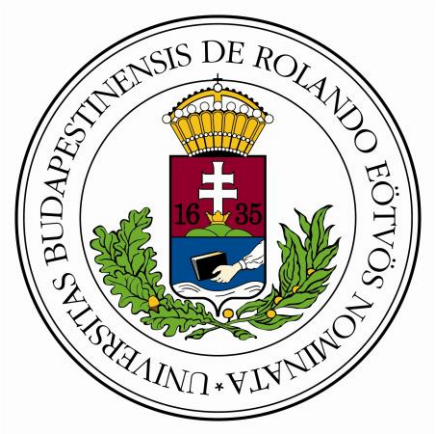


Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Többplatformos játékfejlesztés libGDX-szel

Mészáros Mónika

Tanársegéd

Máthé Levente

Programtervező Informatikus BSc

Budapest, 2017.

Témabejelentő: a szakdolgozat bekötve kell, hogy tartalmazza a kitöltött es jóváhagyott (az Informatikai Kar dékánja által aláírt) Szakdolgozat-téma bejelentőt. Ennek a lapnak a helyére kell bekötni.

Tartalomjegyzék

1. Bevezetés

A dolgozat a témája a platformfüggetlen (PC és mobil) játékfejlesztés lehetőségeinek megismerése, előnyeinek és hátrányainak bemutatása egy 2D platformjáték elkészítésén keresztül. Bemutatom a játékok készítése során gyakran alkalmazott tervezési mintákat, játék architektúra opciókat, feltérképezem a (2D platform) játékok szokványos elemeit, mint az animáció, pályaszerkesztés, fizika, felhasználói interakció stb.

Mivel a platformfüggetlen programozás a dolgozat fő témaköre, külön kitérek az ebből következő kihívásokra, mint például az irányításra, ami az eszközök különbözősége miatt az egyik legnehezebb témakör a jó felhasználói élmény elérése szempontjából: a játék ugyanolyan könnyen irányítható kell, hogy legyen billentyűzettel, mint érintőképernyős, virtuális gombokkal. Hasonlóan kihívásokkal teli a játékmenet és a grafika is: az okostelefonok kisméretű (bár egyre növekvő) képernyőjén, és nagyobb méretű modern monitorokon (vagy akár televíziókon) egyaránt könnyű navigációt és minőségi grafikát kell tudni biztosítani.

Természetesen a technikai határokkal is foglalkozok: bár az okostelefonok rohamos ütemben fejlődnek, az olcsóbb, de még akár a középkategóriás készülékek is jóval erőforrás szegényebbek a modern PC-knél, így ügyelni kell az erőforrások hatékony csomagolására és betöltésére, vagy például a túlzott Garbage Collector használatra.

A platform játékok történelme az 1980-as években kezdődik, ebbe a műfajba tartoznak például a korábbi Mario játékok, a Contra, és a Castlevania sorozat, modern képviselői pedig például a Super Meat Boy vagy a Trine sorozat. A hagyományos 2D megoldások óta a játékok nagy fejlődésen mentek keresztül: ma már 3D vagy 2.5D megoldásokkal is találkozhatunk, és a játékmenet is sokféle: míg például Super Meatboy játékmenete viszonylagosan egyszerű, a Salt and Sanctuary egy hatalmas képességfával és sokféle felszereléssel rendelkező oldalnézetű szerepjáték.

A mobil eszközök (okostelefonok, táblagépek) népszerűségének emelkedésével megjelentek a játékok is a platformokon, mára pedig az alkalmazások egyik legnagyobb részét teszik ki. Hatalmas sikernek örvendtek példeául az Angry Birds játékok, újabban pedig a Clash of Clans, vagy a Clash Royale. A készülékek fejlődése lehetővé tette a régebbi számítógépes és konzol játékok portolását is, és a korábban csak PC-ken elérhető címek iOS-en és Androidon is megjelentek. A népszerű kártyajáték, a HearthStone iOS

és Android verziói ugyanabban az évben jelentek meg, mint a PC kiadás, de a Minecraftank is van mobil verziója: a Pocket Edition.

Egy modern, bár 2D grafikával rendelkező játékot készítettem, átugorható akadályokkal; ellenségekkel, akikkel a játékos megküzdhet, vagy akiket megpróbálhat elkerülni. Csontalapú animációt és kézzel készült, csempe alapú pályákat használtam.

2. Felhasználói dokumentáció

2.1. A program témája

A játék egy - a középkori Európa által inspirált - fantáziavilágban a játszódik, a főhős pedig Ragnar Lothbrok, az izlandi sagák egyik szereplője. Ragnar a társaival éppen a frankok földjén fosztogatott, amikor katonák jelentek meg a semmiből, és Ragnar a keletkezett zűrzavarban elválasztódott bajtársaitól. Így egyedül kell megmenekülnie, úgy, hogy közben a szerzett kincsre is ügyelnie kell.

Műfaját tekintve a program 2D, oldalnézetes platform játék, a játékos Ragnart irányítja, és a pálya egyik feléből el kell jutnia a másikba, közben akadályokat átugorva, ellenségeket legyőzve, és ügyelve, hogy minél több kincs megmaradjon.

2.2. Rendszerkövetelmény

A játék által támogatott platformok: Windows, Linux, macOS személyi számítógépen, Android mobil eszközökön. A minimális ajánlott felbontás 1280 x 720, az ajánlott képarány pedig 16:9.

2.2.1. PC-n

- Java Runtime Environment 7 vagy újabb
- OpenGL 4.1-képes videókártya vagy jobb

2.2.2. Androidon

- Android 4.0.3 vagy újabb
- OpenGL ES 2.0 vagy jobb

2.3. A program üzembehelyezése

2.3.1. PC-n

A futtatáshoz szükség van a Java Runtime Environment szoftverre. Ha nincs a számítógépen telepítve, le kell tölteni a https://java.com/en/download/ címről. A letöltött fájlt futtatva kövessük a megjelenő utasításokat a Java telepítéséhez.

2.3.2. Androidon

A Lothbrok.apk fájlt először át kell másolni az Android készülék Downloads vagy Letöltés mappájába a számítógépről. Ehhez csatlakoztatni kell az eszközt a PC-hez, majd Androidon kiválasztani a File Transfer, azaz Fájlok átvitele opciót a megjelenő USB beállítások közül. Ezután megnyitható az Android készülék fájlrendszere a számítógépen és át lehet másolni az .apk fájlt.

A telepítés előtt engedélyezni kell a külső forrásból érkező alkalmazások telepítését. Ezt a Settings/Beállítások menü Security/Biztonság alpontjában tehetjük meg, az Unknown Sources/Ismeretlen Források opció bekapcsolásával.

Az átmásolt állomány megjelenítéséhez és futtatásához szükség van egy fájlkezelő alkalmazásra az Android készüléken. Amennyiben nincs ilyen gyárilag telepítve, le kell tölteni egy ilyen alkalmazást, például a következő linken található appot: https://play.google.com/store/apps/details?id=com.asus.filemanager&hl=en.

A fájlkezelő alkalmazásban keressük meg a Downloads/Letöltés mappát, és érintsük meg az átmásolt Lothbrok.apk ikont. Ezzel telepíthetjük a játékot a készülékre.

2.4. A program futtatása

2.4.1. PC-n

Ha telepítve van a Java a számítógépen, a játék futtatható a Lothbrok.jar fájlra való dupla kattintással, vagy parancssorból a következő parancs kiadásával: java -jar Lothbrok.jar

2.4.2. Androidon

Telepítés után a program elérhető a telefon vagy táblagép alkalmazás fiókjában, vagy parancsikonként valamelyik kezdőképernyőn. Ezek valamelyikét megérintve futtatható az alkalmazás.

2.5. A program használata

A program indulásakor rövid töltés és töltési képernyő után a főmenü fogad. Itt két lehetőségünk van, a „Play” gombra kattintva indíthatjuk el a játékot, a „Quit” gombbal pedig kiléphetünk a programból.

A játék indulásakor Ragnar leesik az égből az egyik platformra, már ekkor irányítható. A cél: elérni a pálya túlsó oldalán lévő csillagot. A képernynő a pályán és a karaktereken kívül egyéb információkat láthatunk, mint például a játékos fennmaradó élete. Ez az ún. „Heads-up Display”, röviden HUD. A képernyő tehát két fő részből áll: a játéktérből és a HUDból. Androidon láthatók az irányításhoz szükséges gombok és a virtuális joystick is. A játék szüneteltethető, ekkor a „Pause” menü jelenik meg. Innen kiléphetünk a játékból, vagy folytathatjuk azt.

2.1. ábra A főmenü és a játéktér

2.5.1. A játéktér

A platformokon lehet jobbra-balra közlekedni, ugrálni. A kamera követi a játékost, így mindig Ragnar közvetlen környezete látható. A pálya alján helyenként víz található, ha ebbe beleesünk, az azonnali halállal jár és véget ér a játék.

A platformokon ellenségek járőröznek. A sárga szakállas, kék ruhás karakter a játékos, a többiek ellenfelek. Ha egy ellenfél közelébe érünk, az elkezd Ragnar felé mozogni, és ha elég közel ér, megtámadja a játékost. Ragnar három kardcsapást kibír, az ellenségeknek viszont egy is elég.

Minden mozdulat, a mozgás, ugrás, támadás esetén Ragnar kincse fogy: arany pénzérmék potyognak a poggyászából, ezért minden lépés megfontolandó!

2.5.2. A HUD

A bal felső sarokban egy gyémánt ikon mellett láthatjuk a maradék kincsünket: ebből mindig egyet veszítünk, amikor a játéktéren is látható, hogy elgurul egy pénzérme.

A jobb felső sarokban látható a maradék élet: annyi szív ikon, ahány élete még maradt a játékosnak, ami kezdetben három.

2.5.3. Irányítás

Az irányítás PC-n és Androidon különböző, a platform sajátosságai miatt.

Androidon a karaktert a képernyőn megjelenő gombokkal és joystickkal lehet irányítani, a játékot szüneteltetni és a „Pause” menüt is egy ilyen gombbal lehet elérni. PC-n az irányításhoz a billentyűzet használható.

2.5.4. Irányítás PC-n

A játékos karakter a következő billentyűkkel irányítható:

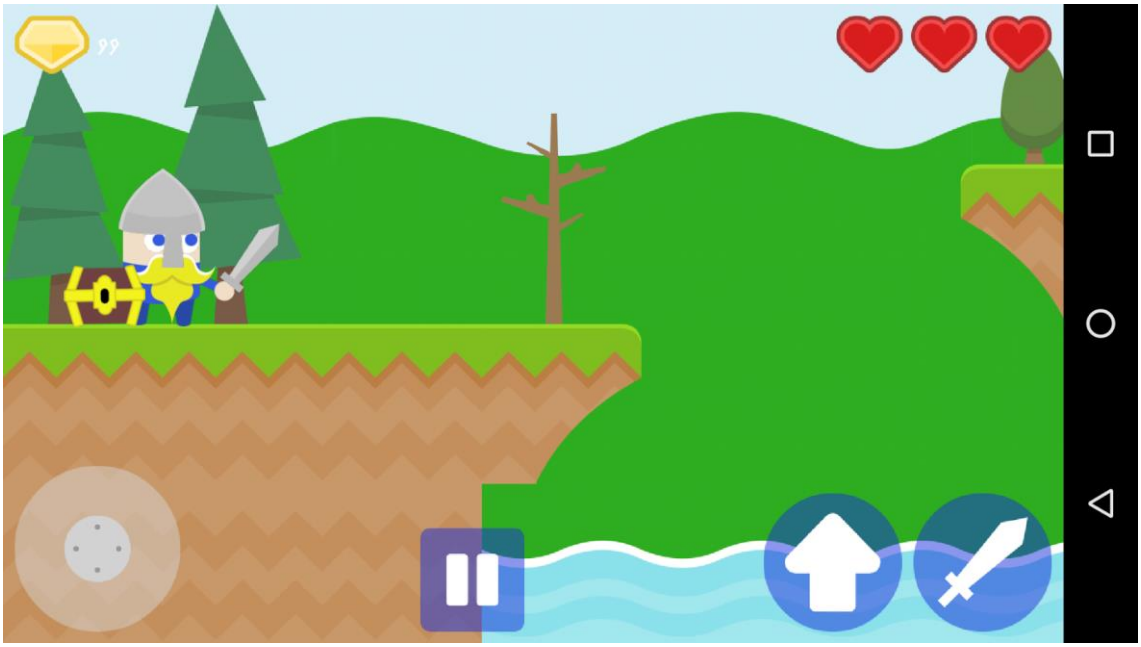
- **„A”** mozgás balra
- **„D”** mozgás jobbra
- **„W”** ugrás
- **„H”** támadás

A „Pause” menü az **„ESC”** billentyűvel hozható elő

2.5.5. Irányítás Androidon

A karakter jobbra és balra mozgatható a virtuális joystcikkal, ami a bal alsó sarokban található. A jobb alsó sarokban lévő kard ikonnal jelzett gombbal támadni, a nyíl ikonos gombbal pedig ugrani lehet.

A „Pause” menü az alul középen található, négyszögletes gombbal jeleníthető meg.



2.2. ábra Az irányításhoz használható virtuális kezelőfelület androidon

2.5.6. A játék vége

A játéknak kétféleképpen lehet vége: a játékos meghal (beleesik a vízbe, vagy legyőzi az egyik ellenfél), vagy eléri a csillagot és nyer.

Előbbi esetben a „Game Over” képernyő, utóbbiban pedig a „You won” képernyő jelenik. Mindkét esetben visszatérhetünk a főmenübe „Main Menu” gombra kattintva.

2.3. ábra A "játék vége" menük: Game Over és You Won

3. Fejlesztői dokumentáció

3.1. A megvalósítandó feladat

A megvalósítandó program egy 2D oldalnézetes platform játék. Rendelkezik egy főmenüvel, ahonnan elindítható a játék. A játékmenet szüneteltethető, és a „Pause” menüből visszajuthatunk a főmenübe. A játék végén a „Game Over” vagy „You Won” menüből szintén visszajuthatunk a főmenübe. A navigációt gombok segítik.

A játék egy 2d, csempékből és dekorációbál álló pályán játszódik. A csempék alkotta platformokon a karakterek mozoghatnak. A háttérelemek parallax mozgással a 3D illúzióját keltik.

A játékos egy platform játék esetén elvárható fizikával rendelkezik: tud jobbra-balra mozogni, ugrani és esni, viszont nem tud felborulni, pattogni, vagy csúszni, és nem „ragad” a falhoz. Mozgás vagy támadás esetén kincset veszít, ami viszont valós fizika szerint mozog: esik, pattog és gurul.

Az ellenfelek jobbra-balra járőröznek a kezdeti pozíciójuk egy sugarában. Ha a játékos a sugáron belülre kerül, az ellenfél elindul felé, majd ha elég közel érnek, megtámadja. A játékos szintén meg tudja támadni az ellenfeleket, és legyőzhetik egymást.

A játékos meghal, ha az élete elfogy: ha a vízbe esik, az összes életét elveszíti, ha pedig egy ellenfél eltalálja, egyet veszít.

A játékos élete és kincse legyen számon tartható egy Heads-up Display (HUD) segítségével.

Az irányítás PC-n billentyűzettel, Androidon pedig a HUD-on megjelenő gombokkal és joystickkel történik.

