**Szegedi Tudományegyetem**

**Informatikai Intézet**

**Szakdolgozat**

**Pokrovszkaja Katerina**

**2021**

Szegedi Tudományegyetem

**Informatikai Intézet**

**My First Adventure –**

**Egy játék fejlesztése JavaFX-ben**

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | Pokrovszkaja Katerina |  | Dr. Siket István |  |
|  | programtervező informatikus hallgató |  | egyetemi adjunktus |  |

Szeged

2021

**Tartalomjegyzék**

Feladatkiírás 4

Tartalmi összefoglaló 5

Bevezetés 5

**1. Alkalmazott technológiák 7**

1.1 Java 7

1.2 JavaFX 7

1.3 SceneBuilder 7

1.4 Tiled 7

1.5 SQLite, JDBC 8

1.6 Maven 8

1.7 Git 8

1.8 IntelliJ 8

**2. A játék megvalósítása 5**

2.1 Játékstruktúra 6

2.2 Entitások 6

2.2.1 Sprite 6

2.2.2 Játékos 6

2.2.3 Ellenség 6

2.2.4 NPC 6

2.2.5 Hátizsák 6

2.2.6 Tárgy 6

2.3 Fejlesztés alatti kihívások 6

2.3.1 A játék, mint game loop 7

2.3.2 Lenyomott gombok kezelése 7

2.3.3 A pálya kirajzolása 7

2.3.4 Több controller használata 7

2.3.5 A játékon belüli entitások irányítása 7

**3. Adatok tárolása az adatbázisban 5**

3.1 Technológia kiválasztása 5

3.1.1 Player 5

3.1.2 Enemy 5

3.1.3 Map 5

3.1.4 Backpack 5

3.1.5 Item 5

3.1.6 Gameplay 5

**4. A játék bemutatása 6**

4.1 Új játék 6

4.2 Játékos irányítása 6

4.3 Mentés betöltése 6

**5.Tesztelés 6**

5.1 Manuális tesztelés 6

5.2 Statikus kódelemzés PMD-vel 6

Irodalomjegyzék 7

Nyilatkozat 8

Köszönetnyilvánítás 8

**Feladatkiírás**

A feladat egy RPG stílusú játék készítése Java és JavaFX alkalmazásával. A cél egy olyan alkalmazás megvalósítása, ahol a felhasználó egy játékost tud irányítani, amelyikkel képes szörnyeket ölni, kereskedni az NPC-vel, a pályán mozogni stb. A felhasználó számára biztosítani kell az új játék indításának a lehetőségét, a játék mentését és annak betöltését.

**Tartalmi összefoglaló**

* A téma megnevezése: RPG játék fejlesztése JavaFX technológia alkalmazásával.
* A megadott feladat megfogalmazása: RPG játék fejlesztése JavaFX technológia alkalmazásával, ahol a felhasználó egy karaktert irányít. A karakterrel tud mozogni, harcolni a szörnyek ellen, szintet lépni és kommunikálni az NPC-vel.
* A megoldási mód: A feladat megoldásához egy teljesen új JavaFX projektet kellett létrehoznom, ahol a logikáért a Java volt a felelős. Minden elkészített funkciót manuálisan le kellett tesztelnem.
* Alkalmazott eszközök, módszerek: A fejlesztés során IntelliJ fejlesztői eszközt használtam, a grafikai elemek szerkesztésére SceneBuilder nevű programot, a pályák megtervezéséhez pedig Tiled nevű alkalmazást. Szükségem volt egy lokális adatbázisra, amihez nem kell külön szervert indítani, így SQLite-ot választottam. A különböző függőségek buildelésére Mavent használtam, a minőségbiztosításhoz pedig PMD statikus kódelemzőt.
* Elért eredmények:

Játék elérése:<https://github.com/Eyill/Szakdolgozat/tree/adventure>

* Kulcsszavak: Java, JavaFX, Scene, Sprite, entitás, Maven, PMD

**Bevezetés**

A dolgozat egy játékot mutat be, amely Java és JavaFX segítségével lett elkészítve. Azért is választottam ezt a feladatot, mert mindig is kíváncsi voltam arra, hogy milyen a játékfejlesztés. Első körben elhatároztam azt, hogy szeretnék egy kétdimenziós játékot készíteni tapasztalatszerzés céljából. Ezután ki kellett választanom, hogy milyen műfajú legyen a játék. Ebből a szempontból sokféle választék van, például stratégiai, akció, horror stb., de végül az RPG mellett döntöttem. Az RPG (Role-Playing Game) egyik legelterjedtebb műfaj, amelyben számtalan híres játék készült, mint például a Diablo sorozat, The Witcher, The Legend of Zelda, World of Warcraft stb. Az RPG játékok elsősorban azért is népszerűek, mert a játékosok kapnak saját karaktert, amellyel teljesíthetik a küldetéseket, megölhetik a szörnyeket és folyamatosan tudnak fejlődni. Másrészt pedig ezek a játékok olyan történetet mesélnek el, amivel magukkal ragadják a játékost.

A stílus meghatározása után szükségem volt textúracsomagokra, amivel megjeleníthettem a játékost, a pályát és egyéb szükséges entitásokat. Kisebb böngészés után sikerült találnom olyan forrásokat [14], ahonnan ingyenesen le lehet tölteni ezeket a csomagokat. Miután kiválogattam a számomra szimpatikus textúrákat, a Tiled nevű program segítségével megrajzoltam a pályát.

**1.fejezet**

**Alkalmazott technológiák**

**1.1Java**

A Java egy közismert programozási nyelv. A nyelv népszerűsége miatt rengeteg online tananyag, lecke, fórum található. Előnyei közé tartozik a platform függetlenség, és a hordozhatóság, így a programunkat bármilyen gépen futtathatjuk, ugyanis a kód nem függ a gép architektúrájától. Ezen a kívül a nyelv objektumorientált, ami azért is jó, mert egyrészt a projektet fel tudjuk bontani csomagokra, osztályok összeségére, másrészt a kódot újra lehet felhasználni. Véleményem szerint a Java programozási nyelv felhasználói barát, emiatt a nyelv elsajátítása nem nehéz.

**1.2 JavaFX**

A JavaFX a Java egyik keretrendszere, amely felelős a grafikus felületért. 2008-ban jelent meg, és gyakorlatilag a Java Swing utódjának tekintik. Jelenleg az Oracle JavaFX-es támogatása elérhető a Java JDK 8-ban egészen 2025-ig, azonban ha újabb verziójú Java projekthez szeretnénk társítani a JavaFX-et, akkor külön le kell tölteni a könyvtárat, és „buildelést” kezelő rendszerhez hozzáadni. Sikeres integrálás után kétféleképpen tudjuk kezelni a grafikai elemeket: vagy létrehozunk egy fxml fájlt, amelyben leírjuk az oldalunk felépítését (például milyen elemek legyenek benne, hol helyezkedjenek el stb.), vagy egy adott Java osztályban létrehozzuk ezeket az objektumokat. Továbbá, ha szeretnénk formázni a megjelenítést egy elkülönített fájlban, akkor erre is van lehetőség a CSS stíluslapokkal.

