Eötvös Loránd Tudományegyetem

Informatikai Kar

Informatikatudományi Intézet

Információs Rendszerek Tanszék

Egyedi 2D platformer játék fejlesztése Unity környezetben, FastAPI alapú szerverrel

Szerző:

Gáspár Róbert

Programtervező informatikus BSc.

Belső konzulens:

Kotroczó Roland

Egyetemi tanárségéd

Külső Témavezető:

Török Zoltán Ákos

Technikai vezető

Budapest, 2025

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:
Név: Gáspár Róbert
Neptun kód: HVH9YF

Képzési adatok:
Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)
Tagozat : Nappali
Külső témavezetővel rendelkezem

Külső témavezető neve: Török Zoltán Ákos
munkahelyének neve:**Heitec Hungary Kft.**
munkahelyének címe: **1117 Budapest, Kaposvár utca 14-18**
beosztás és iskolai végzettsége: **Senior DevOps Engineer, ELTE IK Bsc**
e-mail címe: **tzoltan.akos@gmail.com**

Belső konzulens neve: Kotroczó Roland
munkahelyének neve, tanszéke: **ELTE-IK, Informaciós rendszerek Tanszék**
munkahelyének címe:**1117, Budapest, Pázmány Péter sétány 1/C.**
beosztás és iskolai végzettsége: **Egyetemi tanársegéd, programtervező informatikus MSc**

A szakdolgozat címe: Egyedi 2D platformer játék fejlesztése Unity környezetben, FastAPI alapú szerverrel

A szakdolgozat témája:
(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben szakdolgozat témájának leírását)

A szakdolgozatomban egy 2D platformer játékot fejlesztek a Unity játékmotor segítségével, amely a Bloody Trapland játékmenetét veszi alapul. A projekt célja, hogy a felhasználók különböző pályákon navigálva leküzdjék az akadályokat, miközben saját ügyességüket és reakcióidejüket próbára teszik. A felhasználóknak lehetőségük van egyedi pályák létrehozására, amelyeken más játékosok is játszhatnak, így a közösségi élmény is hangsúlyt kap. Minden pályához tartozik egy rangsor, amelyben a játékosok teljesítménye alapján rangsorolódnak. Az elkészített pályák nem szerkeszthetők, ezzel biztosítva a verseny tisztaságát és a ranglisták megbízhatóságát. A projekt kulcsfontosságú eleme a FastAPI alapú backend, amely a felhasználói regisztrációk, bejelentkezések és a pályák kezelését végzi. Az alkalmazás az AWS EC2 infrastruktúráján fut, amely a felhőszolgáltatások előnyeit kihasználva biztosítja a rendszer stabilitását és skálázhatóságát. A játék adatainak kezelésére PostgreSQL adatbázist alkalmazok, amely Docker konténerben fut. Ez a megoldás lehetővé teszi a könnyű telepítést és karbantartást, valamint az adatbázis környezetek közötti zökkenőmentes migrációját. A PostgreSQL adatbázis tárolja a felhasználói információkat, a pályákat és a ranglistákat. A FastAPI és a PostgreSQL közötti kapcsolat biztosítja, hogy a felhasználók által létrehozott tartalmak és a teljesítmény statisztikák megbízhatóan és gyorsan elérhetők legyenek. A jelszavak biztonságos titkosítással kerülnek tárolásra, ezzel megvédve a felhasználói adatokat. A projekt célja egy összetett, de felhasználóbarát rendszer megvalósítása, amely szórakoztató és kihívásokkal teli játékélményt nyújt a felhasználóknak. A technológiai megoldások – beleértve a FastAPI-t, PostgreSQL-t és Docker-t – lehetővé teszik a rugalmas és skálázható rendszer kialakítását, miközben a Unity motor biztosítja a játék fejlesztésének hatékonyságát. A fejlesztés során a célom a legmodernebb technológiai megoldások integrálása, amelyek biztosítják a játék hosszú távú fenntarthatóságát és folyamatos bővítésének lehetőségét.

Budapest, 2024. 10. 11.

Tartalomjegyzék

1. Bevezetés

A videójátékok napjaink egyik legnépszerűbb digitális szórakozási formáját jelentik, amelyek az elmúlt évtizedekben jelentős technológiai fejlődésen mentek keresztül. A játékipar fejlődésének egyik mérföldköve a Super Mario megjelenése volt, amely forradalmi hatással volt az iparágra, lefektetve a modern 2D játékok alapjait és irányt mutatva a későbbi fejlesztések számára. A játék sikere gazdasági szempontból is jelentős volt, széles körben ismertté téve a műfajt. A Super Mario által inspirált játékok a klasszikus játékmenetet ötvözik pixel art vizuális stílussal, precíz mozgásmechanikával és fokozatosan nehezedő pályákkal. Ilyen alkotások például a Bloody Trapland és a Super Meat Boy, amelyek lehetőséget adnak a játékosoknak ügyességük és reakcióidejük fejlesztésére, miközben a játékélmény egyszerre kínál kihívást és szórakozást.

Ebből a hagyományból kiindulva, valamint a játékok és a modern fejlesztési eszközök iránti érdeklődésem, továbbá az iparban elterjedt Unity játékmotor nyújtotta lehetőségek inspiráltak szakdolgozatom témájának kiválasztására, amely egy saját 2D pixel art játék fejlesztése.

Célom, egy olyan játék létrehozása, amely megtartja a klasszikus platformerek alapvető játékmenetét, ugyanakkor lehetővé teszi, hogy a felhasználók kreativitásukra építve egyedi pályákat készítsenek, megoszthassák azokat, és versenyre hívhassák egymást. A játék ösztönzi a közösségi interakciót és a kreativitást, miközben élvezetes és kihívásokkal teli élményt nyújt minden játékos számára.

A dolgozat további részei a játék részletes bemutatására és a fejlesztés folyamatának ismertetésére fókuszálnak. A felhasználói dokumentáció részletezi az alkalmazás főbb funkcióit, a telepítési folyamatot, valamint a játék kezelését, biztosítva, hogy a program használata a kezdő felhasználók számára is egyértelmű legyen. Ezt követően a fejlesztői dokumentáció bemutatja a szoftver belső felépítését, az alkalmazott technológiákat, a szervert, az adatbázist, valamint a játék működéséhez szükséges algoritmusokat, továbbá a tesztelési folyamatokat. Az összefoglalásban tömören bemutatom a dolgozat eredményeit és a továbbfejlesztési lehetőségeket.

2. Felhasználói dokumentáció

Ebben a fejezetben a játék használatához szükséges gyakorlati tudnivalók kerülnek bemutatásra. A leírás tartalmazza a rendszerkövetelményeket, a telepítés és indítás lépéseit, valamint a játék kezeléséhez és funkcióihoz kapcsolódó útmutatókat.

2.1. Rendszer és hardware követelmények

A program platformfüggő, így kizárólag Windows operációs rendszeren futtatható. Mivel az alkalmazás erőforrásigénye alacsonyabb, mint a Windows futtatásához szükséges minimális követelmények, ezért a játék futtatásához elegendő, ha a számítógép megfelel a Windows 11 alapvető rendszerkövetelményeinek.

Ajánlott hardware követelmények:

- Operációs rendszer: Windows 11 (64-bites).
- Processzor: 1 GHz-es vagy gyorsabb, legalább 2 magos, kompatibilis 64 bites processzor.
- RAM: 4 GB
- Grafikus kártya: DirectX 12-kompatibilis vagy újabb, WDDM 2.0 típusú illesztőprogrammal.
- Tárhely: 85 MB szabad hely.
- Internetkapcsolat: Folyamatos internetkapcsolat az online funkciók használatához.
- Kijelző: 1920×1080 felbontás, a játék kizárólag 16:9 képarányú kijelzőn jelenik meg megfelelően, mivel a blokkok képaránya nem reszponzív [1].

2.2. Telepítő beszerzése

Az egyszerű telepítés érdekében a játékot az „Inno Setup” programmal készített telepítőcsomag formájában biztosítottam, mely a következő linken érhető el:

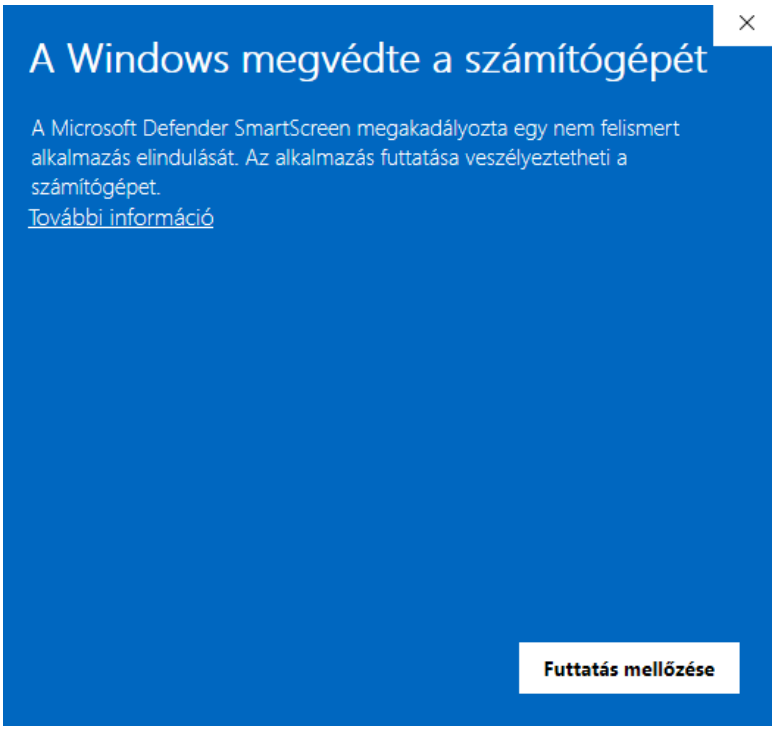
https://github.com/grobi447/Thesis-Client/releases/

A fenti linken a „SweetPain.exe” fájlra kattintva a letöltés automatikusan megkezdődik a számítógépre.

2.3. Telepítés

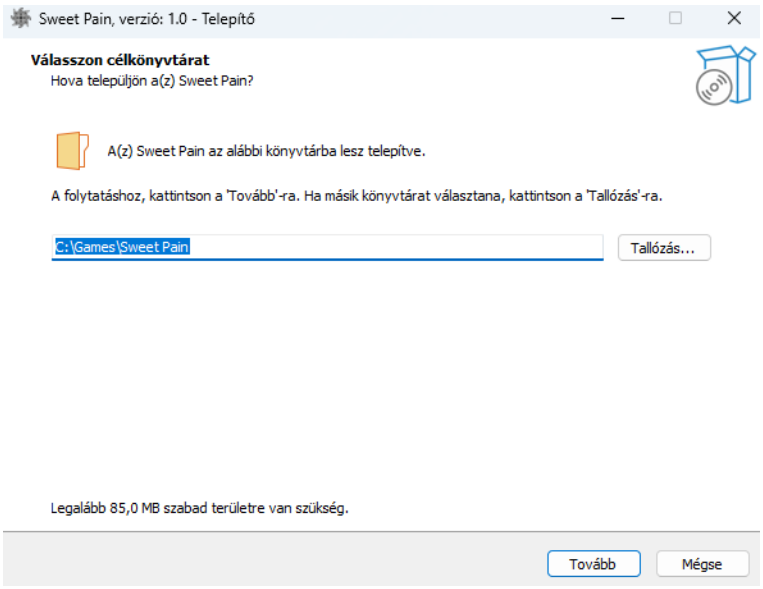
1. ábra: A telepítő állomány

A telepítő állomány megnyitásakor előfordulhat, hogy a Windows Defender (lásd: 2. ábra) figyelmeztetést jelenít meg, mivel a program nem rendelkezik digitális tanúsítvánnyal. Ebben az esetben a „További információ” gombra kattintva elérhető a telepítő futtatásának lehetősége.



2. ábra: Windows Defender figyelmeztetés

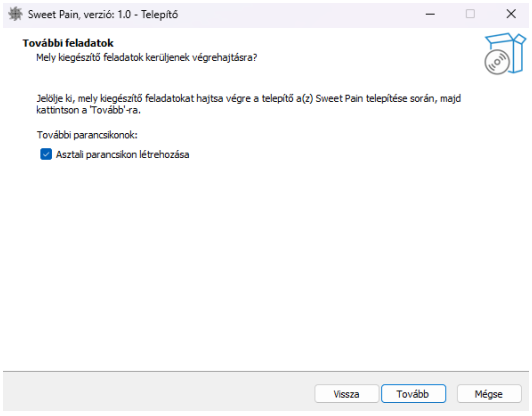
A telepítő megnyitásával megjelennek a nyelvi beállítások, ahol az alapértelmezett magyar mellett az angol is választható. Ezt követően megadható a telepítés célkönyvtára, amelybe a program fájljai kerülnek. Az alapértelmezett útvonal módosításához a „Tallózás” gomb használható.



3. ábra: Célkönyvtár kiválasztása

A „Tovább” gombra kattintva eldönthetjük, hogy készüljön-e asztali parancsikon a programhoz, amely megkönnyíti a későbbi indítást. A következő panelen már csak a telepítés indítása szükséges az erre szolgáló gombbal.

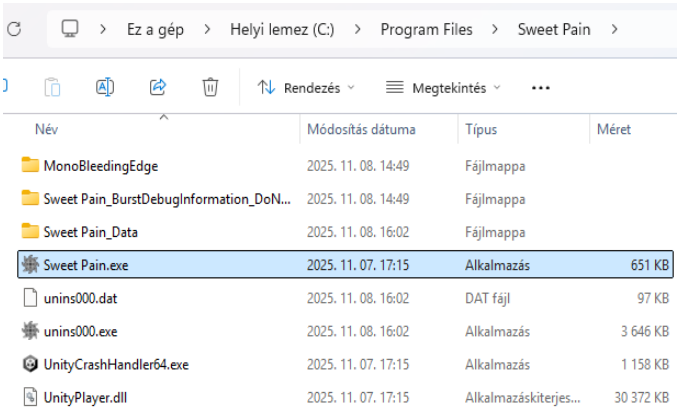
2.4. Indítás



4. ábra: Program indítása telepítőn keresztül

A telepítő befejezése előtt lehetőség nyílik a program automatikus indításának beállítására, melyet az ábrán látható jelölőnégyzet szabályoz. Ha a játékot inkább manuálisan szeretnénk elindítani, az opció kikapcsolható.

A telepítés során létrehozott asztali parancsikon esetén a játék a „Sweet Pain.exe” ikonra kattintva indítható el az asztalról. Parancsikon hiányában a program a telepítéskor megadott célkönyvtárban található „Sweet Pain.exe” fájl futtatásával indítható.



5. ábra: A program mappa tartalma

2.5. Kezdőképernyő

6. ábra: A játék kezdőképernyője

A játék indítását követően a kezdőképernyő jelenik meg, amely az online funkciók elérésének kiindulópontja. A játék használatához minden játékosnak szüksége van egy saját felhasználói fiókra, amelyhez az elért eredmények és az elkészített pályák kapcsolódnak. A felületen lehetőség van új felhasználó regisztrálására vagy meglévő fiókba való bejelentkezésre. Amennyiben a játékos nem kíván bejelentkezni, a „QUIT” gombbal kiléphet az alkalmazásból.

2.5.1. Regisztráció

A „REGISTER” gombra kattintva megnyílik a regisztrációs panel, amely három beviteli mezőt tartalmaz. Az „Enter username” mezőbe a felhasználónév írható, míg a másik két mező a jelszó megadására és megerősítésére szolgál. A sikeres regisztrációhoz a következő feltételeknek kell teljesülniük:

- A felhasználónévnek egyedinek kell lennie, tehát, nem szerepelhet már létező név az adatbázisban.
- A felhasználónév legalább 3 karakter hosszú legyen.
- A két jelszóbeviteli mező tartalma egyezzen meg.
- A jelszó legalább 5 karakter hosszúságú legyen.

Amennyiben bármelyik feltétel nem teljesül, a „REGISTER” gomb megnyomása után a rendszer angol nyelvű hibaüzenetet jelenít meg a képernyő jobb felső sarkában (lásd 7. ábra).

7. ábra: Hibás regisztráció értesítés

Ha minden követelmény teljesül, a regisztráció megtörténik, és a felhasználó a főmenüre kerül átirányításra. Ekkor a rendszer egy sikeres regisztrációt jelző üzenetet jelenít meg.

8. ábra: Sikeres regisztráció

2.5.2. Bejelentkezés

Egy korábban létrehozott felhasználói fiókba a 6. ábrán látható „LOGIN” gombbal lehet belépni. A megnyíló panel két beviteli mezőt tartalmaz: az „Enter username” mezőbe a felhasználónév, míg az „Enter password” mezőbe a jelszó írható, amelyet a regisztráció során adtunk meg. Ha a megadott adatok nem egyeznek az adatbázisban tároltakkal, a rendszer a képernyő jobb felső sarkában hibaüzenetet jelenít meg hasonlóan a 7. ábrához. Biztonsági okokból az üzenet nem részletezi, hogy a felhasználónév vagy a jelszó volt helytelen. Sikeres bejelentkezés esetén a felhasználó automatikusan a főmenübe kerül, ahol a 8. ábrához hasonló értesítés jelzi, hogy a bejelentkezés sikeres volt.

2.6. Főmenü

9. ábra: A játék főmenüje

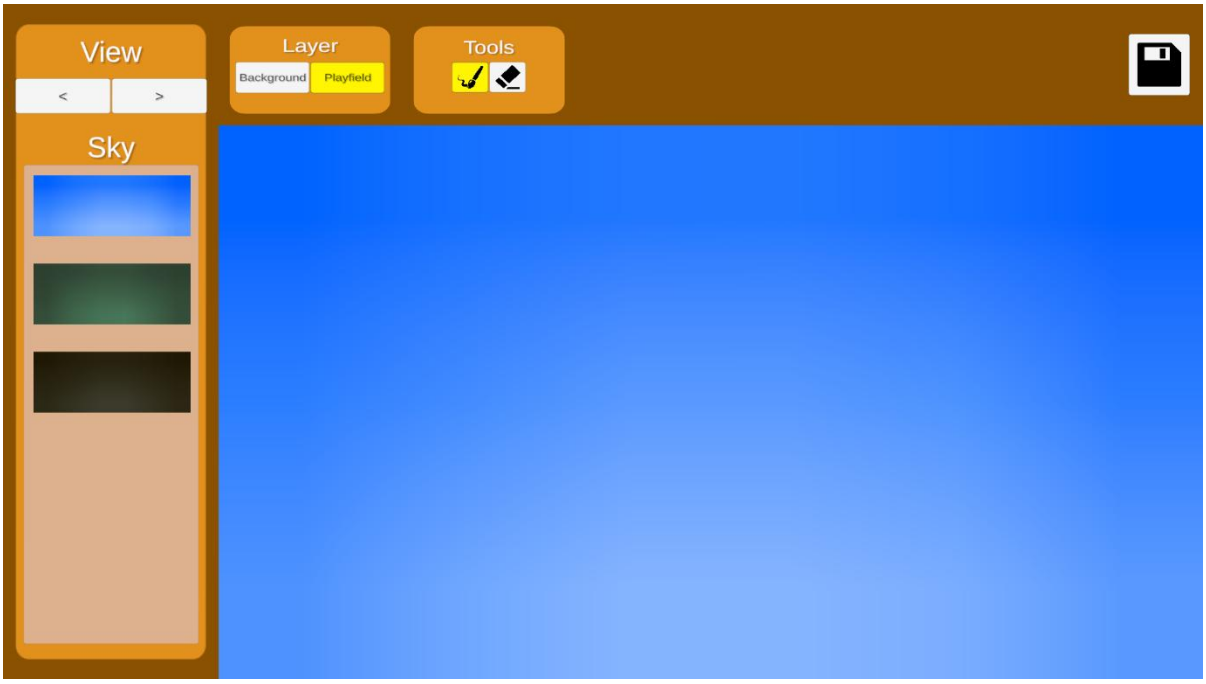
Bejelentkezés után a főmenü jelenik meg, ahol új navigációs gombok válnak elérhetővé. A legfelső „PLAY” gomb a pályaválasztó felületre irányít, míg a „MAPEDITOR” elindítja a

pályaszerkesztőt, ahol saját egyedi pályák hozhatók létre. Amennyiben több felhasználói fiókkal rendelkezünk, a „LOGOUT” gombbal kijelentkezhetünk az aktuálisból. A főmenüben, a kezdőképernyőhöz hasonlóan, lehetőség van az alkalmazás bezárására is.

2.7. Pályaszerkesztő

A pályaszerkesztő elindításakor egy új, interaktív kezelőfelület jelenik meg, ahol a játékos saját pályákat hozhat létre, melyek elérhetővé válnak más felhasználók számára. A szerkesztés teljes szabadságot biztosít, tetszőleges kinézetű és nehézségű pályák készíthetők, minden a játékos kreativitásán múlik. Érdemes azonban figyelembe venni, hogy a pálya ne csak látványos, hanem teljesíthető is legyen, hogy ne romoljon a játékélmény.

2.7.1. Panelek áttekintése

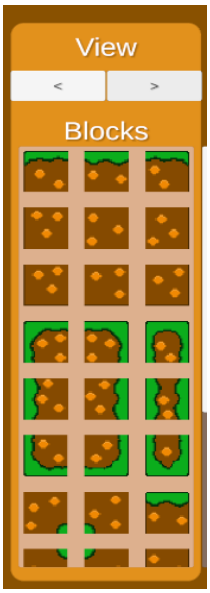


10. ábra: A pályaszerkesztő felépítése

A pályaszerkesztő központi eleme a képernyő közepén elhelyezkedő munkaterület, ahol a szerkesztések és módosítások történnek. A bal oldali panel szolgál a különböző elemek kiválasztására. Itt három különböző nézet („View”) érhető el:

- Sky: Az alapértelmezett nézet, amelyben a játék hátterét lehet megváltoztatni.

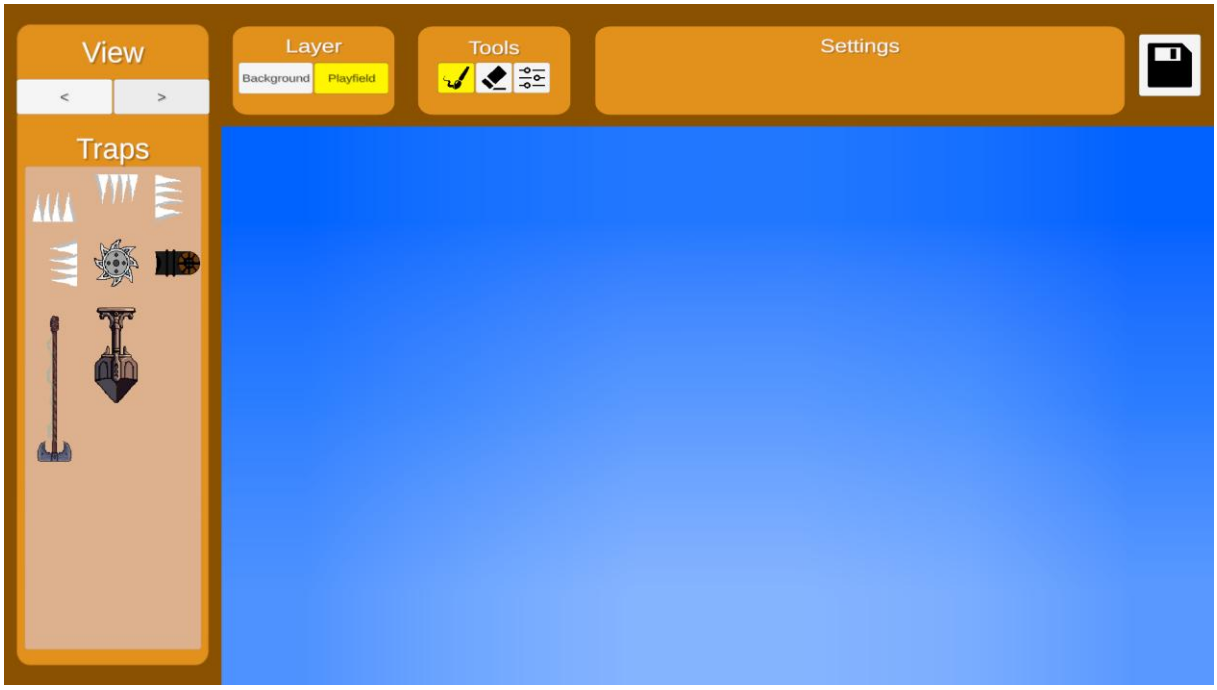
– Block: Ebben a listában találhatók azok a blokkok, amelyek elhelyezhetők a szerkesztőfelületen.
– Trap: A különféle csapdákhoz való hozzáférést biztosítja (lásd 12. ábra).



11. ábra: "Block View" tartalma

A nézetek között a panel tetején elhelyezett nyilak segítségével lehet váltani. Minden nézet megőrzi a legutóbb kiválasztott elemét, ezzel megkönnyítve az átváltást például blokkok és csapdák között.

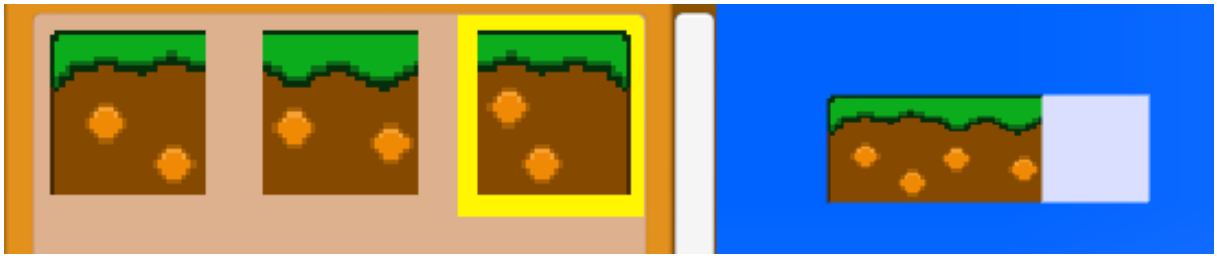
A képernyő felső részén található a rétegválasztó, amely két különböző szerkeszthető réteget kínál. A „Playfield” réteg a játéktér, ahol a karakter mozog, így célszerű a csapdákat is erre a rétegre elhelyezni. A „Background” réteg a háttér szerkesztését szolgálja, amely a játszható karakter számára nem elérhető, kizárólag vizuális mélység és díszítés céljából használható.



