



Eötvös Loránd Tudományegyetem

Informatikai Kar

Informatikatudományi Intézet

Komputeralgebra Tanszék

Top-Down Shooter

Szerző:

Strasszer Ádám

Programtervező informatikus BSc.

Témavezető:

Magyar Péter

Egyetemi tanársegéd

Szombathely, 2025

EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

Hallgató adatai:
Név: Strasszer Ádám Ferenc
Neptun kód: JJ34QK
Képzési adatok:
Szak: programtervező informatikus, alapképzés (BA/BSc/BProf)
Tagozat: Nappali

Belső témavezetővel rendelkezem

Témavezető neve: Magyar Péter
munkahelyének neve: Eötvös Loránd Tudományegyetem, Informatikai Kar
munkahelyének címe: 9700 Szombathely, Károlyi Gáspár tér 4.
beosztás és iskolai végzettsége: Egyetemi Tanársegéd, Programtervező informatikus MSc

A szakdolgozat címe: Top-Down Shooter

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

Egy 2D, felülnézetes, gyors-tempójú pixel-art stílusú akciójáték. A játék menete: A Karakter kiválasztása után a játékos egy épület előtt találja magát. A cél az, hogy az épületben tartózkodó összes ellenséget semlegesítsük. Egy épületen belül több kisebb szoba van. A kihívást a mesterséges intelligencia által irányított ellenségek gyors reakciója és elhelyezkedése nehezíti. Képesek járőrödni, illetve észlelik a játékos által kiadott hangokat is. Több fajta ellenség létezik, például: maffia tag (normál sebesség, alacsony életerő), őr (lassabb sebesség, több életerő, ütőfegyverek nem használnak ellene), vezér (speciális támadási mozdulatok, sok életerő). A játékosnak stratégikusan fegyverhez kell jutnia, majd egyenként kell végeznie minden célponttal, mindezt úgy, hogy ne vonja fel magára a figyelmet. Különböző fegyvereknek különböző előnyei és hátrányai vannak (például: ütőfegyverek, hangtompított fél-automata pisztoly, automata géppuska, sörétes puska), és minden karakternek más speciális képessége van (például: gyorsabb mozgási sebesség, több munió), ezzel tovább bonyolítva a küldetést. Miután végeztünk, a játék pontszámozza, osztályozza a teljesítményt. Bónusz pont jár a combo-támadásokra, a pontosságra, és az időre. A kiértékelés után jön a következő szint. Minden szinten egyre több ellenség van. A projekt egy modern játékmotorral készül.

Budapest, 2024. 10. 15.

Tartalomjegyzék

1.	Bevezetés	1
1.1.	Motiváció.....	1
2.	Felhasználói dokumentáció	2
2.1.	Rendszerkövetelmények	2
2.2.	Futtatás.....	2
2.3.	Játékmenet	3
2.3.1.	Képességek	3
2.3.2.	Fegyverek	4
2.3.3.	Ellenség típusok.....	5
2.4.	Irányítás és kezelőfelület.....	5
2.4.1.	Irányítás	5
2.4.2.	Kezelőfelület	6
3.	Fejlesztői dokumentáció	8
3.1.	Fejlesztői környezet.....	8
3.2.	Technikai megoldások, törekedések	8
3.3.	Osztályok és programstruktúra	10
3.3.1.	PlayerController	10
3.3.2.	PlayerEquipment	18
3.3.3.	SoundDetectionField	21
3.3.4.	PowerUpManager	23
3.3.5.	EnemyController	26
3.3.6.	EnemyEquipment	35
3.3.7.	BossEnemy	37
3.3.8.	IncomingSoundDetector	40

3.3.9.	EnemyData	42
3.3.10.	WeaponPickup	43
3.3.11.	WeaponPickupTrigger	44
3.3.12.	WeaponData.....	46
3.3.13.	Bullet	50
3.3.14.	BackgroundColor	55
3.3.15.	FloorManager	57
3.3.16.	FloorAccessController	58
3.3.17.	DoorController és TriggerHandler	61
3.3.18.	EntranceDoubleDoorController	64
3.3.19.	LockedDoorController	69
3.3.20.	LevelExitTrigger	72
3.3.21.	UIManager	75
3.3.22.	TimerController	79
3.3.23.	AmmoDisplay	82
3.3.24.	Restart	84
3.3.25.	ScoreManager	85
3.3.26.	ScoreScreenManager	90
3.3.27.	LevelSelectManager	91
3.3.28.	MainMenuManager	93
3.3.29.	VolumeManager	95
4.	Összefoglalás, továbbfejlesztési lehetőségek	97
5.	Irodalomjegyzék, harmadik féltől származó felhasznált erőforrások	98

1. Bevezetés

1.1. Motiváció

A huszonegyedik század szórakoztatóiparának legmeghatározóbb újdonsága cím egyértelműen a videójátéknak szól. A tradicionális formákhoz képest kétség nélkül egy jóval komplexebb dologról beszélünk. A legsikeresebb játékok valójában több művészeti ágazat tökéletesen összevetett egyvelege: történetírás, hang-dizájn, karakterdizájn, szoftverdizájn, valamint a vizuális művészetek, az interaktív narratíva és a technológiai innováció egyvelege. A videójátékok nem csupán szórakoztatnak, hanem immerzív világokat teremtenek, ahol a játékos nem pusztán szemlélő, hanem aktív résztvevő, döntéseivel formálva a történetet és a környezetet. Ez az interaktivitás, komplexitás és az ágazatok közötti együttműködés teszi lehetővé, hogy a fogyasztó mélyebben kapcsolódjon a cselekményhez.

Talán nyolc éves lehettem, amikor először megtapasztaltam a videójátékok által nyújtott élményt. Bár volt internet, az még a telefonkábelén keresztül jött, sávszélessége lassú volt, és nem igazán értettem hozzá. Emiatt a többjátékos mód nem volt lehetséges, és a kínálat is rendkívül limitáltnak bizonyult, ami azt eredményezte, hogy a birtokomban lévő játékokat rengetegszer végigjátszottam. Ilyen volt többek között a Quake, Medal of Honor és Call of Duty széria több része. Sokadik teljesítés után, mikor már nem volt lehetőség magasabb nehézségi fokozatra váltásra sem, elkezdett érdekelni, vajon hogy épülnek, hogy készülnek ezek a játékok.

Az utóbbiakban jelentős időt töltöttem el a Hotline Miami címmel és folytatásával. A színes, retró esztétika, az egyszerű, ám kihívásokkal teli adrenalindús játékmenet, valamint a halványan előadott, ugyanakkor jól meghatározott történetszál és a kiváló zenei választéka magával ragadott.

Hosszas töprengést követően kihívásként a Unity játékmotort választottam, mert nagyobb rugalmasságot és több tanulási lehetőséget biztosít a modern játékfejlesztési gyakorlatok terén, emellett az iparban rendkívül népszerű, jól támogatott. A scriptek a számomra ismert C# nyelven íródtak, a grafikák Asepriteben készültek, a hanghatások FL Studioban lettek szerkesztve. A játék címe: FEROCITY.

2. Felhasználói dokumentáció

2.1. Rendszerkövetelmények

Minimális rendszerkövetelmények¹:

- Operációs rendszer: Windows 10 21H1 vagy újabb (x86) vagy Ubuntu 22.04 (x64)
- Processzor: 2GHz órajelű, SSE2 utasításkészlettel
- Memória: 512 MB RAM
- Videokártya: Windows esetén DirectX 10 kompatibilis videokártya, Linux esetén OpenGL 3.2+ képes videokártya.
- Szabad tárhely: 200 MB

Ajánlott rendszerkövetelmények:

- Operációs rendszer: Windows 10 21H1 vagy újabb (x64) vagy Ubuntu 24.04 (x64)
- Processzor: Intel Core vagy AMD Ryzen széria
- Memória: 1GB RAM
- Videokártya: Windows esetén DirectX 11 vagy DirectX 12 kompatibilis videokártya (NVIDIA GeForce GTX 660 / AMD Radeon HD 7870 vagy jobb), Linux esetén OpenGL 3.2+ képes videokártya
- Szabad tárhely: 200 MB

2.2. Futtatás

A játék nem igényel hagyományos telepítési folyamatot. A futtatáshoz:

1. Töltsd le a játékot tartalmazó tömörített mappát a megfelelő rendszeredhez (például „ferocity_release_v1.1.zip”).
2. Csomagold ki a letöltött fájl tartalmát egy tetszőleges mappába a számítógépeden (például „C:/Games/Ferocity”).
3. Nyisd meg a kicsomagolt mappát.
4. Platformtól függően futtasd a „ferocitygame.exe” vagy „ferocitygame.x86_64” fájlt.

¹ A minimum rendszerkövetelmény egy ahhoz hasonló rendszeren tesztelve lett, megítéléséhez a Unity dokumentációt vettem alapul: [\[1\] https://docs.unity3d.com/Manual/system-requirements.html#desktop](https://docs.unity3d.com/Manual/system-requirements.html#desktop)

2.3. Játékmenet

A FEROCITY egy gyors tempójú, felülnézetes akciójáték. A játékos egy karaktert irányít, a feladat egyszerű: A pályán minden ellenfelet meg kell ölni. Indulás előtt a játékos választ egyet a négy képesség közül. Az elérhető fegyverek közül mindegyik valami másban jeleskedik, hatékony használatuk stratégikus gondolkodást igényel. Az ellenfelek realisztikus reakcióidővel rendelkeznek, ugyan azon fegyvereket használják. Az adott épületszint teljesítése után a játékos a lépcső használatával mehet a következőre. Amint az összes ellenfél halott, a játékosnak vissza kell mennie a kezdőponthoz. Ezt követően a játék osztályozza a teljesítményt.



ábra 1: FEROCITY: játékmenet

2.3.1. Képességek

- double ammo: A földről felvett fegyverek tárában kétszer annyi töltény van
- two lives: A játékost kétszer találhatják el, mielőtt a játéknak vége.
- speed boost: A játékos mozgási sebessége 1,3-as szorzót kap.
- accuracy boost: A leadott lövések szórása sokkal alacsonyabb.

A képességek oly módon lettek kiegyensúlyozva, hogy mindegyik egyaránt hasznos lehet különböző játékosoknak.

2.3.2. Fegyverek

- **Bayonet**

Nem figyelmezteti az ellenségeket, eldobva sebzi őket



- **Makarov**

Fél-automata pisztoly



- **Makarov PB**

Hangtompítóval felszerelt változat, nem figyelmezteti az ellenségeket.



- **IZh-27**

Lefűrészelt, duplacsövű sörétes puska, szórása nagy.



- **PPSh-41**

Dobtáras, magas tűzgyorsaságú automata géppisztoly. Szórása és tárkapacitása nagy.

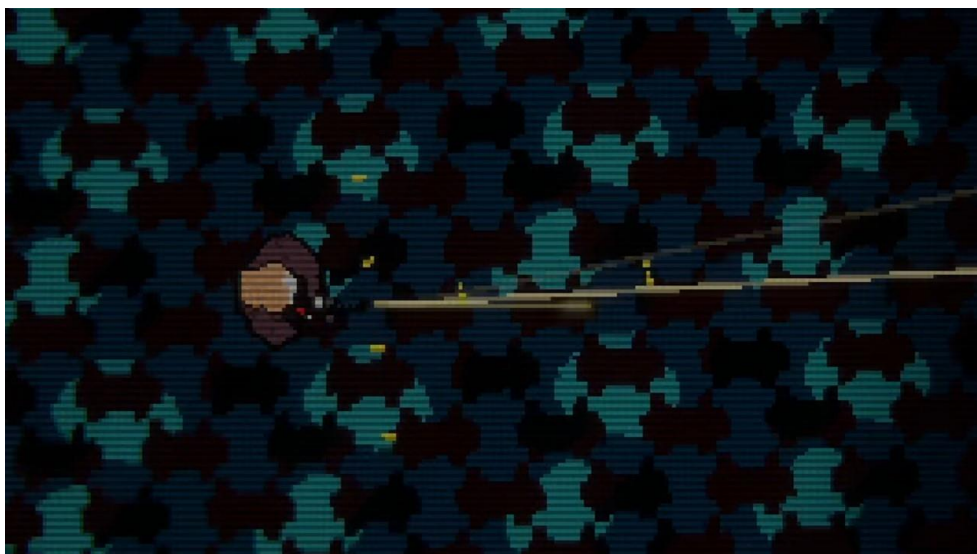


- **AKM**

Pontos, automata géppuska.



A játék az 1980-as évek szovjet Oroszországában játszódik, az fegyverek történelmileg beleillenek ebbe a korszakba, és valós karakterisztikákkal rendelkeznek. (a részecske effektus is minden fegyvernél egy hűséges reprezentációja a valódinak)



ábra 2: FEROCITY: PPSH-41

2.3.3. Ellenség típusok

- **Enemy (alapértelmezett)**

A leggyakoribb ellenség típus

Bármelyik fegyvert használhatja²

Egy találat után meghal

Kezéből kiesik a fegyver, ha a játékos megdobja a sajátjával³



- **Guard (őr)**

Csak közelharcos támadással bír

Sok életerő

Lassabb mozgási sebesség



- **Boss (főellenség)**

A legritkább ellenség típus

Immunis a lőfegyverekre

Több életerő

Speciális „rohamtámadás” képesség



2.4. Irányítás és kezelőfelület

2.4.1. Irányítás

- Játékos mozgása: W, A, S, D
- Kamera mozgása: Egérrel, Bal SHIFT nyomva tartásával megnő a látóhatár
- Támadás/Lövés: Bal egérgomb
- Fegyver felvétele/eldobása: Jobb egérgomb
- Újrakezdés: R gomb nyomva tartásával
- Kilépés: Escape gomb

² Pontosabban a játékos számára elérhető fegyverek enyhén gyengített verzióit használhatják

³ Ha Bayonettel dobja meg, fegyvere eldobása helyett az ellenség meghal

2.4.2. Kezelőfelület

2.4.2.1. Főmenü (5. ábra)

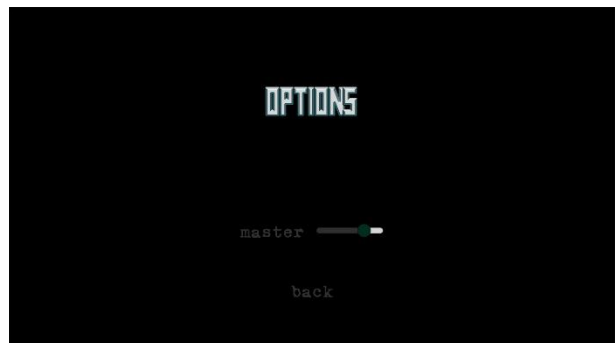
A játék indítása után rögtön a főmenü jelenik meg. A „new game” gombra kattintva a pályaválasztó képernyő fogadja a játékost.



3. ábra: Főmenü

2.4.2.2. Opciók (6. ábra)

Ebben a menüpontban egy szimpla hangerőszabályzó található. A csúszka segítségével beállított hangerőt a játék megjegyzi.



4. ábra: Opciók

2.4.2.3. Pályaválasztó (7. ábra)

A különböző pályák képeire kattintva a játék a választást elmenti, majd továbbvisz a képesség kiválasztó képernyőre.

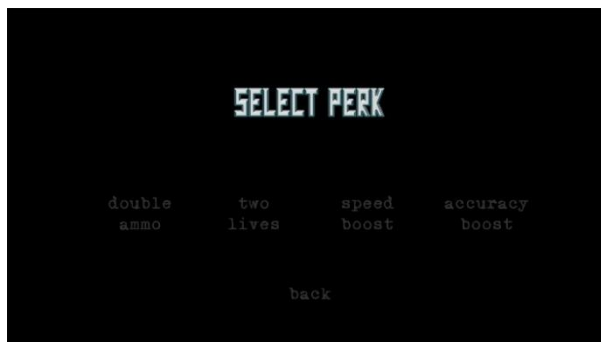


5. ábra: Pályaválasztó

2.4.2.4. Képességválasztó (8. ábra)

Ezen a képernyőn lehet kiválasztani a négy korábban felsorolt képesség közül egyet.

Kattintás után kezdődik a játék.



6. ábra: Képességválasztó

2.4.2.5. Osztályozó képernyő

A szint teljesítése után ez a képernyő fogadja a felhasználót.



7. ábra: Osztályozó képernyő

Ölésenként 100 pont jár. Az időbónusz a pályán lévő ellenségek számától függően, dinamikusan van meghatározva. A közelharci támadások a pontosságot nem befolyásolják.

A végső pontszám a következőképpen alakul:

(ölések + combo bónusz⁴) * pontosság + időbónusz

Az elérhető osztályozások⁵:

SS, S, A, B, C, D

⁴ Combo bónusz: Adott időkereten belüli többszörös ölés extra pontot ad. Az időkereten belül minden egymást követő ölésért 10%-kal több pont járul. Például: három másodpercen belül három ölés 330 pontot eredményez (100 + 100 + 100 + 10 + 20 pont)

⁵ A kapott osztályozást főként a pontosság határozza meg. SS: 100% pontosság, S: 85%+, stb.

3. Fejlesztői dokumentáció

3.1. Fejlesztői környezet

Játékmotor: Unity, verzió: 6000.0.41f1 LTS

Programozási Nyelv: C#

Fejlesztői Környezet: Visual Studio Code

Felhasznált szoftverek: Aseprite (grafikához), FL Studio 21 (hangok szerkesztéséhez)

Verziókezelés: Github: <https://github.com/adxv/ferocitygamep>

3.2. Technikai megoldások, törekedések

A FEROCITY a Unity által preferált komponensalapú architektúrát követi. A játék entitásai (GameObject-ek), mint például a játékos, az ellenségek, a lövedékek vagy a környezeti elemek, komponensek (Component) gyűjteményeiként épülnek fel. Minden komponens egy specifikus viselkedést vagy adatot tartalmaz (pl. mozgás, életerő, grafikai megjelenítés, fizikai test). A főbb logikai rendszerek különálló C# szkriptekben (MonoBehaviour leszármazottak) vannak megvalósítva, melyeket a megfelelő GameObject-ekhez csatoltam a Unity szerkesztőjében.

Egy olyan szerkezet létrehozására törekedtem, amit könnyen és gyorsan lehet bővíteni. A pályákat a játékmotor által natívan támogatott csempékkel (Tilemap) [2] készítettem, emiatt rendkívül hatékonyan, a jelenet nézetben rajzolva úgy lehet elhelyezni a fal, lépcső, és padló textúrákat, hogy azok automatikusan a megfelelő helyen és orientációban jelenjenek meg.

A WeaponData és EnemyData ScriptableObject entítások lehetővé teszik, hogy csupán pár lépésben új típusú fegyvereket és ellenség típusokat lehessen létrehozni.

Több képkockás animációk helyett (ezek extra fejlesztési és rajzadási időt igényeltek volna) próbáltam más megoldásokat alkalmazni, ahol lehetett:

- Az ellenségek és a játékos hátrafele csúszik, amikor meghal.
- Beépített Particle (részecske) effektusok használata lövéskor és sebzéskor.

A játék az új, Unity 6-ban kiadott Input Systemet használja, ami egységesen támogatja a különböző beviteli eszközöket, és lehetővé teszi az akciókhoz kötött input kezelést, tisztább kódot eredményez. A jövőben engedélyezheti a játékosok számára az irányítás testreszabását.

A valós idejű fények kezeléséért és az utófeldolgozásért (Post Processing) a beépített URP (Universal Rendering Pipeline) [\[3\]](#) felelős. Ezen grafikai effektusokat úgy állítottam be, hogy azok semmilyen esetben ne legyenek zavaróak, számottevő konfiguráció kipróbálása után jutottam el a kész játék kinézetének megvalósításához. A szerkesztőben létrehozott Volume Profile paramétereinek állításával lehet az utófeldolgozás effektusainak tulajdonságait és erősségét állítani:

- **Bloom** (Fényes területek körül ragyogó hatás)
- **Vignette** (A kép szélei sötétednek, a figyelem középre irányul)
- **Film Grain** (Zaj, ennek megvalósításához egyedi textúra van használatban)
- **Chromatic Abberation** (Képernyő szélein történő színeltérés)
- **Motion Blur** (Mozgási elmosódás, alig észrevehető, de lövés leadása esetén különbséget tesz)

Awake: Az osztály példányosításakor fut le.

Start: Az első képkockában fut le, ha a script engedélyezve van.

Update: Minden képkockában lefut

FixedUpdate: Fix időközönként (200FPS, azaz 0.0083 másodpercenként) fut le, ennek értékét lehetne finomhangolni.

A legfőbb elvárásom az volt, hogy intuitív, fluid, és rezponzív legyen minden interakció a felhasználó és játék között, ennek érdekében hajlandó voltam néhány esetben a kód szépségének rovására áldozatot hozni, egyéb esetben törekedtem az optimális megoldások használatára. [\[4\]](#) Néhány mechanika tökéletesítésére jelen fázisban nem került sor, ezeket a dokumentum végén felsorolva közlöm.

3.3. Osztályok és programstruktúra⁶

3.3.1. PlayerController

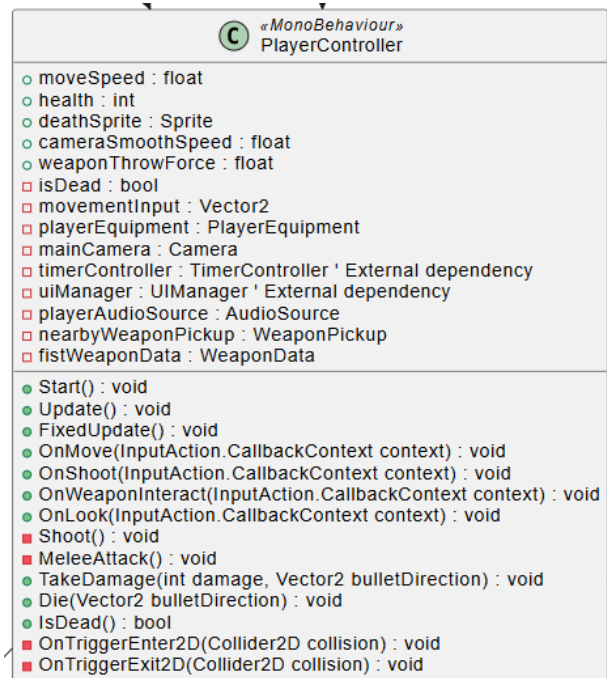
A PlayerController osztály a játékos karakter központi irányító szkriptje. Felelős a játékos mozgásának, fegyverhasználatának (lövés, közelharc, fegyverváltás), életerejének és halálának kezeléséért, valamint a játékoshoz kapcsolódó kameramozgásért és input feldolgozásért.

Inicializálás (Start):

- Az osztály indulásakor lekéri a szükséges komponenseket:

Rigidbody2D a fizikai szimulációhoz, PlayerEquipment a fegyverek és a játékos kinézetének kezeléséhez, AudioSource a hangeffektek lejátszásához, és a Camera komponenst a kamera manipulációjához.

- Alapértelmezett fegyverként beállítja az „ököl” (FistWeaponData) fegyvert, amennyiben a játékosnak nincs éppen felszerelt fegyvere, biztosítva, hogy a játékos mindig rendelkezzen valamilyen támadási képességgel. Inicializálja a lövés időzítését (lastFireTime), hogy a játékos azonnal tüzelhessen a játék kezdetekor.
- Lekéri a TimerController és UIManager singleton instance-okat a játékidő és a felhasználói felület kezeléséhez.
- Végül, a Start metódus végén meghívja az ApplyPowerUps metódust, hogy alkalmazza a játék elején esetlegesen már aktív power-upokat (pl. egy előző pályáról áthozott képességet).



⁶ A Teljes UML Állapotgép diagram elérhető a következő linken: [5]

<https://raw.githubusercontent.com/adxv/ferocitygamep/refs/heads/main/plantuml-diagram.png>

3.3.1.1. Input Kezelés:

Az OnMove, OnShoot, OnWeaponInteract, és OnLook metódusok fogadják az Input System által küldött eseményeket (InputAction.CallbackContext objektumokon keresztül).

OnMove:

- Beolvassa a mozgási inputot (egy Vector2 érték).
- Amint a játékos először elmozdul (movementInput != Vector2.zero), elindítja a játékidő mérését (TimerController.StartTimer()) és a pontszámítást (ScoreManager.Instance.StartLevel()). Ez biztosítja, hogy az idő és a pontszám csak az aktív játékmenet során növekedjen.

OnShoot:

Kezeli a lövés gomb lenyomását és felengedését.

A shouldShoot logikai változó biztosítja a konzisztens tűzgyorsaságot, különösen félautomata fegyvereknél. Az Input System performed eseménye ugyanis nem feltétlenül aktiválódik minden egyes frame-ben, amíg a gomb lenyomva van, így a shouldShoot flag fenntartja a tüzelési szándékot a gomb felengedéséig. Félautomata fegyvereknél a shouldShoot a lövés után azonnal false-ra áll az Update ciklusban.

