetébe lehet építeni, új várost pedig nem lehet a már meglévő városok körzetén belül létrehozni. Ha egy város megsemmisül, a körzetében lévő összes épület is megsemmisül.

A technológiák általában valamilyen új épületet, egységet tesznek elérhetővé, az egységek vagy az épületek tulajdonságát javítják, vagy valamilyen funkcionalitást bővítenek ki. A technológiák függésben állnak egymással, bizonyos technológiák elérhetővé tételéhez szükséges először további technológiák megtanulása.

#### 4.1.2.1 Felhasználói felület

A játék felhasználói felülete a Unity keretrendszer segítségével készült, játék közbeni megjelenítését a 2. ábra szemlélteti.



2. ábra: A játék felhasználói interfésze

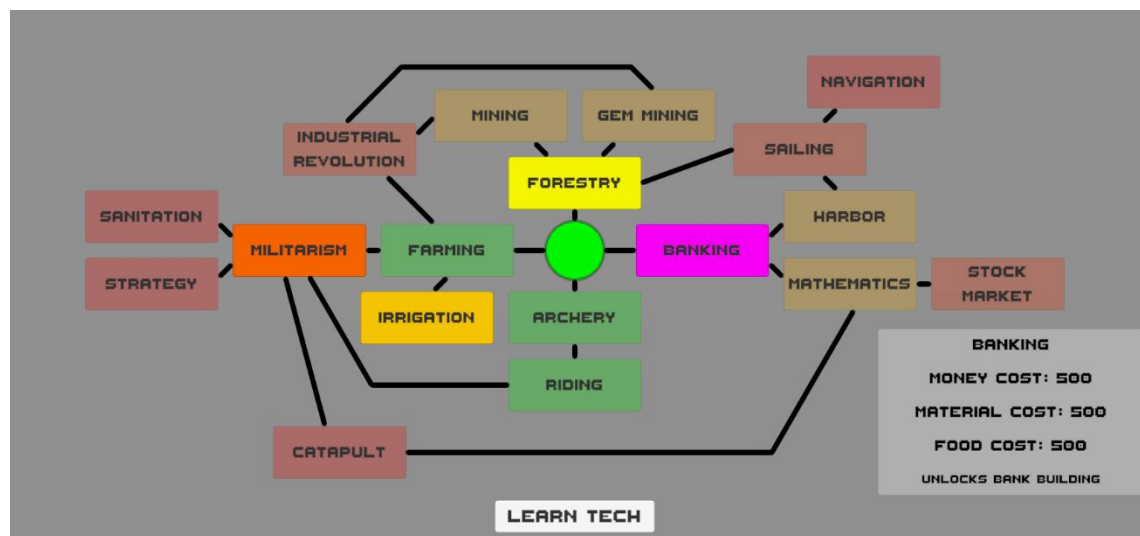
A játékban a kamerát a WASD gombok segítségével mozgathatjuk, emellett, ha lenyomva tartjuk az egér jobb gombját, akkor az egér mozgásával tudjuk elforgatni.

A felület tetején található információs sáv jobb szélén látható az éppen soron következő játékos neve, mellette pedig a fizetőeszközök típusa és az aktuálisan rendelkezésre álló mennyiségek.

Az alsó panelen található az akciógombok, amivel a játékos a parancsokat tudja kiadni. A panel bal szélén látható gombbal a technológia fát jeleníthetjük meg, ahol a technológiákat lehet megvásárolni. A panel jobb szélén található gombbal pedig továbbadhatjuk a kört a következő játékosnak.

Az egységekkel, valamint épületekkel való cselekvések végrehajtásához először a pályán kattintással ki kell választanunk azt, amivel végre szeretnénk hajtani a cselekvést. A fenti ábrán éppen egy város van kiválasztva, ezt a játék kiemeléssel jelzi. A kiválasztás hatására az alsó panelen a technológia fát megnyitó gomb mellett megjelenik az a cselekvés, amit végre tudunk hajtani. Jelen esetben a várossal egységeket képezhetünk ki, a listából kiválaszthatjuk az elérhető egységek közül a megfelelőt, majd a *Train* gombra kattintva az egység elkészül a város mezőjén.

Egység kiválasztásakor kiemelésre kerülnek azok a mezők, ahová az egységgel lépni tudunk, emellett támadó egység esetén a megtámadható egységek és épületek is. A passzív egységek, vagyis az építész és a telepes esetében pedig az alsó sávon a *Train* helyett egy *Build* feliratú gomb jelenik meg, alatta pedig az építhető épületek listája.



3. ábra: A játékhoz tartozó technológia fa

A játékhoz tartozó technológia fa megjelenítését a 3. ábra mutatja. A fában az egyes technológiák egymástól való függését a közöttük lévő vonalak határozzák meg, mindig a sötétebb színű függ a világosabb színűtől. A már megvásárolt technológiák színe

zöld, az éppen elérhetőek színe élénkebb, a nem elérhetőek halványabbak. Ha rákattintunk az egyik elérhető technológiára, akkor kiválasztásra kerül, a fában a színe lilás lesz, valamint a hozzá tartozó információk megjelennek a jobb alsó sarokban lévő panelen. A kiválasztott technológiát az alul található *Learn Tech* gomb megnyomásával fejleszthetjük ki, amennyiben rendelkezünk elég fizetőeszközzel.

#### 4.1.2.2 Konfiguráció

A játékban található egységek, épületek, technológiák tulajdonságait, valamint a pályagenerálási paramétereket a játékhoz tartozó konfigurációs fájlokban, a *PropertiesSettings.json* és a *MapGenerationSettings.json* fájlokban lehet beállítani.

#### 4.1.3 Naplózó rendszer

A játék elején, a pálya generálását követően a pálya adatai, vagyis a pályán található mezők típusa mentésre kerülnek egy JSON formátumú naplófájlba. A fájlban egy *Tiles* nevű tömb található, benne annyi tömbbel, ahány sora van a pályának, minden sor tömb pedig az adott sorban lévő mezők típusait tartalmazza, egy-egy karakterláncként.

A játék folyamán, a játékosok által végzett cselekvések is naplózásra kerülnek egy másik naplófájlba, ez a *Map* nevű tulajdonságában tartalmazza játék elején elmentett pálya útvonalát, illetve a játékosok listáját a hozzájuk tartozó kezdőpozíciókkal, ezen adatokat követik a cselekvésekből előállított naplóbejegyzések. Egy-egy ilyen bejegyzés tartalmazza a játékos nevét, az elvégzett cselekvés nevét, valamint a cselekvéshez tartozó paramétereket. A paraméterek közül mindig csak az adott cselekvéshez szükséges mezők kerülnek kitöltésre. A bejegyzések a történésük sorrendjében kerülnek mentésre, ezáltal a visszajátszó rendszer könnyedén be tudja olvasni és sorrendhelyesen meg tudja jeleníteni az eseményeket. A bejegyzések formátuma a következő:

```
{
  "Name": "Bob",
  "Action": "Train",
  "Parameters": {
    "Start": [11, 1],
    "End": [],
    "Neighbors": [],
    "Tech": "",
    "Building": "",
    "Troop": "Builder"
  }
}
```

A példában egy építész kiképzése látható, amely a *[11, 1]* koordinátájú városban jön létre, a cselekvést a *Bob* nevű játékos hajtja végre.