12. ábra: A pályaszerkesztő „Trap” nézetben

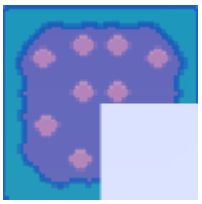
A rétegválasztó mellett találhatók az eszközök, amelyek között balról jobbra haladva a festő, a radír, valamint a beállítási funkció kap helyet, amely kizárólag „Trap” nézetben érhető el. Ebben a nézetben megjelenik a „Settings” panel is, amely alapértelmezetten üres, és csak akkor jelenít meg adatokat, ha egy csapda ki van választva. A panel lehetőséget biztosít a kiválasztott csapdák tulajdonságainak részletes módosítására. A képernyő jobb felső sarkában található mentés ikon segítségével a kész pálya elmenthető.

2.7.2. Pályaszerkesztés folyamata



13. ábra: Blokk kiválasztás és letétele

A pályaszerkesztés folyamata az alapértelmezett „Sky” témában kezdődik, ahol lehetőség van a háttér megváltoztatására a megfelelő háttérkép kiválasztásával. A szerkesztés a kívánt blokk kiválasztásával indul, ezt követően meg kell határozni a szerkesztési réteget és a használni kívánt eszközt. Festés módban a kurzor mozgatásakor láthatóvá válik a pálya rácsos elrendezése, valamint egy vizuális kiemelés jelzi, hogy a blokk elhelyezhető-e az adott pozícióban. Mivel a blokkok bárhová elhelyezhetők, ellentétben bizonyos csapdákkal, ezért nem jelenik meg külön színjelzés a lehelyezhetőségre vonatkozóan, mivel a blokkok minden esetben felülírják az adott mezőt. A blokkok moduláris felépítésűek, így megfelelően illesztve őket egy összefüggő, folyamatos terep hozható létre. A bal egérgomb lenyomásával a kiválasztott blokk azonnal elhelyezésre kerül, lenyomva tartása pedig lehetővé teszi, hogy az egér mozgatásával egyszerre több mező is lehelyezésre kerüljön, ezáltal gyorsabbá téve a szerkesztési folyamatot. A már elhelyezett blokkok bármikor átfesthetők egy másik blokk kiválasztásával.



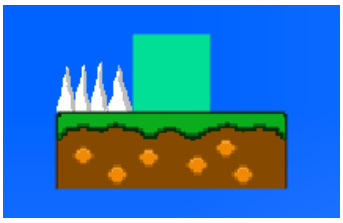
14. ábra: A háttér rétegre elhelyezett blokkok

Amennyiben a „Playfield” réteg helyett a „Background” rétegen dolgozunk, a blokkok halványabb árnyalatban jelennek meg, ezzel is jelezve, hogy azok a háttér részét képezik. A törlési funkció működése megegyezik a festésével: az egérrel kijelölt mezők tartalma törlésre kerül. Fontos, hogy a törlés csak az aktuálisan kiválasztott rétegen hajtható végre. A csapdák elhelyezése és szerkesztése technikailag ugyanúgy működik, mint a blokkoké, azonban bizonyos csapdák esetében további elhelyezési feltételek érvényesülnek, amelyeket a következő fejezet részletez.

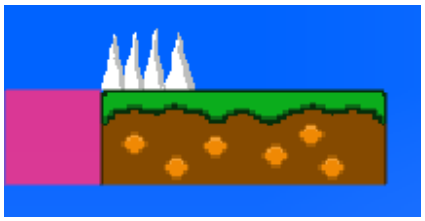
2.7.3. Csapdák bemutatása

A csapdák közül elsőként a tüske és annak különböző elforgatott változatai érhetők el. A tüske kizárólag egy blokk szomszédságába helyezhető el, a megfelelő irányba forgatva. Például egy felfelé néző tüske csak akkor helyezhető el, ha közvetlenül alatta található blokk. Az érvényes elhelyezést zöld színű kiemelés jelzi, míg a helytelen pozíciót, például blokk hiányát vagy hibás elforgatást, piros kiemelés mutatja, amely esetben a csapda nem helyezhető el. A sikeres elhelyezést követően a tüske visszahúzódik, majd megjelenik, ezzel szemléltetve a csapda működését. A többi tüske esetében a működés ugyanilyen, csupán a forgatás iránya változik: jobbra, balra vagy lefelé néző tüske esetén mindig a megfelelő blokkpozíció mellett helyezhetők el. Az érvényes és érvénytelen próbálkozások jelzése, valamint a működést bemutató animáció megegyezik a felfelé néző tüske esetével.

15. ábra: Felfelé néző tüske kiválasztása



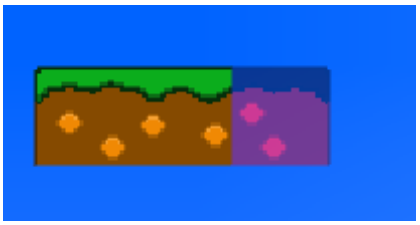
16. ábra: Tüske helyes elhelyezése



17. ábra: Tüske helytelen elhelyezése

A következő elérhető csapda a fűrész. A fűrész kiválasztásakor a „Settings” panelen megjelenik a „Make a rail” felirat, mellette pedig a sín ikonnal ellátott gomb, amely segítségével síneket helyezhetünk el a pályán. A sínek létrehozása elengedhetetlen, mivel a fűrész kizárólag ezekre helyezhető el. Sín csak üres területre rajzolható, és bármilyen irányban szabadon meghosszabbítható. A sín textúrája a rajzolt iránynak megfelelően automatikusan illeszkedik, így a pálya vizuálisan is követi a szerkesztés irányát. Miután a sínek elkészültek, a festőeszközre váltva az egér sín fölé mozgatásakor zöld színnel jelenik meg a kijelölés, ami jelzi, hogy a fűrész az adott helyen lehelyezhető.

18. ábra: Sín készítő gomb

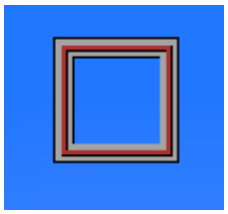


19. ábra: A sín helytelen elhelyezése

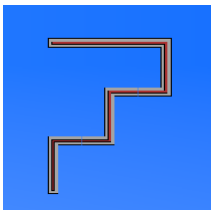
Ha a sín csak egyetlen blokkból áll, a fűrész egy helyben marad és a saját tengelye körül folyamatos forgást végez. Hosszabb pálya esetén a fűrész végighalad a létrehozott szakaszon.

Kör alakú sínpálya esetében a mozgás végig ugyanabba az irányba halad, míg két végponttal rendelkező sín esetében a fűrész a pálya végére érve megfordul, és az ellenkező irányba halad tovább. Amennyiben a sín megszakad vagy egy része törlésre kerül, a fűrész automatikusan felismeri a változást, és a megmaradt sínszakaszon folytatja mozgását.

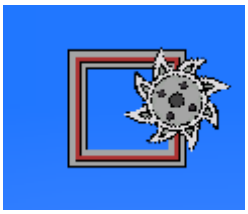
20. ábra: Egy blokknyi sín



21. ábra: Körsín



22. ábra: 2 végpontú sín

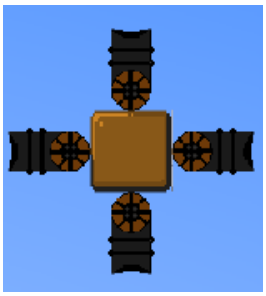


23. ábra: Fűrész a körsínen

A következő csapdatípus az ágyú. Kiválasztásakor a „Settings” panelen megjelenik a „Directions” nevű beállítás, amely négy iránygombot tartalmaz. Ezek segítségével meghatározható, hogy az ágyú melyik irányba nézzen és tüzeljen. Az ágyú elhelyezése hasonló módon történik, mint a sín esetében, tehát levegőben vagy egy blokk mellé is lerakható. Mivel az ágyú nem fed le egy teljes blokkot, külön megoldásra volt szükség annak érdekében, hogy megfelelően illeszkedjen a környezetéhez. Ennek köszönhetően, ha a blokk élével párhuzamosan helyezzük el, az ágyú automatikusan a blokk széléhez simul, így természetesebben illeszkedik a pálya szerkezetébe. Merőleges elhelyezés esetén az ágyú a blokk élének középpontjával kerül egy vonalba.

24. ábra: Ágyú irányai

Elhelyezést követően az ágyú automatikusan működésbe lép, és meghatározott időközönként lövedékeket lő ki a kiválasztott irányba. A lövedék folyamatosan halad, amíg blokkal vagy a pálya szélével nem ütközik, ekkor megsemmisül.



25. ábra: Merőleges ágyú elhelyezés

26. ábra: Párhuzamos ágyú elhelyezés

A következő elérhető csapdatípus a bárd, amely három blokk magas. Kiválasztásakor ugyanaz a „Directions” panel jelenik meg, mint az ágyú esetében (lásd 24. ábra), azonban ezen felül elérhető egy új beállítás is „Movement” névvel. Ez két ikont tartalmaz, amelyek a mozgás típusát jelzik, vagyis lengő és forgó mód közül lehet választani. A kívánt mozgástípus kiválasztása után a bárd megkötés nélkül elhelyezhető a pályán.

27. ábra: "Movement" panel

Lengő mozgás esetén a „Direction” beállítás határozza meg, hogy elforgatva melyik irányból kezdje meg a 180°-os lengést. Forgó mozgás esetén a bárd teljes körben, 360°-ban forog, így ebben az esetben a „Direction” beállításnak nincs szerepe. A lehelyezést követően a bárd automatikusan működésbe lép, és a kiválasztott mozgásformának megfelelően kezdi meg a mozgást, folyamatosan megtartva a beállított lengő vagy forgó viselkedést.

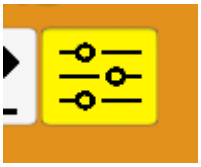
28. ábra: Lehelyezett bárd

Az utolsó elérhető csapda a guillotine, amely a pályán korlátozások nélkül elhelyezhető. Működése szerint a játékos a guillotine alá érve a csapda automatikusan leesik, a pályszerkesztőben pedig a játékos szerepét a kurzor helyettesíti, így ha a kurzort a guillotine alá visszük, az lecsapódik, lehetővé téve a tesztelést. Ha nincs alatta blokk, a guillotine a pálya széléig zuhan, míg ha akadály van, akkor felette áll meg. A guillotine mozgása időhöz kötött, ezért a lecsapódás és visszahúzódás mindig ugyanannyi ideig tart, függetlenül az esés távolságától, így különböző magasságokból indulva is egyszerre érik el a végpontjukat és térnek vissza kiindulási pozíciójukba. Ha a guillotine alatti blokkokat módosítjuk, a rendszer automatikusan újraszámítja az esés távolságát és az időt, így a mozgás mindig a pálya aktuális állapotához igazodik. A guillotinet mindig a penge részénél kell elhelyezni, ezért a szárhoz érdemes a penge felett egy blokkot hagyni.

29. ábra: A guillotine

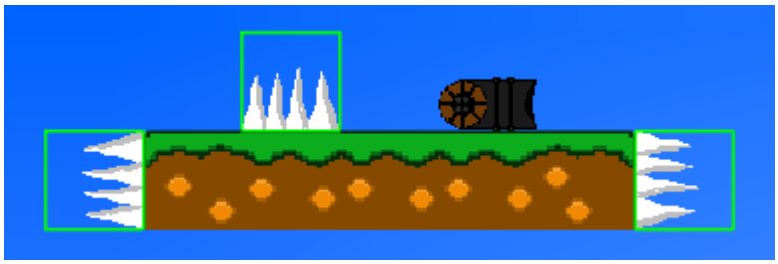
2.7.4. Csapdák beállítása

A pályaszerkesztő eszköztárában található utolsó eszköz a csapdák részletes beállítására szolgál. Ezzel az eszközzel a pályán elhelyezett csapdákra kattintva kijelölhetők, így lehetőség nyílik a tulajdonságaik finomhangolására. Több csapda kiválasztásához a SHIFT billentyűt kell lenyomva tartani, azonban egyszerre mindig csak azonos típusú csapdák jelölhetők ki. A



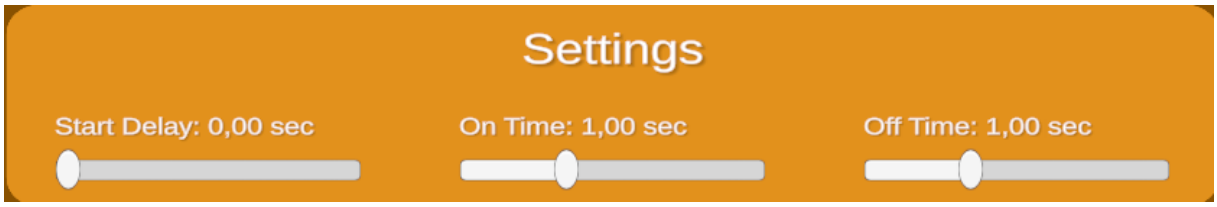
30. ábra: Beállítások eszköz

kiválasztott csapdák tulajdonságai a „Settings” panelen módosíthatók, ahol minden beállítás az adott típusra vonatkozik.



31. ábra: Közös tulajdonságú csapdák kiválasztása

A tüskék esetén a „Settings” panel három fő beállítást kínál. A „Start Delay” a tüske első aktiválódását időzíti, ami például tüskehullámok létrehozásakor hasznos, hogy az egyes elemek egymáshoz képest kis késéssel aktiválódjanak. A késleltetés másodpercben állítható a csúszka segítségével, így pontosan meghatározható, mikor induljon az adott tüske. Az „On Time” határozza meg, mennyi ideig marad a tüske aktív, míg az „Off Time” az inaktív időtartamot szabályozza, így precízen testre szabható a csapda működése a pálya igényei szerint. A tüskéket időben szinkronizálni kell, hogy valós idejű megjelenésük biztosított legyen. Minden tüske elhelyezése vagy módosítása esetén az összes tüske alaphelyzetbe kerül, amely a „Start Delay” értékével kezdődik, így a tüskék egymáshoz képest is szinkronban aktiválódnak, megkönnyítve a tesztelést és a pontos beállítást.

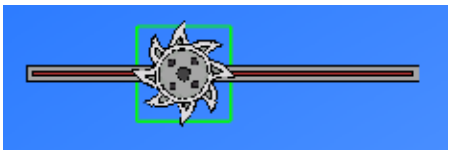


32. ábra: Tüskék beállításai

Fűrész kiválasztása esetén a 19. ábrán látható sínpálya készítő eszköz mellett megjelenik az új beállítás, ami a „Speed”. Ez a tulajdonság a fűrész mozgásának sebességét szabályozza a létrehozott sínpályán, így a játékos igényeihez vagy a pálya nehézségéhez igazítható a csapda sebessége. Fontos megjegyezni, hogy amennyiben a sínpálya csupán egyetlen blokk hosszú, a sebesség beállításának nincs gyakorlati jelentősége, mivel a fűrész ekkor nem képes tényleges mozgásra, csak saját tengelye körüli forgó animációt végez. A 33. ábra és 34. ábra szemlélteti az aktív fűrészt a sínpályán, valamint a „Speed” panel felületét, amelyen a sebesség módosítható. Új fűrész lehelyezése esetén minden fűrész visszakerül a spawnpointra, amely az a pont, ahova először tettük a fűrészt, ezzel biztosítva a szinkronizációt. Emiatt nem érdemes

több fűrészt ugyanarra a blokkra helyezni, mert ha a sebességük is megegyezik, akkor csak 1 fűrészt fogunk látni.

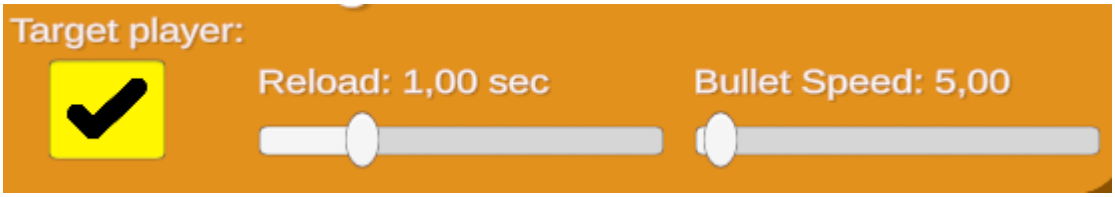
Egy ágyú kiválasztásakor lehetőség nyílik az ágyú irányának módosítására a 24. ábrán látható panel segítségével. Emellett több további beállítás is elérhető a „Directions” mellett.



33. ábra: Aktív fűrész

34. ábra: „Speed” csúszka

Az első a „Targeting Player” funkció, amely lehetővé teszi, hogy az ágyú ne statikusan egy irányba lőjön, hanem kövesse a játékost. Mivel a pályaszerkesztőben nincs tényleges játékos, az ágyú a kurzor irányába fordul, és elkezdi lőni.



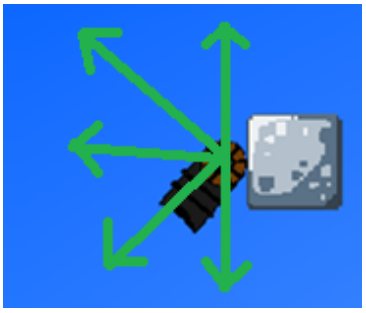
35. ábra: Ágyú további beállításai

Az ágyú nem lát át a blokkokon, így a karakter vagy kurzor a blokkok mögé bújhat. Ha az ágyú nem találja a célpontot, kereső mozgást kezd, amely során a látómező határain belül forgó mozgást végez. A látómező azonban a blokkok elhelyezkedésétől függően dinamikusan változik.

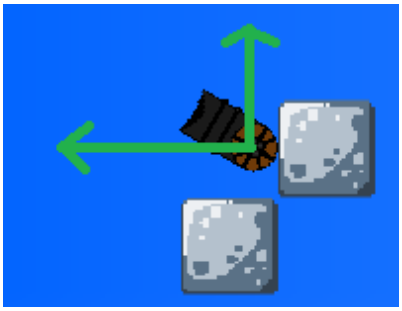
36. ábra: Az ágyú nem látja a kurzort

Konkrét példát nézve, ha az ágyú alatt blokk található (lásd 38. ábra), akkor a látómező 90°-ra szűkül, és az ágyú csak az előtte és felette lévő területet képes követni, mivel az alsó irányokat a blokk kitakarja. Alapértelmezés szerint az ágyú 180°-os szögben képes követni a játékost (lásd 37. ábra), viszont ha az ágyú alatt és felett is blokk van (lásd 39. ábra), a látómező kizárólag az előtte lévő irányra korlátozódik, és csak akkor kezd tüzelni, ha a játékos pontosan ebbe a szűk látósávba kerül.

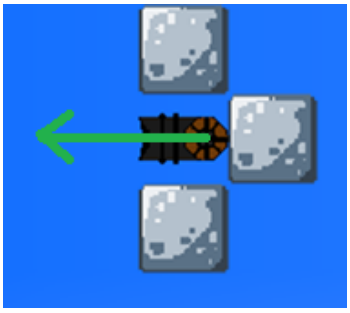
A blokkok tehát aktívan meghatározzák az ágyú látóterét és mozgási viselkedését.



37. ábra: Az ágyú 180°-ban lát



38. ábra Az ágyú 90°-ban lát



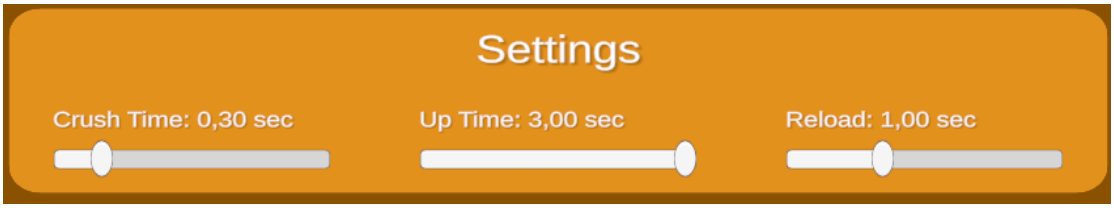
39. ábra: Az ágyú egyenesen lát

További beállítási lehetőségek a 35. ábrán látható „Reload”, amely az újratüzelésig eltelt időt határozza meg másodpercben, valamint a „Bullet Speed”, amely a kilőtt lövedékek sebességét szabályozza, lehetővé téve a csapda finomhangolását a pálya igényei szerint.

A forgó bárdot kiválasztva az ágyúhoz hasonlóan az eddigi meglévő tulajdonságait is változathatjuk és az újonnan megjelenő 34. ábrához hasonlóan a lengés vagy forgás sebességét tudjuk változtatni.

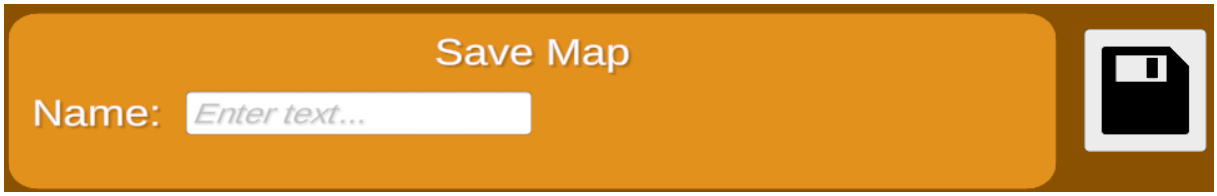
40. ábra: Kiválasztott Bárd

A guillotine aktiválásakor a „Settings” panelen három csúszka válik elérhetővé, amelyek a csapda mozgásának ütemezését szabályozzák. A „Crush Time” határozza meg azt az időt, amely alatt a penge teljesen lecsapódik, függetlenül attól, hogy mekkora távolságot kell megtennie. Ennek következtében, ha a távolság nagy, de a „Crush Time” értéke alacsony, a mozgás gyorsabbnak és erőteljesebbnek fog hatni. Az „Up Time” beállítás a visszahúzódás időtartamát szabályozza, hasonló működési elvvel, mint az előző paraméter. Végül a „Reload” értéke határozza meg, mennyi idő teljen el a csapda újraaktiválása előtt, vagyis mennyi ideig marad inaktív a guillotine a teljes lecsapódás és visszahúzódás után. Ez az időzítő a „Up Time” lefutása után indul el, így biztosítva a mozgások közötti szünetet és a működés ritmusát.



41. ábra: A guillotine beállításai

2.7.5. Mentés és kilépés



42. ábra: "Save Map" panel

A pályaszerkesztő jobb felső sarkában található mentés ikonra kattintva a „Settings” panel helyett a „Save Map” panel jelenik meg, amely tartalmaz egy „Name” input mezőt. Ide kell beírni a pálya nevét, amely a pályaválasztón és a pálya betöltésekor is meg fog jelenni. A pálya mentéséhez több kritériumnak kell megfelelni.

A pályán szükséges legalább egy „Spawn” és egy „Finish” elem elhelyezése, amelyek a „Block View” utolsó két elemének felelnek meg. A „Spawn” jelöli a játékos kezdeti megjelenési pontját a pályán, vagyis innen indul a játékos, amikor a pálya betöltődik. A „Finish” pedig a pálya célpontja, ahová a játékosnak el kell jutnia a pálya teljesítéséhez.

43. ábra: "Spawn" és "Finish" kinézete

Ezek elhelyezésére szigorú szabályok vonatkoznak. Csak blokkok fölé lehet őket helyezni, ellenkező esetben a rács piros színnel jelzi, hogy a lerakás nem lehetséges. Továbbá, a pályán csak egy „Spawn” és egy „Finish” lehet, a pályszerkesztő nem enged több elem elhelyezését. Ha módosítani kívánjuk valamelyik elemet, azt előbb törölni kell, majd újra elhelyezni. Emellett a pályának kötelező nevet adni a „Name” mezőben.

Amennyiben valamelyik kritérium nem teljesül, a „Save Map” panelen, az alábbi ábrákon látható értesítések jelenthetnek meg:

44. ábra: „Spawn” hiánya

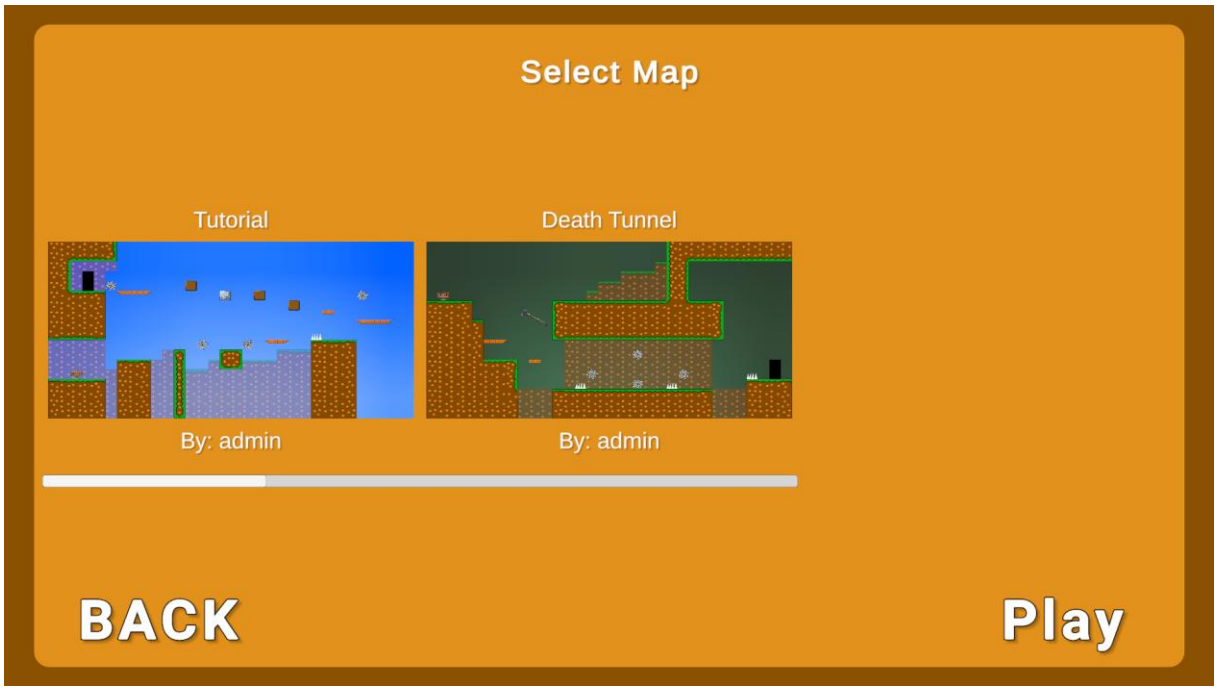
45. ábra: „Finish” hiánya

46. ábra: Pályanév hiánya

Ha a pálya minden kritériumnak megfelel, a rendszer automatikusan készít róla egy képernyőképet, amely később a pályaválasztó felületen jelenik meg. Ezt követően a játék visszairányít a kezdőképernyőre, és a jobb felső sarokban egy értesítés tájékoztat a pálya sikeres mentéséről.

