# A projekt struktúrája

Mivel egy hálózati játék fejlesztésénél elengedhetetlen, hogy legyenek megosztott és nem megosztott kód részletek, valamivel bonyolultabb projekt struktúrára van szükség. A következő szekció ezt hivatott bemutatni.

## GameLogic

Ez a projekt tartalmazza az osztott játék logikai kódot. Mind a NeonTDS (ami igazából GameClient is lehetne) mind a GameServer referenciaként tartalmazza.

A projekt felelősségei többek között:

* Entitás menedzsment
  + Létrehozás
  + Frissítés
  + Törlés
* Ütközés detektálás
* Power up logika
* Player input lekezelése

### Platform kompatibilitási megfontolások

A projekt egy .NET standard 2.0-s projekt mivel a GameServer linuxon is futtatható, ezért .NET core kompatibilisnek kellett lennie a logikának.

Pontosan ebből kifolyólag nem is találunk a grafikai motort támogató könyvtárra semmiféle dependenciát (Win2D). Ezért kellett kissé bonyolultan elszeparálni az entitást és a kirajzolt entitást.

## NeonTDS

Ennek a projektnek a felelőssége a játék motor (Game Engine) megvalósítása. Itt valósul meg az imént említett Drawable – Entity szeparáció is.

Tartalmazza ezen kívül még a kliens specifikus hálózati kódot.

A project egy UWP alkalmazás, hogy tudjuk használni a Win2D könyvtárat a grafika megvalósítására. Emiatt .NET framework-ot és nem core-t vagy standardet használ, mint a többi projekt.

## GameServer

A GameServer egy .NET core console application, hogy tudjon futni linux alatt (mivel egy Debian VPS ált a rendelkezésünkre a teszteléshez).

A GameServer használja a GameLogic és Networking Shared Library-ket, de azon kívül igen letisztult. Csak a GameClient-el egészíti ki a Networking-et. Illetve megvalósítja a szerver oldali Game loop-ot, amiben összeszedi és csomagolja az üzeneteket, amiket a kliensnek küld.

## Networking

A Networking szintén egy .NET standard Shared Library, amit a kliens és a szerveroldali kód is használ. Ez valósítja meg a hálózati architektúrát, amiről bővebben a *Szerver-kliens kommunikáció megvalósítása* című fejezetben lesz szó. Nagy vonalakban egy egyszerű és egy multiplexelt UDP kliens valamit az üzenet típusok és a szerializációjuk tartozik ide.  
**<Ide érdemes rakni egy packages UML diagramot>**

# Éles teszteléshez használt környezet

Mivel ez a játék nehezen tesztelhető lokálisan (két UWP appot nem lehet egyszerre debuggolni vagy csak nagyon körülményesen), ezért érdemes volt felállítani egy remote szervert.

A remote szerver egy Debian VPS, de legjobb tudomásom szerint bármilyen linux disztribúción működnie kéne (ahol a .Net core supportált).

Ahhoz, hogy a játék is tudja, hol a szerver, egy config.json-t használ, amely így néz ki:

{

"ServerIP": "134.209.232.177", // Ezek a defaultok is[[1]](#footnote-1)

"ServerPort": 32132

}

A szerver szintén tartalmaz egy szinte teljesen ugyanilyen JSON fájlt, csak az IP nélkül.

## Build szerver

A kényelmes fejlesztés jegyében a VPS-en fut egy Build szerver, amely kiválasztott branchek-hez futó szerver instance-okat rendel. Emelett github-os webhook-okkal képes detektálni a pusholást és azonnal lepullolja és újrabuildeli az adott instancet. Egyszerre continous integration és process daemon.

A build server Node.js (express) technológiával készült, mert itt nem volt megkötve a kezünk és a C# túlbonyolult lenne ehhez. Magát a szevert egy PM2 nevű node-os process menedzser futtatja a VPS-en belül.

**<Ide érdemes rakni egy deploymentes UML diagramot>**

# A játék logika

## Játékszabályok

A játék alapvetően egyszerű. A játékosok egy limitált (de egész nagy) pályán mozoghatnak. A játékosok, ha egymásnak mennek mind a ketten meghalnak.

