
**Budapesti Műszaki és Gazdaságtudományi Egyetem**
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

# Házi feladat

# Fejlesztői dokumentáció

## Beágyazott rendszerek szoftvertechnológiája

*Készítették:*
Hornyák Máté Dániel
Molnár Marcell
Poleczki Ákos

2019. május 11.

# Az Android alkalmazás

## 1.1. A feladat rövid összefoglalása

A megvalósított feladat egy kétdimenziós, oldalnézetes lövöldözős játék Android operációs rendszer alatt, mely a következőképpen működik.

Egyszerre két játékos játszik egymás ellen, összesen fejenként három életponttal, websocket alapú kapcsolaton keresztül. A játék célja az ellenfél életpontjainak lenullázása. A játék elindításakor a felhasználó egy felugró menüpontban adhatja meg a csatlakozni kívánt szerver IP címét, illetve lehetősége van ezen kívül az alapvető beállításokat változtatni a játék megkezdése előtt. Tehát a vezérlési módok és az egyéni ízlésnek megfelelő háttér kiválasztására, valamint a háttérzene ki-be kapcsolására. A beállítások elvégzésével és a csatlakozás gomb megnyomásával a felhasználó csatlakozik a megadott szerverhez. A beállításokat az alkalmazás két indítása között is megőrzi. Amennyiben mindkét játékos sikeresen csatlakozott, a játék egy visszaszámlálást követően elkezdődik.

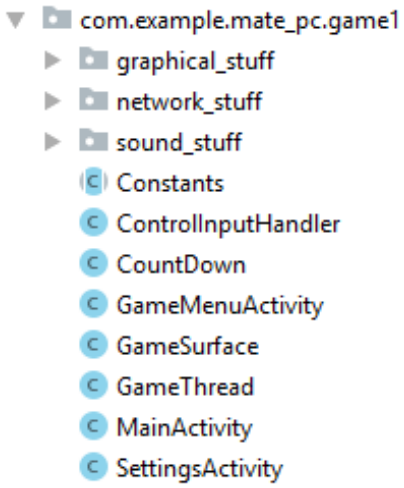
A beállítások alkalmával megadott vezérlő módok alapján vagy gombokkal vagy joystick segítségével van lehetőség a karakter mozgatására, az ellenfél megsebzésére pedig egy dedikált nyomógomb szolgál. A nyomógomb lenyomásakor egy számláló indul el, mely néhány másodpercig blokkolja az újratüzelést, ezzel megakadályozva sorozatos támadást. A játék dinamikájának kiszélesítésének érdekében a blokkoláson kívül akadályok is elhelyezésre kerülnek a pályán, amelyekre akár fel is lehet ugorni a karakterrel.

## 1.2. A program felépítése

Az programot alkotó osztályok az alábbi képen látható struktúra szerint épülnek fel.

A fő package-en belül található:

- a *MainActivity*,
- a beállításokért felelős *SettingsActivity*,
- a főmenüt megjelenítő *GameMenuActivity*,



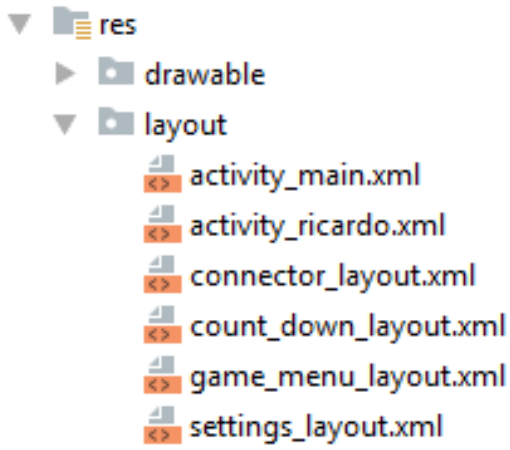

**1.1. ábra.** *Az osztályok struktúrája*

- a felhasználói bemeneteket kezelő $ControlInputHandler$,
- a $GameThread$ osztály, amely egy szálban megfelelő időközönként kirajzoltatja az aktuális nézetet,
- a $GameSurface$ osztály, amely a kirajzolást és a játék logikáját valósítja meg az egyes játék objektumok állapotának frissítésével,
- a $CountDown$ osztály, amely egy lebegő dialógus ($Dialog$) segítségével egy visszaszámlálót valósít meg mielőtt a játék elkezdődne,
- és az absztrakt $Constants$ osztály, amelyben az alkalmazáson belül több helyen használt konstansokat tárolja.

További három package-et találunk a források között, amelyek a grafikus megjelenítés, a hálózati kommunikáció és az audió effektek szempontjából lényegesek:

- a $graphical\_stuff$-ban találhatók a játékosok karaktereit, az akadályokat és a lövedékeket reprezentáló osztály leírások,
- a $network\_stuff$-ban a hálózatkezelésért felelős socket osztály, a program indulásakor használatosa a hálóati kapcsolat kiépítéséért felelős osztály és egy külön szálat futtató osztály található. Ez utóbbi periódikusan küldi át a lényeges információkat a szerver felé.
- a$sound_stuff$-ban a háttérzenét kezelő és a különböző hangeffekteket lejátszó osztályok vannak.

