**Szegedi Tudományegyetem**

**Informatikai Intézet**

**2D Horde Survival Roguelite játék**

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | **Szabó Gergő** |  | **Dr. Jász Judit** |  |
|  | Programtervező informatikus szakos hallgató |  | Egyetemi adjunktus |  |

Szeged

2025

# FELADATKIÍRÁS

A szakdolgozat témája egy 2D horde survival roguelite játék, Godot-ban kiépítve GDScript-et használva. A játékosnak végtelen hullámokat kell túlélnie, miközben folyamatosan fejleszti képességeit és fegyvereit. A játékmenet aktívabb felhasználói élményt nyújt a hagyományos horde survival játékokhoz képest. A játék célja, hogy a játékos minél erősebb legyen így annál tovább bírja a harcot, eközben valamilyen formában pénzt szerez, amit állandó fegyver és képesség fejlesztésre használhat két kör között.

# TARTALOMJEGYZÉK

[FELADATKIÍRÁS 1](#_Toc1739753935)

[TARTALOMJEGYZÉK 2](#_Toc1497479228)

[TARTALMI ÖSSZEFOGLALÓ 2](#_Toc1833054696)

[BEVEZETÉS 2](#_Toc1676873632)

[1. TECHNOLÓGIÁK BEMUTATÁSA 2](#_Toc1023412149)

[1.1 Godot 2](#_Toc376619411)

[1.1.1 AnimationTree és AnimationNode rendszer 2](#_Toc1051268354)

[1.1.2 TileMap és TileSet 2](#_Toc1114132147)

[1.2 GDScript 2](#_Toc139436151)

[1.3 Git 2](#_Toc950186036)

[1.4 Pixilart 2](#_Toc1304355762)

[2. SPECIFIKÁCIÓK 2](#_Toc5830874)

[2.1. Játékmenet 2](#_Toc1134314834)

[2.1. UI 2](#_Toc1283435291)

[2.2. Player 2](#_Toc933123404)

[2.2.2. Élet, sebzés, halál, szintlépés 2](#_Toc43520015)

[2.2.3. Interakciók 2](#_Toc1781760126)

[2.2.4. Fegyverek 2](#_Toc1951734581)

[2.3. Környezet 2](#_Toc447285563)

[2.4. Ellenségek 2](#_Toc534851568)

[2.4.1 Elit típusok 2](#_Toc1424017775)

[2.5. Fejlesztések 2](#_Toc825772227)

[2.6 Mentések 2](#_Toc787655285)

[3. TERVEZÉS 2](#_Toc2015239132)

[3.1. Játékos 2](#_Toc1986349642)

[3.1.1. Moduláris karakterfelépítés 2](#_Toc1315680385)

[3.1.2. Statok összevonása komponensekből 2](#_Toc396865284)

[3.1.3. Életkomponens és sebzéskezelés 2](#_Toc1408544053)

[3.1.4. Vizuális megjelenítés komponensenként 2](#_Toc537129165)

[3.1.5. Állapotgép (State Machine) és animáció kezelés 2](#_Toc261767886)

[3.1.7. Mozgásmechanika 2](#_Toc1889692304)

[3.1.6. Fegyverkomponens rendszer 2](#_Toc1067916772)

[3.2. Ellenségek 2](#_Toc542974319)

[3.3.1. Alapfelépítés és viselkedés 2](#_Toc1234289559)

[3.3.2. Elit ellenségek és stratégia minta 2](#_Toc315064794)

[3.3.3. Ellenség spawner rendszer 2](#_Toc2137028390)

[3.4. Környezet 2](#_Toc665560343)

[3.4.1. Körön belüli pályagenerálás 2](#_Toc1181861088)

[3.4.2. Körök közötti központi tér 2](#_Toc1415398884)

[3.5. Fejlesztések 2](#_Toc1789079125)

[3.5.1. Fejlesztéstípusok kezelése 2](#_Toc1222243018)

[3.5.2. Fejlesztés alkalmazása 2](#_Toc94133900)

[3.6. Mentési rendszer 2](#_Toc1658835096)

[3.6.1. Beállítások mentése 2](#_Toc1625435554)

[3.6.2. Játékadatok mentése Resource-ba 2](#_Toc64441420)

[3.6.3. Futásidejű adatok: RuntimeSaves globális script 2](#_Toc1648375290)

[4. MEGVALÓSÍTÁS 2](#_Toc1175341823)

[ÖSSZEGZÉS 2](#_Toc1233508621)

[IRODALOMJEGYZÉK 2](#_Toc52154668)

[KÖSZÖNETNYÍLVÁNÍTÁS 2](#_Toc1813597999)

[MELLÉKLETEK 2](#_Toc2054844643)

# TARTALMI ÖSSZEFOGLALÓ

* *A téma megnevezése:*

2D Horde Survival Roguelite játék

* *A megadott feladat megfogalmazása:*

Egy horde survivor típusú, PC-re készülő játék fejlesztése, amely a klasszikus műfaji elemeket (folyamatosan érkező ellenséghullámok, időalapú túlélés) ötvözi egy aktívabb, játékosközpontú játékmenettel. A cél egy olyan játékélmény kialakítása, ahol a játékos nem pusztán a fejlesztésekre támaszkodva túlél, hanem döntéseivel és aktív beavatkozásaival (pl. támadás, kitérés, interakciók) alakítja a játékmenetet.

* *A megoldási mód:*

A játék a Godot játékmotor 4-es verziójának felhasználásával készült, GDScript programozási nyelv alkalmazásával. A játékmenet vizuális elemei pixel art stílusban készültek.

* Alkalmazott eszközök, módszerek,

Godot Engine – nyílt forráskódú játékmotor, amely lehetővé teszi a 2D-s játékfejlesztést hatékonyan és platformfüggetlenül.

GDScript – Godot saját szkriptnyelve, amelyet a játék logikájának és rendszereinek implementálására használtam.

Pixilart – online pixel art szerkesztő, amelyet a játék grafikai elemeinek (karakterek, ellenfelek, környezeti elemek) létrehozásához használtam.