OnWeaponInteract:

- Kezeli a fegyver felvételét és eldobását.
- Ha van a közelben felvehető fegyver, és a játékos megnyomja a jobb egérgombot, felveszi azt.
- Ha a játékosnak már volt fegyvere (nem az ökke), akkor azt először eldobja a DropWeapon metódussal.
- Ha nincs felvehető fegyver a közelben, de a játékos nem az öklét használja, akkor a gombnyomásra eldobja a jelenlegi fegyverét és visszavált az öklre.
- Egy rövid időkorlát (weaponActionCooldown) van érvényben a fegyverinterakciók között, hogy megakadályozzuk a fegyverek gyors, akaratlan cserélgetését vagy eldobását. Tesztelés közben azt vettem észre, hogy ennek értékét jóval alacsonyabbra lehet állítani, mint ahogy azt eredetileg megítéltem

OnLook:

Kezeli a „nézelődés” funkciót (Shift billentyű).

- Beállítja az isLooking logikai változót, ami a FixedUpdate-ben befolyásolja a kamera követési offsetjének mértékét, lehetővé téve a játékosnak, hogy „előrébb nézzen” az egérkurzor irányába. Gyakorlatilag megnöveli a látótávot.

3.3.1.2. Mozgás és Forgás (FixedUpdate, Update):

FixedUpdate:

A fizikai mozgás itt kerül kezelésre.

- A Rigidbody2D komponens linearVelocity tulajdonságának közvetlen beállításával mozgatja a karaktert a movementInput alapján. Ez a megközelítés biztosítja a fizikai motorral való konzisztens interakciót, mivel a FixedUpdate fix időközönként fut. (200FPS)
- A kamera pozíciójának frissítése (simított követés Vector3.Lerp-pel és az egérpozíciótól függő offsettel) szintén itt történik, hogy a mozgása független legyen a képfrissítési rátától. A kamera rázkódását (ShakeCamera) is itt alkalmazza a shakeTimeRemaining alapján.
- Ha a játékos meghalt (isDead), a mozgást és forgást leállítja.

Update:

A játékos karakterének forgatása itt valósul meg.

- A karakter transform.up vektorát (a karakter „tetejét”) folyamatosan az egérkurzor irányába állítja.

Ez azért az Update-ben történik, mert ez a ciklus minden frame-ben lefut, így a célzás a lehető legreszponzívabb lesz a játékos számára. A lövés logikájának egy része (tűzgyorsaság ellenőrzése, shouldShoot alapján a Shoot vagy MeleeAttack metódus meghívása) szintén itt található, hogy a lövési inputra adott válasz is azonnali legyen.

3.3.1.3. Támadás (Shoot, MeleeAttack):

Shoot:

- Ellenőrzi, hogy a játékosnak van-e fegyvere (`playerEquipment.CurrentWeapon`), az tud-e lőni (`canShoot`), és van-e elég lőszere (`HasAmmo`).
- Ha nincs lőszer, meghívja a `PlayEmptyClickSound` metódust.
- Meghívja a `WeaponData UseAmmo` metódusát, ami csökkenti a lőszert és visszaadja, hogy sikeres volt-e (volt-e lőszer). Ha sikeres, növeli a kilőtt lövések számát a `ScoreManager`-ben.
- Létrehozza a lövedék (`projectilePrefab`) és a torkolattűz (`muzzleFlashPrefab`) példányait a megfelelő pozícióban (`transform.position + bulletOffset` és `bulletOffsetSide`) és a játékos aktuális forgásának (`transform.rotation`) megfelelően. A torkolattűzet egy rövid idő után (`muzzleFlashDuration`) megsemmisíti.
- Kezeli a fegyver szórását (`spread`) és a sörétes puskák működését (`pelletCount`). A szórás mértékét befolyásolhatja a `PowerUpManager.AccuracyMultiplier`. Sörétes puskáknál (`pelletCount > 1`) több lövedéket hoz létre egyetlen lövésre. Minden sörét (`pellet`) külön lövedékként jön létre, de egyedi azonosítót (`shotgunBlastID`) kapnak. Ez az azonosító a `Bullet` szkriptben kerül felhasználásra, hogy egy adott ellenséget egyetlen sörétes lövésből származó több sörét közül csak egy sebezzen meg először, elkerülve az irreálisan magas sebzést egyetlen lövéssel, és hogy a pontosság számítása is értelemszerű legyen. A sörétek szöge a `spreadAngle` alapján oszlik el.
- Beállítja a létrehozott lövedék paramétereit (sebesség, hatótáv, sebzés) a hozzá tartozó `Bullet` szkripten keresztül (`SetBulletParameters`, `SetDamage`, `SetWeaponData`). Átadja a lövő objektum referenciáját is.
- Lejátssza a lövés hangját (`shootSound`) a `playerAudioSource`-on keresztül, változó hangmagassággal (`Random.Range(1.1f, 1.3f)`).
- Elindítja a kamera rázkódását (`ShakeCamera`) a fegyverhez rendelt értékekkel (`shootShakeDuration`, `shootShakeMagnitude`).
- Frissíti a `lastFireTime` értékét a következő lövés időzítéséhez. Félautomata fegyvereknél a `shouldShoot` változót `false`-ra állítja.

MeleeAttack:

- Ellenőrzi, hogy a jelenlegi fegyver közelharci-e (isMelee).
- Elindít egy Coroutine-t (AttackAnimation) a támadási animáció (sprite csere) és annak időzítésének kezelésére.
- A Physics2D.OverlapCircleAll segítségével megkeresi a támadás hatókörén (range) belül lévő összes Enemy rétegen található kolliderrel rendelkező objektumot. A keresés középpontja a játékos előtt van (attackOrigin).
- Minden eltalált, még élő (!enemy.isDead) ellenségen meghívja a TakeDamage metódust a fegyver sebzésével (meleeWeapon.damage), és létrehoz egy vér effektust (Particles/Blood) az ellenség pozíciójában.
- Lejátssza a találati (hitSound) vagy elvétési (missSound) hangot attól függően, hogy talált-e el ellenséget (didHit). A hangmagasság itt is véletlenszerűen változik.
- Kamera rázkódást alkalmaz a közelharci fegyverhez rendelt értékekkel.

3.3.1.4. Sebződés és Halál (TakeDamage, Die):

TakeDamage:

- Csökkenti a játékos életerejét (health) a kapott sebzéssel (damage).
- Létrehoz egy vér effektust a játékos pozíciójában, a sebzés irányából (bulletDirection).
- Ellenőrzi, hogy az életerő 0 alá esett-e. Ha igen, meghívja a Die metódust, átadva a halálos lövés irányát.

Die:

- Beállítja az isDead állapotot true-ra, ami letiltja a legtöbb játékos akciót (mozgás, lövés, input feldolgozás).
- Leállítja a játékos mozgását (`rb.linearVelocity = Vector2.zero`)
- Megállítja az időzítőt (`TimerController.StopTimer`), megjeleníti a Game Over képernyőt (`UIManager.ShowGameOver`), lecseréli a játékos sprite-ját a halál sprite-ra (`deathSprite`)
- Kikapcsolja a játékos Collider2D komponensét, hogy ne ütközzön tovább
- Visszaállítja az AmmoDisplay-t
- Ha a halált egy lövés okozta (`bulletDirection != default`), a Rigidbody2D típusát Dynamic-ra váltja, és egy kis lökést ad a karakternek hátrafelé a lövés irányával ellentétesen (`rb.AddForce`), majd egy rövid késleltetés (`Invoke(nameof(StopAfterNudge), 0.2f)`) után visszaállítja Static-ra, hogy a holttest ne csússzon tovább.

3.3.1.5. Fegyver Kezelés (`OnWeaponInteract`, `DropWeapon`, `UpdateClosestWeaponPickup`, `ProcessWeaponPickup`):

`DropWeapon`:

- Létrehozza a jelenleg használt fegyverhez tartozó pickup prefabot (`pickupPrefab`) a játékvilágban, a játékos pozíciója előtt.
- Beállítja a létrehozott pickup lőszerét (`SetCurrentAmmo`) a fegyver aktuális lőszerére.
- Fizikai erővel (`weaponThrowForce`) és véletlenszerű forgással elhajítja a pickupot.
- Ezután a játékost visszaváltja az alapértelmezett ököl fegyverre (`fistWeaponData`), és frissíti a `lastFireTime`-ot.
- Beállít egy cooldown-t (`lastWeaponPickupTime`), ha a `applyCooldown` paraméter true.

UpdateClosestWeaponPickup:

Ez a metódus az Update-ben hívódik meg folyamatosan.

- A Physics2D.OverlapCircleAll segítségével keresi a játékos körüli (2.0f sugarú körben) Pickup rétegen lévő WeaponPickup objektumokat.
- Kiszámolja a távolságokat, és a legközelebbit eltárolja a nearbyWeaponPickup változóban.
- Ha nincsenek pickupok a közelben, null-ra állítja. Ez azért történik, hogy az OnWeaponInteract mindig a legközelebbi, felvehető fegyverrel lépjen interakcióba, még akkor is, ha a játékos nem lépett be pontosan a pickup triggerébe, vagy ha több pickup van egymás mellett. Ez a megközelítés jobb, mint pusztán a OnTriggerEnter/OnTriggerExit eseményekre hagyatkozni. Az utóbbi módszert is próbáltam, viszont több okból kifolyólag a jelenlegi megoldást választottam. Egy ilyen problémára példa: ha van fegyver a játékos mellett, és eldobta a már nála lévő, akkor mivel a trigger csak a rádiuszba belépve aktiválódik, új fegyvert nem tud felvenni, ahhoz el kellett hagyni, majd újra be kellett lépnie a játékosnak a triggerbe.

ProcessWeaponPickup:

Akkor hívódik meg, amikor a játékos felvesz egy fegyvert (az OnWeaponInteract-ból vagy a Start-ból az ApplyPowerUps-on keresztül).

- Ellenőrzi, hogy a „Double Ammo” power-up aktív-e (PowerUpManager.HasDoubleAmmo).
- Ha igen, és a felvett fegyver nem közelharci és van tárkapacitása (magazineSize > 0), akkor a fegyver aktuális lőszerét (currentAmmo) a tárkapacitás kétszeresére állítja.

OnWeaponInteract: Lásd az Input Kezelés szekciót.

3.3.1.6. Kamera Kezelés (FixedUpdate, ShakeCamera, OnLook):

FixedUpdate:

- A kamera pozícióját itt frissítjük. A célpozíció a játékos pozíciója, plusz egy offset, ami az egérkurzor képernyőn elfoglalt relatív helyétől függ (mouseScreenPos). Az offset mértékét az offsetFactor és maxOffset változók szabályozzák, amelyek értéke attól függ, hogy a játékos lenyomva tartja-e a „nézelődés” gombot (isLooking).
- Ha isLooking igaz, nagyobb offsetet engedélyezünk (shiftOffsetFactor, shiftMaxOffset), lehetővé téve a távolabbra tekintést.
- A kamera tényleges pozícióját Vector3.Lerp segítségével simítottan közelítjük a célpozícióhoz (smoothedPos), megakadályozva a hirtelen ugrásokat. Ehhez a simított pozícióhoz adjuk hozzá a rázkódásból származó elmozdulást (shakeOffset), ha aktív (shakeTimeRemaining > 0).

ShakeCamera:

Ez egy egyszerű metódus, ami beállítja a kamera rázkódásának hátralévő idejét (shakeTimeRemaining) és erősségét (shakeMagnitude). A tényleges rázkódást a FixedUpdate végzi el Random.insideUnitSphere segítségével.

OnLook: Lásd az Input Kezelés szekciót.

3.3.1.7. Power-upok (ApplyPowerUps, ProcessWeaponPickup):

ApplyPowerUps:

A Start metódusban hívódik meg.

- Lekérdezi a PowerUpManager állapotát, és alkalmazza a releváns módosítókat:
 - Ha a HasDoubleHealth igaz, beállítja az életerőt 2-re.
 - Megszorozza a moveSpeed-et a MovementSpeedMultiplier-rel.
 - Ha a HasDoubleAmmo igaz, meghívja a ProcessWeaponPickup-ot az aktuálisan felszerelt fegyverre.

ProcessWeaponPickup: Lásd a Fegyver Kezelés szekciót.

3.3.1.8. AttackAnimation (Coroutine):

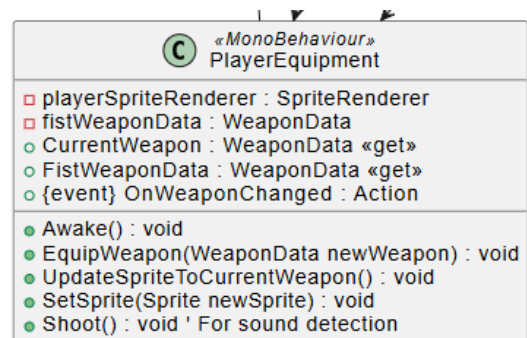
A közelharc támadás vizuális megjelenítésére. A Coroutine használata lehetővé teszi, hogy a sprite csere egy meghatározott ideig (weapon.attackDuration) tartson anélkül, hogy blokkolná a fő programszálát.

- Lecseréli a sprite-ot (opcionálisan váltogatva két sprite között, közelharc fegyverek esetén: useAlternatingSprites), vár a megadott ideig (WaitForSeconds), majd visszaállítja az eredeti fegyver sprite-ot.
- Ha a játékos új támadást indít vagy fegyvert vált/dob el, a futó Coroutine leállításra kerül (StopCoroutine).

3.3.2. PlayerEquipment

A PlayerEquipment osztály felelős a játékos által aktuálisan használt fegyver kezeléséért. Ez a szkript gondoskodik arról, hogy a játékos karakter sprite-ja megfelelően frissüljön a kiválasztott fegyverhez, nyilvántartsa az aktuális fegyvert. Együttműködik a PlayerController osztállyal

(amely a fegyver felvételét/eldobását kezdeményezi), a WeaponData ScriptableObject-ekkel, amelyek a fegyverek specifikus tulajdonságait tárolják, valamint a SoundDetectionField komponenssel a hangérzékeléshez. Alapértelmezett „fegyverként” az ököl (fistWeaponData) szolgál.



3.3.2.1. Inicializálás (Awake):

- Ellenőrzi, hogy a szükséges komponensek – a `playerSpriteRenderer` (a játékos sprite-jának megjelenítéséért felelős komponens) és a `fistWeaponData` (az alapértelmezett „ököl” fegyver adatai) – hozzá vannak-e rendelve az Inspectorban.
- Ha bármelyik hiányzik, hibaüzenetet dob a konzolra
- Ha minden rendben van, alapértelmezettként felszereli az „Fists” fegyvert az `EquipWeapon` metódus meghívásával. Ez biztosítja, hogy a játékos mindig rendelkezzen valamilyen (akár alap) felszereléssel.

3.3.2.2. Fegyver Felszerelése (`EquipWeapon`):

Ez a publikus metódus teszi lehetővé a játékos fegyverének lecserélését. Paraméterként egy `WeaponData` objektumot vár.

- Első lépésként ellenőrzi, hogy a játékos nem próbálja-e ugyanazt a fegyvert felszerelni, amivel már rendelkezik (`newWeapon == CurrentWeapon`). Ha igen, a metódus azonnal visszatér, elkerülve a felesleges műveleteket.
- Ha új fegyverről van szó, frissíti a `CurrentWeapon` property-t az új fegyver adataival.
- Ezután meghívja az `UpdateSpriteToCurrentWeapon` metódust, hogy a játékos sprite-ja megfeleljen az új fegyvernek.
- Végül kiváltja az `OnWeaponChanged` eseményt. Az eseményalapú megközelítés lehetővé teszi más rendszerek (pl. a felhasználói felületet kezelő `UIManager`) számára, hogy értesüljenek a fegyverváltásról anélkül, hogy közvetlen függőség jönne létre a `PlayerEquipment` és ezen rendszerek között. Ez egy laza csatolást (*loose coupling*) eredményező tervezési minta, ami növeli a kód modularitását és karbantarthatóságát. Alternatívaként: közvetlenül hívhatnám más komponensek metódusait (pl. `UIManager.UpdateAmmoDisplay()`), de ez szorosabb csatolást és merevebb architektúrát eredményezne.

3.3.2.3. Sprite Kezelés (SetSprite, UpdateSpriteToCurrentWeapon):

SetSprite:

Segédmetódus, amely közvetlenül beállítja a playerSpriteRenderer sprite-ját a kapott Sprite objektumra, feltéve, hogy a renderer és a sprite sem null.

UpdateSpriteToCurrentWeapon:

Segédmetódus, az aktuálisan felszerelt fegyver (CurrentWeapon) playerSprite tulajdonságát használja a játékos sprite-jának frissítésére. Ha CurrentWeapon vagy annak playerSprite-ja null, akkor visszaáll egy olyan sprite-ra, ami elérhető.

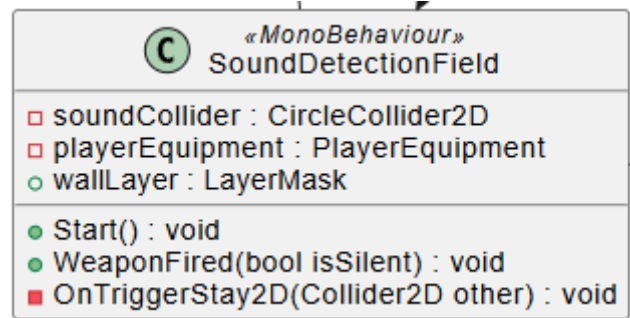
Lövés Hangjának Kezelése (Shoot):

Ezt a metódust a PlayerController hívja meg, amikor a játékos lő.

- Ellenőrzi, hogy van-e aktuális fegyver (CurrentWeapon != null) és hogy az nem hangtompítás-e (!CurrentWeapon.isSilent).
- Ha mindkét feltétel teljesül, megkeresi a játékos objektum hierarchiájában (a gyerek objektumokon) a SoundDetectionField komponenst a GetComponentInChildren<SoundDetectionField>() segítségével.
- Ha talál ilyet, meghívja annak WeaponFired metódusát, jelezve, hogy hangos lövés történt. Ez a mechanizmus teszi lehetővé, hogy az ellenségek észlelhessék a játékos lövéseit. A GetComponentInChildren használata rugalmasságot biztosít a hierarchia felépítésében, de futásidejű keresést igényel. Mivel ez csak lövéskor történik, és nem minden képkockában, a teljesítményre gyakorolt hatása elhanyagolható.

3.3.3. SoundDetectionField

A SoundDetectionField osztály felelős azért, hogy érzékelje a játékos által keltett fegyverhangokat, és értesítse a hatókörön belüli ellenségeket, amennyiben a hang nincs blokkolva akadályok által, és az aktuális fegyveren az isSilent property



False. Ez a komponens a játékos karakter egy gyermek objektumán helyezkedik el, és egy CircleCollider2D komponenst használ triggerként a hang terjedési sugarának szimulálására.

3.3.3.1. Inicializálás (Start):

A Start metódus lekéri a szükséges komponenseket: a CircleCollider2D-t, amely a hangérzékelési zónát definiálja, és a szülő objektumon található PlayerEquipment komponenst.

3.3.3.2. Fegyverhasználat Észlelése (WeaponFired, ResetWeaponFiredFlag):

WeaponFired(bool isSilent):

Ezt a publikus metódust külső szkriptek (valószínűleg a PlayerController vagy egy fegyverkezelő szkript) hívják meg, amikor a játékos elsüt egy fegyvert. A isSilent paraméter jelzi, hogy a fegyver hangtompítás-e (vagy közelharci).

- Ha a fegyver nem halk (!isSilent), a weaponFired logikai változó true-ra állítódik.

ResetWeaponFiredFlag (Coroutine):

- Egy rövid késleltetés (fél másodperc) után visszaállítja a weaponFired változót false-ra.

3.3.3.3. Ellenségek Értesítése (OnTriggerStay2D):

- Ez a metódus minden fizikai frissítéskor lefut, amikor egy másik Collider2D a SoundDetectionField trigger zónáján belül tartózkodik.
- Először ellenőrzi, hogy a játékos lőtt-e nemrég (weaponFired értéke true).
 - Ha nem, a metódus nem tesz semmit.
- Megpróbálja lekérni az IncomingSoundDetector komponenst a triggerbe lépő objektumról.
- Ha ez sikerül (tehát egy ellenség hangérzékelőjéről van szó):
 - Értesítési Késleltetés: Ellenőrzi az enemyNotificationTimes szótárban, hogy az adott ellenség (soundDetector) kapott-e értesítést a közelmúltban. Az autoWeaponNotificationDelay (alapértelmezetten 1 másodperc) határozza meg a minimális időt két értesítés között ugyanazon ellenség számára. Ez megakadályozza, hogy a gyorsan tüzelő (automata) fegyverek folyamatosan „spam”-eljék az ellenséget hangjelzésekkel.
 - Ha túl kevés idő telt el az utolsó értesítés óta, a metódus visszatér.
 - Látó-/Hallóvonal Ellenőrzés: Physics2D.Linecast segítségével egy vonalat húz a hang forrása (a SoundDetectionField pozíciója) és az ellenség hangérzékelőjének pozíciója között. Ellenőrzi, hogy ez a vonal metszi-e a wallLayer-en lévő kollidert (azaz falat).
 - Értesítés Küldése: Ha a Linecast nem talál akadályt (hit.collider == null), és az értesítési késleltetés is letelt, akkor meghívja az ellenség IncomingSoundDetector komponensének DetectSound() metódusát, és frissíti az adott ellenséghez tartozó időbélyeget az enemyNotificationTimes szótárban a jelenlegi időre (Time.time).

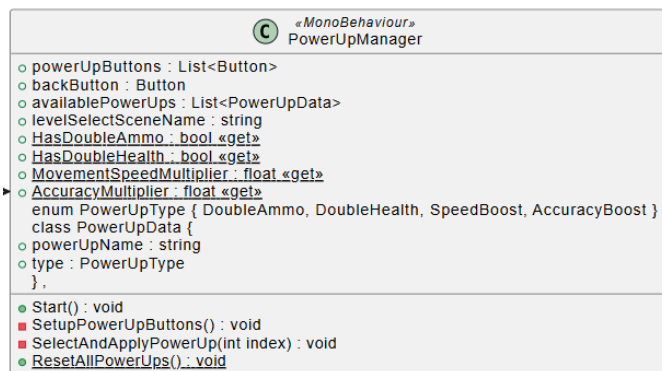
3.3.3.4. Régi Értesítések Tisztítása (Update):

Az Update metódus felelős az enemyNotificationTimes szótár karbantartásáért.

- Végigmegy a szótár bejegyzésein, és összegyűjti azokat az ellenségeket (IncomingSoundDetector kulcsokat), amelyek utolsó értesítése régebbi, mint az autoWeaponNotificationDelay kétszerese.
- Ezeket a régi bejegyzéseket eltávolítja a szótárból, hogy az ne növekedjen feleslegesen az idő múlásával olyan ellenségekkel, amelyek már rég nem relevánsak a hangérzékelés szempontjából.

3.3.4. PowerUpManager

A PowerUpManager osztály felelős a játékos által a pálya megkezdése előtt választható képességek kezeléséért. Ez az osztály biztosítja a felhasználói felületet a power-upok kiválasztásához, feldolgozza a játékos választását, alkalmazza a kiválasztott képesség hatását a játékmenetre (statikus változókon keresztül), és betölti a megfelelő játékjelenetet.



3.3.4.1. Inicializálás (Start):

- Visszaállítja az összes power-up állapotát az alapértelmezettre a ResetPowerUps metódus meghívásával, biztosítva, hogy ne maradjanak aktívak korábbi játékmenetektől származó hatások.
- Ezután a PlayerPrefs.GetString segítségével lekéri a pályaválasztó képernyőn kiválasztott és elmentett pálya nevét (sceneToLoad).
- Beállítja a felhasználói felületen található gombokhoz tartozó eseménykezelőket a SetupPowerUpButtons metódussal, valamint konfigurálja a „Vissza” gomb (backButton) működését, ami visszaviszi a játékost a pályaválasztó képernyőre (GoBackToLevelSelect).

3.3.4.2. Power-upok és UI Kezelés (SetupPowerUpButtons, PowerUpData, PowerUpType):

A választható képességeket az availablePowerUps lista tárolja, amely PowerUpData típusú objektumokat tartalmaz. A PowerUpData egy egyszerű belső osztály, ami összekapcsolja a power-up felhasználó által látható nevét (powerUpName) és annak belső típusát (type), ami egy PowerUpType enum érték (pl. DoubleAmmo, DoubleHealth, SpeedBoost, AccuracyBoost).

