Szegedi Tudományegyetem
Informatikai Intézet

RPG játék fejlesztése Unityben

Szakdolgozat

Készítette:
Horváth Krisztián
informatika szakos
hallgató

Témavezető:
Dr. Nagy Antal
egyetemi docens

Szeged
2023

Feladatkiírás

A feladat egy olyan 2 dimenziós játék fejlesztése top-down nézetben, ami RPG elemeket tartalmaz továbbá egy szerkesztővel is rendelkezik, ahol a játékos saját pályát készíthet el, ahova a későbbiekben eljuthat. A feladat megvalósítása során törekedni kell a jól strukturált fejlesztésre, hogy a későbbiekben könnyen lehessen fejleszteni a projektet.

Tartalmi összefoglaló

- ***A téma megnevezése:***

RPG játék fejlesztése Unityben

- ***A megadott feladat megfogalmazása:***

A feladat egy 2 dimenziós top-down játék készítése, ahol a játékos saját karakterét kreálhatja meg, ellenfelekkel küzdhet és kazamatákba léphet be. A harcokból fegyver és gyógyital vásárlásra használható pénzt és fejlődéshez szükséges tapasztalatpontokat nyerhet. Továbbá a játékos elkészíthet egy saját kazamatát a beépített pályaszerkesztővel.

- ***A megoldási mód:***

A Unity játékmotor adta szerkesztő felület és függvénykönyvtárak használatával és a C# nyelvre kiadott csomagok segítségével készült el az alkalmazás.

- ***Alkalmazott eszközök, módszerek:***

A fejlesztéshez a Unity fejlesztőkörnyezet volt használva, C# programozási nyelvvel. A scriptek megírásához a Visual Studio IDE volt igénybe véve. A pályaszerkesztőhöz procedurális generálás volt alkalmazva.

- ***Elért eredmények:***

Az alkalmazás sikeresen elkészült, a karakter és pályaszerkesztő is használható.

- ***Kulcsszavak:***

Unity, C#, Cinemachine, procedurális generálás, JSON, NewtonSoftJson

Tartalomjegyzék

BEVEZETÉS

A számítógépes játékok a mai világban igen elterjedtnek mondhatók, így nem véletlen, hogy sok játék készül évente, akár számítógépre, akár a különböző konzolokra. Szakdolgozatomban egy 2D Top-Down RPG műfajú játék fejlesztését mutatom be, amelynek motivációja a korábbi játékokkal kapcsolatos élmények.

A mai világban megannyi 3D-s RPG játékkal játszhatunk, azonban úgy gondolom, hogy azok a régi értékek, amiket a 2D-s játékok képviselnek még ma is érvényesülnek. Hisz már egészen a 80-as évektől kezdve jelentek meg 2D-s játékok, és ezek között szerepeltek a top-down változatok is, amik letisztultak, egyszerűek ám mégis szórakoztatóak voltak. Ezek függvényében célom az volt, hogy egy az ezekhez hasonló klasszikus játékot hozzak létre.

Mivel tanulmányaim során még sosem készítettem játékot, ezért nem voltam túl tapasztalt a játékfejlesztésben, úgyhogy mindenképpen egy baráti fejlesztő környezetet választottam. Sok különböző játékmotor is rendelkezésre áll ilyenkor a fejlesztők számára, mint a Godot, Unreal, Armory és a Unity. Választásom azért esett a Unityre, mert népszerűsége igen kimagasló és megannyi módon segíti a játékkészítőt a szoftver elkészítésében, ám ennek ellenére kellő kihívást rejt.

A Unity szerencsére több programozási nyelvet is támogat, amivel elkezdhetjük fejleszteni saját játékunkat, ilyen például a C++, vagy a C#. A két nyelv közül, mivel már évek óta érdekelt a C#, jobbnak gondoltam az utóbbit választani.

Kezdésként több példaprogramot is megvalósítottam, hogy kipróbáljam a fő funkciók egyszerűsített változatát, mint a karakter mozgását, a harc rendszert, vagy az ellenfelek logikáját. Továbbá ez a Unityvel való megismerkedést is elősegítette, így egy konkrétabb képet kaphattam a szerkesztő felületről, a beépített metódusokról és függvénykönyvtárakról.

1. FELHASZNÁLT ESZKÖZÖK

A program megvalósításához alapvetően 2 fő eszközt használtam, a Unity fejlesztői környezetet és a C# programozási nyelvet a Visual Studio felületével. Mivel céljaim közt több magasabb szintű funkció megvalósítása is szerepelt, ezért több alapvetőnek nem mondható csomagot és eszköz használatát vettem igénybe. Az alábbiakban ezeket

részletezem, mivel azok megléte és használata nem triviális. Ezen eszközök és csomagok alkalmazása lehetővé tette, hogy a Unity és a C# nyújtotta lehetőségeket mélyebben kihasználjam, és így gyorsítsam munkámat, aminek eredményeképpen magasabb színvonalú alkalmazást hozhattam létre.

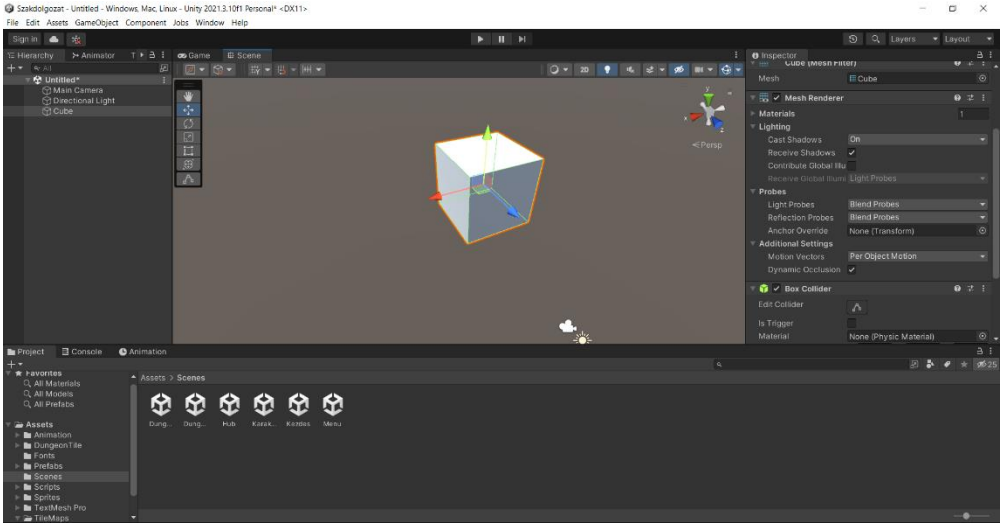
1.1. Unity

A Unity napjaink egyik legnépszerűbb játékfejlesztő szoftvere és keretrendszere, amely lehetővé teszi a felhasználók számára, hogy interaktív 2D és 3D játékokat, szimulációkat és animációkat készítsenek számítógépre, mobil eszközökre, vagy akár konzolokra is. A keretrendszer egy egyszerű és könnyen használható felhasználói felülettel rendelkezik a Unity Editorral (lásd 1.1.1. ábra), amely lehetővé teszi a fejlesztők számára a játék összes, vagy legalábbis a legtöbb paraméterének testre szabását, mint például a grafikát, a hangot, és a fizikát.

Ahhoz viszont, hogy teljes értékű alkalmazásokat tudjunk készíteni szükségünk van egy programozási nyelvre is, Unity esetén ilyen például a C#, amiben megírhatjuk külön script fájlokba kiszervezve a játék működésének logikáját, annak szabályrendszerét. Ilyenkor lényegében a Unity API-ját használjuk fel, hogy hivatkozzunk a különböző játékobjektumokra és módosítsuk azok viselkedését, avagy tulajdonságát.

Az Unity használatakor lényeges, hogy ismerjük az alapfogalmakat, mint például a *GameObject*, ami minden egyes elemet jelöl a keretrendszerben. Fontos továbbá tisztában lenni a *Start* és *Update* metódusokkal, azok működésével, amelyek előre definiáltak, ha egy C# scriptet hozunk létre. A *Sprite*-ok ismerete és a különbség a 2D és 3D között szintén nélkülözhetetlen az alkalmazás fejlesztése során. Ezen információk megértése alapvetőnek tekinthető, ha ezzel a játékmotorral kezdünk el foglalkozni, ezért az én első lépésem is az volt, hogy utánajártam ezeknek a fogalmaknak.

Amennyiben kezdők vagyunk a témában, akkor is számos segítség áll rendelkezésünkre, akár a scriptable objectekkel kapcsolatban is. Egyik ilyen opciónak említeném a Unity közösség fórumát, melynek köszönhetően keresett kérdéseinkre választ kaphatunk.