* *Elért eredmények:*

A projekt során elkészült egy 2D horde survivor játék, amely több szempontból is eltér a klasszikus Vampire Survivors-féle megközelítéstől. A játékos aktívan vált a fegyverei között, kitérhet támadások elől, valamint döntései közvetlen hatással vannak a játékmenetre. A játék tartalmaz egy moduláris ellenség-spawn rendszert, különböző típusú fegyvereket, fejlesztéseket, valamint egy skálázható nehézségi rendszert. A vizuális megjelenés egységes pixel art stílusban valósult meg.

* *Kulcsszavak:*

Horde Survivor, Godot, komponens, Node, GdSript

# BEVEZETÉS

Az elmúlt évek során a horde survivor típusú játékok, különösen a Vampire Survivors és az ahhoz hasonló címek megjelenésével, jelentős népszerűségre tettek szert. Ezek a játékok egyszerű irányíthatóságuk és rendkívül intenzív, gyakran bullet hell-re emlékeztető játékmenetük révén nagy játékosbázist építettek ki. A képernyőt betöltő ellenségek, lövedékek és effektek kaotikus, mégis élvezetes játékélményt teremtenek, miközben a játékos különféle fejlesztések közül választhat a túlélés érdekében.

A dolgozatban bemutatott játék egy, a klasszikus horde survivor elemeket felhasználó, de azokat kibővítő és továbbgondoló 2D akciójáték, amelyet a roguelite műfaj alapelveinek megfelelően terveztem meg. Ennek megfelelően a játékmenet egyik alapvető eleme, hogy a játékos az egyes körök (run-ok) során megszerzett fejlesztések egy részét megőrizheti, így minden új körben erősebben indulhat, mint korábban. Ez a rendszer nemcsak újrajátszhatóságot biztosít, hanem azt az érzetet is, hogy a játékos folyamatosan fejlődik, és tapasztalatait, döntéseit hosszabb távon kamatoztathatja.

Ugyanakkor a klasszikus horde survivor játékok egyik hátrányaként említhető, hogy a játékmenet előrehaladtával a játékos közvetlen kontrollja csökken. A legtöbb esetben a karakter automatikusan támad, a játékos szerepe pedig az időnként felbukkanó fejlesztések kiválasztására korlátozódik. Ezzel párhuzamosan a játék akcióintenzitása folyamatosan nő, ami bár kezdetben izgalmas, hosszabb távon a játékos mentális fáradtságához és az élmény "elfásulásához" vezethet.

A saját fejlesztésű játék egyik célja éppen ennek a problémának a kezelése. A játékmenet során minimális szerepet kapnak az automatikus, passzív effektek, helyettük nagy hangsúlyt fektettem az aktív játékosinterakciókra. A játékos két fegyvertípust használhat: egy elsődleges fegyvert, amellyel közvetlenül támad, és egy másodlagosat, amelynek használatához lőszert kell gyűjteni. A lőszertermelés az elsődleges fegyverrel végzett támadások során történik, így a két rendszer szoros kapcsolatban áll egymással. Ez a mechanika arra ösztönzi a játékost, hogy aktívan részt vegyen a harcban, és ügyességével, döntéseivel közvetlenül befolyásolja a túlélési esélyeit.

A játékmenet kialakítása során kiemelt figyelmet fordítottam az akciódús és pihentető szakaszok megfelelő váltakozására is. Az intenzív, bullet hell-szerű harci fázisokat (úgynevezett "uptime" szakaszokat) a körök végén, illetve a játékos halálakor egy rövidebb levezető vagy fejlesztési szakasz ("downtime") követi. Ez lehetőséget ad az értékelésre, tervezésre, és pihenésre is, így segít elkerülni, hogy a játékos belefáradjon a folyamatos akcióba, miközben fenntartja az élmény frissességét.

Összességében a cél egy olyan roguelite horde survivor játék létrehozása volt, amely ötvözi a műfaj pozitívumait – a fejlődés érzetét, a kaotikus, izgalmas játékmenetet – az aktív, játékosközpontú irányítással és az átgondolt ritmussal, amely váltakoztatja a feszültséggel teli és nyugodtabb szakaszokat.

# 1. TECHNOLÓGIÁK BEMUTATÁSA

## **1.1 Godot**

A játék fejlesztéséhez a nyílt forráskódú Godot Engine legfrissebb stabil verzióját használtam. A Godot egy gyorsan fejlődő, rugalmas és könnyen tanulható játékmotor, amely különösen alkalmas 2D játékok készítésére, de 3D-s képességekkel is rendelkezik.

A Godot Engine eredetileg Juan Linietsky és Ariel Manzur által indult projektként jött létre 2007-ben, és 2014-ben vált hivatalosan nyílt forráskódúvá (MIT licenc alatt). A motor azóta is aktív közösségi támogatással és folyamatos fejlesztéssel bővül, egyre több funkcióval és eszközzel támogatva a fejlesztőket. A Godot célja, hogy egy teljes értékű, platformfüggetlen fejlesztési környezetet biztosítson független fejlesztők és kisebb csapatok számára.

A Godot moduláris Node alapú rendszere lehetővé teszi, hogy a játék különböző elemei (karakterek, UI, háttérrendszerek) önálló, jól elkülönített komponensekből épüljenek fel. Ez a struktúra nagy mértékben támogatja az újrafelhasználhatóságot, a fejlesztés átláthatóságát és a kód karbantarthatóságát.

A játékom fejlesztése során kiemelten használtam a következő beépített rendszereket:

### 1.1.1 AnimationTree és AnimationNode rendszer

A karakterek és ellenfelek mozgásait és támadásait az AnimationTree rendszer segítségével könnyedén tudtam animálni. Ez lehetőséget biztosít állapotgépek (StateMachine) létrehozására, melyekkel különböző animációs állapotokat kezelhetünk például a mozgást, támadást és sebződés. A rendszer támogatja a paraméterek alapján történő dinamikus váltást és az animációk keverését is, ami különösen fontos volt a játék gyors tempójú játékmenetéhez.

### 1.1.2 TileMap és TileSet

A pálya elemit a TileMap és TileSet rendszerek segítségével helyeztem el és kezeltem. A TileSet segítségével előre definiáltam a csempék grafikáját, valamint azokhoz tartozó metaadatokat például vizuális rétegeket és ütközéshez szükséges területeket. A TileMap rendszerrel pedig gyorsan, hatékonyan tudtam generálni pályákat, és a kézzel készített területeket.

