**Szegedi Tudományegyetem**

**Informatikai Intézet**

# SZAKDOLGOZAT

**Potyesz Máté**

**2021**

**Szegedi Tudományegyetem**

**Informatikai Intézet**

# 3D túlélő játék fejlesztése Unity-ben

Szakdolgozat

Készítette:

**Potyesz Máté**

gazdaságinformatika szakos hallgató

Témavezető:

**Dr. Bodnár Péter**

egyetemi adjunktus

Szeged

2021

# Feladatkiírás

A 21. század informatikájában hatalmas nagy teret hódítottak a különböző játékszoftverek, melyeket számítógépen, konzolokon, mobilokon vagy egyéb eszközökön lehet játszani. A hallgató feladata egy háromdimenziós túlélő játék fejlesztése, mely során megismeri a korszerű fejlesztői eszközöket és egy komplex feladat keretében mélyíti el a programozási irányelveket. A játék funkciói többek között egy működő eszköztár, barkácsolás, karakter státuszpontok (életerő, éhség, szomjúság, energia, állóképesség), eszközök pályán lévő elhelyezése és velük való interakciója, nyersanyagok és eszközök szerzése objektumok kiütésével, mentési rendszer, idő rendszer, éjszaka és nappal váltakozása, mesterséges intelligencia, ellenfelekkel való küzdelem, pálya elkészítése a rendelkezésre álló modellekből, valamint már meglévő animációk implementálása és működésbe helyezése.

# Tartalmi összefoglaló

- ***A téma megnevezése:***

*3D túlélő játék fejlesztése Unity-ben.*

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

*A feladat egy olyan háromdimenziós túlélő játék fejlesztése PC-re, mely funkcióit, grafikáját és a játékélményét tekintve megfelel a jelenlegi játékpiacon elérhető alsókategóriás, indie játékoknak.*

- ***A megoldási mód:***

*A megoldáshoz Unity keretrendszer kerül felhasználásra. A grafikai elemek, animációk nagy része az asset store-ból kerül beszerzésre, mivel ezek bonyolultságuk okán nagyon megnövelnék a fejlesztésre fordítandó időt, miközben nem adnának ezzel arányos mennyiségű tapasztalatot. Az animációk implementálása Unity Animatorral történik, a saját modellek Blender-rel készülnek. A játék többi része (mesterséges intelligencia, hangok, játékfunkciók, stb.) szintén Unity segítségével valósul meg.*

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

*A fejlesztés Unity játékmotorban történik, C# programozási nyelven. A játékban lévő mentési rendszer megvalósításához JSON szöveg alapú szabvány kerül felhasználásra. A saját készítésű grafikai elemek Blender-ben készülnek.*

- ***Elért eredmények:***

*Egy szép grafikájú, megfelelő mennyiségű funkcióval rendelkező játékprogram, mely optimálisan és kiegyensúlyozottan fut a legtöbb számítógépen és akár feltölthető legnépszerűbb játékmegosztó oldalakra.*

- ***Kulcsszavak****:*

*Játékfejlesztés, C#, Unity, 3D, Blender*

# Tartalomjegyzék

# Bevezetés

A **játékfejlesztés** már három évtizedes múltra tekint vissza, és fejlődésük a kétezres évek óta jelentős növekedést mutatott. A videójátékokra használt elektronikai eszközök a „***platform***” megnevezésként ismertek, melyek két főcsoportja a személyi számítógépek, valamint az otthoni és játéktermes használtra szánt videójáték-konzolok. A feladatom egy személyi számítógépen játszható játék elkészítése volt.

A játékokat több tulajdonság alapján is besorolhatjuk különböző típusokba, legyen ez a műfaj, platform vagy a játékosok száma. A műfajokon belüli típusok megnevezésénél gyakran angol rövidítéseket használunk. Ilyen műfajok például: Akció, ügyességi, kaland, szerepjáték, stratégiai játék stb. Az akció műfaján belül megkülönböztetünk három típust is: *FPS*: (*First Person Shooter*) vagyis belső nézetű, *TPS* (*Third-person shooter*), azaz külső nézetű és oldalnézetű játékot. A szakdolgozatom műfaja **FPS** (belső nézetű) akciójáték. A játékos száma szerint is megkülönböztetjük a videójátékokat: *single player* (egyjátékos) és *multiplayer* (többjátékos). A játékom **single player**, azaz egyjátékos módban játszható. A videójátékokat irányíthatjuk valamilyen kontrollerrel vagy egér és billentyű-kombinációval. A játékomat egérrel és billentyűzettel lehet irányítani. Egy fontosabb típus, ami alapján megkülönböztetünk még játékokat, az a dimenzió száma. Létezik *2D* (2 dimenziós) *3D* (3 dimenziós), valamint a piac elkülöníti a *2.5D* játékokat is. A *2.5D* egy olyan játék, mely 3 dimenziós környezetet ábrázol, viszont *2D* játékmenetet foglal magában, vagy pont még azokat a játékokat is ide soroljuk, amik *3D* játékmenetet használ, de csak látszólag, mivel a *3D* modellek helyett 2 dimenziós úgynevezett *sprite-okat* használ. Általában a *3D* játékok igénylik a fejlesztés során a legtöbb időt, forrást, energiát, és tapasztalatot. A szakdolgozatomnak egy ***3D*** játékot készítettem el. Az utolsó megkülönböztetési típus, ami megemlítendő az a *3D* játékunk grafikájának minősége. Ezt angol szakszavakkal szokták megnevezni. Megkülönböztetünk *low-polly* és *high-polly* játékokat. A *low-polly* játékok alacsony számú poligont használnak a modellezéshez, míg a *high-polly* magas számú poligont. Értelem szerűen a *low-polly* játékprojektek kevesebb időt és erőforrást igényelnek, valamint az elkészült modellek mérete is töredéke a *high-polly* modelleknek. A játékomban ***high-polly*** *3D* modelleket használok, ezért a teljes projektfájl nagyméretű, valamint csak pár, egyszerűbb modellt készítettem (pl. csákány, balta) saját magamnak. A játékban előforduló többi modell és a hozzá tartozó textúra a *Unity Asset Store*-ból lett letöltve, melyek az irodalomjegyzékben megtalálhatók.

Szakdolgozatomhoz több, különböző típusú szakirodalmat használtam fel az egyetemen megszerzett tapasztalatok mellett. Az írott segédletekből legtöbbet a Unity dokumentációját, a Unity fórumát és a fejlesztők számára elkerülhetetlen weboldalt, a *Stackoverflow*-t használtam. Utóbbi két helyen a világ minden tájáról származó kezdő/tapasztalt fejlesztők tesznek fel kérdéseket, illetve segítenek egymásnak. Másrészt rengeteg videó formátumú tananyagot is megnéztem az elmúlt hónapokban, köztük nagy segítségemre vált pár *e-learning* tanagyag is (az irodalomjegyzékben is fel vannak tüntetve).

A szakdolgozatom felépítését tekintve úgy érzem először mindenképpen fontos szót ejteni a Unity sajátosságairól, elemiről. Ezt követően szeretném nagyon röviden bemutatni magát a játékot az olvasónak, melyet felhasználói dokumentációnak szokás nevezni (indítás, rendszerkövetelmény, beállítások, irányítás és a játék célja). A dolgozat lényeges részét képezi a játékfunkciók megvalósításának részletezése, másnéven a fejlesztői dokumentáció.

# 1. Unity játékmotor

A **Unity** egy *cross-platform* videójáték-motor, amelyet a *Unity Technologies* fejleszt. Segítségével háromdimenziós illetve kétdimenziós videójátékokat, ezen kívül egyéb interaktív jellegű tartalmakat lehet létrehozni. Támogatja a C, C++, C# és a Javascript programozási nyelvet, de az elsődleges nyelvnek a C#-pot tekinti. A Unity szerkesztője (*Unity editor*) támogatja Windows, macOS és Linux platformot, míg maga a motor jelenleg több mint 19 különböző platformon támogatja a játékok készítését, beleértve a mobil, asztali, konzol és virtuális valóság platformokat is.

## 1.1 A Unity Editor főbb elemei

A *Unity Editor* egyik legfontosabb eleme az ***Inspector*** ablak. Ebben az ablakban tudjuk szerkeszteni az adott játékobjektumunkat (hozzáadott *komponenseit* és azoknak a tulajdonságát). Az *Inspectorban* hozzáadhatunk *komponenseket* a kiválasztott objektumhoz.

A ***Scene*** (jelenet) ablakban láthatjuk az adott jelenetünket. Minden projekt áll egy vagy több *Scene*-ből, melyeket a *Build Settings* menüpontban tudunk rendszerezni.

A **Hierarchia** ablak tartalmazza az összes objektumot, ami az adott jelenetben szerepel. Ezek között az objektumok között pedig hierarchiát építhetünk fel.

A Unity egyik legnagyobb előnye a ***Play mode*** funkciója, vagyis a lájátszási módja. Ez lehetővé teszi a projekt futtatását már közvetlenül a szerkesztőben. Meg tudjuk nézni a játék futását a felhasználó szemszögéből, ráadásul közben szerkeszthetjük a projektet és folyamatosan figyelemmel követhetjük a változtatás eredményét. Fontos megemlíteni azonban, hogy a lejátszási mód alatt a szerkesztőben végzett módosítások a lejátszási módból való kilépéskor visszaállnak.

## 1.2 Objektumok

A ***GameObject*** a legfontosabb elem a Unity-ben. A játékban minden objektum egy *GameObject*, ami azonban önmagában nem tud semmit sem csinálni. Tulajdonságokkal kell ellátni, mielőtt karakterré, környezetté, speciális effektussá, vagy bármi használhatóvá válhatna. Úgy is mondhatjuk, hogy egyfajta tárolóként szolgálnak a komponensek számára, amelyek a funkciókat valósítják meg. Attól függően, hogy milyen objektumot akarunk létrehozni, a komponensek különböző kombinációit kell hozzáadni a *GameObject-hez*.

Az objektumokhoz erősen hozzátartozik a ***Prefab*** fogalma is. A Unity *Prefab* rendszere lehetővé teszi, hogy létrehozzunk, konfiguráljunk és tároljunk egy *GameObject-tet* az összes komponensével, tulajdonságértékével és gyermek *GameObjectjével* együtt. Ha egy bizonyos módon konfigurált *GameObject-tet* több helyen is fel szeretnénk használni a jelenetben (*scene*), vagy a projekt több jelenetében is, akkor azt *Prefab-bé* érdemes alakítani. Ez sokkal jobb, mint a *GameObject* egyszerű másolása és beillesztése, mert a *Prefab* rendszer lehetővé teszi, hogy az összes másolatot automatikusan szinkronban tartsa. Azaz ha a színtérben az adott *prefab-ból* több is el van helyezve és mi módosítjuk a *Prefab-et*, akkor azonnal végrehajtódik a jelenetben elhelyezett példányokon is a változtatás.

## 1.3 Komponensek

A már említett *GameObject*-ek viselkedését a komponensek határozzák meg. A komponensekkel való kapcsolatba lépés kétféle képpen történhet. A szerkesztőben hozzáadhatunk és beállíthatunk komponenseket egy adott objektumnak. A másik lehetőségünk, hogy egy adott szkriptben hozzuk létre, vagy módosítjuk azt. A Unity-t komponens alapú rendszernek is tekinthetjük.

A legalapvetőbb komponens a ***Transform***. Ez alapértelmezett minden objektumnál és nem is tudjuk törölni. Ez határozza meg az adott objektum pozícióját, forgását, méretarányát. Ezeket X, Y és Z koordináták alapján tudjuk módosítani.

A második legfontosabb komponens, ami nélkül lényegében nincs is játék, a ***Camera*** (kamera komponens). A Camera komponenssel az objektumunk egy kamera lesz a színtérben és több tulajdonságát is beállíthatjuk az *editorban*. A kamera segítségével tudunk látni a játékban.

A ***Collider*** komponens felruházza az objektumot a más objektumokkal való érintkezésre, ütközésre. A *2D* játékokban ez a *Collider2D* komponens, a *3D* játékokban kicsit nehezebb a dolgunk, hisz több is van belőle: *CapsuleCollider* (kapszula alak), *SphereCollider* (gömb alak), *MeshCollider* (a bonyolultabb objektumokra szokás rakni, ugyanis automatikusan felveszi az objektum alakját) és *BoxCollider* (kocka alak).

Ha a *Colliderrel* rendelkező objektumot a játék során mozgatni kell, akkor egy ***Rigidbody*** komponenst is csatolni kell az objektumhoz. Ez a komponens tulajdonképpen lehetővé teszi, hogy a gravitáció hathasson az objektumra és fizikai kölcsönhatásba léphessünk más objektumokkal.

A játékunk egyik legfontosabb eleme az animáció lesz, amit az ***Animator*** komponens segítségével tudunk hozzáadni egy objektumhoz. Az Animator komponens vár egy *AnimatorControllert*, ahol definiáljuk, hogy milyen animációkat fogunk használni, és hogy ezek hogyan, milyen hatásra játszódhatnak le.

Az utolsó fontos komponens amit említeni szeretnék az a *Mesh*. Erről oldalakat lehetne írni, de röviden ez egy pontokból vagy csúcsokból álló háló. A lényeg viszont az, hogy Unity-ben két elsődleges *renderelési* komponens van: A hálószűrő (***Mesh Filter***), amely a modell hálóadatait tárolja, és a háló renderelő (***Mesh Renderer***), amely a háló adatokat anyagokkal (*Material*, amikhez pedig *Texture* van hozzáadva) kombinálja az objektum rendereléséhez Unity-ben.

### 1.4 MonoBehaviour

A Unity legfontosabb, beépített osztálya. A *MonoBehaviour* az az alaposztály, amelyből minden Unity szkript származik. C# nyelv használata esetén kifejezetten a *MonoBehaviour*-ből kell származtatnunk. Egy C# szkript létrehozásakor láthatjuk, hogy az osztályunk automatikusan örököli a *MonoBehaviour* osztályt. Több fontos metódusa van, melyek ismerete elengedhetetlen a Unity alapszintű használatához is, ezek közül ismertetek párat.

A **`Start()`** a script meghívásakor egyszer hívódik meg (legelőször), még az Update() metódus előtt, a játék indításakor.

Az **`Update()`** metódus a script meghívása után minden képkockakor lefut. Nagyon fontos a képkocka kifejezés, hisz nem időegységenként fut, számít tehát az adott számítógép teljesítménye. Ezáltal a fejlesztőnek oda kell figyelni arra, ha valamit időegységenként akar kezelni.

Az **`OnTriggerEnter()`** érzékeli, ha az objektum érintkezik egy másik objektummal. Az **`OnTriggerExit()`** pedig azt érzékeli, ha már nem érintkeznek egymással.

A **`Destroy()`** statikus metódussal törölhetünk meglévő objektumokat. Szintén statikus metódus még a `FindObjectOfType()` metódus, mely visszatér az első megadott típusú objektummal, ami aktív a színtéren.

# 2. Felhasználói dokumentáció

A játék neve ***Zenit Survival***, melyet Windows operációs rendszerrel rendelkező számítógépen futtathatjuk a *ZenitSurvival.exe* fájllal.

