



EÖTVÖS LORÁND  
TUDOMÁNYEGYETEM  
INFORMATIKAI KAR  
PROGRAMOZÁSI NYELVEK ÉS  
FORDÍTÓPROGRAMOK TANSZÉK

## Hálózaton játszható szerepjáték Unity alapon

**Témavezető:**

Pataki Norbert

Adjunktus, PhD

**Szerző:**

Prucs Ákos András

Programtervező Informatikus Bsc

Budapest, 2022

Bevezetés	4
Motiváció a témaválasztás mellett	4
A szakdolgozat témája	4
Felhasználói dokumentáció	6
Rendszerkövetelmények	6
A program telepítése és indítása	6
Szerver létrehozásához szükséges hálózati beállítás	7
A játék célja	7
Irányítás	7
Felvehető tárgyak	8
Színkód	8
Fejlesztői dokumentáció	10
Unity bemutatása	10
Netcode for Gameobjects bemutatása	11
Felhasználói esetek	12
Program szerkezete	14
Modell	17
Item és csomagoló osztályok (collider, rigidbody, spawn, INetworkSerializable, NetworkBehaviour, NetworkObject, NetworkTransform)	17
Inventory (események, delegációs függvények)	20
Chest	21
Enemy (logika, navmeshagent, prefab)	24
Bullet Movement (spawn stratégia)	26
Damageable (komponens)	22

Player Controller (input, iflocalclient, rotation, kamera, egyedi collision detection, prefab, NetworkClientTransform)	28
UiHealthBar, UiExperienceBar	<b>Hiba! A könyvjelző nem létezik.</b>
UiInventory (drag and drop)	31
ItemAssets (Spriteok, felbontások)	32
Pálya felépítése (tilemap, collider, layers)	23
Game Manager	32
Connection Manager	32
UI Manager	32
Játék tesztelése	33
Játék bővíthetősége	33
Hivatkozások	34
Köszönetnyilvánítás	35

## Bevezetés

### *Motiváció a témaválasztás mellett*

A videójátékok már gyerekkorom óta részei az életemnek. Hálával tartozom, amiért általuk olyan sok barátot sikerült megismernem. Barátokat, akik noha különböző utakat járnak és másfajta életet élnek, de a közös hobbink összehoz minket. A játék remek kapocs egy közösségben és segít egy nyelvet beszélni. Szerencsésnek tartom magam, amiért már olyan korán belecsöppentem ebbe a világba.

A játékfejlesztés gondolata gyakran megfordult a fejemben. Mindig is érdekelt a kérdés, hogy vajon hogyan készül el egy-egy játék, próbáltam mögéjük látni, kíváncsi voltam az alkotásuk mögötti technikai részletekre. Sokáig azonban nem éreztem magam késznek, hogy saját magam is belevágjak egy játék elkészítésének. Megijesztett az összetettségük és kételkedtem a saját képességeimben.

Az egyetemi képzésem alatt, ahogy bővült a tudástáram, úgy egyre magabiztosabb lettem. Elkezdtem úgy érezni, hogy a gondolataimban létező akadályok eltűnnek. Többé már nem riadtam vissza a játékfejlesztéstől. Így jutottam el ahhoz a gondolathoz, hogy a szakdolgozatomban egy játék programot fogok megvalósítani.

### *A szakdolgozat témája*

A szakdolgozatomban egy „Top-down RPG” játékot fogok elkészíteni. A top-down játékok legtöbbször egy kétdimenziós világot képeznek, ahol a játékos a karakterét fenti nézetből képes irányítani. Az RPG a role-playing game avagy szerepjáték rövidítése, ahol a játékos felveheti egy képzeletbeli karakter szerepét.

A tipikus játékelemei egy ilyen műfajnak, hogy a karakter szabadon mozoghat a világban és interakciókat végezhet az abban lévő objektumokkal, más játékosokkal vagy a számítógép által irányított lényekkel. Az interakciók által megszerezhető tárgyak a játékos hátizsákjában tárolódnak. A megszerzett ruhákat és fegyvereket felszerelheti a karakterére, amik így bónuszokat nyújtanak neki a világban.

A megvalósítandó játékhoz a fent említett játékelemeket fogom megalkotni. Ehhez a Unity Technologies által fejlesztett Unity videójáték-motort fogom használni és a C# nyelvet. A hálózati kommunikációért a Unity-hez készült Netcode for Gameobjects nevezetű fejlesztői csomagot fogom használni.

# Felhasználói dokumentáció

## *Rendszerkövetelmények*

A program a Unity 2021.1.14f1 verziója alatt fut, és a rendszerkövetelmények igazodnak a verzió követelményeihez:

- Legalább Windows 7-es operációs rendszer vagy annál jobb.
- SSE2 utasításkészlettel rendelkező 32 vagy 64 bites processzor.
- DirectX10, 11 vagy 12 kompatibilis videokártya.
- Maximum 2 GB memória.
- Maximum 50 MB szabad tárhely a háttértárolón.

A Unity lehetőséget ad arra, hogy több platformmal is kompatibilis legyen az alkalmazásunk, azonban ennél a játéknál én csak Windowsra fókuszáltam.

A játék fejlesztéséhez Windows 10 operációs rendszert használtam, Intel Core i7-8750H 2.20 gigahertzes processzort, GeForce RTX 2060 típusú videokártyát és 16 GB memóriát.

## *A program telepítése és indítása*

A program nem igényel telepítést. Az állomány letöltése is kicsomagolását követően a futtatható bináris fájlal egyből indítható. A játék megnyitását követően a felhasználó egy menüben találja magát, ahol két opciója van. A „Host Game” gomb megnyomásával saját játékot indíthat. A „Join Game” gomb megnyomásával pedig lehetősége nyílik már futó, a hálózaton elérhető játékhoz csatlakoznia.

Saját játék indításakor a program kéri egy port megadását. A szerver ezen a porton figyel a kívülről érkező kliensek kapcsolódási kérévényeit és üzeneteit. Amennyiben a felhasználó egyedül kíván játszani, a portot figyelmen kívül hagyhatja és elindíthatja a játékot.

Amennyiben egy meglévő hálózathoz kíván csatlakozni a program egy címet és egy portot kér, amin a szerver elérhető.

## *Szerver létrehozásához szükséges hálózati beállítás*

Amennyiben a játékot készítő fél (host) belső hálózattal rendelkezik, szükséges lehet a porttovábbítás beállítása, hogy a kívülről érkező felek is kapcsolódni tudjanak a játékba. Ez a belső hálózatért felelős router konfigurálásával lehet megtenni. A folyamat lényege, hogy azokat a publikus címre érkező üzeneteket, melyeket az általunk megadott portra küldenek, a router továbbítsa a saját állományunkhoz.

Amennyiben a játékhoz a belső hálózatról akarnak kapcsolódni a kliensek, a porttovábbításra nincsen szükség. Csupán a hálózatunkon kiosztott IP címünket kell megadnunk nekik.

Ha a játékhoz egy gépről akarunk csatlakoztatni több klienset, akkor a hálózati beállításoktól eltekinthetünk. A játék által felkínált IP címet és portot használjuk, mivel az alapértelmezetten a lokális állományunk címe van megadva.

## *A játék célja*

Sikeres indításkor megjelenik a karakterünk az előre elkészített pályán. Két fajta pályarész létezik: a kezdő, ahonnan a játék indul és a dinamikusan megjelenő részek, avagy szobák, amik feloldódnak amint teljesítettük az adott szintet. A cél, hogy minden szobát kitisztítsunk és minél magasabb szintre jussunk el. Idővel nehezedik a játék, megnő az ellenfelek száma és erőssége, néhány körönként pedig egy különleges ellenfél jelenik meg, az eddigiekhez képest kiemelkedő értékekkel.

## *Irányítás*

A karaktert irányítani a w, a, s, d gombokkal és a nyilakkal lehet a billentyűzeten. Az irány követi az egér mozgását és támadni a bal klikk lenyomásával lehet, amennyiben rendelkezik felszerelt fegyverrel. A TAB lenyomásával megnyílik a táska, itt lehetőség van a megszerzett tárgyak használatára vagy eldobására. Amennyiben az adott tárgy egy ruhadarab vagy fegyver, akkor azt fel lehet ruházni a karakterünkre a megfelelő rubrikába húzással vagy a jobb klikkel.

### *Felvehető tárgyak*

Ahhoz, hogy minél nagyobb szintre eljussunk szükség van minél jobb minőségű ruhákra és fegyverekre. Minden ellenfél legyőzésekor esély nyílik rá, hogy tárgyakat - a továbbiakban itemeket - hagyjon maga mögött a földön. Ezeket úgy vehetjük fel, ha rásétálunk, feltéve, hogy nem telt meg a hátizsákunk, ekkor ki kell dobunk a már nem szükséges dolgokat. A pályán elhelyezkednek kincsesládák is, melyek kinyitáskor itemeket hagynak hátra a földön. Egy ládát érintkezéssel lehet kinyitni.

Fegyvereken és ruházaton kívül három másik típusú item vehető fel a földről:

- Health Potion: Használatkor feltölti a karakterünk életerejét.
- Gold: Nincs különösebb haszna.
- Experience point: Növeli a karakterünk tapasztalatát.

A kör elején maximum életerővel rendelkezünk. Amennyiben a harcok során az életerő nullára csökken a karakterünk veszít, elvesznek a tárgyai és a kezdő pályarészen indul újból. Egy másik játékelem a tapasztalat pont. Lehetőségünk van a karakterünk erősítésére azáltal, hogy tapasztalatot gyűjt. Amikor elér egy mennyiséget szintet lép és erősödik. Ebben a játékban szintlépéskor megnő a mozgási sebességünk.

### *Színkód*

Ahogy az egy tipikus RPG játékban lenni szokott, a ruhák és fegyverek minőségét a színük adja meg. A minőséget négy kategóriába van sorolva a gyengétől a legerősebbig:

- Common, szürke
- Good, zöld



- Rare, lila
- Legendary, sárga

# Fejlesztői dokumentáció

## *Unity bemutatása*

A Unity elsősorban a játékfejlesztésről híres, azonban az alkalmazást használják animációk készítéséhez, építészeti vizualizációkhoz és szimulációkhoz is. A motor képes két- és háromdimenziós „világok” megjelenítésére is. A játékom elkészítéséhez a kétdimenziós konfigurációt választom és a *Unity Editorban* végzem a fejlesztést.

