**Tartalomjegyzék**

[Bevezetés 3](#_Toc375807777)

[1. Az XNA projekt felépítése 4](#_Toc375807778)

[1.2. SpriteBatch 5](#_Toc375807779)

[1.3. LoadContent() 5](#_Toc375807780)

[1.4. Initialize() 5](#_Toc375807781)

[1.5. Draw() 5](#_Toc375807782)

[1.6. Update() 5](#_Toc375807783)

[1.7. UnloadContent() 5](#_Toc375807784)

[2. A 2D játékok alapjai 7](#_Toc375807785)

[2.1. Képek importálása 7](#_Toc375807786)

[2. 2. Rajzolás és animálása sprite-okkal 7](#_Toc375807787)

[2.2.1. A spriteBatch.Draw() metódus 8](#_Toc375807788)

[2.2.2. Animáció 8](#_Toc375807789)

[2.3. Grafikus objektumok ütközéseinek kezelése 9](#_Toc375807790)

[2.3.1. Rectangle collision detection 9](#_Toc375807791)

[2.3.2. Pixel collision detection 10](#_Toc375807792)

[3. Felhasználói bemenetek kezelése 11](#_Toc375807793)

[4. XNA menü készítése 13](#_Toc375807794)

[4.1. Menü készítése XNA-val 13](#_Toc375807795)

[4.2. Menü készítése XNA és Silverlight összevonásával 13](#_Toc375807796)

[4.2.1. Silverlight 14](#_Toc375807797)

[4.2.2. XAML 14](#_Toc375807798)

[5. Hangok hozzáadása a játékhoz 15](#_Toc375807799)

[5.1. XACT audió programozása 15](#_Toc375807800)

[6. Multiplayer játék fejlesztése 16](#_Toc375807801)

[6.1. Peer-to-peer hálózatok 16](#_Toc375807802)

[6.2. Kliens-szerver hálózatok 17](#_Toc375807803)

[6. Adatkezelés Windows Phone 7.5 Mangon 19](#_Toc375807804)

[6.1. Isolated Storage 19](#_Toc375807805)

[6.1.1. Settings 19](#_Toc375807806)

[6.1.2. Files and Folders 19](#_Toc375807807)

[6.1.3. Relation Data 19](#_Toc375807808)

[7. A játék részletes specifikációja 21](#_Toc375807809)

[7.1. Egyszemélyes játék 21](#_Toc375807810)

[7.2. Játék ellenféllel 22](#_Toc375807811)

[8. A játék tervezése 24](#_Toc375807812)

[8.1. Menü komponens tervezése 25](#_Toc375807813)

[8.2. Háttér animáció tervezése 27](#_Toc375807814)

[8.3. A játékos osztály tervezése 29](#_Toc375807815)

[8.4. Pálya kezeléséért felelős osztály tervezése 30](#_Toc375807816)

[8.5. Ütközés kezelő komponens tervezése 33](#_Toc375807817)

[8.6. Egyéb kisegítő komponensek 35](#_Toc375807818)

[8.7. Két személyes játék tervezése 37](#_Toc375807819)

[8.8. Összefoglalás 40](#_Toc375807820)

[9. A játék implementálása 41](#_Toc375807821)

[9.1. Kiemelt megvalósítások 41](#_Toc375807822)

[9.1.1. Pálya feldolgozás és kirajzolás 41](#_Toc375807823)

[9.1.2. Repülő mozgásának megvalósítása 43](#_Toc375807824)

[9.1.3. Gombkezelő komponens implementációja 49](#_Toc375807825)

[9.1.4. Hálózaton keresztüli játék implementációja 50](#_Toc375807826)

[10. A játék tesztelése 56](#_Toc375807827)

[11. Továbbfejlesztési lehetőségek 60](#_Toc375807828)

[Irodalomjegyzék 62](#_Toc375807829)

Bevezetés

Az XNA Framework a Microsoft által megálmodott keretrendszer melyet játékfejlesztésre alkottak meg (nem keverendő játékmotorral). A rendszer .NET alapú és nem csak Windows Phone fejlesztésre alkalmazható, hanem tudunk vele készíteni asztali alkalmazásokat és Xbox játékokat is. Az ok, amiért ezt a rendszert kialakították, hogy a hobbifejlesztők is egyszerűen képesek legyenek kisebb vagy nagyobb komplexitású játékokat írni. A fejlesztő nagy hangsúlyt fektethet a játék logikájának kidolgozására, az optimalizációt és a játék működésének az alapjait biztosítja az XNA keretrendszer. Az ára, hogy a Microsoft adott egy ilyen jól átlátható, egyszerű és átgondolt keretrendszert az, hogy nem a legfrissebb natív grafikus platformra van optimalizálva. Windows Phone játékok fejlesztése során is az XNA Framework került választásra. Két elterjedt megoldás létezik asztali számítógépes játékok fejlesztésére, ez az OpenGL és a DirectX. Ez a két API túlságosan komplex és magas programozói tudást igényel ezért a Microsoft maradt a meglévő XNA keretrendszer mellett. [1, 3]

**XNA keretrendszer helye a .NET struktúrában**



0.1. ábra – XNA rendszer helye a .NET keretrendszerben [1]

1. Az XNA projekt felépítése

Ebben a fejezetben az a program működéséhez tartozó funkciókat mutatom be, melyeket a program előír. Az 1.1. ábrán jól látszik, hogy az inicializálások után a játék belép egy végtelen ciklusba, ami addig tart, amíg ki nem lépünk az alkalmazásból. Tehát a játékfejlesztés nem eseményvezérelt programozáson alapul, hanem azt vizsgáljuk meg mindig frissítéseknél, hogy történt e valamilyen felhasználói interakció. Eseményvezérelt fejlesztésnél, ha keletkezik, valamilyen felhasználói bemenet akkor az létrehoz egy eseményt melyen végrehajtódik egy metódus.

















* 1. *ábra – Az XNA játékciklus blokkdiagramja [3]*

1.1. GraphicsDeviceManager

Minden XNA projekt tartalmaz egy objektumot a fent említett osztályból. Ezzel a példánnyal tudjuk beállítani és kezelni a játék bizonyos beállításait (például: a játék felbontását, ami Windows Phone 7 esetén mindig 480\*800 pixel, a játékos profilbeállításait stb).

# *1.2. SpriteBatch*

Ennek az objektumnak a metódusaival tudjuk kirajzolni a képernyőre a textúrákat illetve karakterláncokat (kétdimenziós objektumokat). Ez a példány a főosztályban deklarálódik és a LoadContent() metódusban hoz létre belőle egy példányt.

# *1.3. LoadContent()*

Ennek a metódusnak a segítségével tudunk betölteni a játékba képet, hangot vagy betűtípust a content pipeline segítségével. Fontos, hogy csak a betöltés után használhatóak a tartalmak.

# *1.4. Initialize()*

Ez a metódus egyszer fut le, amikor a játék elindul. Itt érdemes beállítani az olyan tulajdonságait a játéknak, amit nem fogunk megváltoztatni a futás alatt, illetve a szükséges kezdeti értékeket.

# 1.5. Draw()

Ez a metódus felelős a rajzolásért. Először törli a képernyőt, utána beállítja a képernyő színét majd kirajzolja a grafikus elemeket a megfelelő sorrendben.

# 1.6. Update()

Ebben a metódusban kezeljük le a felhasználó által előidézett interakciókat. Itt végezzük el a különböző objektumok tulajdonságainak a módosítását, amivel reagálunk a felhasználói bemenetekre, vagy frissítjük az automatikus folyamatokat.

# 1.7. UnloadContent()

Ez a metódus szabadítja fel a memóriát, ha a játék futása befejeződött. Itt kell végrehajtani az olyan műveleteket, melyeket a játék kilépése előtt szeretnénk lefuttatni.

Ezek az előre legenerált metódusok nagymértékben segítik a játékfejlesztést. Jól értelmezhető kódot ad mivel jól elválnak egymástól a játék felépítéséhez szükséges metódusok. Természetesen csak akkor igaz az előző állításom, ha ezeket mind megfelelően használjuk, mert a kirajzolásért felelő metódusban ugyanúgy tehetünk üzleti logikát, mint a frissítés metódusban és az inicializálásért felelő metódusba illeszthetünk betöltést is. Érdemes betartani ajánlott szétválasztásokat. [3]

2. A 2D játékok alapjai

A kétdimenziós játékfejlesztés tulajdonképpen képek egymás utáni kirajzolása ezért véleményem szerint olyan játékot kell alkotni, amelyben nem csak a grafikai megvalósítások fogják meg a felhasználót, hanem a játék logikája is. Lehetséges grafikailag szép játékot készíteni 2 dimenzióban is különböző féle térhatást keltő transzformációkkal, például elmosással (a távolabb vagy közelebb levő objektumokat homályosítjuk különböző mértékben, attól függően, hogy mi milyen távolságra van).

2D objektumok kirajzolására egy kétdimenziós koordináta-rendszer áll rendelkezésünkre, mely X és Y tengelyekből áll. Windows Phone 7.5 Mango operációs rendszer esetén a tengelyek mérete a következőképen alakul:

x

480 pixel

0

y

800 pixel

*2.1. ábra – Windows Phone 7 koordináta-rendszere*

# 2.1. Képek importálása

Játékfejlesztés során sok képet használunk fel ezért az, fontos hogy milyen formátumú és nagyságú képeket importálunk. XNA Framework a következő típusokat fogadja el: bmp, dds, hdr, jpg, pfm, png, ppm, tga. A méretét érdemes jól átgondolni, mert ha túl nagy a fájl mérete, akkor sok helyet foglal a memóriában illetve több ideig tart a feldolgozása, a túlságosan alacsony felbontású kép pedig rombolja a felhasználói élményt.

# 2. 2. Rajzolás és animálása sprite-okkal

Erre a célra a következő osztályokat használjuk:

* GraphicsDeviceManager
* SpriteBatch
* Texture2D

Képet először hozzá kell adni a projekthez (ezt érdemes a Content mappába másolni) majd be kell importálni a forráskódba a content pipeline-on keresztül a LoadContent() metóduson belül. Betöltés a fent bemutatott „Texture2D” osztály egyik objektumával történik. Kirajzolásért a Draw() metódus felel, amit a „SpriteBatch” osztály egy objektumán keresztül érünk el. Ennek az eljárásnak többféle felülírt változata van, melyekkel tudunk objektumokat transzformálni, forgatni, színezni, nyújtani, elhelyezni X és Y koordináták megadása segítségével.

## 2.2.1. A spriteBatch.Draw() metódus

Fontosnak tartom a fent említett eljárás paramétereinek részletezését, mert ez megjelenítés legfontosabb része. Első paraméterként egy textúra objektumot kér, amely tárolja a képet. Második paramétere egy vektor példány, amely a kép koordinátáit tartalmazza. X és Y koordinátákra a kép jobb felső sarkát teszi alapbeállításban.

Következő fontos paraméter a forgatást végzi el, amelyet radiánban kell megadni, tehát teljes elforgatás az 2\*π. Jelentősége, hogy ha több hasonló példányt szeretnénk elhelyezni a képernyőn akkor azt nem kell külön-külön textúrákban eltárolni, hanem tudjuk transzformálni. Például ha irányító gombokat használunk az irányításra, akkor azokat csak forgatni kell.

Az ez utáni paraméterrel a „sprite” példány origóját lehet beállítani, ez főleg az előbb említett forgatásnál játszik fontos szerepet. Ha megadjuk ezt a belső origót, akkor e körül a pont körül forgatja a képet.

A következő paraméter is egy transzformációs beállítás lehetőségét adja, a példány méretét tudjuk növelni vagy csökkenteni. Egy olyan értéket kell átadni, ami nem egész szám típus. Az 1.0f érték az eredeti méret, a 0.5f a kép méretét a felére csökkenti és így tovább.

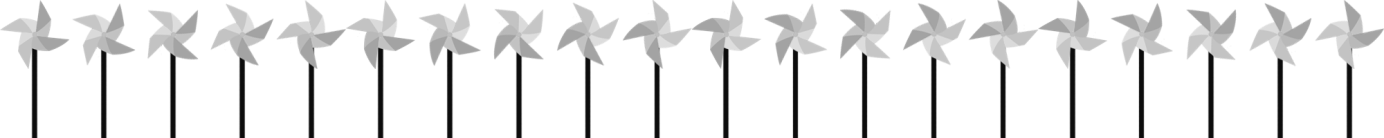
Az utolsó paraméter a „layer depth” mellyel a kirajzolt objektumok egymáson való elhelyezkedését lehetséges szabályozni. Alap esetben ez 0.0f, ekkor kirajzolási sorrendben helyezkednek el egymáson a képek.

## 2.2.2. Animáció

Kétdimenziós grafikai animálásra egy régi, de jól bevált technológiát használnak általában. A képek egymás utáni kirajzolásán alapul. Kétféleképpen szokták ezt használni.

Az egyik az, hogy minden egyes képkockát külön-külön, képekben tárolunk el és ezeket egymásra rajzoljuk.