A játékosok lelőhetik egymást ezzel sebezve a másikat. Minden játékosnak van életereje, ami kiegészülhet egy pajzserővel is ha pajzs power up-ot vesznek fel.

A pályán folyamatosan spawnolnak power up-ok, amiket fel véve valami extrára lehet szert tenni.

Power up-ok:

* Rapid fire (sárga színű) – gyorsabb lövés pár másodpercig
* Shield (sötétkék színű) – pajzserő az életerőhöz (gyakorlatilag második független életerő, ami prioritást élvez)
* Sniper (világoskék színű) – azonnal (shielden keresztül is) ölő és mindenen áthaladó lövedék, amiből csak egy darabot kap a felvevő játékos

Ha valamely játkos (igazából bármely entitás) a játék széléhez ér megöli a lézer azonnal.

## Kliens és szerveroldali logika elszeparálása

Annak jegyében, hogy minél egyszerűbben megosztható legyen a GameLogic kódja és a kliens, illetve szerver specifikus részeket egy helyen lehessen implementálni az EntityManager tartalmaz egy IsServerSide flag-et. Ez alapvetően nem a legjobb megoldás, hiszen így mind a két oldal tartalmaz halott kódot, amit sosem használ. Jobban belegondolva rájöhetünk, hogy egy definiált compile time változó lett volna a jó ötlet, de ez csak későn jutott eszünkbe.

## A játékszabályok implementálása

A legtöbb játékszabályt a Player entitás implementálja, innen tudjuk meg mi történik, ha a Player más entitásokkal ütközik. Ez az osztály implementálja a respawn logikát is.

A kliens nem sebezheti és respawn-olhatja a Player-t így ezek a logikák IsServerSide=true esetén érvényesülnek csak.

## Entitás menedzsment

Az entitások menedzseléséért az EntityManager felelős, amely a létrehozást, törlést és frissítést végzi.

Az EntityManager egy .Net-es Dictionary (HashMap)-ben tárolja az entitásokat az azonosítójuk alapján.

### Entitás azonosító

Az entitások azonosítója egy egyszerű 32 bites unsigned integer, amit a hálózati architektúra is használ az azonosításhoz. Ehhez pluszba egy egyedi rendszert alkalmaztunk a csak kliens oldali entitások ID-ja a legfelső bitet 1-re állítja míg minden nem csak kliensoldali entitás legfelső bitje 0. Erre végső soron nem volt feltétlen szükség.

### Entitás létrehozás és törlés

Mivel egy foreach loop-on belül nem lehet törölni, erre bevett szokás, hogy egy ideiglenes listába pakoljuk a létrehozandó és törlendő entitásokat, amit a végén alkalmazunk a tényleges entitás listára.

Ez a két ideiglenes lista egy-egy HashSet-ként lett implementálva, mivel a sorrend mindegy és ezzel könnyen kiküszöbölhető ugyanannak az entitásnak a többszöri hozzáadása.

### **<Ide kéne egy entity create és vagy destroy szekvencia diagram>**

### Update függvény

A már említett entitás lista kezelésen túl, ebben a függvényben frissülnek maguk az entitások (meghívjuk az Update függvényüket).

Ez a függvény kezeli le az „entitás gyilkos” pályahatárokat, amelyre két okból van szükség. Egyrészt nem akartunk végtelen pályát, szóval egyfajta gameplay oka is van a dolognak. Másrészt fontos, hogy a lövedékeink soha az életben nem tűnnek el hacsak nem mondjuk nekik, ezért felelős még a pályahatár, hogy ha érintkeznek vele az entitások, azonnal törlődjenek.

Ezen kívül ide került még a Power up spawn-olással kapcsolatos logika is, aminek igazából nincsen jó oka.