**1.3 SceneBuilder**

Ez egy olyan asztali alkalmazás, amely lehetővé teszi a grafikai objektumok manipulálását. Segítségével meg tudjuk adni, hogy hogyan nézzen ki az adott felület. A működése úgynevezett „drag and drop” elven alapszik, ahol a felhasználónak ki kell választani a hozzáadni kívánt elemet, majd az ablak középső részébe behúzni azt az elemet. Az adott elemnek meg lehet változtatni a kinézetét, ugyanis a SceneBuilder erre is ad lehetőséget. A felület mentése során az alkalmazás generál egy fxml fájlt, amikben rögzítve vannak a változtatások, így nem kell kézzel létrehozni és módosítani az objektumokat. Továbbá, ha az objektumoknak adunk azonosítót az applikáción belül, akkor az kigenerálja nekünk a controller osztály „csontvázát”. Onnantól már csak társítani kell az fxml fájlhoz az osztályt és az is az alkalmazáson keresztül megtehető.

**1.4 Tiled**

A Tiled segítségével létre lehet hozni különböző kétdimenziós pályákat a játékhoz. Ehhez meg kell adnunk, hogy mekkora legyen egy képkoca, illetve a képkockák számát. Ezután be lehet tölteni azt a képet (tileset), ahová ki vannak gyűjtve a különböző képkockák, textúrák (például fű, kő, fal stb.). Ha a méreteket megfelelően adtuk meg, akkor a program automatikusan szétvágja a képkockákat, így ha valamelyikre rákattintunk, majd rákattintunk a pálya valamelyik mezőjére, akkor az adott képkockát kirajzolja az adott helyen. Alapvetően egy réteg jön létre, de lehet több réteget is létrehozni, így a képkockákat egymásra tudjuk illeszteni. Abban az esetben, ha az illesztés nem megfelelő (például az egyik illesztett képkockánál a bizonyos részek nem átlátszóak), akkor ki tudjuk jelölni azokat a részeket, amelyeket meg szeretnénk jeleníteni. Mentés során txm fájl jön létre, de szükség esetén átírható a kiterjesztés xml-re.

**1.5 SQLite, JDBC**

A JDBC egy olyan Java interfész, amely lehetőséget ad arra, hogy Java alkalmazáson keresztül tudjunk használni adatbáziskezelőket. Használatához be kell importálni modulként a projektbe. Segítségével tudunk SQL lekérdezéseket írni, és azokat továbbítani az adatbáziskezelő felé. Abban az esetben, ha létezik megfelelő driver az adott adatbáziskezelőhöz, akkor tudunk vele kommunikálni az osztályon keresztül.

Az SQLite C nyelven íródott nyílt forráskódú programkönyvtár, mely lényegében egy adatbáziskezelő és adatbázismotor. Előnye, hogy nem kell külön szervert indítanunk ahhoz, hogy működésbe lépjen, azaz tudjuk lokálisan futtatni. Azért is esett a választásom az SQLitera, mert a játék egyszemélyes, így elegendő a lokális adattárolás. Az SQLite egy keresztplatformos fájlban tárolja az adatokat, így biztosított a hordozhatóság a különböző eszközök között. Nagyon sok programozási nyelvvel kompatibilis, köztük Javaval is.

**1.6 Maven**

A Maven egyik legismertebb, elsődlegesen Java projektekhez megírt build keretrendszer. Lehetőséget nyújt az automatizált buildelésnek, így a fejlesztőnek nem kell ezt manuálisan elvégeznie, illetve kisebb a hibázási lehetőség. A pom.xml-ben tudjuk leírni a projektet és annak a függőségeit. Szükség esetén a Maven le tudja tölteni a hiányzó komponenseket, majd ezeket beépíteni a projektbe. A Maven segítségével tudunk JAR fájlba csomagolni a projektünket (Java ARchive), amely biztosítja a program hordozhatóságát, illetve a szükséges adatok eltárolását (például .class fájlok, mappák stb.).

**1.7 Git**

A Git egy verziókövető rendszer, mellyel a felhőben tudunk projekteket tárolni, illetve a bennük lévő változtatásokat követni. A fejlesztők számára elengedhetetlen eszköz, ugyanis a lefejlesztett funkciók és minden szükséges változtatást elküldik az úgynevezett repository-ba („raktár”), ahol a forráskódok össze lesznek fésülve. Fejlesztés során javasolt a Git használata, mert egyrészt könnyen fel lehet tölteni a projekt legfrissebb verzióját, másrészt ha az eszköz (például számítógép) megsérül, vagy tönkremegy, akkor nem fog elveszni a teljes munka, hiszen le lehet tölteni a projektet. A csoportos fejlesztésnél a fejlesztők külön branchet (másnéven fejlesztés ág) hoznak létre a funkciók megvalósításához, majd összefésülik a főággal. Szakdolgozatom készítése során egy fejlesztési ágat használtam, mert egyedül fejlesztettem a projektet, illetve a code review (átvizsgálás), mint általános folyamat elmaradt, így a sikeres teszt után rögtön be is kerültek a változtatások a fejlesztői ágba. A gitignore nevű fájl segítségével ki tudjuk zárni azokat a fájlokat és könyvtárakat, amelyeket nem szeretnénk feltölteni.

**1.8 IntelliJ**

Az IntelliJ a JetBrains által fejlesztett integrált fejlesztői eszköz. Egyik legnagyobb előnye az, hogy programozás során vizsgálja a kódot, és szükség esetén javaslatot tesz a kód átírására (például if-else ágat megírja egy soros változatba), szükséges részek pótlása (például fájlkezeléskor try-catch blokk használatát javasolja). Ezen kívül rendelkezik saját beépített eszközökkel (például beépített Mavennel, verziókövető rendszer stb), melyekkel szintén egyszerűbbé és gyorsabbá teszi a fejlesztést. Ezeknek az előre beépített pluginoknak köszönhetően a fejlesztői eszköz produktívabb munkát biztosít. Az IntelliJ projekt szinten fel tudja ismerni a használt eszközöket, így a fejlesztői eszköz el tudja végezni az automatikus szinkronizálást és a szükséges beállításokat.

**2. fejezet**

**A játék megvalósítása**

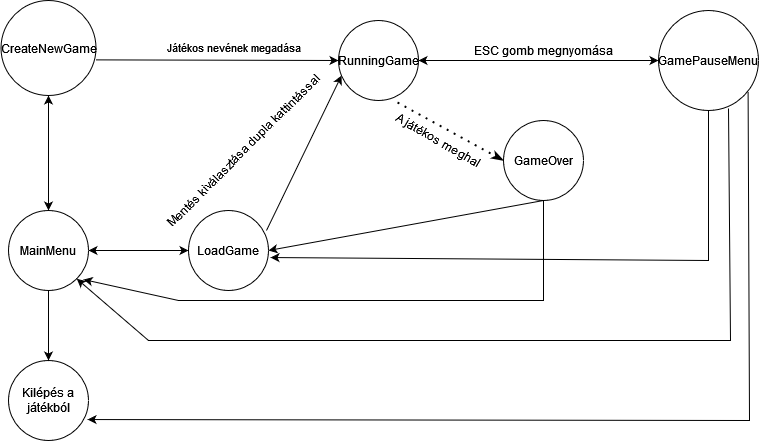
**2.1 Játékstruktúra**

Ebben az alfejezetben be szeretném mutatni a projekt jelenlegi felépítését. A játékok többsége rendelkezik menükkel (például főmenü, játékon belüli menü stb.), amelyek a játékost navigálják az applikáción belül. A játék elindításakor először a főmenüt látjuk, ahol ki lehet választani, hogy új játékot szeretnénk-e kezdeni, vagy egy mentett játékot akarunk betölteni.



2.1 ábra. Indításkor megjelenő főmenü

A 2.2 ábra alapján látható, hogy bizonyos menük és az értük felelős controllerek milyen kapcsolatban állnak egymással. Ahogy korábban említettem, a főmenüből elérhető az új játék és a mentés betöltése funkció. A játék tényleges betöltése a kiválasztott mód alapján történik.

