### Győri SZC Jedlik Ányos Gépipari és Informatikai Technikum és Kollégium Szoftverfejlesztő és – tesztelő

### OKJ 5 0613 12 03

S Z A K D O L G O Z A T

**SPACEY**

### Készítette: Scheuer Patrik Ábrahám Domonkos

Konzulens: Horváth Norbert Beadási határidő: 2022. április 29.

......................... .........................

Kottra Richárd Módos Gábor  
 inf. igh. igazgató

SPACEY

Készítette: Scheuer Patrik, Ábrahám Domonkos

Tartalomjegyzék

1. [Bevezetés 4](#_TOC_250031)
2. [Specifikáció 4](#_TOC_250030)
   1. [Szoftver alapvető jellemzői 5](#_TOC_250029)
   2. [Szoftver funkciói 5](#_TOC_250028)
3. [Alkalmazott technológiák 6](#_TOC_250027)
   1. [Felépítése 6](#_TOC_250026)
   2. [Apk felépítése 7](#_TOC_250025)
   3. [C# 8](#_TOC_250024)
   4. [Fejlesztői környezet 9](#_TOC_250023)
4. [Felhasználói dokumentáció 11](#_TOC_250022)
   1. [Telepítés 11](#_TOC_250021)
   2. [Játék főbb jellemző 11](#_TOC_250020)
   3. [Játék kezdése 12](#_TOC_250019)
   4. [Játék 13](#_TOC_250018)
   5. [Játék vége 14](#_TOC_250017)
5. [Fejlesztői dokumentáció 15](#_TOC_250016)
   1. [Első lépések 15](#_TOC_250015)
   2. [Menük 16](#_TOC_250014)
   3. [Játék szerkezete 16](#_TOC_250013)
   4. [Labirintus 19](#_TOC_250012)
   5. [Pálya 22](#_TOC_250011)
   6. [Mozgás 23](#_TOC_250010)
   7. [Ütközés 24](#_TOC_250009)
   8. [Játékos osztály 26](#_TOC_250008)
   9. [Játékos adatok tárolása 27](#_TOC_250007)
   10. [Bájitalok 27](#_TOC_250006)
   11. [Játék indítása NFC-n keresztül 28](#_TOC_250005)
   12. [Tesztelés 30](#_TOC_250004)
6. [Összefoglalás 32](#_TOC_250003)
   1. [Jövőbeli tervek 32](#_TOC_250002)
7. [Felhasznált irodalmak 33](#_TOC_250001)
8. [Melléklet 33](#_TOC_250000)

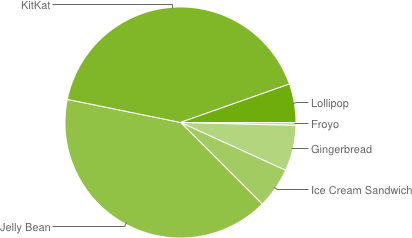
# Bevezetés

Az elmúlt években az űrhajózás annyira közel került ahhoz, hogy civilek is részt vehesenek űrutazásban, hogy mára már közel sem elkpézelhetetlen az eféle űrutazás. Napjaink meghatározó űrkutatási cége a SpaceX áll legközelebb annak megvalósításához, hogy akkár nagyobb mennyiségben civileket küldjön az űrbe. Ennek apropóján jött az ötlet a szakdolgozat témájaként, hogy egy olyan komplex webes alkalmazást készítsünk amely különböző űrutazásokat kínál civileknek.

Az űrutazásra jelentkezők sok információt kaphatnak az adott utazással kapcsolatban. Különböző ismertetőkön keresztül az utazó megismerheti magát az űrhajót és az adott bolygót részletesen. Az utazással kapcsolatos alap információkat is megtalálhatja az oldalon. A mobil applikációban\*

# Specifikáció

A Szakdolgozatunk egy űrutazással foglalkozó webes alkalmazás. A fejlesztés első lépéseként meg kellett határoznunk, hogy mit tudjon a programunk, miben végezük a backend, frontend, az adatbázis fejlesztését. Mielőtt kiválasztottuk az adott fejlesztői környezeteket, az előbb említetekre lebontva felmértük,  
hogy az ötleteinket, elképzeléseinket melyikekben tudnánk tudásunk szerint a lehető legjobb módon megvalósítani. A mobil applikáció fejlesztői környezete már eleve adott volt számunkra. A választások a következő program nyelvekre estek: Backend: NodeJs Frontend: VueJs Adatbázis: MongoDb Mobil applikáció: Xamarin.



## Szoftver alapvető jellemzői

* + - Szoftver elnevezése, címe: SPACEY
    - Szoftver kategóriája: Webes alkalmazás
    - Szoftver alapvető funkciója: Űrutazások vásárlása, kezelése, ismertetése
    - Fejlesztői környezet: Backend: NodeJs Frontend: VueJs Adatbázis: MongoDb Mobil applikáció: Xamarin
    - Programozási nyelvek: C#, JavaScript
    - Platform: Windows

## Szoftver funkciói

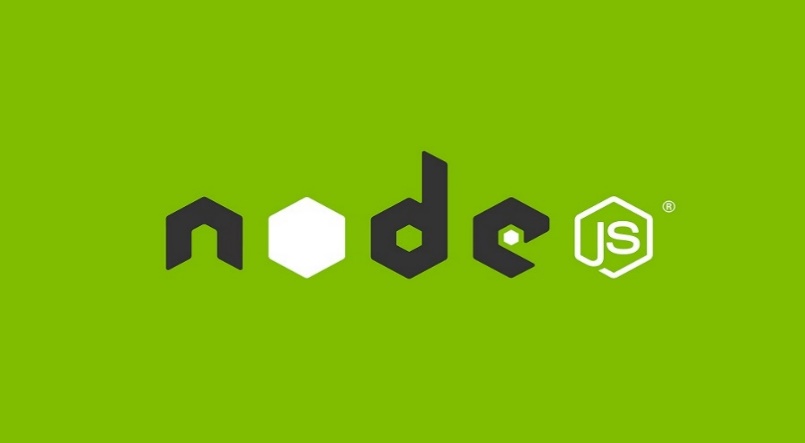
Egyjátékos módú labirintus játék, aminek a fő célja az, hogy megkeresd a sajtot és így mimnél gyorsabban visszatérj az egér várba, hogy megmentsd a hercegnő életét. A pálya véletlenszerűen generált labirintus, aminek csak egy részét látja a játékos így nehezítve a célba jutást. A pályán különböző italok vannak elhelyezve, aminek a hatására falból részek eltűnhetnek vagy visszajöhetnek, lassulhat a játékos, vagy bizonyos esetben átmehet a falon. A játék vezérlését a telefonba épített gyorsulásmérő szenzor végzi. Így a telefon megfelelő irányú bedöntésével lehet irányítani. A játék menüi az Android gyári felületének elemeit használják és egymásra épülő hierarchiát alkotnak. A játék végén megtudhatjuk, hogy mennyi idő alatt vittük végig a játékot.

# Alkalmazott technológiák

# NodeJs:

A Ryan Dahl által 2009-ben létrehozott, Windowson, Linuxon és Mac OS-en is futó, nyelvként JavaScriptet használó Node.js egy olyan, ingyenes szerverkörnyezet, amely a Google V8-as motorjára épül. Platformfüggetlen megoldás, amely tökéletes skálázható online alkalmazások megvalósítására.

Lehetővé teszi a gyors és hatékony JavaScript fejlesztést és a könnyű prototypingot ezzel a leghatékonyabban használható szerveroldali programnyelvek egyike. Egyre népszerűbb, számos nagy cég (Linkedin, Microsoft, Yahoo!, Walmart, PayPal, Netflix…stb.) használja.



VueJs:

A Vue.js egy JavaScript könyvtár. Sok esetben keretrendszernek szokták hívni, pedig valójában nem az: az [Angular, React, Vue](https://gremmedia.hu/edukacio/bejegyzes/angular-react-vagy-vue-melyiket-lehet-erdemes-hasznalnunk) trióból egyedül az Angular minősül frameworknek. 2014-ben jelent meg először, a Google egyik volt fejlesztője készítette eredetileg. Azóta hihetetlenül sok újításon ment keresztül, és a használati tendenciája egyre inkább felfelé ívelőnek látszik.



MongoDb:

A MongoDB egy nyílt forrású adatbáziskezelő rendszer (DBMS), amely dokumentum-orientált adatbázis-modellt használ. A MongoDB C ++ nyelven íródott. A MongoDB támogatja az adatok különféle formáit. A MongoDB az adatokat sima fájlokban tárolja, saját bináris tárolóobjektumaik segítségével. Ez azt jelenti, hogy az adattárolás nagyon kompakt és hatékony, kiválóan alkalmas nagy adatmennyiségek tárolására. A MongoDB az adatokat JSON-szerű dokumentumokban tárolja, ami nagyon rugalmasvá és méretezhetővé teszi az adatbázist.



