# 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 játékkörnyezetben 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 paraméterezhetőségre és az adaptivitásra, hogy a generált pályák ne csak változatosak és kihívást jelentőek 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.

# Az oldalnézetes játékok, valamint a játékmotorok jellemzői

## A fejezet célja

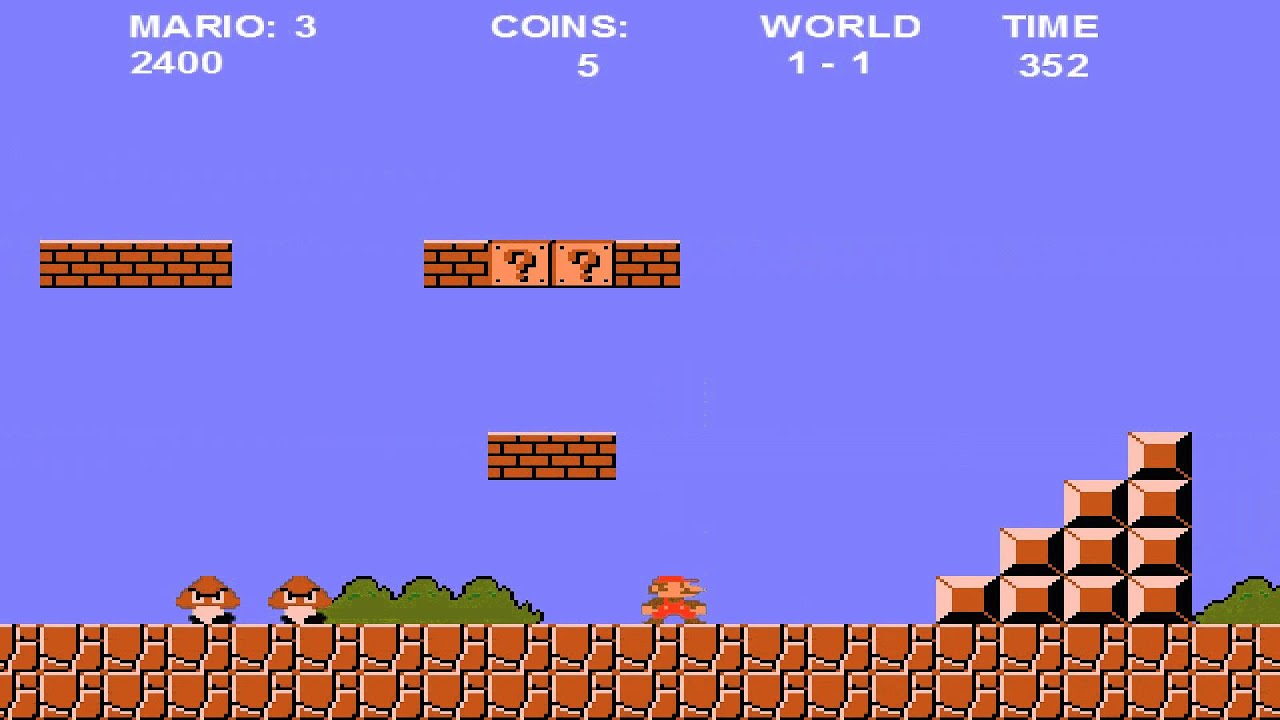
Ebben a fejezetben arra fogok törekedni, hogy részletesen bemutassam az oldalnézetes (side-scroller) játékok és a pályageneráló algoritmusok világát. Megvizsgálom az oldalnézetes játékok történelmi fejlődését, általános jellemzőit, és azt, hogy hogyan kapcsolódnak ezek a játéktípusok a pályageneráláshoz.

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

## Az oldalnézetes játékok

### 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.
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.
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.

2.2.1.1. ábra A Super Mario Bros. játékmenete

1. **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.
2. A képen rajzfilm, Számítógépes játék, játék, Stratégiai videojáték látható

   Automatikusan generált leírás**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.

2.2.1.2. ábra A Crash Bandicoot játék

1. A képen szöveg, köd, képernyőkép, fekete látható

   Automatikusan generált leírás**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.

2.2.1.3. ábra A Limbo játék

1. **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.