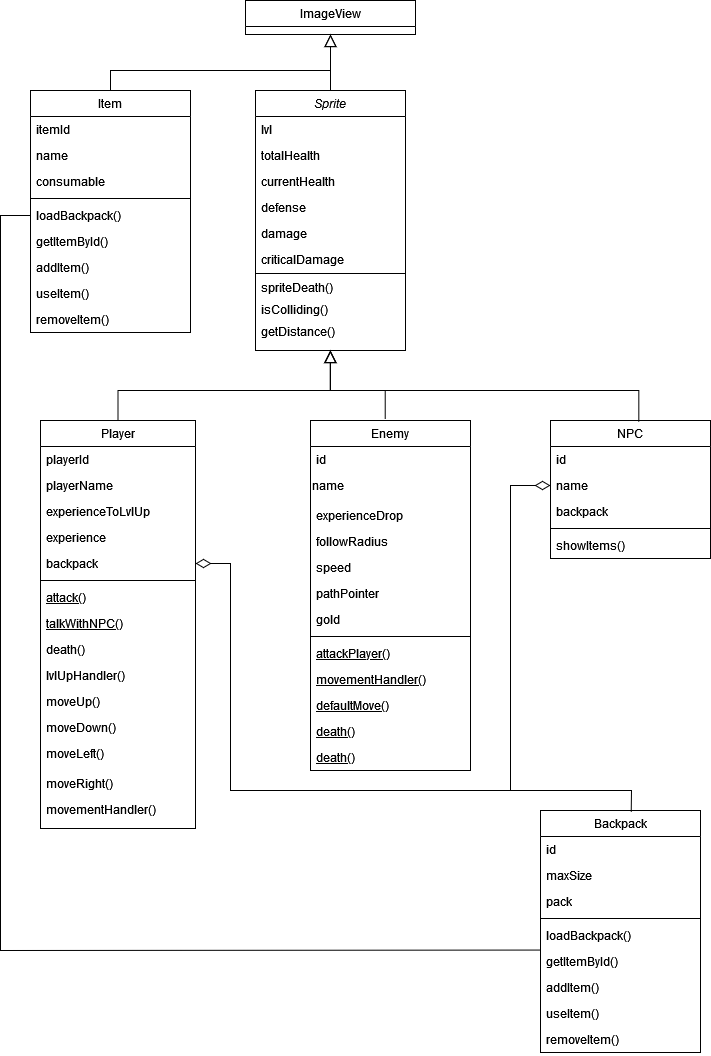
****

2.2 ábra. A játékon belüli controllerek

A játék elején fontos a controllerek cserélése menügombok megnyomásánál. Alkalmazás elindításakor automatikusan létrejön egy úgynevezett „szintér” objektum (Stage), és előtte pedig a „jelenet” objektum (Scene), amely reprezentálja az aktuális felületet. Amikor a beépített gombok segítségével felületet váltunk (például a főmenünél rákattintunk a „Create new game” gombra), akkor valójában csak egy új Scene objektumot hozunk létre, és azt átadjuk a Stage objektumnak a setScene függvényen keresztül. Erre azért van szükség, mert ha új Stage objektumot hoznánk létre, akkor ezzel együtt egy új ablak jönne létre a már meglévőn kívül, ami nyilván nem célszerű, mert magának a játéknak 1 ablakon kell futnia. Ezt a fajta megoldást több helyen is fel kellett használnom, így létrehoztam egy olyan absztrakt osztályt (CommonMenu), ahol elhelyeztem azt a függvényt, ami megvalósítja a Scene objektumok cserélését, majd ebből az osztályból leszármaztattam a controllereket. Azonban annál az osztálynál, ahol ténylegesen elindul a játék (RunningGame) ki kellett vennem a leszármaztatást, mert különben nem működött a billentyűkezelés.

**2.2 Entitások**

A játék egyik fontos építőköve az entitások. Ide tartozik összes olyan objektum, amitől az alkalmazás játszhatóvá válik, például maga a játékos, a szörnyek stb. Az entitásokat fontos külön kiemelni, mert lényegében a játék során ezek az objektumok fognak egymással kommunikálni. Az alábbi ábra mutatja az osztályok közötti kapcsolatot.

****

2.3 ábra. Osztálydiagram az entitásokhoz

**2.2.1 Sprite**

A játékfejlesztéssel kapcsolatos kutatásaim során többször találkoztam a Sprite-tal, mint általános grafikai elemmel, ezért követtem a mintát, és létrehoztam saját Sprite osztályt. Ez az osztály szülő osztálya azoknak az entitásoknak, amelyek a játék során „cselekvőképesek”. Ez az osztály csak a képi megjelenítésért lett volna felelős, de végül ki lett bővítve azokkal az adattagokkal, amelyekkel az entitások rendelkeznek (például életerő, jelenlegi szint stb.). Eredetileg az osztály rendelkezett volna egy ImageView objektummal, viszont szimpatikusabbnak tartottam leszármaztatni az ImageViewból a Sprite osztályomat. Ennek köszönhetően elegendő volt az ős konstruktornak átadni az entitás képét, és magára a Sprite objektumra meghívni a setLayoutX és setLayoutY függvényeket, illetve őt hozzáadni a Pane gyerekeihez.

**2.2.2 Játékos**

A Player, mint Sprite egyik leszármazott osztálya reprezentálja a játékosunkat. A játékos tud szintet lépni a tapasztalatpontok segítségével, amelyeket a szörnyektől tud szerezni. Ha elég közel áll a szörnyhöz, akkor meg tudja sebesíteni, de ha ügyetlen, akkor a szörny le fogja győzni, és ilyenkor a Game Over felület fog megjelenni, ahol vagy az utolsó mentést tudjuk betölteni, vagy visszalépni a főmenübe. A játékos gyorsabban tud mozogni az ellenséghez képest, így el is tud menekülni előle.



2.4 ábra. A játékos alapértelmezett képe

**2.2.3Ellenség**

A Sprite másik fontos gyerekosztálya, az Enemy. A Player objektummal szemben nem képes önálló fejlődésre, azaz a szintje fix érték. A szörny elpusztítása során valamennyi tapasztalatpontot ad a játékosnak. Ha a közelében megtalál egy játékost, akkor elkezd felé mozogni. Abban az esetben, ha a játékos hatósugarában helyezkedik el, akkor elkezdi őt támadni egészen addig, amíg a játékos nem hal meg, vagy a játékos nem pusztítja el a szörnyet, vagy a játékos nem menekül el előle.

****

2.5 ábra. Példa az ellenségre

**2.2.4 NPC**

Az NPC (Non-player character) egy olyan szereplő a játékban, amit a felhasználó nem tud irányítani. A játékos tud vele kommunikálni, de nem tudja megtámadni. A szörny nem lép interakcióba az NPC-vel. Az NPC rendelkezik saját hátizsákkal, így tud kereskedni a játékossal. Akárcsak a szörny, maga az entitás nem tud önállóan fejlődni, így az attribútumai (például jelenlegi szint, támadó erő stb.) változatlanul maradnak.



2.6 ábra. Példa az NPC-re

**2.2.5 Hátizsák**

A hátizsák egy olyan objektum, amely nem rendelkezik Sprite attribútumokkal, ugyanis logikailag nem tartoznak hozzá. A hátizsák Item objektumokat tárol, amik a játékbeli tárgyakat reprezentálják (például ital stb.), illetve eltárolja azoknak a mennyiségét. 24 féle tárgy lehet, de az adott tárgyból lehet több darab a hátizsákban. A hátizsák megjelenítéshez saját modalt használok, ami egy Gridpane, és minden cellájában van egy ImageView objektum, ami maga az Item objektum képi megjelenítése.

