Beszámoló a szakmai gyakorlatról



Miskolci Egyetem

2D platformer játék fejlesztése   
a Unity játékmotor használatával

**Készítette**:

Szabó Martin

Programtervező informatikus

Neptun-kód: WJFAOO

**Üzemi instruktor**:

Piller Imre

Tanársegéd

+36 30 427 5717

**Miskolc, 2022**

Tartalom

[**1.** **Technológiák ismertetése** 3](#_Toc111618894)

[**1.1.** **Az Unreal Engine játékmotor** 3](#_Toc111618895)

[**1.2.** **A Godot játékmotor** 4](#_Toc111618896)

[**1.3.** **A Unity játékmotor** 4](#_Toc111618897)

[1.3.1. A Unity szerkesztője 5](#_Toc111618898)

[**2.** **A Reddy The Hero játék bemutatása** 6](#_Toc111618899)

[**2.1.** **A játék leírása** 6](#_Toc111618900)

[**2.2.** **Reddy, a játékom főhőse** 6](#_Toc111618901)

[2.2.1. A Hero osztály 7](#_Toc111618902)

[2.2.2. A MainCamera objektum 10](#_Toc111618903)

[2.2.3. Reddy animációi 11](#_Toc111618904)

[**2.3.** **A környezet, az érmék és az ellenségek elkészítése** 12](#_Toc111618905)

[2.3.1. A Grid objektum, valamint a Tile Palette használata 12](#_Toc111618906)

[2.3.2. Érmék, ládák meganimálása és számlálása 13](#_Toc111618907)

[2.3.3. A Háttér 14](#_Toc111618908)

[2.3.4. Az ellenségek, valamint a víz animációjának az elkészítése 14](#_Toc111618909)

[**2.4.** **A főmenü, valamint a felhasználói felület** 15](#_Toc111618910)

[2.4.1. A Főmenü 15](#_Toc111618911)

[2.4.2. A GameOverPanel objektum 16](#_Toc111618912)

[**3.** **Összegzés** 18](#_Toc111618913)

# **Technológiák ismertetése**

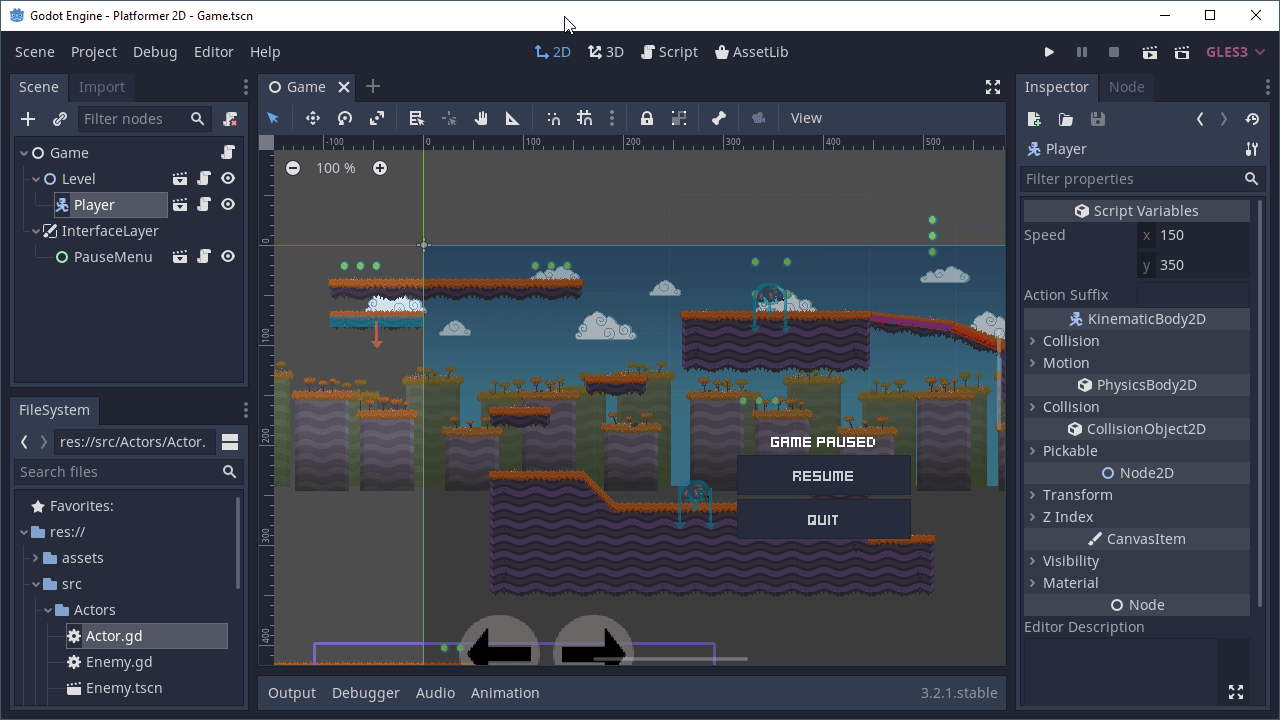
A technológiák ismertetése a következőkben a játékmotorok bemutatását fogja takarni. Ezek közül az Unreal Engine-t, a Godot-ot, valamint a Unity-t fogom részletesebben bemutatni.

## **Az Unreal Engine játékmotor**

Az Unreal Engine egy játékfejlesztésre használt, nyílt forráskódú motor, amelyet az Epic Games fejlesztett ki 1988-ban. Híres az általa készített játékok gyönyörű grafikájáról. A fizikai motorja fejlett, melyet „Chaos” néven ismerünk. Több platformra is lefordíthatjuk a játékunkat, például Windows, MacOS, Linux konzolokra, valamint mobileszközökre is. Ez a játékmotor a C++ programozási nyelvet használja, nekem ezért nem volt szimpatikus és választottam inkább a Unity-t. Egyébként, ha valaki nem jártas a programozásban, akkor az használhatja a visual scripting rendszerét, melyet Blueprint-nek hívnak. Nekem személy szerint a szerkesztői panelja bonyolultnak tűnt, mely az alábbi ábrán meg is tekinthető.