SetupPowerUpButtons:

- A metódus végigiterál a powerUpButtons listán (melyek a UI gombok referenciái) és az availablePowerUps listán.
- Minden gombhoz hozzárendel egy onClick eseménykezelőt, amely a SelectAndApplyPowerUp metódust hívja meg az adott power-up indexével.

3.3.4.3. Power-up Kiválasztás és Alkalmazás (SelectAndApplyPowerUp):

SelectAndApplyPowerUp:

Amikor a játékos rákattint egy képesség gombra, ez a metódus hívódik meg a megfelelő indexszel.

- Ez a metódus először meghívja a ResetPowerUps metódust, hogy biztosítsa, csak az újonnan kiválasztott power-up hatása érvényesüljön.
- Ezután a kapott index alapján kikeresi a kiválasztott PowerUpData objektumot az availablePowerUps listából.
- Egy switch utasítás segítségével, a power-up típusa (selectedPowerUp.type) alapján beállítja a megfelelő statikus változót.
- A megfelelő statikus változó beállítása után a metódus meghívja a LoadGameScene függvényt, ami betölti a korábban (Start-ban) lekért játékjelenetet.

3.3.4.4. Állapotkezelés (Statikus Változók):

A kiválasztott power-up hatását statikus változók (HasDoubleAmmo, HasDoubleHealth, MovementSpeedMultiplier, AccuracyMultiplier) tárolják.

3.3.4.5. Visszaállítás (ResetPowerUps, ResetAllPowerUps):

ResetPowerUps:

A metódus minden egyes képesség kiválasztása előtt lefut, hogy visszaállítsa az összes statikus változót az alapértelmezett értékre (pl. MovementSpeedMultiplier = 1f).

ResetAllPowerUps:

Egy public static metódus, amely lehetővé teszi a power-up állapotok teljes visszaállítását kívülről, például egy új játék indításakor vagy a főmenübe való visszatéréskor, anélkül, hogy szükség lenne a PowerUpManager egy konkrét példányára.

3.3.4.6. Jelenetkezelés (LoadGameScene, GoBackToLevelSelect, PlayerPrefs):

Ez az osztály kezeli a power-up választó képernyőről a játékjelenetbe (LoadGameScene) és a pályaválasztó képernyőre (GoBackToLevelSelect) való navigációt a SceneManager.LoadScene segítségével. A betöltendő pálya nevét (sceneToLoad) a PlayerPrefs-ből olvassa ki.

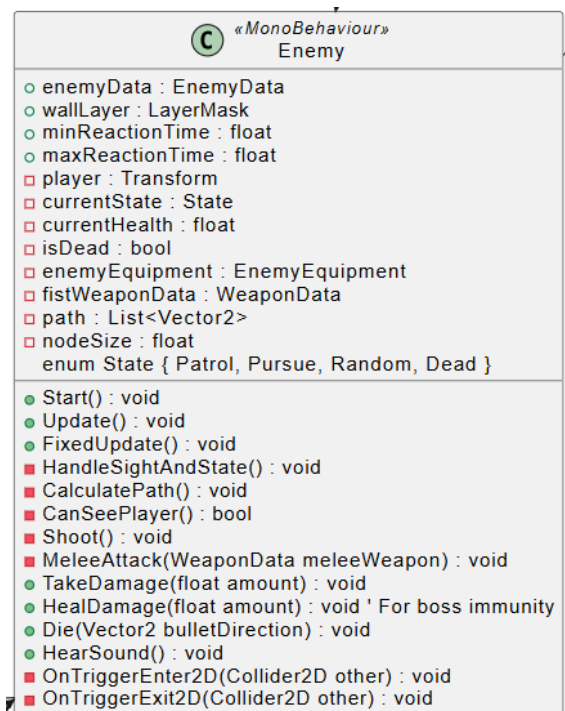
A PlayerPrefs használata egyszerű módja az adatok (itt a kiválasztott pálya neve) átadásának jelenetek között. Azonban a PlayerPrefs elsősorban a felhasználói beállítások tartós mentésére szolgál, és nem ideális eszköz az átmeneti állapotok (mint a következő betöltendő jelenet) kezelésére. Nagyobb rendszerekben jobb megoldás lenne egy dedikált GameStateManager, SceneController statikus változója, vagy egy DontDestroyOnLoad script használata az ilyen információk átadására.

3.3.5. EnemyController

Az Enemy osztály felelős az ellenséges karakterek mesterséges intelligenciájának megvalósításáért. Ez magában foglalja az ellenség állapotainak kezelését (járőrözés, üldözés, véletlenszerű mozgás, halál), a játékos észlelését (látás, hallás), a mozgást (beleértve az útvonalkeresést és az akadálykerülést), a harcot (lövés és közelharc), valamint az életerő és a halál kezelését.

Állapotgép (State Machine):

Az ellenség viselkedését egy egyszerű állapotgép (State enum) vezérli, amely a következő állapotokat definiálja:



- **Patrol (Járőrözés):** Az ellenség egy előre meghatározott vagy egyszerű mintát követve mozog (pl. egyenesen halad, falnál 90 fokban elfordul), amíg nem észleli a játékost.
- **Pursue (Üldözés):** Az ellenség aktívan követi és támadja a játékost, amint észlelte. Az A* algoritmust használja az útvonaltervezéshez.
- **Random (Véletlenszerű):** Az ellenség céltalanul bolyong a pályán, véletlenszerű irányokba fordulva és rövid ideig várakozva. Ebben az állapotba kerülhet például, miután elvesztette a játékost szem elől.
- **Dead (Halott):** Az ellenség meghalt, minden aktív viselkedése leáll.

Az állapotváltásokat elsősorban a HandleSightAndState metódus kezeli, a játékos láthatósága (CanSeePlayer), hallása (HearSound), a játékos halála és az enemyData.forgetTime (az idő, ami után az ellenség „elfelejti” a játékost) alapján.

3.3.5.1. Inicializálás (Start):

- Lekéri a hozzárendelt EnemyData ScriptableObject-et, amely az alapvető attribútumokat (életerő, sebesség stb.) tartalmazza. Hiba esetén letiltja a komponenst.
- Beállítja a kezdeti állapotot: 20% eséllyel Random, egyébként Patrol.
- Inicializálja az életerőt (currentHealth) az EnemyData alapján.
- Megkeresi a „Player” címkével ellátott játékobjektumot és eltárolja annak Transform komponensét. Hiba esetén naplózza.
- Lekéri a Rigidbody2D komponenst, beállítja Kinematic típusúra és lefagyasztja a forgást az X és Y tengelyen.
- Lekéri az EnemyEquipment komponenst a felszerelés kezeléséhez. Hiba esetén letiltja a komponenst.
- Lekéri az ököl fegyver adatait (fistWeaponData) az EnemyEquipment-től.
- Inicializálja a lövési és támadási időzítőket (nextShootTime, lastAttackTime), figyelembe véve a fegyver tűzgyorsaságát és egy véletlenszerű reakcióidőt (minReactionTime, maxReactionTime).
- Véletlenszerűen eldönti a járőrözés közbeni fordulás irányát (turnRight).
- Beállítja a GameObject rétegét „Enemy”-re.
- Inicializálja az A* útvonalkereséshez és az elakadás-észleléshez (stuck detection) szükséges változókat.

3.3.5.2. Alapvető Logika (Update, FixedUpdate, HandleSightAndState):

Update:

Felelős a magas szintű döntéshozatalért és időzítésekért.

- Ellenőrzi, hogy az ellenség vagy a játékos halott-e; ha igen, leállítja a logikát.
- Kezeli az elakadás-észlelést (stuck detection): Rendszeres időközönként (stuckCheckTime) ellenőrzi, hogy az ellenség pozíciója változott-e számottevően (stuckThreshold). Ha több egymást követő ellenőrzés során elakadt (consecutiveStuckFrames >= stuckFrameThreshold), megpróbálja újraszámolni az útvonalat (ideiglenesen megnövelt nodeSize-zal), végső esetben pedig egy véletlenszerű irányba indul (GetClearRandomAngle).
- Meghívja a HandleSightAndState metódust az állapotváltások kezelésére.
- Pursue állapotban kezeli a támadási logikát:
 - Prioritizálja a közelharcot (MeleeAttack), ha a játékos az ököl hatótávolságán belül van (fistWeaponData.range) és az időzítés (lastAttackTime) engedi.
 - Ha nem közelharcban van, és a játékos látható (CanSeePlayer), lő (Shoot), ha a fegyver engedi (canShoot), van lőszer (HasAmmo), és a reakcióidő (nextShootTime) letelt. Ha nincs lőszer, üres tár hangot játszik le (PlayEmptyClickSound) maximum háromszor.
- Random állapotban kezeli a várakozási időzítőt (waitTimer).
- Patrol és Random állapotban simított forgást (Quaternion.Slerp) alkalmaz a targetRotation felé.
- Pursue állapotban periodikusan (pathUpdateTime) újraszámolja az A* útvonalat (CalculatePath).

FixedUpdate:

Felelős a fizikai alapú mozgásért és az akadálykerülésért. A FixedUpdate használata biztosítja a konzisztens mozgást a fizikai motorral, függetlenül a képfrissítési gyakoriságtól.

Pursue állapot:

- Követi a kiszámított A* útvonalat (path). Meghatározza az irányt a következő csomópont (path[currentPathIndex]) felé.
- Alkalmaz egy akadály- és ellenségtaszító erőt (CalculateRepulsion) a közvetlen környezet elkerülésére.
- A Rigidbody2D.MovePosition segítségével mozgatja az ellenséget a célirányba.
- Ellenőrzi, hogy elérte-e az aktuális csomópontot (pathNodeReachedDistance), és lépteti az indexet (currentPathIndex).
- Optimalizálásként átugorhat csomópontokat, ha egy távolabbi csomópont közvetlenül látható (CanSeePoint).
- Lassít a kanyarok előtt a természetesebb mozgás érdekében (a sebességszorzó a következő irányváltozás szögétől függ).
- Ellenőrzi a lehetséges ütközést (WouldCollide) a következő pozícióval. Ha ütközne, megpróbál kitérni egy kissé eltérített irányba.
- Ha nincs érvényes útvonal, közvetlenül a játékos felé mozog taszítással.
- Simítottan forog (Quaternion.Slerp) a mozgás iránya felé.

Patrol állapot:

- Ha éppen fordulás utáni szünetben van (isPausedAfterTurn), nem mozog.
- Egyenesen mozog a patrolDirection irányába, figyelembe véve a taszítást (CalculateRepulsion).
- Physics2D.Raycast segítségével ellenőrzi, van-e fal előtte (safetyDistance).
- Ha falat észlel, véletlenszerűen 90 fokkal jobbra vagy balra fordul (turnRight), beállítja az új targetRotation-t és patrolDirection-t, majd egy véletlen hosszúságú szünetet tart (turnPauseTimer, minTurnPauseDuration, maxTurnPauseDuration).
- Ha nem ütközne (WouldCollide), mozog a MovePosition segítségével.

Random állapot:

- Ha nem várakozik (isWaiting), a patrolDirection irányába mozog taszítással.
- Ha falat észlel, új, tiszta véletlenszerű irányt választ (GetClearRandomAngle), beállítja a targetRotation-t és patrolDirection-t, majd véletlen ideig várakozik (waitTimer).
- Ha nem ütközne, mozog.

Dead állapot:

- Leállítja a mozgást, ha az ellenség halott, vagy ha halott játékost üldöz.

HandleSightAndState:

- Ellenőrzi a játékos láthatóságát (CanSeePlayer).
- Ha a játékos láthatóvá válik:
 - Beállítja a hasSpottedPlayer flag-et.
 - Rögzíti az észlelés idejét (lastSpottedTime).
 - Reseteli a hasReacted flag-et.
 - Beállít egy véletlenszerű reakcióidőt (Random.Range(minReactionTime, maxReactionTime)), ami után az ellenség először lőhet (nextShootTime) vagy támadhat (lastAttackTime).
 - Ha az állapot nem Pursue volt, átvált Pursue-ra, és kényszeríti az útvonal újraszámítását (lastPathUpdateTime = -pathUpdateTime).
- Ha a játékos nem látható vagy halott:
 - Ha a forgetTime letelt az utolsó észlelés óta, és az ellenség Pursue állapotban volt: Visszavált Random állapotba (ha a játékos él) vagy Patrol állapotba (ha a játékos halott), törli az útvonalat, megállítja a mozgást, és reseteli a hasSpottedPlayer, hasReacted flag-eket.
 - Ha a játékos meghal, miközben üldözik (de a forgetTime még nem telt le), az ellenség azonnal Patrol állapotba vált.

3.3.5.3. Útvonalkeresés Implementáció: A*

Az ellenség Pursue állapotban az A* algoritmust használja. Az A* egy széles körben elterjedt és hatékony útvonalkereső algoritmus, amely garantáltan megtalálja a legrövidebb utat (ha létezik) egy súlyozott gráfban vagy rácson.

Működése:

- Rács és Csomópontok: A világot egy logikai rácsra osztja. Minden cella egy potenciális PathNode.
- Adatszerkezetek: openSet (lista a még vizsgálándó csomópontokról), closedSet (HashSet a már vizsgált csomópontokról), nodes (Dictionary a csomópontok adatainak tárolására: gCost, hCost, parent).
- Költségszámítás:
 - gCost: Valós költség a start csomóponttól az adott csomópontig.
 - hCost: Heurisztikus becslés a költségről az adott csomóponttól a célig (itt Manhattan távolság: HeuristicDistance).
 - fCost: Teljes becsült költség (gCost + hCost). Az algoritmus mindig a legalacsonyabb fCost-ú csomópontot választja ki az openSet-ből.
- Szomszédok vizsgálata: Ellenőrzi a 8 szomszédos csomópontot (vízszintes, függőleges, átlós). Az átlós mozgás költsége nagyobb (14 vs 10).
- Járhatóság (IsPositionBlocked): Physics2D.OverlapCircle segítségével ellenőrzi, hogy egy csomópont pozíciója ütközik-e fallal. Tartalmaz logikát az átlós „sarokvágás” megakadályozására is (ha két szomszédos, nem átlós csomópont blokkolt, az átlós út sem járható).
- Útvonal rekonstrukció (ReconstructPath): Miután elérte a célcsomópontot, visszafelé követi a parent referenciákat a startig, hogy összeállítsa az útvonalat.
- Útvonal egyszerűsítés (SimplifyPath): Optimalizálja a rekonstruált útvonalat. Eltávolítja a felesleges köztes pontokat, ha két nem szomszédos pont között egyenes vonalú akadálymentes út van (CanSeePoint segítségével ellenőrizve). Ez simább és természetesebb mozgást eredményez.
- Optimalizálás: Ha a játékos közvetlenül látható (CanSeePlayer), az A* számítást kihagyja, és egyenes útvonalat használ. Az útvonal csak periodikusan frissül (pathUpdateTime), nem minden képkockában, ami kíméli a processzort.

3.3.5.4. Harc Logika (Shoot, MeleeAttack, AttackAnimation):

Célzás:

Mindkét támadástípus előtt az ellenség a játékos felé fordul a `transform.up = directionToPlayer` beállításával. A forgást ezután a Z tengelyre korlátozza.

Shoot (Lövés):

- Ellenőrzi a feltételeket (van fegyver, löhető, van lőszer, nem halott stb.).
- Felhasználja a lőszert (`currentWep.UseAmmo`). Ha nincs, üres tár hangot játszik (`PlayEmptyClickSound`).
- Létrehozza a lövedék (`projectilePrefab`) és a torkolattűz (`muzzleFlashPrefab`) példányait a megfelelő pozícióban és irányban.
- Kezeli a fegyver szórását (`spread`) és a sörétes puskák működését (`pelletCount > 1`), több lövedéket hozva létre egy lövésre, a `spreadAngle` alapján elosztva az irányokat.
- Beállítja a létrehozott lövedék (Bullet komponens) paramétereit: sebesség, hatótáv, sebzés, a lövő (`SetShooter`), és a fegyver adatai (`SetWeaponData`).
- Lejátssza a lövés hangját (`shootSound`) véletlenszerű hangmagasság-változtatással.

MeleeAttack (Közelharc):

- Ellenőrzi a feltételeket (van fegyver, közelharci, nem halott).
- Elindítja az `AttackAnimation` Coroutine-t a vizuális visszajelzéshez.
- `Physics2D.OverlapCircle` segítségével keresi a játékost a támadás hatókörén (`meleeWeapon.range`) belül, a fegyver előtti pontból (`attackOrigin`).
- Ha a játékos eltalálható (nincs fal közöttük - `Physics2D.Raycast` ellenőrzés), sebzést okoz (`playerController.TakeDamage`), vér effektust hoz létre, és beállítja a `didHit` flag-et.
- Lejátssza a találati (`hitSound`) vagy elvétési (`missSound`) hangot.

AttackAnimation (Coroutine):

- Időzítetten kezeli a közelharci támadás vizuális effektusát: lecseréli az ellenség sprite-ját a fegyver `attackSprite`-jára (vagy véletlenszerűen az `attackSprite2`-re, ha `useAlternatingSprites` igaz) a `weapon.attackDuration` idejére.

3.3.5.5. Érzékelés (CanSeePlayer, HearSound, OnTriggerEnter2D/Exit2D):

CanSeePlayer:

Kombinált módszert használ a játékos észlelésére:

- Először ellenőrzi, hogy a játékos a látómezőt (FOV) reprezentáló trigger collideren belül van-e (playerInFOV). Ezt az OnTriggerEnter2D és OnTriggerExit2D metódusok kezelik. Ez egy gyors, alacsony költségű előszűrés.
- Ha a játékos a triggerben van, egy Physics2D.Raycast-et lő a játékos felé, hogy ellenőrizze, nincs-e fal (wallLayer) a kettő között (tisztá rálátás).
- Ez a kétszintű ellenőrzés hatékonyabb, mintha folyamatosan raycastokat lőne minden irányba.

HearSound:

- Lehetővé teszi, hogy külső események (pl. játékos lövése) kiváltsák az ellenség figyelmét és Pursue állapotba léptessék, szimulálva a hanghallást. Ugyanúgy alkalmazza a reakcióidőt, mint a vizuális észlelés.

3.3.5.6. Sebződés és Halál (TakeDamage, Die, HealDamage):

TakeDamage:

- Csökkenti a currentHealth-et a kapott sebzéssel.
- Létrehoz egy vér effektust az ellenség pozíciójában.
- Ellenőrzi, hogy az életerő 0 alá esett-e. Ha igen, meghívja a Die metódust.
- Biztosítja, hogy a sebzés ne okozzon nem kívánt forgást (rb.freezeRotation).

Die:

- Beállítja az isDead flag-et és a State.Dead állapotot.
- Leállítja a mozgást és minden aktív Coroutine-t (pl. AttackAnimation).
- Jelzi a ScoreManager-nek az ellenség legyőzését.
- Beállítja a halál sprite-ot (enemyData.deathSprite) az EnemyEquipment segítségével.
- Átállítja a sprite renderelő rétegét („DeadEnemies”), hogy a holttestek más objektumok alatt jelenjenek meg.
- Módosítja a méretarányt (transform.localScale).

- Eldobja az ellenség fegyverét (ha nem az ökle volt), létrehozva annak pickupPrefab-ját a világban, és fizikai erőt (AddForce, AddTorque) alkalmaz rá.
- Kikapcsolja az ellenség Collider2D-jét.
- Átállítja a Rigidbody2D típusát Dynamic-ra (rövid időre), hogy a halált okozó lövés iránya (bulletDirection) alapján egy kis lökést (AddForce) kaphasson hátra. Az Invoke(nameof(StopAfterNudge), 0.2f) segítségével egy kis késleltetés után megállítja ezt a mozgást.

HealDamage:

- Visszaállítja az életerőt, de nem engedi túllépni a maximális értéket. A Boss típusú ellenség halhatatlanságához.

3.3.5.7. Segédmetódusok (WouldCollide, CalculateRepulsion, GetClearRandomAngle, PlayEmptyClickSound):

WouldCollide:

- Physics2D.OverlapCircle segítségével ellenőrzi, hogy egy adott pozíció ütközne-e fallal, figyelembe véve az ellenség sugarát (enemyRadius). A mozgás előtt használatos az ütközések elkerülésére.

CalculateRepulsion:

- Kiszámít egy taszító vektort a közeli falaktól és más ellenségektől.
- Több irányba (rayCount) lő rövid Physics2D.Raycast-eket a falak észlelésére, és Physics2D.OverlapCircleAll-t használ a közeli ellenségek (Enemy réteg) megtalálására.
 - A taszítás erőssége a távolságtól függ.
 - Az ellenségektől való taszítás erősebb, hogy megakadályozza a karakterek egymásba csúszását.

GetClearRandomAngle:

- Megpróbál találni egy olyan véletlenszerű irányt (szöget), amerre nincs közvetlen akadály egy bizonyos távolságon belül. Több próbálkozást tesz (maxAttempts), mielőtt feladná és egy teljesen véletlen szöget adna vissza. A Random állapotban használatos új irány választásakor.

PlayEmptyClickSound:

- Lejátszik egy hangot, amikor az ellenség lőni próbál, de nincs lőszere. Korlátozza az egymás utáni lejátszások számát (emptyClickSoundCount), hogy ne legyen zavaró.

3.3.6. EnemyEquipment

Az EnemyEquipment osztály felelős az ellenséges karakterek felszerelésének, elsősorban a fegyverük és a hozzá tartozó sprite kezeléséért. Ez a komponens biztosítja, hogy az ellenség rendelkezzen egy alapértelmezett fegyverrel, képes legyen új fegyvereket felvenni, és hogy a kinézete (sprite) mindig a jelenleg használt fegyvernek megfelelő legyen.

C «MonoBehaviour» EnemyEquipment	
□	enemySpriteRenderer : SpriteRenderer
□	defaultWeaponData : WeaponData
□	fistWeaponData : WeaponData
○	CurrentWeapon : WeaponData «get»
○	DefaultWeaponData : WeaponData «get»
○	FistWeaponData : WeaponData «get»
○ {event}	OnWeaponChanged : Action
●	Awake() : void
●	EquipWeapon(WeaponData newWeapon) : void
●	EquipWeaponFromPickup(WeaponPickup pickup) : void
●	SetSprite(Sprite newSprite) : void
●	UpdateSpriteToCurrentWeapon() : void

3.3.6.1. Inicializálás (Awake):

- Ellenőrzi és lekéri a szükséges SpriteRenderer komponenst. Ha nem található, hibát naplóz és letiltja a komponenst.
- Ellenőrzi a fistWeaponData (ököl) meglétét. Ha hiányzik, figyelmeztetést naplóz, és megpróbálja az alapértelmezett fegyvert (defaultWeaponData) használni.
- Ellenőrzi a defaultWeaponData meglétét. Ha hiányzik, figyelmeztetést naplóz. Ha van fistWeaponData, azt szereli fel alapértelmezettként. Ha egyik sincs, hibát naplóz és letiltja a komponenst.
- Ha a defaultWeaponData megvan, meghívja az EquipWeapon metódust az alapértelmezett fegyver felszerelésére. Ez biztosítja, hogy az ellenség mindig rendelkezzen valamilyen fegyverrel az induláskor.

3.3.6.2. Fegyver Felszerelése (EquipWeapon):

Ez a központi metódus az új fegyverek kezelésére.

- Ellenőrzi, hogy a kapott newWeapon nem null-e, és hogy különbözik-e a jelenlegitől (CurrentWeapon).
- Létrehoz egy másolatot (Instantiate) a kapott WeaponData ScriptableObject-ból, hogy minden ellenségnek saját, független fegyverpéldánya legyen.

- Inicializálja az új fegyverpéldány lőszerét (currentAmmo) a tárcapacitásra (magazineSize).
- Beállítja a CurrentWeapon referenciát az új példányra.
- Frissíti az ellenség sprite-ját az új fegyverhez (UpdateSpriteToCurrentWeapon).
- Kiváltja az OnWeaponChanged eseményt, értesítve más komponenseket (pl. az Enemy osztályt) a fegyverváltásról.

3.3.6.3. Sprite Kezelés (SetSprite, UpdateSpriteToCurrentWeapon)

SetSprite:

Lehetővé teszi a sprite közvetlen beállítását. Elsősorban speciális esetekben (pl. halál sprite beállítása az Enemy osztályban) vagy ideiglenes vizuális effektusokhoz (mint az AttackAnimation Coroutine) használatos.

