**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Programozási Nyelvek és Fordítóprogramok Tanszék**

**Kétdimenziós platformjáték C# és Unity alapon**

Szerző: Témavezető:

**Fodor Attila Krisztián** **Pataki Norbert**

Programtervező informatikus BSc. Habilitált egyetemi docens

**Budapest, 2025**

Tartalomjegyzék

[Köszönetnyilvánítás 1](#_Toc196974209)

[1. Bevezetés 2](#_Toc196974210)

[1.1. Motiváció 2](#_Toc196974211)

[1.2. Szakdolgozatom témája 2](#_Toc196974212)

[1.3. Készítés során felmerülő főbb akadályok 3](#_Toc196974213)

[2. Felhasználói dokumentáció 4](#_Toc196974214)

[2.1. Rendszerkövetelmények 4](#_Toc196974215)

[2.2. A játék telepítése és futtatása 4](#_Toc196974216)

[2.3. A játék ismertetése 5](#_Toc196974217)

[2.3.1. A játékmenet leírása 5](#_Toc196974218)

[2.3.2. Főmenü 5](#_Toc196974219)

[2.3.3. Beállítások 6](#_Toc196974220)

[2.3.4. Új játék 6](#_Toc196974221)

[2.3.5. Játékfelület 7](#_Toc196974222)

[2.3.6. Szüneteltetés menü 10](#_Toc196974223)

[2.3.7. Irányítás 11](#_Toc196974224)

[2.3.8. Pályaelemek 14](#_Toc196974225)

[2.3.9. Csontvázak 18](#_Toc196974226)

[2.3.10. Pályák 20](#_Toc196974227)

[2.3.11. Győzelmi képernyő 22](#_Toc196974228)

[3. Fejlesztői dokumentáció 24](#_Toc196974229)

[3.1. Használt eszközök és technológiák 24](#_Toc196974230)

[3.1.1. Unity Engine választásának indoklása 24](#_Toc196974231)

[3.1.2. C# programozási nyelv választásának indoklása 25](#_Toc196974232)

[3.1.3. A dolgozathoz használt fontos Unity fogalmak 25](#_Toc196974233)

[3.2. Felhasználói esetek 32](#_Toc196974234)

[3.3. A játék főbb komponensei 33](#_Toc196974235)

[3.3.1. Detekció és ütközéskezelés 33](#_Toc196974236)

[3.3.2. Harcrendszer 35](#_Toc196974237)

[3.3.3. Ellenfelek 37](#_Toc196974238)

[3.3.4. Játékos 39](#_Toc196974239)

[3.3.5. Erősítések 42](#_Toc196974240)

[3.3.6. Más felvehető objektumok 47](#_Toc196974241)

[3.3.7. Széttörhető tárgyak 48](#_Toc196974242)

[3.3.8. Checkpoint rendszer 49](#_Toc196974243)

[3.3.9. Perzisztenciarendszer 51](#_Toc196974244)

[3.4. Egyéb szkriptek 56](#_Toc196974245)

[3.4.1. Felhasználói felület 56](#_Toc196974246)

[3.4.2. Menük 58](#_Toc196974247)

[3.4.3. PersistentCamera és PersistentCanvas 59](#_Toc196974248)

[3.4.4. SceneController 59](#_Toc196974249)

[3.4.5. AudioManager 60](#_Toc196974250)

[3.5. A játék tesztelése 60](#_Toc196974251)

[3.5.1. Manuális tesztelés 60](#_Toc196974252)

[3.5.2. Automatizált tesztelés Unit Testekkel 60](#_Toc196974253)

[4. Összefoglalás 62](#_Toc196974254)

[5. Fejlesztési lehetőségek 63](#_Toc196974255)

[6. Melléklet és felhasznált források 64](#_Toc196974256)

[6.1. Felhasznált Unity csomagok 64](#_Toc196974257)

[6.2. Grafikus anyagok 65](#_Toc196974258)

[6.3. Hang és zenei anyagok 66](#_Toc196974259)

[6.4. Egyéb források 66](#_Toc196974260)

Ábrajegyzék

[1. ábra: Főmenü 5](#_Toc196974135)

[2. ábra: Beállítások 6](#_Toc196974136)

[3. ábra: Életerő-sáv 7](#_Toc196974137)

[4. ábra: Életek 7](#_Toc196974138)

[5. ábra: Erősítéstár 8](#_Toc196974139)

[6. ábra: Tűzlabda indikátor 8](#_Toc196974140)

[7. ábra: Kulcs indikátor 9](#_Toc196974141)

[8. ábra: Pontszámláló 9](#_Toc196974142)

[9. ábra: Csontváz számláló 10](#_Toc196974143)

[10. ábra: Törés számláló 10](#_Toc196974144)

[11. ábra: Szüneteltetés menü 11](#_Toc196974145)

[12. ábra: Mozgás 11](#_Toc196974146)

[13. ábra: Ugrás 12](#_Toc196974147)

[14. ábra: Közelharc 12](#_Toc196974148)

[15. ábra: Távolharc 13](#_Toc196974149)

[16. ábra: Kijárat 14](#_Toc196974150)

[17. ábra: Inaktív éledési pont 15](#_Toc196974151)

[18. ábra: Aktív éledési pont 15](#_Toc196974152)

[19. ábra: Tüskék 15](#_Toc196974153)

[20. ábra: Kis életerő 16](#_Toc196974154)

[21. ábra: Nagy életerő 16](#_Toc196974155)

[22. ábra: Kis sebességnövelő 16](#_Toc196974156)

[23. ábra: Nagy sebességnövelő 16](#_Toc196974157)

[24. ábra: Gravitációcsökkentő 16](#_Toc196974158)

[25. ábra: Doboz 17](#_Toc196974159)

[26. ábra: Szürke váza 17](#_Toc196974160)

[27. ábra: Barna váza 17](#_Toc196974161)

[28. ábra: Hordó 17](#_Toc196974162)

[29. ábra: Szív 18](#_Toc196974163)

[30. ábra: Kardforgató csontvázharcos 18](#_Toc196974164)

[31. ábra: Pörölyös csontvázharcos 19](#_Toc196974165)

[32. ábra: Távolsági csontvázharcos 19](#_Toc196974166)

[33. ábra: Első betöltés 20](#_Toc196974167)

[34. ábra: Dungeon pálya 21](#_Toc196974168)

[35. ábra: A Village szint bejárata 21](#_Toc196974169)

[36. ábra: Village platformok 22](#_Toc196974170)

[37. ábra: Győzelmi képernyő 23](#_Toc196974171)

[38. ábra: SceneView 26](#_Toc196974172)

[39. ábra: GameView 27](#_Toc196974173)

[40. ábra: Inspector komponens 29](#_Toc196974174)

[41. ábra: Animator komponens 30](#_Toc196974175)

[42. ábra: Tile Palette komponens 31](#_Toc196974176)

[43. ábra: Felhasználói eset diagram 33](#_Toc196974177)

[44. ábra: DetectionZone osztálydiagramja 34](#_Toc196974178)

[45. ábra: TouchingDirections osztálydiagramja 34](#_Toc196974179)

[46. ábra: Damageable osztálydiagramja 35](#_Toc196974180)

[47. ábra: Attack osztálydiagramja 35](#_Toc196974181)

[48. ábra: Projectile osztálydiagramja 36](#_Toc196974182)

[49. ábra: ProjectileLauncher 37](#_Toc196974183)

[50. ábra: Ellenfelek típusai 37](#_Toc196974184)

[51. ábra: Skeleton osztálydiagram 38](#_Toc196974185)

[52. ábra: PlayerController attribútumai 40](#_Toc196974186)

[53. ábra: PlayerController metódusai és függvényei 41](#_Toc196974187)

[54. ábra: PowerupEffect osztálydiagramja 42](#_Toc196974188)

[55. ábra: Erősítések típusai 43](#_Toc196974189)

[56. ábra: Powerup osztálydiagramja 43](#_Toc196974190)

[57. ábra: HealthBuff osztálydiagramja 44](#_Toc196974191)

[58. ábra: GravityBuff osztálydiagramja 44](#_Toc196974192)

[59. ábra: SpeedBuff osztálydiagramja 45](#_Toc196974193)

[60. ábra: LifeBuff osztálydiagram 45](#_Toc196974194)

[61. ábra: PowerupInventory osztálydiagramja 45](#_Toc196974195)

[62. ábra: PowerupSpawnManager 47](#_Toc196974196)

[*63. ábra: GetFireball osztálydiagramja* 47](#_Toc196974197)

[64. ábra: KeyPickup osztálydiagram 48](#_Toc196974198)

[65. ábra: LifeBuffPickup osztálydiagramja 48](#_Toc196974199)

[66. ábra: Széttörhető elemek típusai 49](#_Toc196974200)

[67. ábra: Breakable osztálydiagram 49](#_Toc196974201)

[68. ábra: Checkpoint osztálydiagramja 50](#_Toc196974202)

[69. ábra: CheckpointManager osztálydiagramja 50](#_Toc196974203)

[70. ábra: DataPersistenceManager osztálydiagram 51](#_Toc196974204)

[71. ábra: FileDataHandler osztálydiagram 53](#_Toc196974205)

[72. ábra: GameData osztálydiagram 54](#_Toc196974206)

[73. ábra: PowerupData osztálydiagramja 55](#_Toc196974207)

[74. ábra: Test Runner komponens 61](#_Toc196974208)

# Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani mindazoknak, akik hozzájárultak a *Dungeon Crawler* játék jelen verziójának megvalósításához.

Először is köszönettel tartozom a témavezetőmnek, Pataki Norbertnek, hogy betöltötte ezt a szerepet. A rendszeresen tartott konzultációink, a gyors és pontos visszajelzései, rugalmassága nélkül ez a dolgozat nem készült volna el.

Hálás vagyok édesapámnak, aki az egész fejlesztés alatt folyamatos támogatást nyújtott. Bár nincs otthon a programozás világában, mégis ösztönzött, noszogatott, biztatott. Érdeklődött és meghallgatott, amikor kétségbeesetten próbáltam elmagyarázni egy újabb hiba technikai oldalát és lehetséges forrásait. Megpróbálta megérteni, amit felvázolok, még akkor is, amikor a nagy része teljesen absztrakt és megfoghatatlan volt számára. A vele folytatott beszélgetések sokszor vezettek felfedezésekhez és megoldásokhoz, melyekre csak oly módon tehettem szert, hogy ténylegesen kimondtam, elmagyaráztam egy-egy felmerülő problémát.

Továbbá szeretném megköszönni a barátnőmnek, aki végtelen türelmével és biztatásával még akkor is támaszt nyújtott, amikor ő magának is szüksége lett volna rá.

Szeretném még köszönetemet kifejezni édesanyámnak és a nagyszüleimnek, akikre mindig számíthattam, amikor a dolgozat megszületésében kételkedtem. Szavaik hatására minden alkalommal megújult erővel folytattam a dolgozat írását.

Végül, de nem utolsó sorban szeretném megköszönni mind az egyetemen megismert, mind a régebbről megmaradó barátaimnak, akik nélkül elképzelhetetlennek tartom a szakdolgozatom megszületését. A tőlük kapott támogatás, odaadás és visszajelzés tette lehetővé, hogy a legnehezebb kihívásokkal is szembenézzek.

# Bevezetés

## Motiváció

A videójátékok mindig is fontos szerepet töltöttek be az életemben. Már gyerekkoromban is kikapcsolódást, menedéket jelentettek - egy módot arra, hogy eltereljem a figyelmemet a mindennapi problémákról. Bár ma már leginkább kompetitív, valós idejű játékokat kedvelek, a kezdetek platformjátékokhoz kötődnek: az első játékom egy oldalnézetes platformjáték volt, amit egy Playstation 2 konzolon játszottam. Akkor természetesen még nem foglalkoztatott, hogyan működik egy játékkonzol. Nem kérdőjeleztem meg, mi történik a „fedél alatt”, azt, hogy mi minden szükséges ahhoz, hogy a kontroller bizonyos gombnyomásaira azonnali vizuális visszajelzést kapjak a tévémtől. Nem gondolkodtam azon sem, hogy mennyi munka van egy ilyen játékélmény megteremtése mögött. Csupán élveztem, és megfeledkeztem minden másról - mintegy elvarázsolva. Ahogyan azonban telt az idő, egyre jobban érdekelni kezdett, hogy hogyan működnek ezek a világok a háttérben: hogyan áll össze egy játék logikája, hogyan lesz a sok apró darabból egy teljes, koherens kép. Ezért is választottam a témámat a játékfejlesztés világából.

## Szakdolgozatom témája

A szakdolgozatom célja egy kétdimenziós platformjáték fejlesztése, amelyben a játékos feladata a kiszabadulás. A platformjáték, vagy más néven „platformer*”*, egy olyan videójáték-műfaj, amelyben a játékos karakterének mozgása, különösen az ugrás és más vertikális mozgásformák, kiemelt szerepet kap. A műfaj jellegzetessége, hogy a játékos különböző szinteken, emelvényeken, akadályokon, tereptárgyakon - tehát platformokon - navigálva halad előre, gyakran időzítésre és ügyességre épülő kihívásokkal szembesülve. A platformjátékok gyakran kombinálják az ügyességi elemeket harci, felfedező vagy logikai mechanikákkal, ezáltal változatos játékmenetet kínálva. A dolgozatom témájaként szolgáló játék tehát ebbe a műfajba illeszkedik. A kiszabaduláshoz a játékosnak különböző pályákat kell felfedeznie, olyan mozgási mechanikákat felhasználva, mint például a futás, ugrás, illetve dupla ugrás, hiszen bizonyos pályaelemeket csupán ilyen módon képes elkerülni. A pályákon különféle ellenfelek találhatók, amelyek mind eltérő módon próbálják megállítani a játékost a kiszabadulásban. A felhasználónak a játék során erősítések (*power-up*) gyűjtésére is lehetősége van. Ezek keresése és használata nem kötelező, de erősen ajánlott, mivel ezek teszik lehetővé bizonyos pályaszekciók elérését, és az ellenfelekkel való küzdelemben is nagy segítséget nyújthatnak.

## Készítés során felmerülő főbb akadályok

Ez a projekt az első egyedülálló játékfejlesztési projektem, és számos kihívás elé állított. A Unity játékmotort korábban nem használtam, így a kezdeti lépések során több alapvető problémával szembesültem. Először is akadályt jelentett a motor felépítésének, működésének és logikájának megértése. A különböző scriptelési lehetőségek, valamint a játékmenet működését vezérlő eseményrendszer mind kidolgozatlan fogalmak voltak a fejemben, ezért jelentős időt kellett szánnom a megértésükre. Eleinte problémát jelentett a jelenetek közötti átmenetek és azok játékelemekre való hatásának megértése is, de számos más Unity-specifikus működést meg kellett tanulnom. Akadály volt még például az animációk készítése is, pontosabban az animációk közötti átmenetek feltételeinek finomhangolása és kezelése, amely mind játékmotoron, mind kódon belül megfontolt konfigurációt igényelt. A fejlesztés során előfordult, hogy egyes dolgokat a kezdeti, hiányos ismereteimből kiindulva nem megfelelő módon valósítottam meg, ami miatt később problémákba ütköztem. A dokumentációk, fórumok és oktatóanyagok használata ekkor elengedhetetlenné vált, főleg, amikor egy-egy ilyen hiba okát nehéz volt pontosan beazonosítani. Ám amikor a felmerülő problémát megvizsgáltam, komponensekre bontottam, majd a rendelkezésre álló segédanyagok segítségével eljutottam egy helyes megoldásra, óriási sikerélményben volt részem. Technikai nehézséget okozott viszont, hogy a félév nagy részében csak otthoni asztali gépen tudtam dolgozni, mivel a jelenlegi, fejlesztéshez alkalmas laptopomat csak a szemeszter végére sikerült beszereznem. Emiatt sokszor voltam a számítógéphez kötve, ami nemcsak fizikailag volt megterhelő, hanem a társas kapcsolataimra is hatással volt.

# Felhasználói dokumentáció

## Rendszerkövetelmények

A dolgozat elkészítéséhez a Unity 6000.0.37f1-es verzióját használtam, aminek az alábbiak a minimum rendszerkövetelményei:

|  |  |
| --- | --- |
| **Operációs rendszer** | Windows 10 21H1-es verzió (19043-as build) vagy újabb (64 bites), illetve Windows 11 21H2-es verzió (22000-as build) vagy újabb (Arm64) |
| **Processzor** | X64 architektúra SSE2 instrukciós szett támogatással, Arm64 |
| **Grafikus API** | DX10-, DX11-, DX12- vagy Vulkan-kompatibilis grafikus kártya |
| **További követelmények** | A hardvergyártó által hivatalosan támogatott illesztőprogramok szükségesek. |

## A játék telepítése és futtatása

A program használatához - a letöltött tömörített állomány kicsomagolása után - a „Build/DungeonCrawler.exe” futtatható állományt kell futtatni. Ezenkívül explicit telepítésre nincsen szükség.

