



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

ALGORITMUSOK ÉS ALKALMAZÁSAIK TANSZÉK

## Pac-Man játék

*Témavezető:*

Nagy Sára

Mesteroktató

*Szerző:*

Vass Miklós Szilveszter

Programtervező informatikus Bsc

*Budapest, 2021*

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>3</b>
1.1. Pac-Man . . . . .	3
1.2. Maci Laci . . . . .	4
1.2.1. Cselekmény . . . . .	4
1.3. Feladat . . . . .	4
<b>2. Felhasználói dokumentáció</b>	<b>5</b>
2.1. Rendszerkövetelmények . . . . .	5
2.2. Telepítés . . . . .	5
2.2.1. Java JRE telepítése . . . . .	5
2.2.2. A játék telepítése . . . . .	6
2.2.3. A játék indítása . . . . .	6
2.2.4. Játékmenet . . . . .	7
2.2.5. New Game/Új játék indítása . . . . .	9
2.2.6. Continue / Játék folytatása . . . . .	9
2.2.7. Highscores / Toplista . . . . .	10
2.2.8. Exit game / Kilépés . . . . .	10
<b>3. Fejlesztői dokumentáció</b>	<b>11</b>
3.1. Megoldandó feladat . . . . .	11
3.2. Használt környezet és eszközök . . . . .	11
3.2.1. Java . . . . .	11
3.2.2. Használt csomagok . . . . .	12
3.3. A program felépítése . . . . .	12
3.3.1. pac . . . . .	12
3.3.2. Display . . . . .	16

3.3.3. Entity . . . . .	16
3.3.4. Creatures . . . . .	19
3.3.5. Input . . . . .	25
3.3.6. States . . . . .	25
3.3.7. Tiles . . . . .	26
3.3.8. World . . . . .	27
3.3.9. Utils . . . . .	28
3.3.10. Gfx . . . . .	29
3.3.11. Highscore . . . . .	30
3.4. Tesztelés . . . . .	34
3.4.1. Menü tesztelése . . . . .	34
3.4.2. Játék tesztelése . . . . .	35
3.4.3. Fejlesztés és tesztelés során felmerült korábbi problémák, hibák	38
<b>4. Összegzés</b>	<b>40</b>
4.1. Áttekintés . . . . .	40
4.2. Továbbfejlesztési lehetőségek . . . . .	40
<b>Irodalomjegyzék</b>	<b>42</b>
<b>Ábrajegyzék</b>	<b>43</b>
<b>Táblázatjegyzék</b>	<b>44</b>
<b>Forráskódjegyzék</b>	<b>45</b>

# 1. fejezet

## Bevezetés

Szakdolgozatom témájának ötlete 2019 nyarán merült fel. Ebédszünetben kollégáimmal az egyetem végéről beszélgettünk, és témákat szerettünk volna kitalálni, mivel mindannyian közeledtünk az egyetem végéhez. Ekkor javasolta Bércesi Dániel kollégám, hogy egy régi játékot, a Pac-Man egy még régebbi rajzfilmsorozattal, a Maci Lacival ötvözzük, mert az biztosan érdekes téma lehet, és egy szórakoztató játék lenne az eredménye. Ezzel az ötlettel kértem fel Nagy Sára[1] tanárnőt, hogy legyen témavezetőm, aki elfogadta a felkérésem.

### 1.1. Pac-Man

A Pac-Man a Namco játéka, amelyet 1980-ban adtak ki a játéktermekbe, és később különféle platformokra. Igazi klasszikussá vált, a világ egyik legismertebb játéka. A játék labirintusban játszódik, amelyben egy kis sárga fejet kell irányítani. A cél az, hogy a pontokat megegye, és elkerülje a négy szellem ellenséget, amelyek el akarják kapni Pac-Mant. A dolgozatom célja, hogy ezt a játékot egy új formába öntsem. Pac-Man Maci Laci bőrébe bújlik, és Bubu társaságában almákat szerez meg, hogy az adott szintet teljesítse. Ellenfelei is új bőrből bújnak, és a vadőrök képében próbálják megakadályozni, hogy Maci Laci megszerezze az almákat. Míg Pac-Man egyetlen labirintusban kalandozott, hogy felvegye az összes pontot, Maci Lacira és Bubura egy egész erdő vár, hogy felfedezzék.

## 1.2. Maci Laci

A Maci Laci (eredeti címén Yogi Bear) 1961-től 1962-ig futott amerikai televíziós rajzfilmsorozat, amelyet a Hanna-Barbera Productions[2] készített. A tévéfilmsorozat forgalmazója Warner Bros. [3]. Először 1958-ban a Foxi Maxi című sorozatban jelent meg, még csak mellékszereplőként. Végül azonban nagyobb népszerűsége tett szert a Foxi Maxinál. 2010-ben 3D-s számítógépes animációs filmet készítettek belőle.

### 1.2.1. Cselekmény

Maci Laci és barátja, Bubu a Yellowstone Nemzeti Park erdőjében élnek (ami a rajzfilmben JellyStone Parkként szerepel). Maci Laci folyton butaságokat követ el a kíváncsisága miatt, és rossz szokása fosztogatni a kirándulók piknikkosarait. A park vadőre, Smith folyton ellenőrzi és szidja az elkövetett rendbontások miatt, amit sokszor neki kell helyrehoznia. Maci Laci barátnőjét Cindynek hívják, ő ritkábban jelenik meg a sorozatban.

## 1.3. Feladat

A fentebb említett Pac-Man játékot a rajzfilmsorozat stílusának megfelelően egy idősek és fiatalok számára egyaránt szórakoztató időtöltéssé alakítani. A program célja, hogy az idősebb közönséget hosszú nosztalgiazásokba idézze, míg a fiatalok reflexeit, logikáját fejlessze, hosszas szórakoztatás útján.

## 2. fejezet

# Felhasználói dokumentáció

### 2.1. Rendszerkövetelmények

A játék futtatásához szükséges hardverkövetelmények:

- 2GB memória
- Minimum 500 MB lemezterület a telepítendő szoftverek számára
- További lemezterület a Java JRE környezet telepítéséhez: 124 MB, Java JRE Update: 2 MB.

A játék futtatásához szükséges szoftverkövetelmények:

- Windows 10 operációs rendszer
- A Java JRE futtatási környezet telepítése, JAVA 8as verziótól kezdődően
- A felhasználó C:\Users \ (Felhasználónév) mappájába való írás joga a játék számára.

### 2.2. Telepítés

#### 2.2.1. Java JRE telepítése

A Java JRE telepítése elengedhetetlen a játék futtatásához. Telepítéséhez érdemes a legújabb csomagot letöltenünk az Oracle[4] weboldaláról az operációs rendszerünknek megfelelően.

### 2.2.2. A játék telepítése

A játék telepítéséhez csomagoljuk ki a mellékelt .zip állományt, mely tartalmazza az összes forrásfájlt a játék futtatásához.<sup>1</sup> A játékot a kicsomagolt parancsikkal tudjuk elindítani.

### 2.2.3. A játék indítása

A Java JRE megfelelő telepítése, valamint a játék telepítése után a játékot elindíthatjuk. A játék indítása a start.bat fájl, vagy a PacMan parancsikon futtatásával lehetséges. Indítás után megjelenik a játék főmenüje, mely a következő menüpontokat tartalmazza:

- New Game (Új játék indítása)
- Continue (Játék folytatása)
- Highscores (Toplista)
- Exit Game (Kilépés)



2.1. ábra. A játék főmenüje

---

<sup>1</sup>A tömörített állományhoz nem tartozik Java JRE telepítő.

### 2.2.4. Játékmenet

A játékos célja, hogy a játéktéren lévő összes almát, illetve kosarat felvegye, mielőtt elkapják a vadőrök. Maci Laci mindig a játéktér bal felső sarkában kezd, a vadőrök pedig a játéktéren egy véletlenszerűen választott üres mezőn. Maci Lacit a nyilakkal tudja a játékos irányítani. A játékos almák felvételével növeli a pontszámát, míg kosarak segítségével nem csak a pontját növeli, hanem a vadőröket azonnal visszaküldi a kiindulási pontjukra. A játéktérre hegyek illetve fák vesznek körbe. A játéktér minden pályán egy erdő, mely labirintust alkot.



2.2. ábra. Maci Laci, akit a játékos irányít

### Bubu, a segítő

A játékost a győzelem elérésében Bubu Maci is segíti. Bubu nem annyira elkötelezett, mint Maci Laci, így nem tudatosan veszi fel az almákat, hanem csak kóborol az erdőben. Útja során minden útba eső almát begyűjt. Bubu nem tud kosarakat begyűjteni, így azokat mindenképp a játékosnak kell felvennie. Bubut a vadőrök nem tudják elkapni.



2.3. ábra. Bubu Maci, akit a számítógép irányít



## Kétjátékos mód

Bubut alapértelmezetten a számítógép irányítja. Ezt azonban lehet játék közben változtatni. Egy második játékos a W, A, S, D gombok lenyomásával irányíthatja Bubu Macit. Ekkor a számítógép nem lép közbe. Amint a játékos felengedi a billentyűket, azaz nem irányítja Bubut, a számítógép újra átveszi az irányítást, és az eddig kijelölt almához indul el ismét. Fontos megemlíteni, hogy a kamera minden esetben Maci Lacit követi. Bubu képes a képernyőn kívülre menni. Ez nem befolyásolja a pályán haladásban, vagy az almák begyűjtésében.