## 1.2 GDScript

A Godot saját szkriptnyelvét, a GDScriptet használtam a játék logikájának megvalósításához. A nyelv szintaxisa erősen hasonlít a Pythonhoz, ezáltal könnyen tanulható, ugyanakkor kifejezetten a Godot motor működéséhez optimalizált, így gyors és hatékony fejlesztést tesz lehetővé.

A játék fejlesztése során komponensalapú architektúrát alkalmaztam, amelyben az egyes játékelemek (például játékos, ellenségek, fegyverek, projektilok, effektek) saját, különálló szkripttel rendelkező Node-okból állnak. Ez a struktúra elősegítette a modularitást, a kód újrafelhasználhatóságát, és a tesztelhetőséget.

A GDScript egyik leghasznosabb funkciója a signal rendszer, amely események kezelését teszi lehetővé laza csatolással a különböző komponensek között. A signal rendszer lehetővé teszi, hogy a komponensek ne ismerjék egymást közvetlenül, így az egyes rendszerek szabadon bővíthetők vagy újrafelhasználhatók más projektekben is.

A játékmenet során emellett számos állapotkezelést (state machine) is GDScriptben valósítottam meg, különösen a játékos viselkedését, a fegyverek működését és az ellenségek viselkedésmintáit illetően. A GDScript rugalmassága lehetővé tette, hogy a játékmenet fontosabb részeit gyorsan iteráljam és többször újratervezzem a tesztelés alapján.

## 1.3 Git

A fejlesztés során verziókezeléshez a Git rendszert használtam, amely a szoftverfejlesztés egyik legelterjedtebb verziókövető eszköze. A Git lehetővé teszi, hogy a projekt teljes fejlesztési története visszakövethető legyen, a különböző fejlesztési irányok pedig biztonságosan elválaszthatók és szükség esetén összevonhatók legyenek.

A Git használatával elkerülhetőek a véletlen adatvesztések, és hatékonyabban lehet új funkciókat kipróbálni vagy hibákat visszakeresni.

A Git parancssoros használata helyett a vizuális felületet kínáló GitHub Desktop alkalmazást használtam, amely egyszerűbbé és gyorsabbá tette a commitolás és a szinkronizálás folyamatát. A projekt távoli tárolására a GitHub szolgáltatását vettem igénybe, így a munkám biztonsági mentése mindig elérhető volt online.

## 1.4 Pixilart

A játékhoz szükséges 2D sprite-ok és animációs képkockák elkészítését Pixilart nevű online pixel art szerkesztő segítségével végeztem. A program egyszerű felületet kínál a pixelgrafikák készítéséhez, támogatja a rétegeket, az animált sprite készítést, valamint lehetőséget ad az exportálásra átlátszó háttérrel, amit közvetlenül be tudtam importálni a Godot projektbe.

# 2. SPECIFIKÁCIÓK

## 2.1. Játékmenet

A játék egy köralapú, folyamatosan nehezedő rendszerre épül, amely a roguelite műfaj alapjaira támaszkodik. A játékos minden kör elején eldöntheti, milyen nehézségi módosítókkal szeretné elindítani a túlélési szakaszt. Ezek a módosítók különböző negatív hatásokkal járnak, ugyanakkor a kör végén nagyobb jutalmazással járnak, így stratégiai döntések elé állítják a játékost. A nehézség fokozása történhet például az ellenségek számának növelésével, statjaik – mint sebesség, életerő vagy sebzés – emelésével, illetve olyan speciális módosítók bevezetésével, amelyek a játékos képességeit befolyásolják. Egy példa erre a Tank effektus, amely drasztikusan lelassítja a játékost, cserébe megduplázza az életerejét. Ezek a mechanikák lehetőséget adnak a játékos számára, hogy saját kihívást szabjon magának, és ezzel együtt növelje a kör végén kapható jutalmak értékét.

A játékmenet ciklikusan épül fel: a játékos egy akciódús túlélési szakaszban (uptime) harcol a folyamatosan érkező ellenséghordákkal, majd egy kör végi szünet (downtime) során új fejlesztésekből választhat, vagy a karakterét alakíthatja tovább. Ez a szünet nemcsak lehetőséget ad a stratégiai döntésekre, hanem tudatosan kezeli az akció és pihenés váltakozását, hogy a játékos ne fáradjon bele a folyamatos harcba. A túlzott akcióterhelés (akciósűrűség) rendszeres megtörése segít abban, hogy a játék ne váljon kimerítővé, és mindig frissen tudja tartani az élményt.

A downtime szakasz emellett nem csak a fejlesztésről szól: a játékos egy külön területen mozoghat, ahol többféle interakciós lehetőség is rendelkezésre áll. Itt találhatók az edzőbábuk, amelyek lehetőséget nyújtanak az elsődleges és másodlagos fegyverek kipróbálására, a támadások hatásmechanizmusának gyakorlására, vagy újonnan megszerzett fejlesztések és képességek azonnali tesztelésére. Ez a tesztelési lehetőség különösen hasznos a játékos számára, hiszen így kockázat nélkül megismerkedhet az új eszközökkel, és felkészülhet a következő kör kihívásaira.

A körök közötti fejlődési rendszer központi eleme a játék hosszú távú motivációs struktúrájának. A játékos az egyes körök során gyűjtött erőforrásokból különböző permanens fejlesztéseket oldhat fel. Ezek között szerepelnek:

* Közelharci fegyverek, amelyeket egy fejlesztési fa (skill tree) segítségével lehet testre szabni, új támadásokat, effekteket és viselkedéseket adva hozzájuk.
* Másodlagos fegyverek, amelyekből a játékos választhat, és amelyek szorosan kapcsolódnak az elsődleges fegyverhez (pl. az elsődleges támadások töltik a másodlagos fegyver lőszerét).
* Alap statisztikák fejlesztése, mint mozgási sebesség, maximális életerő, vagy támadási sebesség.
* Kinézet változtatás, különböző ruhák és felszerelések feloldásával, amelyek nemcsak esztétikai célokat szolgálnak, hanem akár statisztikai bónuszokat is adhatnak.