#### 4.1.4 Visszajátszó rendszer

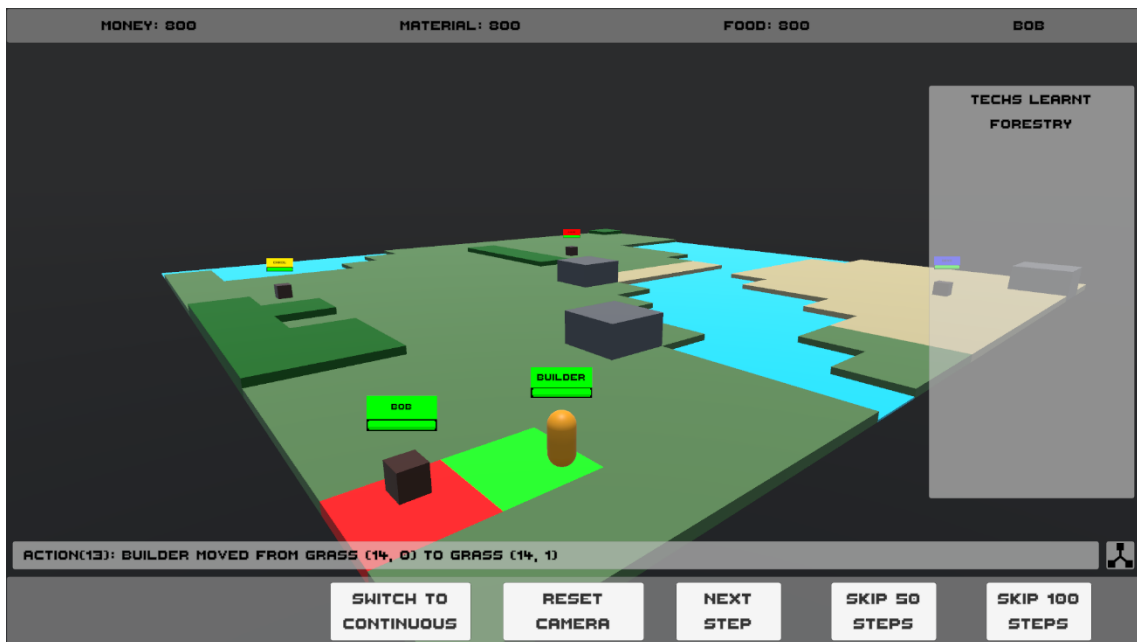
A visszajátszó rendszer *ReloadManager* osztálya betölti az indításkor megadott naplófájlt, a benne található, a pálya fájljához tartozó elérési útvonalat továbbítja a játéklógika pályakezelő, *MapManager* nevű osztályának, amely az útvonal alapján betölti és összeállítja a játékhoz szükséges pályát. A mezők logikai összekötését követően létrejönnek a megjelenített mezők is, ezek természetesen páronként megfeleltethetők egymásnak. A pálya betöltése után létrejönnek a játékosok is a naplófájlban megadott névvel, valamint a hozzájuk tartozó kezdő városok is elhelyezésre kerülnek, ezzel megtörtént a visszajátszó rendszer inicializálása.

Az inicializálás megvalósításához szükség volt minden, az alapjáték megjelenítéséhez használt osztályt – beleértve a különböző játékmechanikákat megvalósító menedzser osztályokat, illetve a játékoshoz és a játékbeli elemekhez tartozó osztályokat is – leképeznem a visszajátszó rendszerbe egy új osztály formájában. Ezek tartalmazzak átfedéseket a játékbeli osztályokkal, viszont a visszajátszáshoz szükséges viselkedéseket valósítják meg. Jó példa erre a korábban is említett *MapManager* osztály, melynek feladata az alapjáték során egy új pálya generálása, a visszajátszó rendszerbeli változata viszont egy fájlból kell betöltsse a pályát. Attól függetlenül, pálya előállítás eltérő módon történik, a mezők összekötése például ugyanúgy zajlik. Fejlesztéskor fontos volt átgondolni, hogy melyek azok a viselkedések, amelyek változnak, és melyek azok, amelyek ugyanúgy tudnak működni a visszajátszás során is.

A játék inicializálását követően a felhasználó már használhatja a grafikus felületen található gombokat, melyekkel vezérelheti a visszajátszást. A visszajátszási folyamat során az egységnyi lépés egy naplóbejegyzés visszajátszása, ezek feldolgozását a *ReplayManager* osztály végzi, amely egy listában tartalmazza az összes naplóbejegyzést, illetve számon tartja, hogy éppen hányadik lépésnél jár a visszajátszás. Ezen osztályból külön kiemelendő a *PlayAction* nevű metódus, amely egy naplóbejegyzést dolgoz fel. A függvény paraméterként egy *JsonActionObject* típusú példányban kapja meg az éppen aktuálisan következő naplóbejegyzés tartalmát. Ezen osztály feladata egy naplóbejegyzésben található adatok tárolása. A függvény a paraméterben található *Action* nevű tulajdonságának értéke alapján meghívja a hozzá tartozó, visszajátszást végző

metódust. Ezen metódusok mindegyike megfeleltethető a játékban végrehajtható egy-egy cselekvésnek. Egy ilyen metódus a meghívásakor először kiveszi a *Parameters* mezőből a szükséges értékeket, majd meghívja a játéklogika *Controller* egységének a megfelelő metódusát a tényleges végrehajtáshoz. A cselekvés végrehajtását követően a grafikus megjelenítés a változásoknak megfelelően módosul.

#### 4.1.4.1 Felhasználói felület



4. ábra: A korábbi játékállás a visszajátszó rendszerben

A visszajátszó rendszer szintén rendelkezik egy, a Unity keretrendszer segítségével megvalósított grafikus megjelenítéssel, erről mutat egy képernyőképet a 4. ábra. Szemben az alapjátékkal, a visszajátszó rendszer a játék megjelenítéséhez a sokkal egyszerűbb, fejlesztői megjelenítést használja annak érdekében, hogy a visszajátszás során a tartalmi, nem vizuális dolgokra lehessen koncentrálni.

A visszajátszó rendszerben a kamera mozgatása megegyezik az alapjátékban használt irányítással, emellett a felhasználói felületen található felső panelen is ugyanazon információkat láthatjuk.

Az alsó panelen viszont más gombok kaptak helyet, középen a kamera alaphelyzetbe állító gombja, tőle balra a visszajátszás módját változtató gomb, jobbra pedig az a három, melyek segítségével léptethetjük a visszajátszást. Diszkrét visszajátszási módban a gombok a rajtuk látható feliratnak megfelelő mennyiségű cselekvéssel léptetik előre a játékot. Folytonos módba kapcsolva a gombok viselkedése

(illetve a rajtuk látható felirat is) megváltozik, ebben az esetben – balról jobbra haladva – a következő feladatokat látják el: léptetés szüneteltetése (*Pause*), automatikus léptetés (*Play*), gyors automatikus léptetés (*Fast forward*). Ezen viselkedés implementációja során felhasználtam két, a Unity keretrendszerben található metódust, melyekkel meg tudtam valósítani az időközönkénti automatikus léptetést:

```
InvokeRepeating(string methodName, float time, float repeatRate);  
CancelInvoke();
```

Az első metódus az *InvokeRepeating*, amely a megadott időközönként meghívja az első paraméterben kapott metódust. Az implementáció során ennek a függvénynek átadtam a visszajátszó léptetését végző metódust, a kezdeti késleltetést nullára, a harmadik paramétert pedig a visszajátszásnak megfelelő sebességre állítottam, ami a *Play* esetében 1, a *Fast Forward* esetében pedig 0.2 másodperc. A másik, *CancelInvoke* nevű metódus pedig az *InvokeRepeating* által végzett folyamatos ismétlés leállítására használható, a *Pause* gombra kattintva ennek hatására áll meg a visszajátszó.

