Pannon Egyetem

Műszaki Informatikai Kar

Villamosmérnöki és Információs Rendszerek Tanszék

Programtervező Informatikus BSc szak

SZAKDOLGOZAT

Tower Defense Stílusú Többszemélyes Játék Fejlesztése

Géringer Ábel Róbert

Témavezető: Dr. Guzsvinecz Tibor

2021

FELADATKIÍRÁS

- Téma: **Tower defense stílusú többszemélyes játék fejlesztése**
- Cím: **Tower defense stílusú többszemélyes játék fejlesztése**
- Végleges cím:
- Oktatók: **Dr. Guzsvinecz Tibor**
- Jelentkezés dátuma: **2021.09.07. 13:14:44**
- Elfogadás dátuma: **2021.09.22. 7:54:26**
- Beadás dátuma:
- Bemutatás dátuma:
- Védés dátuma:
- Külső téma: **Nem**
- Leírás:

- Nyelv: **magyar**
- Szervezeti egység: **MIVIR**
- Szakdolgozat státusz:
- Oktatói vélemény: **Támogatott**
- Beosztás eredménye: ✔
- Elfogadó: **Bálint Adrienn**
- Visszavonás dátuma:
- Védés eredménye:
- Titkos: **Nem titkos**
- Url:
- Sorszám: **SZD21092207546484**

Tower Defense témájú játék. A lényege, hogy egy (vagy több) kijelölt úton jönnek ellenségek és azokat tüzelő tornyok megvásárlásával és elhelyezésével a pályán kell megvalósítani. - Hang: saját zenét és hangeffektek - Multiplayer: egyszerre 2 játékos játszhatna és a pálya fel lenne osztva, hogy ki melyik részen építhet - WaveSpawner: a hullámok random mintában jönnek - Nehézségi fokozatok - Pályaszerkesztő: A játékosok megszabhatnak közel mindent a pályán, mennyi pénzzel kezdjenek, mekkora legyen a pálya, milyen dizájn elemeket tennének rá, mennyi irányból jöjjenek a hullámok, mennyi helyre érkezzenek, mennyi élet legyen, milyen gyakran jöjjenek a hullámok stb... - Custom Maps: az előző pontból jön ez a lehetőség, hogy elmentsék az eddig készített pályákat és bármikor játszhassanak - Saját modellek Kérésre kiírt téma.

NYILATKOZAT

Alulírott Géringer Ábel Róbert diplomázó hallgató, kijelentem, hogy a szakdolgozatot a Pannon Egyetem Villamosmérnöki és Információs Rendszerek Tanszékén készítettem Programtervező informatikus BSc diploma (BSc in Computer Science) megszerzése érdekében.

Kijelentem, hogy a szakdolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy a szakdolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Veszprém, 2021. dec. 03.

aláírás

Alulírott Dr. Guzsvinecz Tibor témavezető kijelentem, hogy a szakdolgozatot Géringer Ábel Róbert a Pannon Egyetem Villamosmérnöki és Információs Rendszerek Tanszékén készítette mérnök informatikus BSc diploma (BSc in Computer Science) megszerzése érdekében.

Kijelentem, hogy a szakdolgozat védésre bocsátását engedélyezem.

Veszprém, 2021. dec. 03.

aláírás

KÖSZÖNETNYILVÁNÍTÁS

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Guzsvinecz Tibornak a bizalmáért, mellyel lehetőséget biztosított számomra a szakdolgozatom elkészítéséhez. Kiváló szakmai tudásával, tapasztalatával és átfogó látásmódjával mindvégig segítségemre volt a munkám során. Szeretnék köszönetet mondani családomnak, akik megértésükkel és odaadó segítségükkel mindig támogattak. Nekik köszönhetem, hogy idáig eljuthattam.

TARTALMI ÖSSZEFOGLALÓ

A szakdolgozat Tower defense stílusú többszemélyes játékot mutat be. Az alkalmazás a Unity tátékfejlesztő környezet segítségével csináltam C# nyelven, melyet Microsoft Visual Studio-ban írtam. A fejlesztés Windows 10 operációs rendszeren végeztem. A program mind egy- és többjátékos módban játszató, melynek lényege a torony megvédése. Az ellenség egy adott útvonalon, hullámokban, tart a torony felé, hogy elfoglalhassák. A játékos(ok) feladata ezt megakadályozni, különböző védő tornyok átgondolt elhelyezésével, valamint fejlesztésével a pályán.

Az alkalmazás funkciói:

- Állítható nehézségi fokozatok
- Pályaszerkesztő
- Többjátékos mód
- Grafikai és egyéb beállítások

Dolgozatom ismerteti a felhasznált technológiákat, és bemutatja az alkalmazás megtervezésének folyamatát. Ismertetem a fejlesztett program működését, majd értékelem az elkészült alkalmazást. Végül szót ejtek a továbbfejlesztési lehetőségekről.

Kulcsszavak: Tower Defense, Microsoft Visual Studio, játék, Genetikus Algoritmus

ABSTRACT

My thesis is about a Tower Defense styled multiplayer game. The application is made by the Unity Engine and was written in C# in Microsoft Visual Studio. The development was going under Windows 10 operation system. The game can be played both alone and with a friend. The players mission in the game is to defend the tower from the incoming enemy waves by strategically placing and also upgrading turrets on the map to prevent them from occupying it.

The functions of the application:

- Multiple difficulties
- Map creator
- Multiplayer
- Graphical and other options

My thesis reviews the used technologies and presents the process of designing the application's user interface and back-end. It describes how the finished application operates. Lastly, I evaluate the work done and write about the possible improvements.

Keywords: Tower Defense, Microsoft Visual Studio, game, Genetic Algorithm

TARTALOMJEGYZÉK

1. Fejezet

1.1 Bevezetés

Szakdolgozatom egy régebbi játék stílusa adta meg az ihletet, melyet még 10-15 éve még Windows XP-n játszottam a böngészőben a Flash Player-rel 2D-ben. A „Tower Defense” stílus, mely a stratégiai játékok egy alcsoportjába tartozik, aranykora egészen az 1990-es évekre vezethető vissza, az Árkád játékokhoz, amikor megjelent a „Rampart” 1990-ben, melyet az akkor ismertebb Atari Games adott ki. Mindig is érdekelt a játékfejlesztés és gondoltam ez egy remek alkalom lesz arra, hogy elsajátíthassam annak alapjait, miközben egy kedvenc műfajomból csinálok játékot egy kis extrával. Szerettem volna valami kis pluszt is belevinni a játékba, így a játékot 3D-ben csinálom, melyben lesz lehetőség online, barátokkal együttesen visszaverni az érkező ellenségeket. A játékon belül van lehetőség egy egyszerű kampány módra, melyben az ellenségek egy adott számú körön át érkezik, valamint van lehetőség olyan módra melyben csakis kizárólag a mi ügyességünkön múlik, hogy meddig bírjuk, ebben a lehetőségben mindig lesz egy következő hullám, melyben helyt kell állni. Ennek megvalósítására az Unity 3D játékfejlesztő környezetet választottam, ugyanis ez a program az egyik legnépszerűbb jelenleg a piacon, vezető cégek ezzel a programmal dolgoznak hosszú évek óta, mely folyamatos fejlesztés alatt van.

1.2 Tower Defense Stílus

A „Tower Defense” a stratégiai játékok egyik alkategóriája sok más osztállyal együtt, csak hogy egy pár közismertebbet megemlítsek: TBS (Turn-Based Strategy), RTS (Real Time Strategy), MOBA (Multiplayer Online Battle Arena). A TBS egy körökre osztott játék, akár a sakk. Minden egyes körben egy valaki végezheti el stratégiai lépéseit. Az RTS egy olyan stratégiai játék, melyet nem körökre bontva kell játszani, hanem valós időben történik minden. A MOBA lényege, hogy 2 csapat játszik egymás ellen egy előre megadott pályán. Sokan gondolják laikusként, hogy egyetlen pályán játszani minden alkalommal nagyon repetitív, viszont ebben az esetben más elemekkel teszik változatossá a játékokat. Ha példának vesszük a DOTA2 című játékot, minden egyes játék teljesen különböző, bár egy pálya van csak, melyet úgy érnek el, hogy 123 karakterrel lehet játszani, melyeknek különböző képességei, tulajdonságai, szerepei vannak a játékban.

1.3 Hasonló Stílusú Játékok

Ebben a stílusban sok játék jelent meg az évek során, melyek nagy népszerűségnek is örvendenek, csak hogy néhányat említsek, itt van talán a legismertebb a genre-ban a „Plants Vs zombies” melyet az Electronic Arts (EA) leányvállalata a „PopCap Games” fejlesztett. Az első része a játéknak még 2009-ben jelent meg, mely egy 2D-s játék, amit szinte minden platformra kiadtak a fejlesztők. Emellett érdemes megemlíteni még egy jelenleg is nagy népszerűségnek örvendő játékot a „Bloons TD” -t, mely szintén 2D-ben készült először, 2007-ben, valamint egy személyes kedvencemet a „Dungeon Defenders”, mely merőben eltérő az előbb említett 2 játéktól. 3D-ben készült már az első része is még 2011-ben. Nagy újdonság volt akkor az új mechanika, miszerint egy hőst kellett irányítanunk a térképen úgy nevezett „Third Person View” -ban (A karakterünk hátát látjuk és irányítjuk) a több választható és nagyban különböző karakterek közül és vele építeni meg a különböző védőtornyokat és beszállni a védésbe.

2. Fejezet

2.1 Felhasználói Dokumentáció

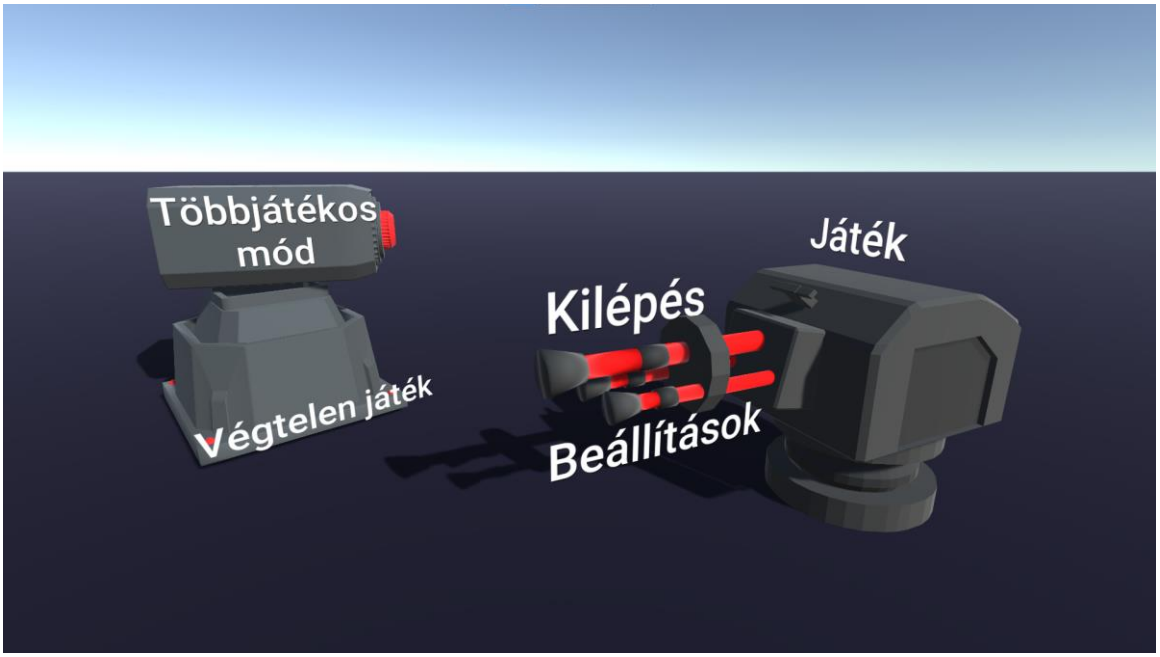
2.1.1 A program használata

A játékot Windows operációs rendszeren futtathatjuk. Telepíteni nem kell, a TowerDefense.exe fájllal indíthatjuk.