## A játék ismertetése

### A játékmenet leírása

A játék egy oldalnézetes, kétdimenziós platformer, amely egy elfeledett föld alatti tömlöcben veszi kezdetét. Főhősünk a csontvázharcosok fogságába esett, és most a játékos segítségével próbál kiszabadulni. A cél minden pályán megtalálni a kijárathoz vezető kulcsot, leküzdeni az útba eső akadályokat és ellenfeleket, valamint felhasználni a különféle erősítéseket a továbbjutáshoz. A játék támogatja a mentést és visszatöltést is. Betöltéskor a játékos a legutóbb aktivált checkpointnál folytatja a kalandot, és az állapotok - például az életpontok, a próbálkozások száma, az összegyűjtött erősítések, valamint a megszerzett pontszám és legyőzött ellenfelek száma - mind megőrzésre kerülnek. Az interaktív pályaelemek, például a széttört objektumok vagy felvett tárgyak, az aktuális állapotuknak megfelelően jelennek meg újratöltés után is.

### Főmenü

A program futtatásakor megjelenő főmenüben (1. ábra: Főmenü) a játék neve alatt négy gomb jelenik meg. Ezek közül a legfelső *New Game* gomb lehetőséget kiválasztva a felhasználó új játékot indíthat a bevezető pályáról kezdve. A *Continue* gombra kattintva a játékos folytatja az elmentett játékmenetet a legutoljára aktivált checkpointnál. Az *Options* gomb megnyomása szintén az elvárt hatást éri el, átirányít a beállítások menüjébe. A *Quit* gomb használatával pedig a játékos bezárhatja a programot.



1. ábra: Főmenü

### Beállítások

A beállítások menüben (2. ábra: Beállítások) az *Options* felirat alatt három csúszka található. A legfelső, *Main* csúszka a játék általános hangerejét szabályozza, a második, *Music* csúszkával a háttérzene hangereje módosítható, míg a harmadik, *SFX* csúszkával a játékmenetben használt hangeffektek erőssége állítható be. A felhasználónak lehetősége van továbbá a kívánt felbontás és a teljes képernyős nézet konfigurálására. Amikor pedig a főmenüre szeretne visszatérni, a képernyő alján megtalálható *Back* gombot kell használnia. A beállítások indítások között is megmaradnak.



2. ábra: Beállítások

### Új játék

Eleinte mind a *New Game*, mind a *Continue* gomb hatására a *Tutorial* pályára kerülünk. Ez a rabul ejtésünkre szolgáló kazamatában található. Itt a pályán felfüggesztett szöveges utasítások segítségével a játékos megkezdheti a játékkal és az irányítással való ismerkedést. Először viszont kitérnék a játék felületére és működésére.

### Játékfelület

A játék felhasználói felülete egyszerű, letisztult és intuitív módon tájékoztatja a játékost, ikonokat és egyszerű szövegeket alkalmaz.

#### Életerő-sáv

A képernyő bal felső sarkában látható az életerő-sáv (3. ábra: Életerő-sáv), amely százalékos értékű, alatta pedig egy felirat jelzi a karakter aktuális életpontjait.



3. ábra: Életerő-sáv

A felirat alatt további, a játékállapot szempontjából elengedhetetlen információ-tartalommal rendelkező felületi elemek találhatók.

#### Életek

A hátralévő próbálkozások (4. ábra: Életek) számát három darab szív ikon jelzi, vagy betöltött, vagy betöltetlen állapotban. Ezek az ikonok jól láthatóak, így a játékos mindig pontosan tudja, mennyi hibalehetősége maradt.



4. ábra: Életek

#### Erősítéstár

A közvetlenül a szívektől jobbra látható erősítéstár (5. ábra: Erősítéstár) mutatja, hogy a játékosnak milyen erősítései milyen mennyiségben vannak eltárolva. A három különböző erősítésfajtának pontos működésére később kitérek, egyelőre viszont azt szeretném kiemelni, hogy bizonyos fajtákból létezik több alfajta (például: kis életerő/nagy életerő erősítés). Ezen alfajták játékosra gyakorolt hatása ugyanaz, csupán a hatás mértéke vagy időtartama változik.

A képen képernyőkép, rajzfilm, művészet látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

5. ábra: Erősítéstár

Az egyes alfajták más megjelenésűek, és bár hasonló ikonjuk van, mégis fontos megkülönböztetni őket. Ennek oka az, hogy amelyiknek az ikonját az erősítéstár éppen mutatja, az az alfajta lesz - a megfelelő gyorsgomb megnyomásakor - elhasználva. Az elhasználás azt az erősítést részesíti előnyben, amelyet a játékos hamarabb megszerzett, nem pedig amit legutóbb. Az erősítés típusok megjelenési sorrendje nem meghatározott, a felvétel sorrendjében jelennek meg a szívek után, balról jobbra sorakozva. Az egyes fajták ikonjain elhelyezkedő szám jelzi, hogy mennyi maradt még abból a típusból a játékos tárában.

#### Tűzlabda indikátor

Az elhasználható erősítések után helyezkedik el a tűzlabda erősítés indikátora (6. ábra: Tűzlabda indikátor), mely felvételre megjelenik, következő szintig pedig láthatóvá válik.

A képen pixel, képernyőkép, Színesség, tervezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

6. ábra: Tűzlabda indikátor

#### Kulcs indikátor

A kulcs indikátor (7. ábra: Kulcs indikátor) hasonló módon működik, ha egyszer azt a játékos megszerezte, a pálya erejéig aktívvá válik.

A képen képernyőkép, pixel látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

7. ábra: Kulcs indikátor

#### Pontszámláló

A játékban pontokat kell gyűjteni. Ezen pontok megjelenítéséhez mentések között perzisztens számlálók találhatók meg a képernyőn. A fent felsorolt játékfelületi elemek alatt, a képernyő bal oldalán jelenik meg a pontszámláló (8. ábra: Pontszámláló), mely a játékos által összegyűjtött összpontszámot mutatja. Ez minden legyőzött ellenfél vagy elpusztított pályaelem után frissül.



8. ábra: Pontszámláló

#### Csontváz számláló

A képernyő jobb felső sarkában fellelhető a csontváz számláló, amely minden legyőzött csontváz után növeli értékét. A számláló növekedésekor az összpontszám is változik. 100 pont jár egy kardforgató vagy pörölyös, 200 pedig egy távolsági csontvázharcos legyőzéséért. A 9. ábrán láthatjuk, hogy a felhasználó az új játék kezdete óta összesen 1 darab csontvázat tett el láb alól.



9. ábra: Csontváz számláló

#### Törés számláló

A 10. ábrán látható számláló közvetlenül a fenti alatt jelenik meg, minden széttört pályaelem után eggyel nő az értéke. Ez szintén felülírja az összesített pontszámot, elemenként 50 pontot jelent.



10. ábra: Törés számláló

A HUD (*head-up display*) minden eleme úgy van elhelyezve, hogy az ne zavarja meg a játékélményt, ugyanakkor minden fontos információ könnyen elérhető és gyorsan értelmezhető legyen a játékos számára. A dizájn célja az volt, hogy a felhasználó számára ne okozzon fennakadást a pályák felfedezése és az akció közbeni döntéshozatal.

### Szüneteltetés menü

A játék szüneteltetése elengedhetetlen része bármely játéknak, hiszen valamilyen módon „kiutat” kell biztosítani a felhasználónak - akár a főmenübe, akár az alkalmazás bezárásához. Ezt a funkciót tölti be a 11. ábrán látható menüpont, amelyet a felhasználó a játék közben az *Escape* billentyű lenyomásával érhet el. A megjelenő menü négy gombot tartalmaz: a játékmenet folytatását lehetővé tevő *Resume* gombot, a főmenüben már bemutatott beállításokat megnyitó *Options* gomb, a kezdőképernyőre visszalépést biztosító *Back to Menu* gombot, valamint a programot bezáró *Quit* gombot. A játék bármely módon történő elhagyásakor automatikus mentés történik.

A képen szöveg, képernyőkép, Multimédiás szoftver, áramkör látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

11. ábra: Szüneteltetés menü

### Irányítás

#### Mozgás

A játékos a karaktert az **A** vagy a **Balra nyíl** billentyű lenyomásával balra, illetve a **D** vagy a **Jobbra nyíl** billentyű lenyomásával jobbra tudja irányítani (12. ábra: Mozgás). A **Bal Shift** vagy **Jobb Shift** billentyű egyidejű lenyomva tartásával a karakter futni kezd az adott irányba.

A képen képernyőkép, szöveg látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

12. ábra: Mozgás

A **W**, **Felfelé nyíl** vagy **Space** billentyűk lenyomásával a játékos ugrást (13. ábra: Ugrás) hajthat végre. Amennyiben a karakter a levegőben tartózkodik, és ismételten lenyomja az ugrásra szolgáló billentyűk egyikét, egy második ugrást (dupla ugrást) hajt végre. Fontos megjegyezni, hogy platformról leesés után is csak egyetlen extra ugrás áll rendelkezésre.

A képen képernyőkép, szöveg, Multimédiás szoftver látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

13. ábra: Ugrás

#### Küzdelem

A közelharci támadás (14. ábra: Közelharc) a **Bal egérgomb** vagy az **O** billentyű lenyomásával aktiválható, amikor a játékos a földön áll. Az első ütés 10 sebzést okoz és kisebb visszalökődést (*knockback*-et) eredményez az ellenségen. Egy rövid időablakon belül végrehajtott második ütés 20 sebzést okoz, és valamivel nagyobb visszalökődést vált ki.

A képen képernyőkép, Videojáték-szoftver, Számítógépes játék, Digitális képszerkesztés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

14. ábra: Közelharc

A távolharci támadások (15. ábra: Távolharc) a **Jobb egérgomb** vagy a **P** billentyű lenyomásával indíthatók. Ha a játékos nem rendelkezik tűzlabdával, követ hajít el az aktuális nézési irányba, amely 10 sebzést okoz és kisebb visszalökődést eredményez. Amennyiben a játékosnál tűzlabda található, azt lövi ki, amely 20 sebzéssel és nagyobb visszalökő hatással jár. Mindkét támadást csak földön állva, az animáció ideje alatt helyben maradva lehet végrehajtani.

A képen képernyőkép, Digitális képszerkesztés, Videojáték-szoftver, Számítógépes játék látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

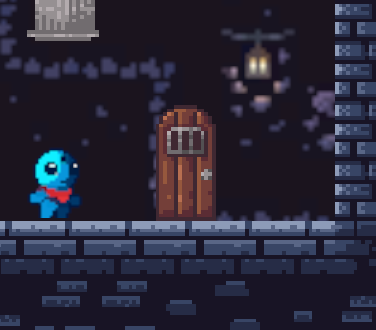
15. ábra: Távolharc

#### Erősítés használata

A játékos különböző eltárolt erősítéseket alkalmazhat az **1**, **2**, illetve **3** számgombok segítségével. Az **1-es** gombbal életerőt növelő (*HealthBuff*) erősítést aktiválhat, feltéve, hogy nem teljes az életereje. A **2-es** gombbal sebességnövelő (SpeedBuff), míg a **3-as** gombbal gravitációcsökkentő (*GravityBuff*) hatást idézhet elő, amely magasabb ugrásokat tesz lehetővé. Fontos figyelmeztetés, hogy az újonnan használt erősítés az előző azonos típusú erősítés hatását azonnal felülírja, így a felelőtlen használat az erősítések gyors elpazarlásához vezethet. Figyelni kell még arra is, hogy a fenti számgombok mindig a megadott erősítéstípusból használják fel a soron következőt, függetlenül az erősítéstípusok *user interface*-en megjelenő sorrendjétől.

#### Pálya elhagyása

Amennyiben a játékos megszerezte a pályán elrejtett kulcsot, a kijárat közelében (16. ábra: Kijárat) az **E** billentyű megnyomásával átléphet a következő szintre.



16. ábra: Kijárat

Ekkor a képernyő lassan elsötétedik, amikor pedig az új pálya betöltődött, fokozatosan újra láthatóvá válik. A pályaelhagyás megtartja a játékos minden adatát, de a tűzlabda erősítést és az előzőleg megtalált kulcsot elveszíti, hiszen ezek az előző szint végéig érvényesek elemek.

### Pályaelemek

#### Platformok

A játék - műfajának nevéből adódóan - platformokon játszódik. A platformok a pályák felépítésének és a játékmenet dinamikájának alapvető építőkövei. Különböző típusú felületek lehetnek: egybefüggő szilárd pályaelemek, lebegő blokkok vagy kisebb-nagyobb emelvények. A játékos ezekre a platformokra képes felugrani, rajtuk mozogni, illetve róluk tovább ugrani. Ezeken fognak elhelyezkedni a széttörhető tárgyak és járőrözni az ellenséges csontvázak.

#### Éledési pontok

Minden pályán megtalálhatók jellegzetes kinézetű éledési pontok (17. ábra: Inaktív éledési pont), amelyek azt a célt szolgálják, hogy a játékos életvesztés esetén ne legyen kénytelen a szintet teljes egészében újrakezdeni, vagy ismételten végighaladni a már teljesített pályaszakaszokon. Ezek a pontok megjelenésükben egyértelműen eltérnek a környezettől, így könnyen felismerhetők. Minden éledési ponthoz tartozik egy prioritási érték: minél magasabb ez az érték, annál közelebb helyezkedik el a pálya kijáratához.



17. ábra: Inaktív éledési pont

Amikor a játékos először megközelít egy éledési pontot, az aktiválódik (18. ábra: Aktív éledési pont), arany színnel jelezve, hogy ettől kezdve a játékos ezen a ponton éled újra életvesztés után. Amennyiben a játékos egy már aktivált, vagy a jelenleginél alacsonyabb prioritással rendelkező checkpoint közelébe érkezik, nem történik változás, az aktuális éledési pont marad érvényben.



18. ábra: Aktív éledési pont

#### Csapdák

A pályákon elszórtan a platformokra telepített csapdák (19. ábra: Tüskék) találhatók. A játékosnak mindenképpen el kell kerülnie ezeket az akadályokat, ugyanis érintkezéskor a játékos - függetlenül aktuális életerőpontjaitól - azonnal elveszít egy életet.

A képen képernyőkép, Videojáték-szoftver, Digitális képszerkesztés, 3D modellezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

19. ábra: Tüskék

#### Ideiglenes erősítések

A játékban jelenleg három különböző ideiglenes erősítéstípussal találkozhatunk.

1. **Életerő erősítések (*HealthBuff*):**

* **Kicsi (Small):**
* legfeljebb 25 életerőpontot tölt vissza

20. ábra: Kis életerő

A képen rajzfilm, clipart, illusztráció látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* **Nagy (Large):**
* legfeljebb 50 életerőpontot tölt vissza

21. ábra: Nagy életerő

1. **Gyorsaság erősítések (SpeedBuff):**

* A képen labda látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Kicsi (Small):**
* Sebességnövelés mértéke: 1.2-szeres
* Időtartam: 10mp
* Elhasználáskor felülírja az aktuális gyorsaság erősítés időtartamát és mértékét

22. ábra: Kis sebességnövelő

* A képen művészet látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Nagy (Large)**
* Sebességnövelés mértéke: 1.4-szeres
* Időtartam: 5mp
* Elhasználáskor felülírja az aktuális gyorsaság erősítés időtartamát és mértékét

23. ábra: Nagy sebességnövelő

1. A képen rajzfilm, művészet látható

   Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Gravitációs erősítés (GravityBuff):**

* Csökkenti a gravitációs mező erejét, magasabb ugrást és lassabb zuhanást eredményezve
* Időtartam: 6 mp

24. ábra: Gravitációcsökkentő

* Elhasználáskor felülírja az aktuális gravitációs erősítés időtartamát

#### Széttörhető pályaelemek

A széttörhető pályaelemeket a játékos támadásai mellett a lövedékek is elpusztíthatják. Bármilyen sebzés esetén azonnal összetörnek, megsemmisülésük után pedig 50% eséllyel tartalmazhatják a rájuk specifikus erősítést. Az erősítésekhez hasonlóan több típusú pályaelem különböztethető meg, mindegyik más gyakorisággal. A tárgy gyakorisága a benne rejlő erősítés mérlegelt hasznától függ. Például egy kis sebességgyorsítás kevesebb játékhelyzetben lehet olyan hasznos, mint 50 garantált életerő.

1. *A képen képernyőkép, pixel, tér látható

   Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.***Doboz (Box)**

* Kis életerő
* Kis sebességnövelő
* Gyakoriság: Gyakori (*Common*)

25. ábra: Doboz

1. A képen képernyőkép, pixel látható

   Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Szürke váza (Grey vase)**

* Nagy sebességnövelő
* Gyakoriság: Nem gyakori (*Uncommon*)

26. ábra: Szürke váza

1. A képen képernyőkép, pixel, tervezés látható

   Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Barna váza (Brown vase)**

* Gravitációcsökkentő
* Gyakoriság: Ritka (*Rare*)

27. ábra: Barna váza

1. A képen képernyőkép, pixel, tér látható

   Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.**Hordó (Barrel)**

* Nagy életerő
* Gyakoriság: (*Epic*)

28. ábra: Hordó

#### A képen pixel, képernyőkép, Színesség látható Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.Felvehető szívek

Az utolsó említetlen pályaelem az életet adó szív (29. ábra: Szív). Ez általában nehezen megközelíthető helyen, bizonyos esetekben csak erősítés használatával kaparintható meg. Egy extra próbálkozás megmentheti a játékost a játék végétől, így keresésük erősen ajánlott.

29. ábra: Szív

### Csontvázak

#### Kardforgató

A kardforgató csontvázharcosok (30. ábra: Kardforgató csontvázharcos) gyorsan támadó és mozgó ellenfelek, akik pillanatok alatt a játékoshoz szökkennek, hogy aztán lesújtsanak rá pengéjükkel.