A cél az, hogy a játékos minden újabb körrel érezze a fejlődést, és egyre mélyebb mechanikai rétegek nyíljanak meg előtte, miközben a játék folyamatosan friss kihívásokat kínál. A roguelite struktúra, a növekvő nehézség, a személyre szabható kihívások és a pihenést kínáló downtime-rendszer együttesen biztosítják, hogy a játék hosszú távon is motiváló és újrajátszható maradjon.

## 2.1. UI

A játék felhasználói felülete több különálló, de egymással összhangban működő részből áll, amelyek célja, hogy a játékos számára átlátható, informatív és könnyen kezelhető játékélményt nyújtsanak. A játék indításakor a főmenü jelenik meg, ahol a játékos elindíthatja az új játékot, beléphet a beállításokba vagy kiléphet az alkalmazásból. A beállítások menü lehetőséget biztosít a képernyőfelbontás megváltoztatására, a teljes képernyős mód ki- és bekapcsolására, valamint a kontrollok testreszabására.

Játék közben a HUD (Heads-Up Display) szolgáltatja a játékos számára a legfontosabb visszajelzéseket. A képernyőn megjelenik az aktuális lőszer mennyisége a másodlagos fegyverhez, a dash-ek száma, az életerő jelző csík, valamint egy XP csík is, amely vizuálisan mutatja, hogy a játékos milyen közel áll a következő szintlépéshez. Mindez valós időben frissül, hogy a játékos gyorsan reagálhasson a harc közbeni helyzetekre.

Egy kör végeztével a játékos egy összegző képernyőre kerül, ahol visszajelzést kap az adott kör teljesítményéről. Itt látható, hogy mennyi ideig élt túl, hány ellenséget győzött le, és hogy mennyi fejlesztési pontot szerzett az adott körben. Ezeket a pontokat a következő kör előtti fejlesztési szakaszban használhatja fel. A körök közötti időszakban egy fejlesztési felület is található, ahol a játékos három fő komponensét az elsődleges (közelharci) fegyvert, a másodlagos (távolsági) fegyvert és a karakter ruházatát külön fejlesztheti.

Mielőtt új kör kezdődne, a játékos egy külön UI-felületen választhat további nehézségi módosítókat, melyekkel növelheti a kihívást és ezzel együtt a megszerezhető fejlesztési pontok mennyiségét is. Ezek a módosítók lehetnek ellenség-erősítések (nagyobb életerő, több megjelenő ellenség), vagy játékosra ható negatív hatások, például mozgáslassítás. A rendszer szabad kezet ad a játékosnak, hogy saját kockázatvállalása szerint alakítsa a kihívást, miközben a jutalom is ennek megfelelően skálázódik.

## 2.2. Player

2.2.1 Felépítés és mozgásmechanika

A játékos karakter három fő modulból épül fel: egy elsődleges fegyverből, egy másodlagos fegyverből és egy ruhából. A ruha határozza meg a játékos alapvető statisztikáit, mint például a mozgási sebességet, maximális életerőt vagy támadássebességet, és emellett vizuálisan is módosítja a karakter megjelenését. Ez a moduláris struktúra lehetővé teszi, hogy a játékos különböző felszerelési kombinációkat állítson össze, saját játékstílusához igazítva a karakterét. Minden modul külön fejleszthető és cserélhető, így a rendszer nagyfokú rugalmasságot biztosít.

A karakter irányítása top-down nézetből történik, teljes 360 fokos mozgási lehetőséggel. Az alapmozgást egy dash képesség egészíti ki, amely egy gyors lökést biztosít a választott irányba. Ez a mozgásforma kulcsszerepet játszik a harcban: alkalmas az ellenséges támadások elkerülésére, gyors helyváltoztatásra vagy taktikai pozícióváltásra. A mozgásrendszer kialakításánál cél volt, hogy egyszerűen kezelhető, mégis dinamikus és reaktív élményt nyújtson a játékos számára.

### 2.2.2. Élet, sebzés, halál, szintlépés

A játékos életerejét egy különálló életkomponens kezeli, amely felelős az összes életállapot-változásért, mint például sebződés, gyógyulás, maximális élet változása vagy halál. Ez a komponens nemcsak a játékosra, hanem más entitásokra (pl. ellenségek) is újrahasznosítható.

Amikor az életerő megváltozik, a rendszer vizuális visszajelzést biztosít a játékosnak – például egy lebegő szám (float text) jelenik meg a karakter felett, amely mutatja, hogy mennyi sebzést vagy gyógyítást kapott. Ezzel a játékos azonnali információhoz jut anélkül, hogy a HUD-ot kellene figyelnie.

A halál után a játékos állapota visszakerül az alapértelmezett kezdőhelyre, és a kör véget ér. A karakter tapasztalati pontokat (XP) is gyűjt, és szintlépés esetén automatikusan aktiválódnak a szintlépési funkciók – például statnövekedés, új képesség nyitása vagy vizuális effekt. A szintállapotot a rendszer külön kezeli, tárolva az aktuális XP-t, a szintet, és kezelve az új szint eléréséhez szükséges XP küszöböt.

### 2.2.3. Interakciók

A játékos képes különböző objektumokkal interakcióba lépni a körön belül és kívül. A körök közötti zónában a kereskedő jelenik meg, akitől fejlesztések vásárolhatók vagy felszerelések cserélhetők. A körön belül a játékos ideiglenes fejlesztési pontokkal gyógyítás, ideiglenes statbónusz, tud kapcsolatba lépni.

### **2.2.4. Fegyverek**

A játékos két külön fegyvermodullal rendelkezik: egy elsődleges közelharci és egy másodlagos távolsági fegyverrel. Mindkét típus támogatja a fejlesztéseket, amelyek lehetnek egyszerű statisztikai módosítások, mint például sebzésnövelés vagy hatótávbővítés, illetve komplexebb viselkedésváltoztatások is, mint például lánctámadás vagy páncéláttörés. A fegyverek közötti váltás során automatikusan frissülnek a hozzájuk tartozó animációk, találati zónák és statisztikák, így a játékos mindig pontos visszajelzést kap a választott felszerelés hatásairól.