## **A Godot játékmotor**

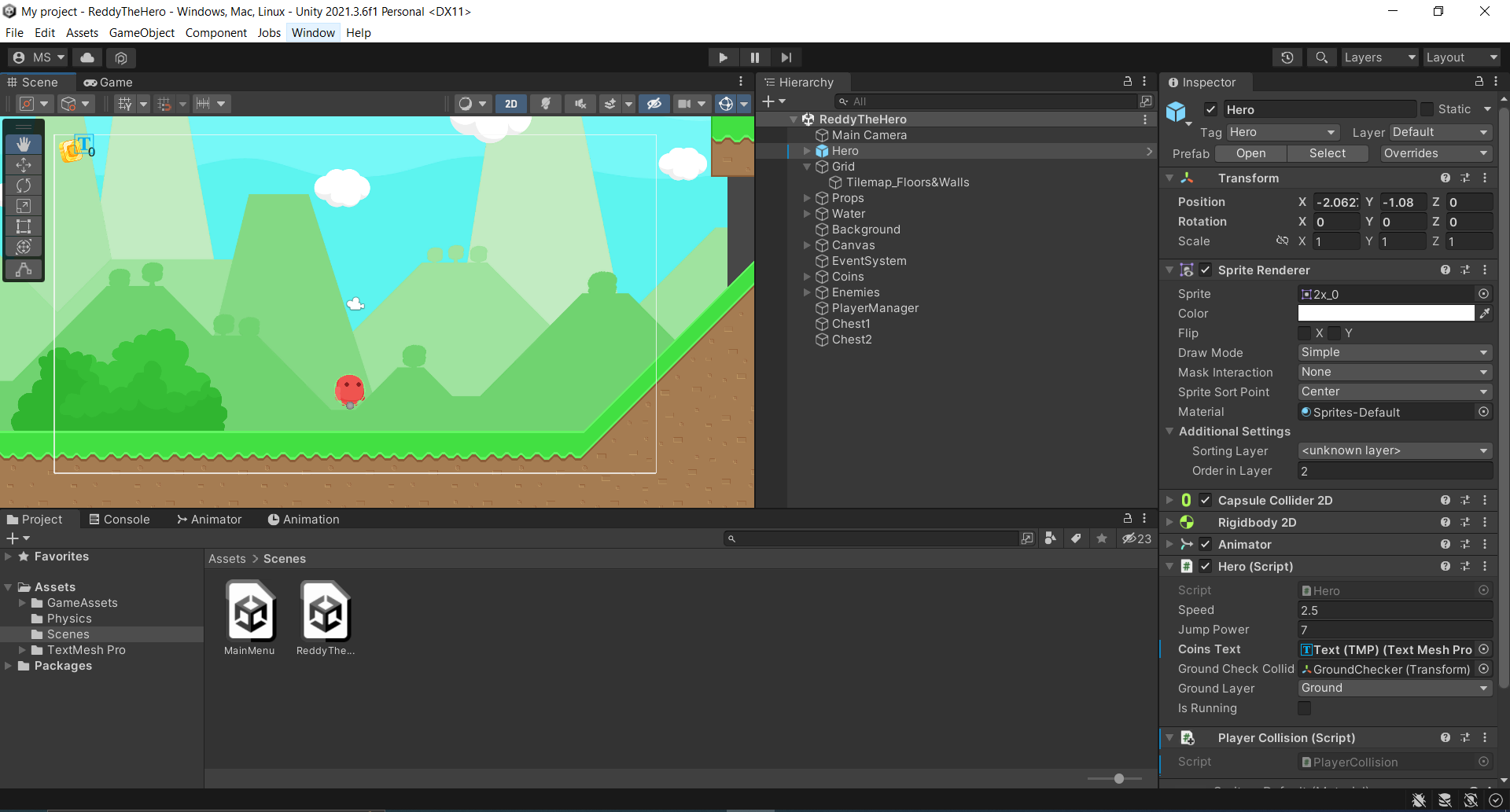
A Godot szintén egy nyílt forráskódú, platformfüggetlen játékmotor, amely MIT licensz alatt lett kiadva. A fejlesztőkörnyezet Linux-on, macOS-en és Windows-on fut. A Godot-tal mobil-, PC- és webes játékokat is lehet fejleszteni. Saját nyelve a GDScript, egy Python-hoz hasonló magas szintű dinamikus programozási nyelv, de a fejlesztett játékokat C#-ban is meg lehet írni. A kódolás mellett elérhető itt is az úgynevezett *visual scripting*, ahol egy grafikus felületen lehet kódot szerkeszteni. A Godot szerkesztője megtekinthető az alábbi ábrán.

## **A Unity játékmotor**

A Unity egy olyan játékmeghajtó motor, amellyel egyszerre lehet fejleszteni PC-re, MacOS-re, iOS-re, Android-ra, Wii-re valamint next-gen konzolokra (PlayStation, Xbox). A Unity Technologies fejlesztette, és adta ki 2005-ben. Sajnos nem nyílt forráskódú, viszont rendkívül alapos és jól használható dokumentációval rendelkezik. Három féle nyelven tudjuk programozni a játékunkat, ezek a C#, a JavaScript, valamint a Boo, de lassan kezdi bevezetni a visual scripting megoldást is. A Unity 25 célplatformot támogat, ezáltal az általam bemutatott game engine-ek közül a legtöbbet. Alapvetően ingyenes, viszont csak akkor, ha nem érjük el a vele készített játékkal az évi 100000$-os bevételt. Az ingyenes licensz viszont tökéletes arra, hogy a szárnyainkat próbálgassuk, és egyáltalán megnézzük, hogy mire vagyunk képesek.