### Általános jellemzőik

A platformer játék, más néven platform videojáték, 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 feladatokat teljesítsen, magas pontszámokat érjen el, vagy egyszerűen csak életben maradjon. Mivel ez a játékműfaj az évek során jelentősen megváltozott, sok ilyen játéknak más lehet a látványvilága. A következő jellemzők azonban gyakran megtalálhatók a platformjátékokban.

1. **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.
2. **Third-Person nézőpont:** A játékos által irányított karakter az előtte lévő képernyőn látható, mivel sok platformjátékot úgynevezett „third-person” perspektívából készítenek.
3. A képen rajzfilm, Animációs film, clipart látható

   Automatikusan generált leírás**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.

2.2.2.1. ábra Horizontáls és vertikális mozgás

1. **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.
2. **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.

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.

### 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.

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 biztosíthat a játékunkban, sokkal alacsonyabb előzetes előállítási költségekkel.

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 terepviszonyokkal 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.

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.

A képen képernyőkép, szöveg látható

Automatikusan generált leírásTöbb játék is nagyszerűen használta az procedurális generálást. Például a Roguelight című játék bemutatja, hogy a procedurális generálással hogyan lehet mélyebb és sötétebb környezetet létrehozni, amely minden egyes játékmenettel változik.

2.2.3.1. ábra A Roguelight játék

A képen Grafikus tervezés, szöveg, képernyőkép, poszter látható

Automatikusan generált leírásA "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.

2.2.3.2. ábra A Diskophoros játék

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

## 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 elé tárnak.

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.

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 projektekhez fejleszti, ami lehetővé teszi a játék követelményeihez jobban illeszkedő és testre szabható funkciókat.

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

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.

### Az Unreal Engine játékmotor

Az Epic Games által fejlesztett Unreal Engine gazdag múltra tekint vissza a videojáték-fejlesztés 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.

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.

A képen képernyőkép, szöveg, Multimédiás szoftver, szoftver látható

Automatikusan generált leírás

3.1.1.1. ábra. Az Unreal Engine Blueprint nevezetű visual scripting rendszere

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 produkciókban.

### 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. 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.

A képen szöveg, képernyőkép, Grafikai szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.1.2.1. ábra. A Godot játékmotor visual scripting rendszere

A Godot nagy és aktív közösséggel rendelkezik, amely rengeteg forrást, oktatóanyagot és fórumot biztosít a tanuláshoz és problémamegoldáshoz. A Godot felhasználóbarátnak számít, különösen a kezdők számára, köszönhetően a könnyű kialakításának, a különböző hardvereken nyújtott hatékony teljesítményének és az aktív közösségnek, amely folyamatosan hozzájárul a fejlesztéséhez.

### 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 nyelvének 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.

Aki nem ért annyira a programozáshoz, annak sem kell csüggednie, 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-rendszereket 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.

A képen képernyőkép, diagram látható

Automatikusan generált leírás

3.1.3.1. ábra. A Unity Bolt névre hallgató visual scripting rendszere

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.

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 szinesítsük, valamint értékes fejlesztési időt takarítsunk meg.

A képen szöveg, képernyőkép, szoftver, Számítógépes ikon látható

Automatikusan generált leírás

3.1.3.2. ábra. A Unity Asset Store nevezetű boltja, ahonnan a játékunkhoz szerezhetjük be a megfelelő sprite-okat

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.

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

# Pályageneráló algoritmusok

Ebben a fejezetben a különböző generálási módszerekről fogok leírást adni.

## Perlin-zaj (Perlin noise)

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 „side-scroller” terep létrehozásához, vagy kézzel írt vonalak illúziójának megteremtésére.

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 alábbiakban néhány képet láthatunk a különböző méretű zajokról és néhány felhasználási módjukról futás közben:

|  |  |  |
| --- | --- | --- |
| Zaj dimenziószáma | A nyers zaj (szürkeárnyalatos) | Felhasználási mód |
| 1 | A képen 1D perlin-zaj látható | A képen kör, vázlat, minta látható  Automatikusan generált leírás  A zaj offsetként való használata kézzel írt vonalak létrehozásához. |
| 2 | A képen szürke, képernyőkép, természet látható  Automatikusan generált leírás | A képen minta, csomagolópapír, térkép, Színesség látható  Automatikusan generált leírás  A zajt szigetek létrehozásához is lehet használni |
| 3 | A képen vázlat, fekete-fehér, minta, táj látható  Automatikusan generált leírás | A képen térkép látható  Automatikusan generált leírás  Egy módosított Perlin-zaj implementációval dombok, völgyek és barlangok hozhatók létre |

Amint láthatjuk, a Perlin-zaj számos természetben előforduló jelenségre alkalmazható.