A MongoDB dokumentum-orientált adatbázis-modell. Minden MongoDB adatbázis gyűjteményeket tartalmaz, amelyek viszont dokumentumokat tartalmaznak. Az egyes dokumentumok különbözhetnek, és a különböző mezők számától függhetnek. Az egyes dokumentumok mérete és tartalma különbözik egymástól. Az adatmodell funkciói lehetővé teszik tömbök és komplexek hierarchikus kapcsolatban strukturált tárolását.

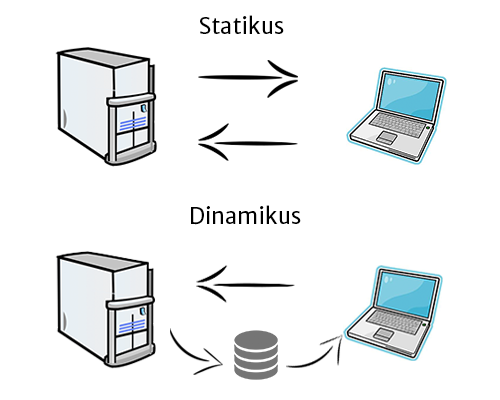
Xamarin:

A Xamarin egy nyílt forráskódú platform, amellyel a C# vagy F# használatával számos különböző eszközre hozhat létre mobilalkalmazásokat.



## Felépítése

A honlapok lehetnek statikusak és dinamikusak. A statikus weblapok tartalma állandó, nem változik (például bemutatkozó oldalak) csak manuális átszerkesztéssel. A dinamikus weblapok alapeleme az ún. motor, amelynek segítségével ismételten végezhetőek azonos műveletek egyszerűen (például hírportálok) és nem kell a forráskódot manuálisan szerkeszteni, mert azt a honlapmotor állítja elő sablonszerűen. A statikus weblapok létrehozására alkalmas a HTML és JavaScript és minden kliens oldalon futó webes nyelv. A dinamikus weblapokhoz szükség van egy-egy olyan leíró nyelvre (például PHP, Java), amely szerveroldali és a szerveren képes a felvitt információkat/adatokat rögzíteni akár szöveges (fizikai fájlok), akár adatbázisba betárolás formájában (például mySQL, postgreSQL).



A legtipikusabb ilyen nyelvek:

* [kliens](https://hu.wikipedia.org/wiki/Kliens)-oldali, a [böngészőben](https://hu.wikipedia.org/wiki/B%C3%B6ng%C3%A9sz%C5%91) végrehajtott [szkriptek](https://hu.wikipedia.org/wiki/Szkript):
* JavaScript
* [szerver](https://hu.wikipedia.org/wiki/Szerver)-oldali [szkriptek](https://hu.wikipedia.org/wiki/Szkript) (a [böngésző](https://hu.wikipedia.org/wiki/B%C3%B6ng%C3%A9sz%C5%91) statikus [HTML](https://hu.wikipedia.org/wiki/HTML)/[XHTML](https://hu.wikipedia.org/wiki/XHTML) lapot   
  és esetleg [kliens](https://hu.wikipedia.org/wiki/Kliens)-oldali [szkripteket](https://hu.wikipedia.org/wiki/Szkript) kap):
* PHP
* ASP
* JSP
* Perl

A [szkripteken](https://hu.wikipedia.org/wiki/Szkript) túl lehetnek egy lapba ágyazva további speciális objektumok is, például [Flash](https://hu.wikipedia.org/wiki/Adobe_Flash) ([SWF](https://hu.wikipedia.org/wiki/SWF)) vagy ún. [Java](https://hu.wikipedia.org/wiki/Java_programoz%C3%A1si_nyelv) kisalkalmazások.

## Webalkalmazás felépítése

A programozási nyelvek fejlõdését figyelemmel kisérõ programozók fülében ismerõsen cseng a Java szó. A Java napjaink talán legígéretesebb programozási nyelve, melyet a Sun Microsystems és a Netscape Communications Corporation együtt fejlesztettek ki. A Java nyelv nem más, mint egy hatalmas, tisztán objektum-orientált programozási nyelv, melynek nagy ereje, hogy segítségével platformfüggetlen alkalmazások készíthetõk. A Java nyelven megírt programokat egy ún. köztes kódra (byte kód) kell lefordítani, melyek különbözõ platformokon, különbözõ operációs rendszerek alatt futtathatóak - egy értelmezõ program segítségével. A Java szóval napjainkban - a WWW világában - három területen találkozunk.

* Egyrészt beszélünk Java programokról, mely programokat le kell fordítani és önállóan futnak. (Nem támaszkodnak más programokra.)
* A JavaScript a Java nyelv "kis testvérkéje"- mondják sokan. A JavaScript-ek a HTML lap szövegébe épülnek be, csak a böngészõben képesek futni.
* A Java appletek (programocskák) - kis Java programok -, melyek a Java-t támogató böngészõkbõl futtathatóak. Ezek tehát felhasználják a böngészõk által nyújtott szolgáltatásokat. Az appleteket is le kell fordítani és - ahogyan azt már korábban bemutattam - ezek HTML dokumentumokból hívhatók.

A JavaScript, mint neve is mutatja, egy script nyelv, amit legelõször a Netscape Navigator 2.0 támogatott. A JavaScript-eket HTML lapokba lehet beágyazni, a lappal együtt töltõdnek le, majd a böngészõ értelmezi és futtatja azokat. Hasonlít a Java-ra, de ugyanakkor sokkal kisebb, egyszerûbb, korlátozottabb mûködésû is annál.

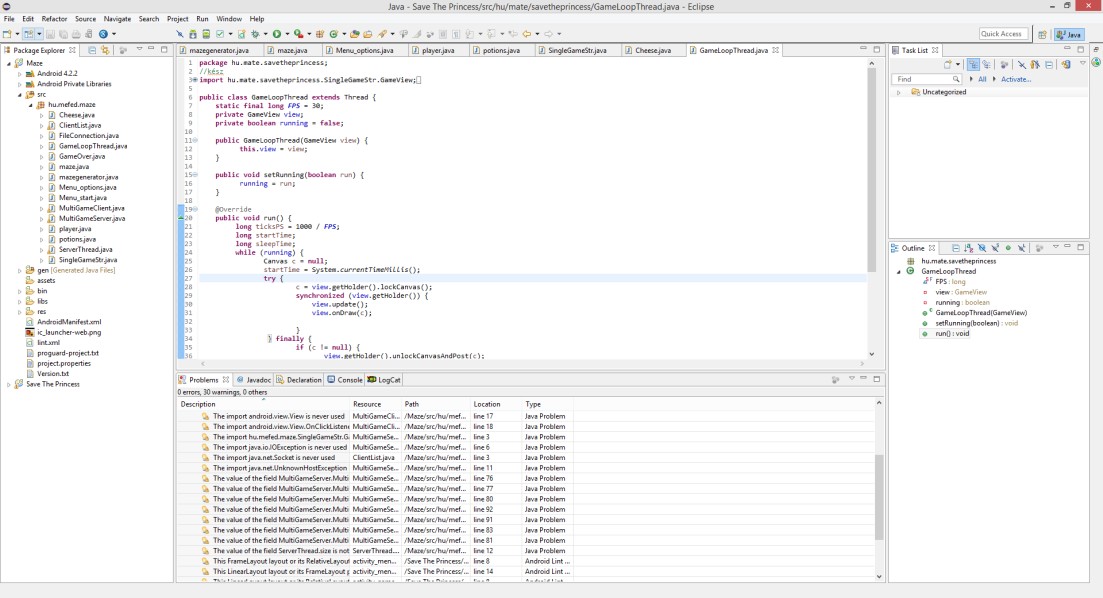
Nézzünk meg egy összehasonlító táblázatot arra vonatkozóan, hogy mik a fõbb hasonlóságok és különbségek a JavaScript és a Java appletek között:

|  |  |
| --- | --- |
| JavaScript | Java applet |
| A kliens értelmezi (interpretálja). | A szerveren fordított - letöltés után - kliensen futó. |
| Objektum-alapú. Beépített, kiterjeszthetõ objektumokat használ, de nincsennek osztályok és öröklõdés. | Objektum-orientált. Az appletekben vannak osztályok és öröklõdés. |
| A program beágyazódik a HTML dokumentumba. | Az appletek a HTML lapból hívhatók meg, de azoktól elkülönülnek. |
| A változók típusát nem kell deklarálni. (Nem típusos nyelv.) | A változókat deklarálni kell. (Erõsen típusos nyelv.) |
| Dinamikus kötés. Az objektum-hivatkozások ellenõrzése futásidõben történik. | Statikus kötés. Az objektum-hivatkozásoknak már fordításkor létezniük kell. |
| Nem tud önállóan a merevlemezre írni. | Nem tud önállóan a merevlemezre írni. |

A JavaScript-et azért fejlesztették ki, hogy az egyébként statikus lapokba dinamizmust vigyenek. Egyrészt látványosabbá tehetjük vele azokat (dinamikusan, futásidõben generálhatunk HTML lapokat), másrészt számos praktikus dologra használhatjuk fel (pl. jelszót kérhetünk a lapunk megtekintéséhez, vagy pl. ellenõrizhetjük az elküldendõ adatok helyességét még a kliens oldalon - hálózati erõforrásokat spórolva meg, stb).