A Unity által szimulált világban megjelenő elemek a **GameObjectek** (a későbbiekben sokszor objektumként is hivatkozom rájuk). Ezeket az objektumokat képesek vagyunk egymásba ágyazni és ezáltal fa struktúrákat kialakítani, ezzel megkönnyítve az összetett tárgyak elkészítését. GameObjectek egy általunk megadott gyűjteményét **jelenetnek** nevezünk és könnyedén válthatunk jelenetek között futási időben és a Unity Editorban is. Egy egyszerű példa a *jelenetek* alkalmazására egy videójáték esetében a főmenü és a játék jelenetének váltakozása. Az általam elkészített játékban egyetlen jelenetet használok és kód segítségével futási időben változtatom annak állapotát.

Ahhoz, hogy egy GameObjectet tulajdonságokkal ruházzunk fel **komponenseket** kell hozzárendelnünk. A Unityben rengeteg előre elkészített komponens található. Némelyek csupán egy egyszerű geometriai alakzatot rendelnek hozzá az objektumunkhoz és vannak, amik bonyolultabb logikai képességeket. Minden GameObject tartalmaz egy *Transform* komponenset, ami a világban lévő pozícióját, rotációját és méretét tartja számon. Játékoknál és szimulációknál hasznos komponensek a Colliderok, amik detektálják és jelzik, ha két objektum összeütközik. Valamint a Rigidbody komponensek, amik a gravitációt és fizikai erőt szimulálni képes logikával rendelkeznek.

Lehetőségünk van saját logikát létrehozni, majd egy komponens formájában hozzárendelni a GameObjecthez. Ehhez egy olyan C# nyelven írt osztályt kell létrehoznunk, amely leszármazottja a Unity könyvtárában lévő **MonoBehaviournak**. A leszármazottak öröklik a *Start* és *Update* metódusokat. Az előbbi egyszer fut le, az objektum inicializálásakor, ellenben

az *Update* minden képkocka frissítéskor lefut. Ezeket a metódusokat nem kötelező használni minden *MonoBehaviour* leszármazottnál, azonban hasznos grafikai elemeknél és a felhasználótól érkező input feldolgozásánál.

Még egy említésre való elem a Unityben a **Prefabek** jelenléte. A Unity Editorban előre elkészíthetünk adott komponensekkel felszerelt *GameObject*-et és elmenthetjük későbbi használatra. A Prefabek segítenek elkerülni a kód redundanciáját és futási időben elérhetőek. A későbbiekben én is alkalmazom őket a játékos karaktere előkészítéséhez vagy a pályán elhelyezkedő szobák elkészítéséhez.

## *Netcode for Gameobjects bemutatása*

A Netcode for Gameobjects – továbbiakban Netcode – egy magas szintű fejlesztői csomag, ami hálózati képességeket nyújt a *GameObject* és *MonoBehaviour* alapú munkafolyamatoknak. A használatához a jelenetben létre kell hozni egy **NetworkManager** komponenssel (scripttel) rendelkező *GameObject*-et, ami egy *Singleton* osztály és elérhető lesz a hálózat bármelyik oldalán.

A hálózaton szereplő feleknek három fajtája létezik: *client (kliens)*, *server (szerver)* és *host*. A játékban azonban csak a játékot készítő *host*, és az ahhoz kapcsolódó *client*-ek lesznek jelen. A *host* egy olyan fél, amin egyszerre fut a *server* és a *client* programja. Amikor lentebb a szerverre utalok, akkor a *host* állományán futó szerverre gondolok. Amikor *kliensről* van szó, akkor a szerverhez csatlakozó állományokra és a *host*-on futó *kliens*-re is gondolok egyaránt.

A Netcode kibővíti a *MonoBehaviour* osztályt egy **NetworkBehaviour** osztály kiterjesztésével, ezáltal a hálózati működéshez szükséges funkcionalitást hozzáadva az objektumoknak. A *NetworkBehaviour* használatának feltétele, hogy az adott *GameObject* vagy egy szülője rendelkezzen a **NetworkObject** komponenssel, ami futási időben egy a hálózat minden felén megegyező azonosító számot rendel az objektumhoz.

A *NetworkBehaviour* rendelkezik az *OnNetworkSpawn* és *OnNetworkDespawn* metódusokkal, amik **spawnolás** és **despawnolás**kor hívódnak meg. Unityben, ha egy prefabet létre akarunk hozni a jelenetben, ahhoz a UnityEngine könyvtár statikus *Instantiate*

metódusát kell meghívunk. Ez viszont nem elég, ha a hálózat minden felén létre akarunk hozni egy objektumot. Ehhez regisztrálnunk kell a prefabet a NetworkManager konfigurációjában, majd ezt követően a NetworkObject *Spawn* metódusával minden kliensnél létrehozhatjuk az objektumot.

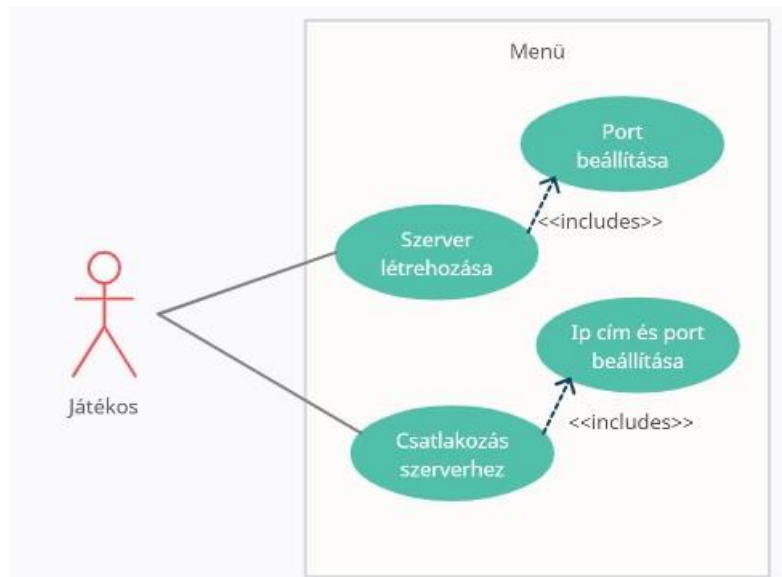
Egy felmerülő probléma, hogy ha minden félnél jelen van az objektum, rajta a logikáért felelős kóddal, akkor hogyan különítjük el a szerver és a kliens feladatait. Erre nyújt megoldást a NetworkManager *IsClient*, *IsServer*, *IsHost* boolean típusú változói, melyekkel eldönthetjük, hogy éppen a hálózat melyik oldalán fut a kódunk. A játékban legtöbbször ezek segítségével választom el a szerver (ez esetben host) és a kliens logikáját. Az ellenfelek esetében például a host végzi a logikáért felelős számításokat, a kliens csupán lefrissíti az ellenfél pozícióját az *Update* metódussal.

A kommunikáció állományok között távoli eljárashívással vagy **RPC**-vel (Remote Procedure Call) történik. A Netcode megkülönböztet Client RPC-t és Server RPC-t. Az előbbit a szerver hívja meg, és a kliensek állományán fut le a kód. Az utóbbit egy kliens hívja meg és a szerveren fut a kód. Ahol pedig szükség van állapotokra és azok szinkronban tartására a felek között ott a Netcode könyvtár által biztosított **NetworkVariable<T>** generikus változót használom. Ennek előnye, hogy csak a szerver rendelkezik írás joggal, valamint állapotváltozáskor jelzést kapnak a kliensek is, hogy reagálni tudjanak rá.

A Netcode további, általam felhasznált részeit lentebb fogom részletezni, ahol egyből a felhasználásukról is írni tudok.

## *Felhasználói esetek*

A felhasználói esetek diagram két részből áll. A menüből, amivel a felhasználó a játék indításakor találkozik, és a játék eset diagramjából.

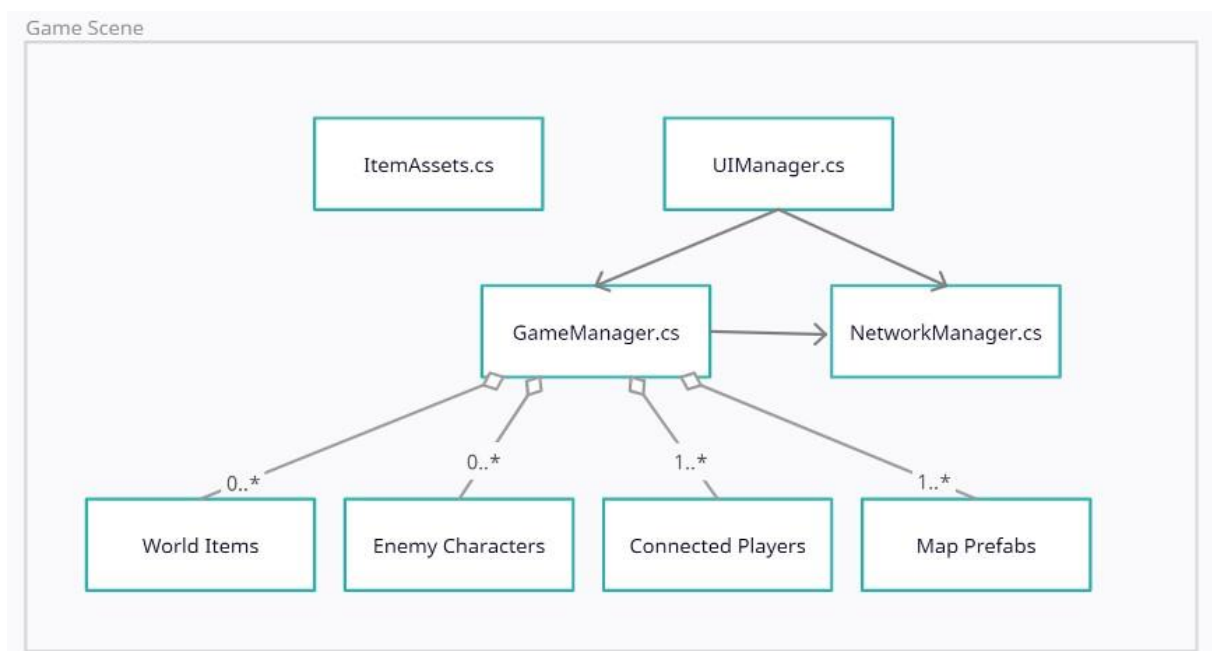


A menüben a felhasználónak két lehetősége van, létrehozhat egy saját játékot a host szerepeként, vagy csatlakozhat egy már létező játékhoz a hálózaton. Saját játék létrehozásakor a felhasználónak be kell állítani egy portot, amin a szerver figyelheti a bejövő üzeneteket. A szerver portja alapértelmezetten 7777. Már meglévő játékhoz csatlakozásnál a program egy IP címet és egy portot kér a felhasználótól. Alapértelmezetten az IP cím a lokális állomány címe és a port megegyezik a fent említettel.

