Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Vecsei Gábor

Játékfejlesztés Unity3D keretrendszerrel

Önálló Laboratórium

KONZULENS

Hideg Attila

BUDAPEST, 2016

Tartalomjegyzék

1 Bevezetés **3**
2 A játékról **4**
2.1 Leírás 4
2.2 Erők 4
2.3 Játékosok 5
2.4 Irányítás 5
3 Megvalósítás **6**
3.1 Unity3D felülete 6
3.2 A játékosok 7
3.2.1 Komponenseik 8
3.2.2 A PlayerController Script 9
3.3 Kamera 16
3.4 Box Spawning 18
3.5 Game Manager 19
3.6 Energy Bar megvalósítás 20
4 Összefoglalás **22**
5 Project elérhetősége **23**

1 Bevezetés

Az Önálló laboratóriumi témám az a Játékfejlesztés Unity3D keretrendszerrel volt. Azért választottam ezt a témát, mivel mindig is nagyon érdekelt a játékfejlesztés, és igazából ezen keresztül ismerkedtem meg még általános iskolában a programozással. E mellett álló érvekből még nagyon sok van, de ezt kiemelném, hogy már foglalkoztam játékfejlesztéssel és ugyan ezt a keretrendszert alkalmaztam így nem volt ismeretlen a környezet. Ez azért is volt jó, mert így rögtön arra tudtam koncentrálni, amit meg szeretnék valósítani és nem kellett a környezettel ismerkednem.

2 A játékról

2.1 Leírás

Egy 2D local multiplayer platformer-t készítettem, ami azért local multiplayer mert nem a hálózaton keresztül lehet játszani, hanem elég hozzá csak egyetlen gép és annak a billentyűzetén tud két játékos játszani egymás ellen. A 2D platformer pedig egy olyan játék típus ahol egy sík térben a játékossal/játékosokkal kell menni jobbra, balra és ugrálni platformok között. Innen is jön az elnevezése, mivel a játék ilyen platformokból épül fel.

Az én játékomban két felhasználó irányíthatja külön kis embereit, akiknek az a feladata, hogy megszerezzenek egy labdát, ami a pályán található és minél tovább maguknál tartsák. Amelyik a legtovább tudja magánál tartani az lesz a nyertes a kör végén. Kétféleképp lehet nyerni. Az egyik az az, amikor a játék idő letelik, és akkor megnézzük, hogy kinél volt tovább a labda. A másik lehetőség mikor még nem telt le az idő, de az egyik játékosnál olyan sokáig volt a labda, hogy feltelítette az „energia szintéjt". Természetesen elég kicsi az esély, hogy pontosan ugyan annyi ideig legyen a labda mind a két félnél, de ez sem lehetetlen, így a döntetlen is a játék egyik kimenetele.

2.2 Erők

A játékban találhatók power up-ok amiket összeszedve a felhasználók különleges erőkre tesznek szert. Három féle erő létezik:

- Gyorsaság: Ha valaki ezt szerzi meg akkor egy adott ideig nagyon gyorsan tud mozogni a pályán.
- Magas Ugrás: Ezzel az erővel majdnem kétszer akkorát tud ugrani az emberke, és így feljuthat esetleg olyan helyekre ahova nem ér fel a másik.
- Játékos megfagyasztás: Ez szó szerint megfagyasztja a másikat és így nem tud mozogni, ugrani. Ennek is az a lényege, hogy magunknak adjunk egy kis előnyt a küzdelem során.

Az erőket a pályára leeső dobozokból tudjuk összeszedni. Így nem csak a labdát kell megszereznünk, hanem az ilyen leeső dobozokat is, mivel ha véletlenül a másik kaparintja meg akkor nagy előnyben részesülhet.

2.3 Játékosok

Két játékosunk van, akiket külön-külön lehet irányítani egy billentyűzeten. Ahhoz, hogy ne tarthassa a labdát nagyon sokáig magánál valaki, van még két opció, amit bevethet a másik ellen. Az egyik az a labda kiütése a másiktól. Ezt 10 másodpercenként teheti meg, szóval nem lehet folyton elütni a labdát, de van rá lehetőség, ha az ember kivárja a megfelelő alkalmat. A másik az pedig a labdának az eldobása. Ez először fura lehet, de gondoljunk bele, hogy mi van, ha sarokba szorítanak, és nem tudunk sehova menni. Ekkor csak fel kell ugranunk és eldobni a labdát nehogy kiüssék tőlünk és aztán utána tudunk menni.

2.4 Irányítás

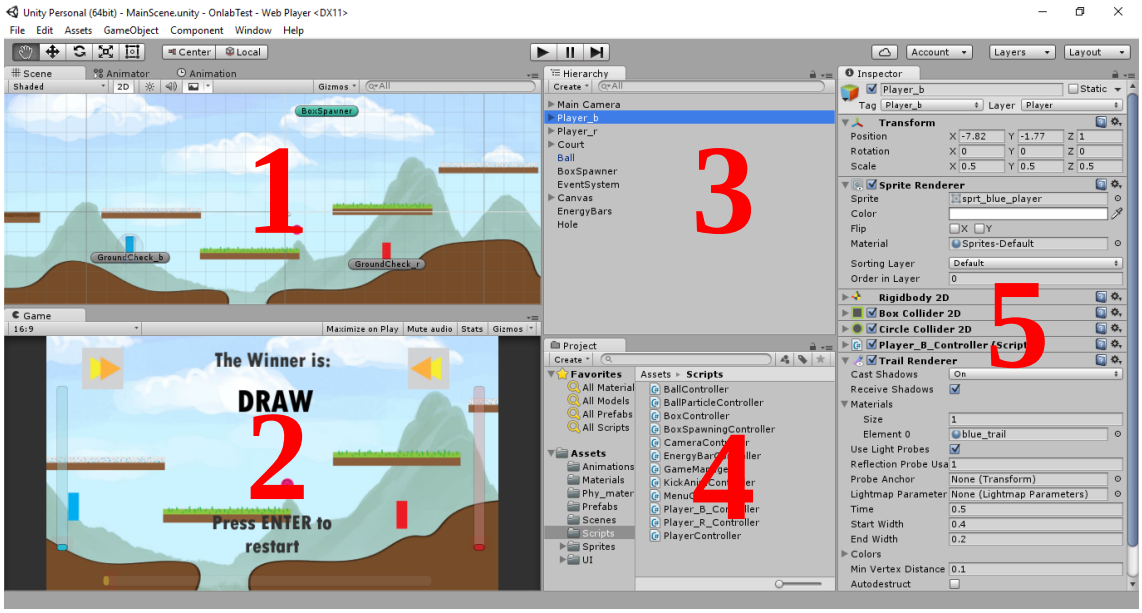
Játékos	Mozgatás	Labda eldobása / kiütése	Erő elhasználása
Kék játékos	W,A,S,D	E	Q
Piros játékos	I,J,K,L	U	O

3 Megvalósítás

Ebben a fejezetben fogok arról írni, hogy mit és hogyan valósítottam meg a már fent említett keretrendszer segítségével. Ennek a különlegessége az, hogy bizonyos megoldások több féle képpen is elérhetőek mivel kódon keresztül és a grafikus felületen is képes vagyok objektumokat, képeket stb. betölteni és a tulajdonságaikat megváltoztatni.