## A vadőrök

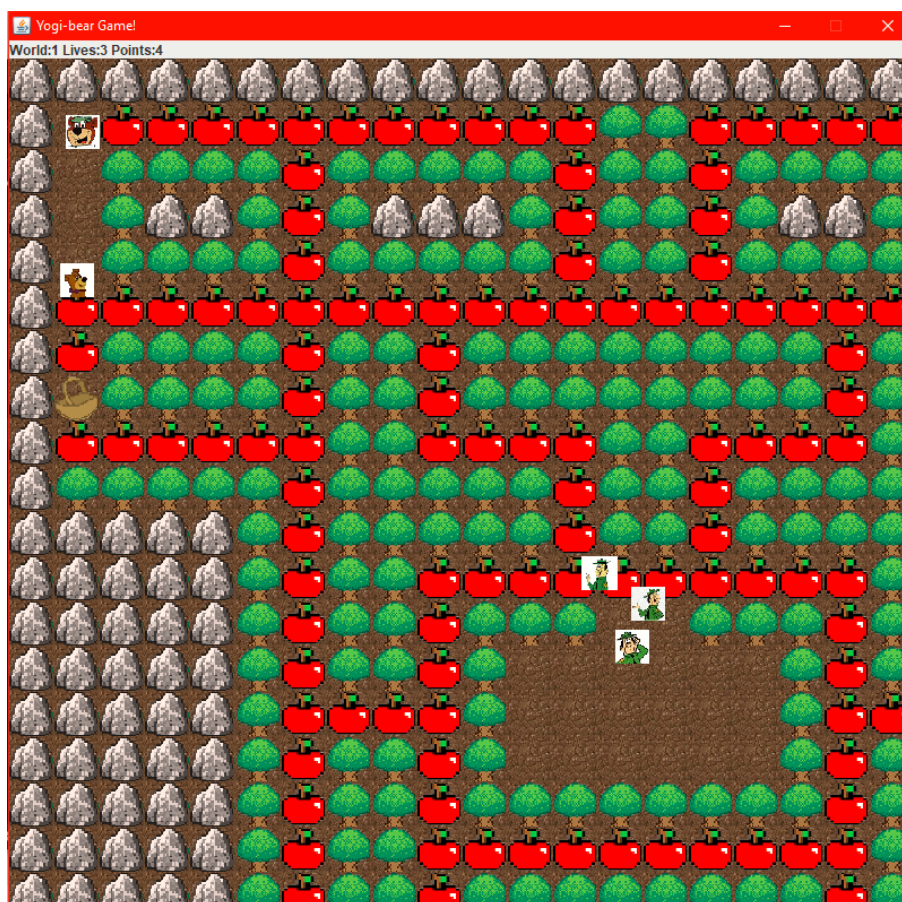
Célja elérésében a játékost négy vadőr akadályozza meg. Mind a négy vadőr más más stratégiát használ, hogy célját elérje, és elkapják Maci Lacit. A vadőrök nem foglalkoznak Bubuval, hiszen a káoszt Jellystoneban mindig Maci Laci okozza, így ő a fő célpontjuk. A vadőrök taktikája a következő: Az egyikőjük véletlenszerűen közlekedik az erdőben, hogy meglepje Maci Lacit, amikor nem számít rá. A második vadőr pontosan tudja, hogy hol van Maci Laci, így egyenesen neki veszi az irányt, és üldözőbe veszi. A harmadik és a negyedik vadőr két társuktól taktikusabb megközelítést alkalmaz, és megpróbálják bekeríteni Maci Lacit. Elkapni viszont nem feltétlenül fogják, csak csapdába ejteni. Maci Laci elkapását az esetek többségében a második vadőrre fogják bízni.



2.4. ábra. A vadőrök, akik Maci Lacit üldözik

### 2.2.5. New Game/Új játék indítása

A **New Game** gombra kattintva egy új játékot indíthatunk. Új játék során a játékos az első pályáról indul 0 ponttal, és 3 élettel. Az előző mentés ilyenkor törlődik. Ha a játékos kilép, vagy megnyeri az első pályát, a **Continue** gombbal tudja folytatni azt.



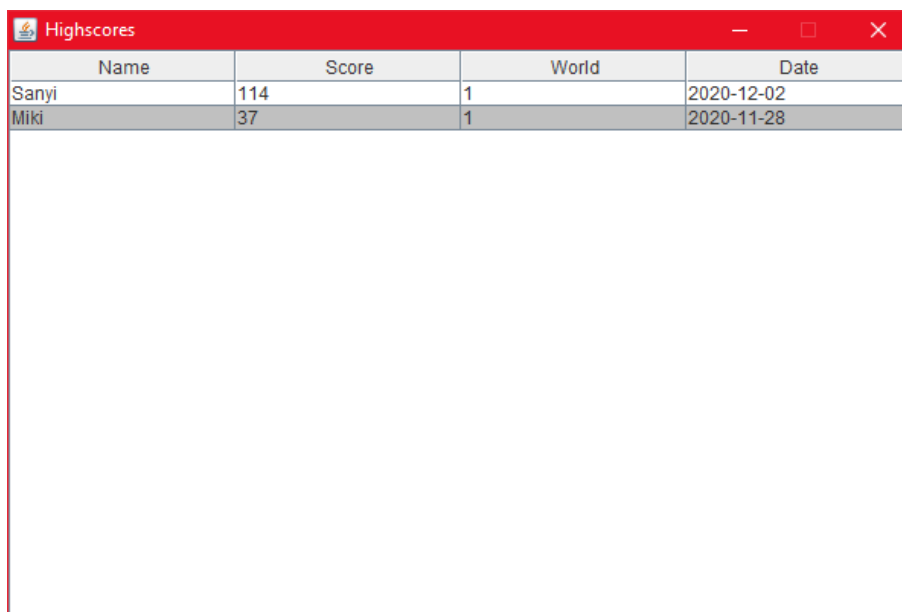
2.5. ábra. Új játék, első pálya

### 2.2.6. Continue / Játék folytatása

A **Continue** gombra kattintáskor a játékos az utolsó mentett játékot fogja folytatni. Ha sikerült teljesíteni egy adott pályát, és úgy kerül vissza a főmenübe, akkor a **Continue** gombra kattintva a következő pálya fog betöltődni. A játék minden **Exit Game** gombra kattintáskor elmenti az utolsó játékot. A mentésre kerülő adatok az életek száma, az utolsó látogatott világ, illetve a játékos pontszáma. Játék folytatása esetén ezen adatok kerülnek betöltésre, és így folytatódik a játék. Ezeket az adatokat új játék kezdése törli.

### 2.2.7. Highscores / Toplista

Megjeleníti az aktuális toplistát. A toplista a következő adatokat tárolja el: Név, Elért pontok száma, Világ, Dátum. A toplistába csak vereség árán lehet bekerülni. Amint a játékos elveszti utolsó életpontját, a játék kéri a játékos nevét. A felugró ablakon a Name mezőbe kattintva annak tartalma törlődik, és itt adhatja meg a játékos a nevét, melyet az Enter gombra kattintva küldhet el. Ha a játékos pontszáma bekerül a legjobb 10 közé, akkor láthatja nevét a toplistán.



Name	Score	World	Date
Sanyi	114	1	2020-12-02
Miki	37	1	2020-11-28

2.6. ábra. A toplista

### 2.2.8. Exit game / Kilépés

Ez a gomb elment minden eddigi változást a játékban, így véglegesíti az utolsó mentést is. Mentés után a játék leáll.

## 3. fejezet

# Fejlesztői dokumentáció

### 3.1. Megoldandó feladat

Egy olyan felhasználóbarát játék készítése Java nyelven, melyet kicsik és nagyon egyaránt élveznek. Játék kialakítása során főbb szempontok a kinézet, funkcionalitás, és egyértelműség. Fontos, hogy a játékos tudja, hogy melyik vadőr merre fog közlekedni illetve, hogy mikor vegyen fel kosarat a vadőrök kezdőhelyükre való visszaküldéséhez. Mindezt egy könnyen kezelhető felület kialakításával, és igényes kialakítással szeretném véghezvinni.

### 3.2. Használt környezet és eszközök

#### 3.2.1. Java

A fejlesztéshez ideális nyelv a Java, két főbb szempont miatt. Az első, hogy az objektum orientált nyelv ideális egy játék fejlesztéséhez. A második szempont pedig, hogy a java alpból tartalmazza a fejlesztéshez szükséges csomagokat, így nem kell külön csomagokat telepítenünk, frissítenünk.

A nyelv mellé tartozik egy IDE - fejlesztői környezet - is. A NetBeans (újabbban Apache NetBeans)[5] nevű fejlesztői környezetet használtam. Ez a környezet ingyenesen beszerezhető, és minden fontos beépített elemet tartalmaz, melyre szükségem lehet. A dokumentációban használt UML diagrammok készítéséhez Umletet [6] használtam.

### 3.2.2. Használt csomagok

A fejlesztéshez használt csomagok a következők:

- gfx - Játéktér megjelenítésért felelős
- javax.imageio - Képek feldolgozásáért felelős
- javax.swing - Gombok és menüelemek megjelenítéséért felelős
- awt - Gombok és menüelemek megjelenítéséért felelős
- io - Fájlok olvasásáért felelős
- nio - Mappaszerkezet vizsgálatáért felelős
- util - Listákért felelős
- time - Idővel kapcsolatos műveletekért felelős

A gfx, imageio, swing, awt csomagok legfőképp a megjelenésért és megjelenítésért, az io és nio csomagok a mentésért és betöltésért, a time és a util csomagok pedig az egyéb kisebb feladatokért felelősek.

## 3.3. A program felépítése

### 3.3.1. pac

#### Launcher és Handler

A pac csomag feleltethető meg a program törzsének. Itt található a main függvény, a **launcher** osztályban. Ez a program belépési pontja, itt példányosítunk egy új **Menu**-t. A **Handler** felelős a játék, és a világ összekötéséért. Így a játék a játékosokkal, a világ pedig a megjelenítendő pályával foglalkozik.