47. ábra: A pálya sikeresen elkészült

2.8. Pályák kiválasztása

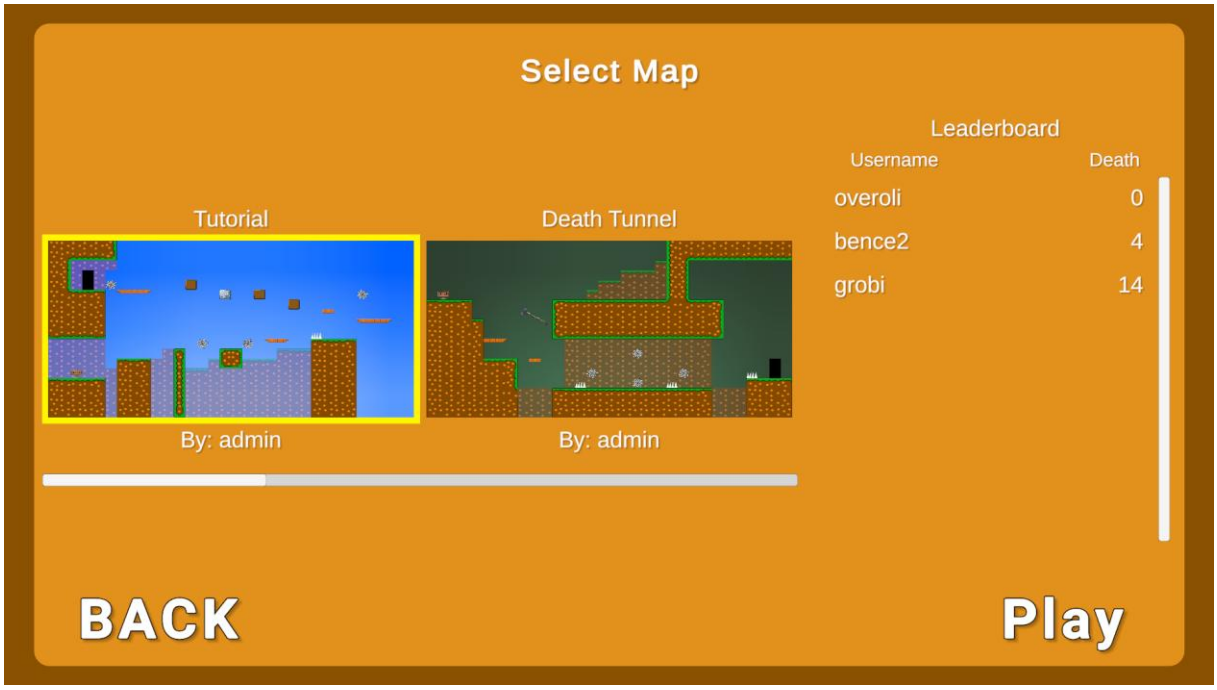


48. ábra: Pályaválasztó

A főmenüben a „PLAY” gomb megnyomásával megnyílik a pályaválasztó panel, ahol az összes eddig elkészített pálya megtekinthető egy függőlegesen görgethető listában. Minden pálya megjelenítéséhez tartozik a pálya neve, egy előnézeti kép, amely a pálya kinézetét mutatja, valamint az alkotó neve, aki a pályát készítette. A panel megnyitásakor az alkalmazás automatikusan letölti az adatbázisban tárolt pályákat.

A kívánt pálya kiválasztása a listaelemre kattintva történik, amelyet egy sárga háttér szín jelez, így egyértelműen látható, melyik pálya van aktívan kiválasztva. A képernyő jobb oldalán ezzel egyidőben megjelenik a „Leaderboard” panel, amely megmutatja, hogy a többi játékos hányszor halt meg az adott pályán, mielőtt azt sikeresen teljesítette volna. Minden pályához egyedi eredménytábla tartozik, így a statisztikák kizárólag az adott pályára vonatkoznak. A játékos neve akkor kerül be ebbe a listába, amikor sikeresen végigjátssza a pályát.

A kiválasztott pálya indítása ismét a „Play” gomb megnyomásával történik. Amennyiben nincs kijelölt pálya, a gomb megnyomása nem vált ki semmilyen műveletet, így a játék nem indul el.



49. ábra: Pálya kiválasztása

2.9. Játékmenet

50. ábra: Töltő képernyő

A pálya indításakor egy rövid töltőképernyő jelenik meg, amelynek középső részén látható a pálya neve, alatta pedig a „Created by:” felirat után a pálya készítőjének neve olvasható. A töltőképernyő rövid áttűnés után a kiválasztott pálya játszható formában jelenik meg.

A játékos karakter mindig a „Spawn” blokkon jelenik meg, innen indul a játékmenet. A mozgás az A és D billentyűk, illetve a balra és jobbra nyilak segítségével történik. Ugrani a W billentyűvel vagy a felfelé nyíllal lehet, míg az S billentyű és a lefelé nyíl lenyomásával a karakter leguggolhat, ami sok esetben hasznos egy-egy csapda elkerülésénél.

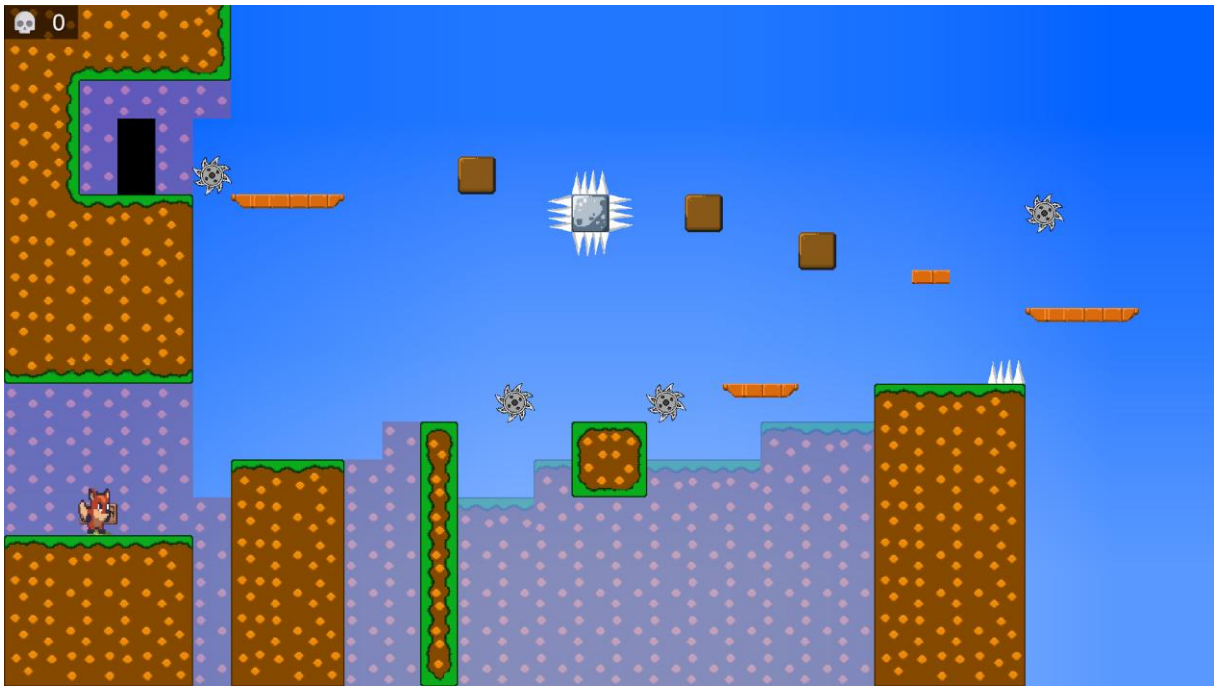
A játék többféle ugrási mechanizmust támogat. Ha az ugrás gombot nyomva tartjuk, a karakter két blokk magasra tud felugrani. Ha viszont az ugrás gombot csak röviden nyomjuk meg, a karakter egy blokk magasra ugrik. Ez a megoldás lehetővé teszi, hogy olyan pályák is készüljenek, amelyek teljesítése kizárólag a rövid ugrással lehetséges.

A pályán elhelyezett csapdák azonnal megölik a karaktert, és a képernyő bal felső sarkában található halálszámláló értéke minden halál után növekszik. A halálesetek száma automatikusan mentésre kerül az adatbázisban, így elkerülhető, hogy a játékos az alkalmazás bezárásával újra nulla halállal kezdje a pályát. A pálya újranyitásakor a halálszám onnan folytatódik, ahol az előző játék megszakadt.

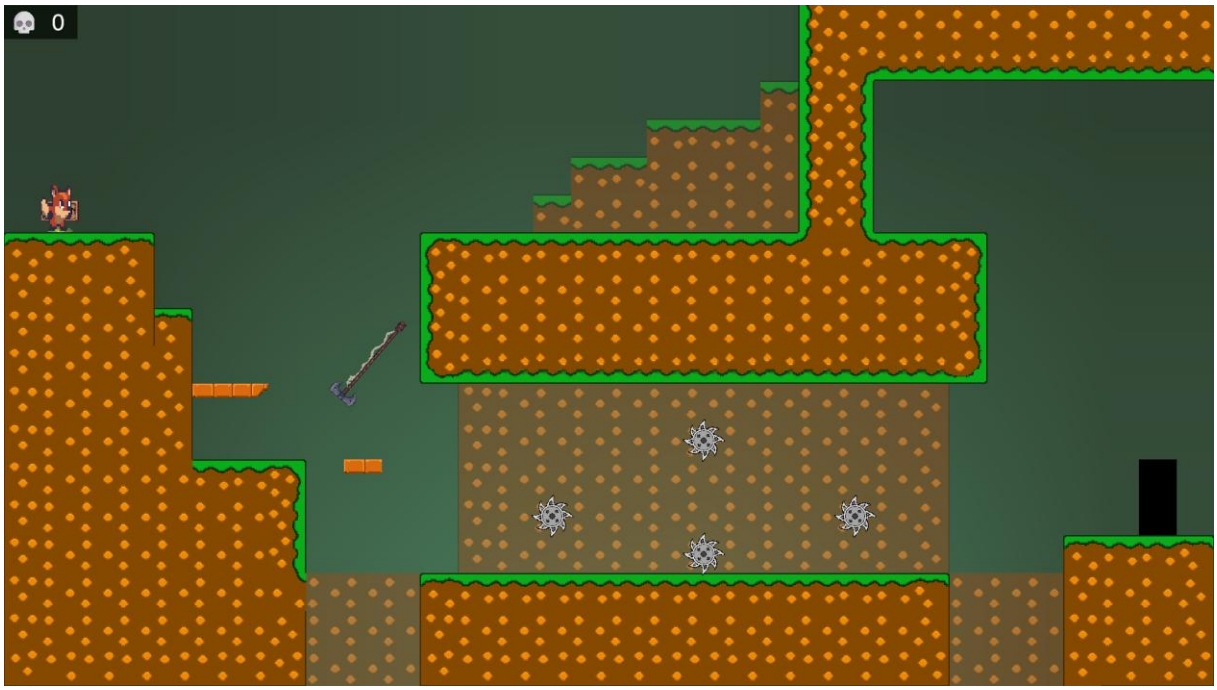
A pálya teljesítése akkor történik meg, amikor a játékos eléri a „Finish” blokkot. Ekkor a képernyő fokozatosan elhalványul, majd ismét megjelenik a töltőképernyő, amely a következő pálya betöltését jelzi. Amikor a játékos végigvitt egy pályát, az adott próbálkozás halálszáma megjelenik a „Leaderboard”-on. Ha a játékos javítani szeretne az eredményén, újra elindíthatja a pályát, ekkor a halálszám lenullázódik, és ismét nulláról kezdheti a próbálkozást.

A játék bármely pontján megnyitható a „Pause” menü az ESC billentyű lenyomásával. Ilyenkor a játék megáll, és megjelenik a menü, amely a játék folytatását, a főmenübe való visszatérést vagy az alkalmazásból való kilépést teszi lehetővé.

51. ábra: „Pause” menü



52. ábra: 1. pálya példa



53. ábra: 2. pálya példa

3. Fejlesztői dokumentáció

A fejlesztői dokumentáció bemutatja a játék tervezési folyamatát, az adatbázis, a szerver és a kliens felépítését, valamint a köztük zajló kommunikációt. Ismerteti a megvalósítást, a projekt inicializálását, a fontosabb osztályokat és algoritmusokat, valamint a fejlesztés során hozott döntéseket és tesztelési folymatokat.

3.1. Megoldási terv

A tervezés dokumentálásához a kliens oldalon a „Doxygen” generátort használtam, a különböző UML-diagramokat pedig a „PlantText” és a „Graphviz” szoftverrel készítettem. Ezek a diagramok a „Doxygen” által generált HTML és PDF dokumentációban is elérhetőek. A szerver oldalon a „FastAPI OpenAPI” automatikusan generált dokumentációját használtam.

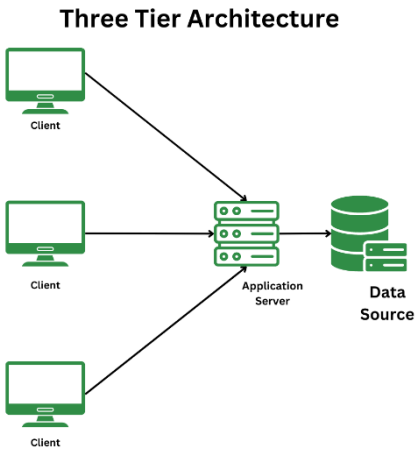
3.1.1. Architektúra

A szoftverfejlesztés során gyakori, hogy az alkalmazásokat egy meghatározott architekturális paradigma modell alapján építik fel. A modern szoftverarchitektúrák egyik legelterjedtebb típusa a háromszintű, vagy 3-tier architektúra. Ebben a modellben az alkalmazás három elkülönült rétegre tagolódik [2]:

- Presentation Tier: A felhasználói felületet kezeli, és ahol a felhasználóval való interakció történik. Ez a réteg felelős az adatok megjelenítéséért és a felhasználói bemenetek feldolgozásáért.
- Application Tier: A felhasználói kérések feldolgozását végzi, számításokat hajt végre, és döntéseket hoz. Ez a réteg közvetítő szerepet tölt be a „Presentation Tier” és a „Data Tier” között.
- Data Tier: Az adatok tárolásáért és kezeléséért felel. Itt történnek az adatbázis-műveletek és az adatok lekérése, biztosítva a rendszer számára a szükséges információkat.

A háromszintű architektúra előnyei közé tartozik a skálázhatóság, mivel minden réteg külön-külön bővíthető, például az alkalmazásréteg növelhető a megnövekedett terhelés kezelésére anélkül, hogy a többi réteget érintené. A karbantarthatóság is javul, mert egy réteg frissítése nem befolyásolja a többieket. A rugalmas fejlesztés lehetővé teszi például a felhasználói felület módosítását a további 2 réteg érintése nélkül [3].

A háromszintű architektúra a rendszeremben a kliens-szerver architektúrán belül valósult meg. A kliens-szerver modell egy hálózati architektúra, amelyben a hálózathoz kapcsolódó eszközök, a kliensek, kéréseket küldenek a központi szerver felé, amely ezeket a kéréseket feldolgozza és biztosítja az erőforrásokat vagy szolgáltatásokat [4]. Így a továbbiakban a „Presentation Tier”-re kliensként, az „Application Tier”-re pedig szerverként hivatkozok.

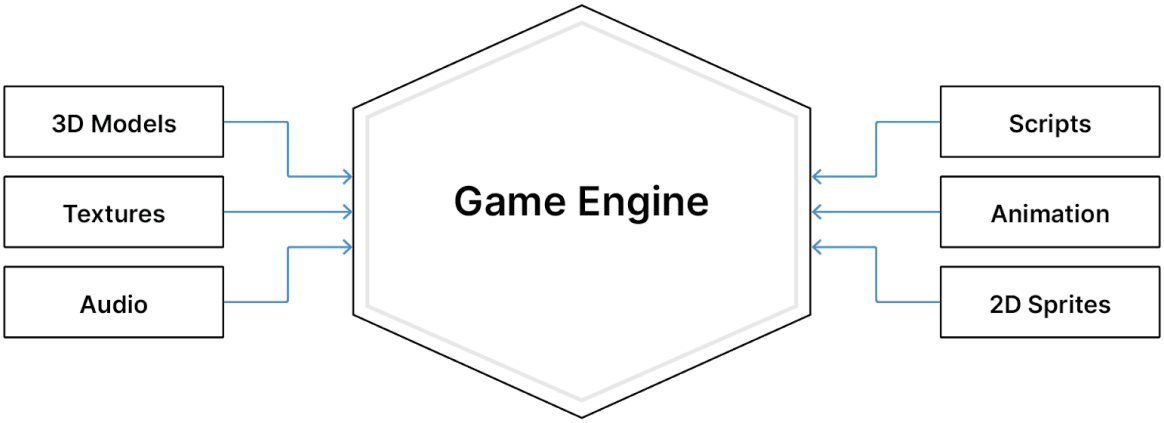


54. ábra: 3 szintű kliens-szerver architektúra [3]

3.1.2. Kliens oldal

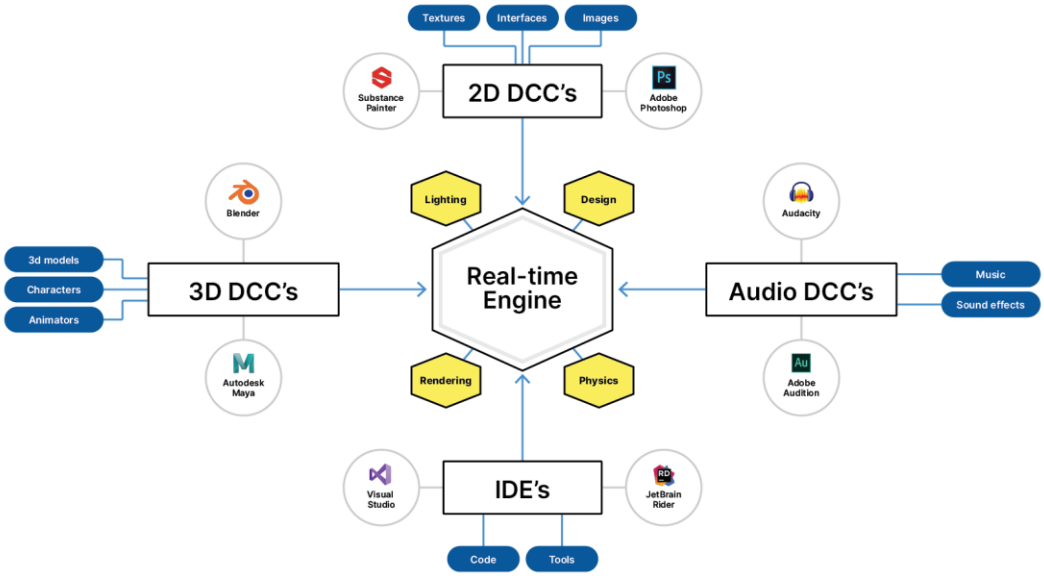
Az architektúra 1. rétege a kliens oldal. A kliensoldal megvalósításához a piacon széles körben elterjedt és modern játékmotort, az Unity Game Engine-t választottam. Az Unity egy valós idejű motor, amely lehetővé teszi az interaktív fejlesztést.

Az Unity a játék létrehozásának minden aspektusát összefogja. A játékok, mint minden alkalmazás, kisebb részekből állnak, például 3D modellekből, scriptekből és hangfájlokból. Ezek összekapcsolásával jön létre a teljes felhasználói élmény. Ha a 3D modellek, scriptek és hangfájlok az összetevők, az Unity a környezet és az eszköz, amely biztosítja ezek működését [5]. Kizárólag a C# objektumorientált programozási (OOP) nyelve támogatott, így tervezésnél ezt is figyelembe kellett venni.



55. ábra: Game Engine fő összetevői [6]

A valós idejű (real-time) működés biztosítja, hogy a felhasználói interakciók és a tartalmak azonnal, késleltetés nélkül jelenjenek meg, miközben a motor a háttérben kezeli a grafikai, fizikai és hangkezelési feladatokat. Így a fejlesztő a játék mechanikáinak és a felhasználói élmény kialakításának megvalósítására koncentrálhat [6].



56. ábra: Valós idejű munkafolyamatok [5]

3.1.3. Szerver oldal

Az architektúra második rétege a szerver oldal. A kliens és az adatbázis közötti kommunikáció API-n keresztül történik. Ehhez a FastAPI keretrendszert használtam, amely lehetővé teszi a gyors és egyszerű végpontok létrehozását, támogatja az adatbázis-kezelést, valamint validációt és hibakezelést biztosít a Python típusannotációk segítségével. Emellett automatikusan interaktív dokumentációt generál a végpontok könnyű teszteléséhez és megértéséhez. Ennek köszönhetően nincs szükség külső tesztelőeszközökre, mint például a „Postman”. A szerver három fő osztályt kezel [7]:

- Felhasználók (User): regisztráció, bejelentkezés, adatlekérés.
- Pályák (Map): pályák feltöltése, lekérése és tárolása.
- Rangsor (Leaderboard): pontszámok kezelése és frissítése.

A kliens HTTP request formájában küldi el a kérést a szerver felé, amely feldolgozza azt, majd JSON formátumban válaszol.

Sikeres feldolgozás esetén a szerver visszaküldi a megfelelő adatokat, hiba esetén pedig a HTTP-protokollnak megfelelő státuszkódot és hibaüzenetet ad vissza, így a kliens egyértelműen kezelni tudja a hibát.

FastAPI 0.1.0 OAS 3.1

/openapi.json

default

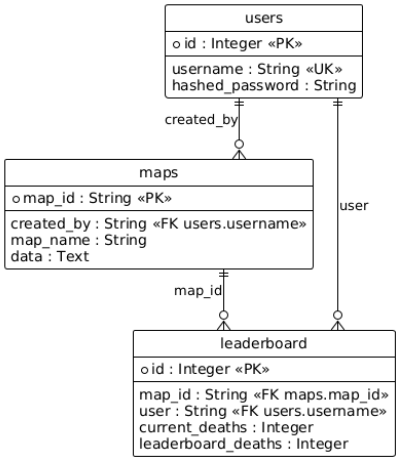
GET	/	Root
POST	/register/	Register
POST	/login/	Login
POST	/map-creator/	Create Map
GET	/maps/	Get Maps
GET	/maps/{map_id}	Get Map
GET	/download/{map_id}	Download Map Image
POST	/leaderboard/	Create Leaderboard Entry
GET	/leaderboard/{map_id}	Get Leaderboard
GET	/leaderboard/{map_id}/{user}	Get Leaderboard Entry
PATCH	/leaderboard/current/{map_id}/{user}	Update Current Deaths
PATCH	/leaderboard/leaderboard/{map_id}/{user}	Update Leaderboard Deaths

57. ábra: API végpontok

3.1.4. Adatbázis

Az architektúra második rétege az adatbázis. Erre PostgreSQL objektum-relációs adatbázis kezelő rendszert (ORDBMS) használtam, mely tökéletesen beleillik a C# objektumorientált programozásba (OOP) és a szerver JSON kiszolgálásához

Az adatbázist a „DBeaver” eszközzel kezeltem, ahol a táblák közötti kapcsolatok virtuális kulcsokkal vannak definiálva. A szerver három fő osztályt kezel, amelyeket három külön táblán valósítottam meg.



58. ábra: Az adatbázis táblái

Az alábbiakban bemutatom az egyes táblák mezőszerkezeti leírását és kapcsolatokat:

1. táblázat: User tábla mezőszerkezete

Column	Type	Key	Uniqe	Index	Leírás
id	Integer	Primary_key	True	True	Felhasználó egyei azonosítója
username	String	-	True	True	Felhasználónév
hashed_password	String	-	-	-	Jelszó hash

2. táblázat: Map tábla mezőszerkezete

Column	Type	Key	Uniqe	Index	Leírás
map_id	String	Primary_key	True	True	Pálya egyei azonosítója
created_by	String	Foreign_key (User.username)	-	True	A pályát létrehozó felhasználó
map_name	String	-	-	-	Pálya neve
data	Text	-	-	-	Pálya adatai JSON formátumban

3. táblázat: Leaderboard tábla mezőszerkezete

Column	Type	Key	Uniqe	Index	Leírás
id	Integer	Primary_key	True	True	Rangsor egyedi azonosítója
map_id	String	Foreign_key (Map.map_id)	-	-	Pályához tartozó rangsor
user	String	Foreign_key (User.username)	-	-	Felhasználóhoz tartozó pontszám
current_deaths	Integer	-	-	-	Aktuális halálozások száma
leaderboard_deaths	Integer	-	-	-	Legjobb halálozások száma

Az adatbázis nem statikus, ezért a táblák közötti kapcsolatok beállítását a következő kódrészlet szemlélteti:

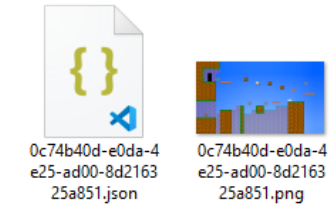
```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String, unique=True, index=True)
    hashed_password = Column(String)

...
class Map(Base):
    __tablename__ = 'maps'

    map_id = Column(String, primary_key=True, index=True, unique=True)
    created_by = Column(String, foreign_key='users.username')
    map_name = Column(String)
    data = Column(Text)

...
class Leaderboard(Base):
    __tablename__ = 'leaderboard'

    id = Column(Integer, primary_key=True, index=True, unique=True)
    map_id = Column(String, foreign_key='maps.map_id')
    user = Column(String, foreign_key='users.username')
    current_deaths = Column(Integer, nullable=True)
    leaderboard_deaths = Column(Integer, nullable=True)
```

59. ábra: Virtuális kulcsok szemléltetése

3.1.5. Fájlkezelés

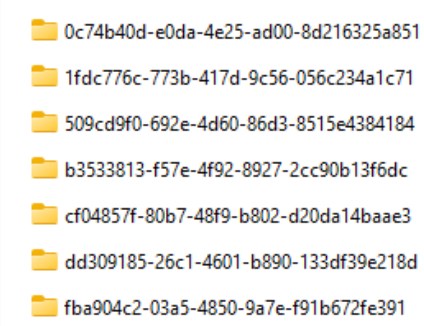
A fájlkezelés a rendszer mindhárom rétegében kulcsfontosságú szerepet tölt be. Amikor egy játékos pályát készít, a kliens oldalon a pálya adatai JSON formátumban és egy screenshot képfájlban jönnek létre. Ezeket egy mappában tároljuk, amely tartalmazza a pálya összes releváns adatát.