UpdateSpriteToCurrentWeapon:

- Felelős azért, hogy az ellenség SpriteRenderer-ének sprite-ja megegyezzen a CurrentWeapon playerSprite (az ellenség által használt sprite) értékével.
- Ellenőrzi a SpriteRenderer meglétét.
- Megpróbálja beállítani a CurrentWeapon sprite-ját.
- Ha a CurrentWeapon vagy annak sprite-ja érvénytelen (null), megpróbál visszalépni (fallback) az alapértelmezett fegyver (defaultWeaponData) sprite-jára. Ha a CurrentWeapon is null volt, megpróbálja újból felszerelni az alapértelmezett fegyvert.
- Ha az alapértelmezett fegyver sprite-ja is érvénytelen, megpróbál visszalépni az ököl (fistWeaponData) sprite-jára. Ha a CurrentWeapon null volt, megpróbálja felszerelni az öklöt.
- Ha egyik sprite sem érvényes, hibát naplóz.

3.3.6.4. Fegyver Felvétele Pickupból (EquipWeaponFromPickup):

Segédmetódus, amely lehetővé teszi, hogy egy WeaponPickup objektumból származó WeaponData-t közvetlenül felszereljen az EquipWeapon meghívásával. Ezt jellemzően akkor hívja meg egy külső logika (pl. az Enemy osztály), amikor az ellenség interakcióba lép egy fegyver pickup objektummal.

3.3.7. BossEnemy

A BossEnemy osztály a „Boss”, vagyis Főellenség speciális ellenségtípusának a viselkedését definiálja. A BossEnemy fő jellegzetessége egy erőteljes rohamtámadás (dash attack), valamint egyedi sebződési szabályok (immunitás a távolsági fegyverekre). A rohamtámadást a PerformDashAttack Coroutine valósítja meg,

«MonoBehaviour» BossEnemy	
○ dashSpeed : float	
○ dashDuration : float	
○ dashCooldown : float	
○ dashTelegraphTime : float	
○ dashTelegraphColor : Color	
○ dashSound : AudioClip	
□ baseEnemy : Enemy	
□ player : Transform	
□ isDashing : bool	
● Awake() : void	
● Update() : void	
● TakeDamage(float amount, WeaponData weapon) : void	
● HandleThrownWeapon(WeaponData thrownWeapon) : void	
■ CanSeePlayer() : bool	
■ PerformDashAttack() : IEnumerator	

ennek indítását az Update metódus vezérli a megfelelő feltételek (időzítés, játékos láthatósága és távolsága) teljesülése esetén.

3.3.7.1. Inicializálás (Awake)

Az osztály példányosításakor az Awake metódus fut le, amely elvégzi a szükséges kezdeti beállításokat:

- Lekéri a kötelezően megkövetelt Enemy komponenst, valamint a Rigidbody2D, SpriteRenderer, AudioSource és EnemyEquipment komponenseket.
- Eltárolja a SpriteRenderer eredeti színét (originalColor), hogy a roham után visszaállíthassa azt.
- Megkeresi a „Player” címkével ellátott játékokobjektumot és eltárolja annak Transform komponensét.
- Beállítja a rohamtámadás kezdeti időzítését: véletlenszerűen meghatároz egy várakozási időt (currentCooldown) a dashCooldownMin és dashCooldownMax értékek között, és a lastDashTime értékét úgy állítja be, hogy az első roham a kezdeti várakozási idő letelte után azonnal lehetséges legyen.

3.3.7.2. Dash Logika (Update, CanSeePlayer)

Update:

Felelős a rohamtámadás indításának feltételeinek ellenőrzéséért minden képkockában:

- Ellenőrzi, hogy a Főellenség vagy a játékos halott-e; ha igen, a logika futása leáll.
- Ellenőrzi, hogy eltelt-e a szükséges várakozási idő (currentCooldown) az utolsó roham (lastDashTime) óta, hogy a Főellenség éppen nem rohamoz-e (!isDashing), és hogy látja-e a játékost a CanSeePlayer metódus szerint.
- Ha minden feltétel teljesül, elindítja a PerformDashAttack Coroutine-t a roham végrehajtásához.

CanSeePlayer:

Speciális feltételeket vizsgál a roham indításához:

- Ellenőrzi, hogy a játékos létezik-e.
- Kiszámolja a játékoshoz viszonyított irányt és távolságot.
- Csak akkor tekinti a játékost „láthatónak” a roham szempontjából, ha a távolság egy meghatározott intervallumon belül van (itt 5 és 20 egység között).
- Egy Physics2D.Raycast segítségével ellenőrzi, hogy van-e akadály (a baseEnemy.wallLayer rétegen) a Főellenség és a játékos között. Csak tiszta rálátás esetén ad vissza igaz értéket.

3.3.7.3. PerformDashAttack (Coroutine):

Végrehajtja a rohamtámadás teljes folyamatát lépésről lépésre.

- Beállítja az isDashing flag-et igazra, jelezve a roham aktív állapotát.
- Eltárolja a Rigidbody2D eredeti beállításait (típus, forgás befagyasztása, megkötések), hogy a roham végén visszaállíthassa azokat.
- Vizuális előrejelzés (Telegraphing): A SpriteRenderer színét megváltoztatja a dashTelegraphColor-ra, jelezve a játékosnak a közelgő támadást.
- Lejátszik egy hanghatást (dashSound), ez megegyezik a Bayonet elvétett támadásával.
- Vár a dashTelegraphTime ideig (yield return new WaitForSeconds).
- Ellenőrzi, hogy a játékos és a Főellenség még mindig élnek-e a várakozás után.
- Ha igen, meghatározza a roham irányát a játékos aktuális pozíciója felé.

- Beállítja a Főellenség sprite-jának irányát (transform.up) a roham irányába.
- Módosítja a Rigidbody2D beállításait a roham idejére: Dynamic típusúra állítja, kikapcsolja a gravitációt.
- A roham mozgását egy while ciklus valósítja meg, amely a dashDuration ideig tart. A FixedUpdate-ben beállított sebesség (rb.linearVelocity) végzi a tényleges mozgást. A Coroutine itt inkább az időzítésért és az állapotkezelésért felel.
- A roham időtartamának letelte után megállítja a mozgást (rb.linearVelocity = Vector2.zero).
- Visszaállítja a Rigidbody2D eredeti beállításait.
- Visszaállítja a SpriteRenderer eredeti színét.
- Az isDashing flag-et visszaállítja hamisra.
- Rögzíti a roham befejezésének idejét (lastDashTime).
- Új, véletlenszerű várakozási időt (currentCooldown) állít be a következő rohamhoz.

3.3.7.4. Fizikai Mozgás (FixedUpdate)

FixedUpdate:

- Ha a Főellenség éppen rohamoz (isDashing), és a játékos, valamint a Főellenség él, akkor a Rigidbody2D linearVelocity tulajdonságát közvetlenül beállítja.
- A sebességvektor iránya a játékos felé mutat, nagysága pedig a dashSpeed értékkel egyenlő. Ez egy direkt sebességbeállítás, amely felülír minden más fizikai hatást (pl. súrlódás, ütközésekből származó erők), biztosítva a gyors és megállíthatatlan rohamot.

3.3.7.5. Sebződés Kezelése (TakeDamage, HandleThrownWeapon)

A BossEnemy felülírja vagy kiegészíti az alap Enemy sebződési logikáját:

TakeDamage:

Ez a metódus egyedi implementációt kapott, ami csak erre az ellenség típusra vonatkozik.

- Ellenőrzi a sebzést okozó fegyvert (WeaponData).
 - Ha a fegyver nem közelharci (!weapon.isMelee), a Főellenség immunis: nem szenved el sebzést, és egy „Immune” részecskeeffektus jelenik meg.
 - Ha a fegyver közelharci, akkor továbbhívja az alap Enemy osztály TakeDamage metódusát a normál sebzés feldolgozására.

HandleThrownWeapon:

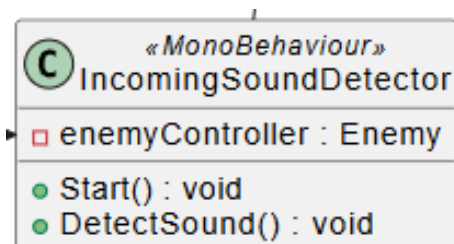
Kezeli az eldobott fegyverek által okozott találatokat.

- Ellenőrzi a sebzést okozó fegyvert (WeaponData).
 - Hasonlóan a TakeDamage-hez, ha az eldobott fegyver nem közelharci, immunitást jelez egy részecskeeffektussal.
 - Ha az eldobott fegyver közelharci, akkor fixen 1 sebzést okoz az Enemy.TakeDamage meghívásával. Ez egy speciális interakció az eldobott közelharci fegyverekkel.

3.3.8. IncomingSoundDetector

Ez egy egyszerű segédkomponens, amelynek elsődleges célja, a hanghatások által kiváltott jelzéseket továbbítani a szülő GameObjecten található Enemy komponens felé. Lényegében egy „hallgatóként” funkcionál, amely értesíti az ellenséget, ha a közelében „hang” történik,

lehetővé téve az ellenség számára, hogy reagáljon rá (például az üldözési állapotba (Pursue) váltással). Ez a komponens elősegíti a hangforrások és az ellenséges MI közötti lazább csatolást.



3.3.8.1. Inicializálás (Start):

- A GetComponentInParent<Enemy>() metódussal megkeresi a hierarchiában felfelé haladva az első Enemy komponenst. Ez feltételezi, hogy az IncomingSoundDetector vagy ugyanazon a GameObjecten van, mint az Enemy, vagy annak egy leszármazottján.
- Eltárolja a talált Enemy komponenst az enemyController változóban.
- Ha nem talál Enemy komponenst a szülő objektumok egyikén sem, hibaüzenetet naplóz.

3.3.8.2. Hang Észlelése (DetectSound):

Ez a publikus metódus szolgál a komponens fő funkciójának kiváltására:

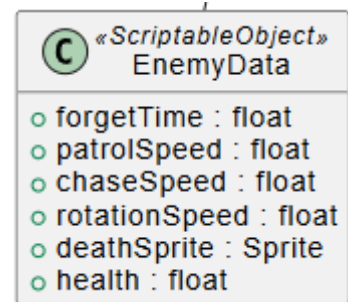
- Meghívásakor ellenőrzi, hogy az enemyController referencia érvényes-e (azaz sikerült-e Enemy komponenst találni a Start metódusban).
- Ha érvényes a referencia, meghívja az enemyController HearSound() metódusát.

Használat:

Az ellenség objektum olyan gyerek objektumára kell elhelyezni ezt a komponenst, aminek van egy trigger komponense (bármilyen Collider, aminek a Trigger tulajdonsága igazra van állítva). Ez a Collider az ellenség hallási hatótávolságát reprezentálja. Amikor egy hangforrás ebbe a triggerbe lép, a hangforrás szkriptje meghívhatja az IncomingSoundDetector DetectSound() metódusát.

3.3.9. EnemyData

Az EnemyData egy ScriptableObject típusú adatkonténer, amely az ellenséges karakterek alapvető attribútumait és viselkedési paramétereit tárolja. A ScriptableObject használata lehetővé teszi az ellenségadatok elkülönítését a konkrét ellenségpéldányoktól. A különböző ellenségi típusok mind más EnemyData-t használnak.



- Kódszerkesztés nélkül módosíthatóak az ellenségek tulajdonságai (sebesség, életerő stb.) az EnemyData assetek szerkesztésével.
- Az EnemyController felelős a viselkedésért, míg az EnemyData szolgáltatja az ehhez szükséges paramétereket.

Az EnemyData asseteket a „Create/Game/EnemyData” menüpontjával lehet létrehozni.

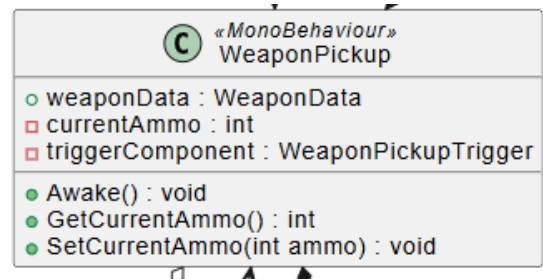
Adattagok:

Az EnemyData osztály a következő publikus változókat tartalmazza, amelyeket az EnemyController használ fel az ellenség viselkedésének meghatározásához:

- forgetTime: (float) Az az időtartam másodpercben, amelynek el kell telnie azután, hogy az ellenség utoljára látta a játékost, mire „elfelejti” őt és abbahagyja az üldözést
- patrolSpeed: (float) Az ellenség mozgási sebessége, amikor Patrol (Járőrözés) állapotban van.
- chaseSpeed: (float) Az ellenség mozgási sebessége, amikor Pursue (Üldözés) állapotban van. Magasabb, mint a patrolSpeed.
- rotationSpeed: (float) Az ellenség forgási sebessége. Meghatározza, milyen gyorsan tud az ellenség a mozgás iránya vagy a célpontja felé fordulni.
- deathSprite: (Sprite) Az a sprite (2D grafika), amelyet az ellenség megjelenít, miután meghalt (Die metódus). Ezt az EnemyEquipment komponens használja a sprite lecserélésére.
- health: (float) Az ellenség kezdeti és maximális életeréje. Ezt az értéket használja az EnemyController az inicializáláskor (Start) a currentHealth beállítására, és a TakeDamage metódusban a sebződés számításához.

3.3.10. WeaponPickup

A WeaponPickup komponens egy a játékvilágban elhelyezett, felvehető fegyvert reprezentál. Felelős a fegyver vizuális megjelenítéséért (a hozzárendelt SpriteRenderer segítségével, amelyet a RequireComponent attribútum meg is követel), a hozzá tartozó fegyveradatok (WeaponData) tárolásáért, valamint annak nyomon követéséért, hogy mennyi lőszer van az adott fegyverpéldányban. (Az ellenség által eldobott fegyverek mindig tele tárral képződnek, viszont a játékos által eldobottakban megjegyződik a maradt lövedékek száma). Együttműködik egy „TriggerZone” nevű gyermekobjektumon található WeaponPickupTrigger szkripttel, amely a játékos közelségének érzékeléséért és a felvételi folyamat elindításáért felelős.



3.3.10.1. Inicializálás (Awake)

- Lekéri a kötelező SpriteRenderer komponenst a vizuális megjelenítéshez.
- Megkeresi a „TriggerZone” nevű gyermekobjektumot a transform.Find segítségével. Ez a gyermekobjektum várhatóan tartalmaz egy Collider2D komponenst (triggerként beállítva) a játékos érzékeléséhez.
- Lekéri a WeaponPickupTrigger komponenst a „TriggerZone” objektumról. A külön trigger zóna használata lehetővé teszi, hogy a játékos fel tudja venni a fegyvereket, de ne ütközzön velük. (Ez egy gyakori eljárás, a szülő objektum nem trigger Collider-je kezeli az ütközést a falakkal)
- Ellenőrzi, hogy a weaponData (a fegyver tulajdonságait leíró ScriptableObject) hozzá van-e rendelve és tartalmaz-e érvényes playerSprite-ot.
- Ha a weaponData érvényes:
 - Beállítja a GameObject nevét a hierarchiában a fegyver nevére (weaponData.weaponName) és a „Pickup” utótagra az azonosíthatóság érdekében (pl. „Makarov_Pickup”).
 - Inicializálja a currentAmmo változót. Ha a currentAmmo értéke a kezdeti -1 (ami azt jelzi, hogy még nem lett külsőleg beállítva), akkor a fegyver tárának maximális méretére (weaponData.magazineSize) állítja be.
- Ha a weaponData vagy annak playerSprite-ja hiányzik, figyelmeztetést naplóz.

3.3.10.2. Lőszer Kezelése (GetCurrentAmmo, SetCurrentAmmo)

Ezek a metódusok lehetővé teszik más szkriptek számára, hogy lekérdezzék vagy beállítsák az adott fegyver-pickupban lévő lőszer mennyiségét.

GetCurrentAmmo:






Visszaadja a currentAmmo futási idejű változóban tárolt aktuális lőszerszámot.

SetCurrentAmmo:

Lehetővé teszi a currentAmmo értékének külső beállítását. A PlayerController hívja meg, amikor eldobja az aktuális fegyverét.

3.3.11. WeaponPickupTrigger

A WeaponPickupTrigger komponens a WeaponPickup GameObject egy gyermekobjektumán („TriggerZone”) helyezkedik el, az eldobott fegyverek speciális interakcióinak

	«MonoBehaviour» WeaponPickupTrigger
	throwTime : float
	isThrown : bool
	OnEnable() : void
	OnTriggerEnter2D(Collider2D collision) : void

(az ellenségekkel való ütközéskor) és a játékos fegyverfelvételének kezeléséért felelős. Működéséhez szükséglet, hogy a GameObject, amelyen található, rendelkezzen egy Collider2D komponenssel, amelynek Is Trigger tulajdonsága be van kapcsolva.

3.3.11.1. Inicializálás és Dobás Észlelése (OnEnable)

Amikor a „TriggerZone” GameObject (és ezzel együtt ez a komponens) aktívvá válik (például amikor a szülő WeaponPickup GameObject létrejön a világban), az OnEnable metódus lefut:

- Megpróbálja lekérni a szülő objektum WeaponPickup komponensét.
- Ha talál WeaponPickup komponenst (ami azt jelenti, hogy ez a trigger egy érvényes fegyver-pickuphoz tartozik), akkor beállítja az isThrown flag-et true-ra és rögzíti az aktuális időt (Time.time) a throwTime változóban. Tehát az OnEnable lefutása jelzi a dobás pillanatát.

3.3.11.2. Ütközéskezelés Eldobás Után (OnTriggerEnter2D)

Ez a metódus felelős az eldobott fegyver interakcióinak kezeléséért, amikor a trigger zóna más objektumokkal ütközik:

- Először ellenőrzi, hogy az ütközés az eldobást követő rövid időablakon (weaponKnockbackWindow, alapértelmezetten 0.2 másodperc) belül történt-e (`isThrown && Time.time - throwTime <= weaponKnockbackWindow`). Ez biztosítja, hogy csak a frissen eldobott, még „repülő” fegyverek váltsák ki a speciális hatásokat, a földön fekvő, statikus pickupok ne.
- Ezután ellenőrzi, hogy az ütköző objektum az „Enemy” réteghez tartozik-e (`collision.gameObject.layer == LayerMask.NameToLayer("Enemy")`).
- Ha ellenséggel ütközött:
 - Lekéri az ellenség Enemy komponensét, és ellenőrzi, hogy az ellenség él-e (`!enemy.isDead`).
 - Lekéri az eldobott fegyver WeaponData-ját a szülő WeaponPickup komponensből.
 - Ellenőrzi, hogy az ellenség egy BossEnemy-e.
 - Ha igen, meghívja a `bossEnemy.HandleThrownWeapon(thrownWeaponData)` metódust, amely a Főellenség specifikus logikáját hajtja végre az eldobott fegyverekre
 - A return utasítás megakadályozza a további feldolgozást.
 - Ha nem Főellenségről van szó:
 - Ha az eldobott fegyver közelharci (`thrownWeaponData.isMelee`)
 - 1 egységnyi sebzést okoz az ellenségnek az `enemy.TakeDamage(1)` meghívásával. A return itt is megállítja a további feldolgozást.
 - Ha az eldobott fegyver nem közelharci, kiváltja a lefegyverzés mechanikáját:
 - Lekéri az ellenség EnemyEquipment komponensét.
 - Ellenőrzi, hogy az ellenség rendelkezik-e felszerelt fegyverrel (nem az öklét használja), és hogy ennek a fegyvernek van-e hozzárendelt pickupPrefab-ja (azaz eldobható-e).
 - Ha a feltételek teljesülnek:

- Létrehoz egy új WeaponPickup példányt az ellenség aktuális fegyveréből (enemyWeapon.pickupPrefab) az ellenség pozíciójában, véletlenszerű forgatással.
- Alkalmaz egy véletlenszerű irányú és erősségű impulzust (AddForce, AddTorque) az újonnan létrehozott pickup Rigidbody2D-jére, hogy az realisztikusan repüljön el.
- Az
 enemyEquipment.EquipWeapon(enemyEquipment.FistWeaponData)
 meghívásával arra kényszeríti az ellenséget, hogy váltson az öklére, gyakorlatilag lefegyverezve őt.

3.3.12. WeaponData

A WeaponData egy ScriptableObject típusú adatkonténer, amely a játékban található fegyverek összes releváns tulajdonságát és viselkedési paraméterét központosítottan tárolja. Hasonlóan az EnemyData-hoz, a ScriptableObject használata itt is lehetővé teszi a fegyveradatok elkülönítését a fegyvereket használó komponensektől (pl. PlayerController, EnemyController, EnemyEquipment) és a fegyverek fizikai reprezentációjától a játékvilágban (WeaponPickup). Ez a megközelítés kulcsfontosságú a játék fegyverrendszerének rugalmassága és karbantarthatósága szempontjából.

A ScriptableObject használatának előnyei:

- Új fegyvertípusok vagy meglévők variációinak létrehozása egyszerűen egy új WeaponData asset létrehozásával és annak paramétereinek beállításával történik a Unity Editorban, kódírás nélkül.

C «ScriptableObject» WeaponData	
○	weaponName : string
○	playerSprite : Sprite
○	weaponIcon : Sprite
○	pickupPrefab : GameObject
○	canShoot : bool
○	isMelee : bool
○	isSilent : bool
○	projectilePrefab : GameObject
○	fireRate : float
○	isFullAuto : bool
○	damage : float
○	bulletOffset : float
○	bulletOffsetSide : float
○	range : float
○	bulletSpeed : float
○	spread : float
○	magazineSize : int
○	pelletCount : int
○	spreadAngle : float
○	shootSound : AudioClip
○	emptyClickSound : AudioClip
○	shootShakeDuration : float
○	shootShakeMagnitude : float
○	muzzleFlashPrefab : GameObject
○	muzzleFlashDuration : float
○	hitSound : AudioClip
○	missSound : AudioClip
○	attackSprite : Sprite
○	attackSprite2 : Sprite
○	useAlternatingSprites : bool
○	attackDuration : float
Runtime	
○	currentAmmo : int
●	HasAmmo() : bool
●	UseAmmo() : bool
■	OnEnable() : void

- A fegyverek sebzésének, tűzgyorsaságának, tárkapacitásának és egyéb tulajdonságainak módosítása egyetlen helyen, a megfelelő WeaponData assetben elvégezhető, ami jelentősen megkönnyíti a játékegyensúly finomhangolását.
- Az adatok (mit csinál a fegyver) és a logika (hogyan használja a karakter a fegyvert, hogyan működik a lövedék) szétválasztása hozzájárul a rendezettebb és könnyebben bővíthető kódhoz.

Az WeaponData asseteket a Unity Editor „Assets/Create/Game/Weapon Data” menüpontjával lehet létrehozni.

3.3.12.1. Adattagok:

Az osztály számos publikus változót tartalmaz, amelyek a fegyver különböző aspektusait definiálják:

Azonosítás és Megjelenés:

- `weaponName`: (string) A fegyver neve
- `playerSprite`: (Sprite) Az a sprite, amelyet a fegyvert viselő karakter megjelenít, amikor ez a fegyver van aktívan felszerelve.
- `weaponIcon`: (Sprite) A fegyver ikonja. Jelenleg nincs használatban, de UI elemként a töltényszámláló mellett lehetne megjeleníteni.
- `pickupPrefab`: (GameObject) Az a Prefab, amely példányosításra kerül a játékvilágban, amikor ez a fegyver eldobásra kerül.

Viselkedési Jelzők:

- `canShoot`: (bool) Meghatározza, hogy a fegyver képes-e lőni
- `isMelee`: (bool) Explicit jelző, hogy a fegyver közelharci-e. Ez befolyásolja a támadási logikát (MeleeAttack metódus hívódik meg Shoot helyett)
- `isSilent`: (bool) Jelzi, hogy a fegyver használata (lövés) hangtalan-e.

Lövési és Lövedék Tulajdonságok:

- `projectilePrefab`: (GameObject) A lövedék Prefabja, amelyet a fegyver kilő. Csak akkor releváns, ha `canShoot` igaz.
- `fireRate`: (float) A fegyver tűzgyorsasága, lövés/másodperc mértékegységben.

- **isFullAuto:** (bool) Jelzi, hogy a fegyver teljesen automatikus-e. Ha true, a tüzelés gomb nyomva tartásával folyamatosan lehet lőni a fireRate ütemében. Ha false, minden lövéshez külön gombnyomás szükséges.
- **damage:** (float) Az egy lövedék vagy egy közelharcos támadás által okozott sebzés mértéke.
- **bulletOffset:** (float) Eltolás mértéke előre, ahol a lövedék és a torkolattűz effektus létrejön.
- **bulletOffsetSide:** (float) Oldalirányú eltolás a lövedék keletkezési pontjához.
- **range:** (float) A fegyver hatótávolsága. Távolsági fegyvereknél a lövedék maximális távolsága, ameddig sebzést okoz. Közelharcos fegyvereknél a támadás maximális távolságát jelenti a karaktertől.
- **bulletSpeed:** (float) A kilőtt lövedék sebessége.
- **spread:** (float) A fegyver szórásának mértéke. Meghatározza, mennyire térhet el a lövedék iránya a célzott iránytól. Nagyobb érték nagyobb szórást jelent.
- **magazineSize:** (int) A fegyver tárának kapacitása. Közelharcos fegyvereknél általában irreleváns.

