Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Bozó Bálint Vid

RPG JÁTÉK FEJLESZTÉSE ANDROID PLATFORMRA LIBGDX KERETRENDSZER SEGÍTSÉGÉVEL

KONZULENS

Sik Dávid

BUDAPEST, 2022

Tartalomjegyzék

1 Bevezetés.....7
1.1 Az okostelefon és a szórakozás.....7
1.2 Műfaj és cél.....8
1.3 Felépítés.....8
2 Irodalomkutatás és technológiák.....9
2.1 Az RPG alapjai.....9
2.2 Alap és hibridműfajok.....9
2.2.1 Klasszikus rpg.....9
2.2.2 Akció RPG.....10
2.2.3 Taktikai RPG.....11
2.2.4 Egyéb.....12
2.3 Piac.....12
2.3.1 Az RPG eredményei számokban.....13
2.3.2 Játékosok véleménye.....14
2.3.3 Sikerjátékok közös elemei.....15
2.4 Összegzés.....16
3 Felhasznált technológiák.....17
3.1 LibGDX.....17
3.1.1 Felépítése.....17
3.1.2 Bővítmények.....18
3.1.3 Életciklus metódusok.....20
3.1.4 Kirajzolás.....21
3.2 Tiled térképkészítő.....22
3.3 Pinta.....23
3.4 GDX Texture Packer.....23
4 Tervezés.....24
4.1 Feladat bemutatása.....24
4.1.1 Műfaji sajátosságok.....24
4.1.2 Népszerű játékelemek.....25
4.1.3 LibGDX lehetőségei.....25

4.1.4 Játékmenet....26
4.2 Architektúra....29
4.2.1 MVC....30
4.2.2 Tervek....30
4.2.3 Egyéb tervezési minták....31
5 A megvalósítás....33
5.1 *Game* és alapok....33
5.2 *MenuScreen* és a kirajzolható osztályok....34
5.2.1 A képernyőn megjelenő elemek....35
5.2.2 A kamera és képernyőfelbontás....37
5.3 CharacterScreen....37
5.4 GameScreen....38
5.4.1 GameHud és bemenetkezelés....38
5.4.2 World és világépítés....39
5.4.3 World működése....39
5.4.4 MapContainer és térképkészítés....41
5.4.5 Entitás menedzser és animációk....43
5.4.6 Entitások és mozgásuk....45
5.4.7 Karakterek, ellenségek és harcrendszer....49
5.5 UpgradeScreen....50
6 Önálló munka értékelése és továbbfejlesztési lehetőségek....51
7 Irodalomjegyzék:....53

HALLGATÓI NYILATKOZAT

Alulírott **Bozó Bálint Vid**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 05. 20.

...…….
Bozó Bálint Vid

Összefoglaló

Napjainkban a mobiltelefon átvette a videójáték-piac vezetését a többi platformmal szemben. A legnagyobb bevételű műfajok között a szerepjátékok dominálnak. Megfogalmazódhat a kérdés, hogy ez miért van így, ezzel egyetemben felmerül az igény arra, hogy hogyan lehetne egy hasonlóan népszerű szerepjátékot fejleszteni.

A szakdolgozat végigveszi, hogy mik ennek a műfajnak a jellemzői, feltérképezi azt, hogy a játékosok mit gondolnak róla és miért szeretik azt, majd megvizsgálja a legnagyobb címek közös tulajdonságait. Mindemellett megismertet egy népszerű telefonos platformot támogató játékfejlesztő keretrendszert, a libGDX-et.

A mindezekből leszűrt tanulságok alapján megtervezésre kerül egy kétdimenziós, izometrikus megjelenésű egyszerűbb akció szerepjáték, melynek a könnyen bővíthetőség a fő alapelve, illetve nagy hangsúly kerül benne a megjelenítést kezelő objektumok hatékony kiszervezése. Minden elemének a működése részletesen le lesz írva.

Zárásként egy összefoglaló következik, mely leírja az elért eredményeket, illetve fejlesztési lehetőségeket.

Abstract

Nowadays, the mobile phone has taken the lead in the video game market over the other platforms. Role-playing games dominate among the highest-income genres. The question arises as to why this is the case, as well as the need to develop a similarly popular role-playing game.

The dissertation goes through the characteristics of this genre, maps out what players think of it and why they love it, and then examines the commonalities of the biggest titles. In addition, it introduces a game development framework that supports a popular telephone platform, libGDX.