### Entitás események

Az entitások az EntityManager-től kapnak információt arról, amikor létrejönnek vagy törlődnek ezért felelős az OnCreate és az OnDestroy virtuális függvénypár, amit kedvünkre felüldefiniálhatunk az entitásokban. az entitásokban. Ezen kívül az entitások tartalmaznak egy CollidesWith függvényt, amelyet az érintett ütközés logika hív meg (performancia megfontolások miatt nem csinálunk automatikus ütközés detektálást mindig minden entitásra).

### EntityManager események

Az EntityManager események feladata kettős. Egyrészt a kliens feltud iratkozni az egyes eseményekre ilyen olyan effektek létrehozása céljából. Másrészt a szerver oldal feltud iratkozni, hogy üzeneteket tudjon belőle csinálni. Gyakorlatilag ez a híd a szerver oldali üzenet gyártás és a GameLogic kód között.

Az események rendes C# event-ek, néhányukhoz tartozik egy Invoke<Event> függvény is amit maguk az entitások hívnak meg.

Használható események:

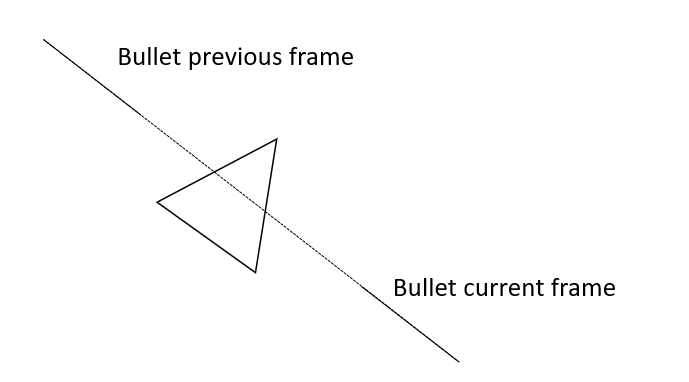
* EntityCreated
* EntityDestroyed
* PlayerRespawned
* PlayerActivePowerUpChanged
* PlayerHealthChanged

**<Ide kéne egy letisztultabb UML class diagram a EntityManager és Entity esetleg Player ha kifér csak>**

## Ütközés detektálás

Az algoritmus szakaszok metszésén alapul. Player (vagy jövőben más zárt körvonalú entitások) tesztelésére a TestClosedShapes algoritmus használható, amely minden szakaszt minden szakasszal megpróbál ütköztetni. Ez bár nem a legeffektívebb megoldás, a gyakorlatban egész szépen működik.

A Bullet – Player ütközést máshogyan kell detektálni hiszen a Bullet igen nagy sebességű lehet, tehát simán átugorhatja a Player-t.



1. ábra A lövedék elkéne, hogy találja a játékost

Ezért egyfajta ray tracelésre van szükség, erre való a TestBulletHit függvény.

## Játékos bemenetének kezelése

A játékos (felhasználó) alapvetően a következő féleképpen képes hatni a játékra:

* W – Sebesség növelése (bizonyos maximumig)
* S – Sebesség csökkentése (bizonyos minimumig)
* A – Fordulás balra
* D – Fordulás jobbra
* Egér pozíciója felé néz az ágyú
* Egér balgomb lenyomva tartása – tüzelési állapot igaz (elengedés esetén hamis)

Ezeket az inputokat alapvetően a szerver kezeli le, de lokálisan is megtörténik (a gyakorlatban ez nem igazán látszik).

A Player entitást indirekten tudjuk szabályozni. A mozgást az adott irányba vett egyenes vonalú egyenletes mozgás valósítja meg, ami minden Entitás esetében az Update függvény alapértelmezett logikája (mivel a legtöbb entitás kihasználja).

A lövedékek az ágyú irányába fognak kilőni, ezt a lövedék spawn-olást a szerver utasítására végezheti csak a kliens, vagyis alapvetően ez is IsServerSide=true logika.