A képen koponya, rajzfilm, csont, képernyőkép látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* Gyors közelharci támadások, melyek találatkor sebeznek és visszalöknek
* Sebzés: 20
* Életerő: 150
* Támadás sebessége: nagyon gyors
* Ha a játékost észreveszi, futni kezd felé, fokozatosan gyorsulva.

30. ábra: Kardforgató csontvázharcos

#### Pörölyös

A pörölyös csontvázharcosok (31. ábra: Pörölyös csontvázharcos) kinézetük ellenére igen fürgék, támadásaik pedig mindenáron elkerülendőek. Ha a játékos a pálya egy fontos mérföldkövéhez ér, szinte biztos lehet benne, hogy találkozni fog eggyel.

A képen képernyőkép, rajzfilm látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

* Lassan végrehajtott közelharci támadások, melyek találatkor nagy sebzést és jelentős visszalökést okoznak
* Sebzés: 40
* Életerő: 250
* Támadás sebessége: igen lassú
* Ha a játékost észreveszi, futni kezd felé, fokozatosan gyorsulva

31. ábra: Pörölyös csontvázharcos

#### Távolsági

A távolsági csontvázharcosok (32. ábra: Távolsági csontvázharcos) lassabban mozognak, ám sűrűn indítanak távolsági támadásokat. Lövedékeik elkerülése különösen fontos, főleg a levegőben tartózkodó játékos számára. Ha például a játékos éppen a második ugrását használja, és eközben találatot kap, a visszalökés hatására könnyen elvétheti a célplatformot, és veszélyes felületre zuhanhat. A közelharc szintén körültekintést igényel: rosszul időzített támadás esetén az ellenfél elrúgja magától a játékost, ismét távolsági küzdelembe kényszerítve őt.

* A képen képernyőkép, koponya látható

  Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.Közepes időközönként folytatott távolsági támadások, melyek találatkor sebzést és visszalökést okoznak
* Lövedék sebzése: 10
* Lövedék sebessége: közepesen gyors
* Rúgás sebzése: 15
* Rúgás támadási ideje: nagyon gyors
* Életerő: 200
* Ha a játékost észreveszi, távolsági támadást indít ellene, majd gyorsabban mozog az irányába

32. ábra: Távolsági csontvázharcos

### Pályák

#### Tutorial

Új játék indításakor a játékos a bevezetőként szolgáló *Tutorial* pályán találja magát (33. ábra: Első betöltés). A pályán rögzített paneleken megjelenő instrukciók segítségével megismerkedhet a játék irányításával, a játékban szereplő különböző pályaelemekkel, erősítésekkel és ellenfelekkel.

A képen képernyőkép, szöveg, Digitális képszerkesztés, Multimédiás szoftver látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

33. ábra: Első betöltés

#### Dungeon

A bevezető pályáról kiszabadulva, a játékos még mindig a kazamaták mélyén elhelyezkedő *Dungeon* szintre jut (34. ábra: Dungeon pálya). Ezen a pályán már nagyobb kihívást jelentő akadályokkal kell szembenéznie, és a bevezető pályához képest sokkal nagyobb terület felfedezésének néz elébe. A szint elrendezése vertikális hangsúlyú, tehát a kiindulópont és a kijárat között jelentős magasságbeli különbség van. A játékos itt már minden megtanult mechanikát alkalmazhat, és alkalmaznia is kell.

A képen szöveg, képernyőkép, Számítógépes játék, Digitális képszerkesztés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

34. ábra: Dungeon pálya

#### Village

A kazamatákból kiszabadulva a játékos új környezetben találja magát (35. ábra: A Village szint bejárata). A föld alatt játszódó, rideg színezésű első szinthez képest a második szint színekkel teli és pozitívabb hangulatú.

35. ábra: A Village szint bejárata

A képen szöveg, képernyőkép, fa látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

A *Village* szint (36. ábra: Village platformok) egy elhagyatott faluban játszódik, melyet csontvázharcosok szálltak meg. A korábbi pályához képest lineárisabb felépítésű, hosszan elnyúló terület várja a játékost, nehezebb akadályokkal berendezve. A ritkábban elhelyezett éledési pontok, a keskenyebb platformok és hosszabb csapdazónák fokozott precizitást követelnek, számos hibára adva lehetőséget. Jelenleg a játékban mindössze két szint érhető el, ám a rendszer könnyedén bővíthető további pályákkal.

A képen képernyőkép, Videojáték-szoftver, Számítógépes játék, 3D modellezés látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

36. ábra: Village platformok

### Győzelmi képernyő

Az utolsó pálya teljesítése után a felhasználó a győzelmi képernyőre kerül (37. ábra: Győzelmi képernyő). Itt megtekintheti a játék során összegyűjtött pontszámot, valamint a legyőzött csontvázak és széttört pályaelemek számát. A képernyő alján egyetlen gomb található *Back to Menu* felirattal, amely visszavezet a játék főmenüjébe. Innen a játékos csak új játékot indíthat, mivel a jelenlegi verzióban nincs lehetőség pályaválasztásra; újrajátszáskor a szinteket kizárólag a bemutató szintről, lineáris sorrendben lehet végigvinni.

A képen szöveg, képernyőkép, felhő látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

37. ábra: Győzelmi képernyő

# Fejlesztői dokumentáció

## Használt eszközök és technológiák

### Unity Engine választásának indoklása

A játék fejlesztéséhez a Unity játékmotort választottam, mert jelenleg az egyik legelterjedtebb és legsokoldalúbb eszköz ezen a területen. A Unity különösen jól használható 2D-s és 3D-s projektekhez is, de mivel én egy 2D platformjátékot készítettem, számomra főleg a kétdimenziós fejlesztéshez kínált eszközei voltak igazán hasznosak. Például a *Tilemap* rendszerrel sokkal gyorsabban tudtam megépíteni a pályákat, az *Animator* segítségével pedig a karakterek és a különböző játékelemek animációját kezeltem egyszerűen.

A Unity másik nagy előnye, hogy multiplatform támogatással rendelkezik. Ez annyit jelent, hogy a játékot szinte különösebb változtatás nélkül ki lehet adni Windowsra, Mac-re, mobilokra vagy akár konzolokra is. Ráadásul már a fejlesztés elején is lehetőség van gyors prototípusokat készíteni, amit később könnyen lehet optimalizálni és véglegesíteni, hála a motor erős erőforrás-kezelési megoldásainak.

**Fő előnyök, amiket tapasztaltam:**

* **Multiplatform támogatás:** ugyanaz a projekt könnyedén futhat többféle eszközön.
* **Gyors fejlesztés:** a sok beépített eszköznek köszönhetően gyorsan lehet haladni a munkával, legyen szó pályatervezésről vagy játékmenet fejlesztésről.
* **Kifejezetten jó 2D támogatás:** a Unity erre is komoly hangsúlyt fektetett, ami egy platformjáték esetében óriási előny volt.

**Hátrányok:**

* Bár a Unity nagyon jó eszköz 2D játékok fejlesztésére, elsősorban 3D-re optimalizált motor, így a 2D-s játékok is „3D környezetben” futnak.
* Ez néha indokolatlanul magas erőforrás-használathoz vezethet (CPU, RAM, GPU), különösen mobilokon vagy gyengébb gépeken.

Mielőtt a Unity mellett döntöttem, megnéztem más lehetőségeket is**:**

* **Godot Engine:** Ez egy könnyen kezelhető, ingyenes motor, ami szintén jól működik 2D projektekhez. Viszont a kisebb közösség miatt nehezebb megoldásokat találni, ha valami problémába ütközünk.
* **Unreal Engine:** Technológiailag nagyon erős, főleg 3D-ben. De épp emiatt túlzás lett volna egy egyszerűbb 2D-s platformerhez, ráadásul a kezelése is jóval bonyolultabbnak tűnt számomra.

Összességében azért maradtam a Unitynél, mert egyszerre adott gyors fejlesztési lehetőséget, erős 2D támogatást és a multiplatform megjelenést is megkönnyítette. Ezek együtt olyan előnyök voltak, amik a projektemhez tökéletesen passzoltak.

### C# programozási nyelv választásának indoklása

A fejlesztés során C#-ot használtam, mivel a Unity alapból erre a nyelvre épül. A C# egy erőteljes, objektumorientált nyelv, ami jól illik a játékfejlesztéshez. A szintaxisa egyszerű, könnyen átlátható, ráadásul rengeteg hasznos könyvtárral és előre megírt megoldásokkal szolgál. A legnagyobb előnye az volt, hogy nagyon jól működik együtt a Unity motorral, így minden funkciót könnyedén elértem vele.

### A dolgozathoz használt fontos Unity fogalmak

A Unity-ben a fejlesztési folyamat egyik alapköve a **SceneView** és a **GameView** használata.

#### SceneView

A *SceneView* (38. ábra: SceneView) lényegében a "szerkesztői nézet" – itt tudjuk kézzel összeállítani a jeleneteket, elhelyezni az objektumokat, finomhangolni a világot. Olyan, mint egy műhelyasztal, ahol még szabadon pakolgathatunk bármit, mielőtt ténylegesen "élesbe" menne.

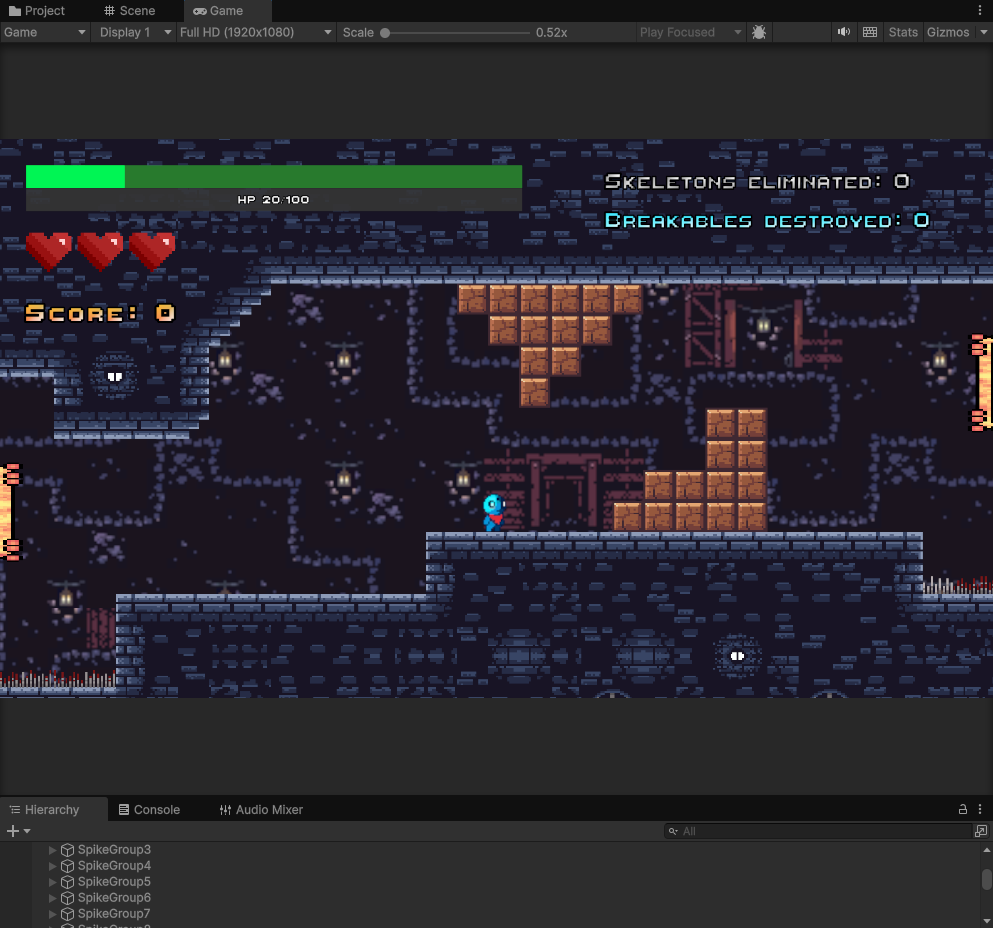
A képen képernyőkép, szöveg, Digitális képszerkesztés, Grafikai szoftver látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

38. ábra: SceneView

#### GameView

A 39. ábrán látható *GameView* („*Play* mód”) pedig az, ahol futtatni tudjuk a játékunkat közvetlenül a szerkesztőből. Itt látjuk, hogyan viselkednek az objektumok, hogyan működnek a scriptjeink éles környezetben, tehát valós idejű tesztelési lehetőséget ad. Fontos, hogy amit *Play* módban változtatunk (pl. mozgatunk egy objektumot), az nem marad meg, ha kilépünk a *Play* módból - ilyenkor a szerkesztő visszaáll az eredeti állapotba.



39. ábra: GameView

#### MonoBehaviour

A Unity-ben a legtöbb script alapja a **MonoBehaviour** osztály. Ez az, amit akkor használunk, amikor egy scriptet közvetlenül egy *GameObject*hez akarunk kötni. Például, ha egy karakter mozgását, egy platform viselkedését vagy egy ellenség mesterséges intelligenciáját akarjuk vezérelni, akkor *MonoBehaviour*-t öröklünk.

#### ScriptableObject

A **ScriptableObject** ezzel szemben egy kicsit más jellegű. *ScriptableObjectek*et akkor érdemes használni, amikor adatok tárolására van szükségünk függetlenül attól, hogy azok éppen melyik *GameObject*hez tartoznak. Például karakterstatisztikákat, fegyverek adatait vagy pálya konfigurációkat lehet így jól strukturálni. Nagy előnye, hogy memóriában hatékonyabb, és egyszerre több *GameObject* is hivatkozhat ugyanarra a *ScriptableObject* példányra.

Röviden:

* *MonoBehaviour* = "viselkedést" ad egy *GameObject*hez
* *ScriptableObject* = "adatot" ad függetlenül a jelenet objektumaitól

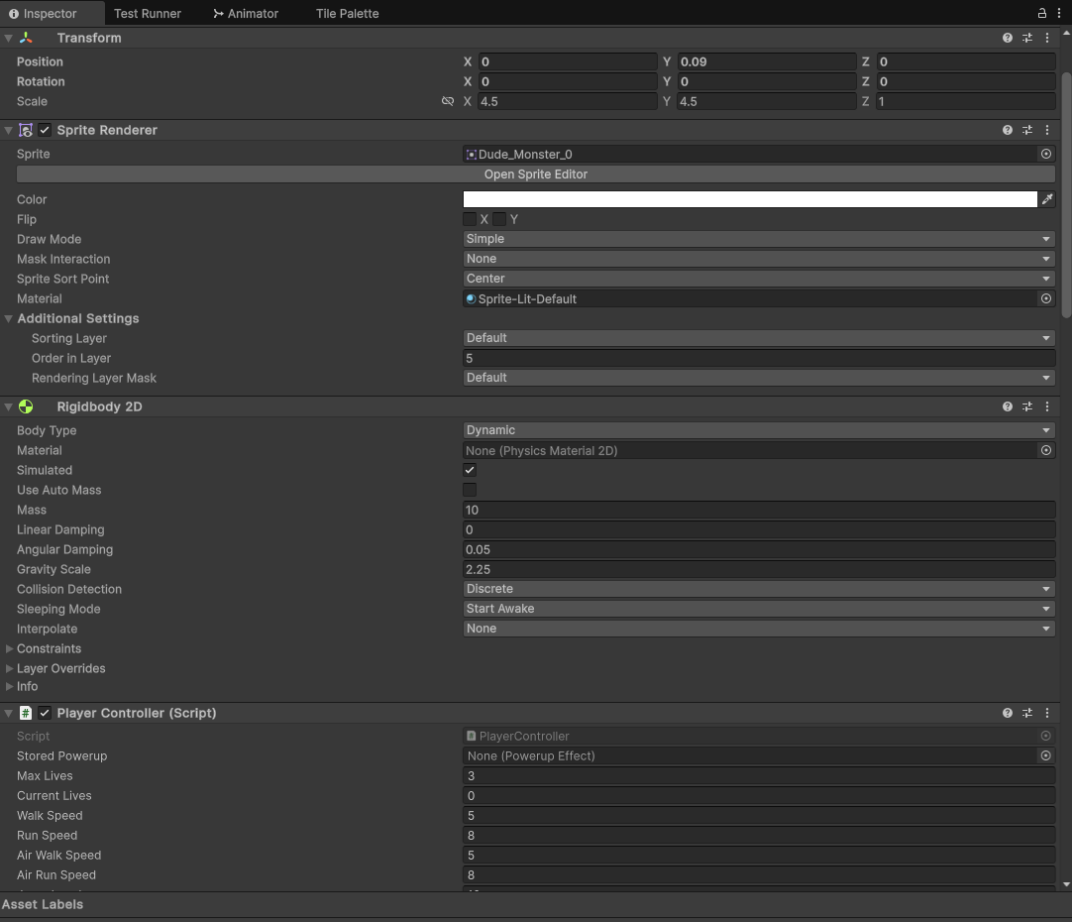
#### Alapvető metódusok

A Unity scriptjeiben többféle "életciklus" metódus is rendelkezésre áll, amikkel pontosan meg lehet határozni, mikor mit akarunk csinálni.