Az éppen aktuálisan bemutatott lépés részletei az alsó panel fölött, egy információs sávban jelennek meg, valamint a cselekvés során használt mezők kiemelésre kerülnek a játéktéren. Ennek működését a cselekvésekhez tartozó metódusok végzik, a *ReplayManager* osztályban található *actionText* mező *text* tulajdonságának beállításával, a mezők színezését pedig a *HighlightManager* osztály *Add* metódusa végzi, amely paraméterül megkapja a kiszínezni kívánt mező koordinátáit, illetve a színt is. A kiszínezett mezők később a *Clear* metódus hatására kerülnek eredeti állapotukba. A képen például azt láthatjuk, ahogy az építész egység a pirosan kiemelt mezőről átlép a zölden kiemelt mezőre.

Az információs sáv mellett jobbra található még egy gomb, ezzel nyithatjuk meg és zárhatjuk be a kép jobb oldalán látható függőleges panelt, amiben az aktuális játékos által megtanult technológiák nevei láthatók. A gombra kattintva lekérdezésre kerülnek a játékoshoz tartozó technológiák, majd ezek neve felkerül a panelre, ami ezt követően megjelenik.

## 4.2 Automatizáló rendszer

Az automatizáló rendszer feladata, hogy lebonyolítsa a weboldalon elindított versenyt a felületen beállított konfigurációkkal, eközben pedig tájékoztassa a felhasználót a verseny állapotáról.



## 4.2.1 Működése

A rendszer első eleme a verseny konfiguráló weboldalhoz tartozó szolgáltatás, a *TournamentService* osztály, melynek a *StartTournament* metódusa kerül meghívásra verseny indításakor.

```
public async Task StartTournament(string map, List<string> zipFileNames,
    TournamentType tournamentType, int roundCount, int maxTurnsInMatch)
{
    TournamentResult tournamentResult = new()
    {
        DateTime = DateTime.Now,
        MapFileName = map,
        TournamentType = tournamentType.ToString(),
        Finished = false
    };
    _dataContext.TournamentResults.Add(tournamentResult);
    _dataContext.SaveChanges();

    OnTournamentStarted?.Invoke(zipFileNames.Count);
    CopyFilesToNetwork(zipFileNames);

    List<string> clientIDs =
        zipFileNames.Select(z => z.Split('.')[0]).ToList();
    Dictionary<string, int> results = new();

    if(tournamentType == TournamentType.League)
    {
        results = await RunLeagueTournament(map, clientIDs,
            roundCount, maxTurnsInMatch);
    }
    else
    {
        results = await RunKnockoutTournament(map, clientIDs,
            maxTurnsInMatch);
    }

    RemoveFilesFromNetwork(zipFileNames);
    await Network.Client.ClientManager.RemoveImages(clientIDs);

    tournamentResult.Finished = true;
    _dataContext.TournamentResults.Update(tournamentResult);
    foreach(var result in results)
    {
        _dataContext.ResultItems.Add(new ResultItem
        {
            FileName = result.Key,
            Value = result.Value,
            TournamentResultId = tournamentResult.Id
        });
    }
    _dataContext.SaveChanges();
    OnTournamentCompleted?.Invoke(results, tournamentType);
}
```

A függvény paraméterként megkapja a konfigurációhoz szükséges adatokat, vagyis a pályát, a kiválasztott mesterséges intelligenciák nevét listaként, és a verseny típusát. A maradék két paraméter a *roundCount* és a *maxTurnsInMatch*, előbbi a fordulók száma, azaz, hogy hány játszma legyen két kliens között, utóbbi pedig egy adott játszma hosszát adja meg, körökben mérve. (A *roundCount* paraméternek csak a *League* típusú versenyeknél van jelentősége, a *Knockout* típusú versenyek esetén mindig egy mérkőzést játszanak a játékosok egy párosítás során.)

Először a metódus létrehozza és adatbázisba elmenti a versenyről tárolni kívánt adatokat, ez tartalmazza a verseny kezdési idejét, a pálya nevét, a verseny típusát, valamint egyelőre a befejezettségét hamisra állítja. A mentést követően az *OnTournamentStarted* eseménnyel jelzi a weboldalnak a verseny indulását, átadva a résztvevők számát is.

A verseny kezdésekor a *CopyFilesToNetwork* metódus segítségével átmásolja a kliensek tömörített fájljait a játékszervert megfelelő könyvtárába, ezt követően a fájlnevekből előállítja a játékosok neveit, majd a verseny típusa alapján aszinkron meghívja a megfelelő, mérkőzéseket lebonyolító metódust. Amikor a verseny összes mérkőzése véget ért, akkor a függvény befejezi futását és visszatér az eredményekkel, melyek tartalmazzák minden játékos nevét és az általa elért eredményt.

Az eredmények megérkezését követően törli a verseny elején átmásolt fájlokat a szerverről, valamint megsemmisítésre kerülnek a verseny során létrejövő, a kliensekhez tartozó Docker image fájlok is. Az image fájlokat azért nem törli minden lejátszott mérkőzés után – szemben a konténerekkel – mert az image fájlok újra felhasználhatóak konténer készítésére abban az esetben, ha egy kliens több mérkőzésben is részt vesz. Az image fájlok megtartásával így erőforrást, valamint időt is spórol a rendszer.

Ezt követően ér véget a verseny, a metódus a versenyeredmény objektumban igazra állítja a verseny befejezettségét, rögzíti az adatbázisban a játékosokhoz tartozó eredményeket, hogy az eredmények később is megtekinthetők legyenek, majd menti az adatbázismódosításokat. Utolsó lépésként az *OnTournamentCompleted* esemény segítségével értesíti a weboldalt a verseny befejeződéséről, paraméterként pedig átadja az eredményeket, illetve a verseny típusát.

A verseny során a mérkőzéseket a verseny típusától függően a *RunLeagueTournament*, illetve a *RunKnockoutTournament* metódusok bonyolítják le.

Előbbi a *League* típusú versenyeket vezényli le, ilyenkor minden játékos annyi mérkőzést játszik mindenkivel, amennyi a paraméterként kapott *roundCount* értéke. A metódus lényegi működését az alábbi sorok végzik:

```
List<string> lobby = new() { clientIDs[i], clientIDs[j] };
OnMatchStarted?.Invoke(lobby);
string matchWinner = await RunGame(map, maxTurnsInMatch, lobby.ToArray());
points[matchWinner] += 1;
OnMatchCompleted?.Invoke(matchWinner);
OnPointsUpdated?.Invoke(points, TournamentType.League);
```

Először kiválasztásra kerül a soron következő két játékos neve, akik játszani fognak egymás ellen, ezt követően a mérkőzés kezdetéről az *OnMatchStarted* eseménnyel értesíti a felületet a szolgáltatás. A játék a *RunGame* metódusban zajlik le, melynek visszatérési értéke a játszma nyertese. Ebben a játékmódban minden győzelem 1 pontot ér, ez bekerül az eredményeket tartalmazó tárolóba, ezt követően a mérkőzés végéről az *OnMatchCompleted* esemény értesíti a weboldalt, átadva a nyertes játékos nevét. A pontszámok változásáról pedig az *OnPointsUpdated* eseményen keresztül értesül az oldal.