A másodlagos, távolsági fegyverek rendszere ennél is részletesebb. Nemcsak az alap statisztikák, hanem a kilőtt lövedékek viselkedése is testreszabható. Egy fegyver képes lehet több lövedéket egyszerre vagy egymás után kilőni, sorozat vagy szórás formájában. A lövedékek sebessége, sebzése, robbanási képessége és hatóköre mind külön paraméterezhető, ahogyan olyan speciális tulajdonságok is, mint az ellenségek közötti pattogás vagy az extra effektusok alkalmazása.

A fegyverrendszer teljesen moduláris módon épül fel, így lehetővé teszi a játékos számára, hogy szabadon kombinálja és fejlessze fegyvereit, akár egyetlen kör alatt ideiglenesen, akár hosszabb távon a játékmenetek között elérhető tartós fejlődési rendszer révén.

## 2.3. Környezet

A játék kétféle pályaszerkezetet alkalmaz: a körön belüli területek procedurálisan, míg a körök közötti központi zóna kézzel tervezve jön létre.

A körök során a játéktér végtelennek érződik, amit úgy oldottam meg, hogy a pályát kisebb egységekre, úgynevezett "chunk"-okra osztva generálom és töltöm be. Minden chunk egy generált zajtérkép (noise texture) alapján kerül feltöltésre különféle elemekkel – például dekorációkkal, objektumokkal vagy interaktív elemekkel. A zajtérkép adott értékei határozzák meg, hogy hová milyen típusú elem kerüljön, ezáltal biztosítva a változatos, mégis szabályos elrendezést. Ez a rendszer biztosítja, hogy a pálya dinamikusan bővüljön a játékos mozgásával, és sose érjen véget.

Ezzel szemben a körök közötti zóna – a főbázis vagy pihenőterület – teljesen kézzel készült, fix határokkal rendelkező pálya. Ennek célja, hogy minden fontos funkció (mint például a bolt, vagy a következő körbe teleportáló portál) egyértelműen elérhető legyen a játékos számára, anélkül hogy az végtelenül barangolna, elkerülve a tényleges játékot. A kézi tervezés ezen a részen nemcsak funkcionális, de vizuálisan is gazdagabb, részletesebb környezetet biztosít, ami jól elkülönül a procedurálisan generált területektől.

## 2.4. Ellenségek

A játékban az ellenségek kialakításánál a fő szempont a mennyiség, nem a minőség. Egy horde survivor típusú játékban a játékmenet akkor működik jól, ha rengeteg ellenség van egyszerre a képernyőn, így ezeknek az egységeknek rendkívül egyszerűnek és erőforrás-takarékosnak kell lenniük. Az animációk, mozgásminták és viselkedések minimálisra vannak szorítva: a legtöbb ellenség csupán a játékos felé mozog, akadálykerülés vagy komplex logika nélkül. Ez biztosítja, hogy a rendszer képes legyen nagy létszámú ellenfél-lekezelésre, és ne lassuljon le a játékmenet.

A látványvilág is ehhez igazodik: az ellenfelek halálakor egy gyors sprite-váltás vagy egyszerű effekt elég a vizuális visszajelzéshez. Egyes ellenségek távolról támadnak, mások pedig közelharcra specializálódtak, de mindegyikük viszonylag kis számú támadási vagy mozgási variációval rendelkezik. Ezt a minimális ellenség logikát kihasználva a rendszer úgy lett kialakítva, hogy az ellenségek viselkedése egy egységes sablonra épül, ezáltal az új típusok könnyen hozzáadhatók.

### 2.4.1 Elit típusok

A ritkábban megjelenő elit ellenségek az alap típusok megerősített változatai, amelyek különleges statisztikai bónuszokkal és megjelenésbeli eltérésekkel rendelkeznek. Ezek nem teljesen új entitások, hanem az alap ellenségek speciális, buffolt verziói, amelyek kinézetükben is jól elkülönülnek. Például a gyors elit egy kisebb méretű változat élénkebb színnel, ami sokkal sebesebben mozog és nehezebben található el. Az elit ellenségek nem csak mechanikai szinten jelentenek nagyobb kihívást, hanem vizuálisan is azonnal felismerhetők, ez segíti a játékost abban, hogy gyorsan dönteni tudjon, érdemes-e először őket kiiktatni vagy kikerülni.

## 2.5. Fejlesztések

A fejlesztések három különböző típusba sorolhatók: állandó, körspecifikus, és ideiglenes fejlesztések. Ezek mind hasonló módon befolyásolják a játékos statisztikáit és képességeit, de eltérő módon kerülnek megszerzésre és megtartásra.  
  
Az állandó fejlesztések tartós bónuszokat nyújtanak, amelyek megszerzés után minden körre megmaradnak. Ezek fejlődési pontok felhasználásával szerezhetők meg a körök közötti menüben, és a játékos hosszú távú fejlődését szolgálják. Ilyenek lehetnek például az új fegyvertípusok elérhetősége, vagy passzív bónuszok a különböző elemek statisztikáira.  
  
A körspecifikus típusú fejlesztés csak az aktuális kör alatt marad aktív. A játékos ezeket jellemzően szintlépéskor egy választási lehetőségek közül szerezheti meg, és a játékmenet során fokozatosan erősödhet általuk. Bár hatásuk erősebb lehet, mint az állandó fejlesztéseké, a következő kör kezdetével ezek automatikusan elvesznek, ezzel biztosítva a körönkénti újrakezdés élményét.

Az ideiglenes fejlesztések a legrövidebb ideig hatnak, mivel egy meghatározott időtartamig vagy limitált idejű hatásként működnek. Ezeket a pályán szétszórva lehet megtalálni, mint interaktív elemek. Jellemzően erőteljes hatásúak, céljuk, hogy rövid időre kiutat biztosítsanak a kritikus helyzetekből, vagy előnyt adjanak nagyobb ellenséghullámok közepette.

Ez a háromféle fejlesztési mechanika együttesen biztosítja, hogy a játékos mind hosszú távon tervezhessen fejlődést, mind pedig a körök közben és azon belül is reagálhasson a változó helyzetekre.

