

SZAKDOLGOZAT



MISKOLCI EGYETEM

Pályagenerátor algoritmus fejlesztése oldalnézetes játékokhoz

Készítette:

Szabó Martin

Programtervező informatikus

Témavezető:

Dr. Földvári Attila József

MISKOLC, 2024

MISKOLCI EGYETEM

Gépészszmérnöki és Informatikai Kar

Alkalmasztott Matematikai Intézet Tanszék

Szám:

SZAKDOLGOZAT FELADAT

Szabó Martin (WJFAOO) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: C# programozás, térkép generálás

A szakdolgozat címe: Pályagenerátor algoritmus fejlesztése oldalnézetes játékokhoz

A feladat részletezése:

A számítógépes játékok esetében általános problémát jelent a pályák, térképek el-készítése. Ezt jellemzően valamilyen pályaszerkesztő szoftver használatával oldják meg. Az utóbbi években tendenciaként megjelent, hogy mivel nagy mennyiségen van igény ilyen jellegű tartalmakra, ezért különféle generálási módszereket használnak. A szakdolgozat azt mutatja be, hogy erre milyen módszerek és lehetőségek vannak oldalnézetes platformer (side-scroller) játékok esetében.

Megtervezésre és implementálásra kerül (Unity keretrendszerben, C# programozási nyelven) egy platformer játék, és egy olyan szoftver, amely a játékos által megadott paraméterezés alapján képes automatikusan létrehozni egy teljes, játszható térképet. A dolgozatban részletesen bemutatásra kerülnek az ehhez felhasznált paraméterek, a generáló algoritmus és a kapott pályák értékelése, elemzése.

Témavezető: Dr. Földvári Attila József, egyetemi adjunktus

A feladat kiadásának ideje: 2022. szeptember 26.

.....
szakfelelő

EREDETISÉGI NYILATKOZAT

Alulírott **Szabó Martin**; Neptun-kód: WJFA00 a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Pályagenerátor algoritmus fejlesztése oldalnézetes játékokhoz* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc,évhónap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

.....

dátum

témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):

.....
.....
.....

konzulens (dátum, aláírás):

.....
.....
.....

3. A szakdolgozat beadható:

.....

dátum

témavezető(k)

4. A szakdolgozat szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....
.....

egyéb mellékletet (részletezve)

tartalmaz.

.....

dátum

témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve:

.....

dátum

szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata:

a bíráló javaslata:

a szakdolgozat véleges eredménye:

Miskolc,

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. Bevezetés	1
2. Az oldalnézetes játékok, valamint a játékmotorok jellemzői	2
2.1. A fejezet célja	2
2.2. Az oldalnézetes játékok	2
2.2.1. Fejlődésük	2
2.2.2. Általános jellemzőik	4
2.2.3. A pályageneráló algoritmusok kapcsolódása a platformer játékokhoz	5
2.3. Játékmotorok	7
2.3.1. Az Unreal Engine játékmotor	8
2.3.2. A Godot játékmotor	8
2.3.3. A Unity játékmotor	9
3. Pályageneráló algoritmusok	12
3.1. A térképgenerálás megtervezése	12
3.2. A térképgenerálás logikáját megvalósító algoritmusok bemutatása	13
3.2.1. A GenerateArray() metódus	13
3.2.2. A RenderMap() metódus	14
3.3. A Perlin zaj	15
3.3.1. A Perlin-zaj algoritmus implementálása és vizsgálata	17
3.4. A celluláris automata	19
3.4.1. A celluláris automata algoritmus implementálása és vizsgálata	20
3.5. A Random Walk algoritmus (Véletlen bolyongás)	25
3.5.1. A véletlen bolyongás algoritmus implementálása és vizsgálata	26
4. 4 2D platformer játék tervezése és fejlesztése a Unity játékmotor segítségével	37
4.1. A játék leírása	37
4.2. A Unity szerkesztője	38
4.2.1. 4.2.1 A GameObject, valamint a szülő-gyerek kapcsolat	38
4.3. Az irányítható hős és a kamera	39
4.3.1. A PlayerMovement osztály	41
4.3.2. A BetterJump osztály	47
4.3.3. A Grappling Hook mechanizmus	48
4.3.4. A MainCamera objektum és a hozzá tartozó szkript	50
4.4. Az irányítható karakter meginmálása	56
4.5. A szintek elkészítése	58
4.5.1. A Grib objektum, valamint a Tile Palette használata	58

4.5.2. A RuleTiles használata	59
4.6. Az ellenség és a felvehető objektumok elkészítése, valamint a karakter életvesztésének és halálának az implementálása	61
4.6.1. Az EnemyAI osztály	61
4.6.2. A LifeCount osztály	64
4.6.3. Az InteractionSystem és az Item szkript	64
4.7. A LevelManager és a SoundManager osztály	67
4.8. A felhasználói felület elkészítése	69
4.8.1. A főmenü	69
4.8.2. A szint kiválasztásáért felelős menü	70
4.8.3. Az opciók menü	71
5. A kész játék bemutatása	75
6. Összefoglalás	78
Irodalomjegyzék	79

1. fejezet

Bevezetés

A modern számítógépes játékpiac rohamos fejlődése mellet a játékfejlesztők folyamatosan keresik azokat az innovatív megoldásokat, amelyekkel újszerű és lenyűgöző játékélményt hozhatnak létre. Az oldalnézetes (side-scroller) platformer játékok, amelyek az elmúlt évtizedekben jelentős népszerűségre tettek szert, különleges teret biztosítanak a kreativitás és a technológia találkozásának. Azonban ezeknek a játékoknak a fejlesztése komplex kihívásokkal jár, különösen a pályatervezés tekintetében, ahol a fejlesztőknek egyensúlyt kell találniuk az innováció, a játékélmény és a fejlesztési erőforrások között.

A szakdolgozatomban kettős célt tűztem ki: egyrészt egy teljesen működő, játszható 2D platformer játék tervezése és implementálása a Unity keretrendszerben, C# programozási nyelven, másrészt egy hozzá kapcsolódó pályagenerátor algoritmus fejlesztése, amely képes automatikusan, a felhasználó preferenciáit alapul véve változatos és kihívást jelentő pályákat létrehozni. Ez a kettős megközelítés lehetővé teszi, hogy nem csak elméleti síkon vizsgáljuk a pályageneráló algoritmusokat, hanem valós játékörnyezetben is teszteljük azok hatékonyságát és hatását a játékélményre.

A szakdolgozatom kiterjed a platformer játék fejlesztésének minden aspektusára, beleértve a játékmechanika megtervezését, a grafikai elemek integrálását, valamint a felhasználói interfész megvalósítását. Mindezek mellett a fő hangsúly a pályagenerátor algoritmuson van, amely a játék alapvető részét képezi. Az algoritmus tervezésekor különös figyelmet fordítok a paraméterezenhetőségre és az adaptivitásra, hogy a generált pályák ne csak változatosak és kihívást jelentők legyenek, hanem jól illeszkedjenek a játék dinamikájához és stílusához.

A szakdolgozat során a platformer játék fejlesztési folyamatának minden lépését alaposan dokumentálom, a kezdeti koncepciótól a végleges implementációig. Ezen túlmenően, az algoritmus tervezése és implementációja során részletesen bemutatom a különböző programozási kihívásokat, a paraméterezési stratégiákat, és azokat a tesztelési módszereket, amelyekkel az algoritmus teljesítménye és a generált pályák játékbeli hatékonysága értékelésre kerül. A szakdolgozatom így nem csak egy konkért algoritmus kidolgozására vállalkozik, hanem hozzájárul a videójáték fejlesztés megismeréséhez is.

2. fejezet

Az oldalnézetes játékok, valamint a játékmotorok jellemzői

2.1. A fejezet célja

Ebben a fejezetben arra fogok törekedni, hogy részletesen bemutassam az oldalnézetes (side-scroller) játékok és a pályageneráló algoritmusok világát. Megvizsgálom az oldalnézetes játékok történelmi fejlődését, általános jellemzőit, és azt, hogy hogyan kapcsolódnak ezek a játéktípusok a pályageneráláshoz.

Ezen felül részletezni fogom a játékmotorok különböző típusait, előnyeit és hátrányait.

A hivatkozások jelentős része ehhez a fejezethez szokott kötődni. (Egy hivatkozás például így néz ki [11].) Itt lehet bemutatni a hasonló alkalmazásokat.

2.2. Az oldalnézetes játékok

2.2.1. Fejlődésük

A 2D-s platformjátékok fejlődése gazdag és változatos utazás a videójátékok történetében. Egy rövid áttekintést szeretnék adni a fejlődésükről:

1. A kezdetek: Ez a műfaj a „Space Panic”-kel (1980) kezdődött, de a „Donkey Kong” (1981) volt az, amely a létrák és az ugrálás kombinálásával igazán megteremtette a mércét. Ezek a korai játékok többnyire egyképernyős platformerek voltak.[11]
2. „Side-scroller” korszak: A „Super Mario Bros.” (1985) forradalmasította a műfajt a „side-scroller” pályákkal, emlékezetes karaktereket, „power-up”-okat és titkos útvonalakat vezetve be. Ebben a korszakban olyan játékok is megjelentek, mint a „Mega Man” és a „Metroid”, amelyek ezt a típusú játékstílust más elemekkel, például lövöldözéssel és felfedezéssel vegyítették.[11] A Super Mario Bros. játékmene a 2.1. ábrán látható.
3. A technológia fejlődése: A technológia fejlődésével a játékok elkezdtek pszeudo-3D elemeket tartalmazni. Az 1990-es években az olyan játékok, mint a „Crash Bandicoot” a platformer koncepciókat valódi 3D-s környezetbe helyezték.[11]



2.1. ábra. A Super Mario Bros. játékmenete

4. A 16 bites korszak: A „Mega Man X” és a „Donkey Kong Country” figyelemre méltó példái ennek az időszaknak. A 16 bites konzolok bevezetése lehetővé tette a feljavított párhuzamos görgetést és a részletesebb sprite-okat.[11]
5. Korai 3D korszak: A korai 3D platformerek közé tartoztak a 2,5D-s címek és a 3D-s perspektívájú, de 2D-s grafikájú platformerek. Az olyan játékok, mint a "Crash Bandicoot", lineáris pályákon maradtak, de vegyítették a járműveket és a másodpercek töredékeiben történő platformozást.[11] A Crash Bandicoot játék a 2.2. ábrán látható.



2.2. ábra. A Crash Bandicoot játék

6. Az indie újjáéledés: A 2000-es évek végén és a 2010-es években az indie fejlesztők jelentős szerepet játszottak a 2D-s platformjátékok újjáélesztésében, és inkább a

történetre és az innovációra összpontosítottak. Az olyan játékok, mint a "Braid", a "Limbo" és a "Super Meat Boy" egyedi mechanikájukkal és narratívájukkal mutatták be ezt a trendet.[11] A Limbo játék a 2.3. ábrán megtekinthető.



2.3. ábra. A Limbo játék

7. Modern korszak: A 2D platformjátékok az utóbbi években továbbra is népszerűek, gyakran a hagyományos játékmenetet modern tervezési elvekkel ötvözik. Az olyan címek, mint a "New Super Mario Bros." sorozat és a különböző indie játékok élénk és változatos műfajt tartanak fenn.[11]

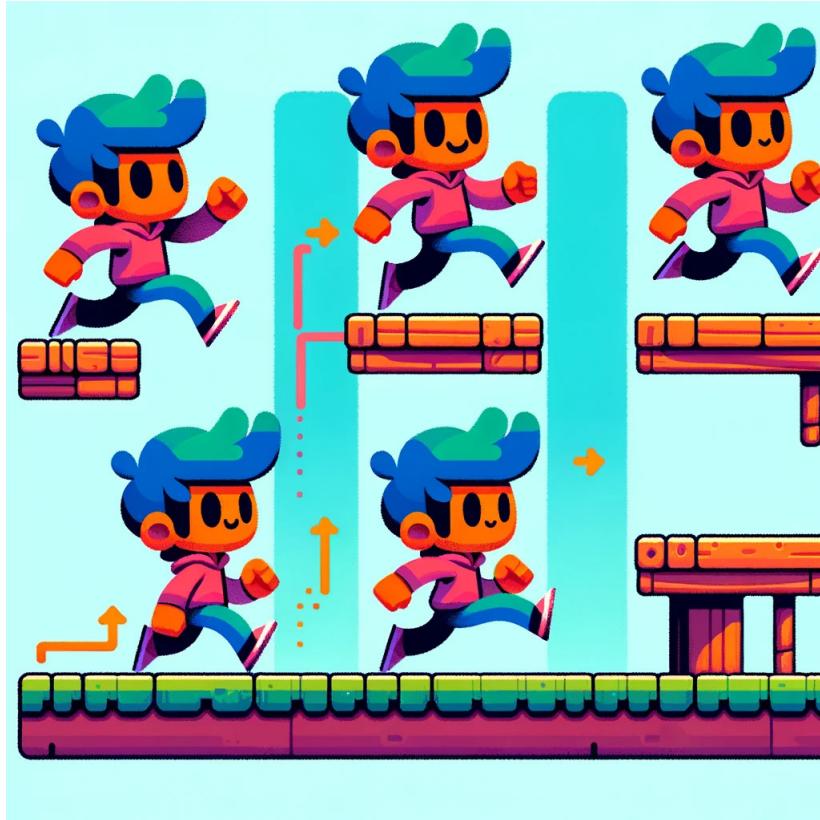
2.2.2. Általános jellemzők

A platformer játék, más néven platform videójáték, egy olyan játéktípus, amely jellemzően kétdimenziós grafikával rendelkezik, és amelyben a játékosok a képernyőn különböző platformokon ugráló vagy mászkáló karaktereket irányítanak.

A platformjátékokban egy karakter egy pályán navigál, hogy feladatait teljesítsen, magas pontszámokat érjen el, vagy egyszerűen csak életben maradjon. Mivel ez a játék-műfaj az évek során jelentősen megváltozott, sok ilyen játéknak más lehet a látványvílága. A következő jellemzők azonban gyakran megtalálhatók a platformjátékokban.[15]

- Interaktív környezet: Azt, hogy egy karakter mit tehet egy játékban, nagyban befolyásolja a szint vagy a környezet kialakítása. A platformjátékok célja különösen az, hogy próbára tegyék a játékost, miközben a főhőst olyan összetett akadályok elé állítja, mint a szöges platformok, halálos csapdák vagy lávával esetleg vízzel teli szakadékok.[15]
- Third-Person nézőpont: A játékos által irányított karakter az előtte lévő képernyőn látható, mivel sok platformjátékot úgynevezett „third-person” perspektívából készítenek.[15]
- Vízszintes és függőleges mozgás: A platformjátékok többsége kétdimenziós „side-scroller” játék, ami azt jelenti, hogy a játékos oldalról látja a karakterét, miközben

a képernyő vízszintesen vagy függőlegesen mozog vele együtt.[15] A horizontális és vertikális mozgásról egy kép a 2.4. ábrán látható.



2.4. ábra. A horizontális és vertikális mozgás

- Az ugrás kontrollálása: A játékos irányítja a karakter ugrási képességét, ami a platformjátékok egyik fő szempontja. Ezekben a játékokban az ugrás gyakran szükséges a környezetben való mozgáshoz és a következő szintre jutáshoz.[15]
- Történetmesélés és világépítés: Bár a korai platformjátékokban nem volt ennyire elterjedt, a modern 2D-s platformjátékok gyakran tartalmaznak gazdag történetmesélést és részletes világépítést a játékélmény fokozása érdekében.[15]

Ezek a funkciók együttesen hozzák létre azt az egyedi és gyakran kihívást jelentő élményt, amely a 2D platformer játékokat jellemzi. A műfaj az évek során jelentősen fejlődött, és minden játék a maga újításait és fordulatait vezette be ezekbe az alapvető összetevőkbe.

2.2.3. A pályageneráló algoritmusok kapcsolódása a platformer játékokhoz

A számítógépes játékokban sokszor sokkal több tartalom megjelenítését szeretnénk elérni, mint amennyit valójában elő tudunk állítani vagy el tudunk tárolni. Vagy a tartalom előállítása során felmerülő korlátozások miatt - pl. egy kis gyártócsapat esetében -, vagy a tartalom tárolása, esetleg forgalmazása miatt. [4]

Ezt azonban megkerülhetjük a procedurális generálással. Ez az, amikor a játék menetközben, játékidőben generál új tartalmat, ahelyett, hogy csak a korábban előállított

tartalmat használná fel. Ha jól csináljuk, ez gyakorlatilag korlátlan tartalmat biztosít hat a játékunkban, sokkal alacsonyabb előzetes előállítási költségekkel.[4]

A 2D-s platformjátékokban a térképgeneráló algoritmusok döntő szerepet játszanak a dinamikus és magával ragadó játékkörnyezetek létrehozásában. Ezek az algoritmusok generálhatnak térképeket előre létező szakaszok összerakásával vagy változó terepvízonyokkal rendelkező tájak rajzolásával. Egy gyakori módszer egy alapvonal megrajzolása (amely a talajt jelképezi), majd annak a magasságának a tájban való megváltoztatása a változatosság megteremtése érdekében.[4]

Ezeknek az algoritmusoknak a 2D platformerekben való használata lehetővé teszi egyedi, procedurálisan generált világok létrehozását, ami növeli az újrajátszhatóságot és a játékosok érdeklődését. minden egyes játékmenet más-más élményt nyújthat, a tájak az egyszerű és lapostól a komplex és többszintesig terjedhetnek.[4]

Több játék is nagyszerűen használta az procedurális generálást. Például a Rogue-light című játék, amely a 2.5. ábrán látható, bemutatja, hogy a procedurális generálással hogyan lehet mélyebb és sötétebb környezetet létrehozni, amely minden egyes játékmenettel változik.



2.5. ábra. A Rougelight játék

A "Diskophoros" egy másik érdekes cím, amely a gyors tempójú multiplayer akciót procedurálisan generált pályákkal kombinálja, így minden egyes játékmenet során új élményt tud nyújtani a játékosok számára. A Diskophoros játék a 2.6. ábrán látható.

Ezek a példák szemléltetik a procedurális térképgenerálás változatos alkalmazásait a 2D-s platformjátékokban, jelentősen hozzájárulva a játéktervezéshez és a játékosok általi érdeklődés növeléséhez.



2.6. ábra. A Diskophoros játék

2.3. Játékmotorok

A játékmotor olyan szoftveres keretrendszer, amelyet elsősorban videojátékok fejlesztésére terveztek. Ezek a motorok lehetővé teszik a játékfejlesztő cégek számára, hogy az összes munkájukat egy kész termékké egyesítsék. Manapság majdnem minden videójáték egy játékmotor segítségével készült. Azért nevezzük „motoroknak”, mivel ezek működtetik a teljes játékvilágot, amit az ember előre tárnak.[28]

A játékmotorok igen sokféle funkciót kínálnak, például 2D vagy 3D grafikus megjelenítést, ütközésérzékelő és -reagáló fizikamotort, hangot, szkriptelést, animációt, mesterséges intelligenciát, hálózatot, streaminget, memóriakezelést, szálkezelést, lokalizációs támogatást, jelenetgrafikát és videótámogatást a filmes jelenetekhez. Ezek a funkciók megkönnyítik a játékfejlesztés összetett folyamatait azáltal, hogy automatizálják a legtöbb játékprojektben előforduló ismétlődő feladatokat és így jelentősen csökkentik a költségeket, a komplexitást és a piacra kerülési időt.[28]

A játékmotoroknak két fő típusa van: a harmadik féltől származó motorok és a saját fejlesztésű motorok. A harmadik féltől származó motorokat vállalatok fejlesztik ki, hogy más stúdióknak adják bérbe őket. Ezeket a motorokat úgy tervezték, hogy különböző játékműfajokat és játékstílusokat támogassanak. A jól ismert harmadik féltől származó motorok közé tartozik az Unreal Engine és a Unity. Ugyanakkor a saját fejlesztésű motorokat egy játékstúdió házon belül, konkrét projektekhöz fejleszti, ami lehetővé teszi a játék követelményeihez jobban illeszkedő és testre szabható funkciókat.[28]

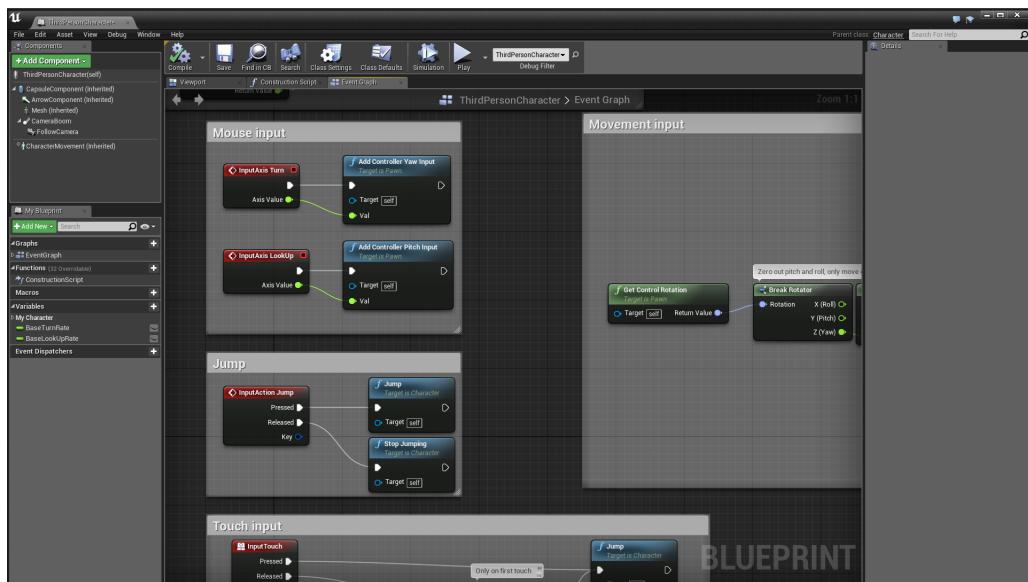
A játékmotorokat a játékfejlesztő csapat szinte minden tagja használja. A pályatervezők, az animátorok és a környezettervező művészek jelentős időt töltnek a motoron belüli munkával, a játék különböző elemeit alakítva a környezettől kezdve a karakterek mozgatásán át a világításig.[28]

A megfelelő játékmotor megválasztása több tényezőtől függ, például a projekt-költségvetéstől, a játék terjedelmétől, valamint a játékfejlesztő cég méretétől is függhet. Míg a kisebb stúdiók a költség- és erőforrás-korlátok miatt harmadik féltől származó motorok mellett dönthetnek, addig a nagyobb, több erőforrással rendelkező stúdiók saját motorokat fejleszthetnek a játékuk speciális igényeinek kielégítésére.[28]

2.3.1. Az Unreal Engine játékmotor

Az Epic Games által fejlesztett Unreal Engine gazdag múltra tekint vissza a videojátékfejlesztés világában. A motort eredetileg Tim Sweeney alkotta meg az 1998-ban megjelent "Unreal" című „first-person” lövöldözős játékhoz, de az évek során jelentősen fejlődött. Az első generációja a szoftveres renderelési képességeiről volt nevezetes, később pedig a dedikált grafikus kártyák teljesítményének kihasználásáról.[30]

Az Unreal Engine egy teljes körű, fejlett fizikai motorral rendelkező, nyílt forráskódú játékmotor, amelyet, ha nem kereskedelmi célra használunk, akkor ingyenes. Az Unreal Engine emellett támogatja a különböző platformokra való telepítést, többek között a Windows PC, PlayStation, Xbox, macOS, iOS és Android platformokra, és visszafelé kompatibilis az Unreal Engine 4 egyes korábbi verzióival. A játékmotort C++ nyelven írták, és ez is a hivatalos scripting nyelve, de a kezdő programozók bátran használhatják a motor Blueprint névre hallgató visual scripting rendszerét, amely a 2.7. ábrán látható.[30]



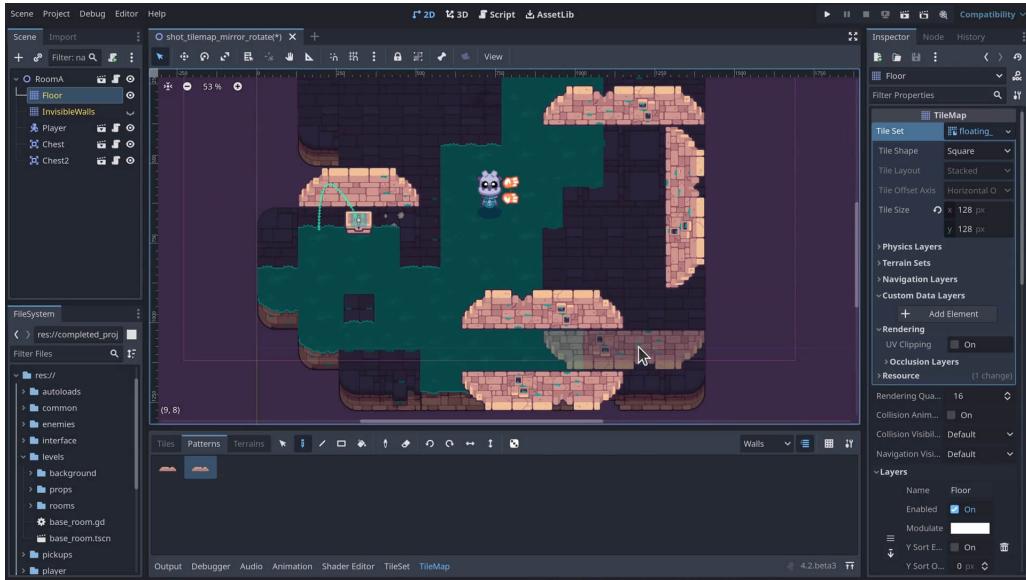
2.7. ábra. Az Unreal Engine Blueprint nevezetű visual scripting rendszere[9]

Az Unreal Engine népszerűsége és sokoldalúsága nem csak a rendkívül valósághű grafikai képességeinek köszönhető, hanem annak is, hogy a játékokon kívül is széles körben használják, például a film- és televíziós produkciónban.

2.3.2. A Godot játékmotor

A Godot Engine egy sokoldalú, ingyenes és nyílt forráskódú játékmotor 2D-s és 3D-s játékok készítéséhez. A Godot lehetővé teszi a videojáték-fejlesztők számára, hogy 3D-s és 2D-s játékokat készítsenek több programozási nyelv, például C++, C# és GDScript használatával. A programozásban kevésbé jártas játékfejlesztők használhatják a Godot visual scripting funkcióját is, amely a 2.8. ábrán látható. A fejlesztés megkönnyítése érdekében csomópontok hierarchiáját használja. Egy csomóponttípusból osztályok származtathatók, hogy speciálisabb csomóponttípusokat hozzanak létre, amelyek öröklik a viselkedést. A Godot szerkesztője támogatja az olyan asztali platformokat, mint a Linux, a macOS és a Windows, valamint az androidos telefonokat és táblagépeket.