1.1.1. ábra – Unity Editor

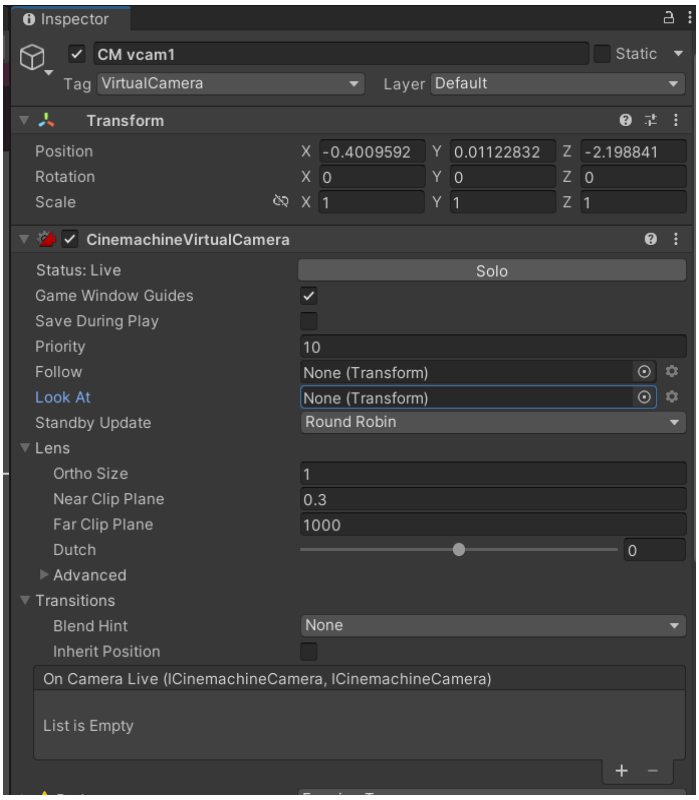
1.2. Felhasznált csomagok

Az alábbi pontokban részletezem jobban az általam vélt legfontosabb 3 csomagot, amik munkámat nagy mértékben előre segítették, így megengedve, hogy a lényegi részre koncentráljak. Az első ilyen csomag a *Cinemachine*, ami a fő kamera kezelését támogatta, a második a *Sprite Library Asset*, ami a spriteok, képek kezelésében nyújtott segítséget és a harmadik a *NewtonSoft JSON*, ami a JSON fájlok írását és olvasását egyszerűsítette. [1,2,3]

1.2.1. Cinemachine

A Unityben a megjelenítésért egy speciális játékobjektum felel, ami a *Camera*. A *Camera* segítségével korlátozhatjuk és befolyásolhatjuk azt, amit láttatni szeretnénk a felhasználóval, azonban ennek a *gameobjectnek* alapvetően nincs megvalósítva a logikája. A *Camera* logikájának implementálására különböző elterjedt lehetőségek állnak rendelkezésre, egyik ilyen az ingyenesen elérhető *Cinemachine* csomag a Unityn belül.

A *Cinemachine* használatakor egy új virtuális kamerát hozhatunk létre, aminek átadhatjuk a Main Cameránkat a *CinemachineVirtualCamera* komponensen keresztül. Ezek után pedig megannyi attribútum érhető el a komponensen belül, mint a *Look At, Follow,* vagy az *Ortho Size* (lásd 1.2.1.1. ábra). Ezekkel közvetetten állíthatjuk a *Main Camera* objektumunk tulajdonságait, így például azt is, hogy melyik objektumra fókuszáljon, vagy hogy milyen gyorsan kövesse azt. [1]

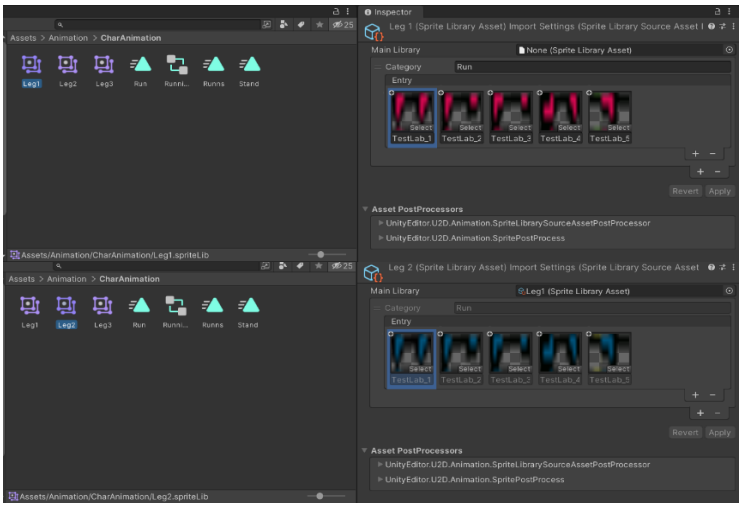


1.2.1.1. ábra – A CinemachineVirtualCamera komponensei

1.2.2. Sprite Library Assets

A 2D játékok során az egyes játékelemek textúráját úgynevezett *spriteokból* szokás létrehozni. Például, ha a karakter mozgásának animációját szeretnénk megvalósítani, akkor valószínűleg több spriteot/képet is elkészítünk a karakterről, a mozgás egyes állapotait reprezentálva. Ezek után a mozgáshoz már csak képkockánként kell váltogatnunk a karakter spriteját, így azt az illúziót keltve, hogy mozog.

Több *sprite* együtt kezeléséhez a Unityben a *Sprite Library Asset* könyvtár nyújthat segítséget. Alkalmazásakor, ha azonos méretű spriteokat használunk, akkor kezdésként egy alapértelmezett *sprite* listát, kategóriát hozunk létre (lásd 1.2.2.1 ábra), majd a továbbiakban ezt másolva és felhasználva, tudjuk a másolatokra is alkalmazni ugyanazon animációkat, vagy más egyéb módosításokat. Ilyenkor természetesen elegendő egyetlen egy animációt létrehozni, hiszen csak a *spriteok* kinézete változik. Ha nem használnánk ezt a csomagot, akkor annyi animációt kellene megvalósítani, ahány féle kinézetet akarunk az adott elemnek. [2]



1.2.2.1. ábra – Egy sprite lista (felül) és másolata (alul)

1.2.3. Newtonsoft JSON

A *C#* maga rendelkezik olyan beépített függvénykönyvtárral, ami képes *JSON* formátumot kezelni, azonban vannak esetek, amikor ez nem nyújt optimális megoldást.

A *NewtonSoft Json* egy olyan ingyenesen elérhető külső könyvtár, aminek módszerei egyszerűsitik a *JSON* kiterjesztésű fájlok, írását és olvasását. A csomaghoz C# esetén a NuGet Packageken keresztül férhetünk hozzá. [3]

Egyik leghasznosabb funkciója, hogy az általunk létrehozott osztályokat könnyedén szerializálhatjuk *JSON* formátumúvá, szövegként kezelve. Ha pedig deszerializálunk, akkor a *JSON* objektumot stringként tudja majd kezelni, amiből képes vissza alakítani azt egy saját típusú objektummá. (lásd 1.2.3.1 – kódrészlet)

```
private void ExampleJson()
{
string p = "\\dir1\\dir2\\file.json"; //Útvonal
SpawnPoint point = new SpawnPoint();

//Itt a SpawPoint tipusu objektumot alakítja stringgé, JSON formázással
var json = JsonConvert.SerializeObject(point, Formatting.Indented);

// Itt egy egész számokból álló vektor halmazt hoz létre a JSON fileból
var deserialized =
JsonConvert.DeserializeObject<HashSet<Vector2Int>>(File.ReadAllText(p));
}
```

1.2.3.1. kódrészlet – Szerializálás és Deszerializálás

2. MEGVALÓSÍTÁS

Az elsődleges célom a program elkészítése során az volt, hogy megvalósítsam azokat a funkciókat, amelyek az egész játék alapját képezik, vagyis a fő karaktert és annak logikáját. Tudtam, hogy az ellenfelek, a felvehető tárgyak, a pályák és a nem játszható karakterek mind a főhőshöz kapcsolódnak, ezért ennek a funkciónak kellett az elsők között elkészülnie. Miután az alapját megalkottam, foglalkoztam a karakter szerkesztővel, az ellenfelekkel, a pályák kialakításával és a harcrendszerrel, valamint a különböző tárgyak hozzáadásával a játékhoz.

A program elkészítése során fontos volt, hogy az elmentett scriptek jól strukturáltak legyenek, továbbá a komplex problémák minél kisebb részegységekből álljanak, hogy a további fejlesztések egyszerűsödjenek. Az egyes funkciókat külön fájlokban rendeztem, és ezeket a következő fejezetekben részletesen bemutatom.