A kliens a JSON-t és a képet API-n keresztül a szervernek küldi, ahol a JSON adatokat az adatbázisban tároljuk, míg a képfájlokat a szerveren futó „NGINX” szolgálja ki statikus fájlként. Fontos, hogy az adatbázisban nem szükséges külön elérési URL-t tárolni a képekhez, amennyiben a kép neve megegyezik a pálya azonosítójával, hiszen minden kép egy „uploads” gyűjtőmappában található. A szerver a művelet sikerességéről visszajelzést ad a kliensnek, amely továbbítja azt a játékos felé.



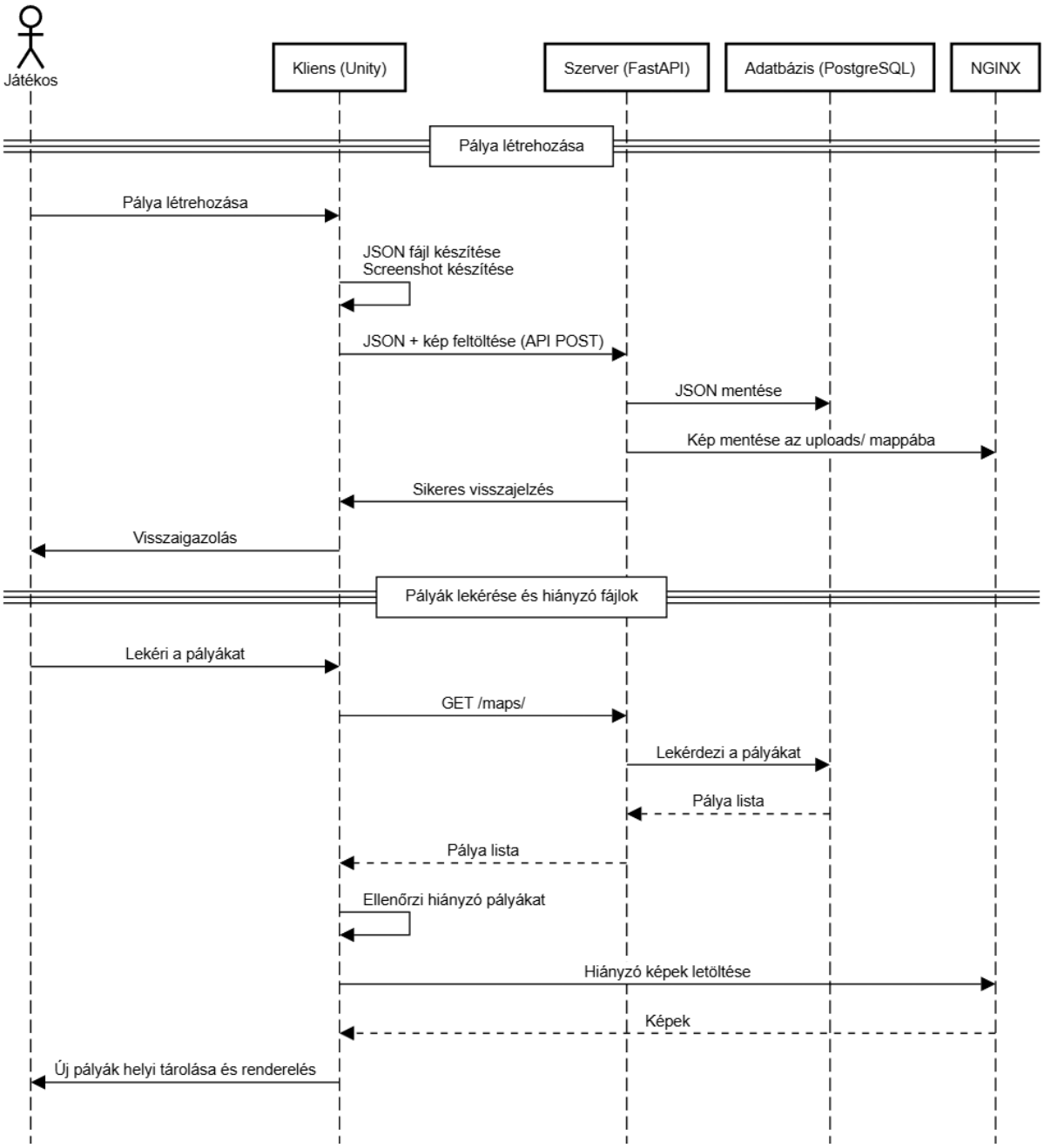
60. ábra: Egy pálya mappa tartalma

Amikor a játékos a pályaválasztó felületre lép, a kliens lekéri az összes elérhető pályát az adatbázisból, majd ellenőrzi, hogy mely pályák hiányoznak a helyi tárolóból. A hiányzó pályák képei a „NGINX” szerverről kerülnek letöltésre a kliens „Maps” mappájába, ahol tárolódnak a pályák a rendereléséhez. Ez a megoldás lehetővé teszi, hogy a kliens helyben használja az adatokat, minimalizálva a szerver terhelését.



61. ábra: "Maps" mappa tartalma

A szekvencia diagram szemlélteti a játékos, a kliens, a szerver, az adatbázis és a „NGINX” közötti interakciókat, bemutatva a pálya létrehozásának és a pályák lekérésének folyamatát, valamint a fájlok kezelését és tárolását.



62. ábra: A file kezelés szekvencia diagramja

3.1.6. Docker

A szerver architektúrája konténerizált környezetben működik, ami jelentősen leegyszerűsíti a fejlesztést, a telepítést és az üzemeltetést. A „FastAPI” alkalmazás, az „NGINX” szerver és a „PostgreSQL” adatbázis egymástól izolált, de mégis összehangolt „Docker” konténerekben futnak. A „Docker” lehetővé teszi a függőségek egységes kezelését, így a rendszer minden környezetben azonos módon viselkedik.

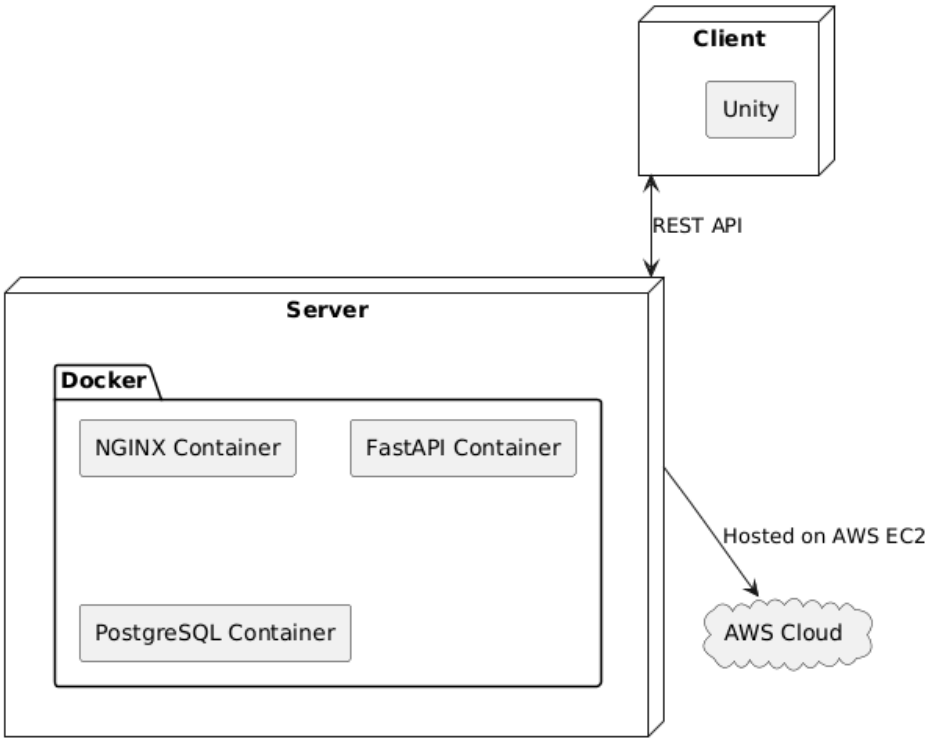
A többkonténeres architektúra „Docker Compose” segítségével kerül összehangolásra, amely deklaratív módon definiálja a szolgáltatásokat, a hálózatot és a szükséges környezeti változókat.

Ennek köszönhetően egyetlen parancs segítségével létrehozható és újraindítható a teljes szerveroldali infrastruktúra. A „dockerizáció” további előnye, hogy jelentősen csökkenti a telepítési hibák esélyét, mivel a környezet reprodukálható és önmagában hordozza a szükséges modulokat és konfigurációkat [8].

3.1.7. Felhő infrastruktúra

A szerver hostolását érdemes egy felhőalapú környezetben megvalósítani, hogy az alkalmazás bármikor, bárhonnan elérhető legyen, valamint ne függjön a fejlesztői környezet (localhost) által. A projektben az Amazon Web Services (AWS) Elastic Compute Cloud (EC2) szolgáltatását használtam, amely rugalmas, skálázható virtuális szervereket biztosít.

Az EC2 lehetővé teszi, hogy az alkalmazás tetszőleges számú virtuális gépen fusson, és a kapacitás szükség szerint növelhető vagy csökkenthető. A szerver hardveres konfigurációja az instance típustól függ, így a rendelkezésre álló memória, CPU és hálózati erőforrások a projekt igényeire szabhatók. Az EC2 előnye, hogy gyorsan beállítható és stabil teljesítményt biztosít [9].



63. ábra: „Deployment” diagram

3.1.8. Verziókezelés

A projekt verziókezeléséhez a „GIT” rendszer és a „GitHub” szolgáltatás került alkalmazásra. A kliens és a szerver kódja két különálló „GitHub” repository-ban található, ami lehetővé teszi a komponensek egymástól független fejlesztését.

Mivel a szerver egy virtuális gépen fut, így SSH-kapcsolaton keresztül egyszerűen elvégezhető a frissítések telepítése. A fejlesztési folyamat során elegendő volt a szerveren „git pull” parancsot futtatni, amely letölti az új változtatásokat a „GitHub” repository-ból.

A FastAPI fejlesztői módba konfigurálva fut, így a kódban történt módosításokat automatikusan felismeri. A reload mechanizmusnak köszönhetően a szerver újraindítás nélkül tölti be a változásokat, ami gyors és hatékony fejlesztést tesz lehetővé. Ez különösen előnyös a gyakori API-módosítások és hibajavítások során, mivel minimalizálja a megszakításokat és gyorsítja a fejlesztést.

3.1.9. Alternatív megoldások

A kliensoldalon az „Unreal Engine” jelenthet alternatívát az „Unity” helyett. Az „Unreal Engine” C++ és vizuális „Blueprint” programozást is támogat, viszont jelentősebb erőforrásigénye lassíthatja a fejlesztést. Az „Unity” C# nyelven fejleszthető, könnyebben kezelhető a 2D tartalom, és a fejlesztői tapasztalatom alapján gyorsabb fejlesztést tesz lehetővé, így a projekt „Unity” mellett maradt.

Szerveroldali alternatívaként az „Azure” felhőszolgáltatás kínál hasonló képességeket, mint az AWS EC2, skálázható virtuális gépeket és egyszerű integrációt „Docker” konténerekkel.

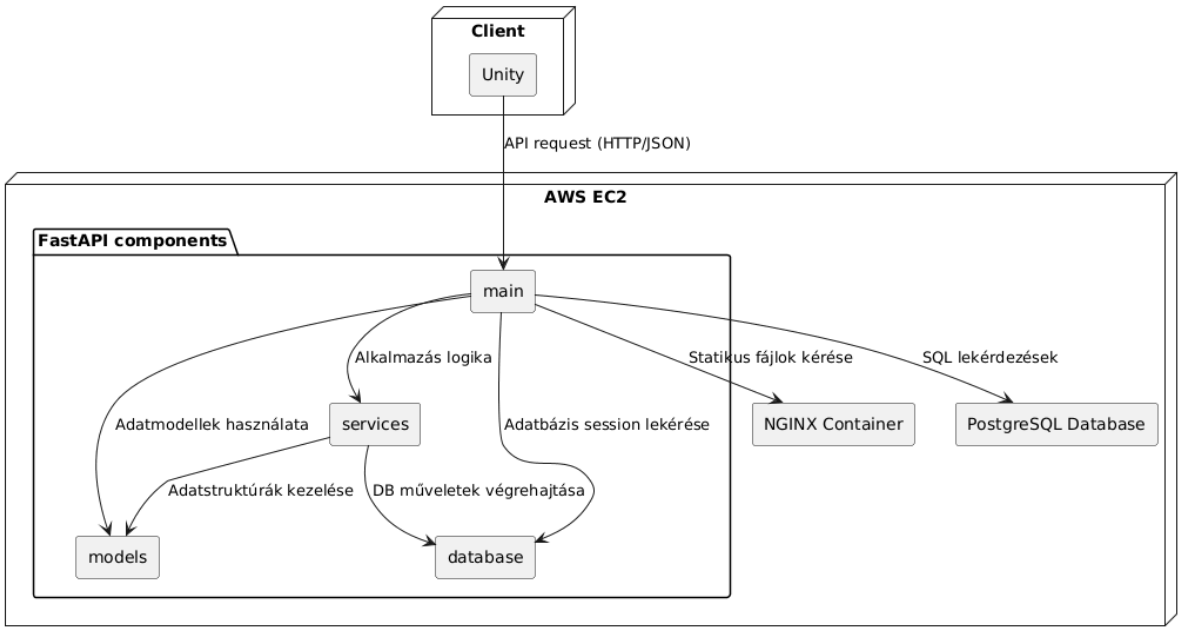
Adatbázis szinten a „MongoDB” lehet alternatíva a „PostgreSQL” helyett, mivel nem-relációs adatmodellje révén jól kezeli a JSON alapú adatszerkezeteket, ami pálya adatok tárolásánál előnyös lehet.

Verziókezelés esetében a „GitLab” kínál alternatívát a „GitHub” helyett, különösen, ha a projekthez CI/CD pipeline-ok is tartoznak, mivel egyszerűbben automatizálható a buildelési és tesztelési folyamat.

3.1.10. Komponensek

Az alkalmazás moduljai korábban áttekintésre került, most az Unity és a FastAPI komponensei kerülnek részletes bemutatásra. A szerver 4 ő komponensből áll:

- Main: A FastAPI alkalmazás belépési pontja. Itt kerülnek definiálásra az API végpontok, melyeken keresztül a kliens kommunikál a szerverrel. A komponens feladata az érkező HTTP kérések feldolgozása, a bemeneti adatok ellenőrzése és validálása. Továbbá a main komponens kezeli a statikus fájlok elérését, például a pályaképek letöltését az NGINX konténerből.
- Services: Itt történik a felhasználók, pályák és rangsorok létrehozása, jelszavak titkosítása és ellenőrzése.
- Models: Definiálja az adatbázisban tárolt entitásokat, modelleket. Itt találhatóak a felhasználó, pálya és rangsor struktúrái. A komponens biztosítja, hogy az adatok konzisztens formában kerüljenek tárolásra, valamint lehetővé teszi az adatok típusellenőrzését a bejövő kérések során.
- Database: Létrehozza az adatbázist és a szükséges táblákat. A main komponens ezen keresztül hajtja végre a commit, refresh és egyéb SQL műveleteket.

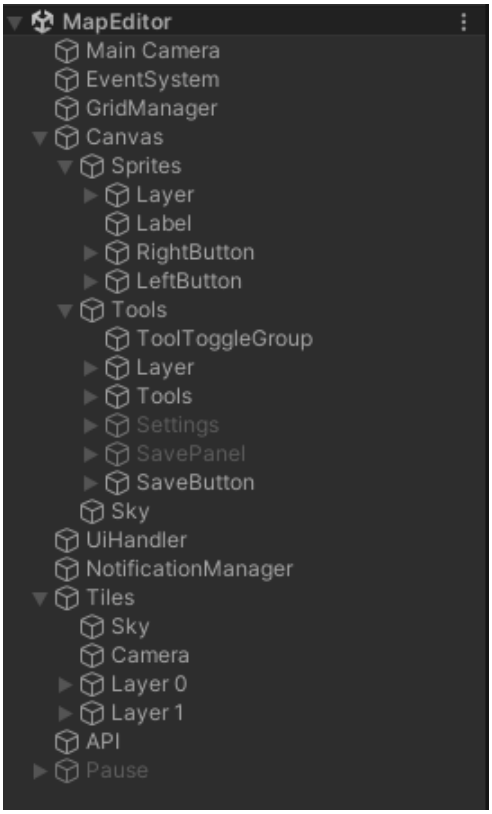


64. ábra: FastAPI komponens diagramja

A kliens oldalon három fő komponens található, amelyeket Unity-ben „Scene”-ként lehet megvalósítani a játék funkcionalitásának elkülönítésére:

- MainMenu: A játék főmenüjét biztosítja, innen indítható a játék, betölthető a pályaszerkesztő, vagy elérhetők a beállítások.

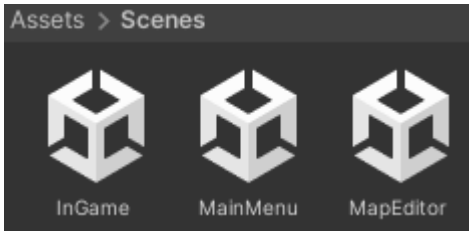
– MapEditor: A pályaszerkesztő, ahol a felhasználók saját pályákat hozhatnak létre és menthetnek.
– InGame: A tényleges játék, ahol a felhasználók végigjátsszák a pályákat, és megjelennek a pályán elhelyezett objektumok.



65. ábra: "MapEditor Scene" felépítése

A scene-ek tartalmazzák a játék objektumait, például karaktereket, környezetet, akadályokat. Minden egyedi scene fájl egyedi szintként vagy funkcionális területként értelmezhető. Ez a felépítés lehetővé teszi, hogy a játék különböző részei izoláltan fejleszthetők és kezelhetők legyenek, miközben összekapcsolódnak a teljes játékfolyamatban [10].

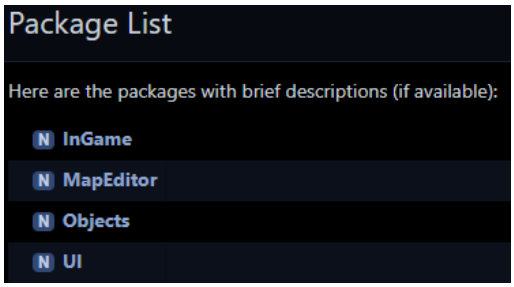
A komponensek közötti navigáció és adatátadás is a scene-ek között történik, például a felhasználói adatok és pályainformációk átadása a „MainMenu” és az „InGame” között. Ez világossá teszi a komponensek szerepét és működését a játék egészében.



66. ábra: A 3 fő komponens

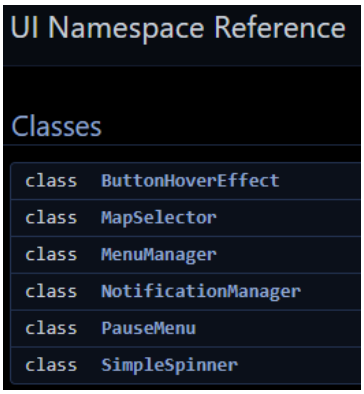
3.1.11. Osztályok

A rendszerben található osztályok funkcionális szerepük alapján külön package-ekbe rendeztem, amelyek az UI, az InGame, a MapEditor és az Objects. A moduláris felépítés lehetővé teszi, hogy a különböző játékrészek logikailag elkülönüljenek, ugyanakkor jól áttekinthető szerkezetet biztosít a teljes projekt számára.



67. ábra: Package lista

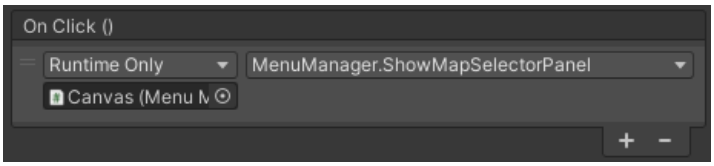
Az „UI” package tartalmazza mindazokat az osztályokat, amelyek a „MainMenu scene” vizuális és logikai működéséért felelnek. A felhasználói felület „Unity UI” elemekből épül fel, mint például gombok, panelek, input mezők és visszajelző komponensek.



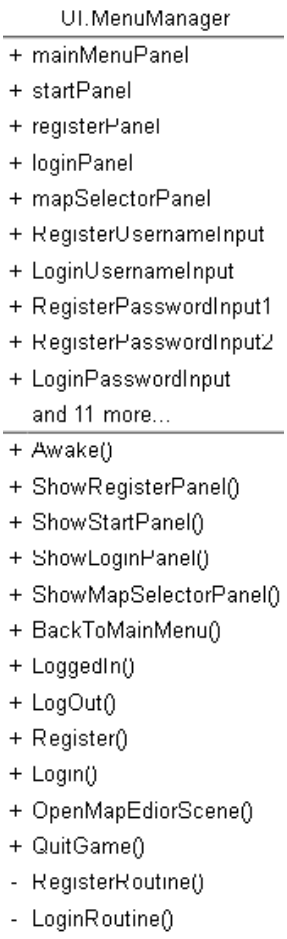
68. ábra: UI package tartalma

A felület működésének központi eleme a „MenuManager”, amely tulajdonképpen a „MainMenu scene” vezérlője. Feladata, hogy a különböző „UI” panelek megjelenítését koordinálja, figyelje a gombnyomásokat, és ezek hatására a rendszer megfelelő funkcióit meghívja. A „MenuManager” felelős a regisztrációs, bejelentkezési és pályaválasztó panelek láthatóságának kezeléséért, valamint a felhasználói állapot kezeléséért is. Mivel a bejelentkezés és regisztráció API-n keresztül történik, a „MenuManager” aszinkron kommunikációt bonyolít le a szerverrel, miközben vizuális visszajelzést biztosít a „NotificationManager”

A felület gombjai „OnClick” eseményekkel rendelkeznek, amelyekhez önálló metódusok rendelhetők.

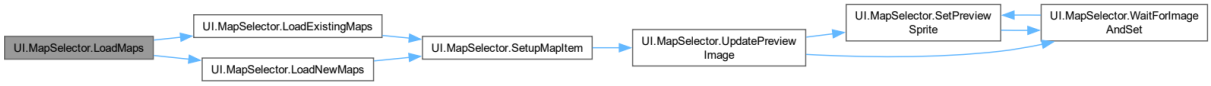


69. ábra: Unity gomb handler

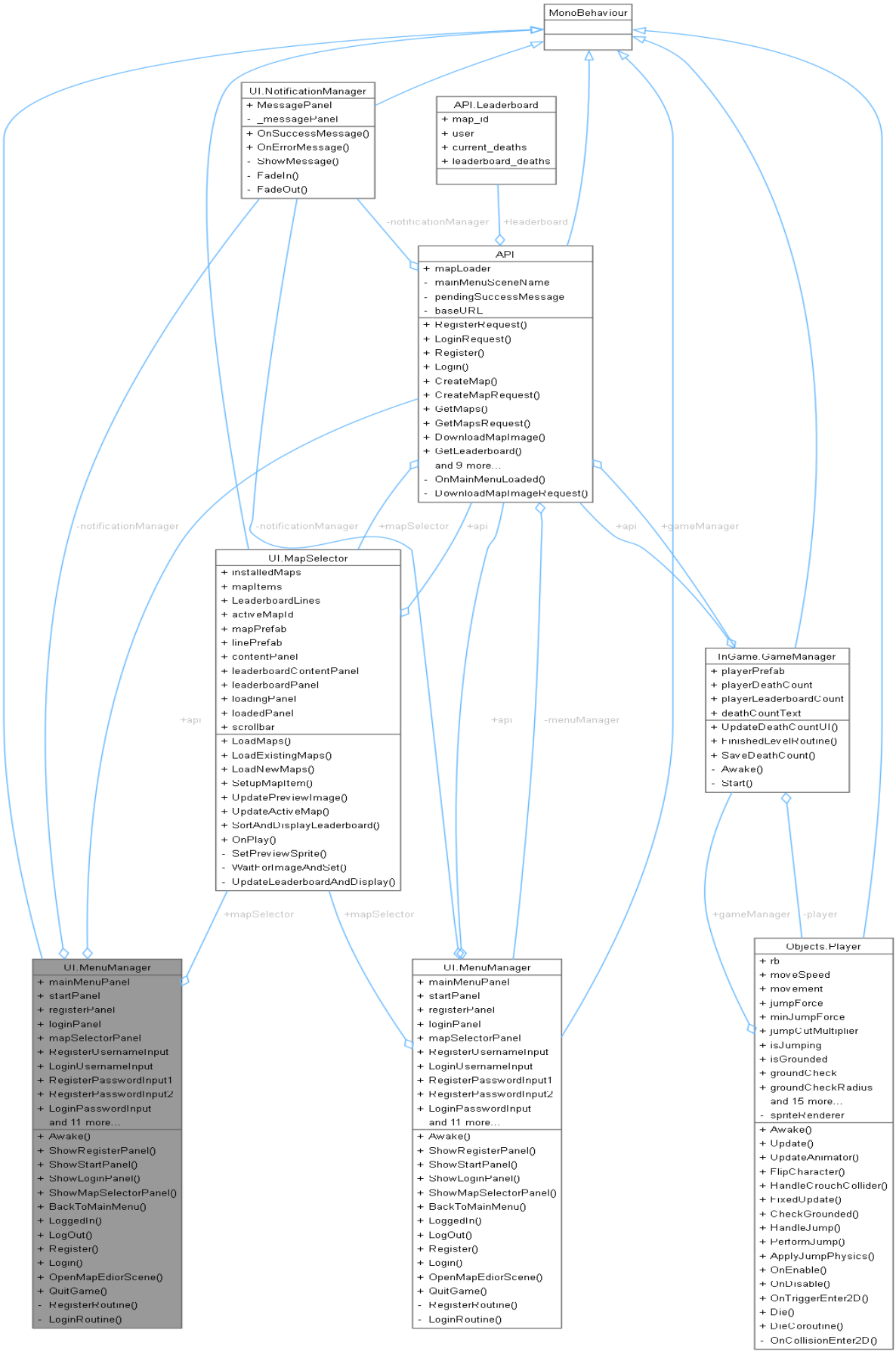


70. ábra: "MenuManager" osztálydiagramja

„LoadMaps” függvény hívási gráfja bemutatja a pálya betöltéshez kapcsolódó metódusok közötti kapcsolatokat, míg a „MenuManager” együttműködési diagramja azt szemlélteti, hogyan kommunikál a központi vezérlőosztály az „UI” további komponenseivel és a háttérben futó folyamatokkal.