## 2.1 Gépigény

- Véleményem szerint a **minimum gépigény** (nem feltétlen fut maximum grafikán, de érdemes kipróbálni): *I3*-as processzor, *2GB* memóriával rendelkező VGA, *4GB* RAM.
- Egy becsült **ajánlott gépigény**: újabb típusú *I5* processzor, *4GB* memóriával rendelkező VGA, *8GB* RAM.
- Saját számítógép (laptop), melyen a fejlesztés és tesztelés történt (átlagosan 60 FPS): Intel I7-10750H 2.6 GHz processzor, NVIDIA GeForce GTX 1660 Ti videókártya, 16 GB RAM.

## 2.2 Irányítás

A karakterünket a *w/a/s/d* vagy a nyíl billentyűkkel tudjuk irányítani. Az *I* billentyű lenyomásával tekinthetjük meg az eszköztárunkat és a barkácsolási felületet mellette. A *bal shiftet* lenyomva tudunk futni. A *bal egérgomb* lenyomásával tudunk ütni, lőni, az adott fegyverrel támadni, fát vágni, dobozokat/hordókat kiütni és bányászni. A pályán lévő *itemeket* (tárgyakat) az *E* gomb lenyomásával vehetjük fel, valamint ugyan ezzel a billentyűvel interakcióba lépni az adott eszközökkel, vagy a kereskedővel. A *Q* gomb lenyomásával rúgni is tudunk. A *jobb egérgomb* a zseblámpa be/ki kapcsolására, az öngyújtó ki/begyújtására való. A *space* billentyű lenyomásával ugrani tud a karakterünk. Az eszközöket úgy tudjuk használni, kézbe venni, hogy először a *hotbarba*, vagyis a képernyő alján található eszköztár egyikébe (1-8) rakjuk őket, majd az *alfanumerikus számbillentyűk* közül azt a számot nyomjuk meg, amelyik számú helyen van az eszköz.

## 2.3 Játékleírás

A **játék célja** minél tovább túlélni egy elhagyatott szigeten. A játék során a feladatunk, hogy minél több *itemet* (tárgyat) szerezzünk, melyekkel túl tudunk élni minél több napot.

### 2.3.1 Életerő stabilizálása

Ha a karakter életerője 0-ra csökken, a játéknak vége. A szomjúság, éhség és energia érték folyamatosan csökken, ez a csökkenés futáskor sokkal gyorsabb. Ha bármelyik érték eléri a 0 értéket, akkor az életerő is elkezd csökkenni.

Az **éhséget** ételekkel lehet csökkenteni, pontosabban növelni az éhség státusz értékét. Ételt szerezhetünk állatoktól (húsok), a hordókból[1], dobozokból, földről (gomba), fáról (alma, banán), egy kereskedőtől, stb. Az egyes ételeket, például a húsokat meg is tudjuk sütni tábortűzön, ezáltal több életerőt töltünk vissza. A **szomjúságot** vízzel, valamint energiaitallal tudjuk visszatölteni. Az **energiaszintünket** alvással tudjuk feltölteni, amihez egy sátrat kell barkácsolnunk kőből[2] és fából, de az energiaital is segít a visszatöltésben. Az **életerőt** gyógyszerrel és kötszerrel tudjuk visszatölteni.

### 2.3.2 Tárgyak szerzése és barkácsolás

A játék során először a földről kell összegyűjtenünk megfelelő mennyiségű követ és fát. Ezekből baltát és csákányt is barkácsolhatunk. A baltával **fát vághatunk** a kőcsákánnyal pedig köveket és vasat. A kibányászott vasat elvihetjük egy kemencébe, amit elhagyatott faházakban lehet találni. Megfelelő mennyiég kiégetése után vascsákányt készíthetünk belőle. A pályán található bronz, ezüst és arany érc[3], melyeket vascsákánnyal tudunk **kibányászni** és szintén a kemencében tudjuk kiégetni őket. A kiégetett ércekből bronz, ezüst és arany érmét tudunk készíteni, amikkel tudunk **vásárolni** a kereskedőtől különböző ritkább tárgyakat. Vasból és fából lehet kést, mint fegyvert **barkácsolni**. A pályán találhatunk elemlámpát, melyet tudunk használni, ha elegendő elemünk van hozzá. Pisztolyhoz is juthatunk, melyhez tárra és lőszerekre van szükség.

### 2.3.3 Harcrendszer

A játékban találhatók **barátságos** és **agresszív** ellenfelek. A barátságos ellenfelek állatok, amelyekhez ha közel kerülünk, akkor elfutnak. Ilyen például az őz, szarvas, nyúl vagy disznó. Az agresszív állatok, valamint az őslakosok pedig nekünk támadnak ha közel kerülünk hozzájuk. Ilyen állatok a medvék vagy farkasok. Ezek az ellenfelek különböző értékű életerővel és sebzéssel rendelkeznek. Védekezni ellenük pedig kézzel, baltával, csákánnyal, késsel vagy pisztollyal lehet, de ezek természetesen más sebzéssel rendelkeznek.

[1] https://assetstore.unity.com/packages/3d/environments/industrial/barrels-63623 - Elérési idő: 2021.11.26
[2] https://assetstore.unity.com/packages/3d/environments/landscapes/rocky-hills-environment-light-pack-89939 - Elérési idő: 2021.11.26
[3] https://assetstore.unity.com/packages/3d/props/exterior/mining-ore-ingots-minerals-kit-26357 - Elérési idő: 2021.11.26

# 3. Játékfunkciók megvalósítása

A játék funkcióinak megvalósítására C# programozási nyelvet használtam. A legtöbb játékszoftverhez hasonlóan ez is 3 rétegre bontható: felhasználói felület, üzleti logika, adatbázis réteg. A **felhasználói felüle**t a felhasználókkal való kommunikációért felelős. Unity-ben az *EventSystem* valósítja meg a felhasználói interakciók és a vizuális elemek közötti interakciókat. Az **üzleti logika** (játék logika) az ebben a szekcióban leírt játék funkcionalitást végzi. Ilyenek többek között a C# szkriptek melyek a játékvezérlésért felelősek. Az **adatbázis rétek** a játékban lévő adatok tárolásáért felelős. Unity-ben általában a *JSON* szöveges fájlformátumot használnak erre, így én is ezt a módszert választottam.

## 3.1 Irányítás

Az elkészített játékom belső nézetű (*FPS*), azaz a karakter szemszögéből láthatunk. A mozgás során a karakterünk lábait és a kezeit láthatjuk és ezekre érvényesek a mozgási animációk is.

### 3.1.1 Mozgás

A karakter mozgási funkciója az **`AgentMovement`** osztályban lett megvalósítva. Két mozgási lehetőség van: **gyaloglás** és **futás**. Futni a *bal shift* lenyomásával tudunk, ekkor a mozgási sebesség megnő, ugrani pedig a *space* billentyűvel tudunk. A mozgásért a **`MovePlayer()`** *metódus* felel. Először az *input* irányokat kérem be vertikálisan és horizontálisan, majd az irányt határozom meg ezek alapján.

```
float vertical = Input.GetAxis("Vertical");
float horizontal = Input.GetAxis("Horizontal");
moveDirection = transform.forward * vertical + transform.right *
horizontal;
moveDirection *= currentSpeed;
```

A *`moveDirection`* változó lesz az irány, amerre a karakterünk haladni fog. A `transform.forward` a már megadott vertikális irány felé fog menni a `transform.right` pedig a horizontális irány felé.

```
float currentSpeed = isWalking ? walkingSpeed : canPlayerRun ?
runningSpeed : walkingSpeed;
```

Az aktuális sebességhez meg kell nézni, hogy a karakterünk éppen sétál-e. Ha sétál, akkor a gyaloglási sebesség lesz az érték, ha nem, akkor pedig a futási sebesség. A kettő értékhez azonban meg kell néznünk, hogy tud-e a karakterünk futni, ez lesz a

`canPlayerRun` bináris változó. Ezt majd a játékos státuszánál fogjuk változtatni, hogy az aktuális állóképességszintje alapján képes-e futni vagy sem. Hogy sétálunk-e vagy sem, azt az inputok vizsgálatával tudjuk megnézni.

```
bool isWalking = true;
if ((Input.GetKey(KeyCode.W) || (Input.GetKey(KeyCode.S)
|| (Input.GetKey(KeyCode.A) || (Input.GetKey(KeyCode.D)))))
&& moveDirection != Vector3.zero &&
Input.GetKey(KeyCode.LeftShift))
{
isWalking = false;
}
else if((Input.GetKey(KeyCode.W) || (Input.GetKey(KeyCode.S) ||
(Input.GetKey(KeyCode.A) || (Input.GetKey(KeyCode.D)))))
&& moveDirection != Vector3.zero)
{
isWalking = true;
}
```

A metódus végén pedig megtörténik maga a valós idejű mozgatás. A karakterünkre rakott *characterController* komponens beépített `Move()` metódusa végzi ezt el. Fontos, hogy a meghatározott `moveDirection` változót meg kell szorozni a `Time.deltaTime`-mal, ez ugyanis valós időben fogja mozgatni a karaktert. Ez azért kell, mert az `Update()` függvényben hívjuk a metódust és az képkockánként hívódik meg.

```
characterController.Move(moveDirection * Time.deltaTime);
moveDirection.y -= gravity * Time.deltaTime;
```

Ugrásnál a *Space* billentyű lenyomását figyeljük és az *y* irányban mozgatjuk a karakterünket egy adott, előre meghatározott értékkel.

### 3.1.2 Kamera

Amit a játékban látunk a karakter szemszögéből, azért a *MainCamera* objektum *Camera* komponens felelős és természetesen az egérrel tudjuk irányítani, mint az összes *FPS* játékban. Ezt a **`MouseLook`** osztály valósítja meg.

```
float mouseX = Input.GetAxis("Mouse X");
float mouseY = Input.GetAxis("Mouse Y");
xRotation -= mouseY;
xRotation = Mathf.Clamp(xRotation, -50f, 55);
transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
playerTransform.Rotate(Vector3.up * mouseX);
```

Az x és y tengelyen tudunk forgatni, de az x tengelyen csak két bizonyos szög között, hogy élethűbb legyen. Az első négy sor ezért lesz felelős. A kamera forgatásához pedig a `Quaternion.Euler` beépített függvényt hívtam segítségül, ami behatárolja a

forgási szöget, majd a *Transform* komponens `Rotate()metódusa` végzi el a tényleges forgatást a *mauseX* paraméter alapján.

### 3.2 Karakter státusz

A karakterünknek az éhség, szomjúság, energia, állóképesség (*stamina*) pontjaiért a `PlayerStats` osztály felel. A `PlayerStatsBarChanger` szkript pedig ezeknek az értékeknek az ikonjainak a megjelenítéséért és módosításaiért felelős. Többek között az életerő működését valósítja meg a `HealthScript` osztály. A **`PlayerStats`** osztályból három értéket elég csupán az éhségre bemutatni, hiszen a másik kettő, vagyis a szomjúság és az energia statisztika teljesen hasonlóan valósult meg. Eltároljuk először az aktuális értéket, valamint a maximális értéket. Ezen kívül pedig egy nem egész számot, amivel a játék során az adott statisztika csökkenni fog. Továbbá szükségünk lesz egy szöveges objektumra, ami majd megjelenik a képernyőn az adott érték formájában.

```
public float hunger = 5;
private float hungerNormal;
private float hungerFallRate = 0.1f;
public Text hungerText;
```

Ha fut a karakter, akkor az éhség egy bizonyos értékkel gyorsabban csökkenti a `hungerFallrate`-et, azaz az éhségcsökkentést.

```
hunger -= Time.deltaTime * hungerFallRate * 1.3f;
```

Ellenkező esetben pedig folyamatosan csökken a `hungerFallrate` értékével szépen lassan, amíg persze el nem éri a 0 értéket. Azt is figyelni kell, hogy az aktuális pontszám ne legyen nagyobb, mint a maximális, hiszen ezeket az értékeket a játék során tudjuk majd növelni is. A szöveges változónak pedig a `hunger` értéket adjuk értékül és így fog megjelenni a képernyőn.

A játékos *staminája* folyamatosan csökken futás közben. Ha eléri a minimumot, akkor nem tud tovább futni, kifárad, amit hanggal is jelez és meg kell várni amíg visszatölt, hogy újra tudjunk futni. Ez tulajdonképpen egy *Slider*-en fog megjelenni, ami egy Unity *UI* elem, magyarul egy csúszka, ami az aktuális értéket kitöltéssel mutatja.

Az `UpdateStamina()` elosztja az aktuális *staminát* a maximális *staminával*, így kapva egy értéket, amit ki kell tölteni.

```
void UpdateStamina(int value)
{
      staminaProgressUI.fillAmount = stamina / staminaNormal;
      sliderCanvasGroup.alpha = 1;
}
```

Ezek után már csak az a dolgunk, hogy a metódusban lévő `staminaProgressUI.fillAmount`-ot csökkentsük ott, ahol a futást érzékeltük. Ha pedig nem történik futás és még nem értük el a maximális *staminát*, akkor egyszerűen növeljük ezt az értéket. Itt kell meghívnunk az `UpdateStamina()` metódust is.

*3.1. ábra - Játékos stamina*

A játékos életerő értékéért a **`HealthScript`** osztály felelős. Ha a játékos energiája, éhsége vagy szomjúsága nullára csökken, akkor az életerőt egy adott értékkel folyamatosan csökkentem addig, amíg vissza nem töltődik minden 0 fölé.

```
if (playerStat.hunger <= 0 || playerStat.thirst <= 0 ||
playerStat.energy <= 0)
{
      if(health < 0)
          playerStat.Display_HealthStats(0);
      else
      {
          health -= Time.deltaTime * 0.5f;
          playerStat.Display_HealthStats(health);
       }
}
```

A `PlayerStat` scriptben szerepel még egy `Display_HealthStats` nevű metódus is, ami annyit csinál, hogy a játékos életerőjét értékül adja az ahhoz tartozó Text változónak és így kiíródik az. A `HealthScript`-ben valósul meg az is, hogyha a játékosunk életerője 0, akkor a játéknak vége lesz, és a *DeathMenu* színtér jelenik meg.

## 3.3 Eszköztár

Az *Inventory System*[4] az egyik legbonyolultabb funkció a projektben, ezért megpróbáltam valamilyen forrás után nézni. Rátaláltam egy fizetős külsőnézetű, *low-polly* mini túlélőjáték elkészítését bemutató, *tutorial* sorozatra. A *Sunny Vally Studios Udemy* fizetős kurzusa[5] nagyon sokat segített a projektben lévő *Inventory System* és *Placement System* elkészítésében, hisz mint később kiderült, ezek elég komplex rendszerek.

