**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Komputeralgebra** **Tanszék**

2D puzzle platformer játék

Szerző: Témavezető:

Zsoldos Dániel Magyar Péter

Programtervező informatikus BSc. Egyetemi Tanársegéd

**Szombathely, 2025**

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT TÉMABEJELENTŐ

**Hallgató adatai:**

**Név:** Zsoldos Dániel

**Neptun kód:** X62Z3Y

**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:** 2D puzzle platformer játék

**A szakdolgozat témája:**

Egy oldalnézetes platformer puzzle játék ami az űrben játszódik és egy MARVIN nevű javítórobottal kell átjutni 10+ darab pályán. A cél: elérni a hajó hídját, hogy az űrhajó folytathassa az útját egy új bolygó felé.

A játék elkészítéséhez a Godot játékmotor lesz felhasználva, a grafika pixel art stílusú lesz és valós időben fog futni.

A pályák különböző mechanikai és logikai kihívásokat rejtenek, amelyeket eltérő gravitációs viszonyok között kell megoldani. Amikor erősebb a gravitáció, akkor kisebbet ugrik a robot , amikor gyengébb, akkor az ugrásai nagyobbak. A kihívásokat nehezítik a pályákon található veszélyforrások, mint például tüskék, beragadt ajtók és zárlatos energiafalak amelyek a meghibásodott védelmi rendszer részét képezik. A logikai feladatok közé fog tartozni például, hogy karok és/vagy gombok segítségével jeleket küldünk egy kapcsolótáblának, amelyben egymáshoz kötött logikai kapuk találhatók olyan céllal, hogy egy 1-es jel jusson el a következő pályához vezető zárt ajtóhoz.

A javítórobot több fajta eszközzel is rendelkezik, mint például egy „jump pack”, amivel tud egyszer ugrani a levegőben és egy teleportációs készülékkel, amellyel előre tud ugrani egy kis távolságot, bármilyen szilárd akadályon áthaladva. Ezeket az eszközöket újra használhatja, miután földet ér, vagy a pályán elhelyezett frissítő tárgyat szed fel. Továbbá egy csavarhúzót is tart magánál, amellyel a kapcsolótáblák burkolatát le tudja szedni és meg tudja nézni, hogy milyen logikai kapuk találhatók benne, valamint azt is, hogy azok hogyan vannak bekötve. A csavarhúzóval meg tudja bökni a zárlatos energiafalakat, hogy a jump pack-be gyűjtsön még egy ugrásra elegendő energiát. A csavarhúzót mindig magánál tartja, de a jump pack-et és a teleportációs készüléket adott pályák teljesítésével fogja tudni megszerezni.

Budapest, 2024. 10. 14.

Tartalomjegyzék

[1. Bevezetés 1](#_Toc196005809)

[1.1. A Godot játékmotorról 1](#_Toc196005810)

[1.2. A platformer műfajról 1](#_Toc196005811)

[2. Felhasználói dokumentáció 2](#_Toc196005812)

[2.1. A játék célja 2](#_Toc196005813)

[2.1.1. Célközönsége 2](#_Toc196005814)

[2.2. Rendszerkövetelmények 2](#_Toc196005815)

[2.3. Játék indítása 3](#_Toc196005816)

[2.4. Irányítás 3](#_Toc196005817)

[2.5. Funkciók ismertetése 4](#_Toc196005818)

[2.5.1. Játékfelépítés 4](#_Toc196005819)

[2.5.2. Karakter képességei 4](#_Toc196005820)

[2.5.3. Szinteken megtalálható elemek 5](#_Toc196005821)

[2.5.4. Kapcsolótábla és áramkörök működése 9](#_Toc196005822)

[2.5.5. Felhasználói felületek 13](#_Toc196005823)

[3. Fejlesztői dokumentáció 16](#_Toc196005824)

[3.1. Megoldási terv 16](#_Toc196005825)

[3.1.1. SwitchBox terv 17](#_Toc196005826)

[3.2. A játékprogram 18](#_Toc196005827)

[3.2.1. World 18](#_Toc196005828)

[3.2.2. Player 22](#_Toc196005829)

[3.2.3. A szintek alapkialakítása 26](#_Toc196005830)

[3.2.4. HitBox HurtBox 27](#_Toc196005831)

[3.2.5. InteractionArea és InteractorArea 27](#_Toc196005832)

[3.2.6. ScrewDriverInteractor és InteractionArea 27](#_Toc196005833)

[3.2.7. screw\_driver.tscn és screw\_driver\_script.gd 28](#_Toc196005834)

[3.2.8. PWR osztály, globalPowerState.gd 28](#_Toc196005835)

[3.2.9. End Of Level Door 29](#_Toc196005836)

[3.2.10. Felvehető tárgyak osztályai: 30](#_Toc196005837)

[3.2.11. Spikes 31](#_Toc196005838)

[3.2.12. Electric Fence: 31](#_Toc196005839)

[3.2.13. GravityArea és NoTeleportArea 31](#_Toc196005840)

[3.2.14. Door-kapu 32](#_Toc196005841)

[3.2.15. PowerLine 32](#_Toc196005842)

[3.2.16. PowerGrid 32](#_Toc196005843)

[3.2.17. Lever – kar 33](#_Toc196005844)

[3.2.18. SwitchBox – kapcsolótábla 34](#_Toc196005845)

[3.3. Tesztelés 36](#_Toc196005846)

[4. Összefoglalás és további fejlesztési lehetőségek 38](#_Toc196005847)

[5. Irodalomjegyzék 39](#_Toc196005848)

# Bevezetés

A szakdolgozatom fő célja az, hogy egy 2D-s platformer játék megvalósításával jobban megismerhessem a játékműfajt és a Godot játékmotor működését és lehetőségeit. A 2D-s grafika választása tudatos döntés volt, mivel ez a stílus lehetőséget adott arra, hogy a szükséges grafikai elemeket saját kezűleg elkészítsem. A platformer műfajt azért választottam, mert számos hasonló stílusú játékkal játszottam korábban, így már rendelkezem tapasztalatokkal arról, mi lehet élvezetes vagy kihívást jelentő a játékos számára.

## A Godot játékmotorról

A Godot egy nyílt forráskódú, ingyenesen használható játékmotor, amely támogatja a 2D-s és 3D-s játékfejlesztést is. A motor könnyen kezelhető, platformfüggetlen fejlesztői környezettel rendelkezik, és saját fejlesztésű, egyszerűen tanulható programozási nyelvet (GDScript) használ. Emellett támogatja a C# és C++ nyelvek használatát is.

## A platformer műfajról

A platformer műfaj első képviselőjeként gyakran az 1981-ben a Nintendo által kiadott Donkey Kong játékot említik. Ebben a játékban a játékos a később világhírűvé vált Mario karaktert irányítja – aki akkor még a Jump Man néven szerepelt. A cél az volt, hogy egy építkezési területen létrákon és platformokon keresztül haladva, hordókat átugorva elérje a gorillát, Donkey Kongot, és megmentse a fogságba esett Pauline-t.

A poster of a video game

AI-generated content may be incorrect.