Based on the lessons learned from all this, a simpler action role-playing game with a two-dimensional, isometric appearance is designed, the main principle of which is easy to expand, and a great emphasis is placed on the efficient outsourcing of the objects that handle the display. The operation of each of its elements will be described in detail.

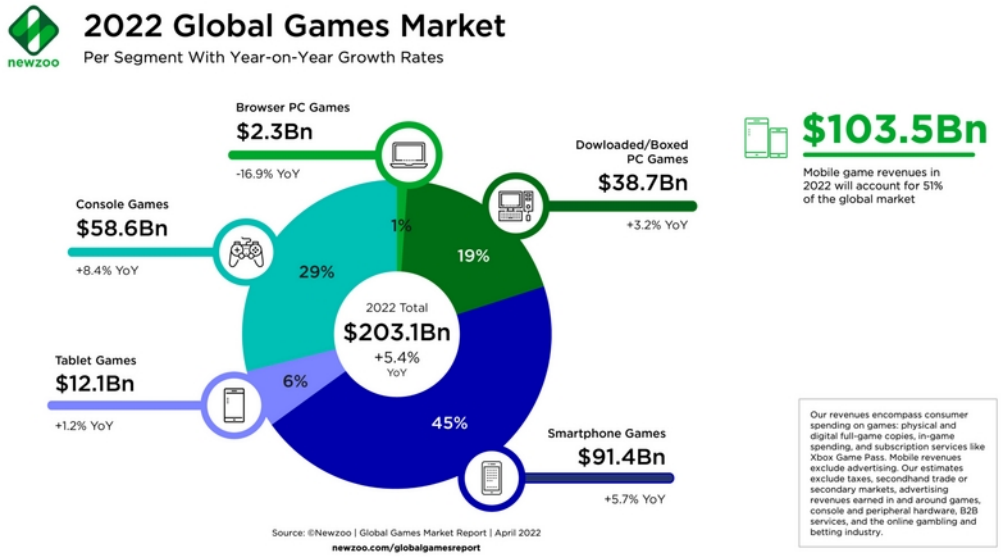
In conclusion, a summary is provided describing the results achieved and the opportunities for improvement.

1 Bevezetés

1.1 Az okostelefon és a szórakozás

Az okostelefonok manapság már mindenki zsebében ott lapulnak, a világbank adatai szerint 2020-ban 100 főre átlagosan 106 mobiltelefon előfizetés jut [1]. Az eredetileg csak telefonálásra kifejlesztett eszköz profilja jelentősen átformálódott az elmúlt 15 év alatt, köszönhetően a nagy ütemű technológiai fejlődésnek. Eredeti funkciója mellett szabadidőnk eltöltése helyének egyik legjelentősebb platformjává vált, az Egyesült Államokban átlagosan több mint négy órát töltenek el naponta a használatával [2], ami jelentősen több, mint amennyit asztali számítógép előtt ülünk. Ezt az átalakulást lekövette mind a közösségi média, mind a szórakoztatóipar, utóbbiban is az egyik legnagyobb nyertes, a videójáték-ipar.

A telefonok terjedésével a telefonos játékok egyre hatalmasabb piacot tudhattak magukénak. A különböző műfajok minden típusú szórakozási igényt ki tudnak elégíteni, kezdve az egyszerű utazás alatti unaloműző alkalmazásoktól kezdve, a családtagokkal és barátokkal játszható társasokon át, az asztali számítógépeken népszerű komplexebb szerepjátékokig is. Sokszínűségének és elterjedtségének köszönhetően rövid idő alatt a legjövedelmezőbb szegmensévé vált a videójáték-iparnak.



1. Ábra: Videójátékok platformokra lebontott bevételeinek eloszlása és értéke. A mobiljátékoknak otthont adó eszközök 51%-át teszik ki a teljes piacnak [3]

1.2 Műfaj és cél

A játékok közötti egyik legnagyobb bevételt elkönyvelő - egyben a gyerekkoromnak is kedvenc műfaja - a régebbi platformokon is népszerű szerepjátékok (Role Playing Games, röviden RPG), melyek a telefonos játékok piaci bevételeinek 21%-át teszik ki [3]. A lényege röviden, hogy egy fiktív karakter bőrébe bújva küzdhetjük le az elénk álló nehézségeket, melyek miben léte egészen vegyes skálán mozog, hisz lehet akár egyszerű logikai feladatok megoldásától kezdve, akció alapú harcon át, stratégiai tervezést igénylő körökre osztott küzdelem is. Másik fontos eleme emellett a fejlődési rendszer, mely megadja a lehetőséget, hogy a teljesített feladatok haladtával erősebb legyen mind a játékos, mind az előtte álló kihívás keménysége is.