A képen ablak, állvány, körülvéve látható

Automatikusan generált leírás

2.7 ábra. Üres hátizsák ábrázolása a játékon belül

**2.2.6 Tárgy**

A tárgy egy egyszerű játékbeli entitás. Létrehoztam Item osztályt, amelyet leszármaztatom az ImageView osztályból, ugyanis rendelkezik képi attribútummal, illetve ugyanúgy el lehet helyezni a pályán, így a játékos fel tudja venni. Az Item osztály szintén nem lett leszármaztatva a Sprite osztályból, mert az attribútumai nem tartoznak hozzá. Ebben a játékban egyszerű *boolean* típus alapján el lehet dönteni, hogy az adott tárgy fogyasztható-e vagy sem. Kereskedés szempontjából szükség van egy *integer* adattagra, ami eltárolja az alapértelmezett árat. Legtöbb játékban előfordul, hogy a játékos kevesebb pénzért tudja eladni a tárgyat, viszont sokkal drágábban tudja megvenni a kereskedőtől. Mivel a játékos és az NPC-k hátizsákja el van különítve, így módosíthatóak a tárgyak



2.8 ábra. Életerő ital, mint játékbeli tárgy

**2.3 Fejlesztés alatti kihívások**

Ebben az alfejezetben ki szeretném emelni azokat a problémákat, és érdekes tényezőket, amelyek miatt nem ment gördülékenyen a fejlesztés. Mint minden keretrendszernél, természetesen JavaFX-nek is megvannak a szépség hibái, így kutatni kellett az alternatív megoldások után, illetve átgondolni a lehetőségeket.

**2.3.1 A játék, mint game loop**

Azon felül, hogy a játékban szerepelnek a szükséges entitások (játékos, szörnyek stb.) mindenképpen kell „valami”, amitől a játék futni fog folyamatosan egészen addig, amíg a játékos nem hal meg, nem fejezi be a teljes játékot, vagy esetleg kilép belőle. Ez a valami, nem más, mint a végtelen ciklust biztosító folyamat, azonban itt olyan folyamat kell, ami bizonyos idő után frissíti a játék állapotát. Legtöbb esetben törekednek a 60 fps (frame per second, azaz képkocka másodpercenként) beállításra, úgyhogy ezt vettem alapul.

A legegyszerűbb és egyben a legközismertebb megoldás a while ciklus használata (például addig fusson a játék, amíg a játékos élete nem nulla). Azonban amikor megpróbáltam alkalmazni ezt a megoldást, akkor súlyos hibákat kaptam. Itt legfőképpen az úgynevezett szálkezelésével volt probléma. Szerencsére a JavaFX-nek van saját „időzítő” osztálya (AnimationTimer), amely képes meghívni az objektum törzsében lévő metódusokat bizonyos időközönként (alapértelmezetten 60-szor hívja meg másodperceként).

1. new AnimationTimer() {
2. @Override
3. public void handle(long now) {
4. healthBar.setProgress((double) player.getCurrentHealth() / (double) player.getTotalHealth());
5. expBar.setProgress(player.lvlUpHandler());
7. player.setLayoutX(player.getLayoutX());
8. player.setLayoutY(player.getLayoutY());
10. hpLabel.setText(player.getTotalHealth() + " / " + player.getCurrentHealth());
11. expLabel.setText(player.getExperienceToLvLUp() + " / " + player.getExperience());
12. lvlLabel.setText(String.valueOf(player.getLvl()));
14. UserDataHandler.checkEnemyLife();
16. for (Enemy enemy : enemies) {
17. enemy.movementHandler(player);
18. }
19. }
20. }.start();

Abban az esetben, ha a játékos meghalna, vagy a felhasználó ki akar lépni a főmenübe, akkor az AnimationTimer osztálynak a stop függvényét meg kell hívni. A két esetnél megtörténik a felületváltás, de a háttérben ugyanúgy futhat az időközönként meghívott függvények, és emiatt súlyos hibák és kivétel léphetnek fel. A GameOver funkció tesztelése során tapasztaltam a hibát, illetve amikor a játékossal közelebb mentem az ellenséghez, majd kiléptem a főmenübe.

**2.3.2 Lenyomott gombok kezelése**

A játék fejlesztése során foglalkoznom kellett a felhasználói inputok kezelésével, ugyanis erre szükség van például akkor, amikor a karakterünkkel mozogni szeretnénk, vagy esetleg megtámadni egy szörnyet. A JavaFXnek van KeyEvent osztálya, amely reprezentálja azt az eseményt, amely billentyűgomb lenyomásakor generálódik. Az eseménynek le tudjuk kérni a kódját getCode függvény segítségével, így bizonyos gombokhoz tudunk algoritmusokat rendelni.

1. public void handleKeys(KeyEvent event) {
2. switch (event.getCode()) {
3. case W:
4. this.moveUp(2);
5. break;
6. case S:
7. this.moveDown(2);
8. break;
9. case A:
10. this.moveLeft(2);
11. break;
12. case D:
13. this.moveRight(2);
14. break;
15. default:
16. break;
17. }
18. }

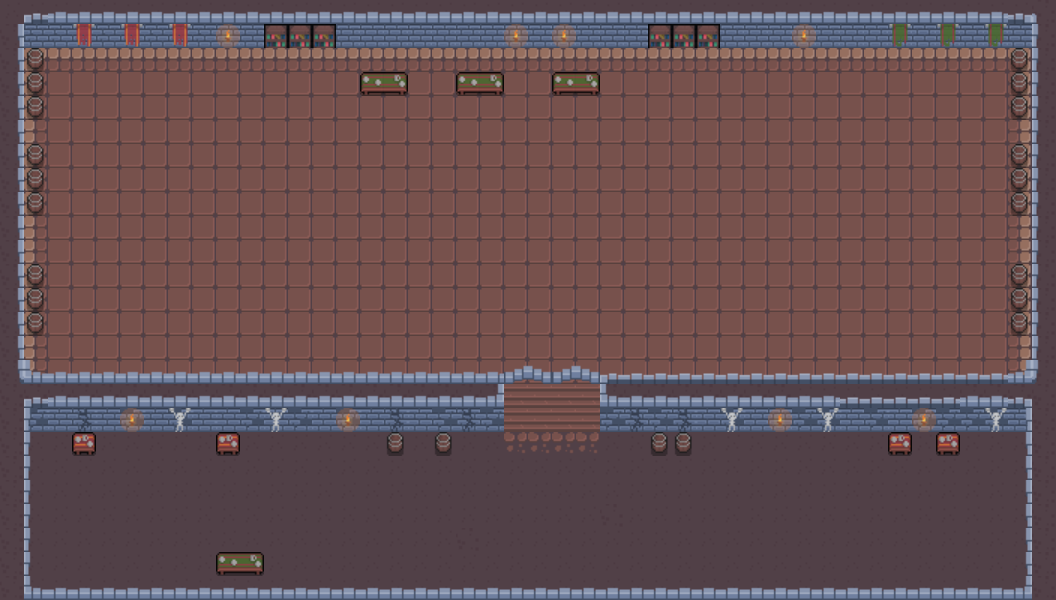
Amikor leteszteltem az implementált funkciót, akkor gomb lenyomáskor nem történt semmi. Találtam egy cikket, melyben a szerző hasonló problémába ütközött, viszont megfigyelt egy érdekes jelenséget. Ha a felületen csak alakzat jellegű objektumok szerepelnek (például téglalap, kör, szöveg, kép stb.), akkor a Scene objektum nem fogja kezelni az eseményeket, ezért legalább egy Button objektumot létre kell hozni. Arról is írt, hogy ha a Controller objektumra meghívjuk a focus függvényt, akkor az szintén megoldja a problémát, de nálam csak a gombos verzió lett hatásos. Ezen kívül az is akadályozta a működést, ha a RunningGame osztályt leszármaztattam a CommonMenu osztályból, úgyhogy ezt ki kellett vennem.