## Fejlesztői környezet

Bármely program / szoftver, amelyet látunk vagy használunk, a háttérben futó kóddal működik. Hagyományosan a kódolást a hagyományos szerkesztőknél, vagy akár az alapvető szerkesztőknél, mint például a Jegyzettömb használják! Ezek a szerkesztők alapvető támogatást nyújtottak a kódolók számára. Néhányuk annyira alapvető volt, hogy nagyon nehéz volt benne írni az alapvető angol szintű programokat. Az idő múlásával néhány programozási nyelvnek speciális keretre és támogatásra volt szüksége a további kódoláshoz és fejlesztéshez, ami ezen szerkesztőkkel nem volt lehetséges. A VI szerkesztő, a Sublime szövegszerkesztő és a Visual Studio kód a sokféle szerkesztő közül, amelyek léteznek. A legszembetűnőbb, és szinte minden kódoló nyelvet támogat, a VISUAL STUDIO CODE. A Visual Studio Code szolgáltatásai lehetővé teszik a felhasználó számára, hogy a felhasználásnak megfelelően módosítsa a szerkesztőt, azaz a felhasználó letöltheti a könyvtárakat az internetről, és integrálhatja a kódhoz az igényei szerint.



1. *ábra Eclipse*

A Visual Studio Code egy laikus szempontból kódszerkesztő. Meghatározásaként a Visual Studio Code „ingyenes szerkesztő, amely segíti a programozót a kódírásban, segít a hibakeresésben és a kód javítását az intelli-sense módszerrel”. Normál esetben megkönnyíti a felhasználót, hogy egyszerűen írja a kódot. Sokan azt mondják, hogy az IDE fele és szerkesztője, de a döntés a kódolóktól függ.

A Visual Studio kódot támogató leggyakoribb nyelvek:

* C #
* Visual Basic
* Java-Script
* R
* XML
* Piton
* CSS
* MEGY
* PERL

Egy másik szolgáltatás, amelyet a naiv felhasználók vagy bárki azonnal láthat, és különbözik a többi szerkesztőtől, a Visual Studio kód felhasználóbarát jellege. A Visual Studio Code használhatóságát nagyon könnyű kezelni. A fájl hierarchikusan van elrendezve, és rendszeres szoftverekkel rendelkezik, mint például eszköztár, állapotsor és oldalsáv. Ezenkívül van egy lebegő Windows explorer ablaka is, amelyet a kényelem alapján egy helyen rögzíthetünk, amely a fájlok könyvtárstruktúrájából áll. Ezek a fájlok (kódfájlok, képmappák stb.) Innen nyithatók vagy átnevezhetők, és a változások automatikusan tükröződnek a tárolóban.

# Felhasználói dokumentáció

## Telepítés

Mivel a játékom Android platformra íródót ezért a telepítéséhez szükség van egy

„.apk” kiterjesztésű állományra. A játékot csak Android 4.0 Ice Cream Sandwich verzió feletti eszközökre lehet feltelepíteni. A telepítéséhez be kell kapcsolni a beállítások/biztonság menüpont alatt lévő Ismeretlen forrásokat. Ez azért lényeges, mert nem Play Store-ból kell telepíteni, hanem a telefon tárhelyéről. A feltelepítéshez meg kell keresni a telepítőt és annak indításakor előugrik egy ablak. A telepítéskor előugró panelban meg lehet tekinteni a futáshoz szükséges jogokat. Ha ezzel megvolnánk, akkor lehet csak rányomni a telepítésre. A feltelepítés után a menüben létrejön a játéknak az ikonja. Az első használatkor hozza csak létre a futásához szükséges mappát és abban az elemeket. Ez után már lehet is játszani a játékkal.

## Játék főbb jellemző

A játék kijelző méretétől független és mindegyik eszközön hasonló játék élményt ad. A futásához szükséges minimum 256 MB ram és 20 MB szabad hely a belső tárterületen. Csak azokon az eszközökön fut el, amelyikben van gyorsulás mérő szenzor és elérés a fájlrendszerhez.



*4. ábra Induló ablak*

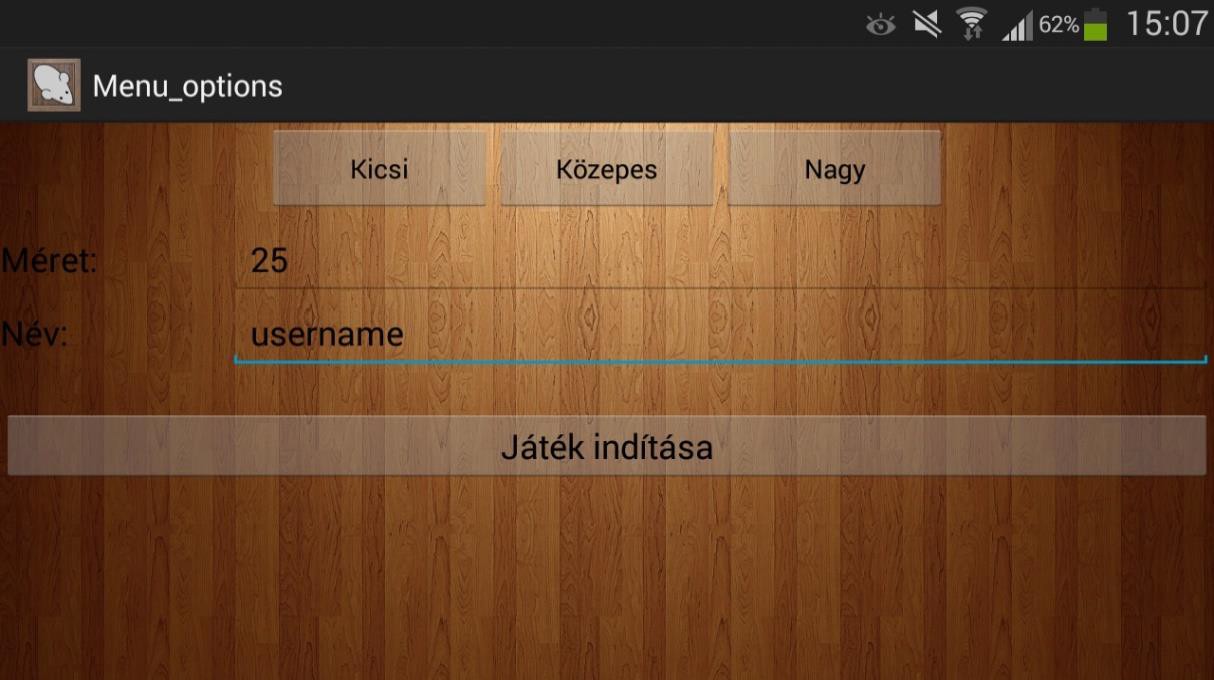
A játékos a beállított pálya méretét át adhatja NFC-n keresztül egy másik játékosnak, csak az a lényeg, hogy mindkét telefonon rajta legyen a játék és mindkét eszközön be legyen kapcsolva az NFC. Az NFC bekapcsolására, mint a

3. ábrán is látszik, a készülék figyelmeztet. Az NFC-n történő kommunikációhoz arra van szükség, hogy a küldő félen fusson a játék, és ha a telefon hátulját összeérintik, akkor a fogadó félnél megjelenik a játék a menüvel és ott be lesz állítva a küldött pálya mérete.