. ábra: Donkey Kong szórólap

# Felhasználói dokumentáció

## A játék célja

A játék célja az, hogy MARVIN a játékos által irányított javítórobot elérjen az űrhajú hídjára és aktiváljon egy vezérlőpanelt, amellyel kijavítja a hajó pályáját az új bolygó felé. Az út során a játékosnak különféle akadályokat kell leküzdenie, melyek próbára teszik ügyességét és alapvető problémamegoldó képességeit. A kihívások logikai és mechanikai feladatokat tartalmaznak, amelyek megoldása szükséges a továbbhaladáshoz.

### Célközönsége

A játék minden korosztály számára ajánlott, különösen azoknak, akiket érdekelnek az oldalnézetes platformer vagy puzzle-alapú játékok. A játék tartalmaz ügyességi és logikai elemeket is, így ideális választás lehet mindazok számára, akik szívesen tesztelik vagy fejlesztik problémamegoldó képességüket.

A játék külön értéke, hogy bevezeti a játékost az alapvető logikai műveletek és logikai kapuk világába, melyek a matematikai és informatikai logika alapját képezik. Így a játék nemcsak szórakoztató, hanem oktatási szempontból is hasznos lehet – különösen fiatalok, diákok számára.

## Rendszerkövetelmények

Minimum:

Operációs rendszer: Windows 7+

CPU: Intel Core 2 Duo E8200, AMD Athlon XE BE-2300

GPU: Intel HD Graphics 5500, AMD Radeon R5 Graphics

RAM: 2GB

Tárhely: 150MB

Optimális:

Operációs rendszer: Windows 10+

CPU: Intel Core i5-6600K, AMD Ryzen 5 1600

GPU: NVIDIA GeForce GTX 1050, AMD Radeon RX 460

RAM: 4GB

Tárhely: 200MB

## Játék indítása

Az indítást a játékmappában elhelyezett marvin\_in\_space.exe megnyitásával tehetjük meg. Telepítésre nincsen szükség, mivel a játék hordozható állapotban van exportálva. Ilyenkor a főmenübe érkezve a [New Game] gombra kattintva elindítható a játék.

## Irányítás

|  |  |
| --- | --- |
| **Billentyű:** | **Funkció:** |
| A, Jobbra nyíl | Jobbra mozgás |
| D, Balra nyíl | Balra mozgás |
| W, Felfelé nyíl, Szóköz | Ugrás |
| Shift | Teleportálás a karakter haladási irányába |
| E, Bal klikk | Interakció (Kar, Kapcsolótábla, Ajtó) |
| M, Tabulátor | Az adott szinten a karakter- és a szintkamera közötti váltás. |
| S, Lefele nyíl | A szinteken áthelyezett platformokon lefelé való átesésre szolgál. |
| Escape | Menü megnyitása, szünet |
| X | A szint újraindítása |

. táblázat: Irányítás

## Funkciók ismertetése

### Játékfelépítés

A játék 12 szintből áll melyekből az első 3 bemutatja a játékost a főbb mechanikáknak (mozgás, ugrás, teleportálás). Ezt követő 3 szint ügyességi, melyek után a logikai pályák következnek. A 12. szint után fog MARVIN elérni a hídra ami a játék vége. A játék felépítése lineáris, a pályák egymás után következnek, visszafelé haladás nem lehetséges.

### Karakter képességei

Ugrás, jump pack: A karakter adott sebességgel ugrik felfelé. Alapállapotban csak egy ugrásra képes, kizárólag a földről indulva. Amennyiben rendelkezik a jump pack-kel, lehetősége nyílik egy második ugrásra is, így összesen egymást követően kettőt hajthat végre, akár levegőből indulva. Az eszköz aktuális állapotát egy ikon jelzi a képernyőn. A jump pack akkor töltődik újra, ha a karakter ismét talajt ér, vagy ha a pályán elhelyezett frissítő tárgyat vesz fel.

Teleportációs készülék: A teleportációs készülék segítségével a karakter egy meghatározott távolsággal előrébb teleportálható, úgy, hogy szilárd falba nem érkezhet. A játék kezdetén ez az eszköz még nem áll rendelkezésre, de a játékos rövid időn belül hozzáférést nyer. Fontos odafigyelni a teleportáció használatára, mivel az eszköz lehetővé teszi, hogy a karakter kiugorjon a hajó vékony külső falán. A teleportáció elérhetőségét egy ikon jelzi a képernyőn.

Csavarhúzó: Ezzel képes leszedni kapcsoló táblák burkolatát, ilyen módon meg lehet nézni hogy milyen logikai kapuk vannak benne. Továbbá ha már rendelkezik a karakter jump pack-el akkor egy elektromos kerítés megérintésekor kap egy ugrást, de az egy időben tartott ugrások száma nem mehet 2 fölé.

A pixelated cartoon of a robot

AI-generated content may be incorrect.

. ábra: MARVIN kezében a csavarhúzóval,

feje fölött a Jump Pack és a Teleport ikon

### Szinteken megtalálható elemek

* Tüskék: Amennyiben a karakter megérinti a tüskét, az aktuális szint újraindul.



. ábra: Tüskék

* Elektromos kerítés (receiver): Le és föl lehet kapcsolni karok/kapcsolótáblák által. Amennyiben a karakter csavarhúzója érintkezik vele a karakter ugrást kap, és ha a karakter maga érintkezik vele akkor a szint újraindul.

A red and white line

AI-generated content may be incorrect.

. ábra: Elektromos kerítés

* Platform: Normál állapotban a karakter át tud rajta ugrani alulról és állhat rajta. Az S billentyű vagy a lefele nyíl megnyomásával a karakter képes átesni rajta.

A black and white image of a bridge

AI-generated content may be incorrect.

. ábra: Platform

* Gravitációs zóna: Ez egy téglalap alakú zóna a pályán melyen belül melyen belül a karakterre nagyobb gravitációs erő hat, gyorsabban esik és kisebbet tud ugrani. A gravitáció mértéke színkóddal van jelölve: minél magasabb, annál pirosabb, míg alacsonyabb érték esetén a zóna kékebb árnyalatú. Emellett egy Xg formátumú felirat is jelzi a gravitációs értéket.

A grey rectangular object with red text

AI-generated content may be incorrect.

. ábra: Gravitációs zóna

* Teleportálást blokkoló zóna: Ez egy sárga áttetsző téglalap alakú zóna, melyben a karakter nem teleportálhat akár milyen irányba. Ilyenkor nem jelenik meg a teleportációs készülék ikonja.

A green square with white text

AI-generated content may be incorrect.

. ábra: Teleportálást blokkoló zóna

* Jump Pack: Felvétele után, lehetővé teszi a karakter számára, hogy duplát ugorjon – vagyis a levegőben is végre tud hajtani egy második ugrást. A levegőben 2 ugrás elvégzése után akkor használható újra, ha a karakter földet ér vagy Jump Pack frissítőt vesz fel

A pixelated video game of a pair of blue objects

AI-generated content may be incorrect.

. ábra: Jump Pack

* Teleportációs készülék: Miután a karakter felveszi ezt az eszközt, lehetővé válik számára, hogy egy kis távolságot előre teleportáljon, még akkor is, ha szilárd akadály (például fal) van előtte. A készülék használata korlátozott, és csak akkor válik ismét elérhetővé, ha a karakter földet ér.

A pixelated blue and black diamond

AI-generated content may be incorrect.