A Button objektumok alapértelmezetten érzékenyek a SPACE és az ENTER gomb lenyomására. Ha ezt a tulajdonságot kikapcsoljuk (focus traversable), akkor az eseménykezelő nem fog működni. A játék esetében ez abban nyilvánul meg, hogy a karakter jelenlegi állapotát jelző sávban lévő gomb aktiválódik. Meghagytam a SPACE billentyű használatát, ugyanis a legtöbb retro játékban a SPACE billentyűvel vagy ugorni lehet, vagy támadni.

**2.3.3 A pálya kirajzolása**

Elsőkörben azzal kísérleteztem, hogy magának a Pane objektumnak beállítottam a hátterét egy kép segítségével, ami reprezentál egy játékbeli pályát. Ez a megoldás nem volt célszerű, mert egyrészt a pályán belüli akadályokat kézzel kellett létrehozni. Másrészt így nem volt dinamikus a pálya kirajzolása, hiszen új pályáknál újra kellett definiálni az akadályok elhelyezkedését. A fejlesztés során már használtam a Tiled nevű programot, így elgondolkodtam azon, hogy fel kéne használni azt az xml fájlt, amit készítettem a pályához, illetve szükségem lesz egy pályát kirajzoló algoritmusra, ami a fájl alapján megjeleníti a textúrákat. Maga a rajzolható felület egy Canvas objektum, amit hozzáadok a Pane-hez. A képkockákhoz készítettem egy enumot (Block), amely eltárolja a képkocka azonosítóját(*id*), képkocka képének a helyét (*path*), és azt, hogy akadály-e (*collidable*). Ezután mindegyik képkockát eltároltam enumként, amelyek az adott pályán szerepelnek. Így már csak meg kellett írni egy olyan függvényt, amely végig iterál az xml fájlon belüli sorokon, és rárajzolja a canvasre a textúrákat az azonosítójuk alapján.

A pálya kirajzolása során helytelenül jelentek meg a kirajzolandó tile-ok, ugyanis a canvasan egymásra voltak csúsztatva. Ebből arra tudtam következtetni, hogy a drawImage függvény hívása során nem megfelelő pozíciók kerülnek átadásra. Maga a kiindulási képlet ez volt *: gc1.drawImage(image, x, y, 16, 16)*, amelyben image a kirajzolandó textúra, x és y az aktuális pozíciót jelöli, és az utolsó két érték pedig a szélességet és a magasságot. Mivel a tile-ok, amiket használtam 16x16 pixel méretűek, ezért megpróbáltam beszorozni az x és y változót 16-tal, és így helyesen jelentek meg a textúrák. A 2.2-es ábra szemlélteti a végeredményt.



2.9 ábra. A tile-ok segítségével kirajzolt pálya

**2.3.4 Több controller használata**

A játéknak a felhasználói felületének készítése során felmerült bennem, hogy szükség lehet a beépített felugró ablakokra, másnéven modalokra. A JavaFX-nek létezik saját osztálya erre a célra, viszont az külön ablakként tud megjelenni, ami számomra nem volt megfelelő, úgyhogy egy teljesen új modalt kellett készítenem. Először magába a RunningGame osztályba raktam az elkészített modalt. Ez a megoldás működött, de ha a többi modalt is ugyanitt elhelyeztem volna, akkor az nagyon sok sort eredményezett volna, ezért szerettem volna elkülöníteni a modalt a RunningGame osztálytól.

A modalokhoz ugyanúgy készítettem fxml leírást a SceneBuilder segítségével, mert egyrészt úgy egyszerűbb és gyorsabb volt megtervezni a modalt. Másrészt ha a hierarchiában lévő elemeket leszármaztatjuk egy főelemből (például egy Pane objektumba „csomagoljuk” az összes elemet), akkor a bizonyos módosításokat elegendő végrehajtani a főelemen (például ha elrejtjük a főelemet, akkor automatikusan a gyerekei is el lesznek rejtve). Az fxml fájlok mellett készítettem külön osztályokat (például GamePauseModal), amik a konstruktorjukban betöltik a megfelelő fxml fájlt. A probléma az volt, hogy az FXMLLoader loader függvénye nem tölti be teljesen az objektumhierarchiát, hanem egy Pane (vagy annak a leszármazott gyerekosztály) objektumot, illetve az fxml fájlhoz tartozó controllert, emiatt nem működtek a modalban lévő gombok, mert valójában null volt az értékük. A probléma megoldásához szimplán létre kellett hozni külön controller osztályt a modalhoz.

**2.3.5 A játékon belüli entitások irányítása**

A játékosnak csak akkor kell elmozdulnia, hogyha a felhasználó megnyomta a billentyűt (például W-t), előtte pedig meg kell vizsgálni, hogy a következő pozíció, ahová lépnénk akadály-e, vagy sem. Mivel a játékost nekünk kell irányítani, így rálátásunk van az egész pályára és tudunk megfelelő döntést hozni.

A szörnyek esetében két esetet kell vizsgálni: közelében van-e a játékos, vagy sem. Ha a szörny nem találta meg a játékost a közelben, akkor elkezd mozogni a kis környezetében úgy, hogy az akadályokra nem lép rá. Első lépésként kigenerál egy tetszőleges irányt, ahová lépne (fel, le, jobbra, balra), majd egy véletlenszerű számot generál 1 és 50 között, ami a lépéseknek a számát jelöli. Ezután meg kell vizsgálni, hogy a lehetséges új pozíció akadály-e. Ha nem az, akkor ki kell számolni az új koordináta és a jelenlegi koordináta közti különbségét, ebből vektort képezni, és fokozatosan elmozdítani a szörnyet az adott vektor irányában, miközben arra kell ügyelni, hogy a vektoron belüli pont akadály-e. A pályán belüli tile típusokat egy kétdimenziós 25x44 db elemű tömbben tárolom. A tile típus lekéréséhez, amelyiken éppen elhelyezkedik az entitás az aktuális pozíciót le kell osztanom 16-tal (mint korábban említettem a felhasznált képkockák 16x16 pixelesek).