A célom a szakdolgozattal, hogy megvizsgáljam, hogy miben is különbözik RPG műfaj a többitől, és milyen tulajdonságok teszik a piacon sikeressé. Utána mindezen tényezők figyelembevételével egy olyan RPG játékot tervezzek és készítsek, ami az alapvető játékmechanikára fókuszál, hogy azt később már elég legyen csak tartalommal megtölteni. Cél emellett alaposan megismerni, és használni az alacsonyabb szinten programozható multiplatform játékfejlesztő keretrendszert, a libGDX-t.

1.3 Felépítés

Az első fejezetben lesz megtekintve az RPG specifikus tulajdonságai, honnan alakult ki, milyen játékelemek különböztetik meg a többi műfajtól, illetve milyen alfajai léteznek. Utána a piac lesz megvizsgálva, hogy mi teszi a műfajt népszerűvé a telefonos platformon, emellett mit preferálnak maguk a játékosok benne, illetve mik azok a jellemzők, amik kiemelik a legsikeresebb címeket. A felhasznált technológiákon belül lesz kifejtve a libGDX keretrendszerben rejlő lehetőségek összegzése és annak legfontosabb mechanikái. Ezt követően az egyéb, fejlesztés során felhasznált technológiákról lesz leírás olvasható. A tervezésnél az irodalomkutatásból leszűrt tanulságok alapján meg lesz határozva, hogy milyen feltételeket kell teljesítenie a készülő játéknak. Közvetlen utána érkezik a felsőszintű architektúra megtervezésének folyamata, közben bemutatva az azt befolyásoló elveket. Mindezek után a program részletekbe menő leírása következik ábrák és kódrészletek segítségével, a tervezetben leírt felosztás alapján. A dolgozat legvégén egy rövid összegzés található, az elért eredményekről és a fejlesztési lehetőségekről.

2 Irodalomkutatás és technológiák

2.1 Az RPG alapjai

Az RPG definíció szerint olyan videójáték műfaj, ahol a játékos egy fiktív karakter irányításán keresztül vállalhat és teljesíthet küldetéseket egy kitalált világban. Ezen felüli általánosságban a leírása viszont nem egy kizárólagos módon egyértelmű eredményt hozó feladat, mivel az elmúlt idők során különösen szerteágazóvá vált működése [4].

A tradicionális szerep-videójátékok öt alapvető elemen osztoznak:

1. Lehetőség a játékbeli karakter tulajdonságai erősségének vagy a szintjének a növelésére, amelyek befolyásolni tudják játékmenetet.
2. Tartalmaz menüalapú harcrendszert, több különböző képességgel és varázslattal, valamint leltárrendszert, hordható felszerelésekkel (például páncél vagy fegyver).
3. Főküldetéssel rendelkezik, amely története mentén halad a játékos, emellett van lehetősége további mellékküldetéseket teljesíteni.
4. A környezettel interakcióba lehet lépni eredeti játékmeneten felüli képességekkel (például csapda hatástalanítás vagy zártörés).
5. Több különböző karakterosztály létezése, melyek meghatározzák az adott karakter tulajdonságait és képességeit (például varázsló, harcos, tolvaj).

A modern és hibrid RPG-k manapság nem feltétlenül tartalmazzák mindegyik pontot, általában egy-kettőt használnak fel és kombinálnak más műfajok tulajdonságaival.

2.2 Alap és hibridműfajok

2.2.1 Klasszikus RPG

A ma ismert RPG videójátékok az asztali szerepjátékokból alakultak ki, mint a klasszikus Dungeons&Dragons. Alapvető lényege az volt, hogy a játékosok egy-egy fiktív karaktert személyesítenek meg, szóban megfogalmazva annak a cselekedeteit és gondolatait.