3.1 Unity3D felülete

Ez egy olyan engine vagy keretrendszer, amivel nagyon jó „minőségű” játékokat tudunk készíteni. A neve nem kell, hogy megtévesszen minket mivel nem csak 3D, de most már 2D támogatottsága is van. Én is utóbbit használtam mikor elkészítettem a játékomat mivel az egy 2D platformer. A unity3D felhasználó barát módon nyújt olyan szolgáltatásokat, amik ha nem lennének, akkor sokat szenvednénk azoknak az elkészítésével.



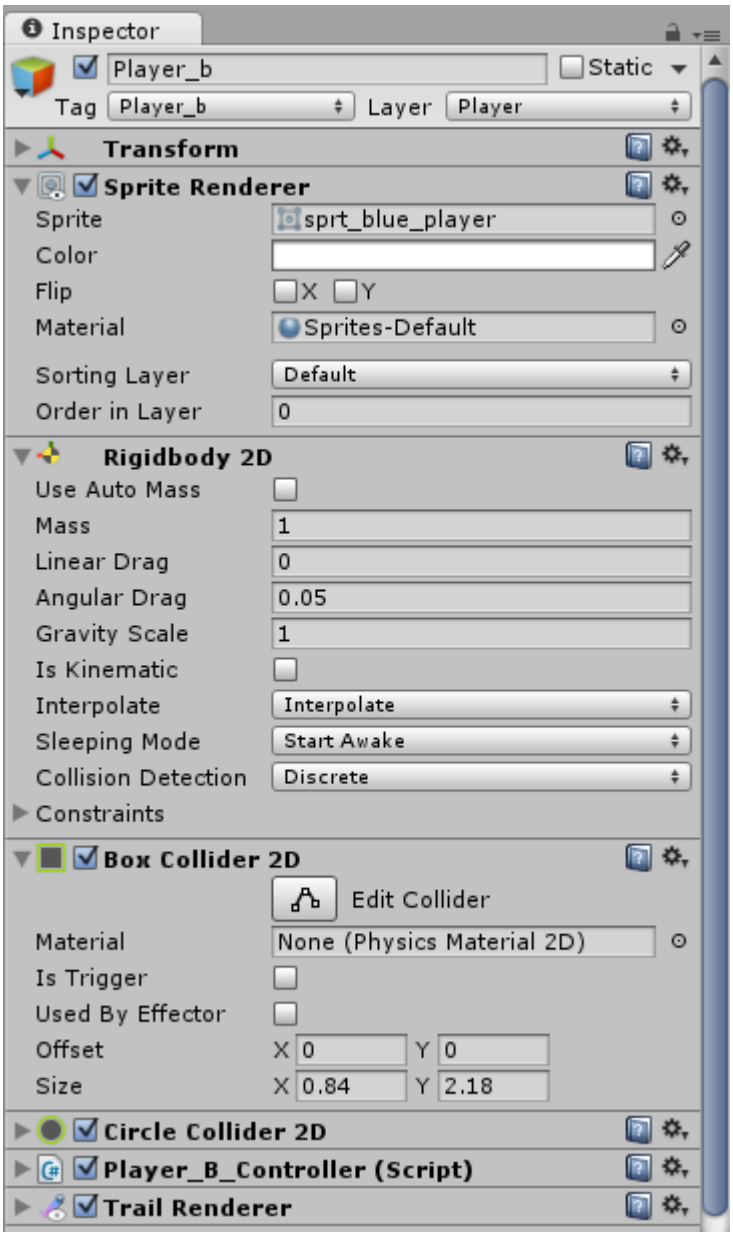
1. ábra A Unity3D grafikus felülete

- **1**: Ezt hívják Editor-nak. Itt lehet összerakni a pályát, és itt látszanak extra információk, amik esetleg a fejlesztés közben jól jöhetnek, de a játékban nem szívesen látjuk őket.
- **2**: Ez a Preview nézet. Itt nézhetjük meg, hogy hogy is fog kinézni a kész játékunk. A felhasználók ugyan ezt látják majd.
- **3**: Hierarhia. Ezen a tabon láthatjuk az elhelyezett GameObjecteket a Scene-n.
- **4**: Ez egy explorer avagy fájlkezelő. Itt láthatjuk hogy milyen folderek vannak a projectünkben meg milyen fájlok.
- **5**: Ez a tab egy Gameobject komponenseit tartalmazza (pl.: Rigidbody, Collider). Itt állíthatók be bizonyos tulajdonságok és itt kell hozzáadni a scripteket is.

3.2 A játékosok

Két játékosunk van a pályán, de mind a kettő ugyan azokkal a komponensekkel és scriptekkel rendelkeznek, mert nem különböznek semmiben, csak az irányítás módjában.