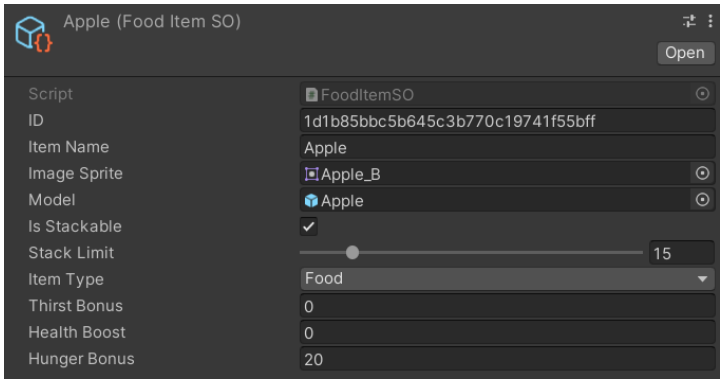
A komplexitásól adódóan csak a fontosabb, lényeges részekre térek ki. Az osztályokat külön mappákba rendeztem úgy, mint az összes többi kódot a projektben, így átláthatóan elkülönülnek logikájuk és funkcionalitásuk szerint.

[4] Eszköztár rendszer
[5] https://www.udemy.com/course/unity-2019-make-a-3d-survival-game/ - Elérési idő: 2021.11.25

### 3.3.1 Tárgytípusok

A tárgytípusokra vonatkozó osztályok az **ItemType** mappában találhatók. Ezek lesznek felelősek azért, hogy a különböző tárgyak értékeit be tudjuk állítani az *editorban*. Ezért a ***ScriptableObject*** beépített osztály lesz a felelős, ebből fogjuk származtatni az itteni fő osztályunkat, az **`ItemSO`**-t. Az összes többi osztály, amik az *item* típusok osztályai, ebből a főosztályból fognak származni.

A *ScriptableObject*[6] egy olyan adattároló, amely nagy mennyiségű adat mentésére használható, az osztálypéldányoktól függetlenül futási idő közben. Ezek olyan objektumok, amik kizárólag adatok tárolására használhatóak.




*3.1. ábra - Étel típusú ScriptableObject*

Az **ItemSO** szkriptben történik az eszközök beazonosítása a már említett *ScriptableObject* segítségével. Ezzel tudunk egy új tárgyat létrehozni, nevet adni neki, megadni a képét, ami majd megjelenik az eszköztárban, megadni a modellt, a *stacklimitet*[7] és azt, hogy *stackelhető*-e. Egy *enum*[8]-ba pedig az összes tárgytípust megadtam, majd csináltam mindegyikből egy osztályt ami örökölni fogja az *ItemSO*-t és csak az adott tárgytípus tulajdonságait adom meg ott. Példaként `FootItemSO` tartalma:

```
public int thirstBonus = 0, healthBoost = 0, hungerBonus = 0;
```

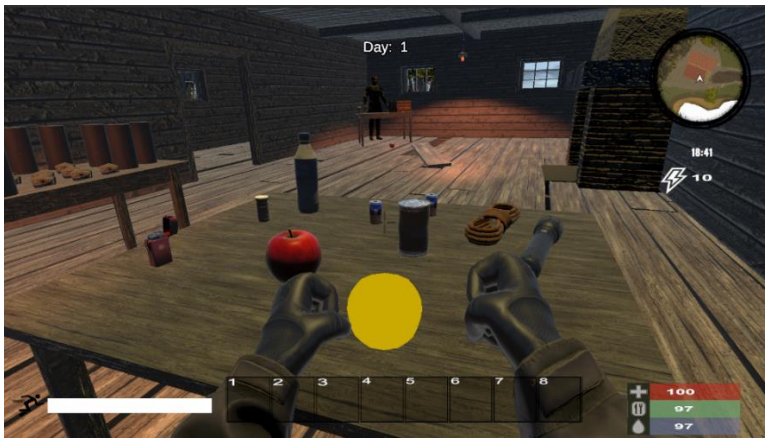
### 3.3.2 Tárgyak felvétele a pályáról

A ***PickingItems*** mappában lévő kódok azért felelősek, hogy a pályán található tárgyakat fel tudjuk venni és azok bekerüljenek az eszköztárunkba. A **`PickableItem`** szkriptet kell csatolni az adott tárgy objektumhoz komponensként. A komponens vár egy *ScriptableObject* típusú tárgyat és annak az *ID*-jét, valamint a darabszámát fogja nézni.

---

[6] https://docs.unity3d.com/Manual/class-ScriptableObject.html - Elérési idő: 2021.11.25
[7] Halmozható eszközöknél a maximális halmozható érték az adott eszközből egy adott eszköztárhelyen.
[8] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum - Elérési idő: 2021.11.25

Tehát már tudjuk, hogy az adott objektum amit felveszünk milyen tárgy lesz és mennyit kapunk belőle. A tárgy felvételéért a **`DetectionSystem`** osztály felel, azon belül is a fő metódus a `PerformDirection()` lesz. Itt a karakterünkre egy úgynevezett érzékelőt rakunk, aminek a segítségével a körülöttünk lévő objektumok *Collidereit* figyeljük folyamatosan. Ha szeretnénk, hogy az objektum felvehető legyen, akkor a *Layer*-ét[9] *Interactible* típusúra állítjuk. Ha egy *objektumnak Interacible Layere van*, akkor az objektum színét beállítja sárgára, jelezve, hogy ezt az objektumot fel tudjuk venni. A színváltoztatásért a `SwapOriginalMaterial()` és `SwapSelectedMaterial()` függvények felelősek.




*3.2. ábra - Tárgy kijelölése felvétel előtt*

A ***State*** mappában lévő `InteractState` osztály felelős azért, hogy az adott tárgy a tárgyhelyünkre is bekerüljön. Az `AddToStorage()` függvény hozzáadja az eszköztárunkhoz az adott tárgyat. Az **`IPickable`** névtérben található egy `IInventoryItem` típusú **`Pickup()`** metódus. Az IInventoryItem névtérben pedig egy *item* típus változói találhatóak: *`ID`*, *`count`*, *`IsStackable`* és `StackLimit`, amiket értékül adunk a felvett *item* értékeivel.

```
var resultCollider =
controllerReference.detectionSystem.CurrentCollider;
if (resultCollider != null)
{
      var ipickable = resultCollider.GetComponent<IPickable>();
      var remainder =
      inventorySystem.AddToStorage(ipickable.Pickup());
      ipickable.SetCount(remainder);
      if (remainder > 0)
          Debug.Log("Can not pick it up!");
 }
```

[9] A Unity-ben a rétegek határozzák meg, hogy mely *GameObject-tek* léphetnek kapcsolatba különböző funkciókkal.

### 3.3.3 Tárgyak eltárolása

A **`Storage`** mappában lévő kódok fontos metódusait felsorolásszinten fogom elmagyarázni. Tömören az itt lévő osztályok felelősek azért, hogy az adott eszköz bekerüljön az eszköztárunkba és azért, hogy az adott eszközöket tudjuk is használni. A megjelenítés és a tényleges tárgytárolás tehát elkülönül egymástól.

Az **`InventorySystemData`** osztály az *itemek* törléséért, kiválasztásáért, cseréjéért, eltárolásáért, stb. felelős adatszinten. Néhány fontos metódus:

*`AddToStorage(IInventoryItem item)`*: A megkapott *itemet* tárolja el. Megnézi, hogy van-e már ilyen *item* a *hotbarban* és az *inventory*ban. Ha van mindkettőben, *stackelhető* és még nem érte el az *item* maximális *stackelhetőségi* értékét, akkor hozzárakja a többihez az adott *invetory* helyen. Ha csak a *hotbarban* van, akkor ugyan ezeket a lehetőségeket nézve helyezi el a *hotbarban* az *itemet*. Ha egyik helyen sincs, akkor az *inventory* legelső szabad helyére rakja, az `AddItem()` metódus segítségével.

*`EqupItem()`* és *`UnEquipItem()`*: Még nem volt szó azokról az eszközökről, amiket a kezünkbe vehetünk. A metódus lényegében annyit csinál, hogy az osztályban már inicializált `equippedItemStorageIndex` változó értékét állítja be arra az indexre, amelyik *hotbar* indexen van az a tárgy, amit használni szeretnénk. Az `UnEquipItem()` pedig szimplán -1 értéket ad neki, azaz jelenleg nincs *item* felszerelve.

*`SwapStorageItemsInsideInventory()`*,*`SwapStorageItemsInsideHotbar()`*,*`SwapStorageHotbarToInventor()`*,*`SwapStorageInventoryToHotbar()`*: A metódusok paraméterei: `int droppedItemID` (a cserélendő *item*), `int draggedItemID` (az éppen cserélni kívánt, egérrel megfogott *item).* Az első metódus az inventoryban, a második a hotbarban történő két item közötti cseréért, a harmadik metódus a hotbar és inventory, a negyedik pedig az inventory és hotbar közötti item cseréért felelős. A cserét a `Storage` osztályban lévő `SwapItemWithIndexFor()` végzi. Paraméterében kéri a cserélendő és cserélni kívánt tárgyakat.

A **Storage** osztályban is vannak fontos metódusok:

*`AddItem(string ID, int count, bool isStackable = true, int stackLimit = 100)`*: Ez a metódus egy új *itemet* ad hozzá a legelső szabad indexű *inventoryra,* vagy *hotbarra*. A `TryAddingToAnExistingItem()` metódus nézi meg azt, hogy hozzá tudunk-e rakni egy már meglévő ilyen itemhez. Ez a metódus

visszaadja azt a számot, ami már nem fér el az adott helyen és új helyre kell rakni a maradék *itemet*. Ebben esetben keresünk egy szabad helyet és ott az `Add(newStorageItem)` metódussal új elemet adunk a `storageItem`[10] listához.

`TakeItemFromStorageIfContainsEnough(string ID, int quantity)`: Ez a metódus megnézi, hogy van-e a paraméterben kapott *ID-jű* (egy random *string*) itemből elég mennyiség. Egy *for* ciklusban addig hívjuk a már említett `TakeFromItem()` metódust, amíg el nem fogy a darabszám, közben pedig folyamatosan ellenőrizzük, hogy van-e még elég *itemünk*. Csak akkor tér vissza igazzal, ha sikeresen lezajlott a `quantity` mennyiségű tárgycsökkentés, azaz volt elég *itemünk*.

A **StorageItem** osztályban lévő metódusok egy adott *item* műveleteiért felelnek.

*`AddToItem()`*: Hozzáad az itemhez, ha pedig már nem fér el, akkor visszatér a maradék számmal, nullával ha nem marad és sikeresen hozzáad mindent egy helyre.

*`TakeFromItem()`*: Az adott *item* darabszámát, a paraméterben megadott darabszámmal csökkenti. Visszatér az értékkel, amennyi el lett véve.

### 3.3.4 Eszköztár végleges megvalósítása és megjelenítése

Az *inventory UI* részéért az **Inventory** mappában található scriptek felelősek. Az **InventoryItemPanelHelper** osztály az inventoryban és a hotbarban lévő itemek megjelenítéséért, törléséért, és az itemek közötti cserének megjelenítéséért felelős.

*`SetItemUI(string name, int count, Sprite image)`:* Az item paraméterben kapott értékeit fogja megjeleníteni az inventoryban vagy a hotbarban. A metódus beállítja az item nevét, képét, darabszámát a paraméterben megadottakra. A `ModifyEquippedIndicatorAlpha()` metódus segítségével pedig az adott *itemslotot*[11] megváltoztatja, ha az ott lévő tárgy fel van éppen szerelve. Ez az utóbbi metódus annyit csinál, hogy az *itemslotra* ráhelyezett indikátor alfa értékét állítja 100%-ra (látható) vagy 0%-ra (nem látható), így be ki tudjuk kapcsolni az adott keretet amit a *hotbarslotok* köré raktam.

[10] A tárgyak, amik már el vannak éppen tárolva.
[11] Az adott indexű hotbar hely


*3.3 ábra - A hotbarban lévő csákány kijelölése felszerelt állapotban*

`OnPointerClick(), OnDrag(), OnBeginDrag(), OnEndDrag():` Ezek a metódusok aktiválják az adott Action típusú *Callbackeket*[12]. Az első metódust akkor hívjuk meg, mikor rákattintunk egy *itemre*, a másodikat mikor éppen egy *itemet* húzunk valamerre az egérrel, a harmadikat a megfogás előtt, az utolsót pedig lerakáskor.

Az **`InventoryPanelIsOn`** osztályom felelős azért, hogy megnézze van-e valamilyen eszköz felszerelve. Ha nincs, akkor azt a *prefab-ot* aktiválja, amiben nincs semmi a kezünkben. Megnézi azt is, hogy valamilyen panel aktiválva van-e, és ha igen akkor letiltja az animációkat, hogy ne tudjunk ütni pl. barkácsolás közbeni kattintáskor.

A legfontosabb osztályunk a ***Controllers*** mappában található **`InventorySystem`** osztály. Ez a *szkript* az összes eddig leírt metódust felhasználja.

`RemoveItemFromInventory(int ui_id)`: A törlésért felelős metódusokat hívja meg. Ha a paraméterben kapott tárgy valahol létezik az *inventoryban*, akkor töröli azt, majd megnézi, hogy véglegesen töröltük-e. Ha igen akkor kitöröli a *UI*-ból az *itemet*, ha nem akkor csak frissül az eszköz darabszáma.

`HotbarShortKeyHandler(int hotbarKey):` Ez a metódus az adott *hotbar* indexet hozzárendeli az ottani *itemekhez*. A megnyomott *hotbarkey* (*1-8*) még nem az index lesz, ahhoz ki kell vonnunk 1 et a megnyomott számból.

`UseItem(ItemSO itemData, int ui_id)`: Ez a metódus felel azért, hogyha megnyomjuk az adott *hotbar* számát, akkor megjelenjen az *item prefab*-je a karakterünknél. Ha már használatban van az *item*, akkor Az `UnequpItem()` metódust hívom meg, ha pedig nincs akkor az `EquipItem()` metódust. A `CreateItemObjectInPlayerHand()` metódus egy meghatározott pozícióba fogja

[12] http://codesaying.com/action-callback-in-unity/ - Elérési idő: 2021.11.25

megjeleníteni az *item prefabját*. A `RemoveItemFromPlayerHand()` töröli azt az adott *prefabot* ami éppen meg van jelenítve.

*`PutDataInUI():`* A metódus, ami *for ciklussal* végighalad az összes *inventory sloton* és az ott lévő *itemek* adatait (név, darabszám, kép) inicializálja a már eltárolt *itemData* adataival.

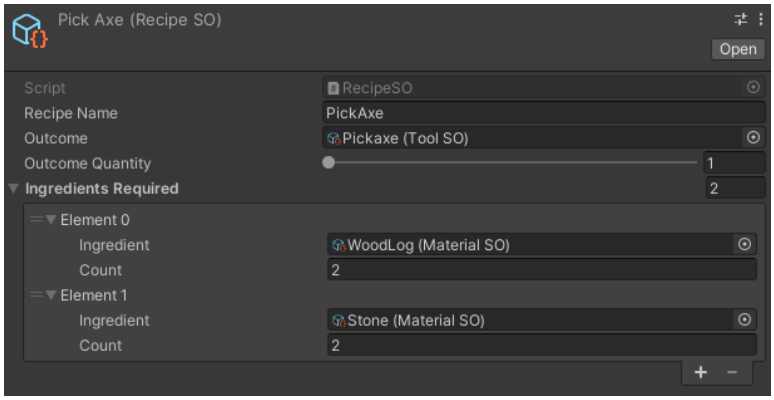
*`DropHandler():`* Végre meg is tudjuk valósítani az *itemek* cseréjét és húzgálását az *inventoryban* és a *hotbarban*, vagy éppen a kettő között, hiszen már minden metódus a rendelkezésünkre áll. Megnézzük, hogy mikor kattintottunk egy *itemre*, akkor az a *hotbarban* vagy az *inventoryban* van-e. Példaként legyen most ez *inventory*. Ekkor jön egy újabb `if()` ág, ami most azt nézi meg, hogy amikor letesszük azt az adott *itemet*, akkor hova tesszük azt le. Ezek alapján hívjuk meg a már fent említett 2 lehetséges metódusokat (vagy csere két *inventory item* között, vagy *inventory* és *hotbar* között, ha inventoryban volt az eszközünk).