Sörétes Puska Specifikus Beállítások:

- **pelletCount:** (int) Az egy lövéssel kilőtt sörétek (lövedékek) száma. Alapértelmezetten 1. Ha 1-nél nagyobb, a fegyver sörétes puskaként viselkedik, több lövedéket hoz létre egyetlen lövésre.
- **spreadAngle:** (float) A sörétek szóródásának szöge. Meghatározza, hogy a pelletCount számú lövedék milyen széles szögben terül szét a célzott irány körül. Csak akkor releváns, ha pelletCount > 1.

Effektek:

- shootSound: (AudioClip) A hang, amelyet a fegyver elsütésekor lejátszik.
- emptyClickSound: (AudioClip) A hang, amelyet akkor játszik le, ha a játékos/ellenség megpróbál lőni egy üres tárral.
- shootShakeDuration: (float) A képernyőrázkódás időtartama másodpercben, amelyet a fegyver elsütése kivált.
- shootShakeMagnitude: (float) A képernyőrázkódás erőssége.
- muzzleFlashPrefab: (GameObject) A torkolattűz részecskeeffektus Prefabja, amely a lövés pillanatában jön létre a bulletOffset által meghatározott pozícióban.
- muzzleFlashDuration: (float) A torkolattűz effektus élettartama másodpercben.

Közelharci Fegyver Specifikus Beállítások:

- hitSound: (AudioClip) A hang, ami akkor játszik le, ha a közelharci támadás eltalál egy ellenséget.
- missSound: (AudioClip) A hang, ami akkor játszik le, ha a közelharci támadás nem talál el semmit.
- attackSprite: (Sprite) Az a sprite, amelyet a karakter megjelenít a közelharci támadás animációja közben.
- attackSprite2: (Sprite) Egy alternatív támadási sprite.
- useAlternatingSprites: (bool) Ha igaz, a közelharci támadások során váltakozva, véletlenszerűen használja az attackSprite-ot és az attackSprite2-t.
- attackDuration: (float) A közelharci támadás animációjának időtartama másodpercben.

Futásidejű Adatok:

- currentAmmo: (int) A fegyver aktuális lőszerszáma a tárral. Ez egy futásidejű állapot, amelyet a játék során a UseAmmo metódus csökkent. A [System.NonSerialized] attribútum biztosítja, hogy ez az érték ne kerüljön mentésre az asset részeként a Unity Editorban.

3.3.12.2. Inicializálás és Segédmetódusok:

OnEnable:

Ez a metódus automatikusan meghívódik, amikor a ScriptableObject létrejön.

- A currentAmmo értékét inicializálja a magazineSize-ra.

HasAmmo:

- Visszaadja, hogy van-e még lőszer a fegyverben.
- Közelharc fegyverek esetén mindig True-t ad vissza, mivel azoknak nincs szükségük lőszerre.
- Egyébként ellenőrzi, hogy a currentAmmo nagyobb-e nullánál.

UseAmmo:

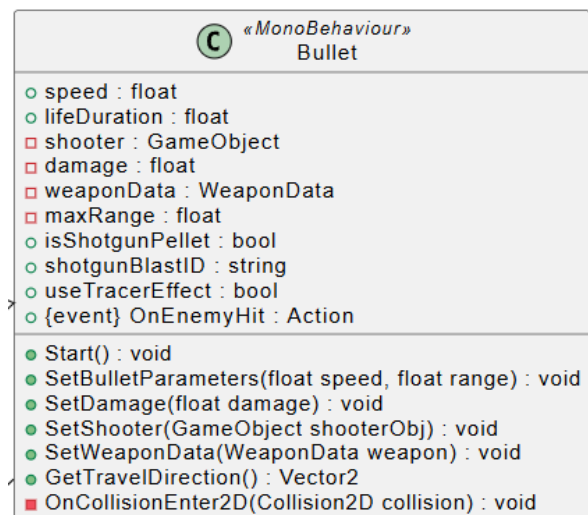
- Csökkenti a currentAmmo értékét eggyel.
- Visszatérési értéke jelzi, hogy sikeres volt-e a lőszer felhasználása (azaz volt-e lőszer).
- Közelharc fegyverek esetén nem csinál semmit, és mindig True-t ad vissza.

3.3.13. Bullet

A Bullet osztály felelős a kilőtt lövedékek viselkedésének szimulálásáért. Ez magában foglalja a lövedék mozgását, élettartamát, ütközésérzékelését, sebzésokozását, valamint a vizuális hatások kezelését

3.3.13.1. Inicializálás (Start):

- Lekéri a szükséges komponenseket: Rigidbody2D, SpriteRenderer, Collider2D.
- Beállítja a Rigidbody2D kezdeti sebességét (linearVelocity) a lövedék előre (transform.up) iránya és a speed paraméter alapján.
- Eltárolja a mozgás irányát (travelDirection) későbbi felhasználásra (pl. sebzés irányának átadása).



- Beállítja a lövedék automatikus megsemmisítését a `lifeDuration` időtartam után a `Destroy` függvénnyel. Ez megakadályozza, hogy a lövedékek a végtelenségig a pályán maradjanak.
- Eltárolja a lövedék indulási pozícióját (`startPosition`) a hatótávolság ellenőrzéséhez.
- Speciális logika sörétes lövedékek (`isShotgunPellet`) esetén:
 - Ha a lövedék egy sörétes lövés része és rendelkezik egyedi azonosítóval (`shotgunBlastID`), rögzíti a létrehozás idejét a `shotgunBlastTimes` statikus szótárban (ha még nem szerepel).
 - Ha ez az első lövedék ebből a lövésből (azaz az azonosító még nem szerepel a `shotgunHitsRecorded` szótárban), elindít egy `Coroutine`-t (`CleanupOldShotgunEntries`) a régi bejegyzések későbbi törlésére a statikus szótárból.
- Nyomjelző effektus (`useTracerEffect`) inicializálása:
 - Ha engedélyezve van, lekéri a `LineRenderer` komponenst.
 - Beállítja a `LineRenderer` tulajdonságait: pozíciók száma (2), szélesség (`tracerWidth`), anyag (alapértelmezett `sprite shader`), és egy véletlenszerűen módosított szín (`RandomizeColor`, `tracerColor`, `colorVariation`).
 - Beállítja a vonal kezdő- és végpontját az indulási pozícióra.
 - Inicializálja a nyomjelző elhalványulásának időzítőjét (`tracerTimer`).

3.3.13.2. Paraméterek Beállítása (`SetBulletParameters`, `SetDamage`, `SetShooter`, `SetWeaponData`):

Ezek a publikus metódusok beállítják a lövedék futásidejű tulajdonságait a példányosítás után:

`SetBulletParameters`:

- Beállítja a lövedék sebességét (`speed`) és maximális hatótávolságát (`maxRange`), majd frissíti a `Rigidbody2D` sebességét.

`SetDamage`:

- Beállítja a lövedék által okozott sebzést (`damage`).

SetShooter:

- Eltárolja a lövedéket kilövő GameObject referenciáját (shooter). Az ön-ütközés elkerülésére és annak eldöntésére, hogy ki lőtt ki.

SetWeaponData:

- Eltárolja a lövedéket kilövő fegyver WeaponData referenciáját. Ez szükséges például a BossEnemy sebzskezeléséhez, amely ellenőrzi a fegyver típusát.

3.3.13.3. Mozgás, Hatótáv és Nyomjelző Frissítése (Update)

Update:

- Hatótávolság ellenőrzése: Kiszámítja a megtett távolságot és összehasonlítja a maxRange-gel. Ha a lövedék túllépte a hatótávolságot, beállítja az isOutOfRange flag-et true-ra. Ez a flag később az ütközéskezelésnél kerül felhasználásra, hogy a hatótávon túli lövedékek ne okozzanak sebzést.
- Nyomjelző frissítése (useTracerEffect esetén):
 - A LineRenderer második pontját (SetPosition(1, ...)) a lövedék aktuális pozíciójára állítja, így a vonal követi a lövedéket.
 - Kezeli a nyomjelző elhalványulását: Csökkenti a tracerTimer-t. Ha az időzítő lejár, fokozatosan csökkenti a LineRenderer színének alfa értékét amíg az teljesen átlátszóvá nem válik.

3.3.13.4. Lövedék Deaktiválása (DisableBullet)

Ez a privát metódus felelős a lövedék „kikapcsolásáért” ütközés után, anélkül, hogy azonnal megsemmisítené azt. Ez lehetővé teszi, hogy a nyomjelző elhalványulása befejeződjön¹.

- Ellenőrzi, hogy a lövedék nem lett-e már korábban deaktiválva (hasHitSomething).
- Beállítja a hasHitSomething flag-et true-ra.
- Kikapcsolja a SpriteRenderer-t és a Collider2D-t.
- Megállítja a Rigidbody2D mozgását (linearVelocity = Vector2.zero) és Kinematic típusúra állítja, hogy ne befolyásolják további fizikai erők.

3.3.13.5. Ütközéskezelés (OnCollisionEnter2D)

Akkor hívódik meg, amikor a lövedék Collider2D-je egy másik Collider2D-vel ütközik. Ez a lövedék logikájának központi része:

- Ellenőrzi, hogy a lövedék aktív-e még (!hasHitSomething).
- Kihagyja az ütközést, ha az a lövedéket kilövő objektummal (shooter) történt.

Egy switch utasítással kezeli a különböző ütközési célpontokat az ütköző objektum címkéje (tag) alapján:

- **Enemy:**
 - Ha ellenség lőtte a lövedéket, azonnal deaktiválja azt (DisableBullet), megakadályozva a baráti tüzet az ellenségek között.
 - Ha játékos lőtte: Lekéri az Enemy komponenst, és ellenőrzi, hogy az ellenség él-e és a lövedék hatótávon belül van-e (!isOutOfRange).
 - Találat naplózása (ScoreManager): Mielőtt sebzést okozna, naplózza a találatot a ScoreManager-ben, ha a lövő játékos volt.
 - Sörétes logika: Ha sörétes lövedékről van szó (isShotgunPellet), ellenőrzi a shotgunHitsRecorded statikus szótárat. Csak akkor naplózza a találatot a ScoreManager-ben, ha ehhez a shotgunBlastID-hez még nem történt naplózás, hogy egyetlen sörétes lövés (amely több lövedékből áll) csak egy találatként számítson a pontozásban, függetlenül attól, hány sörét találta el ugyanazt vagy különböző célpontokat.
 - Beállítja a hasRecordedHit flag-et is.
 - Meghívja az OnEnemyHit eseményt.
 - Sebzés alkalmazása:
 - Ellenőrzi, hogy az ellenség BossEnemy-e. Ha igen, meghívja annak speciális TakeDamage metódusát, átadva a weaponData-t is (az immunitás ellenőrzéséhez).
 - Ha normál ellenség, létrehoz egy vér effektust a találat pontján (collision.contacts[0].point), és meghívja az enemy.TakeDamage(damage) metódust.

- Halál kezelése: Ha a sebzés következtében az ellenség meghal (`enemy.isDead`), meghívja az `enemy.Die(travelDirection)` metódust, átadva a lövedék irányát, hogy az ellenség a megfelelő irányba „essen el”.
- Deaktiválja a lövedéket (`DisableBullet`).
- Player:
 - Ha a lövedék hatótávon belül van (`!isOutOfRange`), létrehoz egy vér effektust és sebzést okoz a játékosnak a `player.TakeDamage(1, travelDirection)` meghívásával (a sebzés értéke fixen 1, de lehetne a `damage` változó is), átadva a lövedék irányát is.
 - Deaktiválja a lövedéket.
- Environment (vagy bármi más):
 - Egyszerűen deaktiválja a lövedéket.

3.3.13.6. Tisztítás és Pontozás (`OnDestroy`):

Amikor a lövedék `GameObject` megsemmisül (akár a `lifeDuration` lejárta, akár más ok miatt), az `OnDestroy` metódus lefut:

- Ha a lövedéket a játékos (shooter) lőtte ki, és az nem talált el semmit (`!hasHitSomething`), értesíti a `ScoreManager`-t az elvétett lövésről (`ScoreManager.Instance.LogMissedShot()`).

3.3.13.7. Sörétes Találatok Kezelése és Tisztítás (CleanupOldShotgunEntries):

A sörétes puskák egy lövéssel több lövedéket (pellet) indítanak. Annak érdekében, hogy egyetlen lövés ne számítsion több találatnak a pontozásban, ha több sörét is eltalál egy vagy több célpontot, a rendszer egyedi azonosítót (shotgunBlastID) használ minden egyes lövéshez.

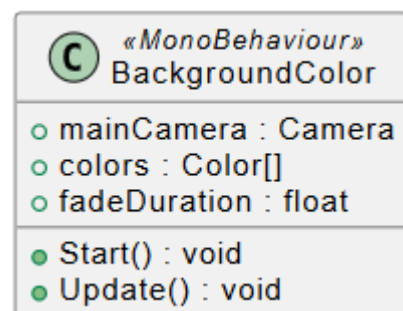
- Statikus Szótárak: Két statikus szótárat (shotgunHitsRecorded, shotgunBlastTimes) használ a rendszer. A shotgunHitsRecorded tárolja, hogy egy adott shotgunBlastID-jű lövéshez naplóztak-e már találatot. A shotgunBlastTimes tárolja, hogy mikor jött létre az adott lövés (az első sörét létrehozásakor). A statikus jelleg miatt ezek a szótárak az összes Bullet példány számára közősek.
- Találat Naplózása: Amikor egy sörét eltalál egy ellenséget, ellenőrzi a shotgunHitsRecorded szótárat. Ha az adott shotgunBlastID-hez még nincs bejegyzés, vagy a bejegyzés false, akkor naplózza a találatot a ScoreManager-ben, és a szótárban true-ra állítja az értéket. Ha már true az érték, a találatot nem naplózza újra.

CleanupOldShotgunEntries (Coroutine):

- Az első sörét létrehozásakor elindul ez a Coroutine, amely egy rövid várakozás (10 másodperc) után végignézi a shotgunBlastTimes szótárat, és eltávolítja azokat a bejegyzéseket (és a hozzájuk tartozó bejegyzéseket a shotgunHitsRecorded-ból is), amelyek egy meghatározott időnél (itt szintén 10 másodperc) régebbiek.

3.3.14. BackgroundColor

A BackgroundColor komponens egy egyszerű vizuális effektusért felelős: a játék fő kamerájának háttérszínét átmenettel változtatja egy előre definiált színsorozat elemei között.



3.3.14.1. Inicializálás (Start)

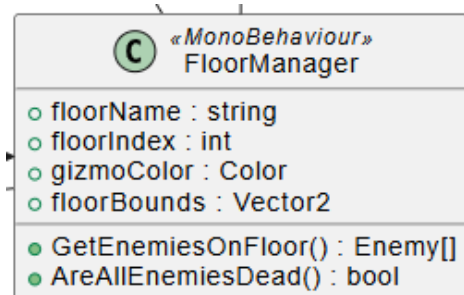
- Ellenőrzi, hogy a `mainCamera` változóhoz hozzá van-e rendelve egy kamera az Inspectorban. Ha nem (null), akkor megpróbálja automatikusan megtalálni és hozzárendelni a jelenet fő kameráját (`Camera.main`).
- Ellenőrzi, hogy a `colors` tömb (amely a ciklikusan váltakozó színeket tartalmazza) tartalmaz-e elemeket. Ha a tömb üres (`colors.Length == 0`), akkor alapértelmezettként egyetlen elemet ad hozzá: a fekete színt (`Color.black`).
- Beállítja a `mainCamera` kezdeti `backgroundColor` tulajdonságát a `colors` tömb első elemére (`colors[0]`).

3.3.14.2. Színátmenet Logikája (Update):

- Először ellenőrzi, hogy a `colors` tömb tartalmaz-e legalább két színt (`colors.Length < 2`). Ha nem, akkor nincs értelme színátmenetet végezni, így a metódus futása leáll (return).
- Növeli a `fadeTimer` változót az előző képkocka óta eltelt idővel (`Time.deltaTime`). Ez az időzítő követi nyomon, hogy mennyi idő telt el az aktuális színátmenet kezdete óta.
- Kiszámítja az interpolációs faktort a `fadeTimer` és a `fadeDuration` (az egy teljes színváltáshoz szükséges idő) hányadosaként. Az `t` értéke 0 és 1 között mozog, ahol 0 a kezdő színnek, 1 pedig a cél színnek felel meg.
- Meghatározza az aktuális szín indexét (`currentColorIndex`) és a következő szín indexét (`nextColorIndex`) a `colors` tömbben. A modulo operátor (%) biztosítja, hogy az indexek ciklikusan ismétlődjenek a tömb végére érve.
- Lekéri az aktuális (`currentColor`) és a következő (`nextColor`) színeket a tömbből az indexek alapján.
- A `Color.Lerp(currentColor, nextColor, t)` függvény segítségével lineárisan interpolál (kever) a két szín között az `t` faktor alapján. Az eredményül kapott köztes színt beállítja a `mainCamera.backgroundColor` tulajdonságának.
- Ellenőrzi, hogy az interpoláció befejeződött-e (`t >= 1f`). Ha igen, az azt jelenti, hogy a háttérszín elérte a `nextColor`-t. Ekkor:
 - Frissíti a `currentColorIndex`-et a `nextColorIndex`-re, hogy a következő ciklusban ez legyen a kiinduló szín.
 - Lenullázza a `fadeTimer`-t, hogy a következő színátmenet előlről kezdődhessen.

3.3.15. FloorManager

A FloorManager osztály felelős a játékszinteken belüli emeletek definiálásáért és kezeléséért. Elsődleges célja, hogy meghatározza egy adott emelet fizikai határait, és lehetővé tegye az ezen a területen tartózkodó ellenségek lekérdezését, valamint annak ellenőrzését, hogy az adott emeleten minden ellenséget meghalt-e.



3.3.15.1. Emelet Definíció és Vizualizáció:

Az osztály lehetővé teszi az egyes emeletek egyedi azonosítását és konfigurálását a Unity Editorban.

- floorName (string): Emelet neve
- floorIndex (int): Egy numerikus index az emelet azonosításához.
- floorBounds (Vector2): Meghatározza az emelet kiterjedését az X (szélesség) és Y (magasság) tengelyeken a FloorManager GameObject pozíciójához képest.
- gizmoColor (Color): A szín, amellyel az emelet határai megjelennek a Unity Editor „Scene” nézetében.

Az OnDrawGizmos metódus kizárólag a Unity Editorban fut le. Egy téglateetet (Gizmos.DrawWireCube) rajzol a FloorManager pozíciója köré, a floorBounds által meghatározott méretekkel és a gizmoColor színnel.

3.3.15.2. Ellenség Kezelés(GetEnemiesOnFloor, AreAllEnemiesDead):

Az osztály két fő metódust biztosít az emeleten található ellenségek kezelésére:

GetEnemiesOnFloor:

Ez a metódus visszaadja az összes olyan Enemy komponenst, amely fizikailag az adott FloorManager által definiált határokon belül található.

- Lekéri a jelenetben található összes aktív Enemy komponenst a FindObjectsByType<Enemy> függvényvel.
- Létrehoz egy Bounds objektumot a FloorManager pozíciója és a floorBounds alapján.
- Végigiterál az összes megtalált ellenségen.

- A Bounds.Contains metódussal ellenőrzi, hogy az adott ellenség transform.position-ja a definiált határokon belülre esik-e.
- A határokon belül lévő ellenségeket hozzáadja egy ideiglenes listához.
- Végül visszaadja a listát tömb formátumban.

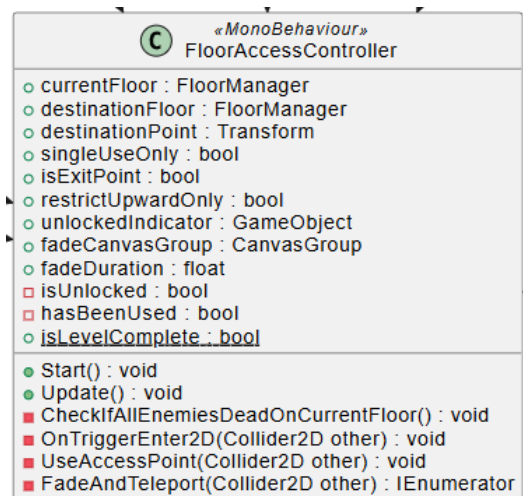
AreAllEnemiesDead:

Ez a logikai metódus ellenőrzi, hogy adott emeleten az összes ellenség halott-e.

- Meghívja a GetEnemiesOnFloor metódust az emelethez tartozó ellenségek listájának lekéréséhez.
- Végigiterál ezeken az ellenségeken.
- Minden ellenségnél ellenőrzi az isDead logikai változó értékét.
- Ha akár csak egyetlen élő ellenséget (!enemy.isDead) talál, false értékkel tér vissza.
- Ha a ciklus úgy ér véget, hogy nem talált élő ellenséget, true értékkel tér vissza, jelezve, hogy az emelet tiszta.

3.3.16. FloorAccessController

A FloorAccessController osztály felelős a játékon belüli szintek vagy területek közötti lépcsők működésének vezérléséért. Kezeli az átjárók feloldási feltételeit, a játékos átteleportálását a célpontra, valamint a kapcsolódó vizuális visszajelzéseket és a képernyő elsötétítését.



3.3.16.1. Inicializálás (Start)

- Ha a currentFloor nem található vagy nincs beállítva, hibaüzenetet naplóz.
- Beállítja az átjáró kezdeti állapotát (zárolt: isUnlocked = false, nem használt: hasBeenUsed = false).
- Megkeresi és eltárolja a feloldott állapotot jelző vizuális elem (unlockedIndicator) referenciáját és annak kiindulási pozícióját (indicatorStartPosition) az animációhoz.
- Az indikátort alapértelmezetten kikapcsolja.
- Meghívja az UpdateVisuals metódust a kezdeti vizuális állapot beállításához.

3.3.16.2. Feloldási Logika (Update, CheckIfAllEnemiesDeadOnCurrentFloor)

Update:

- Folyamatosan ellenőrzi a feloldási feltételeket, amennyiben az átjáró még zárolt (!isUnlocked) és nem volt még használva (!hasBeenUsed). A feloldás logikája függ az átjáró irányától és a restrictUpwardOnly beállítástól.
- Ha az átjáró felfelé vezet (destinationFloor.floorIndex > currentFloor.floorIndex), vagy ha a restrictUpwardOnly hamis, akkor a CheckIfAllEnemiesDeadOnCurrentFloor lefut.

CheckIfAllEnemiesDeadOnCurrentFloor:

- Ez a metódus lekérdezi az aktuális szint FloorManager komponensétől (currentFloor.AreAllEnemiesDead()), hogy az összes ellenség meghalt-e az adott szinten.
 - Ha igen, az átjáró feloldott állapotba kerül (isUnlocked = true) és a vizuális megjelenítés frissül (UpdateVisuals). Alternatívaként a feloldás köthető lenne kulcstárgyak megszerzéséhez vagy kapcsolók aktiválásához is
- Ha az átjáró lefelé vezet (destinationFloor.floorIndex < currentFloor.floorIndex) és a restrictUpwardOnly igaz, akkor a feloldás feltétele a FloorAccessController.isLevelComplete statikus változó igaz értéke.
- Ha az átjáró egy kijárat (isExitPoint), akkor szintén a isLevelComplete állapotot ellenőrzi a feloldáshoz az OnTriggerEnter2D metódusban.

3.3.16.3. Vizuális Visszajelzés (Update, UpdateVisuals)

UpdateVisuals:

- Felelős a feloldott állapotot jelző unlockedIndicator GameObject láthatóságának beállításáért. Az indikátor akkor látható, ha az átjáró feloldott (isUnlocked) és még nem volt használva (!hasBeenUsed), különben rejtett marad.
- Ha az unlockedIndicator aktív, az Update metódus egy egyszerű, szinusz hullámon alapuló lebegő animációt (indicatorBobSpeed, indicatorBobAmount) alkalmaz rá a Time.time felhasználásával.

3.3.16.4. Játékos Interakció (OnTriggerEnter2D):

- Ez a metódus akkor aktiválódik, amikor egy Collider2D belép az átjáró trigger zónájába. Először ellenőrzi, hogy nem zajlik-e már átmenet (isTransitioning), hogy elkerülje a többszöri aktiválást.
- Ezután ellenőrzi, hogy a belépő objektum rendelkezik-e a Player címkével.
- Megállapítja a mozgás irányát (felfelé vagy lefelé) a currentFloor és destinationFloor indexei alapján.
- Ellenőrzi a használat feltételeit:
- Ha az átjáró egy kijárat (isExitPoint) és a pálya teljesítve van (isLevelComplete), meghívja a UseAccessPoint metódust.
- Ha az átjáró feloldott (isUnlocked):
 - Felfelé irányuló mozgás esetén ellenőrzi, hogy az aktuális szinten minden ellenség halott-e (currentFloor.AreAllEnemiesDead()).
 - Lefelé irányuló mozgás esetén ellenőrzi, hogy a pálya teljesítve van-e (isLevelComplete).
- Ha a feltételek teljesülnek, meghívja a UseAccessPoint metódust a teleportálási folyamat elindításához.