## Játék kezdése

A játék elindításakor megjelenik egy gomb (4. ábra), amire ha rányomunk, akkor beléphetünk a beállításokba (5. ábra) az itt lévő elemek kitöltése kötelező. Az előzőleg beállított értékek eltárolódnak és azzal lehet ugyanolyan játékot nyitni. A menüben az első 3 gombnak az a funkciója, hogy 3 alappálya méretet állítson be. Ezek a következők:

* + - Kicsi ez 50\*50-es
    - Közepes 80\*80-as
    - Nagy 100\*100-as

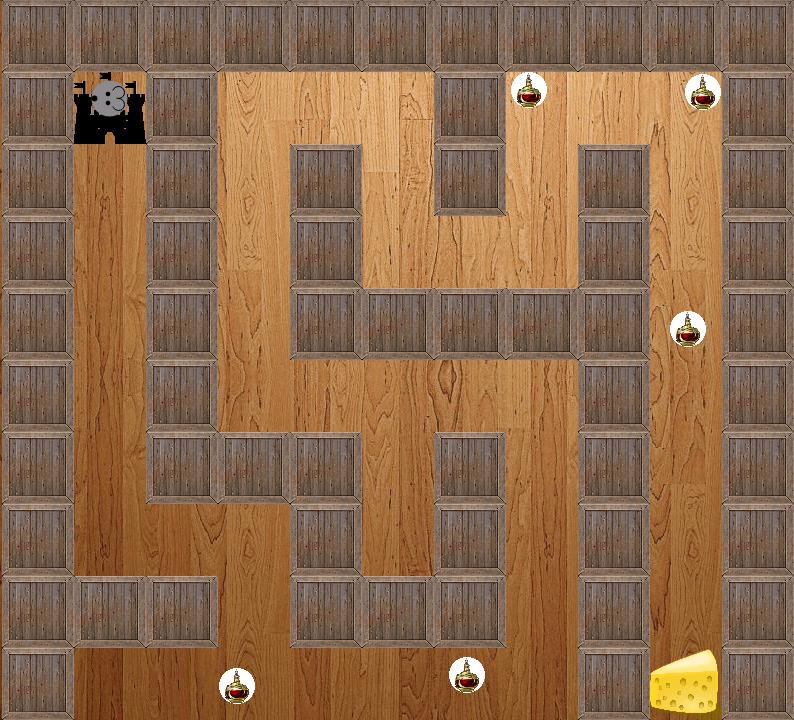


1. *ábra Menü*

Ezen felül a beviteli panelba lehet beírni az egyénileg kiválasztott pálya méretét, aminek 9 és 251 közötti számnak kell lennie. Ha már régebben beállítottál adatokat, akkor ezek megjelennek, és rögtön kezdhetsz is vele játszani. Ez után kell beállítani a játékos nevét. Ez akárhány karakter lehet.

A játék elindításához meg kell nyomni a játék indítása gombot, ami ilyenkor ellenőrzi az adatokat, és ha valami hibát észlel, akkor azt a mezőt kinullázza. Ekkor újra ki kell azt a mezőt tölteni. Ha mindent sikerült helyesen beállítani elindul a játék

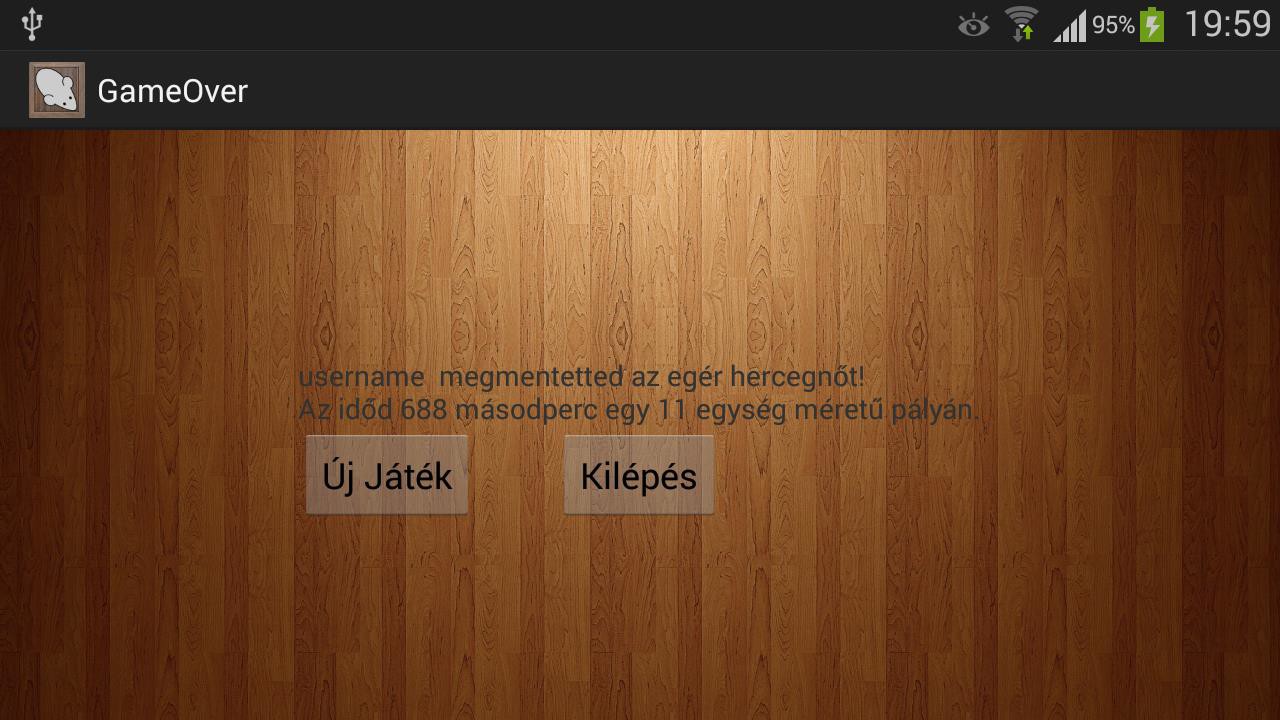
## Játék

A játékot elforgatott képernyőn lehet játszani, amire automatikusan átvált az elinduláskor. A játékos egy egérrel van, akit a telefon döntésével tud irányítani. Az egér mindig menetirányba fordul így mutatva, hogy merre dől a telefon. A sebessége a dőlésszögtől függően változik és körülbelül 20°-os szögben éri el a maximumot. A súrlódás miatt van egy küszöb, ami felett indul el csak a golyó, de ezt követően a sebesség a kiindulási alá is mehet.

1. *ábra Játék*

Az egér célja, hogy megkeresse a pálya túloldalán lévő sajtot és azt visszavigye a várhoz, ami a játék kiinduló pontján van. A játékot izgalmasabbá teszi az, ha a játékos bájitalok segítségével jut el a sajthoz, akkor még nem ismeri pontosan az odavezető utat és ez így megnehezíti a dolgát. Menet közben különböző bájitalokat lehet felszedni, amivel alternatív útvonalakat nyithat meg, vagy csukhat be a játékos. Ezen kívül még van, kettő bájital. Az egyik azt eredményezi, hogy a játékos át mehet a falon, a másik pedig azt, hogy a játékos lelassul. Mivel mind a négy ugyan úgy néz ki, ezért felszedés előtt nem lehet tudni, hogy az jó lesz-e vagy sem. Felszedésük után a felül megjelenő szöveg árulkodik a hatásukról. A lassulás és a falon történő átmenethez tartozik egy a ball felső sarokban megjelenő számláló, ami 100%-ról számol visszafelé és azt mutatja, hogy mennyi százalék van még hátra a bájital hatásából. Ez időben körülbelül 3 másodpercet jelent. A bájitalok a pálya területén véletlenszerűen generálódnak. Egy pályán csak korlátozott számú ital van. A sajt megszerzése után vissza kell térni a várba, hercegnő megmenekül, amivel véget is ér a játék.

## Játék vége

A játék végén megjelenik (7. ábra), hogy menyi idő alatt és mekkora méretű pályán mentette meg a játékos a hercegnőt. Ez után át lehet menni az Új Játék gombbal a játék beállításai menüponthoz, vagy ha a Kilépés gombot választjuk, akkor pedig visszalép a kezdő ablakba és onnét egy vissza gombbal el lehet hagyni a játékot.

1. *ábra Játék vége*

# Fejlesztői dokumentáció

A fejlesztés során több Android specifikus jellemzőt figyelembe kellet vennem. Az alacsony verem méret, korlátozott erőforrások és viszonylag kicsi kijelző méret. Ezek alapján született meg a végleges játékom. A játék felületét canvasbe rajzolom ki, ami leegyszerűsítette a felületre való rajzolást. Mivel 2D-os a játékom ezért ez nem jelentett nagy problémát. A fejlesztésem célja egy a játékomhoz szükséges egyszerű játékmotor kifejlesztése is volt. Ez végzi az ütközéseseknek a számítását és a kirajzolás vezérlését.

## Első lépések

A Fejlesztést a grafikus felület tervezésével kezdtem és menük szintjének a kialakításával. Itt az elsődleges szempontok a letisztultság, egyszerű kezelhetőség és gyors navigálás volt. A kezdő képernyőn egy nagyméretű gombot helyeztem a gyors játék indításhoz. Ebből nyitom meg a menüt ahol a játékos nevét, pálya méretét lehet beállítani. Minden adatot egy szöveges fájlba mentek le és innen mindig vissza lehet tölteni, így ugyanazokkal az adatokkal elég gyorsan lehet új játékot indítani. Az egyjátékos mód a menü után rögtön indítható. A játék véget érése után megjelenik a játékos neve, alatta a játékidő, új játék és a kilépés gombok.