## 1.3. A nézetek




**1.2. ábra.** *Az activity-kben használt layoutok*

Összesen 6 layout van definiálva az alkalmazáshoz. Ebből 5 tartozik valódi "AppCompatActivity" vagy "Activity" osztályhoz. Ezek az osztályok a "Manifest" állományban is megtalálhatóak. A "count_down_layout" a visszaszámláló "CountDown" osztályhoz tartozik, amely egy "Dialog" ablakot helyez a képernyőre. A "count_down_layout" ennek a dialógusnak a nézete.

## 1.4. MainActivity

A program indulásakor a MainActivity-ből fog egy példány létrejönni. Ennek az osztálynak a nézetében (*activity_main*) található

- egy *LinearLayout*, amelyre a játék elemeit rajzoljuk ki,
- egy *LinearLayout*, amely a joystick alapú irányításnál fogja a *TouchEvent*eket "elkapni",
- valamint 4 gomb, amelyek a lövésért és a 3 irányú mozgásért felelősek.

A bemenő eseményeket egy külön osztályban kezeljük, ezért a gombokra és a joystick nézetére olyan *OnTouchListener*t hozunk létre, amely egy függvényhívással továbbadja az eseményeket a *ControlInputHandler* osztálynak.

Létrehozzuk a *GameSurface* osztályt (,amely a *SurfaceView* osztályból származik), és játék felületét megtestesítő *LinearLayout* nézetéhez hozzáadjuk.

Ezután elindítjuk a csatlakozásért felelős nézetet egy *ConnectorClass* formájában, majd várjuk a felhasználói interakciót.

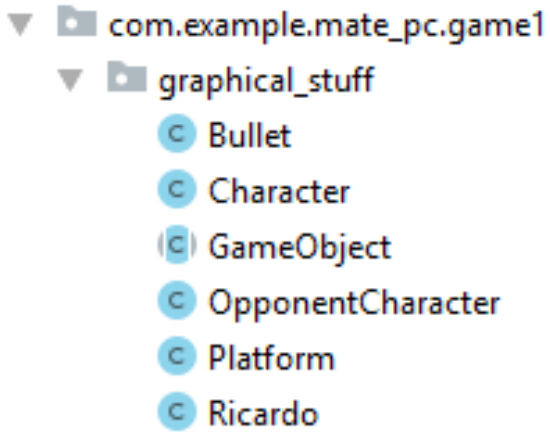
## 1.5. GameSurface

A "GameSurface" osztály elnevezésével ellentétben a játék logikáját is megvalósítja. Ebben az osztályban tároljuk a karaktereket, az akadályokat és minden más játék objektumot. A két legfontosabb függvénye a *public void update*() és a *public void draw*(*Canvas canvas*). Ezek felelősek a játék objektumainak állapotfrissítéséről és arról, hogy ezeket az objektumokat a képernyőre rajzolhassuk.

## 1.6. GameThread

Fontos, hogy a játék logikája és a kirajzoltatás egy külön szálon fut. Ezt a szálat valósítja meg a "GameThread" osztály, amely periodikusan meghívja a "GameSurface" osztály *update*() és *draw*(..) függvényeit.

## 1.7. Grafikai elemek




**1.3. ábra.** *A grafikai elemek*

### 1.7.1. GameObject

A "GameObject" osztály egy absztrakt osztály, amiből a "Bullet", a "Character", az "OpponentCharacter"[1] és a "Platform" osztály leszámrazik. Ez egy olyan osztály, amelyben olyan adatokat tárolunk, amelyek minden, a felsorolt osztályokban megtalálhatók. Ilyen például egy objektum pozíciója, mérete és az általa megjelenített kép.

[1] Az "OpponentCharacter" közvetlenül a "Character" osztályból származik le"

### 1.7.2. Character

Ez az osztály reprezentálja a játékos karakterét. A két legfontosabb függvénye a *public void update*() és a *public void draw*(*Canvas canvas*). Ezeket a "GameSurface" hívja meg a saját *update*() és *draw*(..) függvényeiből. Fontos, hogy a karakter helyzete kizárólag az "update()" függvényben változik meg, mivel az osztály tárolja a karakter sebességét, ami alapján változik a pozíciója. Érdekesség, hogy az ugrás dinamikáját a gravitáció valós modellezésével oldottuk meg. Ezen kívül "getter", "setter" és egyéb segédfüggvények vannak benne.

### 1.7.3. OpponentCharacter

Az "OpponentCharacter" a sima "Character" osztályból származik le. A különbség ott van a két osztály között, hogy az ellenfél karakterének pozícióját a web socketen keresztül kapjuk meg, ezért ennek a karakternek rendelkeznie kell az $x$ és $y$ koordinátária vonatkozó "setter" függvénnyekkel.