. ábra: Teleportációs készülék

* Jump Pack frissitő: Miután a karakter felveszi ezt az elemet, visszatölti a Jump Pack kapacitását, vagyis ismét legfeljebb két ugrás végrehajtására lesz képes. A frissítő akkor is hasznos, ha a karakter levegőben van és már elhasználta az ugrásait.

A pixelated image of a cross

AI-generated content may be incorrect.

. ábra: Jump Pack frissitő

* Vezeték: 2 eszközt összekötő vonal, ami piros ha nem hordoz információt és zöld ha igen. Legtöbbször egy jeladótól egy vevő - elektromos kerítés, kapu, ajtó - vagy kapcsolótábla felé viszi az információt. Kapcsolótáblából tovább viheti a jelet egy másik, vagy ugyan abba a kapcsolótáblába, vagy fogyasztóba.

A cartoon of a person holding a vent

AI-generated content may be incorrect.

. ábra: Vezeték egy bekapcsolt kartól egy kapcsolótábláig

* Kar (Jeladó): A karakter képes meghúzni a kart, amelynek hatására megváltozik a kar és az összekapcsolt eszközök energiaállapota (pl. bekapcsolnak vagy kikapcsolnak). Ez a változás gyakran szükséges a továbbjutáshoz vagy a szint logikai feladványainak megoldásához.

A pixel art of a lamp

AI-generated content may be incorrect.A pixel art of a microphone

AI-generated content may be incorrect.

. ábra: Be és kikapcsolt karok

* Ajtó (Vevő): Minden szint elején található egy ajtó, amely nem interaktív, mivel a pályák között nem lehet visszafelé haladni. A szint végén lévő ajtó viszont zárt vagy nyitott állapotban lehet. Amennyiben zárt, annak kinyitása a szint logikai feladványának része, például kapcsolók vagy logikai kapuk megfelelő aktiválásával válik átjárhatóvá.

A black rectangle with blue handles

AI-generated content may be incorrect.A grey and black rectangles with a blue stripe

AI-generated content may be incorrect.

. ábra: Nyitott és zárt ajtó

* Kapu (Vevő): Kapukkal három állapotukban találkozhatunk a pályákon. Beragadva, amikor a kapu semmilyen esetben sem fog kinyílni, így egy alternatív megoldást kell találni az áthaladásra. Zárva, amikor a kapu kinyitása a szint logikai feladványának része, amely megoldásával nyitott állapotba kerüljön. Nyitott állapotban akadály nélkül át lehet haladni rajta.

A red white and blue line

AI-generated content may be incorrect.A grey rectangular object with black lines and red and green stripes

AI-generated content may be incorrect.

. ábra: Beragadt, zárt és nyitott kapu

* Kapcsolótábla (Jeladó): Pályákon csak zárt állapotban találhatóak meg, ki lehet őket nyitni MARVIN csavarhúzójával. A felnyitást követően a játékos megtekintheti a belső logikai felépítést, azaz milyen logikai kapuk találhatók benne, és hogyan vannak összekötve. A bemenő vezetékek sorrendje megegyezik a belső felépítésben láthatóval. Amikor a bemeneti vezetékeken megváltozik az energiaállapot, a logikai rendszer újraértékeli az állapotot, és a kimeneti vezetékeken keresztül továbbítja az eredményt.

A pixel art of a vent

AI-generated content may be incorrect. A pixelated image of a person

AI-generated content may be incorrect.

. ábra: Zárt és nyitott kapcsolótábla

* Főkonzol: A játékos végső célja, hogy MARVIN-nal elérje az űrhajó hídján található vezérlőkonzolt. Ezzel interakcióba lépve MARVIN véglegesen kijavítja az űrhajó pályáját, lehetővé téve, hogy a hajó folytassa útját az új bolygó felé. A konzollal történő interakció a játék befejezését jelenti.

A video game screen with a video game controller

AI-generated content may be incorrect.

. ábra: Főkonzol a hajó hídján

### Kapcsolótábla és áramkörök működése

Áramkörök fő részei:

* Vezetékek: Ezek viszik át az energiaállapotot egy Jeladó felől egy Vevő felé.
* Jeladók: Ezek adják ki az energiaállapotokat a vezetékeken keresztül.
* Vevők: Ezek csak fogadják az energiaállapotot és használják, pl. kinyílik az ajtó/kapu; bekapcsol egy elektromos kerítés.

Tehát egy egyszerű áramkörnek tekinthetjük azt, hogy egy jeladó egy vevőnek ad jelet, hogy be vagy ki legyen kapcsolva például egy kar ami meghúzásával bekapcsol egy elektromos kerítést (16. és 17. ábra).

A pixelated image of a stick

AI-generated content may be incorrect.

. ábra: Kikapcsolt egyszerű áramkör

A video game screen with a red and white smokestack

AI-generated content may be incorrect.

. ábra: Bekapcsolt egyszerű áramkör

Kapcsolótáblákban megtalálhatóak logikai kapuk - a nem (NOT) kapu kivételével mindegyik 2 bemenetes - amelyek logikai műveleteket végeznek el az energiaállapottal, tekintsük a bekapcsolt állapotot 1-nek a kikapcsolt állapotot 0-nak, ekkor:

* Vagy (OR): 0 OR 0 -> 0; 0 OR 1 -> 1; 1 OR 0 -> 1; 1 OR 1 -> 1
* Kizáró vagy (XOR): 0 XOR 0 -> 0; 0 XOR 1 -> 1; 1 XOR 0 -> 1; 1 XOR 1 -> 0
* Nem vagy (NOR): 0 NOR 0 -> 1; 0 NOR 1 -> 0; 1 NOR 0 -> 0; 1 NOR 1 -> 0
* És (AND): 0 AND 0 -> 0; 0 AND 1 -> 0; 1 AND 0 -> 0; 1 AND 1 -> 1
* Nem és (NAND): 0 NAND 0 -> 1; 0 NAND 1 -> 1; 1 NAND 0 -> 1; 1 NAND 1 -> 0
* Nem (NOT): NOT 0 -> 1; NOT 1 -> 0

A black and white image of two black and white letters

AI-generated content may be incorrect.

. ábra: Logikai kapuk

Speciális rész a kapcsolótábla mely fogad és ad is le energiaállapotot, de jeladóként van nyilvántartva elsődlegesen. A kapcsolótáblákban többféle elrendezés is létezik, tekintsünk példának egy 1 bemenettel rendelkező kapcsolótáblát (18. ábra) és működését, ahol egy karból jövő jelet módosít és ad tovább egy ajtó felé (19. ábra):

A pixel art of a grey square with a triangle and a black triangle

AI-generated content may be incorrect.

. ábra: 1 bementes kapcsolótábla belső szerkezet

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: 1 bemenetes kapcsolótábla működése

További lehetséges kapcsolótábla belső elrendezéseire példák:

* 2 bemenetes kapcsolótábla: A két bemeneti szálon egy-egy egybemenetes logikai kapu található, melyek kimenete egy közös, kétbemenetes logikai kapuba csatlakozik.

A pixel art of a light switch

AI-generated content may be incorrect.

. ábra: 2 bemenetes kapcsolótábla elrendezés

* 3 bemenetes kapcsolótábla: Az első két bemenet egy kétbemenetes logikai kapuba vezet, majd ennek kimenete a harmadik bemenettel együtt újabb kétbemenetes kapuba torkollik. A harmadik bemeneti szálon opcionálisan egy egybemenetes kapu is helyet kaphat.