A számítógépen futó formájukban lehetővé vált a játékmester (aki narrátora és bírája is volt egyben a szóbeli játéknak) automatizálása, hogy programozott logika döntsön a nem játékos által irányított szereplők (Non-Player Characters, későbbiekben röviden NPC) és ellenfelek akcióiról.

A legtöbb RPG egy kitalált világban játszódik, hagyományos fantasy vagy sci-fi elemekkel, amelyek beépülnek a játék mechanikájába. Például egy faj, a törpe, ember vagy az elf kiválasztása befolyásolhatja a karakter játékbeli döntéseit, vagy módosíthatja tulajdonságait vagy varázslatos képességeit.

A videójáték formátum a papír-toll játékokhoz hasonlóan indultak, de a kockákat felváltották a képernyő mögött végrehajtott automatizált dobások. A harc ellenfelekkel interaktív menük segítségével lett lebonyolítva, és történhetett körökre osztott vagy valós idejű módon. A játékos irányíthat egyszerre több karaktert is, mivel azok felváltva, egymás után cselekszenek, az ellenséggel egyetemben. A harcon kívül játéktérben a mozgás általában kötött, nem ad nagy szabadságot.

A modern játékok azonban számos hibrid változatot vezettek be, amelyek jelentősen kibővítették a műfajt és annak lehetőségeit.

2.2.2 Akció RPG

Az Akció RPG (ARPG) valós idejű harcra helyezi a hangsúlyt, ahol a játékos közvetlenül irányíthatja a karakterét, szemben az eredeti kör- vagy menüalapú harccal, miközben továbbra is a karakter fejleszthető statisztikai tulajdonságait használja fel (például erő, ügyesség, varázserő stb.), amely meghatározza a relatív erejét és képességeit. Megszűnik az egyszerre több karakter irányításának lehetősége, viszont sokkal könnyebben le tudja kötni a játékost folytonos harc és mozgás. Utóbbi kötetlenné vált, tetszőleges irányba tud mozogni a felhasználó. Általánosságban maga a harcrendszer van fókuszban, a történeti narratíva mellékes szokott lenni.

A technikai újítások miatt ezek fejlesztése egy fokkal nehezebb, mint régebbi társáé. Míg elődjében felváltva történtek az események, itt minden egyszerre zajlik, ami nagyobb mennyiségű erőforrást igényel, alacsonyabb szintű programozói keretrendszerekkel nagyobb matematikai tudásra lehet szükség.

A legnépszerűbb képviselője a műfajnak a Diablo széria játékai, melynek minden része a korának legsikeresebb címe volt.

2. Ábra: Az izommetrikus megjelenésű Diablo 2 ARPG felülete

2.2.3 Taktikai RPG

Azokra a címekre vonatkozik, amelyek stratégiai videójátékokból emelik át a harcrendszert. A játékos egy véges létszámú csapatot irányít, hasonló mennyiségű ellenséggel szemben. Az összecsapás általában egy négyzetrács (vagy hatszögrács) alapú játéktérben zajlik, amelyben a sakkhoz hasonlóan felváltva lépnek a felek, és a pozícióbeli előnyt kihasználva küzdenek. Szerepjáték elemük, hogy az összecsapások között fejleszthetőek az egységek, amivel gyakran többet számíthat a felkészültség, mint maga a harcban nyújtott teljesítmény.

Gyengesége a többi műfajjal szemben, hogy döntő többségében csak egyjátékos módot tesz lehetővé, illetve hiányzik belőle a felfedezés élménye.

Egyik legnagyobb klasszikusa a Heroes of Might and Magic széria, melyhez több hasonló appot találhatunk már telefonra is.

2.2.4 Egyéb

Még rengeteg hibrid műfaj létezik, ezek közül csak a telefonokon is legnépszerűbbeket érdemes megnézni.

A Massively Multiplayer Online Role-Playing Game (röviden MMORPG*)*, olyan videójátékokat foglal magába, amely egy állandó állapotú világban játszódik, és több ezer vagy akár millió felhasználó játszik és fejleszti karakterét szerepjátékos környezetben. A közös világ soha nem statikus, még akkor is ha egy ember ki van jelentkezve, olyan események történnek benne, amelyek hatással lehetnek rá, amikor újra bejelentkezik. Hangsúlyos eleme a játékosok közötti interakció. Legnépszerűbb képviselője androidon a LineAge M, amely 2021-ig 3,5 milliárd dollár (több mint 1285 milliárd forint) bevételt termelt [5].