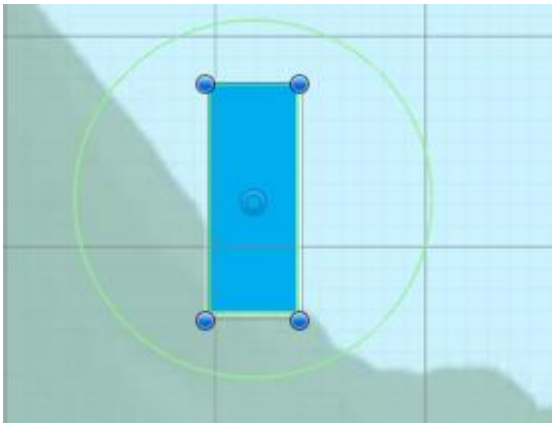
3.2.1 Komponenseik



2. ábra Player komponensei

Látszik, hogy több komponensből épül fel egy játékos, amik együttese adja meg azt, hogy hogy is fog viselkedni a GameObject maga a pályán. A *Transform* adja meg a pozícióját a koordináta rendszerben. A *Sprite Renderer* felelős a sprite (kép) megjelenítéséért. *Rigidbody2D*–t azért adtam hozzá, mert ezzel lehet megoldani, hogy a fizika beleszólhasson a GameObject viselkedésébe, tehát miután ezt hozzáadtam, a gravitáció hatott rá és elkezdett lefelé esni. A *Box Collider 2D* miatt detektálja az ütközést más Colliderekkel és így nem esik tovább. A Ground-nak is van Collider-e és így ezek hatnak egymásra és a játékos „fennakad” a ground-on. Van egy *Circle Collider*

2D is a játékosoknál, de ez azért van, hogy érzékelje a másik játékos jelenlétét. Végül egy *Script*-et látunk, amiben benne van a játékos irányítása és, minden egyéb funkciója.



3. ábra A kék játékos

3.2.2 A PlayerController Script

Ez az a script, ami megvalósítja a játékosok mozgását, és funkcióit. Ezt hozzáadva egy GameObject-hez könnyedén játékost készíthetünk. A *PlayerController.cs* az egy base class azaz a játékosoknál van egy *Player_B_Controller.cs* és egy *Player_R_Controller.cs.*

A fő funkciók tehát a PlayerControllerben vannak implementálva, csak mivel más és más gombokat használunk az irányításnál, így az ebben lévő függvényeket hívják meg a saját beállításaikkal, miközben hozzáférnek mindenhez ami a base class-ban van.

3.2.2.1 Kis kitérő a scriptekre

A Unity-ben készülő scriptekről annyit kell tudni, hogy vannak beépített függvények. Ezek közül a legfontosabbak:

```
void Start() {}
```

Ez akkor kerül meghívásra, amikor a játék elindul és aktiválódik a GameObject.

```
public void Update() {}
```

Az Update függvény minden frame-nél meghívódik.

```
void FixedUpdate() {}
```

Ezt a függvényt akkor kell használni amikor Rigidbody-van a GameObject-hez hozzárendelve, így akkor, amikor a fizikával kapcsolatos dolgot csinálunk. Tehát az Update helyett azt inkább itt kell megvalósítani.

3.2.2.2 A mozgatás

A kis kitérő után jöhet a tényleges megvalósítása a játékosoknak. Ebben a szekcióban a mozgatásról fogok írni. Ez azért volt egy kis kihívás, mert sokszor sokat tudnak segíteni az Unityben előre elkészített dolgok, de ez az én esetemben nem így volt, mivel szép és jó érzést kelő mozgatást akartam a játékosokhoz hozzáadni, miközben a fizikát is figyelembe kellett vennem a platformok különbözősége miatt. A jobb érzetet az adja, hogy nem úgy mozgatom a játékost, hogy erővel hatok rá, hanem, hogy annak a gyorsulását változtatom. Viszont ezzel a megoldással az a gond, hogy nem, ha így mozgatom, akkor nem hatnak rá a különböző fizikai anyagok, így például az, se ha egy platform-ot csúszóssá teszek. Ehhez kellett egy jó megoldást találnom. Szerencsére találtam is. Az Objektumoknak a pályán lehet különböző tag-eket adni és különböző rétegekre rárakni őket. Minden plaftorm a ground rétegen van, így tudjuk, hogy min lehet mozogni, miről lehet elugrani, viszont különböző taggel láttam egy a különböző viselkedésű platformokat (normál, csúszós, ragadós).

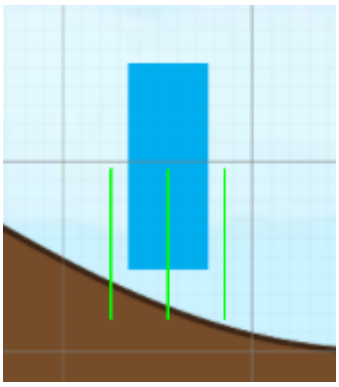
Lehet RayCastolni ami annyit tesz, hogy a végtelenségbe (vagy egy megadott távolságra) lövök egy sugarat, ami ha eltalál egy Collider-t akkor egy RayCastHittel tér vissza, amiből megtudhatom, hogy mi az eltalált tárgy. Így meg lehet tudni a tárgy tag-jét is.

Ezzel a módszerrel több részre lehet szedni a mozgást és van, hogy gyorsulás alapon mozgatom és van, hogy erő alapján mozgatom a játékost. Mindig csak azt kell figyelni, hogy éppen mi van alattunk. Ha nincs semmi vagy *NormalGround* tag-el rendelkezik a platform, akkor gyorsulás alapon mozgatom.

```
rb.velocity = new Vector2 (MoveX * MaxSpeed, rb.velocity.y);
```

Viszont ha valami olyan platform van alattam, ami rendelkezik egy physical material-al azaz valami eltérő anyagtulajdonsággal (csúszós, ragadós) akkor erő alapon van a mozgatás és így hat rá ez az anyagtulajdonság is.

Azt csináltam, hogy 3db raycastot indítottam útnak a játékosomtól. Az egyik a jobb oldalán van a másik a bal oldalán és van egy még középen is (ezeket meg is jelenítettem az editorban).



4. ábra Raycastok a játékosnál

Ha csak a jobb vagy a baloldal érzékel valamit, akkor tudhatjuk, hogy a platform szélén állunk, így erővel kell rábírnunk a játékost, hogy essen le (ha a gyorsaságát változtatjuk, akkor ez nem következik be).

```
if ((materialCheckBase.collider == null && materialCheckRight.collider != null)
|| (materialCheckBase.collider == null && materialCheckLeft.collider != null)) {
            rb.AddForce (new Vector2 (MoveX * MaxSpeed, 0));
}
```

Ha sima platformon állunk, akkor nem kell erőhatás. Ezt csak a középső raycasttal vizsgáljuk. De ha a GamObject tag-ja *SlipperyGround* vagy *StickyGround* akkor már erőhatással kell bírnunk, hogy érvényesüljön a hatás.

```
else if ((materialCheckBase.collider != null) && (materialCheckBase.collider.tag ==
 "NormalGround")) {
            //Ha sima platformon állunk akkor nem kell erőbehatás
            rb.velocity = new Vector2 (MoveX * MaxSpeed, rb.velocity.y);
} else if ((materialCheckBase.collider != null) && (materialCheckBase.collider.tag
== "SlipperyGround")) {
            //Viszont ha csúszós platformon állunk akkor kell erő hatás, mivel így
érvényesül a Physics material
            rb.AddForce (new Vector2 (MoveX * MaxSpeed, 0));
} else if ((materialCheckBase.collider != null) && (materialCheckBase.collider.tag
== "StickyGround")) {
            //Viszont ha ragadós platformon állunk akkor kell erő hatás, mivel így
érvényesül a Physics material
            rb.AddForce (new Vector2 (MoveX * MaxSpeed, 0));
        } else {
            rb.velocity = new Vector2 (MoveX * MaxSpeed, rb.velocity.y);
}
```

Ami ezt valósítja meg függvény azt a FixedUpdate-ben kell meghívni mivel fizikával kapcsolatos dolog történik a metóduson belül.