A pixel art of a switch

AI-generated content may be incorrect.

. ábra: 3 bemenetes kapcsolótábla elrendezés 1

* Alternatív elrendezésű 3 bemenetes kapcsolótábla: A középső bemeneti szál kettéágazik, és a keletkezett két ágban egy-egy egybemenetes logikai kapu helyezkedhet el. Ezek a kapuk különböző kétbemenetes logikai kapukba csatlakoznak, ahogy az a 23. ábrán is látható.

A pixel art of a diagram

AI-generated content may be incorrect.

. ábra: Alternatív 3 bemenetes kapcsolótábla elrendezés

### Felhasználói felületek

#### Főmenü

A játék elindításakor a főmenübe érünk, innen tudjuk a [NEW GAME] gombbal elindítani a játékot. A játékból innen ki is tudunk lépni a [QUIT] gombbal.

A screenshot of a video game

AI-generated content may be incorrect.

. ábra: Főmenü

#### Játékmenü

A játék közben az [ESC] bilentyűvel megjeleníthető menü, innen vissza tudunk menni a [főmenübe](#_Főmenü) a [MAIN MENU] gombbal. Továbbá kiléphetünk a játékból a [QUIT] gomb segítségével. Ennek a menünek a megnyitásakor a játék megáll. valamint kiléphetünk belőle a [RESUME] gombbal ami a játékot újraindítja.

A screenshot of a video game

AI-generated content may be incorrect.

. ábra: Játékmenü

#### Játék végi menü

A játék teljesítése után (ha a játékos használja a főkonzolt a hajó hídján) jelenik meg. Innen vissza lehet menni a főmenübe a [MAIN MENU] gomb megnyomásával, és ki tudunk lépni a játékból a [QUIT] gomb megnyomásával.

A screenshot of a video game

AI-generated content may be incorrect.

. ábra: Játék végi menü

#### Kapcsolótábla belső elrendezés menü

Ez egy nyitott kapcsolótáblával való interakció után jelenik meg, amíg a képernyőn van addig a játék leáll (szünetel). Az [OK] gomb megnyomásával vissza lehet térni a játékhoz.

A screenshot of a video game

AI-generated content may be incorrect.

. ábra: Kapcsolótábla belső menü

# Fejlesztői dokumentáció

Ez a fejezet bemutatja a játék architektúráját, az egyes komponensek felépítését, működését és a közöttük lévő kapcsolatokat.

## Megoldási terv

A játék Godot játékmotor 4.4-es verziójában, a motor saját scriptnyelvében GDScriptben lett elkészítve. A játékmotor támogatja 2D-s játékok hatékony fejlesztését és kezelését. A Godot egy node alapú rendszert alkalmaz ami segíti az egyes funkciók moduláris szétválasztását és menedzselését. A játékelemek saját önálló Scene-ekbe (jelenetekbe) vannak szervezve (Pl: lever\_scene.tscn, main\_menu\_scene.tscn). Ezeket a scene-eket (jeleneteket) példányosítani lehet egy másik scene-ben. Létezik egy „fő scene”, amit elindításkor példányosít a motor, és ez úgymond a SceneTree-be kerül bele (a SceneTree szüneteltethető, ami azt jelenti, hogy minden folyamat felfüggesztésre kerül, kivéve azok, amelyek PROCESS\_MODE.ALWAYS módban futnak – ezek továbbra is aktívak maradnak). A GDScript beépített függvényei alsóvonallal kezdődnek (Pl. \_init(), \_ready(), \_process, \_physics\_process()).

* \_init(): Ez a függvény akkor fut le amikor az adott osztály (script) példányosítása történik. Fontos, hogy MINDIG a \_ready() függvény előtt fut le.
* \_ready(): Ez a függvény akkor fut le amikor teljesen betöltődött a scene-be, bekerült a SceneTree-be és minden gyerekének lefutott a \_ready() függvénye.
* \_process(): Ez a függvény minden képkocka (frame) megjelenítése előtt meghívódik, tehát annyi alkalommal fut le másodpercenként, amennyi a gép által elért képkockaszám (frame rate). Ezért alkalmas nem-fizikai tevékenységek, például animációk, UI-frissítések vagy logikai műveletek kezelésére.
* \_physics\_process(): Ez a függvény minden futási ciklusban meghívódik, másodpercenként általában 60 alkalommal, azaz fix időközönként. Emiatt ideális a fizikai számítások, például mozgás, ütközéskezelés vagy gravitáció kezelésére.
* \_input(): Ez a függvény akkor fut le, amikor a program bemeneti eseményt érzékel.

A \_process és \_physics\_process függvényeket lehet egy delta paraméterrel hívni, amely megadja a két lefutás között eltelt időt.

A karakter alapötlete az, hogy egy kis robot, ami valamennyire hasonlít a Wall-E-hoz, saját preferencia miatt és alapötletben 2 lába volt és az ugrás animációban a lábai voltak a rakéták amelyekkel tudott ugrani, de ezt később, mivel nem tudok jól rajzolni, átalakítottam úgy, hogy lánctalpa van Marvin-nak és a hátán helyezkedik el a jumpPack-je.

### SwitchBox terv

Előzetes tervezések közül megemlítésre méltó a SwitchBox amelynek a belső logikája és a megjelenítése lett megtervezve. Ki kellett számolni, hogy pontosan mekkora ábra kell a belső logika megjelenítéséhez, ami végülis 128x128 pixeles képek négyszeres nagyításával lett megoldva. A belső logika pedig előre definiált minták alapján lett implementálva (lásd 29.ábra).

A black background with white triangles

AI-generated content may be incorrect.

. ábra: SwitchBox mintái

## A játékprogram

### World

#### World.tscn

A world scene a játék fő vezérlője, neki a gyerek (child) node-jai lesznek a player karakter és a szint amin a karakter van. Statikusan 2 gyermeke van, amelyek egy TileMapLayer, ami a megjelenítésben vesz részt és az űrt szimbolizálja a hajó körül. Valamint egy Node – DeathAnimContainer - aminek 2 gyereke van, egy AnimationPlayer – ami valamilyen animáció lejátszásáért felel - és egy ColorRect – amely egy színes téglalap. Ezek együtt alkotják a „Death Animation”-t amely azt jelenti, hogy egy fekete téglalap jelenik meg és tűnik el amikor az adott szint újraindul.

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: World package diagramja

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: World Scene a Godot editorban

#### World osztály – world\_srcipt.gd

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: World class diagramja

**Változók:**

levelIndex: Ez alapján töltődik be a szint.

LevelSpawnPoint: Az adott szint kezdőpontját mentjük bele, könnyebb hozzáféréshez.

Player: A karakter-t ide példányosítjuk és innen kezeljük.

MainMenu, PauseMenu és Endscreen: A 3 menüt ezekbe példányosítjuk és innen kezeljük őket.

Level-, Player-, MenuCamera: A felsorolt elemek kameráit ide mentjük bele könnyebb hozzáféréshez.

collectibleGot: Változó ami nyilvántartja, hogy a játékos felszedett-e egy tárgyat amely az EndScreenen fog majd megjelenni.