Játék létrehozásakor vagy sikeres csatlakozáskor a felhasználó a játékban találja magát. Itt lehetősége van a karaktere mozgatására a nyilak vagy a w, a, s és d gombokkal. A bal klikk lenyomásával támad a karaktere. A tabulátor lenyomva tartásával megjelenítik a táskáért felelős felhasználói felület. Itt lehetősége van a jelenleg nála lévő tárgyak használatára vagy eldobására. Amennyiben a felhasználók mind kiesnek a játékból az újraindul és megpróbálhatják meg egyszer. Több játékos esetén, ha az egyikük kiesik, akkor elveszíti a karaktere fölötti irányítást és a kamera a még bent lévő játékosokat figyeli.

## Program szerkezete

Az alábbi részben átfogóan bemutatom a játék felépítését, a különálló részeit és azok kapcsolatát egymással.



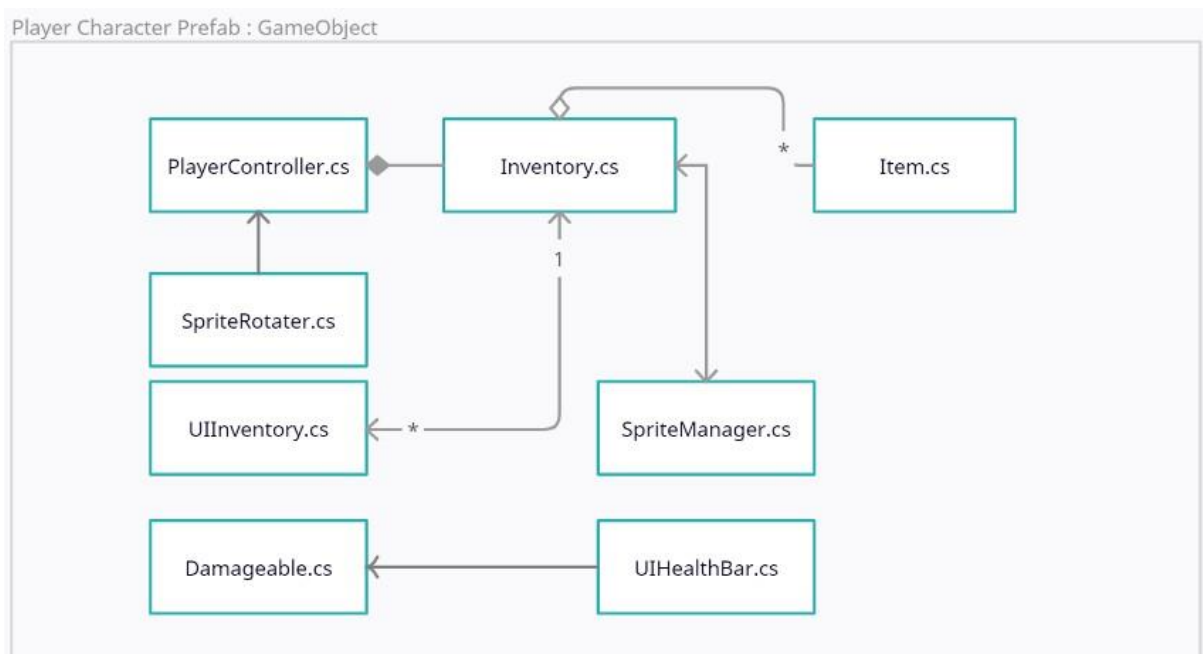
A játék jelenetének felépítése.

Korábban említettem, hogy a Unity által kínált jelenetek funkcionalitását nem fogom kihasználni, ugyanis én egyetlen jelenetet használok és az általam írt kód felel ennek a jelenetnek a váltakozásáért. A fenti ábrán a jelenet felépítését ábrázolom. A „.cs” végződésű elemek C# kódok (scriptek) komponensként a saját GameObjectjeiken. Ezek az osztályok jelen

vannak a programban az indítástól kezdve a bezárásig. Elérhetőek bárhol a jelenetből, ezért információt juttathatnak más osztályoknak. Erre példa az `ItemAsset` osztály, aminek feladata indításkor betölteni szükséges Prefabeket és a játékban később felhasználandó grafikai elemeket.

A játék indulását követően a felhasználót eligazító egyszerű menü az `UIManager` osztály felelőssége. Kommunikál a `NetworkManager` osztállyal a hálózati beállításokról és tájékoztatja a `GameManager` osztályt a játék indulásáról. Az utóbbi osztály főképp szerveroldali kódot tartalmaz. Figyelemmel kíséri a játék állapotát, létrehozza az ellenfeleket, legenerálja a pályát és szamontartja a csatlakozott játékosokat. Amikor egy felhasználó sikeresen csatlakozik a játékhoz, a `GameManager` létrehoz neki egy karaktert az előkonfigurált `Player Prefab` alapján és átadja a tulajdonjogot az adott kliensnek.

A `NetworkBehaviour` egyik adottsága, hogy a létrehozó fél tulajdonjoggal rendelkezik felette, amit később átadhat más feleknek is. Egy `NetworkBehaviour` leszármazott osztály tulajdonjogával rendelkező kliens kaphat írásjogot a saját hálózati változóira, egyedül ő hívhat meg RPC-eket az adott scriptből és ellenőrizhetjük éppen melyik állományon fut a kód az örökölt `IsOwner` és `IsLocalPlayer` változókkal.



A játékos karakterén lévő komponensek és azok viszonyai egymással.

Amikor egy fél csatlakozik a játékba a *Player Prefab* alapján a GameManager létrehoz neki egy karakter objektumot. Ez egy több egymásba ágyazott GameObjectból álló egység. A hierarchia legtetetjén helyezkedő szülő objektum tartalmazza a fő komponensét a játékosnak, a **PlayerController** osztályt. Ez egy MonoBehaviour osztály, ami felel az input feldolgozásáért, a táska (Inventory) osztály példányosításáért és információt továbbít a közvetlen alatta lévő gyermek GameObjecteknek. A legfelső szinten még egy különálló osztály a **Damageable** amely sebezhetővé teszi az objektumot a játékban, valamint kapcsolatban van a felhasználói interfészt nyújtó *UIHealthBar* osztállyal. Az utóbbinak feladata hogy egy életerő sávot rajzoljon a karakter fölé a Damageable osztály információi alapján.

A karakter felrajzolásáért szintén egy gyermek objektum felel. Ez tartalmazza az alap karakter és a ruha rétegekért felelő objektumokat. Két komponensem van az említett GameObjecten: a **SpriteManager**, ami szemmel követi az Inventoryban a karakteren szereplő ruhákat. És a **SpriteRotater**, ami a PlayerContorllertől elkéri az egér jelenlegi pozícióját és abba az irányba forgatja a játékos karakterét.

Egy szintén UI (User Interface) elem a táska megjelenítéséért és annak kezeléséért szolgáló **UIManager** osztály. Ennek feladata, hogy lehetőséget biztosítson a felhasználónak a táska kezelésére. Az osztály jelen van játék közben és a tabulátor lenyomásakor aktiválja a panelt, amivel a táskát ábrázolja.



Az ellenfelek objektumán szereplő komponensek.

A játékmenetért felelős GameManager osztály a szintekre ellenfeleket hoz létre. Ezeket az Enemy Prefab alapján hozza létre és ezt követően felkonfigurálja a paramétereit – ezzel



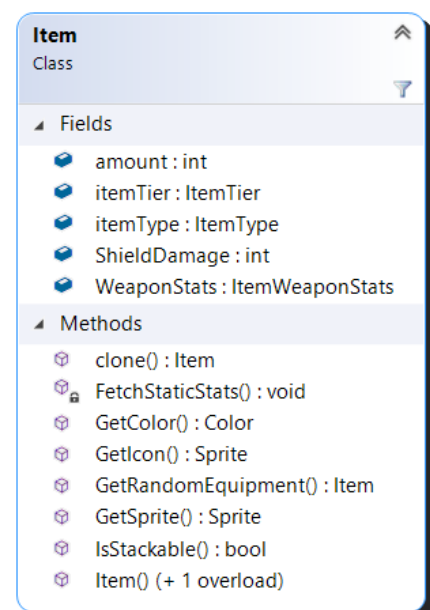
beállítva az adott egység nehézségi fokát. Az ellenfél objektuma kevésbé összetett a játékoséhoz képest. Fő komponense az **Enemy** osztály, ami vezérli az adott NPC-t (Non-Playable Character). Érdeemes megemlíteni, hogy főként szerver oldali logikát tartalmaz, ugyanis a klienseknek nem lehet beleszólása egy ellenfél mozgásába. Míg a szerver kiszámolja az ellenfél mozgását, addig a kliensek csupán kiolvassák a pozícióját és irányát a hálózati változókból.