2.1.2 Irányítás

- jobbra: az egér jobbra húzása
- balra: az egér balra húzása
- fel: az egér felhúzása húzása
- le: az egér lefele húzása
- közelítés/távolítás: görgővel

2.1.3 Menürendszer



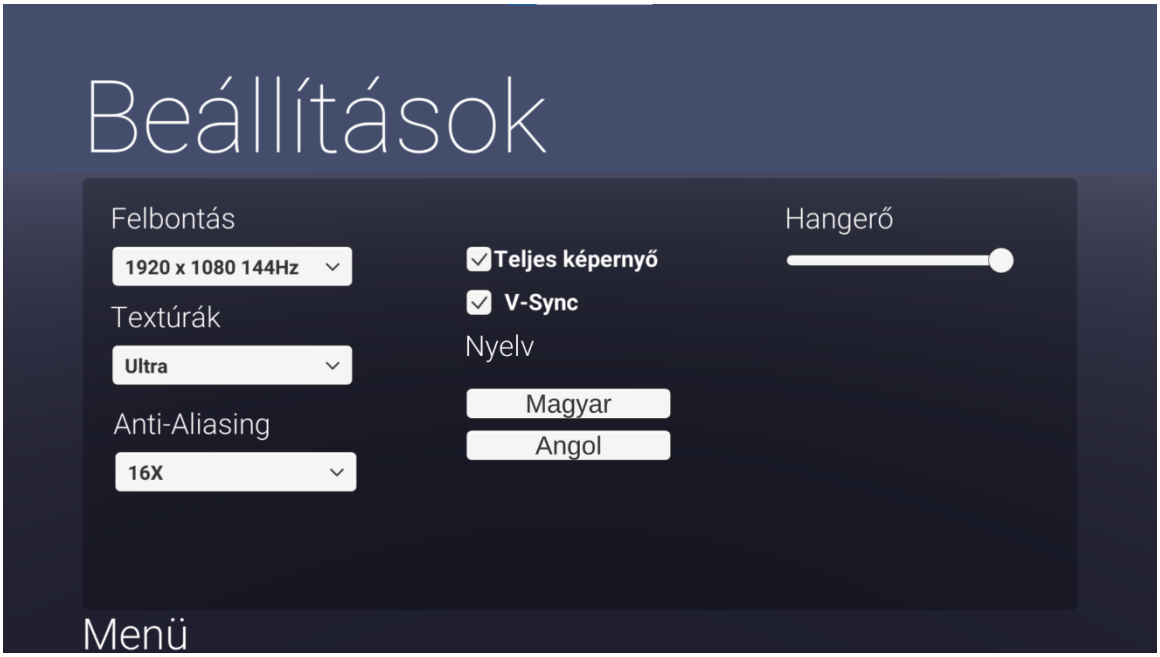
A program indulásakor a főmenüben 5 menüpont közül lehet választani:

- **Játék:** ezzel a gombbal mehetünk el a pályakiválasztó menübe
- **Többjátékos:** ezzel a gombbal mehetünk el a többjátékos módhoz
- **Végtelen játék:** ezzel a gombbal jutunk el a végtelen hosszú játékmódba

- **Beállítások:** ezzel a gombbal mehetünk el a beállítások menübe
- **Kilépés:** ezzel a gombbal léphetünk ki az alkalmazásból

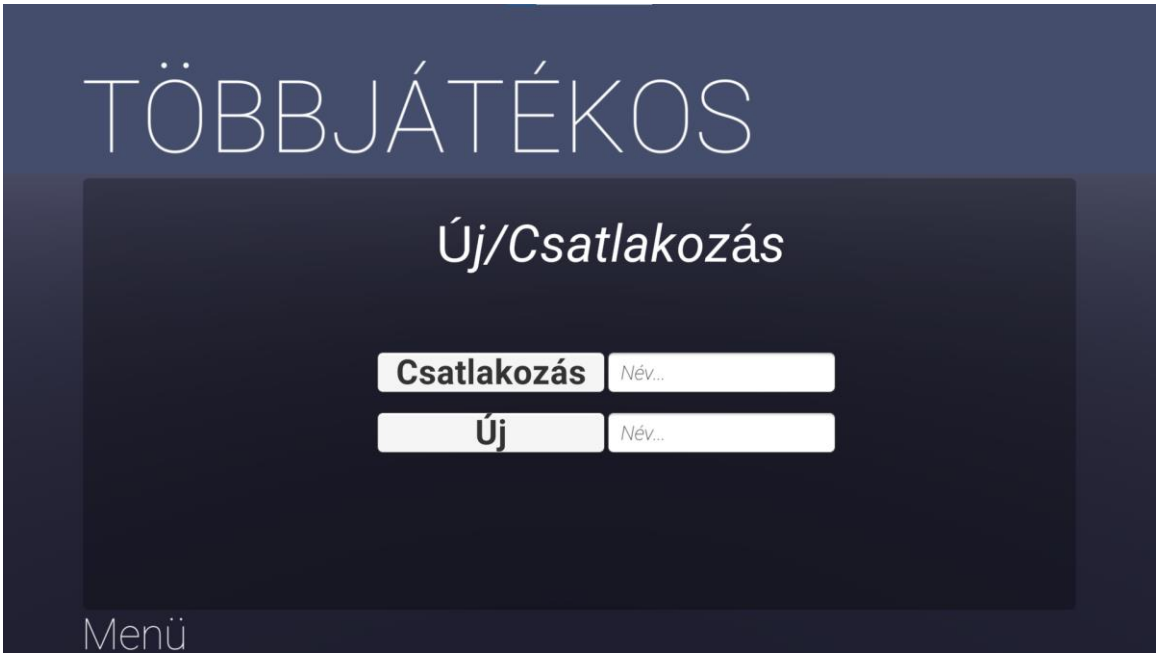
2.1.4 Beállítások

- **Teljes képernyő:** kiválaszthatjuk, hogy teljesképernyőn szeretnénk-e játszani.
- **Felbontás:** kiválaszthatjuk a játék ablakának felbontását.
- **Textúra:** kiválaszthatjuk a játék minőségét.
- **Anti-Aliasing:** Az élsimítás a mintavételezett jelek, például digitális képek vagy digitális hangfelvételek álsimítási problémáinak leküzdésére szolgáló számos technika bármelyikére utalhat.
- **V-sync:** A függőleges szinkronizálás segít a stabilitás megteremtésében azáltal, hogy szinkronizálja a játék vagy az alkalmazás képkockasebességét a képernyő-monitor frissítési gyakoriságával. Ha nincs szinkronizálva, az a képernyő beszakadásához vezethet, aminek következtében a kép hibásnak vagy vízszintesen duplikáltnak tűnik a képernyőn.
- **Hangerő:** beállíthatja a játékos a hangerősségét a játéknak.
- **Angol – Magyar:** kiválasztható nyelvek melyek a teljes játékban alkalmazva lesznek.



2.1.5 Többjátékos mód

Ebben a menüben tudunk bárkivel együtt játszani, akinek szintén van hozzáférése az internezhez. Annyit kell tennünk, hogy beírjuk a szobánk nevét, amelyben szeretnénk játszani és létrehozzuk a create gombbal, amellyel rögtön elindítjuk a játékot. Ha már létre van hozva a szoba és csak be szeretnénk csatlakozni a barátunkhoz, akkor a szobanév beírása után csak rá kell kattintanunk a join gombra, mely egyből be fog dobni minket a játékba. Ebben az opcióban, egyelőre csak egy pályán tudunk játszani online, de később lehetőség lesz arra, hogy egy lobbyban kiválaszthassuk a pályát is.

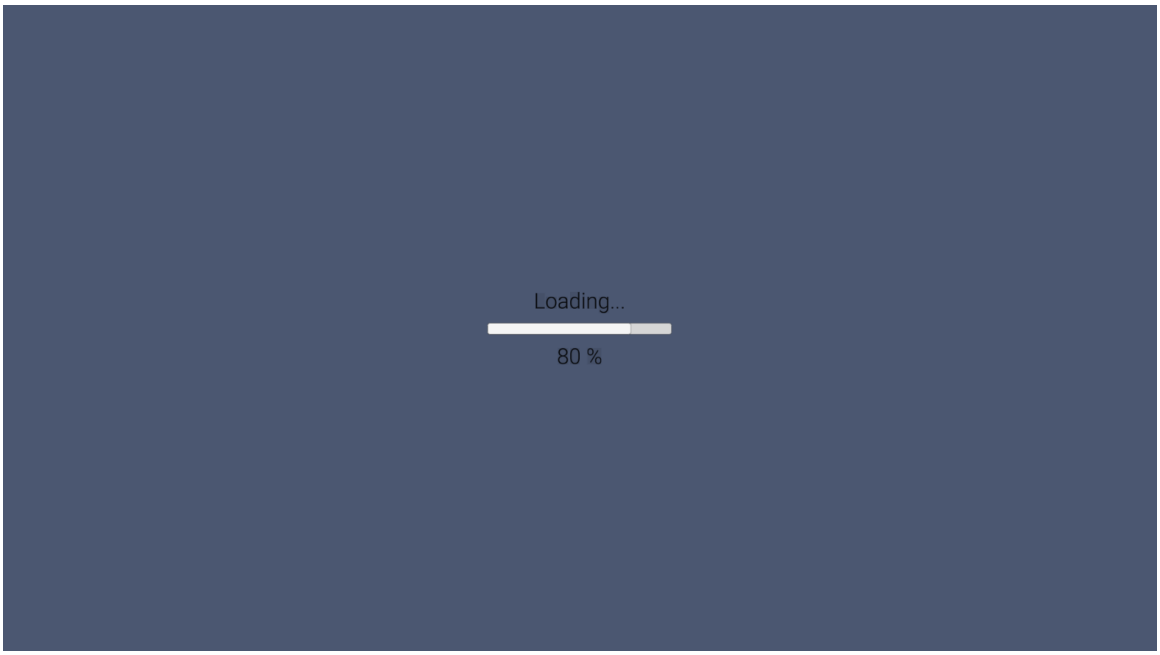


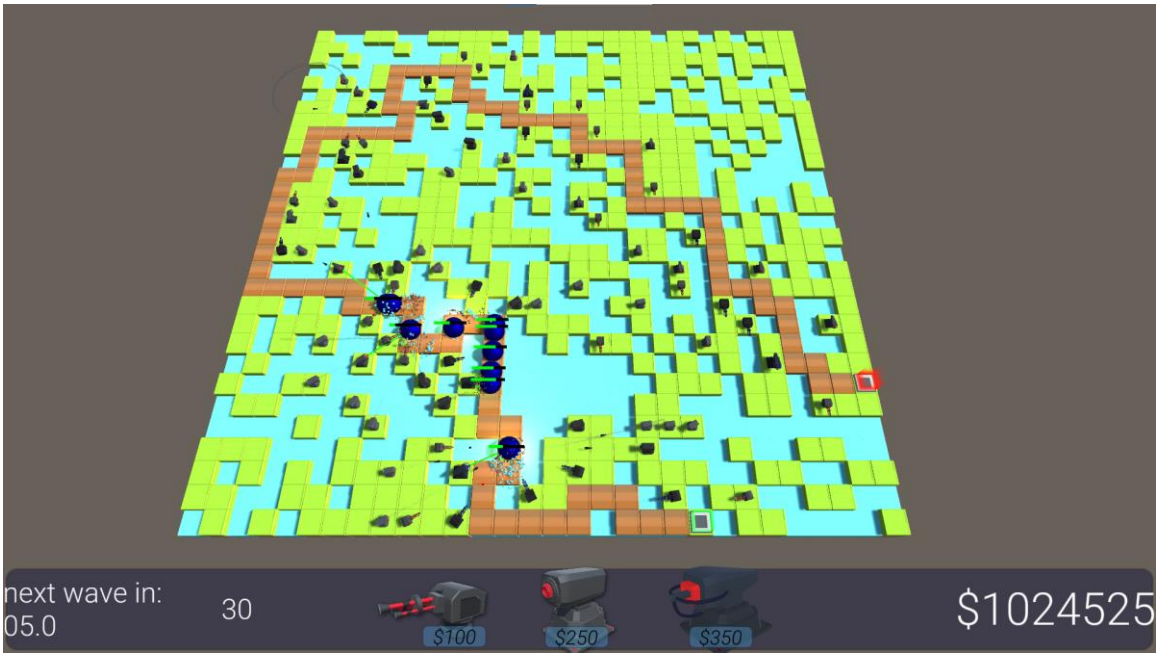
2.1.6 Játék

A játék gombra kattintva az alábbi képen látható menübe juthatunk el. Ez a játék kampány pályáit mutatja meg, melyben 5 pályát tud játszani a játékos, ha pedig végzett velük, újrakezdheti, vagy random pályákon van lehetősége új és új kihívásokat találni.

2.1.7 Végtelen mód

A végtelen mód egy rövidebb töltőképernyővel indul, majd miután végzett indul el a játék és elkezdhetünk egy olyan pályán játszani, melyen még senki más.





2.1.6 Szabályok

A játék célja, hogy a hullámokban érkező ellenségeket, különböző lőtornyokkal megsemmisítsük és meggátoljuk, hogy elérjék a tornyot. A megsemmisített ellenségekért a játék pénzel jutalmaz meg, melyet újabb tornyokra, vagy pedig fejlesztésekre költhetünk. Minden egyes pályán, meg van adva, hogy a játékos mennyi ellenséges egységet engedhet oda a várhoz a pálya tetején, bal oldalán, hogy hányadik hullámnál tartunk, valamint a pálya jobbalsó sarkában látjuk, mennyi idő van még vissza a következő csapat ellenségig. A játékablak alján, kiválaszthatjuk a nekünk kellő védőtornyokat, melyek ára, valamint feladata is sok mindenben különbözik! A játékban szereptő tornyok modelljeit, egy ingyenes Asset Pack-ból szereztem (https://devassets.com/assets/tower-defense-assets/).