**Függvények:**

\_ready(): A feladata, hogy inicializálja a MainMenu, PauseMenu és EndScreen változókat és utána betöltse a Main Menüt. Valamint beállítja a nekik és a DeathAnimationHandler-nek, hogy akkor is futhasson amikor a SceneTree le van szüneteltetve.

\_input():

* A [TAB] billentyű megnyomásakor meghívja a switchCameraState() függvényt amivel kameranézetet vált a LevelCamera és a PlayerCamera között.
* Az [ESC] billentyű megnyomásakor szünetelteti a tree-t és meghívja a loadMenu() függvényt és átváltja a kameranézetet a MenuCamera-ra.

switchCameraState(desiredCameraState): A kameranézetet átváltja a desiredCameraState által meghatározott CameraState-re. A LevelCamera-ra váltáskor a kamera pizícióját a szint közepére állítja.

A group of ovals with text

AI-generated content may be incorrect.

. ábra: Kameranézet state diagramja

loadLevel(): Példányosítja és meghívja a tree-be a levelIndex számozású szintet és a LevelSpawnPoint-ot az adott szint SpawnPoint nevű Node-jára állítja. Minden egyes szint teljesítésekor (amikor a karakter használja a szint végi ajtót) meghívódik.

loadPlayer(): Példányosítja és meghívja a tree-be a karaktert. Ez csak a Main Menu-ből kilépéskor hívódik meg.

loadMenu(): Meghívja a Pause Menu-t a tree-be és inicializálja a MenuCamera-t.

resetPlayerPosition(): A karakter sebességét lenullázza és a pozícióját a LevelSpawnPoint pozíciójára állítja. Ez a függvény az új szint betöltésekor és a szint újraindításakor van meghívva.

nextLevel(): Meghívja a deathAnimation() függvényt és lenullázza a LevelCamera és a LevelSpawnPoint változókat. Ezt követően meghívja a Level változó queue\_free() függvényét amely egy beépített függvény, aminek hatására a node törlésre kerül a tree-ből. Majd inkrementálja a levelIndex változó értékét és meghívja az initLevel függvényt, és ha minden készen áll akkor meghívja a deathAnimationRev() függvényt.

deathAnimation(): Ez a függvény elindítja a DeathAnimationHandler „DeathScreenIn” animációját, amelyben a DeathAnimContainerben megtalálható DeathScreen fekete téglalap áttetszőségét 100%-ról, 0%-ra csökkenti.

deathAnimationRev(): Ez a függvény elindítja a DeathAnimationHandler „DeathScreenOut” animációját, mely az előbb említett DeathScreen áttetszőségét 0%-ról 100%-ra emeli.

intiLevel(level): Először is ha létezik a tree-ben a Main Menu, akkor eltávolítja. Ezt követően az argumentumban megadott levellel egyenlővé teszi a levelIndex változót és meghívja a loadLevel() függvényt, valamint, ha nem létezik a karakter a tree-ben, akkor meghívja a loadPlayer() függvényt és elvégzi a szintekhez rendelt utasításokat, valamint alaphelyzetbe állítja a karakter pozícióját és átváltja a kamerát.

showMainMenu(): Eltávolít minden menüt, a karaktert és a szintet a tree-ből és meghívja a Main Menüt a tree-be.

resumeGame(): Eltávolítja a Pause Menu-t a SceneTree-ből, a szüneteltetést feloldja és a kamerát a PlayerCamera-ra állítja.

showEndScreen(): Eltávolítja a szintet és a karaktert a tree-ből és meghívja az EndScreen-t.

### Player

#### player.tscn

Ez a játékos által irányított karakter megjelenítését, mozgását és a szint részeivel való kapcsolatba lépését kezeli.

A screenshot of a computer program

AI-generated content may be incorrect.

. ábra: A Player Scene a Godot editorban

A player scene-nek sok gyereke van, de kiemelhető közülük:

* BaseCollShape: Ez a karakter ütközési rétege, amellyel például megállhat a földön vagy beleütközhet egy falba.
* MainSprite: Amely a karakter animációiért felel, kódban van változtatva, hogy éppen melyik animáció játszódik.
* HurtBox2D: Egy olyan Area2D amely a HitBox2D megfelelője, amely a Hitbox2D-vel való érintkezéskor meghívja a player takeDamage() függvényét.
* InteractorArea2D: Ez egy olyan Area2D, amelyet az InteractionArea2D node képes érzékelni, mivel ugyanazon a collision layeren helyezkedik e.
* TeleportArea: Egy Area2D, amely által el lehet dönteni, hogy a karakter teleportálása megtörténhet-e (a végcél nem egy fal belsejében van).

#### Player osztály – player\_script.gd

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: Player osztály osztálydiagramja

**Változók:**

SpeedChange, speedSlow, speedCap: a mozgás alapvető részei jobban kifejezve a movementHandler() függvénynél.

gravity: Az a sebesség amivel a karakter gyorsul lefelé amikor nem áll a földön, valamint ezt az értéket multiplikálja a multGravity() függvény.

jumpCharges(): Számon tartja hogy hány ugrása van még a karakternek.

jumpForce: Az a sebesség amivel a karakter ugrani képes felfelé.

maxJumpCharges: Számon tartja, hogy a karakternek hány darab jumpCharge-a lehet maximum (1 vagy 2, ha felvette már a JumpPack-et).

canGetJump: Ez a logikai változó az elektromos kerítések által adott jumpCharge-ot tartja számon, úgy hogy amíg a levegőben van a karakter addig csak 1-szer kaphat elektromos kerítéstől.

canTeleport: Az a logikai változó megmondja, hogy a karakter képes a teleportálásra, vagy nem.

currentState: Eltárolja a karakter éppen játszódó AnimationState-jét.

interactions[]: Egy tömb, amelyben azok a Node-ok vannak nyilvántartva, amelyekkel a karakter kapcsolatba léphet.

canInteract: Egy logikai változó, amely megmondja, hogy a karakter tud-e interaktálni.

**Függvények:**

\_ready(): Beállítja a 33.ábrán látott: JetpackIcon1, JetpackIcon2, TeleportIcon pozícióját és a TeleportArea ütközési maszkját, amellyel észleli, hogy az ütközik-e valamivel.

\_input():

* Az [E] billentyű megnyomásakor meghívja az interactions tömb első elemének interact() függvényét, ha a karakter kapcsolatba tud lépni tárgyakkal (canInteract).
* Amíg a [S] billentyű vagy a lefele nyíl le van nyomva, addig kikapcsolja a karakter 5ös ütközési rétegét és maszkját, amit a gombok felengedésével visszakapcsol.
* Az [X] billentyű lenyomásakor meghívja a takeDamage() függvényt, amivel az aktuális pálya újraindul.

\_physics\_process(): A karakter mozgását a movementHandler() függvény kezeli, amely beállítja a horizontalDirection változó értékét – 1, ha a karaktert jobbra, –1, ha balra szeretnénk mozgatni, és 0, ha nem érkezik mozgásra vonatkozó input –, emellett kezeli az ugrást és teleportálást, valamint meghívja az updateScrewDriverPosition() függvényt is, amely a csavarhúzó pozícióját frissíti.