#### Menu

A **Menu** felelős a főmenüért. Nem csak a funkcionalitásáért, de a megjelenítésért is. Bár ellenzendő a gui és logikai elemek egy osztályba való fésülése, mivel ez egy konstans elem a játék során, így ettől eltekintettem. Itt tárolom a játékos

aktuális tulajdonságait is, melyek a játékos aktuális játékát határozzák meg, így a jelenlegi világ számát, életek számát, pontokat, illetve egy kapcsolóban azt is, hogy mentett játék következik-e, avagy egy már meglévő játékot folytatna a játékos. Ez a **Continue** gomb lenyomásakor fontos. A menü 4 gombból épül fel, mindegyik lambdaként meghívja a megfelelő metódusokat, az adott feladatok végrehajtásához. Így a új játék kezdésénél mindent alaphelyzetbe állítunk és egy új játékot indítunk, folytatásnál előbb említett kapcsoló alapján betöltünk, majd folytatunk, toplistánál a toplistát jelenítjük meg, kilépésnél pedig mentünk és kilépünk. A gombok funkcióinak beállítása után beállítjuk a menü megjelenítését is. Az itt használt metódusok a következők:

- Getterek és setterek
- `increaseWorldId()` - világ számát növeli eggyel
- `resetWorldId()` - 1re állítja vissza a világ számát
- `lowerLives()` - csökkenti az életek számát eggyel
- `resetLives()` - 3ra állítja vissza az életek számát
- `setScore(int newScore)` - beállít egy új pontszámot
- `resetScore()` - 0ra állítja vissza a pontszámot
- `getSavedDetails()` - Megnézi, hogy szerepel-e a felhasználó **home** könyvtárában a `.pacmangame/last.txt`. Ha igen, akkor abból beolvassa, majd betölti az adatokat. Ha nem létezik a `last.txt` a megadott helyen, akkor létrehozza azt, és feltölti az alapvető adatokkal. Ezt a függvényt játék folytatásához használjuk.
- `saveDetails()` - Elmenti a jelenlegi adatokat a felhasználó **home** könyvtárában lévő `.pacmangame` mappába egy `last.txt` nevű fájlba. A fájl pontosvesszőkkel tagolva tartalmazza a világot, az életek számát, illetve a pontszámot.

```
1 public void saveDetails(){
2     try{
3         String path = (System.getProperty("user.home") + "/" +
4             "pacmangame/last.txt");
5         RandomAccessFile input = new RandomAccessFile(path, "rw");
6         input.seek(0);
7         input.writeBytes("" + worldId + ";" + lives + ";" + score + ";"
8             );
9     } catch (IOException e){
10        System.out.println("Error with last visited world file");
11    }
12 }
```

### 3.1. forráskód. Mentések betöltése

## Game

A **Game** osztály felelős a játék futásáért. Itt található a játékciklus. Példányosításkor elindul a Run() metódus, mely először meghívja az init() metódust. Az init() metódus felelős a játék inicializálásáért. Itt jön létre a felugró ablak, illetve itt jön létre a billentyűk lenyomásáért felelős **keyManager** is. Beállítjuk továbbá a handlert, és a kamerát, valamint a játékstátuszt is. Inicializálás után elkezdődik a játékciklus, mely fix 60 képkocka/másodpercre van állítva. Ez garantálja azt, hogy a játék elemei frissüljenek 1/60 másodpercenként.

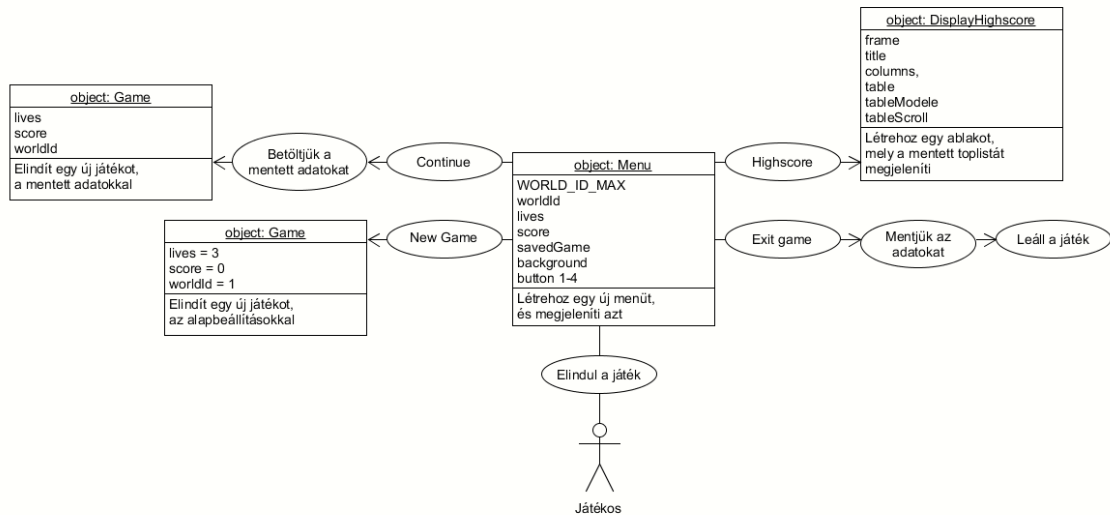
```
1 @Override
2 public void run(){
3     init();
4
5     int fps = 60;
6     double timePerTick = 1000000000 / fps;
7     double delta = 0;
8     long now;
9     long lastTime = System.nanoTime();
10    long timer = 0;
11
12    while(running){
13        now = System.nanoTime();
14        delta += (now - lastTime) / timePerTick;
```

```
15         timer += now - lastTime;
16         lastTime = now;
17
18         if(delta >= 1){
19             try{
20                 tick();
21                 render();
22                 delta--;
23             }catch(Exception e){}
24         }
25
26         if(timer >=10000000000){
27             timer = 0;
28         }
29     }
30     stop();
31 }
```

### 3.2. forráskód. Játékciklus

A frissítés a `tick()` illetve a `render()` metódusokból áll. A `tick` metódus a mozgásokat, illetve a játékelemek közötti interakciókat figyeli, míg a `render()` metódus ezeket kirajzolja. Mindez egy `while(running)` ciklusba van téve, így amint a játék leáll kilépés vagy játékos halála esetén, kilépünk a ciklusból, és meghívódik a `stop()` metódus. A `stop()` metódus leállítja a játékot. A **Game** osztályban két további fontos metódus található a gettereken és a settereken kívül. Ezek az `increaseScore()` illetve a `closeGame()` metódusok. Az `increaseScore()` adja át a **Menu**-nek a játékos pontszámát, illetve írja ki a játéklablak tetején lévő label-be az aktuális pontszámot. A `closeGame()` metódus a játék leállításáért felel. Ezt a metódust hívják meg a játék különböző elemei, ha a játék leállítását szeretnék. Ez előfordulhat akkor ha nyer, vagy veszít a játékos.





3.1. ábra. A játék főmenüjének működése

### 3.3.2. Display

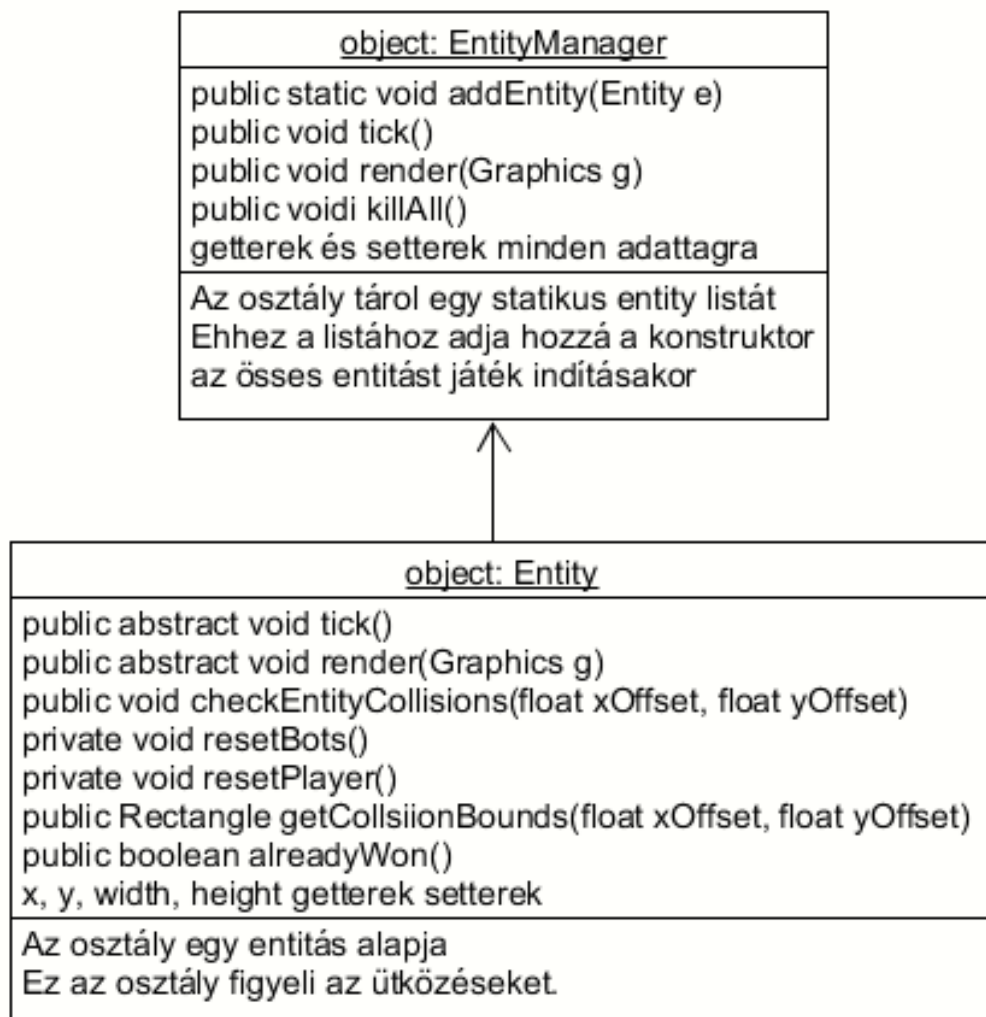
A **Display** csomag egyetlen osztályt tartalmaz, a **Display** osztályt. A display osztály szerepel a játék megjelenítéséért. Itt jön létre a játék indításakor felugró ablak, illetve az azon szereplő label, mely a játékos információit (világ száma, életek száma, pontok száma), illetve a játéktérteret rajzolja ki. A játéktér egy canvas elem segítségével rajzolódik ki. Mindez a `createDisplay()` metódusban valósul meg. Ezen kívül az osztály csak `getCanvas()`, `getFrame()`, `getLabelText()` és `setLabelText(String text)` metódusokat tartalmaz, melyek az adott tulajdonságokat kéri le, illetve módosítják.

### 3.3.3. Entity

Az **Entity** csomag két osztályból áll. Az **Entity** és az **EntityManager** osztályokból. Az **Entity** osztály egy-egy mozgó játékelemet reprezentál, ezek lehetnek:

- Maci Laci, player
- Bubu, friendlyBear
- Vadőr, angry, helpful, problem, smith

Az **EntityManager** osztály ezen elemek frissítéséről, illetve tárolásáról gondoskodik.



3.2. ábra. Az Entity és EntityManager osztályok kapcsolata

## Entity

Az **Entity** osztály egy-egy mozgó játékelemet reprezentál. Ezen játékelemeknek az x, illetve y koordinátáján kívül a szélességét, illetve magasságát, valamint az elem határait egy négyzetként. Ezt az osztályt valósítja meg minden karakter.

Az **Entity** osztály két abstract metódust tartalmaz, melyek a render() illetve a tick() metódusok. Ezeket a metódusokat minden karakter külön-külön implementálja, hiszen minden karakter máshogy néz ki, illetve máshogy mozog, viselkedik.