A weboldal a szolgáltatásban található eseményekre való feliratkozásának köszönhetően mindig láthatja a felhasználó, hogy az éppen folyamatban lévő mérkőzésen mely játékosok vesznek részt, mik voltak a korábban befejeződött játszmák eredményei, illetve mi a verseny aktuális állása.

A *Knockout* típusú, vagyis az egyenes kieséses rendszert megvalósító *RunKnockoutTournament* metódus is hasonlóan működik, ott viszont pontok helyett helyezéseket érnek el a játékosok, melyek a játékosok száma alapján kerülnek kiszámításra, ezért is fontos, hogy az *OnPointsUpdated* eseményben átadásra kerüljön a verseny típusa is, mivel az eredmények megjelenítése típustól függően eltérő lehet a weboldalon.

Egy játék tényleges menete a *RunGame* metódusban zajlik le, ennek lényegi implementációja a következő:

```

private async Task<string> RunGame(
    string map, int maxTurnsInMatch, string[] clients)
{
    string winner = "";

    ...

    ProcessStartInfo startInfo = new()
    {
        FileName = exePath,
        Arguments = $"{map} {maxTurnsInMatch} {string.Join(" ", clients)}",

        ...
    };

    using (Process process = new())
    {
        process.StartInfo = startInfo;

        ...

        process.Start();

        ...

        await process.WaitForExitAsync();
    }
    return winner;
}

```

Ez a metódus megkapja a pályát, a játékban játszható maximális körök számát, valamint a játékosok tömbjét. Ezeket a létrehozott *ProcessStartInfo* típusú objektum *Arguments* tulajdonságában adja át. Ezt követően létrehoz egy *Process* típusú objektumot, ami maga a játék lesz, ennek beállítja, hogy a korábban összeállított objektumot használja fel indulása során. A megadott adatok alapján elindul a játékot futtató szerver, ezt követően megkezdődik a kliensek csatlakoztatása, ezt a *ClientManager* osztály *StartClients* metódusa végzi.

```

public static async Task StartClients(
    string serverAddress, List<string> clients)
{
    ...

    _clients = clients;
    for(int i = 0; i < _clients.Count; i++)
    {
        Unzip(_clients[i]);
        CreateDockerContainer(_clients[i], serverAddress, defaultPort + i);
    }
    await StartContainers();
}

```

A kliensek egy-egy tömörített fájlban találhatók meg, amelyek a verseny indítása előtt kerültek átmásolásra. A tömörített fájlok kicsomagolásra kerülnek az *Unzip* módszer segítségével, ezt követően a *CreateDockerContainer* függvény létrehozza a könyvtárban található Dockerfile alapján először a kliensek Docker image fájlját, az image fájlokból pedig létrehozza a konténereket, valamint a *StartContainers* módszer el is indítja azokat. A kliensek az elindulásukkor automatikusan csatlakoznak a játék szerverhez. A csatlakozásokat követően végbemegy a játék, melynek befejeződését a *RunGame* módszer aszinkron módon várja meg, hogy ezt követően visszatérhessen a nyertes nevével.

#### 4.2.1.1 Hálózati kommunikáció

Az automatizáló rendszerben található hálózati kommunikációs interfész teremti meg az összeköttetést a szerver és a kliensek között. Ennek segítségével tudja a szerver fogadni a kliensek csatlakozási kéréseit, a csatlakozásokat követően ezen keresztül történik az adatátvitel, a játék befejeztével pedig az interfész lecsatlakoztatja a klienseket a szerverről.

Ha egy kliens csatlakozik a játékhoz, a szerver elküldi neki a játék beállításait, ami a két konfigurációs fájlban lévő adatokat tartalmazza. Ha elegendő kliens csatlakozott, a szerver elindítja a játékot. A soron következő kliens mindig egy *StartTurn* üzenetet kap, innen tudja, hogy ő következik. Ezt követően a kliens már lekérdezheti a játék állapotát és elküldheti az általa végrehajtani kívánt lépéseket.

A játékban végrehajtható cselekvésekhez tartozik egy JSON formátumú adatmodell, melynek paramétereit megfelelően kitöltve tudja kiválasztani a kliens a végrehajtandó lépést. Az adatmodell formátuma az alábbi:

```
{
  "Name": ...játékos neve...,
  "Action": ...parancs neve...,
  "Parameters": {
    "Start": ...parancs kezdő koordinátái...,
    "End": ...parancs végső koordinátái...,
    "Troop": ...egység típusa...,
    "Building": ...épület típusa...,
    "Tech": ...technológia típusa...
  }
}
```

Az adatmodellben a név (*Name*) és a parancs (*Action*) mezők értékének kitöltése kötelező, a paraméterek (*Parameters*) közül viszont csak azt szükséges megadni, ami a

cselekvéshez szükséges. A parancs mező értéke a végrehajtható cselekvések alapján alábbiak lehetnek:

Parancs neve	Leírás	Szükséges paraméterek
<i>AttackBuilding</i>	épület megtámadása	<i>Start, End</i>
<i>AttackTroop</i>	egység megtámadása	<i>Start, End</i>
<i>Build</i>	épület építése	<i>Start, Building</i>
<i>EndTurn</i>	kör befejezése	-
<i>GetActions</i>	elérhető cselekvések lekérdezése	-
<i>GetGameState</i>	játék teljes állapotának lekérdezése	-
<i>Learn</i>	technológia megtanulása	<i>Tech</i>
<i>Move</i>	egységgel való lépés	<i>Start, End</i>
<i>Train</i>	egység kiképzése	<i>Start, Troop</i>

1. táblázat: A parancsok listája

## 4.2.2 Mesterséges intelligencia fejlesztése

A projekt legfőbb célja, hogy a játékhoz saját mesterséges intelligenciákat lehessen fejleszteni, amelyek egymás ellen mérhetik össze tudásukat.

### 4.2.2.1 Játék stratégiák

A játékhoz írt mesterséges intelligenciák különböző stratégiákat követhetnek a győzelem megszerzése érdekében. A játékot alapvetően az nyeri, akinek sikerül megvédenie a birodalmát, tehát marad legalább egy városa, viszont a maximális kör limitet is figyelembe kell venni, mert olyankor a felhalmozott fizetőeszközök összege számít.

Egy lehetséges megközelítés lehet a támadó típusú stratégia, amikor a játékos igyekszik minél rövidebb idő alatt sok egységet kiképezni, amikkel gyorsan lerohanhatja az ellenfeleket. Itt nem számít, hogy mennyire fejlettek az egységek, az a fontos, hogy minél több legyen belőlük, ezáltal az ellenfél nem tudja mindet megsemmisíteni. A stratégia hátránya lehet, hogy kevés épülettel, várossal rendelkezik, ezáltal kevésbé stabil a birodalom.