Ezt követte a dizájn kialakítása. Mivel a játékom lényegét tekintve egy labirintus ezért arra jutottam, hogy a fa illene legjobban hozzá. A háttérnek a textúrája mindenhol padlóra hasonlít. A Játékteret úgy alakítottam ki mintha egy raktárban lenne. Ezt a hatást azzal kívántam elérni, hogy fa dobozokat alkalmaztam textúraként a falon. A menünek az elemeit, vagyis a gombokat és egyéb paneleket az Android gyári elemeivel oldottam meg. Ezeknek az enyhén áttetszősége jól mutat a háttérrel.

A játékos egy Egér és elég kicsi ahhoz, hogy képként jól nézzen ki, így annál úgy döntöttem, hogy kódból fogom generálni. A bájital kinézeténél amellett döntöttem, hogy egy fehér háttéren lesz egy piros, sárga színű rajz. Ezért jól elkülönül, könnyen felismerhető. Mivel a kijelzők mérete különböző ezért egyes eszközökön nagyobb, máshol pedig kisebb az elemeknek a mérete. Ez nem befolyásolja a játék élményt és nem ad senkinek előnyt a nagyobb kijelző.

## Menük

A menüket külön activity-kben valósítottam meg, amik egymásra épülnek. A grafikus felületet XML állomány generálja le és mindegyikhez egy külön Java osztály tartozik. Itt a kihívást a menü pontok meghívása okozta amiket intentekkel oldottam meg (1-2.kódrészlet)

Intent intent = **new** Intent (Menu\_start.**this**, Menu\_options.**class**); intent.putExtra(***EXTRA\_MESSAGE***, "Single");

startActivity(intent);

1. *kódrészlet Intent hívás*

Mivel a túloldalon lévő menü a felelős az egyjátékos mód mellett a többjátékos mód szerver és a kliens beállításáért is, ezért küldök egy plusz üzenetet (2. kódrészlet), amivel tudom azonosítani a módot. A menü és a játék között pedig a játék indításához nélkülözhetetlen adatokat.

Intent intent = getIntent();

String[] input = intent.getStringExtra(Menu\_start.***EXTRA\_MESSAGE***);

1. *kódrészlet Intent fogadás*

A Menu\_option.java állományom tartalmazza az összes beállítást ennek a tartalmának a konfigurációs állományba való kiírását egy külön osztály vezérli. Ez az állomány tartalmazza a változókat is. A menüben lévő gombok mindegyikére kódból helyeztem rá a funkciókat. Az adatokat csak akkor ellenőrzöm, amikor a játékos rányom a játék indítására és ilyenkor a hibás részeket módosítania kell előtte.

Mivel a java nyelvben nincsen TryParse mint a c#-ban ezért erre csináltam egy saját függvényt. Ami azt csinálja, hogy megpróbálja a szöveget számmá konvertálni, és ha nem sikerül, akkor kinullázza a beviteli mezőt.

## Játék szerkezete

A játék osztály meghívása után, ami a SingleGameStr az OnCreate()-ben létrehozok egy SensorManager-t a gyorsulás mérőből való adat kiolvasására (3. kódrészlet) ez minden egyes alkalommal lefut, amikor újadat érkezik az érzékelőtől. Az abból kapott adatokat két változóban tárolom el és ezeket majd a

játékom dolgozza fel. A senorból kiolvasott adat egy tömbben tárolódik, aminek az első eleme telefonoknál bedöntött képernyőnél megfelel az y-nak és a második 1-es index eleme pedig az x-nek.

**public void** onSensorChanged(SensorEvent event) {

**switch** (event.sensor.getType()) {

**case** Sensor.***TYPE\_ACCELEROMETER***: accellerometerSpeedX = event.values[1]; accellerometerSpeedY = event.values[0]; **break**;

}

}

1. *kódrészlet Szenzorból való adat kiolvasása*

A Szenzor hívás után hozom létre a játékteremet ez a GameView osztályban található, és mielőtt ennek átadnám a kijelzőre való rajzolást, beállítom a képernyőt úgy, hogy ne kapcsoljon ki menet közben és teljes képernyős módba váltson (4.kódrészlet).

getWindow().addFlags(WindowManager.LayoutParams.

***FLAG\_KEEP\_SCREEN\_ON***);

requestWindowFeature(Window.***FEATURE\_NO\_TITLE***); getWindow().setFlags(WindowManager.LayoutParams.***FLAG\_FULLSCREEN***,

*WindowManager.LayoutParams.***FLAG\_FULLSCREEN***);*

1. *kódrészlet Teljes képernyő, képernyő kikapcsolásának tiltásai*

Ezek után adom át a vezérlést a GameView-nak ami elösször létrehoz egy szálat és ez a szál fogja majd késöbb az időzített frisítést irányítani. Ezt követi a képek betöltése és az osztályok incializálása. Mivel minden egyes készülék tipusnak eltérő a képmérete, ezért létrehoztam egy változót ami azt tartalmazza, hogy egy blokknak mekkora legyen a mérete. Ennek a változónak az értéke mindig a kijelző függőleges tengelyének a pixelszámának a tizede. Így minden eszközön ugyan annyi jelenik meg a pályából egyszerre. A képek méretét betöltéskor lecsökkenti a blokknak a magasságára. A GameView osztályomnak két fő metódusa van, egyik

a frisítésekért felelős a másik pedig a kirajzolásért. Ezeket mindig az előbb már említett szál fogja egymás után meghívni másodpercenként harmincszor.

**public void** run() {

**long** ticksPS = 1000 / ***FPS***;

**long** startTime; **long** sleepTime; **while** (running) {

Canvas c = **null**;

startTime = System.*currentTimeMillis*();

**try** {

c = view.getHolder().lockCanvas();

**synchronized** (view.getHolder()) { view.update(); view.onDraw(c);

}

} **finally** {

**if** (c != **null**) { view.getHolder().unlockCanvasAndPost(c);

}

}

sleepTime = ticksPS-(System.*currentTimeMillis*()

– startTime);

**try** {

**if** (sleepTime > 0)

*sleep*(sleepTime);

**else**

*sleep*(10);

} **catch** (Exception e) {}

}

}

1. *kódrészlet Frissítés vezérlő*

A frisítés metódus (5. kódrészlet) alatt hivja meg egyesével a szükséges frissítendő elemeket. Először még itt megnézi, hogy a játékos hova mozgott vagyis utána pedig, hogy van-e valamilyen hatás, ha van akkor a szerint dönt, ilyen példáúl hogy áttudok menni a falom vagy sem. Ha igen akkor a játékos osztálynak más ellenörző metódusát hivja meg. Itt elenőrzi a fal elvételt és hozzáadást is. A frissítés osztály tartalmazza a játék végének kiértékelését is, mivel ha vége van akkor át kell irányítani a játék vége képernyőre. A frissítés utolsó pontja a kamera mozgatása.

A frissítés lefutása után fut le a kirajzolás. Ezt úgy oldottam meg, hogy mindegyik osztály amelyiknek kell a képernyőre rajzolnia az a saját kirajzolását maga végzi

el. Így azokat egyesével meghivja. A kirajzoláshoz mindig be kell szorozni a blokk mérettel a pozíciókat, mert az lesz a tényleges pozíciója a képernyőn.

**private void** cameramove() {

pozx = ((displaysize.x / 2) - player.getPoz().x \* blocksize)

/ blocksize; pozy = ((displaysize.y / 2)- player.getPoz().y \* blocksize)

/ blocksize;

**if**(pozx > 0) pozx = 0;

**if**(pozx < pozxmax) pozx = pozxmax;

**if**(pozy > 0) pozy = 0;

**if**(pozy < pozymax) pozy = pozymax;

}

1. *kódrészlet Kamera mozgatása*

Mivel a kijelző mérete nagyon kicsi a számítógépekhez képest és ez azt eredményezi, hogy nem lehet megoldani a teljes pálya kirajzolását és utólagos maszkolását, ezért csúsztatnom kell a teljes képet minden irányba. Ennek a megvalósítását úgy csináltam meg, hogy van egy vektorom amivel csusztatom a képet. A mozgatásnál mindig megpróbálom a képernyő közepén tartani a játékost ha pedig már közeledik a pálya széléhez, vagyis már nem tud a pálya mozogni akkor pedig a játékos kezd el mozogni. Ezt kódból úgy valósítottam meg, hogy még a játék indulásakor megnézem a képernyő és a pálya közti különbségeket. Ez lesz a vektorom maximuma. Ez után minden egyes frissítésben kiszámolom a játékos és a képernyő közepe közti különbséget és ha ez 0 és a elöbb kiszámolt maximum között van akkor egyenlő lesz a számításban lévővel, ha kisebb akkor egyenlő lesz nullával, ha nagyobb akkor pedig a maximum lesz. A pozx-et és y-ot mindig hozzáadom az összes kirajzolandó komponenshez.