Ez az osztály tartalmazza a checkEntityCollisions(float xOffset, float yOffset) metódust, mely az ütközéseket vizsgálja. Az ütközéseket négy részre bontottam fel. Játékos és alma, játékos és kosár, Bubu és alma, valamint játékos és minden ellenfél.

Minden rész az adott elágazásban megvizsgálja, hogy az adott elemek ütköznek-e, és ütközéstől függően más-más metódusokat hív meg. Almák és kosarak esetén növeljük a pontszámot, és az adott pályakockát földre állítjuk át. Kosár esetén az ellenfeleket visszaküldjük kiindulási helyükre. Bubu csak az almákat képes felvenni, így itt csak az almákat vizsgáljuk. Ha a játékos ellenfelekkel ütközik, akkor mindenkit visszaküldünk a kezdőpontjára Bubu kivételével, és csökkentjük a játékos életeinek számát. A helyzetek visszaállításáért a `resetBots()` illetve a `resetPlayer()` metódusok gondoskodnak.

```
1 if(e instanceof Player && e.getCollisionBounds( 0f, 0f).intersects(  
    getCollisionBounds(0f, 0f)) && handler.getWorld().getTile(((int)  
    e.x/40), ((int)e.y/40)).getId() == 3){  
2     this.handler.getWorld().setTile(((int)(e.x/40), (int)(e.y/40),  
        0);  
3     this.handler.getGame().increaseScore();  
4     if(this.alreadyWon()){  
5         handler.getGame().getMenu().increaseWorldId();  
6         handler.getGame().closeGame();  
7     }  
8  
9 }
```

### 3.3. forráskód. A játékos felvesz egy almát

A `resetBots()` metódus lekéri az adott botok x és y kezdőkoordinátáit, majd az aktuális x és y koordinátáikat ezekre a lekért koordinátákra állítja be.

A `resetPlayer()` metódus visszaküldi a játékost a pálya bal felső sarkába, mert minden pálya ott kezdődik. Az osztály tartalmaz továbbá egy `getCollisionBounds(float xOffset, float yOffset)` metódust, mely egy négyzetet ad vissza, ami az adott entitás ütközési határait tartalmazza, illetve egy `alreadyWon()` metódust, mely a pályán lévő almákat (és kosarakat) számolja meg. Ha a visszakapott érték 0, akkor a játékos (és Bubu) felvette a pályán található összes almát és kosarat, így teljesítette a pályát. Az **Entity** osztály további metódusai a getterek és setterek az x, y, width, és height attribútumokra.

## EntityManager

Az **EntityManager** osztály a mozgó játékelemek frissítésért és tárolásáért felel. Legfőbb eleme a statikus Entityket tároló ArrayList. Ez fogja tárolni az összes játéktéren lévő karaktert. Inicializálás során létrehozuk az összes karaktert a konstruktorban, majd hozzáadjuk őket ehhez a listához. Ehhez egy `addEntity(Entity e)` függvényt használunk. Az **EntityManager** további két függvénye a `tick()`, illetve a `render(Graphics g)` függvény, melyek minden entitásra meghívják azok `render()` és `tick()` függvényeit.

### 3.3.4. Creatures

Az **entity** csomagon belül található egy **creatures** csomag, mely a **Creature**, **Player**, **FriendBear**, illetve a vadőrökhöz tartozó osztályokat, valamint az **Astar** és **Node** osztályokat tartalmazza. Ez a ycsomag felel az egyes karakterek mozgásáért, és viselkedéséért.

## Creature

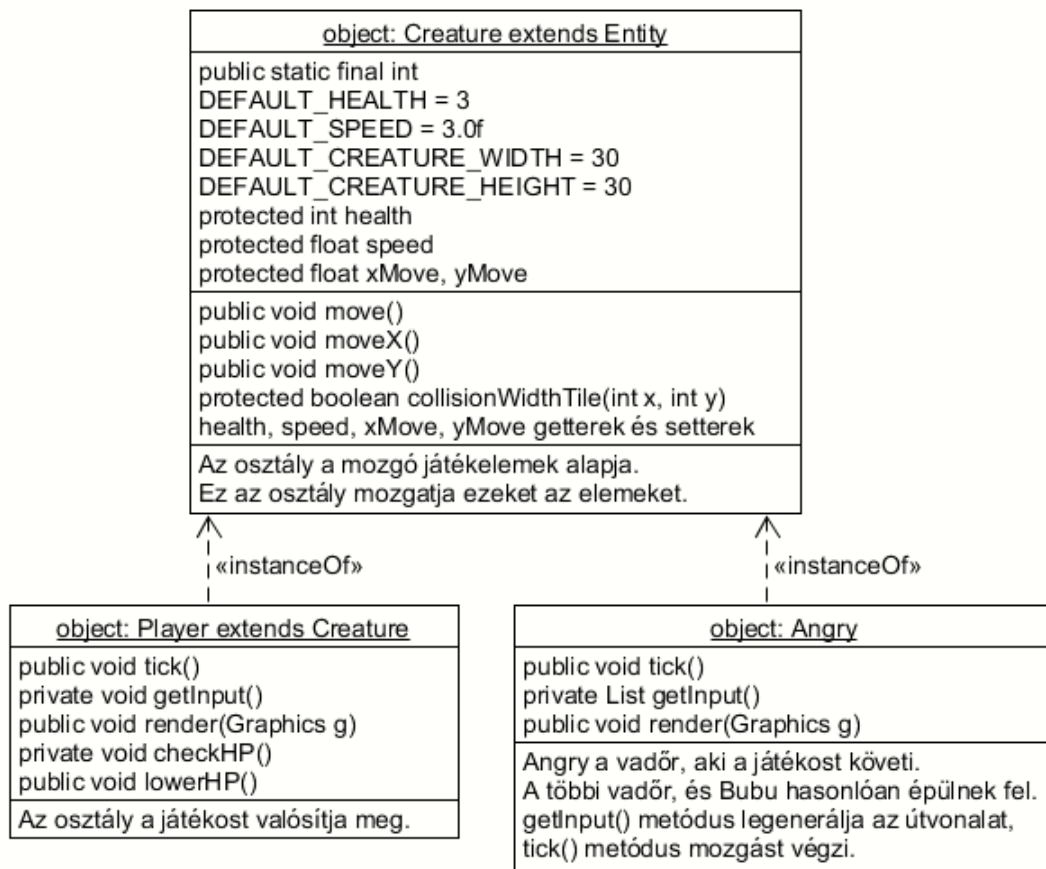
A **Creature** osztály az **Entity** osztályt terjeszti ki. Négy statikus végleges adattaggal rendelkezik, melyek a

- `default health = 3`
- `default speed = 3.0f`
- `default creature width = 30`
- `default creature height = 30`

Az élet a játékos miatt 3 inicializáláskor. A kezdő sebesség minden karakterre vonatkozik, de a vadőrök mozgásánál, a saját osztályukban a mozgási sebesség mellé tartozik egy 0.8as szorzó is. Ez biztosítja majd, hogy a játékos gyorsabb, mint a vadőrök, így is növelve az esélyeit. A karakter szélessége és magassága a csempék szélességéhez és magasságához van igazítva. Így bárhonnán felveszi a játékos az alját, mégis könnyebben tud közlekedni a pályán.

A **Creature** osztály három fő részből áll: ütközés ellenőrzése, mozgás ellenőrzése, mozgás. A mozgás egymásbaütközés vizsgálással kezdődik. Meghívódik az **Entity**

osztály `checkEntityCollisions(float xOffset, float yOffset)` függvénye. A lekezelt ütközések után a karakter mozog. Ez a `moveY()` és `moveX()` függvényekkel történik meg. A függvény ellenőrzi, hogy az adott karakter `xMove` vagy `yMove` attribútuma változott-e. Ha igen, és nem ütközik az adott karakter falba, azaz a `collisionWithTile(int x, int y)` függvény hamis értékkel tért vissza, akkor az `xMove` és `Ymove` attribútumoknak megfelelően fel vagy le, illetve balra vagy jobbra mozgatja el a karaktert.



3.3. ábra. A Creature osztály, és az azt megvalósító osztályok kapcsolata

## Player

Ez az osztály felel a játékosért. Tárolja az életek számát, pozícióját, határait, szélességét és magasságát. Két fő metódusa van: `tick()` és `render()`. A `tick()` metódus figyeli a bemenetet a `getInput()` metódus segítségével, mely a **KeyManager** osztályt használja. Ennek megfelelően mozgatja a játékost a pályán a `move()` metódus, melyet a **Creature** osztályban definiáltunk. Mozgás után a kamerát állítja be, hogy a játékost figyelje. Így nem csúszik el a kamera a játékos felől. Ez után ellenőrzi a játékos életét. Ha ez elfogy, akkor vége a játéknak. Ezt a `checkHP()` függvény teszi meg. Játék végén lenullázza a világ számát, lekéri a pontszámot, majd létrehozza az `EndGamePanel`-t. Ez után bezárja a játékot.

Ebben az osztályban található továbbá a `lowerHP()` függvény is, mely a játékos életét csökkenti eggyel, majd a képernyőn lévő `label`-en átírja az életek számát az új értékre.

```
1 private void getInput(){
2     xMove = 0;
3     yMove = 0;
4
5     if(handler.getKeyManager().up){
6         yMove = -speed;
7         this.handler.getGame().getKeyManager().pressedKeys.add(
8             "u");
9     }
10
11     if(handler.getKeyManager().down){
12         yMove = speed;
13         this.handler.getGame().getKeyManager().pressedKeys.add(
14             "d");
15     }
16
17     if(handler.getKeyManager().left){
18         xMove = -speed;
19         this.handler.getGame().getKeyManager().pressedKeys.add(
20             "l");
21     }
22
23     if(handler.getKeyManager().right){
24         xMove = speed;
25     }
26 }
```

```
21         this.handler.getGame().getKeyManager().pressedKeys.add(  
22             "r");  
23     }  
24     if(handler.getKeyManager().escape){  
25         handler.getGame().closeGame();  
26     }  
27 }
```

### 3.4. forráskód. A játékos bemenet figyelése

## Astar és Node

Az **Astar** osztály az A\* algoritmust megvalósító osztály. A Node osztály egy segédosztálya az **Astar** osztálynak. Ez a két osztály együtt felelős az útkereső algoritmusért, amely megadja Bubunak és a vadőröknek, hogy hogyan mozogjanak. A **world.World** osztály a játékeret Nodeokká alakítja a `nodificateMap()` metódus segítségével, majd ezt átadjuk az **Astar** konstruktorának.