3.2.2.3 Ugrás

Az ugrás egyszerűen úgy oldottam meg, hogy figyeljük, hogy történt e gomb lenyomása (Update-ben) ami az ugrás kiváltja, és ha igen akkor ugorhat is a játékos. Persze csak akkor, ha nincs a levegőben, hanem a földön tartózkodik. A gomb lenyomását ez a függvény figyeli, amit a Player_B_Controller.cs/Player_R_Controller.cs-ben hívunk meg a saját input string-ünkkel:

```
public bool IsJumping(string inputButton){
        bool jump = Input.GetButtonDown (inputButton);
        return jump;
}
```

A FixedUpdate-n belül pedig megnézzük, hogy teljesülnek e a kritériumok és ha igen akkor el is ugorhatunk a földről. Az isGrounded változó True, ha a földön tartózkodik a játékos.

```
isGrounded = Physics2D.OverlapCircle (groundcheck.transform.position, 0.3f
, whatIsGround);

//Ugrik a játékos
if (jumping == true && isGrounded == true) {
   rb.AddForce (new Vector2 (0, JumpForce));
   //Ne tudjon még 1X ugrani
   jumping = false;
}
```

3.2.2.4 A labda megszerzése

Azt is kéne érzékelni, ha a labdát megszereztük, mivel amíg nálunk van, a labda addig egy számláló folyton számol, felfelé ami alapján el tudjuk dönteni, hogy ki a nyertes.

A labdának is van egy *Circle Collidere*, hogy ne essen le a földről, és ezt felhasználva tudjuk nézni az interakciót a játékosokkal. A Uniyben van egy függvény

```
void OnCollisionEnter2D(Collision2D coll){}
```

Amivel érzékelhetjük, ha ütközés történt és a *coll*-on keresztül megtudhatunk információkat az ütközésről és a GameObjectről amivel ütköztünk. Ez a függvény akkor hívódik meg, ha ütközés történik.

Ezt kihasználva nézzük a GameObject tag-jét amivel ütköztünk, hogy az a Ball e és ha igen akkor megsemmisítjük a Ball-t hogy ne legyen a pályán, jelezzük, hogy nálunk van a labda így elindulhat a számláló és létrehozunk egy particle-t ami azt jelzi, hogy a labda nálunk van.

```
void OnCollisionEnter2D(Collision2D coll){
        //Ha a labdával ütközünk
        if (coll.gameObject.tag == "Ball") {
            Destroy (coll.gameObject);
            iHaveTheBall = true;
            GameObject go;
            //Létrehozunk egy particle-t ami azt jelzi, hogy kinél vana  labda
            go = Instantiate (ballParticle, transform.position, Quaternion.identity
) as GameObject;
            //Beállítjuk, hogy a gyereke legyen a játékosnak így a pozíciójuk meg f
og egyezni, és együtt mozognak majd
            go.transform.parent = this.transform;
        }
}
```

3.2.2.5 Labda eldobása és kiütése

Az eldobást és a kiütést két külön függvényként valósítottam meg. Lehet, hogy különbözőnek tűnnek, de nagyon sok mindenben hasonlítanak. Szinte ugyan azokat a változókat kell leellenőrizni mindegyiknél.

A labda eldobásánál meg kell nézni, hogy nálunk van e a labda mert ha nincs akkor úgyse tudunk semmit se eldobni. Viszont ha nálunk van, akkor simán eldobhatjuk. Ehhez viszont tudni kell, hogy merre nézünk, mert akkor arra az irányba kell hajítani egy erővel.

```
void ThrowTheBall(){
            GameObject go;
            if (faceingLeft) {
                go = Instantiate (ballGo, transform.position + new Vector3 (-
1, 0, 0), Quaternion.identity) as GameObject;
                go.GetComponent<Rigidbody2D> ().AddForce (new Vector2 (-500, 500));
            } else {
                go = Instantiate (ballGo, transform.position + new Vector3 (1, 0, 0
), Quaternion.identity) as GameObject;
                go.GetComponent<Rigidbody2D> ().AddForce (new Vector2 (500, 500));
            }
            Destroy (transform.Find ("BallParticle(Clone)").gameObject);
            iHaveTheBall = false;
    }
```

A kiütésnél is meg kell nézni, hogy nálunk van e a labda mert ha nálunk van akkor csak eldobni lehet, nem lehet kiütni a másiktól. Látható, hogy ez egy if-else–nek megfelel, sőt még az gomb is ugyan az, amit meg kell nyomni így egy másik függvény ellenőrzi le, hogy kinél van a labda és a szerint vagy kiüti, a labdát a másiktól vagy pedig eldobja. Ezt a függvényt a játékosok külön scriptében hívom meg a saját inputjukkal.

```
void KickTheBall(){
        GameObject go;
        GameObject go_kickBallAnim;
        kickTime = 0;
        canKick = false;
        //Ha kiütöttük akkor már nincs a másiknál a labda
        otherPlayer.GetComponent<PlayerController> ().iHaveTheBall = false;
        //Random erő az ütéshez
        Vector2 randomForce = new Vector2 (Random.Range (-
250, 250), Random.Range (400, 1000));
        Vector3 otherPlayerPos = otherPlayer.transform.position;
        //Most a labda ott fog teremni ahol a másik játékos van
        go = Instantiate (ballGo, otherPlayerPos + new Vector3 (0, 1f, 0), Quaterni
on.identity) as GameObject;
        //Az előzőleg kitalált erővel elütjük
        go.GetComponent<Rigidbody2D> ().AddForce (randomForce);
        //Lejátszuk az animációt ami a kiütést szemlélteti
        go_kickBallAnim = Instantiate(kickBallAnim, transform.position, Quaternion.
identity) as GameObject;
        //Meg is kell semmisíteni a jelző particle-t
        Destroy(otherPlayer.transform.Find("BallParticle(Clone)").gameObject);
    }
```