* **Awake()**: Akkor fut le, amikor az objektum a játék indításakor betöltődik. Itt kell beállítani a legfontosabb kezdeti értékeket.
* **Start()**: Ez már csak akkor hívódik meg, amikor az objektum aktívvá válik a játékban. Bizonyos kezdeti értékeket érdemes lehet csak itt beállítani az *Awake* helyett.
* **Update()**: Ez a játék minden egyes képkockájánál lefut. Ide kell tenni azokat a kódrészeket, amik folyamatosan figyelnek valamit, például a játékos inputját.
* **FixedUpdate()**: Ez hasonló az *Update*-hez, de nem a képkockák számához, hanem fix időlépésekhez igazodik. Főleg fizikával kapcsolatos számításokhoz ajánlott használni, mert itt megbízhatóbb az időzítés.

#### Inspector komponens

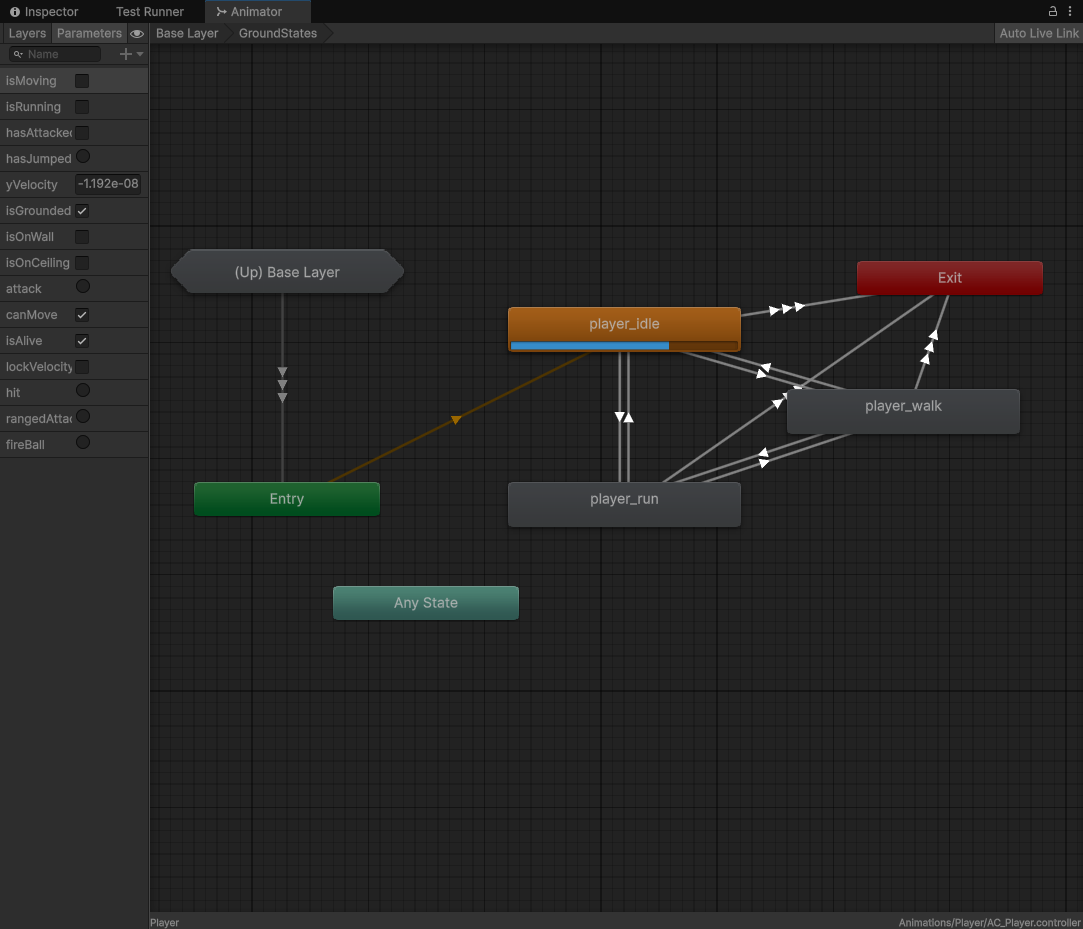
Az **Inspector** (40. ábra: Inspector menü) egy kulcsfontosságú szerkesztőpanel a Unity Editoron belül, amely lehetővé teszi az egyes *GameObjectek*hez rendelt komponensek és azok tulajdonságainak megtekintését és szerkesztését. Amikor kiválasztunk egy objektumot a Hierarchy nézetben, az Inspector panel automatikusan megjeleníti az ahhoz az objektumhoz kapcsolt összes komponenst - például a *Transform*ot, *Rigidbody*-t, *Collider*-t vagy bármely egyedi *MonoBehaviour* scriptet. *Az Inspector* segítségével tehát vizuálisan, kódírás nélkül módosíthatjuk egy objektum viselkedését és megjelenését.



40. ábra: Inspector komponens

#### Animator komponens

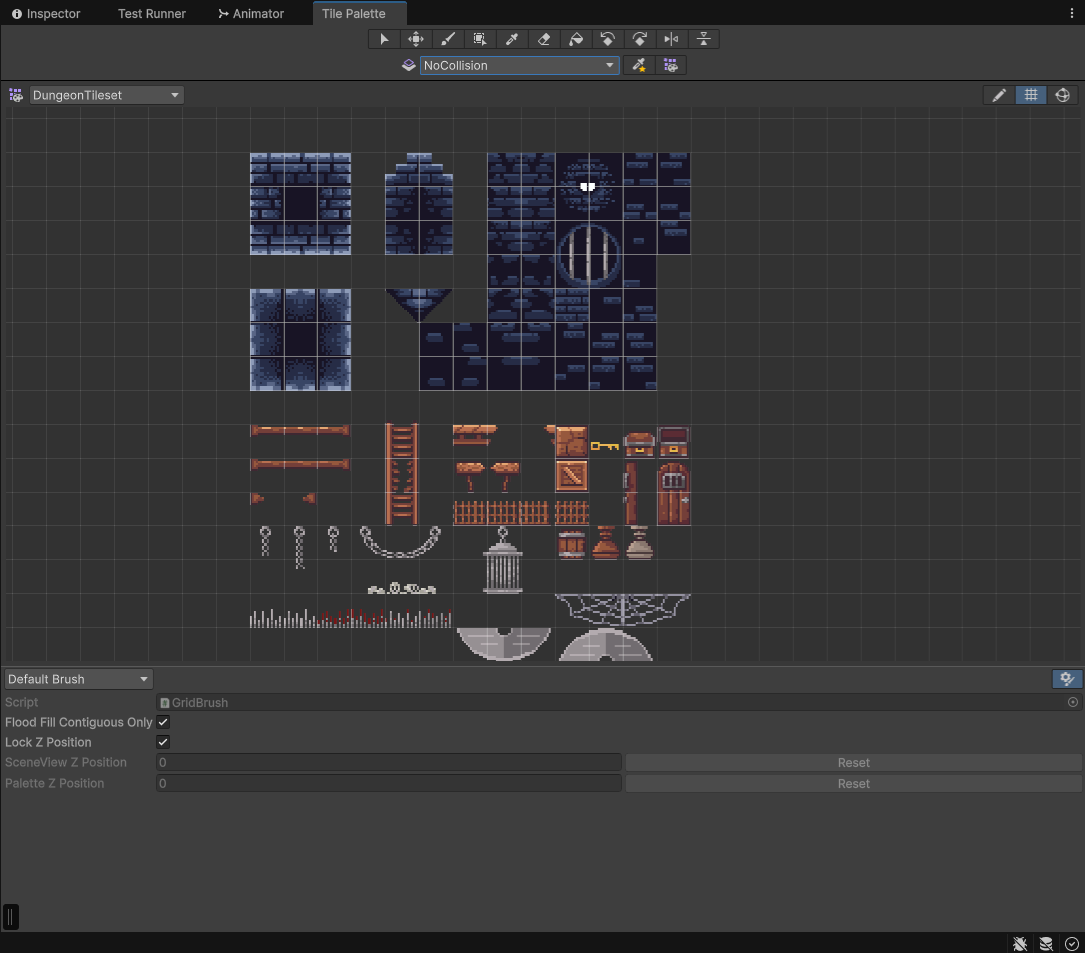
Az **Animator** (41. ábra: Animator komponens) a Unity animációs rendszerének központi része. Ezzel tudjuk irányítani a karakterek és más játékelemek animációit, például egy futás, ugrás, vagy támadás mozgásait. Az *Animator*ban állapotgépek használatával meghatározzuk, hogy az egyes animációk között milyen feltételekkel lehet váltani. Például, ha a játékos "futás" állapotban van, és megnyom egy gombot, akkor az *Animator* átválthat "ugrás" állapotba. A rendszer vizuálisan is jól áttekinthető: húzogatni lehet a kapcsolatokat az animációk között, és be lehet állítani különféle paramétereket (pl. sebesség, gombnyomás), amik alapján dönt, hogy mikor váltson.



41. ábra: Animator komponens

#### Tilemap és Tile Palette komponens

A **Tilemap** egy elképesztően hasznos eszköz, főleg 2D-s játékoknál. Gyakorlatilag egy rácshálóra tudunk kis csempéket (*tile*-öket) "rajzolni", amiből gyorsan és egyszerűen lehet pályákat összerakni. A *Tilemap* rendszer nagyon erős abban, hogy nagyobb, részletesebb pályákat is könnyen kezelhetővé tesz. Nem kell minden kis faldarabot, padlót vagy akadályt külön objektumként elhelyezni, elég egyszerűen a **Tile Palette**-ből (42. ábra: Tile Palette komponens) kiválasztani az adott csempét, és "kifesteni" vele a pályát. Ráadásul a *Tilemap*-hez könnyen hozzárendelhetünk *collider* rétegeket is, így a játékos vagy az ellenségek rögtön érezni fogják a világ határait anélkül, hogy kézzel kellene mindenhol egyesével *collider* komponenseket feltenni. Ennek hiánya a fejlesztés nehézségét hatványozta volna, így örömmel használtam ki a komponens nyújtotta lehetőségeket.



42. ábra: Tile Palette komponens

#### A Unity beépített fizikai rendszere

A Unity beépített fizikai rendszere (*Physics System*) gondoskodik arról, hogy a játékban szereplő objektumok "valóságosan" mozogjanak, ütközzenek, reagáljanak az erőhatásokra.

Két nagy része van: **2D Physics** (pl. *Rigidbody2D*, *Collider2D*) és **3D Physics** (pl. *Rigidbody*, *Collider*). A projekt során a *2D* *Physics* rendszer használatára volt szükségem.

Amikor egy objektumhoz *Rigidbody2D* komponenst adunk, akkor az bekerül a fizikai motor számításai közé, tehát például súlya lesz, leesik a ráható gravitációs erő miatt, vagy nekiütközik más dolgoknak (*collide*-ol). A physics számítások a *FixedUpdate* során frissülnek, hogy a fizikai interakciók stabilak és kiszámíthatóak legyenek. A *collider* komponensek pedig meghatározzák, hogy milyen formája és mérete van az objektumnak fizikai szempontból. Az egész rendszer célja az, hogy a fejlesztőnek ne kelljen kézzel kiszámolnia ütközéseket, gyorsulásokat, forgásokat - ezt a motor automatikusan megoldja, rengeteg időt megspórolva. Úgy gondolom, hogy az effektív programozás egyik legfontosabb eleme az a belátás, hogy vannak dolgok, amiket nem érdemes „újra feltalálni”. Véleményem szerint az alapvető fizikai hatások rendszere jelen projekt esetében tökéletes példának számít.

## Felhasználói esetek

A 43. ábrán látható a programhoz kapcsolódó felhasználói esetek diagramja. A program indításakor a felhasználó a főmenüből (*Main Menu*) indul. Innen lehetősége nyílik a beállítások (*Options*) menübe lépve személyre szabni a játék hangerejét, kiválaszthatja a felbontásra és teljes képernyős nézetre való preferenciáit. A program bezárásához viszont mindenképpen vissza kell lépnie a főmenübe. Amennyiben a felhasználó új játékot indít*,* minden elmentett játékadat - ha van - felülírásra kerül. Játék folytatásakor viszont a program az aktuális adatfájljában eltárolt adatok alapján tölti vissza a játékmenetet. Amennyiben nincs ilyen adatfájl - a játékos először játszik a játékkal, mégis megpróbál „folytatni” egy játékmenetet -, a program létrehoz egyet. Betöltés után a játékos a diagramon látható esetekkel találkozhat, ekkor már a játék összes funkciója elérhetővé válik számára. A szüneteltetés (*Pause*) menü pedig lehetőséget ad neki a játék folytatására, a beállítások szerkesztésére, a főmenübe való visszalépésre és a program bezárására is. Visszalépés és bezárás esetén is automatikus mentés történik, viszont, ha a játékos próbálkozásai elfogynak, a főmenüből már csak új játék indítható: a mentés ilyenkor törlődik.

A képen szöveg, képernyőkép, Betűtípus, diagram látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

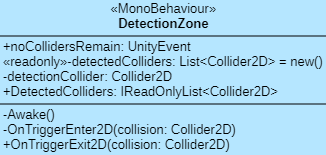
43. ábra: Felhasználói eset diagram

## A játék főbb komponensei

Ebben a pontban szeretném bemutatni a játékmenet alapvető komponenseit, funkcionális területekre bontva azokat.

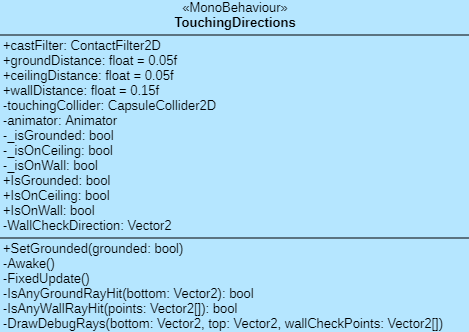
### Detekció és ütközéskezelés

A játékfejlesztésben a detekció és ütközéskezelés (*collision detection*) alapvető technikák, amelyek lehetővé teszik, hogy a játékbeli objektumok reagáljanak egymás közelségére vagy ütközésére. Ide tartozik például az, amikor egy karakter eléri a talajt, felvesz egy tárgyat, aktivál egy csapdát vagy belép egy adott zónába. Ezek az interakciók nemcsak a játékmenet kulcselemei, hanem az élmény és irányíthatóság szempontjából is kritikusak.

A **DetectionZon*e*** osztály (44. ábra: DetectionZone osztálydiagramja) egy láthatatlan trigger-zónát valósít meg, amely figyeli, hogy más *Collider2D* komponensek belépnek-e vagy elhagyják a zónát. Ez lehet például egy csapda aktiváló területe. A *detectedColliders* lista biztosítja, hogy mindig naprakészen tudjuk, mi tartózkodik a zónában, és ha az utolsó *collider* is távozik, egy esemény (*noCollidersRemain*) váltható ki. Az *UnityEvent* rendszer használata különösen hasznos, mivel lehetővé teszi a viselkedések testreszabását az *Inspector*ból újrakódolás nélkül.

44. ábra: DetectionZone osztálydiagramja

A **TouchingDirections** osztály (45. ábra: TouchingDirections osztálydiagramja) célja a karakter környezeti kapcsolatainak érzékelése: például, hogy a karakter talajon áll-e, érint-e falat vagy plafont. Ezek az információk elengedhetetlenek például az ugrás, futás vagy animációs logikák működtetéséhez. Az észlelést *Raycast* technikával végeztem, több pontból indítva a sugarakat, hogy stabilan detektálhassuk a környezeti elemeket. A falérzékelés is több szinten történik, így például lépcsők vagy egyenetlen felületek esetén is megbízható marad. Az osztály az *Animator* komponenssel is szorosan együttműködik: a detektált állapotokat közvetlenül szinkronizálja az animációs paraméterekkel, így tiszta, elkülönített felelősségű rendszer jön létre, ami jól példázza az egy felelősség elvét (*single responsibility principle*).

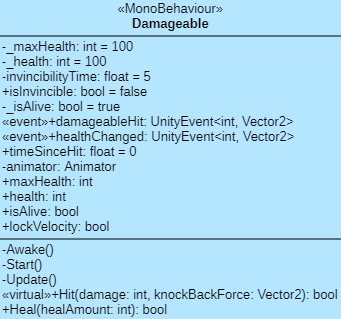


45. ábra: TouchingDirections osztálydiagramja

### Harcrendszer

A harcrendszer logikáját négy különálló, de szorosan együttműködő komponens valósítja meg. Ezeket a komponenseket úgy terveztem, hogy egyrészt jól elválaszthatók legyenek a karakter- és animációs logikától, másrészt könnyen bővíthetők legyenek új támadástípusokkal vagy interakciókkal*.*

#### Damageable

A harcrendszer egyik központi osztálya (46. ábra: Damageable osztálydiagramja). Ez felel az életerő nyilvántartásáért, a halál állapotának kezeléséért, valamint az ideiglenes sebezhetetlenség (*invincibility frame*) biztosításáért, amit sok akciójátékban alapelvként alkalmaznak. A *Hit* metódus az animációkat is elindítja (*hitTrigger*, *lockVelocity*), valamint eseményeket küld (*damageableHit*, *characterDamaged*) más rendszereknek - például a *UI*- vagy hangkezelésnek.

46. ábra: Damageable osztálydiagramja

Külön pozitívum, hogy a gyógyítás (*Heal*) is itt történik, és szintén eseményt generál, ami a rendszer modularitását erősíti. Mindez lehetővé teszi, hogy az életerőváltozásokra más komponensek is reagáljanak anélkül, hogy szorosan össze lennének kötve ezzel az osztállyal.