- **Rakétakilövő állomás:** rakétákat lő, melynek robbanása következtében, egy kisebb sugarú körön belül minden ellenséges egységet sebez. Vigyázz, sebzése nem nagy, cserébe messzire lő!

- **Lézer projektor:** egy erős, zöld lézersugarat lő ki magából, mellyel egy rosszakarót tud sebezni és lassítani is egyszerre egy kisebb sugarú körben.

- **Egyszerű löveg:** kis töltényeket lő ki magából, mellyel egy a Rocket Launcher-nél kisebb, de a Leaser Beamer-nél nagyobb sugarú körben tudja sebezni az ellent.

A játék során figyelni kell, hogy a pénzedet okosan használd fel! Nem mindegy, hogy egy tornyot fejlesztesz, vagy esetleg veszel egy másikat, valamint nem mindegy, hogy a pálya mely pontjain helyezed el melyik tornyot! A játékban többféle ellenség van, amikből több különböző erősségű is van, melyekre érdemes odafigyelni a játék előrehaladtával.

- **alap ellenség:** ez egy alap ellenség, melynek nincsen kiemelkedő tulajdonsága, élete, sebessége, valamint a vele elnyert pénz mennyisége is a többi között átlagos. Ismertető jele, hogy a kék 3 különböző árnyalatát veszi fel.
- **Gyors ellenség:** ennek a lénynek nagy jellemzője, hogy sokkal gyorsabb az összes többinél, és egy kicsivel többet is kapni belőle, és ennek van a legkevesebb élete az összes többi közül. Onnan lehet felismerni, hogy sárga.
- **Tank:** az egyik leglassabb fajta rosszakaró, melyért kicsivel több pénz jár, viszont annál nehezebb megsemmisíteni, mivel több élete van, mint az eddigieknél. Felismerni onnan lehet, hogy piros.
- **Fő ellenség:** a legnehezebb ellenfél az egész játékban, nagyon sok pénzt lehet nyerni leküzdése után, nagyon sok élete van és gyorsabb, mint egy standard enemy. A játékban csak bizonyos hullámokban kerül elő, hisz fel kell készülni rá a játékosnak. Arról ismerni fel, hogy ez a legnagyobb féle ellenség a játékban.

3. Fejezet

3.1 Fejlesztői Dokumentáció

3.1.1 Felhasznált Technológiák

Unity játékfejlesztő környezetben készült a program, a scipt-eket pedig az JetBrains Rider-ben írom. Választásom azért jutott ezekre, mivel az egyetem biztosít lehetőséget a JetBrains termékeinek használatára, melyeket nagyon szeretek és nagyon kényelmes használni őket, legyen akár jelen esetben a C# scriptek írása, vagy Java, Kotlin, de akár PHP is. Az Unity-t pedig a hatalmas közössége miatt választottam, valamint, hogy a kezdő Unity fejlesztők is könnyebben tudnak elindulni. A projekt verziókezelése Github segítségével valósult meg, mert a legtöbb nagy cég is ezt használja és biztonságos, valamint könnyedén követhető a projekt fejlődése, hiba esetén egy korábbi verzió visszaállítása. A többjátékos mód a Photon PUN (Photon Unity Network) mely egy Realtime Cloud service, hogy a világon bárki játszhasson egymással, ha van hozzáférése internethez. Ez a technológia nagyon sokoldalú és kiváló integrációja van az Unity játékfejlesztő környezettel, bármely „Room Based" (kisebb online szobákra alapozó) játék megteremtéséhez. Választásom azért került a Photon PUN-ra, mivelhogy ez egy olyan ingyenes lehetőség a többjátékos mód megvalósítására, melyben, ha meghaladnám az ingyenes keretet is véletlen, sem kapnék elsőre egy csekket, amit be kellene fizetnem. A legbiztosabb alapja a Photon sebességének a „client-2-server-architecture", ez egy olyan modell melyben a szerver irányítja az erőforrások nagy részét és a szolgáltatásokat pedig a kliens. Más ismertebb elnevezései a „networking computing model" vagy „client/server network", mert minden a hálózaton keresztül történik. Ha ez mind nem volna elég, akkor lehetőség van „Cross Platform" -ra is, tehát nem számítana, hogy milyen operációs rendszeren, vagy eszközön játszunk, van lehetőség együtt, egy online szobában visszaverni az ellent.

3.1.2 Funkcionális követelmények

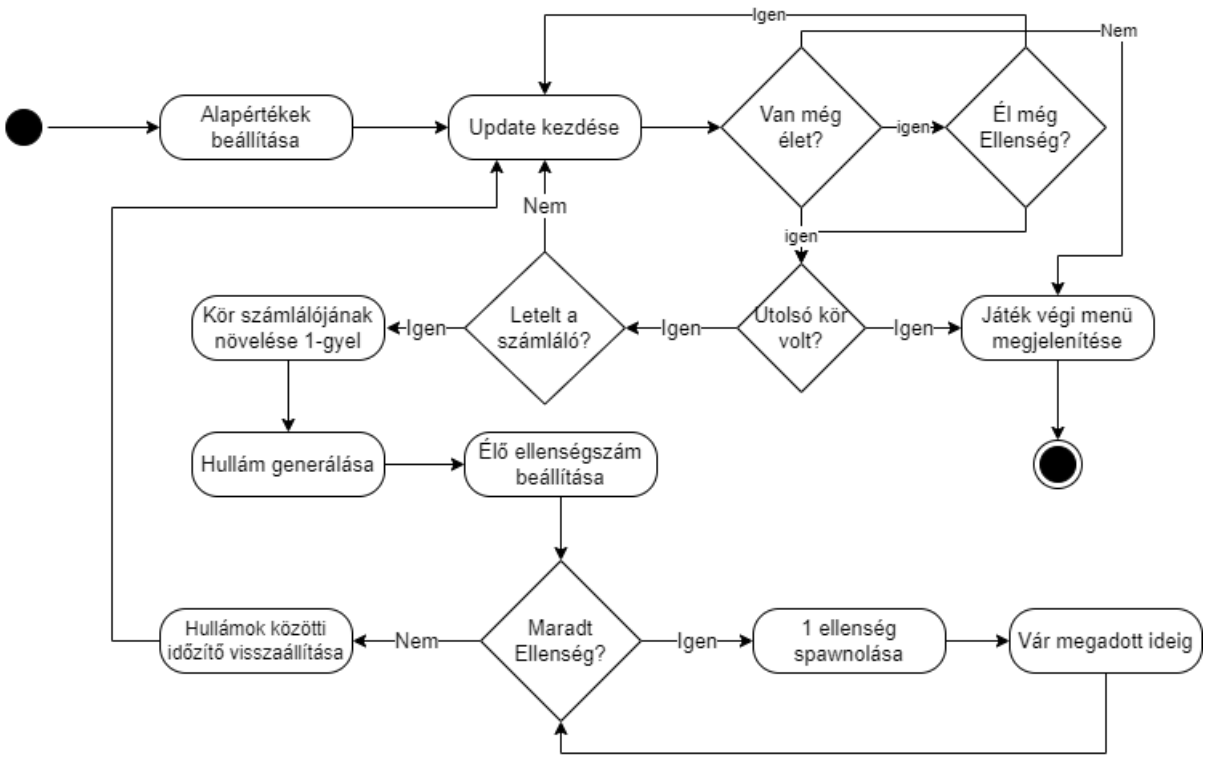
- A programban van lehetőség különböző grafikai beállításokra, mint például:
  - Felbontás
  - Teljes képernyős mód (Fullscreen és Windowed)
  - Textúrák minősége
  - Élsimítás (Anti-aliasing)

    - Vertikális szinkronizáció (V-sync)
    - Nyelv (angol vagy magyar)
- A programban a játékos tud egyedül és barátokkal is játszani
- A programban lehet saját pályát készíteni
- A saját pályákat el lehessen menteni
- A programban lehet random generált pályán is játszani
- A programban az ellenségek véletlenszerűen jönnek a nehézségi beállítástól függően
- A programban az egyjátékos módban, több pálya is van melyeket csak sorban lehet játszani elsőre.
- A programban az egyjátékos módban a pályák kioldhatók legyenek csak, hogyha nyertek az előtte lévő pályán
- A programban könnyen lehessen navigálni
- A játék közben, fontosabb információk, mint a pénz, hányadik hullám és a hátralévő életek legyenek láthatók
- A játékban legyen minimum 3 féle torony, melyet a játékos letehet
- A játékban legyen minimum 8 féle ellenség
- A program indításakor az elmentett beállítások betöltődnek
- A játék közben lehessen szünetet tartani és újrakezdeni az adott pályát

2.2.2 Nem Funkcionális Követelmények

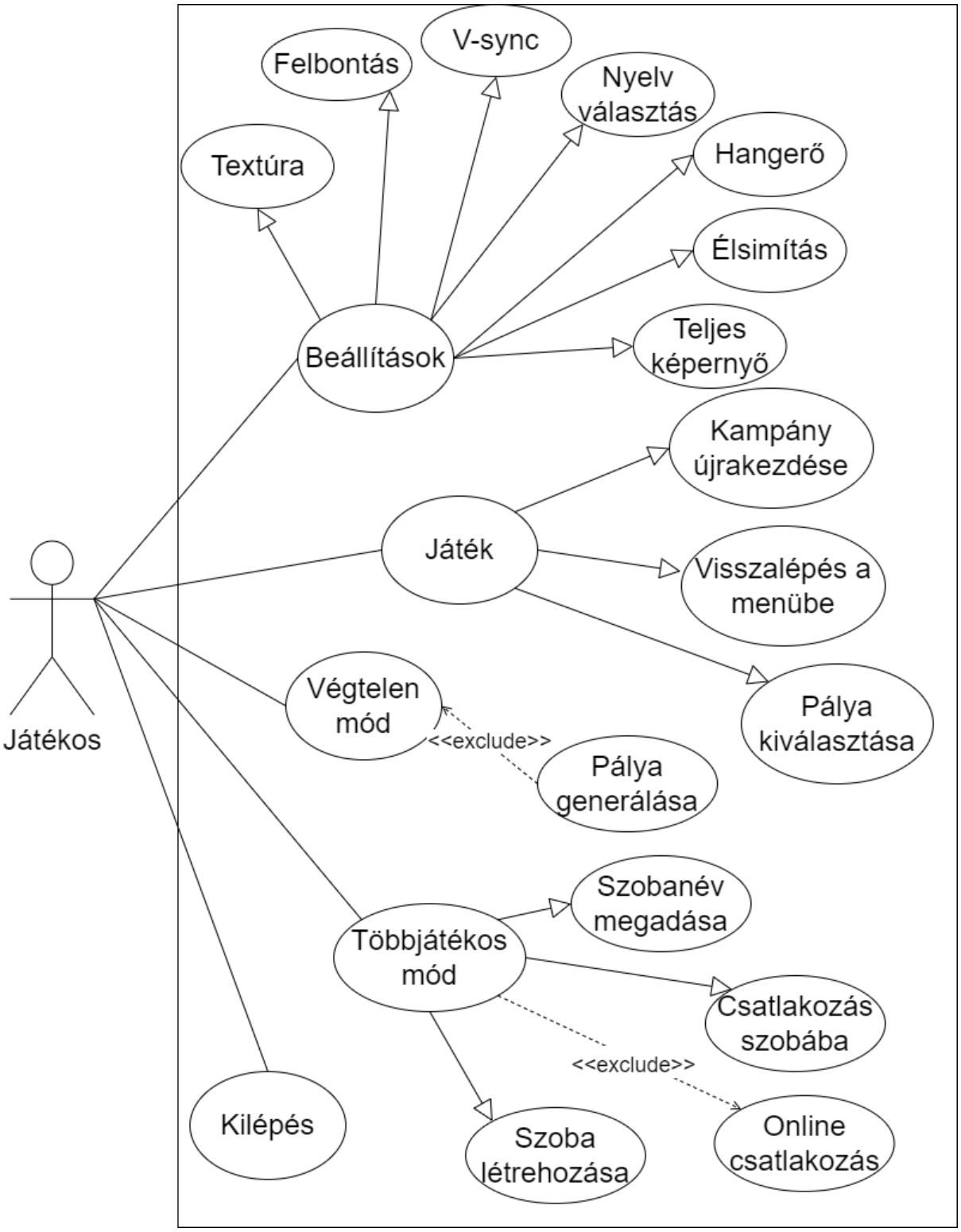
- Kis rendszerigénye legyen
- Lehessen futtatni PC-n, Linuxon és MAC-en
- A játék átlátható legyen
- Könnyen lehessen navigálni a menüpontok között
- Ne sértse meg a szerzői jogok védelmét

3.2.1 Előre Definiált Hullám Generálása



Az első lépésben inicializálni kell néhány alap változót, hányadik hullám ellenségnél tart a játék, valamint hányan vannak pályán életben még, ezeket mind a kettőt 0-ra, egyelőre. Az update metódus, minden frame-ben, ezt a függvényt használják a legtöbbet, ha valamilyen játékon belüli scriptet írunk. Minden egyes híváskor meg kell vizsgálnunk, hogy vége lett-e a játéknak, vagy még folytathatjuk, melyhez meg kell vizsgálni, hogy van-e még ellenség, van-e még hátralévő hullám, amelyet le kell győznünk. Ha teljesen elfogytak az ellenségek, akkor nyertünk, megjelen a győzelmi menü, és eldönthetjük, hogy újra próbálnánk, haladnánk a következő pályára, vagy ki szeretnénk lépni a menübe. Abban az esetben, ha van még hullám, akkor elindul egy visszaszámláló, majd elkezdődik a spawn (valamely GameObject létrehozása a pályán). A spawn egy előre, statikusan megadott hullámot kezd el legenerálni, mely elindul a kijelölt útvonalon.