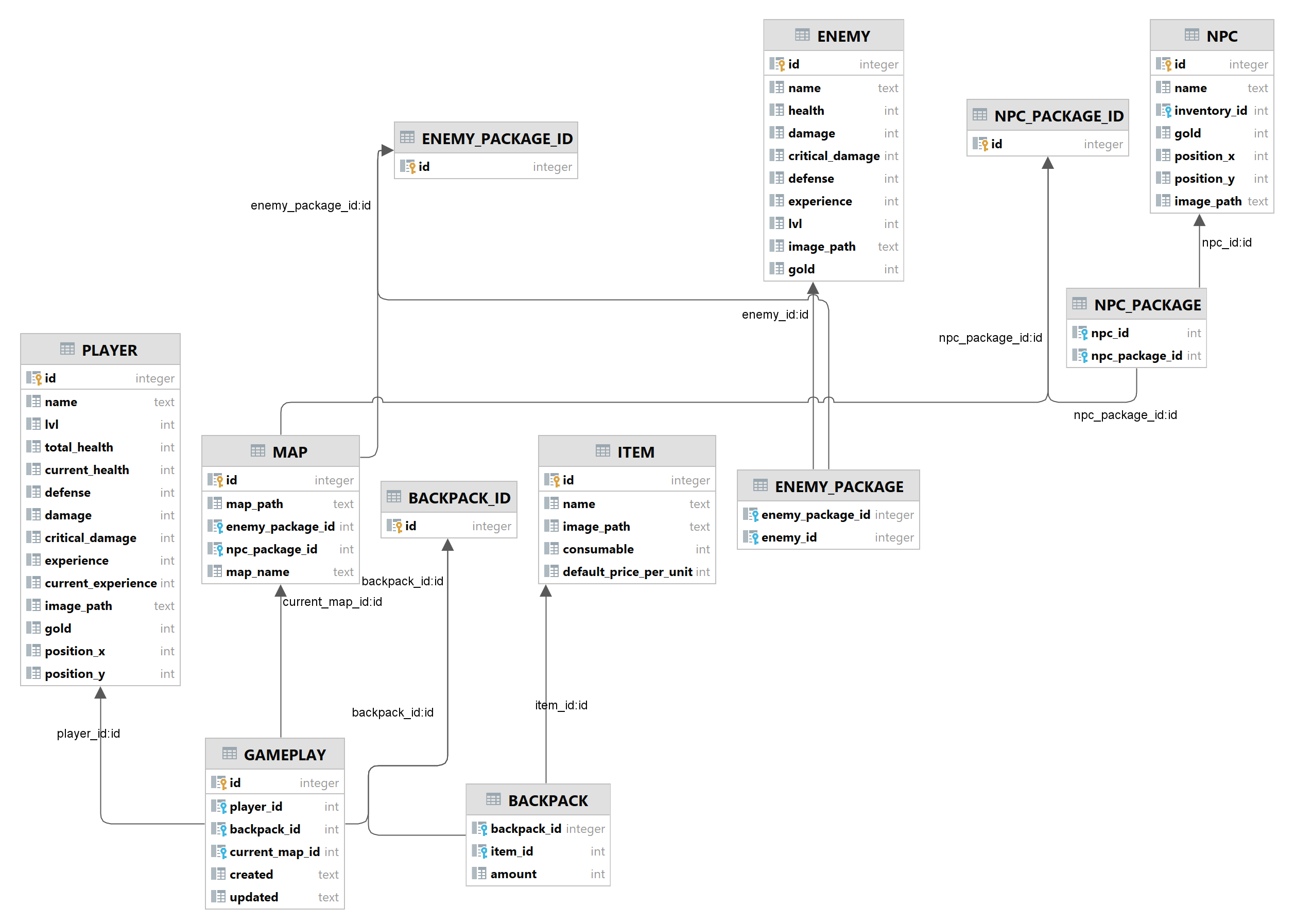
Ha a szörny mégiscsak talált egy játékost, akkor lényegében ugyanazokat a műveleteket végrehajtja, mint az „alapértelemezett” mozgásnál, viszont a játékos aktuális koordinátáit veszi célba, és egészen addig halad felé, amíg nem kerül annyira közel hozzá, hogy elkezdje a támadást. Ehhez A\* algoritmust használtam. A szörny megvizsgálja a jelenlegi pozíciójából egy lépésből elérhető mezőket. Az akadályos mezőket eldobja a lehetséges opciók közül, majd kiválasztja azt, amelyikkel közelebb kerülhet a játékoshoz.

1. public void defaultMove() {
2. int x\_position = (int) getLayoutX();
3. int y\_position = (int) getLayoutY();
4. List<String> givenList = List.of("left", "right", "up", "down");
6. Random rand = new Random();
7. int randomIndex = rand.nextInt(givenList.size());
8. String direction = givenList.get(randomIndex);
9. int randomSteps = rand.nextInt(100);
11. switch (direction) {
12. case "left":
13. x\_position -= randomSteps;
14. break;
15. case "right":
16. x\_position += randomSteps;
17. break;
18. case "up":
19. y\_position -= randomSteps;
20. break;
21. case "down":
22. y\_position += randomSteps;
23. break;
24. default:
25. break;
26. }
27. int new\_x = (int) Math.ceil(((double) x\_position) / 16);
28. int new\_y = (int) Math.ceil(((double) y\_position) / 16);
30. if (new\_y < 25 && new\_x < 43 && new\_y >= 0 && new\_x >= 0) {
31. if (!TileManager.gameMap[new\_y][new\_x].collidable) {
32. Vector target = new Vector(x\_position, y\_position);
34. double x = target.x - getLayoutX();
35. double y = target.y - getLayoutY();
36. x \*= speed;
37. y \*= speed;
39. int next\_position\_x = (int) Math.ceil((getLayoutX() + x) / 16);
40. int next\_position\_y = (int) Math.ceil((getLayoutY() + y) / 16);
42. if (!TileManager.gameMap[next\_position\_y][next\_position\_x].collidable) {
43. setLayoutX(getLayoutX() + x);
44. setLayoutY(getLayoutY() + y);
45. }}}}

**3. fejezet**

**Adatok tárolása az adatbázisban**

**3.1. Technológia kiválasztása**

A fejlesztés elején ragaszkodtam ahhoz, hogy a különböző adatok betöltését és mentését egy sima fájlban kell elvégezni (például egy txt-ben vagy egy xml-ben). Később bebizonyosodtam róla, hogy érdemes adatbázist használni, mert SQL lekérdezéseket viszonylag gyorsan lehet írni, egyszerűbb a fájl kezelésnél és valamelyest biztonságosabb. Miután megismertem a SQLite adatbáziskezelőnek a lehetőségeit elgondolkodtam azon, hogy ez lenne a megfelelő választást, mert mint korábban említettem, a lokális adattároláshoz nem kell külön szervert indítani. Mivel Windows 10-ben fejlesztettem, így az SQLite telepítése után külön fel kellett vennem a könyvtárat a rendszer környezeti változói közé. Ezután az IntelliJ már automatikusan felismeri a db kiterjesztésű fájlokat, és automatikusan létre tudja hozni a benne lévő táblákat.

3.1 ábra. Az entitások tárolása adatbázisban

**3.1.1 Player**

A *PLAYER* táblán belül nem szerepel olyan oszlop, amely külső kulcsra hivatkozna, illetve nincs összetett adattagja. A tábla sémája *PLAYER(id, name, lvl, total\_health, current\_health, defense, damage, critical\_damage, experience, current\_experience, image\_path, gold, position\_x, position\_y)*, ahol az *id* az elsődleges kulcs, és új rekord beszúráskor automatikusan generálódik a következő azonosító.

**3.1.2 Enemy**

Az *ENEMY* táblában szintén nem szerepel összetett adattag, a sémája *ENEMY(id, name, health, damage, critical\_damage, defense, experience, lvl, image\_path, gold).* Ha több fajta szörnyet szeretnék megjeleníteni egy pályán belül, ahhoz szükségem lesz egy segédtáblára, ami jelen esetben az *ENEMY\_PACKAGE* és *ENEMY\_PACKAGE\_ID* tábla. Az *ENEMY\_PACKAGE\_ID* táblában csak egy azonosító kerül tárolásra, amely az ellenfelek „csomagját” azonosítja, emiatt az sémája *ENEMY\_PACKAGE\_ID(id)*. Ez az érték kerül átadásra az *ENEMY\_PACKAGE* tábla *enemy\_package\_id* oszlopának. Ezen kívül még az adott szörny azonosítóját (*ENEMY.id*) kell átadni az *ENEMY\_PACKAGE.enemy\_id* oszlopának.