A konstruktor várja a sorok illetve oszlopok számát, valamint a kezdő és a cél Node-ot. Ezen kívül várja a függőleges/vízszintes illetve átlós költséget. Ez a konstruktor jelen esetben felül van definiálva egy másik konstruktorral, mely a függőleges/vízszintes költségeket 1-nek, az átlósakat 10nek adja meg. Jelen esetben ez azt jelenti, hogy a vadőrök nem közlekednek átlósan, csak függőlegesen és vízszintesen. Az **Astar** példányosításakor megy végbe a heurisztika kiszámolása is. Mivel minden kockához újraszámoljuk a legrövidebb utat (hogy effektív legyen az útkeresés), így ez a legjobb hely a heurisztika kiszámolásához. A heurisztikát a `calculateFinalCost()` metódusban használjuk fel, mely a végső értékét adja meg egy adott Node-nak. Ezt az 'f' attribútum fogja tárolni. Ezt fogja összehasonlítani `PriorityQueue<Node>` a felüldefiniált `compare(Node 0, Node 1)` metódusban, hogy a `PriorityQueue<Node>` megfelelő sorrendet állítson be az útvonalak között.

Példányosítás után be kell állítani a Node-ok értékeit, így megadhatjuk, hogy mely területeken közlekedhetnek a gépi játékosok. Jelen esetben ha 1 vagy 2 értéket talál a térképen, akkor az fa vagy szikla, így `setBlock(int row, int col)` függvénnyel beállítjuk, hogy ott nem közlekedhetnek. Végül meghívjuk a `findPath()` metódust, mely a tényleges útkeresést végzi el.

## Útkeresés

Az útkeresés a legfontosabb része a játéknak. Ezen áll, vagy bukik a játékelmény. Ha a az ellenfelek túl gyorsak, túl jól mozognak a játékos lehetőségeihez képest, akkor a játékos a játék elejétől kezdve vereségre van ítélve. Ha túl gyengék az ellenfelek, akkor a játékos könnyedén végigjátszik minden pályát, és nem fog kihívást okozni a játék. Útkereséshez a már korábban említett A\* algoritmust használtam, mellyel Gregorics Tibor[7] tanár úr Mesterséges Intelligencia előadásán ismerkedtem meg. Az algoritmus a következőképp épül fel:

---

### Algoritmus 1 A\* algoritmus

---

**Funct** public List<Node>findPath()

- 1: Legyen egy nyitott, és egy zárt listánk.
  - 2: Adjuk hozzá a nyitott listához a kiinduló Node-ot
  - 3: **while** ( A nyitott lista nem üres ) **do**
  - 4:   Vegyük ki a nyitott listából a következő elemet, ez lesz a jelenlegi elem.
  - 5:   Adjuk hozzá a jelenlegi elemet a zárt listához
  - 6:   **if** A jelenlegi elem a cél **then**
  - 7:     **return** Elemhez vezető utat visszaadjuk a szülők segítségével
  - 8:   **else**
  - 9:     Adjuk hozzá a környező Nodeokat a nyílt listához úgy, hogy a jelenlegi elem a szülő
  - 10:   **end if**
  - 11: **end while**
  - 12: **return** Üres lista, hiszen ha megtaláltuk, akkor már visszaadtuk a 7. lépésben
- 

A gépi játékosok az így visszkapott legrövidebb utat veszik a játékoshoz. Ám minden vadőr módosít ezen, hogy nehezebb legyen a játék. Ezt a nehézséget azal kompenzáltam, hogy a vadőrök egy kicsit lassabbak, mint Maci Laci. Ezeket a változtatásokat a vadőröknél tovább taglaltam.

```
1 private List getInput(){
2     Node [][] world = handler.getWorld().nodificateMap();
3     Astar astar = new Astar(world.length, world.length,
4         world[(int)(y / Tile.TILEWIDTH) ][(int)(x / Tile.
            TILEHEIGHT) ],
```



```
5         world[(int)(handler.getWorld().getEntityManager().
6             getPlayer().getY() / Tile.TILEWIDTH)]
7             [(int)(handler.getWorld().getEntityManager
8                 ().getPlayer().getX() / Tile.TILEHEIGHT)
9                 ]);
10
11     astar.setNodeValues(handler.getWorld().getWorld());
12     List<Node> nodes = astar.findPath();
13     if(nodes.size() > 1){
14         nodes.remove(0);
15     }
16     return nodes;
17 }
```

3.5. forráskód. Útvonalkeresés

## Vadőrök

Minden vadőr külön osztállyal rendelkezik, hiszen minden vadőr más taktikát használ Maci Laci elkapásához.

- Angry - Pontosan követi Maci Lacit, és a legrövidebb utat választja minden esetben.
- Helpful - Alulról próbálja meg Maci Lacit bekeríteni, Astarnál `destY += 4`
- Problem - Jobbról próbálja meg Maci Lacit bekeríteni, Astarnál `destX += 2`
- Smith - Véletlenszerűen választ ki egy mezőt a játéktéren, és oda megy. Ha odaért, új célt választ.

Ezen kívül a vadőrök teljesen azonosak. Kiterjesztik a `Creature` osztályt, van egy statikus `path` Listájuk, mely az útvonalat tartalmazza, illetve statikus `destinationTileX` és statikus `destinationTileY` adattagjaik. Konstruktoruk a méreteiket és pozíciójukat állítja be. Függvényeik:

- `tick()` - lekéri a bemenetet, mely a `path` lista első eleme. Ennek megfelelően mozog
- `getInput()` - útvonalat keres az előző fejezetben említett módon.

- `render(Graphics g)` - A vadőr pozíciójára kirajzolja a vadőr képét.

A vadőrök hátránya Maci Lacival szemben, hogy lassabbak, mint Maci Laci. Ezt a `tick()` metódusban implementáltam, a vadőrök mozgásánál. Minden irányú mozgásukhoz tartozik egy 0.8f szorzó, mely a sebességüket az egységes sebesség 80%-ra csökkenti.

## FriendBear

Ez az osztály valósítja meg Bubut. Bubu viselkedése megegyezik Smith viselkedésével, azzal a különbséggel, hogy, ha egy második játékos lenyomja a W,A,S,D gombok egyikét, akkor ezt veszi alapul, és a gép addig nem változtat irányt. Így nem "küzd" a játékos a gép ellen az irányításért.

### 3.3.5. Input

Az **Input** csomag egyetlen osztályt tartalmaz, a **KeyManager** osztályt, mely a billentyű bemeneteket kezeli le, ezeket egy `final boolean[] keys`-ben tárolja. A `tick()` metódus figyeli, hogy melyik gombot nyomjuk le a billentyűzeten, és az annak megfelelő tömb elemet igazra állítja át. A billentyű felengedésénél a `keyReleased(KeyEvent e)` metódus segítségével az az elemet visszaállítja `false`-ra.

### 3.3.6. States

A **states** csomag szülőosztálya a **State** osztály. Egy egyszerű absztrakt osztály, mely egy `private static State currentState` adattaggal, illetve egy `protected Handler` handler adattaggal rendelkezik. Metódusai:

- `public State(Handler handler)` - konstruktor
- `public static void setState(State state)` - state beállítása
- `public static State getState()` - state lekérése
- `public abstract void tick()`
- `public abstract void render(Graphics g)`

## GameState

A **GameState** osztály a **State** osztályt terjeszti ki. Egy privatre final World world adattaggal rendelkezik, mely a világ számát tárolja. Ezt a beolvasáshoz használja a konstruktorban. A példányosításhoz szükséges egy Handler handler, és egy int worldId. A konstruktor meghívja a super metódust, melynek átadja a handlert, majd a world adattagot egy új Worldként példányosítja, a worldId segítségével. Ez után a handler.setWorld(WorldId) parancs beállítja a világ id-t. Két további metódusa van, a tick, mely meghívja a world.tick() metódust, illetve a render(Graphics g), mely meghívja a world.render(g) metódust.

### 3.3.7. Tiles

A tiles csomag a pálya alkotásáért felelős. Itt találhatók a pályán látható elemek, mindegyik elem külön osztályban. A **Tile** osztály tárolja el ezeket az elemeket, és itt példányosul minden elem. Minden elemnek van egy BufferedImage texture, és egy protected final int id adattagja. Négy függvénye van a **Tiles** osztálynak, melyek:

- Tile(BufferedImage texture, int id) - konstruktor, inicializálás után tiles tömbbe felveszi az adott elemet
- public void render(Graphics g, int x, int y) - Kirajzolja az adott elemet x, y koordinátájú helyre
- public boolean isSolid() - mely alaphól false értéket ad vissza.
- public int getId() - visszaadja az id-t.

A **Tiles** csomag minden további osztálya egy-egy "csempét" valósít meg, és mind a Tile osztályt terjeszti ki. Minden osztály konstruktora meghívja a super() metódust a megfelelő textúrával az Assets osztályból, illetve a kapott id-vel. Az osztályok felüldefiniálják az isSolid() metódust is attól függően, hogy az adott terület tömör, avagy bejárható. Az osztályok a következők:

- AppleTile
- BasketTile

- DirtTile
- RockTile
- TreeTile

### 3.3.8. World

A **World** csomag a játék pályáját hozza létre, és tölti be. Adattagjai a handler, width, height, spawnX, spawnY, melyek a játékos kezdőpontját tárolják, int[][] tiles, mely a pályát magát tárolja, entityManager, mely a játékelemekért felel, illetve a gépi játékosok kezdőhelyei.

A konstruktor betölti a világot a loadWorld(path) metódussal, beállítja a vad-őrök kezdőhelyeit, definiálja az entityManagert a megfelelő paraméterekkel, valamint beállítja a játékos tulajdonságait is. A tick() és render() metódus itt is megtalálható, a tick() az entityManager tick() metódusát hívja meg, míg a render() metódus a kamera által megjelenített területet rajzolja ki a képernyőnkre. A Tile getTile(int x, int y) és void setTile(int x, int y, int id) metódusok egy, a private int[][] tiles tömbben tárolt területet kérdeznek le, és állítanak be.

A pálya betöltéséért a void loadWorld(String path) metódus felel, mely a Utils.loadFileAsString(path) metódust hívja meg. Ezt követően sorokra bontja fel a kapott stringet, és így végzi el a beolvasást. A pálya szélessége és magassága után a játékos kezdőpontja jön, utána a pálya inicializálása a tiles tömb egészekkel való feltöltésével.