A másik azaz, hogy minden mozzanatot (képkockát) egy képben tárolunk, úgy hogy azokat egymás mellé vagy egymás alá helyezzük el (2.2.2.1. ábra). Kirajzolásnál meg tudjuk adni, hogy a kép melyik részét rajzolja ki, ezt egyszerűen tudjuk automatizálni, ha a képek szélessége és magassága megegyezik (márpedig egy karakter vagy bármilyen más objektum animálásánál a mérete nem változik csak a különböző részeinek a helyzete), akkor egy ciklussal be tudjuk járni a képet.



*2.2.2.1. ábra – Animáció*

# 2.3. Grafikus objektumok ütközéseinek kezelése

Ütközésvizsgálat fajtájának a kiválasztásakor fontos figyelembe vennünk, hogy milyen formájú objektumaink vannak. Ha szabályos alakzatokat (pl. négyzet, háromszög, kör) használunk, akkor megfelel a „Rectangle collision detection” technika használata, viszont ha amorf formákkal dolgozunk, akkor érdemesebb a „Pixel collision detection” technológiát alkalmazni, mert pontosabb és növeli a felhasználói élményt is.

## 2.3.1. Rectangle collision detection

Lényege, hogy az formáinkat egy négyzettel vagy téglalappal rajzoljuk körül és ezt tekintjük az elem széleként. Az ábrán (2.2. ábra) látszik ezzel a probléma, hogy valójában nem történt átfedés vagy ütközés ezért ezt akkor érdemes használni, ha nem fontos a játék szempontjából és nem észrevehető, csak kis mértékben.

*2.2. ábra - Rectangle collosion detection*

*2.3. ábra - Pixel collosion detection*

## 2.3.2. Pixel collision detection

A vizsgálat, hasonló az előzőhöz csak bevezetünk plusz feltételt, ami megvizsgálja, hogy ahol éppen az objektum egy színes pixele van az fedi-e egy másik objektum színezett pixelét. A mellékelt ábrán jól látszik a különbség (2.3. ábra). Ez a fajta ütközéskezelés időigényesebb és erőforrás igényesebb, viszont sokkal pontosabb és növeli a felhasználói élményt. [3]

3. Felhasználói bemenetek kezelése

A Windows Phone 7 játékoknál és bármilyen telefonos operációs rendszerre írt alkalmazásnál fontos, hogy az irányításnak milyen logikát adunk, mivel a képernyő mérete kicsi. Vizuálisan megjelenített gombok használata szerintem előnytelen, mert akkor még kisebb a játékterünk.

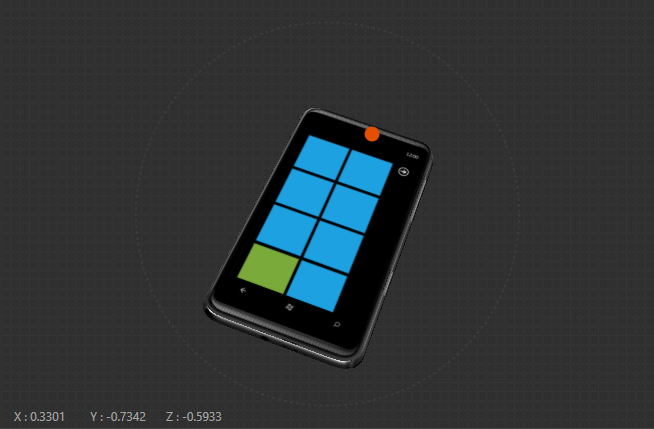
Mint a legtöbb okos telefon operációs rendszer így Windows Phone 7 is egyszerre több érintést tud kezelni. Ahhoz, hogy ezeket az érintéseket feldolgozzuk, az operációs rendszer biztosít feldolgozó metódusokat, amiket a programozó eseményként kap meg, de azt is meg tehetjük, hogy erre saját algoritmust írunk, mivel minden érintésnek megkapjuk az X és Y koordinátáit, amiket egy listában tárol el. A Windows Phone 7 operációs rendszerbe beépített érintési minták a következőek: „Tap”, „Double Tap”, „Touch and Hold”, „Pan”, „Flick”, „Pinch and Stretch” (3.1. ábra). Ahhoz, hogy az XNA játékunkban is használni tudjuk ezeket a mintákat, ahhoz engedélyezni kell.



*3.1. ábra – Windows Phone 7 gesztusok*

Felhasználói bemenetek közé sorolnám a gyorsulásmérőt is, amely játékfejlesztő és felhasználó szempontjából is előnyös (3.2. ábra). Ez az eszköz megadja, hogy milyen irányban hat a gyorsulás a telefonra. Alaphelyzetben egy Föld középpontja felé mutató vektor melynek van X, Y és Z koordinátája, ahogy forgatjuk, a telefont úgy változnak ezek a koordináták. Ezt az eszközt is külön engedélyezni kell, ahogy az előbb említett gesztusmintákat.

Giroszkópot is építenek a legtöbb Windows Phone-ba amely az előzőtől eltérően nem gyorsulást mér hanem a telefon mozgását a térben. [1]



*3.2. ábra – Gyorsulásmérő szemléltetése*

4. XNA menü készítése

Menü készítése minden játéknál nyilvánvalóan szükséges mivel az eredményeket menteni kell, megjeleníteni, játék beállítási lehetőséget biztosítani a felhasználónak, pályát választani stb. Külön menükészítésre tervezője nincsen az XNA-nak. Két lehetőségünk van menüszerkesztésre: XNA játékciklusba programozunk egy menüt vagy XNA és Silverlight összevont alkalmazást készítünk.

# 4.1. Menü készítése XNA-val

Játékciklusban körülményes menüt készíteni, nincsenek események, amelyeket generál a felhasználó, hanem egy végtelen ciklus van. Nincsenek gombok vagy bármilyen grafikus vezérlők, amiket használunk, hanem minden használni kívánt elemet, gombot, listát nekünk kell programozni vagy szerkeszteni. Ugyanúgy kell kezelni ezeket a felhasználói inputokat mintha a játékos játszana, azaz pixelterületeket kell vizsgálnunk. Ha gomb területén belülre tapintott, akkor arra reagáljon valahogy a program. Fontos arra figyelni, hogy ilyenkor még a játékciklus folytatódik, tehát ha például egy gombot lenyom a felhasználó, akkor arra csak egyszer reagáljon a program. Ha folyamatosan lenyomva tartja például 1 másodpercig, akkor 30-szor következik be az esemény, ha ezt nem kezeljük.

# 4.2. Menü készítése XNA és Silverlight összevonásával

Ebben az esetben a menü elkészítése eseményvezérelt programozással elkészíthető, ami megkönnyíti a fejlesztő munkáját. Ez a típusú hibridalkalmazás a Windows Phone 7 Mango-val jelent meg. A projekt létrehozásakor automatikusan létrejön két xaml fájl. Az egyik az maga az XNA játék (GamePage.xaml), a másik pedig egy Silverlight alapú oldal (MainPage.xaml).

A GamePage.xaml fájl xaml kódját nem engedi szerkeszteni, itt csakis a „code-behind”-ba írhatunk. Ha megnézzük a mögöttes kódot látszik, hogy a legenerált XNA metódusokat tartalmazza.

A MainPage.xaml fájl pedig minden olyan feladatra alkalmas, ami nem köthető a játék logikájához. A projektnek az a része eseményvezérelt, tehát rendelkezésre állnak a Microsoft által biztosított alap vezérlőelemek (gombok, listák, szövegmezők stb.). Erről az oldalról el tudunk navigálni a GamePage.xaml-re illetve a vissza is tudunk navigálni a játék oldaláról.

## 4.2.1. Silverlight

A Silverlight technológia előtt alkalmazások fejlesztésére a Windows Forms állt rendelkezésre a programozóknak. A felhasználói igények manapság megváltoztak ezért a Microsoft a Silverlight-tal újított. Az interaktív felhasználói felület létrehozására alkalmatlan volt az előbbi megoldás, merev, kötött volt a GUI fejlesztése.

A Windows Forms helyettesítésére a Microsoft belekezdett az Avalon projektbe, amit később átkereszteltek Windows Presentation Foundation (WPF) névre. Ez a technológia rugalmasabb és szebb, interaktívabb felhasználói felülettervezést biztosított. Ez a technológiai újítás 2006-ban jelent meg a .NET 3.0 keretrendszerrel. Megjelent sok új vezérlőelem, új vezérlőmodell és már megkülönböztetnek csapat projekten belül XAML fejlesztőt.

## 4.2.2. XAML

A WPF-vel együtt megjelent a felhasználói felületet leíró nyelv, amely az eXtensible Application Markup Language (XAML). Szintaktikája az XML-hez hasonló, melyhez tartozik egy XAML Parser ami értelmezi, ellenőrzi ezt az alkalmazás leíró nyelvet és létrehozza a felület struktúráját. Ezt a nyelvet nem csak WPF technológia kapcsán alkalmazzák, hanem a Microsoft úgy vélte, hogy ezt érdemes lenne a webes felületek készítésénél is alkalmazni, ennek eredményeképp jött létre a Silverlight. Jelenleg úgy néz ki, hogy ez a jövőben ez lesz a cég fő irányvonala. A Windows Phone 7.0 és a Windows Phone 7.1 megjelenésével terjedt el, de most a legújabb operációsrendszerük a Windows 8 alapja is XAML alapú, és az asztali alkalmazásfejlesztés is ezt az irányt követi. [1]

5. Hangok hozzáadása a játékhoz

A hangok és a zene fontosak a játékélmény növelésében. Minden játékra jellemző, hogy, szól alatta van valamilyen zene illetve a különböző játékbeli történésekhez hangeffektek vannak csatolva.

A hangokat ugyanúgy, mint a képeket, a „Content pipeline” segítségével érjük el. Használata hasonló az eddig említett tartalmakhoz, be kell tölteni először a hangfájlt, utána szabadon használható.

XNA Framework-höz tartozik a XACT audió programozása. Ennek segítségével nem egyszerűen hangokat tudunk hozzákapcsolni a különböző akciókhoz, hanem azokat programozni is tudjuk. Például ha egy objektum közeledik, akkor az elején halk, és ahogy egyre közelebb ér annál hangosabb lesz a hangja.

# 5.1. XACT audió programozása

Főleg háromdimenziós játékoknál van nagy szükség a hangok programozására így ezért ezt a témát nem fejtem ki. A következő formátumokat engedélyezi a XACT: WAV, AIF, AIFF. Ha MP3-at szeretnénk hozzáadni, akkor nem szükséges a XACT használata akkor elég a Song osztályt használni. [3]

6. Multiplayer játék fejlesztése

Azért esett a választásom multiplayer játék készítésére, mert manapság már nagyon széles körben használnak mobilinternetet a felhasználók. Általában aki vásárol egy okos telefont az előfizet ezzel együtt mobilinternetre is. Tapasztalataim alapján nem sok multiplayer játék szerepel a „Piactéren” pedig véleményem szerint lenne rá kereslet.

Fontos döntés az, hogy a megjelenítést, hogyan kezeljük több játékos esetén. Az én esetemben egyszerre 2 játékos lehet a játékban.

Két választásom van, az egyik, hogy a képernyőt két részre osztom és az egyik részén a saját játékterét látja, a másik részén pedig az ellenfél játékterét mutatja. Ezzel a megoldással nem lenne probléma, ha egy nagyobb képernyő állna rendelkezésre, például egy táblagép vagy számítógép. Mivel a játékot egy kis képernyős eszközre készítem, ezért ez a módszer nem lenne túl optimális.

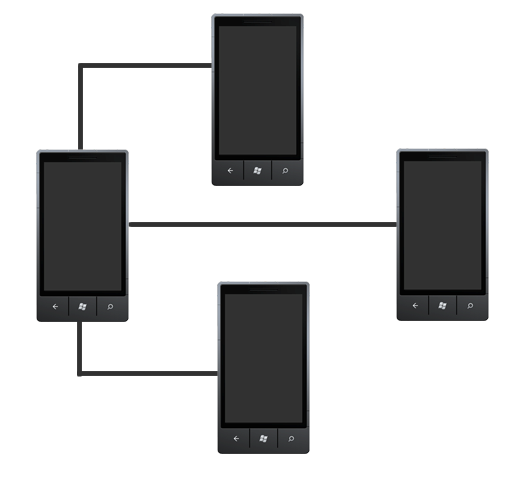
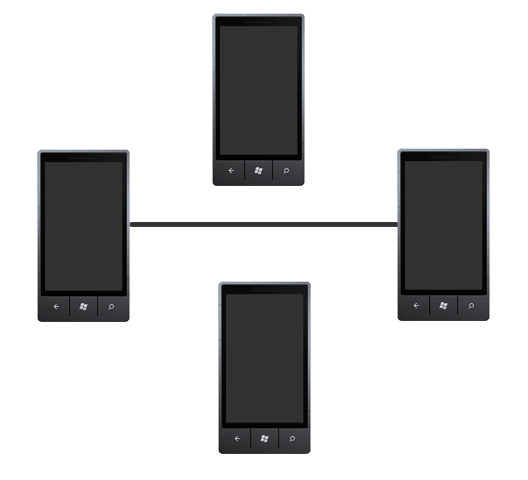
A másik választásom az, hogy nem osztom ketté a megjelenítést, hanem egy térben szerepeltetem a két játékos karakterét. Ezt a megoldást fogom választani a fejlesztés során, mert jobb mértékben használja ki a kis átmérőjű képernyőt, illetve a játék logikájával is ez a megoldás sokkal szemléltetőbb. Ebben az esetben szükséges lesz kezelnem az ütközéseket. Kétdimenziós játékot készítek, mégis a két karaktert térben majd úgy kell kezelem mintha az egyik közelebb lenne mint a másik. Természetesen ez a játék logikájától függ.