## Modell

A továbbiakban részletezni fogom az általam készített komponensek működését és technikai részleteit. Az egyszerűbb scriptektől fogom kezdeni és haladok az egyre összetettebb komponensekig.

### Item és csomagoló osztályok

Az **Item** osztály írja le a játékban létező tárgyak tulajdonságait. Ezek a tárgyak elhelyezkedhetnek a játékos táskájában vagy a pályán begyűjthető elemenként. Három fontos változója a *mennyiség*, *típus* és *minőség*. Típustól függően értéket kapnak a fegyver attribútumok vagy a pajzs által levédett sebzés értéke, ezeket a *FetchStaticStats* metódussal töltöm be és a ruhák/fegyverek értékeit statikus változóiban tárolom. Minden típusú és minőségű itemhez tartozik két Sprite (kép), ami a játékban megjelenik. Egy a földön felvehető tárgyak képe, a másik a karakteren elhelyezkedő ruha réteg képe. Ezeket az elemeket a *GetSprite* és *GetIcon* metódusok adják meg az *ItemAssets* osztálytól való lekérdezéssel.



Ahhoz, hogy egy tárgyat megjelenítsek a játék pályáján elhelyezve, az **ItemWorld** osztályt használom. Létezik egy ItemWorld Prefab objektum is, amin a megfelelő Unity által nyújtott komponensek elő vannak készítve. A komponensek sorban a SpriteRenderer, TextMeshPro, Collider2D, Rigidbody2D és NetworkTransform.

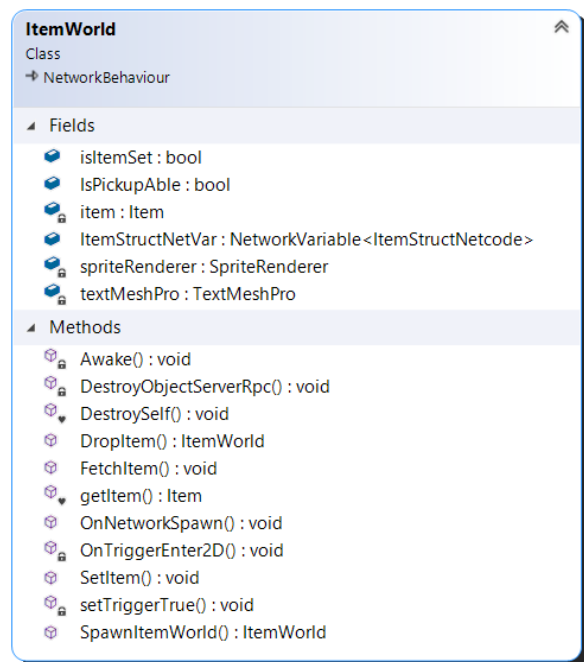
A **SpriteRenderer** feladata egy kép megjelenítése a GameObjecten. A többi komponenshez hasonlóan az azonos objektumon lévő kóddal el lehet érni és futási

időben dinamikusan lehet változtatni a megjelenítendő képét. Jelen esetben az ItemWorld által tartalmazott Item *GetSprite* metódusát jeleníti meg a komponens. A SpriteRenderert több helyen is használom, például a játékos karakterének kirajzolásakor, az ellenfelek megjelenítésekor, de a pálya kirajzolásáért felelős TileMap is hasonló Renderer osztályt alkalmaz.

A **TextMeshPro** egy nagyfelbontású szövegdobozt nyújtó szolgáltatás. Jelen esetben arra használom, hogy a felvehető és halmozható tárgyak (Gold, Health potion, Experience point) mennyiségét kiírjam a megjelenítendő Sprite elé.

A **Collider2D** az ütközés érzékelésért felelős Unity komponens. Amikor átfedésbe kerül, vagy ütközik egy másik Collider objektummal akkor azt egy felülírható metódus meghívásával jelzi. Ez a metódus az *OnTriggerEnter2D* és jelen esetben arra használom, hogy falka ütközéskor megállítsam az objektumot.

Az előző komponens gyakori párja a **Rigidbody2D**, ami fizikai tulajdonságokkal ruházza fel az objektumunkat. Arra használom, hogy egy tárgy eldobásakor egy erőt adjak a komponensnek egy tetszőleges vagy az egér irányába. Ez az erő a beállított gravitációs paraméterektől függően idővel elhalványul, vagy pedig falba ütközik az objektum és megáll a mozgásban.



Ahhoz, hogy a hálózaton mindenhol példányosuljon az ItemWorld objektum, nem elég csupán az egyik kliensen létrehozni. A GameObjectnek adni kell egy **NetworkObject** komponenst és regisztrálni kell a Prefabet a Netcode által biztosított NetworkManager objektumon. Így már minden kliensen létrehozhatjuk a Prefabet a *Spawn* módszerrel.

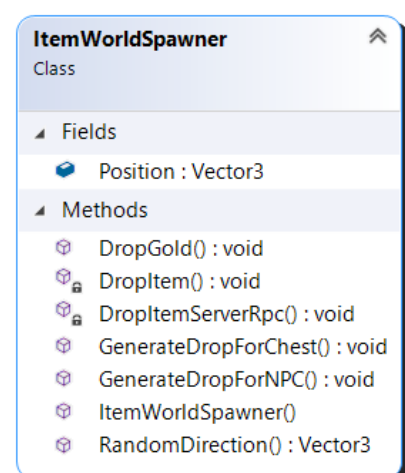
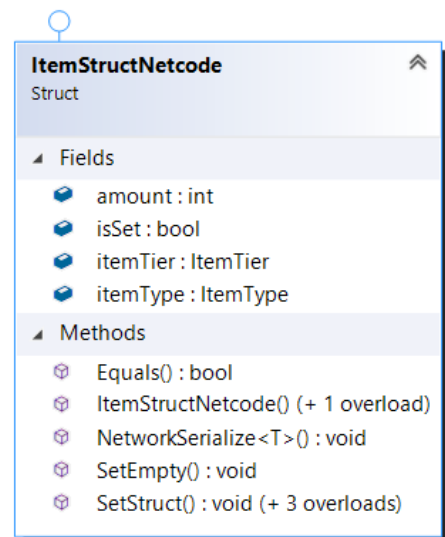
Jelen van egy úgynevezett **NetworkTransform** komponens is, amit a Netcode biztosít az egyszerűbb használat érdekében. Ez megoldja a GameObject Transform komponensének (pozíció, rotáció, méret) szinkronizálását a hálózaton, így ezzel nem kell törődnünk.

Az ItemWorld osztálynak kell egy hálózati változó, ami az állapotát tárolja és szinkronban van minden kliensen. Ehhez használom az *ItemStructNetVar:NetworkVariable<T>* generikus változót.

Alapértelmezetten a Netcode csak primitív típusok tárolását engedi ezekben a típusokban. Ahhoz, hogy az Item tulajdonságait egy hálózati változóban tároljuk létre kell hoznunk egy struktúrát, ami megvalósítja az **INetworkSerializable** interfészt. A *Struct* nyelvi elemre azért van szükség, mert egy olyan típus kell, ami nem veheti fel a *null* értéket.

Így, hogy a hálózat minden felén létrejön az ItemWorld objektum a megfelelő Item tulajdonságokkal már csak egy konverzió kell **ItemStructNetcode** és Item között, amit a konstruktorokkal oldok meg.

Az ItemWorld objektumok spawnolásához még létrehozok egy erre való osztályt, az **ItemWorldSpawner**nt. Erre azért van szükség mert a spawnoláshoz csak a szervernek van jogosultsága, ezért kell egy modul, ami RPC-vel értesíti a szervert, ha a kliens egy ItemWorld objektumot szeretne létrehozni. Ezt az osztályt minden alkalommal példányosítom, amikor egy tárgyat szeretnék a pályán létrehozni, majd ezt követően a C# személggyűjtőjére bízom a memória felszabadítását.



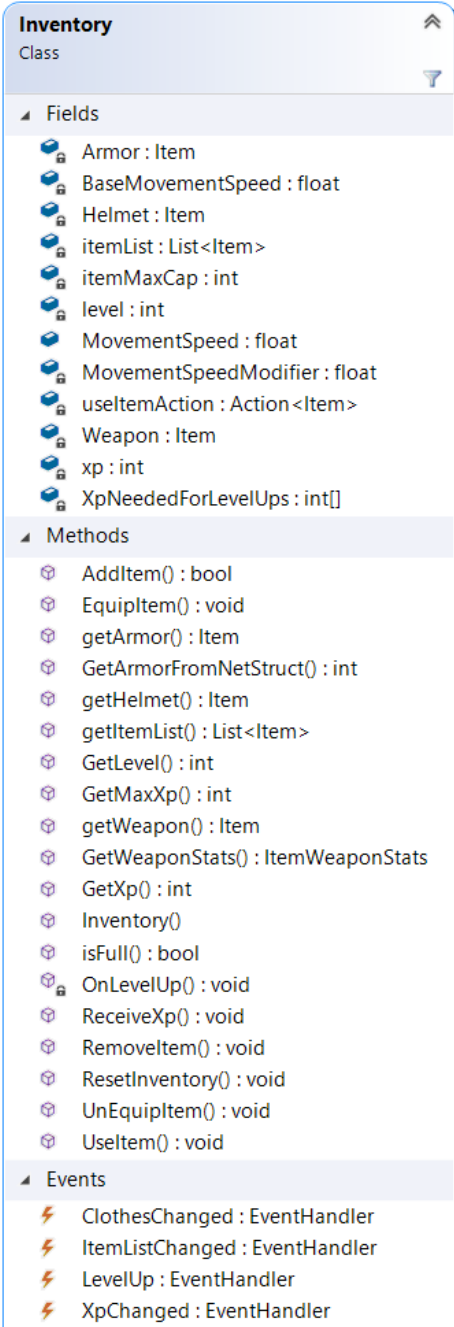
Külön metódust írtam arra az esetre, ha egy ládát nyit ki a játékos vagy egy ellenfelet győz le, mivel az akkor kidobott tárgyak minősége és mennyisége véletlenszerű. Az esélyek százalékát az Item osztály statikus mezőiben tárolom, mivel ott könnyen elérhetőek és egyszerűen lehet kísérletezni velük.