## Labirintus

A labirintus generálás a játékom fő komponense ennek a megvalósításával telt el a legtöbb időm különböző hibák miatt.

Először egy rekurzív algoritmuson gondolkodtam, aminek el is készítettem a megvalósítását egy már meglévő algoritmust alapul véve. Ezt még sima Java

futatási környezetben Windowson teszteltem és hibátlannak tűnt csak akkor kezdődtek a gondok, amikor megpróbáltam áthelyezni Androidra. Első körben a különböző nyelvi elemek jelentetek gondot, de ezek kijavítása után már megfelelően működött. Ezt követően elkezdtem tesztelni akkor olyan hibák jöttek elő, amire nem számítottam. Az 50 \* 50 –es méret felett egyszerűen lehalt az alkalmazásom. Ez komoly hibának tűnt és elsőre nem is találtam rá magyarázatot, hogy miért dobott „StackOverFlowExceptiont”-t. Mint már utólag kiderült az Android alkalmazásokat futató Dalvik virtuális gép verem mérete telefontól függően 8-15 MB-t körül van alaphelyzetben. Ami igaz, hogy menet közben a terhelés hatására növekedhet, de ez túl lassú volt a rekurzív algoritmusomnak. Előbb elfoglalta a memória területet, mint ahogy a virtuális gép újat becsatolt volna. Ezért a rekurzív algoritmusok korlátozott mértékben működő képes, de nem olyan méreteknél, mint ahogy én használtam. Ennek a megoldására írtam át az algoritmust úgy, hogy egymást kövessék a lépések.

Az eredeti rekurzív algoritmus annyit csinált, hogy a labirintust feltöltötte falakkal és utána ebbe járatokat vájt. Egy pontból mindig minden lehetséges irányba elindult. Az után az azt követő ponttal pedig meghívta magát újra, és így minden egyes új pont újabb 4-et hívott meg, ami 50\*50–nél is már több mint 2500 hívást jelentett. A stack méretnek a növekedését lehet a manifest állományban növelni, de ez sem vált be így ezután jött létre a következő algoritmus. Az eredeti rekurzív algoritmus alapul véve úgy alakítottam át, hogy megfeleljen az alacsony verem méreteknek. Az eddigi ideiglenes pontokkal történő meghívás helyett egy listában tároltam el, ennek mindig az utolsó elemét nézem újra és újra, amíg ez a lista nem lesz üres. A cikluson belül pedig a lista utolsó eleme által megadott pont körül megnézem, hogy merre lehet tovább menni. Ebből a pontból összeszedem a lehetséges irányokat, ha tovább tudok haladni, akkor a lehetséges irányok közül választok egyet és onnét megyek tovább, ha pedig már egyik irányban sem lehet tovább menni, akkor pedig törlöm azt a pontot (7. kódrészlet).

Ezzel az algoritmussal már sikerül 1000 \* 1000 pályát is legenerálnom, ami elég játszhatatlannak bizonyult, de ez bizonyítja az algoritmusom működő képességét. A legenerált pályát egy 2d logikai mátrixban tárolom. Azért döntöttem mellette, mert ez kisebb helyet foglal, mintha egy listában tároltam volna el a koordinátákat

és könnyen meghatározható benne a játékos pozíciója. Ebben a tömbben a hamis érték jelöli a falat és az igaz a járatot.

Alaphelyzetben egy helyes út van a sajthoz, de mivel a falakat kiszedhetem menet közben ezért alternatív útvonalak is létrejöhetnek. Ezek kisebb méretben egyszerűsíthetik a játékot, de nagyon pályánál érezhetően rontott a játékosok idején.

**private void** generalas(int x, int y){ maze[x][y] = false;

ArrayList<Point> directionarray = new ArrayList<Point>(); directionarray.add(new Point(x,y));

**while** (!directionarray.isEmpty()){

**int** index = (directionarray.size() - 1); Point point = directionarray.get(index); maze[point.x][point.y] = false;

**ArrayList**<Integer> irany = new ArrayList<Integer>();

**if**(point.x >= 2 && maze[point.x - 2][point.y] ==

**true**) irany.add(0);

**if**(point.x < size - 2 && maze[point.x + 2][point.y]

== **true**) irany.add(1);

**if**(point.y < size - 2 && maze[point.x][point.y + 2]

== **true**) irany.add(2);

**if**(point.y >= 2 && maze[point.x][point.y - 2] ==

**true**) irany.add(3);

**if**(!irany.isEmpty()) {

int direction = rnd.nextInt(irany.size()); switch (irany.get(direction)) {

case 0:

**if** (maze[point.x - 2][point.y] == **true**) { maze[point.x - 1][point.y] = **false**; directionarray.add(new Point(point.x

- 2 , point.y));

}

break;

case 1:

**if** (maze[point.x + 2][point.y] == **true**){ maze[point.x + 1][point.y] = **false**; directionarray.add(new Point(point.x

+ 2, point.y));

}

break; case 2:

**if** (maze[point.x][point.y + 2] == **true**) { maze[point.x][point.y + 1] = **false**; directionarray.add(new

Point(point.x,point.y + 2));

}

break; case 3:

**if** (maze[point.x][point.y - 2] == **true**) { maze[point.x][point.y - 1] = **false**; directionarray.add(new

Point(point.x , point.y - 2));

}

break;

}

}

**else**{

directionarray.remove(index);

}

}

}

1. *kódrészlet Labirintus generálás*

## Pálya

Ez az osztály kapja meg a kész labirintust és későbbiekben itt is lehet csak elérni. Ezen kívül ennek az osztálynak a feladata a fal elvétele és visszarakásának a megvalósítása. A fal eltávolításánál véletlenszerűen törlök 5 és a pálya méretének kétszerese közötti számot. A kitörölt falakat egy listában tárolom le. A visszarakásnál csak olyan falat rakok vissza, amit már egyszer eltávolítottam. Ez azért kell, mert ha mást is visszaraknék, akkor holtágak keletkezhetnének, amibe a játékos beleszorulhat. A kirajzolást blokkonként hajtom végre és minden egyes blokknál megvizsgálom, hogy az adott elem látszik-e ha igen akkor rajzolom csak ki, így gyorsítva fel a játék futását.

**public void** addwall(){

**if**(rm.size() >= 1){

**int** count = rnd.nextInt(rm.size());

**for** (**int** i = 0; i < count; i++) { maze[rm.get(i).x][rm.get(i).y] = **true**; rm.remove(i);

}

}

}

**public void** removewall(){

**int** count = (5 + rnd.nextInt(maze.length \* 2));

**for** (**int** i = 0; i < count; i++) {

**int** x; **int** y; **do**{

x = rnd.nextInt(maze.length - 2) + 1 ;

y = rnd.nextInt(maze[0].length - 2) + 1;

}

**while**(!maze[x][y]); maze[x][y] = **false**; rm.add(**new** Point(x,y));

}

}

1. *kódrészlet Fal eltávolítása és hozzáadása*

## Mozgás

Mozgatás során a gyorsulás mérőből kiolvasott adatok alapján növelem a játékos sebességét. A maximális sebesség a lassító bájital hatására megfeleződik. A sebesség vektor alaphelyzetben nem lehet nagyobb, mint 0.25. A sebesség kiszámításakor először Pitagorasz tétellel sebesség nagyságát számolom ki, és ha ez nagyobb, mint a maximális akkor megnézem a dőlésszöget itt az atan függvényt használom. A dőlésszög alapján pedig szinusz és koszinusz függvénnyel meghatározom az x és y koordinátákra eső maximális sebességvektort. Itt a problémám az volt, hogy a tangens két-két negyedben ugyan azt az értéket adja vissza, amivel kiszámolom az új sebességet aztán pedig a megfelelő irányba transzformálom. A Javában lévő atan2 függvény ugyan ezt a szerepet tölti be, mint az én algoritmusomban lévő, de szerettem volna saját megvalósítást csinálni a problémára.