## 3.4 Barkácsolás

A lényeg az *CraftingSystem* mappában lévő osztályokban van. Fontos kijelenteni, hogy erre a rendszerre 4 funkciót is kiépítettem. Az ételek sütése, az ércek kiégetése, a tárgyak megvásárlása és a barkácsolás is ugyan azzal a logikával és kóddal működik. Ez alapján a *3.4.1*-es pontban fogom elmagyarázni a rendszert, a maradék háromban pedig csak képekben mutatom meg az adott funkciót, mivel a megvalósítási mód megegyező.

### 3.4.1 Eszközök készítése

A **`RecipeSO`** osztály egy már említett *ScriptableObject* típust hoz létre, azon belül is egy barkácsolható *item* típust. Egy ilyen típusnak 4 adattagja lesz: recept név, a kapott mennyiség, a kapott tárgy, és az *itemek*, amikre szükségünk van a barkácsoláshoz. Egy szótárban tároljuk ezeket az adatokat ugyanúgy, mint az *Inventorynál*.




*3.4. ábra - Craftolható ScriptableObject típusú item*

Az **`IngredientUIElement`** osztály felelős az *itemek* megjelenítéséért a barkács *panelben*. Értéket ad az itemeknek, valamint van egy metódusa ami azért felelős, hogy módosítsa a hozzávalókban lévő *itemek* hátterét, ami azért kell, hogy lássuk, van-e elegendő mennyiség az adott hozzávalóból van sem (piros háttér, ha nincs elég itemünk).

A **`UICrafting`** osztály összeköti a barkácsolás felhasználói felületet az adat résszel:

*`AddIngredient(string ingredientName, Sprite ingredientSprite, int ingredientCOunt, bool enoughItems)`*: A paraméterben kapott értékeket fogja megjeleníteni a hozzávalók panelen.

*`PrepareRecipeItems(List<RecipeSO> listOfRecipes)`*: Egy listával tér vissza, amely a játékban lévő recepteket tartalmazza. Egy *for* ciklusban végig lépkedünk a paraméterben kapott lista elemein és a szükséges értékeket inicializáljuk.

A **`CraftingSystem`** osztályban lévő `ReciepeClickedHandler(int id)` metódus felel a barkácsolásért. A paraméterben a szükséges recept ID-t várja. A metódusban lévő *for* ciklus lesz felelős azért, hogy megnézze, van-e elég a szükséges *itemből* az eszköztárunkban a barkácsoláshoz. A hozzávalókat egy szótárban tároljuk és az elemein lépkedünk egy for ciklusban. Az `enoughItemFlag` egy logikai változó és az `onCheckResourceAvailability()` igazra állítja, hogyha az adott hozzávalóból van elég az eszköztárunkban. A `blockCraftButton` logikai változónak ez alapján fogunk értéket adni, ami azért lesz felelős, hogy a barkácsolásra szolgáló gombot aktiválja, vagy ne. Az `AddIngredient()` metódus pedig megjeleníti a hozzávalókat.

```
foreach (var key in ingredientsIdCountDict.Keys)
{
      bool enoughItemFlag =
      onCheckResourceAvailability.Invoke(key,
      ingredientsIdCountDict[key]);
      if(blockCraftButton == false)
            blockCraftButton = !enoughItemFlag;
      uiCrafting.AddIngredient(ItemDataManager.instance.GetItemNa
      me(key), ItemDataManager.instance.GetItemSprite(key),
      ingredientsIdCountDict[key], enoughItemFlag);
}
```

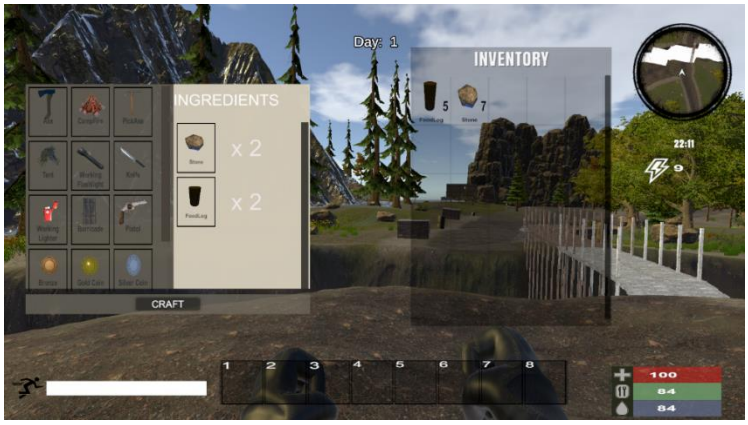
Az **`InventorySystem`** főosztályban egy fontos metódusunk lesz:

*`CraftAnItem(RecipeSO recipe)`*: A paraméterben kapott recept típuson egy *for* ciklusban lépkedünk és az összes hozzávalót kitöröljük az inventoryból.

```
foreach (var recipeIngredient in recipe.ingredientsRequired)
{
```

```
        inventoryData.TakeFromItem(recipeIngredient.ingredient.ID,
        recipeIngredient.count);
}
```

Az `inventoryData` osztály `AddToStorage(recipe)` metódusával pedig hozzáadjuk a barkácsolt *itemet* az eszköztárunkhoz.




### 3.4.2 Ételek sütése

A játék során barkácsolhatunk egy tábortüzet fa[13] és kövek segítségével. A tábortüzet a *Placement* rendszer segítségével elhelyezhetjük a pályán. Az „*E*” billentyű lenyomásával pedig elérhetővé válnak a megsüthető ételek. Ilyenek például az állatok után kapott húsok megsütve több éhségpontot töltenek vissza.

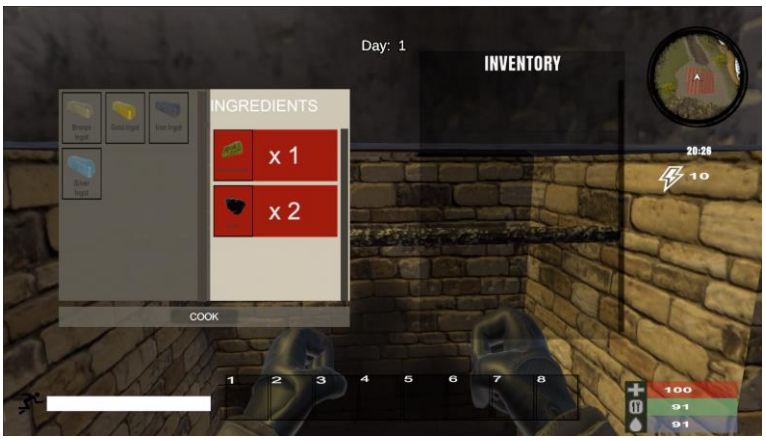

*3.6. ábra - Ételek sütése*

### 3.4.3 Ércek kiégetése

A játékban a csákányokkal bányászhatunk különböző érceket és szenet is, nem csak követ. Ezeket a faházakban lévő kemence[14] segítségével tudjuk kiégetni. Az ércekből így rúd készül (vas, arany, ezüst, bronz rúd). A vas rudakból vas csákányt lehet barkácsolni,

[13] https://assetstore.unity.com/?q=%20PBR%20Log&orderBy=1 – Elérési idő: 2021.11.26
[14] https://assetstore.unity.com/packages/3d/props/medieval-props-41540 - Elérési idő: 2021.11.26

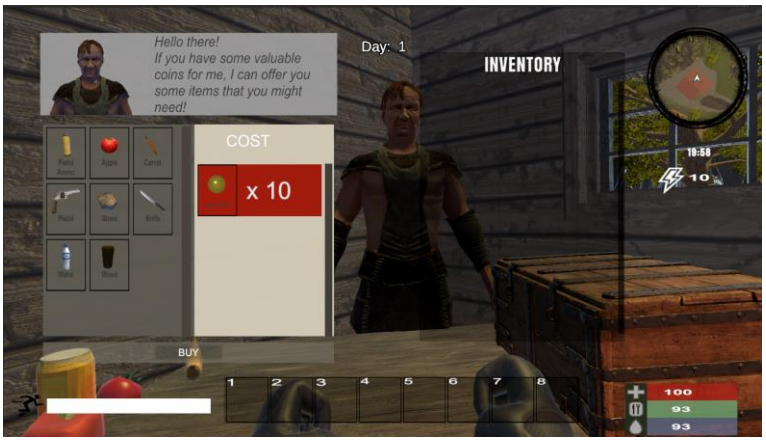
amivel ki tudjuk ütni az arany és az ezüst érceket is (kőcsákánnyal nem). Az ércek kisütéséhez szénre van szükségünk.




*3.7. ábra - Ércek kiégetése*

### 3.4.4 Tárgyak vásárlása

A bronz, ezüst és arany ércekből érméket is barkácsolhatunk magunknak, amivel az egyik faházban lévő kereskedő[15] őslakosnak lehet fizetni különböző hasznos túlélő tárgyakért.




*3.8. ábra - Tárgyak vásárlása*

## 3.5 Tárgyak elhelyezése a pályán

A játékban az egyes barkácsolható elemeket le is rakhatjuk a pályán, majd akár használhatjuk is azt. Ilyen tárgyak például a tábortűz, sátor[16] vagy egy fa barikád[17].

A tábortűzről már volt szó, a sátorban pedig aludni lehet. Ekkor a karakter energiaszintje (jobb felső sarok a térkép alatt) visszatölt egy bizonyos szintet és másnap reggel kel fel. Aludni csak egy bizonyos időszakban lehet (*18:00* és *04:00* között).

[15] https://assetstore.unity.com/packages/3d/characters/humanoids/humans/andan-16389 - Elérési idő: 2021.11.26
[16] https://assetstore.unity.com/packages/3d/environments/hovel-77502 - Elérési idő: 2021.11.26
[17] https://assetstore.unity.com/packages/3d/props/exterior/wooden-barricades-111243 - Elérési idő: 2021.11.26.

A rendszerért a ***PlacementSystem*** mappában lévő osztályok felelnek. Minden használható és egyben lerakható eszközre külön létrehoztam egy osztályt ami azért lesz felelős, hogy érzékelje, ha körülötte vagyunk és interakció billentyűt ( „*E*”) nyomunk. Ezekben az osztályokban szerepel egy `Use()` metódus, ahol megívom a `FindObjectOfType<AdottTipus>().ShowAdottFunkcioUI()` metódust, ami az adott típusú osztály egy `ShowUI()` metódusát fogja meghívni.

A **`PlacementHelper`** osztály felel az egész rendszerért. Ebben is sokat segített a *SunnyValleyStudios Udemy* kurzusa, nélküle ezt nem tudtam volna elkészíteni.

*`Structure PrepareForPlacement()`*: Ez a metódus felkészíti az adott tárgyat az elhelyezési állapotra. A *RigidBody* komponensét ideiglenesen töröli és a *collider triggerjét*[18] is kikapcsolja.

A `FixedUpdate()`[19] metódusban fogunk az adott lerakható tárgynak *RayCastokat*[20] adni a *BoxCollider* alapján.

```
Vector3 topLeftCorner = bottomCenter + new Vector3(-
boxCollider.size.x / 2f, 0, boxCollider.size.z / 2f);
```

Ebből természetesen kell még egy középső pont, valamint a *topRight*, *bottomLeft* és *bottomRightCorner* is, hogy megkapjunk egy négyzetet a modell alján. Ezek után egy bizonyos nagyságú (*raycastMaxDistance*) *Raycastot* indítunk ezekből a sarkokból, hogy érzékeljük, van-e valami az alsó négyzet fölött és elhelyezhetjük-e a tárgyat[21].

```
bool result1 =
Physics.Raycast(transform.TransformPoint(topLeftCorner) +
Vector3.up, Vector3.down, out hit1, raycastMaxDistance,
layerMask);
```

Ezek után egy tömbben eltároljuk a *RayCastok* értékeit, majd megnézzük a legkisebbet és a legnagyobbat. Ha a legkisebb és a legnagyobb érték különbsége egy adott, előre meghatározott értéknél nagyobb, akkor ne tudjuk lehelyezni a tárgyat (mert mondjuk ferde, vagy egy másik tárgyban van éppen az egyik fele). Akkor se tudjuk lehelyezni, ha a legkisebb *RayCast* kisebb, mint 0, mert ez azt jelentené, hogy valahogy a föld alá került éppen a tárgy, amit lehelyeznénk. Ha egyik sem teljesül, akkor beállítjuk a *RigidBody* pozícióját és egy logikai segédváltozót (ami alapján tudjuk, hogy elhelyezhetjük-e a tárgyat) igazra állítunk.

---

[18] Nem fog így elakadni objektumokban az adott elhelyezendő *item*, miközben sétálunk vele és próbáljuk lerakni valahova.
[19] Olyan, mint az `Update()`, csak ez a metódus minden fixált képkockában fut le
[20] Egy adott pontból, adott hosszúságú sugarat küld egy adott irányba a *colliderek* felé
[21] Ezt mindegyik oldalra létre kell hozni hasonlóan mint a példa kódban.

A lerakást a **PlacementState** osztályban fogjuk megvalósítani. A `HandlePrimaryAction()` metódust akkor hívjuk meg, ha a bal egérgombot megnyomtuk. Ha fentebb említett segédváltozó igaz, akkor meghívja a `PlaceMentHelper` osztály metódusait, majd töröli az inventoryból az *itemet*.


*3.9. ábra – Sátor elhelyezése*

## 3.6 Nyersanyagok és egyéb tárgyak szerzése

A játékban több kiüthető *objektum* is van, amikből különböző nyersanyagokat, tárgyakat szerezhetünk. Minden ilyen objektumtípusra külön osztály lett létrehozva, azonban a funkciójuk nagyrészt megegyeznek. Ezért az első pontban (*fa*) a teljes objektumkiütés és tárgyszerzés rendszer logikája részletezve van, azonban a többiben már csak az egyedi metódusok, funkciók vannak megemlítve.