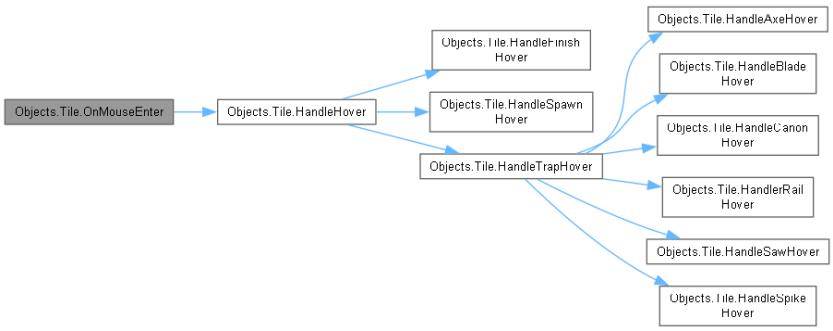
71. ábra: „LoadMaps” függvény hívási gráfja



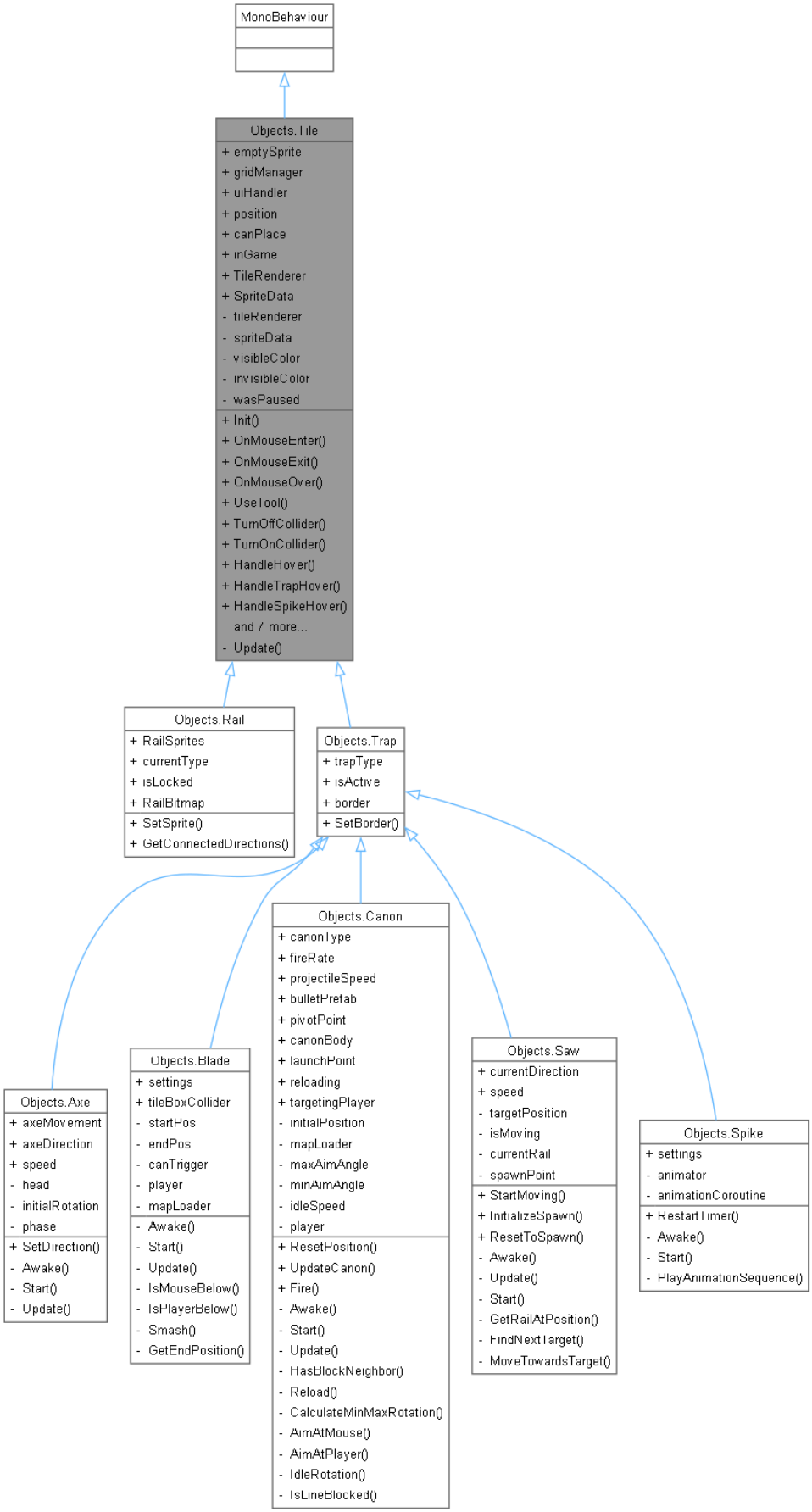
72. ábra: „MenuManager” együttműködési diagramja

Az „Objects” package tartalmazza mindazokat a játékelemeket, amelyek közvetlenül megjelennek és működnek a játéktérben. A rendszer alapját a „Tile” osztály alkotja. Ez az osztály felel minden egyes pályacella vizuális és logikai reprezentációjáért. A pályaszerkesztőben ezek az elemek alkotják a térkép teljes struktúráját, a játék futása közben pedig ugyanennek az osztálynak az objektumai jelennek meg játékelemként. A „Tile” tartalmazza a közös mezőket, például a pozíciót, a „Sprite” adatait, a „SpriteRenderert”, valamint az interakciós logikát. A játék csapdaelemei, mint az „Axe”, „Blade”, „Canon”, „Saw” vagy „Spike”, szintén a „Tile” osztályból származnak. Ezek közös működését a „Trap” absztrakt osztály fogja össze. A csapdák több tulajdonságban egységes viselkedést mutatnak. Kapcsolatba lépve a játékossal annak halálát okozzák, speciális „collider” kezelést használnak, valamint a szerkesztőben eltérő vizuális visszajelzést igényelnek. A „Trap” osztály a közös tulajdonságokat gyűjti össze, és átmeneti rétegként helyezkedik el a „Tile” és a konkrét csapdatípusok között. Így minden csapda továbbra is része marad a „Tile” alapú pályaszerkezetnek, miközben rendelkezik saját specifikus működéssel.

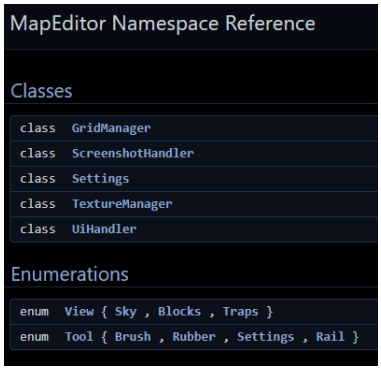
A 74. ábra szemlélteti az öröklődési hierarchiát. A „Tile” osztály a „MonoBehaviour” osztályból származik, a „Trap” pedig erre építve biztosít egységes csapdalogikát, amelyet az egyes csapdatípusok tovább specializálnak. Ez a felépítés egyszerre garantálja a könnyen bővíthető architektúrát és az egységes működési mintákat. A „Tile” osztály felelős a szerkesztőben és a játék közben megjelenő blokkok teljes működéséért. Az Init metódus végzi az objektum inicializálását, míg az „OnMouseEnter”, „OnMouseExit” és „OnMouseOver” metódusok kezelik a felhasználói interakciókat a pályaszerkesztőben. A „Tile” felismeri, hogy a felhasználó milyen eszközt választott ki, és ennek megfelelően különböző „hover” eljárásokat hajt végre. Ezek a műveletek vizuális visszajelzést adnak arról, hogy az adott elem elhelyezhető-e a kiválasztott pozícióban.



73. ábra: Tile.OnMouseEnter hívási gráfja



74. ábra: „Tile” osztálydiagramja és öröklődése



75. ábra: „MapEditor” package tartalma

A „MapEditor” package feladata a pályaszerkesztő vezérlése. A szerkesztő központi eleme a „GridManager”, amely a rácsszerkezet kirendereléséért és folyamatos frissítéséért felel. Ez az osztály biztosítja, hogy a felhasználó a megfelelő pozíciókban tudjon blokkokat elhelyezni.

76. ábra: „GridManager” osztálydiagramja

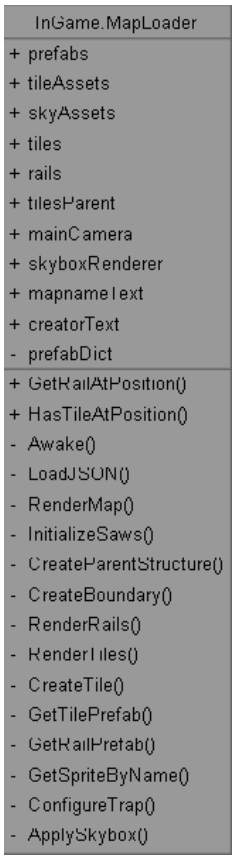
A „TextureManager” végzi a szerkesztéshez szükséges blokkok kezelését. Ez az osztály felelős azért, hogy a felhasználó a grafikus felületen kiválaszthassa a kívánt blokk típust, majd ezt a választást a rendszer belső állapotban rögzítse. A „TextureManager” tárolja az aktuálisan kiválasztott textúrát, így a blokkok elhelyezése következetes és egyszerű folyamat.

A felhasználói interakciók nagy része a „UIHandleren” keresztül történik, amely a különböző panelek működését és az ezekhez tartozó eseményeket kezeli. Ez az osztály felel azért, hogy a felhasználó könnyedén válthasson eszközök, menük vagy szerkesztési módok között, és a teljes felület működése gördülékeny maradjon.

A pályaszerkesztőben található speciális elemek, például csapdák vagy interaktív objektumok konfigurálását a „Setting” osztály biztosítja. Ez az osztály tartalmazza azokat a beállításokat és logikákat, amelyek lehetővé teszik bizonyos objektumok tulajdonságainak módosítását.

A szerkesztés lezárásakor a „ScreenshotHandler” gondoskodik arról, hogy a rendszer a kész pályáról egy képernyőképet készítsen. Ez a kép később a pályaválasztóban jelenik meg, így vizuálisan is beazonosíthatóvá válik a felhasználó által létrehozott térkép. A „ScreenshotHandler” biztosítja, hogy a felvétel megfelelő felbontással és pozicionálással készüljön el.

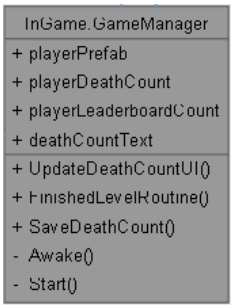
Az „InGame” package felel a pálya betöltéséért, megjelenítéséért és a játék közbeni logikák előkészítéséért. A „MapLoader” osztály, amely a JSON formátumban tárolt pályaadatok betöltését, a „tile” alapú pálya kirenderelését és a csapdák inicializálását végzi. A működés az „Awake” metódusban indul el, ahol a rendszer előkészíti a pálya betöltéshez szükséges erőforrásokat és felépíti a belső adatstruktúrákat. A betöltés maga a „LoadJSON” metódusban történik.



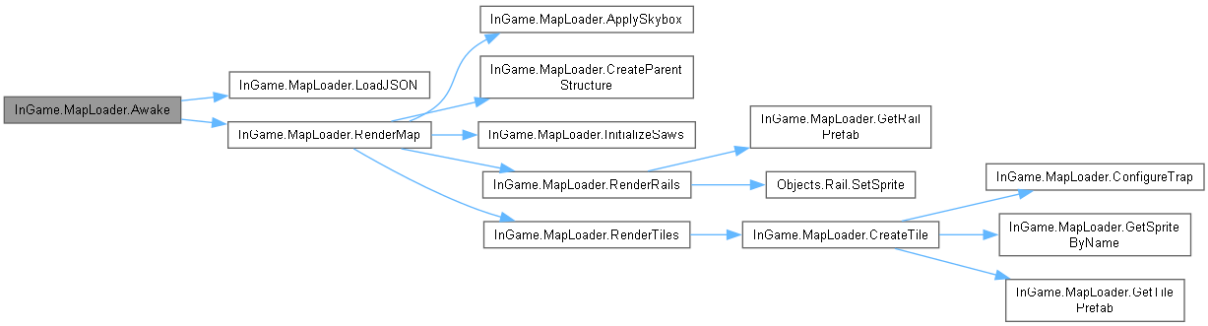
77. ábra: „MapLoader” osztálydiagramja

A teljes pálya felépítése a „RenderMap” feladatkörébe tartozik. Ennek része a „tile” objektumok renderlése a „RenderTiles” metódusban. A „MapLoader” először meghatározza, hogy egy adott blokk milyen típusú prefabnak felel meg, amelyet a „GetTilePrefab” metódus biztosít, majd a „CreateTile” létrehozza és a scene megfelelő pontjára pozicionálja az adott elemet. A „tile” objektumok egy listában kerülnek tárolásra, így a rendszer hatékonyan tudja lekérdezni, hogy egy adott pozíción helyezkedik-e el blokk, amit a „HasTileAtPosition” metódus biztosít.

A játékmenet vezérlését a GameManager látja el. Ez az osztály felel a játék logikai állapotainak kezeléséért. Figyeli a játékos aktuális állapotát, kezeli a haláleseményeket, valamint biztosítja a megfelelő visszajelzéseket a felhasználó számára. A játékon belüli események, például a játék vége, az időzítők állapota vagy a mozgó elemek aktiválódása.

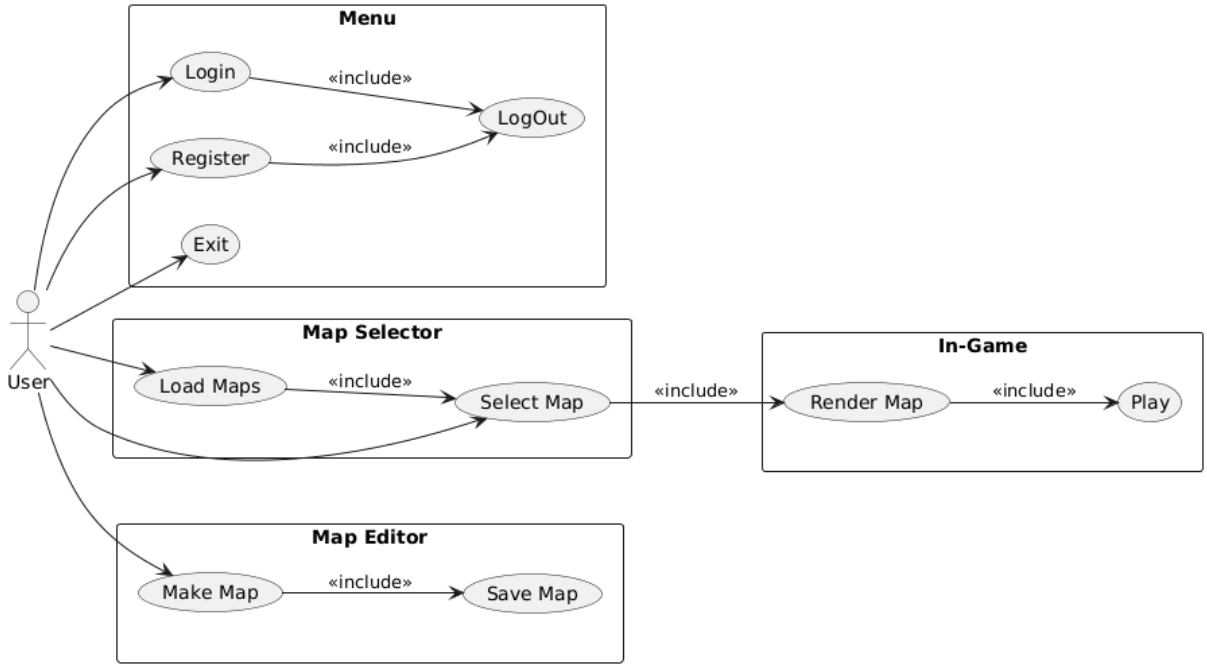


78. ábra: „GameManager” osztálydiagramja



79. ábra: MapLoader.Awake hívási gráfja

3.1.12. Felhasználói esetek

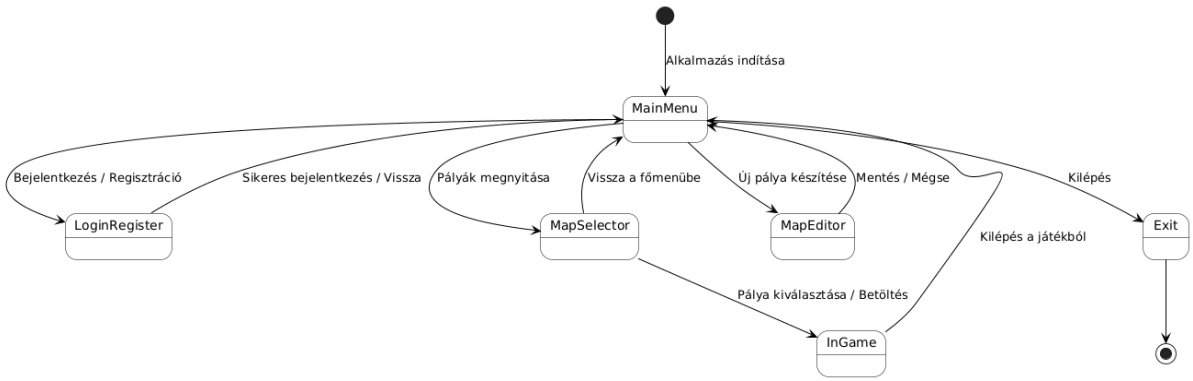


80. ábra: A játék használati eset diagramja

A felhasználói eset diagram az alkalmazás fő funkcióit mutatja be egyetlen felhasználó szempontjából. A felhasználó a főmenüből bejelentkezhet, regisztrálhat, kijelentkezhet vagy kiléphet az alkalmazásból. A pályák betöltése és kiválasztása a „MapSelector” modulban történik, míg új pályák létrehozása és mentése a „MapEditorban” valósul meg. A kiválasztott pálya megjelenítését és a játék indítását az „InGame” modul kezeli, biztosítva, hogy a felhasználó mindig a megfelelő sorrendben férjen hozzá a funkciókhoz.

3.1.13. Felhasználói felület

A rendszer felhasználói felülete több, funkcionálisan elkülönített képernyőből áll, amelyek között a felhasználó egyértelmű navigációs útvonalakon keresztül léphet tovább. A főmenüből érhető el a bejelentkezés és regisztráció felülete, a pályaválasztó képernyő, valamint a pályaszerkesztő modul. A „MapSelector” a mentett pályák listázását és kiválasztását biztosítja, míg a „MapEditor” a pályák elkészítését és mentését teszi lehetővé. A kiválasztott pálya betöltése után a rendszer az „InGame” képernyőre vált, ahol a játék indítása és megszakítása történik. A képernyők közötti lehetséges átmeneteket egy állapotdiagram foglalja össze, amely a tényleges navigációs irányokat és a hozzájuk tartozó eseményeket ábrázolja. Ez a struktúra egyértelműen mutatja, mely képernyők mely műveletek után válnak elérhetővé, és hogyan épül fel a felhasználó teljes útvonala az alkalmazásban.



81. ábra: A felhasználói felület állapotdiagramja

3.2. Megvalósítás

A következő fejezet összefoglalja, milyen eszközök és technológiák kerültek felhasználásra, hogyan épült fel a rendszer a gyakorlatban. A fejezet bemutatja a menü, a pályakezelés és a játék működéséhez szükséges fontosabb objektumokat és algoritmusokat.

3.2.1. Forráskód beszerzése

A forráskódok beszerzéséhez szükségünk lesz a „GIT” telepítésére ezek után a forráskódokat az alábbi parancsokkal letölthetők:

- Client: git clone https://github.com/grobi447/Thesis-Client.git
- Server: git clone https://github.com/grobi447/Thesis-Backend.git

3.2.2. Fejlesztési eszközök

A kliensoldali fejlesztéshez az Unity Hub telepítése szükséges, amely biztosítja a különböző Unity Editor-verziók kezelését. A projekt az Unity 2022.3.63f2 LTS kiadásban készült, ezért a fejlesztői környezet felépítéséhez ennek a verziónak a telepítése szükséges.

A szerveroldali futtatáshoz szükség van a Docker és a Docker Compose telepítésére. A telepítési folyamat operációs rendszerenként eltér:

- Windows: a Docker Desktop telepítése szükséges, amely elérhető a következő címen:

  https://www.docker.com/get-started/

- Linux (Ubuntu): a Docker Engine és a Docker Compose telepítési útmutatója elérhető a dokumentációban:

  https://docs.docker.com/engine/install/ubuntu/

A kódszerkesztéshez Visual Studio Code-t használtam, Unity és Python bővítménnyel.

3.2.3. Függőségek

A szerver függőségei a „requirements.txt” fájlban kerültek definiálásra. A Docker futtatásakor ezek automatikusan telepítésre kerülnek, így a fejlesztőnek külön nem kell foglalkoznia velük.

A szerver gyökérkönyvtárában azonban szükség van egy „.env” fájl létrehozására, amely az alábbi konfigurációs paramétereket kell, hogy tartalmazza:

Server > .env

```
1  DB_URL= postgresql+asyncpg://<felhasználónév>:<jelszó>@<szerver-cím>:<port>/<adatbázis-név>
2  USERNAME = az adatbázis felhasználóneve
3  PASSWORD = az adatbázis jelszava
4  NAME = az adatbázis neve
```

82. ábra: .env fájl tartalma

Ezek a paraméterek teszik lehetővé, hogy a szerver hozzáférjen az adatbázishoz, végrehajtsa a lekérdezéseket, olvassa és írja az adatokat a felhasználók, pályák és rangsorok kezeléséhez. Lokális fejlesztés során a localhost, éles környezetben pedig a szerver IP-címe vagy domain neve adja meg a hostot.

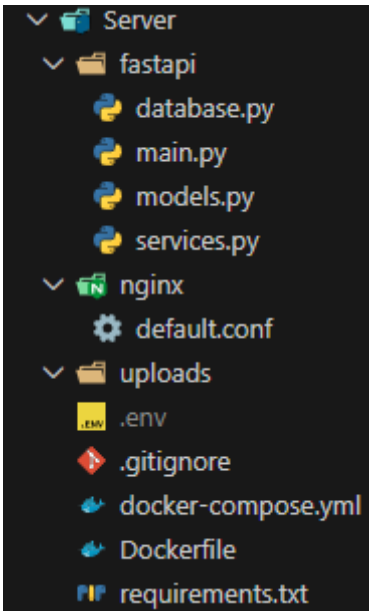
A kliensoldali alkalmazás függőségei a projektben használt fejlesztői környezet az Unity Package Manager automatikusan kezeli a szükséges külső könyvtárak telepítését. Ez lehetővé teszi, hogy a szükséges külső könyvtárak, mint JSON kezelő könyvtárak vagy UI komponensek automatikusan rendelkezésre álljanak a fejlesztés során, így a fejlesztőnek nincs külön telepítési feladata.

3.2.4. Szerver felépítése

A szerver felépítése a következő fő mappákból és fájlokból áll:

- fastapi mappa:
  - main.py: A FastAPI alkalmazás belépési pontja, itt történik az API végpontok regisztrációja és az alkalmazás indítása.
  - database.py: Az adatbázis kapcsolat definiálása.
  - models.py: Az adatbázis modellekleírása, amelyek az adatbázis táblákat reprezentálják.
  - services.py: API végpontokat kiszolgáló függvények implementációja.

– nginx mappa:
    - default.conf: Az Nginx webszerver konfigurációs fájlja, amely a forgalom továbbítását és a statikus fájlok kiszolgálását szabályozza.
– uploads mappa: Feltöltött fájlok tárolására szolgáló könyvtár.
– Egyéb fájlok:
    - docker-compose.yml: A szerver és a kapcsolódó szolgáltatások konténerizált indítását és összekapcsolását definiáló fájl.
    - Dockerfile: A szerver alkalmazás Docker image-ének felépítését leíró fájl.
    - .gitignore: Verziókezelésből kizárandó fájlok listája.
    - requirements.txt: A Python környezethez szükséges csomagok listája.



83. ábra: Szerver mappa felépítése

3.2.5. Szerver konfigurációk

A szerver konfiguráció két kulcsfontosságú elemből áll, a „Docker Compose” és az „Nginx” beállításaiból. A „docker-compose.yml” három fő szolgáltatást definiál. Az adatbázis (db) „PostgreSQL”-t használ a „.env” fájlban definiált hitelesítési adatokkal, a „pgdata volume” biztosítja az adatok tartósságát. A „FastAPI” alkalmazás (web) az „uvicorn” szerveren fut, a forráskód és a feltöltött fájlok „mountolásával”, az adatbázis elérési útját a „DB_URL” változó adja át. Az „Nginx” a bejövő „HTTP” és „HTTPS” forgalmat kezeli, „mountolt” konfigurációs fájlokkal, „SSL” tanúsítványokkal, valamint az „uploads” mappa kiszolgálásával.

```yaml
version: '3'

services:
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: ${USERNAME}
      POSTGRES_PASSWORD: ${PASSWORD}
      POSTGRES_DB: ${NAME}
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

  web:
    build: .
    container_name: fastapi-app
    command: uvicorn main:app --host 0.0.0.0 --port 8000 --reload
    volumes:
      - ./fastapi:/app
      - ./uploads:/app/uploads
    depends_on:
      - db
    ports:
      - "8000:8000"
    environment:
      - DB_URL=${DB_URL}

  nginx:
    image: nginx:alpine
    container_name: nginx-proxy
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/default.conf:/etc/nginx/conf.d/default.conf:ro
      - ./ssl:/etc/nginx/ssl:ro
      - ./uploads:/app/uploads:ro
    depends_on:
      - web

volumes:
  pgdata:
```
84. ábra: „docker-compose.yml” file tratalma

Az „Nginx” konfiguráció két szerverblokkot tartalmaz. A 80-as porton érkező kérések „HTTPS”-re irányítódnak, a 443-as port „SSL”-tanúsítványokkal és megfelelő „TLS” protokollokkal kezeli a forgalmat, miközben a / (root) útvonalon továbbítja a kéréseket a „FastAPI” backend felé. Az /uploads/ útvonal alias segítségével szolgálja ki a feltöltött fájlokat hosszú cache idővel. A konfiguráció biztosítja a biztonságos kommunikációt, a backend elérhetőségét és a statikus fájlok optimális kiszolgálását.

```
server {
    listen 80;
    server_name 52.58.160.54;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name 52.58.160.54;

    ssl_certificate     /etc/nginx/ssl/selfsigned.crt;
    ssl_certificate_key /etc/nginx/ssl/selfsigned.key;

    ssl_protocols       TLSv1.2 TLSv1.3;
    ssl_ciphers         HIGH:!aNULL:!MD5;

    location / {
        proxy_pass         http://web:8000;
        proxy_http_version 1.1;

        proxy_set_header   Host              $host;
        proxy_set_header   X-Real-IP         $remote_addr;
        proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }

    location /uploads/ {
        alias /app/uploads/;
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

85. ábra: "default.conf" file tartalma

3.2.6. API végpontok

Ebben a fejezetben három különböző „HTTP” metódust („GET”, „POST”, „PATCH”) alkalmazó API végpont kerül bemutatásra, amelyek a rendszer fontosabb funkcióit fedik le, mint a pályakép letöltése, felhasználó regisztrációja, illetve játékos halálainak frissítése.

A /download/{map_id} végpont segítségével a kliens letöltheti a kiválasztott pálya képét. A szerver ellenőrzi, hogy a fájl létezik-e az „uploads” mappában, majd „FileResponse” segítségével visszaküldi azt. Hibás kérés esetén a szerver 404-es, míg váratlan hiba esetén 500-as státusz kódot ad.