3.2.3 Használati Eset Diagram



3.2.4 Használati Eset Diagram Leírása

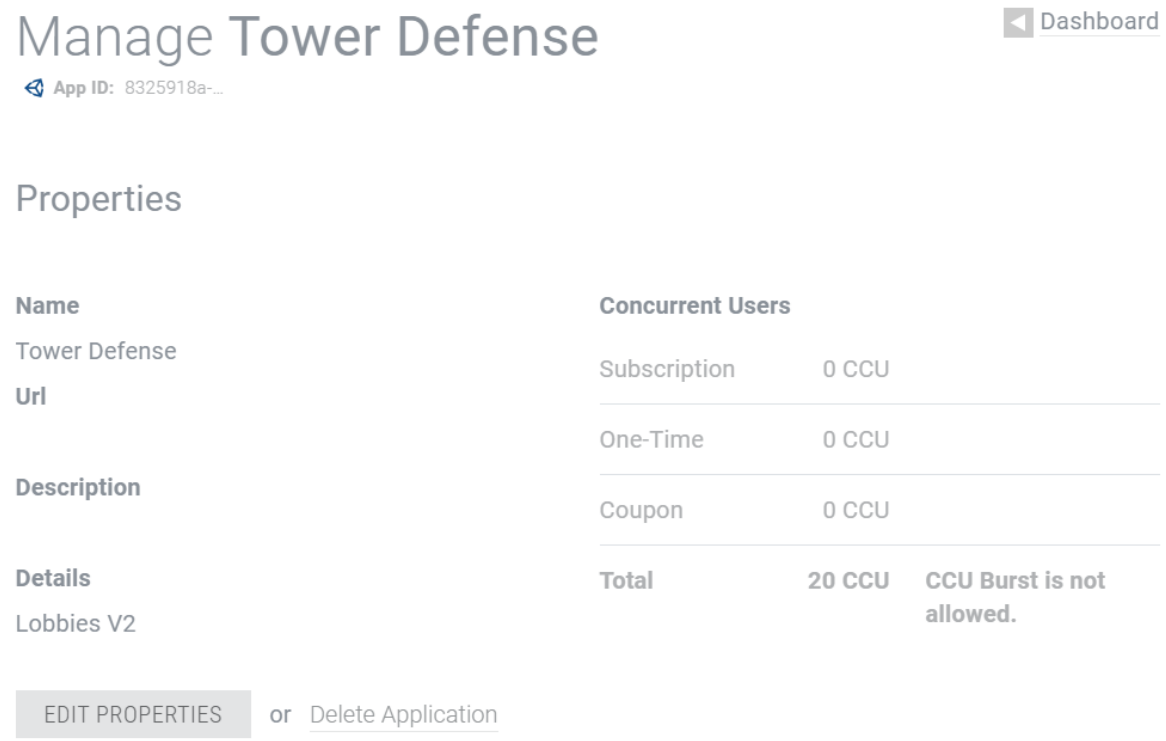
A játékos indításkor először a menüben találja magát, ahol több opció közül választhat. Beállítások, egyjátékos mód, többjátékos mód, végtelen játék és kilépés a játékból. A

beállításokban találja meg a játék grafikai beállításait, hangerő szabályozását, valamint a nyelvi beállításokat. A beállítások között több dolgot is tud a játékos állítani, mely lehet akár grafikai, hanggal kapcsolatos vagy nyelvi beállítás. A grafikai beállítások között megtalálható a Felbontás, textúra minősége, az élsimítás minősége, teljesképernyő és a vertikális szinkronizálás ki és bekapcsolása, a fő hangerő szabályozása, valamint kiválasztható az angol, illetve magyar nyelv. A kilépés gombbal tud kilépni a játékos a játékból. Az egyjátékos mód a szint kiválasztása menübe visz el, ahol ki lehet választani, hogy melyik pályán szeretne játszani a felhasználó, ebben a menüben elsőre csak az első pálya érhető el. A többi pálya kioldásához, mindig az azt megelőző szintet kell teljesíteni. Ha a játékos szeretné, akkor van lehetősége a játék újrakezdésére a visszaállítás gombbal, ez az opció visszaállítja a teljesített pályákat és csak az első szint lesz elérhető, a többihez újból el kell kezdenie minden pálya teljesítését. A többjátékos mód menüponttal a játékos először egy szoba létrehozásához, illetve szobába való csatlakozásához kellő menübe kerül. A csatlakozás a szobába gombbal tud egy már létező szobába becsatlakozni, melynek a nevét meg kell adnia. A szoba létrehozásához a szoba létrehozása gombra kell nyomnia, miután beírta a szoba nevét. Ezután a szoba létrehozója tovább kerül a szint kiválasztása menübe. A végtelen mód menüpont pedig, egy rövid töltőképernyő után, elviszi a játékost egy teljesen véletlenszerűen legenerált pályára, amely minden alkalommal egy teljesen új, procedurálisan létrehozott pályát ad a játékosnak.

3.3 Többjátékos mód

A játék a Photon Pun-nal készül, mely mind online mind offline módban is nagyon hasznos, hiszen nem kell implementálni bizonyos függvényeket singleplayer és multiplayer módra külön-külön, ezzel is elkerülve a tömeges kódismétléseket. A játék indulásakor automatikusan betesz minket a játék az offline módba, illetve egy szobába melynek neve „OfflineRoom”. Ez amiatt szükséges, hogy használhatóak legyenek a játékban az „Instantiate” függvények, melyekkel az ellenségeket, illetve a tornyokat spawnoljuk, mert nem az Unity Instantiate metódusa van meghívva, hanem a PhotonNetwork Instantiate függvénye. Ezzel biztosíthatjuk, hogy online és offline is ugyan azzal a kóddal tudjunk spawnolni. Ha a multiplayer menübe megyünk, akkor a játék online módba kerül a PhotonNetwork CreateUsingSettings meghívásával, majd ezután kezdődhet a szoba létrehozása vagy a szobába való belépés. Ezeket a lépéseket a JoinRoom, CreateRoom illetve a LeaveRoom függvényeket tudjuk végrehajtani. A multiplayer során úgy tudjuk megoldani azt az akadályt, hogy mindenkinél külön történjenek meg a spawnolások, toronyfejlesztés, -eladás és ehhez hasonló

szinkronizálásra szoruló scriptek, hogy a függvény fejléce felett egy sorral meghatározzuk, hogy ez egy RPC (Remote Procedure Calls) függvény a [PunRPC] attribútummal. Ez biztosítani fogja, hogy a hálózaton mindenhol szinkronizálva legyen az az adott dolog, amit a script csinál.



Itt az irányítópulton hozzáférhetünk az alkalmazás használatának mérőszámaihoz és hány CCU-t (egyidejűleg csatlakoztatott felhasználó) csatlakoztattak a játék összes szobájában. A Photonnak van egy ingyenes szintje, amely lehetővé teszi, hogy teszteljük (vagy élesítsük, ha akarjuk) 20 CCU-val és bármikor frissíthetjük, amikor csak akarjuk. Én személy szerint úgy gondolom, hogy ez nagyon jó koncepció és a kis játékoknak továbbá. Ezen felül, ha szükséges tudni, hogy mi történik a háttérben a szerver oldalon, akkor van megoldás a „Self-Hosted" szerverekhez, ugyanaz a megvalósítás, de ahelyett, hogy az övék kiszolgálóira menne, saját szervert tudunk használni. Személy szerint nagyszerű szolgáltatásnak tartom, mivel van lehetőség egy harmadik féltől származó szolgáltatásra közvetíteni, hogy kezelje az összes hálózati eseményt, a méretezhetőséget, az állásidőket, a válaszokat, a régiókat, a mérkőzések lebonyolítását. Ez valóban nagyszerű módja annak, hogy elkezdhesse valaki a többjátékos világát. Ez nem jelenti azt, hogy a Photonon lévők helyett ne tudnád megvalósítani a saját

megoldásodat, ez nem egy mindent vagy semmit, néhány funkció megvalósítható, és a többi funkció egyedül kezelhető.

3.4 Unity

A Unity a Unity Technologies által kifejlesztett többplatformos játékmotor, amelyet először 2005 júniusában jelentettek be és adtak ki az Apple Worldwide Developers Conference rendezvényen Mac OS X játékmotorként. A motort azóta fokozatosan kibővítették, hogy támogassa a különféle asztali, mobil-, konzol- és virtuális valóság platformokat. Különösen népszerű iOS és Android mobiljáték-fejlesztéseknél, könnyen használhatónak tartják a kezdő fejlesztők számára, és népszerű az indie játékfejlesztésben.

A motor segítségével háromdimenziós (3D) és kétdimenziós (2D) játékok, valamint interaktív szimulációk és egyéb élmények készíthetők. A motort a videojátékokon kívüli iparágak is átvették, mint például a filmipar, az autóipar, az építészet, a mérnöki ipar, az építőipar és az Egyesült Államok fegyveres erői.

3.4.1 GameObject és Komponens

Játékobjektumot jelent. A Unity alapvető objektuma, ez azt jelenti, hogy minden, ami a játékban van, az egy GameObject. Önmagukban nincs sok funkciójuk, komponensek hozzáadásával válhat belőlük például karakter, fa, fény.

3.4.2 Prefab

A GameObject-ekből prefab készíthető. Az objektum összes komponensét és beállításait tartalmazza, újra használható sablonként működik.

3.4.3 Komponensek

A GameObject-ek funkciói a hozzá kapcsolt komponensektől függ. A Unity rengeteg beépített komponenst tartalmaz, de a sajátunkat is elkészíthetjük. Néhány fontosabb komponens:

- A Transform komponens az objektum pozícióját, méretét és elforgatásának mértékét határozza meg. Minden GameObject alapvető komponense, nem törölhető belőle

- Animator segítségével irányítjuk az animációkat. Az animációk vagy effektek működését egy irányított gráf formájában adhatjuk meg. Az animációkat átmenetekkel kapcsolhatjuk össze, az átmenetekhez feltételek adhatóak.
- Text komponenssel szövegdobozt jeleníthetünk meg.
- Image komponenssel adható kép az objektumhoz.
- Button komponenssel az objektum gombként funkcionál. Megadható hozzá függvény, ami meghívódik a gombra kattintáskor.
- Photon View felelős a komponensek hálózaton belüli szinkronizálásáért. Hozzá kapcsoljuk azokat a komponenseket, amiknek valamilyen funkcióját, értékét szinkronizálni szeretnénk.
- Photon Transform View-t kapcsoljuk az előző pontban lévő Photon View-hoz. A Transform komponens figyeli.
- Photon Animator View segítségével szinkronizálhatóak az animációk közötti váltások.
- Scriptek is hozzáadhatók komponensekként. Ezzel saját funkciókat adhatunk a GameObjectnek.

3.4.4 Színhely

A játék tárgyait tartalmazzák. Használhatók főmenü, egyéni szintek és bármi más létrehozására. Elképzelhető akár, mint minden egyedi jelenetfájlra egyedi szintként. Minden jelenetben elhelyezhető a környezet, az akadályok és a dekorációk, lényegében darabokra tervezve és felépítve a játékot.

3.5 Adatok Tárolása

A mentés a PlayerPrefs osztály segítségével történik, az osztály képes string, float és integer típusú adatok tárolására a felhasználó platformjának Registry-jében. Az adatok mentésére a SetInt, SetString és a SetFloat függvényeket kell használni, melyek paramétere egy kulcs, amely alapján egyedi adatokat tudunk menteni majd betölteni, illetve maga az adat a megfelelő típussal. A betöltés a GetInt, GetString valamint a GetFloat metódusokkal tehető meg, a paraméterek itt is egy kulcs érték pár, melyek már el lettek mentve a rendszer Registry-ben. Abban az esetben, ha nincsen még elmentve az az adat, akkor PlayerPrefsExeption-t kapunk. A legfontosabb, ami elmentésre kerül, az a játékos teljesítménye. A legfontosabb dolog, ami mentésre kerül, az a játékos azon pályáinak száma, melyeket már teljesített (abban az esetben, ha az újrakezdi a játékot a felhasználó, akkor a pályákra vonatkozó mentéseit is törölni fogja).