## 2.6 Mentések

A mentési rendszer biztosítja a játékos előrehaladásának és a felhasználói beállításoknak a megőrzését. A mentés két fő kategóriára oszlik: játékosadatok és beállítások.

A játékoshoz kapcsolódó adatok, például az állandó fejlesztések, feloldott felszerelések és alap statisztikák kezelésére a Godot által preferált Resource típusú fájl kerül alkalmazásra. Ez lehetővé teszi az adatok közvetlen szerializálását és visszatöltését a motoron belül, egyszerű és natív módon.

A beállítások mint például a hangerő, grafikai mód vagy vezérlési sémák – mentésére a Godot ConfigFile struktúráját használja a rendszer. Ez egy kulcs-érték alapú fájlformátum, amely ideális konfigurációs adatok tárolására.

Ezzel a megközelítéssel a játék képes stabilan és megbízhatóan képes tárolni minden szükséges adatot a játékos élményének folytonossága érdekében.

# 3. TERVEZÉS

## 3.1. Játékos

### 3.1.1. Moduláris karakterfelépítés

A játékos karakter moduláris komponensekből épül fel, amely lehetővé teszi a könnyű bővíthetőséget és karbantarthatóságot. Maga a Player node nem nagyon tartalmaz játékmeneti logikát a mozgáson és a dash újratöltésén kívül, csupán egy vezérlő platformként szolgál, amely koordinálja a hozzá tartozó komponensek működését.

A játékos olyan alapvető elemeket kezel, mint az aktuális tapasztalati pont (XP), szintlépés, dash-mennyiség, aktuális lőszerkészlet és a különböző fegyverek vagy ruhák aktív állapota. Ezek váltása egy, a játékosban definiált váltó metóduson keresztül történik, amelyet egy signal hív meg az adott index értékkel. A signal-t fogadó függvény a megfelelő vagy ?Melee/Ranged/OutfitComponent-et értesíti, amelyek beolvassák és alkalmazzák az új felszerelési adatokat. A jelenlegi felszerelés adatait egy dedikált Resource típusú változó tárolja.

### 3.1.2. Statok összevonása komponensekből

A játékos statjai több forrásból tevődnek össze. Az alapvető értékeket, mint a kezdő életerő, pajzs mennyisége, dash-ek száma, regenerációs sebességeket a ruha komponens határozza meg. A különböző ruhák eltérő statisztikai profilokat biztosítanak, például egy kevesebb életerővel induló ruha nagyobb elsődleges fegyver fejlesztéseket kínál.

Ezek a statok automatikusan összeadódnak a játékos aktuális állapotában, amit a különböző komponensek biztosítanak. Minden komponens felelős a saját statisztikái érvényesítéséért és módosításáért.

### 3.1.3. Életkomponens és sebzéskezelés

A játékos sebzést és regenerációt kezelő logikáját egy önálló HealthComponent végzi. Ez a komponens felel a jelenlegi életerő és pajzs értékek nyilvántartásáért, a regenerációs folyamatokért, valamint a sebzés feldolgozásáért. A játékos HitBox node-ja figyeli az ütközéseket és sebzéseket. Amennyiben találat éri a karaktert, a HitBox jelet küld a HealthComponent számára, amely kiszámítja a kapott sebzést a jel paraméterei alapján.

### 3.1.4. Vizuális megjelenítés komponensenként

A játékos vizuális reprezentációja teljesen a felszereléséhez igazodik. A ruhák a karakter teljes kinézetét meghatározzák, amelyekhez különböző paletták tartoznak a megkülönböztetés érdekében. Az elsődleges és másodlagos fegyverek szintén saját megjelenést használnak, ideértve az egyedi sprite-okat és animációkat.

A közelharci fegyverek esetén egyedi „slash trail” effekt jelenik meg, amely szintén a fegyver típusától függ. A távolsági fegyvereknél a lövedékek kinézete fegyvertípusonként eltérő, ezzel is vizuális visszajelzést adva a játékosnak az aktív felszerelésről.

### 3.1.5. Állapotgép (State Machine) és animáció kezelés

A játékos viselkedése egy StateMachine komponens segítségével van vezérelve, amely egyértelműen szétválasztja a különböző működési állapotokat: Idle, Attack, Combo, Dash, Shoot, és Death. Ez a rendszer biztosítja, hogy adott időpontban kizárólag az aktuális állapothoz tartozó műveletek hajtódjanak végre, így elkerülhetőek az állapotütközések.

Állapotváltások logikája

Az állapotok közötti váltás eseményvezérelt módon történik, jellemzően a bemenetek (input), időzítések és játékon belüli események alapján. Példák az átmenetekre:

Ha a játékos dash-t indít elindul a transition jel, ekkor az aktuális állapottól függően (Idle, Move, vagy Attack) Dash állapotba lép.

Ha a játékos támadást indít ugyanúgy küldünk egy transition jelet, és az Attack állapotba váltok, majd ha egy meghatározott időn belül újabb támadási bemenet érkezik, akkor Combo állapotba kerül.

Ez a szerkezet könnyen bővíthető: új állapotok (pl. blokkolás, gyógyítás, vagy speciális mozdulatok) később is hozzáadhatók, a meglévő rendszer módosítása nélkül.

Animációkezelés – AnimationTree integráció

A különböző állapotokhoz tartozó animációkat egy AnimationTree node kezeli, amely a StateMachine állapotaihoz szinkronizálva működik. Minden főállapothoz külön animáció tartozik. Az állapotváltások során az StateMachine jelenlegi allápotát amit figyel az AnimationTree erre reagálva a fa átugrik a következő állapotra tükrözve a StateMachine-t, így zökkenőmentesen és automatikusan történik az animációváltás.

### 3.1.7. Mozgásmechanika

A játékos mozgását alapvetően a Player szkript kezeli, amely folyamatosan olvassa a bemeneti jeleket, és ezek alapján módosítja a karakter sebességét, irányát és animációs állapotát. A mozgás különleges formája a dash, amely nem a mozgás része, hanem egy különálló állapotként (state) van kezelve a StateMachine komponens segítségével. Amikor a játékos dash-t indít, a karakter automatikusan átlép a Dash állapotba, ahol egy hirtelen sebességlöketet kap az aktuális mozgási irányban. A dash időtartama alatt a normál mozgás le van tiltva, ezután visszatér a Idle állapotba a bemenetektől függően.