Bár konzolokon is futtatható, a nyílt forráskódú licenckorlátozások miatt a népszerű konzolok hivatalos támogatása nem érhető el.[12]



2.8. ábra. A Godot játékmotor visual scripting rendszere [12]

A Godot nagy és aktív közösséggel rendelkezik, amely rengeteg forrást, oktatóanyagot és fórumot biztosít a tanuláshoz és problémamegoldáshoz. A Godot felhasználó-barátnak számít, különösen a kezdők számára, köszönhetően a könnyű kialakításának, a különböző hardvereken nyújtott hatékony teljesítményének és az aktív közösségenek, amely folyamatosan hozzájárul a fejlesztéshez. [12]

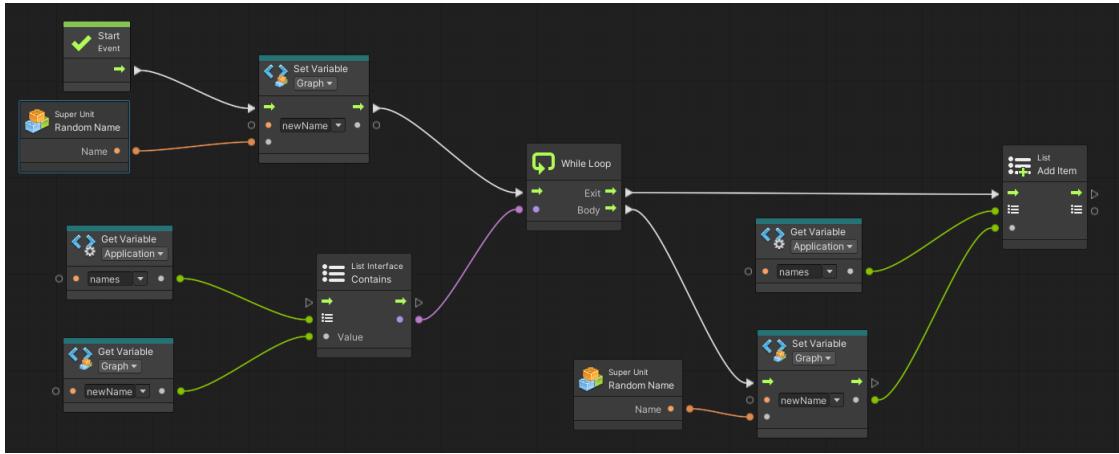
2.3.3. A Unity játékmotor

A Unity egy nagy teljesítményű és sokoldalú játékmotor, amely támogatja a 3D-s és 2D-s játékok, valamint az interaktív szimulációk létrehozását. A 2005-ös megjelenése óta a Unity több fejlesztő számára elérhetőbbé tette a játékfejlesztést különböző platformokon. Dacára annak, hogy a Unity egy Mac OS X játékmotorként indult, mára már számos asztali, mobil, konzolos és virtuális valóság platformot támogat. Különösen népszerű az iOS és Android mobiljátékok fejlesztésében, a kezdő játékfejlesztők számára könnyen kezelhetőnek számít, és népszerű az indie játékfejlesztők körében.

A Unity lehetővé teszi a felhasználók számára, hogy 2D-s és 3D-s játékokat és játékélményeket hozzanak létre. A Unity elsődleges programozási nyelvénél a C# nyelv lett kiválasztva, hozzáférhetősége és sokoldalúsága miatt. A programozási tapasztalattól függetlenül a C# felhasználóbarát környezetet biztosít a játékfejlesztésbe kezdő emberek számára. Egyszerű szintaxisa és az egyszerű felépítése zökkenőmentessé és élvezetessé teszi a tanulást és a kódírást. A C# egy objektumorientált programozási (OOP) nyelv, amely tökéletesen illeszkedik a játékfejlesztéshez. [13]

Aki nem ért annyira a programozáshoz, annak sem kell csüggendnie, hiszen a Unity-nek is van egy beépített visual scripting rendszere, amely a Bolt névre hallgat. Ez egy kódolás nélküli megoldás, amely lehetővé teszi, hogy bárki létrehozzon AI-rendszeret és játéklogikát egy csomópontokon alapuló vizuális felület segítségével. A visual scripting mechanikával vizuálisan meg tudjuk tervezni és össze tudjuk kapcsolni a csomópontokat, hogy komplex interakciókat és viselkedéseket hozzunk létre anélkül, hogy

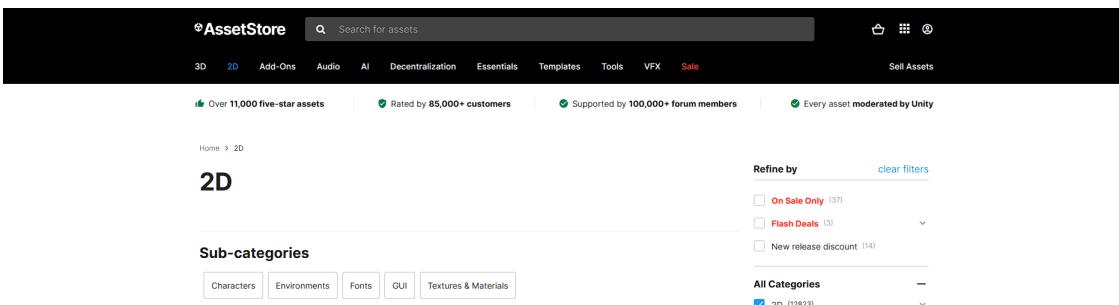
egyetlen sor kódot kellene írnunk. Lehetővé teszi a nem fejlesztők számára, hogy részt vegyenek a játékfejlesztésben, mivel felhasználóbarát és intuitív módot kínál ötleteik életre keltéséhez. [13] A Unity visual scripting rendszere a 2.9. ábrán látható.



2.9. ábra. A Unity Bolt névre hallgató visual scripting rendszere [16]

Érdemes megjegyezni, hogy a visual scripting nem korlátozódik a játéklogikára. A Unity a shaderekhez és vizuális effektekhez is kínál vizuális szkriptelési megoldást Unity Shadergraph néven. A Shadergraph lehetővé teszi lenyűgöző vizuális effektek létrehozását az árnyékolók és effektek megtervezésével egy csomópont-alapú felületen keresztül. [13]

A Unity Asset Store egy kincsesbánya, amely kiegészíti a Unity beépített funkcióit, és lehetővé teszi, hogy könnyedén fejleszthessük a játékunkat. Az ingyenes és fizetős assetek hatalmas választékával az Asset Store több mint 80 000 assethez biztosít hozzáférést, köztük 8000 ingyenes erőforráshoz, amelyeket megvásárlásukat követően közvetlenül a szerkesztőből importálhatunk a projektünkbe. Az Asset Store az assetek széles választékát kínálja az igényeinknek megfelelően. A kész modellektől, animációktól és textúráktól kezdve a sablonokon át a vizuális effektekig (VFX) minden megtalálható, ami ahhoz szükséges, hogy játékunkat színesítsük, valamint értékes fejlesztési időt takarítsunk meg. [13] A Unity Asset Store-jának egy kis részletekkel a 2.10. ábrán megtekinthető.



2.10. ábra. A Unity Asset Store nevezetű boltja, ahonnan a játékunkhoz szerezhetjük be a megfelelő sprite-okat [24]

Unity egy virágzó közösséggel büszkélkedhet, amely felbecsülhetetlen erőforrásként szolgál a fejlesztők számára. A különböző platformok és fórumok segítségével csatlakozhatunk a hasonlóan gondolkodó emberekhez, támogatást kérhetünk, és közösen

dolgozhatunk a többi emberrel a játékunk fejlesztése során. A Unity hivatalos fóruma egy nyüzsgő központot biztosít a fejlesztők számára, ahol vitatkozhatnak, kérdéseket tehetnek fel és megoszthatják tudásukat. Ez az információk kincsesbányája, ahol megoldásokat találhatunk a közös kihívásokra, új technikákat fedezhetünk fel, és naprakészek maradhatunk a legújabb iparági trendekkel kapcsolatban. A Reddit egy másik élénk közösség, ahol a Unity-rajongók összegyűlnek, hogy ötleteket cseréljenek, bemutassák munkájukat és támogassák egymást. Ezek a közösségek értékes platformként szolgálnak a világszerte működő fejlesztőkkel való kapcsolatteremtéshez, inspirációszerzéshez és tapasztalatcseréhez. [13]

Összességében a Unity nagyon jó választás a játékfejlesztéshez, mivel sokoldalú programozási nyelvet, bőséges tanulási forrásokat és egy élénk közösséget kínál.

3. fejezet

Pályageneráló algoritmusok

Ebben a fejezetben három darab generálási módszert fogok bemutatni, amelyek a Perlin zaj, a celluláris automata, valamint a Random Walk algoritmusok. Mindhárom algoritmust a Unity keretrendszerben implementáltam és vizsgáltam meg, hogy melyik módszer illeszkedne legjobban a játékomhoz.

3.1. A térképgenerálás megtervezése

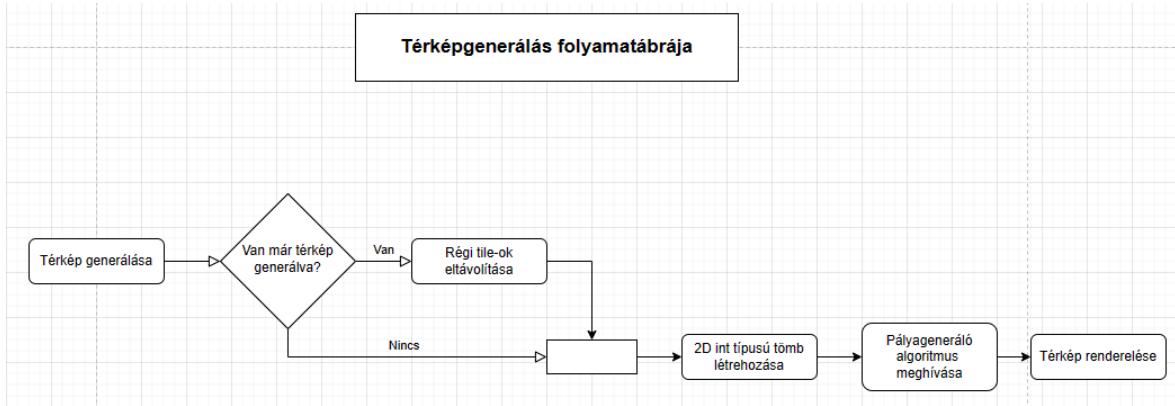
Egy olyan dungeon-stílusú térképet szeretnék létrehozni, amelynek a külső szélei minden falak, amelyek nem engedik leesni a játékost a pályáról, így a játék határai egyértelműen megmaradnak, valamint lehetővé teszi a játékosok számára, hogy szabadon navigáljanak anélkül, hogy áthatolhatatlan akadályokba ütköznek, például olyan falakba, amelyek megakadályozzák őket a tárgyak begyűjtésében. Tehát olyan pályageneráló algoritmus illeszkedne a játékomhoz, amely nem generál elzárt tereket, szobákat.

A térképgenerálás logikája, valamint a menete a következőképpen történik:

$$m_{ij} = \begin{cases} 1 & \text{ha a } \text{cella}(i, j) \text{ teli} \\ 0 & \text{ha a } \text{cella}(i, j) \text{ üres} \end{cases}$$

Bináris használatával különböztetjük meg, hogy mi a tile és mi nem az. Az 1 bekapcsolva, a 0 pedig kikapcsolva. Az összes térképünket egy 2D-s egész számtömbben tároljuk, amelyet minden egyes funkció végén (kivéve, amikor renderelünk) visszaküldünk a felhasználónak.[3]

Az új térkép generálása előtt a térképen a meglévő tile-ok törlődnek. Ez biztosítja, hogy az új térkép generálása üres vászonnal kezdődjön, megakadályozva az új adatok átfedését vagy összeolvadását a régi tile-okkal. Ezután egy metódus segítségével legenerálunk egy N^*N méretű tömböt, amiben az értékek a 0 vagy 1 értéket vehetik fel. Miután a tömb legenerálása befejeződött, meghívjuk a procedurális mapgeneráló algoritmusunkat, majd egy renderelő függvényt, amely az általunk kiválasztott tiletípust festi a térképre. A térképgenerálás folyamatábrája a 3.1. ábrán megtekinthető.[3]



3.1. ábra. A térképgenerálás folyamatábrája

3.2. A térképgenerálás logikáját megvalósító algoritmusok bemutatása

Ebben a fejezetben a térképek generálásáért felelős metódusokat fogom részletesen bemutatni. Ezek a metódusok a procedurális térképgenerálás alapvető keretét jelentik, ahol a `GenerateArray()` metódus létrehozza a térkép adatszerkezetét és kezdeti feltételeit, a `RenderMap()` metódus pedig az adatok játszható vagy látható formába való renderelésének mechanizmusaként szolgál. Ez a kétképes folyamat - az adatok létrehozása és az adatok renderelése - központi szerepet játszik a játékom procedurális tartalomgenerálásban, lehetővé téve a térképek létrehozásának automatizálását.

3.2.1. A `GenerateArray()` metódus

A `GenerateArray()` függvénynek a célja egy 2D-s `int[,]` típusú tömb létrehozása és inicializálása, amely a térképrácsot képviseli a procedurális generáláshoz. Ez a tömb szolgál alapként a procedurális mapgeneráló algoritmusok alkalmazásához a dungeon alaprajzának generálásához.

Paraméterek:

- `int width, int height` : Ezek a paraméterek határozzák meg a térkép méreteit, specifikusan a szélességét és a magasságát.
- `bool empty` : Ez a `bool` típusú változó határozza meg, hogy hogyan inicializáljuk a tömböt. Ha az értéke igaz, akkor üresként inicializáljuk, ha `false`, akkor pedig teliként.

Az `int[,] map = new int[width,height]` sorral egy új 2 dimenziós tömböt inicializálunk. A tömb minden eleme egy egész számot reprezentálhat.

A beágyazott `for` ciklusok a tömb minden egyes elemén végigmennek, az `x` a szélességen, az `y` pedig a magasságon iterál.

A `map.GetUpperBound(0)` és a `map.GetUpperBound(1)` a tömb dimenzióinak felső határainak megadására szolgál. A `GetUpperBound(0)` az első dimenzió (szélesség) maximális indexét adja vissza, a `GetUpperBound(1)` pedig a második dimenzió (magasság) maximális indexét.

A `GetUpperBound` használata általában a tömb megadott dimenziójának utolsó érvényes indexét adja vissza. Egy szélesség és magasság dimenziójú tömb esetében a

`GetUpperBound(0)` valójában a `width - 1`, a `GetUpperBound(1)` pedig a `height - 1` értéket adná vissza, mivel a tömbindexek 0-nál kezdődnek.

A belső cikluson belül az üres paraméter feltételes ellenőrzése dönti el, hogy a tömb helyét 0-val (ami üres vagy nyitott helyet jelez) vagy 1-gyel (ami kitöltött vagy blokkolt helyet jelez) töltse-e ki.

Miután a tömb teljesen feltöltődött, visszakerül a hívó számára. Ez a tömb most a kezdeti rácsállapotként szolgál a további feldolgozáshoz. Az alább látható metódus a [3] oldal alapján készült.

```
public static int[,] GenerateArray( int width ,
    int height , bool empty )
{
    int[,] map = new int[width , height ];
    for ( int x = 0; x < map .
        GetUpperBound( 0 ); x++)
    {
        for ( int y = 0; y < map .
            GetUpperBound( 1 ); y++)
        {
            if ( empty )
            {
                map[ x , y ] = 0 ;
            }
            else
            {
                map[ x , y ] = 1 ;
            }
        }
    }
    return map;
}
```

3.2.2. A RenderMap() metódus

A `RenderMap()` függvény felelős a generált térkép vizuális megjelenítéséért egy Unity Tilemap-en egy adott TileBase segítségével. Ez a függvény a numerikus térképadatokat (egy 2D-s egész szám tömbben tárolva) a játéktérképen lévő tényleges tile-okká alakítja át.

Paraméterek:

- `int[,] map`: Egy két dimenziós tömb, amely a dungeon alaprajzát reprezentálja, ahol az egyes cellák értéke határozza meg, hogy üres vagy tele van-e az adott cella.
- `Tilemap tilemap`: Ez a Unity Tilemap, amelyre a tile-ok fognak felrajzolódni.
- `TileBase tile`: Ez a térkép kitöltött területeinek vizuális ábrázolására szolgáló tile. (Esetben RuleTile)

A `tilemap.ClearAllTiles()` parancccsal, mielőtt az új térképet lerenderelnénk, töröljük az összes meglévő tile-t a térképről. Ez kulcsfontosságú annak biztosítása érdekében, hogy a korábbi tile-ok ne maradjanak láthatóak. Ez a metódus is egymásba ágyazott `for` ciklusokat használ ahhoz, hogy bejárja a map tömb minden elemét.

A cikluson belül a függvény minden egyes koordinátánál (x, y) ellenőrzi az értéket. Az `if(map[x,y] == 1)` feltétel ellenőrzi, hogy az aktuális koordinátáknál lévő cella tele van-e (1 az érték). Ha tele van, akkor a Tilemap megfelelő pozíójára egy új tile kerül. A `tilemap.SetTile(new Vector3Int(x,y,0),tile)` parancs egy tile-t helyez el a Tilemap (x,y) pozíójában. Az alább látható metódus a [3] oldal alapján készült.

```
public static void RenderMap( int[ , ] map ,
    Tilemap tilemap , TileBase tile )
{
    //We clear the map to make
    //sure we don't overlap
    tilemap . ClearAllTiles();

    for ( int x = 0; x <
        map . GetUpperBound ( 0 ); x++)
    {
        for ( int y = 0; y <
            map . GetUpperBound ( 1 ); y++)
        {
            //1 = tile , 0 = no tile
            if ( map [ x , y ] == 1 )
            {
                tilemap . SetTile ( new Vector3Int
                    ( x , y , 0 ) , tile );
            }
        }
    }
}
```

3.3. A Perlin zaj

A Perlin-zaj egy algoritmus, amelyet Ken Perlin hozott létre az 1980-as évek elején, és széles körben használják a játékfejlesztésben bármilyen hullámszerű anyag vagy textúra létrehozásához. Például a Perlin-zajt használhatjuk procedurális domborzati alakzatok (Minecraft szerű domborzati térkép hozható létre a Perlin-zaj algoritmus segítségével), tűzeffektek, víz és felhők létrehozásához. Ezek a hatások főleg a második és harmadik dimenzióban tükrözik a Perlin-zajt, de kiterjeszhető a negyedik dimenzióra is. Ezen kívül az algoritmus használható még az 1 dimenziós térben is, mint például egy „sidescroller” terep létrehozásához, vagy kézzel írt vonalak illúziójának megteremtésére.[29]

Sőt mi több, ha az algoritmust a 2. vagy a 3. dimenzióra is kiterjesztjük, valamint az extra dimenziókra úgy tekintünk, mint az időre, akkor meg is tudjuk a kreált alakzatokat animálni. Az a 3.1. táblázatban néhány képet láthatunk a különböző méretű zajokról és néhány felhasználási módjukról futás közben:[7]

3.1. táblázat. A Perlin zaj felhasználási módjai

Zaj dimenziószáma	A nyers zaj (szürkeárnyalatos)	Felhasználási mód
1		
2		
3		

A táblázatban lévő képeket a <https://adrianb.io/2014/08/09/perlinnoise.html> oldalról használtam fel.

A Perlin-zaj gradiens zajgenerálási technikát alkalmaz, ami a pontok közötti termésszetesebb és simább átmenetet eredményez. Ez a megközelítés élethűbbnek tűnő tájképet hoz létre. Az algoritmus egy rácshálós keretrendszerben működik, ahol a rácsháló minden egyes metszéspontjához egy gradiensvektor tartozik. Ezek a vektorok döntő fontosságúak a zaj mintázatának és irányítottságának kialakításában.[29]

A Perlin-zaj egyik fő jellemzője a rácspontok közötti interpoláció alkalmazása, ami hozzájárul a jellegzetes simasághoz. Ez a sima átmenet éles ellentétben áll a teljesen véletlenszerű zajgenerálásra jellemző hirtelen változásokkal. A Perlin-zajt eredetileg 3D-s grafikához fejlesztették ki, de a 2D-s alkalmazásokban is széles körben használják, többek között a videojátékok terepgenerálásában és a procedurális textúrák létrehozásában.[29]

A generált minták összetettségének fokozása érdekében az algoritmus gyakran alkalmaz rétegezési technikát, amely több "oktavnyi" zajt tartalmaz. minden egyes oktav külön frekvenciával és amplitúdóval működik, és amikor ezeket a rétegeket kombinálják, bonyolultabb és változatosabb mintákat hoznak létre. Az algoritmus állítható paramétereit kínál, mint például a frekvencia, az amplitúdó és a perzisztencia, ami lehetővé teszi a generált zaj megjelenésének részletes szabályozását, és a terep vagy a textúra testre szabott szimulációját.[29]

A játékokban és a számítógépes grafikában való alkalmazásán túl a Perlin-zaj elterjedt más területeken is, mint például tudományos szimulációk készítése, ahol olyan ter-

mészeti jelenségeket modellez, mint a felhőképződmények, vagy egy táj jellegzetességei.[29]■

3.3.1. A Perlin-zaj algoritmus implementálása és vizsgálata

A `PerlinNoiseDungeon()` metódus a Perlin-zaj segítségével módosítja a rácsos térképet, hogy simább átmeneteket hozzon létre a kitöltött és üres terek között.

Paraméterek:

- `int[,] map`: A módosítandó kezdeti térképtömb.
- `float modifier`: Egy skálázási tényező, amely a Perlin zajfüggvény frekvenciáját állítja be. Az értéke 0 és 1 között van.

A metódusban a Perlin-zajt a térképrács minden egyes koordinátájára úgy alkalmazzuk, hogy a koordinátákat megszorozzuk a modifier változóval. Ez a modifier változó befolyásolja a zaj ""zoom"" szintjét - a magasabb értékek kisebb térrészekben belüli gyakoribb változást eredményeznek, ami töredezettebb mintázatot hoz létre, míg az alacsonyabb értékek szélesebb, simább átmeneteket eredményeznek.

Az egyes cellák zajértékének kiszámítása után a zajáértéket egész számra kerekítjük (0 vagy 1), hogy eldöntsük, hogy a cella fal vagy üres tér legyen.

A metódus a térkép határait kifejezetten falaknak állítja be a meghatározott élek fenntartása érdekében.

A függvény visszaadja a módosított térképtömböt az új terepjellemzőkkel együtt, készen állva a renderelésre. Az alább látható metódus a [3] oldal alapján készült.

```
public static int[,] PerlinNoiseDungeon(
    int[,] map, float modifier)
{
    int newPoint;
    for (int x = 0; x <
        map.GetUpperBound(0); x++)
    {
        for (int y = 0; y <
            map.GetUpperBound(1); y++)
        {

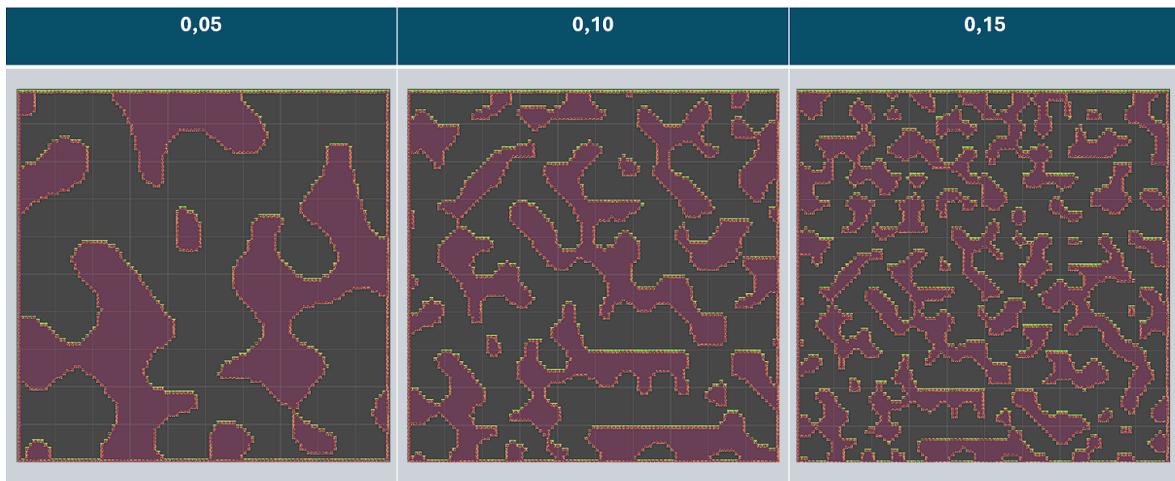
            if ((x == 0 || y == 0 ||
                x == map.GetUpperBound(0) - 1 ||
                y == map.GetUpperBound(1) - 1))
            {
                //Keep the edges as walls
                map[x, y] = 1;
            }
            else
            {
                //Generate a new point using perlin noise,
                //then round it to a value of either 0 or 1
                newPoint = Mathf.RoundToInt(
                    Mathf.PerlinNoise(x * modifier,
                    y * modifier));
            }
        }
    }
}
```

```

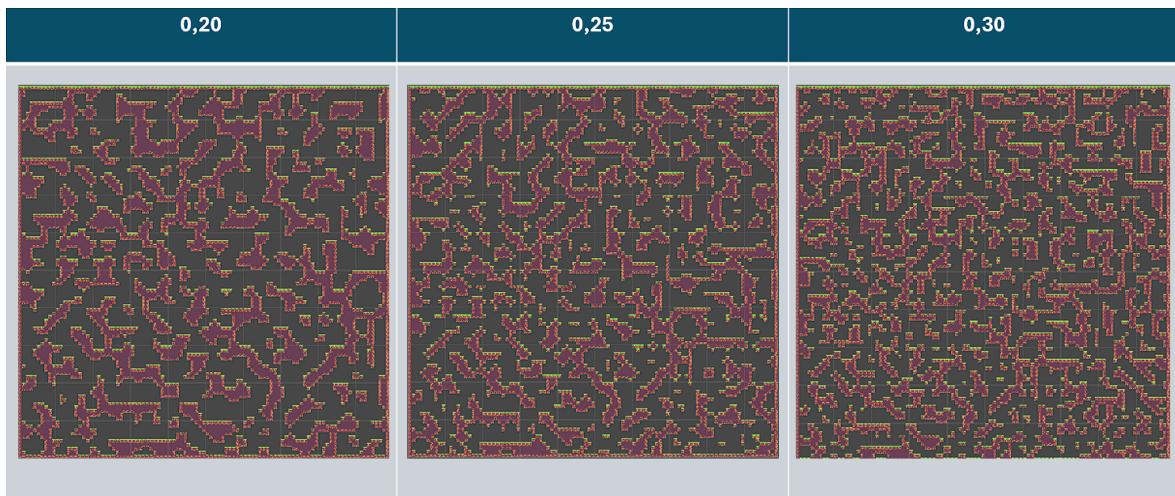
        map[x , y] = newPoint ;
    }
}
return map;
}
    
```

A továbbiakban a Perlin-zaj által generált térképek vizsgálatáról lesz szó.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet 1-esekkel töltünk fel. A modifier változó értékét fogom növelni a vizsgálat során, az első érték a 0,05, a második a 0,10 és ez egészen a 0,30 értékig fog növekedni. A generált térképek a 3.2. ábrán és a 3.3. ábrán láthatóak.



3.2. ábra. A PerlinNoiseDungeon() metódussal generált térképek 0.05, 0.10 és 0.15 modifier értékekkel



3.3. ábra. A PerlinNoiseDungeon() metódussal generált térképek 0.20, 0.25 és 0.30 modifier értékekkel

Ahogy a generált térképeken is láthatjuk, az alacsonyabb modifier értékek általában nagyobb, összefüggőbb régiókat eredményeznek, amelyek szélesebb földtani jellemzőket,

például széles barlangokat vagy nagy nyílt területeket szimulálnak. A magasabb módosító értékek széttöredezettebb és kuszább tereket hoznak létre, amelyek bonyolult alagútrendszeret vagy sűrű barlanghálózatot eredményeznek.