\_process(): Kezeli a karakter animációit (animationHandler() függvényt (animációs anomáliák elkerülése végett, nem a \_physics\_process-ben).

updateScrewDriverPosition(horizontalDirection): Kezeli, hogy melyik kezében legyen a karakternek a csavarhúzója a horizontalDirection és a karakter adott x tengelyen való sebessége alapján.

movementHandler(horizontalDirection): Hogyha a horizontalDirection nem 0, akkor gyorsítja a karaktert az adott irányba a speedChange változó értékével, valamint ha 0 akkor csökkenti a sebességét a speedSlow változó értékével. Ha átlépi a speedCap által megszabott sebességet akkor amennyivel gyorsabb annak az 1/3-ával csökkenti a sebességét végrehajtási ciklusonként.

animationHandler(horizontalDirection): A currentState és a horizontalDirection által megadott adatok alapján a MainSprite megfelelő animációját állítja be valamint a horizontalDirection irányától függően forgatja azt.

takeDamage(): Újraindítja az aktuális szintet a World scene restartLevel() függvénye által. A meghívás előtt meghívja a deathAnimation(), utána a deathAnimationRev() függvényeket.

addInteractable(area): Hozzáadja az area által megszabott Area2D-t az interactions[] tömbhöz, amellyel a karakter kapcsolatba tud lépni.

removeInteractable(area): Az area által megszabott Area2D-t kiszedi az interactions[] tömbből.

multGravity(gravMult): Multiplikálja a gravity változót a gravMult értékkel.

addJumpCharge(num, force): Ad a karakternek num számú jumpCharge-ot, ha force változó igaz, akkor akkor is képes odaadni, ha a canGetJump hamis.

updateJumpChargeSprite(): A karakter feje fölött lévő jump pack ikon frissítéséért felel.

updateTeleportSprite(): A karakter feje fölött lévő teleportációs készülék ikon frissítéséért felel.

### A szintek alapkialakítása

Minden szint statikusan van összerakva a Godot motor editorában. Minden egyes szintet manuálisan kell összerakni és vannak szabályok, amelyek mentén megépülnek:

* Megjelenítés szempontjából áll egy BaseLayer, ami a szint hátterét, valamint egy MapElementsLayer amely a szinten található falakat és platformokat vizualizálja.
* Lennie kell egy kamerának aminek a nevének „Camera” -nak kell lennie
* Egy „Objects” elnevezésű Node2D-nek, amelyben elhelyezkedik a [szinten megtalálható összes elem](#_Szinteken_megtalálható_elemek) (a platform kivételével, mert az a MapElementsLayer része), ezek közül aminek muszáj léteznie az egy „SpawnPoint” elnevezésű Marker2D Node, amely kijelöli a karakter kezdőpozícióját.

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: Példa a szint kialakítására a Godot editorában

### HitBox HurtBox

Amikor a karatkernek baja esik, például ráesik egy tüskére vagy egy elektromos kerítésre, akkor a karakter HurtBox-a detektálja, hogy érintkezett a másik elem HitBox-ával és ezért egy signal-t küld ki, amelyet elkap és ezért meghívja a „gazdájának” a takeDamage() függvényét. Az érintkezést azért tudjuk irányítani, mert úgy van implementálva a Godot motorban, hogy a legtöbb objektumnak van „collision\_layer”-e és „collision\_mask”-ja, amelyek közül a mask érzékeli a layert (a 2-es layer-en tudnak érintkezni) (lásd 37.ábra).

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: Hitbox Hurtbox relációja

### InteractionArea és InteractorArea

Hasonló a működésük a [HitBox és HurtBox](#_HitBox_HurtBox)-hoz (4.2.4). Abban térnek el tőlük, hogy:

* Érintkezéskor: meghívja a karakter addInteraction() és a „gazdája” showInteract() vagy pickUp() függvényét.
* Kilépéskor: meghívja a karakter removeInteraction() és a „gazdája” hideInteract()

### ScrewDriverInteractor és InteractionArea

Hasonló a működésük a [HitBox és HurtBox](#_HitBox_HurtBox)-hoz (4.2.4). Abban térnek el tőlük, hogy:

* Érintkezéskor: meghívja a karakter addJumpCharge() és a „gazdája” (ami egy SwitchBox) startTimer() függvényét.
* Kilépéskor: meghívja a „gazdája” pauseTimer() függvényét.

### screw\_driver.tscn és screw\_driver\_script.gd

#### screw\_driver.tscn

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: ScrewDriver scene felépítése

A ScrewDriverInteractorArea a csavarhúzó végén helyezkedik el aminek segítségével ki lehet nyitni a SwitchBox-okat és lehet jumpCharge-ot kapni az elektromos kerítésekből.

#### screw\_driver.gd

A screenshot of a computer

AI-generated content may be incorrect.

. ábra:Screw Driver osztálydiagramja

**Függvények:**

\_ready(): Beállítja a saját méretezését a szülője méretének 0,75-szörösére

\_physics\_process(): A beépített look\_at() függvénnyel a kurzor felé irányítja magát.

### PWR osztály, globalPowerState.gd

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: PWR osztálydiagramja

Függvényei logikai kapukat jelentenek, szóval a bemenet alapján visszaadják a megfelelő logikai eredményt. Pl. NOT(PowerState.ON) => PowerState.OFF; XOR(PowerState.ON, PowerState.OFF) => PowerState.ON.

### End Of Level Door

#### end\_of\_level\_door.tscn

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: End of level door felépítése

Az InteractionSprite ki van kapcsolva az editor-ban láthatóság miatt és a BaseCollShape ki van kapcsolva, mert ezzel az objektummal nem érintkezhet a karakter.

#### end\_of\_level\_door\_script.gd

A screenshot of a computer program

AI-generated content may be incorrect.

. ábra: EndOfLevelDoor osztálydiagramja

\_ready(): Beállítja a pozícióját az InteractionSprite-nak és a beállítja az alap powerState-et.

showInteract() és hideInteract(): Megjeleníti és eltűnteti az InteractionSprite-ot.

interact(): Meghívja a World nextLevel() függvényét, ezzel elkezdi betölteni a következő szintet.

switchOn() és switchOff(): Kinyitja és becsukja az ajtót (ezzel be/kikapcsolja az InteractionArea collision-jét és lejátssza a megfelelő animációt) és átállítja a currentState változó értékét.

switchState(desiredState, force): Meghívja a switchOn() és switchOff() függvényeket ha váltania kell az állapot és desiredState között, hogyha ki vagy be van kapcsolva. A force argumentummal akkor is megtörténik a váltás, ha az aktuális állapot megegyezik a kívánttal.

### Felvehető tárgyak osztályai:

Ezek az objektumok nem jelennek meg minden szinten, azonban ahol igen, ott a pálya teljesítése szempontjából kulcsfontosságú szerepük van – kivéve egy esetet, amely csak opcionálisan gyűjthető, dekoratív (gizmo) elemként funkcionál.

Ezek az objektumok az InteractionArea2D Node segítségével valósítják meg az interakciót, amely lehetővé teszi, hogy a karakter érintkezés útján „felvegye” őket. Fontos ezért, hogy kivegyük őket a karakter interactions[] tömbjéből a scenetree-ből kivétel előtt.