A Sandbox RPG a játszható világ szinte teljes kontrollját biztosítja a felhasználónak, mint a környezet szinte korlátok nélküli formálása, építése. Legnépszerűbb címe a Google Play sikerlistáján is első Minecraft. [6]

A Roguelike játékokban véletlenszerűen generált pályákon kell végigküzdenie magát a játékosnak, úgy hogy karakterének mindössze egy élete van. Ezzel egy sokkal feszültebb és izgalmas játékélmény szerezhető. Telefonon a Pixel Dungeon számít klasszikusnak.

A Puzzle RPG-ben a harcot logikai játékok váltják fel, telefonon leggyakrabban a „csempe illesztő” kirakós feladványok jellemzőek (az ugyanolyan színű csempéket kell egymás mellé juttatni), és ezek megoldásával lehet fejlődni és továbbhaladni. Egyik legnépszerűbb példa rá a Puzzle&Dragons.

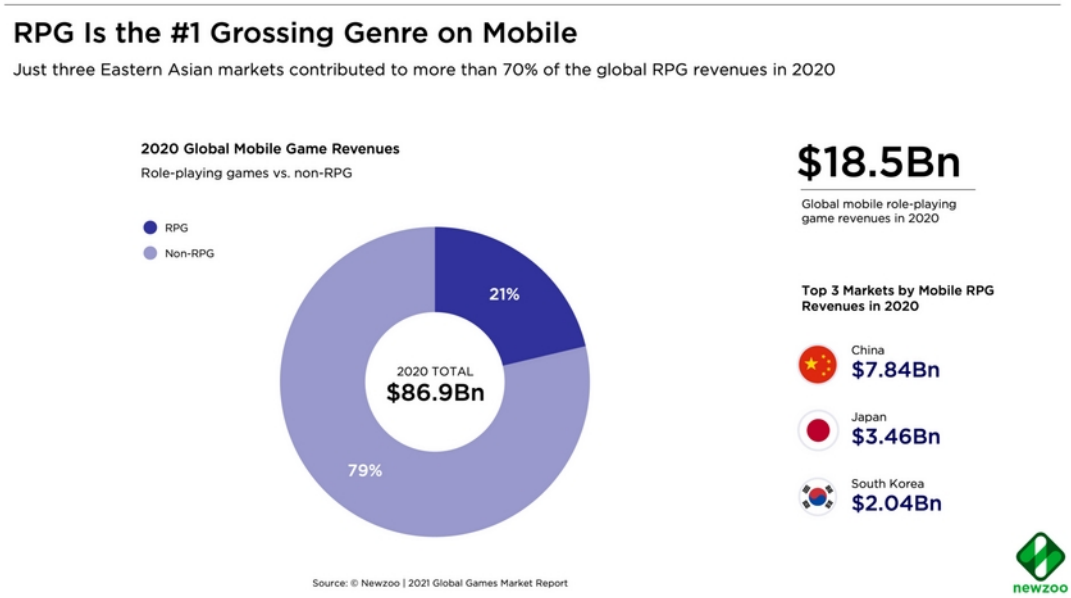
Egy telefon exkluzív műfaj a Location-Based RPG, azaz a hely-alapú szerepjáték. A való világban a mozgásunkat GPS funkcióval követve léphetünk interakcióba az alkalmazásbeli térképen elhelyezett objektumokkal. A ma is legnépszerűbb Pokemon Go megjelenése óta már több mint 1 milliárd letöltésnél jár [7].

2.3 Piac

Ahhoz, hogy megérthessük, hogy mi teszi népszerűvé ennek az összetett műfajnak a játékait, érdemes megvizsgálni a telefonos szerepjáték piaci tényezőit. Megismerésükkel erős tervezési alapot lehet biztosítani a készítendő programnak, elősegítve annak sikerességét.

2.3.1 Az RPG eredményei számokban

A telefonos játékok közül a legnagyobb bevételi mutatókkal a szerepjátékok rendelkeztek az elmúlt években. Ennek kivizsgálása érdekében több átfogó kutatást készítettek az elmúlt években [8] [9].