### A Unity szerkesztője

Amint megnyitjuk a Unity-t, a Unity Hub ablak fogad minket. Itt tudunk projektet létrehozni, frissítéseket letölteni, valamint a Unity-t mint game engine-t jobban megismerni a „Learn” fül alatt. Miután létrehoztuk a projektünket, a Unity szerkesztője fogad majd minket. A Hierarchy ablak azt a célt szolgálja, hogy felsorolja azokat a GameObject-eket, amelyek az aktuális Scene-en megtalálhatóak. A Project ablaknál találhatóak az Asset-ek, amelyeket az Asset Store-ból installálhatunk. Az Asset-ek a 2D-s vagy 3D-s modelleket, textúrákat, anyagjellemzőket tartalmazó fájlokat, háttereket, hangfájlokat jelentik, tulajdonképpen ezekből kreáljuk meg a játékunkat. Valamint megtalálhatóak még a C# szkriptek és az importált csomagok. A Console ablakon keresztül kommunikál a Unity a fejlesztővel, itt jelennek meg a Debug.Log() üzenetek, vagy a fordítási hibák. Az Animator és az Animation ablakok a GameObject-ek animálására szolgálnak. Megtalálható még az Inspector ablak, amely azt a célt szolgálja, hogy a kijelölt GameObject-hez kapcsolódó minden adat megtekinthető, változtatható legyen. Bal felül megtalálható a Scene ablak, ahol a játékunk objektumait, hátterét és a pályát szerkeszthetjük. A Scene fül mellett található a Game ablak, ami kizárólag azt jeleníti meg, amelyet a Scene ablakban található kamera objektum lát. A Unity szerkesztője megtekinthető az alábbi ábrán.



# **A Reddy The Hero játék bemutatása**

## **A játék leírása**

A szakmai gyakorlatom során szerettem volna megismerkedni a játékfejlesztés rejtelmeivel, ezáltal készítettem egy 2D-s platformer játékot, melynek a *Reddy The Hero* nevet adtam. A jaték elkészítéséhez az Asset Store-ból szereztem be egy ingyenes asset-et. A játék irányítása nagyon egyszerű, a A D Space és Shift gombokkal lehet irányítani Reddy-t, akinek a célja, hogy minél több érmét összegyűjtsön az ellenfelek kicselezésével. Ha nem tudja kicselezni az egyik ellenséget, vagy leesik a vízbe, akkor Reddy meghal, és a játékos újrakezdheti a játékot vagy kiléphet a főmenübe. A játék kissé kezdetleges még, mivel azt a célt szolgálja, hogy egy mesterséges intelligencia irányítsa majd a karaktert.

## **Reddy, a játékom főhőse**

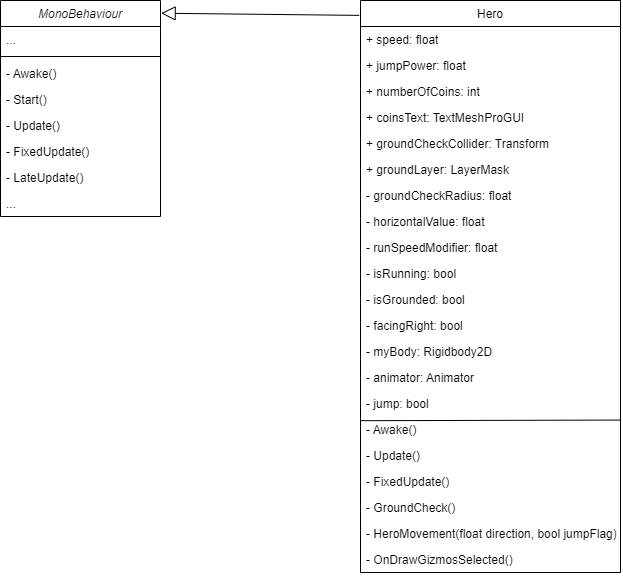
Kezdetben az irányítható karaktert szerettem volna megalkotni, hiszen ez a játék egyik legfontosabb része. Az első lépésben be kell importálni az Asset Store-ból szerzett ingyenes asset-et. Miután a beimportált asset-ből kiválasztottam a karakter 2D-s modelljét, a Hierarchy ablakra húzva megjelenik a Scene képernyőn. A Hero objektumhoz hozzá kellett adnom egy Rigidbody2D, azaz szilárdtest komponenst, így az objektumnak fizikai tulajdonságokat tudtam adni és ennek segítségével a Unity fizikai motorja is befolyásolhatta az objektum mozgását (tehát hatni fog rá a gravitációs erő, és különböző fizikai események). A Rigidbody komponensen belül rengeteg mindent be lehet állítani, én a Body Type-ot állítottam dinamikusra (azért, hogy a karakter ne lebegjen) és a Gravity Scale-t növeltem meg, hogy jobban hasson rá a gravitáció. Az ütközésvizsgálatot nagyon egyszerűen meg lehet oldani a Unity-ben. A Hero objektumhoz hozzá kellet adnom a CapsuleCollider2D nevezetű komponenst, valamint a talajhoz is, így a Hero objektum tud ütközni a talajjal, tehát nem fog a végtelenbe zuhanni. A Hero objektum, valamint a hozzácsatolt komponensek megtekinthetők az alábbi ábrán.

A képen szöveg, elektronika, megjelenítés, számítógép látható

Automatikusan generált leírás

### A Hero osztály

Ahhoz, hogy mozgásra bírjam a karaktert, készítenem kellett egy C# szkriptet, melynek a Hero nevet adtam. A Unity magától létrehozta a Hero osztályt, aminek a MonoBehaviour lett a szülő osztálya, innen fogja örökölni a metódusokat. A Hero osztály osztálydiagrammja az alábbi ábrán látható.