### 3.6.1 Fa

A fa kidőlésért felelős a **TreeProperty** osztály, melyet az Editorban a fa *prefab-ekre* fogunk rakni, mint komponens. A **DestroyTree()** metódus lesz felelős azért, hogy a fa objektumot kitörölje és két megadott érték között random mennyiségű fát adjon. Az *Instantiate*() beépített metódus a paraméterben megadott objektumot fogja megjeleníteni a megadott pozíción és ezt vehetjük majd fel az eszköztárunkba (a példában ez fa lesz).

```
Instantiate(log, thisTree.transform.position,
thisTree.transform.rotation);
```

Az `Update()` metódusban pedig ha egy fa életerője 0, akkor a *rigidbody* gravitációját bekapcsoljuk, a *kinematic* értékét pedig kikapcsoljuk, valamint meghívjuk rá az `AddForce()` metódust, hogy a fa objektum kidőljön a megadott irányba.

A **ChopTree** osztály a fa kiütésért felelős. Arra a *prefab*-re kell ráraknunk a *script*-et, amelyikkel ki akarjuk ütni a fát, ami most a balta lesz:

*bool GetInteractOut(float distance, out Collider hitcollider)*: Megvizsgálja, hogy a paraméterben kapott *distance* távolságon van-e valamilyen *collider*, azaz egy olyan objektum aminek van ilyen komponense és igazzal tér vissza, ha van.

`GetInteract()`: Ebben a metódusban fogjuk megvizsgálni, hogy az adott objektum *Tag*-e[22] megegyezik-e az általunk kívánt *Tag*-gekkel. A fa objektumok Tag-jét *„Tree”*-re állítottam. A metódusban egy random sebzési értéket beállítunk és azt levontjuk a már fent említett `TreeProperty` osztály *durability* értékéből, ugyanis ez lesz az adott fa „életerője”. A fent említett `DestroyTree()` metódusban történtek csak akkor futnak le, ha ez a `durability` 0 alá csökken.

### 3.6.2 Kövek, ércek

A kövek és ércek kiütése annyiban különbözik, hogy a **`StoneProperty`** osztályt fogjuk rárakni a kövekre, ércekre és a szénre. Az `Instantiate()` metódusa pedig nem fákat fog lehelyezni, hanem követ vagy az adott ércet, szenet.

A kövek és az egyes ércek kiütésére a **`MineStone`** szkript szolgál, az arany és az ezüst pedig kap egy külön osztályt. Lényegében ugyan az a kód, csupán mikor az *Objektum Tagek*-ket ellenőrizzük, az első osztályban *„Stone”* és *„Ore” Tag-geket* fogunk vizsgálni, a másikban pedig ugyan ezeket, csak még hozzárakjuk az arany és ezüst objektumok tagjeit, a *„Gold_Silver_Ore*” *Tageket* és azokra is vizsgálunk a `GetInteractOut()` metódusban. A kőcsákányra kerül a `MineStone` osztály, a vascsákányra pedig az arany és ezüst kiütésére is szolgáló osztály.

### 3.6.3 Hordók és dobozok

A hordókra a **`BarellProperty`**, a fa dobozokra pedig a **`BoxProperty`** osztályt használjuk. Itt annyiban fog különbözni az eszközök *spawnolása*[23] a két objektum törlése előtt, hogy a random mennyiségű *itemeken* kívül itt százalékos eséllyel kaphatunk egyes tárgyakat. Ezt is ugyanúgy az `Instantiate()` metódussal valósítom meg, csupán itt az első paraméterben kapott *spawnolandó item* fog random lenni (például kötszer, gyógyszer, víz, alma, kötél, stb.[24]).

[22] A Tag beazonosít egy adott objektumot, amire később a kódban hivatkozni lehet. Az Editorban tudunk *Tag*-gel ellátni egy objektumot.
[23] Pályán lévő megjelenítése
[24] https://assetstore.unity.com/packages/3d/props/tools/survival-kit-lite-92549 - Elérési idő: 2021.11.26

A `Choptree` osztályban a `GetInteractOut()` metódusban külön vizsgálunk egy *„Box” Taget*. Szintén erre az osztályra, valamint a csákányok osztályaira pedig egy *„Barell” Taget.* Tehát fákat csak baltával, köveket és érceket az adott csákánnyal, hordókat csákánnyal és baltával, fadobozokat csak baltával tudunk kiütni.

A favágásra, dobozkiütésre, valamint a kövek és ércek kiütésére egy *ParticleSystem*[25] is lejátszódik minden egyes ütéskor. Ezt a ***ParticleSystem***[26] `Play()` beépített metódusával fogom lejátszani. Az animációról lesz szó, de itt elég annyi, hogy az ütés animáció egy adott pontjában fogom meghívni ezt a metódust a *Particle*-re.

## 3.7 Idő rendszer

A játékban a jobb felső sarokban lévő **kistérkép** alatt folyamatosan láthatjuk az időt. Ez azért is hasznos, mert például az alvási rendszer is erre épül, valamint láthatjuk, hogy körülbelül mikor fog esteledni. A képernyő tetején pedig folyamatosan láthatjuk, hogy hány napot éltünk már túl. A nappal és éjszaka valós idejű (a játékban lévő idő szerint) változása is tehát ebbe a rendszerbe sorolható, amit a nap és hold mozgatásával lehet megvalósítani. Ebben *Aaron Hibberd Youtube* videója[27] segített

### 3.7.1 Aktuális idő

Az aktuális idő folyamatos megjelenítéséért a *Clock* és a *TimeUI* osztály felelős.

```
void Update()
{
      timer -= Time.deltaTime;
      if(timer <= 0)
      {
            Minute ++;
            OnMinuteChanged?.Invoke();
            if(Minute >= 60)
            {
                Hour++;
                Minute = 0;
                OnHourChanged?.Invoke();
                if(Hour >= 24)
                {
                Hour = 0;
                }
            }
            timer = minuteToRealTime;
       }
       TimeCheck();
}
```

[25] https://assetstore.unity.com/packages/vfx/particles/mining-chopping-vfx-199178 - Elérési idő: 2021.11.26
[26] A Unity nagy teljesítményű és sokoldalú részecskerendszer implementációja.
[27] https://www.youtube.com/watch?v=DmhSWEJjphQ&t=655s – Elérési idő: 2021.11.26

A `timer` egy előre beállított érték. A percet nem szeretnénk valóban 60 másodpercre állítani, hisz a többi játékban sem így működik, és egy nap túlélése valóban egy teljes, valós nap lenne. Helyette egy perc a játékban körülbelül 1 másodpercnek fog megfelelni. Ha letelik a *timer* akkor a perc értéke növekedni fog egyel, ha pedig 60-nál nagyobb lenne, akkor az óra értéke fog növekedni és a perc nullázódik. Ha az óra nagyobb lenne, mint 24, akkor az óra nullázódik. A `TimeCheck()` metódust fogjuk folyamatosan meghívni és ellenőrizni az újabb túlélt napokat.

Az `OnOurChanged` és `OnMinuteChanged Action`-nök, az értékük növekedésekor „hívódnak” meg. Ezeket majd a `TimeUI` osztályban kell felhasználni. Ez az osztály felel az idő képernyőre történő folyamatos kiíratásáért. *TextMeshPro*-t[28] használtam a kiíratáshoz. Az *UpdateTime()* metódus fogja kiíratni az adott időt a `Clock` osztályban lévő perc, óra és nap értékek alapján.

```
public void UpdateTime()
{
      timeText.text= $"{clock.Hour:00}:{clock.Minute:00}";
      DayText.text = $"{clock.Day:0}";
}
```

Ezt a metódust folyamatosan hívogatni kell, de csak akkor mikor valamilyen változás történt. Éppen erre kellettek a már fenn említett Action típusok.

```
private void OnEnable()
{
      clock.OnMinuteChanged += UpdateTime;
      clock.OnHourChanged += UpdateTime;
}
```


*3.10. ábra - Jobb sarokban az idő, felül pedig túlélt napok száma látható*

[28] Egy fejletteb Text komponens Unity-ben.

### 3.7.2 Nappal és éjszaka

Egy egyszerű rendszer, melyet a ***sun*** osztály valósít meg. A nap objektum *Directional Light*[29] fénytípus lesz. Erre az objektumra kell rátenni komponensként az osztályt. Az itteni kódok 80%-ka, a nappal és éjszaka mentési rendszeréért felel, azaz elraktározza az aktuális pozíciót és a forgási irányát. Ugyancsak ezt megvalósítjuk a holddal is.

A nap és a hold forgásáért csupát két sor kód felel az `Update()` metódusban. A *transform* beépített `RotateAround`() metódusa az adott objektumot fogja forgatni. Az első paraméterben megadjuk a középpontot a forgatásnak, a másodikban az irányt, a harmadikban pedig a sebességet. Fontos, hogy a nap fénye mindig felénk nézzen a forgás közben, ezért kell a `LookAt()` metódus, aminek a középpontot adjuk meg paraméterül, hisz a forgatási középpont a pálya lesz[30]. A holdat a nap objektum egy gyerekeként fogjuk beállítani és az ellenkező irányba fogjuk elhelyezni. A nappal így együtt fog mozogni a hold és pont az ellenkező irányban lesz és éjszaka látni lehet majd.

```
transform.RotateAround(Vector3.zero,Vector3.right, rotationSpeed
* Time.deltaTime);
transform.LookAt(Vector3.zero);
```


*3.12. ábra - A nap lemenetele*

## 3.8 Mentési rendszer

A játék mentési rendszeréhez ***JSON*** szöveg alapú szabvány lett használva. A játék során minden mentendő elemre ugyan azt a logikát alkalmazom, csupán a mentett adatok lesznek különbözőek, ezért egy példán keresztül mutatom be a rendszert. A játékban az alábbi elemekre alkalmaztam mentést: *inventoryban* lévő tárgyak, játékos pozíciója, nap és hold pozíciója, a pályára elhelyezett tárgyak pozíciója, kiüthető objektumok állapota

[29] Végtelenül messze lévő fény, amely csak egy irányba bocsát ki fényt.
[30] Természetesen ez való életben nem így történik, de egy játékban nehezebb lenne megvalósítani, hogy a pálya (föld) forogjon a nap körül. Az optikai realitást viszont ugyan úgy elérjük vele.

(ha pl. egy fát kiütök és mentek, akkor maradjon kiütve), pályán előre elhelyezett felvehető tárgyak, játék idő, valamint a játékos életerő, éhség, szomjúság és energia.

### 3.8.1 Mentés

A mentéseimet a ***SaveSystem*** osztályban valósítottam meg. A mentés és betöltés rendszert a játékos státuszértékeire fogom bemutatni (éhség, szomjúság, energia és életerő). Először is megadjuk a mentési útvonalat. Fontos, hogy ez olyan útvonal legyen, ami minden számítógépen valós, létező útvonal, ezért használom az *Application.persistentDataPath* megvalósítást. Az első mentéskor a felhasználó számítógépén a következő helyre fogja menteni az adatokat a játék: ****:\Users***\Appdata\LocalLow_vagy_Local\PotyeszMate\ZenitSurvival*

A mentés során először a menteni kívánt adatokat kell *JSON* kompatibilissé alakítani. Minden mentendő elemnek létrehoztam egy *struct-tot,* ahol a menteni kívánt adatokat adom meg. Ezeknek a változóknak fogom értékül adni a játékban lévő értéket.

```
var statData = new PlayeStatData
{
      stamina = playerStat.stamina,
      health = healthScript.health,
      hunger = playerStat.hunger,
      thirst = playerStat.thirst,
      energy = playerStat.energy
 };
```

Ezek után a kapott értékeket a `JsonUtility.ToJson()` metódus segítségével *JSON* formátumúra alakítom és a `WriteAllText()` metódus segítségével kiírom a fájlba a tartalmát a kapott *JSON stringnek.*

```
var jsonString = JsonUtility.ToJson(statData);
System.IO.File.WriteAllText(playerStatsFilePath, jsonString);
```

### 3.8.2 Betöltés

A betöltés során először le kell ellenőriznünk, hogy a létezik-e az a fájl, amit olvasni próbálunk. Ezt a `System.IO.File.Exist(filepath)` metódussal tehetjük meg. Ezt követően a `System.IO.File.ReadAllText(filepath)` metódussal beolvassuk a kívánt fájlt. A `JsonUtility.FromJson<Struct>(jsonstring)` metódus pedig kiolvassa nekünk a JSON *stringből* a megadott *Struct* alapján az értékeket.

```
var jsonSavedPlayerStat =
System.IO.File.ReadAllText(playerStatsFilePath);
var data =
JsonUtility.FromJson<PlayeStatData>(jsonSavedPlayerStat);
```

Ezt követően pedig a beolvasott változókat értékül adjuk az adott funkcióhoz tartozó, játékban lévő változóknak (esetünkben a játékos státusz adatainak).

```
healthScript.health = data.health;
playerStat.stamina = data.stamina;
playerStat.hunger = data.hunger;
playerStat.thirst = data.thirst;
playerStat.energy = data.energy;
```

A mentéseket és a betöltéseket is természetesen külön metódusokban valósítottam meg. A **`GameManager`** osztály felelős a mentések tényleges megvalósításáért amit majd a játékmenüben lévő *„Save”* gombbal tudunk használni, valamint a betöltésért, amit a főmenüben lévő *„Continue”* gomb megnyomásával fogunk aktiválni.

A `SaveGame()` metódusban fogjuk meghívni az összes mentési metódust, `Start()` metódusba pedig az összes betöltési metódust. Ez azonban egy *`if()`* feltételbe kerül, ahol azt vizsgáljuk, hogy a játékosunk a *„Continue”* gombra kattintott-e vagy sem a játék betöltése előtt. Így már meg is van a zárt logikai mentési rendszerünk. Játékmenüben elmentjük a játékot, amivel a `SaveGame()` metódust hívjuk meg és elmentjük az összes menteni kívánt elemünket a játékban. A főmenüben pedig ha a *Coninue* gombra kattintunk, akkor ezt érzékeljük a kódban és betöltjük a mentésünket és folytatjuk a játékunkat.