3. Ábra: A szerepjátékok és nem szerepjátékok bevételeinek aránya, kiegészítve a három legnagyobb összeget hozó országgal [8]

Az RPG 2020-ban összes telefonos műfaj bevételének a 21,3%-át teszi ki, amely összesen 18,5 milliárd USA dollár, ezzel maga mögé utasítva versenytársait. Ennek a döntő többsége Kelet-Ázsiában összpontosul, köztük Kínában, Japánban és Dél-Koreában. Ezzel összefüggésben ezen országokban a 50 legtöbbet kereső játéknak több mint a fele RPG, azaz ott egyértelműen ez a műfaj dominál, szemben a nyugati országokkal, mint az Amerika Egyesült Államok vagy Nagy-Britannia, ahol ez a szám jóval alacsonyabb.

A kutatások kereteiben kikérdezték a felhasználókat arról, hogy miben különböznek a szerepjátékok az átlagos telefonos játékoktól. Több szempont közül három esetben voltak markánsan eltérő értékek a k:

- Az RPG-t negyedannyival kevesebben választják várakozási idő eltöltése miatt, mint egyéb társait, ezzel párhuzamosan hosszabb ideig is játszanak vele alkalmanként.

- Az RPG-nél erősebb a világ és történetszál megismerésének pozitív élménye.
- Ugyanúgy az RPG-nek nagy húzóereje a karakter szerepében, illetve a történetben való elmerülés.

Utóbbi kettő azért is fontos, mert Kelet-Ázsiai országokban a legnépszerűbb, így ha valaki igazán népszerű szerepjátékot szeretne telefonra gyártani, mindenképpen érdemes ezen kultúrák alapján megtervezni a környezetet és a narratívát, de legalábbis az ottani legfontosabb nyelvekre szükséges lefordítania.

2.3.2 Játékosok véleménye

A felmérések alapján, amikor már a szerepjátékok nem biztosítják a fentebb megadott értékeket, elveszti a játékosait. Mindemellett a döntő ok a szerepjátékok befejezésének az, hogy a játékosok unalmasnak, vagy repetitívnek érzik azt (minden másik egyéb ok fontossága különbözik országonként).

Ahhoz hogy sikeres játékot lehessen tervezni, mindenképpen meg kell ismerni, hogy mitől érzi a felhasználó azt, hogy „elmerül” a játékélményben, illetve mit kell elérnünk, hogy ne legyen se unalmas, se repetitív. A kutatásokban 10 000 játékos véleményét kérték ki ezen célból. Ezek közül 7 lényeges pontot lehet kiemelni (természetesen ezek fontosságának arányai is különböznek országonként):

1. Több mint 50 különböző játékbeli karakter.
2. Tárgyak kombinálása/barkácsolása.
3. Napi küldetések.
4. Karakter képességeket befolyásoló tárgyak
5. Gyűjtemények (gyűjthető tárgyak, karakterek vagy kártyák)
6. Ismétlődő események (például minden karácsonykor különleges ellenfél).
7. Nem ismétlődő események.

Figyelemre méltó eredmény emellett, hogy a játék megtartó erejét nem befolyásolja az, hogy annak harcrendszere kör- vagy akció-alapú, vagy hogy pontosan milyen hibrid típusához tartozik az RPG-nek!

2.3.3 Sikerjátékok közös elemei

A top 15 legmagasabb bevételű telefonos játékok megismerése alapján a következő megfigyelések lettek összegyűjtve