2.1. A JÁTÉKOS KARAKTER LOGIKÁJA

A játék fő mozgató rugója a játékos karakter. A fejlesztés kezdeti szakaszában a hozzá köthető funkciókat valósítottam meg. Miután adtam egy *spriteot, collidert* és *Rigidbody 2D*-t a komponensekhez, a mozgást könnyedén hozzá tudtam adni a játékhoz, ezután második lépésként a támadását valósítottam meg. Amikor az alapvető logika elkészült, akkor készítettem el egy szerkesztőt, ahol a játékos minimálisan testreszabhatja a karakter külsőjét.

A játékosnak legfőbb attribútumai az ereje és az életereje, amik majd az ellenfelekből megkapott tapasztalatpontok által elért szintekkel együtt fejlődnek. Egyik talán legfontosabb jellemzője a *Player* objektumnak, hogy ha egy ellenséges objektummal érintkezik, akkor egy *Coroutine* segítségével értékek vonódnak le az életerejéből. Sebződés hatására pedig, rövid időre eltűnik a karakter maga, ezzel mutatva a felhasználónak, hogy a karaktere megsebesült és veszített az életerejéből. A *Coroutine* használata pedig azért volt célszerű, mert szerettem volna több képkockán keresztül mutatni a sebződést.

Továbbá, mivel a játék harc orientált, ezért az életerőt fontos volt számon tartani, így ennek érdekében a képernyő bal felső sarkában helyeztem el az életerő és tapasztalatpont sávokat, hogy jelezzék a játékosnak a veszélyt. (lásd 2.1.1. ábra)

2.1.1. ábra – A karakter életerő és szint sávja

2.1.1. A játékos mozgatása

A játékos ütközés detektálását a Unity 2D *Capsule Collider*rel oldottam meg. Ez korábban *Box Collider 2D* volt, azonban későbbi megfontolásból, hogy a játékos ne akadjon be, jobbnak gondoltam egy íves érintkező felületet adni neki. (lásd 2.1.1.1. ábra) A *collideren* kívül szükségem volt még egy *Rigidbody 2D* komponensre, ami a fizikát engedte befolyásolni, így ezt felhasználva módosíthattam a játékos gyorsulásának irányát és mértékét.

A Top Down nézetű játékok alapvetően korlátozzák a mozgás lehetőségeit, rendszerint 4, vagy 8 irányra. Fejlesztéskor a 2 beépített tengelyt használtam fel, amik folyamatosan vizsgálják, hogy a <WASD> billentyűk közül melyik van lenyomva, és ha a 2 tengelyhez tartozó billentyűk közül egyet-egyet lenyomunk, akkor a ferde irányok egyikébe mozdul el a játékos objektum. Például <W> és <D> hatására jobb, fel irányba mozdul el.

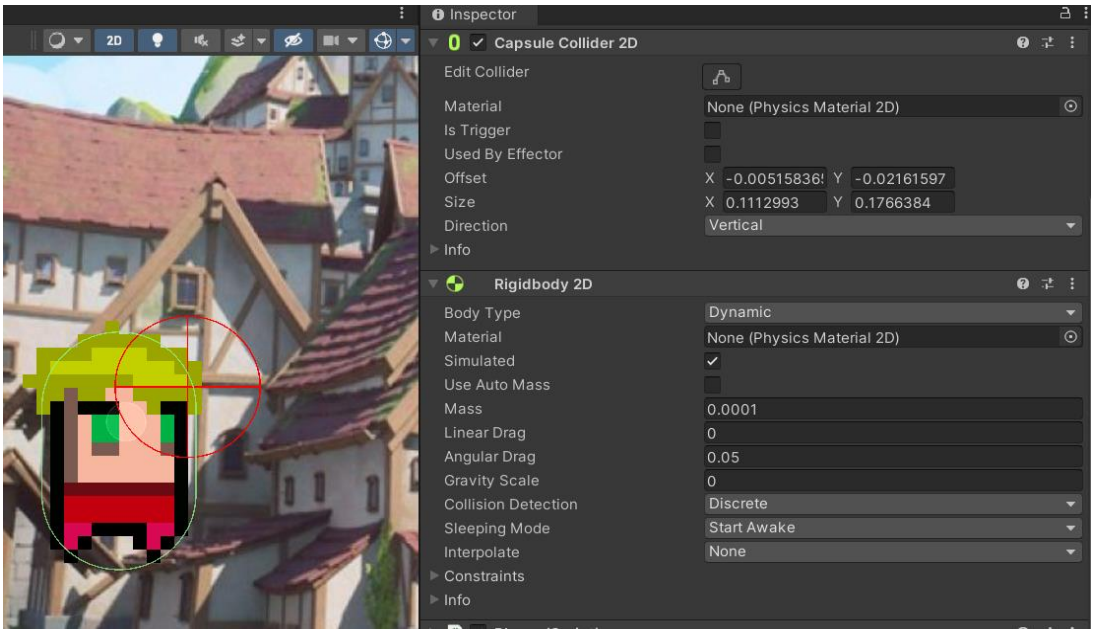
A scriptben több adattagra is szükségem volt, mint a *movementSpeedre* a karakter sebességéhez, a *positionra* a karakter mozgás irányáért, a *rigidbodyra* a fizikai tulajdonságokhoz és az *animatorra*, hogy a megfelelő animáció induljon el mozgáskor.

Ha a játékos már elkezdte a játékmenetet, akkor a *FixedUpdate* beépített függvénnyel, folyamatosan meghívom a *Moving* metódust, ami a mozgásért felel. A *FixedUpdate* használata azért volt célszerűbb az *Update* használatánál, mert ez kifejezetten a fizikai tulajdonságok módosításához lett kitalálva.

A *Moving* metódus, bekéri, a horizontális és vertikális adatokat. Például, ha megnyomjuk az <A> billentyűt, akkor a horizontális input -1 értéket fog visszaadni, ha

pedig nem történik gomblenyomás, akkor 0 az értéke. A bekérések után mivel csak a játékos pozíciójának vektorirányа érdekel, ezért normalizálom. A normalizálást követően, ha történt gomblenyomás, vagyis a horizontális, vagy vertikális inputok nem 0 értékűek, akkor elindul a *run* animáció a lábakra, majd a *rigidbody*, vagyis a karakter gyorsasága a pozíció irányával és a mozgási sebességgel lesz egyenlő. Ennek hatására az elmozdul a kívánt irányba. (lásd 2.1.1.2. kódrészlet)

A metódusban még végzek egy ellenőrzést, a *Flip* eljárással. Ezzel azt vizsgálom, hogy jobbra, vagy balra mozog-e a karakter és ennek megfelelően fordítom meg a spriteot. (lásd 2.1.1.3. kódrészlet)



2.1.1.1 ábra – Capsule Collider 2D és a Rigidbod 2D

```
public void Moving()
{
	position.x = Input.GetAxisRaw("Horizontal");
	position.y = Input.GetAxisRaw("Vertical");
	Flip(position.x);
	position.Normalize();

	if (position.x != 0 || position.y != 0)
	{
		animator.SetTrigger("run");
	}
	else
	{
		animator.SetTrigger("stand");
	}

	rigidbody.velocity = position * movementSpeed;
}
```

2.1.1.2 kódrészlet – A Moving metódus

```
public void Flip(float direction)
  {
        float x1 = this.transform.localScale.x;
        float y1 = this.transform.localScale.y;

        if (direction == -1)
        {
            if (x1 > 0)
            {
                this.transform.localScale = new Vector3(-x1, y1, 1);
            }
        }
        else if (direction == 1)
        {
            if (x1 < 0)
            {
                this.transform.localScale = new Vector3(-x1, y1, 1);
            }
        }
  }
```

2.1.1.3 kódrészlet – A Flip metódus

2.1.2. A játékos támadása

A játékost alapvetően több játékobjektumból építettem fel, mivel erre szükségem volt a karakter szerkesztő és a fegyverek miatt is. Így a *Player* objektum rendelkezik egy *WeaponParent* objektummal. Ennek pedig van egy másik gyerek objektuma, ami a *Weapon*, ehhez pedig szintén tartozik egy gyerek a *CircleOrigin*, ami csak egy gizmo és lényegében érintkezőként szolgál.

A *Weapon* szerepe a megfelelő sprite megjelenítése és annak animálása, míg a *WeaponParent* felel a támadás logikájának megfelelő végrehajtásáért.