Először az Awake() metódus hívódik meg. Ez a Unity inicializáló függvénye, akkor hívódik meg, amikor a játék indításakor a Unity létrehozza azt az objektumot, amihez a program hozzá lett csatolva, ebben az esetben az irányítható karaktert. Egyetlen egyszer fut le az objektum élete során, tehát érdemes olyan parancsokhoz alkalmazni, amiket csak egyszer szeretnénk lefuttatni. Ilyen például az, amikor egy komponens típusú változóban szeretnénk eltárolni azok referenciáit. Esetemben ez a *myBody = GetComponent<Rigidbody2D>()* és az *animator = GetComponent<Animator>()* referenciák eltárolása.

Az utána következő függvény, az Update() is egy Unity által definiált függvény. Ez minden egyes képkockafrissítésnél meghívásra kerül. Ennek van 2 további változata, a FixedUpdate() és a LateUpdate() függvények. Az előbbi határozott időközönként hívódik meg, függetlenül a képkockafrissítéstől. Én az Update() és a FixedUpdate() függvényeket használtam. Az Update() függvény tartalmazza a *horizontalValue* inicializálását, a sebesség növelését amikor a karakter nem sétál hanem fut, valamint az ugrás parancsait is ebben a függvénybe helyeztem el. A FixedUpdate() függvényben hívom meg a HeroMovement() és a GroundCheck() metódusokat.

A továbbiakban a Hero osztály metódusait fogom részletezni.

A HeroMovement() metódus tartalmazza a karakter X és Y tengelyen való mozgatását, valamint azt is, hogyha a karakter földön van és megnyomta a SPACE billentyűt, akkor ugorjon. Alapjába véve, minden GameObject kap egy Transform nevezetű komponenst, ami megadja, hogy az adott objektum hol helyezkedik el a térben (esetemben síkban). Egy objektumot többféle módon mozgathatunk a programon belül. Egy objektum Transform komponensét elérhetjük a *gameObject.transform* osztályon keresztül, ezen belül több lehetőség is van, például a *position-*, a *localScale-*, a *velocity* tulajdonság vagy a Translate() metódus.

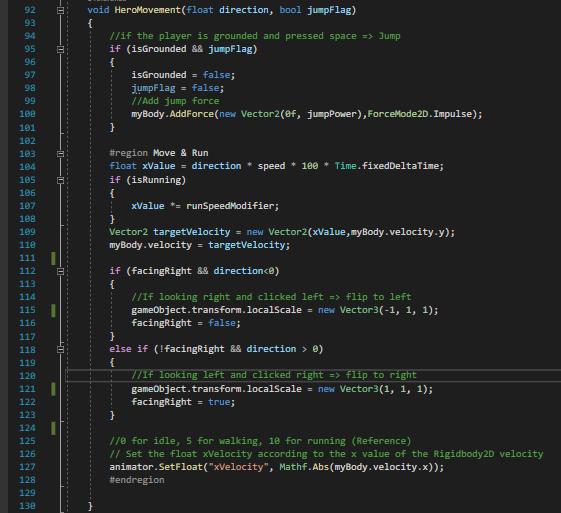
A *velocity* tulajdonságnak megadhatunk egy vektort, ami mentén a Unity áthelyezi a karaktert. Ha ezt mondjuk egy általunk kreált metódusban hajtjuk végre és meghívjuk a FixedUpdate() metódusban, akkor határozott időközönként végrehajtódik. Azonban, ha ezt a kódot egy lassabb és egy gyorsabb gépen is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

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

A *Move & Run* rész felett lévő if függvényt arra használtam, hogy ha a karakter érintkezik a talajjal (tehát az *isGrounded* változó igaz) és a játékos megnyomta a SPACE billentyűt, akkor a karakter az Y tengely mentén mozogjon, tehát ugorjon. Ehhez a myBody.AddForce() metódust használtam, melynek értékéül adtam egy vektort, amelynek az X értékét 0-ra az Y értékét pedig az általam deklarált és inicializált *jumpPower* változóra állítottam, valamint a ForceMode2D.Impulse() metódus segítségével egy löketet (impulzust) is adtam a karakternek, amelynek az erőssége a karakter tömegétől függ, melyet az Inspector fül alatt lévő Rigidbody2D komponensnél lehet beállítani.

Az animator.SetFloat() metódus az Animator Controller állapotgépének szól, ezekben a metódusokban megadott változók értékei szerint vált a Unity a karakter animációi között.

Ezt a metódust a FixedUpdate() függvényben hívtam meg, a HeroMovement() metódus az alábbi ábrán megtekinthető.



A GroundCheck() metódust, ahogy a nevéből is adódik, ütközésvizsgálat céljából hoztam létre. Először is létre kellet hoznom egy GameObject-et a Unity szerkesztőjében, amelynek a neve a GroundCheker lett és a Hero objektumon belül lett része a hierarchiának. Majd a szkriptben létrehoztam egy Transform típusú, *groundCheckCollider* nevezetű, egy LayerMask típusú *groundLayer* nevű, valamint egy float típusú *groundCheckRadius* nevezetű változót amelyeknek az értékeit a GroundCheck() metódusban a Collider2D típusú *colliders* tömbnek adtam át a Physics2D.OverlapCircleAll() metódus segítségével úgy, hogy az első értéknek (Vector2 típusú értéknek) a *groundCheckCollider*.position értéket, a másodiknak a *groundCheckRadius* értéket, a harmadiknak pedig a *groundLayer* értéket adtam meg. Az ezt követő feltételben pedig azt vizsgálom, hogy ha a *colliders* tömb hossza nagyobb, mint nulla (tehát a karakter ütközik egy másik ütközővel rendelkező objektummal), akkor az isGrounded változó értéke igaz lesz. A *colliders* tömbnek átadott változóknak az értékei az általam létrehozott GroundChecker objektum értékei lesznek, amelyet a Unity szerkesztőjében hoztam létre. A [SerializeField] azért van előttük, hogy tudjam őket szerkeszteni a Unity szerkesztőjében anélkül, hogy publikussá írnám át a láthatóságukat. A GroundChecker objektumot a Scene képernyőn a Hero objektum talpához raktam (és mivel a hierarchiában össze van kapcsolva a két objektum, ezért a Hero objektummal együtt fog mozogni), majd egyszerűen csak oda kellett húznom az Inspector menünél lévő Hero (Script) fülnél lévő Ground Check Collider nevű opcióhoz, majd a Ground Layer opciót átállítani „Ground”-ra. Ahhoz, hogy ez működjön, a szilárd talajnak szánt objektumok Layer-ét át kellett állítanom „Ground”-ra. Az animator.SetBool() szintén az Animator Controller állapotgépének szól, jelen esetben a „Jump” logikai értéket fogja hamissá változtatni. A GrondCheck() metódus az alábbi képen látható.