Emellett ami szintén nagyon fontos, az a beállítások, melyek szintén elmentésre kerülnek. Ezek egy index formában kerülnek mentésre, hogy minél kevesebb helyet foglaljon. Az Unity ezen beállításai, listákban vannak és az adott sorszámú beállítás indexe kerül mentésre. Például: a felbontás listájában, ha van 10 elem, és mi kiválasztjuk abból a listából az 1920 X 1080 opciót, ami a 9. akkor a kilenc lesz a Registry-ben.

A Windows Registry egy hierarchikus adatbázis, amely alacsony szintű beállításokat tárol a Windows operációs rendszerhez és a beállításjegyzéket használó alkalmazásokhoz. A rendszermag, az eszközillesztő-programok, a szolgáltatások, a Security Accounts Manager és a felhasználói felületek egyaránt használhatják a rendszerleíró adatbázist. A rendszerleíró adatbázis hozzáférést biztosít a rendszerteljesítmény-profilozáshoz szükséges számlálókhoz is.

3.5.1 Verziókezelés

A GitHub egy internetes tárhelyszolgáltatás szoftverfejlesztéshez és verziókezeléshez Git használatával. Minden projekthez biztosítja a Git plusz elosztott verziókezelését, a hibakövetést, a szoftverfunkciók kérését, a feladatkezelést, a folyamatos integrációt. A kaliforniai székhelyű cég 2018 óta a Microsoft leányvállalata.

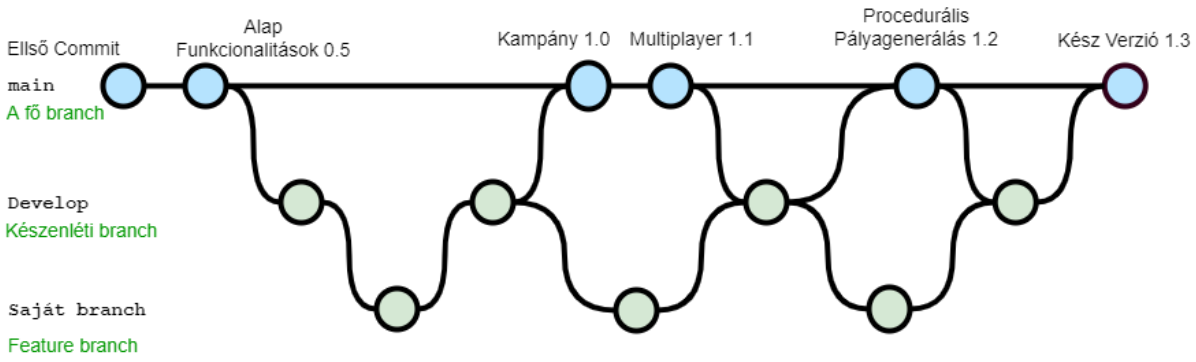
Általában nyílt forráskódú szoftverfejlesztési projektek fogadására használják. 2022 júniusában a GitHub arról számolt be, hogy több mint 83 millió fejlesztővel és több mint 200 millió adattárral rendelkezik, köztük legalább 28 millió nyilvános adattárral. 2021 novemberében ez kezelte a forráskódok legnagyobb részét a világon.

Fontos parancsok a Git verziókezelőben:

- commit: A git commit parancs pillanatképet készít a projekt jelenleg végrehajtott módosításairól. A lekötött pillanatképek a projekt „biztonságos” verzióinak tekinthetők – a Git soha nem fogja megváltoztatni őket, hacsak Ön kifejezetten nem kéri. Használata: git commit -m „commit üzenete”
- add: A git add parancs egy változást ad a munkakönyvtárban az állomásozási területhez. Azt mondja a Gitnek, hogy egy adott fájl frissítéseit szeretné belefoglalni a következő véglegesítésbe. A git add azonban nem igazán befolyásolja a tárat jelentős mértékben – a változtatások ténylegesen nem kerülnek rögzítésre, amíg le nem futtatod a git commit parancsot.
  Használata: git add [mappa vagy fille neve, jelenlegi könyvtár esetében pontot írjunk]

- push: A git push parancsot a helyi lerakat tartalmának távoli tárolóba való feltöltésére használják. A leküldés az a mód, ahogyan a helyi lerakatból a véglegesítéseket egy távoli tárhelyre viheti át. Ez a git fetch megfelelője, de míg az import lekérése a helyi fiókokra vonatkozik, az exportra való kényszerítés a távoli fiókokra. Használata: git push origin [CélBranch]
- merge: Az egyesülés a Git módszere arra, hogy újra összerakja az elágazó történelmet. A git merge parancs lehetővé teszi a git ág által létrehozott független fejlesztési vonalak felvételét és egyetlen ágba való integrálását. Vegye figyelembe, hogy az alább bemutatott összes parancs összeolvad az aktuális ággal. használata: git merge [InduloBranch] [CélBranch]

Saját projektemben az ábrán látható módon zajlott a verziókezelés:



A projekt egy első commit-tal indult el amiben alap dolgokat push-oltam fel csak a repository-ba. Ezután létrehoztam a Develop branch-et a main-ből, majd a Develop-ból egy feature vagy saját branchet. A Develop azt a célt szolgálta, hogy több feature vagy javítás össze lehessen szedve és ha szükséges akkor a kódban lévő konfliktusokat megoldhassam, illetve, ha valami félresikerült ebben a branch-ben, akkor egy plusz biztonsági falként funkcionált, hogy a jól működő production kódokat a main-ben ne rontsa el.

3.6 Főbb Osztályok a Projektben

3.6.1 MonoBehaviour

Minden Unity script osztály ebből származik, olyan alap metódusai vannak, mint a Start mely a script hívásakor hívódik meg, Update, FixedUpdate ami olyan, mint az update, csak az fps fixen 50-re van állítva a fizikai szimulációkhoz, LateUpdate az Update után hívódik meg

minden frame után, OnGUI a GUI eventek kezelésére, OnDisabled és OnEnabled melyek a script ki és bekapcsolásakor hívódik meg.

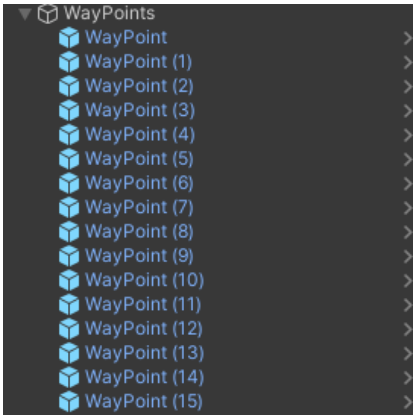
3.6.2 Bullet

Ebben az osztályban található meg a játé lövedék típusainak viselkedése. Az osztály, hogy GameObject-ekhez hozzáadható legyen, a MonoBehaviour osztályból származik. Három fajta bullet típus viselkedése van itt definiálva: sima lövedék, lézer és a rakéta. Mindhárom lövedék saját tulajdonságok alapján sebzi az ellenséget, a lézer lassít egy ellenfelet és folyamatosan csökkenti annak életét. Sima lövedék egy konstans sebességgel halad a célja felé, majd, ha elérte, sebzést okoz neki, valamint a rakéta, mely becsapódáskor egy adott sugarú körben sebez minden ellenfelet, valamint, ha az ellenség meghalna mielőtt becsapódik, akkor elkezd körözni, míg egy újabb ellenfél nem ér sugarába.

3.6.3 Enemy

A script implementál három metódust melyet a Bullet osztály használ, Slow, Die és TakeDamage. Emellett az osztály leírja minden ellenség alap tulajdonságait, mint az élet mennyisége, mennyit ér, milyen gyors, mi a neve, illetve néhány további fontos GameObject-et mint a megsemmisülésekor létrejövő effekt, illetve az életcsík az ellenség felett.

3.6.4 EnemyMovement



Az ellenségek mozgását implementáló script, mely a pályán lévő WayPoint-okat szedi össze és sorban irányítja őket pontról pontra, valamint, ha az utolsó ponthoz ér, akkor levonja a játékos életét.

3.6.5 BuildManager

Egy Singleton osztály, hogy a projektben mindenhonnan könnyedén elérhető legyen és mivel nincsen szükség több példányra belőle. Ebben az osztályban kezelhetők a tornyok építéséhez szükséges GameObject-ek, valamint létrehozásuk. Az osztályban vannak olyan attribútumok, mint az építés és eladás effektje, melyik tornyot építjük, melyik helyre, a toronyhoz tartozó UI, melyen szerepel az eladás.

3.6.6 CameraControl

Az osztály definiálja a kamera mozgását, hogy milyen gyorsan lehessen közelíteni és távolítani, valamint a pályán milyen gyorsan lehet mozogni a kamerával. A szokásos WASD-vel tudunk mozogni előre-jobbra-hátra-balra és a görgővel tudunk nagyítani és kicsinyíteni.

3.6.7 GameMaster

A játék állapotát kezeli az osztály, ha vége van a játéknak mert elvesztette a játékos, vagy megnyerte, akkor a megfelelő UI-t megjeleníti

3.6.8 PlayerStats

A játékos alap adatait tartja számon az osztály melyben statikusan vannak tárolva az adatok a könnyebb hozzáférés miatt.

3.6.9 Settings

Az osztály menti, betölti (Lásd Adatok Tárolása) és beállítja a megadott beállításokat.

3.6.10 Turret

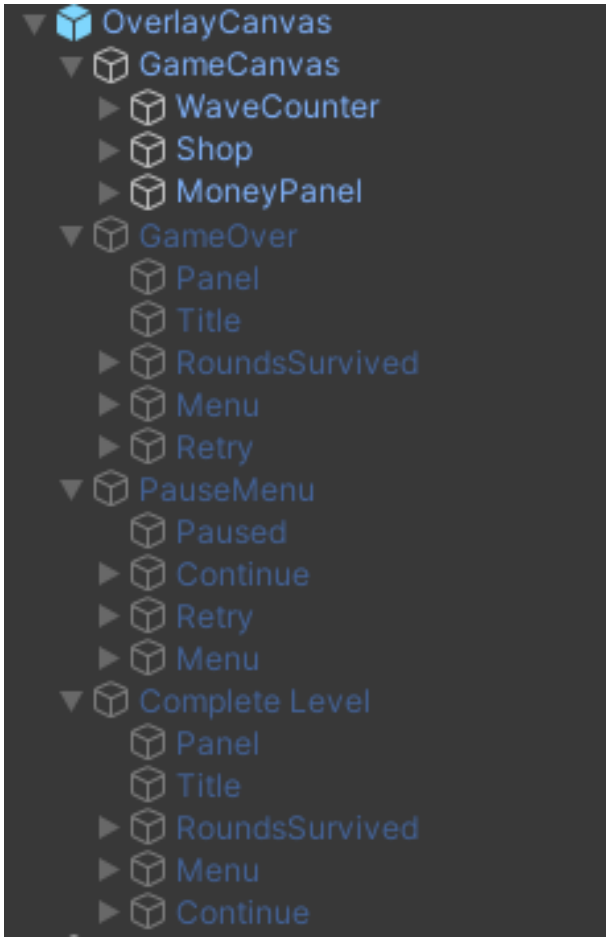
A script leírja a három fajta torony tulajdonságait, mint hogy hogyan és mekkorát, esetleg mekkora körben sebez, hogyan és milyen sebességgel mozog. Emellett az osztályban van meghatározva, az a metódus mely kirajzolja, hogy mekkora körben tudja sebezni az ellenségeket a torony.

