
# ALKALMAZÁS FEJLESZTÉSE XAMARIN PLATFORMON

Fenyvesi Péter, G7CQ2D

Konzulens:

Albert István

# TARTALOM

# Bevezetés

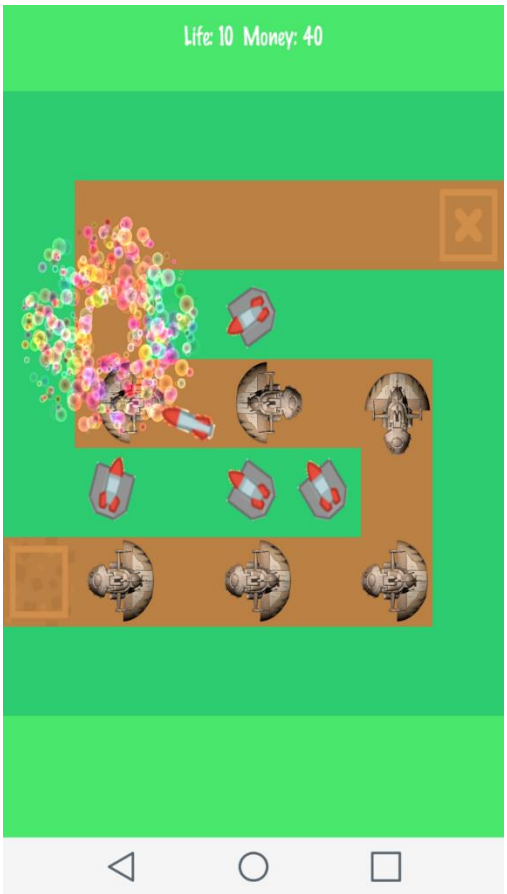
## Motiváció

Ha egy cég egy mobil alkalmazást szeretne kiadni, annak érdekében, hogy versenyképes maradjon, ki kell adnia az alkalmazását a legnagyobb mobil operációs rendszerekre. Ha csak két platformra szeretné kiadni, akkor is látható, hogy kétszeres munkát kell elvégezni, hiszen teljesen más környezetben kell a külön platformra megírt alkalmazásokat megvalósítani. Ezt az akadályt próbálják áthidalni a cross-platform technológiák, mint például a Microsoft tulajdonában álló Xamarin.

Személyes motivációm a témához, hogy érdekelnek a mobiltelefonos technológiák és úgy gondolom, hogy nagy jövője van a cross-platform fejlesztésnek. Mivel a téma nem specifikálta, milyen alkalmazást kell elkészíteni, így úgy gondoltam, hogy egy játékot szeretnék megalkotni, mivel izgalmasabbnak tartom, és nagyobb sikerélmény egy saját játékkal játszani.

## Specifikáció

Az alkalmazás maga egy Tower Defense. A játék egy kétdimenziós játéktéren zajlik. A játékos célja megakadályozni, hogy a pályán keresztül menjenek a tankok, amik végtelenítve jönnek, amíg a játékosnak el nem fogynak az életei. Utána lehetősége van új játékot kezdeni.




A játékos úgy állíthatja meg a tankokat, hogy a pálya azon részeire, ahol a tankok nem mehetnek, tornyokat vehetnek, amik meghatározott időközönként lövik a tankokat. Ha egy tankot kilőtt, akkor abból pénz kap a felhasználó, amit további tornyokra költhet. Ha egy tank beér a célba, akkor fogy az élete.

## JÁTÉKFEJLESZTÉS

Ahogyan egy játék fejlesztése izgalmasabb is lehet, mint egy hagyományos alkalmazásé, úgy igényel egy kicsit más gondolkozást is. A legszembetűnőbb különbség az, hogy a hagyományos alkalmazásokat főként felhasználói események. Egy felhasználói viselkedésre válaszol valahogy az alkalmazás.

Ezzel szemben egy játék egy végtelenített ciklusban megy, amíg véget nem ér. Egy ilyen ciklus alatt sok minden történik.

- Bemenet olvasása
- Játékobjektumok mozgatása felhasználói input, vagy mesterséges intelligencia alapján
- Ütközés detektálása
- Játékállapot változásának ellenőrzése
- Renderelés

Egy ilyen ciklust másodpercenként többször is meg kell tennie a játéknak, attól függően, hogy milyen felhasználói élményt szeretnénk elérni, illetve milyen hardveren fut a játék. Többek között ezért is sokkal erőforrásigényesebb egy játék, így, mivel a telefon erőforrásai végesek, erre is figyelni kell.

Ezeken felül a rengeteg matematikai számítás miatt is erőforrásigényesebb egy játék. Sokkal több matematika és fizika fordul elő, mint hagyományos esetben. Egy háromdimenziós játéknál minden erre épül, de egy kétdimenziós esetében sem lehet elkerülni.