A dash állapot egyik különleges tulajdonsága, hogy tartalmaz egy támadás-buffer rendszert, amely lehetővé teszi, hogy a játékos a dash vége felé beérkező támadási inputokat ne veszítse el. Ha a játékos a dash vége előtt megnyomja a támadás gombot, az input elraktározódik egy belső pufferben, és a dash befejeződése után azonnal végrehajtódik. Ez a megoldás simább és pontosabb játékélményt biztosít, különösen gyors tempójú harci szituációkban.

A dash egyben ideiglenes sebzésimmunitást is biztosít a játékos számára. Ez technikailag úgy valósul meg, hogy a játékos Hitbox komponense a dash ideje alatt deaktiválásra kerül, így a játékos nem kap sebzést.

A láthatóság és egyértelmű azonosítása érdekében a karakter egy particle effect kíséretében hajtja végre a mozdulatot. A particle effektet külön erőforrás (Particle2D vagy GPUParticles2D) kezeli, amely a dash aktiválásakor automatikusan elindul, majd leáll a mozdulat végén.

### 3.1.6. Fegyverkomponens rendszer

A játékban két fő fegyvertípus szerepel: közelharci és távolsági. Mindkét fegyvertípus saját, dedikált komponenssel rendelkezik, amely felelős az adott típus működéséért és adatainak kezeléséért. A fegyverek adatai Resource típusú fájlokban vannak tárolva, amelyeket a fegyverkomponensek betöltenek, és ezek alapján inicializálják a fegyver viselkedését.

A fegyverváltás a játékos által kiváltott signal alapján történik. A signal tartalmazza a kívánt fegyver indexét, amelyet a játékos feldolgoz, és ennek megfelelően frissíti az aktív fegyvereket. A fegyverkomponensek önállóan végzik el az adatbetöltést és inicializálást a megadott Resource alapján.

## 3.2. Ellenségek

### 3.3.1. Alapfelépítés és viselkedés

A játék ellenségei egy egységes sablonból, a BaseEnemy származtatott jeleneteiből épülnek fel. Ezek az egységek a játékoshoz hasonlóan moduláris komponensekre épülnek: tartalmaznak életkezelő komponenst (HealthComponent), állapotgépet (StateMachine), valamint animációvezérlést (AnimationTree), amely lehetővé teszi az állapotokhoz igazodó animációváltásokat.

A legtöbb ellenség egyszerű mesterséges intelligenciával rendelkezik: alapértelmezetten a játékos pozíciója felé mozognak, és ha elérik az ütéstartományt, támadást kezdeményeznek. Egyes típusok, mint például a távolsági ellenségek, egyedi viselkedést mutatnak – sebzés hatására hátrálnak vagy elmenekülnek, mielőtt újrapozicionálnák magukat.

Az ellenségek száma nagy, ezért optimalizálási célból Godot VisibleOnScreenNotifier2D komponensét használom. Ez jelzi, ha egy ellenség kikerül a kamera látóteréből, ebben az esetben az entitás legtöbb komponense (pl. animáció, AI, fizika) deaktiválásra kerül. Csak a mozgási logikája marad aktív, hogy szükség esetén visszatérhessen a képernyőre.

### 3.3.2. Elit ellenségek és stratégia minta

A játékban különleges "elit" ellenségek is megjelenhetnek. Ezek ugyanazokat az alapsablonokat használják, de megjelenésükkor egy ElitStratégia alkalmazásával módosulnak. Az ElitStratégia a Stratégia tervezési minta elvein alapul, és képes bármely ellenségen végrehajtani statisztikai és vizuális módosításokat: például megnövekedett mozgási sebesség, életerő, módosított támadási minták, egyedi színezés vagy effektek. Ez a megoldás lehetővé teszi, hogy bármely alaptípusból egyszerűen létrehozható legyen egy elit változat, anélkül, hogy új jelenetet kellene létrehozni.

### 3.3.3. Ellenség spawner rendszer

Az ellenségek megjelenítését egy központi spawner rendszer vezérli, amely az aktuális hullámokhoz tartozó WaveResource adatait használja. Ezek a resource-ok tartalmazzák a spawnálandó ellenségtípusokat, azok darabszámát, valamint az elit ellenségek előfordulási esélyét.

A spawner véletlenszerűen, a képernyőn kívüli pozíciókra helyezi el az új ellenségeket, hogy fokozatosan közelítsenek a játékos felé. Ha az adott példány elitként kerül kijelölésre, a spawner azonnal alkalmazza rá az ElitStratégiát.

A teljesítmény fenntartása érdekében a rendszer tartalmaz egy maximum aktív ellenség korlátot is, amely megakadályozza, hogy túl sok ellenség legyen egyszerre jelen. Ez különösen fontos a horde-survivor típusú játékmenet során, ahol százával jelenhetnek meg ellenségek.

## 3.4. Környezet

### 3.4.1. Körön belüli pályagenerálás

A körön belüli pálya egy teljesen procedurálisan generált, végtelennek tűnő tér, ahol a játékos a túlélésre és fejlődésre koncentrál. A pálya generálása több lépésből áll, ezek modularitásuk miatt könnyen bővíthetők vagy módosíthatók a jövőben.

A különböző pályaelemek elhelyezésének alapját egy Simplex Smooth-szerű zajgenerátor képezi, amely egy zajképet hoz létre. Ez a zajkép egy adott területen belül ad vissza értékeket, amelyeket típusintervallumokra bontva értelmezünk. Minden típushoz (pl. fű, fa, kő, interaktív tárgy, díszlet) egy Vector2(min, max) értéktartomány tartozik, amely meghatározza, hogy adott zajérték esetén milyen típusú elem kerül elhelyezésre.

A teljes pályát nem egyszerre generáljuk le, hanem egy „chunk” alapú rendszer segítségével, amely a térképet kisebb, 2D rácsra osztott szegmensekre bontja. A rendszer figyeli a játékos aktuális pozícióját, és csak azokat a chunkokat generálja és tartja aktívan memóriában, amelyek a játékos környezetében helyezkednek el. Ez jelentősen csökkenti a teljesítményigényt, és lehetővé teszi egy kvázi-végtelen pályaérzet kialakítását.