3.2. A megvalósítás eszközei

3.2.1. Lehetőségek

A dolgozat írásakor több lehetőség is rendelkezésre állt játékfejlesztéshez használható programozási nyelvek és technológiák tekintetében, ezek három csoportba sorolhatók:

- saját játékmotor készítése
- keretrendszer használata

- 3rd party játékmotor alkalmazása

Ezek közül a középső opciót választottam, mivel a saját játékmotor fejlesztése egy különböző téma, nem érdemes használni, ha a cél a játék- és nem a motorfejlesztés. 3rd party motor alkalmazása pedig a dolgozat témájához túl absztrakt, hiszen a legtöbb módszer és játékelem, amit be szeretnék mutatni, már implementálva van, és el van rejtve a felhasználó elől. Keretrendszert alkalmazva, viszont, be tudom mutatni a játékfejlesztés során alkalmazott módszereket és tervezési mintákat.

Szerencsére játékfejlesztő keretrendszerekből is nagy a választék, szinte minden népszerű programozási nyelvhez találunk megoldást. Ilyenek a teljesség igénye nélkül:

- MonoGame, C#
- Löve, Lua
- Cocos2d-x, C++
- libGDX, Java

A választásom a libGDX-re esett, a minőségi és teljes dokumentáció, és a nagy méretű, segítőkész közösség miatt.

3.2.2. libGDX

A játék Java nyelven, a libGDX keretrendszer segítségével készül. A libGDX egy nyílt forráskódú, cross-platform játékfejlesztő keretrendszer Java nyelven. Segítségével 2D és 3D játékok készíthetők, absztrakcióinak köszönhetően mentesít az alacsony szintű kód írásától (pl. OpenGL), és számtalan platformra kiadható az elkészült termék.

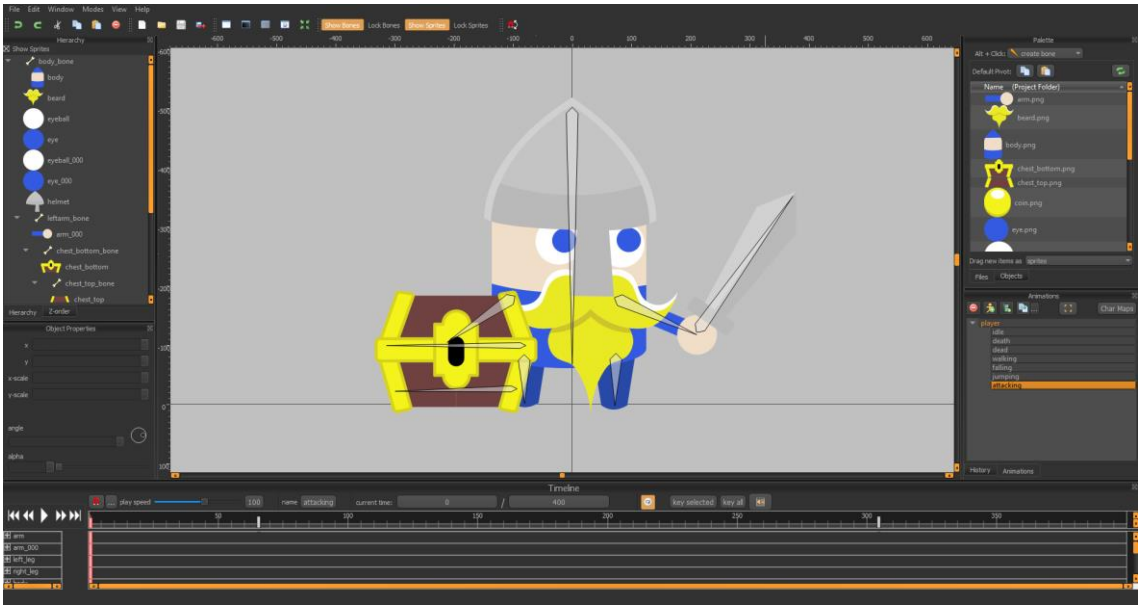
A jó teljesítményt az OpenGL ES alapú megjelenítés, és Garbage Collectort minimálisan használó gazdag API biztosítja.

3.2.3. Animáció – Spriter

A karakterek mozgásához, tevékenységeinek megjelenítéséhez csont alapú animációkat használtam. Ezek lényege, hogy a szerkesztőprogramban a behúzott képek alá egy csontvázat állítunk, és a karaktert a csontok mozgatásával, forgatásával animáljuk. A csontok szülő-gyerek kapcsolatban állhatnak – ha a szülő mozog, a gyerekeit is magával viszi. Minden csonthoz beállíthatunk egy, vagy több, a karaktert alkotó képet, amit mozgat.

A csont alapú animáció előnye a hagyományoshoz képest, hogy gyorsabban lehet látványos eredményt elérni, és minden képből csak egy példányt kell tárolni, mivel az animációt a csontok időponthoz kötött koordinátái és elforgatási szögei határozzak meg.

Az animációk létrehozásához, szerkesztéséhez a Spriter animációs szoftvert használtam. A létrejött animációk a felhasznált képekből és egy .scml kiterjesztésű fájlból állnak – utóbbi tárolja az animáció adatait.



3.1. ábra A Spriter animációs program kezelőfelülete

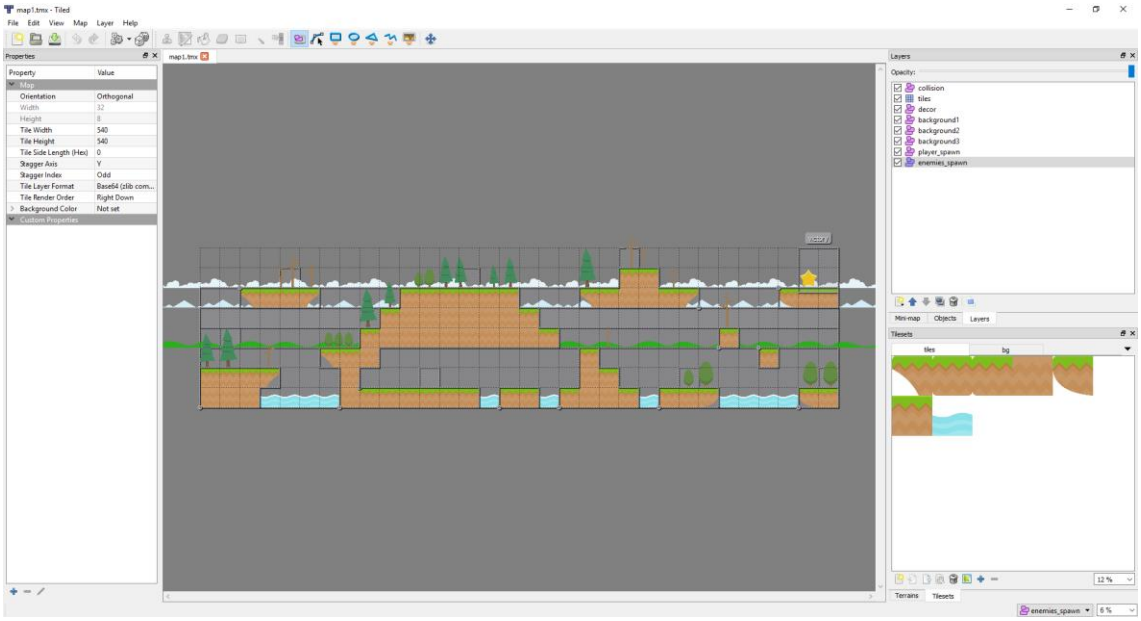
A libGDX nem támogatja az. scml fájlok beolvasását, ezért a közösség egy tagja, trixt0r által készített implementációját alkalmaztam, amit kiegészítettem. [1]

3.2.4. Pályaszerkesztés – Tiled

A játékban csempealapú, ortografikus pályán játszhatunk, ami a platform csempéken kívül háttér- és dekorációs elemeket is tartalmaz, továbbá a Box2d fizikai motor által használt poligonokat, és a játékos illetve ellenfelek kezdő pozícióját is tartalmazza.

Ilyen pályát a Tiled általános célú, csempe alapú 2D pályaszerkesztő programmal készítettem. Egy Tiled pályában egy csempéhez egy előre megadott méretű kép, és tetszőleges mennyiségű tulajdonság tartozik. A grafikus szerkesztőprogramokhoz hasonlóan rétegek hozhatók létre a különböző pályaelemek elválasztására, amik három típusúak lehetnek: Tile layer, Object layer és Image layer. A Tile layereken helyezhetjük el a csempéket, az Object layeren pedig minden mást: poligonokat, képeket, amiket

tulajdonságokkal ruházhatunk fel, hogy felhasználjuk a játék futásakor. Az Image layerben egy képet lehet tárolni, a dolgozat keretein belül nem használtam.



3.2. ábra A Tiled univerzális 2D pályaszerkesztő program kezelőfelülete

Az elkészült pálya egy .tmx kiterjesztésű xml fájlba kerül mentésre, és tömörítést is használhatunk. A Tiled pályák betöltését és kirajzolását támogatja a libGDX, ezért ezt az API-t használtam.

3.2.5. Betűtípus

A betűtípus fájlból a libGDX FreeTypeFont kiegészítőjét alkalmazva hozom létre a játékban megjeleníthető BitmapFontokat.

3.2.6. Fizika – Box2d

Bár a játékos és az ellenfelek saját, egyszerűsített és a célra specializált fizikával rendelkeznek, a hulló, guruló, pattogó kincsek mozgásához a Box2d fizikai motort használtam, ami kiegészítésként elérhető a libGDX keretein belül.

Segítségével valós fizikai paraméterekkel rendelkező testek definiálhatók poligonok használatával – megadható a világ gravitációja, a testeket alkotó alkatrészek sűrűsége, rugalmassága stb.

3.2.7. Projektgenerálás

A projekt létrehozására a libGDX projektgeneráló eszközt használtam [2]. Az eszközben megadható a projekt neve, a csomag, a főosztály neve, a célkönyvtár, valamint be kell állítani az Android SDK helyét. Kiválaszthatók a célzott platformok is, valamint kiegészítéseket is megadhatunk, például a Box2d fizika könyvtárat.

Az eszköz futtatásának eredménye egy Gradle projekt, amelyben a különböző platformok (indító) kódja, valamint a közös kód (core) külön Gradle modulokba kerülnek.

3.2.8. Disztribúció

PC

A desktop Gradle modul dist nevű taskját meghívva készíthetjük el a futtatható .jar állományt, ami a desktop modul build/libs könyvtárába kerül.

Android

Android Studioban a „Build” menü alatt található „Generate Signed APK” opcióval készíthető el az aláírt Android .apk fájl az android modul build/outputs/apk könyvtárában, ami a mobil készülékre másolva telepíthető és futtatható.

3.3. Az erőforrások előkészítése

A PC-k és mobil eszközök különböző karakterisztikái miatt a játék négy méretet támogat, a csomagolt erőforrások szempontjából. Ezek előkészítésének és csomagolásának automatizálására Gradle taskokat alkalmaztam.

Elsősorban a textúrák szempontjából fontos ez – 4K felbontású monitorokon és kisebb képernyős, 720p kijelzővel rendelkező telefonokon is szép kell, legyen a grafika. Mivel a térkép fájlok, animációk függhetnek a textúrák méretétől, ezeket is módosítani kell szükség szerint. A négy támogatott méret a következő:

- XL – 3840 x 2160
- L – 2560 x 1440
- M - 1920 x 1080
- S – 1280 x 720

Az erőforrások tárolására két mappát hoztam létre, két helyen: a core modul „assets_raw” mappájába kerülnek a nyers, csomagolandó erőforrások. Az android modul „assets” könyvtárába kerülnek a csomagolt, kész anyagok. A libGDX is ebben, az utóbbi mappában keresi alapértelmezetten az erőforrásokat, innen kerülnek betöltésre a játékban.

3.3.1. Textúrák

A játékban a hatékony rajzoláshoz a képi elemeket, textúrákat egy nagyobb, közös textúra fájlba kell csomagolni (Texture Atlas), ráadásul az összes támogatott méretben. Ez kézzel nem csak fáradalmas, de nem is feltétlenül eredményezi a leghatékonyabb méretű Atlast. E feladat automatizálására a libGDX TexturePacker kiegészítését alkalmaztam, Gradle taskok keretén belül. A TexturePacker segítségével a megadott paraméterek szerint hatékonyan elrendezett Atlasokat kapunk a kép fájlokból. A kívánt Atlas újra pakolásához csak meg kell hívni az adott Gradle taskot.

3.3.2. Animáció

A Spriter animációk függnek a hozzájuk tartozó képektől, ezért el kell őket menteni az összes kívánt méretben, erre van lehetőség a szerkesztő programban.

3.3.3. Pályák

A Tiled által készült pályák szintén tartalmaznak adatot a textúrák méretéről, viszont nincs lehetőség a különböző méretű mentésre, mint a Spriter esetén. Ezért a pályát csomagoló Gradle task a createAtlasReadyMapCopies.py script segítségével módosítja a .tmx fájlokat, hogy az összes támogatott méretben működjenek.

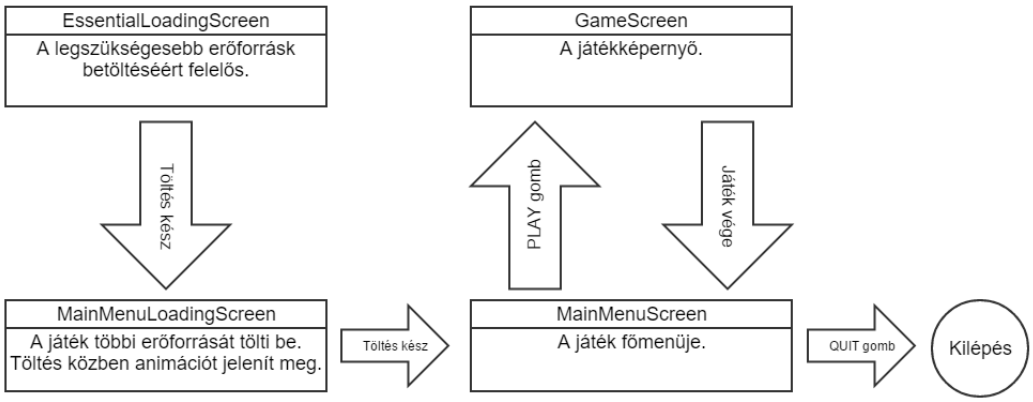
3.4. Megvalósítási terv

3.4.1. A screen tervezési minta

A program háromféle fő komponensre oszthatók a screen játékfejlesztési design pattern szerint, ezek mind a libGDX Srcreen interface-ének implementációi – töltőképernyők, főmenü és játékmenet.

A program indulásokor először az EssentialLoadingScreen jelenik meg, betölti a legszükségesebb erőforrásokat. Ezt követi a MainMenuLoadingScreen, ami a többi erőforrást tölti be, és közben lejátszik egy animációt. Ezzel a töltés végére értünk.

Következő lépésben a MainMenuScreen töltődik be, ahonnan a megfelelő gombbal eljutunk a végső állapothoz – a GameScreenhez.