A Tower Defense fejlesztése során is kellett egyszerű matematikát használnom, mint két pont kötötti távolság mérése, vagy egy vektor irányával és szögével számolni, megvizsgálni, hogy két objektum ütközik-e, illetve a mozgásnál a sebesség egyszerű képletének, a v = s / t-nek a használata volt nélkülözhetetlen. Egy játéktér általában derékszögű koordináta rendszerben épül fel, így a lineáris algebra, a vektorok használata elkerülhetetlen.

Ezek után egy játékfejlesztő csapat is egészen máshogy néz ki, ez hagyományos fejlesztői csapattal szemben. A fejlesztők ugyanúgy az alkalmazás logikájáért felelnek, hogy az specifikációnak megfelelően működjön. Mellette viszont egy grafikusnak sokkal nagyobb szerepe van, mert egy egész világot kell megteremtenie, minden játék béli elemet, képet, objektumot, hiszen a játékélményhez nagyon sokat ad a grafika, legyen az kétdimenziós vagy háromdimenziós.

Ezen felül a játékélmény tökéletesre csiszolásáért felel a játéktervező, aki megtervezi a játék menetét, a sztorit, azt hogy ne csak helyen és szépen működjön, hanem minél érdekesebb legyen. Illetve a játékélményhez a további effektek és hang bejátszások is hozzáadnak, amik menedzselése megint egy másik feladatkör.

## JÁTÉKFEJLESZTÉS XAMARINBAN

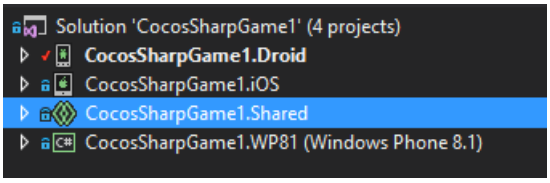
Tehát adott a kérdés, hogyan lehet ezeket megvalósítani Xamarinban. Szerencsére számos technológia is létezik, amiket lehet használni, ha valaki játékot szeretne fejleszteni Xamarinban, mint például CocosSharp, MonoGame, vagy UrthoSharp.

Az UrthoSharp egy főként háromdimenziós játékmotor. Én a játékomban én CocosSharp-ot használtam, mivel az egy kétdimenziós játékokhoz fejlesztett játék engine, így egyszerűen volt implementálható a saját játékom benne. Ezekkel szemben a MonoGame nem egy game engine, csak egy nyílt forráskódú játékfejlesztő API. Ez annyit tesz, hogy a MonoGame használatával közvetlenül kell kezelni a játékobjektumokat, és manuálisan kell kirajzolni őket, és közös

objektumokat is kell implementálni, mint a kamera, vagy a fa struktúrát ami az objektumok kirajzolásánál fontos. A különbséget akkor érthetjük meg könnyedén, ha figyelembe vesszük, hogy a CocosSharp is MonoGame-re épül.

## CocosSharp

A projekt felépítése hasonló a hagyományos Xamarin felépítéséhez. A solutionben található projektek mindegyike az adott platformnak felel meg, ezen felül van egy közös projekt, ahol a közös kódok futnak

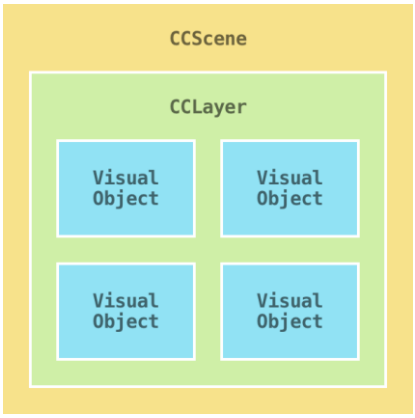



Minden projektben található egy *CCApplication*, ami maga az alkalmazásnak felel meg. Ismeri a készüléket, és annak paramétereit, így létrehozza és beállítja az grafikus alkalmazást, létrehoz egy application delegate-et és elindítja a játékot.

CCApplicationDelegate az alkalmazás életciklusáért felel. Az alkalmazás szintű eseményeket kezeli, mint a következőek:

```
public override void ApplicationDidFinishLaunching(CCApplication application, CCWindow mainWindow)
public override void ApplicationDidEnterBackground(CCApplication application)
public override void ApplicationWillEnterForeground(CCApplication application)
```

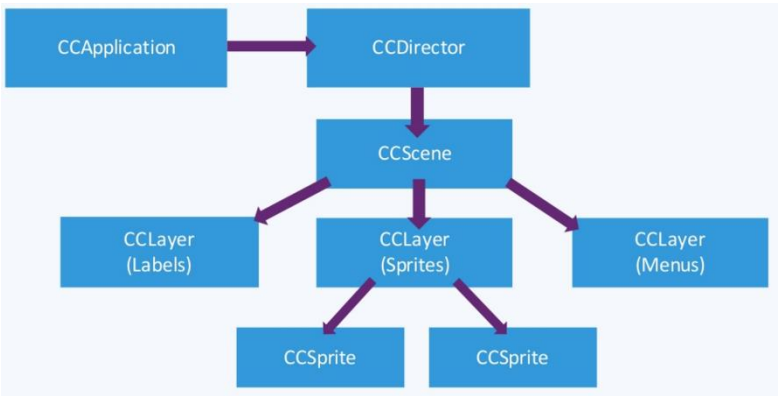
Akkor hívódnak meg, ha az alkalmazás elindult, háttérbe került, vagy előtérbe. Az alkalmazás elindításánál állíthatjuk be a Content foldert, és innen indítja el az első jelenetet. A Content folder-ben tartálhatóak a statikus erőforrások, mint a betűtípusok, képek, hangok, animációk.