**double** z = Math.*sqrt*(Math.*pow*((**double**)(speedx + acx),2) +

Math.*pow*((**double**)(speedy + acy),2));

**if**(z >= speed){

**double** alpha =Math.*atan*((**double**)(Math.*abs*(speedy + acy)

/Math.*abs*(speedx +acx)));

**boolean** xn = **false** , yn = **false**; **if**(acx < 0)xn = **true**;

**if**(acy < 0)yn = **true**;

speedy =(**float**)(Math.*sin*(alpha)\*speed); speedx =(**float**)(Math.*cos*(alpha)\*speed); **if**(xn) speedx \*= -1;

**if**(yn) speedy \*= -1;

}

**else**{

speedx += acx; speedy += acy;

*}*

1. *kódrészlet Sebesség vektor számolás*

## Ütközés

Az ütközésnél megpróbáltam kihasználni azt, hogy egy mátrixban tárolom a labirintusomat és ezt szerintem sikerül is. Az ütköztetés során egy kétlépcsős algoritmust használok, mert külön mindegyiknek van holt pontja. Az ütköztetés előtt megnézem, hogy melyik cella van a játékosom leendő pozíciója fellett, alatt, jobbra és balra utána egyesével megvizsgálom, hogy benne van-e a játékosom. Ha igen akkor pedig a fal mellé állítom ellenkező esetben az új pozíciója marad.

Az első megvalósításomban csak az x és az y irányokat elemeztem külön, aminek az eredménye addig jó is volt, amíg nem ért egy csúcshoz, mert akkor be tudott menni a falba. Ez olyan hibákkal is járt, hogy eltudta hagyni a pályát és akkor IndexOutOfRange hibát dobott, mivel így a tömböt is elhagyta.

Ennek a megoldására írtam egy új algoritmust (10. kódrészlet), ami már vektort húz a régi és az új pontok között, és utána egyenes egyenletét írja fel. Amit aztán az új pozícióból pedig egy kör egyenletének veszem. Ennek a kettőnek megkeresem a közös pontjait aztán az előre felé lévőt kiválasztom, és ha az ütközik, akkor számolom ki a pozícióját. Ennek az algoritmusnak is vannak hibái mivel a sarkokba bele engedte a játékost a falba. Így ez az algoritmus sem bizonyult megbízhatónak minden szempontból.

Ezek után döntöttem úgy, hogy a két algoritmust egyesítem és ennek az eredménye jónnak tűnt és így kitudtam javítani mindkét algoritmus hibáját. Először fut le az x és az y irányon való ütközés keresés. Ez után pedig a vektoros ütközés. Ezzel elértem azt, hogy a vektoros ütközésnél mát nem tud belemenni a sarokba, és ha a tengelyes ütközésnél pedig belemegy a csúcsba a második korrigálja.

**float** A = speedx \* speedx + speedy \* speedy;

**float** B = 2 \* (speedx \* (getPoz().x - newpoz.x) + speedy \*

(getPoz().y - newpoz.y));

**float** C = (getPoz().x - newpoz.x) \* (getPoz().x - newpoz.x)

+ (getPoz().y - newpoz.y) \* (getPoz().y - newpoz.y) - r \* r;

**float** det = B \* B - 4 \* A \* C;

**if**(det > 0){

**float** t1 = (**float**)((-B + Math.*sqrt*(det)) / (2 \* A)); PointF Point1 = **new** PointF(getPoz().x + t1 \* speedx,

getPoz().y + t1 \* speedy);

**if**(mz[(**int**) Point1.x][(**int**) Point1.y]){

**if**(speedx < 0){

newpoz.x = ((**int**)Point1.x) + 1 + 0.17677f; speedx = 0;

}

**else** {

newpoz.x = ((**int**) Point1.x) - 0.17677f; speedx = 0;

}

**if**(speedy < 0){

newpoz.y = ((**int**) Point1.y) + 1 + 0.17677f; speedy = 0;

}

**else** {

newpoz.y = ((**int**) Point1.y) - 0.17677f; speedy = 0;

}

}

}

1. *kódrészlet Ütközés vektorosan*

## Játékos osztály

A Player osztály tartalmazza az összes játékossal kapcsolatos metódust. Ide tartozik a fentebb említett ütközés és a mozgás is. Ezeken kívül itt vannak a mozgással kapcsolatos bájitalok. Mint a lassulásnak és a falon történő áthaladásnak a megvalósítása. A falon történő áthaladáshoz azt a kódot használtam fel, amit akkor írtam mikor az ütközés hibáját kerestem. Ez a kód annyit csinál, hogy nem engedi, hogy a játékos elhagyhassa a pályát.

Ez az osztály tartalmazza még a játékos kirajzolását is. Mivel a játék grafikus elemeinek a megjelenítéséhez canvast használok ezért a játékos kirajzolásához is annak az elemeit használtam fel. Mivel a játékos mindig menetirányba fordul ezért a fejét forgatnom kell. Ehhez lekellett mentenem a canvast. Utána a megfelelő irányba elkellett forgatnom és kirajzolni rá a füleket, szemeket és az orrot. Utána pedig vissza kellet állítani a canvast.

## Játékos adatok tárolása

A játékos adatokat linuxos gondolkodásmód alapján egy könnyen szerkeszthető állományban helyeztem el ami „.conf” kiterjesztést kapott. Ennek a létrehozásához egy külön osztályt hoztam létre amit a menü hív meg. Ez az osztály lett a FileConnection.java. Ennek az osztálynak a példányosításakor lefuttatok több tesztet. Megvizsgálom, azt hogy az eszközön futó játék kapott-e fájlolvasási jogot. Ha kapott, akkor megnézem, hogy létezik-e a mappa, ha nem akkor létrehozom. Ebben a mappában utána megnézem, hogy létezik-e a konfigurációs állomány, ha nem akkor létrehozom és feltöltöm általános adatokkal. Ha létezett és a tartalmi vizsgálatokon is átesett, akkor megfelelő az állományom. Bármilyen hiba esetén nem használható a beállítások mentése, minden egyes játék indítása előtt be kell írni újra az adatokat. De a fájl ellenőrzés helyes értékekkel tér vissza akkor beolvasom a fájl tartalmát és beleírom a változókba. Minden egyes változó private így mindegyikhez tartozik egy get-set metódus.

## Bájitalok

A bájitalok felelősek a játékon belüli segítő vagy rontó hatásokért. a Játék indulásakor legenerálok véletlenszerű helyeken, a pálya méretének felével egyenlő számú bájitalt. Mindegyiknél ellenőrzöm, hogy nincs-e benne a falban vagy nincs-e a játékos alatt. Ezek után megnézem, hogy arra a pozícióra generáltam-e már. Ha ezek közül bármelyik is teljesül, akkor új helyet keresek neki. Ezt úgy valósítottam meg, hogy csináltam egy hátul tesztelő ciklus, aminek van egy változója. Ez a változó, ha igaz értékű az ellenőrzéskor, akkor megvan a pozíciója. Ezzel még csak a blokkját kaptam meg ezután határozom meg a blokkon belüli helyét. Első generálással érem el azt, hogy ugyan abba a blokkba ne kerülhessen még egyszer bájital. A második pedig azért felelős, hogy kikerülhető legyen és ne csak a blokk közepén álljon. Ha minden rendben ment az új ital bekerült a listába és ezt addig ismétlen ameddig le nem generálódik az összes.

**public void** potioncollision(PointF player){

**for** (**int** i = 0; i < pot.size(); i++) {

**if**(Math.*pow*((pot.get(i).x - player.x),2) +

Math.*pow*((pot.get(i).y – player.y),2) <=

.25){

potioneffect = 1 + rnd.nextInt(4); potioneffecttime = 100; pot.remove(i);

}

}

*}*

1. *kódrészlet Bájital játékos ütközés*

Ennek az osztálynak van egy ütközés eleme, ami egy egyszerű kör-kör ütközés. Ennél kiszámolom a két kör középpontjának a távolságát, és ha ez kisebb a játékos és a golyó méretének összegénél, akkor ütköztek. Ekkor határozom meg a bájital hatását és az idejét. Ha ütközés történt, akkor eltávolítom az elemet a listából.

Kirajzolásnál először kirajzolom a listában lévő elemeket. Ez után, ha idővel rendelkezik, akkor a felső sáv baloldalára kiteszek egy számlálót. Ami 100%-tól számol visszafele. A felső sávban az effekt létrejötte után rövid ideig kiírom, hogy mi volt a hatása.

