# 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 pályageneráló algoritmusok jellemzői

## A fejezet célja

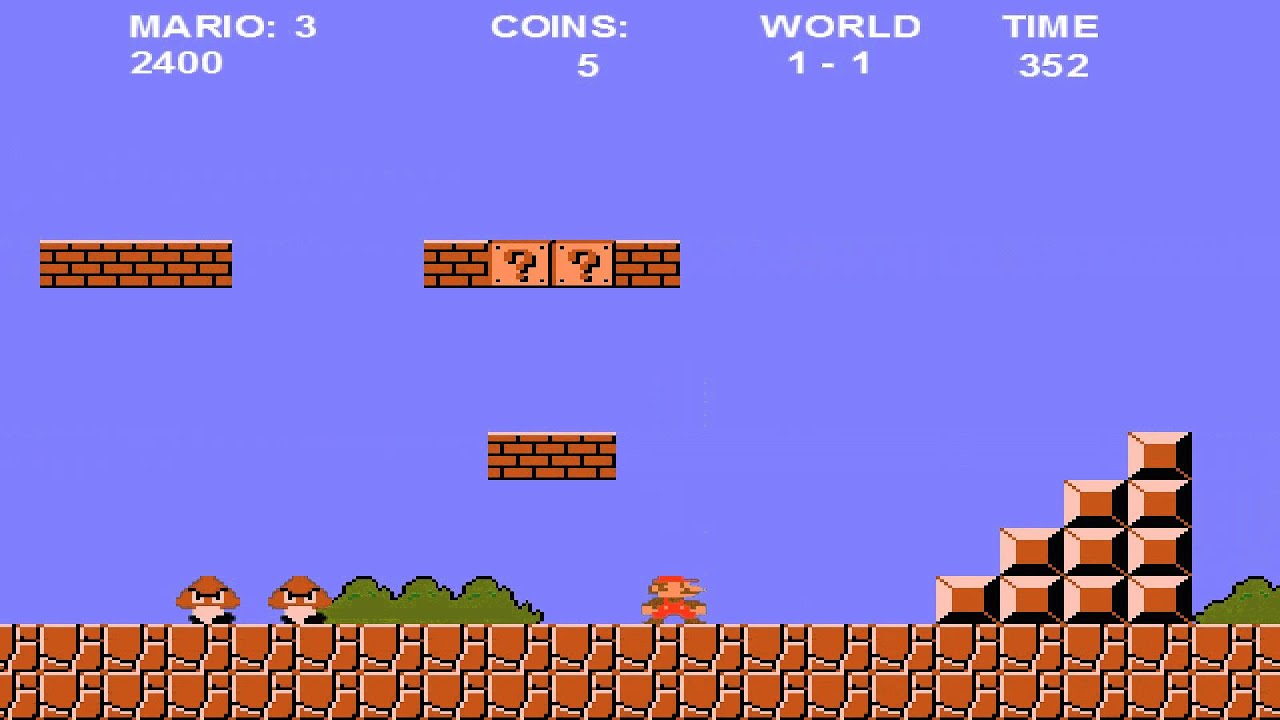
Ebben a fejezetben arra fogok törekedni, hogy részletesen bemutassam az oldalnézetes (side-scroller) játékok és a pályageneráló algoritmusok világát. Megvizsgálom az oldalnézetes játékok történelmi fejlődését, általános jellemzőit, és azt, hogy hogyan kapcsolódnak ezek a játéktípusok a pályageneráláshoz.

Ezen felül részletezni fogom a pályageneráló algoritmusok különböző típusait, módszereit, 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. 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.

2.2.2. 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.

2.2.3. 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.

2.2.4. 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.

2.2.5. 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.

2.2.6. 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.

## 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

2.3.1. 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

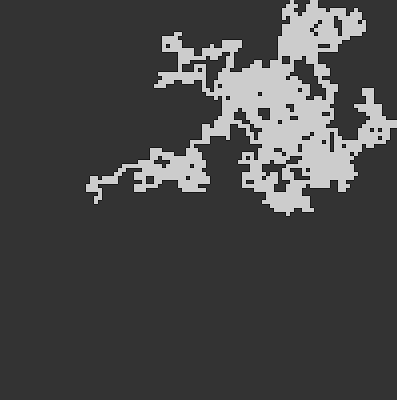
2.3.2. ábra Az automata által generált minták

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.

2.4.1. ábra A véletlen bolyongás algoritmussal generált összefüggő térkép

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

## 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. 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. 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. 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

3.1.4. 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.

## 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. 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

3.2.1. 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

3.3.1. 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

3.3.2. 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

3.4.1. 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

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 FixedUpdate() függvényben hívom meg a Move() és a GroundCheck() metódusokat.

A továbbiakban a Hero osztály metódusait fogom részletezni. 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 velocity tulajdonságnak megadhatunk egy vektort, ami mentén a Unity áthelyezi a karaktert. Ha ezt mondjuk egy általunk kreált metódusban hajtjuk végre és meghívjuk a FixedUpdate() metódusban, akkor határozott időközönként végrehajtódik. Azonban, ha ezt a kódot egy lassabb és egy gyorsabb gépen is lefuttatnánk, akkor az eredmény különböző lenne, ezért létrehoztam egy xValue változót, amelynek az értékét megszoroztam a speed változóval, majd a Time.fixedDeltaTime változóval is, amely a képkockafrissítések között eltelt időt tartalmazza, majd ezt a változót használtam a targetVelocity vektorváltozó X értékének a beállításához és a myBody.velocity tulajdonságot egyenlővé tettem a targetVelocity változóval. Ezáltal a gyorsabb számítógép többször hívja meg a parancsot, de kisebb távolságra, így az eredmény megegyezik majd a két gép között.

A localScale tulajdonságnak is megadhatunk egy vektort, aminek a segítségével elértem azt, hogy ha megnyomtam az A billentyűt, akkor a karakter balra nézzen és ha megnyomtam a D billentyűt, akkor a karakter jobbra nézzem. Ezt egy egyszerű feltétellel meg is tudtam oldani, ahol ha a facingRight változó igaz (tehát jobbra néztünk) és a direction kisebb mint nulla, akkor a localScale tulajdonságnak megadtam egy új vektort, ahol az X értéket -1-re, a facingRight változót pedig hamisra állítottam be, ezáltal a karakter balra fog nézni. Az else if ágban pedig ennek az ellenkezőjét csináltam ahhoz, hogy a karakter jobbra nézzen annak függvényében, hogy a direction értéke nagyobb mint nulla, valamint a facingRight változó pedig hamis (tehát balra nézünk).

A Jumping&Crouching részben