Az alkalmazásban a jelenetek váltásáért a *CCDirector* felel, ez navigál közöttük. Egy CCScene a játék egy jelenete, ez tartalmaz CCLayereket, amik tartalmazzák az összes látható objektumot.

Az alkalmazás látható objektumai egy fába rendezhetően írhatók le. Ennek érdekében a CCScene, CCLayer és minden UI elem a CCNode-ból származik le, ami a fának egy csomópontját jelenti. A fa gyökere a CCScene. Annak gyerekei a rétegek, amin bármennyi és bármilyen elem szerepelhet.

Így CCNode-ból származik a kép megjelenítéséért felelős CCSprite, illetve a szövegért felelős CCLabel, de ezen felül van menü, animáció, vagy épp primitív objektumok, mint sokszögek, matematikai alakzatok.

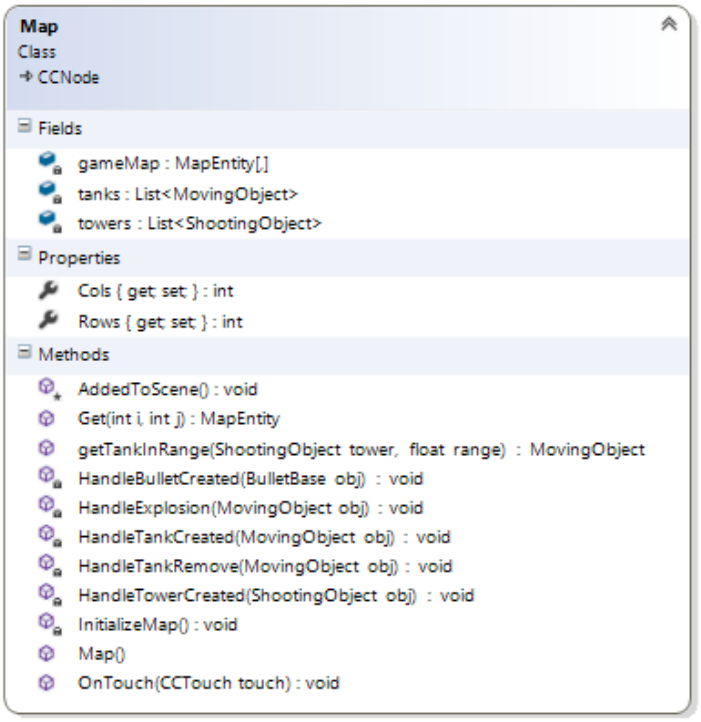


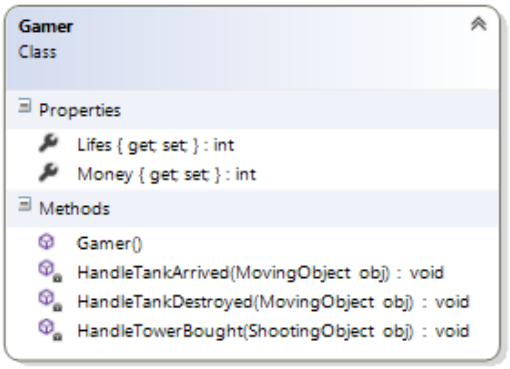

Az érintést a CCTouch osztály kezeli, a layer-en kell az eseménykezelő metódusokat felülírni, mint a TouchesBegan, TouchesMoved, TouhesEnded. A metódusok az összes érintést kapják meg egy listában, így lehet kezelni a multitouch-ot is.

# ALKALMAZÁS ARCHITEKTÚRÁJA

Az alkalmazás fejlesztése közben próbáltam a hagyományos Model-View-Controller architektúrát követni, így szétválasztani a nézetet, kontrollert és modellt megfelelő rétegekbe.

## MODEL




Maga a modell, két fő osztályból épül fel, van egy *Map*, ami a pályáért felel, illetve egy *Gamer* osztályból, ami a játékos maga.

### Map

Felelőssége a pálya felépítése és a pálya elemeinek, tankjainak és tornyainak menedzselése. A *CCNode* osztályból származik, hogy a View ki tudja rajzolni és fa struktúrába tudja szervezni.

Attribútumai és property-jei:

```
        public int Cols { get; private set; }
        public int Rows { get; private set; }

        private MapEntity[,] gameMap;
        private List<MovingObject> tanks;
        private List<ShootingObject> towers;
```

*gameMap,* MapEntity-k kétdimenziós tömbje, maga a pálya.