3.3. ábra A különböző Screenek az alkalmazás életciklusában

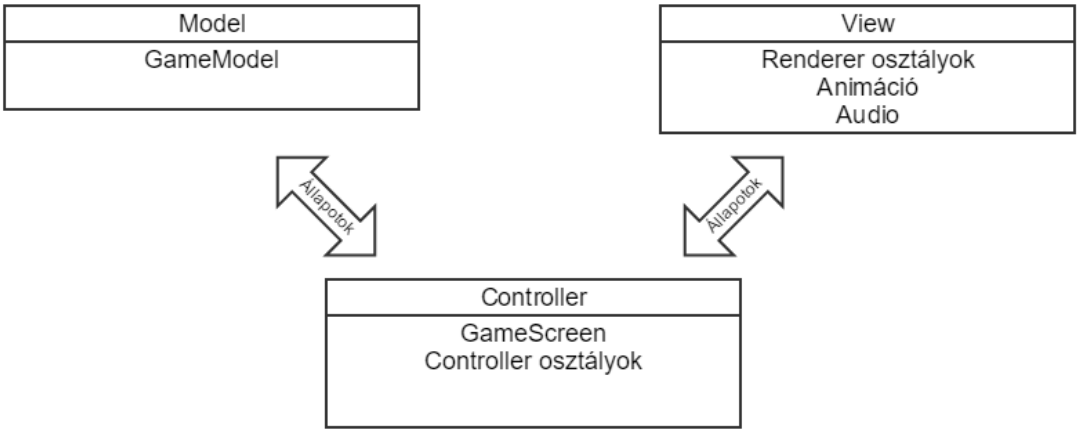
A töltőképernyő osztályok egyszerűek, feladatuk az erőforrások betöltése és töltés képernyő megjelenítése. Hasonlóan egyszerűbb a főmenü felépítése, itt a libGDX Scene2d API-ját használtam, rábízva a megjelenítés és az input kezelését.

Mivel a célom az volt, hogy a játék közben megjelenő menük – a „Pause”, a „Game Over” és a „You Win” – hátterében látható legyen a játékmenet, ezek a menük nem új Screen implentációként, hanem mint a játéktér fölé rajzolt felhasználói felület jelennek meg.

3.4.2. MVC architektúra

A program játékmenettel foglakozó része, a GameScreen, egy MVC-szerű (Model – Modell, View – Nézet, Controller – Irányító) architektúra szerint készült: a modellben csak állapotokat és viselkedést tárolunk, a nézet pedig csak a rajzolásért, illetve hangok vagy zene lejátszásáért felel. A kettőt a controller rétegben kapcsoltam össze: ezek a libGDX Screen interface implementációja (GameScreen), illetve ezen belül az input- és mesterséges intelligencia alapú irányítók – Controller osztályok – ezek irányítják a játékos és az ellenség mozgását, illetve a „Pause” menü elérését is.

A modell közvetlenül nem kommunikál a nézet réteggel, állapotai az irányítón keresztül kerülnek a megjelenítőhöz. A nézet is az irányítón keresztül küld adatokat (mint az ütközésellenőrzéshez szükséges négyszögetek) a modellnek.



3.4. ábra MVC architektúra és a komponensek kapcsolata

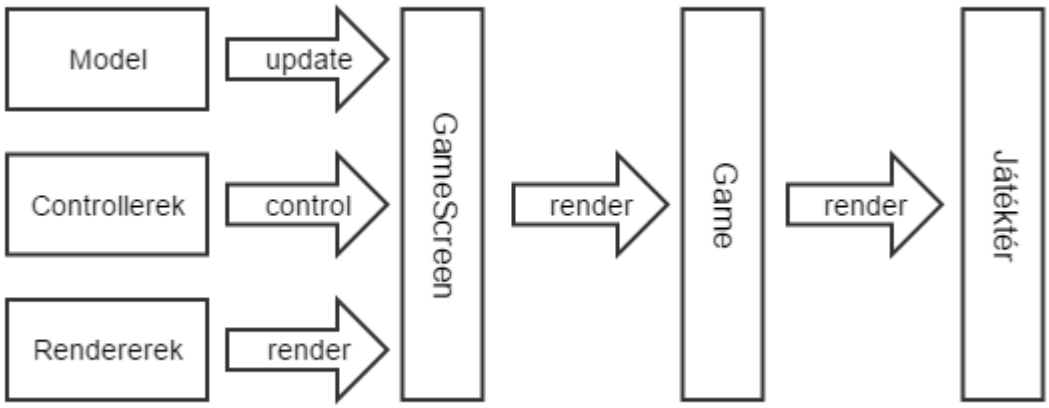
3.4.3. A game loop tervezési minta

A videójátékok klasszikus tervezési mintája, a Game loop szerint egy ciklusban másodpercenként többször (pl. 30, 60 stb.) frissítjük a teljes játékmenetet (GameScreen), és újra rajzoljuk a világot. A főmenü és a töltőképernyők is hasonlóan működnek: előbbi figyel az eseményekre és frissíti illetve kirajzolja a felhasználói felület widgetjeit, utóbbi pedig a háttérben tölt és közben esetleg animációt jelenít meg.

A loop már implementálva van a keretrendszerben, ez az ApplicationListener interface render callback metódusát hívja minden iterációban. A Game osztály implementálja ezt az interface-t és egy Screen kezelő rendszerrel egészíti ki – így az aktív Screen render metódusa is meghívódik.

Hasonlóan kezeli a keretrendszer a többi callback metódust, amit meghívhat a keretrendszer az applikáció élettartama során, például: resize, pause stb.

A program belépési pontja minden platformon a Game osztályból leszármazó Lothbrok osztály.



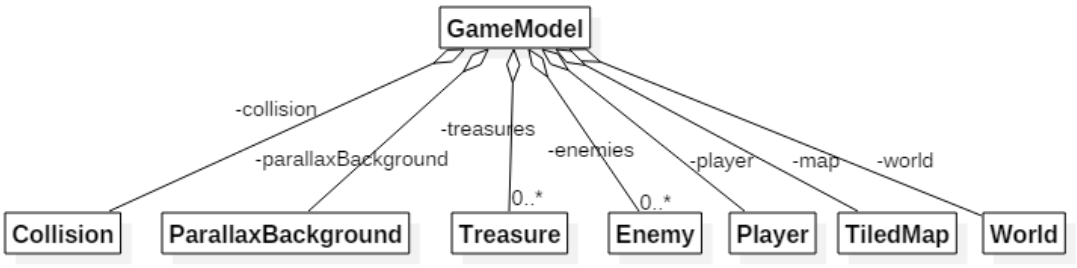
3.5. ábra Az MVC architektúra és a screen, illetve game loop tervezési minták együttműködése

3.4.4. MVC - Model

A játék világát a GameModel osztály modellezi, mezőként megtalálható benne a pálya, a játékos, az ellenségek, a harcrendszer ütközési rendszere, az elveszített kincs és a fizikáért felelős világ.

A pálya tárolja, hogy melyik csempék blokkolnak (a játékos és az ellenségek ütköznek velük, platformként használhatják őket), és hogy hol van a játékos és az ellenségek kezdőpozíciója. Ezen kívül a Box2d fizikához szükséges poligonokat is a térképből nyerhetjük ki. Ezek hozzáadva a Box2d világhoz, ütközőfelületként szolgálnak a játékos által elvesztett kincsek számára.

Kincs megadott időközönként jelenik meg, a játékos ládájánál. Ezt a pozíciót a GameScreenen keresztül a játékos animációjából (nézet réteg) kapjuk meg. Ez alapján rögtön a megjelenéskor erő hat az érmére, így az elrepül, majd a platformokon elgurul a Box2d világ szabályai szerint.



3.6. ábra A modellt alkotó osztályok egyszerűsített UML osztálydiagramja

Az entity-component-system (ECS) tervezési minta

A játékos és az ellenség osztályokat hasonló módon implementáltam. Hagyományosan, a játék entitásokat (játékos, ellenfelek, egyéb karakterek) több szintű leszármazással implementálták.

A közös Entity osztályból származott az összes szereplő, akár több köztes rétegen keresztül, ez azonban többször konfliktust okozott – egy osztály akár több őshöz is tartozhatna, amit a legtöbb modern programozási nyelv - így a Java sem – támogat (többszörös öröklődés). Feltételezve például Tree és Enemy osztályokat, az EvilTree osztálynak mind a kettő őse kéne legyen. [3]

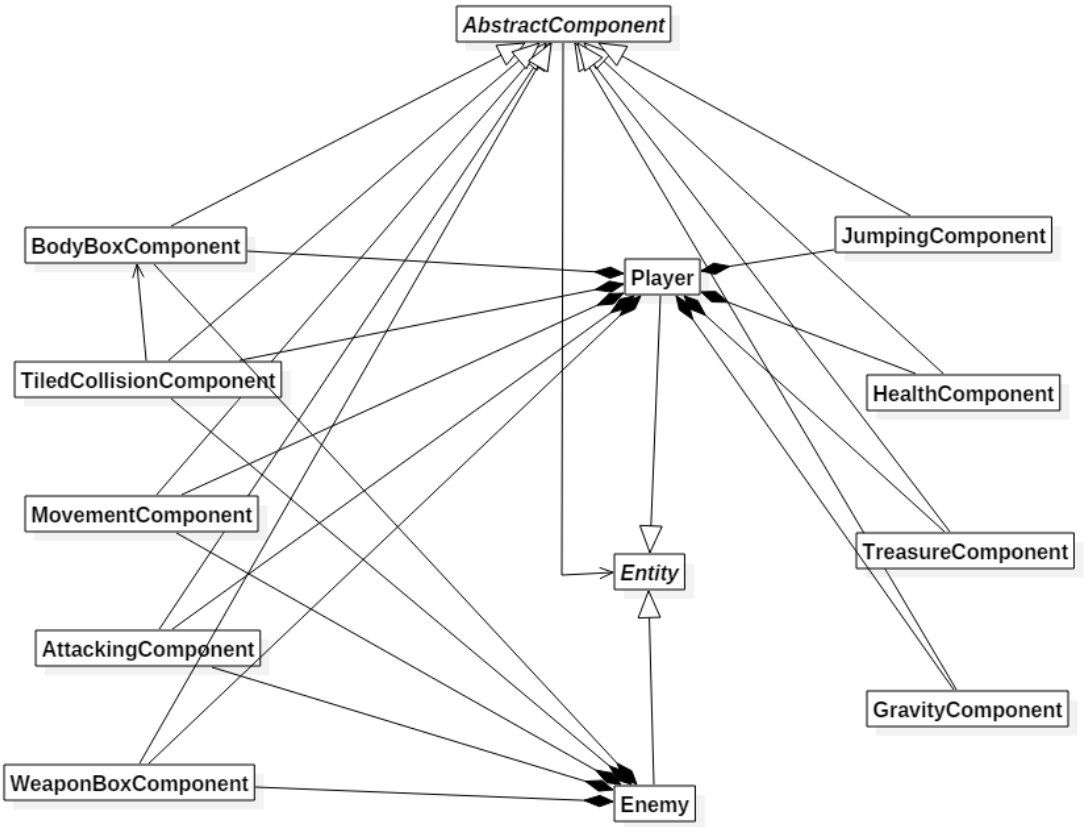
Esetünkben például a játékos egy mozgó, ugró, és támadó entitás. Ennek megfelelően lehetnek a szülei a MovingEntity, a JumpingEntity és AttackingEntity osztályok. Ez azonban többszörös öröklődés lenne, így egyszerűsíteni kellene, hogy minden osztálynak csak egy őse legyen – például a játékos ősosztálya legyen a MovingJumpingAttackingEntity. Ez nem csak túlbonyolítja a modellt, de ha az ellenség nem képes például ugrani, létre kell hozni még egy újabb ősosztályt MovingAttackingEntity névvel.

Ezen rendszer helyett az Entity-component-system tervezési minta egy módosított változatát implementálva alkottam meg a játékos és ellenfél osztályokat.

Az ECS minta szerint az entitások komponenseket tartalmazó halmazok, a komponensek pedig adatok. Az adott komponensekből álló entitásokat pedig adott rendszerekkel lehet módosítani. Például a Movement komponenssel rendelkező entitásokon értelmezve van a MovementSystem rendszer: a rendszer a komponens adatait módosítja - az x, y koordinátákat.

A dolgozat implementációja szerint az Entity osztály egy háromféle állapottal (mozgás, élet, tevékenység), továbbá pozícióval és iránnyal rendelkező főosztály. Ebből származnak az Enemy és a Player osztályok. A többszörös, köztes származási rétegek helyett ezek az AbstractComponent osztályból leszármazó komponenseket tartalmaznak, mint mezők. Ezek a komponensek azonban rendelkeznek alapvető metódusokkal (szemben az ECS mintával), a mozgás komponensbe például már be vannak építve a moveTo, moveLeft és moveRight funkciók, így ezek a komponensek az ECS-szerinti rendszereket is magukba foglalják. A komponenseket együtt alkalmazó funkciók a Player és Enemy osztályokba kerültek.

Mivel a játékban csak két entitás van, elegendő a fenti rendszer és nem szükséges egy teljes ECS implementáció. Ennek előnye, hogy az entitásokat könnyebb elválasztani a program többi részétől, például az irányításól, ami logikailag nem is tartozik a modellhez.

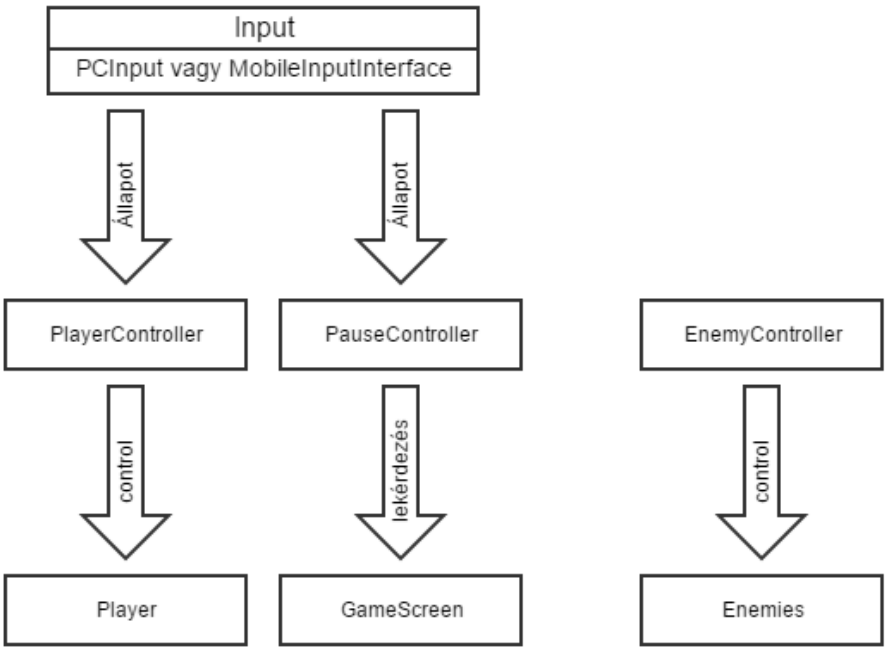


3.7. ábra A játékos és ellenség osztályok, illetve komponenseik egyszerűsített UML diagramja

3.4.5. MVC – Controller

A GameScreenen kívül a Controller osztályok alkotják az irányító réteget. Ezek feladata a szó szoros értelmében vett irányítás: a PlayerController a játékos karakter mozgását, ugrását, támadását, az EnemyController pedig az ellenfeleket irányítja. A PauseController azt figyeli, illetve állítja, hogy a játék meg van-e állítva.

A PlayerController és a PauseController állapotát az input változtatja minden iterációban (tehát PC esetén a billentyűzet, Androidon pedig a képernyőn megjelenő gombok), a Controllerek pedig a beállított állapot alapján adnak parancsot az általuk irányított entitásnak. Ezzel szemben az EnemyController a magába foglalt egyszerű mesterséges intelligencia szerint irányítja az Enemy osztály példányait.



3.8. ábra A controller osztályok működése és kapcsolatuk az inputtal

Platformtól függően két input osztály felelős az általuk irányított controllerekért: a PCInput PC-n a billentyűzettel, a MobilInputInterface pedig Androidon az érintőképernyőn megjelenő gombokkal és joystickkal vezérlik controllereiket. A PlayerController osztályban a lehetséges állapotok el vannak tárolva, az input ezeket kapcsolja ki-be.