```python
@app.get("/download/{map_id}")
async def download_map_image(map_id: str):
    try:
        filename = f"{map_id}.png"
        file_path = f"uploads/{filename}"

        if not os.path.exists(file_path):
            raise HTTPException(status_code=404, detail=f"Image not found for map_id: {map_id}")

        return FileResponse(
            path=file_path,
            media_type="image/png",
            filename=filename
        )

    except HTTPException:
        raise
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error downloading map image: {str(e)}")
```

86. ábra: Pályakép lekérésének API végpontja (GET)

A /register/ végpont új felhasználó létrehozására szolgál. A szerver ellenőrzi a felhasználónév és a jelszó hosszát, valamint a felhasználónév egyediségét. A jelszó titkosítása és az adatbázis-műveletek a services.py fájl megfelelő metódusain keresztül történnek, például a

„hash_password” és a „create_user” függvények meghívásával. A regisztráció végén a szerver visszajelzést küld a sikeres létrehozásról, valamint tartalmazza a felhasználói adatokat.

```python
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str):
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str):
    return pwd_context.verify(plain_password, hashed_password)

async def create_user(db: AsyncSession, username: str, hashed_password: str):
    db_user = User(username=username, hashed_password=hashed_password)
    db.add(db_user)
    await db.commit()
    await db.refresh(db_user)
    return db_user
```

87. ábra: Felhasználó regisztrációs API végpontja (POST)

A jelszókezelés és a felhasználók létrehozása a „services.py” fájlban történik. Itt kerül sor a jelszavak „bcrypt” alapú titkosítására, valamint az új felhasználók adatbázisba mentésére.

```python
def hash_password(password: str):
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str):
    return pwd_context.verify(plain_password, hashed_password)

async def create_user(db: AsyncSession, username: str, hashed_password: str):
    db_user = User(username=username, hashed_password=hashed_password)
    db.add(db_user)
    await db.commit()
    await db.refresh(db_user)
    return db_user
```

88. ábra: „service.py” metódusai

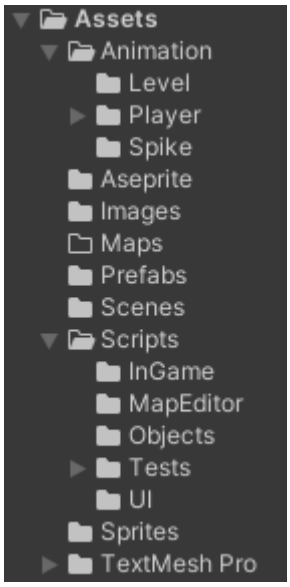
A /leaderboard/current/{map_id}/{user} végpont egy adott pálya adott felhasználó halálát módosítja. A szerver ellenőrzi, hogy a megfelelő rekord létezik-e, majd a „current_deaths” mezőt frissíti. Sikeres frissítés esetén a szerver visszaadja a módosított rekordot, hiba esetén a megfelelő státusz kódot küldi.

```python
@app.patch("/leaderboard/current/{map_id}/{user}")
async def update_current_deaths(map_id: str, user: str, current_deaths: Optional[int] = None, db: AsyncSession = Depends(get_db)):
    try:
        result = await db.execute(select(Leaderboard).where(Leaderboard.map_id == map_id, Leaderboard.user == user))
        entry = result.scalar_one_or_none()

        if not entry:
            raise HTTPException(status_code=404, detail="Leaderboard entry not found")

        entry.current_deaths = current_deaths
        await db.commit()
        await db.refresh(entry)

        return {
            "detail": "Leaderboard entry updated successfully",
            "leaderboard": {
                "map_id": entry.map_id,
                "user": entry.user,
                "current_deaths": entry.current_deaths,
                "leaderboard_deaths": entry.leaderboard_deaths
            }
        }
```

89. ábra: Leaderboard adatainak frissítése (PATCH)

3.2.7. Unity projekt felépítése

Az Unity projekt fő munkaterülete az „Assets” mappa, ahol a fejlesztéshez szükséges összes file és forráskód megtalálható. A mappastruktúra a fájlok kiterjesztése és funkciója szerint van csoportosítva:

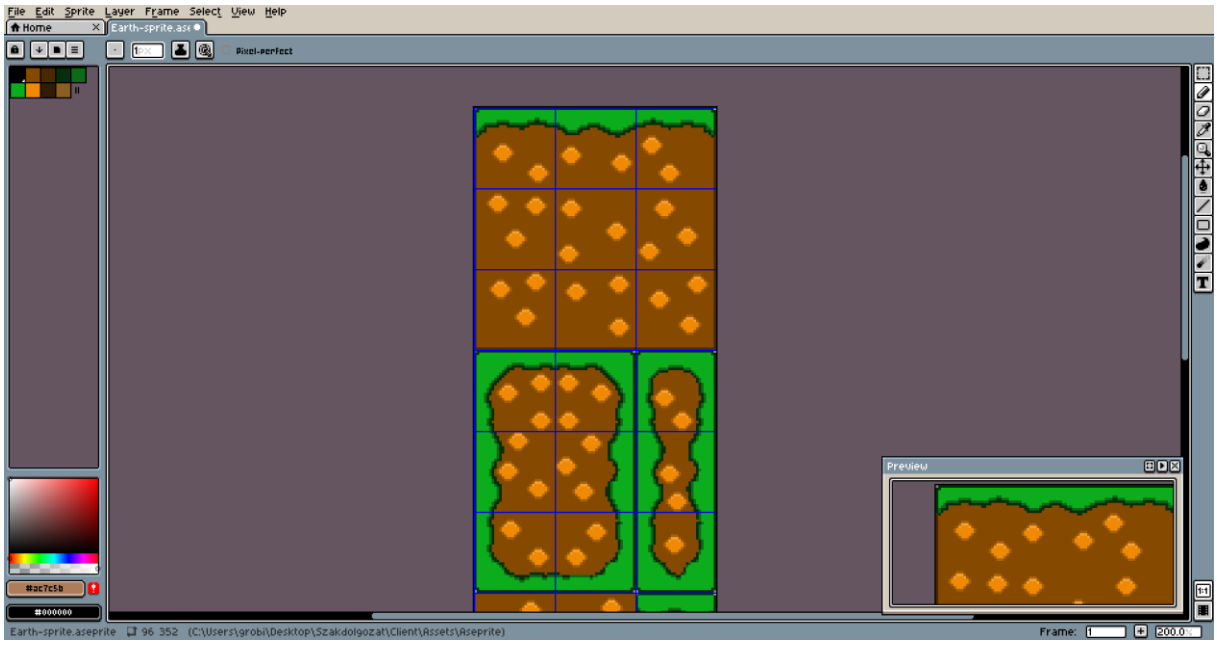


90. ábra: Assets mappa tartalma

- Animation: Az animációk min a pályák közti áttűnés, karaktermozgás, csapdák animációi itt találhatók, Unity animációs „asset”-ként.
- Aseprite: A „sprite”-ok szerkeszthető, nyers „.aseprite” formátumú állományai, amelyekből végleges „spritesheetek” készülnek
- Images: Háttérképek, ikonok, UI elemek PNG formátumban.
- Maps: Ide töltődnek le a pályák, amelyeket a játék vagy a pályaszerkesztő használ.
- Prefabs: Újrahasznosítható, dinamikusan példányosítható objektumok, mint a csapdák és a karakter
- Scenes: A három fő Unity „scene” a főmenü, játék és pályaszerkesztő itt található.
- Scripts: A C# forráskódok funkció szerint további almappákba rendezve.
- Sprites: Az összes „spritesheet”, amely a játék vizuális elemeit tartalmazza.
- TextMesh Pro: A fejlett szövegmegjelenítéshez szükséges „asset”-ek.

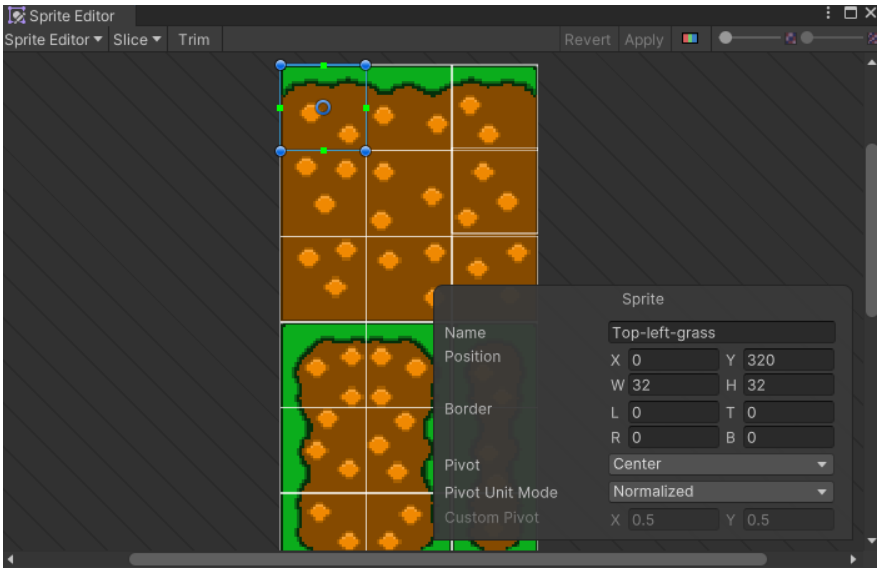
3.2.8. Aseprite és sprite kezelés

A projektben használt „sprite”-ok nagy része saját készítésű, egyedül a játszható karakter grafikája származik külső forrásból [11]. A „sprite”-ok szerkesztéséhez az „Aseprite” alkalmazás került használatra, amely kifejezetten pixel art grafikák készítésére és animálására alkalmas. Az alkalmazás lehetővé teszi „spritesheet”-ek létrehozását, vagyis olyan képfájlok elkészítését, amelyek több egymást követő „sprite”-ot tartalmaznak egyetlen képben (lásd 91).



91. ábra: Aseprite alkalmazás szerkesztő felülete

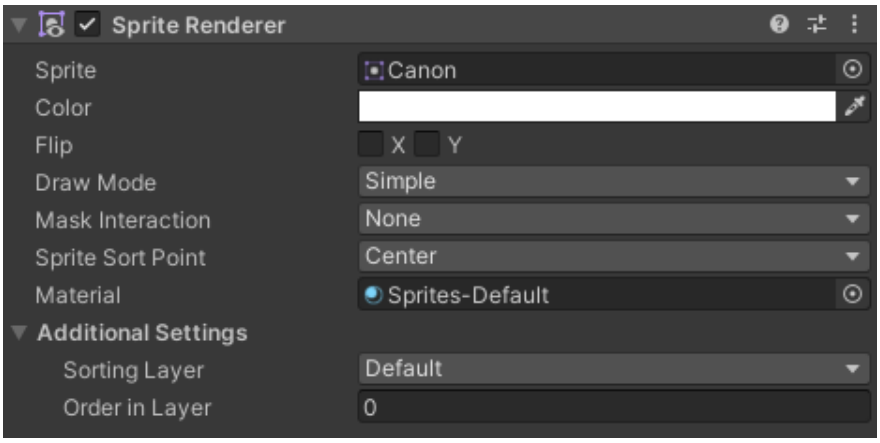
A „spritesheet”-ek Unity-be importálása után a „Sprite Editor” segítségével történik azok feldarabolása. Ebben az eszközben manuálisan vagy automatikus szeleteléssel különálló „sprite”-okra bontható a „spritesheet”, amelyeket ezt követően önálló grafikaként használhatunk.



92. ábra: Unity Sprite Editor

A „sprite”-ok játékbeli megjelenítéséhez a „SpriteRenderer” komponens használható. Ha egy „GameObject”-hez „sprite”-ot szeretnénk csatolni, elegendő hozzáadni ezt a komponenst, majd a megfelelő „sprite”-ot kiválasztani. A „SpriteRenderer” gondoskodik a grafika helyes

kirendereléséről, valamint további beállításokat is biztosít, például a „sorting layer” amely szintén fontos szerepet játszik, hiszen a játék több rétegre épült.



93. ábra: "Sprite Renderer" komponens beállításai

3.2.9. „GameObject”-ek és „Prefab”-ok

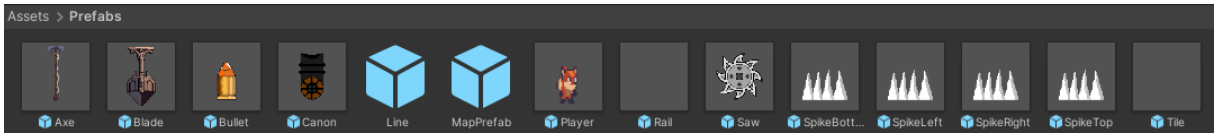
Az Unity működésének alapját a „GameObject” objektumok képezik. Minden, ami a „scene”-ben létezik, legyen az egy karakter, egy interaktív tárgy, egy háttérelem vagy akár egy üres logikai objektum, mint a „GridManager” vagy „GameManager” az „GameObject” formájában jelenik meg.

A GameObject-ek önmagukban még nem tartalmaznak megjelenést vagy működést, ezek csupán konténerek. A viselkedést és a funkciókat a komponensek adják hozzá. Ide tartoznak a megjelenésért felelős elemek, mint a SpriteRenderer vagy az Animator, valamint a fizikai működést biztosító komponensek, például a Rigidbody2D vagy a Collider2D. A saját logika hozzáadását a MonoBehaviour alapú scriptek teszik lehetővé [12].

A MonoBehaviour az az Unity által biztosított alaposztály, amelyből a legtöbb játéklogikát megvalósító script származik. Ezek az osztályok mindig egy GameObject komponenseként léteznek, és a GameObject AddComponent műveletével hozhatók létre. A MonoBehaviour biztosítja azokat a keretrendszer által hívott életciklus metódusokat, amelyek megkönnyítik a fejlesztést, például a minden képkockában futó Update metódust. Ha egy GameObject törlésre kerül, az összes hozzá tartozó komponens, így a MonoBehaviour alapú scriptek is automatikusan eltávolításra kerülnek [13].

A GameObject-ek létrehozhatók kézzel a szerkesztőben vagy futásidőben kódból, azonban a gyakorlatban sokkal elterjedtebb megoldás a Prefab használata. A Prefab tulajdonképpen egy előre konfigurált GameObject sablon, amely tartalmazhat komponenseket, beállításokat és

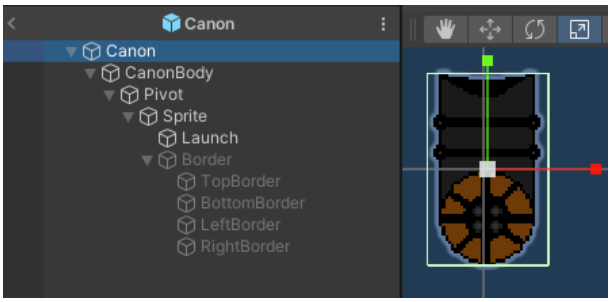
gyermek objektumokat. A Prefab rendszer legnagyobb előnye, hogy ha a Prefab asseten módosítunk, az összes példány automatikusan frissül a jelenetekben. Ez nagyon hatékonyvá teszi az objektumok újrafelhasználását és karbantartását [14].



94. ábra: "Prafabs" mappa tartalma

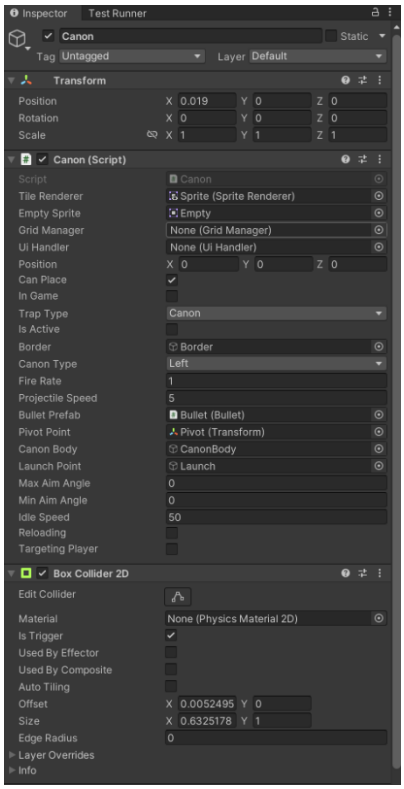
3.2.10. „Canon prefab” felépítése

A „Canon prefab” egy összetett játékelem, amely több egymáshoz kapcsolódó „GameObject”-ből áll. A felépítése olyan módon lett kialakítva, hogy a lövedékek indításához, az ágyú animált forgatásához és a pálya geometriájához való alkalmazkodáshoz minden szükséges vizuális és funkcionális struktúrát tartalmazza.



95. ábra: „Canon prefab” hieararchiája

A prefab gyökéreleme a „Canon” nevű „GameObject”, amely a működésért felelős összes logikai komponenst magában foglalja. Ez tárolja az ágyú pozícióját, iránybeállításait, valamint a működéséhez szükséges időzítési és sebességi paramétereket. A működési logikát a hozzá tartozó kód valósítja meg, amely kezeli az ágyúcső elforgatását, a célzási folyamatot és a lövések indítását.

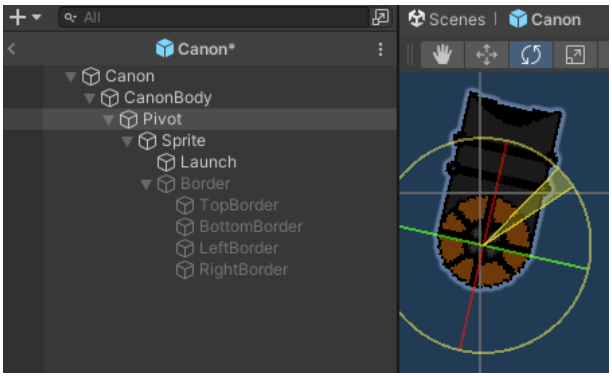


96. ábra: Canon object referanciái

A „Canon” objektum teljes blokkméretű „Collider” komponenssel rendelkezik. Erre elsősorban a pályaszerkesztőben van szükség, mivel így az ágyú egyetlen egységként kezelhető. A map editor a teljes „Canon” objektumot egy blokknyi területre illeszti, ami lehetővé teszi az elhelyezés pontos, rácsalapú kezelését. Ugyanez a „Collider” felel az egérinterakciók, például az „OnMouse” alapú kijelölés és a „hover” visszajelzés érzékeléséért is.

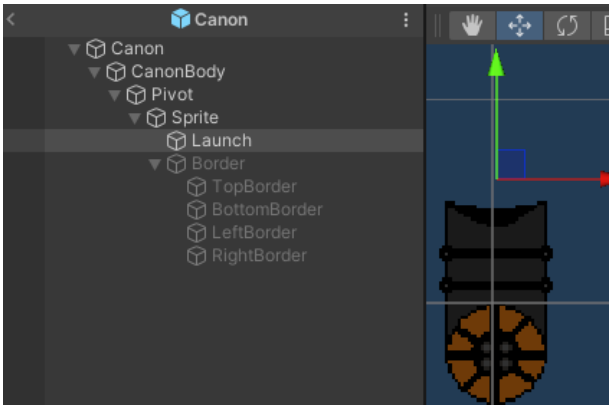
A grafikai megjelenítésért a „CanonBody” nevű gyermekobjektum felel. Ez tartalmazza az ágyú tényleges „sprite”-ját, valamint egy, a sprite méretéhez illeszkedő saját „Collider” komponenst. A „CanonBody” lokális pozíciója nincs fixen a blokk közepére rögzítve: a pálya generálásakor a rendszer a szomszédos blokkok elhelyezkedése alapján eltolásit és forgatást számol, és ezt a „CanonBody” lokális pozíciójára alkalmazza. Ennek köszönhetően ugyanaz a prefab használható különböző környezetekben, miközben az ágyú teste a blokkon belül eltolva, a környező geometriához igazodva jelenik meg, ahogyan azt a 25. ábra és 26. ábra szemlélteti.

A prefab egyik legfontosabb eleme a „Pivot” objektum, amely a forgatás középpontját jelöli. Az ágyúcső minden elfordulása ehhez a ponthoz viszonyítva történik, és a lövedékek iránya is ebből származik. A „Pivot” használata lehetővé teszi, hogy a „CanonBody” pozicionálása és az ágyúcső forgatása egymástól logikailag függetlenül valósuljon meg.



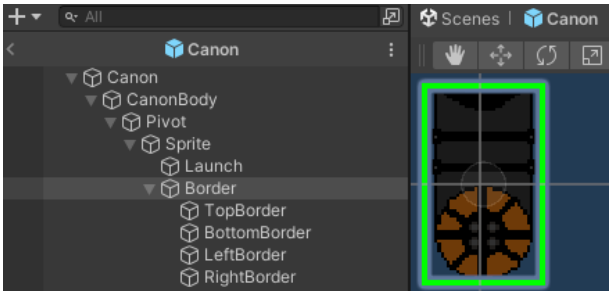
97. ábra: A forgatás középpontja

A lövedékek kiindulási pontját a „Launch” nevű gyermekobjektum határozza meg, amely kizárólag a lövés kezdőpozícióját rögzíti. A lövedék iránya a „Pivot” aktuális elforgatási állapotából származik, így a két objektum együtt biztosítja a pontos célzást és a konzisztens lövedékmozgást.



98. ábra: Lövedék indításának pozíciója

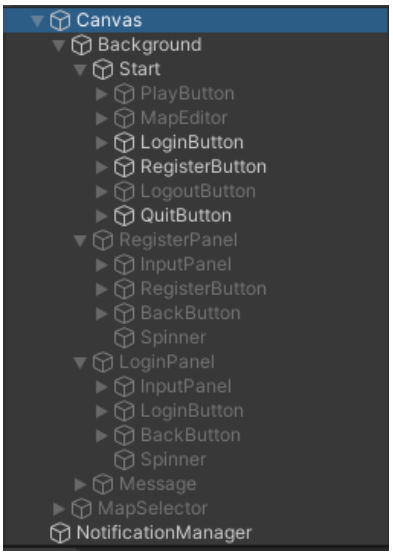
Minden csapda prefab része egy „Border” nevű vizuális segédelem, amely négy vékony „SpriteRenderer” elemből áll. Ezek a „MapEditor”-ban jelennek meg, amikor egy csapdaelemet kiválasztunk, és a blokk köré rajzolt kerettel jelzik az aktív, szerkesztés alatt álló elemet. Játékmódban ezek az elemek rejtve maradnak.



99. ábra: „Border” objektum aktválva

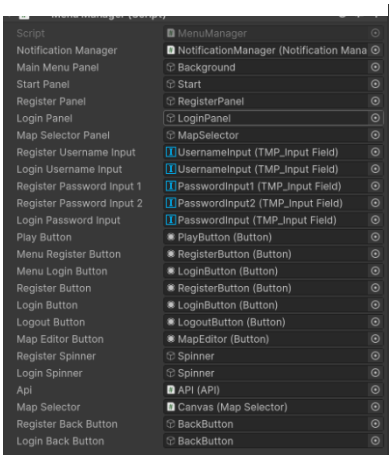
3.2.11. UI panelek vezérlése

A felhasználói felület több elkülönített panelből épül fel, amelyek mind a fő „Canvas” részeként jelennek meg. A panelek irányítása az Unity-ben komponensalapú módon történik. Minden UI-elem a gomb, szövegmező, háttérelem vagy panel egy-egy GameObject-hez kötődik, amelynek megjelenítése vagy elrejtése a „SetActive” metódussal szabályozható. A kezelőlogika lényege, hogy mindig csak az a panel aktív, amelyre a felhasználónak adott pillanatban szüksége van, így a felület rendezett és áttekinthető marad.



100. ábra: „Canvas” objektum tartalma

A panelek közötti váltás minden esetben egy kijelölt vezérlőscriptben történik, amely a Canvas hierarchiájában elhelyezett UI elemekre hivatkozik. A vezérlés lényege, hogy egy panel aktiválása előtt a korábban látható panel inaktiválásra kerül, így elkerülhető az UI elemek egymásra csúszása vagy párhuzamos megjelenése. Az Unity Inspector nézete lehetővé teszi, hogy a panelekhez tartozó GameObject-ek és komponensek, ilyenek a gombok, input mezők, szövegek, töltésjelzők közvetlenül összekapcsolhatók legyenek a kóddal.



101. ábra: GameObjectek referenciája

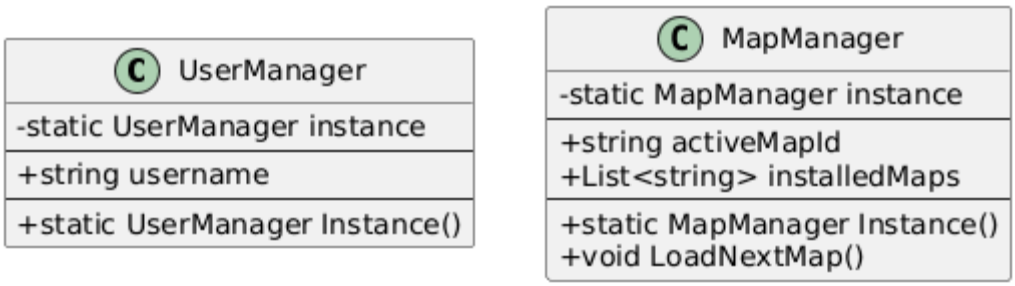
Az alábbi példa szemlélteti a panelek közötti váltás alapelvét. A metódus egy korábban aktív panelt elrejt, majd megjelenít egy másikat

```
public void ShowRegisterPanel()
{
    startPanel.SetActive(false);
    registerPanel.SetActive(true);
}

1 reference
public void ShowStartPanel()
{
    startPanel.SetActive(true);
    registerPanel.SetActive(false);
    loginPanel.SetActive(false);
}
```

102. ábra: Panelek ki-be kapcsolása

3.2.12. Felhasználó és pálya kezelés

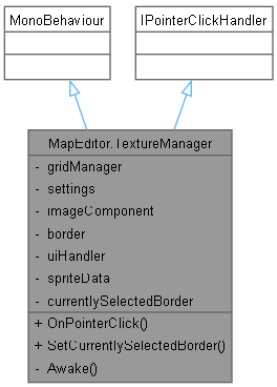
A felhasználó és pályaadatok kezelésére a rendszer „Singleton” mintát alkalmaz, így ezek az adatok minden jelenetből közvetlenül elérhetők. A felhasználói adatokat a „UserManager” tárolja, míg a pályák kezelését a „MapManager” végzi, amely az aktív pálya azonosítóját és a telepített pályák listáját tartja nyilván. A „Singleton” megoldás biztosítja, hogy a pályaszerkesztő és a játékmód ugyanazokkal az adatokkal dolgozzon, és a pályaváltás vagy felhasználói műveletek állapota a jelenetek közötti átlépéskor is megmaradjon.



103. ábra: „UserManager” és „MapManager” osztálydiagramjai

3.2.13. Blokk választó

A pályaszerkesztőben található blokkválasztó feladata, hogy a felhasználó kiválaszthassa, melyik elem kerüljön elhelyezésre a rácson. A választást a „TextureManager” komponens kezeli, amely minden listaelemhez hozzárendelve fut. A komponens az Unity „IPointerClickHandler” interfészét valósítja meg, így minden elem saját kattintáskezelővel rendelkezik.



104. ábra: „TextureManager” osztálydiagramja

A kiválasztott elem adatai egy „SpriteData” objektumban kerülnek eltárolásra, amely tartalmazza a „sprite”-ot, a típust, valamint csapdák esetén a csapdatípust. A típus meghatározása az Unity „Tag”-ek alapján történik, így a rendszer azonnal eldönti, hogy blokk vagy csapda került kiválasztásra, és ennek megfelelően frissíti a beállítási panelt a csapdák speciális paramétereire.

A blokkválasztó vizuális visszajelzését egy, minden elemhez tartozó „Border GameObject” biztosítja. Annak érdekében, hogy több nézet között külön-külön megőrizhető legyen a legutóbb kiválasztott elem, a rendszer egy „View”-onként tárolt dictionaryt használ.