Első lépésként a sebzések adatait különböző tömbökben tárolom, azután, ha a karakter új fegyvert vesz, akkor annak adatait hozzáadom a megfelelő tömbhöz. A jelenleg használt fegyver indexét pedig a *currentWeaponIndex* adattagban tárolom. A fegyver cserélésért a <C> billentyű, a *ChangeWeapon* metódus felel, ahol, ha az index túllépné a jelenleg megszerzett fegyverek számát, akkor visszatér a 0. indexre, így kiküszöbölve az IndexOutOfRangeException -t.

Amint a karakter kiválasztotta a megfelelő fegyvert, akkor egyből támadhat is vele. A támadásért felelős *Attack* függvény (lásd 2.1.2.1. kódrészlet) lényegében csak elindítja a *Weapon* objektum által megkapott *anim* adattag animációját a megfelelő *Trigger* bekapcsolásával. Fontos, hogy mivel nem szerettem volna, hogy a játékos minden egyes támadás után rögtön támadhasson még egyet, ezért létrehoztam egy *Coroutine*-t, a

DelayAttack-ot, ami csak késlelteti az adott fegyver adatai alapján a támadásokat egy *attackBlocked* adattag módosításával.

Mikor elkészült a támadásnak ezen verziója, akkor még hátra volt, hogy az adott fegyver megtudja különböztetni a támadott objektumokat. Ehhez hoztam létre a *DetectColliders* metódust. A metódus figyeli, hogy egy *Collider* bele esik-e a korábban említett *CircleOrigin* gizmoba, amennyiben ez megtörténik és a *Colliderrel* rendelkező objektum egy ellenfél, vagy egy széttörhető tárgy, akkor a fegyver sebzése és a játékos karakter ereje levonódik az érintkezett objektum életerejét reprezentáló adattag értékéből. (lásd 2.1.2.2 kódrészlet)

```
public void Attack()
    {
        if (attackBlocked)
        {
            return;
        }
        else if (isActiveAndEnabled)
        {
            if (GetComponentInChildren<SpriteRenderer>().sprite == null)
            {
                anim.SetTrigger("slashHand");
            }
            else
            {
                anim.SetTrigger("slashWeapon");
            }
            IsAttacking = true;
            attackBlocked = true;
            StartCoroutine(DelayAttack());
 }}
```

2.1.2.1. kódrészlet – Az Attack metódus

```
public void DetectColliders()
{//Előtte még egy foreach-el kikeresem a collidereket
if (collider.GetType() == typeof(CapsuleCollider2D) || collider.GetType()
== typeof(BoxCollider2D))
 {
   if (collider.gameObject.CompareTag("Player"))  continue;
   //ezután még vizsgál a széttörhető tárgyakra is, nem csak az Enemyre
   else if (collider.gameObject.CompareTag("Enemy"))
   {
    Enemy enemy = collider.gameObject.GetComponent<Enemy>();
    if (attackForFirst == true)
    {
      int dmg = weaponDamages[currentWeaponIndex] +
      this.transform.GetComponentInParent<Player>().strength;
     StartCoroutine(enemy.DamageCharacter(dmg, 0.0f,
     GetComponent<Transform>().position,
     weaponPushForces[currentWeaponIndex])
     attackForFirst = false;
    }
  }
```

2.1.2.2. kódrészlet – Részlet a DetectColliders metódusból

2.1.3. A karakterszerkesztő

A karakter, mint korábban is említettem egy több részegységből álló objektum. Korábban a WeaponParentről írtam, most a további részeit fogom kifejteni.

Mind a 4 részegység a *Hair, Head, Body* és a *Legs* azért felel, hogy a megfelelő spriteot jelenítse meg a felhasználó számára. Azonban a *Legs* egyéb feladattal is rendelkezik, hisz amikor elindul a játékos futás animációja akkor a láb spriteoknak kell váltakozniuk. Ehhez a már említett *Sprite Library Assets* könyvtárat alkalmaztam.

Azok után, hogy a főmenüből tovább lépünk egy szerkesztő jelenik meg. (lásd 2.1.3.1 ábra) Itt az említett testrészeket módosíthatja a felhasználó. Ehhez egy *Canvas* objektumot kellett létrehoznom, és egy *Panelt*, amin belül elhelyeztem a látható módosító gombokat. Minden egyes egységnek külön sora van, amihez tartozik egy előző és következő gomb. Amikor ezen gombok egyikére rákattintunk, akkor a *CharacterCustom* script kezd el tevékenykedni.

Ha a „következő” gombra lesz kattintva, akkor az ahhoz bekötött *NextOption* függvény hívódik meg, ahol az *inspectorban* hozzáadott testrész spriteját változtatja meg a következőre. Elsőnek megnöveli a jelenlegi opció indexét és ezután, ha nem a lábakról van szó, akkor egyszerűen kiveszi a listából a következő spriteot, és azt értékül adja az objektum spritejának. Viszont, ha a módosított testrész a *Legs*, akkor a spriteot egy *SpriteLibraryAsset* listából veszi ki, ami azért lényeges, mert így az animációt elég lesz egyszer megvalósítani a mozgáskor, és nem kell minden egyes képre külön-külön.

2.1.3.1. ábra– A karakter módosítása

2.2. ELLENFELEK

Az ellenfelek a játékban leginkább azért szerepelnek, hogy az élményt izgalmasabbá tegyék, és megnehezítsék a játékmenetet, ezzel egyfajta akadályt jelentve a játékos számára.

Alapvetően 3 különböző fajta ellenfél létezik a játékban, amiből 2-nek azonos a logikája. Az első kettő megvalósítás pusztán támadó jelleggel van a pályákon. Ebből következett, hogy a játék így nem túl izgalmas, hiszen ha csak támadnak az ellenfelek és semmi mást nem csinálnak, akkor egy idő után monotonná válik a játékmenet. Ezen probléma megoldására valósítottam meg a 3. fajta nemezist, aminek viselkedése lényegesen eltér az első kettőtől.

Ezen játék objektumoknak hasonló komponenseket adtam, mint a játékosnak magának, tehát egy *Sprite Renderert*, egy *Capsule Collider 2D*-t, és egy *Rigidbody 2D*-t. Az alábbi pontokban ezen ellenséges karakterek logikáját fejtem ki részletesebben.

2.2.1. Az egyszerű támadó ellenfél logikája

A támadó ellenfelekbe az egyszerű *Orc* és *BigOrc* tartozik. Kettőjük között annyi a különbség, hogy a második sebzése és életereje nagyobb, míg mozgási sebessége lényegesen kisebb.

Elsősorban mivel ezek az objektumok némileg hasonlítanak a játékosra, ezért célszerű volt a karakterek osztályból leszármaztatni külön az *Enemy* osztályba. Az osztályon belül a legfőbb metódusok az *OnCollisionEnter2D*, a *DamageFlash* és a *CheckLevelAndPowerUp*. Az első azért felel, hogy ha az objektum egy Player taggel ellátott másik *GameObjecttel* ütközik, akkor elindítja annak a *DamageCoroutineját*, így az elkezd sebződni. A második függvény a látványért felel, vagyis, ha az *Enemyt* megüti a játékos, akkor az pirosan felvillan. Ennek megvalósítása is egy *Coroutinenal* történt, hisz szerettem volna, hogy több képkockán keresztül látszódjon, ahogyan a karakter megsebzi az ellenfelet. A harmadik metódus megnézi, hogy a globális ellenfél szint, ami a játékossal növekszik nagyobb-e, mint a jelenlegi, ha igen akkor az lesz az aktuális. Ezután, hogy az ellenfelek a játékos fejlődése után is kihívást nyújtsanak, az életerejük, az értük járó tapasztalatpont, a sebzésük és a követési sebességük is növekszik. (lásd 2.2.1.1. kódrészlet)

Az ellenfelek igazi mivoltját azonban nem az *Enemy*, hanem a *Wander* script adta meg. Ehhez a Developing 2D Games With Unity könyv által adott kódrészletet vettem alapul. [4]