### 3.9 Menürendszer

A játékban három fajta menü létezik. Az úgynevezett *pause menu* (játékbeli menü), a *main menu* (főmenü) és a *Game Over* (játék vége) *menu.*

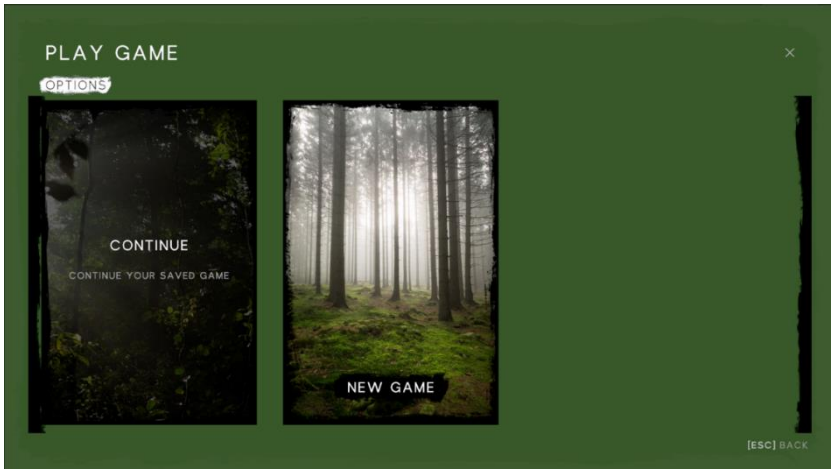
#### 3.9.1 Főmenü

Régebben fejlesztettem már pár 2 dimenziós játékot és még akkor megvásároltam egy akció keretein belül egy főmenüt a *Unity Asset Storeban*.[31] Általában megéri egy komplexebb játék főmenüjét megvásárolni, hisz ez inkább grafikai érzéket és tapasztalatot igénylő munka, valamint a játékfejlesztők is azt szokták mondani, hogy magunknak egy ilyen terméket sokkal több munkaidő lefejleszteni, mint amennyi munkaidővel egyenértékű pénzbe kerül nekünk megvásárolni, ráadásul ezt szabadon módosíthatjuk. Azonban ezt a menürendszert nekem be is kellett valahogy implementálni a játékba és nem árt az sem, hogy hasonló stílusú legyen a kinézete, mint amiről a játék

[31] [30] Unity Asset Store – Michsky – Dark - Complete Horror UI – Elérési ideje: 2021. 11.23 - https://assetstore.unity.com/packages/2d/gui/dark-complete-horror-ui-200569

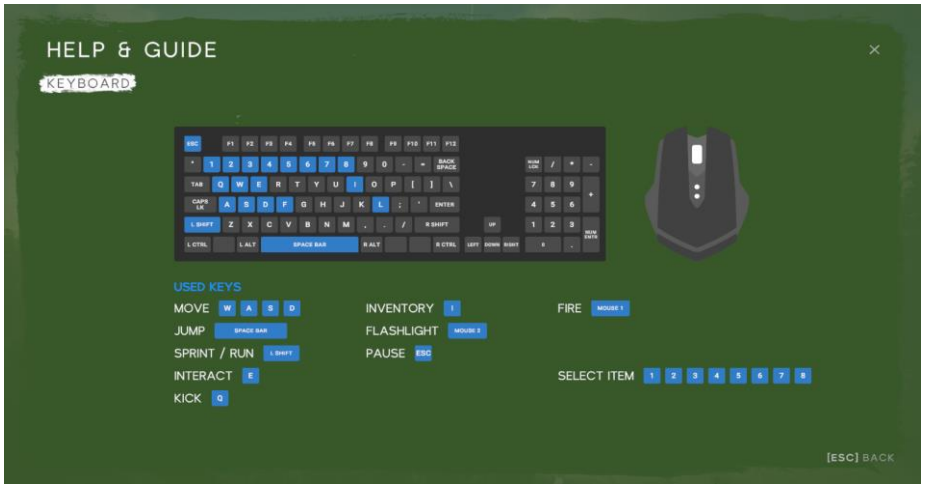
szól. A játék indításakor szépen beúszik a játék neve (*Zenit Survival*) és utána a saját nevem.

***Play Game***: Ezen az opción belül lesz lehetőségünk választani, hogy folytatni szeretnénk a mentett játékunkat, vagy új játékot szeretnénk kezdeni. A *Continue* gomb megnyomásakor a `GameManager` osztály `Start()` metódusában töltjük be a mentéseket és a `SceneManager.LoadScene(MainScene)` metódussal betöltjük a játék színterét. A *New Game* gombbal is a játék színterét töltjük be, viszont a Game Manager `Start()` metódusa figyelmen kívül hagyja a betöltéseket.




*3.12. ábra - Játékindítási lehetőségek*

***Controlls***: Egy képet jelenít meg a billentyűzetről, és kiemeli azokat a billentyűket amikre szükségünk van az irányításhoz.




*3.13. ábra – Irányítás*

***Settings***: Pontosan ezért a beállításért is éri meg egy menürendszert megvásárolni. Itt ugyanis tucatnyi beállítás van, amiket ugyan nem nehéz megvalósítani, de rengeteg időbe telik még az egyszerűbbeket is beállítani. Itt két opció van: *Audio* és *Visuals*.

Az ***Audio***-ban beállíthatjuk a zene, az effektek és a játék hangerőt.


*3.14. ábra - Hangbeállítások*

A ***Visuals*** elég sok beállítási lehetőséget tartalmaz.

***Bal oldalt*** a következő beállítások szerepelnek, amik elég hasznosak: Ablak mód (be-ki), Méretarány (az összes, kb. 20 féle méretarány működik), Fényerő, Gamma, V-Sync, Field of View és Draw Distance (látószög és látótávolság).

***Jobb oldalt*** beállítható a játék grafikai minősége, Árnyékok és reflexiók.


*3.15. ábra - Grafikai, megjelenítési beállítások*

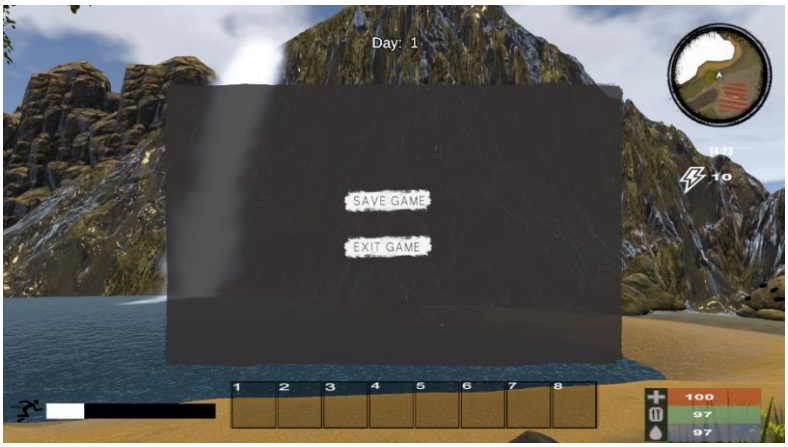
**Exit**: Kilépünk a játékból.

### 3.9.2 Játékmenü

A *pause menu* lényegében a játékan lévő menü, melyet az *Escape* billentyű lenyomásával tudunk előhívni. A nevében is benne van, hogy ilyenkor megáll a játék, tehát tudjuk szüneteltetni. Ebben a menüben a „*Save game*” és az „*Exit game*” gomb található, melyekről már a mentési rendszerben szó volt.

A `HandleEscapeInput()` metódus fogja kezelni az *Escape* billentyű lenyomását. Nemcsak a *pause menu*-t fogja megjeleníteni és elrejteni, de például az

*inventoryt* is elrejti (ha éppen be aktív). Ebben a metódusban hívjuk meg a GameManager osztályban lévő `ToggleGameMenu()` metódust, ami a játékmenü és az egér ki-be kapcsolásáért felel.




*3.16. ábra - Játékbeli menü*

### 3.9.3 Játék vége menü

Amikor a karakterünk életerője 0 alá esik, akkor a játéknak vége. Ilyenkor egy „*Game Over*” menüt dob be a játék, ahol két lehetőségünk van. Vagy kilépünk, vagy új játékot kezdünk. A gombok megnyomásakor a `SceneManager.LoadScene(Scene)` metódus segítségével töltődik be az adott *Scene* (új játék vagy a main menu).


*3.17. ábra - Játék vége menü*

## 3.10 Mesterséges intelligencia

A játékban lévő mesterséges intelligencia egy részét az *„Awesome Tuts”*[32] nevű Youtube csatorna videója segített létrehozni. Az agresszív ellenfelekért az `EnemyController` osztály felel, a barátságosakért pedig a `FriendlyEnemyCondroller`. Az utóbbi csak annyiban különbözik, hogy nincs támadási metódusa és nem ránk támad, hanem elfut

[32] https://www.youtube.com/watch?v=CTWvAwSw98Q - Elérési ideje: 2021. 11.26

előlünk. Az ellenfelek mozgatásához az adott objektumra egy *Nav Mesh Agent*[33] komponenst helyeztem.

### 3.10.1 Ellenfél tétlen viselkedés

A pályán lévő összes ellenfélnek létezik egy „*Patrol*” állapota, melyet magyarul tétlen viselkedésnek nevezhetünk, melyért a **`Patrol()`** metódus felel. A metódus először beállítja a `navagent.speed` értéket a sétálási értékre. A következő lépés, hogy egy segédváltozót folyamatosan növeljünk az idő elteltével és ha ez egyenlő lesz egy előre meghatározott értékkel, akkor egy új pozíciót keressen. Amíg ezt az értéket el nem éri, addig az ellenfél egy helyben fog állni, és egy „*Patrol*” animációt is beállítunk rá (például egy őz a *Patrol* ideje alatt felváltva körbe néz és lehajol enni). Ha pedig új pozícióra megy, akkor egy „*Walk”* animációt állítunk be (sétálás animáció). Az új pozíció meghatározásáért a `SetNewRandomDestination()` metódus felel, ami egy bizonyos sugarú körben egy random pontot fog kiválasztani. A metódus utolsó feladata, hogy a játékos és az ellenfél közötti távolságot figyelje. Ha ez a távolság kisebb, mint az előre meghatározott maximum érték, akkor az *enum* felsorolástípusú `enemy_State` (mely az ellenfél állapotait tartalmazza) *CHASE*-re vált át. Az `Update()` metódusban pedig lekezeljük, hogy CHASE típuskor a *Chase()* metódust hívjuk meg (üldözés).

```
if (Vector3.Distance(transform.position, target.position) <=
chase_Distance)
{
enemy_Anim.Walk(false);
enemy_State = EnemyState.CHASE;
enemy_Audio.Play_ScreamSound();
}
```

*3.18. ábra - Egy őz Idle (tétlen) állapotban*

[33] Ez a komponens lehetővé teszi az adott objektum számára, hogy Nav Mesh segítségével navigálni tudjon a jelenetben a *property-ei* és metódusai segítségével.

### 3.10.2 Ellenfél támadás

A **`Chase()`** metódus felel az ellenfél üldözéséért. A `navAgent.speed` értéket a `run_Speed`-re állítja be, azaz futási sebességre.

A `navAgent.SetDestination(target.position)` beállítja az új pozíciót ami felé az ellenfélnek haladnia kell, ez pedig a karakter aktuális pozíciója lesz. Amíg el nem éri azt a pozíciót, addig egy *Run* animáció hat rá, hisz futva fogja a karakterünket üldözni. Az `if()` feltételben itt azt vizsgáljuk, hogy az ellenfél távolsága elérte-e már a támadási távolságot. Ha igen, akkor minden más animációt megállítunk és meghívjuk az `Attack()` metódust. Azonban még a `Chase()` metódusban azt is meg kell vizsgálni, hogy nagyobb-e a távolság, mint az üldözési távolság (*`chaseDistance`*). Ugyanis ilyenkor az ellenfél már nem kerget minket tovább, a futási animáció megszűnik és meghívjuk a `Patrol()` metódus.

Az **`Attack()`** metódus valósítja meg azt, hogy az agresszív ellenfél meg is tudjon minket támadni. A metódusban a `navAgent.isStopped` *property* értékét igazra állítjuk, hogy megálljon előttünk az ellenfél. A támadási animációt (*Attack*) csak bizonyos időközönként játsszuk le. Ezt a módszert már a favágásnál is alkalmaztam. Kell egy segédváltozó, ami másodpercenként növekszik és csak akkor hívjuk meg a támadási animációért felelős metódust, amikor ez a segédváltozó eléri az általunk kiválasztott értéket. Fontos, hogy itt a `Chase()` metódust is meg kell hívnunk (*enum* segítségével), mivel el is fogunk menekülni, vagyis pozíciót váltani, így addig kell, hogy kövessen minket az ellenfél, amíg a távolsága nem lesz nagyobb az `AttackDistance` és a `ChaseAfterAttackDistance` összegénél (azt adja meg, hogy milyen távolságig kövessen minket az ellenfél, miután mi menekülünk a támadás után).

```
if(Vector3.Distance(transform.position, target.position) >
attack_Distance + chase_After_Attack_Distance)
{
      enemy_State = EnemyState.CHASE;
}
```

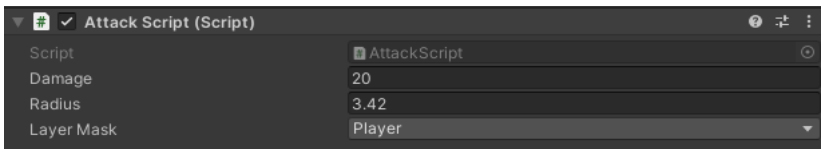
Ahhoz, hogy az ellenfél tudjon támadni, a `Turn_On_AttackPoint` és a `Turn_Of_AttackPoint` metódusok lesznek (részben) felelősek. Ezek a metódusok csupán ki és bekapcsolnak egy objektumot, amihez az `AttackScript` osztályt fogjuk komponensként hozzáadni. A támadási animációban fogjuk meghívni egy rövid időre a bekapcsolásért felelős metódust, majd pedig a kikapcsolásért felelős metódust. Mivel az `AttackScriptben` történtek az `Update()` metódusban valósulnak meg, így a rövid

időre történő bekapcsolás idejére is fog működni a támadás, azaz addig amíg a támadási animáció tart.

Mikor a bekapcsolt objektum (nevezzük hitPoint-nak) hozzáér egy másik *objektumhoz*, akkor a `HealthScript`-ben lévő `ApplyDamage()` metódus fog meghívódni. Ez csupán a paraméterben megadott *damage* értékével egyenlő sebzést fog levonni a hozzáért objektum életerejéből (*LayerMask*). Tehát ha az ellenfél *hitPoint*-ja ez, akkor a karakter életerője csökken, ha pedig az egyik közelharci fegyveren lévő *hitpoint*, akkor az ellenfélé.

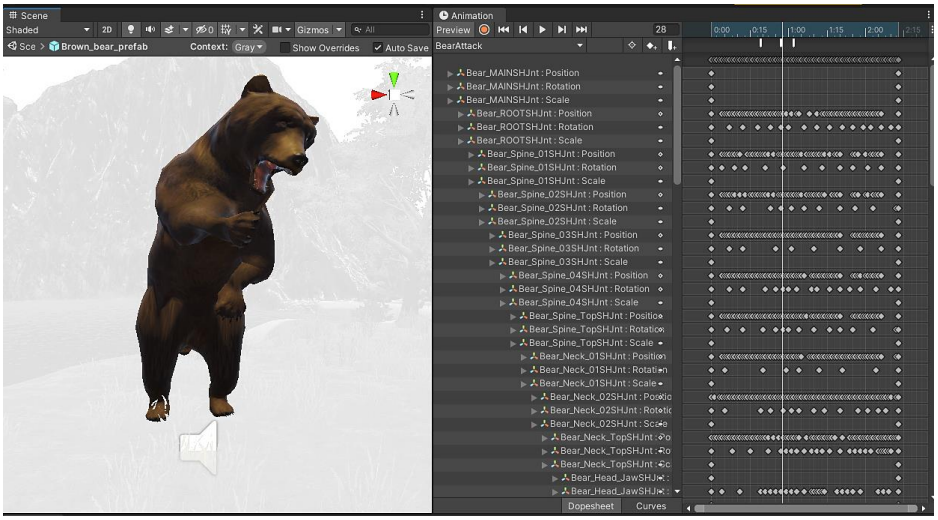
```
Collider[] hits = Physics.OverlapSphere(transform.position,
radius, layerMask);
if (hits.Lentgth > 0
{
      hits[0].gameObject.GetComponent<HealthScript>().Apply
      Damage(damage);
}
```

Mivel ebben az esetben egy ellenfélről beszélünk, így a *LayerMaskot Player-re* állítjuk (az ellenfélnek a *Layerét* „*Enemy*”-re, a karakter *Layerét* „*Player*”-re kell beállítani az Editorban). Beállítjuk, hogy mekkora sebzése legyen és azt is, hogy milyen közelről sebezhet az adott ellenfél (*Radius*).