A Perlin-zaj gradiens zajgenerálási technikát alkalmaz, ami a pontok közötti természetesebb 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.

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.

A generált minták összetettségének fokozása érdekében az algoritmus gyakran alkalmaz rétegezési technikát, amely több "oktávnyi" zajt tartalmaz. Minden egyes oktáv külön frekvenciával és amplitúdóval működik, és amikor ezeket a rétegeket kombinálják, bonyolultabb és változatosabb mintákat hoznak létre. Az algoritmus állítható paramétereket 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.

A játékokban és a számítógépes grafikában való alkalmazásán túl a Perlin-zaj elterjedt más területeken is, mint például tudományos szimulációk készítése, ahol olyan természeti jelenségeket modellez, mint a felhőképződmények, vagy egy táj jellegzetességei.

## Celluláris automata (Cellular Automaton)

A celluláris automata, egy rácsalapú rendszerben működő számítási modell, amely egyszerűségében és összetettségében egyaránt lenyűgöző. Minden egyes sejt ezen a rácshálózaton 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.

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 játékostól 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:

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

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 színtiszta függvénye.) A szabályok ismételt alkalmazása további generációk létrehozásához folytatódik.

A képen tér, diagram, képernyőkép, szöveg látható

Automatikusan generált leírás

2.3.2.1. 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.

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.

A képen vázlat, rajz látható

Automatikusan generált leírás

2.3.2.2. ábra Az automata által generált minták

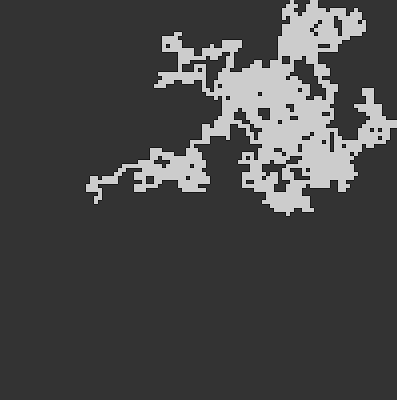
Ezzel a tulajdonságával hatékony eszközzé válik a fejlesztők számára, akik dinamikus és megnyerő környezetet kívánnak létrehozni a 2D platformer játékokban, és az egyedi, változatos szintek létrehozásával jelentősen növeli a játék újrajátszásának az esélyét.

## Véletlen bolyongás (Random Walk)

A Random Walk 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:

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

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 „Random Walk-ot” 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.



2.3.2.1. ábra A véletlen bolyongás algoritmussal generált összefüggő térkép

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

## A játék leírása

A szakdolgozatom során szerettem volna egy 2D-s platformer játékot megalkotni. A jaték 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 mechanika. Ezek a mozgások intuitívek és könnyen kezelhetők, miközben lehetőséget adnak a játékosoknak a pályák különböző kihívásainak megoldására.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

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

A játék több különböző szintből áll, amelyek mindegyike új kihívásokat és izgalmakat kínál a játékosoknak. A szintek száma és a szintek tervezése a játék fejlesztésének későbbi szakaszában lesz véglegesítve. A pályák kihívást jelentőek lesznek az akadályok elhelyezése révén. Ezek az akadályok különböző formákban és méretekben jelennek meg, és stratégiai gondolkodást igényelnek a játékosoktól a leküzdésükhöz. A játék egyik kiemelkedő funkciója a procedurális mapgenerálás, amely a főmenüben választható lesz. A játékosok kiválaszthatják a nehézségi szintet, amely hatással lesz a pályák összetettségére és a gyémántok mennyiségére a mapon.

## 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 3.3.1 ábrán láthatunk.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.1.3.1. 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 találhatóak az Asset-ek, amelyeket az Asset Store-ból installálhatunk, valamint megtalálhatóak még az általunk kreált C# szrkiptek, és a beimportált csomagok. Az Asset-ek a 2D-s vagy 3D-s modelleket, textúrákat, anyagjellemzőket tartalmazó fájlokat, háttereket, hangfájlokat 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 kizárólag azt jeleníti meg, amelyet a Scene ablakban található kamera objektum lát.