Az XNA Framework nagymértékben támogatja a hálózati játékokat, könnyen lehet a segítségével fejleszteni ilyesfajta alkalmazásokat.

# 6.1. Peer-to-peer hálózatok

Az ilyen fajta hálózati megoldás használata akkor előnyös, ha az adatforgalom magas a kliensek között.Ez egy nem központosított rendszer, azaz nincsen köztes szerver, ami végzi az adatfeldolgozást, hanem a hálózaton levő eszközök valósítanak meg egy-egy objektumot, ami végzi a feldolgozást, küldést és fogadást. Ez a fajta architektúra képes arra, hogy minden eszköz lát minden eszközt, amivel egy hálózaton van. Emiatt előszeretettel használják chat programokban, illetve olyan szoftvereknél ahol a biztonság nem magas prioritású. Manapság a kliensek (köztük az okos telefonok is) már olyan erőforrással rendelkeznek, hogy képesek elvárt időn belül magas szintű és gyors feldolgozásra.

Játékszoftver szempontjából ez egy előnyös módszer, ha az eszközök egy hálózaton vannak. Nincs szükség szerverre, ami lassítaná a kommunikációt. Mivel adatot tárolni adatbázisban nem mindig kell, hanem a játékos adatait (pl. pozíció, pontszám) kell megosztani másik játékossal, ezért felesleges beiktatni egy központi eszközt, amin keresztül történik a kommunikáció. Többszemélyes játék esetén, ahol szükséges, hogy minden játékos kommunikáljon minden játékossal ott megoldható a multicast (csoportos üzenetküldés) címzéssel (6.1.1. ábra). Két személyes játék esetén, ha több eszköz van a hálózaton, akkor unicast (6.1.2. ábra) címzést szokás használni, azaz közvetlenül egy eszköz felé történik az üzenetküldés IP cím alapján. [4]

*6.1.1. ábra – Multicast címzés 6.1.2. ábra – Unicast címzés*

# 6.2. Kliens-szerver hálózatok

A kliens szerver architektúra alkalmazása játékoknál ugyanolyan gyakori, mint az előbb leírt, ez játéktípusonként változik. Ez a topológia, úgy épül fel, hogy a szerver nyújt szolgáltatásokat, melyeket a kliensek használnak. Biztonságosabb, mint az előbb említett peer-to-peer architektúra, és jobban felügyelhető, menedzselhető és skálázható. Én a fejlesztés során egyrészt adatok tárolására használnám, amelyek nem játékidőben kapnak fontos szerepet, hanem magáról az adott kliensről tárol információkat. Mindenképpen szükség van a szerverre olyan játék esetén, amikor a kliensek nem egy hálózaton vannak. Ebben az esetben a szerver, mint kapcsolattartó funkcionál az eszközök számára.

Kliens szerver architektúrának két féle szervezési formája van. Az egyik az úgynevezett vékony kliens (6.1. ábra) amikor a végpontok csak a megjelenítő egységként funkcionálnak, azaz nem fut rajtuk semmilyen olyan algoritmus, eljárás, ami az üzleti logikához tartozik. A szerver végzi el az alkalmazás logikájához tartozó feladatokat. Amiért a szerver felel minden számításért ezért nagy lehet az adatforgalom, ami túlságosan leterhelheti a szervert, ha egyszerre sok kliens aktív. Ez nagy hátránya ennek az architektúrának. Előnye, hogy alacsony erőforrással bíró eszközök esetén nem lassul a rendszer, illetve jól használható különböző teljesítményű kliensek esetén is.

Megjelenítés

**Szerver**

**Kliens**

Adatkezelés Alkalmazás feldolgozás

*6.1. ábra – Vékony kliens*

Másik modell a vastag kliens, amely a 6.2. ábrán látható. Ebben az esetben már kap a kliens is feladatot, például a bevitt adatok validálása illetve az alkalmazáslogika egy részét is a kliens oldalán valósítják meg. Ebben az esetben az előbb említett modell hátrányát küszöböli ki, azaz a szervernek a feladata az adatok tárolása és ezeknek a menedzselése. Kihasználja a végpontok képességeit. Ha a kliensek PC-k (képesek számításokat és bonyolultabb eljárásokat is futtatni), akkor az alkalmazás feldolgozás átültethető ezekre az egységekre. [3, 4]

6. Adatkezelés Windows Phone 7.5 Mangon

**Kliens**

**Szerver**

Adatkezelés

Megjelenítés, Alkalmazás feldolgozás

*6.2. ábra – Vastag kliens*

Minden játékban szükséges adatokat tárolni a játék állapotáról, ha azt szeretnénk, hogy a felhasználó vissza tudja nézni az eredményeit, képes legyen úgy fejleszteni a karakterét, hogy azt ne kelljen minden indításkor elölről kezdenie és ne kelljen ahhoz szerverhez kapcsolódnia, hogy elérje a lokális játék státuszát. Erre a célra az úgynevezett Isolated Storage áll rendelkezésre a fejlesztőnek.

# 6.1. Isolated Storage

Windows Phone 7.5 Mango esetén csak ezt területet használhatjuk fájlok írására és olvasására. Magához a fájlrendszerhez nem férünk hozzá, ezért a különböző alkalmazások nem érik el egymás adatait ennek segítségével. Ez az elkülönített tárhely nincs korlátozva, tehát elvileg bármilyen nagyságú helyet tudunk foglalni a memórián az alkalmazásnak. Mivel nincsen korlátozva a helyfoglalás ezért itt ügyelni kell a programozói hibákra. A fájlok törléséről és karbantartásáról a fejlesztőnek kell gondoskodni (ideiglenes fájlok törlése). Három féle adatkezelési módszer létezik erre az operációs rendszerre: Settings, Files and Folders és a Relation Data.

## 6.1.1. Settings

Ha ezt a megoldás használjuk, akkor meglehetősen megvan kötve a kezünk, mert csak kulcs/érték párokat tárolhatunk a memóriában. Akkor érdemes használni, ha valóban ilyen egyszerű adatokat szeretnénk tárolni és olvasni. Általában felhasználói adatok megjegyzésére használják. Például bejelentkező felületnél ne kelljen a felhasználónak mindig begépelni a felhasználónevét és jelszavát. Ebben az esetben egy listához tudjuk hozzáadni ezeket a párokat, tehát nem szükséges konkrét fájl létrehozására vagy törlése amibe írni szeretnénk.

## 6.1.2. Files and Folders

Ebben az esetben a szokásos módon zajlik az írás, olvasás. Tudunk létre hozni fájlokat, mappákat. Az írás/olvasás stream alapú, ahogy azt megszokhattuk.

## 6.1.3. Relation Data

Ez a fajta adattárolás a Windows Phone 7.5 Mango-ban jelent meg először a Windows Phone operációs rendszer alapú telefonoknál. Képesek vagyunk lokális adatbázist létrehozni és ezeken műveleteket illetve lekérdezéseket végrehajtani. Természetesen mivel .NET alapú ezért ezeken többek között „LINQ to SQL” technológiával tudunk végrehajtani lekérdezésket és parancsokat. Ez egy olyan .NET Framework-be ágyazott keretrendszer, amelynek segítségével SQL parancsokhoz hasonló szintaktikájú lekérdezéseket tudunk írni és ez végrehajtódik az adatbázison. Ahhoz, hogy az adatbázist el tudjuk érni egy proxy kell amin keresztül ezt megtehetjük, ezzel az objektummal lehet felépíteni a kapcsolatot a táblákkal. Ezen az úgynevezett „DataContext” példányon keresztül a táblákat és hozzájuk tartózó oszlopokat elérjük, mint entitás illetve attribútum. Ahhoz, hogy fel tudjuk dolgozni a mezőket szükséges létrehozni egy osztályt, ami reprezentálja nekünk az adott táblát. A megfeleltetést az adatbázisban levő tábla és a programkódban szereplő objektum között „Mapping”-el hozzuk létre. [1]

7. A játék részletes specifikációja

# 7.1. Egyszemélyes játék

A cél egy 2 dimenziós, oldalról nézetes játék fejlesztése, melyet egyszemélyes és kétszemélyes módban is lehet játszani. A játék fő karaktere egy papírrepülő melynek teljesíteni, kell a különböző pályákat. A repülőt, a képernyő érintésével lehet irányítani. A repülő sebessége állandó, a felhasználó a repülő magasságát tudja változtatni. Egyszeri képernyőérintéssel attól függően, hogy éppen a repülő felfele repül, vagy lefele száll, az ellentétjére változik a repülő mozgásának iránya. Ha játékos elérte a képernyő tetejét vagy alját, akkor az onnan visszaverődik, azaz irányt változtat. A pályákon akadályok, pontot érő és viselkedést befolyásoló objektumok szerepelnek. Játék célja, hogy az akadályokat kikerülje és minél több pontot szerezzen a játékos.

A pályák előre be vannak építve a játékba. A pályák építése szövegfájlból történik, melyben az előre meghatározott karakterek egy-egy pályaelemet szimbolizálnak. Az éppen betöltött pályát véletlenszerűen választja ki egy funkció. A pálya feldolgozása és betöltése után a játékos kezdheti a játékot. Ha a játékos akadállyal ütközik, akkor mínusz pontot ér mindaddig, amíg a repülő ki nem tér az akadályból. A pontot érő elemek egyenként 5 ponttal növelik a pontszámot. A pályák végesek, ha elérte a játékos a pálya végét, akkor az elért pontszám hozzáadódik az eddigi pontszámaihoz. Nyilvántartom a legjobb eredményt, melyet, ha elér a játék során, akkor a játék eredményéhez hozzáadódik az elért pontoknak az 5 %-a. A játékmenet végét az jelzi, hogy a repülő felgyorsul és horizontálisan eléri a képernyő szélét. Egy pályának a végigjátszását meglehet szakítani, illetve a megszakított állapottól folytatni. Ha kilép a játékból vagy visszanavigál a játék közben a főoldalra, akkor az eddig elért pontszámát elveszíti. Telefonra való fejlesztésnél ügyelni kell a különböző felhasználón kívüli megszakításokra, például telefonhívás. Ha érkezik egy ilyen esemény, akkor a játékállapot szünet állapotba kerül. A pálya, különböző játékmenetet módosító elemeket tartalmaz, melyet a játékos felvehet. Ezek véletlenszerűen módosítják a játékos tulajdonságát, melyek a következők lehetnek:

* Repülő gyorsítása
* Repülő lassítása
* A felszedett érmék értékeinek duplázása

Ponthatárokat határoztam meg melyek szerint kategorizálom a játékost, melynek az oka, hogy a kétszemélyes módban azonos tapasztalattal rendelkező játékosokat sorsoljak össze. A kategóriák a következők:

* „Beginner” - 0 és 3000 pont között
* „Amateur” - 3001 és 7500 pont között
* „Professional” - 7501 és 15000 pont között
* „Hardcore” - 15001 és 25000 pont között
* „Impendent” - 50000 pont felett

# 7.2. Játék ellenféllel

Kétszemélyes módban a játék célja, hogy a játék végén kiértékelt eredmény nagyobb legyen, mint a másik játékosé. Az eredmény két paramétertől függ, az egyik, hogy a játékosok milyen sorrendben érték el a pálya végét, a másik pedig, hogy ki szedett össze több pontot. A végeredmény kiértékelését a következő modell mutatja be:

* Elsőnek beérkezett játékos eredménye: Pontok száma \* 1.05
* Második beérkezett játékos eredménye: Pontok száma
* Ha egyszerre érkeznek a célba, akkor a második szabály érvényes

Fent leírt modell alapján, két fajta stratégiájú játékos különböztethető meg. Mivel a pálya végét, csak úgy lehet rövidebb idő alatt elérni, hogyha vesz fel olyan módosító elemet, ami gyorsítja, ezért kockáztatnia kell, tehát van a „kockáztató játékos”. Ha a játékos nem veszi ezeket a játékot befolyásoló elemeket, akkor is nyerhet, tehát van a „nyugodt játékos”. A fent leírt szabály alapján két végeredmény lehetséges:

* Az egyik játékosnak több pontja van
  + Nyertes pontszáma: Pontok száma \* 1.2
  + Vesztes nem kap pontot
* Döntetlen
  + Mindkét játékos megkapja a szerzett pontját

A játékosok összesorsolása, úgy történik, hogy a felhasználó kiválasztja a kétszemélyes módot, majd kapcsolódik. Az éppen kapcsolódó játékosokat összegyűjti, majd összesorsolja a megfelelő szinttel rendelkezőket. Ha nem talál azonos tudású partnert, akkor erről értesítést kap és megpróbálhat újra kapcsolódni. Ha sikerült kapcsolódni, akkor szinkronizálja a két telefont, hogy egyszerre kezdődjön a játék. A felhasználók látják egymás repülőit és pontjait, a két repülőt úgy különbözteti meg a program, hogy az ellenfél repülőjének a színe szürke színű. Ez a megoldás a specifikáció megírásánál jónak tűnt, viszont a fejlesztés során kiderült, hogy ez a megoldás zavaró. Felesleges a másik játékost megjeleníteni teljes egészében. Ezért másik megoldást kellett keresnem mely, úgy néz ki, hogy egymásról a két játékos azt látja, hogy hol tartanak a pályán. Ezt a képernyő tetején vagy alján jelenítem meg egy kis sávban. Másik megjelenítendő adat az ellenfél pontszáma. A karakterek nincsenek egymással közvetlen kapcsolatban, azaz nincs ütközés vagy más interakció közöttük. A játék végét az jelenti, ha mind a két repülő elérte a pálya végét vagy kilépett. Ha kilép az egyik játékos, akkor azonnal vége a játéknak és a pontszám, amit szereztek nullázódik. Két személyes módban nincs lehetőség szüneteltetni a játékot. Ha az egyik résztvevő előbb éri el a célt, akkor megvárja, míg beér a másik játékos és ezután történik a verseny kiértékelése. A várakozás is a verseny része, ezért ha kilép ugyanaz a szabály érvényes mintha még a játék közben hagyta el volna a játékot.