*tanks,*Tank ősosztályból képzett lista, tárolja a pályán lévő az összes tankot.

*towers,* Torony ősosztályból képzett lista, tárolja a pályán lévő az összes tornyot.

A property-k a pálya méretét adják meg.

Metódusai: Az osztálynak három különböző felelősségű metódusa van.

- Inicializáló metódusok: Konstruktor, ahol az eseményekre iratkozik fel, illetve AddedToScene és InitializeMap, amik a pálya felépítéséért felelnek. Az inicializálás során egy szöveges fájlból olvassa a játék a pályát.

```
public Map()
protected override void AddedToScene()
private void InitializeMap()
```

A következő eseményekre iratkozik fel:

```
GameEventHandler.Self.TankCreated += HandleTankCreated;
GameEventHandler.Self.TankDead += HandleTankRemove;
GameEventHandler.Self.TankDead += HandleExplosion;
GameEventHandler.Self.TankArrived += HandleTankRemove;
GameEventHandler.Self.TowerCreated += HandleTowerCreated;
GameEventHandler.Self.BulletCreated += HandleBulletCreated;
```

- Eseménykezelő metódusok: A játék alatt történt minden változás és felhasználói interakció eseményt vált ki a játékban, ami eseményekre a konstruktorban az osztály fel is iratkozott. Ezek a metódusok kezelik, az adott esemény milyen hatást vált ki a modellben.

```
private void HandleTankCreated(MovingObject obj)
private void HandleTankRemove(MovingObject obj)
private void HandleExplosion(MovingObject obj)
private void HandleTowerCreated(ShootingObject obj)
private void HandleBulletCreated(BulletBase obj)
```

- Getter metódusok: A *Get* pálya egy adott koordinátájú elemét adja vissza, a *getTankInRange* egy adott torony adott távolságában lévő tankok közül visszaad egyet.

```
public MapEntity Get(int i, int j)
public MovingObject getTankInRange(ShootingObject tower, float range)
```

### Gamer

Felelőssége a játékos adatainak nyilvántartása, mint az élete és pénze, és ha elfogy az élete, akkor szóljon a játéknak, hogy vége a játéknak.

Property-k:

```
public int Lifes { get; private set; }
public int Money { get; private set; }
```

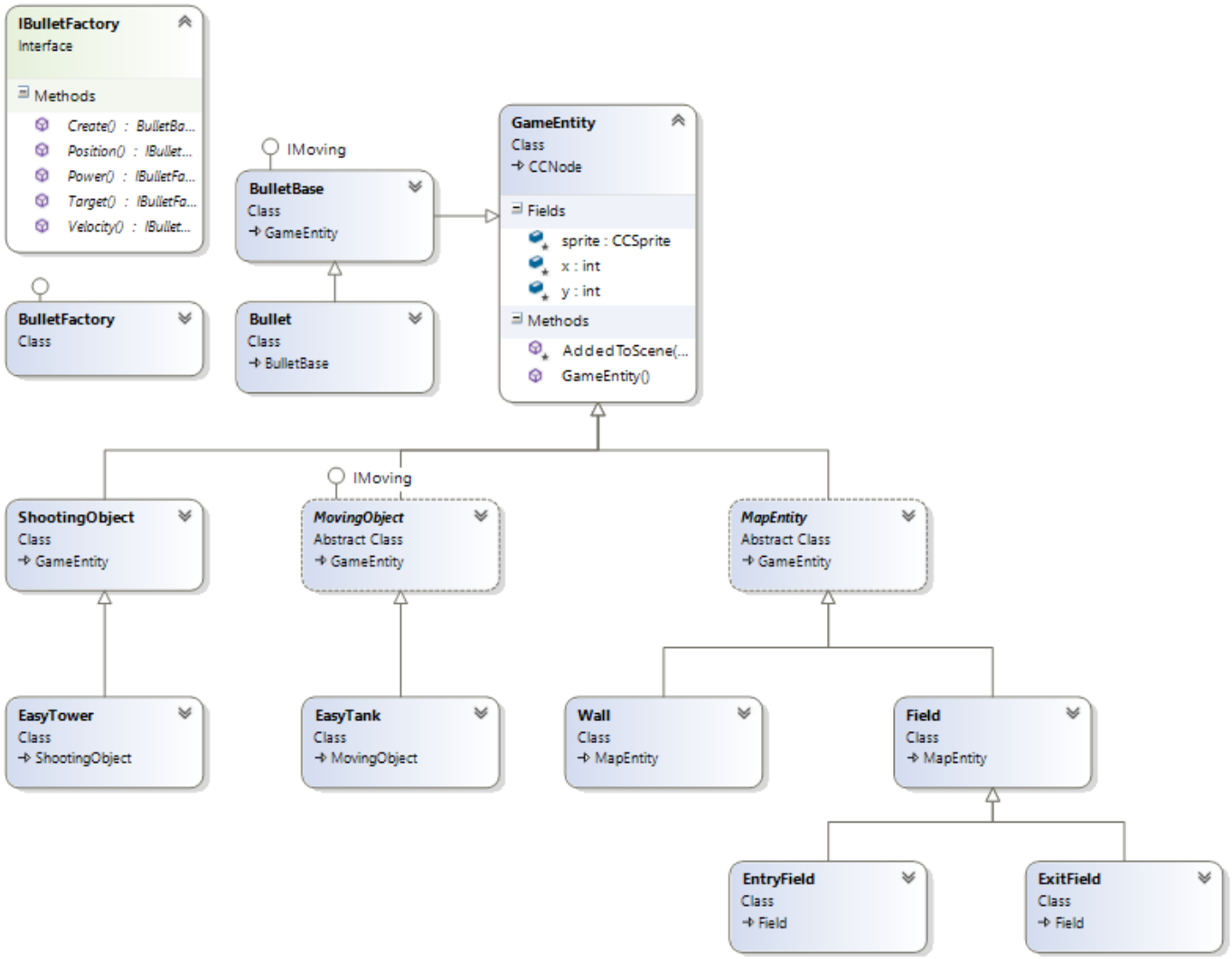
Metódusok:

```
public Gamer()
private void HandleTankDestroyed(MovingObject obj)
private void HandleTankArrived(MovingObject obj)
private void HandleTowerBought(ShootingObject obj)
```