Ezek a különbségek úgy befolyásolhatják a játékmenetet, hogy a nagyobb, nyílt területek megkönyíthetik a navigációt, könnyedebben teljesíthetők lesznek a pályák, míg a bonyolultabb, magasabb modifier értékkel generált pályák nagyobb kihívást jelentők lesznek, hosszabb játékidőt és frusztrációt eredményezhetnek.

Ezek a legenerált pályák nem feleltek meg az általam elvárt kinézetnek, mivel tartalmaznak olyan elzárt részeket, amelybe a játékos nem tud bemenni. Olyan játékoknál alkalmaznám ezt a fajta pályagenerálást, amelyknél implementálva van az, hogy a játékos eltüntetheti, széttörheti a blokkot, mondjuk egy csákánnyal, vagy bombával.

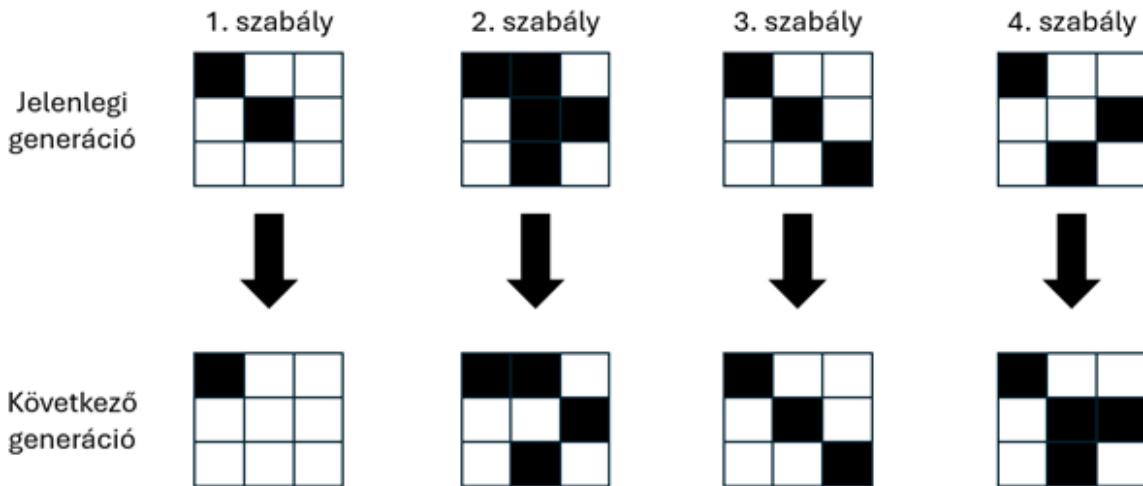
3.4. A celluláris automata

A celluláris automata, egy rácsalapú rendszerben működő számítási modell, amely egyszerűségében és összetettségében egyaránt lenyűgöző. minden egyes sejt ezen a rács-hálózaton két állapotban létezhet, amelyek az "él" vagy a "halott" állapotok. E sejtek fejlődését egyik generációról a másikra egy szabályrendszer határozza meg, amely jellemzően a szomszédos sejtek állapotán alapul. Ez a felállás, bár összetevőit és szabályait tekintve egyszerű, az azonos szabályokat követő sejtek együttes kölcsönhatása révén rendkívül bonyolult mintázatokat képes létrehozni.[26]

A celluláris automaták egyik legismertebb példája Conway „Game of Life” című műve. Ez egy kiváló példa arra, hogy az alapvető szabályok hogyan eredményezhetnek összetett viselkedést, annak ellenére, hogy ez egy „zero-player” játék, ami azt jelenti, hogy a fejlődését a kezdeti állapota határozza meg, és nincs szüksége emberi játékosztól származó cselekedetre, inputra. Az ember úgy lép kapcsolatba a játékkal, hogy létrehoz egy kezdeti konfigurációt, és megfigyeli, hogy hogyan fejlődik. A celluláris automata, valamint a „Game of Life” játék négy szabálya a következő:[27]

1. minden olyan élő sejt, amelynek kettőnél kevesebb szomszédja van, „meghal” (ezt nevezik alulnépesedésnek vagy veszélyeztetettségnek).[27]
2. minden olyan élő sejt, amelynek háromnál több szomszédja van, „meghal” (ezt nevezik túlnépesedésnek vagy túlzsföltságnak).[27]
3. minden élő sejt, amelynek két vagy három élő szomszédja van, változatlanul tovább él a következő generációig.[27]
4. minden halott sejt, amelynek pontosan három élő szomszédja van, életre kel.[27]

A kezdeti minta képezi a rendszer "magját". Az első generáció úgy jön létre, hogy a fenti szabályokat egyszerre alkalmazzák a mag minden sejtjére - a születések és halálozások egyszerre történnek, és azt a diszkrét pillanatot, amikor ez megtörténik, néha ticknek nevezik. (Más szóval, minden egyes generáció az előző generáció szintiszta függvénye.) A szabályok ismételt alkalmazása további generációk létrehozásához folytatódik. [27] A celluláris automata szabályait a 3.4. ábrán láthatjuk.



3.4. ábra. A celluláris automata szabályai szemléltetve

A celluláris automata rugalmassága a testreszabhatóságban rejlik. A fejlesztők a szabályokat és az állapotokat az egyedi igényekhez igazíthatják, befolyásolva olyan szempontokat, mint a térkép sűrűsége és az útvonalak összekapcsolhatósága. A véletlenszerűség beépítésének képessége ellenére az automata determinisztikus jellege biztosítja az azonos kezdeti feltételekből származó konzisztens eredményeket, ami különösen hasznos a reprodukálható szintek létrehozásához.[26]

A celluláris automata nem csak a szintek strukturálásában segít, hanem a vizuális látványt is fokozza, olyan mintákat generálva, amelyek esztétikailag szépek és a játékmenet szempontjából is praktikusak. Ezzel a tulajdonságával hatékony eszközzé válik a fejlesztők számára, akik dinamikus és megnyerő környezetet kívának létrehozni a 2D platformer játékokban, és az egyedi, változatos szintek létrehozásával jelentősen növeli a játék újra játszásának az esélyét.[26]

3.4.1. A celluláris automata algoritmus implementálása és vizsgálata

Ahhoz, hogy a VonNeumannCellularAutomata() metódus létrehozza a randomizált térképünket, először egy módosított térképet kell kreálnunk. A módosítás pedig nem más, mint a randomizáció.

A ModifiedMapForCellularAutomata() metódus megteremti egy celluláris automata folyamat előfeltételeit egy véletlenszerű kezdeti feltételeket tartalmazó térkép létrehozásával, amely aztán a következő lépésekben a von Neumann celluláris automata szabályaival lesz fejlesztve. Funkcionalitását tekintve ugyan azt csinálja, mint a GenerateArray() metódus, de miközben inicializálja a térképtömböt, nem egyenletesen tölti ki a térképteret, hanem randomizálva. minden belső cellához véletlenszerű állapotot rendel (0 vagy 1) a fillPercent által meghatározott valószínűség alapján. Az alább látható metódus a [3] oldal alapján készült.

```
public static int[,] ModifiedMapForCellularAutomata(
    int width, int height, float seed, int fillPercent)
{
    //Seed our random number generator
```

```

System.Random rand = new System.Random(
    seed.GetHashCode());

int[,] map = new int[width, height];

for (int x = 0; x <
    map.GetUpperBound(0); x++)
{
    for (int y = 0; y <
        map.GetUpperBound(1); y++)
    {
        if ((x == 0 || x == map.GetUpperBound(0) - 1 ||
            y == 0 || y == map.GetUpperBound(1) - 1))
        {
            //Ensure the walls on the sides of the map
            map[x, y] = 1;
        }
        else
        {
            //Randomly generate the grid
            map[x, y] = (rand.Next(0, 100) <
                fillPercent) ? 1 : 0;
        }
    }
}
return map;
}

```

A VonNeumannCellularAutomata() metódus a von Neumann szomszédságon alapuló celluláris automata technikát használja a generált térkép fejlesztésére.

Paraméterek

- **int[,] map**: Ez az a térkép, amelyet a függvény módosítani fog.
- **int smoothCount**: Az iterációk (vagy generációk) száma, amelyeken a celluláris automata folyamat keresztülmegy.

A térkép többszörös átmeneteken megy keresztül a cellák állapotának fejlesztése érdekében. A több ismétlés jellemzően simább és összefüggőbb struktúrákat eredményez.

Csak a közvetlen szomszédokat (felfelé, lefelé, balra, jobbra) veszi figyelembe az egyes cellák evolúciója, ami leegyszerűsíti az interakciókat, és inkább az utak vagy nyitott területek létrehozására összpontosít, mint a komplex mintázatokra.

Cellaállapot-frissítési szabályok

- **Kitöltött cella**: Ha egy cellának több mint 2 környező kitöltött cellája van, akkor maga is kitöltött cellává válik.
- **Üres cella**: Ha egy cellának 2-nél kevesebb kitöltött cellája van, akkor üres lesz.
- **Nincs változás**: Ha egy cellának pontosan 2 szomszédja van, az állapota változatlan marad. Ez a szabály stabilitást biztosít a struktúrának, fenntartva a jelenlegi komplexitást.

A megadott számú iteráció elvégzése után a függvény visszaadja a továbbfejlesztett mapot, amely tükrözi a celluláris automata által végrehajtott változásokat. Az alább látható metódus a [3] oldal alapján készült.

```

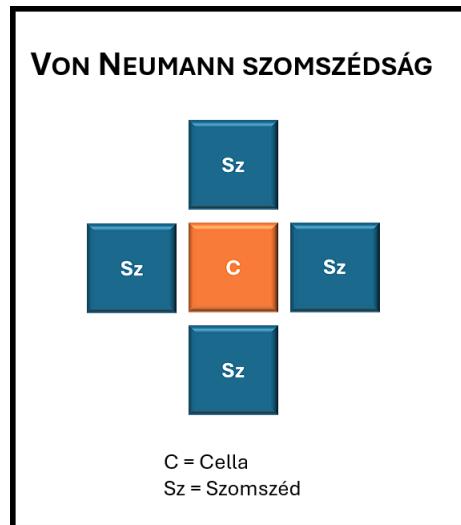
public static int[,] VonNeumannCellularAutomata(
    int[,] map, int smoothCount)
{
    for (int i = 0; i < smoothCount; i++)
    {
        for (int x = 0; x <
            map.GetUpperBound(0); x++)
        {
            for (int y = 0; y <
                map.GetUpperBound(1); y++)
            {
                //Get the surrounding tiles
                int surroundingTiles =
                    GetVNSurroundingTiles(map, x, y);

                if ((x == 0 ||
                    x == map.GetUpperBound(0) - 1 ||
                    y == 0 || y == map.GetUpperBound(1)))
                {
                    //Keep our edges as walls
                    map[x, y] = 1;
                }
                //von Neuemann Neighbourhood requires
                //only 3 or more surrounding tiles
                //to be changed to a tile
                else if (surroundingTiles > 2)
                {
                    map[x, y] = 1;
                }
                //If we have less than 2 neighbours,
                //set the tile to be inactive
                else if (surroundingTiles < 2)
                {
                    map[x, y] = 0;
                }
                //Do nothing if we have 2 neighbours
            }
        }
    }
    return map;
}

```

A fenti metódusban használt `GetVNSurroundingTiles()` metódus [3] célja a környező "kitöltött" cellák számának kiszámítása a von Neumann szomszédság alapján, amely csak a négy szomszédos cellát veszi figyelembe a fő irányokban: felette, alatta,

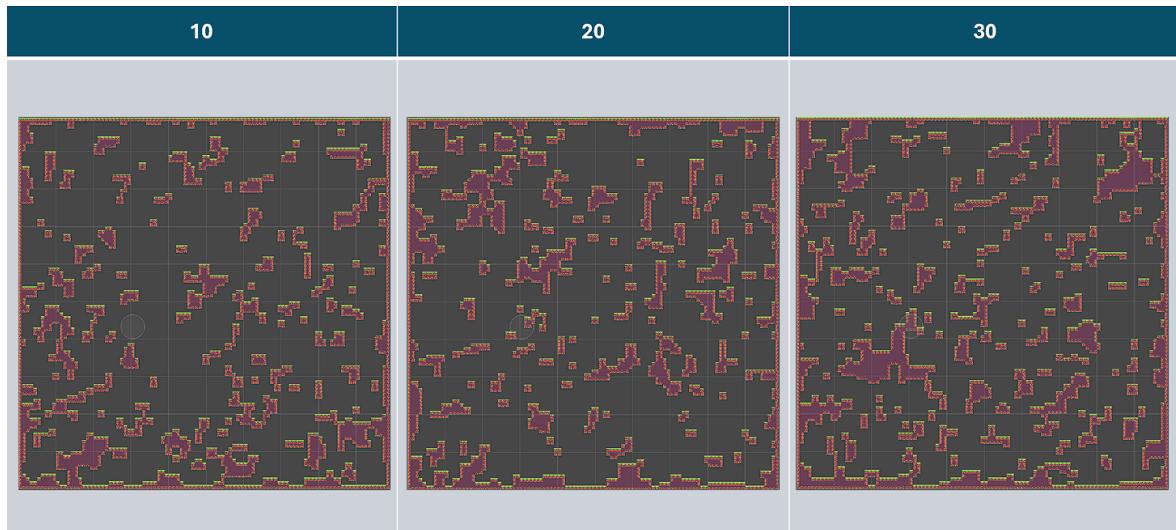
balra és jobbra. A von Neumann szomszédságot reprezentáló ábra a 3.5. ábrán láthatató.



3.5. ábra. A von Neumann szomszédság

A továbbiakban a celluláris automata által generált térképek vizsgálatáról lesz szó.

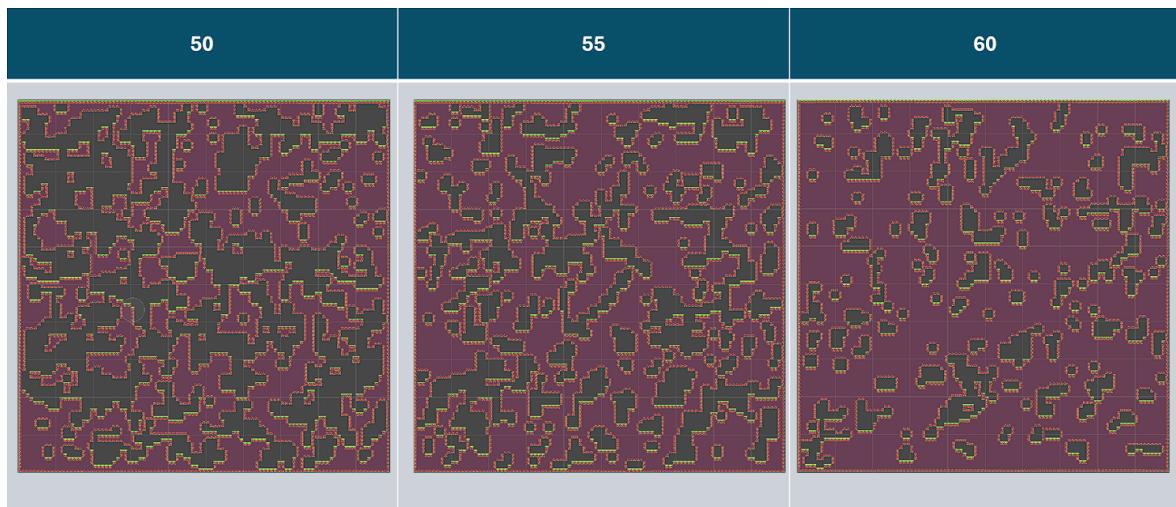
Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet véletlenszerűen töltünk fel 1-esekkel és 0-ákkal. Először a **requiredFillPercent** változó fixálásával és a **smoothIterations** változó növelésével fogom vizsgálni a legenerált térképeket, majd a **smoothIterations** változó értékét fixálom és a **requiredFillPercent** értékét fogom növelni. A generált térképek a 3.6. ábrán láthatóak.



3.6. ábra. Fixált requiredFloorPercent értékkel és változó smoothIterations értékkel generált térképek

A fentebb látható legenerált térképeknél a **requiredFloorPercent** értéke fixen 40, a **smoothIterations** értéke pedig 10,20 és 30. Mint látni is lehet, ha növeljük a **smoothIterations** értékét, akkor a térképen a teli cellák száma növekszik, bár annyira nem látványos a különbség a legenerált térképek között.

A fixált **smoothIterations** értékkel és a változó **requiredFloorPercent** értékkel generált térképek a 3.7. ábrán láthatóak.



3.7. ábra. Fixált smoothIterations értékkel és változó requiredFloorPercent értékkel generált térképek

A fentebb lévő ábrán látható térképeknél a `smoothIterations` értéke fixen 20, a `requiredFloorPercent` értéke pedig 50,55 és 60. Itt már sokkal nagyobb a különbség a legenerált térképek között, minél nagyobb a `requiredFloorPercent` értéke, annál több olyan cella lesz, amelynek az értéke 1, azaz sokkal sűrűbb lesz a térképünk.

A gond ezzel a procedurális mapgeneráló algoritmussal is az, hogy nem lehet garantálni, hogy olyan térképeket generálunk, amelyben nincsenek olyan részek, amelyek teljesen elzártak. Ezt az algoritmust inkább 3D-s, Minecraft szerű játékokban tudnám elképzelni, ahol mondjuk az értékes tárgyak elhelyezkedését lehetne meghatározni vele a föld alatt.

3.5. A Random Walk algoritmus (Véletlen bolyongás)

A véletlen bolyongás algoritmus egy viszonylag egyszerű, de hatékony módszer a procedurális térképgenerálásra, különösen alkalmas kétdimenziós rácsalapú térképekhez. Közismert arról, hogy természetesnek tűnő alakzatokat hoz létre, és összetettebb procedurális generáló rendszerek első lépéseként szolgálhat. Kétdimenziós rácshálózattal összefüggésben a véletlen bolyongást néha „részeges sétának” (Drunkard’s walk-nak) is nevezik, és az elnevezés magától értetődő, ha figyelembe vesszük a működését:[14]

1. Hozzunk létre egy NxM méretű rácshálózatot.[14]
2. Válasszunk egy véletlenszerű kezdő pozíciót a rácshálón.[14]
3. Állítsuk be a pozíciót „visited”-re. (Azaz látogatottra.)[14]
4. Válasszunk egy új véletlenszerű pozíciót az aktuális pozíciótól egyetlen cella elmozgatásával (balra / fel / jobbra / le).[14]
5. Ha a pozíció amire érkezünk érvényes (a pozíció nem esik a rácshálón kívülre), akkor ezt az új pozíciót állítsuk be az aktuális pozíciónak.[14]
6. Menjünk vissza a 4. ponthoz, és addig ismételjük, amíg a befejezési feltétel teljesül (például az ismétlések száma).[14]

A modellezés alapvetően egy olyan egyed, amely minden egyes időlépésnél kiszámíthatatlanul mozog bármilyen irányba. Az entitás a korábban meglátogatott cellákba is visszamehet, így a korábbi iterációk nem befolyásolják az aktuális iterációkat, ami a véletlen bolyongást sztochasztikus / memória nélküli folyamattá teszi. Sőt mi több, garantálja, hogy a térkép teljesen összefüggő lesz, mivel csak a szomszédos cellák között mozog. Ez az algoritmus ideális a játékok barlangjainak és túlvilágainak létrehozására, mivel képes összefüggő és terjedelmes térképeket létrehozni.[14] Egy véletlen bolyongás algoritmussal létrehozott összefüggő térkép a 3.8. ábrán megtekinthető.



3.8. ábra. Egy véletlen bolyongás algoritmussal létrehozott összefüggő térkép[14]

3.5.1. A véletlen bolyongás algoritmus implementálása és vizsgálata

A RandomWalkCave() metódus a térképrács módosítására szolgál a véletlen bolyongás algoritmuson alapuló procedurális generálási technikával. Ez a függvény egy barlang-szerű struktúrát váj ki egy adott kétdimenziós tömbben.

A funkció véletlenszerűen választ ki egy kezdő pozíciót (`floorX`, `floorY`) a térkép határain belül, de nem a széleken, hogy biztosítsa, hogy legyen hely a barlang bővítésére. Innen fog kezdődni a véletlen bolyongás. A megadott tartományon belül minden lehetséges kiindulási pozíciónak egyenlő valószínűsége van, hogy kiválasztásra kerüljön, tehát egyenletes eloszlás alapján kerül kiválasztásra a kezdőpozíció.

A `reqFloorAmount` a térképen lévő összes cellaszám százalékaként kerül kiszámításra. Ez a változó határozza meg, hogy hány cellát kell átalakítani teliből üressé a barlanggenerálás befejezéséhez.

A `floorCount` nulláról indul, és minden alkalommal növekszik, amikor egy cellát teliből üressé alakítunk.

A while ciklus addig megy, amíg a `floorCount` értéke el nem éri a `reqFloorAmount` értékét. A while cikluson belül egy switch utasítást alkalmazok az irányok kezelésére. Az út generálásának a logikája a következő:

- A switch utasítás minden esete egy iránynak felel meg.
- Mielőtt az algoritmus bármilyen irányba mozogna, biztosítjuk, hogy a lépés nem halad ki a térkép határain kívülre.
- 0.eset: Növeljük a `floorY` értékét, ha a felfelé mozgás a határon belül marad.

- 1.eset: Csökkentjük a floorY értékét, ha a lefelé mozgás a határokon belül marad.
- 2.eset: Növeljük a floorX értékét, ha a jobbra mozgás a határokon belül marad.
- 3.eset: Csökkentjük a floorX értékét, ha a balra mozgás a határokon belül marad.

Minden esetben megnézzük, hogy a jelenlegi pozíció teli-e. Ha az, akkor átkonvertáljuk üresre, és a procedúra addig folytatódik, amíg el nem érjük a kívánt cellák számát. Az alább látható metódus a [3] oldal alapján készült.

```

public static int[,] RandomWalkCave(
    int[,] map, float seed, int requiredFloorPercent)
{
    //Seed our random
    System.Random rand = new System.Random(
        seed.GetHashCode());

    //Define our start x position
    int floorX = rand.Next(
        1, map.GetUpperBound(0) - 1);
    //Define our start y position
    int floorY = rand.Next(
        1, map.GetUpperBound(1) - 1);
    //Determine our required floorAmount
    int reqFloorAmount =
        (map.GetUpperBound(1) * map.GetUpperBound(0))
        * requiredFloorPercent) / 100;

    //Used for our while loop,
    //when this reaches our reqFloorAmount
    //we will stop tunneling
    int floorCount = 0;

    //Calculating stepcount and runtime
    int stepCount = 0;
    Stopwatch stopwatch = new Stopwatch();

    //Set our start position
    //to not be a tile (0 = no tile, 1 = tile)
    map[floorX, floorY] = 0;
    //Increase our floor count
    floorCount++;

    stopwatch.Start();
    while (floorCount < reqFloorAmount)
    {
        //Determine our next direction
        int randDir = rand.Next(4);

        stepCount++;
    }
}

```

```

switch (randDir)
{
    case 0: //Up
        //Ensure that the edges
        //are still tiles
        if ((floorY + 1) <
            map.GetUpperBound(1) - 1)
    {
        //Move the y up one
        floorY++;
        if (map[ floorX , floorY ] == 1)
        {
            //Change it to not a tile
            map[ floorX , floorY ] = 0;
            //Increase floor count
            floorCount++;
        }
    }
    break;
    case 1: //Down
        //Ensure that the edges
        //are still tiles
        if ((floorY - 1) > 1)
    {
        //Move the y down one
        floorY--;
        if (map[ floorX , floorY ] == 1)
        {
            //Change it to not a tile
            map[ floorX , floorY ] = 0;
            //Increase the floor count
            floorCount++;
        }
    }
    break;
    case 2: //Right
        //Ensure that the edges are still tiles
        if ((floorX + 1) <
            map.GetUpperBound(0) - 1)
    {
        //Move the x to the right
        floorX++;
        if (map[ floorX , floorY ] == 1)
        {
            //Change it to not a tile
            map[ floorX , floorY ] = 0;
            //Increase the floor count
        }
    }
}

```

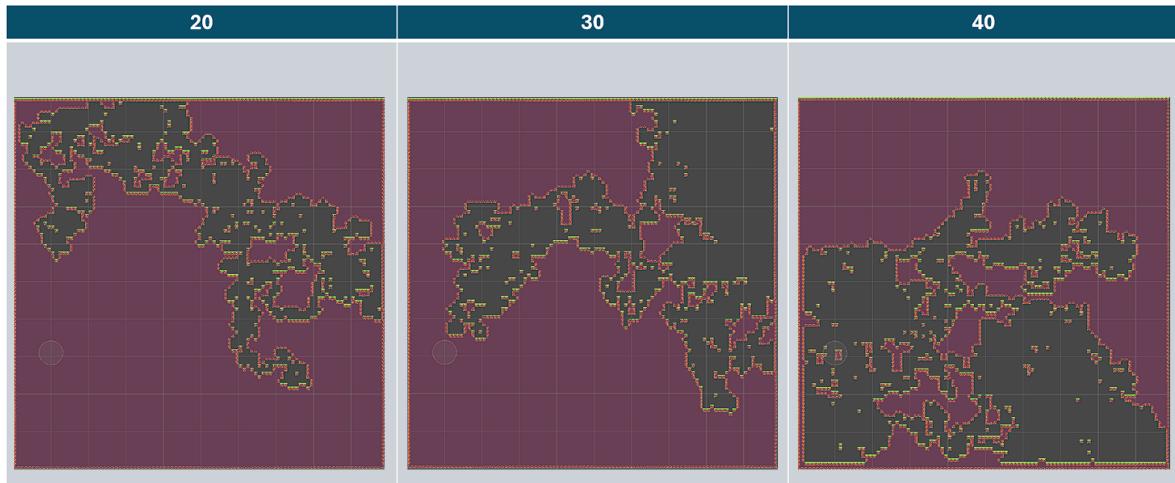
```

                floorCount++;
            }
        }
        break;
    case 3: //Left
        //Ensure that the edges are still tiles
        if ((floorX - 1) > 1)
        {
            //Move the x to the left
            floorX--;
            if (map[ floorX , floorY ] == 1)
            {
                //Change it to not a tile
                map[ floorX , floorY ] = 0;
                //Increase the floor count
                floorCount++;
            }
        }
        break;
    }
    stopwatch.Stop();
    TimeSpan ts = stopwatch.Elapsed;
    UnityEngine.Debug.Log(
        $"The runtime of the RandomWalkCave() method is " +
        $"{ts.TotalMilliseconds}");
    UnityEngine.Debug.Log(
        $"The stepcount of the RandomWalkCave() method is " +
        $"{stepCount}");
    //Return the updated map
    return map;
}

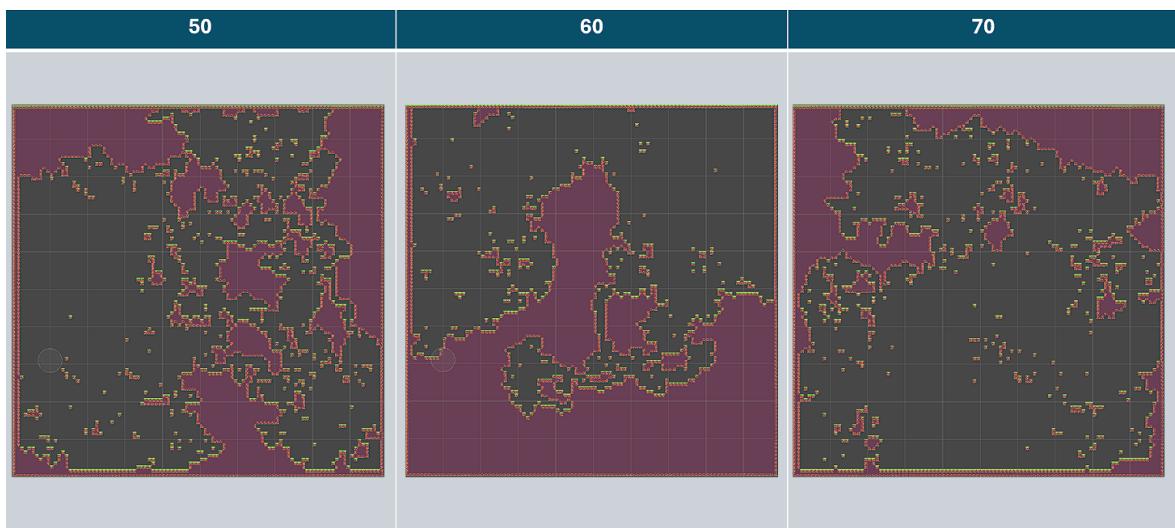
```

A továbbiakban a véletlen bolyongás algoritmussal generált térképek vizsgálatával fogok foglalkozni.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet 1-esekkel töltünk fel. Az első vizsgálat a **requiredFloorPercent** érték növelésével fog történni. Hat esetet fogok vizsgálni, melyeknél a változó értéke rendre 20,30,40,50,60 és 70 lesz. A legenerált térképek a megadott **requiredFloorPercent** értékkel a 3.9. ábrán és a 3.10. ábrán megtekinthetőek.