A képen szöveg látható

Automatikusan generált leírás

### A MainCamera objektum

Úgy is a karakter követésére bírhatjuk a kamerát, hogy egyszerűen ráhúzzuk az objektumra, így létrehozzuk a karakter és a kamera között a szülő-gyerek kapcsolatot a hierarchiában. Ekkor a kamera transform értékeit átírva meg tudjuk adni, hogy mennyire legyen eltolva a karakterhez képest. Kezdésnek megfelel, de ha amikor a karakter már komplexebb mozgásokat végez, akkor célszerűbb egy C# scriptet írni.

A CameraFollow szkript a játék kamerájának a mozgásáért felel. Létrehoztam egy *Transfrom* típusú *target* nevű változót, amely majd egy referenciát fog eltárolni a MainCamera objektumról. Definiáltam még egy *Vector3* típusú *offset* nevezetű változót a kilengés miatt és egy *float* típusú *smoothFactor* nevű változót is az egyenletes követés miatt, melynek 1 és 10 közötti intervallumot határoztam meg, amit a Unity szerkesztőjében tetszés szerint lehet állítani.

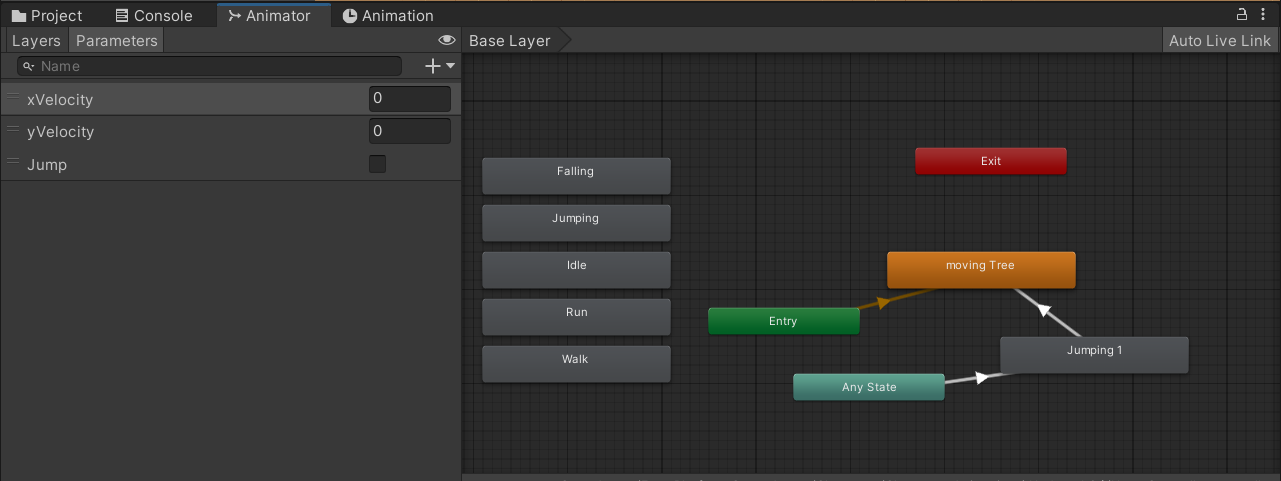
Készítettem egy Follow() metódust, melyben a kamera mozgását programoztam le. Megszabtam a kamerának az X és Y tengelyen lévő maximumát és minimumát (azaz hogy meddig követheti a játékost a képernyőn), amelyet a Mathf.Clamp() metódus segítségével értem el. Az egyenletesség állíthatósága érdekében pedig a Vector3.Lerp() metódust hívtam segítségül, amely lineáris interpolációt végez két megadott pont között, ezzel elérve a kívánt egyenletességet. a CameraFollow osztály az alábbi képen megtekinthető.