3.4.6. MVC – View

A nézet réteget a különböző Renderer osztályok és az Audio osztály alkotják. Ezek a GameScreen mezőiként jelennek meg a programban. Közülük a legösszetettebb a GameRenderer, a játéktér megjelenítéséért felelős osztály.

Tartalmaz referenciákat a modell azon részeire, amelyek szükségesek a megjelenítéshez – a modell állaptának megfelelően választja ki a GameRenderer, hogy mit kell megjeleníteni. Például ha játékos nem mozog vagy támad, akkor az alap, „idle" animációt játsza le.

A többi Renderer a felhasználói interfész megjelenítéséért felelős.

A keretrendszerben a játék világát egy (2D esetén ortografikus) kamerán keresztül vizsgáljuk. Ezért létrehoztam egy kiegészített kamera osztályt a játéktér részére, ami tartalmaz egy kamerát, és annak a mozgatásához szükséges funkciókat. Így a controller rétegben módosítható a kamera helyzete – követheti a játékost – hogy mindig a pálya megfelelő részét lássa a felhasználó. A külön logika egységet képviselő képernyő elemek külön kamerával működnek – így például a HUD és a különböző menük. Ezek egyszerű, statikus kamerát alkalmaznak.

3.4.7. A koordináta-rendszer és a képarány

Egyszerűbb játékok, vagy olyan programok esetén, amik csak előre adott hardveren, vagy előre ismert (kevés vagy csak egy fajta) megjelenítőn lesznek használva, elegendő lehet a pixeleket használni a játéktér vagy menük koordinátáiként.

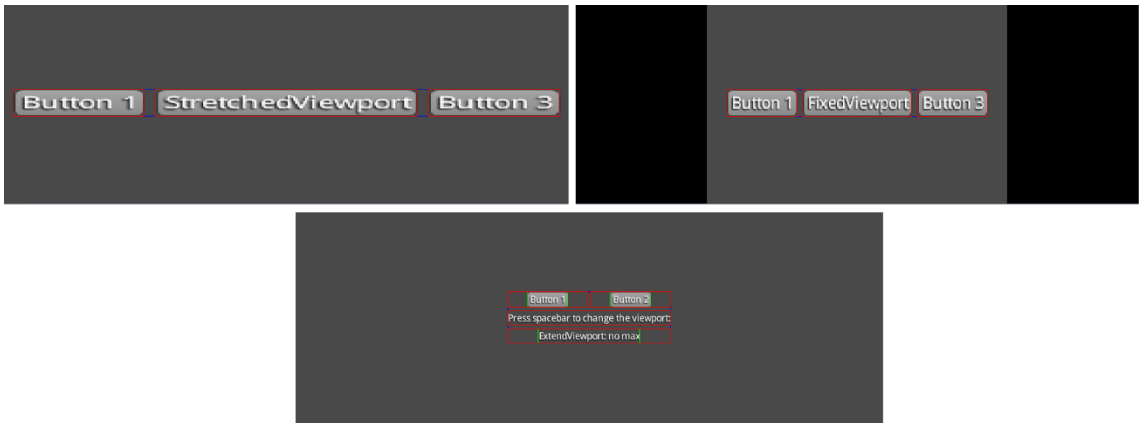
Sőt, ha a fenti feltételek nem teljesülnek, akkor is használható lehet a pixel alapú koordinátarendszer - szükséges esetén skálázással – ha a grafikai elemekhez megfelel a stratégia, például pixel art (olyan grafikai stílus, amelyben a karakterek, tárgyak stb. négyzetekből, azaz „pixelekből" állnak, felelevenítve a korábbi játékok technikai határaiból következő megjelenést) használata esetén, Nearest Neighbour skálázás mellett.

Mivel a dolgozat témájának egyik fenti feltétel sem felel meg, célszerű a játékvilágnak egy új koordinátarendszert definiálni. Így ugyanazok a koordináták használhatók minden platformon, a megjelenítők méretétől és képarányától függetlenül. Ez jó ötlet továbbá azért is, mert a Box2d fizika motor, ami az elveszett kincs mozgásáért felel, szintén saját koordinátarendszert használ: 1 egység = 1 méter.

További probléma a képarány kezelése, amire a modern megjelenítők különbözősége miatt kell odafigyelni. A legegyszerűbb stratégia, hogy figyelmen kívül hagyjuk. Ennek eredményeként a tervezettnél különböző képarányú képernyőkön a kép horizontális vagy vertikális irányban nyúlik, ami (hacsak nem ez volt a cél) nem kellemes látvány a felhasználó számára. Egy másik stratégia szerint, ha különbözik a tervezettől a készülék képaránya, akkor nem nyújtjuk el a képet, hanem kirajzoljuk a középre, így a széleken vagy fekete csíkok jelennek meg, vagy a játéktér egy részét le kell vágni. A legelegánsabb (bár nem mindig megfelelő) megoldás szerint a játéktér kitölti a képernyőt, aránytól függetlenül és nyújtás nélkül, tehát a különböző megjelenítőkön több vagy kevesebb fog látszani a játéktérből. Én a harmadik megoldást választottam.

A játéktér és a képernyő pixeleinek koordinátarendszere közötti egyszerű átváltás a libGDX Viewport API-ja segíti. A Viewport egy alosztályának példányosításakor megadható a világ kívánt mérete, ezután ebben a koordinátarendszerben dolgozhatunk.

A Viewport leszármazott osztályaiból válogatva a fenti három képarány stratégia mindegyikét támogathatjuk a libGDX keretrendszert alkalmazva, ezek rendre: StretchViewport, FitViewport és FillViewPort, ExtendedViewport.



3.9. ábra A három képarány stratégia illusztrációja (4)

Annak megfelelően, hogy a játékteret, vagy a menüt kell megjeleníteni, kétféle stratégiát alkalmaztam.

Játéktér

A játéktérben 1 egység 1 méternek feleljen meg, és 1 csempe legyen 1 egység (azaz méter) széles illetve magas. A koordináták tehát törtek lesznek, az első csempe közepének x koordinátája például 0,5.

A játéktér 4 egység magas, a szélessége pedig a képarány szerint változik.

A képarány változása tehát úgy befolyásolja a játéktér megjelenését, hogy szélesebb vagy szűkebb részletet mutat, míg a magasság mindig 4. Így elkerülhetők a fekete csíkok a képernyő szélein, bár különböző képaránnyal rendelkező eszközökön több, vagy kevesebb fog látszani a pályából. Ez többjátékos vagy kompetitív mód esetén nem eredményezne fair játékmenetet, azonban mivel a játék ilyen funkciókat nem támogat, megfelelő a módszer.

Menük

A menük a scene2d.ui libGDX API segítségével készültek, ami egy html table layouthoz hasonló rendszer alapján rendezi a widgeteket. A helyes működéséhez szükséges, hogy a viewport mérete közel legyen a widgeteket alkotó textúrák méretéhez, mivel az interfész elemek alapértelmezett méretei (amit az elrendezéshez használ a keretrendszer) a textúrák szélessége és magassága. Ennek megfelelően a főmenü ugyan az ExtendedViewport osztályt használja (a független képarány fenntartása miatt), a méretét a felbontásrendszer által megadott adatok alapján állítja be.

3.4.8. A felbontásrendszer

Ugyan a játéktér a képernyőtől független koordináta-rendszerrel rendelkezik, mégis szükséges ismerni kijelző felbontását, amin a játék fut. Egyrészt a megfelelő erőforrások betöltéséhez tudni kell, hogy a támogatott méretek közül melyiket kell betölteni (XL, L, M, S), másrészt a menük pixelközeli koordinátarendszerrel működnek, ami szintén a négy támogatott mérettől függ.

A képernyő méretét tehát a főmenü és az erőforrások betöltése előtt ismerni kell, ezért az elsőként megjelenő Screen, az EssentialLoadingScreen feladata a méret meghatározása és elmentése. Így a későbbiekben a menük lekérdezhetik a méret információt, és az erőforrás-betöltő rendszer is beállítható a megfelelő mérettel.

3.5. Megvalósítás

3.5.1. Az assets csomag

Az assets csomagban az erőforrások (képek, animációk, hangok, pálya, skinek, betűtípusok) betöltéséért és tárolásáért felelős osztályok találhatók.

Assets osztály

3.10. ábra Az Assets osztály UML diagramja

Az Assets osztályban található AssetManager típusú mező, ami a libGDX API egy osztálya, tölti be az erőforrásokat, mindegyiket a hozzájuk tartozó load metódussal (pl. loadPlayerAnimationAssets). A betöltés után az erőforrások a megfelelő get metódussal kérhetők le (pl. getPlayerAnimationAssets). Az erőforrások egy része szinkron, másik része aszinkron módon történik, a hozzájuk kapcsolódó Loader osztály szerint.

Ahhoz, hogy betölthessünk adott típusú erőforrást, regisztrálni kell egy adott Loader osztályt az assetManagerhez, például a Tiled pályák betöltéséhez egy AtlasTmxMapLoader típusú betöltő objektumot. Ez az init publikus metódusban történik, a setLoader metódusok segítségével.

Szintén az init metódusban kerül beállításra a két használt FileHandleResolver interface-t implementáló objektum, ezek feladata, hogy meghatározzák, hol keresse az

assetManager az adott erőforrást. Az internalFileHandleResolver az android modul assets mappájában, a sizeFileHandleResolver pedig a játék mérete alapján meghatározott mappában (szintén az android/assets könyvtárban, pl. xl_) keres.

Több logikailag összetartozó asset egyszerűbb betöltése érdekében létrehoztam két további betöltő metódust: a loadEssentials a töltőanimációt, a loadAll pedig minden mást betölt. Az audio erőforrásokat szétválasztottam zenére és hangokra, ezeket a MusicAssets és SoundAssets típusú mezőkben tároltam. Hasonlóan mezőbe mentem el a pénzérméhez tartozó TextureRegiont, mert a régió lekérdezése az atlaszból költséges művelet, és így csak egyszer kell elvégezni.

A dispose metódust az éppen aktív Screen hívja meg, ezzel felszabadítva az erőforrásokat.

spriter csomag

Itt a Spriter animációk betöltéséhez, tárolásához és lejátszásához használt osztályok találhatók. A LibGdxLoader az animációhoz tartozó képeket tölti be és TextureAtlasba csomagolja. A LibGdxDrawer segítségével rajzolhatók ki az animációk, illetve formák.

Maguk az animációs erőforrások a SpriterAnimationAssets osztályba kerülnek mentésre. Az ScmlReader típusú mező az .scml fájlt olvassa, amit a Data típusú mezőben tárol. A spriteLoader pedig a képeket tölti be és tárolja. Ennek megfelelően rendelkezik loadScml és loadImages nevű metódusokkal, amik a fenti adattagok segítségével végzik a betöltést. Ezt a két metódust a SpriterAnimationAssetsLoader egy példánya hívja meg. Ez egy, az Assets osztály leírásában említett, Loader osztály.

SpriterAnimation

Ez az osztály a már betöltött animációk lejátszására használható. Tartalmaz egy referenciát a betöltött SpriterAnimationAssets típusú erőforrásokra, és egy LibGdxDrawer típusú mezőt azok kirajzolására.

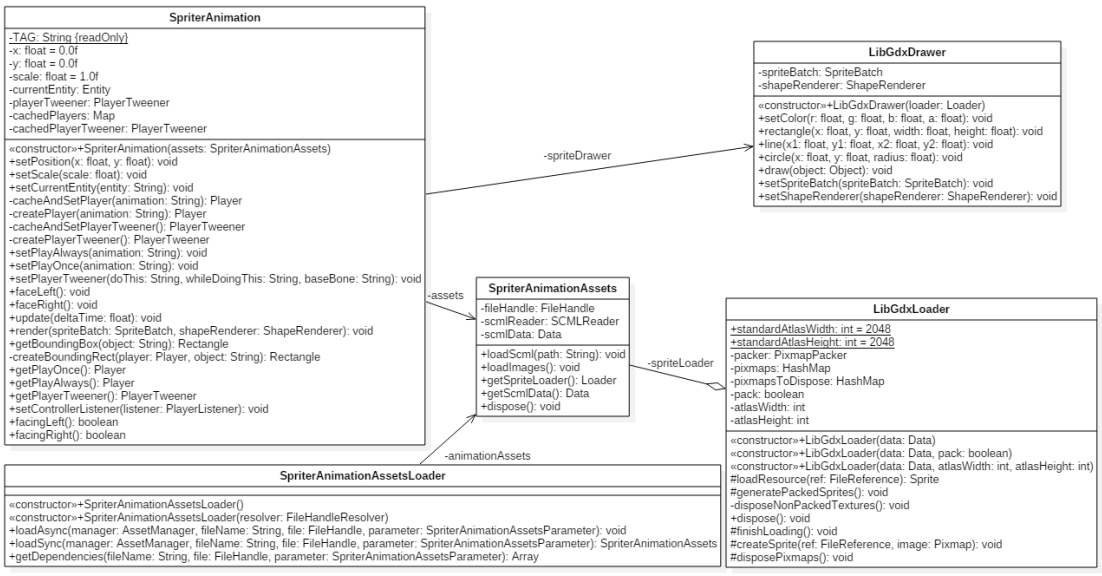
Minden animációnak van pozíciója és skálája, ezeket a megfelelő adattagokban tároltam. A skálára a pixel – világ koordináta átváltás miatt van szükség. Ezeket és a már tárolt Player osztályok értékeit állítják be a setScale és setPosition metódusok.

A helyben definiált Direction nevű felsorolási típusú mezőben azt tároltam, hogy az animáció éppen balra vagy jobbra néz. Ez a faceRight és faceLeft metódusokkal állítható, amik ha szükséges, tükrözik a már eltárolt animációkat is.

A currentEntity az éppen használatban lévő Spriter entitás, ami az animációs programban megadható (például Player).

Az animáció lejátszására három Player típusú mezőt használtam: playOnce, playAlways, és playerTweener. A playerTweener két animáció interpolációjából áll elő, értéke nullra állítódik a lejátszást követően. A playOnce egy egyszer lejátszott animáció, ami a lejátszás után szintén null értéket kap. A playAlways az állandóan lejátszandó animáció – ez kerül lejátszásra, ha a playerTweener és a playOnce null. Az update és a render metódusok ezen stratégia alapján döntik el, hogy melyik animációt frissítsék, illetve játszák le (rendre) a játékciklus adott iterációjában. A három Player mező a három megfelelő setter metódussal állítható be. Ezek először a már eltárolt Playerek („cached" kifejezéssel kezdődő nevű mezők) közül próbálnak egyet betöltetni, ha nem létezik a kívánt objektum , létrehozzák és eltárolják.

A getBoundingBox metódus segítségével a az animáció egyes részeit határoló négyszögek kérdezhetők le, amiket a modell réteg ütközésvizsgálatra használ.