- A felük *Intellectual Property (IP)* játék, azaz már egy létező szellemi tulajdonú franchise részei, azaz már máshol is megjelent a cím vagy történet sorozatokban, filmekben, könyvekben stb. Természetesen ezeknek már van egy kialakult táboruk, így könnyebb nekik érvényesülni. Példa rá a Harry Potter: Hogwarts Mystery vagy a Marvel Strike Force.
- Az MMORPG-ken kívül szinte mindenhol vannak gyűjthető karakterek (sokszor hívják ezeket Collection RPG-nek, azaz gyűjteményes szerepjátéknak). Általában ezeket a karaktereket lehet felhasználni a küzdelmek során, akár egyet irányítva akár többet, ezért is fontos ezekből minél többet és minél jobbat megszerezni. Rendszeresen szokták bővíteni a készletet, így ennek segítségével hosszú ideig lehet lekötni a felhasználókat.
- Néhány kivétellel majdnem mindegyikben megtalálható az úgynevezett *gacha* mechanika. Maga a szó eredetileg azt az „ajándékot” nevesítette, amelyet műanyag kapszulában lehetett vásárolni árusítógépekből úgy, hogy nem tudja a vásárló, hogy melyiket kapja meg. Gyakorlatban a témánkban annyit tesz, hogy játékon belüli valutából véletlenszerű játékbeli tárgyat (kártyát, karaktert, tárgyat) lehet vásárolni. Ezek között vannak gyakoribbak és ritkábbak is, így lehetnek olyanok amikhez nagyon sok próbálkozás kell, hogy megszerezhesse a felhasználó. Gyakran hasonlítják szerencsejátékhoz ezt a rendszert, amely könnyen tud függőséget is kialakítani. (Egyes országokban, például Belgiumban, már megkezdték törvényekbe foglalt a korlátozását.)
- A karakter fejlődés permanens, nincs felső korlát szabva neki.
- Bő többségében létezik „véletlenszerű zsákmány”. Ez azt jelenti, hogy ellenfél legyőzése, vagy feladat elvégzése után előre nem pontosan ismerhető jutalmat kapunk.
- Mindegyik cím rendszeresen kap frissítést a fejlesztőitől. Ezekkel új pályák, karakterek, tárgyak, játékmódok és ideiglenes események érkeznek.

- Ugyanúgy mindben létezik a szociális interakciónak a lehetősége. Legtöbbször ez kimerül online ranglista vagy szövetségi rendszerek létezésében, de sokban van lehetőség egymással megküzdeni, illetve versenyezni is.

2.4 Összegzés

A fenti felsorolásokból nagyon szépen kirajzolódik, hogy a legnépszerűbb játékok közös elemei a felette megfogalmazott felhasználói elvárásokra ad jó megoldásokat. Mindezek leírtak alapján a következő konklúziókat és esetleges alapelveket lehet levonni egy telefonra készülő RPG fejlesztése céljából:

- A harcrendszernél sokkal fontosabb a gyűjthető dolgok sokszínűsége. Az, hogy milyen műfajt valósít meg, nem súlyos jelentőségű.
- Bővíthetőségre kell fókuszálni, hogy egyszerűen lehessen mindig új tartalmat hozzáadni. Ez vonatkozik szinte minden játékelemre, mint irányítható karakter, ellenségek vagy felszerelések, de akár pályák is.
- Szociális interakció lehetőséget kell biztosítani, ahol össze tudja magát mérni a játékos a többiek eredményével, és akár meg tud velük küzdeni is.

3 Felhasznált technológiák

3.1 LibGDX

A libGDX egy ingyenes és nyílt forráskódú játékfejlesztő OpenGL csomagoló (wrapper) keretrendszer. Java nyelven íródott, néhány C és C++ komponenssel a teljesítményfüggő kódokhoz. Multiplatform lehetőséget kínál, tehát lehetővé teszi asztali és mobil játékok fejlesztését ugyanazon kódbázis használatával. Legnagyobb előnye ennek, hogy az Androidra (vagy IOS-re) tervezett alkalmazás futtatásához nem kell emulátort elindítani, hanem azt asztali környezetben is meg lehet nyitni szinte teljesen ugyanazzal a kóddal.

Alapvetően kétdimenziós grafika megjelenítésére alkalmas, bár már létezik hozzá háromdimenziós megjelenést támogató komponens.

Eredeti programozási nyelve a Java, de ezen felül Kotlinnal is lehet használni. A dolgozatban megtervezett program Java nyelven készült, mivel segédanyagok és dokumentációk bő többsége is azzal található meg.

Szinte mindegyik nagyobb Javát támogató fejlesztőkörnyezeten használható (Eclipse, IntellIJ). Mivel a dolgozat célja egy Android alapú alkalmazás megírása, Android Studio-ban készül, mert azon keresztül lehet telefonon is kipróbálni a programot.

3.1.1 Felépítése

Felépítése moduláris, amely a hat alapvető modulon túl továbbiakkal is bővíthető a készítendő játék természetéhez igazodva. Ezek a modulok interfészek formájában biztosítják az operációs rendszerrel történő interakciót. [10]

A hat alapul szolgáló modul, és azoknak funkciója felületesen:

- *Application*: futtatja az alkalmazást, és tájékoztatja az API-klienst az alkalmazásszintű eseményekről, például az ablakok átméretezéséről. Naplózási lehetőségeket és lekérdezési módszereket biztosít, például a memóriahasználatét.
- *Files*: kezeli az adott platform fájlrendszerét. Különböző típusú fájlhelyekről absztrakciót biztosít egy egyéni fájlkezelő rendszeren keresztül.