A konstruktorban feliratkozik az alábbi eseményekre eseményekre:

```
GameEventHandler.Self.TankArrived += HandleTankArrived;
```

```
            GameEventHandler.Self.TankDead += HandleTankDestroyed;
            GameEventHandler.Self.TowerCreated += HandleTowerBought;
```




A térképen lévő összes játékelem a ***GameEntity*** absztrakt ősosztályból származik le. Ez az ősosztály a CocosSharp *CCNode* osztályából származik le, hogy könnyedén fa struktúrába lehessen helyezni az elemeket a UI-hoz.

Attribútumai:

```
:         protected CCSprite sprite;
          protected int x, y;
```

Az x, y a térképen való elhelyezkedésért, a sprite a képért felel.

Konstruktorán kívül, amiben beállítja a koordinátáját, csak egy metódust definiáltam felül az ősosztályából:

```
        protected override void AddedToScene()
```

Ebben a metódusban állítom be a sprite méretét és helyét. és adom hozzá a fához. Ezt azért kellett így csinálni, mert a sprite minden leszármazottban különböző, ezért minden leszármazott a saját képét tölti bele, így az ősosztály konstruktorában az értéke még *null* marad, viszont ez a metódus csak az után hívódik meg, hogy létrejött az objektum.

Négy különböző elem származik le a *GameEntity*-ből, ez a négy különböző osztály a játékban előforduló négy különböző feladatkört látja el:

**MapEntity**:

Magáért a pályáért felel. Egy metódussal egészíti ki az ősét:

```
public abstract bool acceptTank(MovingObject tank);
```

Ez igazzal tér vissza, ha tank ráléphet és hamissal, ha nem.

Két leszármazottja van:

- ***Field:*** Tankot enged menni, tornyot nem, különleges fajtája az első és utolsó mező, az első mező készíti az új tankokat, az utolsó fogadja őket.
- ***Wall:*** Tornyot enged rakni magára, tankot viszont nem.

  Feliratkozik az érintés eseményre, így ha

```
public void OnTouch(CCTouch touch)
```

  Így nem kell minden egyes játékelemre vizsgálni, hogy megérintette-e, és ott lekezelni az érintést, hanem csak azoknak az osztályoknak feliratkozni, amik érdekeltek ebben.

**MovingObject*:***

A tankok ősosztálya. Implementálja az IMoving interfészt, ami a mozgásért felel. Ebből az osztályból leszármazhatnak különböző tankok.

Attribútumai a következők:

```
private Map map;
protected int health;
protected float velocity;
```

Referenciát tárol a pályáról, hogy a mozgásnál le tudja kérni a következő mezőt, illetve van egy élete és egy sebessége az objektumnak.

Metódusai:

```
public void move(float t)
public void getShot(int power)
```

A move függvényt az *IMoving* interfészből implementálja, mozgatja a tankot.

A getShot függvény sebezi a tankot, ha meghal, akkor értesíti a kontrollert.

**ShootingObject*:***

A tornyok ősosztálya. Ebből az osztályból származhatnak le a különböző tornyok. Felelőssége, hogy adott időközönkét rakétát hozzon létre, amivel a tankokat lövi.

Attribútumai:

```
        protected int power;
        protected float range;
        private MovingObject target;
        private Map map;
```

Tudja hogy milyen erős és milyen messzire lát el. Eltárolja a pálya referenciáját is, mert onnan tudja melyik tankot vegye célba.

```
        public ShootingObject(int x, int y, Map map)
        private void rotation(float unused)
        private void shoot(float unused)
```

Konstruktorában feliratkoztatja a másik két metódusát, hogy folyamatosan meghívódjanak. A rotation mindig lekér a pályától egy tankot, amit lát, és afelé forgatja. A shoot három másodpercenként hívódik meg és létrehoz egy rakétát, ha van láthatáron belül tank.