A script feladata leginkább, hogy amíg a játékos nem lép be a meghatározott hatókörön belülre, addig a *ChooseNewEndpoint* (lásd 2.2.1.2. kódrészlet) függvény véletlenszerűen választ egy pontot az *Orc* körül. Ehhez a ponthoz az objektum saját helyzetét és egy szöget vesz alapul. Miután az *endPosition* meghatározásra került, azt az *IEnumerator* típusú *Move* eljárás használja fel. Itt folyamatosan nézi a hátralévő távolságot és ha még nem nulla, akkor folyamatosan mozgatja a pont felé az objektumot, miközben a megfelelő animációt futtatja. Amennyiben a játékos belép a triggerként funkcionáló collider hatáskörébe, akkor a *targetTransform* változóba lementődik a játékos helye, ez pedig majd a *Move*-on belül átadódik az *endPositionnak*, így a követendő pont maga a játékos pozíciója lesz, egészen addig, amíg a játékos életereje le nem csökken 0-ra azáltal, hogy az ellenfél ütközik vele. (lásd 2.2.1.3. kódrészlet)

```
private void CheckLevelAndPowerUp(){

       if(GameManager.sharedInstance.powerForEnemies > currentLevel){
           currentLevel = GameManager.sharedInstance.powerForEnemies;

           maxHealthPoints = currentLevel*5 + baseHP;
           xp = currentLevel*2 + basexp;
           if(hitPoints == maxHealthPoints){
               hitPoints = currentLevel*5 + baseHP;
           }else{
               hitPoints += currentLevel*5 + baseHP;
               if(hitPoints > maxHealthPoints){
                  hitPoints = maxHealthPoints;
               }
           }
           damageStrength = currentLevel + baseStrength;

           if(this.gameObject.name.ToLower().Contains("shaman")){

      this.gameObject.GetComponent<ShamanWander>().pursuitSpeed += 0.05f;
           }else{
            this.gameObject.GetComponent<Wander>().pursuitSpeed += 0.05f;
           }

           startingHitpoints = maxHealthPoints;

       }
   }
```

2.2.1.1. kódrészlet– CheckLevelAndPowerUp metódus

```
void ChooseNewEndpoint()
{
      currentAngle += Random.Range(0, 360);
      currentAngle = Mathf.Repeat(currentAngle, 360);
      endPosition = Vector3FromAngle(currentAngle) + transform.position;
}
```

2.2.1.2. kódrészlet– ChooseNewEndpoint metódus

```
public virtual IEnumerator Move(Rigidbody2D rigidBodyToMove, float speed)
{
float remainingDistance = (transform.position -
endPosition).sqrMagnitude;
    while (remainingDistance > float.Epsilon)
    {
        if (targetTransform != null)
        {   //Itt állítja be a játékos pontját a követési pontnak
            endPosition = targetTransform.position;
        }
        if (rigidBodyToMove != null)
        {
         animator.SetBool("isWalking", true);
         Vector3 newPosition =
         Vector3.MoveTowards(rigidBodyToMove.position,
                             endPosition, speed * Time.deltaTime);
         rb2d.MovePosition(newPosition);


         remainingDistance = (transform.position -
                             endPosition).sqrMagnitude;
         }//Itt vár a következő fizikára optimalizált updatera (frame-re)
         yield return new WaitForFixedUpdate();
    }
        animator.SetBool("isWalking", false);
}
```

2.2.1.3. kódrészlet– A Move metódus

2.2.2. A menekülő ellenfél logikája

A menekülő ellenfél az úgynevezett *ShamanOrc* játékobjektum. Ennek is hasonlóan vannak elrendezve a komponensei az *inspectoron* belül, azonban különlegessége a továbbfejlesztett *Wander* scriptben a *ShamanWanderben* rejlik.

Alapvetően ez az ellenfél is egy véletlenszerű *endPointot* választ ki, azonban a játékos közeledtét egészen máshogy kezeli. Amikor a karakter belép a triggeren belülre, akkor a *Shaman* további négy támadót jelenít meg, úgy hogy azok a játékos és a *Shaman* közötti távolság felére jelennek meg. A segítők ezután rögtön elkezdik támadni a játékost, az eredeti ellenfél objektum pedig elmozdul a megjelenítési pontra és onnan elkezdi meghatározott időközönként növelni a 4 védelmezője életerejét. Amikor a 4 megidézett

objektum legyőzetett, akkor a *Shaman* nem a játékost fogja támadni, hanem ellenkező irányba kezd el futni, vagyis menekül előle.

A 4 *Orc* leidézése az *OnTriggerEnter2D* metódusban történik, ahol beállítom a *followPlayer* adattagot, és meghívom a *FollowFromFar* metódusát a 4 objektumnak, hogy minden esetben elkezdjék követni a játékost. Ezután pedig beállítom az említett pont felét a célpontnak, hogy a *Shaman* odamenjen. (lásd 2.2.2.1. kódrészlet) A gyógyításért egy egyszerű *HealServants* függvény felel, ami végig iterál a 4 objektumon, ha azok léteznek és meghívja azok *Heal* metódusát, ami egy adott értékkel növeli az életerő adattagjukat. A további mozgásért felelős eljárás hasonlóan a sima *Wander* esetén itt is a *Move*, viszont öröklődéskor felül van definiálva a viselkedése, vagyis a megcélzott pozíció akkor lesz a játékoshoz képest az ellenkező irányba, ha az összes védelmező objektuma megszűnt, de még a játékosé nem. Ebben az esetben a játékos pozíciója -1-el lesz szorozva, mert a koordinátarendszerben az ellenkező pontot szeretném megkapni, de hogy ne az origóhoz képest legyen ez a pont számolva, ezért ehhez az értékhez még hozzá kell adnom a *Shaman* objektum pozícióját is, így lényegében ez lesz az új origó és így tükröződik az irány. (lásd 2.2.2.2. kódrészlet) A *Move* függvény a továbbiakban hasonlít az eredetire.

```
if (collision.gameObject.CompareTag("Player"))
{
 for (int i = 0; i<gameObject.transform.parent.transform.childCount;i++)
   {
   if (gameObject.transform.parent.transform.GetChild(i).gameObject.name
       != this.gameObject.name &&
       gameObject.transform.parent.transform.GetChild(i).gameObject.name.C
       ontains("orc"))
     {
       gameObject.transform.parent.transform.GetChild(i).GetComponent
       <Wander>().followPlayer = true;

       gameObject.transform.parent.transform.GetChild(i).GetComponent
       <Wander>().FollowFromFar(collision);
                 }
             }
             currentSpeed = pursuitSpeed;
             targetTransform = collision.gameObject.transform;
             targetPosition = (transform.position +
             targetTransform.position) / 2;
             if (numberOfEnemies != -1) SummonServants(targetPosition);
```

2.2.2.1. kódrészlet– Részlet az OnTriggerEnter2D eljárásból

```
else if (targetTransform != null &&
gameObject.transform.parent.childCount < 2)
{
        endPosition = targetTransform.position;
        endPosition.x = endPosition.x * -1;
        endPosition.y = endPosition.y * -1;
        endPosition += this.transform.position;
        endPosition += this.transform.position;
}
```

2.2.2.2. kódrészlet– Részlet a Move eljárás számításáról

2.3. GAMEMANAGER

A *GameManager* objektum szerepe, hogy a játék során több fontos adatot is eltároljon és ha kell elérhetővé tegye azokat más scriptek számára.

Ebben az esetben ezeken az adatok eltárolásán kívül a *GameManager* script azért is felel, hogy a Singleton design pattern-t felhasználva elérhetőséget biztosítson a *ShowText* metódushoz. Mivel a játékban többször is jelenítek meg szövegeket, tábláknál, ellenfél leidézéseknél, vagy akár a vásárlásnál, ezért szükségszerű volt, hogy kitaláljak egy globális megoldást arra, hogy lebegő feliratokat jelenítsek meg.

Az alapötlet egy *FloatingText* osztály létrehozása volt, ami a megjelenítendő szöveget tartalmazza. Ezt a *FloatingTextManager* osztály kezeli, ahol a *Show* metódus valósítja meg a szöveg megjelenítését. A *GameManager* pedig rendelkezik egy ilyen *FloatingTextManager* adattaggal, és ha szöveget szeretnék megjeleníteni, akkor a *sharedInstance* segítségével meghívhatom a *GameManager ShowText* metódusát, ami a *floatingTextManager* adattagot hívja meg, és annak *Show* függvényét hajtja végre. (lásd 2.3.1 kódrészlet)

```
//A Gamemanagerben lévő metódus
public void ShowText(string msg, int fontSize, Color color, Vector3
position, Vector3 motion, float duration)
{
        floatingTextManager.Show(msg, fontSize, color, position, motion,
duration);
}
//Példa a használatára a ShamanWanderen belül, a leidézéskor
if (fightStarted == false)
{
GameManager.sharedInstance.ShowText("Siess, a sámán elkezdte
gyógyítást!", 70, Color.red, targetTransform.position, Vector3.up *
0.15f, 3.0f);
}
```

2.3.1. kódrészlet– Példa a GameManager ShowText metódusára

2.4. PÁLYASZERKESZTŐ

Szakdolgozatomban az egyik legfontosabb feladat a pályaszerkesztő elkészítése volt. Az alapelképzelés az, hogy miután a játékos végigjátszotta az alappályákat, akkor egy általa elkészített kazamatába léphet be és így teljesítheti azt.