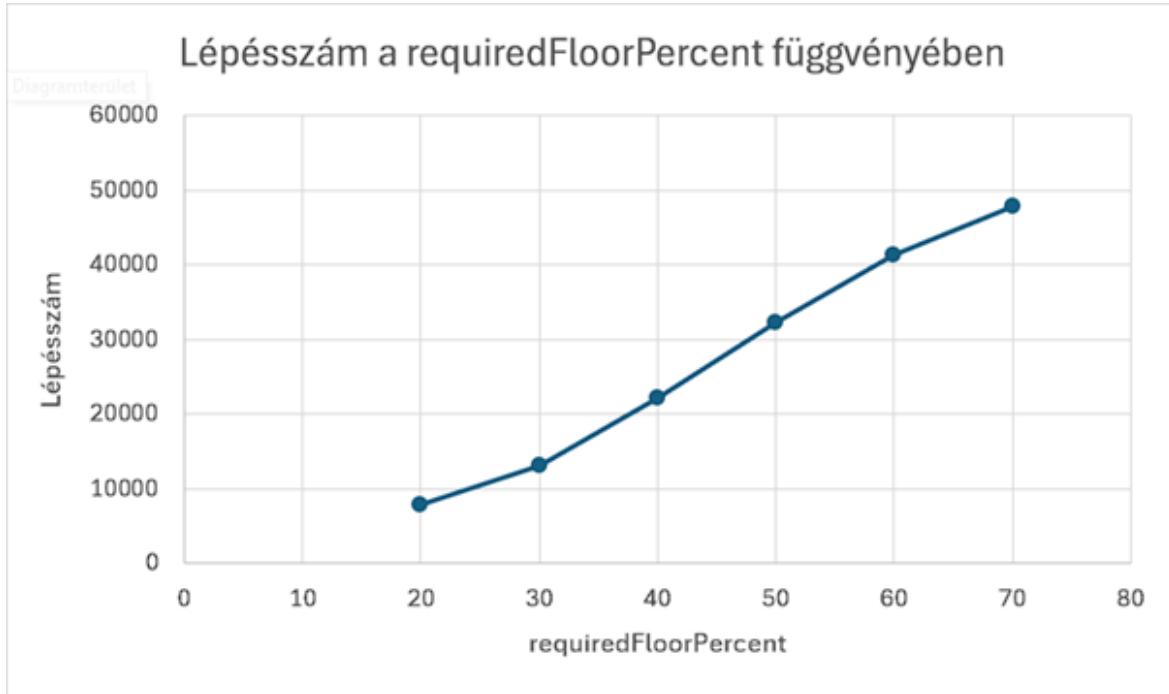
3.9. ábra. Azok a térképek, amelye 20,30 és 40 requiredFloorPercent értékkel lettek generálva



3.10. ábra. Azok a térképek, amelye 50, 60 és 70 requiredFloorPercent értékkel lettek generálva

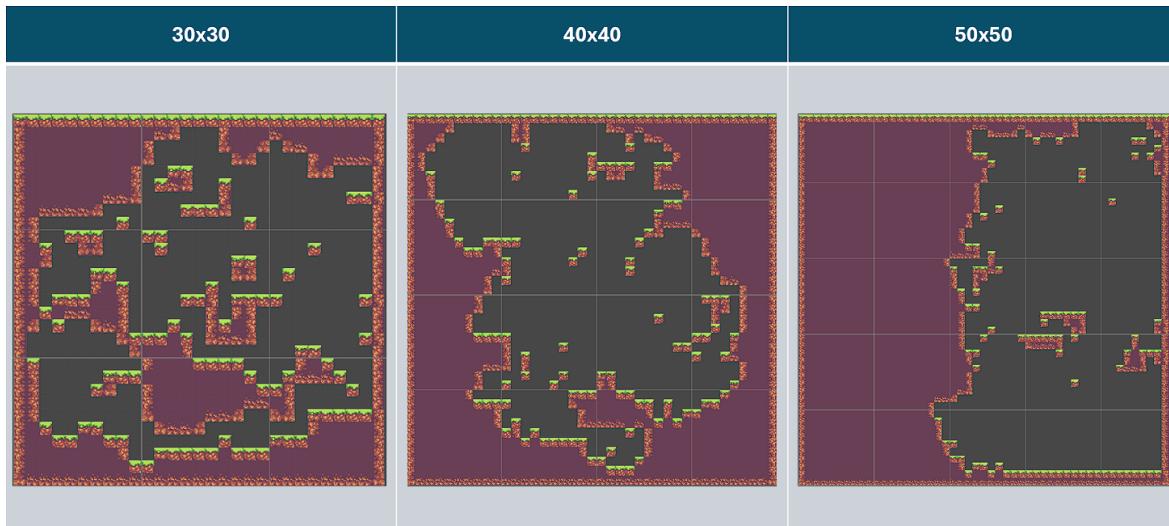
Ahogy az ábrákon is láthatjuk, minél nagyobb a `requiredFloorPercent` változó értéke, annál több alagutat váj ki magának az algoritmus, azaz annál több olyan cella keletkezik, melynek az értéke 0. Azt is észrevehetjük, hogy nem keletkezik olyan rész a térképen, amely el lenne zárva, azaz teljesen összefüggő mapot kapunk minden egyes esetben.

Minél nagyobb a `requiredFloorPercent` értéke, annál nagyobb lesz a lépésszáma az algoritmusnak, valamint a futási ideje is nőni fog. Volt néhány olyan eset, ahol nagyobb `requiredFloorPercent` értéknél kevesebb futási idő, valamint lépésszám volt, de ez elenyésző mennyiségű esetben fordult elő. A lépésszám és a `requiredFloorPercent` korrelációja a 3.11. ábrán megtekinthető.



3.11. ábra. A lépésszám és a requiredFloorPercent kapcsolata

A következő vizsgálat során változó méretű térképeket fogok generálni, viszont a `requiredFloorPercent` értékét 45-re fogom fixálni. Ezen térképek legenárálásának szintén a lépésszámát, valamint az idejét fogom vizsgálni. A térképek rendre 30x30, 40x40 és 50x50 méretűek lesznek. A legenerált térképek a 3.12. ábrán láthatóak.



3.12. ábra. Fixált requiredFloorPercent értékkal, de változó térképmérettel generált mapok

A generáláshoz szükséges futási idő, valamint a lépésszám a 3.13. ábrán látható.

Térkép méretei	30x30	40x40	50x50
Lépésszám	999	2952	6465
Futási idő (milliszekundumban)	0,0819 ms	0,1504 ms	0,264 ms

3.13. ábra. A fixált requiredFloorPercent értékkel és változó térképmérettel rendelkező mapok generálásának lépésszáma és futási ideje

További vizsgálatokat fogok az algoritmuson elvégezni, mivel jelenleg egyenletes eloszlást használ az algoritmus, ahol minden fő iránynak (felfelé, lefelé, balra és jobbra) egyenlő a valószínűsége, amely 0,25. Ez egy torzítatlan sétát eredményez, amely egyik irányt sem részesíti előnyben a többivel szemben. Az egyes irányok valószínűségeinek megváltoztatásával különböző hatásokat és mintákat hozhatunk létre a térképgenerálás során.

Az egyik ilyen vizsgálat során nagyobb valószínűséget fogok rendelni az előző lépésrel megegyező irányba történő előrehaladáshoz és kisebb valószínűséget az irányváltoztatáshoz. Ehhez módosítanom kell az egyes lépések kiválasztásának a logikáját.

Először is egy olyan metódust kell implementálni, amely egy nem egyenletes eloszlás alapján választ irányt, majd az egyszerű `rand.Next(4)` parancsot lecserélni erre a metódusra. A metódusnak a neve a `ChooseDirectionBias()`.

```
private static int ChooseDirectionBias(
    System.Random rand, int[] directions, int lastDirection)
{
    double[] probabilities =
        new double[directions.Length];

    // 70 percent chance to continue in the same direction
    double bias = 0.7;

    for (int i = 0; i < directions.Length; i++)
    {
        probabilities[i] =
            (lastDirection == directions[i]) ?
                bias : (1 - bias) / (directions.Length - 1);
    }

    double roll = rand.NextDouble();
    double cumulative = 0.0;

    for (int i = 0; i < probabilities.Length; i++)
    {
        cumulative += probabilities[i];
        if (roll < cumulative)
```

```

    {
        return directions[ i ];
    }
}

// Fallback
return directions[ directions.Length - 1 ];
}

```

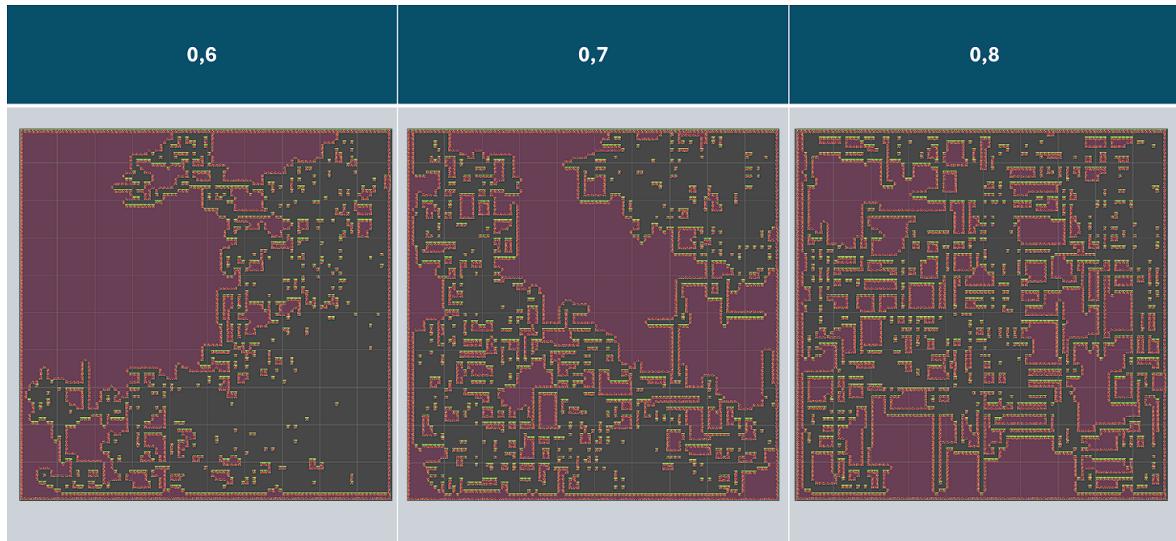
Az első lépés a valószínűségi tömb inicializálása, amely lebegőpontos számokat fog tartalmazni, és a hossza megegyezik a directions tömb hosszával. Ez a tömb tárolja az egyes irányok valószínűségét. Ezután inicializálunk egy **bias** nevű változót, amelynek a változtatásával kontrollálhatjuk, hogy mennyire legyen hajlamos az algoritmus ugyanabba az irányba haladni. Az első **for** cikluson belül számítjuk ki az úgynevezett „elfogult” valószínűséget (biased probability). Ha egy irány megegyezik a **lastDirection** irányával, akkor nagyobb valószínűséget kap (bias paraméter, jelenleg 70%-ra állítva). Az összes többi irány egyenlő arányban részesül a fennmaradó 30%-os valószínűségből.

A következő lépés a random választásnak az implementálása. A **rand.NextDouble()** parancs egy véletlenszerű lebegőpontos számot generál 0 és 1 között, amely tulajdonképpen egy „roll”-ként működik, hogy eldöntse, melyik irány lesz választva. A második **for** cikluson a probabilities tömbön iterálunk végig, és kiszámítjuk a kumulatív valószínűséget. Ezután az irányok kiválasztása következik annak ellenőrzésével, hogy a véletlenszerű „roll” az egyes irányok együttes valószínűségi tartományába esik-e. Az első olyan irány lesz kiválasztva, ahol a kumulatív valószínűség meghaladja a „roll” értékét.

A metódus **return** értékként visszaadja a kiválasztott irányt az eredeti irányok tömbjének megfelelő indexeként.

A továbbiakban a **ChooseDirectionBias()** metódus beépítésével generált térképeket fogom vizsgálni rögzített $n \times n$ méretű pályákon, majd változó méretű pályákon és ezek legenerálásának a futási idejét, valamint lépésszámát is vizsgálni fogom. A vizsgálat során a **bias** változó értékét fogom módosítani a rögzített $n \times n$ méretű pályákon. A **requiredFloorPercent** értéke rögzítve lesz, a 45-ös értéken.

Mindegyik generált térkép 100x100-as mátrixnak felel meg, amelyet 1-esekkel töltünk fel. Hárrom esetet fogok vizsgálni, ahol a **bias** változó értékét fogom módosítani. A **bias** paraméter értéke rendre 0.6, 0.7 és 0.8 lesz. A generált térképek a 3.14. ábrán láthatóak.



3.14. ábra. Változó bias paraméterekkel generált térképek

Az ábráról, amely a generált térképeket különböző **bias** értékekkel szemlélteti, mi közben a **requiredFloorPercent** és a térkép mérete állandó, több következtetést is levonhatunk a **bias** paraméter értékének változtatásának hatásáról.

A bias érték hatása:

- 0.6-os érték:** Ezzel az értékkel a pályák kevesebb egyenes vonallal rendelkeznek, egy olyasmi térképet hoz létre, amely kevesebb falat és egy nagy nyitott területet tartalmaz.
- 0.7-es érték:** A bias érték növelésével összefüggőbb mintázat kezd kialakulni, az ösvények több és egyenesebb kapcsolatokat kezdenek kialakítani a területek között.
- 0.8-as érték:** Ezen a szinten a térkép sok egyenes útvonalat mutat, több kisebb nyitott terüettel, amelyeket ezek a kiterjedtebb útvonalak kötnek össze.

A térképek generálásának lépésszáma, valamint futási ideje a 3.15. ábrán látható.

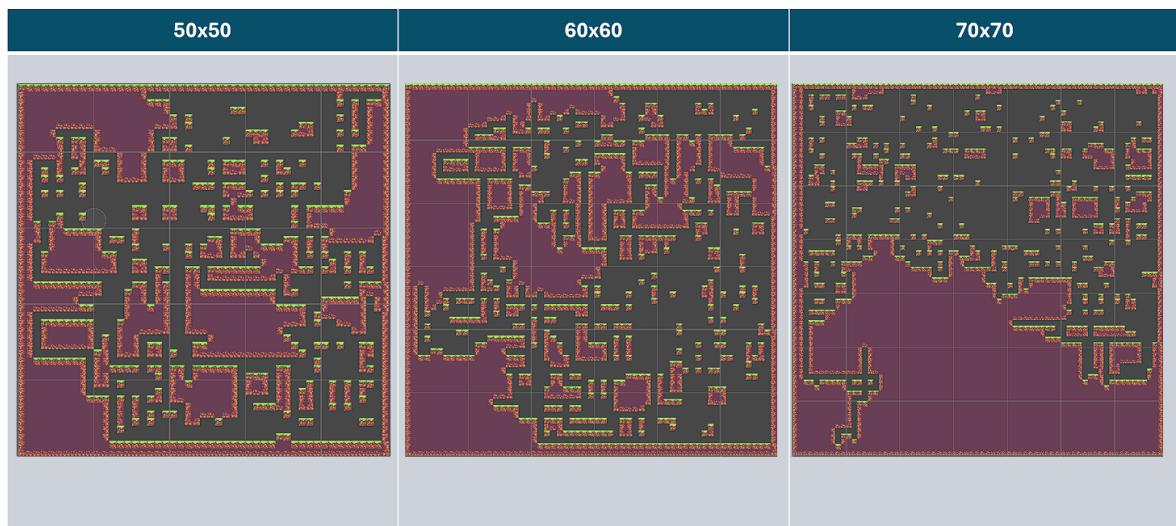
bias értékek	0,6	0,7	0,8
Lépésszám	11341	9374	9156
Futási idő (milliszekundumban)	2,2295 ms	1,9496 ms	1,8217 ms

3.15. ábra. A változó bias értékkal legenerált térképek lépésszáma és futási ideje

Mint láthatjuk, alacsonyabb **bias** érték esetén az algoritmus gyakrabban változtatja az irányt, ami több elfordulást eredményez. Ez összetettebb útvonalakat hoz létre,

amelyek általában további lépéseket igényelnek ahhoz, hogy elérjék a térkép szükséges alapszázelékát, mivel az útvonal megfordulhat önmagában, vagy kevésbé közvetlen útvonalon haladhat.

A továbbiakban a változó méretű térképek generálásának a vizsgálatával fogok foglalkozni. Szintén megvizsgálom a futási időt, valamint a lépésszámot. A térképek mérete rendre 50x50, 60x60 és 70x70 méretűek lesznek, a **bias** változó értékét, valamint a **requiredFloorPercent** értékét is állandóra állítom. A **bias** értéke 0.7, a **requiredFloorPercent** értéke pedig 45 lesz. Az ily módon generált térképek a 3.16. ábrán, a hozzájuk tartozó lépésszám, valamint futási idő a 3.17. ábrán látható.



3.16. ábra. Különböző méretű térképek legenerálása állandó bias értékkel

Térkép méretek	50x50	60x60	70x70
Lépésszám	2298	4083	5536
Futási idő (milliszekundumban)	0,6599 ms	1,0241 ms	1,4310 ms

3.17. ábra. A különböző méretű térképek generálásához szükséges lépésszám, valamint futási idő

Ahogy az várható volt, minél nagyobb a térkép területe, annál több lépésszám szükséges a kívánt requiredFloorPercent értékének eléréséhez, valamint a futás idő is több lesz.

Úgy vélem, ez az algoritmus felelt meg leginkább az általam felállított követelményeknek, hiszen mindenkor minden olyan pályákat generál, amely összefüggő, a térkép szélei mindenkor falak, így a játékos szabadon navigálhat anélkül, hogy törődnie kellene a leeséssel, nincsenek olyan részek, ahová a játékos ne tudna eljutni, hiszen nincsenek elzárt terek. Ahhoz viszont, hogy a játékos eljusson mindenhol, egy bizonyos játékmechanikát fogok bevezetni, ami nem más mint, a grappling hook.

4. fejezet

4 2D platformer játék tervezése és fejlesztése a Unity játékmotor segítségével

4.1. A játék leírása

A szakdolgozatom során szerettem volna egy 2D-s platformer játékot megalkotni. A játék elkészítéséhez az Asset Store-ból szereztem be egy ingyenes asset-et. A játék főszereplője egy agilis, dinamikus karakter, aki a klasszikus platformer hősök hagyományait követi. A karakter alapvető mozgásai közé tartozik a futás, az ugrás és a guggolás, valamint a pályák átvihetősége érdekében bevezetésre került a grappling hook mechatika. Ezek a mozgások intuitívak és könnyen kezelhetők, miközben lehetőséget adnak a játékosoknak a pályák különböző kihívásainak megoldására. A játék vezérlőelemei a 4.1. ábrán láthatóak.

A játék vezérlőelemei	
A	Mozgás balra
D	Mozgás jobbra
S	Guggolás
SPACE	Ugrás
SHIFT	Sprintelés
E	Interakció a felvehető tárgyakkal
T	Grappling Hook

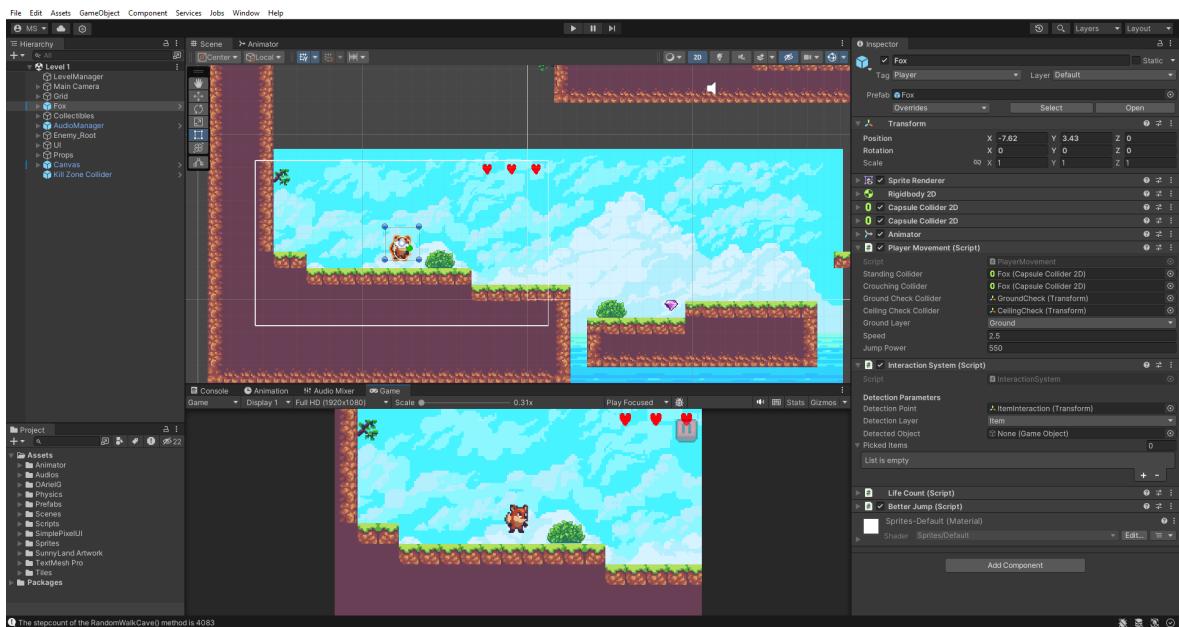
4.1. ábra. A játék közben használt gombok és azok karakterakciói

A játék több különböző szintből áll, amelyek mindegyike új kihívásokat és izgalmat kínál a játékosoknak. A szintek száma és a szintek tervezése a játék fejlesztésének későbbi szakaszában lesz véglegesítve. A pályák kihívást jelentők lesznek az akadályok elhelyezése révén. Ezek az akadályok különböző formákban és méretekben jelennek meg,

és stratégiai gondolkodást igényelnek a játékosoktól a leküzdésükhez. A játék egyik kiemelkedő funkciója a procedurális mapgenerálás, amely a főmenüben választható lesz.

4.2. A Unity szerkesztője

Amint megnyitjuk a Unity-t, a Unity Hub ablak fogad minket. Itt tudunk projektet létrehozni, frissítéseket letölteni, valamint a Unity-t mint game engine-t jobban megismerni a Learn fül alatt. Miután létrehoztuk a projektünket, a Unity szerkesztője fogad majd minket, amelyet a 4.2. ábrán láthatunk.



4.2. ábra. A Unity szerkesztője

A Hierarchy ablak azt a célt szolgálja, hogy felsorolja azokat a GameObject-eket, amelyek az aktuális Scene-en megtalálhatóak. A Project ablaknál találhatók az Assetek, amelyeket az Asset Store-ból installálhatunk, valamint megtalálhatók még az általunk kreált C# szriptek, és a beimportált csomagok. Az Asset-ek a 2D-s vagy 3D-s modellekkel, textúrákkal, anyagjellemzőkkel, hátterekkel, hangfájlokkal jelentik, tulajdonképpen ezekből kreáljuk meg a játékunkat. A Console ablakon keresztül kommunikál a Unity a fejlesztővel, itt jelennek meg a Debug.Log() üzenetek, vagy a fordítási hibák. Az Animator és az Animation ablakok a GameObject-ek animálására szolgálnak. Megtalálható még az Inspector ablak, amely azt a célt szolgálja, hogy a kijelölt GameObject-hez kapcsolódó minden adat megtekinthető, változtatható legyen. Az ábrán látható még a Scene ablak, ahol a játékunk objektumait, hátterét és a pályát szerkeszthetjük. A Scene fül mellett található a Game ablak, ami kizárolag azt jeleníti meg, amelyet a Scene ablakban található kamera objektum lát.

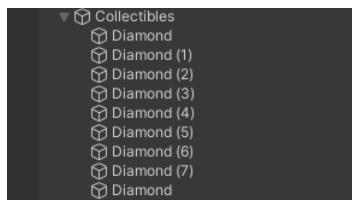
4.2.1. 4.2.1 A GameObject, valamint a szülő-gyerek kapcsolat

A GameObject a Unity egyik legalapvetőbb objektuma. Képviselhet karaktereket, kellékeket, díszletet, kamerákat, útpontokat és még sok más. Lényegében egy GameObject minden olyan objektum, amely elhelyezhető a Scene képernyőn. Fontos megjegyezni,

hogy maga a GameObject nem sok minden csinál; a hozzá csatolt komponensek adják meg a viselkedését, megjelenését és a célját. Például egy "Renderer" komponens hozzáadása láthatóvá teszi a GameObjectt, míg egy "Collider" komponens hozzáadása lehetővé teszi, hogy hasson rá a Unity fizikai motorja.[19]

A Unityben a GameObject-ek rendszerezése a scene-en belül hatékonyan egy szülőgyermek hierarchián keresztül történik. Ez a hierarchikus, fára emlékeztető struktúra lehetővé teszi a GameObject-ek összekapcsolását. A szülő GameObject olyan tárolóként működik, amely hatással van a gyermekéire: a szülőre alkalmazott bármilyen transzformáció, például a mozgatás, forgatás vagy méretezés a gyermek GameObject-ekben is tükröződik. minden gyermek GameObject a szülőjéhez kapcsolódik, és örökli annak transzformációs tulajdonságait. Ez azt jelenti, hogy ha a szülő mozog, a gyermek GameObject is mozogni fog, de megtartja azt a tulajdonságát, hogy függetlenül manipulálható. Például egy autó kerei (gyermek GameObjectk) önállóan is foroghatnak, miközben az autó (szülő GameObject) részei.[20]

A hierarchikus rendszer különösen hasznos az összetett scene-ek kezelésében az összetartozó objektumok csoportosításában. Segít fenntartani a relatív pozíciókat egy objektum különböző összetevői között, amikor az egész objektum mozog. A hierarchia ráadásul különböző szinteken egymásba ágyazható, részletes és szervezett jelenetstruktúrát hozva létre.[20] Például a Collectibles szülő GameObject és annak a Diamond gyermek GameObject-ei a 4.3. ábrán látható.