#### BulletBase

A rakéták ősosztálya, sebezi a célpontnak megadott tankot.

Attribútumai:

```
        private float velocity;
        private int power;
        private MovingObject target;
```

Van sebessége, ereje és iránya.

```
        public void move(float t)
```

Implementálja az *IMovin* interfészt, így megvalósítja annak metódusát, a move-ot. Folyamatosan mozgatja a célpont felé a rakétát adott sebességgel, ha eléri azt, akkor sebezi.

### BulletFactory

Rakétákat *Builder pattern* segítségével hozom létre

```
var newBullet = (new BulletFactory()).Position(position)
                                     .Target(target)
                                     .Power(power)
                                     .Velocity(v)
                                     .Create();
```

Minden rakéta létrehozásánál példányosítom, mivel az összes toronynak saját életciklusa van, saját szálon mozognak, mindig akkor példányosodik a *BulletFactory,* amikor egy torony meghívja, így ezzel biztosítva a szálbiztosságot.

## Controller

A ***GameEventHandler*** osztály kezeli a játékban történő eseményeket. Az osztály követi a *singleton pattern* tervezési mintát, ezzel biztosítva, hogy a játékban biztosan csak egy példány szerepel, és így nem hívódik meg többször ugyanaz az esemény egyszerre.

Minden a játékban történő esemény, legyen az a felhasználó által generált, vagy a játék által generált, ezen az osztályon fut keresztül. Ez eventeket generál belőlük, amikre utána bármelyik osztály feliratkozhat, és lekezelheti.

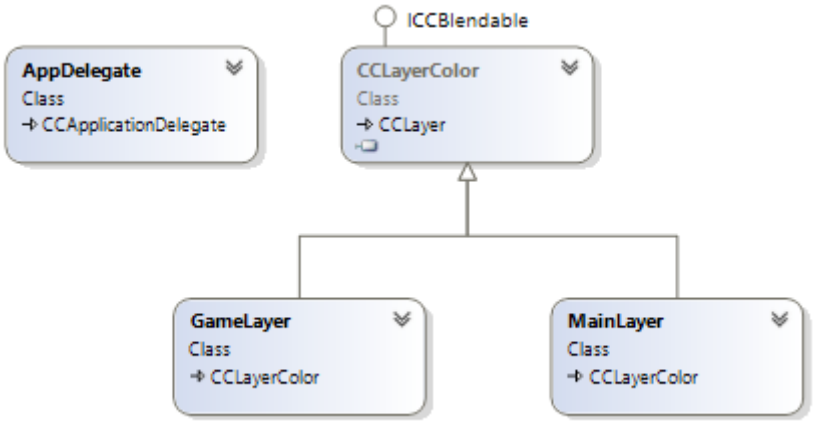
```
public event Action<MovingObject> TankCreated;
public event Action<MovingObject> TankArrived;
public event Action<MovingObject> TankDead;
public event Action<ShootingObject> TowerCreated;
public event Action<BulletBase> BulletCreated;
public event Action GameIsOver;
public event Action<CCTouch> TouchScreen
```

Ezeket az eseményeket akkor süti el, ha egy másik osztály értesíti, hogy bekövetkezett az adott event. Például ha a felhasználó megérinti a kijelzőt, azt a *GameLayer* látja csak eleinte, de ez továbbadja a *GameEventHandler*-nek ami eseményt generál belőle. Másik irányba ugyanúgy működik, amikor nem a View váltja ki az eseményt, hanem a modell, például, ha egy tank beér a célba, a kontroller elsüti a *TankArrived* eseményt. Erre fel van iratkozva a programomban a játékos, hogy csökkenjen az élete, illetve a View, hogy frissítse a kiírást, hány élete maradt a játékosnak.

A következő metódusokkal lehet kiváltani az eseményeket a kontrollerben:

```
public void CreateTank(int x, int y, Direction dir, Map map) // Új tank
public void TankOnExit(MovingObject tank)                    //Tank beér a célba
public void TankDestroy(MovingObject tank)                   //Tank felrobban
public void CreateTower(int x, int y, Map map)               //Új torony
public void isTouched(CCTouch touch, Map map)                //Felhasználó hozzáért a kijelzőhöz
public void CreateBullet(CCPoint position, MovingObject target, int power) //Új rakéta
public void GameOver()                                       //Vége a játéknak
```

## View




Az alkalmazásomnak két nézete van, van egy főoldal, ahonnan új játékot kezdhetünk, illetve maga a játék nézete. A játék nézet betöltésénél építi fel az egész pályát a játék. Az elemeket fa struktúrába rendezve tölti be, a layer-re tölti rá a labelt, ami kiírja a játékos életeinek számát és pénzét, illetve a *map*-et ami gyereke ként tölti be az összes térkép elemet. Amint betöltődött elkezdi a gameloopot. Amint a játék véget ér, visszalép a fő menübe.