*3.19. ábra - Ellenfelen lévő AttackScript komponens: 20 sebzéssel és Player érzékelővel*

Az *Animation tabon* pedig a jobb felső sarokban lévő „*Add Event*” opcióra kattintva tudjuk a be és kikapcsoló metódusokat hozzáadni az animáció adott pontjához.




*3.20. ábra - Agresszív ellenfél támadási animációjában kapcsoljuk be, majd ki a sebzésért felelős objektumot. A példában a medve kezén lesz rajta az érzékelő objektum.*

### 3.10.3 Ellenfél menekülés

A `FriendlyEnemyController` osztály mindenben megegyezik az `EnemyController`-rel, csupán annyi a különbség, hogy nincs `Attack()` metódus és állapot, valamint a `Chase()` metódusban nem a karakter pozíciója felé fut az ellenfél, hanem az ellenkező irányba. A távolság amennyit elfut előlünk így pont az ellenfél és a karakter kötötti távolság lesz.

```
Vector3 directToPlayer = transform.position -
GameObject.FindWithTag(Tags.PLAYER_TAG).transform.position;
Vector3 new_position = transform.position + (directToPlayer);
navAgent.SetDestination(new_position);
```

A `SetDestination()` metódus pedig erre az új pozícióra fogja irányítani az ellenfelünket.

Az ***HealthScript*** osztályban ha az életerő 0 alá csökken, akkor a `PlayerDied()` metódus lesz meghívva. Ez megvizsgálja, hogy milyen típusú objektumról van szó (összes lehetséges ellenfél, vagy a karakter) és hogyha ellenfélről, akkor az *EnemyController* (vagy *FriendlyEnemyController*) komponenst kikapcsolja, valamint a `navagent.isStopped` *property*-t igazra állítja. Ezek után meghívja a `Dead()` animációt majd az ehhez tartozó hangot és töröli az objektumot a pályáról. Így tudjuk legyőzni pályán lévő ellenfeleket.

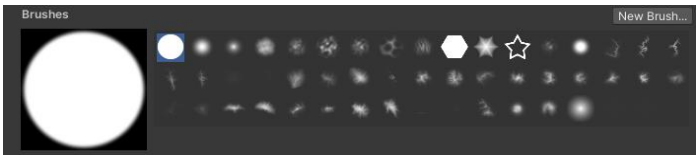
## 3.11 Pálya

A játék pályáját a *Terrain Tools* pályaszerkesztővel készítettem el. A *Unity Asset Store* nagyon sok minőségi pályaelemet biztosít ingyenesen így ezeket fel is használtam[34].

A pálya talaját a ***Paint Textures*** opcióval festettem. Ez egy ecset eszköz ahol több, úgynevezett „*Layer-t*” tudunk létrehozni. Egy ilyen *Layer-nek* különböző tulajdonságait állíthatjuk be. Többek között, hogy egy adott területen milyen sűrűn legyen az adott minta[35], valamint beállíthatjuk az ecset nagyságát, erősségét, stb. Itt tudjuk a talajt festeni adott mintázatúra (pl. fű, homok, sár, kő, föld, stb.)

A ***Stamp Terrain*** egy olyan opció, ahol adott alakzatot lehet kiemelni a talajból. Általában hegyek, dombok kialakítására használhatjuk, így én is arra alkalmaztam.

[34] https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2018-4-32351 - Elérési idő: 2021.11.26
[35] https://assetstore.unity.com/packages/2d/textures-materials/floors/pbr-ground-materials-1-dirt-grass-85402 – Elérési idő: 2021.11.26

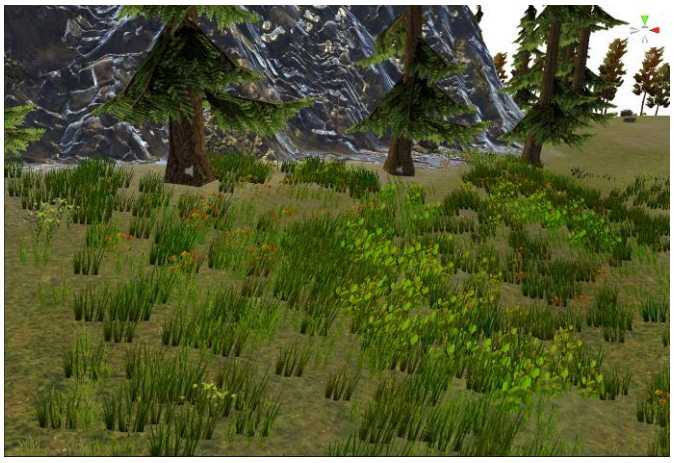



*3.22. ábra - Ecset típusok*

A ***Raise or Lower Terrain*** opció arra szolgál, hogy növelni vagy csökkenteni tudjuk a talaj magasságát az ecset méretével megegyezően. Ezután pedig a ***SmoothHeight*** tudjuk elsimítani a talajt. Így hozhatunk létre hegyeket, dombokat, tavat, folyót, stb.

*3.24. ábra- Pálya*

A ***Paint Details*** segítségével kisebb részleteket tudunk ecset segítségével megjeleníteni a pályán. Ezek főleg valamilyen virágok, füvek, esetleg bokrok[36].



*3.25. ábra - Ecsettel történő részletek festésének eredménye*

[36] https://assetstore.unity.com/packages/3d/vegetation/plants/yughues-free-bushes-13168 – Elérési idő: 2021.11.26

# 4. Animáció

Egy háromdimenziós *high-polly* játékban az animációk elkészítése nagyon sok időbe telik és minőségi munkát is csak nagy tapasztalattal lehet végezni, valamint nem is tudom alkalmazni az egyetemen tanultakat ebben a témában. Ezért a karakterhez és az ellenfelekhez vásároltam animációkat, de ezekhez szerencsére modell is tartozott (*3 asset pack*). Azonban ezeket a modelleket és az animációkat implementálni kell a játékba is. Ehhez *a Unity Editor Animator Tab*-ját[37] használtam az *Animator Controller-rek*[38] létrehozására, majd azokat C# scriptekkel bírtam működésre.

## 4.1 Karakter animáció

A karakterünk két animálható objektummal rendelkezik: láb[39] és kéz[40]. Mint már említettem, a felszerelhető eszközök külön vannak meganimálva ugyan azzal a kéz modellel. Tehát amikor nincs a kezünkben semmi, akkor az a modell lesz aktív (tárgy nélküli kéz), mikor pedig felszereljük az adott tárgyat, akkor inaktív lesz az a modell és az adott tárgyal rendelkező kéz lesz aktív. Itt minden egyes ilyen *Prefab*-re külön kód és *animator controller* van, de logikájukat tekintve teljesen megegyeznek, ezért csak egy példán van bemutatva.

### 4.1.1 Láb

Minden animációért felelős osztály két logikai részből áll. Először az `Update()` metódusban lekezeljük az bemenetet, ami alapján meghívjuk a kívánt animációt megvalósító metódust. A láb animációjáért a ***LegAnimation*** osztály felel.

Az `Update()` metódusban a `canRunAnimation` logikai változó azt a célt szolgálja, hogy a `PlayerStat` osztályban lévő `stamina` érték alapján futás közben is érvényes legyen a sétálás animáció, amikor elfárad a karakterünk. Gyaloglásnál csak a w, a, s és d billentyű lenyomásokat vizsgáljuk, futásnál ezek mellett nézzük még a bal *shiftet* is, a láb animáció esetében pedig a „Q” betű lenyomását is vizsgáljuk, hisz ekkor rúgni fog a karakterünk amivel tudunk sebezni is. Egy ilyen input kezelésre példa a futásnál:

[37] Az Animátor ablak lehetővé teszi az Animátorvezérlő eszközök létrehozását, megtekintését és módosítását.
[38] Az Animator Controller felelős az animációs klipek és a hozzájuk tartozó animációs átmenetek elrendezésért és karbantartásáért (egy karakter vagy objektum számára).
[39] https://assetstore.unity.com/packages/3d/characters/humanoids/soldier-legs-animated-29669 - Elérési ideje: 2010.11.23
[40] https://assetstore.unity.com/packages/3d/characters/animated-hands-with-weapons-pack-132915 - Elérési ideje: 2021. 11.23

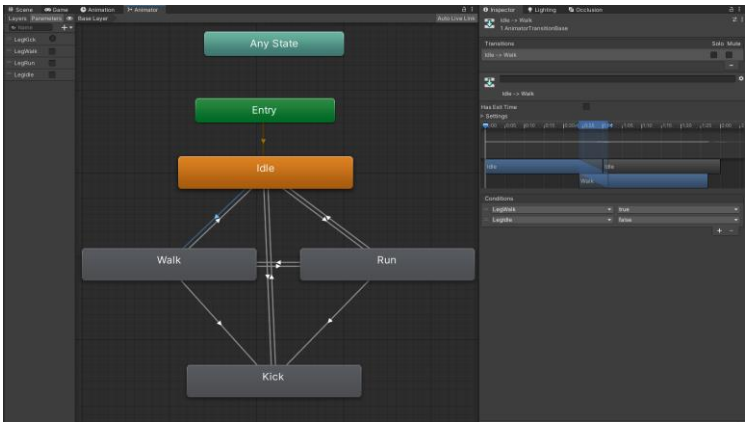
Az `Input.GetKey(KeyCode.Button)` metódus fog igaz értékkel visszatérni, ha a paraméterben megadott billentyűt lenyomtuk. Ha futás közben a `canRunAnimation` hamisra vált, azaz elfáradt a karakter, akkor a `Walk()` metódust hívjuk meg.

```
if ((Input.GetKey(KeyCode.W) || (Input.GetKey(KeyCode.S) ||
(Input.GetKey(KeyCode.A) || (Input.GetKey(KeyCode.D)))))
&& Input.GetKey(KeyCode.LeftShift))
{
      if(canRunAnimation)
            Run();
      else
            Walk();
}
```

Az animációkért felelős metódusokat az `Update()`-en kívül valósítottam meg. Itt fogjuk az ***Animator Controller-ben*** lévő paramétereket ki és bekapcsolni. Az *Animator Controllerben* először létrehozzuk az egyes animációk állapotait, majd ***Transition***-nel összekötjük őket. Ezeknek a *Transition*-nöknek[41] fogjuk a ***Condition***[42]-jeit beállítani. Például a tétlen és a sétálás animáció közötti *tradsitiont* a következő *Condition* párokkal érhetem el: *LegWalk: true* és *LegIdle: False*. Ezt minden ilyen *Transition-re* meg kell valósítani minden egyes *Animator Controllerben*, ezért már az animációk játékba való implementálása, inputokkal való működésre bírása is időigényes. A most megemlített gyaloglási példát a kódban pedig úgy valósítjuk meg, hogy a *Transioton-ben* megadott paramétereket a megfelelő logikai értékre állítom és már le is játszódik az animáció.

```
private void Walk()
{
      animator.SetBool("LegWalk",true);
      animator.SetBool("LegIdle",false);
      animator.SetBool("LegRun",false);
}
```

[41] Animáció átmenet
[42] Állapot



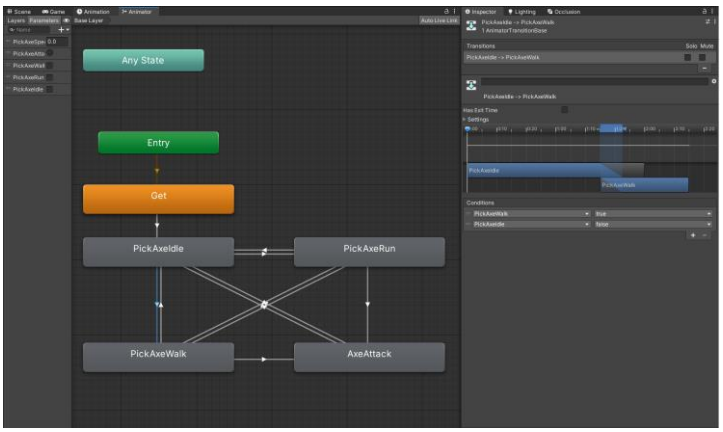

*4.1. ábra - Láb animator controller*

### 4.1.2 Kéz és eszközök

Az animáció megvalósításának logikáját már tudjuk, így most már csak az adott *animator controller* kinézetén és az esetleges egyedi animáció kezelésén lesz a hangsúly. A kézben lévő kőcsákány animációját mutatom be, de az összes többi eszköz animációja is logikailag megegyezően van megvalósítva és természetesen az *animator controllerük* is hasonló.