Egy másik stratégia lehet a megfontolt, eleinte védekező típusú, ilyenkor a játékos kezdetben igyekszik kerülni a kisebb-nagyobb harcokat, csak a védekezéshez elegendő egységet tart fent, a fizetőeszközök nagy részét épületek építésére, technológiák

kifejlesztésére használja fel, hogy utána egy fejlett sereget tudjon kiképezni. A stratégia hátránya, hogy a játék elején még nem rendelkezik nagy sereggel, ez a korai harcoknál problémákat okozhat.

Szintén egy működő stratégia lehet, ha a játékos igyekszik a pálya több, egymástól távolabbi részén is városokat építeni, ezáltal, ha megtámadják, esetleg meg is semmisítik az egyik városát, akkor van ideje a többi, távolabb lévő várost felkészíteni a harcokra, mert idő kell az ellenséges egységeknek, mire odaérnek. Támadáskor is előnyös lehet ez a megközelítés, hiszen több irányból tudja közrefogni az ellenfél egységeit, épületeit. A stratégia hátránya viszont az, hogy ha távol létrehoz egy új várost, akkor az eleinte védtelen, ilyenkor gyorsan meg lehet semmisíteni.

A fizetőeszközök felhalmozására építő stratégia is működőképes lehet, viszont ebben az esetben sokat számít a maximális körök számának értéke, mert kevés körszám esetén könnyebb kivédekezni a játékot, mint a hosszan elhúzódó játszmák esetén, ahol már jelentős hátrányba kerülhet a játékos a kevés harcos egység miatt.

Azt, hogy mi a játékban az ideális stratégia, azt a maximális körszám mellett még sok más tényező is befolyásolja, többek között a pálya, a játékosok kezdőpozíciója, illetve az egységek, épületek, technológiák ára, és egyéb tulajdonságai, melyeket a konfigurációs fájlban lehet módosítani. A felsorolt stratégiák csupán elméleti szintűek, főként mesterséges intelligencia készítéséhez adhatnak kiindulási alapot, a projekt elkészítése során nem volt feladatom kipróbálni különböző stratégiákat, csak az alkalmazás teszteléséhez volt szükségem egy egyszerű logika elkészítésére.

#### **4.2.2.2 Konténerizálás**

Az automatizáló rendszer számára fontos, hogy egységesen tudja kezelni az implementált mesterséges intelligenciákat, a rendszernek nem feladata, hogy különböző implementációk esetén máshogy viselkedjen. Az egységesítés érdekében a mesterséges intelligenciák fejlesztése során szükséges a betartani projekt által definiált szabályokat.

A tömörített mappa gyökerében kell szerepeljen a Dockerfile, amiben szerepel két környezeti változó, egyik a szerver elérési címe, a másik a játékos neve. Ezeket tetszőlegesen inicializálhatjuk, mert a játék indítása során az automatizáló rendszer fog nekik értéket adni. Ezen változók mellett még szükséges felvenni a szerver felé irányuló kommunikációra használt port számát is. A mesterséges intelligencia tényleges implementációja a Dockerfile mellett tetszőleges könyvtárstruktúrában megadható.

#### 4.2.2.3 Hálózati kommunikáció

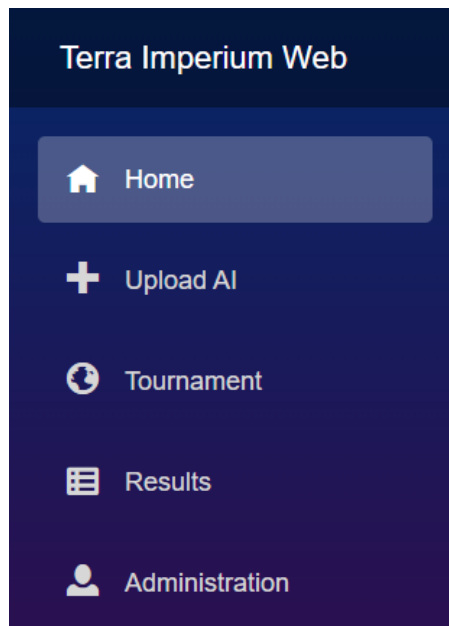
A mesterséges intelligenciát fel kell készíteni a szerverrel való kommunikációra, ehhez implementálni kell a websocket kommunikációhoz szükséges logikát. Az általam készített mesterséges intelligencia kliens az indítást követően elküldi a csatlakozási kérelmet a szerver felé, ha sikeres volt a csatlakozás, akkor a szervertől egy *Setup* típusú üzenetben megkapja a játék kiindulási állapotát, válaszul pedig elküldi a nevét a szervernek, amivel regisztrálásra kerül a játékba. Ezt követően a kliens várakozik, ameddig meg nem kapja a *StartTurn* üzenetet. Ekkor lekérdezi a játékban általa elvégezhető cselekvéseket, majd kiválaszt közülük egyet, ezzel felparaméterezi a választát, elküldi a szervernek, majd újra lekérdezi az immáron módosult állapotot, és újra választ egy lépést. Ezt addig ismétli, ameddig van elérhető lépés, ha már nincs, akkor kiadja a kör befejezését jelző *EndTurn* parancsot, a játék pedig átvált a soron következő játékosra.

A játékhoz elkészíteni kívánt klienseknek ezt a működést kell megvalósítaniuk, a különbségek azon algoritmusokban lesznek, melyek eldöntik, hogy melyik lépés a legjobb választás a kliens számára.

### 4.3 Weboldal

A weboldalon megvalósított funkciók segítségével a felhasználók feltölthetik az általuk implementált mesterséges intelligenciákat, a feltöltött fájlokkal versenyeket tudnak indítani, valamint a végbement versenyek eredményeit megtekinthetik egy összegző oldalon, emellett felhasználókezelést is biztosítanak az adminisztrátorok számára. A felhasználói felület öt különálló oldalból áll, ezek a kezdőoldal, a feltöltés, a verseny indító, az eredmények, valamint a felhasználókezelő oldalak, a teljes menüről készült képet tartalmazza az 5. ábra.





5. ábra: A weboldal teljes menüje

#### 4.3.1 Felhasználókezelés

A weboldal használatakor három különböző felhasználó típust különböztetünk meg. Az első a vendég felhasználó, aki csak a kezdőoldalt tudja megtekinteni, ezáltal megismerkedhet a projekttel.

Ha a felhasználó regisztrál egy saját fiókot, majd bejelentkezik, akkor a második típusba kerül, ami a bejelentkezett felhasználó. A bejelentkezett felhasználók már elérik a mesterséges intelligencia feltöltő oldalt, ahol tetszőleges számú fájlt feltölthetnek, valamint megtekinthetik a versenyek eredményeit listázó oldalt is.

A harmadik típus az adminisztrátori jogokkal rendelkező felhasználó, akinek lehetősége van versenyeket konfigurálni, valamint elindítani, emellett pedig kezelheti az oldalra regisztrált felhasználókat. Biztonsági okokból egy új felhasználó a regisztrációt követően nem kerül automatikusan bejelentkeztetésre, valamint nem is tud bejelentkezni. Ahhoz, hogy használni tudja az oldalt, szükség van arra, hogy egy adminisztrátor elfogadja a regisztrációs kérelmét, ezáltal aktiválja a fiókját, csak ezt követően fog tudni bejelentkezni és megtekinteni a különböző oldalakat. Az adminisztrátori jogok közé tartozik még – a regisztrációs kérelmek elfogadása, illetve elutasítása mellett – a fiókok törlése, valamint az adminisztrátori jogosultságok kezelése.