## Játék logikai esetek

**<Ide kéne egy két filler UML szekvencia diagram a szeveroldali logikáról, Entity eseményekkel főleg> Példák:**

* **Power up felvétele**
* **Bullet hit**
* **Player – Player ütkezés**
* **Entity – pályaszél ütkezés**

# A játék motor

## Sprite generálás

## Renderelés

# Szerver-kliens kommunikáció megvalósítása

## Múltbéli próbálkozások

## Byte alapú üzenet formátum

## Kimeneti MessageQueue

## Bemeneti MessageQueue és a listener

## Üzenet csomagolás

# Dead reckoning használata az effektíven statikus entitásoknál

A játék alapvetően 3 fajta entitásból áll Networking szempontból:

* Player
* Bullet
* Power up

A 3 entitás különböző jellegű üzenetekkel frissül. Egy Bullet esetén semmi értelme nincsen újra és újra lekérni a szerver-t a pozícióról. Csak a létrehozás és törlési üzenetek fontosak, ezért a szerver nem is küld a pozícióról üzenetet. A kliens tudja úgynevezetten prediktálni (jósolni), hol lesz a lövedék a következő frameben. Amikor azt feltételezzük, hogy egy lövedék nem mozog sehova csak amerre fele kilőtték az ún. dead reckoning elvet alkalmazzuk. Ez az elv alkalmazott gyakorlatilag a Power up-ok szempontjából is, hiszen azok pont a másik véglet, nem mozognak.

A Player esetében nem ilyen egyszerű a helyzet így nem is tudunk igazából semmi jól működöt csinálni bonyolultabb algoritmusok nélkül.

Egy speciális eset, hogy a Player entitás ágyújának az irányát mindig „elhihetjük lokálisan”, hiszen az csak a vizualitást fogja befolyásolni és úgymond real time frissül, a szerver nem befolyásolja.

# Lag kompenzációról

A játék hálózati architektúrájánál minden erőfeszítés ellenére nem sikerült értelmes lag kompenzációt összehozni. Így a végső megoldás alapvetően a szerver oldal válasza után lépteti csak az inputot érvénybe. Így bár a kliens oldalon létezik kliens oldali predikcióra lehetőség az nincs kihasználva a Player-ek esetében.

Ennek a gyakorlati következménye, hogy a ping-ünknek (latency) megfelelő input lag-ot tapasztal a játékos. Ez egy limitációja a kurrens implementációnak.

## Hogyan próbáltuk ezt a problémát megoldani?

Mivel ez egy igen nagy probléma és megtöri a játékélményt akár már 50-60 ping esetén is, ezért más megoldásra lenne szükség. Egy lehetséges megoldás, hogy a lokális Player-t a jelenben tartjuk, vagyis „elfogadjuk” az inputját feltételezve, hogy a szerver is ugyanúgy fogja feldolgozni, ami elvben igaz is kis eltéréssel (csomag vesztés, csomag átrendeződés vagy egyszerűen időzítés béli okokból).

A szerver által leküldött lokális Player-t tehát frissíteni kellene, ha egy bizonyos küszöböt átlép a hiba pozícióban, forgásban vagy sebességben. Ehhez tudnunk kellene mi volt a változás amióta a szever feldolgozta az inputot. Ehhez az inputokra egy Sequence number-t (sorszámot) rakunk, amivel azonosítani tudjuk az inputjainkat. A kliens elmenti a lokális Player státuszát a bizonyos sequence number-ekhez és ezt veti össze a szerver által visszaküldött Player státusszal, amely visszaküldi a legutóbb feldolgozott sequence number-t.

Idáig el is jutottunk, ekkor az ötlet az lenne, hogy kivonjuk a kliens mentett pozíciót és hozzáadjuk a szerver által visszaküldöttet. Elvben ekkor a delta ugyanúgy maradna és a játékos egy kisebb „ugrást” venne észre, de semmi több.

Azonban ez a megoldás nem sikerült. Ez a fajta korrekciós algoritmus divergens és túlkorrigál. Valószínű oka ennek az, hogy a szerver és a kliens még sincsenek annyira közel időzítésben. Ennek a problémának megoldásához azonban a fejlesztő csapat kevésnek bizonyult.

1. Tudom, hogy JSON-ban nincsen comment. [↑](#footnote-ref-1)