## Játék indítása NFC-n keresztül

Az NFC egy olyan RFID rendszer, ami kétirányú kommunikációt tesz lehetővé a végpontok között. Az NFC eszköz kiválóan használható egyszerű adatok érintéssel való átadására, mint például a bankkártyás fizetésre. Ezzel a technológiával lehet még gyorsítani a Wi-Fi-s vagy a Bluetooth-os eszközök csatlakozását. Így sokkal hamarabb készen áll az eszköz a csatlakozásra.

public NdefMessage createNdefMessage(NfcEvent nfcEvent) { ndeftext = t\_size.getText().toString();

String message = ndeftext;

NdefRecord ndefRecord = NdefRecord.createMime("text/plain",

message.getBytes());

NdefMessage ndefMessage = new NdefMessage(ndefRecord); return ndefMessage;

}

1. *kódrészlet NFC adat küldés*

Ebből az ötletből jött az, hogy ha a barátok játszani akarnak egy teremben ugyan azzal a pálya mérettel, akkor a játékos már a létrehozott pályaméretet megtudja osztani a többi emberrel a telefonok összeérintésével. Mivel úgy akartam megoldani, hogy ez indítsa el az alkalmazást, ezért az elsőre induló activity-be kellet beleraknom a vevő kódot. Az küldő pedig a menü beállításba és a játékba került bele. Mivel az NFC mivel csak a 16-os API verzió felett érhető el ezért a bekapcsolása előtt ellenőriznem kellett, hogy támogatja-e a telefon, és ha igen akkor be van e kapcsolva. Ha nincs bekapcsolva, akkor figyelmeztetem a felhasználót arra, hogy ha használni szeretné, akkor kapcsolja be. A bekapcsolás figyelés mellet ellenőrzöm, hogy a telefonnak az API verziója megfelelő-e ha igen akkor a telefon tartalmaz-e NFC modult. Ezek az ellenőrzések felelősek azért, hogy ne legyen gond a játék futtatása során és ne dobjon hibákat.

Az NFC használatához engedélyt kell kérni. Amit a manifest állományban lehet megtenni, ehhez két bejegyzés kell egyik egy „user-permission” a másik pedig egy „uses-feature”. Az üzenet fogadáshoz kell még egy szűrőt is beállítani. Ezzel azt lehet elérni, hogy amikor összeérintik a két eszközt akkor a fogadó félnél életre keljen az alkalmazás.

A küldéshez kell egy „CreateNdefMessageCallback”-et implementálni és ennek a metódusait kell felüldefiniálni. A következő 13. kódrészlettel lehet NFC üzenetet készíteni. A metódusnak a visszatérési értéke már egy kiküldhető üzenet.

A küldött adatok fogadását a következő kód végzi. Ez felelős az alkalmazás indulás utáni adatok intentből való kiolvasásáért és értelmezéséért.

mAdapter = NfcAdapter.*getDefaultAdapter*(**this**);

**if**(mAdapter != **null**) {

**if** (!mAdapter.isEnabled()){ Toast.*makeText*(**this**,R.string.***nfcdisabled***,

Toast.***LENGTH\_LONG***).show();

}

**else**{

Intent intent = getIntent();

**if**(NfcAdapter.***ACTION\_NDEF\_DISCOVERED***.equals( intent.getAction())){Parcelable[] rawMessages =

intent.getParcelableArrayExtra(

NfcAdapter.***EXTRA\_NDEF\_MESSAGES***); NdefMessage message = NdefMessage)rawMessages[0]; String input = **new** tring(message.getRecords()[0]

.getPayload());

Intent intent1 = **new** Intent

(Menu\_start.**this**,Menu\_options.**class**); intent1.putExtra(***EXTRA\_MESSAGE***,"Single-"+ input); startActivity(intent1);

}

}

1. *kódrészlet NFC adatok fogadása*

## Tesztelés

A program tesztelését több féle módon valósítottam meg. A fejlesztés során többször alkalmaztam azt, hogy a modulokat külön készítettem el és csak később kapcsoltam bele a tényleges rendszerbe. Ez azért is volt jó, mert így Windowson letudtam tesztelni a program részleteket és így a későbbi portolás során már az Android specifikus dolgokra tudtam leszűkíteni az esetlegesen fellépő hibát. Ez a módszer gyors és hatékony tesztelést tett lehetővé. A modulok tesztelése során alkalmaztam még, hogy előre meghatároztam a bemeneti és kimeneti adatokat és így is sok hibát kitudtam szűrni. Gondot jelentett a Windows és az Android közti

különbség. Ez főleg a labirintus generálásnál jött elő és egy teljesen új irányvonal alkalmazásával járt.

Ez után következett a végső tesztelés. Mivel a fejlesztés során a tesztelést egy telefonon végeztem ezért a végső fázisban fontosnak tartottam, hogy az eszközök széles skáláján teszteljen. A fejlesztés során megpróbáltam kiszűrni a kompatibilitási problémákat. Ennek meg is látszott az eredménye, mivel a legnagyobb hiba az volt, hogy a gyengébb készülékeken akadozott a játék. A programozás során elkövettem egy technikai hibát, ami által hatalmasra nőt a memória igény. Ezt az okozta, hogy a teljes pályát rajzoltam ki és nem csak a látható részét. Ez nagyobb pályaméreteken vagy gyengébb készülékeken jelentett problémát. Ez csak az utolsó fázisban jött elő és ezért újra kellet írnom a pálya kirajzolását. A végleges rendszerben már csak a pálya látható részét rajzolom ki.

A tesztelési fázisban jött elő a statisztikák készítésének az ötlete, mivel jó lett volna összehasonlítani az eredményeket. A megvalósításának a kitalálása közben

,felmerült bennem egy gondolat, hogy ez nem kivitelezhető mivel az útvonal kialakítása mindig változik és van, hogy egyenesen elehet jutni a célig és van, hogy szinte a teljes pálya bejárásával. Ennek a kipróbálására írtam egy tesztet, ami megkeresi a tökéletes útvonalat a labirintusban és végig halad rajta. A gép által számolt adatok nagyon nagy szórást mutattak, ezért úgy döntöttem nem lesz benne a végső játékban.

# Összefoglalás

A szakdolgozatom célja egy egyjátékos módú telefonos játék. A célom ezzel az volt, hogy megtanuljam, hogy milyen játékot fejleszteni Android platformra. Szerintem a célomat elértem és végeredményül egy használható játék került ki a kezem közül. Ennek a megvalósítását Java nyelven végeztem, amihez még hozzájött a különböző XML állományokkal való munka. A kivitelezés közben adódtak komolyabb hibák, amiket meg kellet oldanom. Jelen állás szerint a széles körűen elvégzett tesztelések eredménye ként azt tapasztaltam, hogy tökéletesen működött, de az eszközök széles skálája miatt akadhatnak olyan telefonok és tablettek, amiken nem megfelelő a játék működése.

A következő pontban azokat a lehetőségeket szeretném összegezni, amelyek egy esetleges későbbi fejlesztés során felmerülhetnek, amerre a program a jelenlegi állapotából tovább fejődhet. Ezek közül vannak, amelyek bár belekerülhettek volna a kész programba, valamilyen okból kifolyólag nem kerültek megvalósításra, és vannak olyan pontok is, amelyek csak a program befejeztével merültek fel, mint új lehetséges fejlődési irányvonalként.

## Jövőbeli tervek

A legnagyobb lehetőség a többjátékos mód megvalósítása ezzel meg lehetne azt oldani, hogy a játékosok egy pályán egymással versenyezzenek. Ennek a megvalósítása történhetne szinkron vagy aszinkron módon is. Ennek a módnak a létrehozásához ki kellene bővíteni a játékszabályokat, mivel az egyjátékos módban lévők nem felelnek meg erre a célra. Meg kellene határozni, hogy ki és hogyan szerezheti meg a sajtot a másik játékostól. A többjátékos módot meglehetne valósítani egy központi szerverrel vagy akár úgy is, hogy az egyik eszköz legyen a szerver.

Lehetőség van a ranglista kialakítására, de ehhez lényegében át kellene alakítani a játékot, hogy összehasonlítható legyen, a pályának mindig ugyan annak kellene lennie. Ami meg teljesen ellentétes a programomnak a koncepciójával. Egyébként, ha megvalósulna, akkor egy adatbázisban képzelném el az adatok tárolását, mert onnét egyszerű lenne a statisztikának a megalkotása.

# Felhasznált irodalmak

* Android alapú szoftverfejlesztés (ISBN: 9789639863279)
* <http://developer.android.com/index.html>(2015.04.24)
* https:infoc.eet.bme.hu (2015.04.24)
* <http://commonsware.com/Android/Android-1_0-CC.pdf>(2015.04.24)
* https://androidteam.googlecode.com/files/professional\_android\_applicatio n\_development.pdf (2015.04.24)
* <http://www.cs.columbia.edu/~coms6998-8/lectures/lec4-Android.pdf> (2015.04.24)
* <http://blogs.wrox.com/files/2013/05/Chapter-8-9781118087299-2.pdf> (2015.04.24)
* Beginning Android 4 Application Development (ISBN: 9781118199541)
* Android Application Development For Dummies, 2nd Edition (ISBN: 9781118387108)

# Melléklet

Melléklet:

* 1 darab CD CD mellékelt tartalma:
* 1 darab „.apk” kiterjesztésű állomány
* 1 darab mappa, ami a szakdolgozat állományait tartalmazza