4.3. ábra. A Colletibles objektum szülő-gyerek kapcsolatai

Ez a szülő-gyermek hierarchia a Unity tervezésének egyik alappillére, amely leegyszerűsíti a jelenetszervezést és biztosítja a kapcsolódó GameObjectek közötti koordinált transzformációkat.

4.3. Az irányítható hős és a kamera

Kezdetben be kell importálni az általunk kiválasztott assetet[1] a Unity szerkesztőjébe, amelynek a neve SunnyLand. Ha ezt megtettük, akkor a Project fül alatt fogjuk látni a beimportált kellékeket, amelyekkel a játékunkat elkészítjük. A hierarchia ablaknál egyelőre csak a MainCamera GameObject-et látjuk, amely a nevéből is adódik, a fő kameránk. Ahhoz, hogy legyen egy irányítható karakterünk, először a beimportált asset közül ki kell választani a karakterünk tétlen pozícióját reprezentáló Sprite-ot, majd egyszerűen a Scene ablakra kell húznunk, így létre is jön egy GameObject, amelynek tetszőleges nevet adhatunk. Az én esetben a neve a Fox lett, mivel a karakterem egy róka.

Ha rámegyünk a Fox objektumra, akkor láthatjuk, hogy az első komponens az a Transform komponens. Ez minden egyes GameObject-hez automatikusan hozzá van rendelve, ez a Unity egyik legfontosabb komponense. Ezzel a komponenssel határozhatjuk meg az objektumunk helyzetét, forgását és a méretarányát. Az objektumunk

helyzetét a „position” fülnél lévő X Y és Z értékek változtatásával tudjuk megváltoztatni. A forgását a „rotation” fülnél levő szintén X Y és Z értékek változtatásával tudjuk megváltoztatni. Végül, a „scale” opciónál lévő X Y és Z értékek határozzák meg a GameObject méretét a jelenetben. Ezeket az értékeket szinte minden esetben manipulálják, változtatják és nem csak a Unity szerkesztőjében, hanem szkripteken keresztül is. [23]

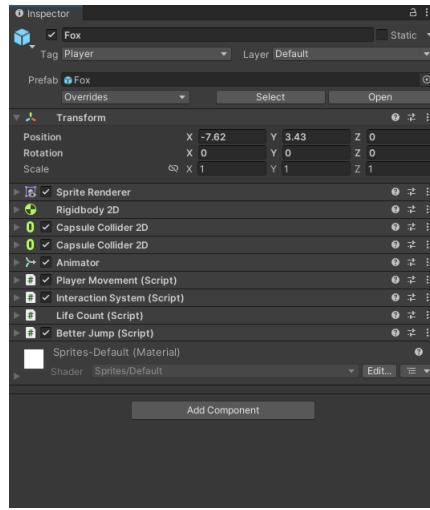
A második komponens pedig a Sprite Renderer komponens, amely lényegében egy olyan eszköz a Unityben, amely a 2D-s képek megjelenítésére és kezelésére szolgál játék közben és nagyfokú ellenőrzést biztosít a képek megjelenítésének és interakciójának módjára.

Ahhoz, hogy ezt a Fox objektumot fizikai alapú objektummá tegyük, hozzá kell adnunk a Rigidbody2D komponenst. A Rigidbody2D komponens a Unityben fizika alapú viselkedést ad a 2D-s sprite-okhoz. Amikor hozzáadjuk ezt a komponenst az objektumunkhoz, akkor a Unity ezt az objektumot a fizikai motorjának az irányítása alá vonja. Ez azt jelenti, hogy hatással lehetnek az objektumunkra különböző fizikai események, például a gravitáció. A Rigidbody2D átveszi az objektum mozgatásának irányítását a Transform komponenstől. Fontos, hogy a szkriptünkben a Rigidbody2D-t mozgassuk a Transform helyett a pontos fizikai szimulációk, valamint az ütközésérzékelés érdekében. Sok dolgot lehet változtatni a Rigidbody2D komponensen belül, mint például a test típusát, amelyet én dinamikusra állítottam annak érdekében, hogy ne lebegjen a levegőben a karakterem. [21]

A másik dolog, amit át kellett állítanom az a „collision detection” érték, amelyet diszkrétről állítottam át folytonosra, hogy egyfolytában érzékelje az objektum, ha egy másik objektummal ütközik. A harmadik fontos dolog, hogy a korlátozások fül alatt lévő „Freeze Rotation Z” rublikát be kell pipálni, mert ha nem tesszük, akkor folyamatosan el fog dőlni a karakterünk.

A következő komponens, amit hozzá kell adnunk az objektumunkhoz, az a Capsule Collider2D, amely egy kapszula alakú ütköztetőt biztosít, amely kölcsönhatásba lép a Unity 2D-s fizikai rendszeréve. Ha egy Rigidbody2D komponens is kapcsolódik ugyanahoz a GameObjecthez, a Capsule Collider 2D lehetővé teszi, hogy az objektum fizikailag kölcsönhatásba lépjen a jelenet más objektumaival. Ez magában foglalja a gravitációra, erőkre és ütközésekre való reagálást. Ezt a komponenst kétszer kellett hozzáadnom az objektumhoz, mivel guggolásnál alacsonyabb méretű és A talajhoz, amelyet kezdetben létrehoztam, ahhoz viszont a Box Collider2D komponenst kellett hozzáadnom.[21] A Fox objektum komponensei a 4.4. ábrán láthatóak.

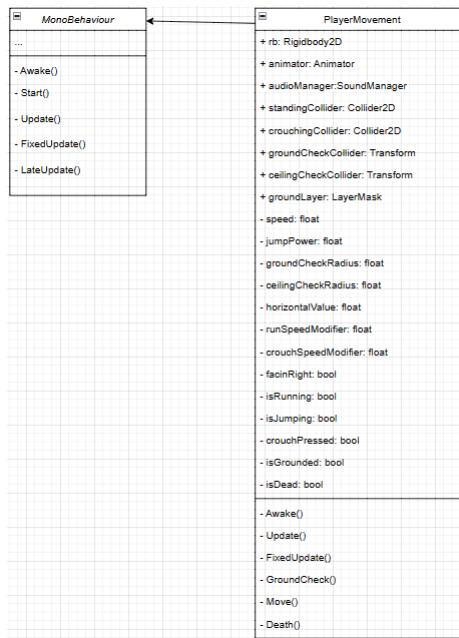
4.3. Az irányítható hős és a kamera



4.4. ábra. A Fox objektum komponensei

4.3.1. A PlayerMovement osztály

Ahhoz, hogy mozgásra bírjam a karakteremet, készítenem kell egy szkriptet. A Project ablakban létrehoztam egy „Scripts” nevű mappát, amelyben létrehoztam egy C# szkriptet PlayerMovement néven. Mielőtt bármit is programoznánk, először is a PlayerMovement szkriptet mint komponenst hozzá kell adnunk a Fox objektumunkhoz. A Unity automatikusan elkészíti nekünk a PlayerMovement osztályt, amely a MonoBehaviour osztályból örököl metódusokat. A PlayerMovement osztály osztálydiagramja a 4.5. ábrán látható.



4.5. ábra. A PlayerMovement osztály osztálydiagramja

Ilyen metódus például az `Awake()` metódus, amely akkor hívódik meg, amikor a jelenetünk betöltődik. Ez a metódus egyetlen egyszer fut le az élete során. A `Start()` metódus az `Awake()` metódushoz hasonlóan szintén egyszer fut le, azzal a különbséggel, hogy az `Awake()` hamarabb hívódik meg, ha minden két metódus szerepel a programban.

Ezeket csak olyan parancsokhoz érdemes használni, amelyeket csak egyszer szeretnénk lefuttatni. Például akkor, amikor referenciakat szeretnénk eltárolni komponens típusú változókban. A `Start()` metódus után következő `Update()` metódus is Unity által definiált, amely minden egyes képkockafrissítésnél hívódik meg. Ebből van több verzió, amelyek a `FixedUpdate()` és a `LateUpdate()`. A `FixedUpdate()` határozott időközönként kerül meghívásra, tehát független a képkockafrissítéstől. A `LateUpdate()` viszont képkockafrissítések között hívódik meg, miután az `Update()` metódus már lefutott.

Én az `Update()` és a `FixedUpdate()` függvényeket használtam. Az `Update()` függvény tartalmazza a `horizontalValue` inicializálását, a sebesség növelését amikor a karakter nem sétál hanem fut, valamint az ugrás és a guggolás parancsait is ebben a függvénybe helyeztem el. A karakterem „halotti állapotát” is az `Update()` függvényben kezelem le, ha az `isDead` változó értéke igaz, a metódus azonnal visszatér, kihagyva minden bemeneti feldolgozást. A `FixedUpdate()` függvényben hívom meg a `Move()` és a `GroundCheck()` metódusokat. Az alábbi metódus a [2] videósorozat alapján készült.

```
void Update()
{
    if (isDead) return;

    //Values between -1 and 1
    horizontalValue = Input.GetAxisRaw("Horizontal");

    //If leftshift is clicked, enable isRunning
    if (Input.GetKeyDown(KeyCode.LeftShift))
    {
        isRunning = true;
    }
    //if leftshift is released, disable isRunning
    if (Input.GetKeyUp(KeyCode.LeftShift))
    {
        isRunning = false;
    }
    if (Input.GetButtonDown("Jump"))
    {
        animator.SetBool("Jump", true);
        isJumping = true;
    }
    else if (Input.GetButtonUp("Jump"))
    {
        isJumping = false;
    }
    if (Input.GetButtonDown("Crouch"))
    {
        crouchPressed = true;
    }
    else if (Input.GetButtonUp("Crouch"))
    {
        crouchPressed = false;
    }
}
```

```
//Set the y velocity in the animator
animator.SetFloat("yVelocity", rb.velocity.y);

}

void FixedUpdate()
{
    GroundCheck();
    Move(horizontalValue, isJumping, crouchPressed);
}
```

A továbbiakban a PlayerMovement osztály metódusait fogom részletezni.

A `GroundCheck()` metódussal azt vizsgálom, hogy a Fox objektum egyik gyermek objektuma, a `GroundCheck` objektum ütközik-e más, a „Ground” rétegen lévő `2DCollider`-ekkel. A `wasGrounded` változó az `isGrounded` változó előző állapotát tárolja. Az `OverlapCircleAll()` metódus a Unity Physics2D-t használja annak ellenőrzésére, hogy a `groundCheckCollider` átfedésben van-e a `groundLayer` bármelyik ütközjével. Ez az ellenőrzés a `groundCheckRadius` által meghatározott körön belül történik. Ezt a metódust egyenlővé tettek a `Colliders2D[]` típusú `colliders` nevezetű változóval, hiszen ha átfedés történik, akkor ez a metódus egy 0-nál nagyobb számmal tér vissza. Ha a `colliders` értéke nagyobb mint 0, akkor az `isGrounded` értékét igazra állítom, valamint, ha a levegőből érkezett a földre a karakterem, akkor egy landolási hangeffekt kerül lejátszásra. Ha a `colliders` értéke egyenlő a 0-val, akkor az `isGrounded` értéket hamisra állítom. A metódus végén a `Jump` animátor paramétert hamisra állítom, hogy az ugrás animáció leálljon. Az alábbi metódus a [2] videósorozat alapján készült.

```
void GroundCheck()
{
    bool wasGrounded = isGrounded;

    //Checking if GroundCheckCollider collides
    //with other 2D Colliders that are in the "Ground" layer
    //If yes (isGrounded true) else (isGrounded false)
    Collider2D[] colliders =
        Physics2D.OverlapCircleAll
            (groundCheckCollider.position,
             groundCheckRadius, groundLayer);
    if (colliders.Length > 0)
    {
        isGrounded = true;
        if (!wasGrounded)
        {
            audioManager.PlaySFX(audioManager.landing);
        }
    }
    else
```

```

{
    isGrounded = false ;
}

//As long as we're grounded
//the jump bool in the animator is disabled.
animator.SetBool("Jump", !isGrounded);
}

```

A `Move()` metódus tartalmazza a karakter X és Y tengelyen való mozgatását, hogyha a karakter földön van és megnyomjuk a SPACE billentyűt, akkor ugorjon, valamint azt is, hogyha a karakter megnyomjuk az S billentyűt, akkor guggoljon. Egy objektumot többféle módon mozgathatunk a programon belül. Egy objektum `Transform` komponensét elérhetjük a `gameObject.transform` osztályon keresztül, ezen belül több lehetőség is van, például a `position`-, a `localScale`-, a `velocity` tulajdonság vagy a `Translate()` metódus.

A `Jumping&Crouching` részben először ellenőrzöm, hogy a `crouchFlag` értéke aktív-e. Ha nem aktív, akkor a `Physics2D.OverlapCircle` segítségével leellenőrzöm, hogy van-e olyan ütköző, amely átfedésben van a megadott pozícióba rajzolt körrrel, amelynek a sugara a `ceilingCheckRadius` változó. Ha az ellenőrzés során bármilyen ütközöt találunk, vagyis valami közvetlenül a Fox objektum fölött van, akkor a `crouchFlag` igaz értéket kap. Ez gyakorlatilag guggoló helyzetben tartja a játékost, amíg egy olyan platform alatt van, amely ütközik a `CeilingCheck` objektummal. A következő fő blokk ellenőri, hogy a játékos földön van-e. Ha igen, akkor feldolgozza az ugrás és a guggolás mechanikáját.

A guggolás mechanikája a következő: ha a `crouchFlag` igaz (mely értéket az S billentyű vagy a fentebb említett feltétel vált ki), akkor a `standingCollider.enabled` értéket hamisra állítom. A guggolás az `animator.SetBool("Crouch", crouchFlag)` segítségével kapcsolható ki és be. Ha a `crouchFlag` értéket igaz, akkor a guggolás animáció lejátszódik, ha hamis, akkor viszont nem. Az ugrás mechanizmusa a következő: még mindig az ellenőrzésen belül, hogy a játékos a földön tartózkodik-e, ha a `jumpFlag` értéke igaz (melyet jellemzően egy ugrás billentyű, esetben a szóköz vált ki), több művelet is történik. Az egyik az, hogy az `isGrounded` értékét hamisra állítom, ezzel jelezve, hogy a játékos ugrást kezdeményezett, és már nem érintkezik a talajjal. Ezután az `audioManager.PlaySFX(audioManager.jumping)` metódus segítségével egy ugrás hanghatás kerül lejátszásra. A `Rigidbody2D` komponensre egy felfelé ható erő fog hatni a `rb.AddForce(new UnityEngine.Vector2(0f, jumpPower))` metódus által. A `Vector2` itt úgy van felépítve, hogy az x értékhez nincs erő rendelve, az y értékhez viszont a `jumpPower` értéket rendelem hozzá, ami függőlegesen fogja felfele tolni a Fox objektumot. Miután lekezeltem az ugrás és a guggolás feltételeit, ismét beállítom a `standingCollider` és a `crouchingCollider` aktív állapotait. A `standingCollider` le van tiltva, ha a `crouchFlag` értéke igaz, ellenkező esetben pedig engedélyezve van. A `crouchingCollider` engedélyezve van, ha a `crouchFlag` igaz, és le van tiltva, ha a `crouchFlag` hamis. Ez azért szükséges, mivel a Fox objektumon kettő darab `CapsuleCollider2D` található (`standingCollider` és a `crouchingCollider`), és ha a játékos ütközne egy ellenséggel, akkor kettő életerőt venne le a játé kostól 1 helyett, mivel minden két komponenssel ütközne az ellenfél. Az alábbi metódus a [2] videósorozat alapján készült.

```

#region Jumping&Crouching
if (!crouchFlag)
{
    if (Physics2D.OverlapCircle
        (ceilingCheckCollider.position,
         ceilingCheckRadius, groundLayer))
    {
        crouchFlag = true;
    }
}

if (isGrounded)
{
    //If we press 'S' we disable the standing collider
    //and animate crouching
    //Reduce the speed by half
    //if released resume the original speed
    //enable the standing collider + disable crouch animation
    standingCollider.enabled = !crouchFlag;
    //If the player is grounded and pressed space -> Jump
    if (jumpFlag)
    {
        isGrounded = false;
        audioManager.PlaySFX(audioManager.jumping);
        //Add jump force
        rb.AddForce(new UnityEngine.Vector2(0f, jumpPower));
    }
}

animator.SetBool("Crouch", crouchFlag);
standingCollider.enabled = !crouchFlag;
crouchingCollider.enabled = crouchFlag;
#endregion

```

A Move&Run részben a karakterem horizontális mozgását, valamint orientációját valósítottam meg. Kezdetben a vízszintes sebesség, melyet az `xVal` változó jelöl, úgy kerül kiszámításra, hogy a `dir` változót megszorozzuk a `speed` attribútummal, majd ezt tovább szorozzuk 100-zal, hogy a Unityben használható sebességskálára konvertáljuk, és a `Time.fixedDeltaTime` segítségével beállítjuk a képkockafrissítési függelenségét. Ez a sebesség módosul annak alapján, hogy a játékos fut vagy guggol: megduplázódik, ha a játékos fut, amit az `isRunning` flag jelez, és megfeleződik, ha guggol, a `crouchFlag` alapján. A kiszámított sebességet ezután a Rigidbody2D-re alkalmazzuk, beállítva annak vízszintes komponensét a meglévő függőleges komponens megtartása mellett, ami lehetővé teszi a gravitációs hatások zavartalan érvényesülését. Ezután a játékos orientációját fogom kezelní a mozgási irány alapján. Ha a játékos irányt vált, a Fox objektumon lévő sprite megfordul, hogy a megfelelő irányba nézzen. Ez úgy történik, hogy a Transform komponens `localScale.x` tulajdonságát -1 vagy 1 értékre állítjuk, így a sprite az `x` tengelyen szükség szerint tükrözve lesz. Végül frissítem az `xVelocity` paramtert az

Animator-ban, hogy tükrözze a Rigidbody aktuális x sebességének abszolút értékét. Ez az érték döntő fontosságú a séta- és futásanimációk vezérléséhez, hogy azok megfeleljenek a karakter tényleges mozgási sebességének. Az alábbi metódus a [2] videósorozat alapján készült.

```
#region Move & Run
float xVal = dir * speed * 100 * Time.fixedDeltaTime;
if(isRunning)
{
    xVal *= runSpeedModifier;
}
if (crouchFlag)
{
    xVal *= crouchSpeedModifier;
}
UnityEngine.Vector2 targetVelocity =
    new UnityEngine.Vector2(xVal, rb.velocity.y);
rb.velocity = targetVelocity;

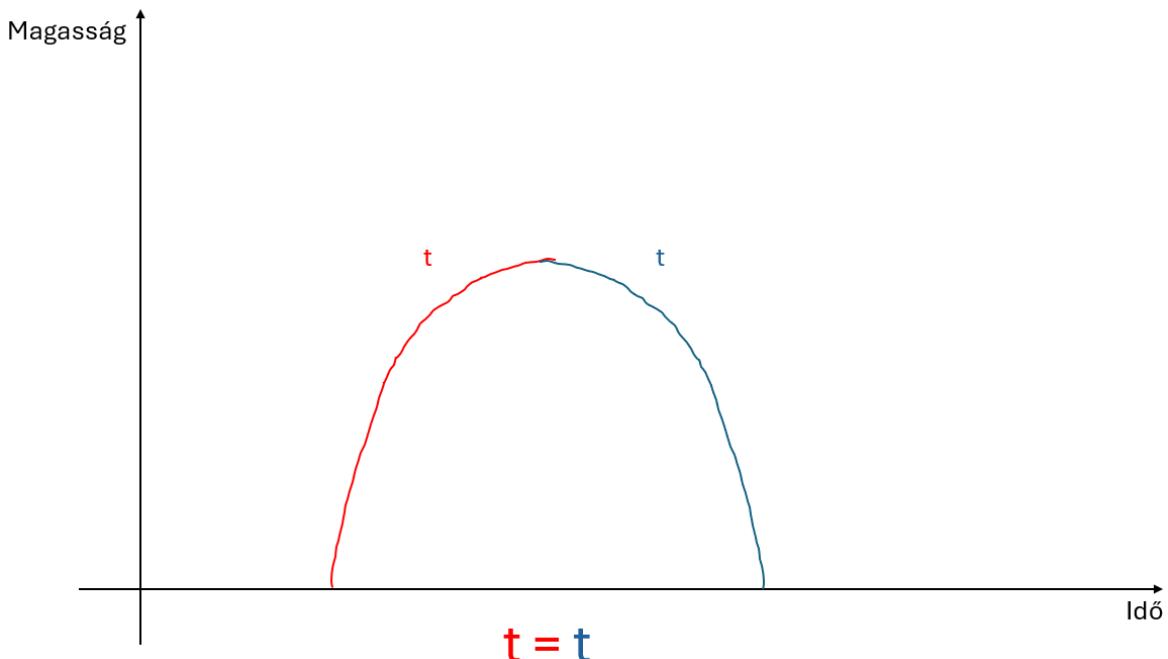
//IF looking right and clicked left ( flip to the left )
if(facingRight && dir < 0)
{
    transform.localScale =
        new UnityEngine.Vector3(-1, 1, 1);
    facingRight = false;
}
// if looking left and clicked right ( flip to the right )
else if (!facingRight && dir > 0)
{
    transform.localScale =
        new UnityEngine.Vector3(1, 1, 1);
    facingRight = true;
}
// set the float xVelocity according to the x value
// of the Rigidbody2D velocity
animator.SetFloat("xVelocity", Mathf.Abs(rb.velocity.x));
#endregion
```

A Death() metódus a játékos halálát kezeli. Az isDead változó értékének az igazra állítása a játékos halottnak jelöli, hogy megakadályozzuk a további bemeneti feldolgozást. Ezután meghívok egy Restart() nevezetű metódust, amely a LevelManager osztályban található, hogy újraindítsam a pályát, ahol a játékos meghalt. Az alábbi metódus a [2] videósorozat alapján készült.

```
public void Death()
{
    isDead = true;
    FindObjectOfType<LevelManager>().Restart();
}
```

4.3.2. A BetterJump osztály

Ezt az osztályt azért hoztam létre, hogy javítsam az ugrás mechanikáját úgy, hogy a Rigidbody2D sebességét a játékos cselekvései és az ugrás állapota alapján sokkal élethűbbre állítom, mivel a karakterem ugrása kezdetben elég furcsának, természetellenesnek tűnt. Ez azért van egyébként, mivel más videójátékokban is a gyorsabb landolás a megszokott, és nem a „lebegős” ugrás. Az igazság az az, hogy a kezdetben implementált ugrás fizikailag helyes, hiszen azt az elképzelést követi, hogy bármennyi időbe telik is az, hogy az ugrásunk csúcsára érjünk, ugyan annyi időbe telik, amíg visszaérünk a földre.[8]



4.6. ábra. Az ugrás és a földet érés időtartama[8]

A videójátékokban ezt teljesen másképp kezelik. Az előbb leírt elmélet azt feltételezi, hogy ugyan annyi képkocka elérni a csúcsot, mint elérni a talajt. Azt kell tehát elérni, hogy több időt töltsön a levegőben a karakter amikor felfelé megy, mint amikor lefelé. Ezt a gravitáció manipulálásával lehet elérni.[8]

Először is létrehozok egy `Rigidbody2d` típusú `rb` nevű változót, amely a `Rigidbody2D` komponensre fog hivatkozni. Utána létrehozok két darab publikus `float` típusú változót, a `fallMultiplier`-t és a `lowJumpMultiplier`-t, amelyek segítségével be fogom tudni állítani a karakteremre ható gravitációs erőt eséskor és alacsony ugráskor (olyan ugrás, amikor az ugrás gombot gyorsan elengedjük), így az ugrás vagy erőteljesebb vagy kevésbé erőteljes lesz. Az `Awake()` metódusban a szkript megkapja a `Rigidbody2D` komponenst a `GameObject`ból, amelyhez a szkript csatlakozik, és az `rb` változóban tárolja.

Az `Update()` metódus aktívan figyeli a játékos ugrási feltételeit, hogy módosításokat alkalmazzon az ugrás fizikájában. Ha a karakter zuhan – amit a játékos sebességében lévő y-komponens jelez -, a szkript növeli a gravitációs erő vonzását. Ez úgy történik, hogy a `Rigidbody` sebességét a `Vector2.up` és a `Physics2D.gravity.y`, valamint a beállított `fallMultiplier` szorzataként kiszámított és a `Time.deltaTime` értékkel

skálázott hozzáadott erővel módosítja. Ez a képlet jelentősen felgyorsítja a játékos esést az ugrás csúcsPontjának elérése után, így valósághűbb és kielégítőbb esést lehet biztosítani.

Ezzel szemben, ha a játékos ugrik, és gyorsan elengedi az ugrás gombot (ezek a pozitív y-sebességgel és az „ugrás” gomb megnyomásának hiányával észlelhető), alacsony ugrást fog eredményezni. Ezt úgy lehet elérni, hogy a `lowJumpMultiplier` segítségével kissé növeljük a gravitációt, ami csökkenti az emelkedés meredekségét és gyorsítja az ereszkedést. Ez a feltétel lehetővé teszi, hogy gyors, kevésbé erőteljes ugrásokat hajtsanak végre a játékosok. Az alábbi osztály a [8] videó alapján készült.

```
public class BetterJump : MonoBehaviour
{
    public float fallMultiplier = 2.5f;
    public float lowJumpMultiplier = 2f;

    Rigidbody2D rb;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    private void Update()
    {
        if (rb.velocity.y < 0)
        {
            rb.velocity +=
                Vector2.up * Physics2D.gravity.y *
                (fallMultiplier - 1) * Time.deltaTime;
        }
        else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))
        {
            rb.velocity +=
                Vector2.up * Physics2D.gravity.y *
                (lowJumpMultiplier - 1) * Time.deltaTime;
        }
    }
}
```

4.3.3. A Grappling Hook mechanizmus

A grappling hook mechanizmus a videójátékokban általában egy kötél, amelyre egy csáklya vagy valamilyen karommal rendelkező eszköz van ráerősítve, majd ezt a játékokban kilövik vagy eldobják, amely megakad a célpontban, és odahúzza a játékost az célponthoz, vagy a célpontot húzza oda a játékoshoz. A játékomban a játékost húzza majd oda a célponthoz, azaz egy falhoz, mivel csak a tile-okhoz tudja majd magát odahúzni a játékosom. Ezt a mechanizmust azért építettem bele a játékba, hogy a

procedurálisan generált pályát a játékosom végig tudja vinni.

Ahhoz, hogy létre tudjam hozni ezt a játékmechanikát, a Fox objektumomhoz hozzá kellet csatolnom egy úgynévezett Distance Joint 2D komponenst, valamint létre kellett hoznom egy üres GameObject-et, amelynek a Rope nevet adtam. Ehhez a Rope objektumhoz hozzá kellett adnom a Line Rendererer komponenst, hogy vizuálisan is látni lehessen a kötelelőt, amelyet majd a Fox objektum pozíciójából a T betű lenyomására kilő a játékos. Ahhoz, hogy a játékos oda tudja magát húzni a megcélzott falhoz, vagy platformhoz, egy szkriptet kellett írjak.

Ennek a szkriptnek a neve Grappling Hook. Ez a szkript a Fox GameObject-hez van csatolva, mivel innen fogja kilőni a játékos a kötelelőt.