### A GameObject, valamint a szülő-gyerek kapcsolat

A GameObject a Unity egyik legalapvetőbb objektuma. Képviselhet karaktereket, kellékeket, díszletet, kamerákat, útpontokat és még sok mást. Lényegében egy GameObject minden olyan objektum, amely elhelyezhető a Scene képernyőn. Fontos megjegyezni, hogy maga a GameObject nem sok mindent csinál; a hozzá csatolt komponensek adják meg a viselkedését, megjelenését és a célját. Például egy "Renderer" komponens hozzáadása láthatóvá teszi a GameObjectet, míg egy "Collider" komponens hozzáadása lehetővé teszi, hogy hasson rá a Unity fizikai motorja.

A Unityben a GameObject-ek rendszerezése a scene-en belül hatékonyan egy szülő-gyermek 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 gyermekeire: 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 kerekei (gyermek GameObjectek) önállóan is foroghatnak, miközben az autó (szülő GameObject) részei.

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. Például a Collectibles szülő GameObject és annak a Diamond gyermek GameObject-ei a 3.3.1.1. ábrán látható.

A képen szöveg, Betűtípus, képernyőkép, tervezés látható

Automatikusan generált leírás

3.3.1.1. 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.

## Az irányítható hős és a kamera

Kezdetben be kell importálni az általunk kiválasztott assetet 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, előszőr a beimportált asset közül ki kell választani a karakterünk tétlen pozícióját reprezentáló Sprite-ot (Ide hivatkozás -> mi az a Sprite), majd egyszerűen a Scene ablakra kell húznunk, így létre is jön egy GameObject, amelynek tetszőleges nevet adhatunk. Az én esetemben 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. 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. 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 ugyanahhoz 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. A Fox objektum komponensei a 3.4.1. ábrán láthatóak.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.3.1.1. ábra. A Fox GameObject komponensei

### 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 3.4.1.1. ábrán látható.

A képen szöveg, képernyőkép, szám, dokumentum látható

Automatikusan generált leírás

3.4.1.1. 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 mindkét metódus szerepel a programban. Ezeket csak olyan parancsokhoz érdemes használni, amelyeket csak egyszer szeretnénk lefuttatni. Például akkor, amikor referenciákat 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 képkockafrissítésenként 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(horizontalValue,isJumping,crouchPressed) és a GroundCheck() metódusokat. Az Update() és a FixedUpdate() függvény a 3.4.1.2. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.4.1.2. ábra. Az Update() és a FixedUpdate() függvény

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” rétegben 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 ütközőjével. Ez az ellenőrzés a groundCheckRadius által meghatározott körön belül történik. Ezt a metódust egyenlővé tettem 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. A GroundCheck() metódus a 3.4.1.3. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.4.1.3. ábra. A GroundCheck() metódus

A Move(float dir, bool jumpFlag,bool crouchFlag) 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 (ceilingCheckCollider.position) rajzolt körrel, amelynek a sugara a ceilingCheckRadius változó. Ha az ellenőrzés során bármilyen ütközőt 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 ellenőrzi, 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 animációi 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, esetemben a szóköz vált ki), több művelet is történik. Az egyik az, hogy az isGrounded értékét hamisra állítom, ezzel jelezve, hogy a játékos ugrás 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 játékostól 1 helyett, mivel mindkét komponenssel ütközne az ellenfél. A Jumping&Crouching rész a 3.4.1.4. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.4.1.4. ábra. Az ugrást és a guggolást megvalósító kódrészlet

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 függetlenségét. 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 kezelni 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 paramtért 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. A Move&Run rész a 3.4.1.5 ábrán látható.

A képen szöveg, képernyőkép látható

Automatikusan generált leírás

3.4.1.5. ábra. A mozgást és a futást megvalósító kódrészlet

A Death() metódus a játékos halálát kezeli. Az isDead változó értékének az igazra állítása a játékost halottnak jelöli, hogy megakadályozzuk a további bemeneti feldolgozást. Ezután meghívók egy Restart() nevezetű metódust, amely a LevelManager osztályban található, hogy újraindítsam a pályát, ahol a játékos meghalt. A Death() metódus a 3.4.1.6. ábrán látható.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

3.4.1.6. ábra. A játékos halálát kezelő metódus

### 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 eléggé 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.

A képen diagram, sor, Diagram, lejtő látható

Automatikusan generált leírás

3.4.2.1. ábra. Az ugrás és a földet érés időtartama

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.