A megjelenítésének kialakításához pixel grafikát alkalmazok, azaz minden elem egy kétdimenziós koordináta rendszerben helyezkedik el. Törekszek 3D hatást keltő kinézetre. A játékmenetben a hátteret úgynevezett „parallax scrolling” technikával oldom meg. Ennek a lényege, hogy a háttér elhelyezése három dimenzióban történik, de valójában az irányítás az kétdimenziós. Háttér több rétegből tevődik össze, melyek egymástól különböző távolságra vannak. A távolabb levő elemek lassabban mozognak és elmosódottabbak, a közelebb levő elemek gyorsabbak és élesebbek.

8. A játék tervezése

Tervezés a legfontosabb szempontja, hogy az implementáció jól paraméterezhető legyen. Finomhangolásokat el lehessen végezni a fejlesztés végén anélkül, hogy bele kelljen nyúlni a logikába. Finomhangolás alatt például grafikai elemek kicserélését, sebességek módosítását, új elemek hozzáadását értem. Jól konfigurálható programkód több kódsort jelent, illetve előreláthatólag több elágazást, ami növeli a futási időt, ezért igyekszem megtalálni az optimumot a gyors működés és a jól paraméterezhetőség között.

Másik fontos szempont a tervezés során a grafikai elemek skálázhatósága. A Windows Phone 8 operációs rendszerrel együtt több féle felbontás jelent meg a telefonokon, ez jelenleg 3 féle lehet ([5]):

* WVGA
  + Felbontás: 480 x 800
  + Képarány: 15:9
* WXGA
  + Felbontás: 768 x 1280
  + Képarány: 15:9
* 720p
  + Felbontás: 720 x 1280
  + Képarány: 16:9
* 1080p
  + Felbontás: 1080 x 1920
  + Képarány: 16:9

Véleményem szerint erre két megoldás létezik. Az egyik, hogy a képek pozícióját nem direkt adom meg, hanem a képernyő szélességéhez és magasságához képest. Ennek a megoldásnak az előnye, hogy bármilyen felbontás esetén ugyanazt az eredményt fogjuk kapni, ha megjelenne egy új, akkor nem kellene módosítani az implementációt. Hátránya viszont, hogy a pozíciók kiszámításához az osztás műveletet kell használni, ami nagy elemszám esetén költséges művelet és csökkenti a futási időt. Ez az elemszám egy játékban nagy, sok a grafikai elem illetve Windows Phone esetén a számításokat másodpercenként akár harmincszor végre kell hajtani. Ez a módszer a hétköznapi alkalmazásokban hatékony lehet.

Másik megoldás, hogy felbontás típusonként más képeket töltünk be és elágaztatjuk a programot attól függően, hogy milyen rezolúciójú telefonon fut éppen a játék. Ennek a módszernek az előnye, hogy nem kell számítási műveleteket végrehajtani. A hátránya viszont, hogy ha megjelenik egy új felbontás, akkor fejleszteni kell hozzá egy új komponenst, ami kezeli ezt az új modellt.

Tervezés előtt figyelembe kell venni, hogy egy ciklusban van mindaddig a játék, amíg ki nem lép a felhasználó. Nincs lehetőség oldalak és nézetek létrehozására illetve nincsenek események, amikre fel lehet iratkozni. Erre egy globális változó a megoldás, ami felsorolás típusú, és eltároljuk benne a játék állapotát. Minden frissítésben meg kell vizsgálni, hogy az előbb említett változónak éppen mi az értéke. Ezek az állapotok a következőek lehetnek a tervezett játékban:

* Főmenü
* Útmutató a felhasználónak
* Egy személyes játék
* Játék ellenféllel
* Játék szüneteltetése
* Játék vége

# 8.1. Menü komponens tervezése

Első fontos átgondolandó probléma az a játékban szereplő menük kezelése. Előző fejezetekben két opciót érintettem ami XNA keretrendszeren lehetséges megoldás. Nem a Silverlight alapú megoldást használom, mert a későbbiekben, ha más operációs rendszerre is implementálni szeretném, akkor nem biztos, hogy a rendelkezésre áll egy ilyen hibrid technológia. Az általam választott megoldás a játékciklusban történő menükezelés.

Az előbb említett megoldás jól átgondolt és újrahasznosítható tervet igényel, mert a fejlesztés során szükség lehet gyorsan és egyszerűen implementálni egy új menüt. Megoldandó probléma a gombok érintésének kezelése. Erre a célra a terv egy osztály készítése mely kezeli ezt az eseményt (8.1.1. ábra).

*8.1.1. ábra – Gomb osztálydiagram*

A bejövő érintéseket kezeli a keretrendszer, eltárolja a pontos koordinátákat és, hogy milyen gesztus történt. Gomb esetében ez a gesztus az egyszeri érintés. Ha a menüben tartózkodik a játékos minden frissítésnél meg kell vizsgálnunk, azt, hogy az érintés a gomb területén belül történt-e. A gombok jelen esetben textúrák melyeknek van magassága és szélessége, így könnyen kiszámolható ezen a területen található pixeleknek a koordinátái.

Gombkezelő véleményem szerint nem elégséges a megfelelő tervezéshez. Mint már említettem menüket kell létrehozni, egy menü az gombok összessége. Egy olyan menü osztályt tervezek, mely gombok gyűjteményét tartalmazza többek között (8.1.2. ábra). Az előbb említett gomb osztályt ki kell egészíteni egy azonosítóval melynek egyedinek kell lennie. A menü osztály tartalmaz egy metódust, ami végig iterál az gomb listán és megvizsgálja, hogy melyik gomb területén történt az érintés. Ha történt érintés, akkor a programnak arra az ágára fut az alkalmazás és átállítja a játék állapotot a megfelelőre.



*8.1.1. ábra – Menü osztálydiagram*

Ezzel a tervezettel sikerült, egy újrafelhasználható eseménykezelőhöz hasonló osztályt létrehozni és lefedni a menükészítést. A következő fejezetekben már a játék logikájára vonatkozó tervezés lesz előtérben.

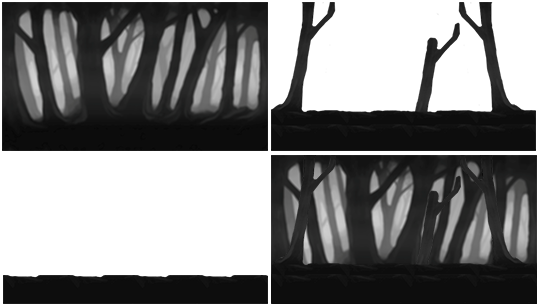
# 8.2. Háttér animáció tervezése

A játékban a háttér görgetése végtelen, azaz mindig ugyanazt a képet mozgatja valamelyik irányba. Mint már említettem, az úgynevezett „parallax scrolling” technikát alkalmazom (8.2.1. ábra). Az XNA keretrendszer rajzoló metódusánál lehet sorrendet definiálni, azaz ha egy textúrát a programkódban később rajzolok ki, de érint más objektumot vagy objektumokat akkor a később kirajzolt kerül előre, így könnyen lehet különböző transzparens rétegeket egymásra helyezni. Az első kép a legtávolabb levő réteg, a második már közelebbi, az utolsó pedig a repülővel egy szintben levő textúra (8.2.2. ábra). Paraméterezhetőnek kell lennie a görgetés sebességének, mivel a közelebb levő rétegek gyorsabban mozognak, illetve a játék során ez dinamikusan változtatható. A sebesség mértékegysége pixel, mely megadja, hogy egy frissítés során hány pixelt ugorjon a réteg. Szükséges a legtöbb osztályban egy függvény, amely kezdőpozícióba állítja a változókat. Ebben az osztályban ez a „RePosition” metódus mely a sebességet visszaállítja a keződsebességre.



*8.2.1. ábra – Háttér osztálydiagram*

Mivel a repülő sebessége állandó és egyirányú, ezért az algoritmus alapja a beillesztett háttér egymás utáni beillesztése. Ha a kép széle eléri a képernyő végét horizontális irányban, akkor ugyanazt képet odailleszti a kép széléhez és a két képet egymás után, egyszerre görgeti (8.2.3. ábra).



*8.2.2. ábra – Háttér rétegei*

*8.2.3. ábra – Háttér görgetésének szekvencia diagramja*

# 8.3. A játékos osztály tervezése

Ez az osztály igényelte a tervezésből a legtöbb időt, két részre osztható téma szerint. Az egyik része az animálás illetve a repülés fizikája, ez a megjelenítéssel kapcsolatos, a másik része pedig a háttérben levő folyamatok, mint a pont számlálása, ütközések kezelése.

A fizika az játék specifikus, viszont az animációt igyekeztem újrafelhasználhatóvá tenni. A tervezés fázisban nem volt még kigondolva az animálás illetve a karakter, ezért próbáltam a legáltalánosabbra tervezni. Definiáltam egy felsorolás típust mely eltárolja a játékos státuszát, jelen esetben ezek a repülés, esés és az egyhelyben repülés. Osztályok megtervezése előtt ezekre különböző féle animációt gondoltam, végül a repülő nem lesz animálva. A frissítés metódusban ettől az állapottól függően rajzolja ki a karaktert.

A következő műveletek keletkeznek repüléskor: repülés felfele, esés, (egy helyben repülés). Az utolsó eseményt is hozzáadtam a játékhoz, de valójában nem lesz használva. A későbbiekben, ha változtatni kívánom a grafikát, akkor a logikát már nem kell módosítani. Az irányítás könnyítése és a repülés fizikájának modellezése miatt a felfele repülésnek és az esésnek is van gyorsulása és végsebessége, utóbbi az osztályon belül konfigurálható. Mint már említettem a specifikációban irányítani a képernyő érintésével lehet.

Másik része a játékos különböző tulajdonságai, melyek nem játszanak szerepet a fizikában. Ilyen a pontszám (amelyre külön osztályt készítettem), ütközött-e akadállyal, vett e fel pontot érő elemet, vagy érintkezett e játékmenetet módosító elemmel.

A pontszámokat külön kellett választanom, mert ehhez tartoznak egyéb funkciók is, melyek logikailag szorosan nem kapcsolódnak a játékoshoz. Két eredményt tárolok el, az egyik az legutóbbi, vagy éppen folyamatban levő játék pontszáma, ezek összesítve, illetve a legmagasabb pontszám. Az összesített pontok alapján a rangot is itt vettem fel, mint tulajdonság, ami egy felsorolás típus. Az összesített pontszámot és a legmagasabb pontszámot a telefon memóriájában tárolom el, ezért ehhez szükségesek metódusok, amik elvégzik a mentést és az olvasást. Nem gondoltam szükségesnek az XML struktúrát ezért erre a célra egy egyszerű szövegfájlt használok.

# 8.4. Pálya kezeléséért felelős osztály tervezése

Terveztem egy olyan osztályt mely kezeli a pályákat a felépítéstől a kirajzolásig. Fontos, hogy később a pályák hozzáadása egyszerű legyen. Arra jutottam, hogy fájlokban célszerű egy pálya szerkezetét eltárolni. Szövegfájl feldolgozása egyszerűen implementálható .NET környezetben. A feldolgozási szabályokat a következőképpen gondoltam.

Meg kell különböztetni akadályt, játékmenetet módosító elemeket és a pontot érő objektumokat. Erre a következő konvenciót terveztem. Akadály az több féle lehet, ezért ezeket a számok jelölik 0-tól 9-ig. Maximum 10 féle akadály különböztethető meg. Játékmenetet módosító elem jelenleg maximum egy fajta lehet ezért ezt a „\*” karakter jelöli. A pontot érő elemeket pedig az „a” karakter szimbolizál. A pálya a végét pedig a „!” karakter jelöli (8.4.1. ábra).