**3.1.3 Map**

A tábla sémája *MAP(id, map\_path, enemy\_package\_id, npc\_package\_id, map\_name)*, ahol az *id* attribútum az azonosító. Látható, hogy a táblában van két külső kulcs, ami a *MAP-*et összeköti a „csomag” táblákkal. Így egyértelmű lesz, hogy az adott pályán milyen szörnyek, illetve NPC-k találhatóak.

**3.1.4 Backpack**

A hátizsáknál hasonló megoldást kellett keresnem, mint az ellenfelek eltárolásánál. Amikor létrehozunk egy új játékost, akkor kell, hogy létrejöjjön egy új hátizsák azonosító, ami jelölni fogja az adott játékos hátizsákját. Ez lesz a *BACKPACK\_ID* tábla, melynek sémája *BACKPACK\_ID(id)*. Továbbá szükség van egy olyan táblára, ami az adott hátizsák tényleges tartalmát el tudja tárolni, így ez lesz a *BACKPACK* tábla. A tábla sémája *BACKPACK(backpack\_id,item\_id,amount)*, amelyben el lesz tárolva az adott hátizsák azonosítója (BACKPACK\_ID.id), a hátizsákban lévő tárgy azonosítója (ITEM.id) és annak a mennyisége. Mivel a hátizsák kapacitása 24 cella, ezért a backenden kell lekezelni azt, hogy hány tárgy férhet bele a hátizsákba. Abban az esetben, ha az adott tárgy mennyisége nulla, akkor az adatbázisban ki kell keresni az őt eltároló sort és kitörölni, hiszen felesleges eltárolni nulla mennyiségű tárgyat.

**3.1.5 Item**

Az *ITEM* táblában nincs összetett adattag, illetve nincs benne külső kulcs. Az ITEM tábla sémája *ITEM(id, name, image\_path, consumable, default\_price\_per\_unit).* Eleinte az egységárakat magában a *BACKPACK* táblában akartam eltárolni, de rájöttem, hogy egyszerűbb eltárolni az *ITEM* táblában az alapértelemzett értéket (*ITEM.default\_price\_per\_unit*), amelyet később lehet módosítani.

**3.1.6 Gameplay**

A *GAMEPLAY* tábla az elmentett játékokat tartalmazza. Az elmentett játéknak tartalmaznia kell az aktuális játékost, a játékos hátizsákját, a pályát, amelyiken jelenleg tartózkodik, illetve a mentés dátumát. A tábla sémája *GAMEPLAY(id, player\_id, backpack\_id, current\_map\_id, created, updated).* A mentett játék betöltésekor szükség van egy azonosítóra, ami alapján be lehet tölteni a játékbeli objektumokat.

**4.fejezet**

**A játék bemutatása**

**4.1 Új játék**

A játék elindítása után lehetőségünk van új játék létrehozására, amelyhez rá kell kattintani a „New game” gombra. Az elején meg kell adnunk a játékos nevét. Ha ezt nem tesszük meg, akkor a mező felett hibaüzenet fog megjelenni.



4.1 ábra Új játékos létrehozása

Miután megadtuk a játékos nevét és rákattintunk a „Continue” gombra, a játék automatikusan létre fog hozni egy mentést alapértelmezett értékekkel. A játék egyszemélyes, ezért elegendő csak az utoljára beszúrt játékmentést vizsgálni, és az alapján betölteni a játékost, a pályát stb. A játékon belüli objektumok inicializálására a UserDataHandler osztályt használom, melyben statikus adattagokként eltárolom az objektumokat, így a RunningGame controllerben ezek elérhetőek lesznek és használhatóak. A játékos létrehozására az alábbi alapértelmezett értékeket használom:

* kép: "/adventure/entities/player/player.gif",
* szint: 1,
* maximális életerő, jelenlegi életerő: 100,
* védelem: 6,
* támadó erő: 10,
* kritikus sebzés: 0,
* szintlépéshez szükséges tapasztalatpont: 40,
* rendelkezésre álló tapasztalatpont: 0,
* x koordináta: 100,
* y koordináta: 120

**4.2 Játékos irányítása**

Maga a játékos irányítása a billentyűk segítségével történik, viszont a különböző menüknél lehet egeret használni.

|  |  |
| --- | --- |
| Parancs | Billentyű |
| felfelé | W |
| lefelé | S |
| jobbra | D |
| balra | A |
| támadás | SPACE |
| interakció (NPC) | E |
| hátizsák | B |
| karakter információ | I |
| mentés | F5 |
| játékmenü | ESC |

A játékmenüre a 4.2 ábrában láthatunk példát. Magát a menüt ESC billentyű lenyomásával tudjuk előhívni. A menü segítségével el tudjuk menteni a játékot, betölteni a játékot, visszamenni a főmenübe és kilépni a játékból.

A játékoknál szokásos megjeleníteni a játékos aktuális állapotát. Ide tartozik a játékos aktuális életereje, jelenlegi szintje és a rendelkezésére álló tapasztalatpontok. A 4.2 ábrában szintén látható a játékos állapotát jelző sáv a bal felső sarokban. Az életerőt és a tapasztalatpontot jelző sávokat ProgressBar objektumokkal valósítom meg. Az életerő sávnál úgy állítom be az értéket, hogy a jelenlegi életerőt elosztom a maximális életerővel. A tapasztalat sávnál értelemszerűen a jelenlegi tapasztalatpontot az elvárt tapasztalatponttal.



4.2 ábra. Játékmenü

**4.3 Mentés betöltése**

Lehetőség van az elmentett játék betöltésére. Ez megtehető a főmenüben, játék közben, vagy ha a játékos meghalt. Az elmentett játékokat egy String típusokat tároló ObservableListben tárolom el, amelyet a ListViewnak átadom, így egy táblázatban fognak megjelenni a kiválasztható mentések. A játék betöltéséhez rá kell kattintani duplán arra a sorra, amelyiket ki szeretnénk választani. Az adott sor tartalmazza a játék azonosítóját, a játékos nevét és a mentés idejét. Ha nincs elmentett játékunk, akkor maga a táblázat üres.

A kattintás során a játék azonosítója átadásra kerül a UserDataHandler osztály loadGame függvényének, amely elvégzi a szükséges betöltéseket. Azért is hoztam létre a UserDataHandler osztály adattagjait és függvényeit statikusan, mert bárhonnan elérhetőek, illetve egyszerűbben le lehet kezelni az új játék betöltését és a mentett játék betöltését.

A képen asztal látható

Automatikusan generált leírás4.3 ábra. Elmentett játékok kilistázása ListViewban

A mentés betöltése funkciónál le kellett kezelnem azt az esetet, amikor az adatbázisban nem szerepelt semmilyen mentés. Ha egy üres ListViewban duplán rákattintottam egy sorra, akkor kivételt dobott az alkalmazás, de egy sima null vizsgálattal meg lett oldva a probléma.



**5. fejezet**

**Tesztelés**

**5.1 Manuális tesztelés**

Minden egyes funkció implementálása után elvégeztem a manuális tesztelést. Ha valamilyen hiba felmerült, akkor rögtön neki tudtam állni a javításnak. Az új játék funkciót többször kellett ellenőríznem adatbázis szinten, mert a sok adattag miatt előfordult, hogy adattagokat rossz oszlopba szúrtam be, így a betöltés sem működött tökéletesen. A mentés funkció megvalósítása után egy kicsit gyorsabb lett a tesztelés folyamata, mert kevesebb kattintással jár a mentett játék betöltése.