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 GameObjectből, 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 esését 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. A BetterJump osztály a 3.4.2.2. ábrán megtekintető.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.4.2.2. ábra. A BetterJump osztály

### 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, növelve 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. A MainCamera objektum, valamint a hozzácsatolt komponensek a 3.4.3.1. ábrán láthatóak.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.4.3.1. ábra. A MainCamera objektum és komponensei

Ha a hierarchiában ráhúzzuk 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.

Alább található a CameraFollow szkript, 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.

A Follow() metódus 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 pozícionálni. 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 biztosításához, hogy a kamera jó rálátást biztosítson az eseményekre, a karaktert a képen tartva, miközben elegendő részt mutat a környezetből.

A target pozíciójának 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 pozícióját 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. A Follow() metódus a 3.4.3.2. ábrán látható.

A képen szöveg, képernyőkép, Betűtípus, szoftver látható

Automatikusan generált leírás

3.4.3.2. ábra. A Follow() metódus

Miután elkészítettem a Follow() metódust, és beállítottam az első pályán a kamera határait, rájöttem, hogy eléggé időigényes, ezért elkezdtem keresgélni, 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 3.4.3.3. ábrán tekinthető meg.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.4.3.3. ábra. A MainCamera objektum általam létrehozott GUI része

Ezt úgy készítettem el, hogy egy feltételes fordítási blokkban (#if UNITY\_EDITOR) 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 nyílvános 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 írányíthatjuk 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 ezeket megvalósító kódrészlet a 3.4.3.4. ábrán láthatóak.

A képen szöveg, képernyőkép, szoftver, Operációs rendszer látható

Automatikusan generált leírás

3.4.3.4. ábra. A "None" állapotnál lévő gombokat és labeleket megvalósító kód.

Ezek az opciók azután jelennek meg, miután a grafikus felhasználói felület végigvitte a felhasználót egy struktúrált beállítási folyamaton. 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 pozícióját, és beállítja azt a minimális határértéknek (script.minValues).

A lépések a 3.4.3.5. ábrán láthatók.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

3.4.3.5. ábra A kamera minimum értékének a 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 a 3.4.3.6. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.4.3.6. ábra. A határértékek beállításáért felelős kód.

## 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. 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 controllert tartalmazó objektum példányosítása után a controller a be lépési állapottól egészen a kilépési állapotig fut az átmenetek mentén. Az állapotátmenetekkel megadhatjuk, hogy mely állapotokból mely állapotokba lehetséges a váltás.[8] 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 controllerben.

A Fox objektum animator controller-e a 3.5.1. ábrán látható.

A képen képernyőkép, Grafikai szoftver, Multimédiás szoftver, 3D modellezés látható

Automatikusan generált leírás

3.4.3.1. ábra. A Fox objektum animator controller-e

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 aktiválódhatnak, 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 fülnél lehet változókat definiálni, a Layers fül alatt pedig rétegeket létrehozni.

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 szisztémát követtem. A Moving nevű Blend Tree a 3.5.2. ábrán megtekinthető.

A képen képernyőkép, 3D modellezés, Grafikai szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.4.3.2. ábra. A Moving Blend Tree

A 3.5.2. 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.

## A szintek elkészítése

A következő lépésnek 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 szkriptelhető 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.

### 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.

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.

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 opciójánál 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 Tile Palette a 3.6.1.1. ábrán megtekinthető.

A képen képernyőkép, szöveg látható

Automatikusan generált leírás

3.6.1.1. 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 mindenképpen 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 3.6.1.2. ábrán látható.

A képen képernyőkép, pixel látható

Automatikusan generált leírás

3.6.1.2. ábra. A játék első szintje

### A RuleTiles használata

Az első szintet még a RuleTiles használata nélkül készítettem el. Sajnos elég monoton és nehézkes volt az elkészítése, mert mindig oda kellett figyelni, hogy a megfelelő tile-t használjam. Ezért keresgélni kezdtem 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-okon. 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.