A zajgenerálás és chunk betöltés után kerülnek elhelyezésre a dekoratív és interaktív objektumok. Ezek különféle tilemap rétegekre, illetve instanciázott objektumként jelennek meg. A dekorációk kizárólag vizuális célt szolgálnak, míg az interaktív elemek például gyógyító pontok, ideiglenes fejlesztéseket adó tárgyak funkcionális hatással bírnak, és egyedi outline shaderrel, valamint lebegő szöveggel vannak megkülönböztetve a környezetüktől. Az interakciós lehetőségeket külön interakció komponens kezeli, amely a játékos közelségét és bemeneteit figyeli.

### 3.4.2. Körök közötti központi tér

A központi tér kézzel készült, nem procedurálisan generált pályarész. A pálya kialakítása során figyelembe kell vanni, hogy a dekoratív elemek ne csak díszítésként szolgáljanak, hanem segítsenek irányítani a játékos figyelmét. Például az utak és vizuálisan kiemelkedő tárgyak arra vezetik a játékost, hogy hamar észrevegye a fontos helyszíneket, mint a bolt vagy a teleport.

### **3.5. Fejlesztések**

A játék fejlesztési rendszere moduláris felépítésű, így könnyen bővíthető és karbantartható. A rendszer alapvető feladata, hogy a játékos és a fegyverek statisztikáit amik külön Resource fájlokban vannak tárolva, külön-külön módosíthatók legyenek a megfelelő fejlesztések alkalmazásával. A statok módosítása a **stratégia tervezési minta** (Strategy Pattern) segítségével történik.

### 3.5.1. Fejlesztéstípusok kezelése

A fejlesztések három fő típusra oszthatók:

* **Állandó fejlesztések**: Ezek hosszú távon módosítják a játékos vagy a fegyverek tulajdonságait. Ezeket a kör közötti bázisban lehet az NPC használatával szerezni.
* **Körön belüli fejlesztések**: Ezeket a játékos a kör során szintlépéskor, szerezheti meg, és csak az adott kör végéig maradnak érvényben. De az implementálása megegyezik az állandó fejlesztésekkel csak nem kerülnek mentésre.
* **Időhöz kötött, ideiglenes fejlesztések**: Ezek egy meghatározott időtartamig aktívak, növelik az adott stat értékét, majd automatikusan a megadott idő elteltével visszaállítják az eredeti értékeket.

### 3.5.2. Fejlesztés alkalmazása

A fejlesztési logika központi eleme a BaseStatModifierStrategy, amely egyfajta absztrakt osztályként szolgál minden fejlesztési stratégia számára. Ez az interfész biztosítja, hogy az összes stat módosító stratégia egységesen legyen kezelve.

Az állandó stratégiák a játékos statisztikáit vagy fegyveradatokat közvetlenül módosítják, és a változások tartósak maradnak a kör végéig, vagy akár a játékos haláláig.

Az ideiglenes stratégiák öröklik az állandó stratégiák működését, de tartalmaznak egy időzítőt is. Amikor ez az időzítő lejár, a stratégia automatikusan visszavonja a stat módosításokat, így visszaállítva az eredeti állapotot.

Ezen kívül léteznek az effekt stratégiák, amelyek nem statokat módosítanak, hanem valamilyen játékmeneti hatást adnak hozzá a célponthoz. Ezek például égés vagy mozgás limitálás, formájában jelenhetnek meg. Az effekt stratégiák a hozzájuk tartozó komponenseket a célpont (például ellenség) node-jához adják hozzá. A komponensek jellemzően csak vizuális elemeket (pl. Particles2D, AnimatedSprite2D) tartalmaznak, míg a komponens gyökéreleme végzi a hatás logikájának kezelését (pl. időzítés, sebzés).

## 3.6. Mentési rendszer

### 3.6.1. Beállítások mentése

A játék technikai beállításait például, felbontást, vezérlési módokat egy ConfigFile típusú fájlba mentem. A ConfigFile formátum (általában .cfg) könnyen olvasható és módosítható, így a felhasználók vagy fejlesztők kézzel is be tudnak avatkozni, ha szükséges. Emellett jól strukturált, kulcs-érték párokat használó formátum, amely ideálissá teszi a ritkán változó, de a futtatás elején szükséges beállítások mentésére.

Az ilyen típusú adatokat nem szükséges futásidőben gyakran módosítani, és nincs szükségük komplex belső logikára, így nem indokolt a Resource alapú mentési forma.

### 3.6.2. Játékadatok mentése Resource-ba

A játékos adatait – például megszerzett fejlesztéseket, permanens pontokat, kinyitott fegyvereket – egyedi Resource típusú állományok tárolják. Ezek a fájlok közvetlenül tartalmazzák az adott adatokhoz kapcsolódó struktúrákat és típusokat, így például az UpgradeResource, PlayerStats, WeaponStats típusú adatokat. A Resource fájlok előnye, hogy közvetlenül integrálhatók a Godot játékmotor adatmodelljébe, így könnyedén példányosíthatók, szerkeszthetők az editorból, vagy futásidőben módosíthatók és menthetők.

### 3.6.3. Futásidejű adatok: RuntimeSaves globális script

A körök közötti, nem véglegesen mentett adatokat egy globális RuntimeSaves nevű script kezeli. Ez a script egy singletonként (autoloader) működik, és a játék teljes ideje alatt elérhető. Olyan adatokat tárol, amelyeknek csak ideiglenesen kell fennmaradniuk – például hogy a játékos milyen nehézségeket alkalmaz a kör során, vagy hogy melyik fegyver volt nála legutóbb.

Ez lehetővé teszi az állapot megőrzését két játékszakasz (pl. kör és központi tér) között anélkül, hogy ezeket az adatokat véglegesen el kellene menteni a fájlrendszerbe.

# 4. MEGVALÓSÍTÁS

# ÖSSZEGZÉS

# IRODALOMJEGYZÉK

# KÖSZÖNETNYÍLVÁNÍTÁS

# MELLÉKLETEK