## Inventory

Az **Inventory** osztály tárolja el a játékos által összegyűjtött Itemeket, a felhúzott ruhákat és kézben tartott fegyvert, a játékos tapasztalati pontját, amit ellenfelek legyőzéséért és ládák kinyitásáért kap valamint egyéb játékot módosító elemeket mint a mozgási sebesség és testen lévő páncélzat értéke. Az Inventory egy önálló osztály, ami nem rendelkezik függőségekkel a Unity iránt.

Az Itemeket egy generikus listában tárolom az interakciókért pedig az *AddItem*, *RemoveItem*, *EquipItem* és *UnequipItem* metódusok felelnek, melyeket a játékos közvetett módon irányíthat. Ezeket a metódusokat használok, amikor a felhasználó karaktere megérint egy tárgyat a földön vagy ha a hátizsák felhasználói interfészén kezeli az ott megjelenő tárgyait. Az Add- és RemoveItemhez egy egyszerű keresés tételt alkalmazok majd attól függően, hogy az adott tárgy halmozható változtatom az Item mennyiség változóját vagy törlöm/hozzáadom az Item listába.

Az Itemekkel interakcióba lehet lépni a UI-n keresztül. Ekkor az Inventory UseItem(Item) **delegációs függvényét** hívom meg. Ennek a



The screenshot displays the Unity Inspector for the **Inventory** class. It is organized into three sections: Fields, Methods, and Events.

- Fields:** Lists 12 public fields: `Armor : Item`, `BaseMovementSpeed : float`, `Helmet : Item`, `itemList : List<Item>`, `itemMaxCap : int`, `level : int`, `MovementSpeed : float`, `MovementSpeedModifier : float`, `useItemAction : Action<Item>`, `Weapon : Item`, `xp : int`, and `XpNeededForLevelUps : int[]`.
- Methods:** Lists 20 public methods: `AddItem() : bool`, `EquipItem() : void`, `getArmor() : Item`, `GetArmorFromNetStruct() : int`, `getHelmet() : Item`, `getItemList() : List<Item>`, `GetLevel() : int`, `GetMaxXp() : int`, `getWeapon() : Item`, `GetWeaponStats() : ItemWeaponStats`, `GetXp() : int`, `Inventory()`, `isFull() : bool`, `OnLevelUp() : void`, `ReceiveXp() : void`, `RemoveItem() : void`, `ResetInventory() : void`, `UnequipItem() : void`, and `UseItem() : void`.
- Events:** Lists 4 public events: `ClothesChanged : EventHandler`, `ItemListChanged : EventHandler`, `LevelUp : EventHandler`, and `XpChanged : EventHandler`.

definícióját nem helyben tárolom, hanem a PlayerController állítja be a hátizsák példányosításakor. Az ötlet, hogy minden játékos maga dönti el, hogyan használja az adott tárgyat. Még ha jelenleg mindenkinél ugyan az a definíció is szerepel. A C# nyelv delegációs függvényeit a System könyvtárban lévő Action/**Action<T>** változóba lehet eltárolni.

A hátizsák módosulását **eseményekkel** jelzi az osztály a külvilág számára. A PlayerController nézetért felelős részei és a UIInventory osztály is ezekre iratkozik fel, hogy értesüljenek a változásokról.

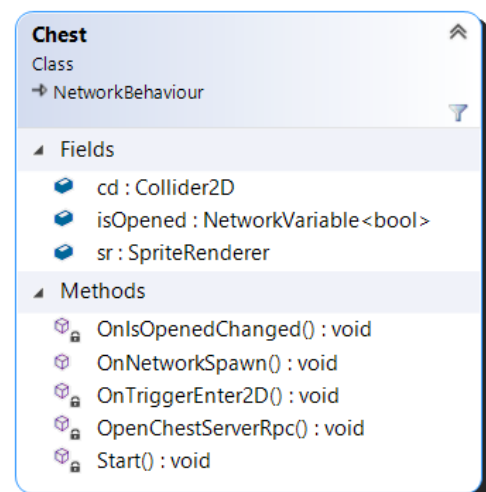
Azt, hogy éppen milyen ruházat és fegyver van egy adott karakternél fontos jelezni a többi kliens felé is, de mivel az Inventory a Unitytől **független** réteget alkot, ezt a feladatot a PlayerController fogja ellátni. Ott hozom létre a játékosnak három NetworkVariable típust, amiben az aktuális ruházatát tároljuk a karaktereknek ItemStructNetVar típussal.

## Chest

A játékba szerettem volna több tartalmat beletenni, ezért készítettem a világba ládákat, melyeket érintésre lehet kinyitni és véletlenszerűen Itemeket ad a játékosoknak. Ezek megvalósításához felhasználok sok fent leírt technikai elemet, ezért egy remek összefoglalása az eddigieknek.

A ládák GameObjectje tartalmaz Collider2D komponenst, hogy észlelni tudjam mikor érintkezik játékoskal. Valamint van egy SpriteRenderere, ami az állapotnak megfelelő nyitott vagy zárt ládának a képét tölti be.

A **Chest** osztálynak van egy NetworkVariable<bool> változója, ami a láda állapotát tárolja a hálózaton. Habár az ütközést minden kliens érzékeli, a NetworkBehaviour IsHost változója segítségével csak akkor kezeljük le, ha a mi állományunkon fut a szerver. Ekkor az megváltoztatja a láda állapotát, létrehoz egy ItemWorldSpawnt és legenerálja a véletlenszerű tárgyakat a világba.



A NetworkVariable típusok egy tulajdonsága, hogy jelzik az összes állomány felé, ha megváltozik az értékük. Ilyenkor továbbítják a korábbi és az új értéket is. A klienseken feliratkozunk erre az eseményre és a meghívásakor a SpriteRenderer képének kicserélésével lekezeljük a láda kinyitását.

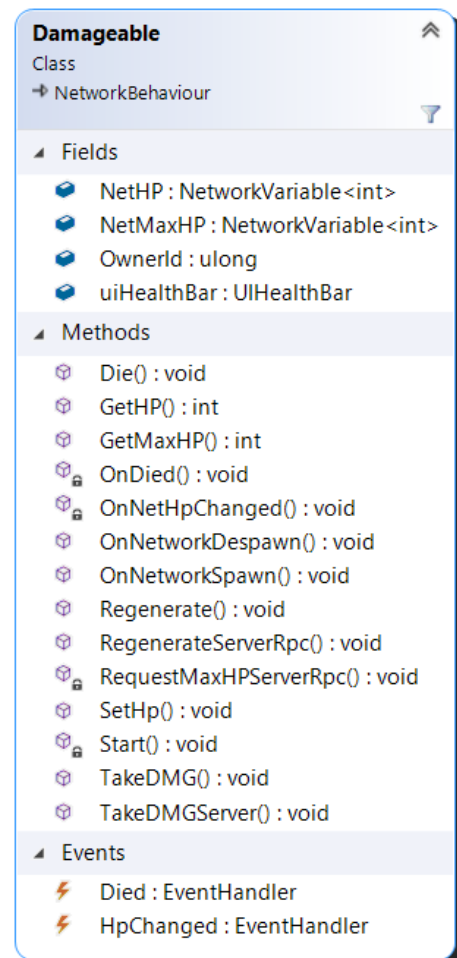
## Damageable

A **Damageable** osztály szerepe, hogy komponensként kerüljön a GameObjectekre és sebezhetőséget adjon nekik. Az osztály szamon tart egy életerő pontot és egy maximum életerőt. Amikor az életerő eléri a nullát egy eseménnyel jelzi a GameManagernek, majd megsemmisül az adott GameObject.

Amikor egy Damageablet eltalál egy lövedék, a szerver a *TakeDMG* metódussal lekezeli a sebzés elszenvedését és módosítja a hálózati változókat. Az objektumok **megsemmisítését** a Unity könyvtár által biztosított *Destroy* végzi el. Amennyiben a szerveren hívjuk meg a metódust egy NetworkObject komponenssel rendelkező objektumon, akkor a Netcode elvégzi a törlést a hálózat minden állományán.

A Damageable osztályt úgy terveztem, hogy bármelyik GameObjectre rá lehessen tenni, azonban egy fellépő probléma volt a NetworkObject felé irányuló függősége. A Unity ilyen esetekre kínál egy megoldást a C#-ban lévő **attribútumok** kibővítésével. A „`[RequireComponent(typeof(NetworkObject))]`” attribútum jelenlétével a Damageable osztály inicializálásakor ellenőrzésre kerül a NetworkObject jelenléte a GameObjecten. És hozzáadja a komponenst amennyiben az hiányzik.

Az olyan NetworkBehaviour osztályoknál ahol alkalmazunk hálózati változókat érdemes az *OnNetworkSpawn* felülírt metódusban elvégezni az inicializálásukat. Ez a metódus kerül



meghívásra amikor egy `NetworkObject` sikeresen spawnol a hálózaton. A `Damageable` osztályban a `Start` metódusban inicializálom a példányt és az `OnNetworkSpawn`-t használom arra, a hálózati változók értékét lekérjem és frissítsem az osztályt aszerint.

A `MonoBehaviour` osztályok nem használják **konstruktort**. Legelőször az `Awake` metódusuk fut le, ami a konstruktor feladatát végzi el. A `Start` is hasonlóan egyszer fut le, viszont mindig az `Awake` után, az első `Update` meghívása előtt. Ezt követően a Unity motorja minden képkocka frissítéskor futtatja le az `Update` metódust.