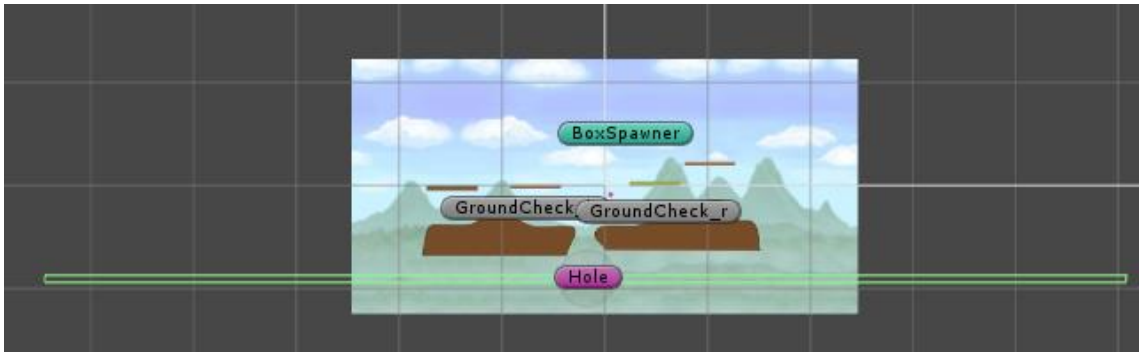
Amit az Update-n belül meg kell hívni:

```
public void KickAndThrow(string inputStr){
        if (Input.GetButtonDown (inputStr)) {
            if (iHaveTheBall) {
                ThrowTheBall ();
            } else if (!iHaveTheBall && canKick) {
                //Ha nincs nálunk a labda és jelezve van, hogy ki tudom ütni akkor
üssük ki a másiktól a labdát
                KickTheBall ();
            }
        }
    }
```

3.2.2.6 Leesés kezelése

Mivel a pályám olyan, hogy a végéről le lehet esni és középen van egy lyuk így valahogy azt az esetet is le kell kezelni, mikor leesik egy játékos vagy pedig a labda. A leesés érzékeléséhez létrehoztam egy új GamObjectet aminek neve *Hole* és ehhez csak egy nagy Collider tartozik. Mert mikor leesünk akkor ezzel fogunk összeütközni és innen fogjuk tudni, hogy kell valamit csinálni. A játék nehezítése miatt mikor leesik

valaki akkor utána csak X idővel éled újra, amikor is random leesik a pályára valahol. Így kap egy kis büntetést a figyelmetlenségéért.



5. ábra Hole és a Collidere

Amikor ezzel ütközik valaki akkor egy Coroutine-t indít el. Azért ezt használtam, mert így tudok csinálni valamit addig, amíg az az idő le nem telik, amit én a büntetésnek szántam. Ezen a függvényen belül generálok egy random x pozíciót ahol majd le fog esni a pályán, és amíg nem telik ez le addig kikapcsolom a SpriteRenderer komponensét, hogy ne látszódjon a pályán.

```
gameObject.GetComponent<SpriteRenderer> ().enabled = false;
```

Miután vége a várakozásnak akkor visszakapcsolok mindent az eredeti állapotába és újra mozoghat a pályán a játékos.

3.2.2.7 Power Ups azaz az Erők

Az erőknek nagyon fontos szerepe van a játékban mivel így előnyre tehet szert az egyik vagy másik játékos. Ezeket úgy tudja megszerezni, hogy a pályára leeső kis dobozokat összeszedi, és akkor kap egy valamilyen erőt. Ezt is annak a segítségével oldottam meg, hogy figyelem az ütközéseket (pont úgy, mint a labdánál). Amikor a játékos hozzáér, azaz összeszed, egy dobozt akkor a dobozt el kell tűntetni azaz, megsemmisítjük és ezen kívül meg kell tudni, hogy milyen erőt kaptunk. Ezt a

```
public string GenerateRandomPower(){}
```

végzi nekünk. Akár mennyire benne van a nevében, hogy random igazából súlyozva vannak az előfordulások.

```
if (rnd < 20) {
            powerUpImage.sprite = powerUpSprt_freezetime;
            return powerUps [1];
        } else if (rnd < 60) {
            powerUpImage.sprite = powerUpSprt_highjump;
            return powerUps [2];
        } else {
            powerUpImage.sprite = powerUpSprt_speed;
            return powerUps [0];
        }
```

A súlyozás azért volt szükséges, mert véleményem szerint valamelyik erő nagyobb értékkel bír, mint a másikak és akkor annak kisebb előfordulási valósínűséget adok. Ez az erő a *FreezeTime* ami a másik játékost megfagyasztja.

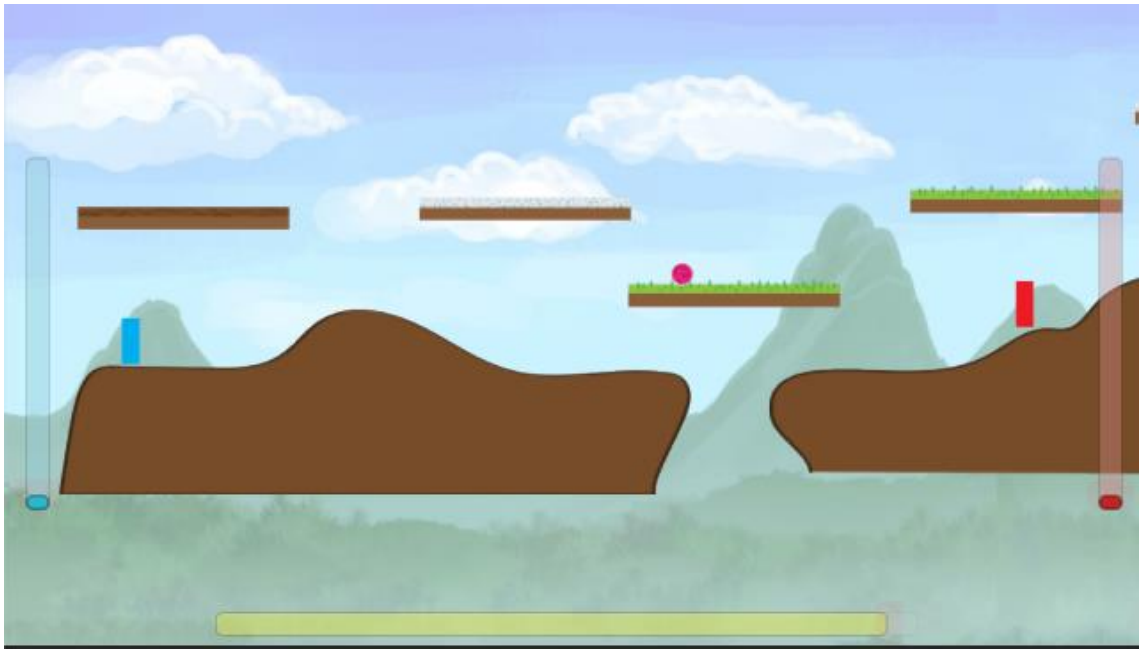
Miután van egy random power up-unk akkor ennek megfelelően megjelenik egy ikon, ami jelzi, hogy mink van. Ez egy egyszerű kép.