A képen szöveg látható

Automatikusan generált leírás

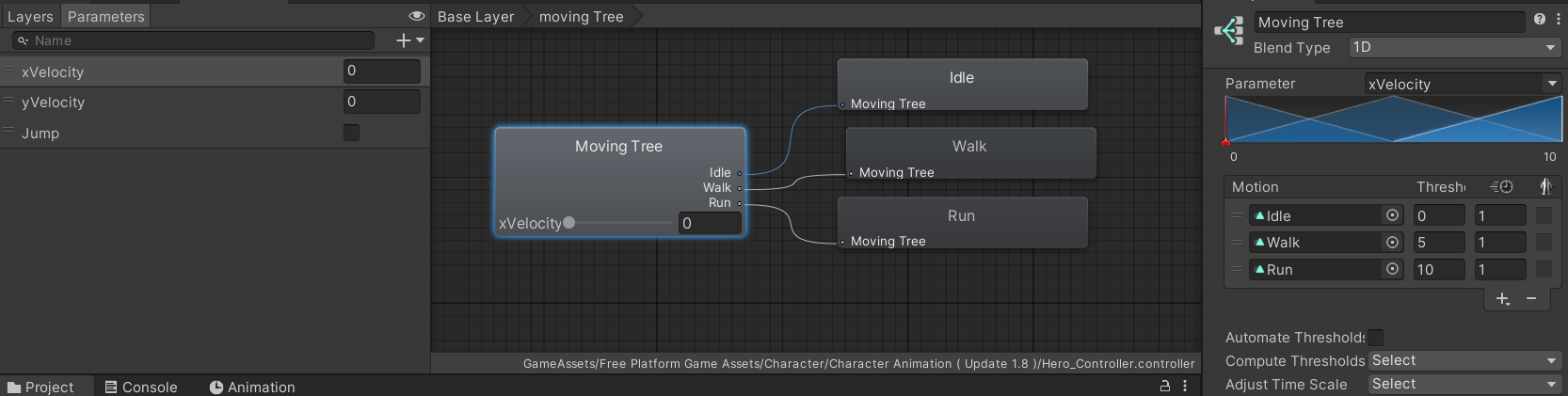
### Reddy animációi

Anélkül, hogy programot írnánk, a Unity képes animálás segítségével mozgatni objektumokat. Erre jó példa a játékomban például az ugrás, futás landolás animációi, az ellenségek, érmék, ládák és a víz animációi. Ezeket az animator controller valósítja meg, melyre gondolhatunk úgy, mint egy állapotgépre. Vannak állapotok, melyek egy-egy animációt reprezentálnak és vannak állapotátmenetek, melyek a végpontjaikban található animációkat felhasználva váltanak az egyik animációról a másikra. Ez a váltás történhet automatikusan, vagy paraméterek, küszöbértékek megadásával is. Az alábbi ábrán látható a Hero objektum animator controller-e.



Az *Entry* állapot a belépési pontot, az *Exit* állapot pedig a kilépési pontot jelenti. A *moving* *Tree* állapot pedig az alap állapot, amely egy *Blend* *Tree* melyben szerepelnek a bal oldalon látható *Idle*, *Walk* és *Run* állapotok, amelyeket az *Animation* fül alatt lehet létrehozni *Sprite*-ok segítségével. Az *Any* *State* állapothoz tartozó animációk bármelyik állapot után aktiválódhatnak, ezért is kapcsolódik hozzá a Jumping, amely szintén egy *Blend* *Tree*. Az Animator ablakon belül a *Parameters* fülnél lehet változókat definiálni, a *Layers* fül alatt pedig rétegeket létrehozni.

A *Blend* *Tree*-t állapotgép helyett használtam annak érdekében, hogy különböző küszöbértékek megadásával váltakozzanak az *Idle*, a *Walk* és a *Run* animációk (ezek a *moving* *Tree*-ben láthatók), valamint az ugrásnál is ugyan ezt a szisztémát követtem. A *moving* *Tree* az alábbi ábrán megtekinthető.

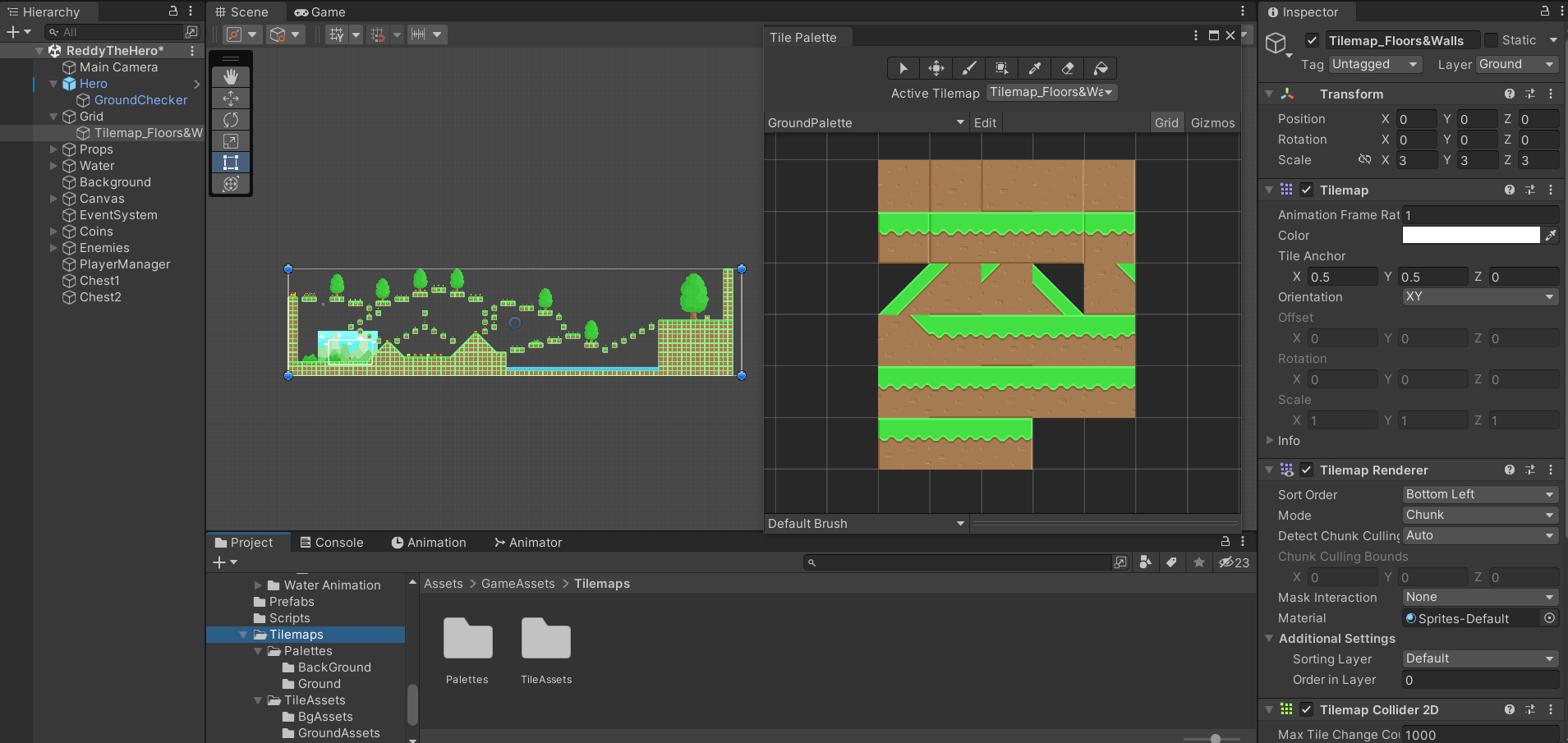


## **A környezet, az érmék és az ellenségek elkészítése**

A következő lépésnek a környezet megalkotását gondoltam, hiszen az elkészített animációkat itt lehet igazán kipróbálni és finomhangolni. Magát a pályát a *Tile* *Palette* segítségével tudtam könnyedén elkészíteni, amely a Unity egyik beépített Tilemap készítő egysége. Ezután a hátteret készítettem el, amely követni fogja a kamera mozgását, majd végül a dekorációkat, érméket, ládákat, az ellenségeket és a vizet, mint animációt készítettem el.