Elsősorban nem akartam a terület alapját is megszerkeszthetővé tenni teljes mértékben, ezért annak alakját procedurális generálással oldottam meg, amihez egy lejátszási listából vettem az alapot. [5]

Másodszor pedig, amikor elkészült a procedurális generálás akkor elkezdhettem fejleszteni a szerkesztő részét. Itt az volt a célom, hogy a játékos ellenfeleket, tárgyakat, gyógyitalokat és pénzt helyezhessen le, továbbá ha valamit véletlenül lehelyezett, akkor törölhesse azt. Az alábbi pontokban a procedurális generálás folyamatát és a pályaszerkesztő felépítését mutatom be.

2.4.1. Procedurális generálás

A procedurális generálás egy sokak által kedvelt eljárás, hogy a digitális játékokban világokat és pályákat készítsenek. Különlegessége abban rejlik, hogy egy adott szabályrendszert felhasználva véletlenszerűen képes a különböző tartalmakat legenerálni, így az én esetemben a kazamaták alapját.

A procedurális generálás folyamata ebben az esetben úgy zajlik, hogy amikor a felhasználó rákattint a Generál gombra, akkor a *RoomsFirstDungeonGenerator* script metódusai hívódnak meg. Abban az esetben, ha a szerkesztő jelenetben van a felhasználó, akkor a *CreateRooms* metódus hívódik meg, (lásd 2.4.1.1. kódrészlet) egyébként, a *CreateRoomsFromJson*. Az előbbi első lépése, hogy a *static SpacePartitioning* osztály *BinarySpacePartitioning* metódusát felhasználva felosztja több részre a termet. Ezután a felosztott részekből elkészíti a padlót a *CreateSimpleRooms* függvénnyel, ahol lényegében csak egy Vector2Int-ből álló halmazt tölt fel vektorokkal. Miután ez kész van, a *NewtonSoftJson* metódusát felhasználva szerializálom ezeket az adatokat, hogy majd lementhessem egy txt fájlba és mivel szükségem van a szobák közepére is, ezért a *BoundsInt* típust kihasználva ezeket az adatokat is *JSON* formátumúvá alakítom. Ekkor a szobák alapja már elkészül, már csak a köztük lévő folyosókat és falakat kell létrehozni.

A folyosók a szoba középpontjaiból jönnek létre a *ConnectRooms* metódussal, ahol lényegében az adott középponthoz ki lesz keresve a legközelebbi még fel nem

használt középpont. Ennek eredménye mindig egy folyosó, és ez lesz egyesítve a többi folyosóval, amikor az összes szoba középpont elfogy a listából, akkor pedig visszaadja az egyesített folyosó halmazt. (lásd 2.4.1.2. kódrészlet)

Amikor a folyosók is elkészülnek, akkor azok egyesítésre kerülnek a padlózattal majd végeredményként a mezőket vizuálisan is megjeleníti, és így már csak a falak lehelyezése hiányzik. A falakat egy külön osztály a *WallGenerator* generálja le. Alapesetben a korábban generált folyosó és padlózat koordinátákat felhasználva megnézi 8 irányban, hogy üres mező található-e, és ha igen akkor oda falat helyez le.

Az említett *BinarySpacePartitioning* első paramétere egy *BoundsInt,* ami lényegében egy terem, vagy tér, amit majd vertikálisan, vagy horizontálisan kettéoszt a függvény. A particiókra osztó metódus, azt vizsgálja, hogy ha a terület szélessége nagyobb, mint a minimum szélesség kétszerese, akkor lényegében vertikálisan osztja fel, ha pedig a magassága nagyobb a minimum kétszeresénél, akkor horizontálisan. Végeredményként egy listát ad vissza, ami a különböző pozíciókat tartalmazza. (lásd 2.4.1.3. kódrészlet)

```
private void CreateRooms()
{
  var roomList = SpacePartitioning.BinarySpacePartitioning(new
BoundsInt((Vector3Int)startPosition, new Vector3Int(dungeonWidth,
dungeonHeight, 0)), minRoomWidth, minRoomHeight);

     floor = CreateSimpleRooms(roomList);
     floor_ = JsonConvert.SerializeObject(floor, Formatting.Indented);

     List<Vector2Int> roomCenters = new List<Vector2Int>();
     foreach (var room in roomList)
     {
      roomCenters.Add((Vector2Int)Vector3Int.RoundToInt(room.center));
     }

      centers_ = JsonConvert.SerializeObject(roomCenters,
                  Formatting.Indented);

      HashSet<Vector2Int> corridors = ConnectRooms(roomCenters);
      floor.UnionWith(corridors);

      tilemapVisualizer.PaintFloorTiles(floor);
      WallGenerator.CreateWalls(floor, tilemapVisualizer);
}
```

2.4.1.1. kódrészlet– A CreateRooms metódus

```
private HashSet<Vector2Int> ConnectRooms(List<Vector2Int> roomCenters)
  {
        HashSet<Vector2Int> corridors = new HashSet<Vector2Int>();
        var currentRoomCenter = roomCenters[0];
        roomCenters.Remove(currentRoomCenter);

        while (roomCenters.Count > 0)
        {
            Vector2Int closest = FindClosestPointTo(currentRoomCenter,
                                 roomCenters);
            roomCenters.Remove(closest);
            HashSet<Vector2Int> newCorridor =
            CreateCorridor(currentRoomCenter, closest);
            currentRoomCenter = closest;
            corridors.UnionWith(newCorridor);
        }
        return corridors;
  }
```

2.4.1.2. kódrészlet– A ConnectRooms metódus

```
public static List<BoundsInt> BinarySpacePartitioning(BoundsInt
spaceToSplit, int minWidth, int minHeight)
    {
     Queue<BoundsInt> queue = new Queue<BoundsInt>();
     List<BoundsInt> list = new List<BoundsInt>();
     queue.Enqueue(spaceToSplit);
     while (queue.Count > 0)
     {
      var room = queue.Dequeue();
      if (room.size.y >= minHeight && room.size.x >= minWidth)
      {
        if (Random.value < 0.5f)
        {
          if (room.size.y >= minHeight * 2)
          {
            SplitHorizontally(minHeight, queue, room)
          }
          else if (room.size.x >= minWidth * 2)
          {
            SplitVertically(minWidth, queue, room);
          }
          else if (room.size.x >= minWidth && room.size.y >= minHeight)
          {
            list.Add(room);
          }
     }
```

2.4.1.3. kódrészlet– A BinarySpacePartitioning metódus

2.4.2. A szerkesztő felülete

Amikor a játékos a menüből kiválasztja a szerkesztő menüpontot, akkor térhet át a pályaszerkesztő részre. (lásd 2.4.2.1. ábra) Itt a korábban említett procedurális

generálással, könnyedén készíthet magának pályát, amire ha akar, akkor megannyi ellenfelet, és tárgyat is lehelyezhet.

Miután a játékos rányomott a generálás gombra, akkor a bal oldalon lévő felületet felhasználva, a bal egérgombot nyomva tartva a pálya területére és csak arra, lehelyezheti a különböző játékobjektumokat. Ezt követően, ha valamelyik mennyiségét sokalja, vagy a pozícióval nem elégedett, akkor egy bal egérkattintással törölheti az objektumot. Ha netán a generált kazamata mérete nagy és nem látszódik a képernyőn megfelelően, akkor a görgőt használva nagyíthat és kicsinyíthet a méreten, illetve a nézet helyzetét is módosíthatja a nyíl gombok segítségével.

Amennyiben a lehelyezett objektumok megfelelnek, akkor a felhasználó a mentés gombra kattintva lementheti az adott munkáját, ami az előzőt ilyenkor felülírja. Az ilyenkor JSON-ként lesz elmentve. Az újonnan létrehozott kazamatát az alappályák elvégzése után a játékos ki is próbálhatja.

2.4.2.1. ábra– A szerkesztő és egy generált kazamata részlet

2.4.3. A lehelyezés

A felülethez, ahol tevékenykedhet a felhasználó a már korábban említett *Canvas*-t használtam fel, és ezen belül hoztam létre több *Slot* elnevezésű objektumot, ami a spriteok

megjelenítéséért felel. Ezen belül található még egy *Object* nevű *GameObject* is, amihez egy *Draggable Object* script tartozik, ami a lehelyezésért felel.

A script első sorban 3 beépített metódust használ fel. Ezeket 3 interfészből veszi ki, az *IBeginDragHandlerből*, az *IDragHandlerből* és az *IEndDragHandler* interfészből. Ezen interfészek implementálásakor elérhetővé válnak az *OnBeginDrag*, *OnDrag*, és az *OnEndDrag* függvények.