Minden pálya txt fájlja a következőképp épül fel:

- Két egyező szám egymás után. Ezek a pálya méretét jelölik. Fontos, hogy a pálya négyzet alakú legyen.
- A játékos kezdőpontja, ez relatív távolság a térkép bal felső sarkától. Ez a szám leosztandó a pályacsempék méretével.
- A következő n sor a pálya. A fájlban a pályaelemek ugyan úgy helyezkednek el, mint ahogyan azok a játékban meg fognak jelenni. Egy-egy elem szóközzel van elválasztva egy sorban. A pályaelemek számkódolva vannak, a következőképpen: 0 - föld, 1 - fa, 2 - szikla, 3 - alma, 4 - kosár.

A `public Node[][] nodificateMap()` metódus az **Astar** és **Node** osztályok egy segédmetódusa, mely minden területhez egy Nodeot hoz létre.

A `public int[] findSpawnPoint()` metódus a gépi játékosok számára keres kezdőhelyet. 4szer próbálja beállítani a kezdőhelyet, negyedjére biztosan beállítja. Ez azért van, hogy ne minden vadőr egy helyen kezdje a játékot, hanem szét szórva.

A `public int countApples()` metódus megszámolja a pályán található almákat, majd visszaadja ezt egy egész számként, míg a `public ovid killEntities()` metódus eltávolítja az összes entitást az `entityManager` listájából, a `setter` segítségével.

Az osztály további metódusai a getterek és setterek a `spawnX`, `spawnY`, `width`, és `height` adattagoknak. Az `entityManager`nek csak `getter` metódusa van, míg a `tiles[][]` mátrixot `getWorld()` metódussal tudjuk lekérdezni.

### 3.3.9. Utils

A **Utils** csomag egy segéd csomag, mely nevéből is következik. Egyetlen osztályt tartalmaz, a **Utils** osztályt. Ennek két metódusa van: `public static String loadFileAsString(String path)`, és `public static int parseInt(String number)`.

A `loadFileAsString(String path)` metódus egy beolvasó metódus. Megpróbálja beolvasni a kapott útvonalon lévő fájlt, majd egy `toString()` metódus segítségével a beolvasott adatokat visszaadni. Mivel a játék egy jar fájl futtatásával indul, így itt fontos megemlíteni, hogy a beolvasott fájlok közül nem mind a jar része. A pontokat, illetve a mentéseket a C: meghajtón a Users könyvtár jelenlegi felhasználójának mappájában, egy `.pacmangame` mappában tárolja a játék. Fontos továbbá, hogy jar fájl nem képes csak streamként beolvasni fájlokat. Így a streamet null értékig olvassuk, majd ezt egy stringként tudjuk továbbadni. Mivel a fájlok viszonylag kis méretűek, így nem kell fájl mérettel, és a String méret limitációival foglalkoznunk.

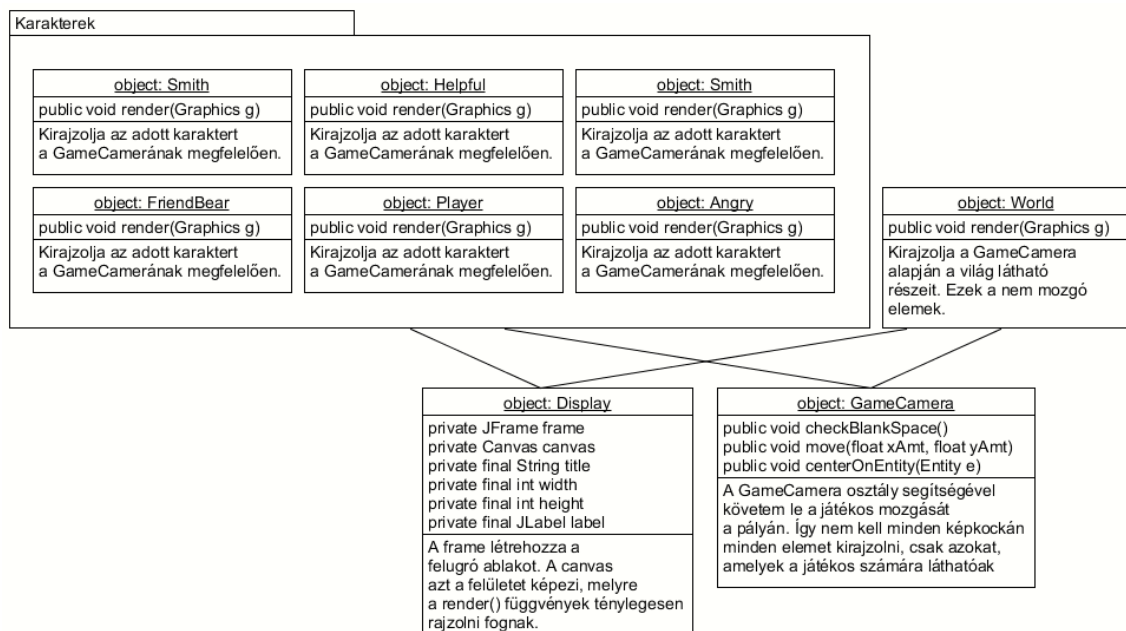
A `parseInt(String number)` egy egész szám konvertáló függvény. Ez pusztán az `Integer.parseInt()` függvény felüldefiniálásnak szerepét játssza, szintén a jar fájl, és beolvasási függőségek miatt.

### 3.3.10. Gfx

A **gfx** csomag a megjelenítendő képekért felelős. Négy osztályból áll: **Assets**, mely a mezők és a karakterek kinézetét tölti be a resourceok közül, és állítja azt be a megfelelő mezőnek és karakternek, **GameCamera**, ami a játék kameráját irányítja, mindig a játékosra fókuszál, és azzal együtt mozog, **ImageLoader**, ami egy képet olvas be egy kapott elérési út alapján, és **SpriteSheet**, ami egy kapott képet SpriteSheet-re konvertál át, így könnyebb kivágni belőle adott elemeket. Mivel a játék során használt képeket egyetlen spritesheetben tároltam (a menü háttér kivételével), így ebből kivágással a legegyszerűbb az adott képeket kezelni.

A játék kirajzolása a következőképp épül fel:

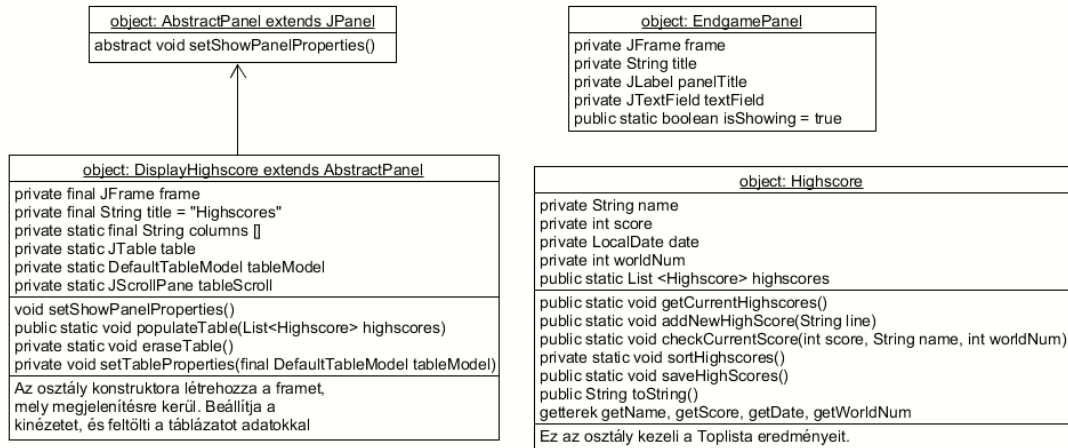
- Létrejön játék indulásakor egy Game
- A Game objektum rendelkezik egy Display objektummal
- A Display objektum valósítja meg a framet, illetve benne egy canvast
- Minden tick()-en belül minden elemre meghívunk egy render() metódust
- A render metódus kirajzolja erre a canvasre az adott objektumot x, y koordináta, szélesség és magasság alapján



3.4. ábra. A kirajzolással foglalkozó osztályok kapcsolatai

### 3.3.11. Highscore

A highscore csomag valósítja meg a toplistát. Négy osztályból áll, ezek a **AbstractPanel**, **DisplayHighscore**, **EndgamePanel**, és **Highscore** osztályok. Ezek következőképp kapcsolódnak egymáshoz:



3.5. ábra. A Toplistát kezelő osztályok kapcsolata

Az **AbstractPanel** egy abstract osztály, mely egyetlen abstract metódust tartalmaz:

```

1 package glvmdl.pac.highscore;
2
3 import javax.swing.JPanel;
4
5 @SuppressWarnings("serial")
6 public abstract class AbstractPanel extends JPanel {
7     abstract void setShowPanelProperties();
8 }

```

3.6. forráskód. AbstractPanel

## DisplayHighscore

Ez az osztály felelős a toplista megjelenítéséért. A toplista egy felugró ablakban jelenik meg, és egyetlen táblázatot tartalmaz. Az osztály konstruktora ezt az ablakot építi fel, illetve a táblázatot tölti fel adatokkal. Ehhez a java awt csomagjában található JTable-t használja. Az osztály felüldefiniálja a setShowPanelProperties() metódust, melyet az **AbstractPanel** osztályból örökölt. Ez a metódus állítja be az ablak méreteit.

A populateTable metódus a kapott highscores listának minden eleméhez létrehoz egy új Objektumot, és egy data[]-be helyezi azt. Ezt a tömböt továbbadja a tableModelnek egy új sorként. Az eraseTable() metódus a table sorainak számát 0-ra állítja, így úgymond törli a táblázatot. A táblázatot a setTableProperties(final DefaultTableModel tableModel) metódus konfigurálja. Itt állítjuk be a táblázat tényleges felépítését, színeit, láthatóságát, méretét.

```
1 public static void populateTable(List<Highscore> highscores){
2     if(tableModel.getRowCount() != 0){
3         eraseTable();
4     }
5
6     for(Highscore highscore : highscores){
7         Object name = highscore.getName();
8         Object score = highscore.getScore();
9         Object worldNum = highscore.getWorldNum();
10        Object date = highscore.getDate();
11
12        Object [] data = {name, score, worldNum, date};
13        tableModel.addRow(data);
14    }
15 }
```

3.7. forráskód. Táblázat feltöltése eredményekkel



## EndamePanel

Ez az osztály valósítja meg a játék végén felugró ablakot. Ez az ablak csak akkor ugrik fel, ha a játékos a legjobb 10 játékos között szerepel. Az ablak felépítése viszonylag egyszerű, így ez az ablak konstruktorában helyezkedik el. Beállítjuk az ablak méreteit, címét, elhelyezkedését, majd létrehozunk egy JTextField-et, ahol a játékos megadhatja a nevét. Ehhez tartozik egy metódus, mely kattintáskor a textField text attribútumát üresre állítja, így ha a játékos belekattint a TextField-be, akkor írhatja a nevét, nem kell a benne szereplő szöveget kitörölnie manuálisan. Hozzáadunk az ablakhoz egy gombot, mely felveszi a pontszámot, illetve elveti az ablakot. A konstruktor további részében az ablak láthatóságát és méretét állítjuk be.