3.3.16.5. Teleportálási Mechanizmus (UseAccessPoint, FadeAndTeleport):

UseAccessPoint:

Egyszerűen elindítja a FadeAndTeleport Coroutine-t

FadeAndTeleport (Coroutine):

- Először beállítja az isTransitioning állapotot igazra.
- Ha a CanvasGroup elérhető, a Coroutine elhalványítja a képernyőt feketére az alfa értékének növelésével a fadeDuration idő alatt.
- A teljes elsötétítés után azonnal átteleportálja a játékost (other.transform.position = destinationPoint.position;) és a fő kamerát (Camera.main.transform.position) a megadott destinationPoint pozíciójába.

- Frissíti az átjáró állapotát: ha `singleUseOnly` igaz, akkor beállítja a `hasBeenUsed` és `isUnlocked` változókat, egyébként csak az `isUnlocked` állapotot állítja vissza hamisra (hogy az esetleges visszatéréshez újra teljesülniük kelljen a feltételeknek). Meghívja az `UpdateVisuals`-t.
- Egy rövid várakozás után (`WaitForSeconds(0.1f)`) visszaállítja a képernyőt az alfa érték csökkentésével.
- Végül visszaállítja az `isTransitioning` állapotot hamisra, lehetővé téve az átjáró újbóli használatát (ha a feltételek engedik).

3.3.16.6. Konfigurációs Opciók:

Az osztály több publikus változóval rendelkezik, amelyek az Inspectorban konfigurálhatók:

- **currentFloor, destinationFloor, destinationPoint:** Az átjáró logikai és fizikai kapcsolódási pontjai.
- **singleUseOnly:** Meghatározza, hogy az átjáró csak egyszer használható-e.
- **isExitPoint:** Jelzi, ha ez a pálya végső kijárata.
- **restrictUpwardOnly:** Szabályozza, hogy a felfelé haladáshoz szükséges-e az ellenségek legyőzése, míg a lefelé haladás más feltételhez (pl. `isLevelComplete`) kötött.
- **unlockedIndicator, indicatorBobSpeed, indicatorBobAmount:** A vizuális visszajelzés és animációjának beállításai.
- **fadeCanvasGroup, fadeDuration:** A képernyő elsötétítés effektusának paraméterei.

3.3.17. DoorController és TriggerHandler

A `DoorController` osztály felelős a játékban található, fizikailag interaktív ajtók működéséért. Kezeli az ajtó nyitását és csukódását külső erőhatások (játékos vagy ellenség általi lökés) következtében, a forgás fizikai szimulációját, a hanghatásokat, valamint az interakciós zóna kezelését.

C «MonoBehaviour» DoorController	
o	maxOpenAngle : float
o	pushForce : float
o	doorMass : float
o	doorDamping : float
o	audioSource : AudioSource
o	doorOpenSound : AudioClip
□	initialRotation : Quaternion
□	currentAngularVelocity : float
□	currentRelativeAngle : float
□	doorTriggerZone : GameObject
●	Start() : void
●	Update() : void
●	HandleTriggerEvent(Collider2D other) : void

3.3.17.1. Inicializálás (Start):

- Rögzíti az ajtó kiindulási forgását (`initialRotation`), amely referenciapontként szolgál a relatív szög számításához.
- Inicializálja az aktuális relatív szöget (`currentRelativeAngle`) nullára, és beállítja a kezdeti zárt állapotot (`wasConsideredClosed`) annak alapján, hogy a szög a zártnak tekintett küszöbértéken (`closedAngleThreshold`) belül van-e.
- Lekéri az objektumhoz csatolt `AudioSource` komponenst a hangok lejátszásához.
- Megkeresi a gyermekobjektumok között a `DoorTriggerZone` nevű objektumot. Ez az objektum definiálja azt a területet, ahol a játékos vagy ellenség interakcióba léphet az ajtóval.
- Ha a `DoorTriggerZone` megtalálható, hozzáad egy `TriggerHandler` komponenst, és inicializálja azt a `DoorController` példány referenciájával. Ez a `TriggerHandler` felelős az ütközési események továbbításáért a `DoorController` felé.
- Amennyiben a `DoorTriggerZone` nem található, figyelmeztetést naplóz.

3.3.17.2. Fizikai Szimuláció és Forgás (Update):

- A szimuláció csak akkor aktív, ha az ajtó éppen mozog (a `currentAngularVelocity` abszolút értéke nagyobb egy kis küszöbértéknél), ez egy kisebb optimalizáció.
- Az aktuális relatív szöget (`currentRelativeAngle`) a szögsebesség és az eltelt idő (`Time.deltaTime`) alapján frissíti.
- A relatív szöget a `Mathf.Clamp` függvénnyel korlátozza a negatív és pozitív `maxOpenAngle` értékek közé, hogy az ajtó a falakon ne forduljon át.
- Az ajtó tényleges forgását (`transform.rotation`) az `initialRotation` és a kiszámított `currentRelativeAngle` alapján állítja be.
- A forgás lassítását (csillapítását) a `currentAngularVelocity` értékének csökkentésével valósítja meg minden képkockában a `doorDamping` érték alapján. Ez a szorzásos csökkentés (`currentAngularVelocity *= (1f - Time.deltaTime * doorDamping)`) egy exponenciális lassulást eredményez.

3.3.17.3. Ütközéskezelés és Erőkifejtés (HandleTriggerEvent, ApplyImmediatePush):

HandleTriggerEvent:

A metódust a gyermekobjektumon lévő TriggerHandler szkript hívja meg, amikor OnTriggerEnter2D vagy OnTriggerStay2D esemény történik.

- Ellenőrzi, hogy az ütköző objektum rendelkezik-e Player vagy Enemy címkével. Csak ezek az objektumok képesek kinyitni az ajtót.
- Ha a címke megfelelő, meghívja az ApplyImmediatePush metódust.

ApplyImmediatePush:

Kiszámítja az ajtó középpontjából az ütköző objektum felé mutató irányvektort (toPusher).

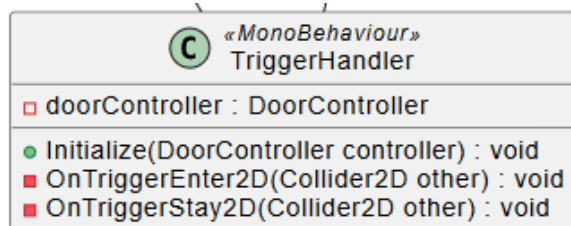
- A Vector2.Dot (skaláris szorzat) segítségével meghatározza, hogy az ütköző objektum az ajtó melyik oldalán helyezkedik el az ajtó (transform.right) vektorához képest.
- A skaláris szorzat előjele alapján meghatározza a lökés irányát (pushDirection: 1 vagy -1).
- Beállítja az ajtó szögsebességét (currentAngularVelocity) a pushForce (lökőerő), a doorMass (ajtó tömege) és a lökés iránya alapján.

3.3.17.4. Hangkezelés (Update):

- Folyamatosan ellenőrzi, hogy az ajtó aktuálisan zártnak tekinthető-e (isCurrentlyClosed) a currentRelativeAngle és a closedAngleThreshold összehasonlításával.
- Összeveti az aktuális zárt állapotot az előző képkockában tárolt állapottal (wasConsideredClosed).
- Hangot csak akkor játszik le, ha az állapot megváltozott (azaz az ajtó éppen most nyílt ki vagy csukódott be).
- További feltétel a hanglejátszáshoz, hogy elegendő idő (minTimeBetweenSounds) teljen el az utolsó hanglejátszás óta (lastSoundTime).
- A hangmagasságot véletlenszerűsíti.
- Az aktuális zárt állapotot eltárolja a wasConsideredClosed változóban a következő képkocka számára.

3.3.17.5. Trigger Kezelés (TriggerHandler és integráció):

Az ajtóval való interakció érzékelése egy különálló TriggerHandler komponens segítségével történik, amely az ajtó egy gyermekobjektumán (DoorTriggerZone)

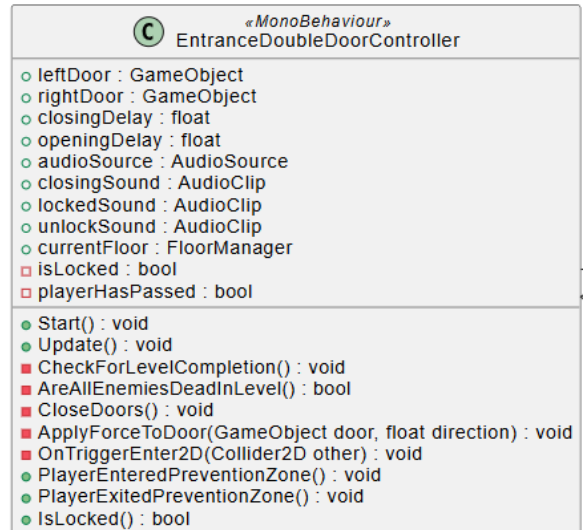


helyezkedik el. Ez azért létezik, hogy az ajtó ne lassítsa le a játékost nyitáskor, és hogy semmiképp sem tudjon elakadni szélsőséges esetekben. A játékos valójában nem tud ütközni az ajtóval.

- A TriggerHandler Initialize metódusát a DoorController hívja meg a Start fázisban, átadva saját magát referenciaként.
- A TriggerHandler OnTriggerEnter2D és OnTriggerStay2D metódusai egyszerűen továbbítják az ütközési eseményt (és az ütköző Collider2D objektumot) a tárolt DoorController referencia HandleTriggerEvent metódusának. Az OnTriggerStay2D használata biztosítja, hogy az ajtó folyamatosan kapjon lökést, amíg a játékos vagy ellenség a trigger zónán belül tartózkodik, nem csak a belépés pillanatában.

3.3.18. EntranceDoubleDoorController

Egy magasabb szintű vezérlő szkript, amely egy pár ajtót (épület bejáratát) kezel. Koordinálja két különálló DoorController példány (egy bal és egy jobb oldali ajtószárny) működését, beleértve azok automatikus bezárását a játékos áthaladása után, lezárását, amíg a szint összes ellensége meg nem hal, és az újbóli kinyitását a szint teljesítésekor. Tartalmaz továbbá egy mechanizmust a „soft lock” (a játékos véletlen bezárásának) megakadályozására.



3.3.18.1. Inicializálás (Start):

- Lekéri a saját AudioSource komponensét
- Ellenőrzi a currentFloor referenciát, és figyelmeztet, ha hiányzik.
- Megkeresi és eltárolja a bal és jobb ajtó GameObject-jeit, valamint a hozzájuk tartozó DoorController és Collider2D komponenseket. Hibaüzenetet naplóz, ha valamelyik ajtó referencia hiányzik.
- Megkeresi a gyermekobjektumok között az EntranceSoftLockPreventor rétegen lévő GameObject-et (softLockPreventor). Ez az objektum egy trigger zónát definiál közvetlenül az ajtók körül.
- Ha a softLockPreventor objektum megtalálható, ellenőrzi, hogy van-e rajta SoftLockPreventorTrigger komponens.

3.3.18.2. Lezárás és Feloldás Logikája (Update, CheckForLevelCompletion, AreAllEnemiesDeadInLevel):

Update:

- Folyamatosan ellenőrzi, hogy az ajtók le vannak-e zárva (isLocked).
- Ha igen, meghívja a CheckForLevelCompletion metódust.

CheckForLevelCompletion:

- Meghívja az AreAllEnemiesDeadInLevel függvényt annak ellenőrzésére, hogy a pálya összes ellensége halott-e.

AreAllEnemiesDeadInLevel:

- Megkeresi az összes FloorManager példányt a FindObjectsByType segítségével.
- Végigiterál a talált FloorManager-eken, és mindegyiknél meghívja az AreAllEnemiesDead metódust.
- Ha bármelyik FloorManager élő ellenséget jelez, a függvény false értékkel tér vissza.
- Ha az AreAllEnemiesDeadInLevel true értékkel tér vissza (minden ellenség halott):
 - Az isLocked állapot false-ra vált.
 - Elindítja a DelayedDoorsOpen Coroutinet az ajtók késleltetett kinyitásához.
 - Meghívja az UpdateDoorColliders(false) metódust, hogy az ajtók fizikai ütközőit újra trigger típusúvá tegye

3.3.18.3. Ajtónyitási Mechanizmus (DelayedDoorsOpen, ApplyForceToDoor):

DelayedDoorsOpen (Coroutine):

- Először vár az openingDelay másodpercig.
- Lejátssza a feloldás hangját (unlockSound) a PlaySound segédfüggvénnyel.
- Meghívja az ApplyForceToDoor metódust mindkét ajtószárnyra (leftDoor, rightDoor). Az erő irányát az openDirectionLeft és openDirectionRight logikai változók határozzák meg, lehetővé téve az ajtók befelé vagy kifelé nyílását.

ApplyForceToDoor:

- Létrehoz egy ideiglenes, láthatatlan GameObject-et ("TempPusher") Player címkével.
- Elhelyezi ezt az objektumot az ajtó megfelelő oldalán, hogy a kívánt irányba "lökje" azt.
- Hozzáad egy trigger típusú BoxCollider2D-t az ideiglenes objektumhoz.
- Lekéri az ajtó DoorController komponensét.
- Ideiglenesen felülírja az ajtó DoorController-ének pushForce értékét az openingForce értékével
- Meghívja az ajtó DoorController-ének HandleTriggerEvent metódusát az ideiglenes lökő objektum colliderével
- Egy rövid késleltetés után megsemmisíti az ideiglenes lökő objektumot.

A lökő objektumra azért volt szükség, hogy pontosan be lehessen állítani a nyitási erőket, ami nem azonos a játékos és az ellenség által használtakkal.

3.3.18.4. Ajtózárási Mechanizmus (CloseDoors, ApplySmoothClosingForce, DelayedDoorsClose, OnTriggerEnter2D):

Kezelik az ajtók automatikus bezáródását és lezárását, miután a játékos áthaladt.

OnTriggerEnter2D:

- Érzékeli, ha egy objektum belép a fő trigger zónába.
- Ha az objektum a Player, és a játékos még nem haladt át korábban (playerHasPassed == false), és az ajtó nincs lezárva (isLocked == false):
 - A playerHasPassed jelzőt true-ra állítja.
 - Ellenőrzi, hogy a játékos a soft lock megelőző zónában van-e (playerInPreventionZone). Ha nincs (vagy nincs ilyen zóna definiálva), elindítja a DelayedDoorsClose Coroutinet az ajtók késleltetett bezárásához.
 - Ha az objektum a Player, de az ajtó már le van zárva (isLocked == true), lejátsza a zárt ajtó hangját (lockedSound).

DelayedDoorsClose (Coroutine):

- Vár a closingDelay másodpercig.
- Újra ellenőrzi, hogy a játékos időközben belépett-e a megelőző zónába. Ha igen, a Coroutine megszakad, és az ajtók nem záródnak be.
- Lejátsza a bezáródás hangját (closingSound).
- Meghívja a CloseDoors metódust.

CloseDoors:

- Az isLocked állapotot true-ra állítja.
- Meghívja az UpdateDoorColliders(true) metódust, hogy az ajtók fizikai ütközőit ne trigger típusúvá tegye, ezzel fizikailag lezárva az utat.
- Naplózza az ajtók bezáródását és lezárását.
- A CloseDoors metódus egyszerűen meghívja az ApplySmoothClosingForce metódust mindkét ajtószárnyra.

ApplySmoothClosingForce:

- Lekéri az ajtó DoorController-ét.
- Meghatározza a záráshoz szükséges forgási irányt az ajtó aktuális relatív szögének (currentRelativeAngle) előjele alapján.
- Ha az ajtó már majdnem teljesen zárva van (a szög abszolút értéke kicsi), nullázza a szögét és a szögsebességét, és visszaállítja az eredeti forgását.
- Ellenkező esetben kiszámít egy záróerőt, amely arányos az aktuális nyitási szöggel ($\text{closingForce} * \text{angleRatio} * \text{forceSensitivity}$).
- Közvetlenül beállítja az ajtó DoorController-ének currentAngularVelocity értékét a kiszámított erő, az ajtó tömege (doorMass) és az irány alapján.

3.3.18.5. Soft Lock Megelőzés (PlayerEnteredPreventionZone, PlayerExitedPreventionZone, SoftLockPreventorTrigger segédosztály):

Van egy külön trigger zóna (softLockPreventor), amely az EntranceSoftLockPreventor rétegen van.

«MonoBehaviour» SoftLockPreventorTrigger	
□	doorController : EntranceDoubleDoorController
●	Initialize(EntranceDoubleDoorController controller) : void
■	OnTriggerEnter2D(Collider2D other) : void
■	OnTriggerExit2D(Collider2D other) : void

Ehhez létezik egy különálló komponens, amely a softLockPreventor GameObject-en helyezkedik el. Egyetlen feladata, hogy érzékelje, amikor a játékos belép vagy kilép a megelőző zónából (OnTriggerEnter2D, OnTriggerExit2D), és értesítse erről a fő EntranceDoubleDoorController-t a PlayerEnteredPreventionZone és PlayerExitedPreventionZone metódusok meghívásával.

PlayerEnteredPreventionZone:

- Beállítja a playerInPreventionZone logikai változót true-ra.
- Ha éppen fut az ajtókat bezáró closingCoroutine, leállítja azt, megakadályozva az ajtók bezáródását, amíg a játékos a zónában van.

PlayerExitedPreventionZone:

- Beállítja a playerInPreventionZone logikai változót false-ra.
- Ha a játékos már áthaladt a fő triggeren (playerHasPassed == true), az ajtók nincsenek lezárva, és a komponens aktív, akkor újra elindítja a DelayedDoorsClose Coroutineet.

3.3.18.6. Collider Kezelés (UpdateDoorColliders):

- Amikor az ajtókat le kell zárni (`lockDoors == true`), a metódus `false`-ra állítja az `isTrigger` tulajdonságot mindkét ajtó colliderén. Ezáltal az ajtók szilárd fizikai akadályokká válnak.
- Amikor az ajtókat fel kell oldani (`lockDoors == false`), a metódus `true`-ra állítja az `isTrigger` tulajdonságot.

3.3.18.7. Hangkezelés (PlaySound):

- Ellenőrzi, hogy az `audioSource` és a megadott clip érvényes-e, és hogy elegendő idő telt-e el az utolsó hang lejátszása óta
- Lejátszás előtt enyhén véletlenszerűsíti a hangmagasságot
- Lejátssza a hangklipet az `PlayOneShot` metódussal.

3.3.18.8. Állapot Lekérdezés (IsLocked):

Egy egyszerű publikus getter metódus, amely lehetővé teszi más szkriptek számára, hogy lekérdezzék az ajtó aktuális zárolt állapotát (`isLocked`) anélkül, hogy közvetlenül hozzáférnének a változóhoz.

3.3.19. LockedDoorController

A `LockedDoorController` osztály egy speciális ajtóvezérlő, amely egyetlen, kezdetben zárt ajtó működését irányítja. Az ajtó automatikusan kinyílik, amint egy hozzárendelt területen (FloorManager által képviselt szinten) minden ellenség elpusztul. Az osztály kezeli az ajtó fizikai viselkedését (forgás, lökés), a zárolt állapotot, a feloldási folyamatot, a hanghatásokat és az

C «MonoBehaviour» LockedDoorController	
○	<code>maxOpenAngle : float</code>
○	<code>pushForce : float</code>
○	<code>unlockForce : float</code>
○	<code>unlockDelay : float</code>
○	<code>audioSource : AudioSource</code>
○	<code>unlockSound : AudioClip</code>
○	<code>lockedSound : AudioClip</code>
○	<code>currentFloor : FloorManager</code>
□	<code>isDoorUnlocked : bool</code>
●	<code>Start() : void</code>
●	<code>Update() : void</code>
■	<code>CheckIfAllEnemiesDeadOnCurrentFloor() : void</code>
■	<code>OnTriggerEnter2D(Collider2D other) : void</code>
■	<code>OnTriggerStay2D(Collider2D other) : void</code>

interakciót a játékoskal és az ellenségekkel. Hasonlóan a `DoorController`-hez, ez az osztály is egy egyszerű, manuális fizikai modellt használ a forgás szimulálására.

3.3.19.1. Inicializálás (Start):

- Rögzíti az ajtó kiindulási forgását (`initialRotation`).
- Az aktuális relatív szöget (`currentRelativeAngle`) nullára állítja.
- Lekéri az objektumhoz csatolt `AudioSource` komponenst.
- Az ajtó kezdeti állapotát zároltra állítja (`isDoorUnlocked = false`).

3.3.19.2. Állapotfrissítés és Fizikai Szimuláció (Update):

Update:

- Zárolás Ellenőrzése: Ha az ajtó még zárolt (`!isDoorUnlocked`) és hozzá van rendelve egy érvényes `FloorManager` (`currentFloor != null`), meghívja a `CheckIfAllEnemiesDeadOnCurrentFloor` metódust.
- Fizikai Szimuláció: Ha az ajtó szögsebessége (`currentAngularVelocity`) nem elhanyagolható, frissíti az ajtó fizikai állapotát:
 - Kiszámítja az új relatív szöget (`currentRelativeAngle`) a sebesség és az eltelt idő alapján.
 - Korlátozza a szöget a `maxOpenAngle` értékre a `Mathf.Clamp` segítségével.
 - Beállítja az ajtó tényleges forgását (`transform.rotation`) az `initialRotation` és a `currentRelativeAngle` alapján.
 - Alkalmazza a csillapítást (`damping`) a szögsebesség csökkentésével (`currentAngularVelocity *= (1f - Time.deltaTime * doorDamping)`).

3.3.19.3. Feloldási Folyamat (`CheckIfAllEnemiesDeadOnCurrentFloor`, `DelayedDoorOpen`, `ApplyUnlockForce`):

`CheckIfAllEnemiesDeadOnCurrentFloor`:

Ha `currentfloor.AreAllEnemiesDead` igaz, azaz a szinten nincs több élő ellenség:

- Az `isDoorUnlocked` állapotot `true`-ra állítja
- Elindítja a `DelayedDoorOpen` Coroutinet, hogy az ajtó ne azonnal, hanem egy kis késleltetés után nyíljon ki.

DelayedDoorOpen (Coroutine):

- Vár az unlockDelay másodpercig.
- Lejátssza a feloldás hangját (unlockSound) a PlaySound metódussal.
- Meghívja az ApplyUnlockForce metódust az ajtó tényleges kinyitásához.

ApplyUnlockForce:

- Meghatározza a nyitás irányát (pushDirection) az openDirection logikai változó alapján (1f vagy -1f).
- Beállítja az ajtó szögsebességét (currentAngularVelocity) az unlockForce, a doorMass és a pushDirection alapján. Ez egy kezdeti lökést ad az ajtónak, hogy automatikusan kinyíljon a feloldás után.

3.3.19.4. Interakció és Lökés (OnTriggerEnter2D, OnTriggerStay2D, ApplyImmediatePush):

OnTriggerEnter2D:

- Ha az objektum Player címkéjű: Ellenőrzi az isDoorUnlocked állapotot. Ha true (az ajtó nyitva van), meghívja az ApplyImmediatePush metódust, hogy a játékos meglökhesse az ajtót. Ha false (az ajtó zárva van), lejátssza a zárt ajtó hangját (lockedSound).
- Ha az objektum Enemy címkéjű és az ajtó már nyitva van (isDoorUnlocked == true): Az ellenségek is meglökhetik a nyitott ajtót, ezért meghívja az ApplyImmediatePush metódust.

OnTriggerStay2D:

Akkor hívódik meg folyamatosan, amíg egy objektum az ajtó trigger zónájában tartózkodik.

- Ha az ajtó nyitva van (isDoorUnlocked == true) és az objektum Player vagy Enemy címkéjű, folyamatosan meghívja az ApplyImmediatePush metódust, lehetővé téve az ajtó folyamatos lökését.

ApplyImmediatePush:

Ez a metódus alkalmazza a tényleges lökést az ajtóra.

- Először ellenőrzi, hogy az ajtó nyitva van-e (isDoorUnlocked). Ha nem, nem tesz semmit.
- Kiszámítja az ajtó középpontjából a lökő objektum felé mutató irányvektort (toPusher).
- A Vector2.Dot (skaláris szorzat) segítségével meghatározza, hogy a lökő objektum az ajtó melyik oldalán van az ajtó lokális jobbra vektorához képest.
- A skaláris szorzat előjele alapján meghatározza a lökés irányát (pushDirection: 1 vagy -1).
- Beállítja az ajtó szögsebességét (currentAngularVelocity) a pushForce, a doorMass és a lökés iránya alapján, szimulálva a lökést.