```
public enum SpriteType
{
    Empty,
    Block,
    Trap,
    Rail,
    Spawn,
    Finish
}

[Serializable]
public class SpriteData
{
    public string name;
    public Sprite sprite;
    public SpriteType type;
    public TrapType trapType;
}
```

105. ábra: „SpriteData” objektum

```
private static Dictionary<View, GameObject> currentlySelectedBorder = new Dictionary<View, GameObject>
{
    { View.Sky, null },
    { View.Blocks, null },
    { View.Traps, null }
};
```

106. ábra: Kiválasztott elem megőrzése

3.2.14. „Tile” objektum

A „Tile” objektum (lásd 74. ábra) felel a pályaszerkesztő rácsának minden mezőjéért. Megjeleníti az adott blokkot vagy csapdát, kezeli az egérrel történő interakciókat, valamint meghatározza a különböző elemek elhelyezhetőségi szabályait. Minden „Tile” egy „SpriteRenderer” segítségével jelenik meg, míg a hozzá tartozó „SpriteData” tartalmazza a „sprite”-ot és az elem típusát.

A „Tile” inicializálását az „Init” metódus végzi, amely beállítja a pozíciót és betölti a „GridManager” és „UiHandler” komponenseket. Az „Update” metódus gondoskodik arról, hogy a „PauseMenu” megnyitásakor vagy szerkesztés közben a blokk megfelelő átlátszósággal jelenjen meg.

A felhasználói interakciót az „OnMouseEnter”, „OnMouseExit” *és* „OnMouseOver” metódusok kezelik. Kurzor belépéskor megtörténik a „hover effect”, kilépéskor visszaállítja az eredeti kinézetet, míg az egér lenyomott állapotában az „OnMouseOver” már ténylegesen elvégzi az adott műveletet, blokk vagy csapda elhelyezését, eltávolítását, illetve sínrajzolást.

Az „UseTool” metódusban történik, az aktuálisan kiválasztott eszközhöz és réteghez kapcsolódó műveletek. Az elhelyezhetőségi szabályokat a „HandleHover” metódus és a hozzá tartozó specializált függvények valósítják meg.

3.2.15. Rácsos szerkezet generálása

```
private void GenerateGrid()
{
    tiles = new Dictionary<Vector3, Tile>();
    for (int z = 0; z < 2; z++)
    {
        GameObject layer = new GameObject($"Layer {z}");
        layer.transform.parent = gridParent;
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                Tile newTile = Instantiate(tilePrefab, new Vector3(x, y, z), Quaternion.identity, layer.transform);
                newTile.name = $"Tile {x} {y} {z}";
                newTile.Init(this, new Vector3(x, y, z));
                newTile.TileRenderer.sortingOrder = z;
                tiles[new Vector3(x, y, z)] = newTile;
            }
        }
    }
    UpdateColliders();
}
```

107. ábra: A rács generálásáért felelős metódus

A pálya rácsszerkezetét a „GridManager” állítja elő. A szerkesztő két külön réteget használ ezért a rács generálása három egymásba ágyazott ciklus segítségével történik. Az X és Y koordináta határozza meg a mező rácsbeli helyét, míg a Z érték azt jelzi, melyik réteghez tartozik az adott „Tile”. A „GridManager” minden Z-réteghez egy külön „GameObject”-et hoz létre, majd a ciklusokban legenerált „Tile” objektumokat ennek a „GameObject”-nek adja szülőként. Minden „Tile” egyedi „Vector3” pozíció alapján kerül eltárolásra a „tiles dictionary”-ban, így a pozíciók később visszakereshetők és gyorsan kezelhetők.

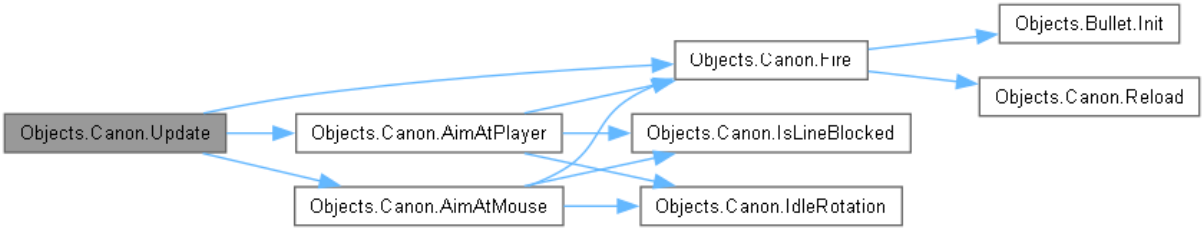
A rács elkészülte után a „GridManager” azonnal frissíti a blokkok „collider” beállításait. A szerkesztőben mindig csak az aktuálisan kiválasztott réteg „collider”-jei vannak bekapcsolva, míg a többi rétegen ez kikapcsolt állapotban marad. Erre azért van szükség, hogy az egérrel végzett műveletek, mint a „hover” effektek vagy a kattintások ne akadjanak össze a háttérréteg objektumaival.

3.2.16. A „Canon” megvalósítása

A csapda inicializálása az „Awake” metódusban történik, ahol eltárolódik a kezdőpozíció. A működés módját a „Start” metódus állapítja meg, amely annak megfelelően tölti be a „GridManager”, „MapLoader” vagy „Player” objektumokat, hogy a csapda szerkesztő módban vagy játék közben fut. A metódus végén meghívódik az „UpdateCanon”, amely beállítja az orientációt és a csapda testének pozícióját a szomszédos blokkok figyelembevételével.

108. ábra: „Start” metódus hívási gráfja

Az „Update” metódus vezérli a csapda folyamatos működését. A csapda vagy a játékost követi, vagy az egérmutatóra céloz, esetleg automatikus pásztázó mozgásban marad. A célzási műveleteket az „AimAtPlayer” és az „AimAtMouse” metódusok valósítják meg. Ezek meghatározzák, hogy a pivot melyik irányba fordulhat el, és figyelembe veszik az adott „CanonType” beállításait. A csapda csak akkor mozdul el, ha a látóvonala szabad, amit az „IsLineBlocked” ellenőriz raycast alapján.



109. ábra: „Update” metódus hívási gráfja

A támadási logika az „Fire” metódusban működik. A csapda ilyenkor egy „Bullet” példányt hoz létre a „launchPoint” pozícióján, majd elindítja a „Reload” metódust, amely meghatározott időre megakadályozza az újabb lövést. A kilőtt lövedék iránya mindig a pivot aktuális orientációjához igazodik.

```
public void Fire()
{
    if (!reloading)
    {
        Bullet bullet = Instantiate(bulletPrefab, launchPoint.transform.position, launchPoint.transform.rotation);
        bullet.Init(projectileSpeed, launchPoint.transform.up);
        reloading = true;
        StartCoroutine(Reload());
    }
}

1 reference
private IEnumerator Reload()
{
    yield return new WaitForSeconds(fireRate);
    reloading = false;
}
```

110. ábra: Lövés mechanizmusa

A csapda testének eltolását az „UpdateCanon” végzi, amely a „HasBlockNeighbor” metódus segítségével megállapítja, hogy milyen blokkszomszédok vannak a csapda körül. Ennek alapján

korrigálja a „canonBody” pozícióját, hogy az illeszkedjen a pálya geometriájához. A metódus végén a „CalculateMinMaxRotation” határozza meg a forgatás lehetséges tartományát.

Amikor a csapda nem tud érvényesen célozni, a „IdleRotation” metódus lép működésbe. Ez lassú pásztázó mozgást hoz létre a megengedett szögtartományon belül, és mindaddig aktív marad, amíg a csapda újra nem talál célpontot.

```
private void IdleRotation()
{
    float range = maxAimAngle - minAimAngle;
    float idleAngle = Mathf.PingPong(Time.time * idleSpeed, range) + minAimAngle;
    if (float.IsNaN(idleAngle) || float.IsInfinity(idleAngle))
        idleAngle = 0f;
    pivotPoint.transform.localEulerAngles = new Vector3(0f, 0f, idleAngle);
}
```

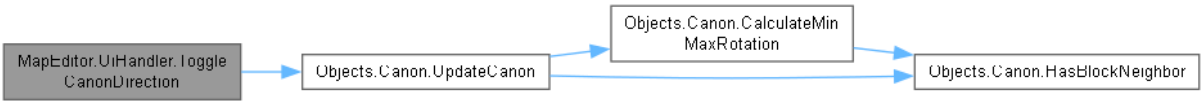
111. ábra: Ágyú forgatása

3.2.17. A „Canon” tulajdonságainak módosítása

A csapdák szerkesztését egységes logika kezeli, az UI-ban végrehajtott módosítások minden kijelölt csapdára azonnal érvényesek, a frissítés pedig mindig a csapda saját típusától függ. A fejezetben bemutatott működés tehát nem kizárólag a „Canon” csapdára vonatkozik, hanem minden csapdatípus ugyanazon implementációs elvet követi, a különbség csupán a módosítható tulajdonságokban és az azokhoz tartozó UI elemekben jelenik meg.

A „Canon” csapda konfigurálása a „Settings” és a „UiHandler” osztályokon keresztül történik, amelyek a csúszkák és „toggle” gombok aktuális állapotát figyelik, majd a változásokat továbbítják az aktív csapdák felé. A módosítások során minden érintett „Canon” objektum azonnal frissül, így a beállítások a szerkesztőben valós időben láthatók.

A csapda irányváltását a „ToggleCanonDirection” metódus végzi. A metódus lekérdezi, melyik irányválasztó „toggle” aktív, majd ennek megfelelően módosítja a „gridManager.currentCanonDirection” értékét. A rendszer ezután minden kijelölt „Canon” objektumban beállítja az új irányt, és meghívja az „UpdateCanon” metódust, amely frissíti a csapdatest pozícióját és a pivot forgatási tartományát.



112. ábra: „ToggleCanonDirection” hívási gráfja

A játékosra való célzás engedélyezését vagy tiltását a „ToggleCanonTargeting” metódus kezeli. A UI „toggle” aktuális állapota alapján frissíti az összes aktív „Canon” csapda „targetingPlayer” értékét, majd a „ResetPosition” metódus visszaállítja a csapda kezdő pozícióját, hogy a módosítások azonnal érvényesüljenek.

113. ábra: „ToggleCanonTargeting” hívási gráfja

A beállító panel megjelenítését az „UpdateCanonSettingsView” metódus végzi. A metódus mindig a szerkesztés alatt álló elem típusához igazítja a felület tartalmát. Ha a kijelölt objektum nem „Canon” csapda, akkor a panel automatikusan elrejtésre kerül. Ha viszont „Canon” került kiválasztásra, a felület minden eleme a csapda aktuális tulajdonságait jeleníti meg, vagyis a „fireRate” és „projectileSpeed” csúszkák a megfelelő értékekre állnak, és az irányválasztó „toggle” gombok is azt az irányt jelölik ki, amelyet a csapda jelenleg használ.

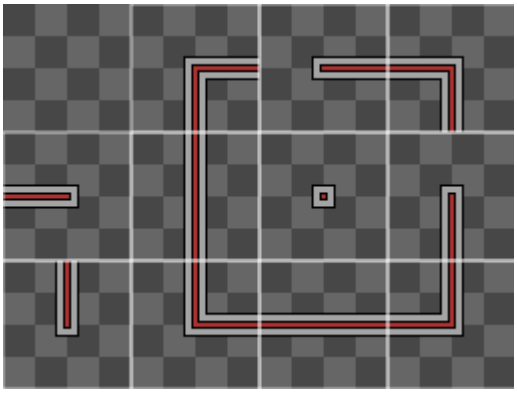
A csúszkák működéséhez tartozó eseményfigyelők a „Settings” osztály „Start” metódusában kerülnek hozzáadásra. A „Reload” és „Bullet Speed” csúszkák „onValueChanged” eseménye közvetlenül módosítja minden aktív „Canon” azonos nevű tulajdonságát, miközben frissíti a hozzájuk tartozó UI feliratokat is.

```
Slider canonFireRate = canonFireRateSlider.GetComponentInChildren<Slider>();
canonFireRate.onValueChanged.AddListener(value =>
{
    TextMeshProUGUI label = canonFireRateSlider.GetComponentInChildren<TextMeshProUGUI>();
    label.text = "Reload: " + value.ToString("0.00") + " sec";
    var canons = gridManager.GetActiveTraps()
        .Where(t => t.trapType == TrapType.Canon)
        .Cast<Canon>();
    foreach (var canon in canons)
    {
        canon.fireRate = value;
    }
});
```

114. ábra: Csúszka eseménykezelés

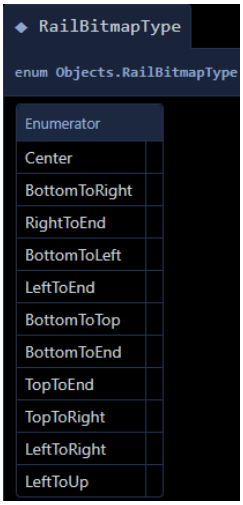
Ennek az egységes, valós időben frissülő szerkesztési logikának köszönhetően a „Canon” és minden más csapda gyorsan és áttekinthető módon paraméterezhető, miközben a beállítások mindig szinkronban maradnak a pálya aktuális állapotával.

3.2.18. Sín „wrappelés”



115. ábra: Sín "spritesheet"

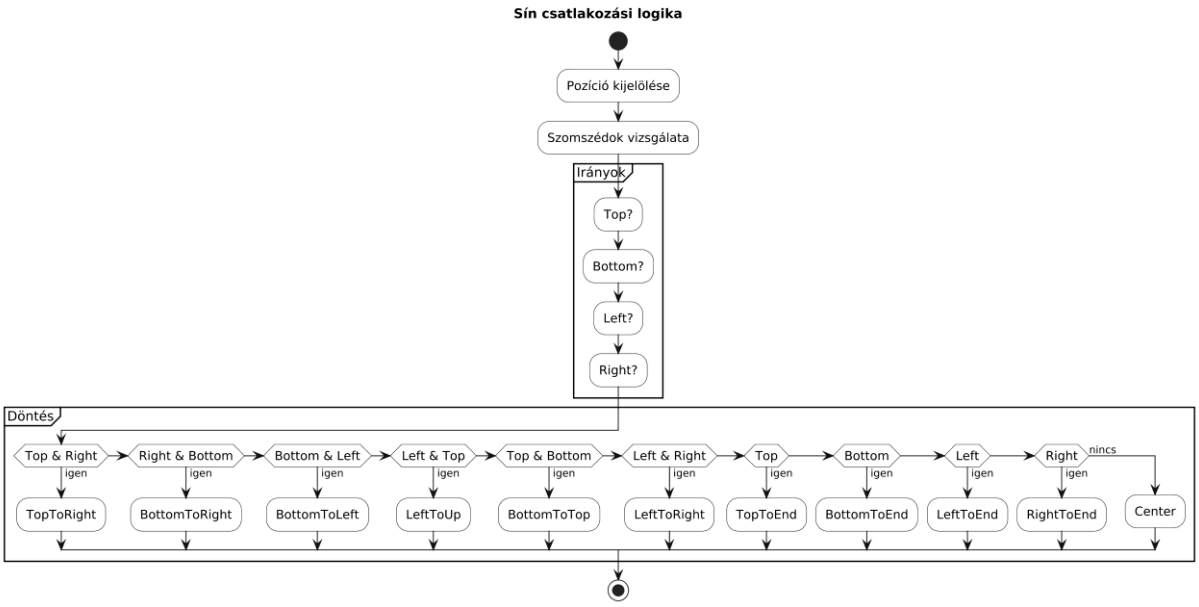
A szerkesztőben elhelyezett minden sín egy előre definiált grafikai formát kap. Ezeket a formákat a „RailBitmapType” felsorolt típus tartalmazza. A felsorolás tizenegy sínformát definiál. Minden forma egy csatlakozási mintát ír le. Például a „BottomToRight” olyan sínszakasz, amely lefelé és jobbra kapcsolódik tovább, a „LeftToRight” pedig egy vízszintes egyenes szakaszt jelöl.



116. ábra: RailBitmapType felsorolás

A megfelelő sínforma kiválasztása a szomszédos sínek vizsgálatán alapul. A „PaintRail” metódus létrehozza az új sín objektumot, majd az „UpdateRailAndNeighbors” meghívásával az algoritmus meghatározza, hogy az adott pozícióhoz melyik sínforma illeszkedik. A vizsgálat négy irányban történik. A rendszer megkeresi, hogy az adott sín felett, alatt, balra és jobbra található-e másik sín. Ezután a „DetermineRailType” metódus ellenőrzi, hogy a szomszédos sínek milyen formákhoz tartoznak, és mely irányokban engedélyezik a további kapcsolódást.

Minden sínforma meghatározza, hogy mely irányokba csatlakozhat tovább. Az algoritmus ezek alapján dönti el, hogy melyik grafikai elem felel meg a vizsgált helyzetnek. Például a „BottomToRight” forma csak akkor választható, ha a sín alatt és jobbra is található olyan sín, amely tovább tud csatlakozni a megfelelő irányba. A „BottomToTop” forma akkor kerül kiválasztásra, ha a sín alatt és felett is sínelem található, és mindkét szomszéd támogatja a függőleges összeköttetést.



117. ábra: Sínlogika folyamatábra

A rendszer nem csupán azt vizsgálja, hogy van-e sín a szomszédos mezőkön, hanem azt is, hogy ezek a szomszédok mely irányokban képesek kapcsolódni. A sín kiválasztása ezért nem egyszerű irányonkénti összeadás eredménye, hanem egy olyan szabályalapú döntési folyamaté, amely a csatlakozások érvényességét vizsgálja.

A sínformák meghatározását követően a „SetSprite” metódus állítja be a megfelelő grafikai elemet, és frissíti a sínhez tartozó adatokat, így biztosítva a helyes megjelenítést és a pontos mentést.

```
public void SetSprite(RailBitmapType type)
{
    this.TileRenderer.sprite = RailSprites[(int)type];
    this.currentType = type;
    this.SpriteData = new SpriteData
    {
        name = RailBitmap[type],
        sprite = RailSprites[(int)type],
        type = SpriteType.Rail
    };
}
```

118. ábra: Sínhez tartozó adatok módosítása

A rendszer egy zárolási mechanizmust is alkalmaz. Egy sín zárolttá válik, ha két érvényes irányba is kapcsolódik. A zárolt síneket az algoritmus később nem módosítja, így nem jöhet létre nem kívánt átrendeződés akkor, amikor a felhasználó új elemeket helyez el a közelben. A „lockolás” lehetővé teszi, hogy több egymás mellett futó sínpálya elkülönülten kezelhető legyen.

Amikor egy sín törlésre kerül, a rendszer először feloldja a környező sínek zárását, majd újraszámolja a törölt sín körül lévő összes elem bitmap típusát. A folyamat rekurzív jellegű, vagyis a törlés minden érintett sínformát újraértékel. Ennek eredményeként a sínrendszer mindig konzisztens marad, a törlések és bővítések után is pontosan illeszkedő formákat jelenít meg.

Ez az algoritmus lehetővé teszi, hogy a szerkesztőben rugalmas sínrendszer jöjjön létre. A sínhálózat minden bővítése és módosítása automatikusan igazodik a környezetéhez, miközben a „lockolási" rendszer gondoskodik arról, hogy a párhuzamos pályák és bonyolult szerkezetek stabilan és kiszámíthatóan működjenek.

3.2.19. API kommunikáció

A szerverrel történő adatcserét a rendszer egy dedikált komponense, az „API" osztály valósítja meg. Ez az osztály felel a teljes hálózati működésért, a „HTTP" kérések létrehozásáért, elküldéséért, a JSON formátumú válaszok feldolgozásáért és a hibakezelésért. Az API minden metódusa „coroutine" formájában fut, így a hálózati kommunikáció nem blokkolja a játékmenetet. A kérések technikai lebonyolításához a „UnityWebRequest" osztály szolgál eszközként.

```
public void GetUserLeaderboard(string mapId, string userId)
{
    StartCoroutine(GetUserLeaderboardRequest(mapId, userId));
}

2 references
public IEnumerator GetUserLeaderboardRequest(string mapId, string userId)
{
    string url = $"{baseURL}leaderboard/{mapId}/{userId}";


    using (UnityWebRequest request = UnityWebRequest.Get(url))
    {
        request.certificateHandler = new AcceptAllCertificatesSignedWithASelfSignedCertificate();

        yield return request.SendWebRequest();

        if (request.result == UnityWebRequest.Result.Success)
        {
            Leaderboard response = JsonConvert.DeserializeObject<Leaderboard>(request.downloadHandler.text);
            gameManager.playerDeathCount = response.current_deaths ?? 0;
            gameManager.playerLeaderboardCount = response.leaderboard_deaths;
            gameManager.UpdateDeathCountUI(gameManager.playerDeathCount);

        }
        else
        {
            yield return CreateLeaderboardEntryRequest(mapId, userId);
            yield return GetUserLeaderboardRequest(mapId, userId);
        }
    }
}
```

119. ábra: „GetUserLeaderboardRequest" metódus

A működés szemléltethető a „GetUserLeaderboardRequest” metóduson keresztül. A függvény egy GET kérést épít fel, majd a „SendWebRequest” hívással indítja el a kommunikációt. A projekt szervere önaláírt tanúsítványt használ, ezért minden kéréshez külön tanúsítványkezelő szükséges. Enélkül az Unity alapértelmezés szerint visszautasítaná a kapcsolatot.

A válasz JSON formátumban érkezik, amelyet az API automatikusan egy „Leaderboard” objektummá alakít. Sikeres válasz esetén a játékos aktuális statisztikái frissülnek, és a felhasználó halálozási adatai megjelennek a játékon belüli felületen. Ha azonban a szerver hibát küld vissza, az általában azt jelzi, hogy a játékosnak még nincs bejegyzése az adott pályához. Ilyenkor az API automatikusan létrehoz egy új rekordot, majd rekurzívan saját magát hívja meg, sikeres beolvasást eredményezve. Ez biztosítja, hogy az első pályaindításkor a „leaderboard” adatok mindig elérhetőek legyenek.

```
[System.Serializable]
8 references
public class Leaderboard
{
    1 reference
    public string map_id;
    2 references
    public string user;
    2 references
    public int? current_deaths;
    5 references
    public int? leaderboard_deaths;
}
```

120. ábra: "Leaderboard" paraméterei

A hívás a pálya betöltésekor történik meg, a rendszer két korábban bemutatott Singleton osztályán, a „MapManager” és a „UserManager” objektumokon keresztül. A projekt további API metódusai, például a regisztráció, a bejelentkezés, a térképfeltöltés vagy az eredménytábla frissítése ugyanezt a mintát követik.

```
api.GetUserLeaderboard(MapManager.Instance.activeMapId, UserManager.Instance.username);
```

121. ábra: "Leaderboard" lekérés hívása

3.2.20. Pályák exportálása

A pályák mentését a szerkesztő felület „Mentés” gombja indítja, amely az „OnSaveButton” metóduson keresztül ellenőrzi a pálya kötelező elemeit. A rendszer csak akkor engedi a mentést, ha a felhasználó létrehozott egy „Spawn” és egy „Finish” pontot, illetve a pályanév sem üres.

A tényleges exportálási folyamat a „CreateMap” metódusban zajlik. A metódus először összegyűjti a teljes rácson található blokkok, csapdák és sínek adatait, majd ezekből egy Map nevű szerkezetet állít össze. A pályához egyedi azonosító készül, amely alapján a szerkesztő létrehozza a mentéshez szükséges könyvtárat. A pálya teljes tartalma JSON formátumban kerül mentésre. A mentés során a rendszer létrehozza a pálya előnézeti képének helyét is, majd minden aktív csapda kijelöltségét megszünteti. A folyamat végén egy külön kamera

segítségével képernyőkép készül a pályáról, és az így létrejött kép és adatfájl a szerveren is rögzítésre kerül.