*8.4.1. ábra – Példa a pályát leíró fájl tartalmára*

A pályaelemek pozícióját a fájlban való elhelyezkedésük határozza meg. Négy szint különböztethető meg vertikálisan, azaz az elem tényleges pozíciója azt a következő egyenlet alapján számolandó: Y = sor \* (képernyő felbontásának magassága / 4). Horizontális elhelyezkedésének kiszámolása hasonló az előbb leírtakhoz, annyiban különbözik, hogy az osztó 4 helyett 8, azaz: X = oszlop \* (képernyő felbontásának szélessége / 8).

A fentiekből azt a következtetést vontam le, hogy meg lehet határozni alap osztályokat melyekből öröklődnek a különböző típusú pályaelemek. Az egyik ilyen struktúra a „pályaelem” osztály, amely magába foglalja a közös tulajdonságokat és funkciókat, mint a textúra, pozíció, kép betöltése kirajzolása és animálása (8.4.2. ábra). Másik ilyen szerkezet a „pályaelemek” osztály, amely egy típusú elemek összességét fogalja magába. Úgy döntöttem, hogy ez egy absztrakt osztály legyen, a funkciók előírások, nem pedig kész algoritmusok, mivel ezek típusonként különbözhetnek.



*8.4.2. ábra – Pályaelem osztály osztálydiagramja*

**

*8.4.3. ábra – Pályaelemek osztály osztálydiagramja*

Tehát egy fajta pályaelem feldolgozásához három osztály szükséges megvalósítani. Az akadály osztálystruktúra felépítésén keresztül mutatom be a tervezetet. A hierarchia tetején a kollektor osztály áll (8.4.4. ábra), ami tartalmazza azokat a metódusokat és mezőket melyek a pályaelem listára vonatkoznak, ez származik a 8.4.3. ábrán látható struktúrából. Ez az osztály tartalmazza a pályaelemekből alkotott tömböt, mely a hierarchia következő szintje, ha fentről lefele haladunk. Ez a szint az egy elemre vonatkozó tulajdonságokat tartalmazza, tehát ez a 8.4.2. ábrán látható osztályból származik (8.4.5. ábra). Egy pályaelemhez tartozik egy index mely típusonként különböző. Az index tartalmaz egy azonosítót mely, típusonként egyedi, illetve itt tartom számon, hogy az elem hányadik sorban és oszlopban helyezkedik el (8.4.6. ábra).



*8.4.4. ábra – Akadály kollektor osztály osztálydiagramja*



*8.4.5. ábra – Akadály elem osztály osztálydiagramja*



*8.4.6. ábra – Akadály index osztály osztálydiagramja*

# 8.5. Ütközés kezelő komponens tervezése

Szükség van egy olyan szerkezetre, ami megint csak jól elválasztható. Két lehetőség közül választhattam. Az egyik, hogy az összes olyan osztályon belül kezelem ahol ez szükséges, de így nagyon szerteágazik, sok különálló komponens esetén. Ezért egy másik megoldást terveztem, mely külön kezeli más komponensektől az ütközéséket. Tartalmazza az összes olyan objektumot melyeknél szükséges az ütközésdetektálás, vagy hatással vannak rá az ütközésnél keletkezett változások. Ezért a következő referenciákat tartalmazza a szerkezet: akadályok, pontot érő elemek, játékmenetet módosító elemek, hátterek és a játékos.

A 2. fejezetben sorra vett megoldások közül a „pixel collosion detection”-t alkalmazom. Az oka ennek az, hogy olyan grafikai elemeket használok melyek nem egyszerű négyzetalakok, ráadásul animáltak és a játék egyik legérzékenyebb része ez.

A tervezés előtt teszteltem mindkét módszert. A téglalap alapú ütközésdetektálásnak az előnye, hogy nem annyira erőforrás igényes, mint a végül használt megoldás, de nagyon pontatlan. A teszt során kiderült, hogy érezhető feldolgozási időbeli különbség nincs a két opció között. Az osztály legfontosabb funkciója az a metódus, ami megvizsgál két objektumot, hogy ütköztek e. A megoldás hasonló a téglalap alapú ütközésvizsgálathoz. Első lépésben ki kell számolni az objektum körüli téglalap területét. Ha az egyik elem területének egy pontja érintkezik a másik elem egy pontjával, akkor még meg kell vizsgálni, hogy az érintett pont az átlátszó vagy színezett pixel. Ha mind a 2 pont színes, akkor detektálható ütközés. A 8.4.6. ábra jól prezentálja a folyamatot, mint szekvencia diagram.



*8.5.1. ábra – Ütközéskezelés szekvencia diagramja*

A vizsgáló metódus kész, a következő feladat a begyűjtött objektumok egymásra való hatásának vizsgálata. A játékos karakter van minden itt összefoglalt elemmel kapcsolatban.

Vizsgálni kell az ütközést egy akadállyal, aminek hatására a játékos „isDead” nevű logikai mezőjét módosítja igaz állításra. Ennek felhasználói felületre való vezetése a játékos osztályban történik, ami jelen esetben az, hogy a repülő színe pirosas lesz és csökkenti a játékos pontszámát. A sorra következő metódus a pontot érő elemmel való ütközésvizsgálatot végzi el. Ha bekövetkezik az érintkezés, akkor az elem törlődik és növeli a játékos pontszámát. Az utolsó vizsgáló metódus a módosító elemekkel való akciót figyeli. Ennek hatása komplikáltabb, mint az eddigiek. Első lépésben törlődik az elem. Második lépésben a hatás kiváltása a cél. Erre a célra szükséges a következő privát tulajdonságokat felvenni:

* Felszedett módosító indexe
* Egy számláló, ami a felszedés pillanatától számolja a ciklusokat
* Egy határ, aminél az előbb említett számláló megáll
* Egy logikai érték, ami eltárolja, hogy módosító folyamatban van e

Szükséges egy olyan metódus, ami megvizsgálja, hogy ha történt ütközés egy módosító elemmel, akkor melyik befolyásoló funkciót szükséges végrehajtani. Ha a felvétel megtörténik, akkor egy véletlen szám generátor dob egy számot, ami eldönti, hogy melyik módosítás hajtódjon végre. A hatás mindaddig tart, amíg a felsorolásban említett számláló el nem éri a határértéket.

Három féle módosítót terveztem, de a későbbiekben az könnyen bővíthető. Az egyik lelassítja a játékost. Mivel a játékosnak nincsen sebessége, csak az őt körülvevő elemeknek, mint a háttér és a pályát alkotó objektumok, ezért tulajdonképpen ezeknek a lassítása jelenti a repülő sebességének csökkentését. A második módosító hasonló az előbbihez, annyi különbséggel, hogy a játékost körülvevő grafikai elemeket gyorsítja. A harmadik pedig az, hogy a felszedett pontot érő elemek duplán számítanak. Azt, hogy ezek a metódusok milyen hatással vannak a játékmenetre azt, megjelenítem a felhasználói felületen szöveg formátumban, ehhez szükséges egy rajzoló metódus.

# 8.6. Egyéb kisegítő komponensek

A legtöbb szoftver fejlesztése esetén szükség van olyan funkciókra és mezőkre melyeket bármelyik programrészben fel lehessen használni példányosítás nélkül. Terveztem egy olyan osztályt, ami a névtér hierarchiában eggyel feljebb szerepel az eddig bemutatottaknál.

Az egyik ilyen komponens csak konstans vagy statikus mezőket tárol. Ide tartozik a telefon felbontásának szélessége és magassága, az akadálykirajzoláshoz szükséges osztók, a különböző betűtípusok és képek elérési útja és a programban többször felhasznált színek.

Másik ebben a névtérben található egység az nem egy osztály, hanem egy felsorolás típus, ami a játékállapotokat írja le, ezek a következőek:

* Játék folyamatban
* Hálózaton keresztüli játék
* Főmenü
* Játék közbeni szünet
* Játék vége
* Játék irányításának bemutatása

Következő osztály a pálya kiválasztásáért felelős egység. Jelenlegi tervezet szerint a pályát nem a felhasználó választja ki, hanem egy automata véletlenszerűen adja a pályát. Jelenleg ez egy funkciót tartalmaz, aminek a visszatérési értéke a kisorsolt pálya. Igaz, hogy egyelőre értelmetlennek tűnik egy metódusnak külön osztályt készíteni, de ha a későbbiekben bővíteni kellene a funkciókat, akkor nem kell újratervezni a felépítést.

Még nem volt szó a különböző állapotok közötti grafikus átmenetekről. Jelenleg ha másik állapotba kerül a játék, akkor a képernyő egyből átvált arra a nézetre és nincs semmilyen átmenet. Ez a felhasználói élményt meglehetősen csökkenti. Erre a problémára a következő megoldást terveztem. Egy statikus osztály mely tartalmaz egy funkciót (8.6.1. ábra), ami az átmenetet végrehajtja, illetve két mezőt, ami eltárolja azt hogy éppen van-e átmenet vagy sem és egy számlálót ami jelenleg 254-ig iterál. Az átmenet egyszerű elhalványítás illetve világosítás. Paraméterként referenciaként egy szín objektumot vár melynek az alfa értékét növeli 254-ig.



*8.6.1. ábra – Átmenet funkció szekvencia diagramja*

Az utolsó statikus kisegítő osztálynak a feladata figyelni, hogy a játék először lett e elindítva vagy sem. Ha először lett elindítva, akkor a játékállapot az átvált az irányítást bemutató menüre. Ez az osztály azt az egy funkciót tartalmazza, ami vizsgálja, hogy létrejött e már az a szövegfájl, ami eltárolja a játékos pontszámát, és ha még nem létezik, akkor létrehozza és a belépés utána kezdőoldal a segítő oldal lesz.

A következő megoldandó probléma a játék állapot váltásakor az első frissítési metódus figyelése. Mivel képernyő váltáskor letörlöm a képernyőről az elemeket és másikakat rajzolok ki, ezért szükséges az inicializálás első lefutáskor. Ilyenkor történik például a játékos, a pályaelemek, gombok és pontok újrapozícionálása illetve a változók értékeinek visszaállítása. Erre a megoldás egy logikai változó a játékciklust futtató osztályban mely igaz, ha játékállapot változás történt. Az első ciklus lefutása után ezt az értéket hamisra kell állítani.

# 8.7. Két személyes játék tervezése

Az első eldöntendő kérdés az, hogy melyik hálózati architektúrát alkalmazom. A 6. fejezetben két lehetőséget ismertettem. A kliens-szerver modellhez szükséges egy szerver, amit publikálnom kell, hogy az interneten elérhető legyen. Másik a peer-to-peer architektúra amihez nem szükséges egy központi szerver ezért ezt megfelelően lehet tesztelni publikálás nélkül. Két eszköz szükséges melyek egy hálózatra vannak felcsatlakozva, ezek adottak.

Tehát az előzőeket figyelembe véve a peer-to-peer architektúra került kiválasztásra. Ennek megvalósítása egy újabb komponenst igényel. Szükséges egy osztály, ami kialakítja a kapcsolatot a hálózaton levő eszközök csoportjával, azaz egy olyan komponens mely egy multicast IP címre kiküldi, hogy játékost keres. Egyszerre több mint 2 játékos lehet a hálózaton ezért a specifikációban már ismertetett módon (azaz játékos tudás szerint) kisorsolja a játékos párokat. Miután megtalálták egymást a megfelelő párok, a multicast IP címzést át kell váltani unicast címzésre. Unicast címzés esetén egy kiválasztott IP címre, mely egy eszközt azonosít, küldi az adatokat. Ez biztosítja, hogy idegen adatok nem érkeznek az eszközre és a két eszköz csak is egymással kommunikál. Innentől kezdve, hogy működik a kommunikáció a megjelenítés és az adatfeldolgozás már csak algoritmustól függ. Fontos kérdés, hogy az adatátvitelhez milyen protokollt használok. Az egyik lehetőség a TCP/IP protokoll, mely biztosítja, hogy minden adat megérkezik sorrendhelyesen. Ez az olyan alkalmazásoknál előnyös ahol nem annyira fontos az idő, ez a protokoll nem biztosítja, azt hogy a csomagok valós időben érkeznek. Játék során fontos az, hogy az adatok pontosak időben pontosak legyenek, ne legyen késés. Másik választási lehetőség az UDP protokoll mely pontosan az előzőnek az ellentéte, azaz nem biztos, hogy minden csomag célba ér viszont nincs késleltetés. Döntésem az UDP protokollra esett, mert fontos, hogy a játékosok az ellenfelükről való információkat valós időben lássák, illetve ha elveszik néhány csomag az nem gond, mert az üzenetküldés folyamat alapú, másodpercenként sok csomag folyik át a csatornán.