#### Attack

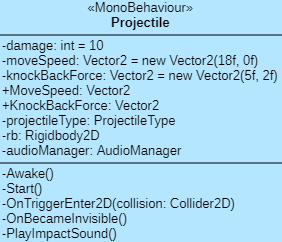
Ez az osztály (47. ábra: Attack osztálydiagramja) reprezentálja a játékos vagy ellenség közelharci támadásait. Amikor egy másik objektum trigger zónájába lép, az *OnTriggerEnter2D* esemény lefut, és megvizsgálja, hogy az objektum sebződő-e (*Damageable*), fizikai hatásnak kitehető-e (*Rigidbody2D*), illetve törhető-e (*Breakable*).

47. ábra: Attack osztálydiagramja

A kiemelkedő rész itt az, ahogyan a *knockback* irányát és erejét dinamikusan számítom, figyelembe véve az ütközés irányát. Ez lehetővé teszi, hogy a játékos mozgásának és pozíciójának tényleges kontextusa befolyásolja az eredményt - nem csupán egy fix értéket alkalmazok. A *knockback* hatást csak akkor alkalmazom, ha az objektum nem törhető, ezzel elkerülve felesleges fizikai számításokat például tárgyakon.

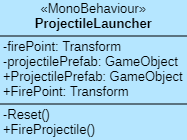
#### Projectile

A lövedékek kezelése külön objektumokon keresztül történik. A *Projectile* komponens (48. ábra: Projectile osztálydiagramja) felel a lövedék mozgásáért, becsapódási logikájáért és annak eldöntéséért, hogy történt-e találat. A lövedékek különféle típusokra oszthatók (***Rock***, ***Fireball***), amelyek vizuálisan és hanghatás alapján is megkülönböztethetők.

A becsapódás során a *Damageable.Hit* meghívásával történik a sebzés és *knockback* alkalmazása, majd a lövedék eltűnik. Ha nem talál célpontot, a képernyőről való kikerülés (*OnBecameInvisible*) szintén eltünteti - ezzel minimalizálva a felesleges objektumokat.

48. ábra: Projectile osztálydiagramja

#### ProjectileLauncher

Ez az osztály leegyszerűsíti a lövedékek kezelését azzal, hogy egy előre definiált pontból (*firePoint*) indítja azokat. Fontos elem, hogy a kilőtt objektum irányát dinamikusan beállítom a támadást végző karakter nézési iránya alapján (*transform.localScale.x*). Ez biztosítja, hogy ugyanaz a *prefab* jobb- vagy balirányban is helyesen viselkedjen anélkül, hogy két külön változatra lenne szükség.

49. ábra: ProjectileLauncher

Összességében ez a harcrendszer jól szervezett, robusztus alapot nyújt közelharci és távolsági támadásokhoz egyaránt. Egyik fő erőssége, hogy komponensalapú felépítése miatt minden elem külön tesztelhető, újrafelhasználható és bővíthető, akár új támadástípusokkal, új animációs válaszokkal vagy eltérő *AI*-reakciókkal.

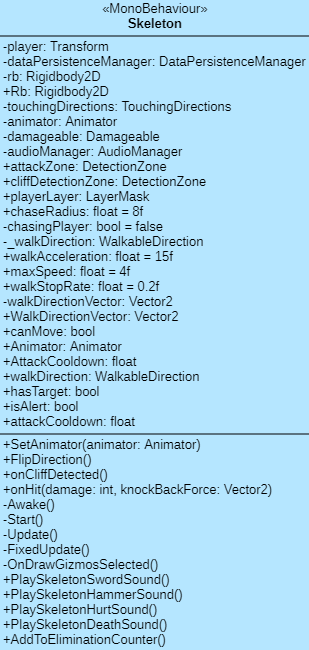
### Ellenfelek

A **Skeleton** ellenfél (50. ábra: Ellenfelek típusai) viselkedésének középpontjában az áll, hogy folyamatosan figyeli a környezetet és dinamikusan reagál a változásokra. Az osztály (51. ábra: Skeleton osztálydiagram) olyan mechanizmusokat implementál, amelyek lehetővé teszik a prediktív, intelligens viselkedést, például észleli, ha a játékos belép az érzékelési zónájába (*throwZone* és *kickZone*), valamint a szakadékok közelében is képes a viselkedését igazítani (*cliffDetectionZone*). Ez a mozgáslogika nem csupán egyszerű *patrol-AI*-t valósít meg, hanem adaptív, „okos” döntéseket hoz, figyelve a helyzetekre és reagálva azok változására.



50. ábra: Ellenfelek típusai

A mozgáslogika tehát úgy lett kialakítva, hogy a karakter képes „okosan” közlekedni a térben. Ha a karakter szakadékhoz ér, automatikusan irányt vált, nem esik le, és a játékos üldözése során is képes folyamatosan váltani az irányt, ha az adott irány már nem optimális, amennyiben a játékos hatótávolságán kívülre ér. Ezen kívül a karakter *sprite*-ja is folyamatosan tükröződik az aktuális mozgási irány alapján, mivel a *walkDirection* változó gondoskodik arról, hogy mindig a megfelelő irányba nézzen.



51. ábra: Skeleton osztálydiagram

A *FixedUpdate* ciklusban történik meg az effektív sebességfrissítés is. Itt figyelembe vesszük a talajérintést, a mozgásképességet (*canMove*) és az animációkat, így a karakter csak akkor mozdul el, ha az animátor engedélyezi. Ez segít megőrizni a mozgás és animáció közti szinkront, például nem mozdul meg, ha épp sebezhető állapotban van. Az *AI* viselkedése a *chasingPlayer* logika alapján skálázódik: ha a játékos a közelben van, a sebesség is növekszik, de a függőleges pozíciókülönbséget is figyelembe veszi, így a karakter nem reagál irreálisan egy magasban lévő játékosra. A mozgási sebesség is ennek megfelelően változik, ami kellemes fokozatos váltásokat eredményez a viselkedésben.

A hangkezelés is el van választva dedikált metódusokba (pl. *PlaySkeletonSwordSound*, *PlayRockThrowSound* stb.), így könnyen módosíthatóak vagy bővíthetőek, ha a későbbiekben hangvariánsokat vagy lokalizált hangkészleteket szeretnénk bevezetni. Ez a megközelítés segít a kód tisztán tartásában és újrahasználhatóságában.

Az ellenség eliminálása után a *DataPersistenceManager* segítségével frissítjük a statisztikákat, mint a teljes eliminációk száma és a játékos pontszáma. Ez hosszú távú motivációs hatással van a játékélményre, mivel a játékos látja, hogy minden egyes legyőzött ellenség hatással van a játékállásra.

A tesztelhetőség is fontos szerepet kapott a kódban, ami a *#if UNITY\_EDITOR || TEST\_MODE* alatti szakaszokban van megoldva. A külső függőségek (mint például a *TouchingDirections* vagy a *Rigidbody2D*) injektálásával a kód könnyen tesztelhetővé válik anélkül, hogy a Unity saját inicializálási ciklusaira kellene támaszkodni. Ez lehetővé teszi az egységtesztek könnyű írását, ami manapság egyre fontosabb a karbantarthatóság szempontjából.

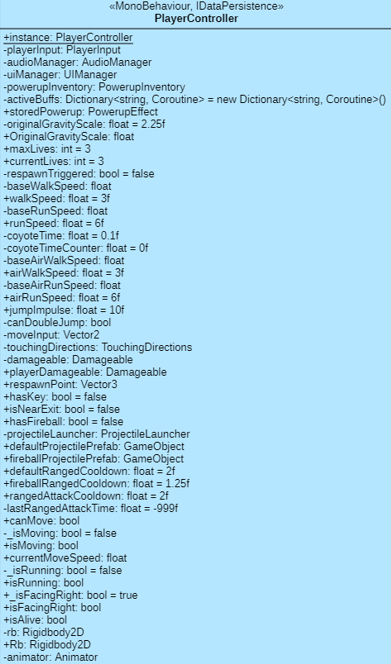
### Játékos

A **PlayerController** osztály (52. ábra: PlayerController attribútumai) a játékban szereplő legkomplexebb osztály, így talán ezt is az elejére venném. Az osztály számos fontos funkcióval (53. ábra: PlayerController metódusai és függvényei) rendelkezik, amelyek meghatározzák, hogyan reagál a játékos a játék világában végzett különböző tevékenységekre, például mozgásra, ugrásra, támadásra és tárgyak felvételére. Emellett ez az osztály felelős a játékos saját attribútumainak kezeléséért, beleértve életerőpontjait, élet-halál állapotát, az újraéledési pontját és a megszerzett erősítéseinek alkalmazását.

Az osztály elején található a *singleton* minta statikus *instance* változója, amely lehetővé teszi az osztály globális hozzáférhetőségét, emellett biztosítja, hogy csak egy példányban fusson a *PlayerController*. Ez a példány az egész játékban elérhető lesz, és ha már létezik, akkor az esetleges következő példányokat automatikusan törli, így elkerülve a potenciális hibákat.

Az *Awake* metódus az osztály inicializálásakor hívódik meg, és beállítja a szükséges komponenseket, mint például az *AudioManager*, *PlayerInput*, vagy az *Animator*.

A *playerInput* és az ahhoz kapcsolódó bemeneti események, mint például az *onMove()*, *onJump()*, *onRun()*, és a támadások (*ranged attack* és *melee attack*) mind az egyes inputok kezelését szolgálják. Az *onMove()* és *onRun()* események a karakter mozgását és futását



52. ábra: PlayerController attribútumai

biztosítják, míg az *onJump()* lehetővé teszi a játékos számára, hogy ugrásokat hajtson végre a talajon, illetve másodlagos ugrást végezzen, ha a *coyote time* még engedi. A mozgásról gondoskodnak a *currentMoveSpeed* és az *isRunning* változók, melyek segítenek meghatározni, hogy a játékos gyorsan vagy lassan mozogjon-e, miközben figyelembe veszik a különböző állapotokat, például a futást, vagy hogy a játékos a levegőben van-e.



53. ábra: PlayerController metódusai és függvényei

A mozgás közben a *touchingDirections* komponens segítségével az osztály kezeli a talajjal való érintkezést, a falhoz vagy a mennyezethez való viszonyt, valamint az ugrásokat, amelyek *coyote time* mechanizmussal vannak ellátva.

Egy másik kulcsfontosságú funkcionalitás a különböző támadások használata. Az *onAttack()*, *onRangedAttack()* és a különböző *OnUsePowerup()* metódusok felelősek az olyan tevékenységekért, mint a közelharci támadás, távolsági támadás és a *power-up*ok aktiválása. Az osztály gondoskodik arról is, hogy a különböző képességek, mint a sebesség- vagy gravitációváltoztatás ideiglenesen megváltoztassák a játékos állapotát, majd visszaállítja az eredeti értékeket, miután a hatás időtartama lejár.

Fontos programozói fogás, hogy az osztályban alkalmazott *IEnumerator* és *Coroutine* mechanizmusok segítségével képesek vagyunk időzített események kezelésére, például a sebesség vagy gravitáció ideiglenes növelésére, anélkül, hogy folyamatosan frissítenénk az értékeket. Ez az aszinkron működés különösen hasznos a játékos karakterének olyan tulajdonságainál, mint a mozgási sebesség vagy a gravitáció, amelyek különböző erősebb hatások alatt változnak.

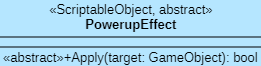
A *Damageable* komponens és a *respawn* mechanizmusok összekapcsolása lehetővé teszi, hogy a játékos élete nyomon legyen követve. Amikor a játékos meghal, egy új élet lehetőségét biztosítja, vagy akár a játék végét eredményezi - amennyiben nem maradt több próbálkozása. Az *onHit()* metódus a karakter sebzését és a visszavágó erőt kezeli, a *RespawnPlayer()* pedig a karakter újraélesztését és pozicionálását végzi el a legutolsó mentett ponton.

Az *IDataPersistence* interfész implementálása biztosítja, hogy az osztály képes legyen adatokat menteni és betölteni, például a játékos életeit, pontszámát, az összegyűjtött tárgyakat és az aktuális pályát. A mentés és betöltés folyamata a *LoadData()* és *SaveData()* metódusok segítségével történik, biztosítva ezzel, hogy a játékos haladása megfelelően tárolódjon el, és a mentett adatok alapján folytathassa a játékot.

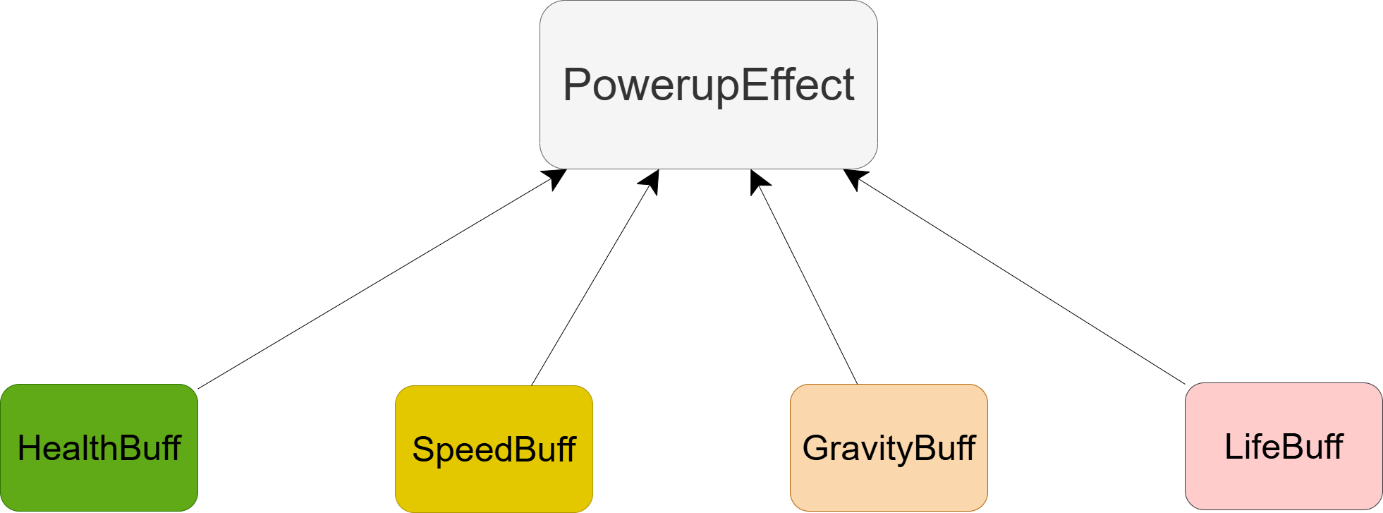
A kód fejlesztése során figyelembe kellett venni a játékos mozgásának és viselkedésének pontos kezelését, például az animációkat, az inputokat és a különböző állapotokat. A *Rigidbody2D* és az *Animator* komponensek összekapcsolása lehetővé tette a karakter fizikájának és vizuális megjelenítésének pontos irányítását, így a mozgások és interakciók zökkenőmentesek és életszerűek lettek. Ezen kívül a kezdeti hibák és hiányosságok megoldása - mint például az animációk szinkronizálása vagy a játékos mozgásának csiszolása - remek lehetőség volt a kezdetleges Unity-ismereteim elmélyítésére.

### Erősítések

#### PowerupEffect

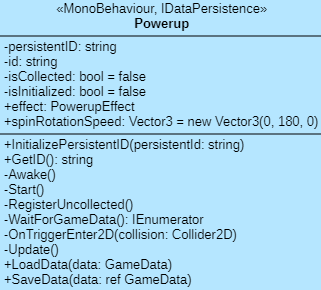
A *PowerupEffect* osztály (54. ábra: PowerupEffect osztálydiagramja) egy absztrakt osztály, amely a különböző *power-up*ok (55. ábra: Erősítések típusai) hatásait reprezentálja. Mivel ez egy *ScriptableObject*, lehetővé teszi, hogy a *power-up* típusait és azok hatásait előre definiáljuk az *Unity Editor*ban, így egyszerűen alkalmazhatunk különböző hatásokat anélkül, hogy azokat kód szinten kellene kezelni. Az *Apply* metódus minden *PowerupEffect* típushoz implementálva van, és felelős a hatás végrehajtásáért a célobjektumon (*target*en).

54. ábra: PowerupEffect osztálydiagramja



55. ábra: Erősítések típusai

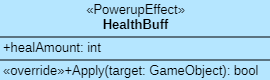
#### Powerup

A **Powerup** osztály (56. ábra: Powerup osztálydiagramja) a játékon belüli erősítések (*power-up*ok) kezelésére szolgál. Minden egyes *Powerup* példányhoz tartozik egy hatás, amely meghatározza, hogy a *power-up* milyen hatást gyakorol a játékosra, amikor az begyűjtésre kerül. A *power-up*okat a játékosok megszerezhetik, amely ezáltal az erősítéstárban kerül tárolásra(*PowerupInventory*). Az osztály az erősítéseket egyedi azonosítóval (*id*) inicializálja, amelyet vagy manuálisan rendelünk hozzá, vagy egy automatikusan generált *GUID* érték formájában. Az *InitializePersistentID* metódus használatával beállíthatjuk a tartós azonosítót. A *RegisterUncollected* metódus biztosítja, hogy a nem begyűjtött *power-up*ok megfelelően regisztrálva legyenek a játék adatállományában, hogy a játékos legközelebb is találkozhasson velük, ha nem gyűjtötte be őket.