Ezután ha a felhasználó el akarja használni a megszerzett erőt, akkor meg kell nyomnia egy billentyűt amivel, aktiválhatja azt. Ha lenyomta, akkor egy másik függvény leellenőrzi, hogy mink van, és a szerint átállít bizonyos tulajdonságokat. Például ha Speed power up-unk van, akkor a MaxSpeed-et állítja át és beállítja azt is, hogy meddig érvényes ez mivel az aktiválással egy időben elindul egy visszaszámlálás, hogy meddig is lehet szuper tulajdonságunk (erőnként külön állítható). Miután letelik az idő azután minden resetelődik és visszakerül az eredeti állapotba.

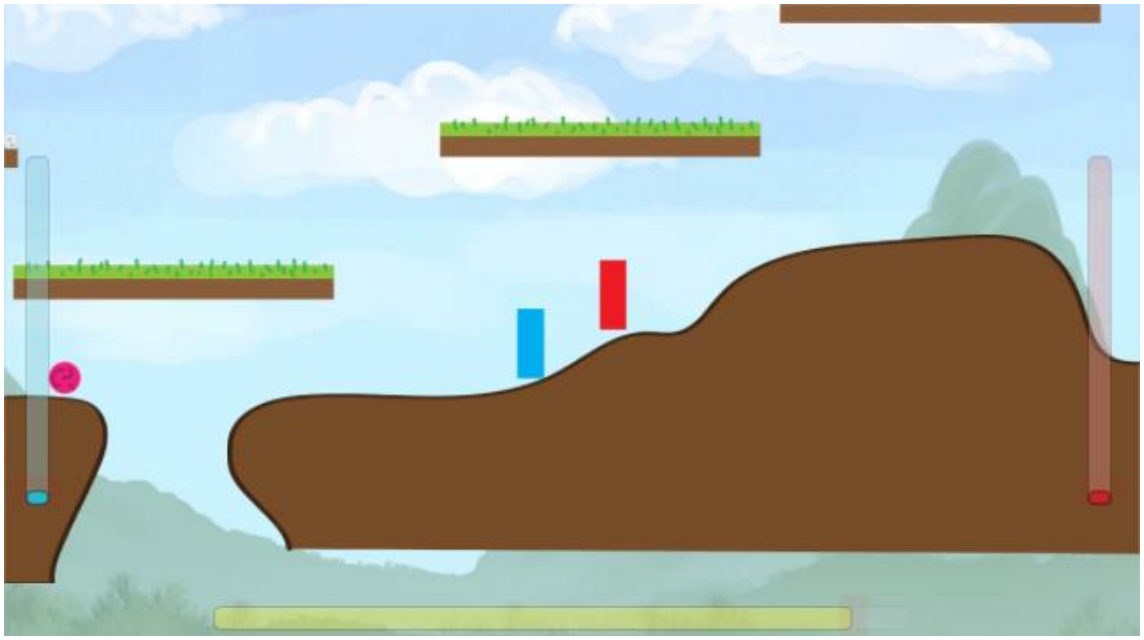
3.3 Kamera

A kamera nagyon fontos része a játéknak mivel ez kontrollálja azt, hogy mit is fogunk látni a játék során. Mivel ez egy többszereplős játék, mindenkinek látszódnia kell a kamera ablakon belül. Véleményem szerint unalmas lett volna egy olyan kamera, ami az egész pályát mutatja, de csak áll egy helyben és semmi dinamizmus nincs benne. Így az egyik ismert játék a Super Mario Smash Bros kameráját készítettem el, ami mindig annyit mutat, a játékból pont amennyit kell. Ehhez egy scriptet kellett készítenem, ami több GameObject pozíciójából számolja ki a kamera helyzetét és méretét. Ezt a scriptet hozzáadva a kamerához már egy olyan képet kaptam, ami mindig arra mozdul, amerre a játékosok vannak és mindenki látszik egyszerre, sőt még a méret

is úgy változik, hogy mindenki látszódjon. A következő képeken láthatunk erre példákat.