# JÁTÉK LOGIKA

Mivel minden játékelem a *CCNode* osztályból származik, így könnyen megvalósítható, hogy minden objektum a saját funkciójának megfelelő logikával működhessen. A *CCNode* osztálynak van egy *Schedule* metódusa, ami minden játékciklusban meghívódik.

**MovingObject logika:**

Mozgást valósít meg, egyszer egyenes vonalú egyenletes mozgás. Mozgatás után ellenőrzi, hogy egyenesen haladva ütközik-e fallal, ha igen, fordul.

**ShootingObject logika:**

Folyamatosan ellenőrzi, hogy van-e hatósugarában tank. Ezt távolságméréssel teszi, ha van akkor kiválaszt egyet, és az arra mutató irányvektor irányába forog. A játékengine API-jában itt található egy bosszantú inkonzisztencia. A *CCNode*-nak property-ben meg lehet mondani, mekkora szöggel legyen elforgatva, illetve egy vektornak is van egy property-je, ami megmondja, mekkora a szöge. Viszont a *CCNode* forgásszöge fokban, míg a vektor szöge radiánban van megadva és az egyik a másikhoz képest 90°-al el van fordítva.

Ezen felül másik felelőssége, hogy bizonyos időközönként új rakétát hoz létre. Ehhez a Schedule metódusnak második paraméterként lehet megadni egy float értéket, ami azt mondja meg mekkora időközönként hívja meg.

**Rakéta logika:**

Mozog, minden pillanatban ellenőrzi, hogy van a célpontja, és felé forgatja magát, illetve az irányvektor irányába halad meghatározott sebességgel. Mozgása végén ellenőrzi, hogy ütközött-e a tankkal, ha igen akkor sebezi, és elpusztítja önmagát.

# MEGVALÓSÍTÁS

## FELBONTÁS

A keretrendszer kezeli az aktuális felbontást, mivel tudja, hogy mekkora az adott készülék. Így csak egy kívánt felbontást kell megadni neki, és azt skálázza az adott készülékre. A játékom csak álló helyzetet támogat, kívánt felbontást 768*1024-re állítottam.

Ezen felül a pálya eleminek mérete is dinamikusan változik, attól függően, hogy mekkora a pálya dimenziói. Ez alapján osztja le az egyes elemeknek a méretét.

## VIZUÁLIS EFFEKTEK

A CocosSharp keretrendszerrel egyszerűen lehet hozzáadni effekteket a játékhoz. A vizuális effektekre a *CCParticle* osztályból lehet leszármazni, és felüldefiniálni a metódusait, hogy a kívánt animációt lejátsza. Az osztály a CCNode osztályból származik, így a fa struktúrához is könnyedén hozzá lehet adni.

A saját alkalmazásomban én nem hoztam létre ilyen új effektet, mivel a keretrendszernek vannak beépítettek is, én onnan használom a *CCParticleExplosion* osztályt. Ez egy egyszerű robbanást csinál a konstruktorában megadott helyzetben. Miután végzett az animáció eltünteti magát, és leválasztja magát a fáról.

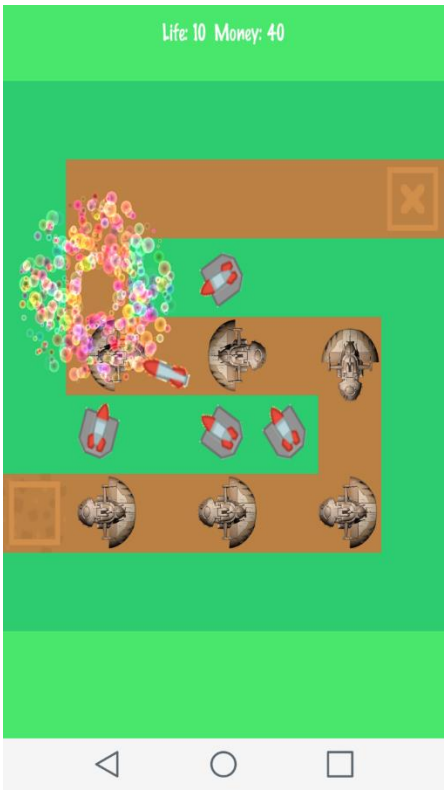
## Hang effektek

Hang lejátszásához a CCMusicPlayer osztályt lehet használni. Az osztály az eddigi osztályokkal szemben nem a CocosSharp névtérben van, hanem a CocosDenshion, ami egy egyszer audio motor CocosSharp-hoz. Miután példányosítottam a játék rétegén, megnyitottam a hangfájlt és elindítottam.