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. Én 9 szabályt adtam hozzá, a piros X az éleket jelöli, míg a zöld nyíl azt jelenti, hogy a tile folyamatos. A szabályok a 3.6.2.1. ábrán láthatóak.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.6.2.1. ábra. A RuleTile szabályai

A szabályok beállítása után már kezdhetjük is a szintek könnyed és gyors elkészítését. A RuleTile segítségével elkészített alakzatok a 3.6.2.2. ábrán láthatóak.

A képen képernyőkép, sor, minta látható

Automatikusan generált leírás

3.6.2.2. ábra. A RuleTile használatával elkészített pédák

## 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 növelve a pálya nehézségét. A pályákat akkor teljesíti a játékos, ha kikerüli az ellenségeket, valamint felszedte az összes gyémántot. A gyémántok számát is növeltem a szintek nehezítése, és a hosszabb játékidő érdekében.

### Az EnemyAI osztály

Ezt az osztályt azért hoztam létre, hogy az ellenséges karaktert két meghatározott pont között oda-vissza mozgásra késztesse.A [RequireComponent(typeof(BoxCollider2D))] attribútum a szkript elején arra szolgál, hogy egy BoxCollider2D komponens automatikusan hozzáadódjon minden olyan GameObject-hez, amelyhez hozzácsatoljuk ezt a scriptet. 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 váltson 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 „járőrútjának” a kezdő és végpontját.

Ezt a szkriptet azért hoztam így létre, hogy ha újfajta ellenséget szeretnék a játékomba, akkor csak elég legyen hozzáadni az új ellenség objektumához. A Reset() és az Init() metódus a 3.7.1.1 ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.7.1.1. ábra. A Reset() és az Init() metódusok

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 iránya 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 elvégzek 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 végrehajtok egy ütközésérzékelést az OnTriggerEnter2D módszerrel. Ha az ellenség ütközik egy „Player” címkével ellátott GameObjecttel, akkor a Fox objektumon (mivel csak ez van ellátva „Player” tag-gel) életvesztést hajt végre a FindObjectOfType<LifeCount>().LoseLife() metódus meghívásával. A MoveToNextLocation() metódus és az ütközésrézékelést végrehajtó metódus a 3.7.1.2. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.7.1.2. ábra. A MoveToNextLocation() és az OnTriggerEnter2D() metódus

### 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 paranccsal 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. A LifeCount osztály a 3.7.2.1 ábrán tekinthető meg.

**A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás**

3.7.2.1. ábra. A LifeCount osztály

### 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 objektum gyermekobjektuma, 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 észlelje.

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 felvevésnek hangja is van, ezért létrehoztam egy SoundManager típusú audioManager változót, amely egy hivatkozás a hangkezelőre, amely a felvevés hangot játsza le.

Az Awake() metódusom inicializálja az audioManager-t az „Audio” címkével ellátott 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. A változók, az Awake() és az Update() metódus a 3.7.3.1. ábrán láthatóak.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.7.3.1. ábra. Az InteractionSystem változói, és néhány metódusa

A DetectObject() metódus a Physics2D.OverlapCircle-t használja a detectionPoint körüli detectionRadiuson belüli ütközők ellenőrzésére. Ezeket az ütközőket a detectionLayer alapján szűri ki. Ha egy ütközőt észlel, a detectedObject-et 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. A DetectObject() és a PickUpItem() metódus a 3.7.3.2. ábrán látható.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.7.3.2. ábra. A DetectObject() és a PickUpItem() metódus

Az Item szkript a játékban lévő tárgyakkal való interakciók kezelésére szolgál, és a felvehető tárgyakhoz nyújt funkcionalitást. 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 paranccsal.

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, de a későbbiekben ez is definiálva lesz.

Az Item szkript a 3.7.3.3 ábrán látható.

A képen szöveg, képernyőkép látható

Automatikusan generált leírás

3.7.3.3. ábra. Az Item szkript

## 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 pozícióját 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 GameObject.FindGameObjectWithTag(„Gem”).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 a FindObjectOfType<PlayerMovement>().transform.position = playerInitialPos parancs segítségével.

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.

Található még a szkripten két metódus, a LoadMainMenu() és a LoadNextLevel() amelyek a szint befejézést jelző panel gombjainak a metódusai. A LoadMainMenu()-vel értelemszerűen a főmenüt töltjük be, a LoadNextLevel() metódussal pedig a következő szintet. A LevelManager osztály a 3.8.1. ábrán látható.