6. ábra Távoli játékos pozíciók esetén a kamera

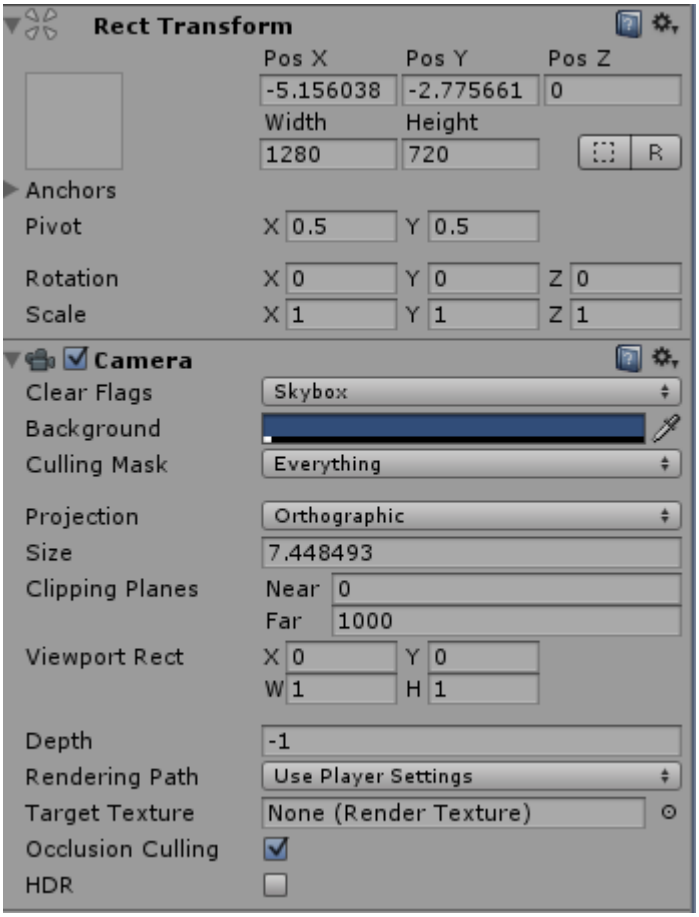


7. ábra Közeli játékos pozíciók esetén a kép

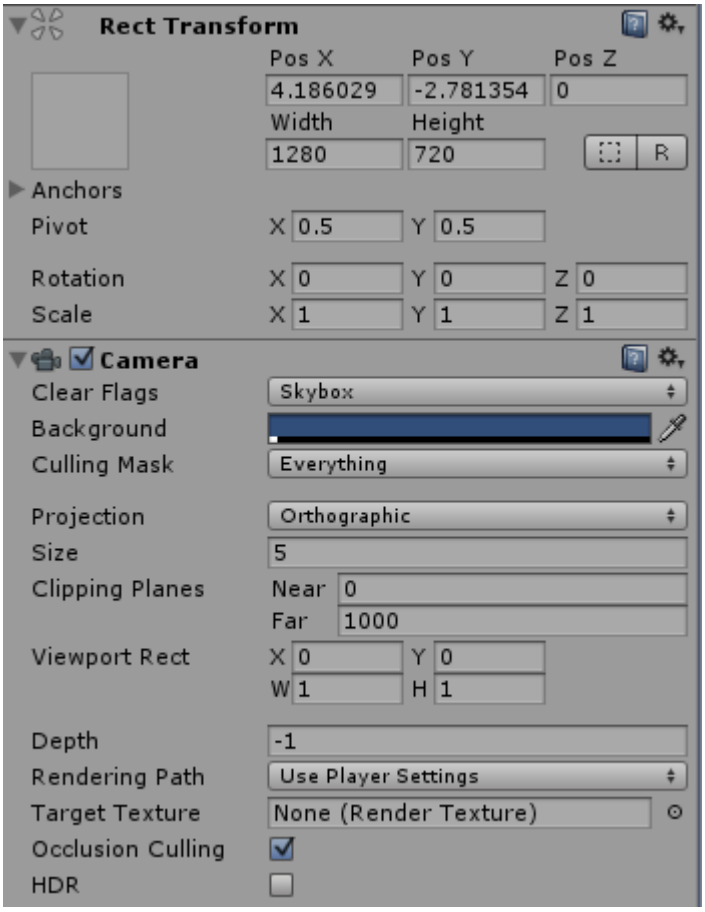
A már említett script-ben egy ArrayList-ben tárolom azokat, akiknek a helyzete számít a végső kalkulációban. A lényege az egésznek, hogy ki kell számolnunk a minimum és maximum X és Y értékeket és így vissza tudunk kapni egy négyszöget.

Ezután a Vector3.SmoothDamp segítségével a kívánt pozícióba mozgatjuk. Ez a függvény egy vektort egy másikba visz át finoman. És mikor már megvan hogy hol kell lennie, akkor a méretét is ki kell számolni, azaz az Ortokgrafikus méretét.

Előző képek alapján a kamera pozíciója és mérete is változott, ezt látjuk a következő képeken.



8. ábra Mikor távol vannak akkor nagyobb a mérete



9. ábra Közeli pozíció-nál kisebb a méret

3.4 Box Spawning

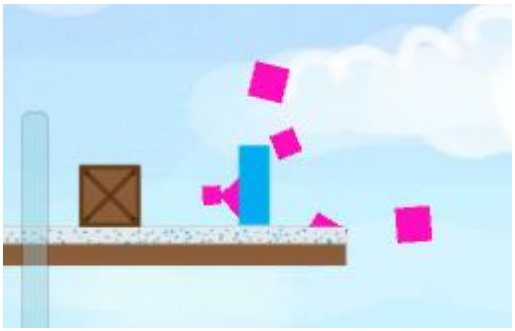
Ahhoz, hogy új ládák teremjenek a pályán egy új scriptet kellett elkészítenem, amit egy „láthatatlan” GameObjecthez adtam hozzá. Ez a pálya tetejénél van, azon az y pozíción ahonnan akarom, hogy majd a ládák leessenek. Ez a BoxSpwning.cs.

Ez is mint már egyszer említettem Coroutine-on alapszik. Ahogy elindul a játék, azaz meghívódik a GameObject Start metódusa, elindítok egy Coroutine-t ami megoldja

a ládák dobálását. A metódusom csak annyit csinál, hogy egy minimum és egy maximum érték között generál egy számot és annyi időt vár mielőtt ledobhatná a dobozt. A várakozás letelte után generál egy random x értéket ahol majd a box meg fog jelenni.

Most már mindenünk megvan, hogy „teremtsünk” egy ládát, amivel majd egy új power up-ot tudunk generálni.

```
float rndXPos = Random.Range (spwPosXMin, spwPosXMax);
Vector2 dropPos = new Vector2 (rndXPos, transform.position.y);
GameObject go = Instantiate (boxPref, dropPos, Quaternion.identity) as GameObject;
```



10. ábra Leesett doboz

3.5 Game Manager

A GameManager az az a script, ami figyeli a játékidőt és azt, hogy ki nyer a játék végeztével vagy akár közben is. Ezeken felül még kezeli a UI-t azaz ki is írja, hogy ki nyert.

Külön kell vizsgálnunk azt az esetet mikor letelt a játékidő, mert akkor össze kell hasonlítani, hogy kinél volt többet a labda. A másik eset az a mikor valaki feltelíti a saját enery bar-ját és így nyert még idő előtt.

Miután megvan a nyertek azt egy egyszerű függvénnyel kiiratjuk a képernyőre. Csak engedélyezni kell a szövegek megjelenését és be kell állítani a szövegeket.

3.6 Energy Bar megvalósítás

Az energy barok a képernyő szélén találhatóak és azt mutatják, hogy mennyi ideig volt nálunk a labda. Ennek egy nagyon egyszerű megvalósítást képzeltem el, amit meg is csináltam.