### A Grid objektum, valamint a Tile Palette használata

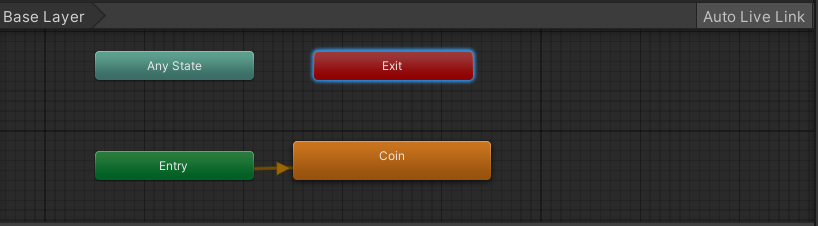
A *Tile* *Palette*-et úgy tudjuk megnyitni, hogy a Windows fül 2D opciójánál kiválasztjuk azt. Ezután be kell importálni azokat az *asset*-*eket*, amelyekkel a pályát fogjuk elkészíteni, ekkor készítünk egy palettát melyet én külön mappába mentettem el, valamint az ahhoz tartozó *asset*-*eket* is. A Grid objektum mivoltját a Scene képernyőn fogjuk *észrevenni*, amikor is cellákat kreál rajta. Ez az objektum lesz a szülője a *Tilemap\_Floors&Walls* palettámnak. A palettán én a celláknak a nagyságát a háromszorosára növeltem, hogy kevesebb blokkot használjak a pálya elkészítéséhez, valamint a látványvilág is sokkal szebb így. A blokkokat egyszerűen ki kell választani a palettából, és már el is tudjuk készíteni a pályánkat. Persze vigyázni kell, hogy a megfelelő blokkot válasszuk ki, különben esztétikailag nem lesz a legszebb a pályánk. Miután elkészítettem a pályát, a *Tilemap-Floors&Walls* objektumhoz hozzá kellett adnom egy úgynevezett *Tilemap* *Collider*-t, ezután a rétegbeli sorrendjét kevesebbre kellett állítanom a *Tilemap* *Renderer* komponensnél, mint a Hero objektumét, különben eltakarta volna a karaktert. Ha sima BoxCollider2D ütközőt használtam volna, akkor a blokkok közé ragadt volna a karakter. Az elkészített pályára elhelyeztem pár darab dekorációt, amelyek hangulatosabbá tették azt. A palettát és magát a pályát az alábbi ábrán láthatjuk.



### Érmék, ládák meganimálása és számlálása

Szerettem volna felszedhető elemeket rakni a játékomba, melyeket a játékos felszedhet és pontokat gyűjthet vele. Ilyen felszedhető elem az érme és a láda. A kettő között a különbség abban rejlik, hogy ha érmét szedünk fel akkor csak eggyel növeli a számlálót, de ha ládát akkor pedig tízzel. Ennek eredményeképp a ládákat nehezebben megközelíthető helyekre raktam, mivel többet érnek, ha felszedjük őket, mintha érméket szednénk fel.

Az animálásukat szintén az animator controller állapotgépének a használatával oldottam meg annyi különbséggel, hogy sima állapotgépeket használtam Blend Tree-k helyett, mivel azt akartam elérni, hogy amíg fut a játék folyamatosan fusson az animációjuk. Az alábbi ábrán látható a Coin objektum animator controller-e*.*



Ahhoz, hogy fel lehessen őket szedni, egy BoxCollider2D komponenst kellet az objektumaikhoz hozzáadnom, ahol az *Is* *Trigger* értékét igazra kellet állítanom, hogy a karakter ne akadjon meg bennük. Valamint egy C# osztályt is kellet készítenem hozzá, melynek a neve CoinCollecting, ahol egy metódust kellet meghívnom, melynek a neve OnTriggerEnter2D(). Ebben a metódusban egy feltételt alkalmaztam, amelyben ha a Coin objektum ütközik egy olyan objektummal, amelynek a címkéje egyenlő a „Hero”-val, mint szöveggel, akkor a Hero osztályban defniált *numberOfCoins* változó értékét növelje meg eggyel, valamint semmisítse meg a Coin objektumot. A Chest objektumnál ugyan ez a metódus lett meghívva annyi különbséggel, hogy az érmék számát nem eggyel, hanem tízzel növeli. Az érmék kiíratása a *TMP* *Text* nevezetű *UI* típusú objektummal lett megoldva, melyet a MainCamera objektum bal felső sarkához igazítottam. Ennek az objektumnak a referenciája a Hero osztályon belül található meg. A CoinCollecting osztály definíciója az alábbi ábrán látható.

A képen szöveg látható

Automatikusan generált leírás

### A Háttér

A hátteremet az importált asset-ek közül választottam ki, egyszerűen csak behúztam a Hierarchy ablakba, és meg is jelent a Scene képernyőn. Beállítottam úgy, hogy teljesen fedje a MainCamera objektumot, valamint a rétegének a szintjét -4-re állítottam be, hogy ne fedjen le semmilyen objektumot.