Inicializáció:

- **LineRendererer line:** A 2D térben pontok közötti vonalakat rajzoló vizuális komponens. Itt a grappling hook kötelének renderelésére használom.
- **DistanceJoint2D joint:** Fizikai komponens, amely két pontot meghatározott távolságra tart egymástól. Ebben az esetben biztosítja, hogy a játékos és a grappling hook horgonya meghatározott távolságot tartson.
- **Vector3 targetPos:** A grappling hook célpozícióját tárolja, ahová az egérkurzor mutat, amikor a játékos aktiválja a horgot.
- **RaycastHit2D hit:** Tárolja az információt arról, hogy a raycast mit talál el, ami annak meghatározására szolgál, hogy a grappling hook tud-e valamire rögzülni.
- **LayerMask layerMask:** Megszűri, hogy a raycast milyen objektumokat találhat el, javítva ezzel a teljesítményt és biztosítva, hogy a grappling hook csak olyan objektumokkal lépjen kölcsönhatásba, amelyekkel kell.
- **public float distance:** Azt határozza meg, hogy a grappling hook milyen messzire érjen el
- **public float step:** Azt határozza meg, hogy a grappling hook milyen gyorsan húzódjon vissza.

A Start() metódussal inicializálom a DistanceJoin2D-t, és letiltom a LineRendererert, mivel a grappling hook kezdetben nem aktív. Az Update() metódus folyamatosan ellenőrzi, hogy a grappling hook aktív-e, és a távolság alapján beállítja a hosszát vagy leállítja, frissíti a kötel vizuális megjelenítését, hogy kövesse a játékosot, valamintf ilyel a játékos bemenetére a kapaszkodási művelet elindításához és leállításához.

A StartGrappling() metódus átalakítja az egér pozícióját világkoordinátákká, és ellenőrzi, hogy van-e érvényes célpont a grappling hook rögzítéséhez egy raycast segítségével. Ha érvényes célpontot talál, létrehoz egy új GameObjectet, amely a grappling hook horgony pontjaként szolgál, és hozzácsatolja a DistanceJoint2D-t. Ennek az objektumnak a létrehozásához azért volt szükség, mivel a DistanceJoint2D csak olyan objektumhoz tud hozzácsatlakozni, amely rendelkezik Rigidbody2D komponennsel. A platformok viszont nem rendelkeznek Rigidbody komponenssel. Ezután beállítja a vizuális vonalat a játé kostól a célpontig. Az alábbi metódus a [5] és a [6] videók alapján készült.

```

void StartGrappling()
{
    targetPos =
        Camera.main.
        ScreenToWorldPoint(Input.mousePosition);
    targetPos.z = 0;
    hit =
        Physics2D.Raycast(
            transform.position,
            targetPos - transform.position,
            distance, layerMask);

    if (hit.collider != null)
    {
        if (hit.collider.gameObject.
            GetComponent<TilemapCollider2D>() != null ||
            hit.collider.gameObject.
            GetComponent<Rigidbody2D>() != null)
        {
            GameObject anchor =
                new GameObject("GrappleAnchor");
            anchor.transform.position = hit.point;
            Rigidbody2D anchorRb =
                anchor.AddComponent<Rigidbody2D>();
            anchorRb.bodyType =
                RigidbodyType2D.Static;

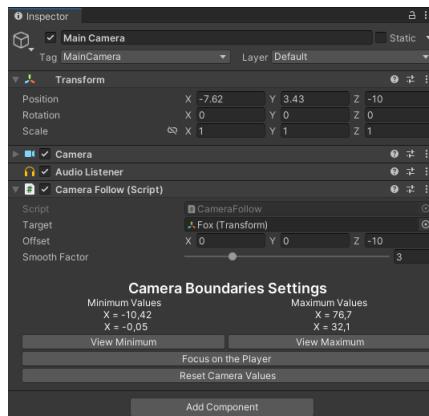
            joint.enabled = true;
            joint.connectedBody = anchorRb;
            joint.connectedAnchor = Vector2.zero;
            joint.distance =
                Vector2.Distance(transform.position,
                    hit.point);
            line.enabled = true;
            line.SetPosition(0, transform.position);
            line.SetPosition(1, hit.point);
        }
    }
}

```

4.3.4. A MainCamera objektum és a hozzá tartozó szkript

A MainCamera objektum az egyik legfontosabb eleme bármelyik projektnek, hiszen a játékos ezen keresztül látja a játékvilágot. A főkamera a játékos szemeként működik a játékkörnyezetben, és megjeleníti a képernyőn megjelenő dolgokat. minden, ami a kamera látóterében van, az az, amit a játékos láthat a játékban. A Unity kamerái perspektivikus vagy ortografikus vetítést használhatnak. A perspektivikus vetítés az

emberi szem látásmódját utánozza, a távolabbi elemek kisebbnek tűnnek, míg a közelebbi objektumok nagyobbnak. Az ortografikus vetítés az objektumokat a kamerától való távolságuktól függetlenül azonos méretben jeleníti meg, ami a 2D-s játékokban nagyon gyakori. Perspektivikus kamerák esetében a látómező (Field of View) határozza meg a megfigyelhető világnak a kiterjedését. A kamerák szkriptek segítségével konfigurálhatók és manipulálhatók, annak érdekében, hogy például kövessék a játékost, filmszerű effekteket hozzanak létre vagy reagáljanak a játék eseményeire, növelte a játék interaktivitását és dinamikáját. A Unity lehetővé teszi több kamera használatát a játékjelenet különböző részeinek egyidejű rendereléséhez. Például az egyik kamera beállítható a játékmenethez, míg egy másik megjelenítheti a felhasználói felületet vagy a speciális effekteket. A Unity ezeket a nézeteket egymásra rétegezi, és így hozza létre a végső képet, amelyet a játékos lát. Az Audio Listener komponens jellemzően a MainCamera objektumhoz kapcsolódik, és a jelenetben elhelyezett hangforrásokkal együtt működik.[18] A MainCamera objektum, valamint a hozzácsatolt komponensek a 4.7. ábrán látható.



4.7. ábra. A MainCamera objektum és komponensei

Ha a hierarchiában röhízzük a kamerát a karakter objektumra, így létrehozva a szülő gyerek kapcsolatot, a kamera követni fogja a játékost. Ekkor a kamera transform értékeit átírva megadhatjuk, hogy mennyire legyen eltolva és elforgatva a karakterhez képest. Kezdésnek ez is elég jó eredményt ad, de amikor már komplexebb mozgásokat szeretnénk megadni a kamerának, programot kell rá írni.

Ezért létrehoztam egy CameraFollow szkriptet, ami a játék kamerájának a mozgásáért, valamint a határainak a beállításáért felel. Létrehoztam egy public Transform típusú, target nevű változót, amely referenciát fog tárolni a Fox objektumról. A Vector3 offset vektor a célpont és a kamera közötti eltolás annak érdekében, hogy a kamera ne legyen közvetlenül a célpont tetején. A float smoothFactor változó a kamera mozgásának az egyenletességét határozza meg. A Vector3 minValues és a Vector3 maxValues vektorok a kamera pozíciójának a minimális és maximális értékeit határozza meg, a kamera mozgását egy adott területre korlátozva.

Az alább látható a Follow() metódus, aminek a célja, hogy zökkenőmentesen kövessen egy célpontot meghatározott határokon belül. A módszer azzal kezdődik, hogy a target aktuális pozíciója alapján meghatározza, hogy a kamerát ideális esetben hová kellene pozicionálni. Hozzáad egy előre meghatározott offset-et a célpont pozíciójához, ami lehetővé teszi, hogy a kamera a célponttól állandó távolságot tartson, ahelyett, hogy közvetlenül a célponton ülne. Ez az eltolás különösen hasznos annak biztosításá-

hoz, hogy a kamera jó rálátást biztosítson az eseményekre, a karaktert a képen tartva, miközben elegendő részt mutat a környezetből.

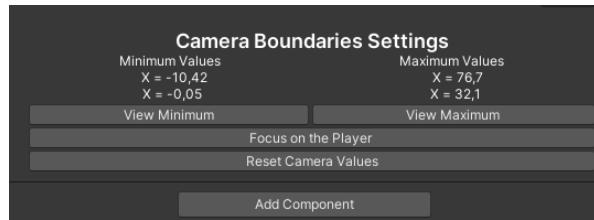
A target pozíójának kiszámítása után a módszer úgy korlátozza ezt a pozíciót, hogy a megadott minimális (`minValues`) és maximális (`maxValues`) határértékeken belül maradjon. Ez a `Mathf.Clamp` segítségével történik, amely a célpozíciót az egyes tengelyekre (`x`, `y`, `z`) vonatkozó `minValues` és `maxValues` által meghatározott tartományra korlátozza. Ez a rögzítés biztosítja, hogy a kamera ne mozduljon ki a játékvilág tervezett területéről, például ne mozogjon a szint szélein túl, vagy olyan területekre, ahol nem történik játék. A kamera végső pozíóját a `Vector3.Lerp` függvény segítségével számítjuk ki. Ez a függvény lineárisan interpolál a kamera aktuális pozíciója (`transform.position`) és a korlátozott célpozíció (`boundPos`) között egy simítási tényező (`smoothFactor`) alapján. Az interpoláció a `Time.fixedDeltaTime` értékkel szorzódik meg, hogy a mozgás sima és egyenletes legyen a különböző képkocka sebességek között. Az alábbi módszer a [2] videósorozat alapján készült.

```
void Follow()
{
    //Define minimum x, y, z values and maximum x, y, z values

    Vector3 targetPosition = target.position + offset;
    // Verify if the target position is out or not
    //Limit it to the min and max values
    Vector3 boundPos = new Vector3(
        Mathf.Clamp
        (targetPosition.x, minValues.x, maxValues.x),
        Mathf.Clamp
        (targetPosition.y, minValues.y, maxValues.y),
        Mathf.Clamp
        (targetPosition.z, minValues.z, maxValues.z));

    Vector3 smoothPosition =
        Vector3.Lerp
        (transform.position, boundPos,
        smoothFactor * Time.fixedDeltaTime);
    transform.position = smoothPosition;
}
```

Miután elkészítettem a `Follow()` módszert, és beállítottam az első pályán a kamera határait, rájöttem, hogy elég időigényes, ezért elkezdtem keresgálni, hogy hogyan is lehetne egyszerűbben, és gyorsabban beállítani ezeket a határokat, mivel nem csak ez lesz az egyetlen szint a játékomban. A legjobbnak tűnő megoldás egy grafikus felhasználói felület elkészítése, amely az Inspector menüben, a CameraFollow szkriptnél jelenik meg. Az általam kreált GUI a 4.8. ábrán tekinthető meg.



4.8. ábra. A MainCamera objektum általam létrehozott GUI része

Ezt úgy készítettem el, hogy egy feltételes fordítási blokkban hoztam létre az egyéni szerkesztőfunkciókat. A `[CustomEditor(typeof(CameraFollow))]` sorral megadtam az alatta implementált osztálynak, hogy melyik futási idejű típusnak lesz a szerkesztője. Ez az osztály a `CameraFollowEditor`, amelynek a szülője nem a `MonoBehaviour`, hanem az `Editor` osztály. Ahhoz, hogy létrehozzak egy egyéni inspector-t, felül kell írjam az `OnInspectorGUI()` metódust, amelyet az `Editor` osztályból örököl az általam kreált `CameraFollowEditor` osztály.

A Unity `DrawDefaultInspector()` metódusát használtam annak a biztosítására, hogy a `CameraFollow` szkript minden nyilvános mezője látható és módosítható legyen. Ahhoz, hogy felhasználóbarát, és könnyed legyen a határok beállítása, különböző GUI-stílusokat és elrendezési elemeket használtam. Specifikus stílusokat határoztam meg, mind például a `defaultStyle`, amely az általános szöveg stílusáért felelős, valamint a `titleStyle`, ami pedig a szakaszcímekért felelős, így a felület sokkal áttekinthetőbbé és látványosabbá vált. Ezeket a stílusokat különböző interaktív GUI-elemekre, köztük címkékre és gombokra alkalmaztam, amelyek végig vezetnek a kamera konfigurációs folyamatán.

Ezek az interaktív elemek különösen hasznosak, hiszen a „View Minimum” és a „View Maximum” gombok lehetővé teszik, hogy az előre meghatározott határokhoz mozogjon a kamera, lehetővé téve a menetközbeni beállításokat és vizuális ellenőrzéseket. A további funkciók közé tartozik még a „Focus On The Player” gomb, amivel azonnal a játékosra irányíthatjuk a kamerát, valamint a „Reset Camera Values” gomb, amely a `CameraFollow` osztály `ResetValues()` metódusát hívja meg, amellyel töröl minden beállítást, és lehetővé teszi a határértékek beállításának az újrakezdését. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
if (script.setupComplete)
{
    GUILayout.BeginHorizontal();
    GUILayout.Label("Minimum Values",
        defaultStyle);
    GUILayout.Label("Maximum Values",
        defaultStyle);
    GUILayout.EndHorizontal();

    GUILayout.BeginHorizontal();
    GUILayout.Label($"X = {script.minValues.x}",
        defaultStyle);
    GUILayout.Label($"X = {script.maxValues.x}",
        defaultStyle);
    GUILayout.EndHorizontal();
}
```

```

GUILayout.BeginHorizontal();
GUILayout.Label($"X = {script.minValues.y}",
    defaultStyle);
GUILayout.Label($"X = {script.maxValues.y}",
    defaultStyle);
GUILayout.EndHorizontal();

GUILayout.BeginHorizontal();
if (GUILayout.Button("View Minimum"))
{
    //Snap the cameraview to the min values.
    Camera.main.transform.position = script.minValues;
}
if (GUILayout.Button("View Maximum"))
{
    //Snap the cameraview to the max values.
    Camera.main.transform.position = script.maxValues;
}
GUILayout.EndHorizontal();

if (GUILayout.Button("Focus on the Player"))
{
    Vector3 targetPos = script.target.position;
    targetPos.z = script.minValues.z;
    Camera.main.transform.position = targetPos;
}

if (GUILayout.Button("Reset Camera Values"))
{
    //Reset the setup complete bool
    //Reset the min max vec3 values
    script.ResetValues();
}

```

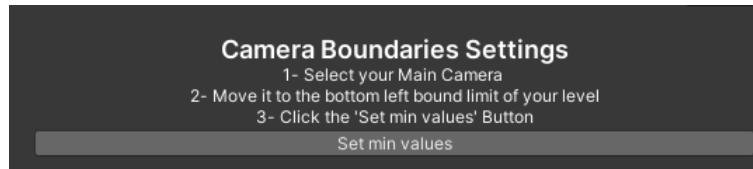
Ezek az opciók azután jelennek meg, miután a grafikus felhasználói felület végigvitte a felhasználót egy struktúrált beállítási folyamatban. A beállítási folyamatnak három állapota van: a Step1, a Step2 és a None. Ha a kamerahatárértékek beállítása még nincs inicializálva (`if(script.state == CameraFollow.SetupState.None)`), akkor a szkript megjeleníti a „Start setting the camera values” gombot. Erre a gombra kattintva a beállítás állapot a „Step1” lesz, és elindul a határértékek beállításának a folyamata.

Az 1. lépést követően a szerkesztő felhasználói felületén utasítások jelennek meg, amelyek a kamera minimális határainak a konfigurálásának a lépései lesznek:

1. A „Select your Main Camera” felszólít minket, hogy a MainCamera legyen az aktív objektum a szerkesztőben
2. "Move it to the bottom left bound limit of your level" arra utasít, hogy manuálisan állítsuk be a kamera pozícióját a játékterület bal alsó sarkába, ahol a kamera határa kezdődik.

3. Kattintsunk a „Set min values” gombra. Erre a gombra kattintva rögzíti a főkamera aktuális pozíóját, és beállítja azt a minimális határértéknek.

A kamerabeállítás lépései a 4.9. ábrán láthatóak.



4.9. ábra. A kamera minimumértékének beállítási lépései

A minimális értékek beállítása után az állapot a második fázisba lép. Ez szinte ugyan az, mint az első fázis, annyi különbséggel, hogy maximális értékeket állítunk be. Ha beállítottuk a maximális értékeket, akkor a beállítás állapota visszatér a „None” állapotba, és a `script.setupComplete` igazra állítódik, jelezve, hogy a kamera határainak a beállítása befejeződött. A határértékek beállításáért felelős kódrészlet az alább megtekinthető. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```

if (script.state == CameraFollow.SetupState.None)
{
    if (GUILayout.Button("Start setting the camera values"))
    {
        //Changes the state to step1
        script.state = CameraFollow.SetupState.Step1;
    }
}
//Step 1: Set up the bottom left boundary (min values)
else if (script.state == CameraFollow.SetupState.Step1)
{
    //Instruction on what to do
    GUILayout.Label($"1— Select your Main Camera",
        defaultStyle);
    GUILayout.Label($"2— Move it to the bottom " +
        $"left bound limit of your level", defaultStyle);
    GUILayout.Label($"3— Click the 'Set min values' Button",
        defaultStyle);

    //Button to set the min values
    if (GUILayout.Button("Set min values"))
    {
        //Set the min values of the cameralimit
        script.minValues = Camera.main.transform.position;
        //Change to step2
        script.state = CameraFollow.SetupState.Step2;
    }
}
//Step 2: Set up the top right boundary (max values)
else if (script.state == CameraFollow.SetupState.Step2)

```

```

{
    //Instruction on what to do
    GUILayout.Label($"1— Select your Main Camera",
        defaultStyle);
    GUILayout.Label($"2— Move it to the top " +
        $"right bound limit of your level", defaultStyle);
    GUILayout.Label($"3— Click the 'Set max values' Button",
        defaultStyle);

    //Button to set the min values
    if (GUILayout.Button("Set max values"))
    {
        //Set the min values of the camera limit
        script.maxValues = Camera.main.transform.position;
        //Change to None
        script.state = CameraFollow.SetupState.None;
        //Last thing is to enable the setupComplete value
        script.setupComplete = true;
    }
}

```

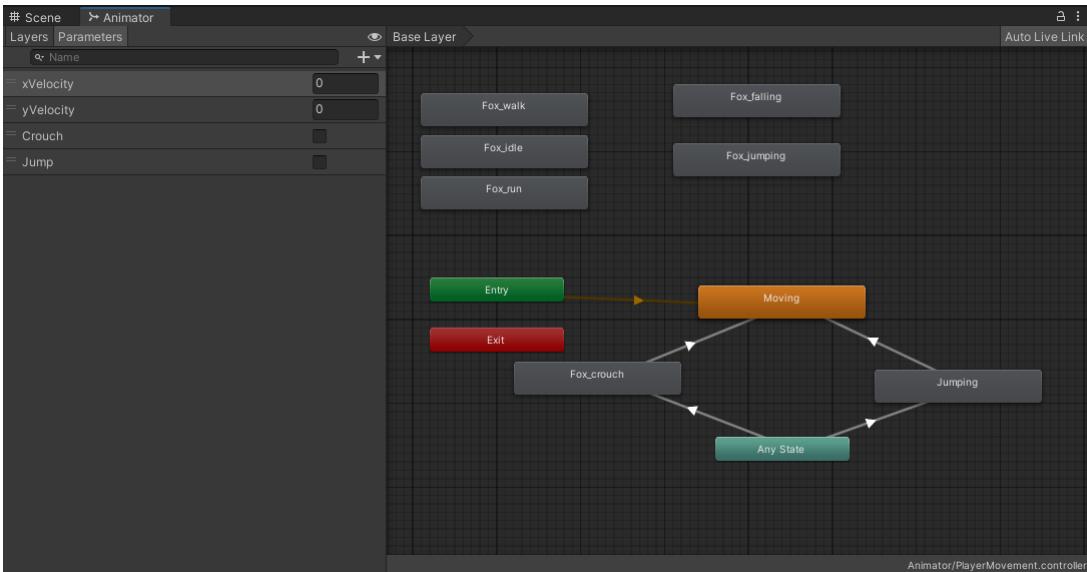
4.4. Az irányítható karakter meganimálása

Anélkül, hogy programot írnánk, a Unity képes animálás segítségével mozgatni objektumokat. Erre jó példa a játékomban például az ugrás, futás, landolás, guggolás, az ellenségek és az érmék animációi. Ezeket az animator controller valósítja meg, melyre gondolhatunk úgy, mint egy állapotgépre[17]. Vannak állapotok, melyek egy-egy animációt reprezentálnak és vannak állapotátmenetek, melyek a végpontjaikban található animációkat felhasználva váltanak az egyik animációról a másikra. Ez a váltás történhet automatikusan, vagy paraméterek, küszöbértékek megadásával is.

Az **animator controller**-t tartalmazó objektum példányosítása után a kontroller a belépési állapottól egészen a kilépési állapotig fut az átmenetek mentén. Az állapotátmenetekkel megadhatjuk, hogy mely állapotokból mely állapotokba lehetséges a váltás. Egyik állapotból egy másik állapotba az átmenet (beállítástól függően) történhet automatikusan (például, ha az állapothoz tartozó animáció véget ért, mehetünk a másik állapotba), vagy paraméterek megadásával, melyeket programon belül állíthatunk be. Mindig csak egy állapot vagy egy átmenet lehet aktív a **controller**-ben. A Fox objektum **animator controller**-e a 4.10. ábrán megtekinthető.

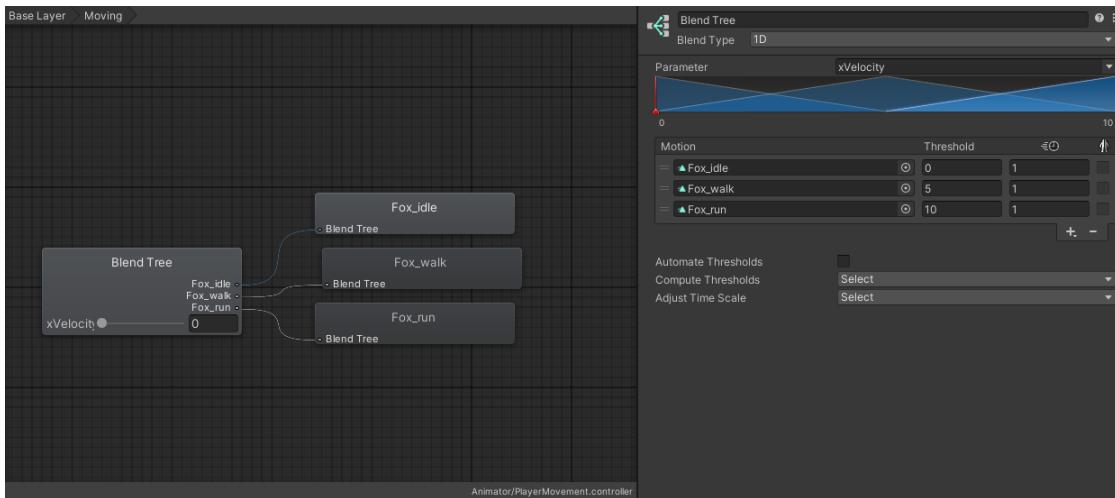
Az **Entry** állapot a belépési pontot, az **Exit** állapot pedig a kilépési pontot jelenti. A **Moving** állapot pedig az alap állapot, amely egy **Blend Tree** melyben szerepelnek a fent lévő **Fox_walk**, **Fox_idle** és **Fox_run** állapotok, amelyeket az Animation fül alatt lehet létrehozni Sprite-ok segítségével. Az **Any State** állapothoz tartozó animációk bármelyik állapot után aktiválódnak, ezért is kapcsolódik hozzá a **Fox_crouch** és a **Jumping** állapot. A **Jumping** szintén egy **Blend Tree**, amelyben a **Fox_falling** és a **Fox_jumping** állapotok szerepelnek. Az Animator ablakon belül a Parameters fünlél lehet változókat definiálni, a Layers fül alatt pedig rétegeket létrehozni.

4.5. A szintek elkészítése



4.10. ábra. A Fox GameObject animator controller-e

A Blend Tree-t állapotgép helyett használtam annak érdekében, hogy különböző küszöbértékek megadásával váltakozzanak az Fox_idle, a Fox_walk és a Fox_run animációk (ezek a Moving Blend Tree-ben láthatók), valamint az ugrásnál is ugyan ezt a szisztemát követtem. A Moving nevű Blend Tree a 4.11. ábrán megtekinthető.



4.11. ábra. A Moving Blend Tree

A 4.11. ábrán látható, hogy a Blend Tree leginkább egy fa gráfra hasonlít. A különböző animációk küszöbértékei a jobb oldalon láthatóak. Ha az xVelocity értéke 0, akkor az Fox_idle animáció játszódik le, ha az értéke 5, akkor a Fox_walk, ha 10, akkor pedig a Fox_run. Programból a paraméterek értékadása az anim.SetBool; és az anim.SetFloat; függvényekkel történnek. Ezeknek a függvényeknek a paraméterei az animátorban beállított paraméter neve és értéke.

4.5. A szintek elkészítése

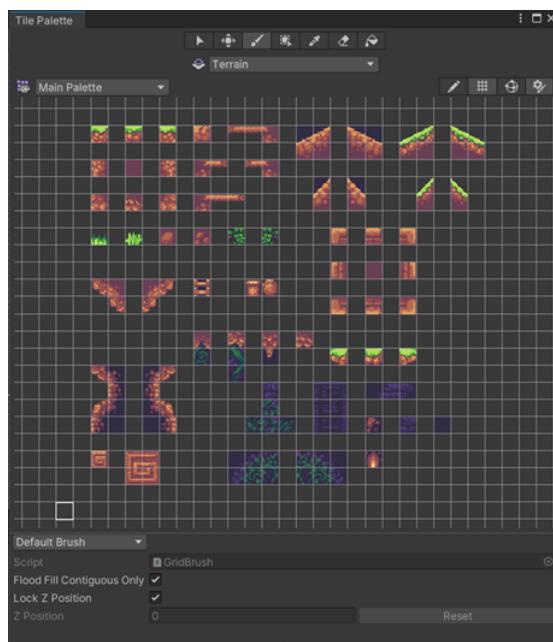
A következő lépéseknek a szintek és a környezet megalkotását gondoltam, hiszen az elkészített animációkat itt lehet igazán kipróbálni és finomhangolni. Magát a pályát a Rule Tiles és a Tile Palette segítségével tudtam könnyedén elkészíteni. A Rule Tiles olyan C# nyelven írt szkripttelhető tile-ok, amelyek elég okosak ahhoz, hogy a megfelelő szomszédos tile-okat használják, és menet közben kezelik a tile-ok animációját, a határokat és az ütközéseket. A Tile Palette pedig a Unity egyik beépített Tilemap készítő egysége. Ezután a hátteret készítettem el, amely követni fogja a kamera mozgását.

4.5.1. A Grib objektum, valamint a Tile Palette használata

A Grid objektum a Tilemaps szülője vagy tárolója. Meghatározza a cellák elrendezését és a benne elhelyezett tile-ok szerkezetét. A Unityben a Grid komponens megszervezi a teret, amelyben a tile-ok elhelyezésre kerülnek, biztosítva, hogy azok egy meghatározott rácsrendszer szerint helyesen igazodjanak egymáshoz. [22]

A Tilemap egy olyan komponens, amely egy rácson belül működik, és a tile-ok gyűjteményét tárolja és kezeli. Lehetővé teszi a 2D tile alapú szintek gyors elkészítését a Tile Palette-ben meghatározott tile-ok felhasználásával.[22]

A Unity Tile Palette egy olyan eszköz, amellyel létrehozhatjuk, rendszerezhetjük és szerkeszthetjük a tile gyűjteményünket. A Tile Palette-et úgy tudjuk megnyitni, hogy a Windows fül 2D opciónál kiválasztjuk azt. Ezután be kell importálni azokat az asset-eket, amelyekkel a pályát fogjuk elkészíteni, ekkor készítünk egy palettát melyet én külön mappába mentettem el. A blokkokat egyszerűen ki kell választani a palettából, és már el is tudjuk készíteni a pályánkat. Persze vigyázni kell, hogy a megfelelő blokkot válasszuk ki, különben esztétikailag nem lesz a legszebb a pályánk. A TilePalette a 4.12. ábrán megtekinthető.