Az játékos kezeinek animációjának beállítása is hasonlóan történik mint a már leírt láb animációja. A különbség itt most annyi, hogy a támadási animációt most a bal egérgomb lenyomásával tudjuk meghívni, amit az `Input.GetKeyDown()` metódus fog érzékelni (itt az egérgomb felengedését kell figyelembe venni). A `timeT` segédváltozó pedig a bizonyos időközönkénti támadás animációért felel. Ez a támadás animáció fog lejátszódni tehát a csákány esetében minden bal egérgomb lenyomáskor

```
if(Input.GetKeyDown(KeyCode.Mouse0) && timeT > 1 &&
!isInventoryOn)
{
        Attack();
        timeT = 0;
}
```




*4.2. ábra - Kéz és csákány animator controller*

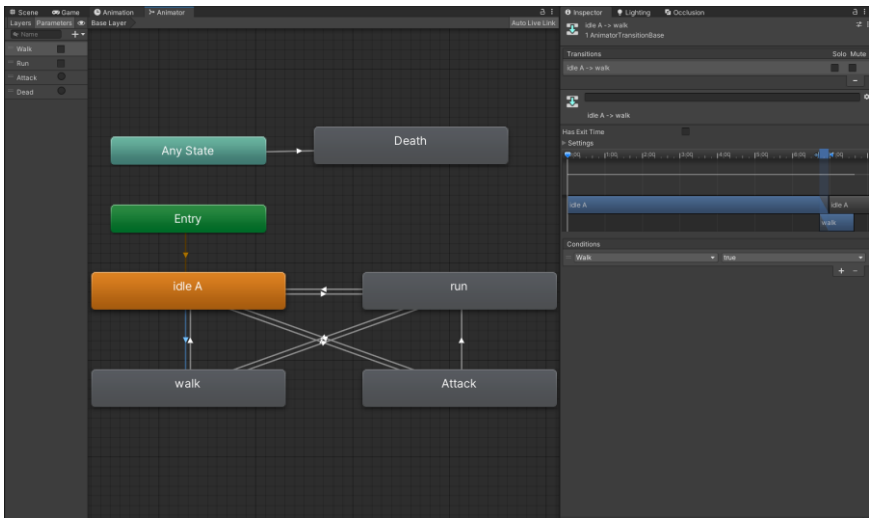
Az `Attack()` metódusban pedig egy *triggert* fogunk bekapcsolni minden egyes támadásnál és ha ez a *trigger* be van kapcsolva, akkor lejátszódik egyszer az animáció.

```
private void Attack()
{
      animator.SetTrigger("PickAxeAttack");
}
```

## 4.2 Ellenfél animáció

Az ellenfél animációjánál[43] viszont semmi újdonság nincs. A mesterséges intelligenciánál már volt szó az ellenfelek viselkedéséről, sőt még az animáció metódusok lejátszásáról is. Az `EnemyAnimation` osztályban lévő metódusok felelősek az összes lehetséges animációk lejátszásáért : `Walk()`, `Attack()`, `Run()`, `Idle()`, `Dead()`. A különbség itt annyi lesz, hogy a metódus paraméterében állítjuk igazra vagy hamisra az animator controllerben lévő paraméterek *condition* értékét. Ez sokkal könnyeb kezelhetőséget ad, hisz például mikor a `Chase()` fázisban van az ellenfelünk, akkor a `Run()` metódusnak csak egy `true` értéket kell adni paraméterben.

```
public void Run(bool run)
{
      anim.SetBool(„Run”, run);
}
```



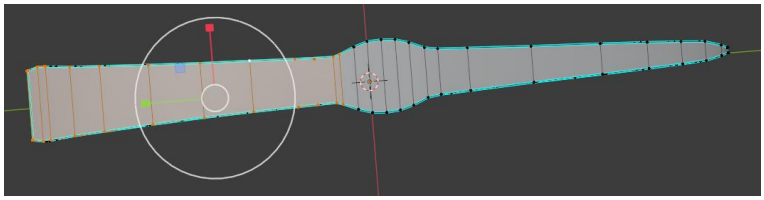

*4.3. ábra - Farkas animator controller*

[43] https://assetstore.unity.com/packages/3d/characters/animals/animal-pack-deluxe-99702 – Elérési ideje: 2021. 11.23 -

# 5. 3D modell készítése Blender-ben

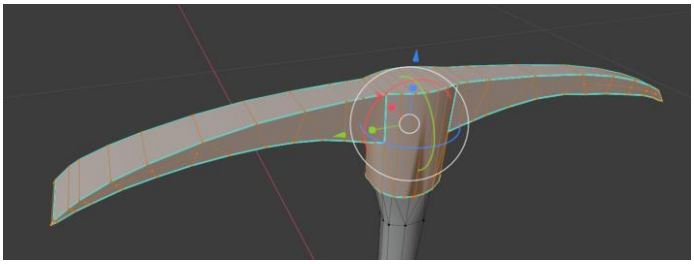
A **Blender** egy háromdimenziós grafikai program, mely nem csak modellezésre, de animálásra, vizuális effektek létrehozására, vagy akár 3D nyomtatásra is használható. Képes a *poligonális*[44] testek gyors, felosztás alapú modellezésére. Az egyetemi képzés során a számítógépes grafika kurzuson tanultam alapvető modellezést a *Blender*-ben (textúrázáson kívül). Mivel most csak a modellezésre használtam fel a programot, ezért csak a „*Modelling*" munkaterületet mutatom be, a textúrázást Unity-ben végeztem el, már meglévő textúrákkal és *materialokkal*.

A ***Scale*** opcióval tudjuk a kijelölt felület vagy egy objektum méretét módosítani (nagyítani vagy kicsinyíteni). Például a csákány egyik részét, ha nagyítani szeretnénk, akkor a *Shift+bal klikk* segítségével kijelöljük, akkor méretezni tudjuk.



*6.1. ábra - Scale*

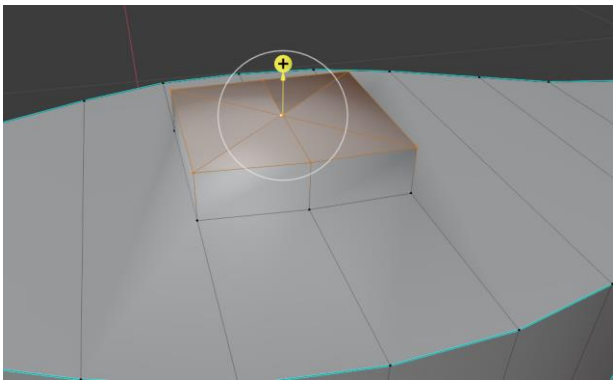
A ***Transform*** sokkal hasznosabb, mert egyszerre biztosítja nekünk a forgatást, méretezést és fogást is. Itt tudjuk például a nyilak segítségével mozgatni a kijelölt éleket, felületeket. Ezt az opciót használtam a legtöbbször, mivel a csákányt ívesen kellett kialakítani, amire a *rotate* állt rendelkezésre és közben felváltva kell kihúzni az éleket és lapokat, hogy megnöveljük a felületet.



*6.2. ábra - Transform*

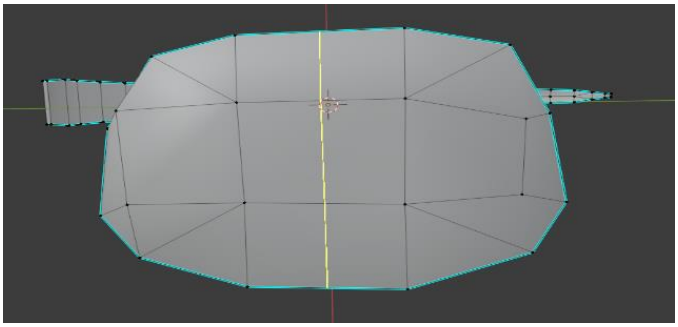
Az ***Extrudenál*** (nyújtás), ha ki van jelölve egy *vertex*[45], felület vagy él, akkor az *extrude* opció megkétszerezi azt a kijelölt részt nekünk, egy hidat képezve az új és a régi darab között.

[44] Sokszögű
[45] Csúcs, csúcspont



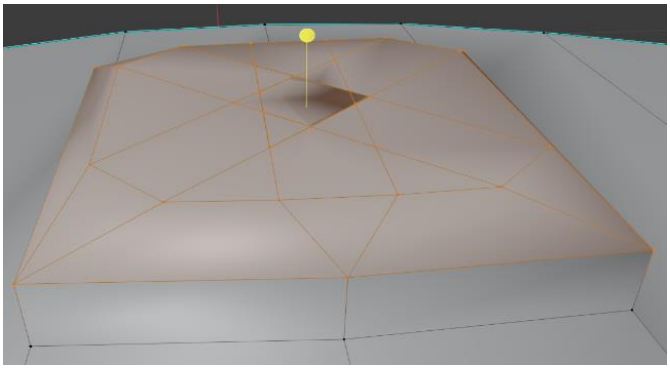
*6.3. ábra - Extrude*

A ***Knife***, vagyis a kés is egy hasznos eszköz, amikor irányítottan szeretnénk újabb éleket létrehozni. Az ***Loop cut***, vagyis élhurok készítő is hasonló eszköz, de ez egy irányban osztja fel a felületet. Itt a csík azt jelenti, hogy találtunk egy loop-ot, vagyis egy felületsort. Kattintás után vágtunk egy új élt a felületre, amit ugyan úgy tudunk majd szerkeszteni.



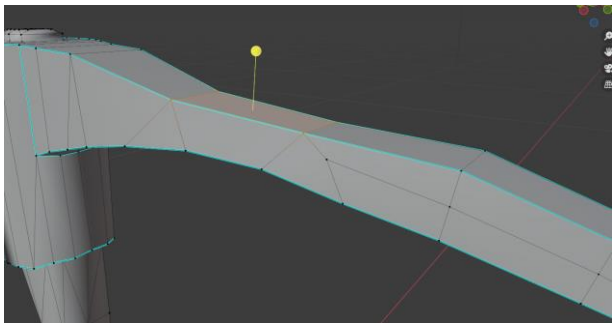
*6.4. ábra - Loop Cut*

A **Bevel** eszköz lehetővé teszi, hogy a geometrián ferde vagy lekerekített sarkokat hozzunk létre. Ezt ritkábban használtam, de objektumok lekerekítésére tökéletes.



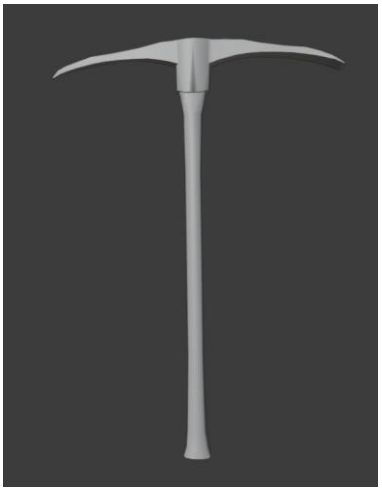
*6.5. ábra - Bevel*

Az **Edge Slide** (élcsúsztatás, mozgatás) „átcsúsztat” egy vagy több élt a szomszédos felületeken, néhány korlátozással. A csákány felső részénél is lehet ezt alkalmazni, ha egy picit formázni szeretnénk a két végét.



*6.6. ábra - Edge slide*

A **Smooth** eszközzel elsimítani tudunk az **Inset Faces**-zel pedig új, hasonló felület tudunk beilleszteni a kijelölt felületre. A leírtakat használtam főleg a csákány és a balta modell létrehozására vagy a meglévő letöltött modellek átalakítására is.



*6.6. ábra - Csákány modell Blender-ben*

A kész modellt **OBJ** formátumban exportáltam, amit a Unity könnyen felismer. Itt a már meglévő, letöltött materialokat ráraktam a modellre egy kis beállítás után és kész is a csákányunk.

*6.6. ábra - Kész csákány modell materiallal*

# 6. Összegzés

A szakdolgozatomat igyekeztem úgy bemutatni, hogy egy olyan olvasó is megértse, aki még nem találkozott *Unity*-vel. Éppen ezért éreztem szükségességét a Unity játékmotor rövid bemutatásának. Azt is fontosnak tartottam, hogy a szakdolgozat elején tisztában legyen a játék céljával, valamint a teszteléshez szükséges irányításokkal, gépigénnyel. Végeredményként úgy érzem sikerült olyan játékszoftvert készítenem, ami funkcionalitását és grafikáját tekintve megállja a helyét a játékmegosztó *platformokon* mint ingyenes, egyszemélyként fejlesztett, legalacsonyabb költségvetésű kezdő játék.

A fejlesztés előtt csak *2D* játékfejlesztési tapasztalatom volt. Már a projekt elején rá kellett jönnöm, hogy első alkalomként egy *3D high-polly* játékkal nagyobb fába vágtam a fejszém. Az fejlesztési idő nagy része utánajárásokkal, *Udemy* kurzus, *youtube* videók és dokumentációk nézésével telt. Azonban azt vettem észre, hogy egyre több időt töltök a projektemmel, többször előfordult, hogy egész nap, reggeltől estig vele foglalkoztam és volt olyan hónap, amikor minden egyes nap, több órát is fejlesztettem. Úgy érzem a képzés alatt ez volt az a projekt (az egyetemen kívüli hobbiprojekteket is beleértve) amiből a legtöbbet tudtam fejlődni programozás szinten. Egyre mélyebbre jutva jöttem rá, hogy milyen nehéz és időigényes is egy komplex *3D* játék létrehozása és mennyire becsülendő a játékfejlesztők munkája.

Mivel a szakdolgozatommal elsősorban nem az volt a célom, hogy animáljak, vagy profi grafikai modelleket készítsek és ezt nem is tudtam volta megvalósítani a rendelkezésemre álló idő alatt sem, ezért a hangsúlyt a funkciók leprogramozására, valamint a Unity széles körű használatára fektettem. Szerettem volna nagyrészt azokkal a területekkel foglalkozni, amik az egyetemi képzés alatt elsajátíthatók voltak és így be is mutatható az elsajátított tudás is. Éppen ezért az animálást és a modellezés 90%-kát *Unity Asset Store*-ból való beszerzéssel spóroltam meg (forráslinkek a lábjegyzetekben).

A játékban azonban még így is rengeteg lehetőség van, amikre már nem maradt időm, vagy a jelenlegi tudásommal még nem tudom megvalósítani. Lehetne például táska funkciót készíteni, ami megnöveli az *invetory* méretét, de le is lehessen venni azt az eszközök megmentésével. A játékban lévő eszközök egy idő után törjenek el és törlődjenek az eszköztárból, hogy mindig újat kelljen barkácsolni. További fejlesztési lehetőség lehetne még például, hogy az egész játékba egy többjátékos módot is implementáljunk. Ezesetben viszont a legtöbb funkció kódját át kellene írni és szervert is biztosítani. Ezeket a fejlesztéseket a jövőben valószínűleg meg is próbálom valósítani.

# *Irodalomjegyzék*


[1] Wikipédia – Videójáték – Elérési ideje: 2021. 11. 27. - https://hu.wikipedia.org/wiki/Vide%C3%B3j%C3%A1t%C3%A9k

[2] Wikipédia – Unity (game engine) – Elérési ideje: 2021. 11. 27- https://en.wikipedia.org/wiki/Unity_(game_engine)

[3] Unity Documentation – Configurable Enter Play Mode – Elérési ideje: 2021. 11.23 - https://docs.unity3d.com/Manual/ConfigurableEnterPlayMode.html

[4] Unity Documentation – GameObjects – Elérési ideje: 2021. 11. 27- https://docs.unity3d.com/Manual/GameObjects.html

[5] Unity Documentation – Prefab – Elérési ideje: 2021. 11. 27- https://docs.unity3d.com/Manual/Prefabs.html

[6] Unity Documentation – Introduction to components – Elérési ideje: 2021. 11.11 - https://docs.unity3d.com/Manual/Components.html

[7] Unity Documentation – MonoBehaviour – Elérési ideje: 2021. 11. 27- https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[8] Unity Forum – Elérési ideje: 2021. 11.27 - https://forum.unity.com/

[9] Raywenderlich – Runtime Mesh Manipulation With Unity – Elérési ideje: 2021. 11.23 - https://www.raywenderlich.com/3169311-runtime-mesh-manipulation-with-unity

[10] Magyar Blender kuckó – Blender leírások – Elérési ideje: 2021. 11. 27- https://blender.hu/index.php?page=tut&l=hdoc/modalap

[11] Blender Documentation – Elérési ideje: 2021. 11. 27- https://docs.blender.org/

[12] Wikipédia – Blender – Elérési ideje: 2021. 11. 27 - https://hu.wikipedia.org/wiki/Blender_(program)

[13] YouTube – mr. Gyarmati – Elérési ideje: 2021. 11. 27 - https://www.youtube.com/channel/UCWsBFX2clAUfU88QRHrwdRw

[14] YouTube – Vézna Kazuár – Elérési ideje: 2021. 11. 27 - https://www.youtube.com/channel/UCHNOGrafIIdrSEUAULtfh0Q

## *Nyilatkozat*

Alulírott Potyesz Máté gazdaságinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Számítógépes grafika Tanszékén készítettem, gazdaságinformatikus 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 a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

*Potyesz Máté*

Dátum 2021.11.23 ...……

Aláírás