Hálózaton keresztül byte típusú tömböt lehet küldeni, összetett típusú objektumokat nem, ezért szükséges meghatározni egy konvenciót, hogy a feldolgozás hogyan történik. Lehet egy olyan megoldás miszerint a jelentést a tömb különböző indexű elemei határozzák meg (pl. az első index az x irányban való pozícióját jelenti, a második indexű elem pedig az y tengelyen való pozícióját stb.). Ez a megoldás jelen esetben nem biztos, hogy elegendő, mert lehet egy érték nagyobb, mint 254. Ha csak a koordináták lehetséges értékeit nézzük az is 480 és 800 értékek közt szerepel. Emiatt másik megoldást kellet keresnem, méghozzá, hogy a byte tömböt karakterlánccá alakítom és így dolgozom fel. Ahhoz, hogy az értékeket külön lehessen választani egy szeparátort kell beillesztenem ami a vessző karakter lett.

Innentől kezdve, hogy rendelkezésre áll a technológia, már alakítható az ehhez tartozó osztályok struktúrája és hierarchiája. Szükséges egy osztály mely a kapcsolatért felelős. Ebben szerepelnek azok a metódusok, melyek felelnek a kapcsolat megnyitásáért, kapcsolat zárásáért, adatok fogadásáért és az adatok küldéséért. Mint már említettem az előző bekezdésekben két féle kapcsolat van, az egyik a multicast, másik pedig az unicast. Eldöntendő volt az komponensek kialakítása során, hogy ezeket egy osztályban vagy külön-külön valósítom meg, végül az egy osztályban való elhelyezés mellett döntöttem, mert nem szükséges bonyolítani ennyi funkció esetén (8.7.1. ábra).



*8.7.1. ábra – Kapcsolatért felelős osztály osztálydiagramja*

A következő osztály a kapcsolat építésének, tartásának és bontásának a logikáját tartalmazza (8.7.2. ábra). Amikor a játékos kiválasztja ezt a módot, akkor először keresni kell másik játékost. Ez a már említett multicast címzéssel hajtódik végre. Egy olyan csomagot küld ki minden frissítése ciklusban, ami egy „0” karaktert tartalmaz, ezzel jelzi, hogy nem talált még párt. Ha nincsen a hálózaton senki, akkor nem talál párt tehát szükséges meghatározni, hogy maximum hányszor küldhet ki egy ilyen párkereső csomagot. Ha meg találta a párt, akkor onnantól kezdve a játékállapot átvált „NetworkGaming” állapotra, lezárja a multicast címzést és megnyitja az unicast címzést a beérkezett eszköz IP címére. A csomag adat része a következőképpen tevődik össze: „X,Y,sebesség,pontszám”. Másik fontos metódus a bejövő üzenet feldolgozása. Egy karakterlánc típusú üzenetet kap, majd ezt szétválasztja a megadott szeparátor alapján. Itt kerül elő a következő osztály, ami a másik játékos mezőit is funkcióit tartalmazza.



*8.7.2. ábra – Kapcsolat logikájáért felelős osztály osztálydiagramja*

A megkapott adatokat szétválogatja a feldolgozó metódus és egy „OtherPlayer” típusú objektumba foglalja bele (8.7.3. ábra). Ez az osztály tartalmazza a feldolgozandó adatokat, azaz a pozíciót, pontszámot és a sebességet. Ezek mellett még megvalósítja a kirajzoló és a betöltő metódusokat.



*8.7.3. ábra – Másik játékos osztályának osztálydiagramja*

# 8.8. Összefoglalás

Az osztályok közötti kapcsolatok a tervezés során létrejöttek. Implementálás előtt szükséges átgondolni a komponensek névtérben való elhelyezését, amit a 8.7.1. ábra mutat be.

*8.8.1. ábra – Névterek kapcsolata*

A tervezés során úgynevezett „Proof of concept”(POC) technikát alkalmaztam. A jelentése, hogy egy-egy kérdéses rész valóban megvalósítható lesz-e az implementálás, prototípuskészítés. Ezzel a módszerrel kiküszöbölhető, hogy az implementálás során tervezést módosítani kelljen, mert időben kiderül, ha egy koncepció nem lesz a későbbiekben megvalósítható.

9. A játék implementálása

A fejlesztés során Team Foundation Server-t használok, a biztonság és a verziózás miatt. Ez egy Microsoft termék, melyet főleg fejlesztő cégek használnak a csapatmunka biztosításáért. Otthoni (korlátozott a csapattagok száma stb.) használatra ingyenes. Az interneten létre lehet hozni egy saját szervert mely bárhonnan elérhető, verziózza a fájlokat, így ha egy jó megoldás módosítva lett az könnyen visszakereshető és visszaállítható.

# 9.1. Kiemelt megvalósítások

Nem szeretném a megvalósítás minden egyes lépését bemutatni ezért következőekben kiválasztottam néhány olyan funkciót mely során a legtöbb probléma merült fel vagy hosszabb átgondolást igényelt.

## 9.1.1. Pálya feldolgozás és kirajzolás

Ez a metódus az összes, olyan osztályban kell, hogy szerepeljen, mely a pálya elemeit dolgozza fel. Tehát szerepelnie kell az akadály osztályában, a pontot érő elem osztályában és a játékot módosító elem osztályában. Mindegyik előbb említett komponensnek meg kell valósítania, de az algoritmus esetenként különböző, ezért ezt a funkciót egy absztrakt osztályban helyeztem el, absztrakt metódusként.

A tervezésben már bemutatott konvenciók adottak (9.1.1.1. ábra), tehát:

* Akadály: szám, 0 - 9-ig
* Pontot érő elem: karakter, a - z-ig
* Játékot módosító elem: ’\*’ karakter
* Pálya vége: ’!’ karakter

*9.1.1.1. ábra – Pálya leírását tartalmazó fájl tartalma*

Jól látszódik, hogy a különböző típusú indexelések miatt szükséges más-más feldolgozó algoritmus. Példaként vegyük az akadály feldolgozó metódust (9.1.1.2. ábra). Az első sorban látható, hogy definiálom a „BarrierList” mezőt, amit az osztályon belül már deklaráltam. Ez a lista „Barrier” típusú elemeket tartalmaz, mely osztályban megtalálhatóak azok a funkciók és tulajdonságok melyek egy akadályra vonatkoznak. A következő sorban levő deklarálás, egy lista, ami karakterlánc típusú elemeket tartalmaz. Ahogy a nevéből is látszik ez a 9.1.1.1. ábrán levő fájl tartalmának egy-egy sorát tartalmazza. A következő egység megnyitja fájl stream-et a „LevelName” alapján, ami a pályát tartalmazó fájl neve. Ezeknek a fájloknak a kiterjesztése „.level” a megkülönböztetés végett. Itt használom a „using” kulcsszót, ami azt jelenti, hogy a benne létrehozott objektumnak (jelen esetben a „stream” és a „reader”) a kódblokk végén meghívja a „Dispose()” metódusát ami felszabadítja az erőforrásokat, ha megvalósítja „IDosposible” nevű interfészt. Általában fájloknál, stream-eknél illetve, olyan osztályoknál szokás használni ahol szükséges megnyitni és lezárni a kapcsolatot vagy bármi mást. Ennek a blokknak a belsejében kiolvassa a fájl tartalmát soronként, mindaddig, amíg el nem éri a sorok az utolsó sort azaz „null” érték nem lesz. Egy-egy sor az egy-egy elem a „rows” listában.

*9.1.1.2. ábra – Pályafeldolgozás implementációja*

A következő egység az előbb feltöltött listát dolgozza fel és tölti fel a „BarrierList” listát. Az első ciklus a sorokon iterál végig, a második ciklus a sorban levő karakterlánc karakterein iterál. Akadály esetén az index azonosítója szám, ahogy ez már a tervezésben eldőlt ezért szükség van az „int.TryParse()” metódusra mely, ha a karaktert egész számmá tudja konvertálni, akkor igaz eredményt ad, ha nem tud konvertálni, akkor pedig hamis eredményt ad. Jelen esetben ez a feltétel dönti el, hogy az éppen vizsgált elem akadály-e vagy sem. Ha a feltétel igaz, akkor, azt még egy elágazás követi mely meghatározza, hogy melyik textúrát kell betölteni a „Barrier” objektumba. Ez a megoldás a későbbiekben könnyen bővíthető plusz akadályokkal, illetve a pálya a szerkezete probléma nélkül alakítható.

A pálya feldolgozása és betöltésnek nagy időigény lehet, ezért szükséges egy olyan játékállapot mely a feldolgozást reprezentálja. Ez alatt az idő alatt egy fekete képernyőt látna a felhasználó, ha nem aszinkron történik a feldolgozás, tehát a pálya betöltés egy másik szálon dolgozza fel a rendszer mely a 9.1.1.3. ábrán látszik.

*9.1.1.3. ábra – Pálya betöltésének implementációja*

## 9.1.2. Repülő mozgásának megvalósítása

Következő érdekesség megvalósítás szempontjából a repülő fizikájának és irányításának a kialakítása. A következő állapotok lehetségesek: repülés, esés, egy magasságban repülés. Ezeket az állapotokat egy felsorolás típusú változóban tárolom melynek a neve „PlayerState”. Fizika kialakításához a következő tényezők szükségesek: maximum sebesség, a felfelé szállás sebessége, gyorsulása és útja illetve a lefelé szállás sebessége, gyorsulása és útja (9.1.2.1. ábra). A változók azért lettek privát láthatósággal felruházva, mert ezeket csak is a benne levő metódusok módosíthatják.

*9.1.2.1. ábra – Repüléshez szükséges fizikai tényezők*

Az idő előrehaladása folyamán a repülő gyorsul felszálláskor és leszálláskor. A két féle gyorsulás egyforma az irányíthatóság miatt, így ha egyforma időközönként váltakozik a le és –felszállás akkor a repülő egy magasságban tartható. A különböző állapotokra külön metódusokat készítettem melyek a repülő mozgását alakítják, tehát külön van „Fly()”, „Fall()”, „InOneAltitude()”.

*9.1.2.2. ábra – Esés metódus implementációja*

Az esésnek a megvalósítása a 9.1.2.2. ábrán látható. Az első sorban megvizsgálja, hogy a sebességes elérte e már a maximum sebességet és csak akkor növeli, ha az kisebb. Az megtett utat, mindig a sebesség értékével növeli. Ezután újrapozícionálja a repülőt, azaz az y tengelyen való elhelyezkedése az előbb növelt megtett út. A következő sorban az előbb említett értéket átadja annak a váltózónak, ami a repüléshez tárolja a y tengelyen való pozíciót, illetve a repüléshez kapcsolódó sebességet nullázza. Ez azért szükséges, hogy ha mozgás irányváltás történik, akkor azok a megfelelő kezdőértékkel induljanak. A következő feltétel azt vizsgálja, hogy elérte a képernyő alját. Jelen esetben, ha ez megtörténik, akkor irányt vált, azaz visszaverődik. A „\_fall\_sy > ZeroPosition.Y – 12” azért szükséges, hogy a repülő ne hagyja el a képernyőt. Mivel a gyorsulás tulajdonképpen azt jelenti, hogy frissítéskor hány pixelt ugrik ezért nem biztos, hogy a képernyőn maradna, ha ezt a feltételt kihagynám mivel akár 8 pixel/frissítésre is felgyorsulhat. Az utolsó sorban meghívja az „Animate()” metódust hívja, mely a repülő animálásáért felelős (lásd később).

A repülést végrehajtó metódus nagyon hasonló az előzőhöz, tulajdonképpen az ellentétje (9.1.2.3. ábra).

*9.1.2.3. ábra – Repülő metódus implementációja*

Az első lépésben megvizsgálja, hogy elérte e már a képernyő szélét, ha elérte, akkor a játékos státusza megváltozik esésre. Ha nem érte el, akkor a következő feltételben azt vizsgálja, hogy elérte e már a maximum sebességet, ha még nem akkor növeli a sebességet. Következő lépésben az új pozícióhoz szükséges y koordinátát csökkenti a sebesség értékével, majd meghatározza a repülő új pozícióját. Ez után az előzőhöz hasonlóan beállítja az eséshez szükséges kezdőértékeket. Legutolsó parancsként pedig meghívja az animáló metódust.

A következő metódus a repülőt egy magasságban tartja (9.1.2.4. ábra). Mindössze annyit csinál, hogy meghívja az animáló metódust.

*9.1.2.4. ábra – Magasságot tartó metódus implementációja*

A karakteranimáló metódus feladata a „belső” animáció azaz, logika szempontjából nem fontos, viszont a játékérzet dinamikusabb lesz tőle (9.1.2.5. ábra). Két paramétere az a szokásos rajzoláshoz szükséges általános paraméterek a „moveCount” egész szám típusú paraméter pedig az animációt alkotó képek számát adja meg, melyet a példányosításkor szükséges megadni. Az „UpdateRectangle()” funkció frissíti a karakter által közrefogott négyszöget. Ez azért szükséges, mert a textúra tartalmazza az animáció minden lépését ezért ennek a területe nem a repülő valódi területét írja le. A „DrawHistory()” metódus a repülő útját mutatja.