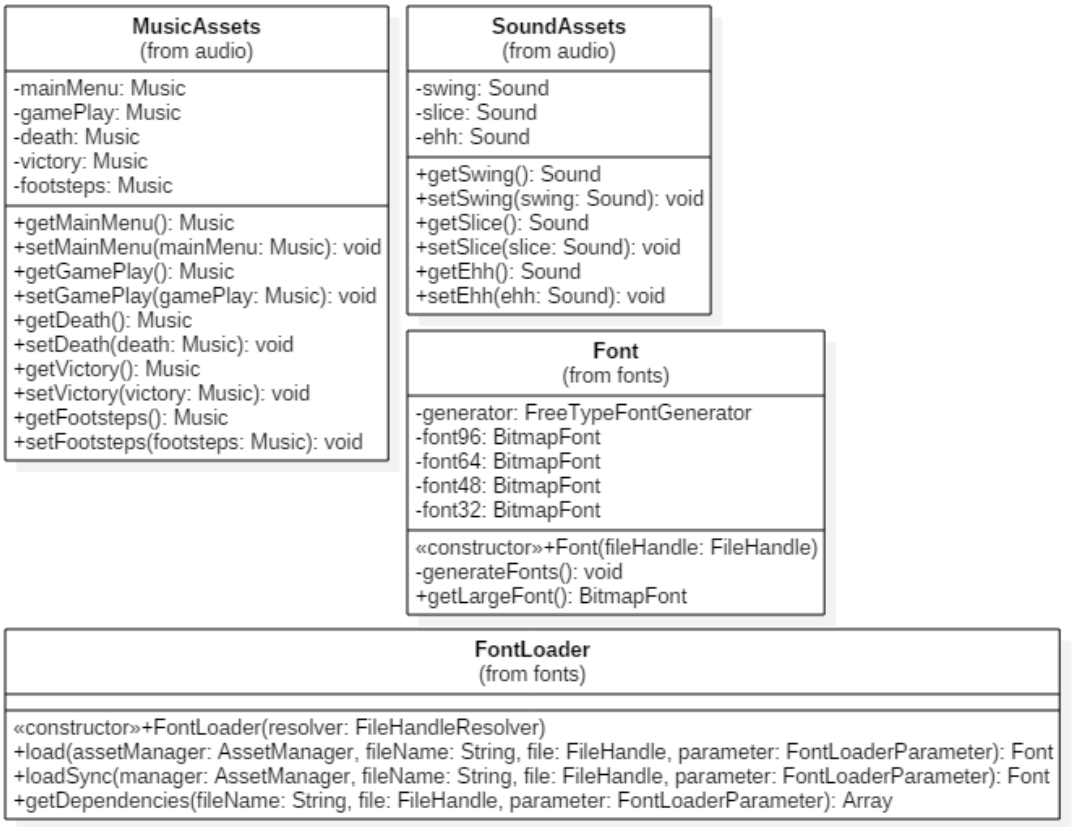
3.11. ábra A spriter csomag tartalma - UML osztálydiagram

entities csomag

Az entities nevű csomagban az erőforrások csoportosítását megkönnyítő, illetve kiegészítő funkciókat tartalmazó osztályokat találunk.

A MusicAssets és a SoundAssets osztályok az audio zenékre és hangokra való csoportosítására használt osztályok. Tárolják a betöltött zenéket és hangokat.

A FontLoader a SpriterAnimationAssetsLoader osztályhoz hasonlóan egy Loader, ami a Font típusú objektumok AssetManagerrel való betöltésére hivatott. A Font

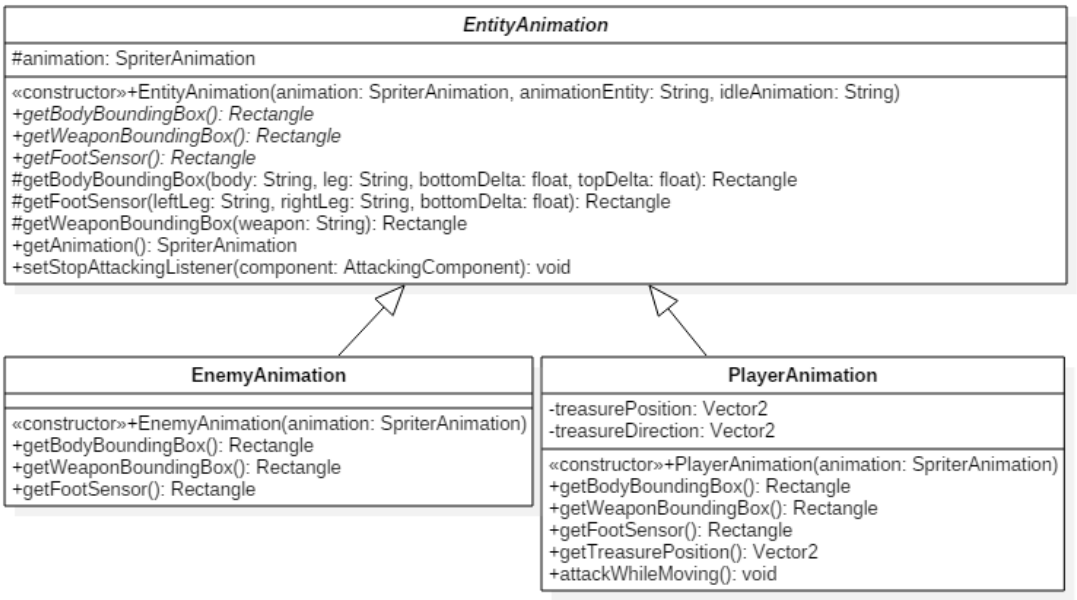


3.12. ábra Az erőforrás entitások UML osztálydiagramja

osztályban a libGDX kiegészítő FreeTypeFontGenerator egy példánya található. Ez felelős a többi mező generálásáért: a betöltött a free type betűtípusokból különböző méretű bitmap betűtípusokat készít. A getLargeFont metódus ezek közül az egyikkel tér vissza, a képernyő méretétől függően, így a menük könnyedén kérhetnek megfelelő méretű, rajzolható betűtípust.

Az EntityAnimation absztrakt osztály a SpriterAnimation kiegészítésére szolgál, attól függően, hogy milyen játék entitás hivatott az adott animáció megjeleníteni. Tartalmazza

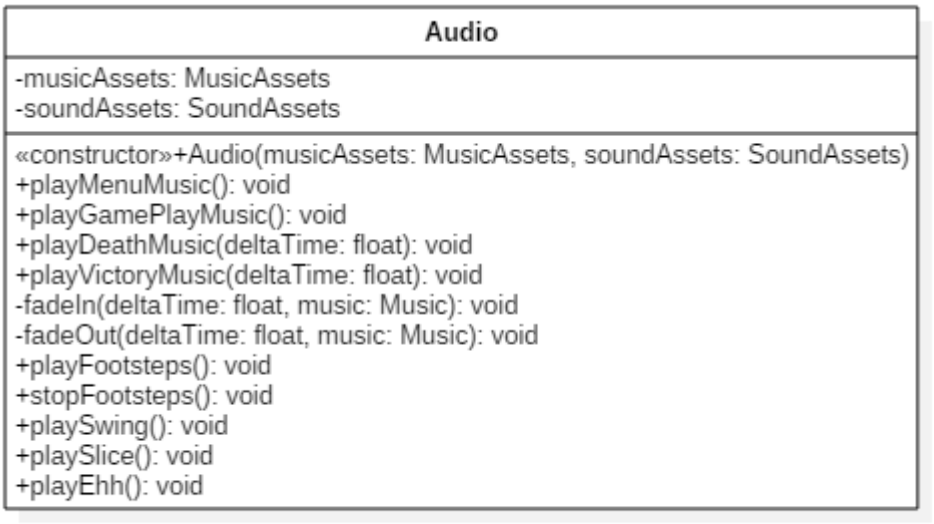
a modell számára szükséges négyszögek lekérdezésére szolgáló metódusokat, mint például a getWeaponBoundingBox, ami a fegyvert körülvevő négyszöget adja vissza. Mindegyikhez tartozik egy paraméter nélküli változat, amit a leszármazott osztályok implementálnak: megadják, hogy az animáció melyik részéből kell a négyszög. Az AttackComponentListener típusú mező feladata a támadás animáció végén való jelzés, a modell ennek a segítségével tudja, hogy mikor érte véget a támadás. A PlayerAnimation és az EnemyAnimation osztályok az EntityAnimation leszármazottai és implementálják a fent említett absztrakt metódusokat.



3.13. ábra Az animációs entitás osztályok UML osztálydiagramja

3.5.2. Az audio csomag

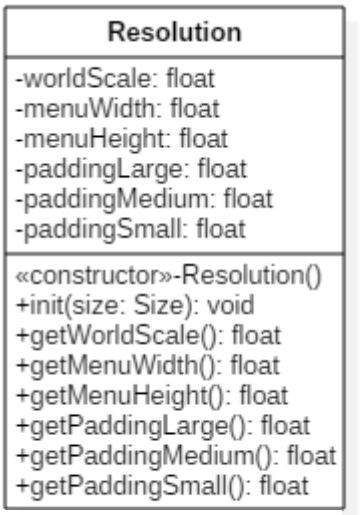
A csomag egyetlen osztályának – Audio – a feladata a hangok és zene lejátszása az adott környezetben. Ezen kívül rendelkezik fadeIn és fadeOut metódusokkal, amik segítik az egyik zenéről a másikra történő váltást.



3.14. ábra Az Audio osztály UML osztálydiagramja

3.5.3. A constants csomag

A constants csomagban különböző String és szám konstansok vannak, amiket a program használ, használati helyétől függően osztályokba csoportosítva. Ilyenek például a különböző erőforrásokhoz tartozó utak, menü feliratok stb.



3.15. ábra A Resolution UML osztálydiagramja

Ezek alól kivétel a Resolution osztály, amiben a képernyő méretétől függő adatokat tároltam: a világ skálája, a menü szélessége és magassága, különböző méretű padding értékek. Ezt az osztályt állítja be az EssentialLoadingScreen, amikor a méretet meghatározza. Singletonként implementáltam – tartalmaz egy instance nevű statikus Resolution változót – így a program bármelyik részében könnyedén lekérdezhetők az adatok.

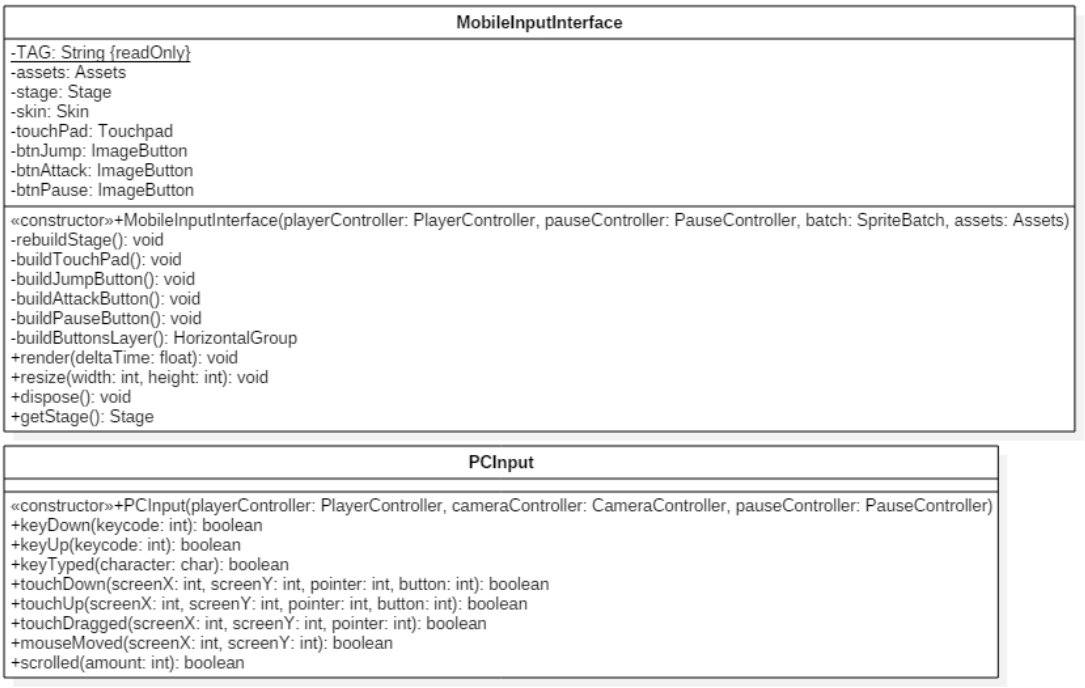
3.5.4. A controller csomag

Az MVC architektúra Controller rétegének képviselőit tartalmazza (kivéve a GameScreent), valamint az inputért felelős két osztály a két platformra.

Input

A PCInput, ahogy a neve is mutatja, az inputért felelős osztály PC platform esetén. Mezői az általa irányított controller objektumok referenciái. Implementálja a libGDX API InputProcessor intefészét, a metódusai ennek megfelelően alakulnak. Ezek közül a keyDown és keyUp metódusokat használtam fel: ezek rendre egy billentyű lenyomásakor és felengedésekor hívódnak meg. Az érintett billentyűtől függően az irányított controllerek valamely metódusát hívja meg, ami az általa tárolt adott állapotot ki-be kapcsolja.

A MobileInputInterface hasonló feladatot lát el, Android platform esetén. Az input kezelésen kívül azonban nézet funkcióval is rendelkezik – hiszen a mobil platform esetén nincs fizikai billentyűzet. A mezői – az általa vezérelt controllerken kívül - a libGDX scene2d.ui API osztályai: a Stage a színpad, ahol a szereplők (jelen esetben UI widgetek) elhelyezkednek. A Skin egy erőforrás, ami a különböző UI elemek stílusát tárolják egy .json fájlban. A Skinek betöltését a fentebb leírt Assets osztály végzi. A konstruktor a privát build metódusok segítségével felépíti a felhasználói felületet, amit a render metódus jelenít meg és frissít a játékciklusban. Az input kezelését a gombokhoz csatolt, helyben létrehozott InputListener implementációk végzik, hasonlóan a PCInput osztályhoz.



3.16. ábra Az inputért felelős osztályok UML diagramja

PlayerController

A játékos irányításáért a PlayerController osztály felel. A benne definiált Commands nevű felsorolási típus a lehetséges állapotokat reprezentálja. Az aktuális állapotot egy Command kulccsal és logikai értékkel rendelkező map modellezi. Így bármelyik parancs aktív lehet egy iterációban. Ezeket az állapotokat a controllerhez kapcsolódó input osztály kapcsolja ki-be a billentyűzet vagy virtuális UI események alapján.

A control metódus a játékosnak parancsokat ad az aktuális állapot alapján. A többi metódust az input használja az állapot beállítására.

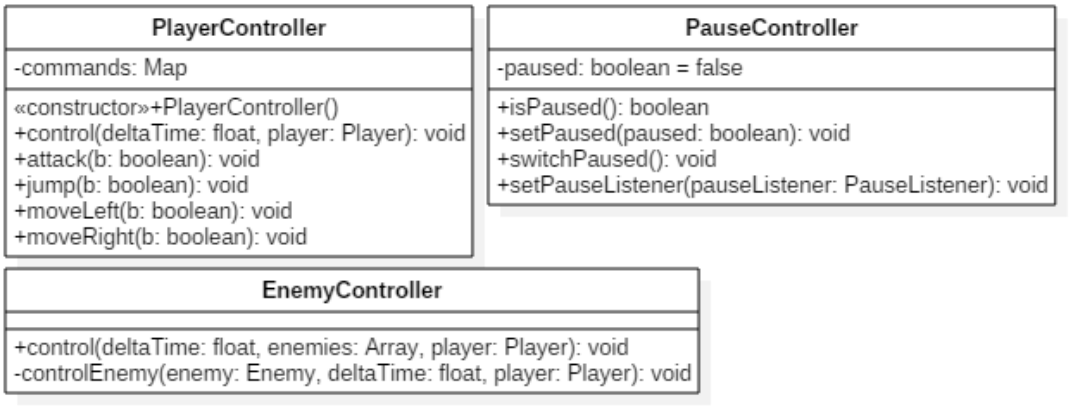
PauseController

Hasonlóan működik a PlayerController osztályhoz, szintén az input vezérli. A pause állapotot tartja nyilván. A hozzá csatlakoztatható PauseListener interfész segítségével lehet figyelni az állapot változását. Erre azért van szükség, mert a „Pause” képernyő megjelenésekor át kell adni az input vezérlését a megjelenő képernyőnek, eltűnésekor pedig vissza kell adni a játéknak. Ezt a GameScreen objektumban, az interfész hozzáadáskor helyben implementált névtelen osztály végzi.