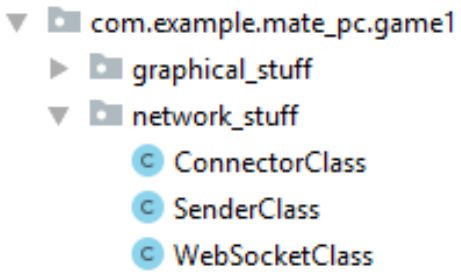
### 1.7.4. Platform

A "Platform" osztály az akadályokat valósítja meg. Ennek egy speciális függvénye a *public boolean isBelow*(*Character character*), amely megmondja, hogy a paraméterként kapott karakter a platform felett van-e. A "GameSurface" ezt minden platformra meghívja és ez alapján dönti el, hogy a karakter meddig tud zuhanni.

### 1.7.5. Ricardo

Ez az osztály egy "Activity", amely győzelem esetén a jutalom videó lejátszásáért felelős.

## 1.8. Hálózati elemek




**1.4. ábra.** *A hálózati elemek*

### 1.8.1. ConnectorClass

Ez az osztály egy "Activity", amely a játék indulásakor az IP-cím megszerzéséért felelős. A nézetében egy szövegdobozba ("EditText") kell megadni az IP-címet. A "MainActivity"-hez tér vissza az osztály, az IP-címet pedig egy "Intent" extrába csomagolva küldi el.

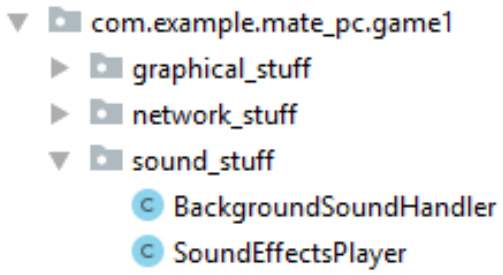
### 1.8.2. SenderClass

A "SenderClass" egy szálat futtat, amelyben periódikusan elküldi a saját játékosunk pozícióját, a másik játékos életét és a még be nem csapódott lövedékek helyzetét. Az ellenfél életének elküldése azért jobb megoldás a találat egyszeri jelzése helyett, mert ha egy hálózati csomag el is vesztődik, a nem jelent gondot, mert a következő csomagban ismét elküldjük ugyanazt az adatot. Ha viszont csak egy találatot küldenénk el, és pont ez a csomag veszne el, akkor a két játékos más és más életpontot látna. Ugyanígy a saját karakterünk sebessége helyett a pozíciót érdemes küldeni, mert így nem kell felesleges szinkronizációt végezni a két eszköz között.

### 1.8.3. WebSocketClass

A "WebSocketClass" a web socketet valósítja meg. Ennek egy belső osztálya a "SocketServerThread", amely egy szálat valósít meg, amelyben a we socket fut. A hálózati adatokat ebben az osztályban létrehozott web socket fogadja.

## 1.9. Hangeffektek




**1.5. ábra.** *A hangokért felelős osztályok*

A hangokért alapvetően két osztály felelős. Az egyik a "BackgroundSoundHandler", amely a háttérzenét szolgáltatja egy MediaPlayeren keresztül. A másik a "SoundEffectsPlayer" a különböző effekteket játsza le, például amikor egy lövedék becsapódik. Ezeket egy "SoundPool" segítségével játszk le.

## 1.10. JavaDoc

Jelen dokumentumhoz egy JavaDoc dokumentum is tartozik. Ez egy HTML, CSS és javascipt alapú dokumentum, amelyet böngészőben érdemes megnyitni és megtekinteni. Ehhez a "\Javadoc\index.html" nevű fájlt kell megnyitni, minden más erről a HTML oldalról érhető el. Ebben minden java osztály és forrásfájl benne van a szokásos java-s dokumentáció szerint.

# A szerver

## 2.1. Felépítés, működés

A szervert a "Ratchet" nevű, PHP alapú könyvtárral valósítottuk meg. Ezzel könnyen és gyorsan fel lehet állítani egy valós időben működő szervert, amelyre a kliensek tudnak csatlakozni. Ebben egy osztályt kell létrehoznunk, amely a "Ratchet"-ben található "MessageComponentInterface" osztályából származik le. Ennek - a java-s web sockethez hasonlóan - 4 alapfüggvénye van:

- $public\ function\ onOpen(ConnectionInterface\ \$conn)$
- $public\ function\ onMessage(ConnectionInterface\ \$from,\ \$msg)$
- $public\ function\ onClose(ConnectionInterface\ \$conn)$
- $public\ function\ onError(ConnectionInterface\ \$conn,\ \backslash Exception\ \$e)$

Ezek közül az *onOpen*(..) függvény hívódik meg amikor egy új kliens kapcsolódik. Ekkor ezt a klienst megjegyezzük és várunk a következő eseményre. Amennyiben üzenetet kapunk, akkor az *onMessage*(..) függvény hívódik meg, amelyben az üzenetet továbbküldjük a másik játékosnak (amennyiben csatlakozott már másik játékos). Ha egy játékos megszakítja a kapcsolatot, akkor az *onClose*(..) függvény hívódik meg és ezért itt eltávolítjuk a megszakított kapcsolatot.

Jelenleg az implementált szerver csupán 2 kapcsolatot tud kezelni. Ezt azonban bármikor ki lehet bővíteni, hogy több kapcsolat kezelésére is képes legyen.