Az *OnBeginDrag* akkor hívódik meg, amikor rákattintunk az objektumra, és elkezdjük azt húzni. A metódus elsőnek lementi az objektum kezdeti pozícióját, ami ilyenkor a felületen lévő koordináta. Attól függően, hogy a lehelyezendő objektum tárgy vagy ellenfél inicializálja azt, vagyis létrehozza, hogy ne csak a képe legyen elmozgatva, hanem ténylegesen is létrejöjjön egy mozgatható objektum. Ezután lementi az objektum eredeti szülőjét, és átállítja a jelenlegit egy a hierarchiában lévő legfentebb lévőre, majd a sorban az utolsó objektummá teszi. Így az mindenen felül fog elhelyezkedni, ha mozgatjuk. (lásd 2.4.3.1. kódrészlet)

Az *OnDrag* egy nagyon egyszerű függvény, hiszen pusztán annyit csinál, hogy amíg mozgatjuk az objektumot, vagyis lenyomva tartjuk a bal egérgombot, addig folyamatosan beállítja a *GameObject* pozícióját az egér helyzetével megegyezőre. (lásd 2.4.3.2 kódrészlet)

Az *OnEndDrag* eljárás akkor fog meghívódni, amikor elengedjük az egérgombot. Elsősorban lekéri a *tilemapet*, vagyis a kazamata felszínét, majd megpróbálja megszerezni az egér pozícióján lévő cella helyzetét. Ehhez elsőnek átalakítja a mutató koordinátáját egy világbelire, majd ezt felhasználva lekéri az ott elhelyezkedő cella pozícóját, a *WorldToCell* beépített metódussal. Ezután visszaállítja a korábban lementett szülőt, hogy később is fel lehessen használni az adott slotot. Ezek után, mivel a mozgatott slot még nem az objektum, eltünteti azt a *SetActive*-val. A következőkben, ha a lehelyezett mennyiség még nem lépte át a határt, vagyis még van lehelyezhető objektum, akkor készít egy másolatot a prefabot felhasználva és beállítja a szülőjét továbbá a pozícióját. (lásd 2.4.3.3. kódrészlet)

```
public void OnBeginDrag(PointerEventData eventData)
    {
        tempPos = transform.position;
        if (item != null)
            InitialiseItem(item);
        else if (destroyable != null)
            image.sprite = destroyable.sprite;
        else
            InitialiseMob();

        parentAfterDrag = transform.parent;
        transform.SetParent(transform.root);
        transform.SetAsLastSibling();

    }
```

2.4.3.1. kódrészlet– Az OnBeginDrag

```
public void OnDrag(PointerEventData eventData)
    {
        transform.position = Input.mousePosition;
    }
```

2.4.3.2. kódrészlet– Az OnDrag

```
public void OnEndDrag(PointerEventData eventData)
    {
     Tilemap tilemap =
GameObject.FindWithTag("FloorTile").GetComponent<Tilemap>();
        Vector3Int cellPosition =
tilemap.WorldToCell(Camera.main.ScreenToWorldPoint(new
Vector2(Input.mousePosition.x, Input.mousePosition.y)));

        transform.SetParent(parentAfterDrag);
        this.gameObject.SetActive(false);

        if (currentQuantity > 0)
        {
            if (tilemap.GetTile(cellPosition) != null)
            {
                GameObject go = Instantiate(prefabObject, transform);
                currentQuantity--;
                go.transform.SetParent(parentOfObjects);
parentOfObjects.gameObject.GetComponent<DeleteClickedObjects>().parentsOf
DeletedObjects.Add(this);
                go.transform.position =
Camera.main.ScreenToWorldPoint(new Vector3(Input.mousePosition.x,
Input.mousePosition.y, 10));
            }
        }
        transform.position = tempPos;
        this.gameObject.SetActive(true);
    }
```

2.4.3.3. kódrészlet– Az OnEndDrag

2.4.4. A törlés

Mivel ezen funkció teljesen független az objektumok típusától, ezért célszerű volt az ezért felelős *DeleteClickedObjects* scriptet a *GameObjectek* szülőjéhez kapcsolni. Így mivel mindegyiknek azonos a szülője, egyszerre lehet ellenőrizni, hogy rákattint-e a felhasználó az objektumokra, vagy sem.

A scriptben egyetlen egy *Update* metódus szerepel, ami vizsgálja, hogy levan-e nyomva a bal egérgomb. Ha a feltétel teljesül, akkor lekéri annak pozícióját, majd végigmegy az összes objektumon és megvizsgálja, hogy annak pozíciójától csak egy kicsit tér-e el a kattintásunk koordinátája. Amennyiben közel kattintottunk, akkor a *parentsOfDeletedObjects*-ben a megfelelő slotra hivatkozva, növeli a lehelyezhető mennyiséget, majd törli a listából a slotot, hogy többször ne hivatkozzunk rá, végül pedig megsemmisíti az objektumot magát, amihez közel kattintottunk. (lásd 2.4.4.1. kódrészlet)

```
void Update()
{
  if (Input.GetMouseButtonDown(0))
  {
Vector3 posOfMouse = Camera.main.ScreenToWorldPoint(Input.mousePosition);

  for (int i = 0; i < transform.childCount; i++)
  {
  if ((transform.GetChild(i).transform.position.x - 0.5f <= posOfMouse.x
                                    &&
       transform.GetChild(i).transform.position.x + 0.5f >= posOfMouse.x)
                                    &&
      (transform.GetChild(i).transform.position.y - 0.5f <= posOfMouse.y
                                    &&
       transform.GetChild(i).transform.position.y + 0.5f >=posOfMouse.y))
   {

      parentsOfDeletedObjects[i].currentQuantity++;

       parentsOfDeletedObjects.Remove(parentsOfDeletedObjects[i]);
      Destroy(transform.GetChild(i).gameObject);
      break;
    }
  }
 }
}
```

2.4.4.1. kódrészlet– A DeleteClickedObjects metódusa

2.4.5. A mentés

A mentés alapvetően akkor, történik meg, ha a játékos rákattint a felületen a mentés gombra. Ilyenkor, az 2 függvényt hív meg, az egyik a *RoomsFirstDungeonGenerator* scriptben található, amikor is a legenerált kazamata alapját menti le, a második eljárás pedig a *SavePositionsOfObjects* scriptben lelhető meg.

Az első eljárás a *MakeJsonFile*, ami a procedurális generálás során lementett *floor_* és *centers_* változókat írja egy-egy JSON fájlba, a beépített *WriteAllText* függvény segítségével. A *floor_* tartalmazza a padlózatot, míg a *centers*, ezeknek a közepét. Mindkettő lényegében egy koordinátákból felépülő JSON-t alkot.

Az objektumok lementéséért a *Save* metódus felel. Kezdetben megnézi, hogy egyáltalán van-e objektum, aminek a pozícióját lementheti. ha van, akkor egy *for* ciklus segítségével végig iterál rajtuk és megvizsgálja, hogy a mentendő objektum egyezik-e egy előző objektum nevével és egyáltalán volt-e előző objektum. Ha nem akkor csak egyszerűen lementi a *listsOfPositions*-be az objektum koordinátáját. Amennyiben volt következő objektum, de a nevük nem egyezik, akkor végigmegy a *Dictionary*-n és megkeresi az előző névvel ellátott objektumot, és ahhoz hozzáfűzi a pozíció listát, így bővítve az adott típusú objektum elemszámát a json-ben. Ezen felül még egy ellenőrzés történik, hogy ha nem találta meg az objektumot a dict-ben, akkor az azt jelenti, hogy

```
for (int i = 0; i < numberOfChildren; i++)
{
 aChild = transform.GetChild(i).gameObject;
 bool alreadyWas = false;
 if (prevName != "" && prevName != aChild.name)
 {
  foreach (var obj in dict){
   if (obj.Key == prevName)
    {
      dict[prevName].AddRange(listOfPositions);
      listOfPositions = new List<Position>();
      alreadyWas = true;
     }
   }
   if (alreadyWas == false)
   {
    dict.Add(prevName, listOfPositions);
    listOfPositions = new List<Position>();
   }
 }
 listOfPositions.Add(new
       Position(aChild.transform.position.x,aChild.transform.position.y,
       aChild.transform.position.z));
 prevName = aChild.name;
 }
```

2.4.5.1. kódrészlet– Részlet a Save metódusból

még nincs ilyen objektum az adott koordinátákkal, szóval hozzáadhatja a jelenlegit. Miután végigment az összes objektumon, felhasználja a NewtonSoftJson SerializeObject függvényét, majd lementi az adatokat az objects.json-be. (lásd 2.4.5.1. kódrészlet)

INVENTORY

A játékban 3 féle gyógyital található, amiket a játékos a ládákból, az egyes ellenfelektől, vagy egy nem játékos karaktertől is vásárolhat pénzért cserébe. Mivel ezeket az eszközöket bármikor fel kell tudnia használnia, ezért szükséges volt egy eszköztár megvalósítása.