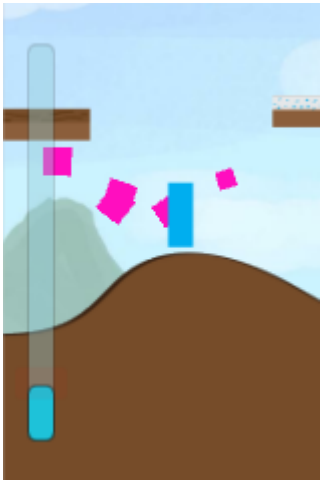
Először is létrehoztam egy slidert ami egy UI elem. Ezzel általában értéket lehet megadni (pl.: hogy miylen hangos legyen a zene). De most nem ezzel adtam értéket, hanem ennek adtam. Script-en keresztül beállítottam a maximális értékét, hogy ne kelljen a grafikus felületen sok minden állítgatni és, hogy nehogy inkonzisztens legyen (például növelem az elérhető értéket, de a slider maximumát nem változtatom akkor a slidernél hamarabb be fog telni, mint az ténylegesen megtörténne).

```
gameTimeBar.maxValue = GameManager.gameTime;
energyBar_r.maxValue = GameManager.maxBallHoldTime;
```

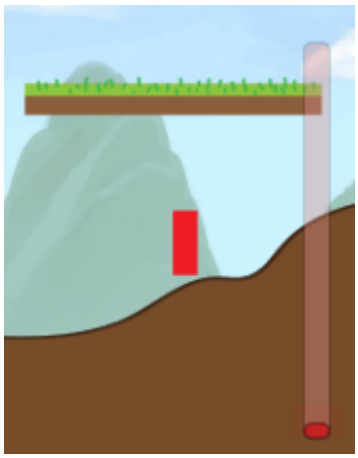
Ezután csak az Update-ben mindig „lekérdezzük” az éppen aktuális időt ameddig tartotta a labdát a játékos és beállítjuk a slider értékévé.

```
energyBar_r.value = player_r.GetComponent<PlayerController> ().BallHoldTime;
energyBar_b.value = player_b.GetComponent<PlayerController> ().BallHoldTime;
```

Most már ahogy növekszik a labda tartási idő úgy fog változni az úgynevezett energy bar is.



11. ábra Kék játékos Energy bar



12. ábraPiros játékos Energy bar

4 Összefoglalás

Számomra ez egy nagyon kedves feladat volt, hogy készítsek játékot. Az egyetemi teendők mellett azt mondanám, hogy még pihentető is volt. Tapasztalat szerzésnek tökéletes volt, mivel most úgy próbáltam meg elkészíteni a játékot, hogy átgondolom részletesebben, hogy milyen osztályokat és hogy kéne elkészítenem ahhoz, hogy jól működjön a programom és nem utolsó sorban, úgy ahogy én elképzeltem. Még szem előtt tartottam most azt is, hogy jól olvasható kódot készítsek, olyat, amit én is szívesen néznék egy ismeretlen projectben. Ez többé kevésbé sikerült a tapasztaltságomnak és a tapasztalatlanságomnak köszönhetően. Ahhoz hogy a saját magam által elvárt szintet teljesítsem, még jó pár játékot el kell készítenem.

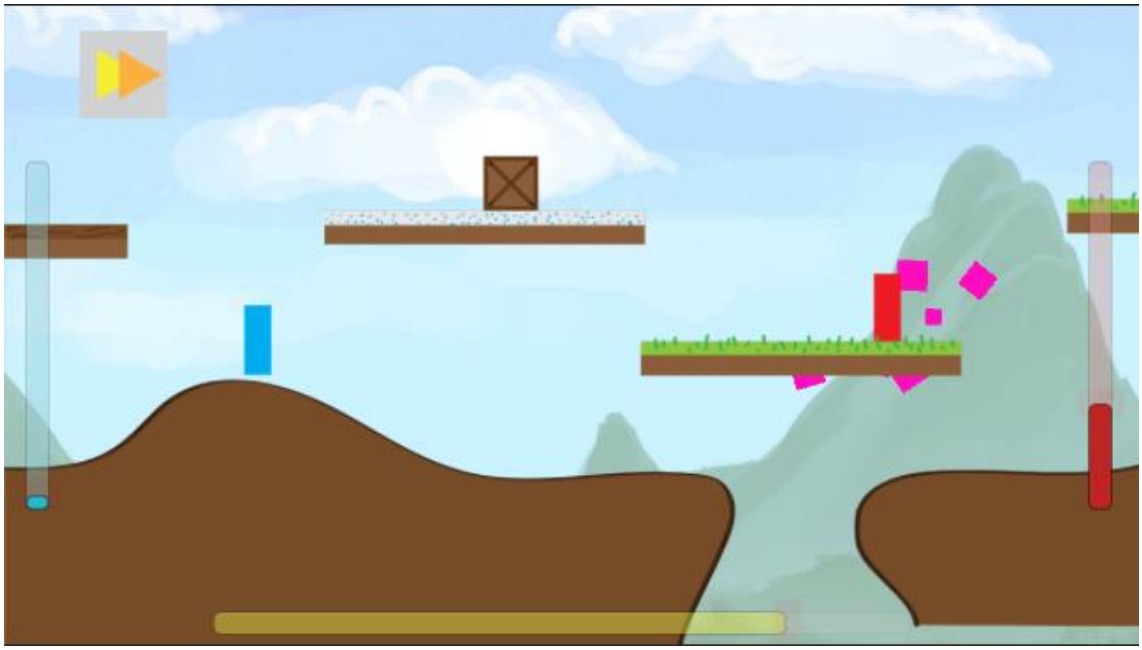
Mivel ismertem a felületet és C#-ban nagyon szeretek programozni így nem az okozta a legnagyobb kihívást, hogy ezeket elsajátítsam, hanem az hogy maradjon időm a félév összes többi teendője mellett, így ezt mondanám a legnagyobb kihívásnak.

Amit még ennek a projectnek köszönhetek az az, hogy megismerkedtem a verziókezelés rejtelmeivel, és aktívan használtam, feltöltöttem nem csak ezt a programomat, de másokat is.

5 Project elérhetősége

https://github.com/gaborvecsei/OnalloLaboratorium

13. ábra Projectem GitHub linkje



14. ábra Játékból képmentés