```
public void CreateMap(string mapName)
{
    string sky = uiHandler.sky.GetComponent<Image>().sprite.name;
    Map map = new Map(tiles, rails, sky);
    string mapId = Guid.NewGuid().ToString();
    var jsonSettings = new JsonSerializerSettings
    {
        NullValueHandling = NullValueHandling.Ignore
    };
    string json = JsonConvert.SerializeObject(map, Formatting.Indented, jsonSettings);

    API.MapData mapData = new API.MapData
    {
        map_id = mapId,
        created_by = UserManager.Instance.username,
        map_name = mapName,
        data = map
    };

    if (!File.Exists(Application.dataPath + "/Maps"))
    {
        Directory.CreateDirectory(Application.dataPath + "/Maps");
    }
    if (!File.Exists(Application.dataPath + $"/Maps/{mapId}/"))
    {
        Directory.CreateDirectory(Application.dataPath + $"/Maps/{mapId}/");
    }
    string path = Application.dataPath + $"/Maps/{mapId}/{mapId}";
    File.WriteAllText(path + ".json", JsonConvert.SerializeObject(mapData, Formatting.Indented, jsonSettings));
    foreach (var trap in activeTraps)
    {
        trap.isActive = false;
        trap.SetBorder();
    }
    ScreenshotHandler.TakeScreenshot_Static(1920, 1080, path, () =>
    {
        api.CreateMap(mapId, UserManager.Instance.username, mapName, json, File.ReadAllBytes(path + ".png"));
    });
}
```

122. ábra: Pálya fájlba mentése

3.2.21. Képernyőkép készítése

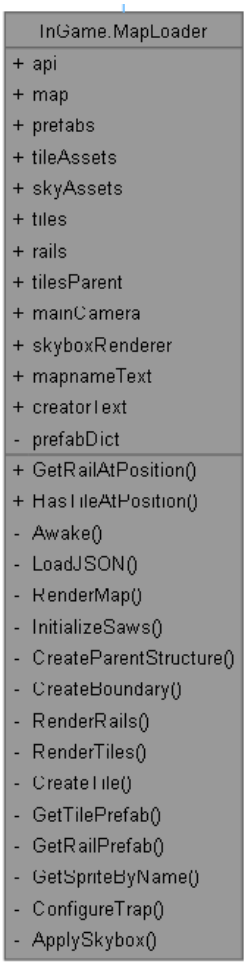
A szerkesztő nem a teljes játékablakról készít képet, hanem külön kamerát használ kizárólag az előnézeti kép előállítására. A „ScreenshotHandler” osztály vezérli ezt a folyamatot. Amikor a „CreateMap” metódus képkészítést kér, a rendszer előkészíti a kamerát egy ideiglenes renderfelülettel, majd a következő képkocka végén az „OnPostRender” függvényben megtörténik a tényleges képrögzítés. A kamera által kirajzolt kép egy „Texture2D” objektumba kerül, amelyet a rendszer PNG formátumba kódol, majd elment a megfelelő mappába. A kép elkészülése után a rendszer visszaadja a vezérlést az export folyamatnak, amely azonnal elküldi a fájlt a szerverre. A módszer előnye, hogy kizárólag a pályarács és a játékmező jelenik meg a képen, az UI elemek nem zavarják az előnézetet.

3.2.22. Pályák importálása

A pályák betöltéséért a „MapSelector” felel, amely egyszerre kezeli a helyileg tárolt pályákat és a szerverről érkező új pályákat. A rendszer először átnézi a helyi mappát, és minden megtalált JSON fájlt, UI elemmé alakít, hogy a felhasználó egyszerűen választhasson közülük. Ezután betölti az API segítségével lekért pályalistát, és minden olyan pályáról, amely nem található meg lokálisan, létrehozza a könyvtárat, elmenti a JSON adatot, majd letölti az előnézeti képet is. A „MapSelector” mindkét forrást egységes módon jeleníti meg, így a felhasználó ugyanabból a listából érheti el saját és mások által készített pályákat is. A pálya kiválasztásakor a rendszer frissíti az aktív pálya azonosítóját.

3.2.23. Dinamikusan renderelt játékterek

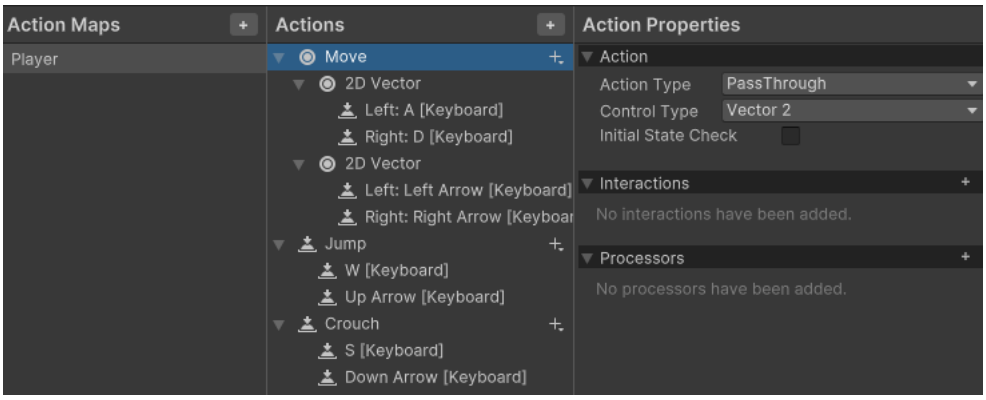
A játék közbeni pályamegjelenítést a „MapLoader” végzi, amely a szerkesztőben elmentett JSON fájl alapján építi fel újra a teljes játékteret. A rendszer először beolvassa a pálya adatait, majd a JSON-ban szereplő minden egyes objektumtípust a megfelelő Unity „prefab” alapján példányosít. A csapdákhoz tartozó beállítások, például időzítések, sebességek vagy mozgásirányok, a JSON-ben tárolt trapSettings struktúrából kerülnek visszaállításra, így minden csapda pontosan ugyanúgy működik, mint a szerkesztés során. A renderelés során külön rétegek jönnek létre a háttér és a játszható réteg számára, a „collider”-ek pedig a réteg alapján kapnak megfelelő fizikai beállításokat. A síneket a rendszer a mentett bitmap típus alapján állítja vissza, így a pályaszerkesztőben kialakított sínformák minden esetben pontosan újraépülnek. A „MapLoader” a háttérképet is beállítja, majd elindítja a mozgást igénylő elemeket. A pálya így teljesen dinamikus módon töltődik be, és minden esetben a szerkesztőben mentett állapotot reprodukálja.



123. ábra: "MapLoader" osztálydiagramja

3.2.24. A karakter

A játékos irányítását a „Player” osztály valósítja meg, amely teljes egészében felelős a mozgásért, az ugrásért, az ütközések kezeléséért, a kamera felé továbbított animációs paraméterekért, valamint a halál és az újraéledés folyamatáért. A komponens egy több részre osztott logikát alkalmaz, amelyben a „Unity Input System”, a fizikai számítások és az animátor működése szorosan együttműködik.



124. ábra: „Unity Input System”

A mozgás alapját a „Rigidbody2D” komponens biztosítja, amelynek vízszintes sebességét a játékos aktuális bemenete határozza meg. A bemeneteket az „InputActionReference” objektumokból olvassa ki. Az aktuális vízszintes sebesség abszolút értéke közvetlenül átadásra kerül az animátornak „Speed” paraméterként, így a sétálás és futás animációi automatizáltan aktiválódnak.

```
if (!isCrouching)
{
    movement = move.action.ReadValue<Vector2>();
    animator.SetFloat("Speed", Mathf.Abs(movement.x));
}
else
{
    movement = Vector2.zero;
}
```

125. ábra: Mozgás vektor beolvasása és a futás animáció beállítása

Az ugrási rendszer több összetevőből áll. A „jump buffer” mechanizmus lehetővé teszi, hogy a játékos akkor is kapjon ugrást, ha a gombnyomás kicsivel a talajra érkezés előtt történik. A fizikai viselkedést külön gravitációs multiplikátorok módosítják. Lefelé gyorsabban esik a karakter, felfelé pedig rövid ugrás valósul meg, ha a játékos idő előtt felengedi az ugrás gombot. Az ugrás állapotai alapján frissülnek az animátor paraméterei is, mint az „IsJumping”, „IsFalling”.

A talajérintést „raycast” határozza meg. A karakter csak „Block” címkével ellátott objektumokat tekint talajnak, így biztosítható, hogy a csapdák vagy egyéb interaktív elemek ne váltsanak ki hamis „grounded” állapotot.

A guggolás két különböző „collider” segítségével működik. A normál álló helyzetben a „BoxCollider” aktív, míg guggoláskor a „CapsuleCollider” váltja fel. A „collider” csere biztosítja, hogy a karakter fizikailag is kisebb legyen, és be tudjon férni az alacsony helyekre.

A halálkezelés a „Die” metódusokon keresztül valósul meg. A karakter minden bemenete letiltásra kerül, a „collider”-ek kikapcsolnak, a „rigidbody” statikussá válik, majd a megfelelő animáció lefutása után egy „újraspawnolási” folyamat indul el. A játékos visszakerül a pálya kezdőpontjára, majd minden komponens visszaáll az eredeti állapotába. Ezzel egyidőben a halálszámláló frissül, majd mentésre kerül az API felé.

A rendszer utolsó elemeként a karakter megfordítása automatikusan történik: ha a mozgás pozitív irányú, a „sprite” jobbra, ha negatív, akkor balra néz. Ez minden animáció esetén konzisztensen működik, mivel közvetlenül a „transform” skáláját módosítja.

3.2.25. Animációk

A karakter animációs rendszere az Unity Animator környezetben készült, és több, egymással összekapcsolt állapotból áll. Az animátor vezérlését teljes egészében paraméterek biztosítják, amelyek a „Player” osztályból kerülnek beállításra minden képkockában. Ezek határozzák meg, hogy a karakter éppen milyen animációt jelenít meg.

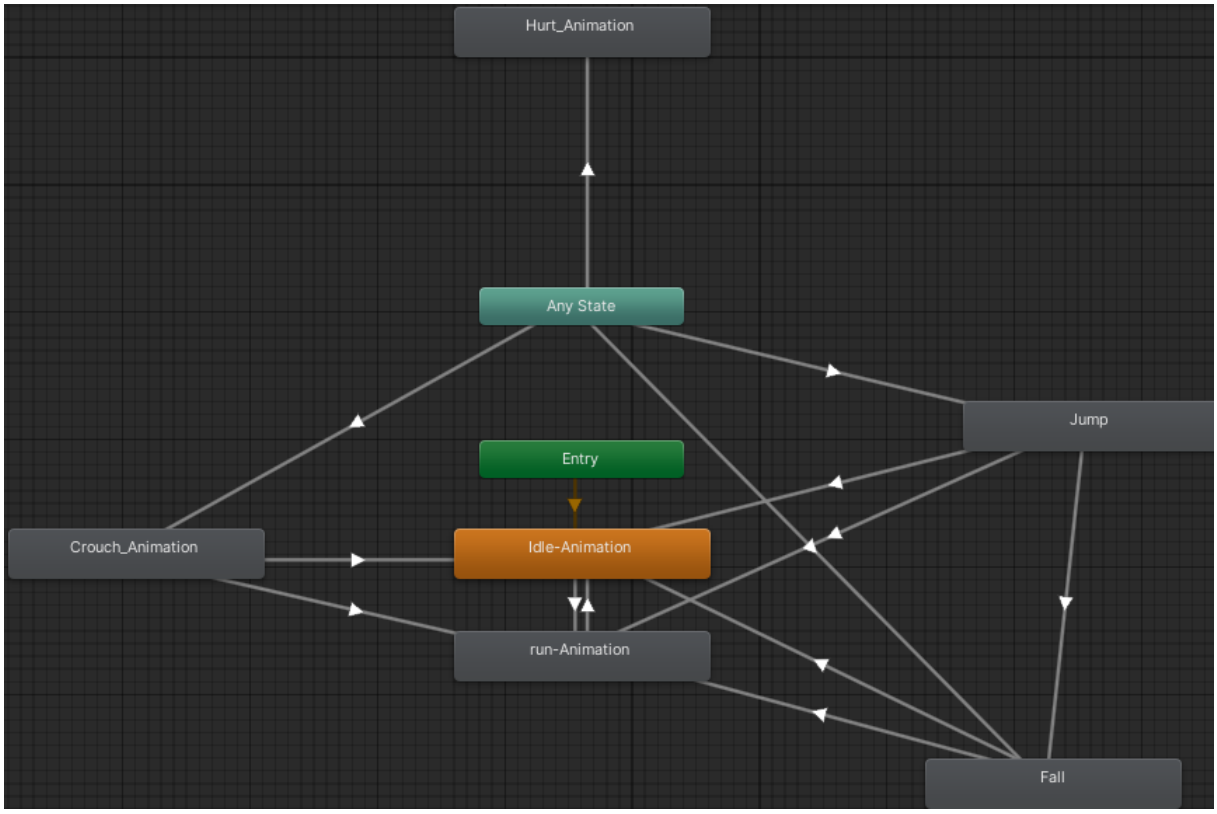
126. ábra: Animációk paraméterei

Az animátor a „Idle-Animation” állapotból indul. Ha a játékos vízszintes bemenetet ad, és a sebesség paraméter pozitív, automatikusan átlép a „run-Animation” állapotba. A guggolás bekapcsolása a „Crouch_Animation” állapotot aktiválja, amely mindaddig fennmarad, amíg a megfelelő InputAction nyomva tartott.

Az ugrás és esés animációi a karakter y irányú sebességéhez kapcsolódnak. Ha a sebesség pozitív, „IsJumping” válik aktívvá, míg lefelé irányuló mozgás esetén az „IsFalling” paraméter lép életbe. Ezek a paraméterek váltják ki a „Jump” és „Fall” animációk lejátszását. A talajra

érkezéskor mindkét paraméter kikapcsol, így a rendszer visszatér a normál földön végrehajtott animációkhoz.

A halál animáció működését a „Die trigger” kezeli. Amikor a karakter bármely csapdával vagy lövedékkel ütközik, a „Player” osztály a „trigger” beállításával áttér a „Hurt_Animation” állapotba. Mivel ez az állapot az Animator „Any State” pontjáról is elérhető, a halál animáció bármely más animációs állapotból azonnal és megszakítás nélkül indítható. A halál animáció végrehajtása alatt a karakter fizikailag inaktív állapotba kerül, majd az újraéledés logikája visszaállítja a standard működést.



127. ábra: A karakter animációs gráfja

3.3. Tesztelés

A rendszer minőségének biztosítása érdekében a fejlesztés során kétféle tesztelési megközelítést alkalmaztam. Manuális funkcionális tesztelést és automatizált egységtesztelést. Az automatizált tesztek elsősorban az egyes objektumok logikai működésére koncentrálnak, és nem az UI elemeit.

3.3.1. Manuális tesztelés

A manuális tesztelést a teljes fejlesztés során folyamatosan végeztem. Minden új metódus, UI elem vagy játékelem elkészítése után azonnal kipróbáltam annak működését, hogy a hibák még

a rendszer összetettebb részei előtt kiderüljenek. A pályaszerkesztőt különböző blokkok és csapdák elhelyezésével, törlésével, rétegek közötti váltással és eltérő rácspozíciókkal teszteltem. Ellenőriztem, hogy a „hover” logika helyesen jelezze a zöld és piros állapotokat. A csapdatípusokat külön-külön vizsgáltam, a tüskéknél a „delay” és az időzítések szinkronját, a fűrésznél a sínekhez való igazodást és irányváltást, az ágyúnál a látómező változását különböző blokkelrendezések mellett. A guillotine-t több magasságban helyeztem el, hogy az esési és visszahúzódási idő minden esetben működjön.

A pálya mentésénél ellenőriztem, hogy a rendszer felismeri-e a hiányzó „Spawn” vagy „Finish” elemeket, helyes hibaüzenetet ad-e, és csak érvényes pályát enged menteni. Minden mentés után megnyitottam a létrehozott JSON-fájlt és az előnézeti képet, hogy a tartalom megfelelő legyen. A betöltést több, helyben tárolt és szerverről érkező pályával is teszteltem, ellenőrizve, hogy a hiányzó képek automatikusan letöltődnek-e, és minden objektum a korábban mentett beállításokkal jelenik meg.

A játék közben a karakter mozgását és a fizikai működést különböző helyzetekben teszteltem. Vizsgáltam a rövid és hosszú ugrás működését, a „jump buffer” hatását, a gyorsított esést, a „crouch collider” váltását. A halál animációt és a „respawn” folyamatát minden csapdatípussal kipróbáltam és ellenőriztem a számláló növekedését valamint az API felé küldött frissítéseket. A pálya befejezésekor teszteltem, hogy a leaderboard mindig a megfelelő pályához frissül-e. A hálózati működést hibás és helyes adatokkal is vizsgáltam. Teszteltem a sikertelen bejelentkezést, a már létező felhasználó regisztrációját, a nem elérhető szerver esetén működő hibakezelést.

A használhatóságot több ismerősöm segítségével vizsgáltam. A pályaszerkesztőt, a menüket és a játékmenetet magyarázat nélkül használták, és a visszajelzések alapján minden funkció egyértelmű volt számukra. Ez azt mutatta, hogy a kezelőfelület, a kiemelések és az értesítések megfelelően vezetik a felhasználót, és a rendszer logikus működése külön magyarázat nélkül is követhető.

3.3.2. Automatizált tesztelés

A projekthez több, egymástól független automatizált teszt. Az „Editor” módban futó tesztek kizárólag az egyes metódusok és osztályok logikáját vizsgálják, míg a „PlayMode” tesztek a futás közbeni viselkedést ellenőrzik, valós időben, várakozási időkkel és Unity viselkedéssel együtt. Az összes teszt fájlt a „Scripts” mappán belüli „Tests” mappa tartalmazza.

Editor tesztek között szerepel a „Tile” osztály működésének ellenőrzése a pályaszerkesztő környezetben. Ezek a tesztek vizsgálják a blokkok helyes inicializálását, a „hover” logikáját, azt, hogy milyen színt kap a blokk különböző kiválasztott „sprite”-ok esetén, valamint azt is, hogy a törlő eszköz vagy a különböző „trap” típusok megfelelően viselkednek-e. A „MapeditorTileTest” kiterjed a blokk viselkedésére akkor is, amikor a játék szüneteltetve van, vagy amikor tiltott lerakási műveleteket próbál végrehajtani a felhasználó.

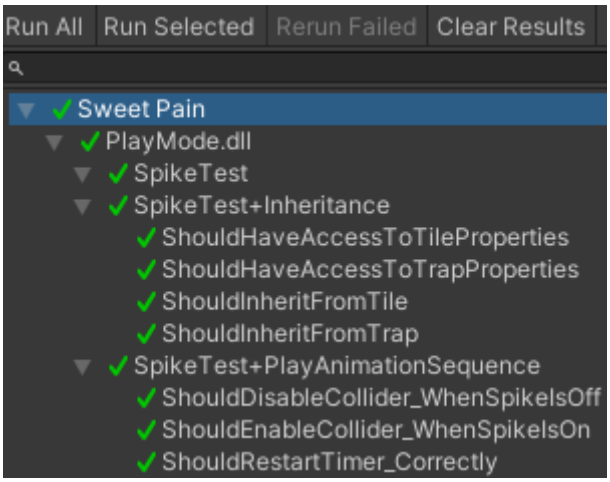
A „TrapTest” az alap csapda működését ellenőrzi, és különösen fontos szerepet tölt be az öröklési hierarchia validálásában. A teszt igazolja, hogy a „Trap” osztály valóban a „Tile”-ból származik, ezáltal hozzáfér a blokk megjelenítésért felelős komponensekhez. A teszt vizsgálja továbbá a csapda aktív és inaktív állapot közti váltást, és azt, hogy a „broder” objektum aktiválása és deaktiválása helyesen reagál-e.

```
[TestFixture]
0 references
public class Inheritance : TrapTest
{
    [Test]
    0 references
    public void ShouldInheritFromTile()
    {
        Assert.IsTrue(trap is Tile);
        Assert.IsInstanceOf<Tile>(trap);
    }

    [Test]
    0 references
    public void ShouldHaveAccessToTileProperties()
    {
        Assert.IsNotNull(trap.TileRenderer);
        Assert.AreEqual(trap.GetComponent<SpriteRenderer>(), trap.TileRenderer);
    }
}
```

128. ábra: Trap öröklődés teszt

A „PlayerTest” a játékos karakter működésének ellenőrzésére szolgál. A tesztek között szerepel a vízszintes mozgás vizsgálata, a halál utáni mozgás letiltása, az ugrási feltételek ellenőrzése és a zuhanási fizika megfelelő alkalmazása. Emellett teszteli a karakter animátorhoz kapcsolódó logikai jelzéseket és a különböző ütközési helyzetekben bekövetkező állapotváltozásokat.

A „PlayMode” tesztek valós időben futnak, várakozási időkkel és a „MonoBehaviour” viselkedést követve. A teszt ellenőrzi a csapda ütemezett animációját, azt, hogy a „collider” a megfelelő időpillanatokban ki és bekapcsol, és hogy a „RestartTimer” hívása valóban új ciklust indít.



129. ábra: "PlayMode" tesztek

A tesztek lefuttatásáról készült képernyőképek alapján minden automatizált teszt sikeresen lefutott, ami igazolja a logikai működés minden vizsgált esetben megbízható.

130. ábra: "EditMode" tesztek

4. Összefoglalás és további fejlesztési lehetőségek

A szakdolgozat elkészítése során egy olyan rendszert hoztam létre, amely méretében és összetettségében messze a legnagyobb saját projekt, amin eddig dolgoztam. A fejlesztési folyamat izgalmas volt, mert egyszerre kellett foglalkoznom a játékmechanikával, a grafikai elemekkel, a hálózati kommunikációval, a szerver felépítésével, a felhőben futó infrastruktúrával és az adatbáziskezeléssel. A munka jelentős része kutatással telt, NGINX konfiguráció, Docker, felhőalapú szerverkörnyezet, Unity belső rendszerei, FastAPI, API és számos kisebb technikai kérdés megoldása.

A cél egy olyan 2D platformer játék volt, amely nem csupán játszható pályákat kínál, hanem saját pályaszerkesztőt is tartalmaz, továbbá felhasználókhoz kötött online funkciókat is biztosít. A rendszer ennek megfelelően három fő részre épült. Az Unity alapú kliensre, a FastAPI szerverre és a PostgreSQL adatbázisra. A kliens oldalon valósult meg a teljes pályaszerkesztő logika, a csapdák működése, a sínrendszer, a karakterirányítás, a renderelés, az import-export folyamat és a játékbeli interakciók. A szerver oldalon készült el a regisztráció, bejelentkezés, pályakezelés, leaderboard kezelés és a fájlok kiszolgálása, míg az adatbázis tartós tárolóként működik.

A projekt minden eleme úgy került megvalósításra, hogy hosszú távon, karbantartható és bővíthető maradjon. A fejlesztést végig manuális teszteléssel követtem, metódusonként és funkciónként vizsgálva a rendszert. Emellett több automatizált teszt is készült a kritikus logikai részekre, például a Tile, Trap és Player működésére. A rendszer végül stabilan, kiszámíthatóan és egységesen működik, és teljes mértékben képes reprodukálni a szerkesztőben elkészített pályák logikáját játék közben is.

A munka eredményeként egy olyan játékrendszer jött létre, amely valódi pályák készítésére, megosztására és végigjátszására alkalmas, és amely technikai oldalról a kliens–szerver modellnek megfelelően működik.

A rendszer teljes egészében úgy került kialakításra, hogy hosszú távon, bővíthető és módosítható maradjon. Az egységes csapda- és pályakezelés, valamint a kliens-szerver kommunikáció elkülönített logikája lehetővé teszi, hogy a projekt több irányban is továbbfejleszthető legyen.

A pályaszerkesztő egyik legkézenfekvőbb bővítési területe a vizuális és tematikus elemek kibővítése. A jelenlegi blokkrendszer összetettebb tematikus készletekkel egészíthető ki, például külön „jég" vagy „tűz" környezetekkel. Ezekhez illeszkedően új csapdatípusok is bevezethetők, például csúszós felületek, időzített lávakiömlések vagy speciális környezeti elemek. A rendszer öröklési modellje és az egységes csapdalogika miatt a csapdák további bővítése minimális szerkezeti átalakítást igényelne.

A jelenlegi háttérválaszték további elemekkel egészíthető ki, és külön rétegezési logika is kialakítható, amely lehetővé teszi több háttérréteg használatát. A játékhangok támogatása szintén külön modulban valósítható meg, amely egységesen kezelné a karakterhangokat, a csapdákhoz tartozó effekteket és a háttérzenét.

A szerveroldalon további fejlesztési lehetőséget jelent egy CI/CD folyamat kiépítése. A Docker-alapú architektúra jól illeszkedik olyan automatizációs eszközökhöz, mint a Jenkins vagy GitHub Actions. A szerverfrissítések, az image-ek újraépítése, az automatizált tesztek futtatása és a deploy teljesen gépesíthető lenne. Ez lehetővé tenné a gyors frissítéseket és a hibák korai kiszűrését.

Funkcionális bővítési irány lehet továbbá egy közösségi rendszer kialakítása, például pályaértékeléssel, hozzászólásokkal vagy részletesebb statisztikákkal. A leaderboard jelenleg halálszámot kezel, de kiegészíthető lenne időeredményekkel vagy további teljesítménymutatókkal. A rendszer szerkezete alkalmas lenne többjátékos módintegrálására is

A játék alapjai stabilak, a rendszer pedig olyan struktúrában készült el, hogy a fenti funkciók mind technikailag megvalósíthatók legyenek anélkül, hogy az alkalmazás alaplogikáját újra kellene tervezni.

5. Irodalomjegyzék

[1] Windows 11 Specs and System Requirements. https://www.microsoft.com/en-us/windows/windows-11-specifications Accessed: 2025-11-08.

[2] Understanding the architecture of a 3-tier application. https://vfunction.com/blog/3-tier-application/ Accessed: 2025-11-13.

[3] Three-Tier Client Server Architecture in Distributed System – GeeksforGeeks. https://www.geeksforgeeks.org/computer-networks/three-tier-client-server-architecture-in-distributed-system/ Accessed: 2025-11-13.

[4] Client-Server Architecture - System Design – GeeksforGeeks. https://www.geeksforgeeks.org/system-design/client-server-architecture-system-design/ Accessed: 2025-11-13.

[5] What is Unity? – Unity Learn. https://learn.unity.com/tutorial/what-is-unity Accessed: 2025-11-13.

[6] What can Unity do? – Unity Learn. https://learn.unity.com/tutorial/what-can-unity-do Accessed: 2025-11-13.

[7] FastAPI - Introduction – GeeksforGeeks. https://www.geeksforgeeks.org/python/fastapi-introduction/ Accessed: 2025-11-14.

[8] What is Docker? | IBM https://www.ibm.com/think/topics/docker Accessed: 2025-11-14.

[9] What is Amazon EC2? - Amazon Elastic Compute Cloud https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html Accessed: 2025-11-14.

[10] Unity - Manual: Scenes. https://docs.unity3d.com/530/Documentation/Manual/CreatingScenes.html Accessed: 2025-11-14.

[11] Ansimuz: Sunny Land | 2D Characters | Unity Asset Store https://assetstore.unity.com/packages/2d/characters/sunny-land-103349 Accessed: 2025-11-15.

[12] Unity - Scripting API: GameObject https://docs.unity3d.com/ScriptReference/GameObject.html Accessed: 2025-11-16.

[13] Unity - Scripting API: MonoBehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html Accessed: 2025-11-16.

[14] Unity - Manual: Introduction to prefabs.
https://docs.unity3d.com/6000.2/Documentation/Manual/prefabs-introduction.html
Accessed: 2025-11-16.