## Highscore

Ez az osztály valósítja meg a toplista egy elemét, és tárolja azt egy statikus highscores tömbben. Az osztály konstruktora egy új eredményt hoz létre. Az osztály első metódusa a public static void getCurrentHighscores(), mely először ellenőrzi, hogy a felhasználó .pacmangame mappája létezik-e, és azon belül megtalálható a highscores.txt. Ha nem, akkor ezeket létrehozza üresen. Ha igen, akkor betölti ebből az adatokat, és feltölti a highscores tömböt a betöltött adatokkal. Ez után rendezzi a highscores listát, hogy az eredmények csökkenő sorrendben helyezkedjenek el.

```
1 public static void getCurrentHighscores(){
2     if(highscores.isEmpty()){
3         try{
4             Path path = Paths.get(System.getProperty("user.home"
5                 ));
6             if(Files.exists(Paths.get(path + "/.pacmangame/"
7                 highscores.txt"))){
8                 BufferedReader br = new BufferedReader(new
9                     InputStreamReader(new FileInputStream(System
10                         .getProperty("user.home") + "/.pacmangame/"
11                         highscores.txt)));
12
13                 String line = br.readLine();
14                 while(line != null){
```

```
10         addNewHighScore(line);
11         line = br.readLine();
12     }
13     br.close();
14
15 }
16 else{
17     Files.createDirectory(Paths.get(path + "/" +
18         "pacmangame"));
19     Files.createFile(Paths.get(path + "/" + "pacmangame"
20         "/last.txt"));
21     Files.createFile(Paths.get(path + "/" + "pacmangame"
22         "/highscores.txt"));
23 }
24
25 }catch(IOException e){
26     System.out.println("Highscore file not found!");
27 }
28
29 }
30
31 sortHighScores();
32 }
```

### 3.8. forráskód. Eredmények betöltése

A következő metódus a `sortHighscores()` metódus, mely a `highscores` lista rendezését végzi. Ezt buborékrendezés módszerrel hajtja végre. Ez nem a legeffektívebb rendezés, de maximum 10 elem rendezéséről beszélünk, így a hatékonyság nem a legfontosabb.

Az utolsó fontosabb metódus a `public static void saveHighscores()` metódus. Ez a benne szereplő útvonal alapján felülírja a `highscores.txt` metódust. Nem kell vizsgálnunk, hogy létezik-e, hiszen ha nem létezik, akkor üresen hozzuk létre. Mentés csak az `ExitGame` gombra való kattintáskor megy végbe, de az adatok a menü indulásával már betöltődnek, így feltehetjük, hogy a `highscores.txt` létezik. A `txt`-t minden esetben felülírjuk, így elég csak a már rendezett `highscores` lista elemeit soronként eltárolni a fájlban. Az osztály további metódusai a `String getName()`, `int getScore()`, `LocalDate getDate()`, `int getWorldNum()`, `String toString()`, melyek mind publikusak.

## 3.4. Tesztelés

A játékot folyamatosan tesztetem a fejlesztés alatt. A felfedezett hibák javításra kerültek. Mivel a játék minden felhasznált eleme grafikus, így ezeket a legcélraveze-tőbbben játékkal lehet tesztelni. A tesztelést a játék két fő alkotóelemére bontottam. Az első elem a menü, a második maga a játék.

### 3.4.1. Menü tesztelése

Menü tesztelése során először a menü attribútumait teszteljük. A két legfonto-sabb az életek száma, és a világ száma. Ezeket egyszerűen lehet tesztelni:

```
1 @Test
2 public void newMenuTest(){
3     Menu testMenu = new Menu();
4     Assertions.assertTrue(testMenu.getLives() == 3);
5     Assertions.assertTrue(testMenu.getWorldId() == 1);
6 }
```

3.9. forráskód. Menü tesztelése

A menü további tesztelése vizuálisan történik. Ellenőrizzük, hogy minden funkció megfelelően működik-e.

- New Game gombra kattintva új játék indul el. Az első pályán van a játékos, 3 élettel, 0 ponttal. Ekkor a játék korábbi mentése törlődik. 'Esc' gomb lenyomásával kiléphetünk a menübe, és Continue gomb lenyomásával folytathatjuk az előbb indított játékot. Ezzel ellenőriztük, hogy a mentés törlődött.
- Continue gombra kattintva a mentett játék folytatódik. Ezt bármilyen pályával ellenőrizhetjük. Fontos, hogy a pontszám és az életek száma is mentésre kerül.
- Highscore gombra kattintva megjelenik a toplista. A toplistán szereplő lista fentről lefele pontszám szerint csökkenő sorrendben jelenik meg. A toplistán egyszerre maximum 10 elem jelenik meg.
- Exit game gombra kattintva a játék menti a jelenlegi játékállást, és kilép a játékból. A mentést a C: meghajtó felhasználók/felhasználó-

név/.pacmangame/last.txt fájlban ellenőrizhetjük. Tárolt adatok: pálya id;életek száma;pontszám.

- X gomb a menü ablakának fejlécében: A játékot bármilyen mentés nélkül leállítja.

### 3.4.2. Játék tesztelése

Sikeres indítás után megjelent előttünk a játékelület. Ellenőrizzük a következőket:

- Megjelent egy új ablakban a játék
- Az ablak megfelelő méretű
- Az ablak neve helyesen van beállítva
- X gombra kattintva a játék leáll, mentés nélkül
- az ablak tetején egy fejlécében megjelent a világ száma, az életek száma, a pontszám, helyesen
- Az ablakban megjelent a játéktér, a world<n>.txt-nek megfelelően
- A játéktéren látjuk a játékost, Bubut, és a 4 vadórt
- A játéktéren látjuk az almákat
- A játéktéren látjuk a kosarakat, ha vannak.

Ezek alapján tudjuk, hogy a játék sikeresen betöltődött. A karakterek ez után elkezdenek mozogni a célpontjaik felé. A következő tesztelendő egység, az ütközés. Az ütközésekkel **Entity** csomagon belül az **Entity** osztály foglalkozik. Az ütközések, melyeket tesztelnünk kell, és hatásaik:

Ütközés	Játékos	Maci Laci
<i>Szikla és Fa</i>	Nem halad tovább	Nem halad tovább
<i>Alma</i>	Nő a pontszám	Nő a pontszám
<i>Kosár</i>	Nő a pontszám, vadőrök kezdőhelyükre kerülnek	Áthalad rajta, nem veszi fel
<i>Vadőr</i>	Veszít egy életet, vadőrök és Maci Laci kezdőhelyére kerül	Áthalad rajta, nem történik semmi

3.1. táblázat. Ütközések vizsgálata

A következő pont, amit ellenőriznünk kell, az a vadőrök, illetve Bubu útkeresésének helyessége. Ennek ellenőrzéséhez a kódba építettem egy részmetódust, mely az útvonalat rajzolja ki. Ez minden említett karakter osztályának render(Graphics g) metódusában megtalálható, és a következőképp néz ki:

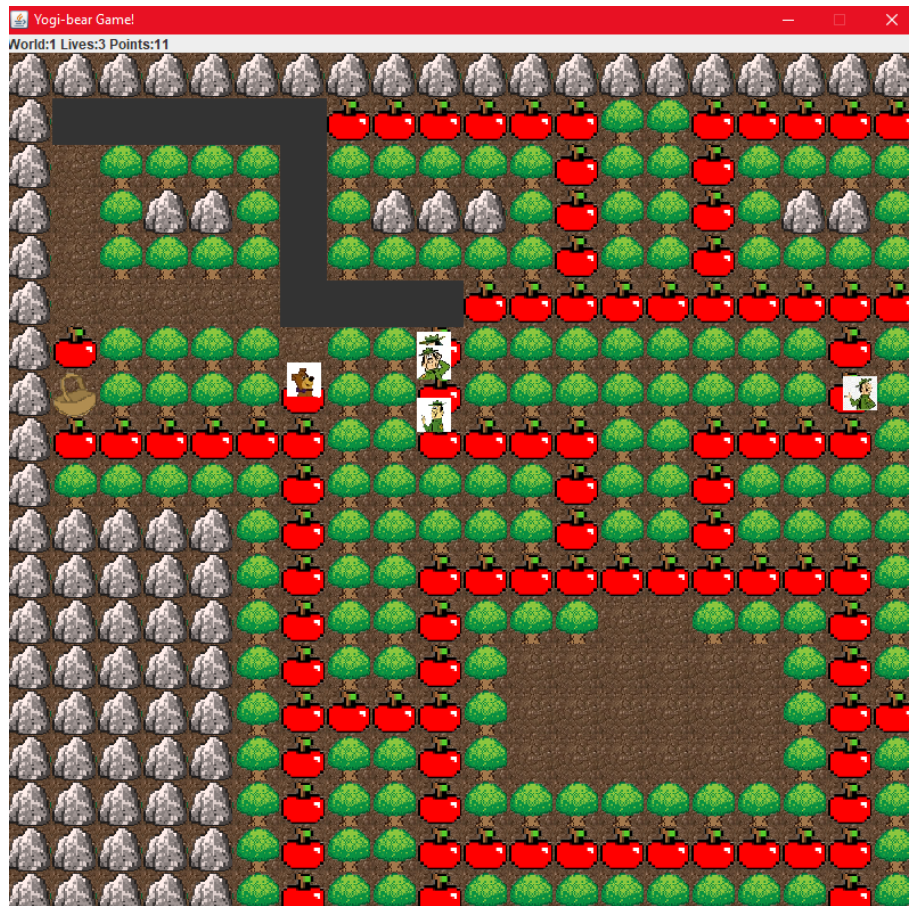
```

1 @Override
2     public void render(Graphics g){
3         g.drawImage(Assets.problem, (int)(x - handler.getGameCamera()
4             ().getxOffset()), (int)(y - handler.getGameCamera().
5             getyOffset()), width, height, null);
6         //         for(int i=0;i<path.size();i++){
7         //             g.drawRect((int)(path.get(i).getCol()*40- handler.
8                 getGameCamera().getxOffset()),(int)(path.get(i).getRow()*40-
9                 handler.getGameCamera().getyOffset()),40,40);
10            //             g.fillRect((int)(path.get(i).getCol()*40- handler.
11                getGameCamera().getxOffset()),(int)(path.get(i).getRow()*40-
12                handler.getGameCamera().getyOffset()),40,40);
13        //         }
14    }

```

3.10. forráskód. Renderelés

A render metódus kommentekkel ellátott sorai egy ciklus segítségével kirajzolják az adott karakter által tárolt útvonalat. Ez a következőképp néz ki:



3.6. ábra. Az útkeresés grafikus megjelenítése

Az utolsó ellenőrizendő esemény a játék vége. A játék kétféleképp érhet véget. Ha a játékos felveszi az összes almát és kosarat, akkor a játék véget ér, és a menübe kerül vissza. Ekkor a következő pálya indul el a Continue gombra való kattintáskor. A másik mód ahogy a játék véget ér, az a vereség. Ekkor a játékos elveszti az összes életét, és a játék leáll. Felugrik egy ablak, ahova a nevet lehet megadni. Ekkor a mezőbe való kattintáskor az ott lévő szöveg eltűnik. Megadhatunk egy nevet, amit majd a toplistában viszont látunk. Ha a játékos nincs benne az első tízben, akkor az ablak nem ugrik fel.