56. ábra: Powerup osztálydiagramja

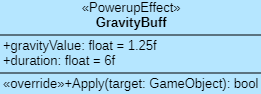
A *power-up*ok begyűjtésekor az *OnTriggerEnter2D* metódus felelős a begyűjtésükért, amely a *PowerupInventory*ba helyezi el azokat, majd eltávolítja őket a játékadatokból. Ez a metódus biztosítja, hogy a *power-up*ok ne legyenek többszörösen begyűjthetőek, és hogy a mentett játékadatok mindig frissüljenek. Az erősítés felvételét követően a megfelelő *audio* visszajelzés is lejátszásra kerül, ha az *AudioManager* elérhető.

#### HealthBuff

A **HealthBuff** osztály (57. ábra: HealthBuff osztálydiagramja) a játékos egészségpontjait növelő *power-up*ot képviseli. Az *Apply* metódus a játékos *Damageable* komponensét használja, hogy erre próbát tegyen. Ha sikerül gyógyítani a játékost, akkor igaz értékkel tér vissza. A játékos aktuális és maximális egészsége kiírásra kerül a konzolra, amely segíthet a hibakeresésben vagy a fejlesztés során.

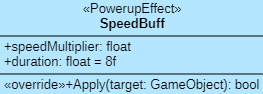
57. ábra: HealthBuff osztálydiagramja

#### GravityBuff

A **GravityBuff** szkript (58. ábra: GravityBuff osztálydiagramja) a gravitációs erő növelésére szolgáló *power-up*ot képviseli, amely a játékos mozgását befolyásolja. Az *Apply* metódusban a játékos *PlayerController* komponensén keresztül alkalmazza a gravitációcsökkentőt, amely az *ApplyGravityBuff* metódus segítségével módosítja a gravitációs erőt a játékos számára. A *buff* időtartama és mértéke beállítható.

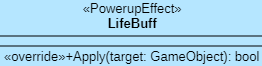
58. ábra: GravityBuff osztálydiagramja

#### SpeedBuff

A **SpeedBuff** (59. ábra: SpeedBuff osztálydiagramja) egy olyan *power-up*, amely növeli a játékos mozgási sebességét. Működése hasonló a gravitációcsökkentő működésével, viszont a sebességnövelés itt egy meghatározott szorzóval történik.

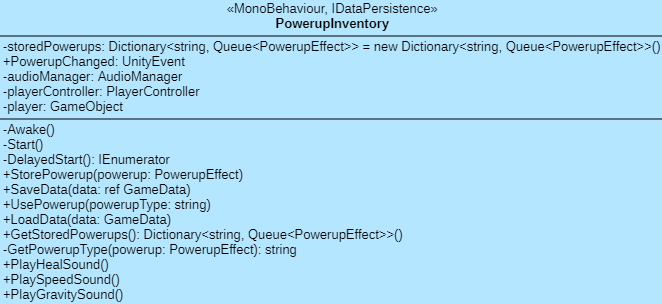
59. ábra: SpeedBuff osztálydiagramja

#### LifeBuff

A **LifeBuff** osztály (60. ábra: LifeBuff osztálydiagram) megnöveli a játékos életeinek számát az *Apply* metódusban, amennyiben a játékosnak nincs teljes számú próbálkozása.

60. ábra: LifeBuff osztálydiagram

#### PowerupInventory

A **PowerupInventory** osztály (61. ábra: PowerupInventory osztálydiagramja) a játékban lévő erősítések tárolásáért és kezeléséért felelős. Az osztály alapvetően egy *dictionary*-t használ, amelyben a kulcsok a *power-up* típusait (például *HealthBuff*, *SpeedBuff*), az értékek pedig egy-egy típushoz tartozó *Queue<PowerupEffect>* sort tárolnak. Ezzel a megoldással könnyen bővíthetjük a *power-up* típusokat, miközben biztosítjuk, hogy mindegyik típushoz egy sorban történik a tárolás.

61. ábra: PowerupInventory osztálydiagramja

A kódban az osztály feladatai között szerepel az erősítések tárolása mellett, azok felhasználása és a mentési/újratöltési adatainak kezelése is. Az *Awake* és *Start* metódusok segítenek biztosítani, hogy a szükséges komponensek (pl. az *AudioManager* és a *PlayerController*) megfelelően legyenek inicializálva a játék indításakor. Különösen figyelembe kell venni a *DelayedStart* használatát, amely késlelteti az inicializálást addig, amíg a játékos karaktere megfelelően elérhetővé nem válik. Ahogy azt a fejlesztés során megtanultam, a Unityben más osztályok példányai nem feltétlenül érhetők el azonnal az *Awake* metódusban, ezért gyakran szükség van egy enyhén késleltetett inicializálásra.

Az osztályba épített *UsePowerup* metódus az erősítések felhasználásának logikáját biztosítja. A *power-up*ot a megfelelő típus szerint kezeli, hanghatásokat is biztosítva. Ezen kívül, ha egy *power-up* felhasználásra kerül, az a megfelelő helyen törlődik, és ha egy típus összes eleme elfogyott, azt a tárolóból is eltávolítja.

Fontos megemlíteni, hogy a mentés és betöltés során az osztály a *GameData* osztályt használja a *power-up*ok nevének tárolására és betöltésére, valamint az erőforrásokat a *Resources.Load* segítségével tölti be. Ez utóbbi döntés egy fontos programozói választás, mivel a *Resources.Load* egy gyakran használt módszer, de a későbbi játékok esetén nem feltétlenül a legjobb választás a hatékonyság szempontjából. Alternatívaként érdemes lehet a *ScriptableObject*eket vagy más memóriakezelési technikákat is fontolóra venni, főleg a nagyobb projektek esetében, amikor a játék előre betöltött adatai nagyobb mennyiségű erőforrást igényelnek.

A *PowerupChanged* esemény használata külön figyelmet érdemel, mivel lehetővé teszi, hogy bárhol a játékban értesülhessünk az *inventory* állapotának változásairól. Ez egy tipikus esemény-alapú megoldás, amely segít abban, hogy a *UI*, vagy egyéb komponensek dinamikusan reagáljanak a játékos erőforrásaiban bekövetkező változásokra.

#### PowerupSpawnManager

A **PowerupSpawnManager** osztály (62. ábra: PowerupSpawnManager) felelős a *power-up*ok világba való *spawn*olásáért, amely szintén kulcsfontosságú a játékmenet szempontjából. Az osztály a *singleton* mintát alkalmazza, amely biztosítja, hogy csak egy példánya létezzen az egész játékban. A minta használata itt indokolt, mivel az erőforrásokat csak egy helyen szeretnénk kezelni, hogy elkerüljük a redundanciát és a potenciális logikai hibákat, amelyek több példányból adódhatnának. Az osztályban található *LoadData* metódus felelős a játékmentésből származó erőforrások világba való betöltéséért. Ha a betöltött *power-up* nem található meg már a világban, akkor az a megfelelő helyen, megfelelő típusú *prefab*-ből kerül létrehozásra. A *prefab*-ek betöltése dinamikusan történik a *PowerupUtils.LoadPowerupPrefab* segítségével, amely lehetővé teszi az erőforrások előzetes kezelését és memóriában való megfelelő elhelyezését. Ezen kívül az *IsPowerupAlreadyInScene* metódus ellenőrzi, hogy az adott *power-up* már létezik-e a világban, ezzel elkerülve a duplikációkat.

62. ábra: PowerupSpawnManager

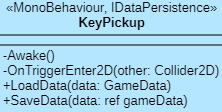
### Más felvehető objektumok

#### GetFireball

Ez az osztály (*63. ábra: GetFireball osztálydiagramja*) a játékban található tűzgolyó (*Fireball*) objektumot kezeli, amely a játékos számára egy erősebb távolsági támadást biztosít, ha azt összeszedi. Az elsődleges feladata, hogy az összeszedéskor beállítja a *PlayerController*ben, hogy a tűzlabda felvételre került. Ez az *OnTriggerEnter2D* metódusban történik, mivel ez akkor fut le, amikor a játékos fizikailag összeér az objektummal. Ennek megfelelően módosítja a *UI*-t (*UIManager*), és menti a játék állapotát a *DataPersistenceManager* segítségével, hogy a következő indításkor is emlékezzen a játék, hogy már összeszedte-e a tűzlabdát.

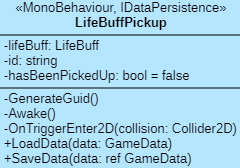
*63. ábra: GetFireball osztálydiagramja*

#### KeyPickup

A **KeyPickup** osztály (64. ábra: KeyPickup osztálydiagram) hasonló elven működik, mint a *GetFireball* szkript: amikor a játékos a kulcsot összeszedi, a *PlayerController*ben ez tárolásra kerül, a *UI* pedig frissítésre kerül. Amikor a játékos összeszedi a kulcsot, egy hanghatás is lejátszásra kerül, ha az *AudioManager* elérhető.

64. ábra: KeyPickup osztálydiagram

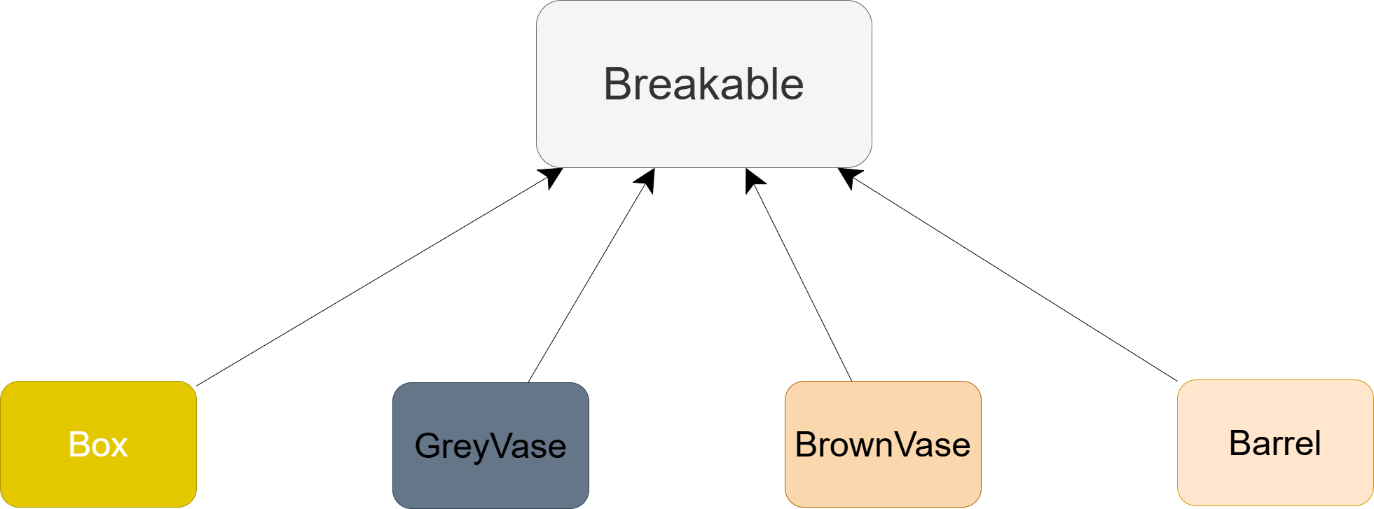
#### LifeBuffPickup

A **LifeBuffPickup** szkript (65. ábra: LifeBuffPickup osztálydiagramja) az élet erősítéseket kezeli, amelyeket a játékos felvehet a játékmenet során, egy extra próbálkozást szerezve. Itt van egy érdekes technikai döntés: a *System.Guid.NewGuid().ToString()* használata az egyedi azonosítók generálására. Ez egy okos módszer, hogy minden egyes *pickup* egyedi azonosítót kapjon, így, ha a játékos például újraindítja a játékot, az objektum újra betölthető, amennyiben még nem került felvételre. Az élet erősítés sikeres felvétele után az objektum eltűnik, a megfelelő hanghatás lejátszásra kerül, és a mentés is frissül.

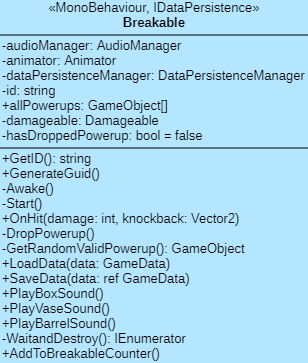
65. ábra: LifeBuffPickup osztálydiagramja

### Széttörhető tárgyak

A **Breakable** osztály (66. ábra: Széttörhető elemek típusai) a játékban lévő törhető objektumok kezelését végzi. Az osztály (67. ábra: Breakable osztálydiagram) fontos eleme, hogy az objektumok elpusztításukkor egy véletlenszerű erősítést dobhatnak le. Az *OnHit* metódus figyeli az objektumot érő ütések hatását, és ha az életerő nullára csökken, akkor a megfelelő animáció lejátszása után létrehozza a megfelelő *power-up*ot.



66. ábra: Széttörhető elemek típusai



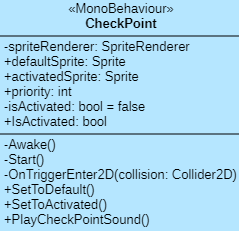
Egy érdekes megoldás a *GetRandomValidPowerup* metódus, amely biztosítja, hogy csak érvényes (nem null) *power-up*ok kerüljenek létrehozásra. A törhető objektumok életciklusának kezelését a mentés és visszatöltés is segíti, így a játékos végigkövetheti, hogy egy objektum már el lett-e pusztítva, és azt az állapotot vissza lehet tölteni, ha szükséges.

### Checkpoint rendszer

67. ábra: Breakable osztálydiagram

#### Checkpoint

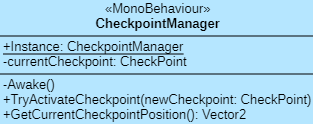
A *CheckPoint* osztály (68. ábra: Checkpoint osztálydiagramja) a játékban található ellenőrzőpontokat kezeli, amelyek segítenek a játékos számára abban, hogy a pálya adott szakaszán mentse az előrehaladását, így, ha meghal, akkor onnan folytathatja a játékot. Az osztály egy *SpriteRenderer* komponenst használ annak érdekében, hogy a *checkpoint* megjelenését dinamikusan változtassa, attól függően, hogy aktiválták-e már.

A fejlesztés során fontos szempont volt, hogy a már aktivált checkpointot hogyan kell vizuálisan megkülönböztetni az alaphelyzetben lévőtől. Ezért készítettem két különböző *sprite*-ot: egy alapértelmezettet (*defaultSprite*), és egy aktiváltat (*activatedSprite*). Az *OnTriggerEnter2D* metódus aktiválja a *checkpoint*ot, ha a játékos érintkezik vele. Ez az interakció alapvetően a *CheckpointManager* osztálytól függ, amely a játékmenet logikáját kezeli, és biztosítja, hogy csak a megfelelő prioritású ellenőrzőpontokat aktiválja, amennyiben azok még nem lettek aktiválva. Ezzel biztosítom, hogy a játékosok a pálya nehezebb részeivel találkozva a magasabb prioritású checkpointokhoz tudjanak visszatérni, ha meghalnak, anélkül, hogy újra végig kellene játszaniuk a könnyebb szakaszokat.

68. ábra: Checkpoint osztálydiagramja

Az osztály a hangok kezelésére is figyelmet fordít: amikor egy *checkpoint* aktiválódik, a *PlayCheckPointSound* metódus lejátszik egy hangot, így a játékos számára az aktiválás egyértelműen visszajelezhető a játék élményén keresztül. A szkript emellett tartalmazza a *SetToDefault* és *SetToActivated* metódusokat, amelyek biztosítják, hogy a *checkpoint* megfelelően váltson állapotot, ha a játékos sikeresen aktiválja vagy deaktiválja azt.

#### CheckpointManager

A *CheckpointManager* osztály (69. ábra: CheckpointManager osztálydiagramja) az a központi komponens, amely az összes *checkpoint* állapotát kezeli a játék során. Ez az osztály biztosítja, hogy amikor a játékos egy új *checkpoint*-ot aktivál, annak adatai (mint például a helye) helyesen frissüljenek, és a játékos éledési pontja is a legutóbb aktivált *checkpoint*-ra legyen beállítva. Az osztály *singleton* mintát használ, így csak egy példány létezik belőle a játék során.

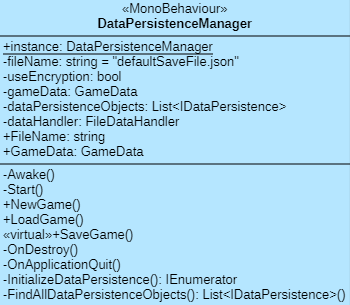
69. ábra: CheckpointManager osztálydiagramja

A *TryActivateCheckpoint* metódus kulcsfontosságú a játékmenet logikájában, mivel ennek segítségével történik az ellenőrzőpontok aktiválása. A metódus figyeli, hogy a jelenlegi éledési pont már aktiválva lett-e, és ha nem, akkor ellenőrzi a prioritását a legutóbb aktivált *checkpoint*hoz képest. Ha az új *checkpoint* prioritása nagyobb az aktuálisénál, akkor az új *checkpoint* aktiválódik, a régi pedig visszaáll a kezdőállapotába, ezáltal biztosítva, hogy mindig csak a legutóbb aktivált *checkpoint* legyen aktív. A metódus a játékos újraéledési pontját is frissíti, ha a *PlayerController* objektum elérhető, így amikor a játékos meghal, mindig a legutóbbi érvényes *checkpoint*nál fogja folytatni a játékot.