EnemyController

Az ellenség vezérlése egyszerű mesterséges intelligencia szerint történik az EnemyController osztály segítségével. A control metódus paraméterként megkapja a játékost és az aktuális ellenségek tömbjét, majd mindegyiken meghívja a privát controlEnemy metódust. Ez első lépésben ellenőrzi, hogy a játékos életben van-e, azaz null értékű-e az objektum. Ha igen, járőrözik.

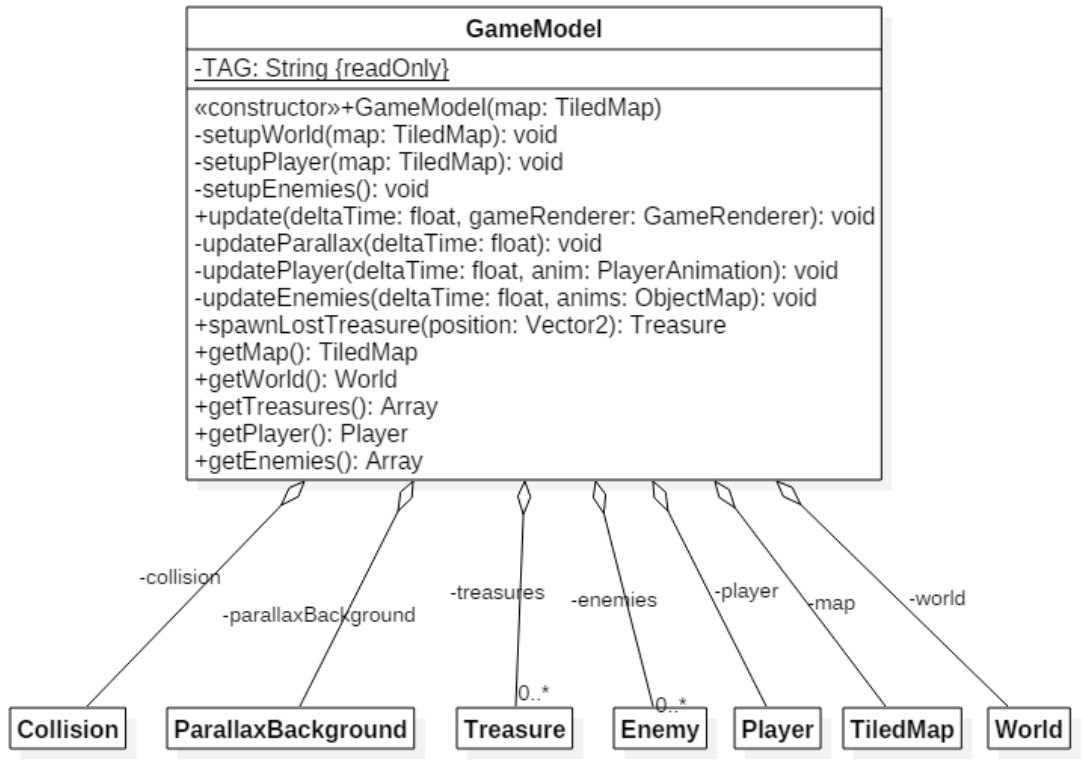
Ha a játékos még életben van, kiszámítja a távolságukat, és hogy egy platform szinten vannak-e. Ha a játékos a megadott rádiuszon belül van, megtámadja, ha ennél távolabb, de a mozgási sugáron belül van, akkor pedig elkezd felé mozogni. Egyéb esetben tovább járőrözik.



3.17. ábra A controller osztályok UML diagramja

3.5.5. A model csomag

Az MVC architektúra Model rétegének képviselőit tartalmazza. A csomag fő osztálya a GameModel, ami a többi modellben használt entitást tömöríti egy helyre. A setup metódusok a tagok beállítását szolgálják, az update metódusban pedig a modell entitások update metódusai kerülnek meghívásra – más szóval itt frissül a modell összes eleme. Az updateParallaxBackground mozgatja a háttérelemeket a parallax stratégia szerint, ha a játékos mozog. Ez a mélység, a 3D illúzióját kelti. Az updatePlayer metódus a játékos frissítésén túl beállítja az ütközéshez szükséges négyszögeit is, amiket az animációtól kap. Hasonlóan jár el az updateEnemies az ellenfelekkel. A spawnLostTreasue segítségével pedig létrehozhatunk egy pénzérme Box2d fizikai testet és az azt enkapszuláló Treasure típusú objektumot.



3.18. ábra A GameModel részletes UML osztálydiagramja

Collision

A Collision osztály a harcrendszer ütközéseinek ellenőrzéséért felel. Számon tartja az aktív ellenfeleket (azaz azokat, akik kellően közel vannak a játékoshoz) és a játékost. Az aktív ellenfelek listája módosítható az addActiveEnemy és removeActiveEnemy metódusokkal, ez a GameScreen feladata. Az update metódusban történik az ütközésellenőrzés. Ez végigiterál az aktív ellenségek listáján, és ellenőrzi, hogy az adott ellenség támad-e és történt-e ütközés a kardja és a játékos között (ezt az animációtól kapott négyszögek segítségével teszi). Azt, hogy az ütés csak egyszer legyen regisztrálva, az ellenfél hit metódusa ellenőrzi. Ezután hasonlóan jár el a játékos oldaláról is.

tiled csomag

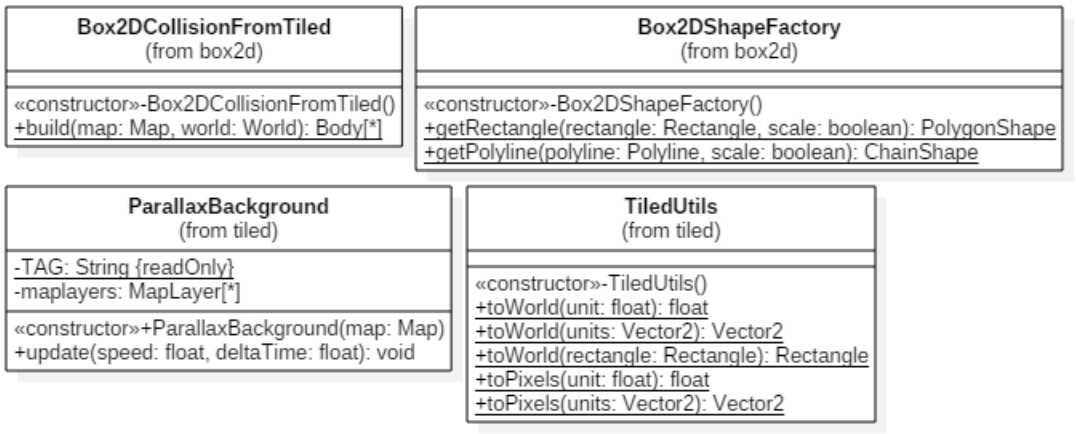
A TiledUtils osztály statikus segítő metódusokat tartalmaz, amivel a világ és pixel koordináták között lehet váltani. Erre azért van szükség, mert a Tiled pályák object rétege (ami a csempéken kívül minden mást tartalmaz) pixel koordinátákat használ.

A ParallaxBackground felelős a háttér parallax mozgásáért. A Tiled pályában három object réteg tartalmazza a három hátteret: a zöld dombokat, a kék hegyeket, és a fehér felhőket. Az osztály ezeket tárolja, és az update metódusban, végigiterálva rajtuk, horizontálisan mozgatja őket. A konstruktor meghívja a resetMap privát metódust, ami visszaállítja a háttérelemek koordinátáit a kezdeti állapotra, így a játékot újra indítva is a megfelelő koordinátákról indul a mozgás.

box2d csomag

A Box2DCollisionFromTiled build statikus metódusa a Tiled pályából létrehozza a Box2d fizika világban a platformokat.

A Box2DShapeFactory statikus segítő metódusokat tartalmaz a Box2d alakzatok létrehozására, amit az előző osztály használ.

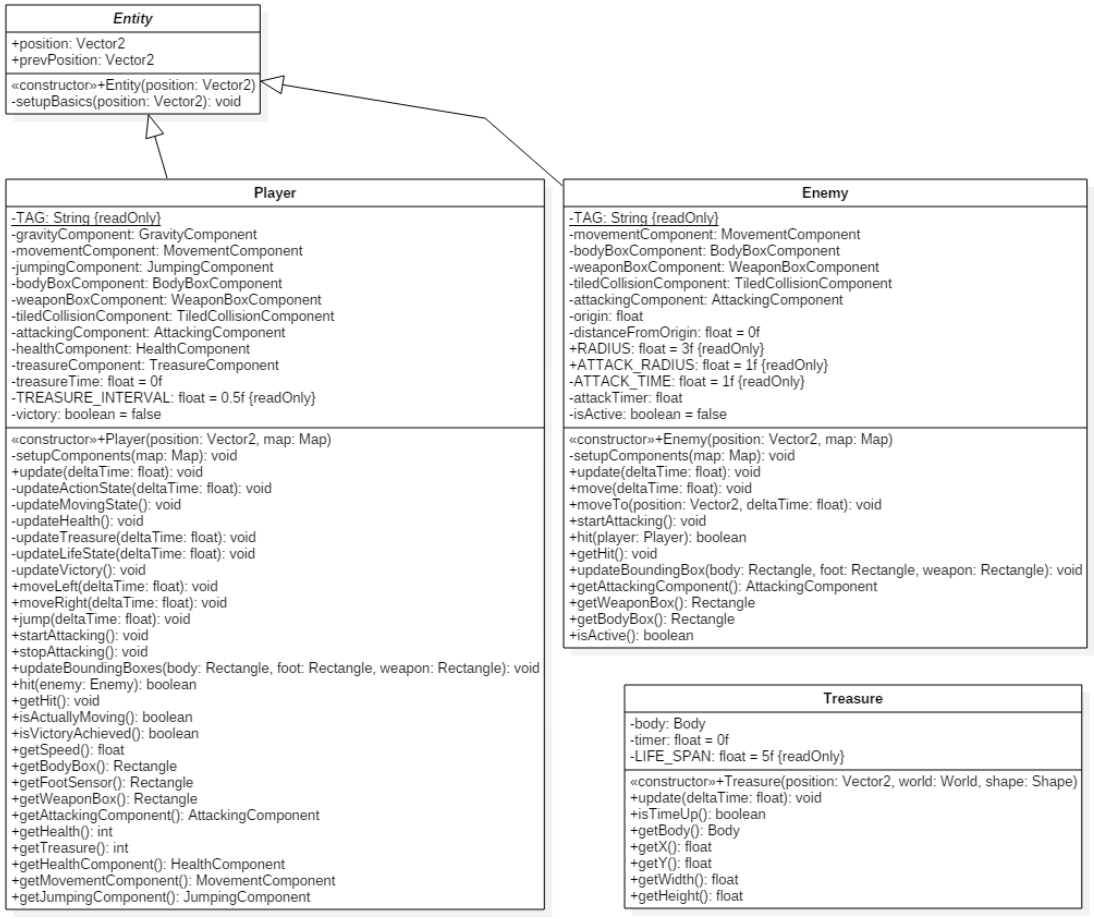


3.19. ábra A modell többi elemének UML osztálydiagramja

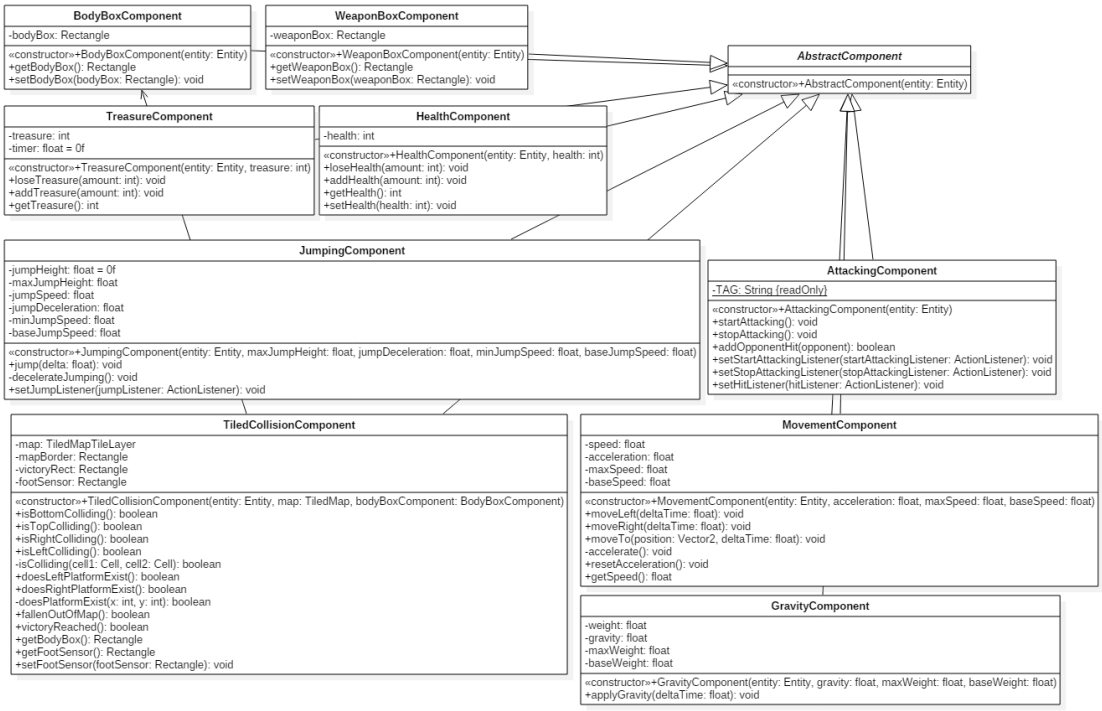
entities csomag

A megvalósítási tervben leírt ECS-szerű rendszer implementációját tartalmazza a csomag. Ennek megfelelően itt található az Entity ősosztály, és a Player, illetve Enemy osztályok. Az ebben a csomagban található ActionListener interface a komponensekhez csatlakoztatható, és különböző eseményekre lehet vele figyelni. Segítségével az egyszer lejátszandó hangok indításának idejét könnyen és pontosan meg lehet határozni.

A Treasure osztály egy kiegészítő osztály, amely magában foglalja a Box2d fizikával rendelkező testet (Body), és számon tartja az érme életartamát, ami az isTimeUp metódussal lekérdezhető.