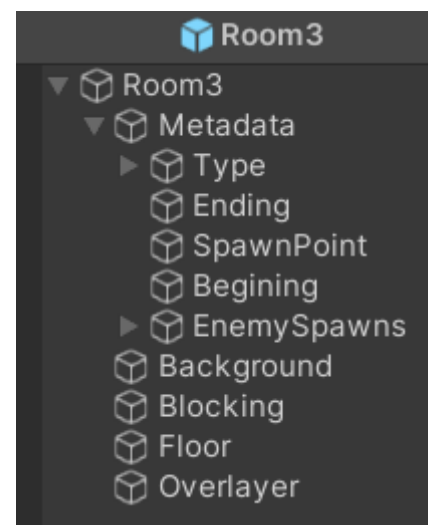
## *Pálya felépítése és `MapInfo` osztály*

A játékban a szobák egymást követve spawnolnak miután az előzőt kitisztítottuk az ellenfelektől, így épül fel a pálya egésze. A szobákat a Unity Editorban rajzoltam, ami nem igényel programozáshoz való tudást. Azonban mégis tartalmaznak olyan elemeket amik ismerete szükséges az ellenfelek vagy a lövedékek logikájának bemutatásához.

A szobák a Unity Tile Map komponensével készültek. Ez egy kétdimenziós **Grid**, melynek kockáit én töltöm fel egy **Tile Palette** alapján. Ez egy kép, ami rengeteg pályaelemet tartalmaz egyetlen stílusban. Én egy fejlesztőknek biztosított ingyenes palettát használok.

A **Tile Map** fontos tulajdonsága, hogy több réteget helyezhetünk egymásra. Minden réteget külön felcímkézhetünk, amik alapján később tájékozódhatunk a scriptekben. Egy másik funkció a rétegek kirajzolási sorrendjének beállíthatósága. Így készíthetünk talajt, ami a karakterünk mögött rajzolódik fel. A falaknak és akadályoknak is készíthetünk egy új réteget „fal” -ként felcímkézve őket. Így már megállapíthatjuk, hogy a karakterünk vagy az eldobott tárgyak mikor ütköznek akadálynak. És készíthetünk a karakterünket eltakaró tetőt is, a `Sorting Layer` megfelelő beállításával.

A szobák tartalmaznak egy metaadatokkal szolgáló `GameObject`-et, ami a szoba egyes pozícióit tárolja el `Transform` komponensekben. Erre szükség lesz később amikor azt szeretném kiszámolni, hogy pontosan hova kell spawnolni az újabb szobát, vagy az ellenfeleket melyik



területre szabad elhelyezni. Az egyszerűség kedvéért készítettem egy **MapInfo** osztályt, ami a szobák metaadataival rendelkező GameObjectet feldolgozza és kinyeri az információt belőlük.

Érdemes megemlíteni, hogy az objektumokat mind fel lehet címkézni. Továbbá a SpriteRenderer és a User Interface elemein is lehet állítani Sorting Layert.

## *Enemy*

Az ellenségek Prefab objektuma egyszerű: A szülőben helyezkedik el a hálózati működéshez szükséges NetworkObject és NetworkTransform, a sebezhetőséget biztosító Damagable script, egy Collider2D az ütközés detektáláshoz és az útkereséshez szükséges **Nav Mesh Agent** komponens, amit szintén a Unity biztosít az AI (Artificial Intelligence) könyvtárában. A szülőn kívül három gyermek GameObjectuma van beágyazva: az egyik a Sprite kirajzolásáért, ami azért van külön, hogy a szülőtől függetlenül tudjon forogni (a mozgás irányába). Egy másik gyermek objektumon az élet (Damageable) kirajzolásáért felelő életerő sáv helyezkedik el. A harmadik gyermek pedig egy üres objektum, amin maga az Enemy osztály helyezkedik el.

Az ellenfelek útkereséséhez Unity szolgáltatást használok. Létrehozok egy **Navigation Mesh** típusú adatstruktúrát, ami információt hordoz magával a pályáról, ezzel az útkeresést segítve. Ezt követően egy Navigation Mesh Agent komponenssel rendelkező GameObjectnek könnyedén kiadhatom az utasítást, hogy jusson el A pontból B-be.

Az ellenfeleket az **Enemy** osztály vezérli. Ebben a scriptben túlnyomó részt szerver oldali logika található, a klienseknek elég mindössze az ellenfél pozíciójáról és irányáról tudni. Egy NPC-nek három állapota létezik: nyugodt állapotban járórként járja a pályát. Amikor egy játékos kerül a hatótávolságába akkor elkezd felé sétálni. Végül mikor elég közel ér a célpontjához megtámadja azt.

Az ellenfelek rendelkeznek egy támadási és egy látási hatótávval. Képkocka frissítésenként ellenőrizzük, hogy vannak játékos címkével ellátott objektumok a közelben. Ehhez a Unity Physics2d könyvtárának az *OverlapCircle* metódusát használom. Amennyiben nincsenek játékosok a közelben, az ellenfél kijelöl egy véletlenszerű pontot a szobában és elkezd felé tartani. Ha látótávban vannak játékosok, akkor kijelöli a legközelebbit és elkezd felé tartani.

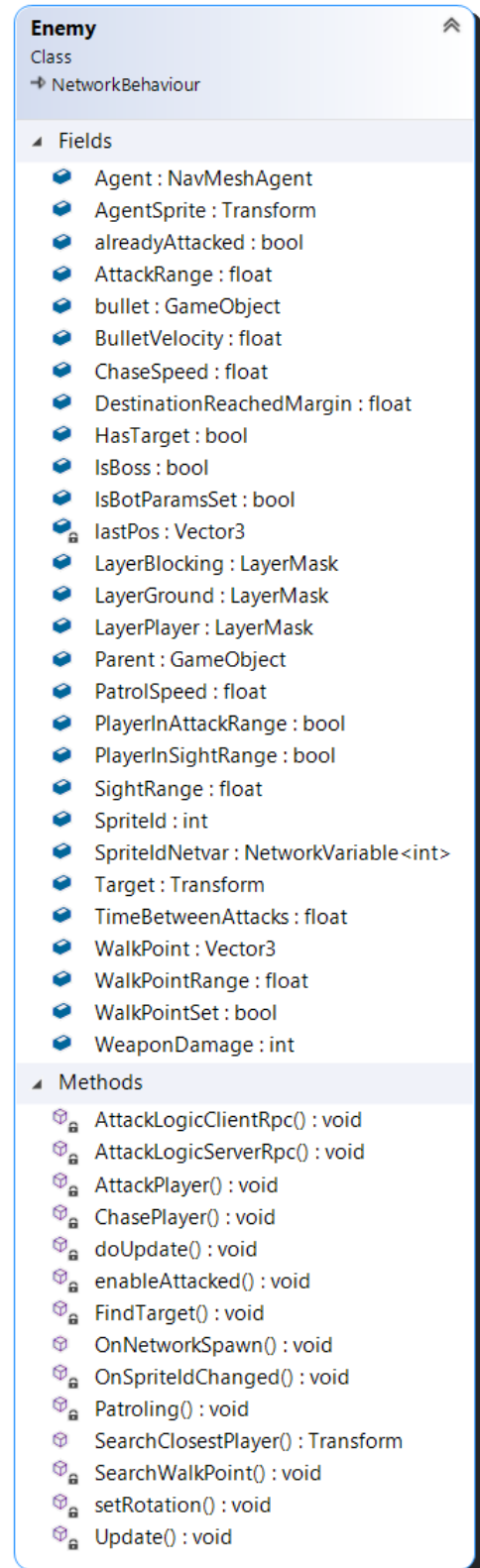


Amennyiben támadási hatótávban van egy játékos, akkor lefut az Enemy támadási logikája. Az *Update* legvégén elmentjük a jelenlegi pozíciót. Ez azért kell, hogy az NPC Sprite objektumát a megfelelő irányba tudjam fordítani. Ehhez egy **irányvektort** kell számolni a két pozíció egymásból való kivonásával.

Amikor őrárat során egy új *walkPoint* kerül kiszámításra ellenőrizni kell, hogy az új pozíció nem a pályán kívül helyezkedik el és azt is, hogy nem egy *blocking* címkével rendelkező objektumon. A pályán *blocking* címkével látom el a falakat és az akadályokat azzal a szándékkal, hogy ne legyenek átjárhatóak. Az egymáson elhelyezkedő rétegeket egy RayCast segítségével ellenőrzöm. Ha az új *walkPoint* nem a földön helyezkedik el, akkor a számítások abbamaradnak az adott *Update* metódusban.

A **RayCast** során egy megadott pozícióból útnak indítunk egy sugarat egy általunk meghatározott irányba. A sugár minden pixelt megvizsgál egy vonalban (háromdimenziós tér esetén) és reagál a térben lévő objektumokra, vagy rétegekre, amiken áthalad. A játék kétdimenziós lapon készül, de játéktér háromdimenziós. Így itt is lehetőség van az alkalmazására a z tengely irányába. Az Enemy osztályban a három *LayerMask* típusú változó nyújt információt a megfelelő címkéjű rétegekről. A RayCast szintén egy Unity által implementált metódus, ami a Physics2D könyvtárban található.

Egy ellenfél erősségét a *WeaponDamage*, *TimeBetweenAttacks*, *ChaseSpeed* és *BulletVelocity* paraméterei adják meg. Az értékek a GameManager által kerülnek beállításra



az ellenfelek spawnolásakor. Mivel az Enemy osztály is főképp szerver oldali logikát tartalmaz, nincs szükség hálózati változókra. A klienseknek nem kell tudniuk ezekről a paramétereikről.

Az egyetlen NetworkVariable az osztályon a Sprite azonosítójáért felelő integer változó. Habár ez egy utólag létrehozott módosítás, de úgy gondoltam élethűnek, ha minden kliens ugyanazt a képet használja fel a hálózaton megegyező ellenfeleknek. Az ItemAssets osztály (ami egy tömbben tárolja az ide illő Spriteokat) minden kliensen azonosan épül fel, így egyfajta **szótárként**, vagy dekóderként is funkcionálhat a hálózaton. A *SpriteIdNetVar* hálózati változón elég a Sprite megfelelő indexét elmenteni, a többi kliens pedig visszafejti a saját ItemAssets példányával.