* **pickUpJumpPack:** Ez az objektum biztosítja a karakter számára a dupla ugrás lehetőségét, azaz a 2 maximum Jump Charge-ot. A karakter kezdeti töltöttségi szintjét a world scene határozza meg, szintfüggően 1 vagy 2 értékkel indítva. A pickUpJumpPack felvétele után a karakter mindig teljes (2-es) töltöttséggel rendelkezik.
* **pickUpTeleporter:** Ezzel a tárggyal válik lehetővé a karakter számára az előre irányuló teleportálás. A játékos az adott szinten veheti fel, előtte a karakter canTeleport változója false, így a teleportálás nem lehetséges. Ennek értékét szintfüggően a World Scene állítja be a karakter példányosításakor- a felvételt követően a canTeleport true lesz.
* **jumpPackRefresh:** Ez a tárgy visszatölti a karakter jump charge-jait (ugráskapacitását). Felvétele után az eszköz inaktívvá válik 5 másodpercig, majd újra használhatóvá lesz. A visszatöltési időt a tárgyhoz tartozó Timer Node szabályozza, amely szükség esetén paraméterezhető.
* **Collectible:** Egy gyűjthető tárgy, amelynek felvételekor a World Scene-ben található collectibleGot változó értéke true lesz. Ez az információ később, a EndScreen scene-be kerül felhasználásra.

### Spikes

A Spikes objektumnak csak saját scene-je létezik, külön scriptet nem tartalmaz. Szerepe, hogy veszélyforrás legyen a szinteken, amik megértinésével azonnal újraindul a szint. Elhelyezhető padlón, falon vagy plafonon, így a szintek során változatos módon akadályozhatja a karakter mozgását.

A legtöbb pályán a tüskék egy SpikeCluster nevű Node2D csoportban találhatók, ami megkönnyíti azok együtt mozgatását és pozicionálását – ez azonban nincs hatással a működésükre.

A játékossal való interakció úgy történik, hogy a Spikes scene tartalmaz egy HitBox node-ot, amely a karakter HurtBox-ával ütközve halálos érintkezést vált ki, ezáltal a pálya újraindul.

### Electric Fence:

Az [End Of Level Door](#_end_of_level_door.tscn_és_end_of_le)-al hasonlóan ki és be lehet kapcsolni, ha egy áramkör része. Használja a PWR.PowerState enumerátort az energiaállapota nyomon követéséhez. Energiaállapot váltásért a switchState() függvénye felel, amely a megjelenését és az ütközési rétegét változtatja. Valamint amiért közös InteractionArea2D Node-ja van a SwitchBox-al ezért egy canGiveJump változó által megmondható róla, hogy képes jumpCharge-ot adni a karakternek.

### GravityArea és NoTeleportArea

Ezek az objektumok az Area2D leszármazottjai, amelyek a karakter gravitációját és a teleportálási képességét változtatják meg.

Amikor a karakter belép egy GravityArea zónába, meghívódik a karakter multGravity() függvénye, a zóna gravityMult változójának értékével. Ezzel a karakterre ható gravitáció megnövekszik vagy csökken, a zóna beállításától függően. Kilépéskor ugyanez a függvény hívódik meg, de a paraméterként átadott érték a gravityMult reciprok értéke, így a gravitáció visszaáll az eredeti szintre. Kinézetéért a \_ready() függvénye felel, amely a gravMult változó alapján színezi a területét (nagyobb 0-nál akkor egyre pirosabb, kisebb 0-nál akkor egyre égszínkékebb).

Amikor a karakter egy NoTeleportArea-ba lép be, akkor a karakter canTeleport változójának az értéke false lesz. Kilépéskor ez visszatér az eredeti true állapotába, visszaállítva a karakter teleportálási képességét.

A screenshot of a computer

AI-generated content may be incorrect.

. ábra: gravityArea osztálydiagramja

### Door-kapu

Ezek a szinten lévő logikai feladatok részei, vagy egy kivételben, a teleportációs készülék használatára tanítják meg a játékost. Az [End Of Level Door](#_end_of_level_door.tscn_és_end_of_le)-al megfelelően be és ki lehet kapcsolni, hogyha egy áramkörben van, ami a nyitás és zárás mozzanatait hozza létre. Ezek animációit a currentState –amely az energiaállapotára utal- változtatásával együtt lejátssza.

### PowerLine

Ez egy objektum amely arra szolgál, hogy a PowerGrid-en belül PowerState-et vigyen át egy jeladó (emitter) felől egy vevő (receiver) felé, amelyek is egy PowerGrid részét képezik. A PowerLine egy mondhatni P2P - peer to peer - kapcsolatot létesít 2 PowerGrid-ben szereplő objektum között. Továbbá rendelkezik a saját PowerState-jével, amit a játékosnak a színe által kommunikál le. Lényegében amikor a PowerGrid-től megkapja az utasítást, hogy váltson PowerState-t, akkor megváltoztatja a színét, és átadja a parancsot a vevő felé. Fontos megjegyezni, hogy a PowerLine switchState() függvényének nem 2 paramétere (desiredState, force) van, hanem 3 (desiredState, force, reset), amelyben a reset feladata, hogy elkerülje azt az eshetőséget, hogy végtelen ciklusban fusson, mert egy SwitchBox kimeneti vezetéke kötődhet saját magába, amely nem hívja meg a SwitchBox belső függvényét, mert az azonnal visszaállítaná a vezeték energiaállapotát, így végtelen ciklust indítana.

### PowerGrid

A PowerGrid képezi a logikai feladatok megoldására használt áramkörök implementálását. Működése biztosítja, hogy például egy meghúzott kar át tudja adni a hozzá kapcsolt PowerLine-nak a energiaállapotát. Ennek az információnak átjutása érdekében a PowerGrid egy úgymond middle-man, aki megkapja a signal-t a kartól (vagy SwitchBox-tól) és eljuttatja a hozzá kapcsolódó PowerLine-(ok)nak. A logikát a refreshGrid() függvény kezeli. Ennek a működésére egy egyszerű példa:

* A játékos meghúz egy kart.
* A kar kiküld egy signal-t.
* A PowerGrid megkapja ezt a signal-t és azt hogy melyik kar küldte.
* A PowerGrid átfut az összes vezeték között és átadja nekik az energiaállapotot.
* A vezeték (PowerLine) megkapja és átállítja a saját energiaállapotát.
* A vezeték továbbadja az energiaállapotot a vevő felé és az lekezeli.

Azért, hogy ez a folyamat problémák nélkül és optimálisan futhasson néhány alapvető szabály-t kell megfogalmazni a PowerGrid használatával szemben:

* Kell lennie 3 Node-nak (sima Node-nak kell lennie!) amiknek sorrendje nem számít, de a nevük igen. A 3 Node neve: Emitters, Wires, Receivers
  + Az Emitters-ben a jeladók foglalnak el helyet: kar, SwitchBox
  + A Wires-ben csak PowerLine2D lehet.
  + A Receivers-ben csak vevő lehet.

Valamint az adott szint újrakezdésekor meghívódik a Reset() függvénye ami minden vezeték és jeladó-t alaphelyzetbe állít. Nem szükséges a vevőket alaphelyzetbe állítani, mivel a refreshGrid() függvény meghívódik többször is az alaphelyzetbe állítás közben.

### Lever – kar

Ez egy játékos által használható objektum, amely rendelkezik egy stateChanged signal-al, ami a PowerGrid refreshGrid() függvényével kommunikál és átadja önmagát, amivel a PowerGrid hozzáfér az energiaállapotához. A PowerGriddel való relációja a 44. ábrán látható.