3.20. ábra Az entitások részletes UML osztálydiagramja



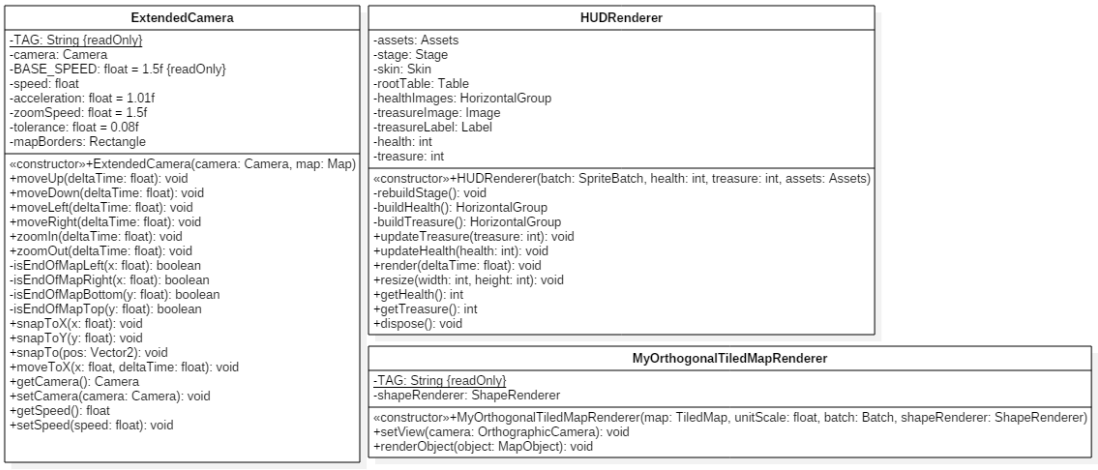
3.21. ábra A komponensek részletes UML osztálydiagramja

3.5.6. A renderers csomag

A renderers csomagban a rajzolással kapcsolatos osztályok találhatók. Ez a csomag képezi továbbá az MVC architektúra View rétegét.

Az ExtendedCamera a mozgással kiegészített kamera osztály. Tartalmaz egy libGDX Camera objektumot, a mozgáshoz szükséges mezőket, mint például a sebesség, és a pálya határait. Utóbbira azért van szükség, hogy biztosítható legyen, hogy a kamera nem hagyja el a pályát, és a pályán kívüli része a világnak nem látszik. Ezt feladatot az „isEndOfMap” kezdetű névvel rendelkező metódusok végzik. A move jellegű metódusok a kamerát a sebessége és gyorsulása szerint mozgatják, míg snapTo verziókkal azonnal az adott pontra lehet helyezni azt. A mozgató metódusokat a GameScreen hívja meg a játékos mozgása alapján.

A MyOrthogonalTiledMapRenderer egy segédosztály, ami leszármazik libGDX OrthogonalTiledMapRenderer osztályából. Azért van rá szükség, mert a Tiled pályák object rétegét nem rajzolja ki, az ezért felelő metódus üres volt, ezért implementáltam a céloknak megfelelően, és így a háttérelemek is kirazjolhatók.



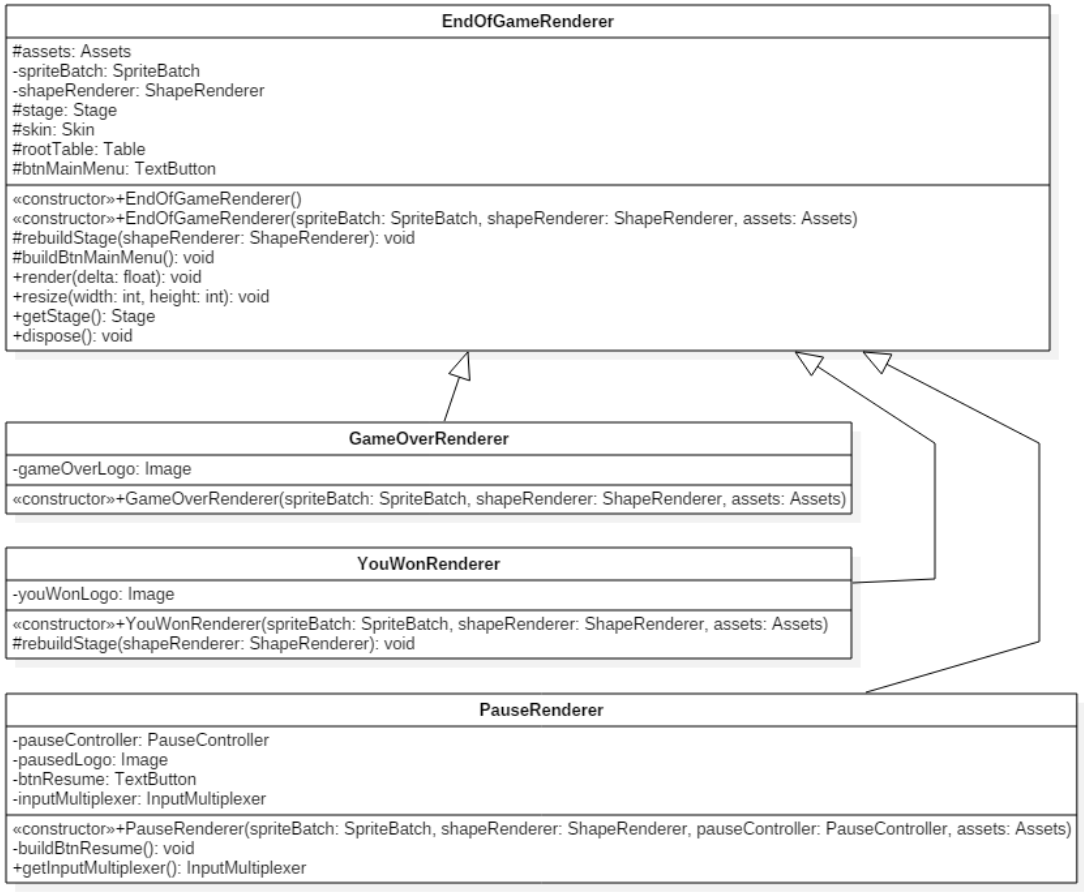
3.22. ábra Nézet réteg beli osztályok UML diagramja

UI rendererek

A HUDRenderer felelős a játékos életének és kincsének megjelenítéséért. A már korábban bemutatott scene2d.ui alapú megjelenítő. Kiegészül az updateHealth és updateTreasure metódusokkal, amelyek segítségével a nézet frissíthető a modellből származó adatokkal.

Az EndOfGameRenderer osztály az őse a játék vége és pause renderer osztályoknak. Egy scene2d.ui felhasználói felület alkotja, így ez is a már korábban látott elemekkel rendelkezik: Stage, Skin stb.

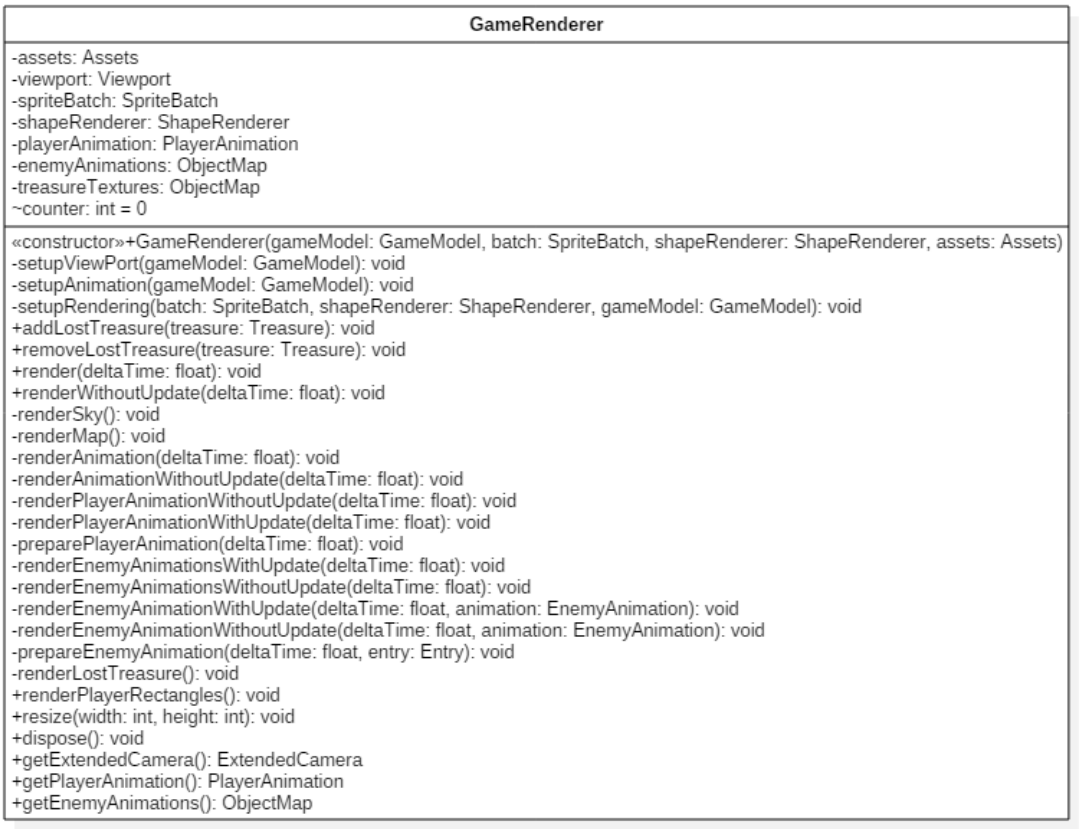
A GameOverRenderer és YouWonRenderer osztályok ezt kiegészítik a megfelelő logókkal. Hasonlóan működik a PauseRenderer osztály is, viszont ez kiegészül még egy gombbal, amivel a főmenübe lehet navigálni.



3.23. ábra Az EndOfGameRenderer típusú menürajzolók UML osztálydiagramja

GameRenderer

A fő render osztály, ami a játékmenet megjelenítéséért felelelős. Tartalmazza a fentebb leírt ExtendedCamera és MyOrthogonalTiledMapRenderer egy-egy példányát. Ezen kívül a PlayerAnimation egy példánya, EnemyAnimaton objektumok egy kollekciója, és kincsekhez tartozó textúrák gyűjteménye található ennek az osztálynak a mezői között. A modell állapotainak lekérdezését egyszerűbbé teszik a játékos, ellenfél és kincs referenciák. A setup metódusok beállítják a viewportot és inicializálják az animációkat. A render metódusok a kirajzolást végzik. Ezeknek két fajtája: razjolás az animációk frissítésével, vagy anélkül. Utóbbira a „Pause” menü miatt van szükség, ahol az animációk legutóbbi állapotát kell kirajzolni, és nem kell frissíteni.



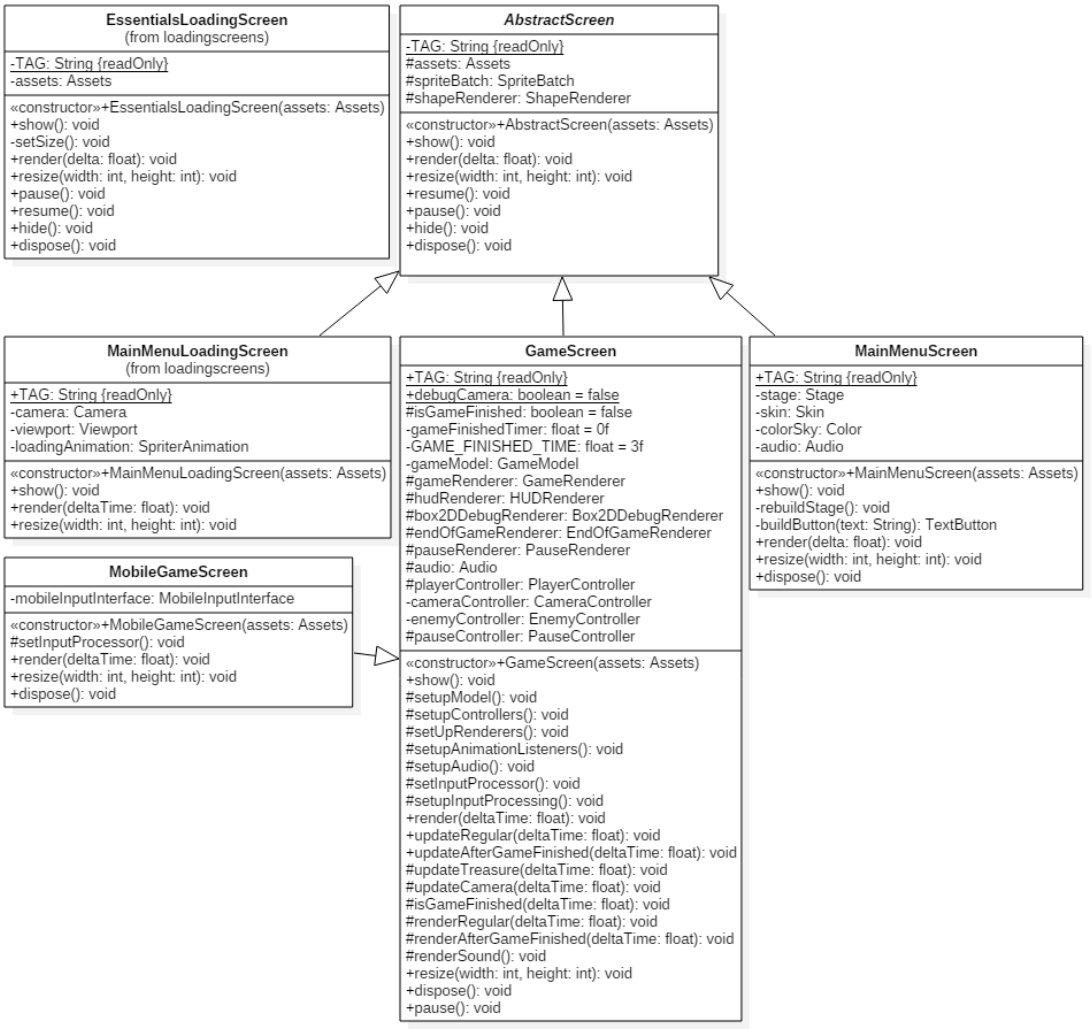
3.24. ábra A GameRenderer osztály részletes UML diagramja

3.5.7. A screens csomag

A Screen tervezési minta osztályai találhatók a csomagban. Az EssentialLoadingScreen betölti a töltéshez szükséges animációt, majd egy MainMenuLoadingScreen példánynak adja át a vezérlést. Ez betölti a maradék erőforrást, és közben lejátsza a töltés animációt. Mivel az erőforrások betöltésének legnagyobb része aszinkron módon történik, miután a show callback metódusban kiadtam a parancsot a töltésre, a render metódusban minden játékciklus iterációban ellenőrizni kell, betöltöttek-e az erőforrások. Amennyiben igen, tovább lehet adni a vezérlést a következő Screen implementációnak, amennyiben nem – a MainMenuLoadingScreen esetén – frissíti és kirajzolja a töltés animációt.

Az AbstractScreen egy absztrakt osztály, amiből az EssentialLoadingScreentől eltekintve az összes többi osztály leszármazik. Tartalmaz referenciát az erőforrásokra, és a rajzoláshoz szükséges mezőket: spriteBatch és shapeRenderer.

A MainMenScreen a korábbi menüknél leírtak szerint működik, hiszen ennek is a scene2d.ui API az alapja.



3.25. ábra A Screen osztályok UML diagramja

3.5.8. GameScreen

A GameScreen az MVC architektúra Controller rétegének egy tagja. Ez a Screen felelős a játéktér és játékmenet különböző részeinek összefogásáért. A különböző update és render metódusaiban a modell, az irányító és a nézet elemeit frissíti és rajzolja.

Mobil platformon az ebből leszármazott MobileGameScreen osztály látja el ugyanezt a feladatot, kiegészülve a mobil irányítás elemével.

3.6. Tesztelés

A játékfejlesztés során megszokott módon a tesztelést már az első funkcionalitás megírásakor elkezdtem, ennek megfelelően a játékprogram részfeladatainak tesztelését implementáció során folyamatosan végeztem. A játékmenet „kipróbálással” és debug

üzenetek kiírásával való tesztelésén kívül, a részben általam, részben a keretrendszer által beépített debug funkciókkal is teszteltem a programot, amik az esetleges hibák megtalálását segítik.