A következő parancs végzi a kirajzolást. Lényege a harmadik paraméter, mely a képnek egy részét rajzolja ki melyben meg kell adni a kezdő x és y koordinátákat illetve a szélességet és a magasságot. A kezdő x koordináta az egy képkockának a szélessége az animáció lépéseinek számlálójának a szorzata. Az y koordináta mindig 0, mert a lépések egymás mellett helyezkednek el a betöltött képen, szélessége az előbb említett képkockának a szélessége, magassága pedig a textúra magassága az előbb említett ok miatt. A következő feltétel azt vizsgálja, hogy a lépésszámláló elérte e már a lépések számát, ha nem akkor eggyel növeli az értéket, ha elérte, akkor pedig nullázza, hogy elölről kezdődjön a számlálás.

*9.1.2.5. ábra – Animáció implementációja*

A következő ehhez a komponenshez tartozó metódus a bejövő felhasználói bemeneteknek a feldolgozása a játékmenetben. A specifikációban egy féle irányítást említettem, de a fejlesztés során találtam egy másik módot a repülő irányítására. Az első módszer az, hogy egy érintés esetén, ha repül, akkor leesik, ha esik, akkor pedig felrepül (9.1.2.6. ábra). Ebben az esetben a Windows Phone API beépített gesztusait használtam. A használt gesztusokat külön engedélyezni kell inicializáláskor. Az irányításhoz egyedül a „Tap”-et kell használni.

*9.1.2.6. ábra – Tapintással való irányítás implementációja*

Első lépésben be kell lépni egy ciklusba mindaddig, amíg van felhasználói bemenet. Majd ezeket a jeleket ki olvassa a „TouchPanel” objektumból melyeket eltárolom. A következő feltételben vizsgálja, hogy a típusa az tapintás volt e, és a jelenlegi játékos státusz az esés vagy egyhelyben repülés, igaz esetben átváltja a játékos státuszát repülésre. Ellenkező esetben pedig megvizsgálja, hogy tapintás típusú felhasználó bemenet érkezett e és a repülő státusza repülés, ha igaz, akkor átvált esés státuszra.

Ez a megoldás sok egymás utáni hirtelen érintést igényelt és ez felhasználói szempontból nem kényelmes. Ezért kipróbáltam a következő módszert mely végül a végleges lett. Ez pedig a következő: ha a felhasználó hozzáér a képernyőhöz, akkor repül, ha pedig nem akkor esik (9.1.2.7. ábra).

*9.1.2.7. ábra – Érintéstartással való irányítás implementációja*

Ebben a megoldásban nem használom az előre megírt Windows Phone gesztusokat. Próbáltam a „Hold” gesztust felhasználni erre a célra, de az csak egy idő vált a tartás állapotra, először úgy kezeli, mintha egyszeri érintés lenne, majd ha elért egy időhatárt, akkor vált át „Hold” állapotra. Első lépésben kiolvassa az érintéseket, azért lista nem pedig egy elemet tartalmazó változó, mert egy frissítés alatt több interakció is bekövetkezhet. Második lépésben végig iterál az előbb begyűjtött érintéseket és vizsgálja ezeket. Ha érintés történt, akkor megvizsgálja, hogy nyomás történt vagy képernyő elengedés.

## 9.1.3. Gombkezelő komponens implementációja

Mint már említettem a tervezés fázisnál ennél a komponensnél fontos, hogy könnyen bővíthető legyen, ezért a tervezett osztályokat megvalósítottam. Kiemelném azt a funkciót mely az „események” lefutásáért. Azért tettem idézőjelbe, mert nem a klasszikus eseménykezelésről van szó, mikor eseményekre tudunk feliratkozatni metódusokat. Jelen esetben egy „switch-case” szerkezetet kell bővíteni, ha új gombot implementálunk (9.1.3.1. ábra).

*9.1.3.1. ábra – Gombok kezelésének implementációja*

Paraméterként vár egy listát, amiben az érintések vannak elmentve, a játék aktuális állapotát, illetve referenciaként egy bizonyos „firstEntry” változót mely a főosztályban (melyben fut a játékciklus) van definiálva. Az ebben tárolt érték jelzi az első belépést, ha játékállapot változás történik. Első lépésben megvizsgálja, hogy van e egyáltalán gomb implementálva. Majd egy ciklussal végigiterál a paraméterben megkapott érintéslista elemein. Ebbe a ciklusba van ágyazva még egy, mely végig lépked a gomblista elemein. Első lépésben beállítja a gombelemen, hogy történt e lenyomás (9.1.3.2. ábra).

*9.1.3.2. ábra – Gomb lenyomás vizsgálatának implementációja*

Ez a művelet a kirajzolást befolyásolja, felhasználó szempontból fontos, hogy legyen visszajelzés, hogy a gomb le van e nyomva vagy sem (9.1.3.3. ábra).

*9.1.3.3. ábra – Gomb kirajzolásának implementációja*

Majd a cikluson belül következik egy feltétel vizsgálat mely akkor igaz, ha a gomb lista elemén történt érintés (9.1.3.4. ábra).

*9.1.3.3. ábra – Gomb érintésvizsgálatának implementációja*

Ha a feltétel igaz, akkor következő lépésként belép egy „switch-case” szerkezetbe, mely a megfelelő gombhoz tartozó parancsokat hajtja végre. Itt szükséges beállítani a „firstEntry” változót és az átmenethez szükséges „IsTransition” és „TransitionCounter” mezőket. Az ábrán mindössze két gomb látható, hogy ne foglaljon feleslegesen sok helyet, de itt szükséges hozzáadni az összes gombot.

## 9.1.4. Hálózaton keresztüli játék implementációja

A komponens alapja a Microsoft által ajánlott megvalósítás, mely az „UdpAnySourceMulticastChannel.cs” nevű osztályon alapul. Ez felelős a multicast kapcsolat kiépítéséért, üzenetküldésért és a kapcsolat bontásáért is. Ezt a megoldást kicsit átalakítva alkalmaztam mely a 9.1.4.1. ábrán látható privát mezőket és konstruktorokat tartalmazza. A „224.109.108.107” egy multicast IP cím, mely „224.0.0.0” és a „239.255.255.255” tartományok között szerepelhet. A konstruktorokban létrejön egy „UdpAnySourceMultiCastClient” típusú objektum mely a hálózaton kommunikál.

*9.1.4.1. ábra – Kapcsolatot felépítő osztály konstruktorai és privát mezői*

Az üzenet küldéséhez és fogadásához először szükséges megnyitni a kapcsolatot, mely a 9.1.4.2. ábrán látható. Első lépésben megvizsgálja, hogy meg van e nyitva már a kapcsolat és csak akkor nyitja meg, ha ez az érték hamis. Következő lépésben hajtódik végre a tényleges kapcsolódás, mely aszinkron módon történik. Az „EndJoinGroup()” metódus befejezi a kapcsolódás, attól függően, hogy mi az kapcsolódás eredménye. Ezután a „GroupIsJoined” mezőt igazra állítja és a kliens a „MulticastLoopback” tulajdonságát hamisra, ami azt jelenti, hogy saját magának nem küldi az üzenetet a kliens.

*9.1.4.2. ábra – Kapcsolat megnyitásának implementációja*

A sorra következő metódus az üzenetküldésért felelős (9.1.4.3. ábra). Paraméterként egy karakterláncot vár mely az üzenetet reprezentálja. Második paraméternek megadhatunk 0 vagy több paramétert, melyet a „string.Format()” dolgoz fel. Az első sorban megint egy vizsgálat jön, mely vizsgálja, hogy a kapcsolat nyitott e, ha igen akkor tovább lép. Majd paraméterként átadott karakterláncot átalakítja byte tömbbé melyet már képes kliens objektum, küldő metódusa továbbítani. Hasonlóan az előző metódushoz ez is aszinkron (ez igaz lesz a továbbiakban tárgyalt kliens funkciókra is). Első paramétere a puffert határozza meg, második az eltolást, a harmadik az üzenet hosszát, az utolsó pedig egy „AsyncCallback” delegáltat mely akkor hajtódik végre, ha befejeződött a küldés, jelen esetben ezt lambda kifejezéssel oldottam meg, mert kódsor szempontjából rövid funkciót valósít meg.

*9.1.4.2. ábra – Üzenetküldés implementációja*

Ha üzenetküldésre szükség van, akkor üzenet fogadására is melyet a 9.1.4.3. ábra mutat. Első lépésben megint csak megvizsgálja, hogy nyitott-e a kapcsolat. Majd kiüríti, fogadott üzenet puffert, hogy ne tároljon téves adatokat. Hasonlóan az előző metódushoz itt is első három paraméterként a puffert, az eltolást és a puffer hosszát kell megadni. Negyedik paraméter egy callback metódus, mely meghívja az „EndRecieveFromGroup()” metódust, ha befejőzött az adatátvitel. Ennek első paramétere a metódus eredménye, második paramétere pedig egy referencia, ami tartalmazza a küldő IP címét és portját, melyet egy „IPEndPoint” típusú objektumban tárol el. Következő lépésben meghívja a „GroupOnRecive()” funkciót mely egy eseményt generál és meghívja az üzenet érkezéséhez rendelt funkciót. Következő lépésben meghívja önmagát a metódus, azaz rekurzív. A kilépés akkor történik meg, ha kapcsolat már nem áll fent a hálózattal.

*9.1.4.3. ábra – Üzenet fogadásának implementációja*

Utolsó rövid, de fontos funkció a kapcsolat bontása (9.1.4.4. ábra). Ez mindössze a „GroupIsJoined” mezőt hamisra állítja és meghívja az osztály „Dispose()” metódusát, amit akkor szokás használni, ha a struktúra kapcsolatot tart fent egy külső forrással és szükség van erőforrás felszabadításra.

*9.1.4.4. ábra – Kapcsolat bontásának implementációja*

A fent említett metódusokkal egy csatornát lehet kialakítani melyen keresztül, a kliens kommunikál a többivel. A tervezés fázisban még úgy volt kialakítva ez a struktúra, hogy szerepelt benne unicast kapcsolat is. Az implementáció során problémákba ütköztem az UDP protokoll és az aszinkron működés miatt. Nehezen volt megtalálható és javítható a hiba. A jelenség az volt, hogy az üzenetek nem mindig értek célba ezért, a kliens nem tudta kiépíteni az unicast kapcsolatot. Volt, hogy az egyik eszköz már unicast üzenetet küldött a megtalált eszköznek, de a másik eszköz még multicast üzeneteket küldött és nem tudta felépíteni az 1-1 kapcsolatot. Ezért másik megoldás után kutattam, mely végül a következő lett: csak multicast üzenetküldés történik, és az üzenetben jelzem, hogy talált e párt az eszköz vagy sem illetve az eszközökön eltárolom a párjának a GUID azonosítóját. Ez a megoldás a játékos osztály bővítését idézte elő egy azonosítóval, ami az előbb említett GUID típusú. A struktúra a Microsoft implementációja, egy egyedi azonosítót generál a létrejött azonosítók tudta nélkül, mert olyan nagy a lehetőségek száma, hogy gyakorlatilag nulla az esélye, hogy két ugyanolyan ID generálódik.

Az ellenfél kezelésére egy új osztályt kellet létrehozni, mely az üzenet feldolgozása során eltárolja az adatokat egy objektumban. Ennek kezelésére szükséges a fogadó eseményhez rendelni egy metódust (9.1.4.5. ábra). Az üzenet karakterlánc típusú és a metódus paramétereként érkezik. Attól függően, hogy az üzenet milyen hosszú és milyen elemeket tartalmaz végrehajtódik az objektumba foglalás. Ha az ellenfél azonosítója még nem rögzített, akkor az azt jelenti, hogy nincsen ellenfél, tehát ha üzenet érkezik, akkor annak az első eleme mindig az eszköz GUID azonosítója.

A következő lépésben a pályák szinkronizálása következik. Az algoritmus egyszerű, ha összehasonlításkor a visszatérési érték egy, akkor az ellenfél által küldött pálya lesz a választott ellenkező esetben pedig a küldött pálya. Felmerülhet olyan eset, hogy az eredmény 0, azaz megegyeznek az azonosítók, de ezt, mint már említettem matematikailag lehetetlen. Látható, hogy a második elemét a tömbnek vizsgálom, mindkettő feltételben. Ez a változó jelzi, hogy a küldő eszköz talált e már ellenfelet, ha az értéke „0” akkor nem, ha pedig „1” akkor igen. Ha létrejött a kapcsolat, akkor az ellenfélt eltároló objektum értékeit beállítja.

*9.1.4.5. ábra – Üzenet érkezését feldolgozó metódus implementációja*

A kapcsolatkereséshez külön játékállapotot rendelek. Mindaddig, amíg ebben a státuszban fut a játék, üzenetet küld az eszköz mely a 9.1.4.6. ábrán látható. Ha talált párt és a pálya is ki lett választva, akkor játékállapotot vált. Azért kell vizsgálni a pályát tároló változónak az értékét, mert az üzenetérkezést feldolgozó metódus aszinkron ezért lehet, hogy már párt talált viszont a pályakiválasztás még nem történt meg.