3.3.19.5. Hangkezelés (PlaySound):

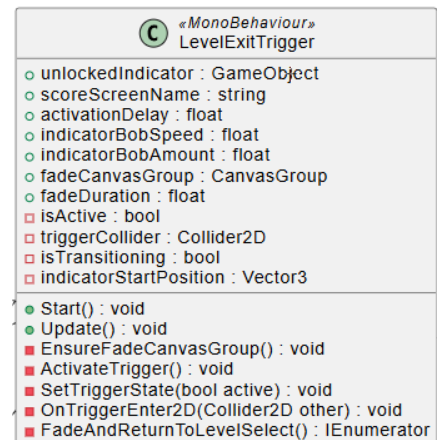
Egy általános segédfüggvény hangklipet lejátszására, ami megegyezik az EntranceDoubleDoorController-ben használt funkcióval.

3.3.20. LevelExitTrigger

A kijáratot aktiválja, amint a pálya teljesítése megtörtént (a FloorAccessController statikus változója alapján), a teleporthoz hasonló módon egy vizuális indikátort jelenít meg és animál, valamint átirányítja a játékost a pontszám képernyőre egy képernyő elsötétítési effekt kíséretében.

3.3.20.1. Inicializálás (Start):

- Lekéri a GameObject-hez csatolt Collider2D komponenst
- Lekéri az unlockedIndicator GameObject referenciáját.
- Ha az indikátor létezik, eltárolja annak kezdeti lokális pozícióját (indicatorStartPosition) a későbbi animációhoz, majd alapértelmezetten kikapcsolja (SetActive(false)).
- Meghívja az EnsureFadeCanvasGroup metódust, hogy biztosítsa a képernyő elsötétítéséhez használt CanvasGroup referenciájának meglétét.
- Inaktív állapotba állítja a triggert a SetTriggerState(false) metódussal.



3.3.20.2. Aktiválás és Állapotfrissítés (Update, ActivateTrigger, SetTriggerState):

Update:

- Folyamatosan ellenőrzi a FloorAccessController.isLevelComplete állapotát. Ha a pálya teljesítetté válik, és a trigger még nem aktív (!isActive), akkor az Invoke segítségével időzíti az ActivateTrigger meghívását az activationDelay késleltetéssel.
- Ha az unlockedIndicator létezik és aktív, az Update ciklusban animálja azt.

ActivateTrigger:

- Meghívja a SetTriggerState(true) metódust.

SetTriggerState:

- Beállítja az isActive logikai változót, amely jelzi a trigger aktív állapotát.
- Ennek megfelelően engedélyezi vagy letiltja a triggerCollider komponenst, valamint megjeleníti vagy elrejtí az unlockedIndicator GameObject-et.

3.3.20.3. Interakció és Átmenet (OnTriggerEnter2D, FadeAndReturnToLevelSelect):

OnTriggerEnter2D:

Akkor aktiválódik, amikor egy Collider2D belép a kijárat trigger zónájába.

- Ellenőrzi, hogy a trigger aktív-e (isActive), nem zajlik-e már átmenet (!isTransitioning), és hogy a belépő objektum a Player címkével rendelkezik-e.
- Ha minden feltétel teljesül, elindítja a FadeAndReturnToLevelSelect Coroutinet a jelenetváltási folyamat megkezdéséhez.

FadeAndReturnToLevelSelect (Coroutine):

- Beállítja az isTransitioning jelzőt true-ra, hogy megakadályozza a Coroutine többszöri elindítását.
- Biztosítja, hogy az idő skálája (Time.timeScale) 1 legyen, feloldva az esetleges játék közbeni szüneteltetést.
- Eltárolja az aktuális jelenet nevét a PlayerPrefs "LastPlayedLevel" kulcsa alatt.
- Meghívja az EnsureFadeCanvasGroup metódust, hogy biztosan legyen referencia az elsötétítő vászonhoz.

- Ha a fadeCanvasGroup létezik:
 - Aktiválja a vászon GameObject-jét.
 - Egy while ciklus segítségével fokozatosan növeli a fadeCanvasGroup alpha értékét 0-ról 1-re a fadeDuration idő alatt, létrehozva a képernyő elsötétítésének (fade-to-black) effektjét. A ciklus a yield return null; utasítással vár a következő képkockáig minden iterációban.
 - Az elsötétítés végén az alpha értéket 1-re állítja
- Vár egy rövid, fix ideig (0.1 másodperc).
- Lekéri az UIManager singleton példányát, és elrejtí a játék közbeni UI elemeket (HUD, Pause Menu, stb.).
- Lekéri a ScoreManager singleton példányát, és kiolvassa a pálya teljesítésével kapcsolatos pontszám adatokat (ölések pontszáma, kombó bónusz, idő bónusz, pontosság, végső pontszám, osztályzat, teljesítési idő).
- Ezeket az adatokat átadja a ScoreScreenManager statikus változóinak.
- Betölti a kiértékelő képernyőt.

3.3.20.4. EnsureFadeCanvasGroup:

Ez a segédfüggvény biztosítja a megbízható hozzáférést az elsötétítéshez használt CanvasGroup komponenshez.

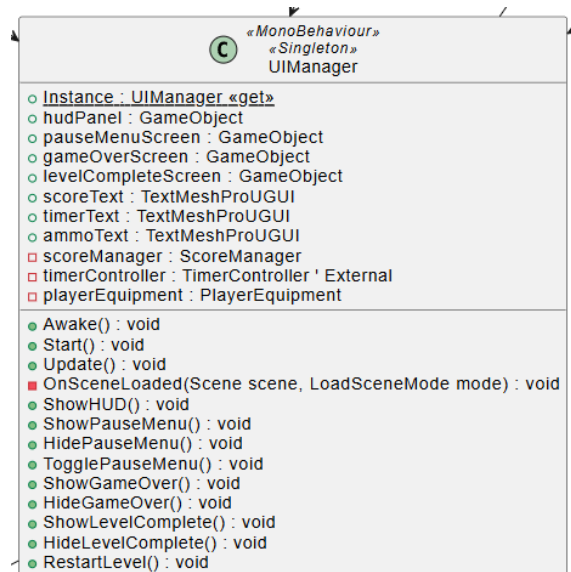
- Ellenőrzi, hogy a fadeCanvasGroup referencia már be van-e állítva.
- Ha nincs, megkeresi a "FadeCanvas" címkével ellátott GameObject-et a jelenetben.
- Ha a GameObject megtalálható, lekéri róla a CanvasGroup komponenst.
- Ha a CanvasGroup sikeresen lekérésre került, alapértelmezettként az alpha értékét 0-ra állítja
- Figyelmeztetést naplóz, ha a CanvasGroup nem található.

3.3.21. UIManager

Az UIManager osztály a játék felhasználói felületének vezérlője. Singleton mintát alkalmaz. Felelős a különböző UI panelek megjelenítéséért és elrejtéséért. Az objektum perzisztens a jelenetek között (DontDestroyOnLoad).

3.3.21.1. Singleton és Perzisztencia (Awake):

Az Awake metódus implementálja a Singleton mintát.



- Ellenőrzi, hogy létezik-e már egy UIManager példány (Instance). Ha igen, és az nem ez az objektum, akkor megsemmisíti ezt az új példányt, megakadályozva a duplikációt.
- Ha még nem létezik példány, akkor beállítja az Instance statikus változót erre az objektumra.
- Meghívja a DontDestroyOnLoad(gameObject) metódust, amely biztosítja, hogy az UIManager objektum ne semmisüljön meg jelenetváltáskor.

3.3.21.2. Jelenet Betöltés Kezelése (OnEnable, OnDisable, OnSceneLoaded):

OnEnable:

- Feliratkozik a SceneManager.sceneLoaded eseményre. Ez azt jelenti, hogy az OnSceneLoaded metódus automatikusan meghívódik minden alkalommal, amikor egy új jelenet betöltése befejeződik.

OnDisable:

- Leiratkozik a SceneManager.sceneLoaded eseményről.

OnSceneLoaded:

- Visszaállítja az idő skáláját 1-re (Time.timeScale = 1f)
- Meghívja a ResetUIState metódust, hogy alaphelyzetbe állítsa a UI elemek láthatóságát (HUD be, többi panel ki).
- Meghívja a SetupUIElements metódust, hogy újra megkeresse és összekapcsolja a szükséges UI komponenseket és adatforrásokat az új jelenetben.

3.3.21.3. Inicializálás és Referenciák (Start, SetupUIElements):

Start:

- Megkeresi a ScoreManager és TimerController példányokat a FindFirstObjectByType segítségével.
- Meghívja a SetupUIElements és ResetUIState metódusokat a kezdeti UI beállításához.

SetupUIElements:

- Ha a TimerController és a timerText létezik, átadja a timerText referenciáját a TimerController-nek.
- Hasonlóképpen, ha a ScoreManager és a scoreText létezik, átadja a scoreText referenciáját a ScoreManager-nek.
- Megpróbálja megtalálni az ammoText referenciáját az AmmoDisplay komponens segítségével (FindFirstObjectByType<AmmoDisplay>). Ha sikeres, lekéri annak ammoText mezőjét.

3.3.21.4. Input Kezelés és Szüneteltetés (Update):

Az Update metódus kezeli az Escape billentyű lenyomását a szünet menü vezérlésére és a főmenübe való visszalépésre.

- Először ellenőrzi az aktuális jelenet nevét. Ha a jelenet "MainMenu" vagy "LevelSelect", akkor a metódus azonnal visszatér, nem dolgozza fel az Escape billentyűt ezekben a menükben.

- Escape lenyomása (`Input.GetKeyDown(KeyCode.Escape)`):
 - Ha a játék éppen szünetel (`Time.timeScale == 0f`), elkezd mérni, mennyi ideig van lenyomva az Escape (`isHoldingEscape = true`, `escapeHeldTime = Time.unscaledTime`). A `Time.unscaledTime` használata kell, mert a normál `Time.time` nem telik, amíg a játék szünetel.
 - Ha a játék nem szünetel, és nincs aktív Game Over vagy Level Complete képernyő, akkor meghívja a `TogglePauseMenu` metódust a szünet menü megjelenítésére/elrejtésére.
- Escape felengedése (`Input.GetKeyUp(KeyCode.Escape)`):
 - Ha az Escape billentyűt lenyomva tartották (`isHoldingEscape == true`), de nem elég ideig a főmenübe való visszalépéshez (a tartás ideje kisebb vagy egyenlő, mint `escapeHoldDuration`), és a játék szünetelt, akkor a felengedéskor elrejt a szünet menüt (`HidePauseMenu`), folytatja a játékot.
 - Visszaállítja a lenyomva tartást figyelő változókat (`isHoldingEscape = false`, `escapeHeldTime = 0f`).
- Escape nyomva tartása (`isHoldingEscape && Time.timeScale == 0f && Time.unscaledTime - escapeHeldTime > escapeHoldDuration`):
 - Ha a játék szünetel, és az Escape billentyűt a megadott `escapeHoldDuration` időtartamnál tovább tartják lenyomva:
 - Visszaállítja a figyelő változókat.
 - Elrejt a HUD panelt és törli az időmérő szövegét.
 - Visszaállítja az idő skáláját 1-re.
 - Betölti a "MainMenu" jelenetet.

3.3.21.5. UI Állapotok Vezérlése (ResetUIState, Show/Hide metódusok):

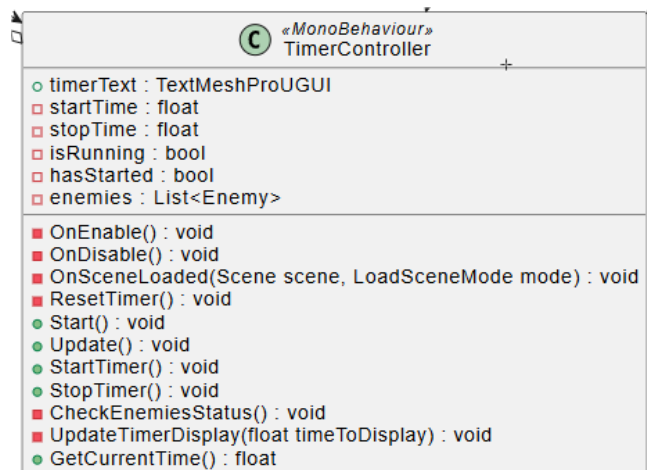
- **ResetUIState:** Alaphelyzetbe állítja a UI-t: megjeleníti a HUD-ot, elrejt a szünet menüt, a játék vége és a pálya teljesítve képernyőket.
- **ShowHUD, HideHUD:** A hudPanel GameObject aktiválását/deaktiválását végzik.
- **ShowPauseMenu:** Megjeleníti a pauseMenuScreen-t, elrejt a többi overlay képernyőt (Game Over, Level Complete), és a Time.timeScale értékét 0-ra állítja
- **HidePauseMenu:** Elrejt a pauseMenuScreen-t és a Time.timeScale értékét 1-re állítja, folytatva a játékot.
- **TogglePauseMenu:** Vált a ShowPauseMenu és HidePauseMenu között.
- **ShowGameOver:** Megjeleníti a gameOverScreen-t, elrejt a szünet és pálya teljesítve képernyőket. Meghívja az AmmoDisplay ResetDisplay.
- **HideGameOver:** Elrejt a gameOverScreen-t.
- **ShowLevelComplete:**
 - Megjeleníti a levelCompleteScreen-t
 - Elrejt a szünet és játék vége képernyőket.
 - Szintén meghívja az AmmoDisplay ResetDisplay metódusát.
- **HideLevelComplete:** Elrejt a levelCompleteScreen-t.

3.3.21.6. Menü Műveletek (ResumeGame, RestartLevel):

- ResumeGame: Egyszerűen meghívja a HidePauseMenu metódust a játék folytatásához.
- RestartLevel:
 - Visszaállítja az idő skáláját 1-re.
 - Visszaállítja a FloorAccessController.isLevelComplete statikus változót false-ra, jelezve, hogy az újraindított pálya még nincs teljesítve.
 - Újratölti az aktuálisan aktív jelenetet a SceneManager.LoadScene segítségével.

3.3.22. TimerController

A TimerController osztály felelős a játék pályáin eltelt idő méréséért és kijelzéséért. Az időmérés jellemzően akkor indul el, amikor a játékos először elmozdul, és leáll, amint a pályán található összes ellenség elpusztul. Az osztály kezeli az időmérő állapotát (fut, leállítva, el sem indult), frissíti a felhasználói felületen megjelenő időt (egy TextMeshProUGUI elemen keresztül, amelyet a UIManager biztosít), és nyilvántartja a pályán lévő ellenségeket a leállítási feltétel ellenőrzéséhez.



3.3.22.1. Inicializálás és Jelenetkezelés (Start, OnEnable, OnDisable, OnSceneLoaded, ResetTimer):

Start:

- Összegyűjti a jelenetben található összes Enemy komponenst a FindObjectsByType segítségével és hozzáadja őket az enemies listához.
- Beállítja az időmérő kijelzőjének alapértelmezett szövegét ("00.000").

OnEnable/OnDisable:

- Feliratkozik, illetve leiratkozik a SceneManager.sceneLoaded eseményre, hogy az OnSceneLoaded metódus meghívódjon minden jelenetbetöltéskor.

OnSceneLoaded:

- Ellenőrzi a betöltött jelenet nevét. Ha a név tartalmazza a "map_" szöveget, meghívja a ResetTimer metódust az időzítő teljes alaphelyzetbe állításához.
- Más típusú jelenetek (pl. menük) esetén csak az alap időzítő változókat nullázza és a kijelzőt állítja alaphelyzetbe, de nem keres ellenségeket.

ResetTimer:

Teljesen alaphelyzetbe állítja az időzítőt.

- Nullázza az időmérési változókat (startTime, stopTime), leállítja az időzítőt (isRunning = false) és visszaállítja az elindulást jelző flag-et (hasStarted = false).
- Frissíti a timerText kijelzőt "00.000" értékre.
- Kitörli az enemies listát, majd újra feltölti azt a jelenetben található összes Enemy objektummal a FindObjectsByType segítségével.

3.3.22.2. Időmérés Logikája (StartTimer, StopTimer, Update):

StartTimer:

- A PlayerController, amikor a játékos először elmozdul hívja meg az időmérés elindításához.
- Csak akkor fut le, ha az időzítő még nem indult el az adott pályán (hasStarted == false).
- Rögzíti az indítási időt (startTime = Time.time)
- Beállítja az isRunning és hasStarted logikai változókat true-ra.

StopTimer:

- CheckEnemiesStatus hívja meg, amikor minden ellenség meghalt.
- Csak akkor fut le, ha az időzítő éppen futott (isRunning == true).
- Rögzíti a leállítási időt (stopTime = Time.time)
- Beállítja az isRunning változót false-ra, és frissíti a kijelzőt a végső, mért idővel

Update:

Ha az időzítő éppen fut (isRunning == true):

- Kiszámítja az eltelt időt (elapsedTime = Time.time - startTime).
- Frissíti a kijelzőt az UpdateTimerDisplay metódussal.
- Ellenőrzi az ellenségek állapotát a CheckEnemiesStatus metódussal, hogy megállapítsa, le kell-e állítani az időzítőt.

3.3.22.3. Ellenségek Követése és Időzítő Leállítása (CheckEnemiesStatus):

Ez a rész felelős az időzítő automatikus leállításáért a pálya teljesítésekor.

Az osztály egy enemies nevű List<Enemy> típusú listában tárolja a pályán található ellenségek referenciáit. Ezt a listát a Start és a ResetTimer metódusok töltik fel.

CheckEnemiesStatus:

- Végigiterál az enemies listán.
- Ha talál legalább egy olyan ellenséget, amely még létezik (nem null) és nincs halott állapotban (!enemy.isDead), akkor a ciklus megszakad, és az időzítő tovább fut.
- Ha a ciklus végigmegy anélkül, hogy élő ellenséget találna, és a lista nem üres (enemies.Count > 0), az azt jelenti, hogy minden ellenség meghalt, ezért meghívja a StopTimer metódust az időmérés leállításához.

3.3.22.4. Kijelző Frissítése (UpdateTimerDisplay):

- Bemenetként egy float értéket kap (az eltelt időt).
- Ellenőrzi, hogy a timerText referencia (amelyet általában az UIManager állít be) érvényes-e. Ha nem, a függvény visszatér.
- Kiszámítja az egész másodperceket (Mathf.FloorToInt) és az ezredmásodperceket (a maradék törtrész szorozva 1000-rel).
- A string.Format segítségével formázza az időt "SS.mmm" (két számjegyű másodperc, három számjegyű ezredmásodperc) formátumra.
- Beállítja a timerText komponens text tulajdonságát a formázott stringre.

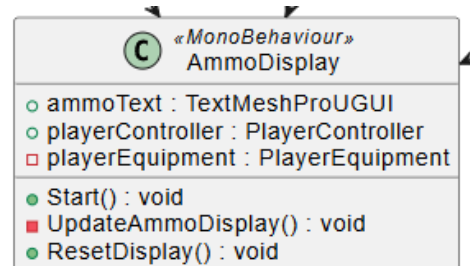
3.3.22.5. Idő Lekérdezése (GetCurrentTime):

Ez a publikus metódus lehetővé teszi más szkriptek számára, hogy lekérdezzék az aktuálisan mért időt.

- Ha az időzítő éppen fut (`isRunning == true`), visszaadja a jelenleg eltelt időt (`Time.time - startTime`).
- Ha az időzítő már elindult (`hasStarted == true`), de már leállt (`isRunning == false`), visszaadja a teljes mért időt (`stopTime - startTime`).
- Ha az időzítő még el sem indult, 0f értéket ad vissza.

3.3.23. AmmoDisplay

Az AmmoDisplay osztály felelős a játékos aktuálisan felszerelt fegyveréhez tartozó lőszerinformáció megjelenítéséért. A lőszer számát vagy egy üres stringet jelez, attól függően, hogy a játékos milyen típusú fegyvert használ (lőfegyver, közelharc fegyver).



3.3.23.1. Inicializálás és Referenciák (Start, OnSceneLoaded):

- Biztosítja, hogy a lőszerkijelző szövegkomponensének (ammoText) GameObject-je aktív legyen.
- Feliratkozik a PlayerEquipment OnWeaponChanged eseményére.
- Meghívja az UpdateAmmoDisplay metódust a jelenet indulásakor.

OnSceneLoaded:

- Egyszerűen újra meghívja a Start metódust.

3.3.23.2. Eseménykezelés és Leiratkozás (OnWeaponChanged, OnDestroy)

Az osztály eseményvezérelt módon reagál a fegyverváltásra.

OnWeaponChanged:

Ez a metódus automatikusan meghívódik, amikor a PlayerEquipment komponens kiváltja az OnWeaponChanged eseményt.

- Meghívja az UpdateAmmoDisplay-t, hogy frissítse a UI-t az új fegyver adatainak megfelelően.

OnDestroy:

Amikor az AmmoDisplay GameObject megsemmisül, ez a metódus lefut, és leiratkoztatja az OnWeaponChanged eseményről (playerEquipment.OnWeaponChanged -= OnWeaponChanged).

3.3.23.3. Kijelző Frissítési Logika (UpdateAmmoDisplay, Update):

UpdateAmmoDisplay:

- Ellenőrzi, hogy a PlayerEquipment és annak CurrentWeapon referenciája érvényes-e.
- Ha van érvényes fegyver:
 - Ha a fegyver tud lőni (weapon.canShoot) és nem közelharci (weapon.isMelee), akkor a fegyver aktuális lőszerkészletét (weapon.currentAmmo) alakítja stringgé és jeleníti meg az ammoText komponensben.
 - Minden más esetben (közelharci fegyver, nem lőhető fegyver, vagy egyéb eset), az ammoText szövegét üres stringre ("") állítja.
- Ha nincs felszerelt fegyver (playerEquipment.CurrentWeapon == null), szintén üres stringet jelenít meg.

Update:

- Hasonló logikát tartalmaz, mint az UpdateAmmoDisplay: ellenőrzi a felszerelt fegyvert, és ha az egy lőfegyver, frissíti az ammoText tartalmát az aktuális lőszerrel.
- A lövés utáni frissítéshez

3.3.23.4. Kijelző Alaphelyzetbe Állítása (ResetDisplay):

- Ellenőrzi, hogy az ammoText referencia érvényes-e.
- Ha igen, az ammoText szövegét üres stringre ("") állítja.

3.3.23.5. Jelenetkezelés (OnEnable, OnDisable, OnSceneLoaded):

OnEnable/OnDisable:

- Feliratkozik/leiratkozik a SceneManager.sceneLoaded eseményre.

OnSceneLoaded:

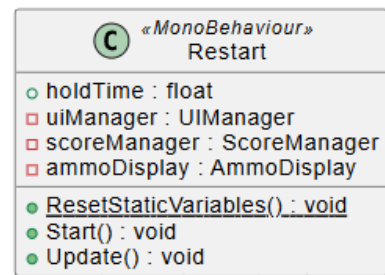
- Újra meghívja a Start() metódust.

3.3.24. Restart

A Restart osztály egy egyszerű funkciót lát el: lehetővé teszi a játékos számára, hogy az aktuális pályát gyorsan újraindítsa az 'R' billentyű lenyomva tartásával egy meghatározott ideig.

3.3.24.1. Statikus Változók Visszaállítása (ResetStaticVariables):

- FloorAccessController.isLevelComplete változót állítja vissza false értékre.



3.3.24.2. Inicializálás (Start):

- Lekéri a UIManager és ScoreManager singleton példányait.
- Megkeresi az AmmoDisplay komponenst a FindFirstObjectByType segítségével.
- Meghívja a ResetStaticVariables() metódust

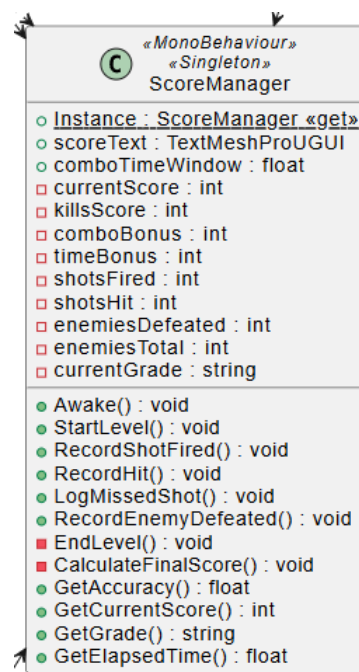
3.3.24.3. Input Kezelés és Újraindítási Logika (Update)

- 'R' Billentyű Lenyomása (Input.GetKeyDown(KeyCode.R)):
 - Beállítja az isHolding statikus logikai változót true-ra
 - Rögzíti a lenyomás időpontját a heldAtTime változóban (Time.time használatával).
- 'R' Billentyű Felengedése (Input.GetKeyUp(KeyCode.R)):
 - isHolding változó visszaáll false-ra
 - heldAtTime nullázódik.