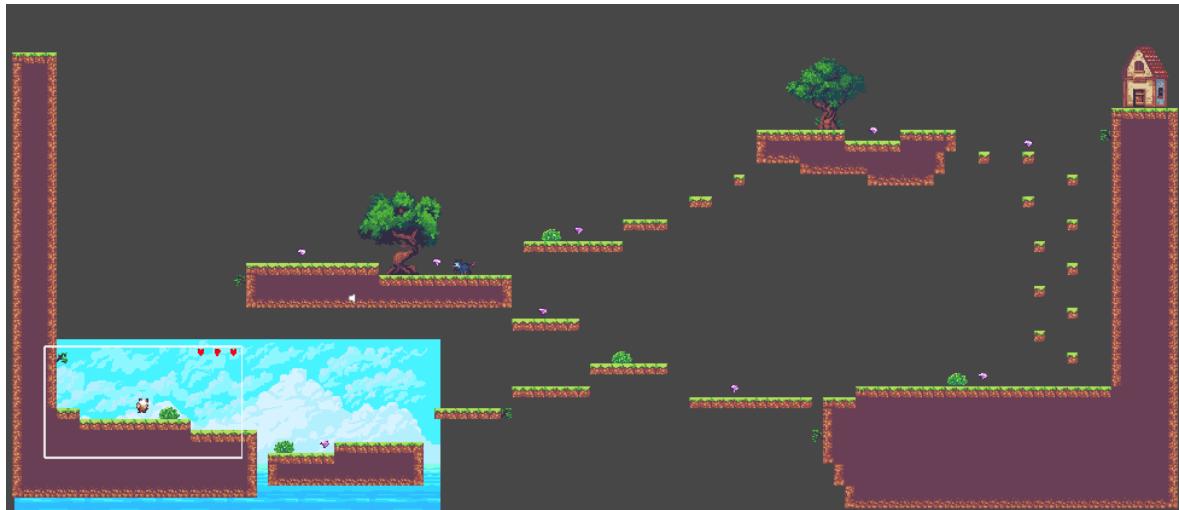


4.12. ábra. A TilePalette

Ezzel a palettával a Terrain nevű Tilemap-re fogok festeni. Fontos tudni, hogy ha több Tilemap-et hoz létre az ember, akkor mindenkorban válassza ki a megfelelőt,

mielőtt nekiáll elkészíteni a pályát. Ennek a palettának a neve a Main Palette, ezen találhatóak azok a sprite-ok, amelyekkel a pályát készítettem el. Miután elkészítettem a pályát, a Terrain objektumhoz hozzá kellett adnom egy úgynevezett Tilemap Collider-t, ezután a rétegbeli sorrendjét kevesebbre kellett állítanom a Tilemap Renderer komponensnél, mint a Fox objektumét, különben eltakarta volna a karaktert. Az elkészített pályára elhelyeztem pár darab dekorációt, amelyek hangulatosabbá tették azt. Ezeket a dekorációkat a Props objektumban helyeztem el a rendszertelenség elkerülése végett.

Az első szint a 4.13. ábrán látható.



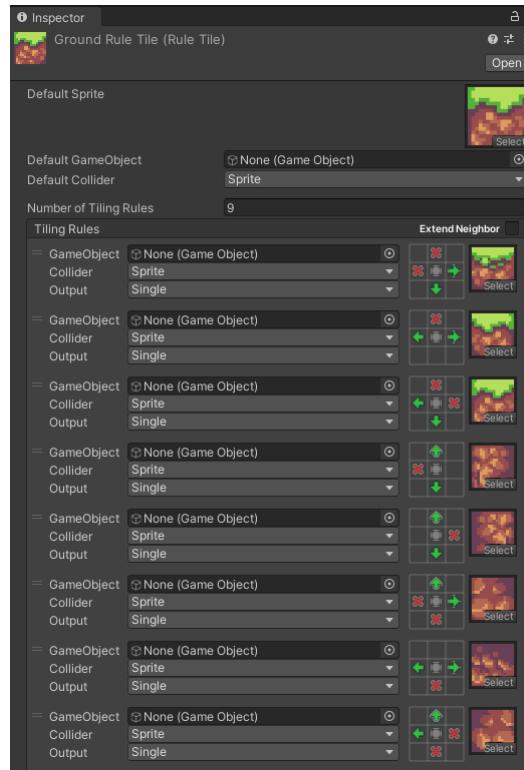
4.13. ábra. A játékom első szintje

4.5.2. A RuleTiles használata

Az első szintet még a RuleTiles használata nélkül készítettem el. Sajnos elég monoton és nehézkes volt az elkészítése, mert minden oda kellett figyelni, hogy a megfelelő tile-t használjam. Ezért keresgéltem az interneten, hogy hogyan is lehetne a pályák elkészítését könnyebbé tenni, és ekkor bukkantam rá a RuleTiles-ra. A Rule Tile Asset egy hatékony és adaptív módja annak, hogy tile-okat fessünk egy jelenetbe. Általában a Tilemap használatakor a tile-ok festése viszonylag gyors, de ismétlődő és nehézkes is lehet, különösen, ha változtatásokat kell végrehajtani a tile-on. A Rule Tiles ezzel szemben képes illeszkedni és alkalmazkodni a környező tile-okhoz, így gyors és kényelmes módját kínálja a tile-ok hozzáadásának vagy módosításának. Ha például egy Rule Tile-t festünk a Scene képernyőre a meglévő tile-ok mellé, a meglévő tile-ok automatikusan frissülnek.[25]

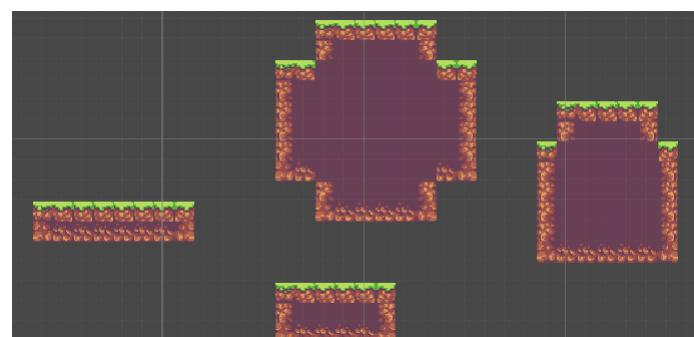
Ahhoz, hogy létrehozzunk egy ilyen RuleTile-t, a fenti legördülő Assets menüpontból ki kell választanunk a Create fült, onnan a 2D-t, utána Tile-t és itt pedig a RuleTile-t amelyet én a Tiles mappába mentettem el Ground RuleTile néven. Ha kiválasztjuk, láthatjuk a tulajdonságait az Inspector fül alatt. Ha ezzel megvagyunk, készítenünk kell egy új palettát, és oda csak szimplán be kell húzni az elkészített RuleTile-t. Miután hozzáadtuk a palettánkhoz, ismét kiválasztjuk az általunk kreált RuleTile-t, és szabályokat kell hozzáadnunk.[25] Én 9 szabályt adtam hozzá, a piros X az éleket jelöli, míg a zöld nyíl azt jelenti, hogy a tile folyamatos. A szabályok a 4.14. ábrán láthatóak.

4.5. A szintek elkészítése



4.14. ábra. A RuleTile-om szabályai

A szabályok beállítása után már kezdhetjük is a szintek könnyed és gyors elkészítését.[25] ■
A RuleTile segítségével elkészített alakzatok a 4.15. ábrán láthatóak.



4.15. ábra. A RuleTiles használatával elkészített példák

4.6. Az ellenség és a felvehető objektumok elkészítése, valamint a karakter életvesztésének és halálának az implementálása

Az ellenség és a felvehető gyémántok animációját is az animator controller segítségével készítettem el. Egy féle ellenség van, egy oda-vissza mozgó oposszum. Az ellenségek száma a pálya szintjétől függ, minél magasabb a pálya szintje annál több ellenséget helyeztem el, így növelte a pálya nehézségét. A pályákat akkor teljesíti a játékos, ha kikerüli az ellenségeket, valamint felszedte az összes gyémántot. A gyémántok számát is növelte a szintek nehezítése, és a hosszabb játékidő érdekében.

4.6.1. Az EnemyAI osztály

Ezt az osztályt azért hoztam létre, hogy az ellenséges karaktert két meghatározott pont között oda-vissza mozgásra késztesse. A `[RequireComponent()]` attribútum a szkript elején arra szolgál, hogy egy BoxCollider2D komponens automatikusan hozzáadódjon minden olyan GameObject-hez, amelyhez hozzácsatoljuk ezt a szkriptet. Készítettem egy `Reset()` metódust, amely akkor hívódik meg, amikor a szkriptet először adjuk hozzá egy GameObjecthez. Ez a metódus meghívja az `Init()` metódust, amely a kezdeti beállításokat konfigurálja. Ezek a beállítások a következők:

1. BoxCollider2D beállítása: A collider-t trigger módba állítja, hogy fizikailag ne blokkoljon más objektumokat, de mégis eseményeket váltszon ki.
2. Root objektum létrehozása: Egy új „_Root” nevű GameObject jön létre, amely az ellenség és annak útpontjai szülőjeként szolgál, így a hierarchia rendezett marad.
3. Útpontok beállítása: Két `waypoint` GameObject jön létre, és ezeknek a szülője, a Waypoints objektum, amely a Root gyereke. Ezek a pontok határozzák meg az ellenség „járórútjának” a kezdő és végpontját.

Ezt a szkriptet azért hoztam így létre, hogy ha újfajta ellenséget szeretnék a játéomba, akkor csak elég legyen hozzáadni az új ellenség objektumához. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
private void Reset()
{
    Init();
}

void Init()
{
    //Make the BoxCollider isTrigger
    GetComponent<BoxCollider2D>().isTrigger = true;
    //Create root object
    GameObject root = new GameObject(name + "_Root");
    //Reset the position of root obj to enemy obj
    root.transform.position = transform.position;
    //Set enemy object as child of root
```

```
transform.SetParent(root.transform);
//Create Waypoints object
GameObject waypoints = new GameObject("Waypoints");
//Reset Waypoints position to root
//Make Waypoints object child of root
waypoints.transform.SetParent(root.transform);
waypoints.transform.position =
    root.transform.position;
//Create two points (gameObject)
//and reset their position to Waypoints object
//Make the points children of Waypoints object
GameObject p1 = new GameObject("Point1");
p1.transform.SetParent(waypoints.transform);
p1.transform.position = root.transform.position;
GameObject p2 = new GameObject("Point2");
p2.transform.SetParent(waypoints.transform);
p2.transform.position = root.transform.position;

//Init points list then add the points to it
points = new List<Transform>();
points.Add(p1.transform);
points.Add(p2.transform);
}
```

Létrehoztam egy `Transform` típusú `points` nevű listát, amely azokat az útvonalakat jelöli, amelyek között az ellenség mozog. A `nextID` a következő útpont indexe a listában, amely felé az ellenség mozogni fog. Az `idChangeValue` egy olyan érték, amely vagy növeli, vagy csökkenti a `nextID`-t, meghatározva a következő útpont célpontját. A `speed` változó pedig az ellenség sebességét szabályozza.

Az `Update()` metódus a `MoveToNextLocation()` metódust hívja meg minden képkockafrissítésnél, hogy a mozgást lekezelje. A `MoveToNextLocation()` metódusban először meghatározzuk az aktuális cél útpontot. Ezután az ellenség orientációját fogjuk megvizsgálni, amely megfordítja az ellenség skáláját a következő pont irányára alapján, hogy arra fele nézzen amelyik útpont irányába halad. A `Vector2.MoveTowards()` funkciót használom az ellenségnek az aktuális pont felé történő mozgatására a meghatározott sebességgel. Ezután elvégzik egy közelség ellenőrzés (proximity check), ami annyit jelent, hogy amint az ellenség elég közel kerül az útponthoz (kevesebb, mint 1 egységnyi távolságra), ellenőrzöm, hogy melyik útpont felé haladjon tovább:

- Ha az utolsó pontnál járunk (`nextID == points.Count - 1`), akkor irányt váltunk az `idChangeValue` -1-re állításával.
- Ha az első pontnál járunk (`nextID == 0`), akkor az `idChangeValue` 1-re történő beállításával irányt változtatunk.
- Ezután frissítem a `nextID`-t, hogy a következő útpontot célozza meg.

A szkript végén végre hajtok egy ütközésérzékelést az `OnTriggerEnter2D` módszerrel. Ha az ellenség ütközik egy „Player” címkelvél ellátott `GameObject`ttel, akkor a `Fox` objektumon (mivel csak ez van ellátva „Player” tag-gel) életvesztést hajt végre

4.6. Az ellenség és a felvehető objektumok elkészítése, valamint a karakter életvesztésének és halálának az implementálása

a `FindObjectOfType<LifeCount>().LoseLife()` metódus meghívásával. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
void MoveToNextLocation()
{
    //Get the next Point transform
    Transform goalPoint = points[nextID];
    //Flip the enemy transform
    //to look into the point's direction
    if( goalPoint.transform.position.x >
        transform.position.x )
    {
        transform.localScale =
            new Vector3(-7, 7, 1);
    }
    else
    {
        transform.localScale =
            new Vector3(7, 7, 1);
    }
    //Move the enemy towards the goal point
    transform.position =
        Vector2.MoveTowards(transform.position,
            goalPoint.position, speed*time.deltaTime);
    //Check the distance between enemy and goal point
    //to trigger next point
    if(Vector2.Distance(transform.position,
        goalPoint.position) < 1f)
    {
        //Check if we're at the end of the line
        //(make the change to -1)
        if(nextID == points.Count - 1)
        {
            idChangeValue = -1;
        }
        //Check if we're at the start of the line
        //(make the change to 1)
        if(nextID == 0)
        {
            idChangeValue = 1;
        }
        //Apply change on the next ID
        nextID += idChangeValue;
    }
}

private void OnTriggerEnter2D(Collider2D collision)
{
```

4.6. Az ellenség és a felvehető objektumok elkészítése, valamint a karakter életvesztésének és halálának az implementálása

```
if( collision.tag == "Player" )
{
    FindObjectOfType<LifeCount>().LoseLife();
}
```

4.6.2. A LifeCount osztály

Ez az osztály egy olyan osztály, amely a játékos életeinek a kezelésére szolgál a játékban, vizuális megjelenítést is biztosít a felhasználói felület elemein keresztül és kezeli a játék állapotát, ha a játékos életet veszít el.

Két publikus változót hoztam létre, az egyik az egy Image típusú tömb, amelyek UI elemek, amik a játékos életeit reprezentálják a felhasználói felületen. A másik az pedig az int típusú livesCount nevű változó, amelynek segítségével meghatározhatjuk, hogy hány élete legyen a játékosnak. A LoseLife() metódusban csökkentük a livesCount értékét eggyel, valamint a lives[livesCount].enabled = false parancssal letiltjuk a livesCount indexnek megfelelő képet a lives tömbben. Ez a művelet vizuálisan megjeleníti egy élet elvesztését azáltal, hogy az egyik élet ikonját elrejti a felhasználói felületről. Ha a livesCount eléri a nulla értéket, akkor a PlayerMovement osztályban definiált Death() metódus hívódik meg. Az alábbi kód részlet a [2] videósorozat alapján készült.

```
public class LifeCount : MonoBehaviour
{
    public Image[] lives;
    public int livesCount;

    public void LoseLife()
    {
        //Decrease the value of lives remaining
        livesCount--;
        //Hide one of the life images
        lives[livesCount].enabled = false;
        //If we run out of lives, we lose the game
        if(livesCount == 0)
        {
            FindObjectOfType<PlayerMovement>().Death();
        }
    }
}
```

4.6.3. Az InteractionSystem és az Item szkript

Az létrehozott InteractionSystem szkript a Fox objektum és a gyűjthető tárgyak (Diamond objektum) közötti interakciók kezelésére szolgál. Először is létrehoztam egy pár darab detektálási paramétert. Az első a Transform típusú detectionPoint, ez az a pont, ahonnan az észlelési terület kiindul. Ez a játékos karakterhez csatlakozik egy a Fox

4.6. Az ellenség és a felvehető objektumok elkészítése, valamint a karakter életvesztésének és halálának az implementálása

objektum gyermekek objektuma, az ItemInteraction GameObject által. A float típusú detectionRadius meghatározza a detectionPoint körüli sugarat, amelyen belül az objektumokkal interakciót lehet folytatni. A LayerMask típusú detectionLayer megadja, hogy az észlelőrendszer melyik réteget vegye figyelembe, lehetővé téve a rendszer számára, hogy csak a megfelelően megjelölt objektumokat észlelj.

Ezután létrehoztam egy GameObject típusú detectedObject nevű változót, amely az éppen detektált objektum referenciáját tárolja, majd létrehoztam egy listát, amely a felvett tárgyakról tart listát. Mivel a felvezésnek hangja is van, ezért létrehoztam egy SoundManager típusú audioManager változót, amely egy hivatkozás a hangkezelőre, amely a felvezés hangot játsza le.

Az Awake() metódusom inicializálja az audioManager-t az „Audio” címkével el-látott GameObject megkeresésével és a SoundManager komponensének elérésével. Ez biztosítja a hanghatások lejátszását. Az Update() metódus ellenőrzi, hogy egy objektum felismerhető-e (if(DetectObject())). Ha a játékos megnyomja az E betűt, amivel interakciót folytathat az objektummal, akkor végrehajtja az Interact() metódust az észlelt objektum Item scriptjéből, és lejátsza a tárgyfelvételkor történő hangot. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
private void Awake()
{
    audioManager =
        GameObject .FindGameObjectWithTag(
            "Audio" ) .
        GetComponent<SoundManager>();
}

void Update()
{
    if ( DetectObject() )
    {
        if ( InteractInput() )
        {
            detectedObject .
                GetComponent<Item>().
                Interact();
            audioManager .
                PlaySFX(
                    audioManager .itemPickup );
        }
    }
}
```

A DetectObject() metódus a Physics2D.OverlapCircle függvényt használja a detectionPoint körüli detectionRadius-on belüli ütközők ellenőrzésére. Ezeket az ütközőket a detectionLayer alapján szűri ki. Ha egy ütközöt észlel, a detektált objektumot az észlelt GameObject-re állítja, és true-t ad vissza. Ha nem észlelt objektumot, törli a detectedObject értéket, és false értéket ad vissza.

A PickUpItem() metódus hozzáadja a megadott tárgyat a pickedItems listához, így ténylegesen nyomon követi a játék során összegyűjtött összes tárgyat. Az alábbi

kód részlet a [2] videósorozat alapján készült.

```
bool DetectObject()
{
    Collider2D obj =
        Physics2D.OverlapCircle
        (detectionPoint.position,
         detectionRadius, detectionLayer);

    if (obj == null)
    {
        return false;
    }
    else
    {
        detectedObject = obj.gameObject;
        return true;
    }
}

public void PickUpItem(GameObject item)
{
    pickedItems.Add(item);
}
```

Az Item szkript a játékban lévő tárgyakkal való interakciók kezelésére szolgál, és a felvehető tárgyakhoz nyújt funkcionálitást. Létrehoztam egy enum típusú változót, amelynek három értéke van: a **NONE**, a **PickUp**, amellyel a felvehetőséget jelzem és az **Examine**, amellyel a megvizsgálhatóságot jelzem. A megvizsgálható tárgyak jelenleg nem szerepelnek a játékomban, de a későbbiekben szeretném ezeket is belerakni. A **public InteractionType type** változó az interakció típusát tárolja. A **Reset()** metódus akkor hívódik meg, ha először csatoljuk egy **GameObject**-hez az Item szkriptet, beállítja a **[RequireComponent(typeof(BoxCollider2D))]** által már felcsatolt **Collider2D** komponenst, hogy triggerként működjön, és hozzárendeli a **GameObject**-et egy adott réteghez a **gameObject.layer = 7** parancssal.

Az **Interact()** metódus egy **switch** utasítást tartalmaz, az interakció típusától függően különböző műveletek kerülnek végrehajtásra:

1. **PickUp**: Hozzáadja az elemet az **InteractionSystem** által kezelt felvett elemek listájához, majd értesíti a **LevelManager**-t, hogy egy Diamond objektumot begyűjtöttek és ezután letiltja a **GameObject**-et.
2. **Examine**: Ez a rész még nincs definiálva a megvizsgálható tárgyak hiánya miatt.

Az alábbi kódrészlet a [2] videósorozat alapján készült.

```

public void Interact()
{
    switch (type)
    {
        case InteractionType.PickUp:
            FindObjectOfType<InteractionSystem>().
                PickUpItem(gameObject);
            FindObjectOfType<LevelManager>().
                GemCollected();
            //Disable the picked up object
            gameObject.SetActive(false);
            break;
        case InteractionType.Examine:
            break;
        default:
            break;
    }
}

```

4.7. A LevelManager és a SoundManager osztály

A LevelManager osztály a játékszint különböző aspektusait kezeli, beleértve a játékosok előrehaladásának nyomon követését, a szintváltások kezelését és a játékbeli események kezelését, például a drágakövek gyűjtését és a szint befejezését. Először is létrehoztam egy Vector2 típusú `playerInitialPos` nevű vektorváltozót, amelyben a Fox objektum kezdeti pozíóját fogom eltárolni a `Start()` metódusban. A `[SerializeField]` `GameObject completionPanel` arra szolgál, hogy a szünet menü referenciáját tárolja el. Az `int` típusú `totalGems` nevű változó megszámolja a szinten lévő felvehető objektumok számát a `Start()` metódusban található `FindGameObjectWithTag().Length` függvény segítségével.

A `Restart()` metódus arra szolgál, hogy ha a játékos elveszti mind a 3 életerejét, akkor a PlayerMovement osztály `Death()` metódusa meghívja azt. Újratöltöm a Scene-t a `SceneManager.LoadScene(SceneManager.GetActiveScene().name)` függvény segítségével, majd a kezdeti pozícióra állítom a játékos helyzetét. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```

public void Restart()
{
    //1 - Restart the scene
    SceneManager.LoadScene(
        SceneManager.GetActiveScene().name);
    //2 - Reset player pos
    FindObjectOfType<PlayerMovement>().
        transform.position = playerInitialPos;
}

```

A `GemCollected()` metódus a `gemsCollected` számlálót minden alkalommal növeli, amikor egy felvehető objektummal interakcióba lépett a játékos. Amikor az összegyűjtött drágakövek száma megegyezik a szinten rendelkezésre álló drágakövek teljes számával (`totalGems`), aktiválja a szint befejezését jelző panelt. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
public void GemCollected()
{
    gemsCollected++;
    if (gemsCollected >= totalGems)
    {
        completionPanel.SetActive(true);
    }
}
```

Található még a szkripten két metódus, a `LoadMainMenu()` és a `LoadNextLevel()` amelyek a szint befejezést jelző panel gombjainak a metódusai. A `LoadMainMenu()`-vel értelemszerűen a főmenüt töltjük be, a `LoadNextLevel()` metódussal pedig a következő szintet. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
public void LoadMainMenu()
{
    SceneManager.LoadScene("Main Menu");
}

public void LoadNextLevel()
{
    SceneManager.LoadScene(
        SceneManager.GetActiveScene().
        buildIndex + 1);
}
```

A `SoundManager` osztály kezeli az összes hanggal kapcsolatos funkciót a játékban, beleértve a háttérzenét és a hangeffekteket (SFX). Először is létre kellett hoznom audiókomponenseket, ezek a hangforrások és a hangklippek. Az egyik hangforrás a `musicSource`, amelyhez a háttérzene lejátszásához hozzárendeltem a Unity szerkesztőjében a `Music` objektumot, amelynek az egyik komponense az `Audio Source`. A másik hangforrás pedig az `sfxSource`, amelyhez a hanghatások lejátszása érdekében ugyan úgy jártam el, mint a `musicSource` esetében, csak az SFX objektumot rendeltem hozzá. A háttérzene és a hangeffektek szétválasztása lehetővé teszi ezek hangerejének és beállításainak független kezelését. Négy darab hangklippet hoztam létre: `background`, `jumping`, `landing` és `itemPickup`.

Az `Awake()` metódusban a singleton mintát használtam annak biztosítására, hogy az `AudioManager` objektum csak egy példánya létezzen a jelenetek között. Ez ellengedhetetlen a folyamatos hanglejátszás (például a háttérzene) fenntartásához a szintek közötti átmentek során. Az alábbi kódrészlet a [2] videósorozat alapján készült.

```
private void Awake()
{
    if (instance == null)
```

```
{  
    instance = this;  
    DontDestroyOnLoad(gameObject);  
}  
else  
{  
    Destroy(gameObject);  
}  
}
```

A `Start()` metódus beállítja és elindítja a háttérzenét, amint a játék elkezdődik. A háttérzenét a Unity szerkesztőjében loop módra kell állítani ahhoz, hogy ha egyszer végigmegy a háttérzene, akkor újrakezdődjön. A `PlaySFX()` metódus egy egyszeri hanghatás lejátszása az `sfxSource` használatával, amely lehetővé teszi bármely átadott `AudioClip` lejátszását. Ez a módszer más szkriptekből is meghívható, például én a `PlayerMovement` szkriptből hívtam meg az ugrás és a landolás hangeffektek lejátszásához, valamint az `InteractionSystem` szkriptből, amikor a játékos felvesz egy gyémántot az `itemPickup` hangeffekt lejátszásához. Az alábbi kód részlet a [2] videóosorozat alapján készült.

```
private void Start()  
{  
    musicSource.clip = background;  
    musicSource.Play();  
}  
  
public void PlaySFX(AudioClip clip)  
{  
    sfxSource.PlayOneShot(clip);  
}
```

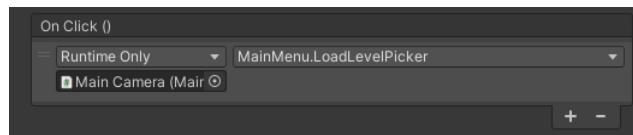
4.8. A felhasználói felület elkészítése

Ebben a fejezetben a játékom fő-, opció-, szintválasztó-, szünet- és a szint teljesítésekkel előugró menü elkészítéséről, a hozzájuk tartozó szkriptek megírásáról lesz szó.

4.8.1. A főmenü

A Unity-nek a beépített UI elemeit használtam fel a Main Menu nevezetű Scene elkészítéséhez. Amikor készítünk egy UI elemet, akkor a Unity automatikusan kreál egy `EventSystem` nevű objektumot, amely az eseményeket kezeli, valamint készít neki egy `Canvas` nevezetű szülőt a hierarchiában, ami igazából egy vászon amire illeszkedni fognak a UI elemek. A `Panel` és a `Button` UI elemeket használtam fel a főmenüm elkészítéséhez. A `Panel` objektumból 2 darab található, az egyik a BackGround, amelynek a Source Image-át a játékomban található background sprite-ra állítottam. A második panel az úgymond egy keret, hogy jobban látszódjanak a `Button` objektumok. Három gombot készítettem el, a `Play`, az `Options` és a `Quit` gombokat. Mindhárom gombnak

egy képet állítottam be háttérnek, amelyeket a Unity Asset Store-on találtam. Ha rá-megyünk bármelyik Button objektumra, akkor az Inspector ablaknál van egy olyan opció, hogy `OnClick()`. Igazából ez egy metódus, és itt tudjuk beállítani, hogy hogyan viselkedjen a gomb, ha az ember rákattint. Az `OnClick()` opció a 4.16. ábrán látható.