A felhasználókezelést végző, vagyis a bejelentkezés, regisztráció, profil megtekintése, valamint a jelszóváltogatás oldalak az ASP.NET Core Identity technológia

által generált felhasználói felülettel rendelkeznek, melyeken kisebb módosításokat végeztem, hogy illeszkedjenek az általam használt adatstruktúrához.

### 4.3.2 Kezdőoldal

A weboldal kezdőoldalán egy, a projektről szóló összefoglaló, valamint a projekthez kapcsolódó fontosabb tudnivalók, hivatkozások találhatók.

### 4.3.3 Mesterséges intelligencia feltöltése

Ezen az oldalon tudják a felhasználók feltölteni az általuk készített mesterséges intelligenciák tömörített fájljait. A feltöltő oldal tartalmát a 6. ábra jeleníti meg.

6. ábra: A feltöltő oldal

Új fájlt a bal oldali oszlopban lehet kiválasztani és beküldeni, fontos, hogy csak a .zip kiterjesztésű fájlokat fogadja el a rendszer, rossz fájlformátum vagy túl nagy fájl méret esetén hibaüzenet jelenik meg. A sikeres beadást szintén egy üzenet jelzi, valamint a jobb oldali oszlopban, a korábbi beadások listájában is megjelenik a feltöltött fájl feltöltéskori neve, valamint zárójelben a rendszer által generált neve. A szerver fájlrendszerén, valamint az adatbázisban a fájl a generált névvel fog szerepelni, így elkerülve az esetleges fájl név ütközéseket. A generált név a felhasználó által regisztrációkor megadott Neptun kódból és a beadás pillanatában generált időbélyegből jön létre. Egy sikeres fájl feltöltést szemléltet a 7. ábra.

7. ábra: Sikeres fájl feltöltés

### 4.3.4 Versenyek lebonyolítása

Versenyeket indítani csak adminisztrátorok tudnak, ezáltal felügyelten történik a szerver erőforrásainak használata, elkerülhető a szerver túlzott terhelése. A lebonyolításhoz két oldal tartozik a felületen, az egyik a *Tournament*, ahol a verseny indítása és követése zajlik, a másik a *Results*, ahol a már végbement versenyek eredményeit lehet megtekinteni.

#### 4.3.4.1 Verseny indító felület

A *Tournament* oldalra navigálva a következő felület jelenik meg, itt tudjuk beállítani a verseny indítási paramétereit, ezt a megjelenítést mutatja a 8. ábra.

8. ábra: A verseny indító oldal

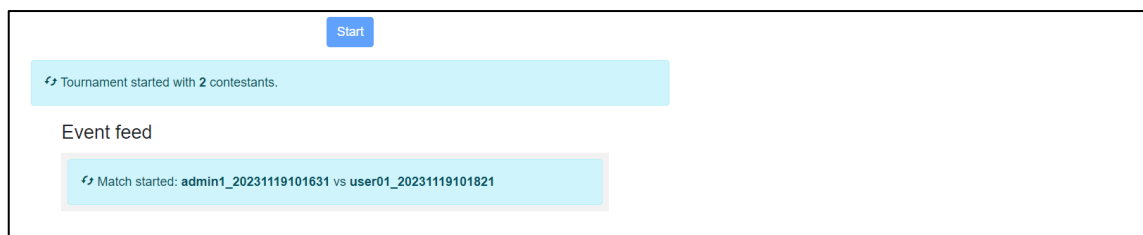
A bal felül látható két listából tudjuk kiválasztani a játékban használni kívánt mesterséges intelligenciák fájljait, illetve a pályát, amin a verseny zajlani fog. Egy pálya kiválasztásakor megjelenik annak előnézete a jobb oldalon látható keretben.

A fájlok kiválasztása alatt meg kell adnunk a verseny típusát. Ez jelenleg kétféle lehet, ez első a bajnokság mód (*League mode*), ennek lényege, hogy a kiválasztott játékosok közül mindenki mindenkivel annyi mérkőzést játszik, ahányat beállítunk a módhoz tartozó plusz beállításként. A másik az egyenes kiesés mód, (*Knockout mode*), ebben a játékosok párokba sorsolva játszanak egy-egy mérkőzést, minden párból a győztes jut tovább, a vesztes kiesik. A győzteseket ezután újra párba sorsolja a rendszer,

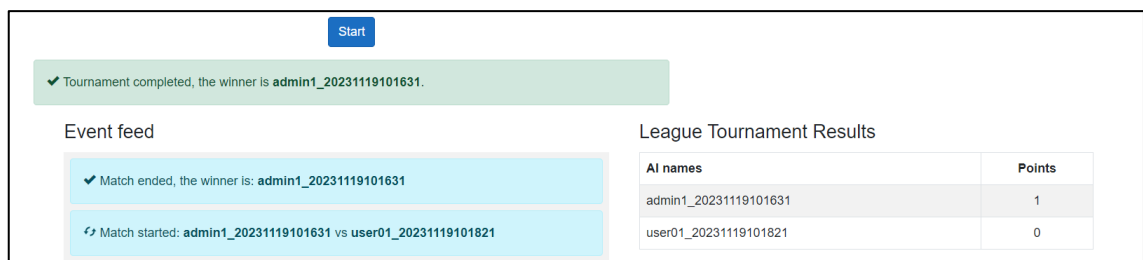
majd ismét játszanak. Ez a folyamat addig tart, ameddig már csak két játékos van versenyben, közülük kerül ki a nyertes az utolsó mérkőzésen.

Az utolsó beállítás, amit még szükséges megadnunk, hogy mennyi legyen a maximális körök száma egy mérkőzés során. Erre azért van szükség, hogy limitálni tudjuk a játszmák hosszát, egy-egy játék ne tartson túlságosan sokáig abban az esetben, ha nagyon kiegyenlített a mérkőzés állása.

Ha minden beállítást megadtunk, akkor a *Start* gomb megnyomásával elindíthatjuk a versenyt. A sikeres indítást egy üzenet jelzi, majd az oldalon lentebb görgetve megjelenik az eseménynapló, ahol a mérkőzéseket tudjuk követni. Ha egy mérkőzés indul, vagy egy futó mérkőzés befejeződik, akkor ezekről létrejön egy-egy bejegyzés az eseménynaplóban, valamint az frissül eseménynapótól jobbra található, az aktuális eredményeket megjelenítő táblázat is. A verseny végét szintén egy bejegyzés jelzi, valamint a végső eredmények mentésre kerülnek az adatbázisba. A verseny indítását követő állapotot a 9. ábra, a verseny végét mutató állapotot pedig a 10. ábra szemlélteti.



9. ábra: Folyamatban lévő verseny



10. ábra: Befejeződött verseny

#### 4.3.4.2 Verseny indító implementációja

A weboldalra való navigáció esetén az oldal betöltésekor először a beépített *OnInitialized* metódus kerül automatikusan meghívásra.

```
protected override void OnInitialized()
{
    TournamentService.OnTournamentStarted += TournamentStarted;
    TournamentService.OnTournamentCompleted += ShowResults;
    TournamentService.OnMatchStarted += MatchStarted;
    TournamentService.OnMatchCompleted += MatchCompleted;
    TournamentService.OnPointsUpdated += UpdateScores;
    GetZipFiles();
    GetMapFiles();
}
```

Ezen függvényen belül történik meg a *TournamentService* szolgáltatás által publikált eseményekre való feliratkozás a hozzájuk implementált eseménykezelő metódusokkal, melyek az oldalon található adatokat frissítik a verseny alatt történő változások alapján. Ezen kívül lekérdezésre kerülnek a szerver fájlrendszerén található tömörített fájlok és pálya fájlok nevei is.