- Ha az `isHolding` igaz (a billentyű le van nyomva), a szkript folyamatosan ellenőrzi, hogy az eltelt idő (`Time.time - heldAtTime`) meghaladta-e a beállított `holdTime` értéket (alapértelmezetten 0.75 másodperc). Ha igen:
 - Az `isHolding` változót visszaállítja `false`-ra, és a `heldAtTime`-ot nullázza, hogy megakadályozza a folyamatos újraindítást.
 - Meghívja a `ResetStaticVariables()` metódust, hogy visszaállítsa a szükséges statikus játékállapotokat. Ez kritikus lépés az újraindítás előtt.
 - A `SceneManager.LoadScene(SceneManager.GetActiveScene().name)` paranccsal újratölti az aktuálisan aktív jelenetet, ezzel hatékonyan újraindítva a pályát.

3.3.25. ScoreManager

A `ScoreManager` osztály a játék központi pontszámítási és teljesítménykövetési rendszere. Singleton, perzisztens módon működik a jelenetek között (`DontDestroyOnLoad`), hogy megőrizze az állapotát. Felelős a játékos pontszámának (alap öléspontszám, kombó bónusz, idő bónusz), a kombó számlálónak, a lövési pontosságnak, az ellenségek számának követéséért, a pálya teljesítési idejének méréséért, a végső osztályzat kiszámításáért, valamint az eredmények átadásáért a pontszám képernyő számára.



3.3.25.1. Singleton és Perzisztencia (Awake):

- Ellenőrzi, hogy létezik-e már `ScoreManager` példány (`Instance`). Ha igen, és az nem ez az objektum, akkor az új példány megsemmisül.
- Ha még nem létezik példány, beállítja az `Instance` statikus változót erre az objektumra.
- Meghívja a `DontDestroyOnLoad(gameObject)` metódust, ami megakadályozza, hogy az objektum megsemmisüljön jelenetváltáskor.

3.3.25.2. Jelenetkezelés és Állapot Visszaállítása (OnEnable, OnDisable, OnSceneLoaded, ResetState):

OnEnable/OnDisable:

- Feliratkozik, illetve leiratkozik a SceneManager.sceneLoaded eseményre.

OnSceneLoaded:

- Meghívja a ResetState metódust az összes belső állapotváltozó (pontszámok, számlálók, időzítők, flag-ek) alaphelyzetbe állításához.
- Ezután egy kis késleltetéssel meghívja a StartLevel metódust az új pálya pontszámításának inicializálásához.

ResetState:

- Alaphelyzetbe állítja az összes pontszámítással kapcsolatos változót.
- Visszaállítja a UI pontszám kijelzőjét (scoreText) "0"-ra.
- Ha fut kombó időzítő Coroutine (comboTimerCoroutine), azt leállítja.

3.3.25.3. Pálya Inicializálása (Start, StartLevel)

Start:

- Beállítja a pontszám kijelző alapértékét
- OnSceneLoaded-hez hasonlóan, késleltetéssel meghívja a StartLevel-t.

StartLevel:

- Nullázza a releváns számlálókat és pontszámokat (shotsFired, shotsHit, killsScore, stb.).
- Rögzíti a pálya kezdési idejét (startTime = Time.time) és beállítja a levelActive flag-et true-ra.
- Megkeresi és megszámolja a jelenetben található összes "Enemy" típusú objektumot. Az eredményt az enemiesTotal változóban tárolja.
- Kiszámít egy célidőt (targetTime) a pálya teljesítéséhez, amely az ellenségek számától függ (minimum 30 másodperc, vagy 6 másodperc ellenségenként). Ez a célidő a későbbi időbónusz és osztályzat számításához szükséges.
- Kitörli a pontszám kijelző szövegét (scoreText.text = "").

3.3.25.4. Pontosság Követése (RecordShotFired, RecordHit, LogMissedShot, GetAccuracy):

RecordShotFired:

- Növeli a kilőtt lövések számát (shotsFired), ha a pálya aktív.
- Naplózza az eseményt és az aktuális pontosságot.

RecordHit:

- Növeli a találatok számát (shotsHit), ha a pálya aktív.
- Naplózza az eseményt és az aktuális pontosságot.

LogMissedShot:

- Naplózza a hibázást és az aktuális pontosságot.

GetAccuracy:

- Kiszámítja és visszaadja a pontosságot a találatok és a kilőtt lövések arányaként.
- Ha még nem történt lövés (shotsFired == 0), 1.0 (100%) értéket ad vissza a nullával osztás elkerülése érdekében.

3.3.25.5. Pontszámítás és Kombó Rendszer (RecordEnemyDefeated, ComboTimer):

RecordEnemyDefeated:

- Növeli az aktuális kombó számlálót (currentCombo) és frissíti az utolsó ölés idejét (lastKillTime).
- Kiszámít egy kombó szorzót (comboMultiplier), amely 10%-kal nő minden egyes kombó szinttel ($1f + (currentCombo - 1) * 0.1f$).
- Kiszámítja az adott ölésért járó pontszámot (scoreForKill) az alap öléspontszám (baseKillScore) és a kombó szorzó alapján.
- Hozzáadja a kapott pontot a teljes öléspontszámhoz (killsScore) és a kombó bónuszhoz (comboBonus).
- Frissíti a teljes pontszámot (currentScore = killsScore) és a UI-t (UpdateScoreUI).
- Újraindítja a kombó időzítő Coroutinet (ComboTimer).
- Növeli a legyőzött ellenségek számát (enemiesDefeated).
- Ellenőrzi, hogy ez volt-e az utolsó ellenség (enemiesDefeated >= enemiesTotal).
- Ha igen, meghívja az EndLevel metódust.

ComboTimer (Coroutine):

- Vár a comboTimeWindow másodpercig.
- A várakozás után ellenőrzi, hogy az utolsó ölés óta valóban eltelt-e a comboTimeWindow idő. Ha igen, akkor a kombó lejár, és a currentCombo számláló nullázódik.

3.3.25.6. Pálya Befejezése (EndLevel):

- Rögzíti a pálya befejezési idejét (endTime = Time.time) és beállítja a levelActive flag-et false-ra.
- Meghívja a CalculateFinalScore metódust a végső pontszám és osztályzat kiszámításához.
- Naplózza a végső pontosságot.
- Az eredményeket (killsScore, comboBonus, timeBonus, accuracy, finalScore, grade, completionTime) átmásolja a ScoreScreenManager statikus változóiba.
- Beállítja a FloorAccessController.isLevelComplete statikus változót true-ra
- Megjeleníti a Level Complete képernyőt

3.3.25.7. Végső Pontszám és Osztályzat Számítása (CalculateFinalScore, CalculateGrade):

CalculateFinalScore:

- Kiszámítja az eltelt időt (elapsedTime) és lekéri a pontosságot (accuracy).
- Kiszámítja az időbónuszt (timeBonus) a targetTime és az elapsedTime különbsége alapján (minden megspórolt másodpercért 5 pont jár).
- Alkalmaz egy pontosság szorzót (accuracyMultiplier) az ölésekből és kombóból származó pontszámra. A szorzó minimum 0.1, maximum 1.0.
- A végső pontszám (currentScore) az így kapott pontszám és az időbónusz összege.
- Meghívja a CalculateGrade metódust az osztályzat meghatározásához.

CalculateGrade:

- Meghatározza a végső osztályzatot ("SS", "S", "A", "B", "C", "D") elsősorban a pontosság alapján (85%+ S, 65%+ A, stb.).
- Ezután módosíthatja az osztályzatot egy szinttel feljebb vagy lejjebb attól függően, hogy a játékos ideje kivételesen jó (targetTime 60%-a alatt) vagy kivételesen rossz (targetTime 150%-a felett) volt-e.

3.3.25.8. UI Frissítés (UpdateScoreUI):

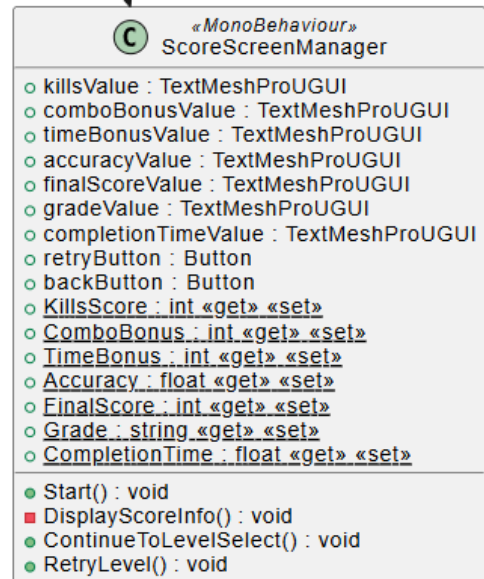
- Ellenőrzi, hogy a scoreText referencia érvényes-e.
- Ha igen, beállítja a text tulajdonságát az aktuális pontszámra (currentScore).

3.3.25.9. Adat Lekérdező Metódusok (Getters):

Számos publikus getter metódus (GetCurrentScore, GetKillsScore, GetComboBonus, GetAccuracy, GetEnemiesDefeated, GetTotalEnemies, GetTimeBonus, GetGrade, GetElapsedTime, GetShotsFired, GetShotsHit) biztosít hozzáférést a ScoreManager által számított és tárolt adatok számára.

3.3.26. ScoreScreenManager

A ScoreScreenManager osztály felelős a pálya teljesítése után megjelenő pontszám képernyő kezeléséért. Feladata, hogy megjelenítse a játékos teljesítményének részleteit (pontszámok, bónuszok, pontosság, idő, osztályzat), amelyeket statikus változókon keresztül kap meg, valamint kezelje a képernyőn található navigációs gombok működését.



3.3.26.1. Statikus Adattárolók

Az osztály számos publikus statikus property-t definiál (KillsScore, ComboBonus, TimeBonus, Accuracy, FinalScore, Grade, CompletionTime).

3.3.26.2. Inicializálás (Start):

- Meghívja a DisplayScoreInfo metódust, hogy a statikus változóknál tárolt adatokkal feltöltse a felhasználói felület megfelelő TextMeshProUGUI elemeit.
- Hozzáadja a listener függvényeket a retryButton és backButton gombok onClick eseményeihez.

3.3.26.3. Eredmények Megjelenítése (DisplayScoreInfo):

- Végigmegy a különböző TextMeshProUGUI referenciákon (killsValue, comboBonusValue, stb.).
- Minden egyes elem esetén ellenőrzi, hogy a referencia nem null-e.
- Ha a referencia érvényes, beállítja a text tulajdonságát a megfelelő statikus property értékének string reprezentációjára.

3.3.26.4. Navigációs Műveletek (ContinueToLevelSelect, RetryLevel):

ContinueToLevelSelect:

- Megpróbálja megkeresni a jelenetben az első CanvasGroup komponenst a FindFirstObjectByType<CanvasGroup>() segítségével. Ha talál egyet, annak alpha értékét 0-ra állítja.
- Betölti a LevelSelect jelenetet.

RetryLevel:

- Visszaállítja a FloorAccessController.isLevelComplete statikus változót false-ra.
- A ContinueToLevelSelect-hez hasonlóan megpróbálja megtalálni és visszaállítani az első CanvasGroup komponenst.
- Lekéri az utoljára játszott pálya nevét a PlayerPrefs-ből
- Betölti a lekért nevű jelenetet

3.3.27. LevelSelectManager

A LevelSelectManager osztály felelős a pályaválasztó képernyő működéséért. Kezeli a felhasználói interakciókat és elindítja a kiválasztott pálya betöltési folyamatát, ami a Power-Up választó képernyőn keresztül történik.

3.3.27.1. Adatszerkezet (LevelData)

Az osztály egy beágyazott, szerializálható LevelData segédosztályt használ az egyes pályákhoz tartozó információk tárolására:

«MonoBehaviour» LevelSelectManager	
○ levelButtonPrefab : GameObject	
○ levelButtonsContainer : Transform	
○ backButton : Button	
○ availableLevels : List<LevelData>	
○ fadeCanvasGroup : CanvasGroup	
class LevelData {	
○ levelName : string	
○ sceneName : string	
○ levelPreview : Sprite	
○ isLocked : bool	
},	
● Start() : void	
■ CreateLevelButtons() : void	
■ SelectLevel(int index) : void	
■ LoadSelectedLevel() : void	

- **levelName:** A pályaválasztó gombon megjelenő név (string).
- **sceneName:** A pályához tartozó jelenet fájlneve (string), amelyet a SceneManager használ a betöltéshez.
- **levelPreview:** A pályaválasztó gombon megjelenő előnézeti kép (Sprite).
- **isLocked:** Logikai érték, amely jelzi, hogy a pálya zárolt-e.

Az availableLevels (List<LevelData>) publikus lista tartalmazza az összes elérhető pálya adatait, amelyeket a Unity Inspector felületén kell konfigurálni.

3.3.27.2. Inicializálás (Start):

- Meghívja a `CreateLevelButtons` metódust.
- Beállítja a `backButton` gomb `onClick` eseménykezelőjét.

3.3.27.3. Pályagombok Dinamikus Létrehozása (`CreateLevelButtons`):

- Végigiterál az `availableLevels` listán.
- Minden egyes `LevelData` elemhez:
 - Létrehoz egy új példányt a `levelButtonPrefab`-ból a `levelButtonsContainer` gyermekeként.
 - Lekéri a létrehozott objektum `Button` komponensét.
 - Megkeresi a gombon belüli `TextMeshProUGUI` komponenst, és beállítja annak szövegét a `levelData.levelName` értékére.
 - Lekéri a gomb `Image` komponensét, és ha van hozzárendelt `levelPreview` sprite, beállítja azt a gomb képének, megőrizve a képarányt (`preserveAspect = true`).
 - Ellenőrzi a `levelData.isLocked` értékét. Ha a pálya zárolt, a gomb `interactable` tulajdonságát `false`-ra állítja.
 - Hozzáad egy `onClick` eseménykezelőt a gombhoz.

3.3.27.4. Pálya Kiválasztása (`SelectLevel`):

- Beállítja a `selectedLevelIndex` privát változó értékét a paraméterként kapott indexre. Ez az index az `availableLevels` listában elfoglalt helyet jelöli.

3.3.27.5. Kiválasztott Pálya Betöltési Folyamata (`LoadSelectedLevel`):

- Ellenőrzi, hogy a `selectedLevelIndex` érvényes index-e az `availableLevels` listában.
- Lekéri a kiválasztott pályához tartozó `LevelData` objektumot.
- Ellenőrzi, hogy a kiválasztott pálya nincs-e zárolva (`!levelToLoad.isLocked`).

- Ha a pálya elérhető:
 - Visszaállítja a FloorAccessController.isLevelComplete statikus változót false-ra.
 - A betöltendő jelenet nevét (levelToLoad.sceneName) eltárolja a sceneToLoad változóban.
 - A kiválasztott pálya jelenetnevét elmenti a PlayerPrefs-be a SelectedLevel kulcs alatt.
 - Betölti a PowerUpSelect jelenetet a SceneManager.LoadScene segítségével. A játékmenet innen folytatódik, lehetővé téve a játékosnak, hogy power-upokat válasszon a pálya megkezdése előtt.

A jelenlegi Buildben nincs implementálva rendszer automatikus pálya zárolás és nyitásra!

3.3.27.6. UI Referenciák és Egyéb Változók:

- **levelButtonPrefab:** A pályagombok létrehozásához használt előre gyártott UI elem (GameObject).
- **levelButtonsContainer:** Az a Transform komponens (általában egy Panel vagy Layout Group), amely alá a pályagombok dinamikusan létrehozásra kerülnek.
- **backButton:** A főmenübe való visszalépést biztosító gomb (Button).
- **availableLevels:** A választható pályák adatait tartalmazó lista (List<LevelData>), amelyet az Inspectorban kell feltölteni.

3.3.28. MainMenuManager

Az osztály a játék főmenüjének működését és felhasználói felületét vezérli.

3.3.28.1. Inicializálás és Gombkezelés (Start):

- Kezdetben elrejtí az opciók panelt (optionsPanel.SetActive(false)) és megjeleníti a főmenü panelt (mainMenuPanel.SetActive(true)).
- Ellenőrzi, hogy a menuButtons listában van-e legalább három gomb. (Start, Opciók, Kilépés).

«MonoBehaviour» MainMenuManager
<ul style="list-style-type: none"> ○ mainMenuPanel : GameObject ○ optionsPanel : GameObject ○ menuButtons : List<Button> ○ controlsImage : Image ○ backButton : Button
<ul style="list-style-type: none"> ● Start() : void ■ ShowOptionsMenu() : void ■ HideOptionsMenu() : void ■ QuitGame() : void

- Ha elegendő gomb van:
 - Az elsőhöz hozzáad egy onClick eseménykezelőt. Ez egy lambda kifejezést `() => { ... }` használ, ami egy rövid, névtelen függvény definiálására szolgál. Amikor a gombra kattintanak, ez a névtelen függvény lefut: először meghívja a `ResetGameState()` metódust, majd betölti a "LevelSelect" jelenetet a `SceneManager.LoadScene` segítségével.
 - A második gombhoz hozzárendeli a `ShowOptionsMenu` metódust listenerként.
 - A harmadik gombhoz hozzárendeli a `QuitGame` metódust listenerként.
- Beállítja a `backButton` (az opciók menü visszalépés gombja) onClick eseménykezelőjét, amely a `HideOptionsMenu` metódust hívja meg.

Az onClick eseményeket közvetlenül a szerkesztőben is hozzá lehet adni.

3.3.28.2. Panelváltás (`ShowOptionsMenu`, `HideOptionsMenu`):

`ShowOptionsMenu`:

- Elrejtí a főmenü panelt (`mainMenuPanel.SetActive(false)`) és megjeleníti az opciók panelt (`optionsPanel.SetActive(true)`).

`HideOptionsMenu`:

- Megjeleníti a főmenü panelt (`mainMenuPanel.SetActive(true)`) és elrejtí az opciók panelt (`optionsPanel.SetActive(false)`).

3.3.28.3. Játékállapot Visszaállítása (`ResetGameState`):

- `FloorAccessController.isLevelComplete` statikus változót állítja false-ra.

3.3.28.4. Kilépés a Játékból (`QuitGame`):

Ez a privát metódus kezeli a játékból való kilépést.


- Meghívja az `Application.Quit()` metódust, amely bezárja a futó alkalmazást.

3.3.28.5. UI Referenciák:

- **mainMenuPanel:** A főmenü elemeit tartalmazó fő panel (GameObject).
- **optionsPanel:** Az opciók menü elemeit tartalmazó panel (GameObject).
- **menuButtons:** A főmenü gombjait (Start, Opciók, Kilépés) tartalmazó lista (List<Button>). A sorrend fontos.
- **backButton:** Az opciók menüből a főmenübe való visszalépést biztosító gomb (Button).
- **controlsImage:** Placeholder, a játékban lévő tutorial pálya miatt nem implementált.

3.3.29. VolumeManager

A VolumeManager osztály felelős a játék globális hangerejének beállításáért és mentéséért. Egy UI csúszkát (Slider) használ a hangerő vizuális reprezentálására és módosítására, és a PlayerPrefs rendszert alkalmazza a beállítás megőrzésére a játékmenetek között.

	«MonoBehaviour» VolumeManager
○	volumeSlider : Slider
●	Start() : void
●	SetVolume(float volume) : void

3.3.29.1. Inicializálás (Start):

- Ellenőrzi a PlayerPrefs segítségével, hogy létezik-e már mentett érték a "MasterVolume" kulcs alatt (PlayerPrefs.HasKey("MasterVolume")).
- Ha nem létezik mentett érték (pl. a játék első indításakor):
 - Beállít egy alapértelmezett hangerőt (1.0f, azaz 100%) a PlayerPrefs-ben a "MasterVolume" kulcsra (PlayerPrefs.SetFloat).
 - Ezután meghívja a LoadVolume() metódust, hogy ezt az alapértelmezett értéket betöltse a csúszkába.
- Ha már létezik mentett érték:
 - Közvetlenül meghívja a LoadVolume() metódust, hogy a korábban mentett értéket betöltse a csúszkába.

3.3.29.2. Hangerő Beállítása (SetVolume):

A volumeSlider OnValueChanged eseményéhez van hozzárendelve a Unity Inspectorban.

- A globális hangerőt az AudioListener.volume statikus tulajdonság beállításával módosítja. Az új értéket közvetlenül a volumeSlider.value tulajdonságból veszi. Az AudioListener.volume 0.0 (néma) és 1.0 (teljes hangerő) közötti értéket vár.

- A hangerő beállítása után azonnal meghívja a `SaveVolume()` metódust, hogy az új értéket elmentse.

3.3.29.3. Hangerő Mentése (`SaveVolume`):

Elmenti az aktuális hangerő beállítást.

- A `PlayerPrefs.SetFloat` metódussal elmenti a `volumeSlider.value` aktuális értékét a "MasterVolume" kulcs alá.

3.3.29.4. Hangerő Betöltése (`LoadVolume`):

Betölti a mentett hangerő beállítást.

- A `PlayerPrefs.GetFloat("MasterVolume")` metódussal lekéri a "MasterVolume" kulcs alatt tárolt értéket.
- A lekérdezett értéket beállítja a `volumeSlider.value` tulajdonság értékének, így a csúszka vizuálisan is a mentett hangerőt mutatja.

4. Összefoglalás, továbbfejlesztési lehetőségek

A játék váza, alapkonceptiója kész, ám messze áll egy valóban kiadható termék mélységétől. A meglévő, implementált komponensek és összetevők elegendőek és kellően polírozottak ahhoz, hogy egyszer egy rendesen kidolgozott cím készülhessen a projektből. A létrehozott erőforrások valóban könnyen skálázhatóvá teszik a szoftvert.

- Az ellenségek és a játékos viselkedése inkonzisztens abban az esetben, amikor egy ajtó választja el őket egymástól
- A fényhatásokkal jelenleg meg vagyok elégedve, de további finomhangolásra később sor kerülhet
- A pontozási rendszer tökéletesítése

Skálázhatósági, Bővítési lehetőségek:

- Több pálya készítése (könnyen megoldható a létező erőforrásokkal)
- További fegyverek hozzáadása (szintén könnyen megoldható)
- Eredetileg több, speciális képesség is tervben volt, ezekről nem mondtam le. Ilyenekre példa:
 - Portálfegyver (Hasonlóan a Portal és Mario játékokban szereplőhöz)
 - Rakéta „ugrás”: Egy rakétavetővel a falaktól való gyors eltaszítása a játékosnak
- Sztori, párbeszéd panel:
 - A játékhoz létezik egy kidolgozott történetyszál. Sajnos időhiány miatt ezt nem tudtam kivitelezni.
- Zenei aláfestés, további hangeffektek

5. Irodalomjegyzék, harmadik féltől származó felhasználói erőforrások

(hozzáférés: 2025.04.28)

- [1] <https://docs.unity3d.com/Manual/system-requirements.html#desktop>

Unity dokumentáció „Unity Player 6 minimális rendszerkövetelmények”

- [2] https://youtu.be/DTp5zi8_u1U?si=a3Vr-WuTatuLu6tR

Unity Tilemap Tutorial

- [3] https://youtu.be/YhrwKF_i-BI?si=qEbJmZliDz2uCqEN

Unity URP bemutató

- [4] <https://www.amazon.com/Game-Programming-Patterns-Robert-Nystrom/dp/0990582906>

Game Programming Patterns

Szerző: Robert Nystrom



ábra 8: Könyv: Game Programming Patterns

- [5] <https://raw.githubusercontent.com/adxv/ferocitygamep/refs/heads/main/plantuml-diagram.png>

Teljes UML Állapotgép diagram a programszerkezetről

- [6] https://youtu.be/7iYWpzL9GkM?si=jd_S_0KgGPq8DMZW

2D felülnézetes Pixel Art RPG játék Tutorial

- [7] <https://youtu.be/Ep-nJNHPc4?si=hMGMP-pAfXJVEfcl>

Videó az Aseprite használatáról és Pixel-Art alapokról

- [8] <https://youtu.be/yWCHaTwVblk?si=UBSuRfuzncd4DL7Q>

Hangerőszabályzó és PlayerPrefs alapok

- [9] <https://lospec.com/palette-list/ufo-50>

Felhasznált színpaletta

- [10] <https://www.dafont.com/pixel-operator.font>
<https://www.dafont.com/x-typewriter.font>

Felhasznált betűtípusok

- [11] <https://mixkit.co/free-sound-effects/>

Fegyverek hangjainak forrása