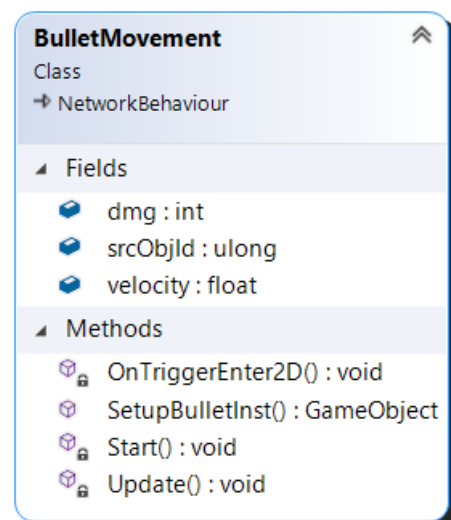
## Bullet Movement

Támadás esetén létrehozunk egy új GameObjectet a jelenetben, ami a lövedéket reprezentálja. Ezen helyezkedik el a **BulletMovement** osztály, ami mozgatja az objektumot, eltárolja a sebzését és gyorsaságát. A lövedék létrehozásához az ehhez készített statikus metódust használom.

Először elhelyezem a világban az előre elkészített Prefabet, ez tartalmaz egy SpriteRenderert a töltény megjelenítéséért és egy Collider2D komponenst az

ütközés észleléséért. Létrehozom a BulletMovement komponenst, felkonfigurálom az értékeit majd beállítom a lövedék irányát. Ehhez egy irányvektort kell használni, amit az egér és a karakter pozíciójából, vagy pedig az ellenfél és karakter egymáshoz viszonyított helyzetéből számolok ki. Unityben az objektumok rendelkeznek a háromdimenziós teret reprezentáló három tengellyel. Ezek értéke függ egymástól, így amikor egy tengelynek új irányt adunk, a többi igazodik hozzá.

A töltény objektumának x tengelyét elforgatom a lövés irányába és a script frissíti a pozícióját minden képkocka frissítéskor. Felmerülő probléma, hogy egy gyorsabb számítógép



többször frissíti a képernyőt és ekkor a töltény is gyorsabban halad előre. Azért, hogy a mozgás egyenletes legyen minden állományon a Unity motor *deltaTime* értékével kell szorozni a mozgás mértékét. A *deltaTime* az előző és a jelenlegi képkocka frissítés között eltelt időt tárolja másodpercekben, lebegőpontos típussal.

Egy másik felmerülő probléma a lövedék szinkronizálása a hálózaton. Ezek az objektumok nagyon gyorsan közlekednek. Gyakran mire sikerült replikálni őket a kliensek állományán már célba értek a szerveren és megsemmisültek. Ezért a távolról kapcsolódó játékosok nem is látták a támadásokat. A megoldásom, hogy nem szinkronizálom őket, hanem minden kliens létrehoz egy saját „hamis” töltényt, ami ütközésig mozog, majd megsemmisül. A Netcode két fajta RPC-t biztosít: **ServerRPC** és **ClientRPC**. Az eddigiekben ServerRPC-t használtam, ahol a kliens értesíti a szervert, hogy az futtasson le egy metódust. Ellenben a ClientRPC esetében a szerver kezdeményez és a metódus a kliensek állományán fut le. Amikor egy támadást kezdeményez az egyik fél, az értesíti a szervert, ami létrehozza a saját BulletMovement példányát, majd a szerver egy ClientRPC-vel létrehozza a hamis töltényeket a kliensek állományán. Magas hálózati késleltetés esetén előfordulhat, hogy egy töltény máshol ér célba a kliens és szerveren, de ez egy sokkal kevésbé zavaró tényező, mint amikor a támadás végbe sem tudott menni a klienseken.

A lövedék ütközés detektálását itt is a Collider2D komponens kezeli. Az OnTriggerEnter2D felülírt metódusban azonban implementálni kell az ütközés logikáját. A metódus paraméterül kap egy *Collider* beépített típust, ami a másik fél Collider2D komponensét reprezentálja. Ebből lekérdezhető, hogy az adott objektum rendelkezik *blocking* címkével, vagy szerepel a *Damageable* komponens a GameObjecten. Ha falba ütközik a töltény akkor egyszerűen töröljük a jelenetből. Ha azonban egy Damageable komponenssel találkozik és a kód a szerveren fut le (amit a NetworkBehaviour IsServer változójával eldönthetünk), akkor meghívjuk a Damageable *TakeDMG* metódusát. Amennyiben az egyik kliensen fut a kód, csak töröljük a töltényt a jelenetből.

Még egy felmerülő problémát jelentett, hogy a támadás pillanatában létrejövő lövedék egyből a támadást indító félbe ütközött. Ezt úgy oldottam meg, hogy a BulletMovement osztályban elmentem a támadó fél karakterének azonosítóját (ami megegyezik a hálózaton)

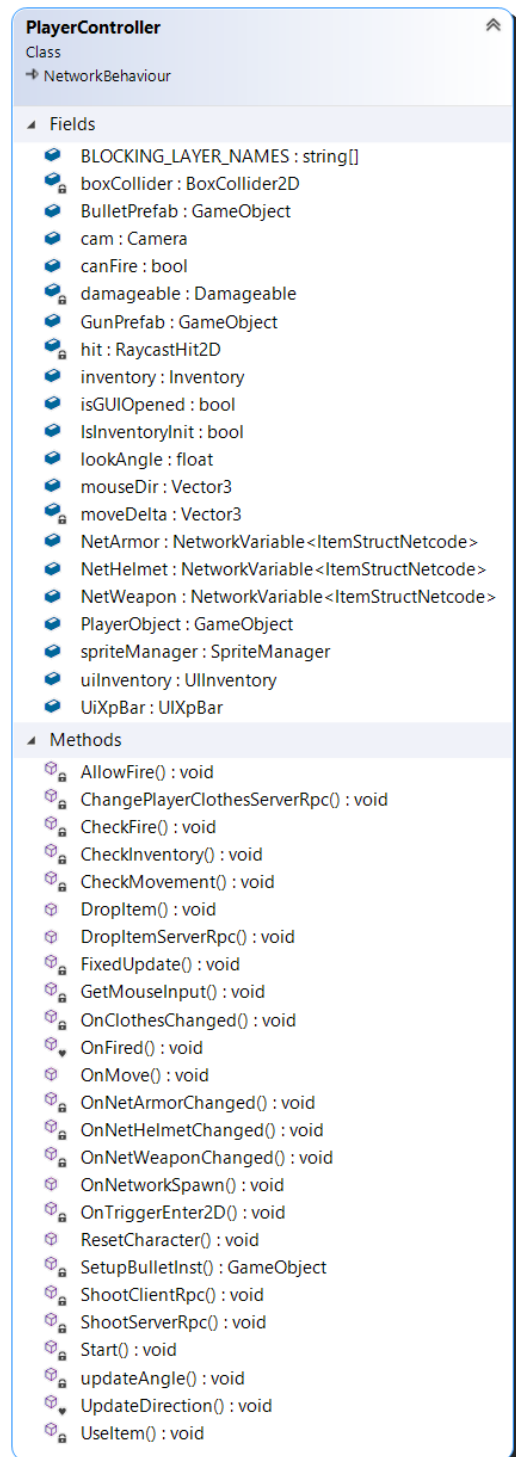
és ütközés esetén külön ellenőrzöm, hogy a két azonosító megegyezik. Amennyiben igen, a támadás logika nem megy végbe és a töltény közlekedik tovább.

## Player Controller

A **PlayerController** osztály a játékosoknak szánt GameObjecteken helyezkedik el. Feladata a kívülről érkező input feldolgozása, az alkomponensek referenciáinak tárolása, a karakter és kamera mozgatása és kommunikáció a további játékost kiszolgáló osztályokkal.

Az osztály örökölt Start metódusát arra használok fel, hogy beállítsam a komponensekre mutató referenciákat, példányosítsam az Inventory osztályt és a karaktert elhelyezzem a GameManager által kijelölt indulópontra

A játékhoz való csatlakozáskor a GameManager hozzáad a jelenethez egy Player Prefabet és átadja a tulajdonjogot is. A játékosok legfeljebb egy karaktert birtokolhatnak, így több játékos esetén is könnyedén el lehet dönteni, hogy melyikük dolgozza fel egy-egy kliens **inputját**. A *FixedUpdate* ismétlődően lefutó metódusban ellenőrzöm a billentyűzeten vagy egeren éppen lenyomva tartott gombokat. Az input leolvasásában a UnityEngine könyvtárba implementált *GetAxisRaw* és *GetButton* függvények segítenek. A *GetAxisRaw* függvény lehetséges értékei a *-1*, *0* és *1*. Ha a kiválasztott függőleges vagy vízszintes tengelyek mentén egy nyilat lenyomunk a billentyűzeten



akkor pozitív irányba 1 a visszatérített érték, negatív irányba -1. Így a játékos mozgatásához lefuttatom a függvényeket a két tengely mentén és a kapott x és y irányba mozgatom a játékost. A mozgás mértékét itt is a Time osztály deltaTime paramétere adja meg, valamint a játékos karakterének egyéni módosítói. A pozíció módosítását megelőzőleg végrehajtok egy *BoxCast* Unity könyvtárba implementált metódust, hogy ellenőrizzem az karakter előtt lévő akadályokat. Ha a metódus végrehajtása során a boxot egy fal vagy másik karakter keresztezi akkor a mozgás művelete megszakad. Az egér lenyomásával az ellenfelek támadásához hasonló műveletek futnak le. A tabulátor lenyomva tartásával pedig a táska felhasználói interfésze nyílik meg. Ennek működését egy későbbi fejezetben részletezem.

A **BoxCast** művelet a RayCasten alapszik. A különbség, hogy itt nem egy vonalat húzunk pixelenként, hanem téglalapokban vizsgáljuk a pixeleket és a többi réteggel való átfedést.

Ideális esetben, amikor a kliens mozgást szeretne végrehajtani, akkor azt egy RPC-vel kéri a szervertől. Ha a mozgás lehetséges, akkor a szerver módosítja a kliens pozícióját és tájékoztatja őt az új pozícióról. Egy egyszerűbb megoldás, ha a pozícióért felelő NetworkVariable írási jogosultságát a szerver helyett a tulajdonos kapja meg. Így a kliens módosíthatja a saját pozícióját és értesíti a többi állományt. Én nem implementálom ezeket a megoldásokat, ehelyett egy előre elkészített **NetworkTransformClient** komponenst használok, ami az utóbb leírt megoldás szerint működik.