Az oldal lényegi működését a *StartTournament* metódus végzi, amely a *Start* gombra való kattintáskor kerül meghívásra.

```
private async Task StartTournament()
{
    ...

    if(validator.Validate(selectedMap, selectedZipFiles, tournamentType,
        roundCount, maxRoundCount, maxTurnsInMatch))
    {
        ...



        await TournamentService.StartTournament(
            selectedMap!.Name, selectedZipFiles,
            tournamentType ?? TournamentType.League,
            roundCount, maxTurnsInMatch);
    }
}
```

Ebben a metódusban történik meg az oldalon kiválasztott adatok validációja, az általam implementált érvényesség-ellenőrző osztály segítségével, amely ellentmondás esetén nem engedi elindítani a versenyt, ilyenkor egy hibaüzenet jelenik meg az oldalon, amelyben a felhasználó megtudhatja, miért nem sikerült elindítani a versenyt. A validátor többek között olyan szempontokat vesz figyelembe, mint hogy van-e kiválasztva pálya, verseny típus, illetve legalább két kliens. Amennyiben minden feltételnek megfelelnek a kiválasztott adatok, akkor az oldal meghívja a *TournamentService* szolgáltatás

*StartTournament* metódusát a megadott paraméterekkel, innentől pedig az automatizáló rendszerben leírtak alapján lebonyolításra kerül a verseny.

#### 4.3.4.3 Eredmények

A végbement versenyek eredményeit a *Results* oldalon tekinthetjük meg, ennek kinézetét a 11. ábra mutatja, az ábrán két korábbi verseny eredménye látható. Az eredmények versenyenként egy külön táblázatban jelennek meg, a táblázatok fejlécében látható a verseny típusa és időpontja. A listában a táblázatok időrendi sorrendben vannak megjelenítve, hogy az oldal tetején mindig a legújabb eredmények legyenek. Adminisztrátorként törölni is tudjuk a mentett eredményeket a táblázat tetején található törlés gombbal.

Tournament results		
Tournament type: League	Date: 2023. 11. 19. 14:39:19	
AI names		Points
user01_20231119101821		1
admin1_20231119101631		0
Tournament type: League	Date: 2023. 11. 19. 14:18:25	
AI names		Points
admin1_20231119101631		1
user01_20231119101821		0

11. ábra: Eredmények nézet

## 5 Az implementáció és a tesztelés folyamata

A projekt elején a visszajátszó rendszer elkészítésével kezdtem a fejlesztést, ehhez először kialakítottam a felhasználói felületet a Unity keretrendszer segítségével. A megjelenítés alapját a játékhoz tartozó felület adta, ezt módosítottam úgy, hogy megfeleljen a visszajátszáshoz szükséges feltételeknek. A felület elkészítése után következett a visszajátszó logika implementálása. Ehhez először minden, a játékban megtalálható osztályhoz készítettem egy-egy leképezést egy új osztály formájában, amely az eredeti osztály módosított, visszajátszáskori viselkedését valósítja meg. Ezt követően implementáltam a naplófájl visszaolvasó logikát, ehhez felhasználtam a naplózó rendszerben található adattároló osztályokat, majd a tényleges visszajátszó logika következett. Ennek működéséhez szükség volt minden, a játékban elérhető cselekvéshez implementálni egy-egy metódust, amelyek elvégzik a játéklogikában azokat a módosításokat, melyeket a játék közben az adott cselekvés végrehajtana. Végül összekötöttem a logikát a felhasználói felülettel, ezt követően pedig teszteltem a visszajátszó működését.

A visszajátszó rendszer megvalósítását követően implementáltam egy egyszerű mesterséges intelligenciát, amely megvalósította az alap elvárt kliens működést, ennek segítségével már a hálózaton keresztül is tudtam játzmákat szimulálni. Ekkor még manuálisan indítottam el a játékhoz tartozó szerveret, és csatlakoztattam hozzá a klienseket, mert nem tartalmazott automatikus kliens kezelést.

A hálózaton a kliensek által lejátszott mérkőzésekből készült naplófájlok visszajátszása során viszont több hiba is fellépett az alkalmazásban. Ezen, a játéklogikában és a hálózati kommunikáció során felmerülő hibák, nem elvárt viselkedések kijavításával folytattam a fejlesztést. Az implementált kliens segítségével könnyebbé vált a hibakeresés, illetve egyszerre tudtam tesztelni a játéklogikát, valamint a hálózati kommunikációs interfészt is. A játéklogikában lévő hibákat egyszerűbb volt megtalálni és kijavítani, általában valamilyen feltétel vizsgálata maradt el, vagy volt helytelen. A hálózati kommunikációs hibákat főként a kliensek konkurens viselkedése okozta, ehhez szükséges volt az erőforrásokon zárat alkalmazni, hogy egyszerre csak egy kliens férjen hozzájuk. A konkurencia mellett előfordultak olyan hibák is, melyeket

a parancsok végrehajtási sorrendje okozott, ezeket kifejezetten nehéz volt felderíteni, kijavításukhoz viszont csak a függvényhívások sorrendjét kellett módosítani.

A hibák kijavítását követően tudtam elkezdeni az automatizáló rendszer megvalósítását. Ehhez először megterveztem a kliensek kezelésének lépéseit, hogy hogyan lesz egy tömörített fájlból Docker konténer. Az automatizáláshoz a klienst is módosítanom kellett, hogy az újonnan hozzáadott a Dockerfile állományból vegye ki a környezeti változókat. Ezt követően implementáltam a klienskezelés lépéseit, melyek az állomány kitömörítése, az image, valamint a konténer létrehozása voltak. Az implementáció után már beépíthettem ezt az egységet is a hálózatkézelő komponensbe, ezt követően teszteltem a konténerizáció helyes működését. A klienskezelés megvalósítását követően ideiglenesen létrehoztam egy verseny kezelő logikát, amely segítségével tesztelni tudtam több játszmából álló versenyeket is. Ezen logikának egy kibővített változatát építettem be később a weboldalhoz tartozó verseny kezelő szolgáltatásba.

Az automatizáló rendszer implementálását követően már csak a webservert elkészítése volt hátra. Az implementáció során weboldalanként haladtam, minden oldalhoz először a felületet, majd a háttérlogikát készítettem el. A verseny konfiguráló oldalhoz implementáltam a versenyek lebonyolításáért felelő háttérszolgáltatást, melynek feladata a verseny előrehaladtával a felület folyamatos értesítése és a részeredmények elküldése is. Az oldalak megvalósítása után hozzáadtam a felhasználókezelést is a projekthez, ehhez létrehoztam az adatbázist, a statikus adatokat pedig lecseréltem az adatbázisból érkező adatokra. Ezt követően implementáltam a jogosultság kezelést, valamint kiegészítettem az alkalmazást egy újabb oldallal, ami lehetővé tette az adminisztrátorok számára a felhasználók felületről való kezelését.