3.6.1. Kihívások

Mivel a játékszoftverek természetüknél fogva összetett programok, és sok különböző funkciójú és jellegű összetevőkből állnak, amik rendszerint egymással is kapcsolatban vannak, az esetleges hibák megtalálása, és a komponensek szétválasztott tesztelése nehézkes. Például, ha egy karakter „beragad", (a földön áll, de nem tud mozogni) és közben az esés animációt játsza le, nehéz megmondani, hogy a hiba a modellből, az ütközés logikából, vagy az animáció komponensből (esetleg mindháromból) származik, és vajon a rossz animáció és a beragadt állapot kapcsolatban van-e egymással.

3.6.2. A rétegek szétválasztása

Az MVC architektúra, a funkciók és rétegek szétválasztása nem csak a program olvashatóságát, bővíthetőségét segíti, de a tesztelést is megkönnyíti, egyszerűbbé téve a fentebb említett kihívásokat. A probléma könnyebb lokalizációja érdekében ugyanis ki- és bekapcsolhatunk komponenseket, vagy módosíthatjuk a viselkedésüket szükség szerint. Így a komponensek közötti kapcsolatokat minimalizálva könnyíthető a tesztelés.

Pédául mozgás, ugrás és esés, illetve az ezekhez kapcsolódó ütközés tesztelésekor kikapcsoltam az animációk közötti váltás funkcióit és az ellenségeket. Így fenyegetés és akadályok nélkül lehetett mozogni, és az animációk változása sem zavarta a tesztelést. Kipróbáltam a mozgást mindkét irányba, akadállyal és anélkül, az ugrást oldalsó és felső akadályokkal és nékül, hasonlóan az esést. Miután meggyőződtem róla, hogy az ütközések és a mozgás megfelelően működnek, egyenként visszakapcsoltam az animációs állapotokat: mozgáskor, ugráskor és eséskor a megfelelő animáció játszódik-e le, a váltás megtörténik-e.

Hasonló példa a támadás tesztelése, először szintén ellenségek és animáció nélkül. Támadni lehet mozgás közben, vagy egy helyben állva, de ugrás közben nem, ennek megfelelően kipróbáltam a támadást helyben, mozogva, a kettő közötti átmenet közben, és ugrás illetve esés közben is. Az állapotok változásait debug üzenetekkel követtem. Hasonlóan a mozgáshoz, fokozatosan kapcsoltam vissza az animációs állapotokat.

Hasonlóan jártam el a többi komponens tesztelésénél.

3.6.3. Debug rajzolók

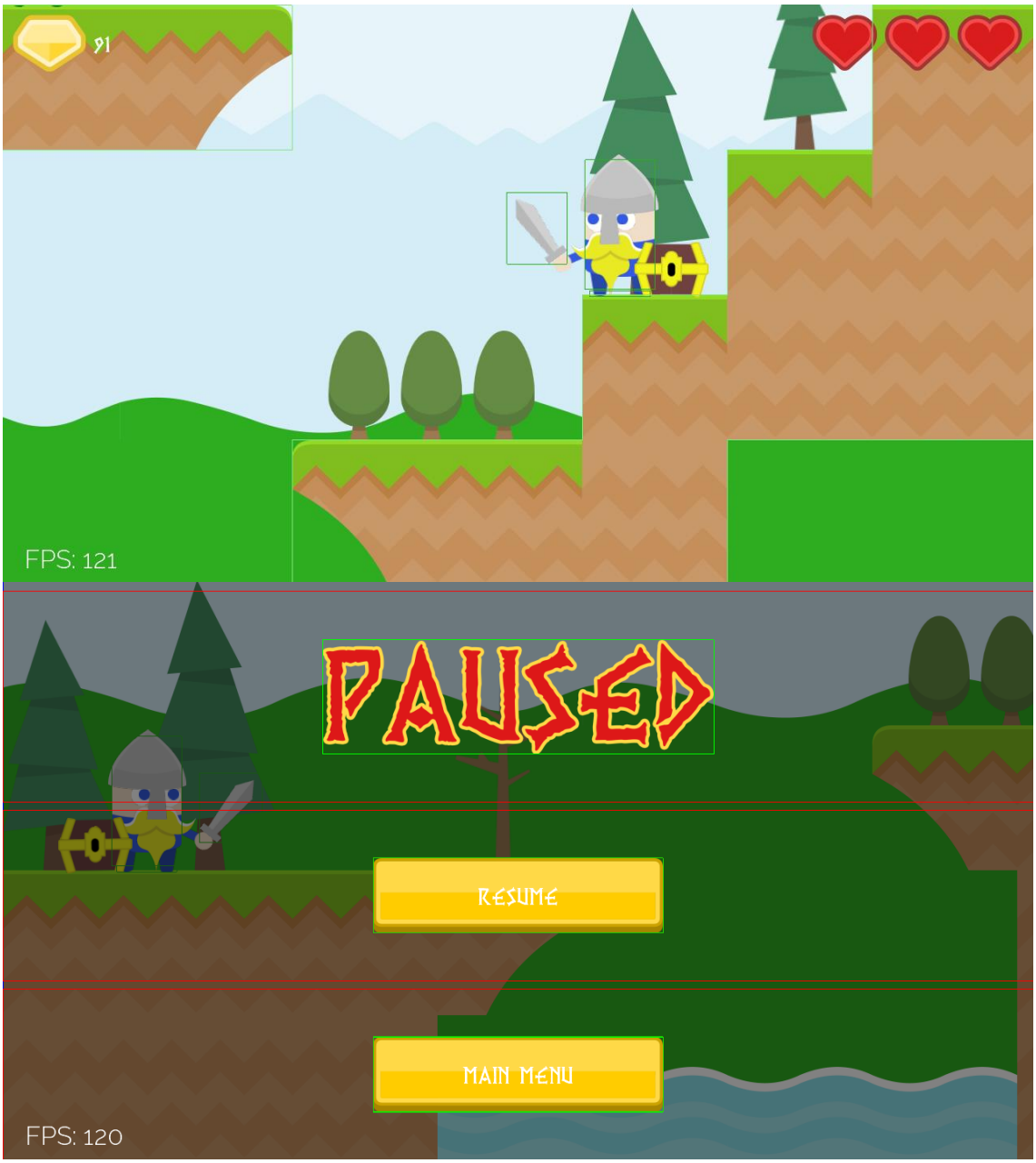
Sok esetben a debug üzenetek és a kirajzolt állapotok nem szolgálnak elegendő információval, hogy megállapíthassuk, honnan ered a hiba. Sőt, gyakran észre sem vehető, hogy hiba történt. Ezért a fejlesztés közben, a tesztelést megkönnyítendő, különböző debug renderer osztályokat használtam – ezek valamilyen többlet információt jelenítenek meg a megszokott szereplőkön kívül.

A libGDX API tartalmaz egy ilyen célra készült osztályt a Box2d objektumok nyomon követésére. A Box2DDebugRenderer osztály példánya kirajzolja a testek és azok alkatrészeinek körvonalát, így könnyen nyomon követhető az elveszett kincsek viselkedése. Azt is láthatjuk továbbá, hogy a pályaszerkesztő programban megfelelően adtuk-e meg az ütközésellenőrzéshez szükséges poligonokat.

A játékos és az ellenségek ütközését a platformokkal, illetve egymás kardjával, négyszögekkel és azok metszésével modelleztem, így az ezekhez használt négyszögeket is kirajzoltam fejlesztés közben.

A felhasználói interfész teszteléséhez a libGDX scene2d.ui API tartalmaz egy egyszerű ki-be kapcsolható funkciót, amivel megjeleníthetők a widgetek és táblák (vagy egyéb container típusok) körvonalai. A UI helyes elrendezésében nagy hasznát vettem ennek az eszköznek, különösen azért, mert a különböző cellák és widgetek négyszögei más-más színnel jellennek meg.

Mivel a játékot több FPS (frames per second, azaz képkocka per másodperc) értékkel is teszteltem, a bal alsó sarokban ezt az értéket is kiírtam. Hasznos ez az információ továbbá a teljesítmény követésére különböző eszközökön való futtatáskor.

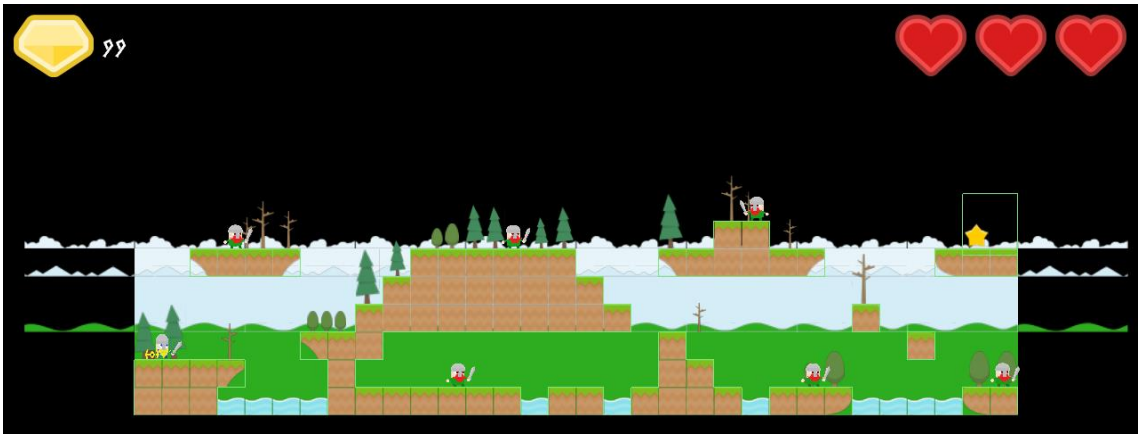


3.26. ábra Debug négyszögek a játékban és a menüben, FPS érték a sarokban

3.6.4. Mozgatható kamera zoommal

Ugyan játék közben is mozog a kamera, hiszen követi a játékost, mindig csak Ragnar közvetlen környezete látható. Ezért fejlesztés közben a billentyűzettel mozgathatóvá tettem a kamerát és zoom funkcióval is elláttam. Ez a funkció megkönnyítette a munkát a pálya betöltésének és rajzolásának tesztelésénél: a zoom segítségével az egész térkép kifér a képernyőre, továbbá az akadályok átugrálása nélkül bárhová hamar el lehet juttatni a kamerát.

Az ellenfelek mozgásának tesztelése így könnyebbé vált, a zoom segítségével egyszerre több ellenség mozgását lehet követni, valamint az egyikről a másikra történő navigáció is gyorsabb.



3.27. ábra A mozgatható kamera tudásának bemutatása: zoom

4. Összefoglalás

Összefoglalásként elmondható, hogy a terveket megvalósító program született, és a kitűzött célokat elértem, a felhasznált módszerek pedig betekintést nyújtanak a modern 2D számítógépes és mobil játékok fejlesztésének folyamataiba. A terveknek megfelelően egy csempékből és dekorációból létrehozott pályából, irányítható játékosból, mesterséges intelligenciával rendelkező ellenfelekből, és különböző menükből álló programot készítettem, és minden platformon élvezhető játékot fejlesztettem. A célok megvalósításához különböző platformok, eszközök, módszerek, tervezési minták vegyes alkalmazására volt szükség, ezek miatt hosszas kutatómunkát is végeztem, hogy a több lehetőség közül ki tudjam választani a legalkalmasabbakat, és amiatt is, hogy ezeket megfelelően beépíthessem a programba.

Kitértem a játékokfejlesztés során használt erőforrások előkészítésére és csomagolására, különös tekintettel a különböző tulajdonságú modern számítógépek és mobil eszközök szempontjából. A tervezett négy méret: XL, L, M és S jól működnek a különböző méretű kijelzőkön.

Mint a cross-platform játékfejlesztés egyik alappillére, külön figyelmet érdemelt az irányítás. A számítógépek és okos eszközök (okostelefonok és táblagépek) nem csak a kijelző és a rendelkezésre álló erőforrások tekintetében különböznek, hanem a használható input eszközök tekintetében is. Míg PC-k esetén feltételezhető a billentyűzet és egér megléte, a mobil eszközök leggyakrabban csak érintőképernyővel rendelkeznek.

Ezt a feladatot mobil platformon virtuális gombokkal és joystickkal sikerült megoldani, míg számítógépen a hagyományos input eszközöket használtam.

Az architektúra és tervezési minták szempontjából kitűzött célokat is teljesíti az elkészült szoftver: a játékmenet egy MVC-szerű architektúra alkalmazásával készült, és a felhasználtam a játékfejlesztés során leggyakrabban használt tervezési mintákat, mint például a screen és ECS (entity-component-system) design patternek.

4.1. Továbbfejlesztési lehetőségek

A program elsősorban a - gyakran alkalmazott - (platformfüggetlen) játékfejlesztési módszerek bemutatásának céljából készült, ezért több lehetőség is van a

továbbfejlesztésére, aminek következtében egy hosszabb, érdekesebb játékmenettel rendelkező és többször újra játszható játékprogrammá fejlődhet.

Egyik lehetőség, hogy a játék ne csak egy pályát támogasson. Legyen a főmenüben egy pályaválasztó menü, ahol a már elért pályák újra játszhatók, az újakat pedig az előttük lévők végig játszásával nyithatjuk meg. Okot adhatna az újra játszásra, hogy a teljesítményt a program a szerzett kincs alapján egy három-, esetleg ötcsillagos értékeléssel méri. Változatosságként pedig különböző környezetben játszódó pályákat is be lehet vezetni, például éjszakai háttér, vagy téli pálya jeges-havas platformokkal.

A játékosok versengésének elősegítése érdekében be lehetne vezetni egy „high score” rendszert, azaz a játék végén a program elmenti a szerzett (elveszített) kincsből származó pontszámot egy szerveren. Így a játékosok láthatják, milyen a teljesítményük a többiekhez képest.

A változatosság érdekében többfajta ellenfél is bevezethető: nem csak külalakban, de viselkedés szempontjából is. A játékban jelenleg implementált ellenfelek a komponens rendszer segítségével könnyen bővíthetők.

4.2. Források

A dolgozatban szereplő megoldásokhoz a Learning Libgdx Game Development [4] és a Game Programming Patterns [5] könyvek adtak inspirációt. Az ötletek a libGDX keretrendszerrel történő megvalósításában a libGDX wiki [6] és dokumentáció [7] nyújtottak segítséget.

5. Irodalomjegyzék

[1] Trixt0r, „Trixt0r/spriter: A Generic Java importer for Spriter animation files.,” [Online]. Available: https://github.com/Trixt0r/spriter. [Hozzáférés dátuma: 29. 06. 2016].

[2] libGDX, „libGDX,” [Online]. Available: https://libgdx.badlogicgames.com/download.html. [Hozzáférés dátuma: 28. 06. 2016].

[3] B. Games, „gamedev.net,” 02. 04. 2013. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013. [Hozzáférés dátuma: 24. 08. 2016].

[4] A. Oehlke, Learning Libgdx Game Development, Packt Publishing, 2013.

[5] R. Nystrom, Game Programming Patterns, Genever Benning, 2014.

[6] tyrondis, „GitHub libGDX wiki,” [Online]. Available: https://github.com/libgdx/libgdx/wiki. [Hozzáférés dátuma: 2016].

[7] libGDX, „libGDX Documentation,” [Online]. Available: https://libgdx.badlogicgames.com/nightlies/docs/api/. [Hozzáférés dátuma: 2016].

[8] Maxim, „GitHub libGDX Wiki,” [Online]. Available: https://github.com/libgdx/libgdx/wiki/Viewports. [Hozzáférés dátuma: 01. 10. 2016].