Ezután készítettem egy FollowTheCamera osztályt, melyben definiáltam két darab float típusú változót, melyeknek a *camerax* és a *cameray* neveket adtam. A LateUpdate() függvényben elértem, hogy a Background object folyamatosan kövesse a kamerának a mozgását a Camera osztály *transform* tulajdonságát felhasználva. A FollowTheCamera osztály definíciója az alábbi képen látható.

A képen szöveg látható

Automatikusan generált leírás

### Az ellenségek, valamint a víz animációjának az elkészítése

A mozgó buzogány és a víz animációját is az animator controller segítségével készítettem el. Kétféle ellenség van, egy lebegő tüskés vasdarab (a Mace objektum), valamint egy mozgó buzogány. A Mace objektum animációját egy C# osztállyal oldottam meg, amely annyit csinál, hogy az Y tengely mentén mozgatja az objektumot. A mozgó buzogány (a MovingMace objektum) animációja annyiból áll, hogy az Animation ablakban felvettem, ahogy az X tengely mentén mozgatom, és az Animator ablakban pedig egy egyszerű állapotgéppel megoldottam, hogy a játék folyamán folyamatosan mozogjon. A víz animáció helyzetében ugyan az a szisztéma, az Animation ablakban készítettem egy animációt, amely az animation controller segítségével a játék menete alatt folyamatosan futni fog. A Mace osztály definíciója az alábbi ábrán megtekinthető.

A képen szöveg látható

Automatikusan generált leírás

Ahhoz, hogy a karakter meghaljon, ha összeütközik az ellenségekkel, készítenem kellet egy PlayerCollision nevezetű osztályt, amelyben az OnCollisionEnter2D() nevezetű metódus segítségével megvizsgáltam, hogy ha a karakter egy olyan objektummal ütközik, amelynek a címkéje az „Enemy” nevet viseli, akkor a Hero objektumot inaktívvá tegye. Ezt a scriptet a Hero objektumhoz kellet csatolnom, valamint az ellenségek címkéjét át kellett állítanom Enemy-re. A PlayerCollision nevezetű osztály definíciója az alábbi ábrán megtekinthető.

A képen szöveg látható

Automatikusan generált leírás

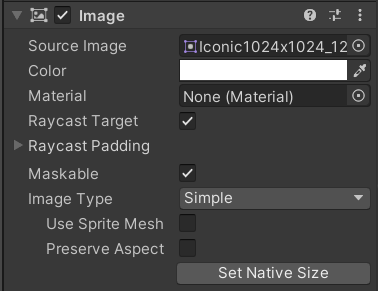
## **A főmenü, valamint a felhasználói felület**

Szerettem volna a játékomhoz elkészíteni egy egyszerűbb főmenüt, valamint egy felugró felhasználói felületet amikor a játékos meghal.

### A Főmenü

Ahhoz, hogy elkészítsem a játékom főmenüjét, készítenem kellett egy teljesen új Scene-t, ahol szintén készítettem egy hátteret, valamint egy palettát. A Unity-nek a beépített UI elemeit használtam fel a MainMenu nevezetű Scene elkészítéséhez. Amikor készítünk egy UI elemet, akkor a Unity automatikusan kreál egy EventSystem nevű objektumot, amely az eseményeket kezeli, valamint készít neki egy Canvas nevezetű szülőt a hierarchiában, ami igazából egy vászon amire illeszkedni fognak a UI elemek.

A Text és a Button elemeket használtam fel, a Button elemhez pedig írtam egy MainMenuController nevű osztályt, amelyben egy általam írt Play() metódus található. A UnityEngine-nek a SceneManager osztályát használtam, azon belül a SceneManager.LoadScene() metódust, hogy a „ReddyTheHero” nevű Scene-t be tudjam tölteni. Ha rámegyünk a PlayButton objektumra, akkor az Inspector ablaknál van egy olyan opció, hogy OnClick(). Igazából ez egy metódus, és itt be tudjuk állítani, hogy hogy viselkedjen kattintásra a gomb. Egyszerűen kiválasztjuk azt az osztályt, amelynek a metódusát szeretnénk használni és kész is. A Button objektumnak be is állíthatunk egy képet a Source Image opciónál, és még tetszetősebb lesz a játékunk.

A képen szöveg látható

Automatikusan generált leírás

### A GameOverPanel objektum

Ennek az objektumnak a létrehozásához két darab Button UI elemet és egy darab TMP Text elemet használtam fel. Ahhoz, hogy csak akkor jöjjön elő a GameOverPanel, amikor a játékos meghalt, készítenem kellet egy külön PlayerManager objektumot és hozzá egy PlayerManager osztályt is.

A PlayerManager osztályban definiáltam egy isGameOver nevű logikai változót, valamint egy gameOverScreen nevezetű GameObject típusú változót, amely majd a GameOverPanel objektum referenciáját fogja eltárolni. Az Awake() függvényen belül beállítottam az isGameOver változó értékét hamisra, az Update() függvényben pedig egy feltétellel megvizsgáltam, hogy ha az isGameOver változó értéke igaz, akkor a GameOverPanel objektumot tegye aktívvá. Ezen kívül készítettem a két Button objektumnak két darab metódust, egy ReplayGame() és egy GoBackToMenu() nevezetű metódust, amelyekben a SceneManager segítségével értem el, hogy ha a ReplayButton gombra kattint a játékos akkor az aktuális Scene-t töltse be, ha pedig a MainMenuButton gombot nyomja meg, akkor pedig a MainMenu Scene-t töltse be. A PlayerManager osztály definíciója az alábbi ábrán látható.

A képen szöveg látható

Automatikusan generált leírás

# **Összegzés**

A szakmai gyakorlatom során szerettem volna megismerkedni a játékfejlesztés rejtelmeivel, amely szerintem sikerült is. Nagyon tetszett a Unity, valamint a játékom készítésének a menete is. Mivel eléggé kezdetleges még a játékom, ezért a jövőben fejleszteni fogom a szakdolgozatomhoz, vagy esetleg a saját magam szórakoztatására.