A projekt végén manuálisan teszteltem a weboldal különböző funkcióit, többek között a verseny lebonyolítását is, ahol igyekeztem minél többféle esetet kipróbálni. A weboldal tesztelése mellett készítettem egy-egy menüt az alapjátékhoz, illetve a visszajátszó rendszerhez is, ezt követően létrehoztam belőlük egy-egy futtatható változatot, ezeket már a fejlesztőkörnyezet nélkül is el lehet indítani.



## 6 Összegzés

Ebben a fejezetben szeretném megosztani a projekt során szerzett tapasztalataimat, valamint kitérni arra is, miképpen lehet a projektben található egyes komponenseket továbbfejleszteni.

### 6.1 Projekt értékelése

A projekt elkészítése során több, általam eddig nem használt technológiával ismerkedhettem meg, ilyen volt például a Docker, valamint a Blazor Server is. Ezen technológiák megismerését mindenképpen hasznosnak tartom, hiszen a Docker egy széles körben használt technológia, melynek ismeretére jó eséllyel szükségem lesz a későbbiek során, a Blazor Server pedig szerintem egy nagyon kényelmes, könnyen használható keretrendszer a weboldalfélesztéshez, melynek segítségével nagyobb probléma nélkül implementálni tudtam az alkalmazáshoz használt weboldalakat, illetve a mögöttük található logikát. Különösen nagy előny volt számomra, hogy ugyanúgy C# nyelven írhattam a programkódot, ahogyan a többi komponens esetében.

A fejlesztés során az új technológiák megismerése mellett jobban el tudtam mélyülni a Unity alapú játékfejlesztésben is, valamint fejleszthettem a programozói képességeimet a különböző tervezési minták, objektumorientált programozási szabályok, elvek alkalmazásával. Ezen projekt elkészítése során is tapasztalhattam, mennyire fontos a modularizáció, a komponensek közötti függőségek minimalizálása, hogy könnyebben lehessen tesztelni, illetve utólagosan módosítani, kibővíteni a kódot.

Összességében szerintem rengeteg tudást, tapasztalatot szereztem a projekt során, eközben pedig egy számomra érdekes és izgalmas témával foglalkozhattam, a játékfejlesztéssel.

### 6.2 Továbbfejlesztési lehetőségek

A projektben található komponensek közül több esetében is felmerülhet a továbbfejlesztés lehetősége. A játéklogika, szinte bármédig bővíthető, a jelenlegi funkciók esetében hozzáadhatunk új épületeket, egységeket, technológiákat. Új funkció lehetne például, hogy a pálya generálásakor kerüljenek elhelyezésre független bázisok, ahol a játék által irányított, minden játékoskal szemben ellenségesen fellépő egységek

tartózkodnak. Az ilyen bázisok támadásokkal nehezítenék a játékosok fejlődését, viszont legyőzésükért cserébe fizetőeszközökhöz jutnának. Egy másik játékbeli funkció lehetne, hogy az épületeknek, egységeknek saját szintjei lennének. A magasabb szintre való fejlesztés fizetőeszközökbe kerülne, viszont a nagyobb szint javítaná a tulajdonságaikat. Egységek esetében ez jelenthet több életpontot, sebzést, mozgási és támadási távolságot, épületeknél több életpontot, valamint magasabb termelési hatékonyságot, a városoknak pedig növekedhetne a körzete is, amelyen belül épületeket lehet építeni. A játék új funkciókkal való kibővítése természetesen magába foglalná a felhasználói felületek, beleértve a visszajátszó rendszer fejlesztését is, illetve a naplózó rendszernek is tudnia kell kezelni ezeket a cselekvéseket. Az új játékfunkciók implementálása emellett komplexebb mesterséges intelligenciákat is jelentenének.

A játék emberi felhasználók számára készült grafikus megjelenítése jelenleg csak az egy eszközön való egymás elleni játékot teszi lehetővé, továbbfejlesztési lehetőség lenne ehhez a játékmódhoz is készíteni egy olyan komponenst, amely lehetővé teszi, hogy a játékosok hálózaton keresztül is tudjanak egymás ellen játszani.

A játék mellett az automatizáló rendszert is tovább lehet fejleszteni, amely jelenleg csak két kliens között tud játékot indítani, a játék viszont képes lenne legfeljebb négy játékos fogadására is. Ezen beállítást is ki lehetne szervezni a versenyekhez tartozó weboldalra. A webszerver jelenleg az egyszerű használatot teszi lehetővé, a hozzá tartozó weboldalak megjelenését, kinézetét különböző tervezési megfontolás alapján tovább lehetne javítani a felhasználói élmény növelése érdekében.

# Irodalomjegyzék

- [1] Wikipedia: *Strategy video game*, [https://en.wikipedia.org/wiki/Strategy\\_video\\_game](https://en.wikipedia.org/wiki/Strategy_video_game) [Hozzáférés dátuma: 2023.11.30]
- [2] Civilization: *About Civilization VI*, <https://civilization.com/> [Hozzáférés dátuma: 2023.11.30]
- [3] Microsoft Learn: *A tour of the C# language*, <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/> [Hozzáférés dátuma: 2023.11.30]
- [4] Microsoft Learn: *What is .NET?*, <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet> [Hozzáférés dátuma: 2023.11.30]
- [5] W3Schools: *What is JSON?*, [https://www.w3schools.com/whatis/whatis\\_json.asp](https://www.w3schools.com/whatis/whatis_json.asp) [Hozzáférés dátuma: 2023.11.30]
- [6] W3Schools: *Introduction to XML*, [https://www.w3schools.com/xml/xml\\_what.asp](https://www.w3schools.com/xml/xml_what.asp) [Hozzáférés dátuma: 2023.11.30]
- [7] MDN Web Docs: *The WebSocket API (WebSockets)*, [https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) [Hozzáférés dátuma: 2023.11.30]
- [8] Python: *What is Python? Executive Summary*, <https://www.python.org/doc/essays/blurb/> [Hozzáférés dátuma: 2023.11.30]
- [9] Read the Docs: *websockets*, <https://websockets.readthedocs.io/en/stable/> [Hozzáférés dátuma: 2023.11.30]
- [10] PyPI: *websocket-client*, <https://pypi.org/project/websocket-client/> [Hozzáférés dátuma: 2023.11.30]
- [11] Docker Docs: *Docker overview*, <https://docs.docker.com/get-started/overview/> [Hozzáférés dátuma: 2023.11.30]
- [12] Docker Docs: *Overview of Docker Desktop*, <https://docs.docker.com/desktop/> [Hozzáférés dátuma: 2023.11.30]
- [13] NuGet: *Docker.DotNet*, <https://www.nuget.org/packages/Docker.DotNet> [Hozzáférés dátuma: 2023.11.30]
- [14] Microsoft Learn: *Overview of ASP.NET Core*, <https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0> [Hozzáférés dátuma: 2023.11.30]

- [15] Microsoft Learn: *Introduction to Razor Pages in ASP.NET Core*, <https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-8.0&tabs=visual-studio> [Hozzáférés dátuma: 2023.11.30]
- [16] Microsoft Learn: *Blazor Server*, <https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-8.0#blazor-server> [Hozzáférés dátuma: 2023.11.30]
- [17] Microsoft Learn: *Blazor WebAssembly*, <https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-8.0#blazor-webassembly> [Hozzáférés dátuma: 2023.11.30]
- [18] Microsoft Learn: *Introduction to Identity on ASP.NET Core*, <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0&tabs=visual-studio> [Hozzáférés dátuma: 2023.11.30]