### Perzisztenciarendszer

A perzisztenciarendszer a játék egyik alapvető része, mivel lehetővé teszi az adatállapotok mentését és visszaállítását, ezáltal biztosítva a játékos előrehaladásának megőrzését. A rendszer célja, hogy az összes játékkal kapcsolatos információ - mint például a játékos állapota, a begyűjtött tárgyak vagy a játékkörnyezet változásai - megmaradjon a játékmenet alatt, illetve egy későbbi indításkor visszatölthető legyen. Az alábbiakban a főbb komponensek és a fejlesztési döntések bemutatására kerül sor.

#### DataPersistenceManager

A *DataPersistenceManager* (70. ábra: DataPersistenceManager osztálydiagram) fő feladata, hogy a játék különböző pontjain jelenlévő adatokat egységes módon elmentse, betöltse, és mindezt átlátható módon kezelje. A cél az volt, hogy egyetlen központi helyen történjen minden mentési logika, ahelyett, hogy külön-külön, szétaprózva lennének a *Save/Load* hívások a játék logikájában.

70. ábra: DataPersistenceManager osztálydiagram

A *Start()* metódusban található *StartCoroutine(InitializeDataPersistence())* hívás biztosítja, hogy a mentési folyamat ne fusson le azelőtt, hogy a Unity minden *Awake* és *Start* hívást be nem fejezett a jelenet összes objektumán. Ez különösen fontos, mivel a játék során előfordulhat, hogy egy *MonoBehaviour*, amely implementálja az *IDataPersistence* interfészt, még nincs teljesen inicializálva. Ez a *coroutine*-alapú megoldás biztosítja, hogy csak akkor kezdjük el az adatbetöltést, amikor a jelenet teljesen kész.

A *LoadGame()* és *SaveGame()* metódusok szimmetrikusan működnek: előbb összegyűjtik a jelenetben található *IDataPersistence* interfészt implementáló objektumokat, majd ciklikusan meghívják azok *LoadData()* vagy *SaveData()* metódusait. Ez a megközelítés azért előnyös, mert lehetővé teszi a laza csatolást a mentési rendszer és a játékmenet elemei között. Bármelyik *GameObject*, amely adatot szeretne menteni vagy betölteni, egyszerűen implementálja az *IDataPersistence* interfészt, és a *manager* automatikusan kezelni fogja - nincs szükség arra, hogy egyenként hozzáadjuk őket valamilyen listához.

A *FindAllDataPersistenceObjects()* metódus egy kis, de érdekes Unity-s trükk. Ahelyett, hogy kézzel hivatkoznánk a játékbeli objektumokra, kihasználjuk a *FindObjectsByType<MonoBehaviour>()* és *LINQ* *OfType<IDataPersistence>()* hívások kombinációját, hogy dinamikusan lekérjük az összes olyan komponenst, amely releváns lehet az adatmentés szempontjából. Ez robusztussá és könnyen bővíthetővé teszi a rendszert.

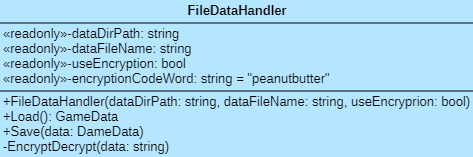
A mentés közvetlenül a kilépéskor (*OnApplicationQuit*) és az objektum megsemmisülésekor (*OnDestroy*) is lefut. Ez egyfajta biztosíték arra az esetre, ha a játékos nem nyomna mentés gombot - legalább a kilépés pillanatában mentjük az állapotot. Ugyanakkor ez hordoz magában egy veszélyt is: ha az objektum *Destroy*-a nem szabályosan történik (pl. *crash* vagy *forced quit*), akkor a mentés nem fut le, ezért fontos volt csupán kiegészítő funkcióként kezelni, nem elsődleges mentési módként.

A *useEncryption* beállítás lehetőséget biztosít arra, hogy a fájlba mentett *JSON* tartalom egy egyszerű *XOR*-alapú "titkosítással" legyen elmentve. Bár ez nyilvánvalóan nem biztonságtechnikai értelemben vett titkosítás, hanem inkább obfuszkáció, mégis jól jöhet, hogy a játékos ne tudja egy egyszerű szövegszerkesztővel manipulálni az elért pontszámokat. Az ehhez kapcsolódó logika magában a *FileDataHandler*-ben történik, nem itt, de a manager kezeli a beállítást.

A *SetInstanceForTesting()* egy apró, de fontos részlet, amely lehetővé teszi az automatizált tesztelés során, hogy a *singleton* példányt manuálisan felülírjuk. Ez a *dependency injection* egy formája, amelyet Unity-s környezetben gyakran így szokás megvalósítani, és sokat segít a *unit* tesztek írásakor.

#### FileDataHandler

A *FileDataHandler* (71. ábra: FileDataHandler osztálydiagram) osztály a háttérben dolgozik, ténylegesen elvégzi a mentési és betöltési műveleteket fájlkezelés szintjén. Míg a *DataPersistenceManager* egyfajta rendező szerepet tölt be az adatkezelés folyamatában, addig ez az osztály a piszkos munkát végzi el: a *JSON*-be való szerializálást, a fájlírást és -olvasást, valamint opcionálisan az adatok titkosítását is.



71. ábra: FileDataHandler osztálydiagram

A konstruktorban megadható három fontos paraméter: az elérési út, a fájlnév, illetve egy *bool* változó, amely a titkosítási igényt jelöli. Így bármilyen játékhoz újrahasznosítható ez az osztály, csak más konfigurációval kell példányosítani. Ezzel előrevetítettem egyfajta skálázhatóságot és moduláris szemléletet.

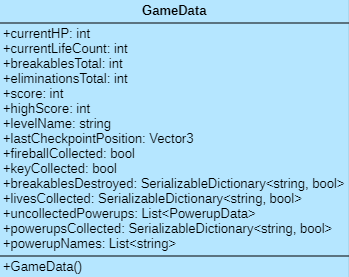
A *Save()* függvény felel az adatok *JSON* formátumban történő mentéséért. A *JsonUtility.ToJson()* metódust használom, amely a Unity beépített *JSON*-szerializálója. Ez gyors és egyszerű, viszont tudni kell róla, hogy csak Unity-annotált osztályokkal és publikus mezőkkel működik megbízhatóan. Itt jegyzem meg, hogy bár más *serializer* (pl. *Newtonsoft Json.NET*) több szabadságot biztosítana, a Unity-s beépített megoldás elegendőnek bizonyult a céljaimhoz.

Mentés előtt opcionálisan titkosítom az adatot, ha a *useEncryption* változó igaz. A titkosítás egy nagyon egyszerű *XOR*-alapú művelet, amit csak "alibi biztonságként" használok. Őszintén szólva, nem valódi adatvédelemről van szó, inkább csak annyi a célja, hogy ne legyen azonnal emberi szemmel olvasható a fájl tartalma. Ennek tudatában bármilyen komolyabb adatvédelemhez más megoldásra lenne szükség, például *AES*-titkosításra vagy valamilyen Unity-kompatibilis kriptográfiai könyvtárra.

A *Load()* metódus a mentett fájlt olvassa be, majd visszakonvertálja *GameData* objektummá. Ha a fájl nem található, egyszerűen null-lal tér vissza, amit a *DataPersistenceManager* később megfelelően kezel. Itt is van egy hibakezelés: ha a beolvasás során hiba történik (például megsérült a fájl vagy nem érvényes *JSON*), akkor a rendszer nem omlik össze, hanem logolja a hibát és egy új, alapértelmezett játékállást hoz létre. Ez felhasználói élmény szempontjából is fontos döntés volt: a cél nem az, hogy a játékos a mentés elvesztése miatt elveszítse a játékhoz való kedvét, hanem hogy ha már baj van, legalább folytatni tudja a játékot.

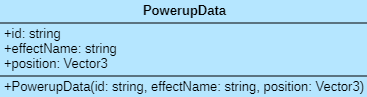
A kódban *a Directory.CreateDirectory()* meghívására is gondot fordítottam mentés előtt. Ez fontos részlet: ha például a mentési mappa valamiért nem jött létre (vagy törölték), ez biztosítja, hogy a fájlírás ne fusson hibára, hanem létrejöhessen az egész struktúra.

#### GameData és PowerupData

A mentési rendszer szívét a *GameData* (72. ábra: GameData osztálydiagram) osztály képezi, amelyben gyakorlatilag az összes lényeges játékmeneti adat egy központi helyen kerül eltárolásra. A mezők között megtalálhatóak klasszikus állapotjelzők, mint az aktuális életerő (*currentHP*), az élet számláló (*currentLifeCount*), a gyűjtött pontszámok és eliminációk száma. Ezek gyakorlatilag a játék pillanatnyi állapotát rögzítik. De helyet kaptak komplexebb szerkezetek is, mint például a *SerializableDictionary* típusú mezők, amelyek lehetővé teszik, hogy rugalmasan nyomon követhessem, melyik élet, törhető objektum vagy *power-up* lett már összegyűjtve, vagy elpusztítva.

72. ábra: GameData osztálydiagram

Azért döntöttem úgy, hogy *Dictionary*-t használok ahelyett, hogy külön *bool* mezőket hoznék létre minden egyes objektumhoz, mert a játék során ezek az objektumok dinamikusan jönnek létre - például különböző pályákon más és más törhető tárgyak vagy életbónuszok vannak. A *Dictionary<string, bool>* típus rugalmasan tudja ezt kezelni, viszont azt is tudni kell, hogy a Unity nem támogatja ezek natív szerializációját. Ezért kellett létrehoznom egy *SerializableDictionary* osztályt, amely listák segítségével képes a kulcs-érték párok konverziójára.

A *GameData* osztályban megtalálható egy *List<PowerupData>* is, amely azokat a *power-up*okat tárolja, amelyeket a játékos még nem szedett fel. Ez a lista szintén dinamikusan változik, ezért külön figyelmet érdemel. Minden *power-up* egy *PowerupData* (73. ábra: PowerupData osztálydiagramja) példányként szerepel, amely tartalmaz egy egyedi azonosítót, egy effektnevet (amely később akár egy *enum*ra is lecserélhető lenne), valamint a pályán elfoglalt pozícióját. Ez lehetővé teszi, hogy a játék újraindításakor pontosan ugyanoda tudjam visszahelyezni az elhagyott bónuszokat, ami különösen fontos egy *checkpoint* rendszerrel rendelkező játékban.

73. ábra: PowerupData osztálydiagramja

Fejlesztés közben arra is gondoltam, hogy ha a játék bővül (pl. új *power-up* típusok vagy komplexebb interakciók jönnek), akkor is elegendő lesz bővíteni a *PowerupData* struktúrát anélkül, hogy az egész mentési rendszert újra kellene írni.

A *GameData* konstruktorában minden mezőt alapértelmezett értékkel inicializálok. Ez egy fontos programozási fogás, ugyanis Unity-ben a *JSON*-ből történő visszatöltés nem mindig hívja meg az osztály konstruktorát. Így a saját *NewGame()* metódusban garantálom, hogy egy új játékindítás során minden állapot kezdőértéken legyen. Ez megint csak azt szolgálja, hogy ne kerüljön instabil állapotba a játék a mentési rendszer miatt.

Utólag visszatekintve azt is mondhatom, hogy egyes adatok (például a *powerupNames*) akár redundánsnak is tekinthetők, ha már van egy *powerupsCollected* *dictionary*. De fejlesztés közben még nem volt teljesen kiforrott, hogyan szeretném a *power-up*okat azonosítani és nyomon követni, így bizonyos duplikációk megmaradtak.

## Egyéb szkriptek

### Felhasználói felület

#### UIManager

A fejlesztés során úgy döntöttem, hogy a *UI* elemek kezelésére dedikált osztályt hozok létre, hogy az összes vizuális elemet centralizálva tartsam, így később egyszerűbben karbantartható maradjon a kódbázis. A *UIManager* osztály feladata, hogy koordinálja a megjelenítendő *UI* elemeket, például a tűzlabda, a kulcs és az erősítéstár állapotát. A szkript ezen elemeket dinamikusan frissíti, hogy azok a játékos állapotát mindig tükrözzék.

A szkript figyeli a különböző eseményeket, például a karakter sebzését (*characterTookDamage*) és gyógyulását (*characterHealed*). A megfelelő *event* aktiválásakor új szövegeket generál a képernyőn, amelyek a sebzés és gyógyulás mértékét jelzik. Ezt a *TMP\_Text* komponens segítségével oldottam meg, amely lehetővé teszi, hogy dinamikusan szöveges elemeket generáljunk a játék képernyőjén. A *UI* dinamikusan frissül akkor is, ha a játékos új erősítést szerez, az életerő-sáv alá kirajzolva, vagy frissítve a már ott megtalálható ikonját.

A *power-up*ok ikonkezelését viszont eleinte nem gondoltam át elég alaposan: nem minden esetben volt garantált, hogy a megfelelő *sprite* jelenik meg, különösen akkor, ha egy előre beállított erőforrás hiányzott. Ezt később a *GetPowerupSpriteFromPrefab* metódusban oldottam meg azzal, hogy gondoskodtam a megfelelő *sprite*-ok visszakereséséről.

További figyelmet fordítottam a játékból való kilépés platformonként eltérő kezelésére is. Az *OnExitGame* metódusban előfeldolgozási direktívákkal *(#if UNITY\_EDITOR* stb.) különböző logikákat alkalmaztam attól függően, hogy a játék milyen környezetben fut, így biztosítva, hogy a kilépés minden platformon megfelelően működjön. Ez a megközelítés nemcsak praktikus, hanem hozzájárul a fejlesztői és éles környezetek közötti kompatibilitás fenntartásához is.

#### Pontszámolásért felelős komponensek

A játék során három állandó *UI*-elem követi a játékos különféle teljesítményeit, ezek kezelésére a **BreakableCounterUI**, az **EliminationCounterUI** és a **ScoreUI** osztályokat hoztam létre. Mindhárom szkript hasonló elven működik: figyelik a megfelelő adatot a játékállásban, és csak akkor frissítik a képernyőn megjelenő szöveget, ha az adott érték ténylegesen változik. Ezt azzal a megoldással értem el, hogy csak akkor frissítem a *UI*-t, ha az új érték eltér az előzőtől, ami különösen fontos olyan helyzetekben, amikor több *UI*-komponens is egyszerre van jelen és aktívan működik a játék során. Ezek az egyszerű, de funkcionálisan fontos *UI*-komponensek hozzájárulnak ahhoz, hogy a játékos mindig pontos visszajelzést kapjon a teljesítményéről, és ezzel javuljon a felhasználói élmény.

#### HealthBar

A **HealthBar** osztály felelős a játékos aktuális életerejének pontos és arányos megjelenítéséért egy csúszka és szöveg formájában, amelyet eseményalapú frissítéssel, valamint biztonsági és teljesítménybeli szempontokat figyelembe véve valósítottam meg.

#### HealthText

A **HealthText** osztály vizuálisan jelzi a játékos sebződését vagy gyógyulását egy animált, fokozatosan elhalványuló és felfelé mozduló szöveg segítségével, amivel a játékélményt nemcsak informatívabbá, hanem látványosabbá is tettem.

#### LifeUI

A **LifeUI** osztály a játékos aktuális életereje alapján dinamikusan frissíti a felhasználói felületen megtalálható szív ikonokat, így a felhasználó mindig egyértelmű, valós idejű visszajelzést kap az próbálkozásainak számáról.

### Menük

#### MainMenuController

A **MainMenuController** osztály a játék főmenüjét kezeli. Felelősségkörébe tartozik a játék indításának, folytatásának és a kilépésének kezelése. A *NewGame* és *ContinueGame* metódusok segítségével kezeli a mentett játékállásokat, míg a *QuitGame* a játék befejezésére szolgál. Fontos megjegyezni, hogy az *Awake* metódus biztosítja, hogy ha már létezik egy *MainMenuController* példány, akkor a későbbiekben ne hozzunk létre újat, ezzel elkerülve a redundanciát.

Új játék kezdésekor a *NewGame* metódus törli a meglévő adatfájlt, ha van, majd elindítja a játékot egy új állapottal. Ezt követően betölti a *Tutorial* jelenetet. A játék folytatásakor a *ContinueGame* ellenőrzi, hogy létezik-e korábban mentett adat, és ha igen, betölti a megfelelő szintet, ha nem, akkor új játékot kezd a *NewGame* meghívásával.

A *QuitGame* metódus különböző platformokon másként működik, például *Unity Editor*ban leállítja a játékot, míg más platformokon kilép a játékból vagy egy speciális kilépő jelenetet tölt be. Az itt alkalmazott platform-specifikus feltétel (#if UNITY\_EDITOR) biztosítja, hogy a játék kilépése ne okozzon hibát a fejlesztési környezetben.

A *DataPersistenceManager* osztály használata, amely az adatok betöltéséért és elmentéséért felelős, kulcsfontosságú eleme az osztálynak, mivel garantálja, hogy a játék előrehaladását megfelelően kezeljük a különböző játékrészletek között.