4.16. ábra. A Button komponensen belüli `OnClick()` metódus

Ehhez létrehoztam egy `MainMenu` szkriptet, amelyben létre kellett hoznom a megfelelő metódusokat, hogy mi történjen, ha mondjuk a játékos rákattint a Play, az Options vagy a Quit gombokra. Ez egy egyszerű szkript, amelyben három metódus található: a `LoadLevelPicker()`, a `LoadOptionsMenu()`, valamint a `QuitGame()`. A Unity-nek a `SceneManager` osztályát használtam fel arra, hogy a `LoadLevelPicker()` és a `LoadOptionsMenu()` metódusok betöltsék a megfelelő Scene-eket. Ezek a parancsok a `SceneManager.LoadSceneAsync(,,Level Picker")` a szintválasztó menühöz, és a opciók menühöz pedig a `SceneManager.LoadSceneAsync(,,Options Menu")`. A `LoadScene()` funkcióval ellentétben, ez a funkció aszinkron módon tölti be az új jelenetet. A háttérben kezdi el a jelenet betöltését, lehetővé téve, hogy az aktuális jelenet tovább fussen, amíg az új jelenet készen nem áll. Ezzel kerülöm el a játék lefagyását a betöltés alatt. Ezt a szkriptet a `MainCamera` objektumhoz csatoltam, hogy a Button objektumok `OnClick()` metódusánál kiválasszam, és a megfelelő metódust rendeljem hozzá a gombokhoz. A Play gombhoz a `LoadLevelPicker()` metódust rendeltem, az Options gombhoz a `LoadOptionsMenu()` metódust, a Quit gombhoz pedig a `QuitGame()` metódust rendeltem hozzá. Az elkészült főmenü a 4.17. ábrán megtékinthető.



4.17. ábra. A játék főmenüje

4.8.2. A szint kiválasztásáért felelős menü

A Level Picker menü elkészítéskor ugyan úgy jártam el, mint a főmenü elkészítésekor, annyi különbséggel, hogy itt nem három, hanem hétféle darab gombot helyeztem el a

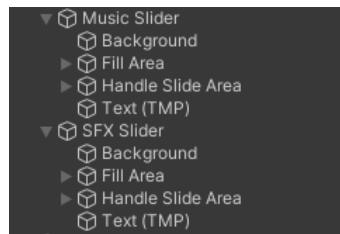
Canvas objektumon. Itt 5 darab gomb a játék szintjeinek a betöltéséért felelős, van egy vissza gomb, amely visszaviszi a játékost a főmenübe, valamint van egy Dungeon gomb is, amelynek segítségével a procedurális mapgeneráló algoritmus által kreált szint töltődik be. Ehhez írtam egy **LevelPicker** nevű szkriptet, amelynek a felépítése hasonlít a **MainMenu** szkript felépítéséhez. A szintválasztó menü a 4.18. ábrán megtekinthető.



4.18. ábra. A játékom LevelPicker menüje

4.8.3. Az opciók menü

Az opciók menü megalkotásánál a fő cél az az volt, hogy a játékban lévő háttérzene, valamint a hangeffektek hangerejét betudja állítani magának a játékos a saját tetszése szerint, valamint egy olyan gomb megalkotása, aminek a megnyomására a Help menü előjön. A hangerőszabályozáshoz a Unity beépített Slider UI elemét használtam. A 4.19. ábrán látható, hogy milyen UI elemekből épül is fel egy Slider. A Background elem az magának a Slider-nek a háttere, vagyis amikor teljesen levesszük a hangerőt, milyen színű legyen a csúszka, én ezt fehérre állítottam. A Fill Area gyermekobjektumát, a Fill objektumot én piros-ra állítottam. Piros színű lesz a slider, ha teljesen felvesszük a hangerőt. A Handle Slide Area gyermekobjektumát, a Handle objektumot alapértelmezetten hagytam, csak a színét állítottam át pirosra. Használtam még Text UI elemet is, hogy meg tudja a játékos különböztetni, hogy melyik hangerőt állítja éppen.



4.19. ábra. A Slider UI objektum



4.20. ábra. Az AudioMixer eszköz

Ahhoz, hogy a játékos tudja állítani a háttérzene és a hangeffektek hangerejét, kézítenem kellett egy AudioMixer-t, valamint egy szkriptet. Az AudioMixer a Unity egyik nagy teljesítményű eszköze, amelyet a projektem hanganyagainak a kezelésére és keverésére használtam. Ez az eszköz lehetővé teszi a fejlesztők számára, hogy különböző hangforrások kezelésével és tulajdonságai dinamikus beállításával szabályozzák a játékuk hangkörnyezetét. Én főleg a hangforrások csoportosításra használtam az AudioMixer-t, hogy kategorizáljam a háttérzene valamint a hangeffekteket. Az AudioMixer-ben, amely a 4.20. ábrán látható, 2 csoportot hoztam létre: a Background és az SFX csoportot, amelyeknek a Master a szülője. Ennek a két csoportnak a Volume változóját exponálnom / fel kellett tárnom a Unity szerkesztőjén belül ahhoz, hogy a **VolumeSettings** szkripten belül elérjem és manipulálhassam őket.

A **VolumeSettings** szkriptben három **[SerializeField]**-el ellátott változót hoztam létre, az egyik az **AudioMixer** típusú **audioMixer** változó, amely a teljes hangkimenetet vezéri. Ezen a keverőn belül lévő speciális paraméterek (**music** és **SFXvolume**), amelyeket fel kellett tárnom a Unity szerkesztőjében, a hangerőszintek beállítására szolgálnak. A **Slider musicSlider** és a **Slider SFXSlider** a létrehozott Slider-ek referenciait tárolják.

A **Start()** metódusban történnek az elmentett hangerőbeállítások betöltése vagy inicializálása. A szkript indításkor ellenőrzi, hogy vannak-e elmentett hangerőbeállítások (**PlayerPrefs.HasKey("musicVolume")**). Ha léteznek beállítások, akkor betölti azokat; ha nincsenek, akkor a hangerőszinteket az aktuális slider-ek pozíciója alapján állítja be. Ez biztosítja, hogy a hangerőre vonatkozó felhasználói beállítások a játékmenetek között is megmaradjanak. Az alábbi kód részlet a [10] videó alapján készült.

```
private void Start()
{
    if ( PlayerPrefs.HasKey("musicVolume"))
    {
        LoadVolume();
    }
    else
    {
        SetBackgroundVolume();
        SetSFXVolume();
    }
}
```

A `SetBackgroundVolume()` és a `SetSFXVolume()` metódusok a hangerőszintek beállítására lettek létrehozva a megfelelő slider értékek alapján. A hangkeverő hangerőszintjét az `audioMixer.SetFloat` segítségével állítjuk be, ahol a hangerő értékét a `Mathf.Log10(volume) * 20` segítségével alakítjuk át. Ez a képlet a `slider` lineáris értékét logaritmikus skálára alakítja át, amely jobban utánozza az emberek hangsintérzékelését. Mindkét módszer elmenti az aktuális `slider`értéket a `PlayerPrefs`-be, így a beállítások tartósak maradnak majd. Az alábbi kódrészlet a [10] videó alapján készült.

```
public void SetBackgroundVolume()
{
    float volume = musicSlider.value;
    audioMixer.SetFloat("music",
        Mathf.Log10(volume) * 20);
    PlayerPrefs.SetFloat("musicVolume", volume);
}

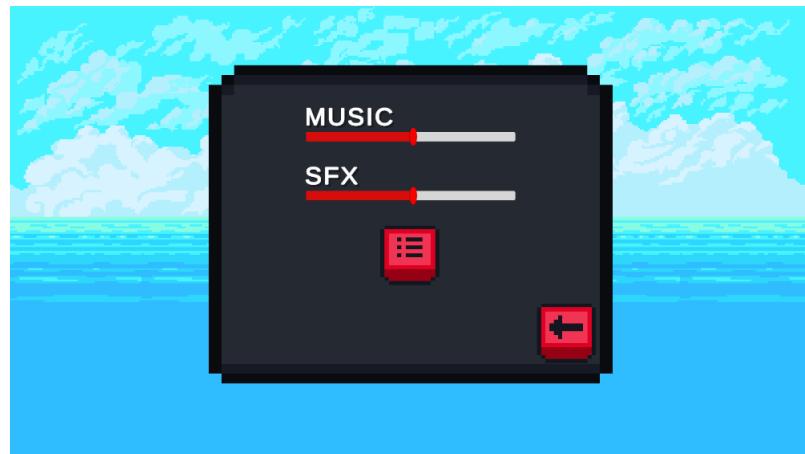
public void SetSFXVolume()
{
    float volume = SFXSlider.value;
    audioMixer.SetFloat("SFXvolume",
        Mathf.Log10(volume) * 20);
    PlayerPrefs.SetFloat("SFXVolume", volume);
}
```

A `LoadVolume()` metódus a `PlayerPrefs`-ból lekérdezi a mentett hangerőbeállításokat, és ennek megfelelően frissíti a csúszkák pozíciót. Itt meghívásra kerülnek a `SetBackgroundVolume()` és a `SetSFXVolume()` metódusok, hogy a hangkeverő beállításai frissüljenek a betöltött értékeknek megfelelően.

Az elkészült szkriptet a Canvas objektumhoz csatoltam, majd ezt a Canvas objektumot a Slider objektumok Slider komponensénél lévő On Value Changed metódusnál referáltam, hogy a megfelelő funkciót ki tudjam választani a két csúszkának.

A következő lépés egy kisebb szkript írása, amely majd a vissza gomb és a Help gomb metódusait fogja tartalmazni. Ezt OptionsMenu nevén neveztem el, és ez a szkript három metódust tartalmaz. Az egyik, a `LoadMainMenu()` amely a vissza gomb metódusa, és betölti a játék főmenüjét, a másik a `LoadHelpMenu()`, amely a Canvas objektumban lévő HelpMenu objektumot hozza elő, valamint az OptionsMenu objektumot eltünteti. A harmadik metódus a `BackToOptionsMenu()`, amely eltüneti a HelpMenu objektumot, és előhozza az OptionsMenu objektumot.

Az OptionsMenu objektum a 4.21. ábrán, a HelpMenu objektum a 4.22. ábrán tekinthető meg.



4.21. ábra. A játék opciók menüje



4.22. ábra. A HelpMenu UI objektum

5. fejezet

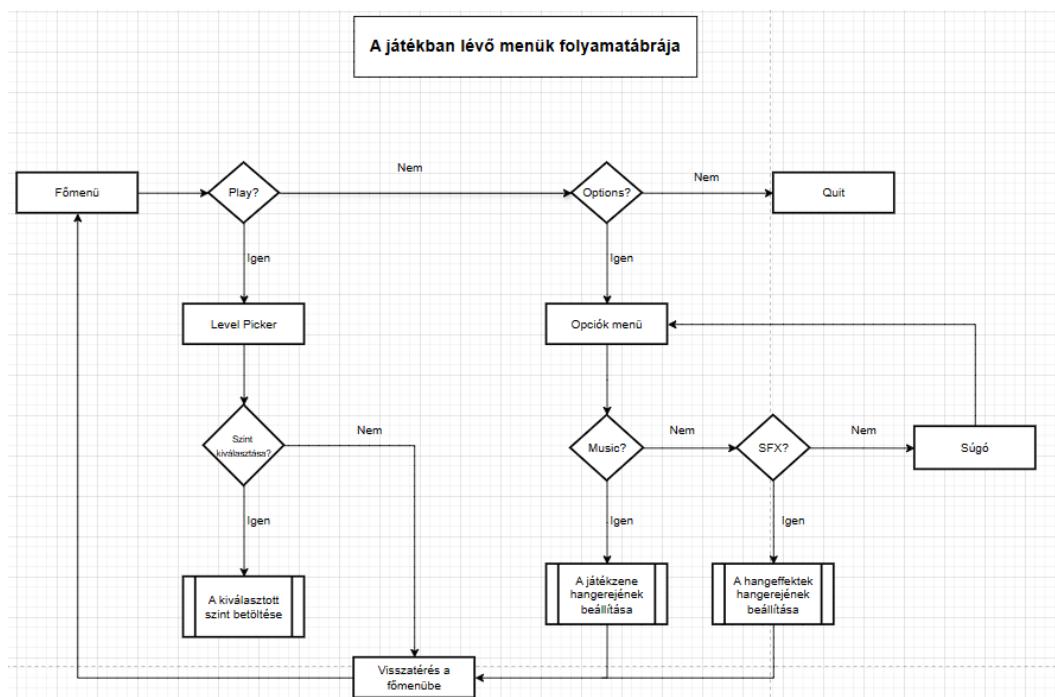
A kész játék bemutatása

Ebben a fejezetben a játék végleges változatát fogom bemutatni, részletezve a felhasználói felületet, a játékmechanikát és a játékosok interakcióit.

A játék elindításakor a játékosok a főmenübe kerülnek. Innen a játékosok három lehetőség közül választhatnak:

- **Start:** "Start" gomb kiválasztásával a játékosok a szintválasztó menűhöz jutnak, ahol kiválaszthatják a kívánt szintet.
- **Options:** Ez a menü lehetővé teszi a játékosok számára az audiolétreállítások módosítását, beleértve a zene és a hangeffektek hangerejét. Ezen kívül a játékosok innen érhetik el a Súgó menüt, ahol a játék mechanikáit tudják megnézni.
- **Quit:** Ha ezt az opciót választják, a játékos kilép a játékból.

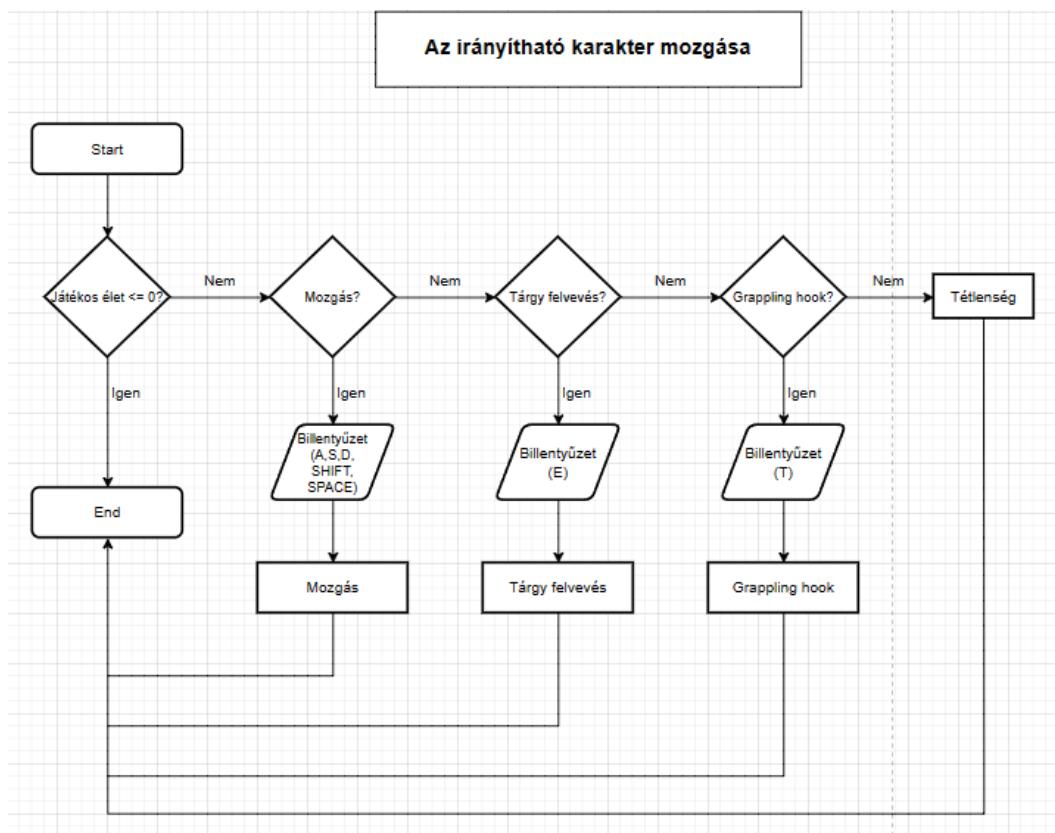
A játék menüinek és azok interakcióinak a folyamatábrája az 5.1. ábrán megtekinthető.



5.1. ábra. A játékban található menüknek a folyamatábrája.

Ha a játékos kiválaszt egy pályát, a játék betöltődik, és a játékos a pálya kezdőpontjára áll. Az összes pályán az elsődleges cél az ellenségek kikerülése, az akadályok átugrása és a rendelkezésre álló gyémántok összegyűjtése. A hátralévő gyémántok számát a képernyő bal felső sarkában lévő számláló mutatja. Egy gyémánt begyűjtése eggyel csökkenti ezt a számlálót.

A játékos az A, D, S, SPACE és SHIFT billentyűkkel tudja irányítani a főhőst. Az A billentyű lenyomásával balra, a D billentyű lenyomásával pedig jobbra tud menni a játékos. Ha a SHIFT gombot megnyomja a játékos és lenyomva tartja, akkor futni fog az irányítható karakter. A SPACE gomb gyors lenyomásával egy kis ugrást tud végezni a karakter, viszont ha a játékos lenyomva tartja a SPACE billentyűt, akkor az magasabba tud ugrani. A játékos a felvehető objektumokat az E billentyű megnyomásával tudja felvenni, a grappling hook játékmechanizmust pedig a T betűvel tudja használni. Az irányítható karakter folyamatábrája az 5.2. ábrán megtekinthető.



5.2. ábra. Az irányítható karakter mozgásának a folyamatábrája

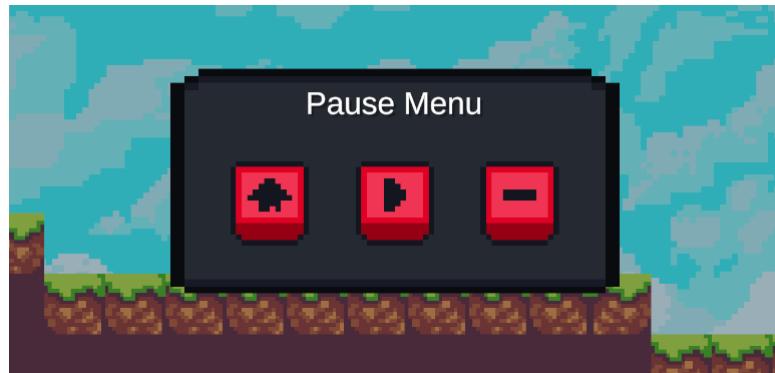
A játékosok minden szintet három élettel kezdenek, amelyeket piros szívek jelképezznek a képernyőn. Egy ellenséggel való találkozás egy élet elvesztését eredményezi. Az összes élet elvesztése azt eredményezi, hogy a játékos újrakezdi a szintet. Ha a játékos leesik a térképről, elveszíti minden életét, és szintén újra kell kezdenie a szintet.

A játék bármikor szüneteltethető a képernyő jobb felső sarkában található szünet gomb megnyomásával. A szünet menüben a játékosoknak három lehetőségük van:

- A házikó megnyomásával a játékos visszatérhet a főmenübe.
- Az elfordított háromszög megnyomásával a játékos folytathatja a játékot ott, ahol abbahagyta.

-
- A minusz jel megnyomásával a játékos újrakezdheti az adott szintet.

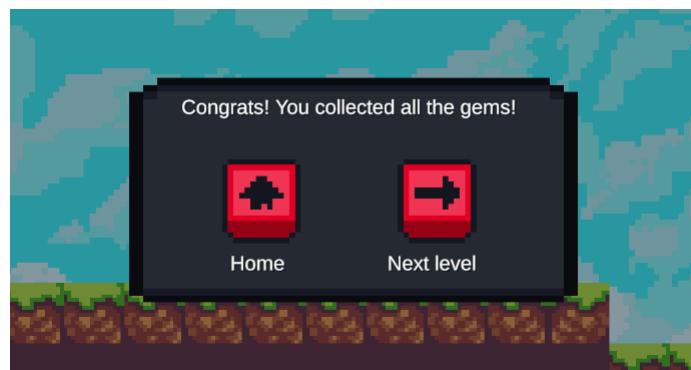
A szünet menü az 5.3. ábrán látható.



5.3. ábra. A játék szünet menüje.

Ha egy szinten belül minden gyémántot összegyűjtött a játékos, megjelenik a szint végét jelző menü, ahol a játékosok választhatnak, hogy továbblépnek-e a következő szintre, vagy visszatérnek a főmenübe.

A szint végét jelző menü az 5.4. ábrán látható.



5.4. ábra. Az adott szint végét jelző menü.

6. fejezet

Összefoglalás

A szakdolgozatomban bemutattam a platformer játékok fejlődését, valamint ezeknek a játékoknak a procedurális mapgeneráláshoz való kapcsolódását. Sikeresen bemutattam a különböző játékmotorok néhány tulajdonságát, a Unity-ben használt mechanizmusok, koncepciók mélyebb részletezésével együtt.

Láthattunk három darab olyan algoritmust, aminek a segítségével automatikusan lehet pályát generálni egy 2D-s játékhoz. Ez a három algoritmus a Perlin-zaj, a celluláris automata valamint a véletlen bolyongás. Én a véletlen bolyongás algoritmus választottam a játékomhoz, mivel ez illett az általam létrehozott preferenciákhoz a legjobban. Sajnos azt nem sikerült elérnem, hogy a játékos preferenciáit figyelembe véve generáljon pályát, de ezt orvosolni fogom egy saját menü létrehozásával. A jövőben szeretném vegyíteni a pályageneráló algoritmusokat, valamint nem csak 2D-s, hanem 3D-s játékhoz is alkalmaznám ezeket.

A szakdolgozatom hozzájárult a videójáták fejlesztés megismeréséhez is, hiszen láthattuk, hogyan lehet GameObject-ekből valamint a hozzájuk csatolható komponensekből egy teljesen működő 2D-s platformer játékot létrehozni. Részleteztem a fejlesztés során létrehozott osztályokat, valamint metódusokat és igyekeztem ezeket minél érthetábban elmagyarázni, azonban néhol túlságosan is implementáció-közeli lett a dolgozatom.

A játék sajnálatos módon nincs még olyan állapotban, hogy a kereskedelmi forgalomban is megállja a helyét. Hiányoznak még belőle a megvizsgálható tárgyak, több szintet kellene létrehoznom a játékban, valamint több játszható karakter is növelné a játék élményét, játszhatóságát.

Összességeiben elégedett vagyok magammal, hogy sikerült elkészítenem életem első játékát, valamint még az automatikus pályagenerálást is sikerült beleépítenem a játékomba. A szerzett tudást mindenképpen fel fogom használni a jövőben, mondjuk egy 3D-s játék készítésénél.

Irodalomjegyzék

- [1] Ansimuz. Sunnyland. <https://assetstore.unity.com/packages/2d/characters/sunny-land-103349>, 10.11.2022. Accessed: 04.2024.
- [2] Antarsoft. Unity 2d platformer tutorial for beginners. https://www.youtube.com/playlist?list=PLjAb99vXJuCRD04EUp8p2az1ILZbq_ZfY, 16.02.2022. Accessed: 04.2024.
- [3] Ethan Bruins. Procedural patterns to use with tilemaps, part 2. <https://blog.unity.com/engine-platform/procedural-patterns-to-use-with-tilemaps-part-2>, 07.06.2018. Accessed: 04.2024.
- [4] Graham Cox. Procedural generation of computer game maps. <https://www.baeldung.com/cs/gameplay-maps-procedural-generation>, 18.03.2024. Accessed: 03.2024.
- [5] DonHaul Game Dev. Grappling hook (part 1) - unity 2d tutorial (c#). https://www.youtube.com/watch?v=sHzWlrTgJo&ab_channel=DonHaulGameDev-Wobble-UnityTutorials, 17.09.2015. Accessed: 05.05.2024.
- [6] DonHaul Game Dev. Grappling hook (part 2) - unity 2d tutorial (c#). , 19.09.2015. Accessed: 05.05.2024.
- [7] Flafla2. Understanding perlin noise. <https://adrianb.io/2014/08/09/perlinnoise.html>, 09.08.2014. Accessed: 03.2024.
- [8] Board To Bits Games. Better jumping in unity with four lines of code. https://www.youtube.com/watch?v=7KiK0Aqtmc&t=104s&ab_channel=BoardToBitsGames, 20.03.2017. Accessed: 04.2024.
- [9] Epic Games. Features. <https://www.unrealengine.com/en-US/features>. Accessed: 03.2024.
- [10] Rehope Games. Unity audio volume settings menu tutorial. https://www.youtube.com/watch?v=G-JUp8AMEx0&ab_channel=RehopeGames, 7.03.2023. Accessed: 05.05.2024.
- [11] Nodwin Gaming. The evolution of platform games in 9 steps. <https://www.redbull.com/in-en/evolution-of-platformers>, 23.03.2017. Accessed: 03.2024.
- [12] Godot. Features. <https://godotengine.org/features/>. Accessed: 03.2024.

-
- [13] Noveltech. The unity advantage: Why developers choose unity as their game engine. <https://www.noveltech.dev/the-unity-advantage>, 13.07.2023. Accessed: 03.2024.
 - [14] Noveltech. Generating a 2d map using the random walk algorithm. <https://www.noveltech.dev/procgen-random-walk>, 17.07.2023. Accessed: 03.2024.
 - [15] Jane Tu. What is a platformer game? (features and examples). <https://www.powerofplatform.com/what-is-a-platformer-game-features-and-examples/>, 22.11.2022. Accessed: 03.2024.
 - [16] Unity. 2d game development. <https://docs.unity3d.com/Manual/Unity2D.html>. Accessed: 03.2024.
 - [17] Unity. Animator controllers. <https://docs.unity3d.com/Manual/AnimatorControllers.html>. Accessed: 04.2024.
 - [18] Unity. Cameras. <https://docs.unity3d.com/Manual/CamerasOverview.html>. Accessed: 04.2024.
 - [19] Unity. Gameobject. <https://docs.unity3d.com/Manual/class-GameObject.html>. Accessed: 04.2024.
 - [20] Unity. The hierarchy window. <https://docs.unity3d.com/Manual/Hierarchy.html>. Accessed: 04.2024.
 - [21] Unity. Introduction to rigidbody2d. <https://docs.unity3d.com/Manual/class-Rigidbody2D.html>. Accessed: 04.2024.
 - [22] Unity. Tilemaps. <https://docs.unity3d.com/Manual/Tilemap.html>. Accessed: 04.2024.
 - [23] Unity. Transforms. <https://docs.unity3d.com/Manual/class-Transform.html>. Accessed: 04.2024.
 - [24] Unity. Unity asset store. <https://assetstore.unity.com/>. Accessed: 04.2024.
 - [25] Unity. Using rule tiles. <https://learn.unity.com/tutorial/using-rule-tiles#>. Accessed: 04.2024.
 - [26] Wikipedia. Cellular automaton. https://en.wikipedia.org/wiki/Cellular_automaton. Accessed: 03.2024.
 - [27] Wikipedia. Conway's game of life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life. Accessed: 03.2024.
 - [28] Wikipedia. Game engine. https://en.wikipedia.org/wiki/Game_engine. Accessed: 03.2024.
 - [29] Wikipedia. Perlin noise. https://en.wikipedia.org/wiki/Perlin_noise. Accessed: 03.2024.
 - [30] Wikipedia. Unreal engine. https://en.wikipedia.org/wiki/Unreal_Engine. Accessed: 03.2024.

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).