A MonoBehaviour három **Update** metódust biztosít: *FixedUpdate*, *Update*, *LateUpdate*. Ezek a leírt sorrendben hívódnak meg egymás után. A FixedUpdate garantáltan lefut és ebbe szokás helyezni a háttérben futó logikát. A LateUpdate fut le utoljára, ebben a metódusban érdemes elhelyezni a nézetbeli változásokat a képernyőn vagy a kamera mozgásáért felelő logikát. Jelen esetben a kamera nem rendelkezik logikával, hanem a Player Prefab gyermek objektumaként együtt mozog a szülővel.

A karakter forgása követi az egér irányát. Ezt egy egyszerű **SpriteRotater** osztály létrehozásával oldottam meg, ami a Sprite szintjén helyezkedik el azzal egy GameObjecten. A PlayerController minden frissítéskor kiszámolja a karaktertől az egér irányába mutató irányvektort és ezt továbbítja a SpriteRotaternek. Ezt követően a Spriteon lévő Transform függőleges y tengelyét teszem egyenlővé az adott vektorral. Azért a y tengelyt, mivel a

karakterek képei úgy lettek rajzolva, hogy alapból felfele néznek, a függőleges tengely pozitív irányába.

A `PlayerController` tartalmaz három `NetworkVariable` osztályt is, amik a karakterre öltött ruhákat és fegyvert reprezentálják. Mivel az `Inventory` egy `Unity`től független osztály, ezért az eseményekkel értesíti a `PlayerController`t, ha változás történt az öltözetben. Változás esetén a kliens kérelmet nyújt a szervernek, ami tartalmazza az új ruhákat (`ItemStructNetcode` típusok) és a szerver ellenőrzés nélkül frissíti ezekkel a hálózati változókat.

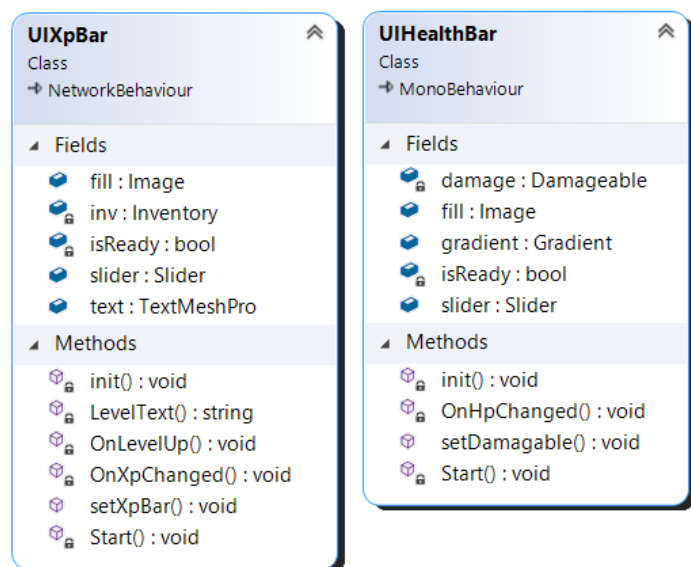
Végül a `SpriteRotater` mellé társul egy `SpriteManager` osztály is, ami figyeli az épp a karakteren lévő ruhákat és eszerint változtatja a karakter kinézetét a képernyőn. A háttérben három egymást elfedő `GameObject` `SpriteRenderer` komponensét kezeli és cseréli ki a képeket.

## Életerőszáv

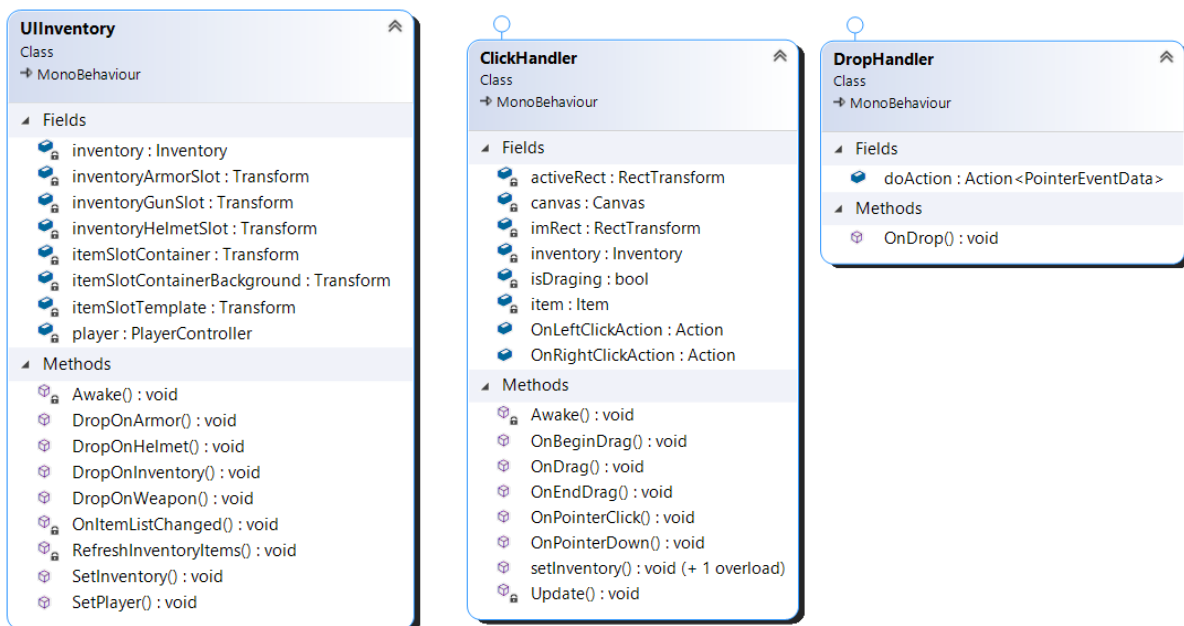
A `Damageable` komponensekhez mindig társul egy életerőpontot felrajzoló grafikus interfész, valamint a játékos karakteréhez egy tapasztalatot mérő sáv. Ezeket az elemeket az `UIHealthBar` és `UIExperienceBar` script valósítja meg.

A működésük igazán egyszerű. Egy `SpriteRenderer`et használok egy téglalap felrajzolásához, aminek a szélességét változtatom amint csökken az életerő vagy nő a tapasztalat pont.

A téglalapot közvetett módon kezelem. Egy **Slider** komponensnek üzenek, ha változik az életerő, az pedig a maximum és a jelenlegi érték alapján kiszámolja a százalékos értéket és beállítja a téglalap méretét. A sáv színe is változik ahogy csökken az életerő. Ezért egy **Gradient** komponens felel, amit színeket ad előre beállított intervallumokon.



## UilInventory



A táska grafikus interfésze a Player Prefab objektumon helyezkedik el. Amikor a PlayerController észleli, hogy a tabulátor gomb le van nyomva a billentyűzeten akkor aktiválja az interfészt.

A Unity kínál beépített **Widgeteket** a UI könyvtárban. Ezeket az elemeket egy Canvasre kell felhelyezni, ami egy kamerához van csatolva. A felbontást a kamera komponensén keresztül lehet beállítani, ami lehet egyéni, de jelen esetben a felbontás megegyezik az ablakéval.

A Canvesen két üres konténerem van. Az egyikben a karakterre felhelyezhető itemek tárolói helyezkednek el. A másik konténer egy **Grid** komponenst tartalmaz, ami a hátizsákot reprezentálja és szintén item tárolókat fog tartalmazni.

Ezek a tárolók úgy működnek, mint a gombok. A **ClickHandler** osztály felel a működésükért, ami megvalósítja az *IPointerClickHandler* interfészt. Ezáltal képes érzékelni és jelezni, ha az egér egy gombjával kattintunk rá. Az osztály tartalmaz két delegációs metódust, melyeknek értékét példányosításkor kapják meg a UilInventory osztálytól. Kattintáskor a gombtól függően

az egyik metódust hívom meg. Jelenleg bal kattintásra a játékos eldobja az adott itemet, jobb kattintásra pedig felhasználja azt.

A kényelmes használatért egy **Drag and Drop** rendszert is készítettem, így a ruhákat kattintás helyett fel lehet húzni a megfelelő tárolóba és fordítva. Ehhez további UnityEngine által nyújtott interfészeket alkalmaztam, hogy értesüljek az egér lenyomásáról és felengedéséről, valamint a lenyomás közben pozícióról. Ezek az interfészek sorban: *IPointerDownHandler*, *IBeginDragHandler*, *IEndDragHandler*, *IDragHandler*. Létrehoztam egy külön **DropHandler** osztályt, ami érzékeli, ha egy elemet ráhúznak a komponensére majd ott elengedik.

A felületet úgy frissítem, hogy újra felrajzolom a felület összes elemét, ez előzőket pedig törölöm. Ezt a *UIInventory* osztály *RefreshInventoryItems* metódusában implementálom.

## *ItemAssets*

Az *ItemAssets* egy **Singleton** osztály ami példányosul a program indításakor, és bárholnan hozzá lehet férni a statikus *Instance* változóján keresztül. Az osztály a játékban felhasznált Spriteokat és Prefabeket tárolja publikus változóiban.

A *MonoBehaviour* szülőosztály lehetővé teszi, hogy a komponensként jelenlévő osztály publikus változóinak a Unity Editorban is értéket adhassunk. Ami még különlegessé teszi, hogy futási idő közben is lehet állítani az értékeit. Ezt a funkciót akkor használtam ki, amikor az ellenségek paramétereinek kerestem az optimális értékeket. Az *ItemAssets* osztálynál nem módosítok futási időben az értékeken, de az Editorban kötöm össze a változóit a megfelelő képekkel és Prefab objektumokkal.

## *Game Manager*

## *Connection Manager*

## *UI Manager*



*Játék tesztelése*

*Játék bővíthetősége*

## Hivatkozások

## Köszönetnyilvánítás