A screenshot of a computer

AI-generated content may be incorrect.

44. ábra: Lever és PowerGrid relációja és működése

### SwitchBox – kapcsolótábla

#### Alapfunkciók

A SwitchBox több funkcióval rendelkezik, de az alapvető működése az, hogy a játékos MARVIN csavarhúzójával interakcióba lépve le tudja szerelni a kapcsolótábla fedelét. Ehhez a játékosnak a csavarhúzót legalább 2 másodpercig a kapcsolótáblán kell tartania. A folyamat során a rendszer példányosítja a switch\_box\_cover.tscn scene-t, amely a fedél leesését szimulálja, majd az 5 másodperc múlva eltűnik. Ezután a SwitchBox használhatóvá válik, és a játékos megtekintheti a belső logikai felépítést.

#### Belső logika megjelenítése

Ennek a megoldása úgy van implementálva, hogy átadja a [subRoutine](#_SwitchBox_terv)-t (29. ábra) és a kapukat amik szerepelnek benne, ezt követően a switch\_box\_menu.gd osztály kiválasztja a megfelelő mintát a SwitchBoxPresets (20-24. ábrák) scene-k közül és beállítja a megfelelő kapukat rajtuk. Erre példa (45. ábra):

* Subroutine: ThreeBaseOne
* Kapuk: Gate0: OR, Gate1: OR, Gate2: NOT

A pixel art of a switch

AI-generated content may be incorrect.

. ábra: SwitchBoxPreset példa

#### Működése

A kapcsolótábla működését lépésekre bontva az alábbiak szerint lehet összefoglalni:

* A SwitchBox bemeneti vezetékein keresztül energiaállapotot kap.
* Elindul a megfelelő subroutine a startSubroutine() függvény meghívásával.
* A subroutine meghívja a callGateFunc() függvényt, amely a megadott vezetékek és logikai kapuk alapján kiszámítja a kimeneti értékeket.
* A callGateFunc() átadja a vezetékek *energiaállapotát* a PWR osztály megfelelő függvényének, amely a logikai számítást végzi.
* Minden logikai kapu meghatározza saját kimenetét – bizonyos esetekben nem közvetlenül a vezeték, hanem a kapu energiaállapotát veszi figyelembe.
* A SwitchBox currentState változója frissül a végső kapu eredménye alapján.
* A SwitchBox végül kibocsátja a stateChanged signal-t, amely megegyezik azzal, amit a kar is használ a rendszerben.

#### Beállításai:

A SwitchBox-ot be kell állítani használat előtt a Godot editorban.

A screenshot of a phone

AI-generated content may be incorrect.

. ábra: SwitchBox beállításai

Lefelé a beállítások:

* Subroutine: Választani kell egyet a 9 subroutine közül. Az összes subroutine kialakítása látható a 29.ábrán.
* A kapukat (Gate 0-3) be kell állítani úgy hogy a subroutine-nak megfeleljen.
* A vezetékeket (Wire 0-2) be kell állítani és nem elég ha itt beállítjuk őket, mert be kell állítani azt, hogy a PowerLine2D-nek a receiver-je legyen a SwitchBox. Valamint hogy a bemeneti vezeték a SwitchBox jó vezetékhelyén foglaljon helyet.

## Tesztelés

A játék fejlesztése során a rendszer működését **folyamatosan saját kezűleg teszteltem**, hogy megbizonyosodjak a funkciók helyes működéséről, a hibák időben történő észleléséről, valamint a játékmenet stabilitásáról. A tesztelés célja az volt, hogy az összes játékelemet – az alap mechanikáktól a logikai rendszerekig – külön-külön és összefüggéseikben is ellenőrizzem.

A pályák tervezése és megvalósítása után minden szintet egyenként többször is végigjátszottam, külön figyelmet fordítva a következőkre:

* **Mozgás, ugrás, gravitáció**: ellenőriztem, hogy a karakter mozgása minden gravitációs zónában megfelelően működjön, és hogy a jump pack használata korrekten korlátozva legyen.
* **Teleportáció**: meggyőződtem róla, hogy a teleportáló eszköz nem engedi a játékost falba teleportálni.
* **Logikai kapcsolók és kapcsolótáblák**: minden lehetséges logikai bemenetet és kombinációt kipróbáltam, hogy a logikai kapuk helyes működést biztosítsanak, és a kimeneti állapot megfelelően befolyásolja a pályaelemeket (pl. ajtók nyitása).
* **Interakciók**: minden felvehető és használható objektumot (pl. karok, frissítők, felvehető tárgyak) külön is ellenőriztem, hogy megfelelően váljanak aktívvá, illetve eltűnjenek felvétel után.
* **Felhasználói felület**: a menük, HUD-elemek és ikonok megjelenését, valamint a szünet és kilépési funkciók működését is végigpróbáltam.

A tesztelés során előforduló kisebb hibákat (pl. pozícióból való kiesés, hibás állapotváltás) folyamatosan javítottam, és a játék működését újra ellenőriztem. Mivel a fejlesztés és a tesztelés kéz a kézben haladt, lehetőségem volt minden hibát közvetlenül a megjelenése után kezelni.

A játék jelenlegi állapotában stabilan és hibamentesen fut, minden szint teljesíthető, és az egyes játékelemek a terveknek megfelelően működnek.

# Összefoglalás és további fejlesztési lehetőségek

Dolgozatom célja egy saját készítésű, 2D-s oldalnézetes puzzle-platformer játék megvalósítása volt, amely során lehetőségem nyílt megismerni a Godot játékmotort, és első kézből tapasztalni meg, milyen folyamat egy játékot a semmiből felépíteni. A fejlesztés során a legnagyobb örömet az jelentette számomra, amikor a napokon át tervezett játékelemek a gyakorlatban is működtek – az ötletekből végül működő játékmenet született.

A fejlesztés részeként megismerkedtem a 2D-s pixel art stílussal is. Bár úgy érzem, hogy ebben még nem vagyok igazán jártas, fontos tapasztalat volt számomra, hogy lássam, hogyan érdemes elindulni ebben az irányban. Bár a végeredmény vizuálisan nem tökéletes, büszke vagyok arra, hogy a saját lehetőségeimhez mérten a maximumot igyekeztem kihozni a projektből.

A jövőben szeretném továbbfejleszteni a játékot, hogy mások számára is érdekes és élvezetes élményt nyújtson – akár csak rövid időre is. A tervek között szerepel újabb logikai eszközök, például időzített gombok vagy nyomólapok beépítése, melyek tovább bővítik a pályák megoldási lehetőségeit. Ezek az elemek újfajta kihívásokat hozhatnak a játékba, és hozzájárulhatnak a játékélmény mélyítéséhez.

# Irodalomjegyzék

1. Godot játékmotor dokumentációja: <https://docs.godotengine.org/en/stable>
2. Hitbox és hurtbox logika: [https://www.youtube.com/watch?v=JWjzSn95bM0](https://www.youtube.com/watch?v=JWjzSn95bM0&ab_channel=GDQuest)
3. Donkey Kong (1981) története: <https://en.wikipedia.org/wiki/Donkey_Kong_(1981_video_game)>
4. Logikai kapuk leírása: <https://en.wikipedia.org/wiki/Logic_gate>