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.7.3.1. ábra. A LevelManager osztály

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, min 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 elengedhetetlen a folyamatos hanglejátszás (például a háttérzene) fenntartásához a szintek közötti átmentek során.

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. A SoundManager szkript a 3.8.2. ábrán látható.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.7.3.2. ábra. A SoundManager osztály és metódusai

## A felhasználói felület elkészítése

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

### 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 rámegyünk bármelyik Button objektumra, akkor az Inspector ablaknál van egy olyan opció, hogy OnClick(). Igazából ez egy metódus, és itt tudjuk beállítani, hogy hogy viselkedjen a gomb, ha az ember rákattint. Az OnClick() opció a 3.9.1.1 ábrán látható.

A képen szöveg, képernyőkép, elektronika látható

Automatikusan generált leírás

3.9.1.1. 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. Ez a parancs a LoadLevelPicker()-nél a SceneManager.LoadSceneAsync(„Level Picker”), a LoadOptionsMenu()-nél pedig 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 fusson, 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. A MainMenu szkript a 3.9.1.2. ábrán, a főmenü pedig a 3.9.1.3. ábán látható.

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Automatikusan generált leírás

3.9.1.2. ábra. A MainMenu osztály és metódusai

A képen szöveg, képernyőkép látható

Automatikusan generált leírás

3.9.1.3. ábra. A játék főmenüje

### A Level Picker menü

A Level Picker menü elkészítésekor ugyan úgy jártam el, mint a főmenü elkészítésekor, annyi különbséggel, hogy itt nem három, hanem hét 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 3.9.2.1. ábrán, a LevelPicker osztály pedig a 3.9.2.2. ábrán látható.

A képen képernyőkép, szöveg, Grafika, Színesség látható

Automatikusan generált leírás

3.9.2.1. ábra. A játék LevelPicker menüje

A képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.9.2.2. ábra. A LevelPicker osztály és metódusai

### 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 3.9.3.1. 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.

A képen szöveg, Betűtípus, képernyőkép, fehér látható

Automatikusan generált leírás

3.9.3.1. ábra. A Slider UI objektum

Ahhoz, hogy a játékos tudja állítani a háttérzene és a hangeffektek hangerejét, készítenem 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 tulajdonságaik dinamikus beállításával szabályozzák a játékuk hangkörnyezetét. Én főleg a hangforrások csoportosításra használtam az AudioMixer-t, hogy kategorizáljam a háttérzene valamint a hangeffekteket. A 3.9.3.2. ábrán látható AudioMixer-ben 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 képen képernyőkép, Multimédiás szoftver, Grafikai szoftver, szoftver látható

Automatikusan generált leírás

3.9.3.2. ábra. Az AudioMixer eszköz

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

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 hangszintérzékelését. Mindkét módszer elmenti az aktuális slider értékeket a PlayerPrefs-be, így a beállítások tartósak maradnak majd.

A LoadVolume() metódus a PlayerPrefs-ből lekérdezi a mentett hangerőbeállításokat, és ennek megfelelően frissíti a csúszkák pozícióit. Meghívja a SetBackgroundVolume() és a SetSFXVolume() metódusokat, hogy a hangkeverő beállításai frissüljenek a betöltött értékeknek megfelelően.A 3.9.3.3 ábrán látható 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 képen szöveg, képernyőkép, szoftver látható

Automatikusan generált leírás

3.9.3.3. ábra A VolumeSettings osztály és metódusai

A képen szöveg, képernyőkép látható

Automatikusan generált leírásA képen szöveg, képernyőkép látható

Automatikusan generált leírásA 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 főmenüjet, a másik a LoadHelpMenu(), amely a Canvas objektumban lévő HelpMenu objektumot hozza elő, valamint az OptionsMenu objektumot eltünteti. A harmadik metódus a BackToOptions(), amely eltüneti a HelpMenu objektumot, és előhozza az OptionsMenu objektumot. Az OptionsMenu objektum a 3.9.3.4. ábrán, a HelpMenu objektum a 3.9.3.5. ábrán, az OptionsMenu szkript pedig a 3.9.3.6 ábrán látható.

3.9.3.4. ábra. A HelpMenu UI objektum

3.9.3.5. ábra. Az OptionsMenu UI objektum

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

3.9.3.6. ábra. Az OptionsMenu szkript és metódusai

### A szünet-, valamint a játék végét jelző menü