3.6.11 MapBrain

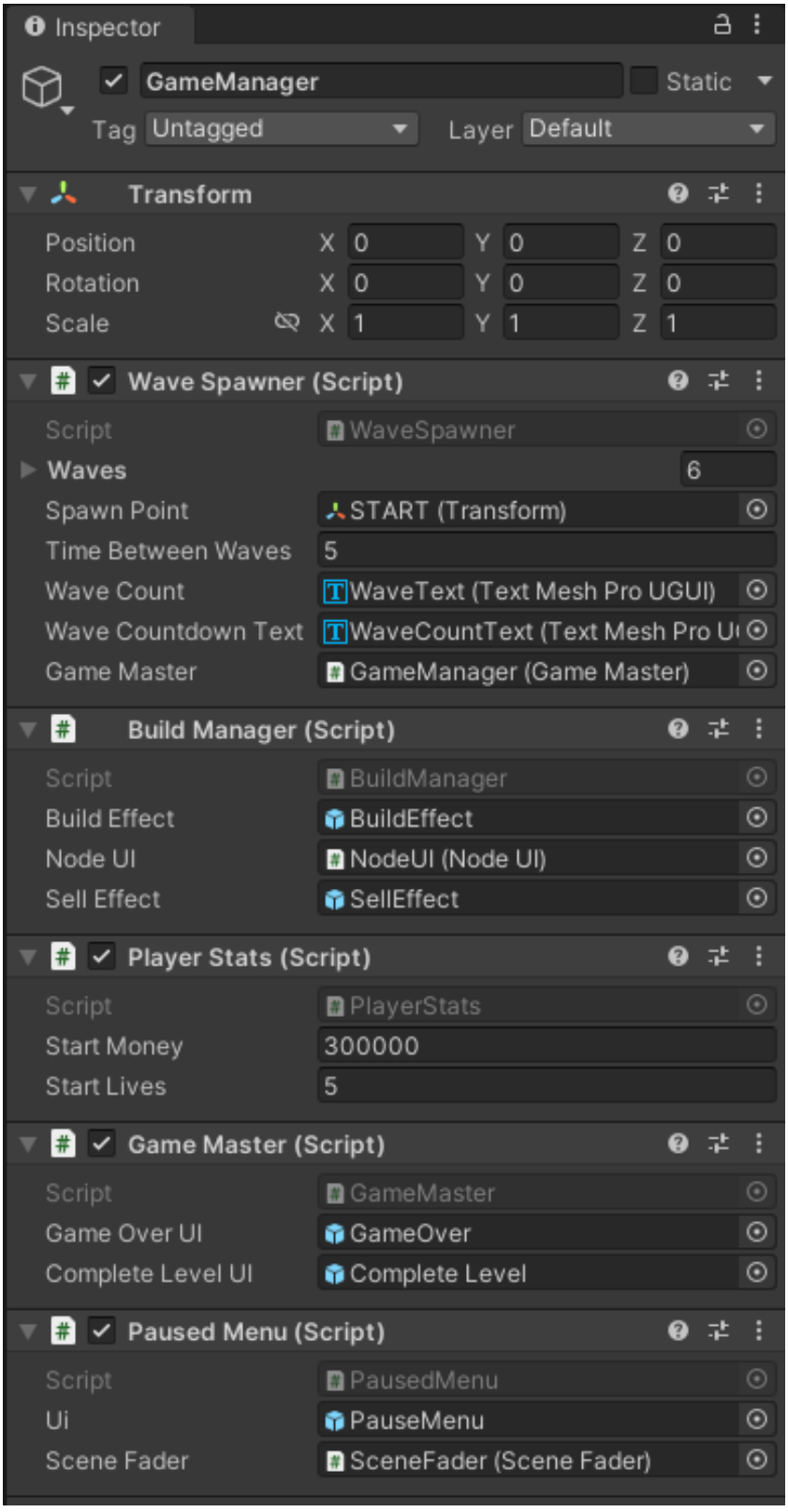
A játék számomra legérdekesebb része itt található. A procedurális pályageneráláshoz szükséges genetikus algoritmus, a rulettkerés kiválasztás és a keresztezés is.

3.7 Színhelyek Fontos GameObject-jei

3.7.1 Pályákon



- GameManager, melyben minden pályán szereplő alap adatok vannak beállítva. A Wave Spawner script a kampány módban van csak, az végtelen játékmódban van egy generáló script. Itt meg kell adni mekkora és milyen hullámok jönnek illetve a GUI-hoz szükséges dolgok, mint például a panelek, melyek jelzik, hogy hányadik hullámnál járunk, mennyi idő van még vissza a következő hullámig, vagy azon menük melyek megjelennek, ha a játéknak vége van, valamint, hogy a játékos mennyi pénzel és élettel kezd.
- OverlayCanvas, egy nagyon fontos elem minden pályán, ez egy 2D-s elem ami a játékos képernyőjén jelenik meg. Ebben található a játékvégi menü, a szünet menüje illetve az álltalános menü mely jelzi az alap adatokat mint a pénz, a hullám száma illetve a bolt melyben a tornyokat tudjuk megvenni. 3.8 Procedurális pályagenerálás



A játékban elérhető egyegy mód, melyben a játékos minden alkalommal egy teljesen új, procedurálisan létrehozott pályán tud játszani. A játékmód a ChessMaze algoritmus és az A* keresőalgoritmuson alapszik. Az A* kereső algoritmus megpróbálja megkeresni a legrövidebb utat. Ez önmagában nagyon jó, de nem lesz élvezhető a játék, ha a pályán mindig

a legrövidebb utat használjuk. Tegyük fel, hogy a kezdő és végpontok egymás mellett vannak egy nagy pályán, ha itt a legrövidebb utat keressük meg akkor a játék nagyon nehéz és élvezhetetlen lesz. A ChessMaze algoritmus erre egy megoldást próbál adni úgy, hogy tetszőleges helyeken a pályán, a nevéből fakadóan, ló sakkbábukat helyez el, melyek akadályként szolgálnak. A ló elnevezés pedig onnan jön, hogy megnézi, hogy az elhelyezett helyeken a lovak hova tudnak lépni és oda is akadályokat helyeznek el, hogy a pályán a legrövidebb út hosszabb, érdekesebb és élvezhetőbb legyen. A megoldás így nem teljesen tökéletes még viszont, mivel nem minden alkalommal biztos, hogy meg tudja találni a legrövidebb utat, vagy hogy az út nem lesz túl rövid és járja be a pályát. Erre a problémára pedig a genetikus algoritmust használtam fel.

3.8.1 A* Algoritmus

Az A* egy informált keresőalgoritmus, avagy legjobbat először keresés, vagyis súlyozott gráfokkal van megfogalmazva: a gráf egy adott kezdőcsúcsából kiindulva arra törekszik, hogy olyan utat keressen az adott célcsomóponthoz, amelynek a legkisebb a költsége (a legrövidebb távolság, a legrövidebb idő stb.). Ezt a kezdőcsúcsból induló utak fájának karbantartásával és az utakhoz az élek egyenkénti hozzáfűzésével teszi, amíg el nem éri a terminálási feltételét.

A fő ciklus minden iterációjánál az A*-nak meg kell határoznia a kiterjesztendő utat. Ehhez az út költségét és a cél eléréséhez szükséges becsült költséget veszi figyelembe. Pontosabban, az A* kiválasztja az {f(n)=g(n)+h(n)}

függvényt minimalizáló utat, ahol n az úton található következő csúcs, g(n) a kezdőcsúcstól n-ig tartó út költsége, és h(n) egy heurisztikus függvény, amely az n-től a célig vezető legolcsóbb út költségét becsli. Az A* akkor fejeződik be, amikor a kezdőcsúcstól a célcsúcsig vezető utat próbálná kiterjeszteni, vagy ha nincsenek kiterjesztendő utak. A heurisztikus függvény problémaspecifikus. Ha a heurisztikus függvény megengedhető (azaz soha nem becsüli felül a cél eléréséhez szükséges tényleges költségeket), akkor az A* garantáltan a kezdőcsúcstól a célcsúcsig vezető optimális utat adja meg.

Az A* tipikus megvalósítása egy prioritásos sort alkalmaz a kiterjesztendő, minimális (becsült) költségű csúcsok ismételt kiválasztásához. Ezt a prioritásos sort nyílt halmaznak vagy peremnek nevezzük. Az algoritmus minden lépésénél a legkisebb f(x) értékű csomópontot eltávolítja a sorból, a szomszédainak f és g értékeit ennek megfelelően frissíti, és

ezeket a szomszédokat hozzáadja a sorhoz. Az algoritmus addig folytatódik, amíg a célcsúcs alacsonyabb f értékkel nem rendelkezik, mint a sorban lévő bármelyik csomópont (vagy amíg a sor ki nem ürül). A cél f értéke ekkor a legrövidebb út költsége, mivel h értéke nulla a célcsúcsban megengedhető heurisztika esetén.

Az eddig leírt algoritmus csak a legrövidebb út hosszát adja meg. A tényleges lépések sorrendjének megtalálásához az algoritmus könnyen módosítható úgy, hogy az útvonalon lévő minden csúcsban eltároljuk a szülőcsúcsát. Az algoritmus terminálása után a célcsúcs az elődjére mutat, és így tovább, amíg valamelyik csúcs elődje a kezdőcsúcs.

Például amikor a térképen a legrövidebb utat keressük, a h(x) képviselheti a célhoz vezető légvonalbeli távolságot, mivel fizikailag ez a lehető legkisebb távolság a két pont között.

Ha a h heurisztika kielégíti a $h(x) \leq d(x, y) + h(y)$ kiegészítő feltételt a gráf minden (x, y) élére (ahol d az adott él hosszát jelöli), akkor h monoton vagy konzisztens. Konzisztens heurisztikával garantált, hogy az A* optimális utat talál egy csomópont többszöri feldolgozása nélkül, és A* egyenértékű Dijkstra algoritmusának futtatásával a $\{d'(x, y)=d(x, y)+h(y)-h(x)\}$ csökkentett költséggel.

3.8.2 Genetikus Algoritmus

3.8.2.1 Definíció

Genetikus algoritmusok alatt olyan keresési technikák egy osztályát értjük, melyekkel optimumot vagy egy adott tulajdonságú elemet lehet keresni. A genetikus algoritmusok speciális evolúciós algoritmusok, technikáikat az evolúcióból kölcsönözték.

3.8.2.2 Módszertan

A genetikus algoritmusokat számítógépes szimulációkkal implementálják. A keresési tér elemei alkotják a populáció egyedeit, melyeket keresztezni (más szóval újra kombinálni) és mutálni lehet, így új egyedek hozhatók létre. A keresési téren értelmezett célfüggvényt ebben a kontextusban szokásos fitnesz függvénynek is nevezni. A genetikus algoritmus működése során egyrészt új egyedeket hoz létre a rekombináció és a mutáció operátorokkal, másrészt kiszűri a rosszabb fitnesz függvény értékkel rendelkező egyedeket és eltávolítja a populációból. Egyes esetekben az ilyen algoritmusok konvergálnak az optimumhoz.

3.8.2.3 Részei

A genetikus algoritmusok sokfélék lehetnek, de az alábbi részeket mindig tartalmazzák:

3.8.2.4 Inicializáció

A kezdeti populációt legegyszerűbb véletlenszerűen generálni. A populáció mérete a probléma természetétől függ, de leggyakrabban néhány száz vagy néhány ezer egyedből áll. Hagyományosan az egyedek a keresési téren egyenletesen oszlanak el, viszont egyes esetekben olyan részeken több egyedet generálnak, ahol sejthető az optimum.

3.8.2.5 Kiválasztás

Minden sikeres generációban a jelenlegi populáció egy része kiválasztásra kerül szaporodásra. Általában fitnesz alapján történik, ahol a fittebb egyedek (a fitnesz függvény szerint) valószínűbben kerülnek kiválasztásra. Bizonyos metódusok minden egyed fitnesz-ét megnézik és választják ki a legjobbat, de más metódusok csak néhány, véletlen példányt néznek meg, mert a teljes folyamat túl hosszú lenne.

A fitnesz függvény a példány *minőségét* méri. A függvény mindig probléma függő.

Néhány problémánál nehéz, akár lehetetlen definiálni a fitnesz számolás műveletét; ilyenkor a példány fenotípusát is használhatjuk, vagy akár interaktív kiválasztást is használhatunk.

3.8.2.6 Szaporítás

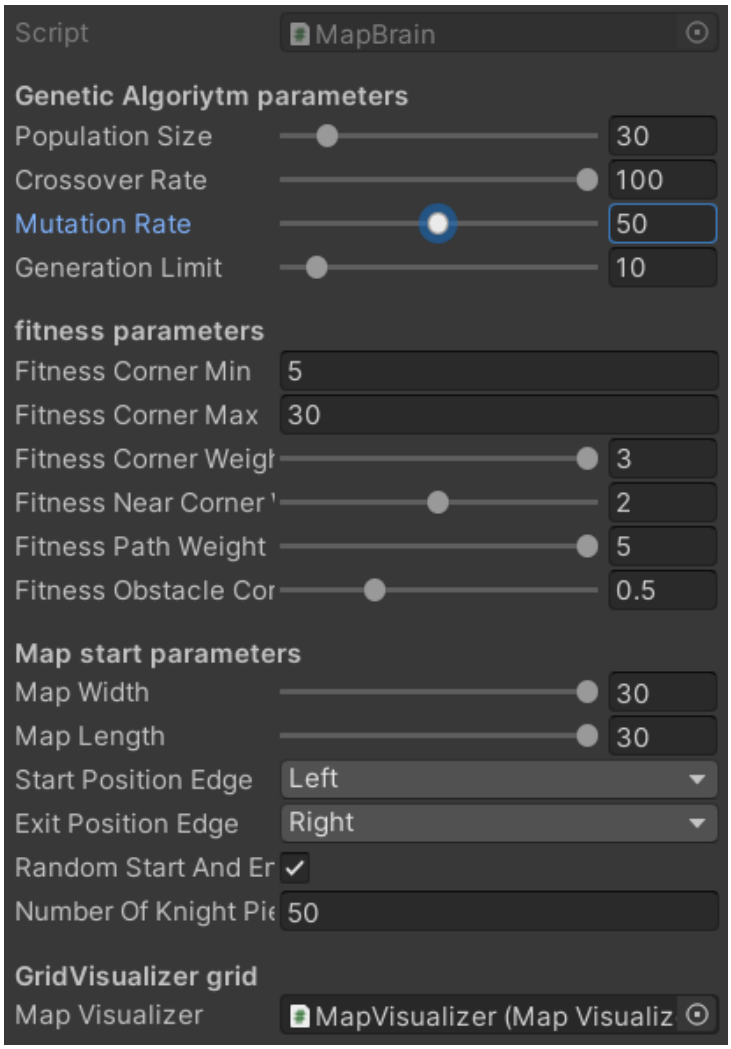
Egyedekből újabb egyedeket a kétoperandusú keresztezés (vagy rekombináció) művelettel és az egyoperandusú mutáció művelettel lehet előállítani. Ezeket az operátorokat általában véletlenszerűen alkalmazzák.