Az Inventory egy a bal alsó sarokbal lévő láda ikonra való kattintással nyitható meg. Ilyenkor a Canvas objektum felhasználásával megjelenik 2 felület. Az első felület a felszerelés maga, ami lényegében csak a kézben tartott fegyver sebzését mutatja. A második felület a tényleges eszköztár, ahol a megszerzett gyógyitalokat felhasználhatjuk. Itt lényegében a szerkesztőhöz hasonlóan imagekkel jelenítem meg az italokat. Ehhez hozzátartozik még 2 gomb is, egy a képre van rátéve átlátszóan, egy pedig a slot sarkába jelenik meg, ha az a slot be van telve. Utóbbival az itemet törölni tudjuk az inventoryból, míg az előzővel felhasználjuk azt. (lásd 2.5.1. ábra)

2.5.1. ábra– Az eszköztár felülete

Ütközéskor automatikusan meghívódik egy *OnTriggerEnter2D* metódus, ami az érintkező itemet felismeri, és ha annak típusa valamelyik potion, akkor az majd az inventoryhoz lesz hozzáadva.

Mivel mindegyik Potion a saját típusán belül mindig ugyanazzal az adatokkal kell rendelkezzen, ezért ennek megvalósításához a legjobb megoldás a Scriptable Object volt, ugyanis ezzel lehetővé válik újrafelhasználhatóvá tenni az objektum adatokat. Segítségével ugyanazokat az információkat, vagyis például a gyógyital életerő növelési értékét elég egyszer eltárolni és nem kell annyiszor, ahány példány van a pályán vagy az inventoryban.

Amikor interaktálunk a megszerzett eszközökkel, akkor felhasználáskor a slothoz csatolt InventorySlot UsItem függvénye fut le. Ez a metódus ilyenkor az item *Use* metódusát fogja meghívni, ami a Scriptable Objectek esetén csak kikeresi a játékos objektum egyik adattagját, és megnöveli azt. Ha törölni kívánjuk az objektumot, akkor az *OnRemoveButton* eljárás hívódik meg, ami pedig az Inventory instancen keresztül törli az objektumot az item listából egy beépített *Remove* metódussal.
(lásd 2.5.2. kódrészlet)

```
public void OnRemoveButton()
 {
  Inventory.instance.Remove(item);
 }
public void UseItem()
 {
  if (item != null)
  {
   item.Use();
   Inventory.instance.Remove(item);
  }
 }
```

2.5.2. kódrészlet– Az OnRemoveButton és a UseItem függvény

3. A JÁTÉK MENETE

Amikor elindul a játék, akkor egy menü jelenik meg. Ezután a szerkesztő menüpontra kattintva a felhasználó elkészítheti a saját pályáját, érdemes ezzel a lépéssel kezdeni hiszen a későbbiekben nem térhet ide vissza.

Amikor megjelenik a szerkesztő, akkor bal oldalt egy felület látható, ahol a különböző lehelyezhető objektumok vannak. Mielőtt még bármit lehelyezhetne a játékos még szükséges generálni egy kazamatát. Ehhez a Generál gombra kell kattintani. Miután létrejön a pálya elkezdődhet annak szerkesztése az ellenfelek és tárgyak lehelyezésével. Ehhez a bal egérgombbal a pálya területre kell húzni a megfelelő objektumot. Ha törölni szeretné a felhasználó valamelyiket, akkor pedig az elemre kattintással teheti meg.

Amennyiben kész a végleges pálya, lementheti azt a Mentés gomb segítségével. Ezek után visszatérhet a menübe a Vissza gombbal.

Második alkalommal az új játékra kattintva megjelenik a karakterszerkesztő. Itt módosítható a gombok segítségével a karakter külsője. Ezután a Kész gombra kattintva elkezdhető a játék. A szerkesztő felületet szintén a bal egérgomb lenyomásával lehet használni.

Rögtön mikor betölt a pályán a játékos, akkor utasítások és segítséget nyújtó feliratok jelennek meg a képernyőn, ha egy-egy lehelyezett tábla közelébe lép a <WASD> billentyűk segítségével. (lásd 3.1. ábra) Az <S> billentyű lenyomásával lefele elindulva lesz az első feladat, ahol egy tábla elmondja, hogy az egyes elemek kiüthetők a támadás gombot használva, ami a bal egérgomb. Ezután egy vagy két ellenfél jelenik meg a jobbra vezető úton, (lásd 3.2. ábra) amik legyőzése után, egy kék mezőre lépve a játékos el teleportál egy köztes térbe.

Ez az úgynevezett Hub arra szolgál, hogy a játékos a megszerzett pénzből, aminek mennyisége a jobb felső sarokban jelenik meg vásároljon italokat, vagy fegyvereket.

Bal oldalra elmenve a fegyver árussal találkozik a játékos, ahol a <1,2,3> billentyűket felhasználva vásárolhat tőle fegyvereket, egy játékmenetben maximum 3-at. (lásd 3.3. ábra) Jobb oldalon a gyógyitalokat veheti meg, szintén hasonló módon.

Miután a felhasználó megvette a szükséges eszközöket, a Hubban felfele elindulva láthatja a kapukat, amik a kazamatákba vezetnek. Balról indulva az első 5 a beépített pálya, míg az utolsó a saját. Utóbbiba akkor léphet be, ha az első ötöt megcsinálta.

Egy pálya akkor lesz elvégezve, ha a játékos a kék mezőtől indulva eljut a pálya végében lévő zöld mezőre, erre rálépve vissza teleportál a Hubba és folytathatja a következő kazamatával.

A játék során a harcok után tapasztalatpontot kap a játékos, amivel automatikusan fejlődik az életereje és ereje, természetesen ezzel párhuzamosan növekednek az ellenfelek attribútumai is, így mindig megtartva a kellő kihívást és izgalmat. A fejlődésnek és a karakterszerkesztésnek hála a játékos könnyebben beleélheti magát a játék folyamán a karakterébe, így létrehozva egy RPG jellegzetességet.

3.1. ábra– A iáték eleie

3.2. ábra– Az első akadály

3.3. ábra– A fegyver árusa

4. ÖSSZEGZÉS

Összességében úgy gondolom, hogy sikerült egy jól funkcionáló játékot készítenem, ami mind a Top-Down és RPG elemeket jól ötvözi. A karakter és ellenfelek fejlődése, a gyógyitalok és az árusok jó alapot biztosítanak, hogy esetlegesen továbbfejlesszem a játékot egy még komplexebb alkalmazássá.

Az alapvető funkciók mellett még egy pályaszerkesztőt is sikerült hozzá készítenem, és amiért sokat kellett kutatnom ezért ennek hála sokkal nőtt a tudásom a

Unityvel és a procedurális generálással kapcsolatban is, de ezeken felül még a játékfejlesztéssel is megismerkedhettem, és ez remek alapot biztosít arra, hogy a jövőben 3D-s játékokkal is elkezdjek foglalkozni.

A fejlesztés során sok akadály állt előttem, kezdve a karakterszerkesztőtől, az ellenfeleken át egészen a pályaszerkesztőig, de ezen problémákra úgy gondolom, hogy ha nem is a legjobb, de elfogadható megoldásokkal álltam elő, és mindvégig figyelembe vettem a teljesítményt és a felhasználóélményt.

Irodalomjegyzék

[1] Developing 2D Games with Unity – 111. oldal (Jared Halpern)

[2] A Sprite Library Assets dokumentációja:
https://docs.unity3d.com/Packages/com.unity.2d.animation@3.0/manual/SLAsset.html (Utolsó látogatás 2022.10.21)

[3] Hello Világ! Hello C#! – 322. oldal (Ruzsinszki Gábor és Lábadi Henrik)

[4] Developing 2D Games with Unity – 277. oldal (Jared Halpern)

[5] A lejátszási lista linkje:
https://www.youtube.com/watch?v=-QOCX6SVFsk&list=PLcRSafycjWFenI87z7uZHFv6cUG2Tzu9v
(A készítő: Sunny Valley Studio, Utolsó megtekintés: 2022.12. 28)

Nyilatkozat

Alulírott Horváth Krisztián programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

dátum, aláírás/név

Köszönetnyilvánítás

Ezúton is szeretném megköszönni a támogatását Berta Virágnak, Hideg Richárdnak, Czibolya Tamásnak és barátomnak Mészáros Tamásnak, aki szakmai tanácsokkal is ellátott. Továbbá köszönöm Stumpf Dávidnak, hogy segédkezett a játék tesztelésében és segített javítani a felhasználói élményt néhány alapötletével.