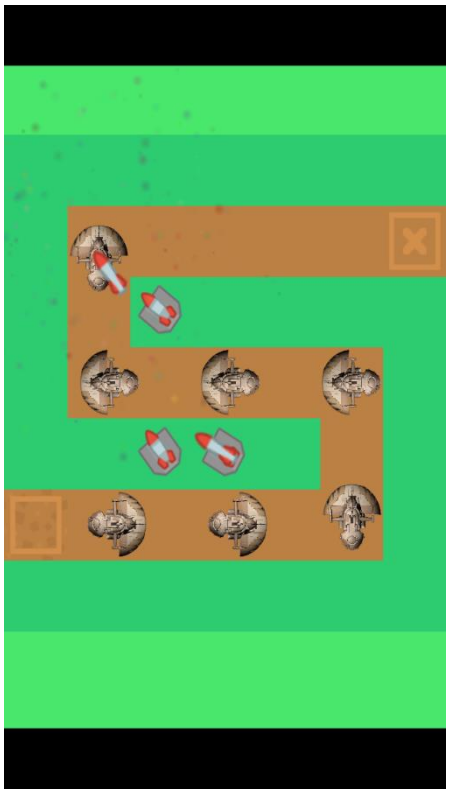
```
CCMusicPlayer backgroungMusic = new CCMusicPlayer();
backgroungMusic.Open("droidsong.mp3", 0);
backgroungMusic.Play(true);
backgroungMusic.Stop();
```

Az osztály Open metódusával lehet a Content folderben található hangfájlt megnyitni, és a Play metódussal elindítani. A play metódus paraméterként kapott bool mondja meg, hogy végtelenítve legyen-e a zene. Ezen felül, amikor a játékos meghal, akkor a Stop metódussal állítom le a zenét.

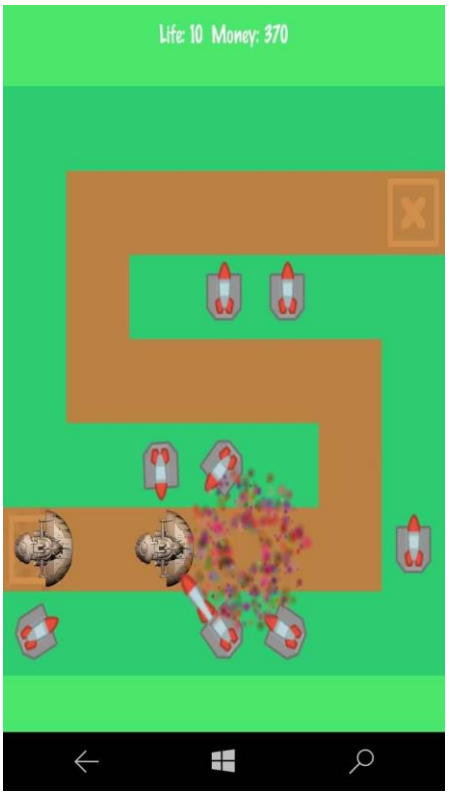
## Cross-platform játék




**Android**



**iOS**




**WP 8.1**

Fejlesztés során saját Android készüléken teszteltem, csak amikor elkészült, akkor teszteltem másik platformokon is.

A Windows Phone-ra fordítva egy hiba keletkezett csak futásidőben. Az audió fájlt nem tudja megnyitni a *CCMusicPlayer*, nem találja a megadott hangfájlt, annak ellenére, hogy az a helyén van. Sajnos a készülék, amire fordítottam, csak kevés ideig volt nálam, így nem tudom, sikerült-e a hibát debugolni.

iOS-re sem sikerült hiba nélkül fordítani elsőre. A játék nem találta meg a betűtípust a Content mappában, így a label-t nem tudja kiírni. Ezt nem sikerült javítani. Ezen felül a szöveges fájlokat máshogy kezeli, így a pálya beolvasásánál hibát dobott. Ezt az alábbi, egyszerű kódrészlettel javítottam:

```
            string mapText = CCFileUtils.GetFileData(filename: "map.txt");
#if __IOS__
            string[] stringSeparators = new string[] { "\n" };
#else
            string[] stringSeparators = new string[] { "\r\n" };
#endif
            string[] lines = mapText.Split(stringSeparators, StringSplitOptions.None);
```

A képeken is látható, hogy a felbontást is máshogy kezelik az eszközök, az Android és Windows Phone skálázza teljes képernyőre, viszont az iOS a kívánt felbontás arányát megtartva skálázza fel a méreteket.

## Összefoglalás

Úgy gondolom, hogy a Xamarin egy remek eszköz, amivel cross-platform alkalmazásokat lehet könnyedén fejleszteni, viszont lehet, hogy nem játékfejlesztésre leghatékonyabb, hiszen vannak technológiák, amik direkt erre a területre céloznak, így nagyobb sikereket érhetnek el, mint például a Unity.

Ennek ellenére nekem egy pozitív élmény volt a projekt, és ha időm engedi, szeretném komolyabbra megcsinálni a játékot, hogy nagyobb játékélmény legyen, illetve tovább fejleszteni, hiszen jelenleg nem nagy kihívás. Az architektúra miatt könnyen bővíthető további tankokkal, tornyokkal és pályákkal, illetve egy többjátékos megoldás is kihívás lehet a játékkal kapcsolatban. Abban viszont biztos vagyok, hogy fogok a jövőben foglalkozni cross-platform technológiával.