3.8.2.7 Leállás

A genetikus algoritmusok rendszerint addig futnak, amíg egy leállási feltétel nem teljesül. Gyakori leállási feltételek a következők:

- Adott generációszám elérése.
- Ha a legjobb egyed fitnesz értéke már nem javul jelentős mértékben egy-egy iterációval

3.9 Genetikus Algoritmus Unity-ben



A generikus algoritmus paraméterei azt szabályozzák, hogy milyen hányszor generáljon pályát és hogy hány keresztezés legyen, hogy a következő generáció létrejöhessen.

- **„Population Size”:** népesség szám, amely meghatározza, hogy menni pálya fog szerepelni egy adott generációban.
- **„Crossover Rate”:** meghatározza, hogy a keresztezés hány százalékban hajtódjon végre. 100 jelenti, hogy a keresztezés 100%-a végbe fog menni.
- **„Mutation Rate”:** a mutációt korlátozza, hogy hányszor hajtódjon végre a gyerekek keresztezése után. 50 jelenti azt, hogy 50% esély van rá, hogy mutálódni fog a keresztezés után a gyerek.

Fitnesz paraméterek abban segítenek, hogy ki lehessen értékelni a létrejött pályákat, hogy mennyire hasonlít arra, amire szeretnénk. A „weight” avagy súly paraméterek a fontosságát

adják meg, hogy legyen vagy ne legyen a pályán. Ezekkel az értékekkel lehet manipulálni, hogy milyen legyen az út és mennyire legyen hosszú, vagy járja be a pályát.

- **„Fitness Corner Min”:** minimum ennyi kanyarnak szerepelnie kell a pályán.
- **„Fitness Corner Max”:** maximum ennyi kanyar lehet a pályán.
- **„Fitness Corner Weight”:** a kanyar súlya megszabja, hogy mennyire legyen sok kanyar.
- **„Fitness Near Corner Weight”:** az egymáshoz közeli kanyarok súlya szabályozza, hogy mennyire sok olyan hely legyen az úton, ahol sok kanyar van egymás után.
- **„Fitness Path Weight”:** az út súlya segít olyan utat létrehozni, ami nem csak nagyon sok kanyarból és rövid egyenesekből áll, hanem vannak benne hosszabb egyenesek is.
- **„Fitness Obstacle Weight:”:** az akadályok súlya megadja, hogy mennyire fontos az, hogy mennyi akadály van a pályán.

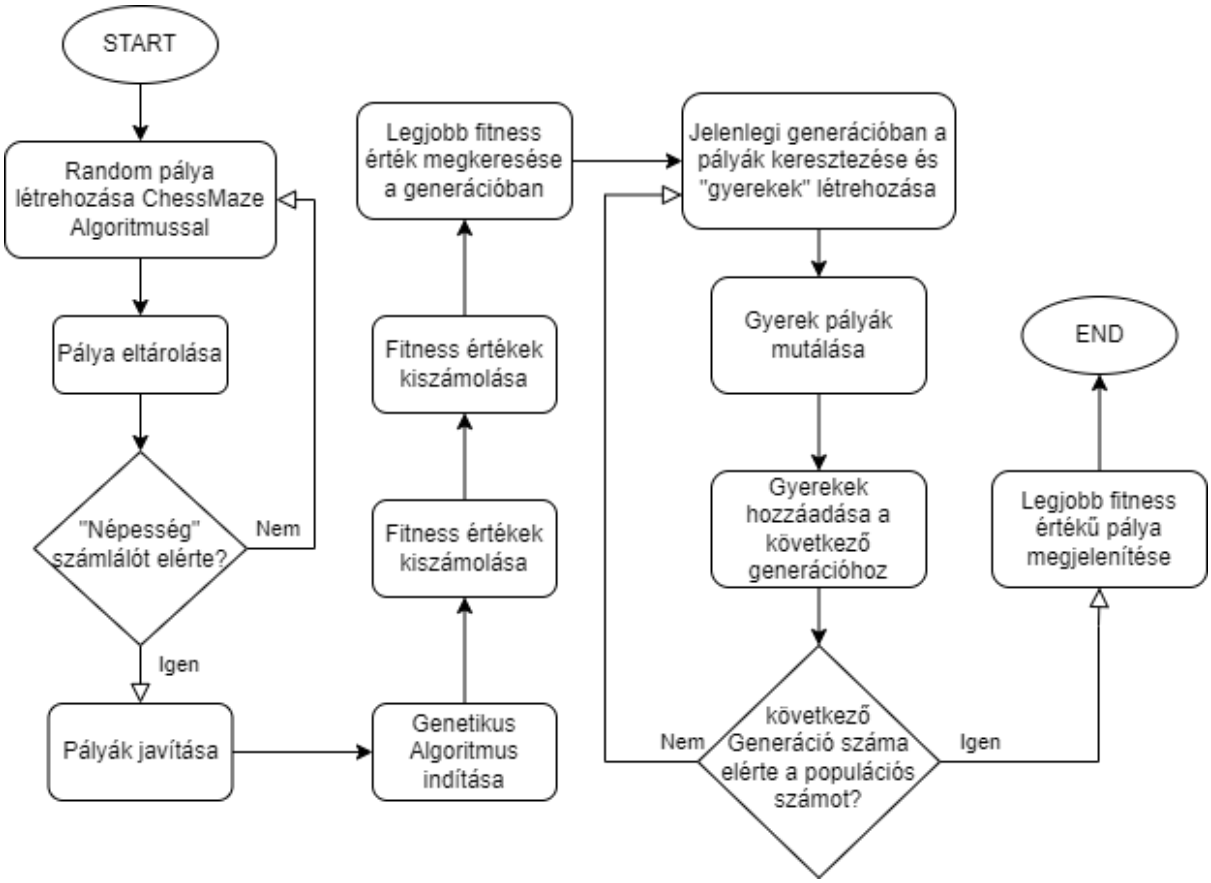
Pálya kezdeti értékek

- **„Map Width”:** a pálya szélességét lehet megadni.
- **„Map Length”:** a pálya hosszúságát lehet megadni
- **„Start Position Edge”:** a kezdőpont melyik szélén legyen a pályának
- **„Exit Position Edge”:** a végpont melyik szélén legyen a pályának
- **„Random Start and Exit Position”:** véletlenszerűen kiválasztható, hogy hol legyen a pálya kezdő és végpontja.
- **„Number of Knight Pieces”:** mennyi ló sakkfigura legyen letéve a pályára, szabályozva ezzel, hogy mennyi helyre nem építhet a játékos és segítve a teljes pálya bejárását.

3.10 Generikus Algoritmus

A ChessMaze algoritmussal generálunk annyi pályát és tesszük bele egy listába, amíg a lista mérete el nem éri a népesség számot, mely egy előre definiált paraméter, mely meghatározza, hogy mennyi pálya legyen egy generációban, majd minden pályát szükség esetén megjavítunk és megkeressük a legrövidebb utat. Ezt követően elindítjuk az első generációval az algoritmusunkat. Kiszámoljuk az összes pálya fitnesz értéket és elmentjük, hogy melyik a legjobb. Ezután elkezdjük létrehozni a következő generációnkat. A jelenlegiben kiválasztunk 2 random pályát a Rulettkerék kiválasztással, melyek legyenek A és B szülők. A

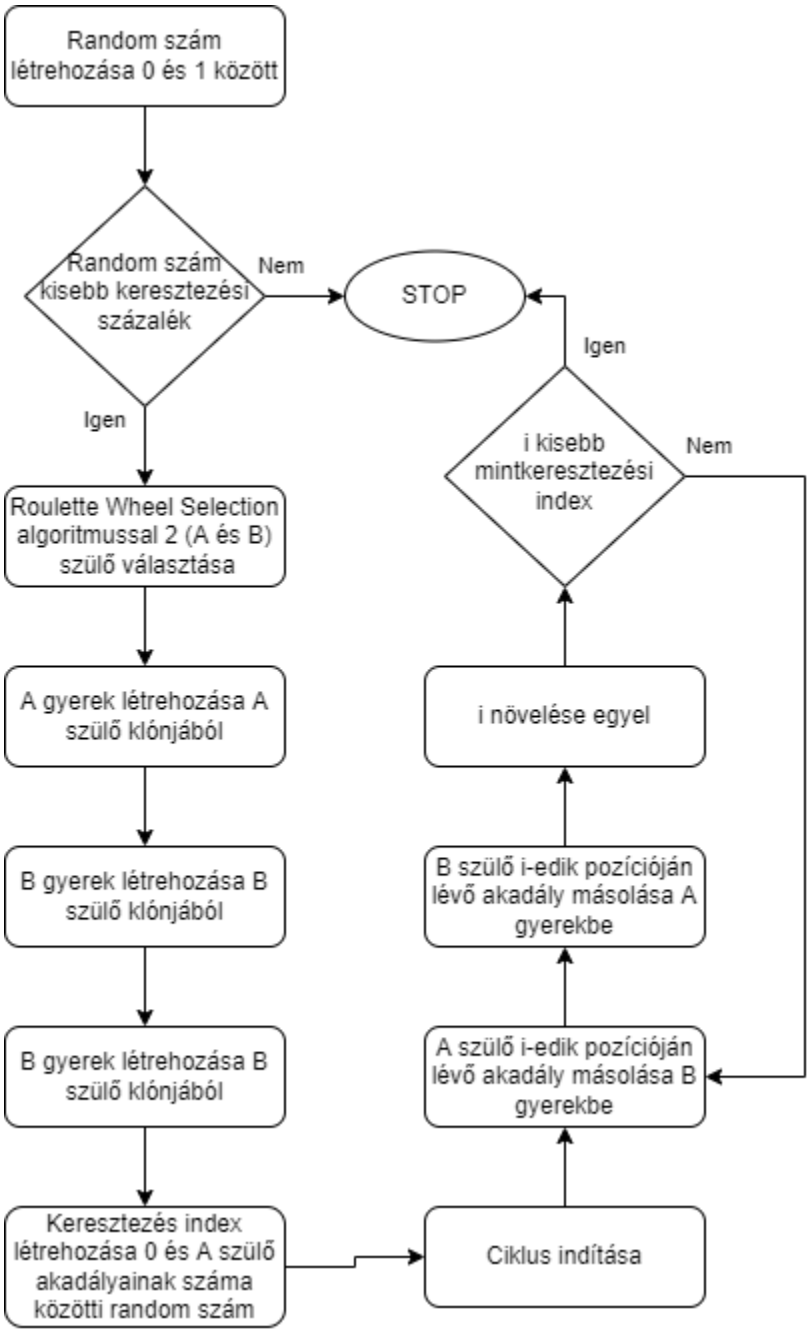
kettő kiválasztott pályát klónozzuk, így létrejön A szülőből A gyerek és B szülőből B gyerek. Ebben a fázisban a szülő és a gyerek egy és ugyan azon pályaadatokkal rendelkezik. Kiválasztás után megkezdjük a keresztezését a négy pályának. A pályák keresztezése után A és B gyerekeken el kell végezni a mutációt, hogy egy új random faktor megjelenjen az algoritmus eredményében, valamint, hogy az algoritmus ne ragadjon meg egy szinten, hanem legyen lehetőség magasabb fittnesz értékkel rendelkező pályát létrehozni. Az így létrejött és módosult A és B gyereket eltároljuk a következő generáció listájában. A keresztezést és a mutálást addig csináljuk amíg el nem érjük a következő generációban a népességszámot. Ha elértük a limitet a következő generációban is, akkor beállítjuk a jelenlegi generációt a következőre és újrakezdjük a folyamatot. Az új generációk gyártása egészen addig folytatódik, amíg nem érjük el a generációs határat.