*9.1.4.6. ábra – Ellenfélkeresés implementációja*

Ha a játék elkezdődött a játékosok között, akkor az üzenet a következőképpen alakul:

10. A játék tesztelése

Megvalósítás közben és után lényeges fejlesztési szakasz a tesztelés. A folyamathoz a Microsoft biztosít egy Windows Phone emulátort mely a Windows Phone SDK-val települ. Ez az eszköz hasznos, ha nincsen biztosítva Windows Phone operációs rendszerrel felruházott telefon kéznél. Az emulátor nagymértékben hasonlít a valódi operációs rendszerre és jól szimulálja azt. Játékfejlesztés szempontjából nem tökéletesen tesztelhető ezzel, mert az emulátor teljesítménye nagymértékben függ a futtató számítógép teljesítményétől. Jelen esetben az algoritmusok tesztelésére használtam, ha nem volt kéznél telefon.

A teszteléshez egy Samsung Omnia W típusú Windows Phone és Nokia Lumia 800 készülékeket használtam, melyeken Windows Phone 7.5 operációs rendszer fut. Ahhoz, hogy futtatni tudjam a saját még nem kiadott alkalmazásomat regisztráltatni kellett, mint fejlesztői készülék. Egy regisztráció egy évig érvényes. Sokat segített a fejlesztés közben, hogy valós eszközökön tudtam tesztelni az alkalmazást, így valós eredményeket kapva.

Az alkalmazást folyamatosan teszteltem, minden komponens elkészülte után, egy hosszabb tesztelési folyamat következett, általában hetente egy-egy rész készült el és ezt teszteltem a következő rész elkészültéig. A részeket a következőképpen osztottam fel:

* Menü komponens
* Háttér animálása
* Repülő irányítása
* Pálya feldolgozó és kirajzoló komponens
* Ütközéskezelés
* Befolyásoló elemek
* Kapcsolat kiépítése két játékos módban
* Játékmenet két játékos módban
* Játék vége és eredmények kiértékelése két személyes módban

Komponensek elkészítése után mással is teszteltettem a játékot, olyan személyekkel, akik nem informatikával foglalkoznak. Fejlesztői szemmel a tesztelés teljesen más, nem veszi észre az olyan hibákat melyekre nem készült fel, sokszor csak az általános használati eseteket próbálja ki, vagy csak azokat melyekre a rendszer fel van készítve, ezért nem veszi észre hibákat.

Ilyen észrevétel volt a repülő irányítása. A tervezés során úgy gondoltam, hogy úgy lesz jól irányítható, ha egy tapintásra repül, és ha még egyszer megérintik a képernyőt, akkor pedig esik. Első tesztelésnél észrevettem, hogy ez nem megszokott irányítás, mert ösztönösen, akikkel teszteltettem a játékot, azt várták, hogy amíg hozzáérnek a képernyőhöz akkor repül, ha elengedik akkor pedig esik. Fejlesztői szempontból ez kis változtatást igényelt viszont a játék sikerességét véleményem szerint növeli.

Másik ilyen nem programozást érintő hibák, a játék logikájával kapcsolatosak. Az implementálás kezdetekor, úgy volt tervezve, hogy ha a repülő eléri a képernyő alját, akkor az onnan visszaverődik. Ebből kifolyólag úgy is mozog a repülő, ha a játékos nem avatkozik bele az irányításába. Repül addig, amíg nem éri el a képernyő tetejét majd esik, amíg nem éri el a képernyő alját. Több pályát is teszteltem anélkül, hogy beleavatkoztam volna az irányításba és így is átlagosan a felszedhetők pontok felét össze lehetett szedni. A probléma megoldása az lett, hogy a képernyő aljáról nem verődik vissza, hanem ott egy magasságban repül.

A hálózaton keresztüli játék tesztelése volt a legnehezebb és a leghosszadalmasabb. Mivel két eszköz szükséges mindenképp ezért minden teszteléskor mindegyik eszközre telepíteni kellet az alkalmazás legfrissebb verzióját. Ha ehhez két telefont használok, első lépésben fordítok az egyikre, majd azt leválasztom és ugyanezt a műveletet elvégzem a másikon is. Hibakeresés szempontjából nehézkes mert ez egyszerre csak az egyiken lehetséges. Keresnem kellett egy gyorsabb megoldást, ez a pedig az, hogy nem két telefont használok, hanem az egyik helyett emulátorral dolgozok, így gyorsabb volt a tesztelés. Miután végeztem, egy-egy részkomponenssel, teszteltem két telefonon. Mint már említettem az emulátor nem képes teljesen ugyanazt produkálni, mint egy valódi eszköz ezért sokszor ami működött egy telefon és emulátor között az hibára futott két telefon között. Többek között ezzel a problémával szembesültem a kapcsolat kialakításakor, mert a hálózati műveletek aszinkronok. Az emulátor lassabban építette ki a kapcsolatot ezért nem volt szükség szinkronizálni, viszont a valódi eszközök túl gyorsan hajtották végre ezt a műveletet és nem mindig kerültek szinkronba ezért ezt kezelnem kellett.

Minden komponens elkészülése után lefuttattam egy kódanalízist a Microsoft Visual Studio 2012 segítségével. Ez az eszköz review-za a programkódot és olyan ajánlásokat ad melyek a logikát nem befolyásolják viszont átláthatóbb és optimálisabb kódot eredményez.

Részegységenként lefuttattam egy analízist mely a következők szerint osztályoz: karbantarthatóság, ciklomatikus komplexitás, öröklődés mélysége, osztálytartalmazás, kódsorok száma. A fejlesztés befejezésével lefuttattam és következőképpen alakult az eredmény (10.1. ábra).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hierarchia | Karbantarthatósági index | Ciklomatikus komplexitás | Öröklődés mélysége | Osztályok használata | Kódsorok száma |
| ScarFly | 44 | 44 | 2 | 41 | 248 |
| .MyClasses | 78 | 12 | 1 | 8 | 37 |
| .LevelelementClasses | 84 | 150 | 2 | 30 | 176 |
| .MenuClasses | 80 | 43 | 1 | 16 | 44 |
| .Networkclasses | 84 | 88 | 2 | 29 | 139 |
| .PlayerClasses | 86 | 223 | 1 | 52 | 367 |
| Szumma | **81** | **560** | **2** | **84** | **1011** |

*10.1. ábra – Visual Studio 2012 „Code metrics” eredmény*

Az első oszlop a karbantarthatósági index mely értéke 0-100-ig terjed. 20-100-ig az eredményt jelző ikon egy zöld színű négyzet, mely azt jelzi, hogy elfogadható tartományban van. 10-19 értékek között a jelzés egy citromsárga háromszög, mely azt jelenti, hogy mérsékelten karbantartható. 0-9 értékek között pedig egy piros kör jelzi, azt hogy a karbantarthatósági szintje nagyon alacsony. Ez az index nagymértékben függ a ciklomatikus komplexitástól és a kódsorok számától. A legalacsonyabb mért érték a 44 és az összesített index 81, amiből látszik, hogy a karbantarthatósága jó. [7]

A következő oszlop a ciklomatikus komplexitás mely, akkor jó, ha a szám minél kisebb. A program elágazásaiból egy gráfot alakít, mely akkor számít komplexnek ha, ezekből sok van. Jelen esetben ez várhatóan sok mivel a logika nagy része ezen alapul. Ha csak a legegyszerűbbeket vesszük sorra, mint például a játékállapot, a játékos állapotok, ütközésvizsgálat stb. A ciklomatikus komplexitás számítási modellje a következőképpen néz ki: ciklomatikus komplexitás = élek száma – levelek száma + 1. A levelek az elágazás feltételeket reprezentálják, az élek pedig a köztük levő műveleteket. [7] [8]

A az öröklődés mélysége az egyszerűen a hierarchia mélységét mutatja. Ez jelen esetben maximum kettő, amit szükség esetén növelhettem volna (ez alkalmazás függő), de úgy gondoltam, hogy az felesleges túlbonyolítás. [7]

A következő mérőszám a benne használt osztály referenciák felhasználására vonatkozik. Annál jobb minél kisebb ez a szám, ha túl nagy, akkor ott hiba van a tervezésben mert túl sok a belső függőség. Ha ez a szám alacsony, akkor a kód jól újrafelhasználható, mert nem nagymértékben nem függ másik komponenstől. [7]

Ez a szám a sorok száma adja eredményül. Fontos, hogy nem a legépelt sorok száma ez, hanem az IL-ben létrejött sorok száma. Ha ez a szám túl nagy, akkor túl sok feladat van az alkalmazásban és ezt el kell osztani. Ha magas a sorok száma, akkor az nehéz karbantarthatóságot idéz elő. [7]

11. Továbbfejlesztési lehetőségek

A későbbiekben tervezem bővíteni a játékot olyan plusz funkciókkal és komponensekkel melyek a játék hosszát illetve élményét növelik, és olyan kiegészítőket melyekkel a játék továbbfejlesztése egyszerűsödik.

Az egyik ilyen fejlesztési lehetőség a repülő karakter fejlesztése a játékos által. A pontok, melyeket szerez a játék során ne csak a többszemélyes módban játsszon kategorizálási szerepet, hanem a pontokért különböző kiegészítőket lehessen vásárolni melyek erősítik a játékost. Ezáltal még egy cél keletkezik, így a felhasználónak érdeke többet játszani, mert látványos fejlődést tud véghezvinni. A megoldás során már komolyabb adattárolás szükséges mind a játékosról, mind a fejlesztő elemekről. Szükséges lehet XML formátumban vagy relációs adatbázisban tárolni ezeket az információkat.

Szükséges lehet még további játékmenetet módosító elemekre, mint például az irányítást nehezítő (véletlenszerű gyorsítás lassítás, ellenkező irányítás). A felhasználói élményt változatossá és színesebbé teszik. Ehhez a már meglévő osztályt kell kibővítenem a megfelelő metódusokkal, tehát ennek a bővítésére elő van készítve a program

Jelenleg a pályákat véletlenszerűen sorsolja ki, ezeket be lehetne sorolni nehézségi szintekbe illetve a pályákat elválasztani témák szerint. A véletlenszerű sorsolás mellett lehessen választani pályát. Egy sorrendet kellene felállítani a pályák között és csak akkor léphet a következő pályára, ha az előzőeket teljesítette egy meghatározott ponthatár felett. Valószínűleg ehhez is szükséges lesz egy relációs adatbázisra, hogy melyik pályához milyen ponthatár tartozik illetve, eltárolni azt, hogy melyeket teljesítette. Ponthatár mellett még különböző rangokat is meglehetne határozni melyeknek a gyűjtése után plusz pontot vagy a karakter fejlődését segítő elemet kap.

Irányítást is jobban kontrolálhatóvá tenném, azaz a felhasználó nagyobb mértékben tudná befolyásolni a repülő mozgását. Jelenleg gyorsulást nem tud befolyásolni csak azt, hogy repüljön vagy sem. A gyorsulást lehetne változtatni az ujj mozgásának sebességétől függően. Ha gyorsan húzza felfele a képernyőn az ujját, akkor gyorsul, ha lassan akkor lassul illetve, ha egy helyben érinti meg akkor pedig egy beprogramozott gyorsulást venne fel. Ez a továbbfejlesztés lehetőséget nyújtana a felhasználónak a finom irányításra.

Az eddigi lehetőségek az egyszemélyes játékmódra fókuszáltak. Kétszemélyes módban, nagyban növelné a játékélményt, ha a játékosok egymásra közvetlen hatással lennének. Azaz lenne egymással való ütközés, vagy fel lehetne szedni olyan módosító elemeket a pályán melyek az ellenfél tulajdonságait módosítják.

Növelné a továbbfejlesztés gyorsaságát egy olyan kiegészítő program készítése mellyel pályát lehet építeni. Ez egy asztali alkalmazás lenne, melyben fel van osztva a pálya négy részre és egységenként be lehet szúrni egy-egy pályaelemet. A további pályák készítésekor ugyanazt a képet adja vissza, mint amit a játékos lát. Ez nagymértékben növelné a hatékonyságát, mert nem kellene az összes új pályát kipróbálni az eszközön.

Irodalomjegyzék

[1] Árvai, Z., Fár, A. G., Farkas, B., Fülöp, D., Komjáthy, Sz., Turóczy, A., Velvárt, A.: Windows Phone fejlesztés lépésről lépésre, *Jedlik Oktatási Stúdió Budapest, 2012*

[2] Tom, M., Dean, J.: XNA Game Studio 4.0 Programming, *Addison-Wesley, 2010*

[3] Stephen, C., Pat, M.: Microsoft XNA Game Studio Creator’s Guide Second Edition, *McGraw Hill, 2009*

[4]Ian, S.: Szoftverrendszerek fejlesztése, *Panem Kft., 2007*

[5]Multi-resolution apps for Windows Phone 8, (<http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206974(v=vs.105).aspx>)*., utoljára megtekintve 2013-11-17*

[6] GUID structure, (<http://msdn.microsoft.com/en-us/library/windows/desktop/aa373931(v=vs.85).aspx)>, *utoljára megtekintve 2013-12-10*

[7] Code Metrics Values, (<http://msdn.microsoft.com/en-us/library/bb385914.aspx>), *utoljára megtekintve 2013-12-15*

[8] CA1502: Avoid excessive complexity, (<http://msdn.microsoft.com/en-us/library/ms182212.aspx>), *utoljára megtekintve 2013-12-15*