Név megadása után a Highscore megnyitásánál a névnek szerepelnie kell a listában, a szerzett pontokkal, és helyes dátummal. A toplistában maximum 10 elem jelenhet meg. A cellákba nem lehet bekattintani, azok értékei nem írhatók felül. A toplista megegyezik a highscores.txt-ben tárolt adatokkal.

Az utolsó tesztelendő funkció a kilépés. Kilépéskor az utolsó játék adatai mentésre kerülnek. Ellenőrizni a last.txt vizsgálatával, tudjuk. Fontos, hogy 0 étellel is

lehet menteni játékot. Ebben az esetben folytatás esetén új játékot kezdünk. Ez a Continue gomb függvényében van lekezelve.

### 3.4.3. Fejlesztés és tesztelés során felmerült korábbi problémák, hibák

Fejlesztés során az első komolyabb mérföldkövet a vadőrök finom hangolása okozta. Ez vonatkozik sebességre, kezdőhelyek beállítására, és az útkeresésre

A sebesség a játék nehézségét szabályozza. A játék jelen állás szerint egyjátékos módban sem könnyű játék. A vadőrök sebességének emelésével - például ha egyenlővé tesszük a játékos sebességével - a játék drasztikusan nehezebbé válik.

A kezdőhelyek beállításának nehézsége a pályák kialakításából ered. A játékos legjobb taktikája a pályán körbe-körbe haladás, és a túlélés. Így ha a játékos olyan helyzetben kezd, ahol a vadőrök alapból körbeveszik, az gyors vereséghez vezet.

Az útkeresés problémája az A\* algoritmus implementálásánál jelentkezett. A probléma abban merült ki ismét, hogy a vadőrök túl jól helyezkedtek a játékoshoz képest. Mivel a vadőrök hatalmas túlerőben vannak, így ha mindannyian tökéletesen helyezkednek, és például csak körbeveszik a játékost, majd fokozatosan lezárják a menekülési útvonalakat, a játékos vereségre van ítélve a játék kezdete előtt. Ezért implementáltam a vadőröket úgy, hogy inkább csak körbevegyék a játékost, és ne mind próbálják elkapni. Így a játékos, ha tudja melyik vadőrnek milyen a viselkedése, megpróbálhat elszökni.

A fejlesztés során nem ütköztem komolyabb problémába addig, amíg a játékot fejlesztői környezetből indítottam. Mivel a játékos nem fejlesztői környezetből fogja indítani a játékot, ezért ezt a fájlok egy .jar fájlba való exportálásával oldottam meg. Ehhez hoztam létre a start.bat fájl, melyhez készítettem egy parancsikont is, hogy akár asztalról is lehessen a játékot indítani. Két probléma merült fel az első indításkor. Az első, hogy a .jar fájlok nem találnak relatív elérési út alapján fájlokat, így a képeket és világokat a .jar fájlban belül kell tárolni resourceként. Ezeket a fájlokat továbbá csak streamként vagyunk képesek beolvasni. Ez alapvetően nem nagy probléma, de a fejlesztés korábbi szakaszaiban erre nem készültem fel.

A második probléma a .jar fájlokkal kapcsolatban, hogy erősen ellenzett az önmagukon való változtatás használata. Ez a mentés/betöltés, illetve a toplista rend-

szer eddigi megoldását döntötte romokba. Erre találtam megoldásként azt, hogy a felhasználó C meghajtóján, a saját mappájába hozok létre egy mappát, melybe mentem ezeket az adatokat. Ehhez persze a felhasználónak jogosultságot kell adni a játék számára, illetve ezek a fájlok tárhelyet fognak foglalni a felhasználó C meghajtóján. Cserébe ezek a fájlok kis méretűek, és mindkettőnek van fix maximális mérete. A last.txt fájl egyetlen sort tartalmaz, a highscores.txt fájl pedig maximum 10 sort tartalmaz.



## 4. fejezet

# Összegzés

### 4.1. Áttekintés

Szakedolgozatomban létrehoztam egy játékot, mely időseknek és fiataloknak egyaránt szórakoztató időtöltést nyújthat. Játékom, mely Maci Laci kalandjait mutatja be az erdőben a vadőrök elől menekülve, nosztalgikus élményt nyújt az idősebb korosztálynak, míg a izgalmakban teli időtöltést nyújt a fiataloknak. Az játék alapját képző Pac-Mant kibővítettem új pályákkal, mentés és betöltés funkcióval, valamint egy toplistával.

### 4.2. Továbbfejlesztési lehetőségek

A dolgozat mérföldkövei közül nem sikerült elérnem az utolsó mérföldkövet, mely a pályakészítő funkció volt. Ezt a funkciót elsőként lehetne implementálni a játék továbbfejlesztésében. Ehhez egy megadott 50x50 méretű üres pályát generálnék hegyekkel körbevéve. A játékelemek kiválasztását gombokhoz kötném, és kattintással lehetne azokat a pályára helyezni. Mentés során ellenőrizném, hogy ez hanyadik pálya, és hogy van-e már ilyen nevű pálya. Ha nincs, és minden szükséges játékelem elhelyezésre került, akkor a pálya mentésre kerül.

További funkció lehetne a pályaválasztó funkció. Ezt a menüben egy újabb gomb hozzáadásával kezdeném. Ekkor egy újabb menü ugrana fel, mely a már mentett pályák neveit tartalmazza gombokon. Adott gombra kattintva a gombhoz tartozó

pálya töltődne be 3 élettal, 0 ponttal. A pályaválasztó funkció nem akadályozná meg korábbi mentett játékok betöltését sem.

Továbbfejlesztés nem csak pályákról szólna, hiszen könnyen a játékhoz adhatóak új elemek is. Hozzáadható lehet adott pontszám fölött eltűnő elem, mely a pálya további részét nyitja meg. Ez azonban hatalmas pályához vezetne, mely nagy kihívást tenne a játékos elé.

Végül, de nem utolsó sorban a továbbfejlesztés közé tartozik a program optimalizálása is. Mint kezdő programozó, nagyon sok tapasztalatot nyújtott a játék fejlesztése. Sok új eszközt használtam, és beépített csomagot ismertem meg. Ehhez hozzátartozik, hogy nem a legoptimálisabban készítettem el a játékot, így ezen lehet javítani.

# Irodalomjegyzék

- [1] *Nagy Sára*. Eötvös Loránd Tudományegyetem, Algoritmusok és Alkalmazásaik tanszék. URL: <https://people.inf.elte.hu/saci/NagySara.html>. (látogatva: 2020.12.09).
- [2] *Hanna-Barbera Productions*. 1957. URL: <https://hu.wikipedia.org/wiki/Hanna-Barbera>. (látogatva: 2020.11.25).
- [3] *Warner Bros. Entertainment*. 1923. URL: [https://hu.wikipedia.org/wiki/Warner\\_Bros..](https://hu.wikipedia.org/wiki/Warner_Bros..) (látogatva: 2020.11.25).
- [4] *Oracle Corporation*. 1977. URL: <http://www.oracle.com>. (látogatva: 2020.11.14).
- [5] *Apache Netbeans*. 2017. URL: <https://netbeans.apache.org/>. (látogatva: 2020.12.11).
- [6] *Umllet*. 2002. URL: <https://www.umlet.com/>. (látogatva: 2020.12.11).
- [7] *Gregorics Tibor*. Eötvös Loránd Tudományegyetem, Programozáselmélet és Szoftvertechnológiai tanszék. URL: <https://people.inf.elte.hu/gt/>. (látogatva: 2020.12.12).

# Ábrák jegyzéke

2.1. A játék főmenüje . . . . .	6
2.2. Maci Laci, akit a játékos irányít . . . . .	7
2.3. Bubu Maci, akit a számítógép irányít . . . . .	7
2.4. A vadőrök, akik Maci Lacit üldözik . . . . .	8
2.5. Új játék, első pálya . . . . .	9
2.6. A toplista . . . . .	10
3.1. A játék főmenüjének működése . . . . .	16
3.2. Az Entity és EntityManager osztályok kapcsolata . . . . .	17
3.3. A Creature osztály, és az azt megvalósító osztályok kapcsolata . . . . .	20
3.4. A kirajzolással foglalkozó osztályok kapcsolatai . . . . .	29
3.5. A Toplistát kezelő osztályok kapcsolata . . . . .	30
3.6. Az útkeresés grafikus megjelenítése . . . . .	37

# Táblázatok jegyzéke

3.1. Ütközések vizsgálata . . . . .	36
-------------------------------------	----

# Forráskódjegyzék

3.1. Mentések betöltése . . . . .	14
3.2. Játékciklus . . . . .	14
3.3. A játékos felvesz egy almát . . . . .	18
3.4. A játékos bemenet figyelése . . . . .	21
3.5. Útvonalkeresés . . . . .	23
3.6. AbstractPanel . . . . .	30
3.7. Táblázat feltöltése eredményekkel . . . . .	31
3.8. Eredmények betöltése . . . . .	32
3.9. Menü tesztelése . . . . .	34
3.10. Renderelés . . . . .	36