3.10.1 Keresztezés

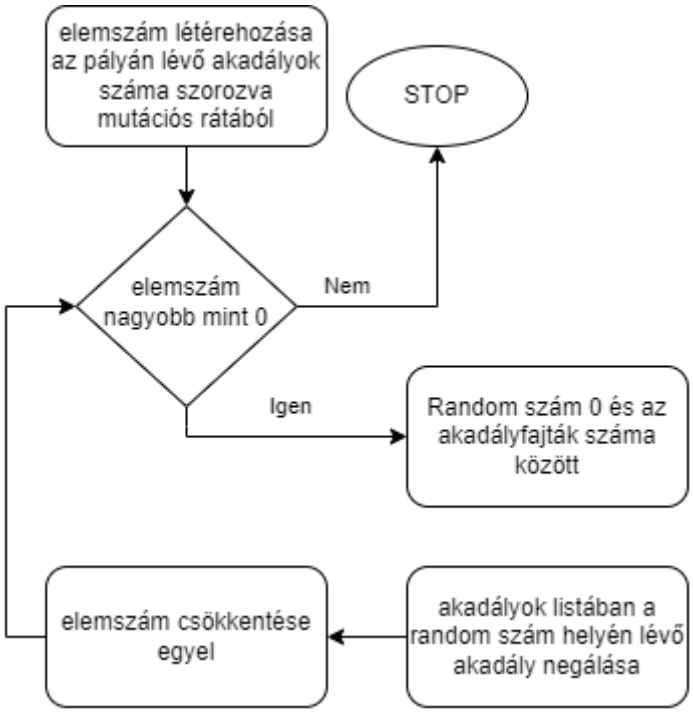
A keresztezés folyamata azzal kezdődik, hogy kiválasztunk egy random számoz 0 és 1 között, és ha ez a szám kisebb, mint a keresztezési ráta akkor megállunk és nem folytatjuk

tovább, kilépünk az algoritmusból. Abban az esetben, ha nagyobb, akkor a Rulettkerék kiválasztó algoritmussal választunk a jelenlegi generációból 2 pályát, melyek A és B szülők lesznek. A kiválasztott pályákat klónozzuk le, A szülőből lesz A gyerek és B szülőből lesz B gyerek. Létrehozunk egy random számot 0 és az A szülő akadályainak száma között és elindítunk egy ciklust i=0-tól a létrehozott random számunkig, a keresztezési indexig. Az A szülő pálya i-edik indexén lévő pályaelemet lemásoljuk a B gyerek ugyan azon helyen lévő elemére, majd a B szülő i-edik indexén elhelyezkedő elemet lemásoljuk az A gyerek i-edik helyén lévő objektumra és növeljük az i-t. Ha i elérte a keresztezési indexet, akkor a keresztezés algoritmus végzett, ha nem akkor a következő ciklus elindul.



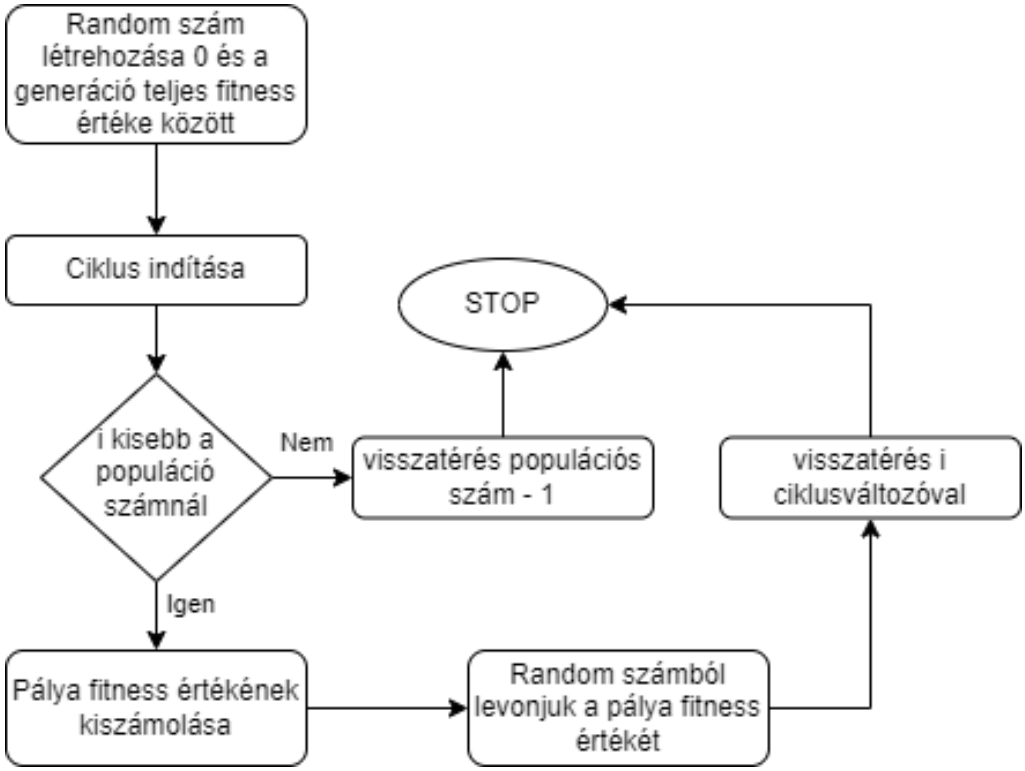
3.10.2 Mutáció

A mutációs művelet elengedhetetlen a genetikai algoritmusban. Megváltoztatja egyes gének értékeit az új gyermekek minőségének javítása érdekében. Annak eldöntésére, hogy egy gén mutált-e vagy sem, a mutációs valószínűséget használjuk. A hagyományos genetikai algoritmusban csak egyetlen állandó érték van a mutáció valószínűségére.



3.10.3 Rulettkerék Kiválasztás

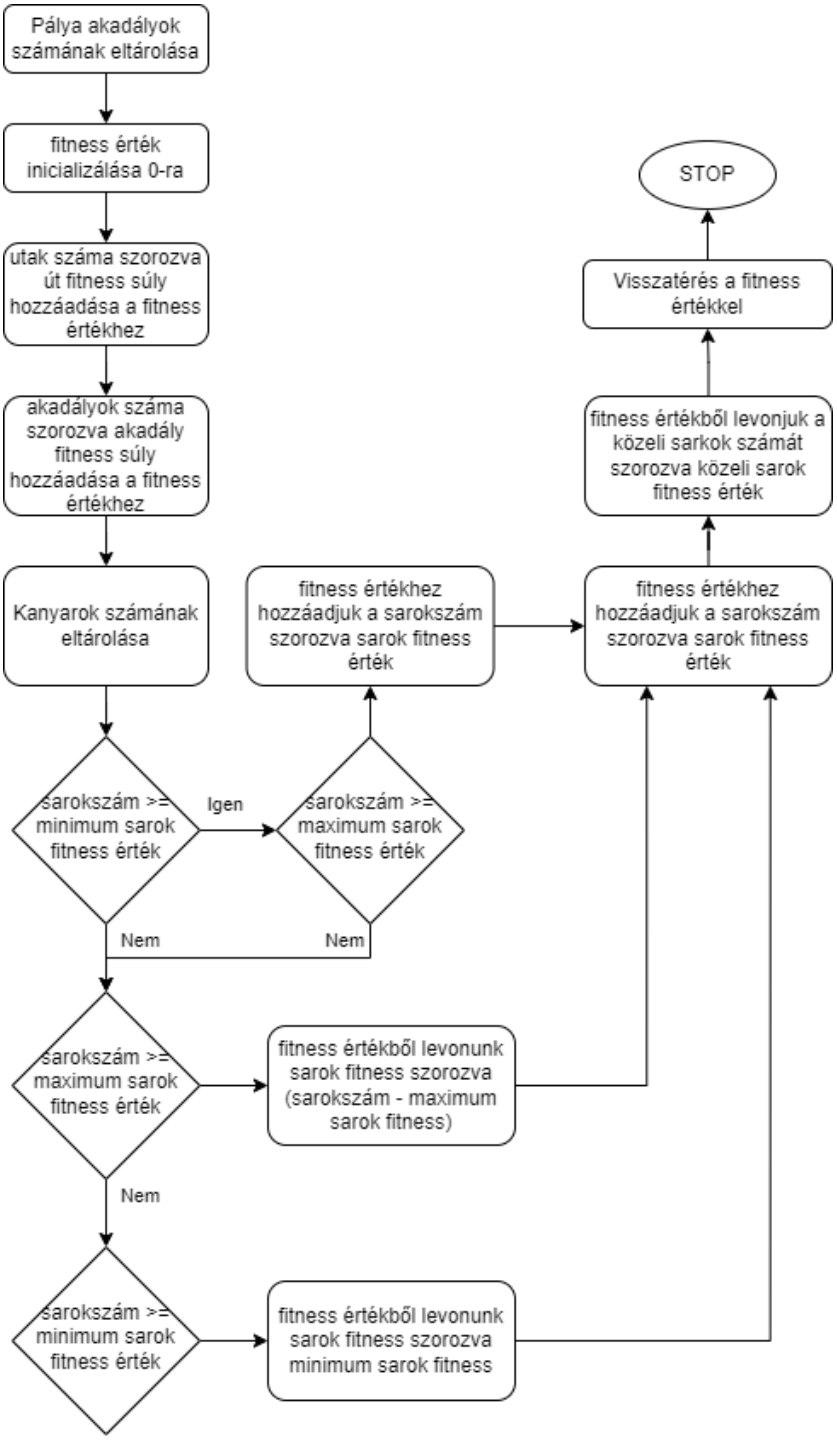
A rulett kerék kiválasztási módszere az összes egyed kiválasztására szolgál a következő generáció számára. Ez egy népszerű szelekciós módszer, amelyet a genetikai algoritmusban használnak. A rulettkerék az egyes egyének relatív fittségéből (az egyéni fittség és a teljes fittség arányából) épül fel.



Véletlenszerűen, létrehozunk egy számot 0 és a legjobb fitnesz érték között és elindítunk egy ciklust i = 0-tól a népesség számig. A random számból levonjuk a jelenlegi generáció i-edik pályájának a fitnesz értékét és ha az így kapott szám kisebb, vagy egyenlő mint 0, akkor visszatérünk i-vel. Ellenkező esetben növeljük az i értékét egyel és elindítjuk a következő ciklust. Ha elértük a népesség szám-t, és a random szám értéke továbbra is nagyobb 0-nál, akkor visszatérünk a népesség szám mínusz egyel és az algoritmus leáll.

3.10.4 Fitness Érték Kiszámolása

A legjobb fitnesz görbe mindig az átlagos fittségi görbe "felett" lesz. Ha a fitnesz, avagy a generációk görbéje csökken, akkor ez általában azt jelenti, hogy a genetikus algoritmus valószínűleg nem tárja fel megfelelően a megoldási teret, és ez jellemzően a nem megfelelő mutációs és keresztezési műveletekre vezethető vissza.


Pálya akadályok számának eltárolása
fitness érték inicializálása 0-ra
utak száma szorozva út fitness súly hozzáadása a fitness értékhez
akadályok száma szorozva akadály fitness súly hozzáadása a fitness értékhez
Kanyarok számának eltárolása
sarokszám >= minimum sarok fitness érték
Igen
sarokszám >= maximum sarok fitness érték
Nem
Nem
fitness értékhez hozzáadjuk a sarokszám szorozva sarok fitness érték
sarokszám >= maximum sarok fitness érték
fitness értékből levonunk sarok fitness szorozva (sarokszám - maximum sarok fitness)
Nem
sarokszám >= minimum sarok fitness érték
fitness értékből levonunk sarok fitness szorozva minimum sarok fitness
fitness értékhez hozzáadjuk a sarokszám szorozva sarok fitness érték
fitness értékből levonjuk a közeli sarkok számát szorozva közeli sarok fitness érték
Visszatérés a fitness értékkel
STOP

4. Fejezet

4.1 Összefoglalás

A szakdolgozat feladata egy olyan többszemélyes játék készítése, amelyhez nincsen szükség a játékosok egy hálózaton léte, játszható egyedül is, van benne lehetőség egy olyan módra, amelyben nincsen megállás, csak akkor, ha a játékos minden élete elfogy. Ezeken felül, hogy legyen benne egy random pálya mód, melyben a játékos minden alkalommal egy teljesen új, procedurálisan létrehozott pályán tud játszani. Sok mindent tanultam mind a játékfejlesztéssel, mind programozási problémák megoldásával kapcsolatban a szakdolgozatom készítése közben. C# és Unity tudásom sokkal előrehaladottam az egy évvel ezelőtti állapotához képest és szeretném a jövőben is ezt a tudásom gyarapítani. Egy játékot mindig lehet tovább fejleszteni, bővíteni, finomítani. Sok erre irányuló ötlet merült fel bennem a fejlesztés során, azonban ezekre idő hiányában nem jutott lehetőség implementálni, de szívesen megvalósítanék még sok funkciót a jövőben.

4.2 Továbbfejlesztési lehetőségek

- Procedurálisan generált pályák elmentése és betöltése későbbi újra játszásra.
- A betöltött pályák többjátékos módban való használása.
- A játék optimalizálása és hatékonyabb megoldások keresése és megvalósítása.
- Nehézségi fokozatok bevezetése melyben random generált ellenséges hullámok érkeznek.
- Egy teljesen egyedi pályaszerkesztő mód, melyben a játékos hozhatja létre magának a pályát.