#### SettingsMenu

A **SettingsMenu** osztály a játék beállításait kezeli: a felbontást, a teljes képernyős módot és a hangerőszabályozást. Az osztály a képernyő felbontását egy lista alapján állítja be, és lehetőséget biztosít arra, hogy a felhasználó kiválassza a számára megfelelő felbontást egy legördülő menüből. A teljesképernyős mód választását egy jelölőnégyzettel, a hangerőszabályozás lehetőségét pedig csúszkákkal biztosítja. A beállítások tárolására a *PlayerPrefs* *API*-t használja, amely lehetővé teszi, hogy a felhasználó által választott beállításokat elmentsük, és a következő futtatásnál betöltsük azokat.

#### PauseMenu

A **PauseMenu** osztály a játék szüneteltetésére szolgáló menüt kezeli. Az osztály feladata többek között, hogy a játékos szüneteltetni tudja a játékot, majd újra folytathassa. Az osztály lehetőséget biztosít arra is, hogy a játékos beállításokat módosítson a szüneteltetett állapotban, és kilépjen a főmenübe, vagy akár teljesen bezárja a játékot.

Az *Awake* metódus biztosítja, hogy a menü ne jöjjön létre többször, ezzel megelőzve az erőforrások fölösleges felhasználását. A játék szüneteltetésének állapotát az *isPaused* változó követi. A megfelelő *UI* elemek, például a *pauseMenu* és a *settingsMenu*, láthatóságukat ezen változó alapján módosítja.

A *PauseGame* és *ResumeGame* metódusok az időt (*Time.timeScale*) manipulálják, hogy a játék szüneteltetve legyen, és a játék folytatásakor az idő újra normális sebességgel fusson. Az akciók kiválasztására használt bemeneti térképek váltása (*SwitchCurrentActionMap*) biztosítja, hogy a játékos a megfelelő *UI* vezérlőket használhassa, például a szüneteltetett menüben a navigáláshoz, ne pedig a játékmenetben próbáljon akciókat végrehajtani (például kattintással támadni).

#### WinMenu

A *WinMenu* osztály a játékmenet végén jelenik meg, megjelenítve a játékos által összegyűjtött pontokat, az összetört tárgyak és legyőzött csontvázok számával egyetemben. Ezenfelül biztosítja a főmenübe való visszatérést, amikor is törli az elmentett játékadatokat.

### PersistentCamera és PersistentCanvas

A *PersistentCamera* és *PersistentCanvas* osztályok biztosítják, hogy a kamera és a felhasználói felület folyamatosan elérhető és konzisztens maradjon a jelenetváltások során.

### SceneController

A *SceneController* egy *singleton*-alapú rendszer, amely szintek közötti átmeneteket, animációkat, mentési és betöltési folyamatokat, valamint a játékos újrapozicionálását kezeli, ezzel biztosítva a játék zökkenőmentes és konzisztens előrehaladását.

### AudioManager

Az *AudioManager* osztály egy központi hangkezelő rendszer, amely elkülönítve kezeli a játékban megjelenő zenéket és hangeffekteket, és egyszerű, egységes hozzáférést biztosít a rá igényt tartó komponensek számára.

## A játék tesztelése

### Manuális tesztelés

A *Dungeon Crawler* készítése alatt több ismerősöm is kipróbálta a játékot, különböző fejlesztési fázisokban. Ezen tesztelések számos hibára rámutattak, lehetővé téve a gyors javításokat. A legelső komolyabb hiba a projekt elején került felszínre, a játékos alapvető irányításainak implementálásakor. Amikor a játékos oldalról egy vékony platform széléhez ugrott, - amely vastagsága kisebb volt magánál a játékosnál is - vertikális sebessége azonnal felülíródott és se nem zuhant, se nem emelkedett tovább, mintegy folyamatosan „zuhanva”. Ennek oka a polírozatlan *hitbox*, és *collider*-kezelésem volt, pontosabban az, hogy az ütközést helytelenül, csupán a játékos feje és lába körül vizsgáltam *Raycast*ek használatával. Ebből kifolyólag, ha egy platform ezen vonalak közé esett, a játékos nem detektálta az ütközést vele, ez pedig kódbéli problémákat okozott a játékfizika kezelésénél. A megoldást egyszerűen sűrűbb *Raycast* használat jelentette. Még egy hiba, melyet egy barátom talált meg manuális tesztelés során: amikor a játékos egy ellenfél fejére elég precízen érkezett ugrást követően, akkor tovább utazhatott a csontváz fején, miközben az tanácstalanul próbálta őt megtámadni. Erre egy igen egyszerű megoldást találtam: az ellenfelek fejére egy *collider*-t helyeztem, mely a játékos *collider*-ével érintkezve - jelképes sebzés kíséretében - visszataszította a játékost, eltántorítva őt a hasonló próbálkozásoktól. Ez később más szempontból is egy jó változtatásnak bizonyult, ugyanis így a pálya alacsonyabb magasságú területein elhelyezett csontvázak megkerülése, átugrása immár nagyobb kihívássá vált a felhasználónak.

### Automatizált tesztelés Unit Testekkel

A Unity-ben végzett unit tesztelés során két jól elkülönülő típusra támaszkodhattam: **EditMode** és **PlayMode** tesztekre. Mindkettő más célra való, és a használatuk során én is gyorsan megtapasztaltam, hogy mikor melyik a hasznosabb.

Az *EditMode* tesztek inkább a nem-jelenetfüggő működés ellenőrzésére alkalmasak. Ilyenkor nincs szükség a játék tényleges futtatására vagy idő-alapú események szimulálására - egyszerűen példányosítunk egy *GameObject*et, hozzárakunk néhány komponenst, beállítjuk a kezdeti értékeket, és ellenőrizzük a viselkedésüket. Például a *Skeleton* osztályom esetében így tudtam gyorsan letesztelni, hogy egy irányváltás valóban megfordítja-e a karaktert, vagy hogy a *knockback* logika helyesen változtatja-e meg a *Rigidbody2D* sebességét.

A *PlayMode* tesztek viszont már magát a játékot, annak időbeli viselkedését is figyelembe veszik - olyasmiket lehet velük vizsgálni, amik csak akkor történnek meg, ha a játék valóban fut. Itt teszteltem többek között a *Damageable* komponensemet is: hogyan reagál egy találatra, miként aktiválódik az *invincibility* mechanika, vagy hogy a karakter halála után ténylegesen leáll-e a sebződés. Itt fontos, hogy az *IEnumerator* alapú tesztelés lehetővé teszi a Unity *frame*-alapú működésének szimulálását, amit az *EditMode* nem tud.

A tesztek egyszerű futtatására a *Test Runner* komponens ad lehetőséget (74. ábra: Test Runner komponens).

A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható

Előfordulhat, hogy a mesterséges intelligencia által létrehozott tartalom helytelen.

74. ábra: Test Runner komponens

# Összefoglalás

Szakdolgozatom írásába bármiféle előzetes Unity tapasztalat nélkül vágtam bele. A motor alapvető komponensei, mint például a jelenetek (*Scenes*) vagy az ezekhez tartozó *GameObject*ek működése, kezdetben teljesen ismeretlen volt számomra. Az első pár indításnál a kezelőfelület használata is nehézséget okozott, mivel a Unity felépítését azelőtt kizárólag csak képernyőképeken keresztül ismertem. Ennek következtében eleinte bizonytalanság és tanácstalanság jellemezte a munkámat. Ez az érzés azonban viszonylag gyorsan elmúlt. Ahogy egyre több időt töltöttem a motor megismerésével, fokozatosan világossá vált számomra, hogy az alapvető működések elsajátítása nem igényel lehetetlen erőfeszítést. Rövid időn belül magabiztosabban tudtam navigálni a különböző nézetek és eszközök között, ami lehetővé tette számomra a gyorsabb haladást.

A *Dungeon Crawler* fejlesztése során kiemelt szerepet kaptak az objektum-orientált programozási (OOP) elvek. Bár a projekt nem minden osztályánál sikerült teljes mértékben megvalósítanom az ideális gyakorlatokat, a fejlesztési folyamat előrehaladtával egyre tudatosabban törekedtem ezek alkalmazására. Így például az enkapszuláció, az egységesség és az újrafelhasználhatóság fontossága világosan kirajzolódott számomra. Az implementált checkpoint rendszer, a mentési mechanizmus, valamint a karakterek és interaktív pályaelemek kezelése során különösen érzékelhető volt az OOP megközelítés előnye.

Összegzésként elmondható, hogy bár a végső kódbázis több területen hagy maga után fejlesztési lehetőségeket, a projekt során megszerzett tapasztalatok jelentősen hozzájárultak az objektum-orientált szemléletem és gyakorlati tudásom fejlődéséhez. A szakdolgozatnak köszönhetően nemcsak technikai ismereteim bővültek jelentősen, hanem megerősödött az elhatározásom is, hogy a jövőben további játékfejlesztési projektekkel foglalkozzak Unity-ben. Célkitűzésem, hogy a jövőbeli munkáim során következetesebben alkalmazzam az objektum-orientált programozási, clean code, valamint szoftvertervezési elveket, ezzel is tovább növelve a fejlesztések minőségét és hatékonyságát.

# Fejlesztési lehetőségek

Ahogy azt korábban is kiemeltem, a játék jelenlegi verziójában több olyan terület is található, ahol a kód tisztábbá és modulárisabbá tehető. A projekt egyes komponensei időnként túllépnek a saját felelősségi körükön - több funkciót látnak el, mint indokolt lenne -, illetve túlzottan függenek más osztályok helyességétől. Mivel a jövőbeli fejlesztések szempontjából kulcsfontosságú, hogy a kód könnyen bővíthető és újrafelhasználható legyen, ezért egy strukturált refaktorálás jelentené a megfelelő folytatást. Ennek keretében több osztály - például a *PlayerController* - rövidebbé és áttekinthetőbbé válhatna, mivel bizonyos függvények átkerülnének más, erre a célra létrehozott moduláris egységekbe. A Unity eseményrendszer (*Unity Event*) tudatosabb és gyakoribb alkalmazása tovább növelhetné a rendszer komponensalapúságát, így megkönnyítve új játékmeneti elemek későbbi integrálását.

A projektben emellett több helyen előfordul kódduplikáció, melyek megszüntetése szintén fontos lépés lenne a karbantarthatóság és a tisztább kódbázis irányába. Az olyan hosszú és összetett metódusok, mint egyes osztályokban például az OnTriggerEnter2D vagy az Update, jól elkülöníthető, kisebb segédfüggvényekre bonthatók lennének. Ezáltal a kód átláthatóbbá, könnyebben tesztelhetővé és újrafelhasználhatóvá válna. A változó- és függvénynevek, valamint az elnevezési konvenciók egységesítése a teljes projekten belül szintén hozzájárulna az általános kódminőség és olvashatóság javításához.

A fent vázolt alapvető refaktorálási lépések elvégzését követően lehetőség nyílna a játék funkcionális bővítésére is. Érdemes lenne megfontolni új mozgásmechanikák - például egy gyors ‘dash’ képesség - bevezetését, amely nemcsak változatosabb játékmenetet eredményezne, hanem új pályaszakaszok és ügyességi feladványok tervezését is lehetővé tenné. A játék dinamizmusát tovább növelhetné mozgó platformok, illetve elkerülendő, interaktív pályaelemek integrálása, amelyek változatosabb kihívásokat kínálnának a játékos számára, így elkerülve a játékmenet monotonitását. Szintén értékes kiegészítést jelenthetne új ellenféltípusok bevezetése, melyek legyőzése eltérő taktikát igényelne, ezzel tovább gazdagítva a játékélményt.

# Melléklet és felhasznált források

A játék fejlesztése során számos Unity-csomagot, valamint külső, ingyenesen felhasználható grafikai és audióelemeket használtam fel, amelyeket ebben a pontban dokumentálok.

## Felhasznált Unity csomagok

Itt csak a legfontosabb csomagokat említem meg, amelyeket a projekt során használtam, és amelyek nem feltétlenül kerülnek automatikusan telepítésre egy hasonló projekt létrehozásakor.

* **2D Animation**: A 2D-s animációk kezelésére szolgáló csomag. Segítségével minden játékon belüli interakció sokkal életszerűbbé tehető. Például, ha egy karakter ugrik vagy támad, a csomag lehetővé teszi az animációk dinamikus, zökkenőmentes lejátszását.
* **2D Sprite**: A 2D grafikák, úgynevezett *sprite*-ok importálására és manipulálására használjuk. E nélkül nem lenne lehetséges a játék vizuális megjelenésének kialakítása.
* **2D Tilemap Editor**: Segít gyorsan és egyszerűen létrehozni a rács alapú pályákat és térképeket. A *tilemap* alapú pályaépítéshez elengedhetetlen csomag.
* **Input System**: Lehetővé teszi az összes felhasználói bemenet, például a billentyűzet, egér vagy kontroller személyre szabható kezelését.
* **Cinemachine**: A kamerakezelést könnyíti meg, automatikusan és dinamikusan irányítja a jelenetekben a kamerát. Maximálisan személyre szabható kameraviselkedést biztosít.
* **Custom NUnit**: Tesztelési eszközként szolgál, biztosítva, hogy a projekt kódja hibamentesen működjön, és segít a tesztelési folyamatok integrálásában. Ha például egy új funkciót adunk hozzá, mint mondjuk egy új karakter képességet, a Custom NUnit segít ellenőrizni, hogy az új kód nem okoz-e problémát más rendszerekben.
* **Test Framework**: Még egy tesztelési eszköz, mely lehetővé teszi, hogy a különböző teszteket egyszerűen végrehajtsuk. Megfelelően strukturált projekt és a tesztfájlok helyes elhelyezése esetén egyetlen gombnyomással elvégezhetjük vele a tesztelést.

## Grafikus anyagok

A játékban megjelenő betűtípus (*Thaleah*) forrása:

<https://assetstore.unity.com/packages/2d/fonts/free-pixel-font-thaleah-140059?srsltid=AfmBOopMM5TR1w9viZNem63g-vcO3zVXKf64_L4wJfBQl9JnWPYcGZ1k>

A játékban felhasznált további pixeles grafikák forrásai:

* Az irányított karakter, lövedék asseteinek forrása: <https://free-game-assets.itch.io/free-tiny-hero-sprites-pixel-art>
* A csontvázak asseteinek forrása: <https://craftpix.net/freebies/chibi-skeleton-warrior-character-sprites/?num=3&count=81&sq=skeleton&pos=12>
* Kulcshoz felhasznált assetek: <https://karsiori.itch.io/pixel-art-key-pack-animated>
* Tűzlabdához felhasznált assetek: [https://bdragon1727.itch.io/fire-pixel-bullet-16x16](https://bdragon1727.itch.io/fire-pixel-bullet-16x16%20)
* Erősítések pixel art-ja: <https://assetstore.unity.com/packages/2d/gui/icons/2d-potions-pixel-art-196023>
* Szív UI és pályaelem forrása: <https://fliflifly.itch.io/hearts-and-health-bar>
* Tutorial és Dungeon pálya építéséhez felhasznált assetek, kijáratként szolgáló ajtó és a széttörhető elemek forrása: <https://incolgames.itch.io/dungeon-platformer-tile-set-pixel-art>
* Tutorial és Dungeon pálya háttere, Dungeon pályán megtalálható emelvények forrása: [https://grafxkid.itch.io/cave-tileset](https://grafxkid.itch.io/cave-tileset%20)
* Tutorial pályán megtalálható instrukciós panelekhez felhasznált assetek: <https://bdragon1727.itch.io/custom-border-and-panels-menu-all-part>
* Village pálya építéséhez felhasznált assetek: <https://cainos.itch.io/pixel-art-platformer-village-props>
* Village pálya és Győzelmi képernyő háttere: <https://free-game-assets.itch.io/free-horizontal-game-backgrounds>

## Hang és zenei anyagok

A játékban hallható hangeffektek (*SFX*) egyik forrása: <https://elevenlabs.io/>.

* Az ElevenLabs egy mesterséges intelligenciára épülő vállalat, amely élethű, érzelmekkel teli mesterséges hangokat készít szövegfelolvasás, hangklónozás és automatikus szinkronizálás segítségével
* Felhasználás: széttörhető elemek széttörésénél lejátszott hangeffekt, tűzlabda, kő lövedék hangeffektjei, csontvázharcosok sebződésének és halálának hangeffektje, játékos sebződésének és halálának hangeffektje

Egy másik forrás, ahonnan ingyenesen felhasználható hangeffekteket töltöttem le:

* <https://pixabay.com/sound-effects/>
* Felhasználás: a fent említett hangeffektek kivételével minden innen ered

A játék háttérzenéinek forrásai:

* Main Menu/Tutorial jelenet: <https://www.fesliyanstudios.com/royalty-free-music/download/8-bit-menu/287> (szerző: David Renda)
* Dungeon szint: <https://www.fesliyanstudios.com/royalty-free-music/download/retro-platforming/454> - Slower Tempo - (szerző: David Fesliyan)
* Village szint: <https://www.fesliyanstudios.com/royalty-free-music/download/8-bit-retro-funk/883> (szerző: David Renda)

## Egyéb források

A fejlesztés során a Unity hivatalos dokumentációján kívül, más segédanyagot is használatba vettem, méghozzá a következő Youtube-csatornákat, melyek segítségével gyorsan megtanulhattam új Unity-s technikákat és megoldásokat.

* Infallible Code (<https://www.youtube.com/@InfallibleCode>)
* Rehope Games (<https://www.youtube.com/@RehopeGames>)
* Shaped by Rain Studios (<https://www.youtube.com/@ShapedByRainStudios>)
* Pandemonium (<https://www.youtube.com/@PandemoniumGameDev>)