Megkértem a családomat, hogy próbálják ki a játékot. A szükséges instrukciókat megadtam, és a tesztelést az én laptopomon zajlott. Tesztelés során nem merült fel súlyos hiba.

A családomon kívül még megkértem a volt hallgatótársamat, hogy ő is próbálja ki a játékot. A játékot Windows 10-ben fejlesztettem, viszont ő kipróbálta Linuxon. Az első szembetűnő hiba az volt, hogy a Brush Script MT betűtípust nem töltötte be az alkalmazás, de helyette Arialt használt a rendszer. Amikor megpróbált egy új játékot létrehozni, akkor súlyos hibát kapott, ami arra utalt, hogy fel kellett telepíteni az SQLite-ot. A telepítés után ugyanúgy felmerült a hiba a sikertelen adatbázishoz való csatlakozás miatt. Közösen megállapítottuk, hogy az application.properties fájlban rossz az útvonal, ugyanis nem relatívat adtam meg, hanem a teljes útvonalat. Az útvonal átírása után sikeresen elindult a játék.

**5.2 Statikus kódelemzés PMD-vel**

A manuális tesztelésen kívül szükség van a statikus kódelemzésre. A programon belül bizonyos sorok hibát okozhatnak (például lezáratlan fájlművelet stb.), amelyeket a fejlesztő nem feltétlenül vesz észre. Előfordulhat az is, hogy például felesleges osztályokat, csomagokat importálunk, ami miatt rosszabb lehet a futásidő. Egy fejlesztőnek eléggé költséges lehet saját magának átnézni a kódját, illetve nagyobb valószínűséggel nem veszi észre a saját hibáit.

A konzulensem javaslatára elvégeztem a statikus kódelemzést PMD segítségével. A Windowsnál elegendő letölteni a PMD-t, a csomagban lévő bin mappa útvonalát felvenni a rendszer környezeti változói közé, és futtatni az alábbi parancsot :

pmd.bat -d src/utvonal -R rulesets/java/quickstart.xml -f text

Az utasítás futtatása után a parancssorban meg lesz jelenítve az összes hiba, amelyet a PMD megtalált. Első alkalommal a legtöbb hibát a felesleges importok okozták és a fel nem használt privát változók. A kód refaktorálása során sorban haladtam a report alapján, azonban előfordultak olyan, hogy a PMD tévesen jelezte a hibát, emiatt ignorálnom kellett ezeket a sorokat. Ilyen volt például, amikor a RunningGame osztálynak a handleOnKeyPressed függvényére azt a visszajelzést dobta, hogy nincs használatban a privát metódus, pedig valójában a JavaFX használja. Az alábbi kódrészlet megmutatja, hogy hogyan néz ki a statikus kódelemzés eredménye.

|  |
| --- |
| 1. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\GameLaunch.java:3:   UnnecessaryImport:  Unused import 'adventure.controller.TempPage' 2. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:4:  UnnecessaryImport:  Unused import 'javafx.beans.property.IntegerProperty' 3. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:5:  UnnecessaryImport:  Unused import 'javafx.beans.property.SimpleIntegerProperty' 4. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:6:  UnnecessaryImport:  Unused import 'javafx.beans.property.SimpleStringProperty' 5. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:7:  UnnecessaryImport:  Unused import 'javafx.beans.property.StringProperty' 6. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:29: FormalParameterNamingConventions:   The method parameter name 'current\_health' doesn't match '[a-z][a-zA-Z0-9]\*' 7. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:137:    UselessParentheses: Useless parentheses. 8. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\Sprite.java:138:    UselessParentheses: Useless parentheses. 9. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\common\_files\configuration\Configuration.java:6: UseUtilityClass:    All methods are static.  Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning. 10. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\controller\CreateNewGame.java:6: UnnecessaryImport:  Unused import 'javafx.scene.Scene' 11. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\controller\GamePanel.java:3: UnnecessaryImport:  Unused import 'adventure.misc.UserDataHandler' 12. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\controller\RunningGame.java:18:  UnnecessaryImport:  Unused import 'javafx.scene.layout.\*' 13. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\controller\RunningGame.java:50:  SingularField:  Perhaps 'gameTile' could be replaced by a local variable. 14. C:\Users\deevi\Asztal\MyFirstAdventure\src\main\java\adventure\controller\RunningGame.java:58:  UnusedPrivateField: Avoid unused private fields such as 'NPCbackpackModal'. |

**Irodalomjegyzék**

[1] [Bruce Eckel – Thinking in Java](https://people.inf.elte.hu/delsaai/java/6Eckel%20-%20Thinking%20in%20Java%20(4th%202006)%20p1079.pdf)

*https://people.inf.elte.hu/delsaai/java/6Eckel%20%20Thinking%20in%20Java%20(4th%202006)%20p1079.pdf*

[2] [Andreas Pomarolli – JavaFX Programming CookBook](http://index-of.es/Varios-2/JavaFX-Programming-Cookbook.pdf)

*http://index-of.es/Varios-2/JavaFX-Programming-Cookbook.pdf*

[3] [Yakov Fain, JavaFX 8: The keyboard events are not being processed if a scene has only shapes](https://yakovfain.com/2015/02/13/javafx-8-the-keyboard-events-are-not-being-processed-if-a-scene-has-only-shapes/)

<https://yakovfain.com/2015/02/13/javafx-8-the-keyboard-events-are-not-being-processed-if-a-scene-has-only-shapes/>

[4] [What is JDBC](https://www.ibm.com/docs/en/informix-servers/12.10?topic=started-what-is-jdbc)

<https://www.ibm.com/docs/en/informix-servers/12.10?topic=started-what-is-jdbc>

[5[] What Is SQLite?](https://www.sqlite.org/index.html)

<https://www.sqlite.org/index.html>

[6] [Git](https://en.wikipedia.org/wiki/Git)

<https://en.wikipedia.org/wiki/Git>

[7] [Projekt repository](https://github.com/Eyill/Szakdolgozat/tree/adventure)

*https://github.com/Eyill/Szakdolgozat/tree/adventure*

[8] [What is Maven?](https://maven.apache.org/what-is-maven.html)

*https://maven.apache.org/what-is-maven.html*

[9] JavaFX [Tutorial](https://www.javatpoint.com/javafx-tutorial)

*https://www.javatpoint.com/javafx-tutorial*

[10] [PMD Source Code Analyzer](https://pmd.github.io/)

*https://pmd.github.io/*

[11][Kedar Vijay Kulkarni, A step-by-step guide to Git](https://opensource.com/article/18/1/step-step-guide-git)

<https://opensource.com/article/18/1/step-step-guide-git>

[12] [JDBC Tutorial](https://www.tutorialspoint.com/jdbc/index.htm)

<https://www.tutorialspoint.com/jdbc/index.htm>

[13] Mike McGrath – Java in easy steps (6th edition)

[14] A projektben használt textúracsomagok:

* o\_lobster , <https://o-lobster.itch.io/simple-dungeon-crawler-16x16-pixel-pack>
* Karwisch, <https://karwisch.itch.io/pxui-basic>
* [0x72](https://0x72.itch.io) , <https://0x72.itch.io/dungeontileset-ii>

**Nyilatkozat**

Alulírott Pokrovszkaja Katerina programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, alapdiploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Továbbá nyilatkozom arról, hogy szakdolgozatom csak elektronikus formában lett beküldve tekintettel a COVID helyzetre (a SZTE TTIK Tanulmányi Osztály felhívása alapján).

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2021. november 30.

aláírás

**Köszönetnyilvánítás**