- *Input*: tájékoztatja az API-klienst a felhasználói bemeneti eseményekről, mint az egér kattintás, billentyű lenyomása, képernyő érintés vagy a telefon gyorsulásérzékelőjének üzenete. A lekérdező és eseményvezérelt feldolgozás is támogatott.
- *Net*: lehetővé teszi az erőforrásokhoz való hozzáférést HTTP-vel és HTTPS-sel a platformok között, valamint TCP szerver és kliens socketek létrehozását biztosítja.
- *Audio*: lehetőséget biztosít a hangeffektusok és zene lejátszására, valamint az audioeszközök közvetlen elérésére a PCM audio csatlakozón keresztül.
- *Graphics*: megjeleníti az OpenGL ES 2.0-t, információkat nyújt az eszköz aktuális kijelzőjének és az alkalmazás ablakának tulajdonságairól, valamint az aktuális OpenGL-környezetről és hozzáféréséről. A tulajdonságok közé tartozik többek között a képernyő mérete, a pixelsűrűsége és a keretpuffer tulajdonságaira vonatkozó információk, mint a színmélység, a mélység pufferek értékei és az élsimítási képességek.

Ezen modulok a *Gdx* osztály statikus mezőin keresztül érhetők el. Ez lényegében globális változók halmaza, amely lehetővé teszi a könnyű hozzáférést a *libGDX* bármely moduljához.

Egy egyszerű példa a *Graphics* modul eléréséhez:

```
//kliens terület szélességének lekérdezése logikai pixelekben
float width = Gdx.graphics.getWidth();
```

A *Gdx.graphics* hivatkozás a háttér-implementációra, amelyet az alkalmazáspéldány az alkalmazás indításakor példányosított. Más modulok is ugyanígy érhetők el, a *Gdx.app* az alkalmazás eléréséhez, a *Gdx.files* a fájlok megvalósításához, és így tovább.

3.1.2 Bővítmények

Ahogy már fentebb is említve lett, a libGDX egyik fontos tulajdonsága a nyílt forráskódú mivolta, amely elősegítette a harmadik fél által megírt bővítmények megszületését. Ezek közül több olyan is van, ami olyan nagy mértékben kiterjeszti a

keretrendszer képességeit, hogy a libGDX projekt generátorában is helyet kapott, amely gradle segítségével automatikusan felépíti azokat indításkor [11].

Érdemes megismerni, hogy milyen lehetőségeket biztosítanak:

- **Bullet:** A Bullet egy C++-ban íródott háromdimenziós testek ütközésének dinamikájával foglalkozó könyvtár. Létrehozhatóak benne puha (szövet, rugó, hajlékony testek) és szilárd (egyszerű, fix formájú) testek.
- **Freetype:** Ezen bővítmény segítségével lehet a programban használt szövegeket vektorgrafikus formában eltárolni. Általános esetben *BitmapFont* típusúak a feliratok, amelynek hátránya, hogy azok pixelgrafikusan jelennek meg, így a képernyő átméretezésénél torzulnak, emellett a betűk képként lesznek elmentve, így azok több tárhelyet foglalnak. Vektorgrafikus tárolás esetén ezen problémák nem állnak fent, a szövegek bármilyen mértékben nagyíthatóak.
- **Tools**: Több kisebb, hasznos programot foglal magába. Két legfontosabb közülük a *TexturePacker,* illetve a *Hiero*. Az előbbi az animáláshoz nyújt hasznos eszközöket. Kétféle animálás stílust támogat: a csontozás- és a képkocka-alapút. A csontozás azt jelenti, hogy egy karakterünket testrészekből, avagy „csontokból” állítjuk össze, amikkel külön megrajzoljuk például egy ember alkarját, felkarját, combját stb, és mozgásnál ezeket a csontokat forgatjuk és toljuk el. Előnye az, hogy könnyen cserélhetőek benne ezek a részletek, például ha új kesztyűt vesz fel az irányított karakter, csak a kesztyűt kell újrarajzolni és lecserélni. A képkocka-alapú animációnál egy mozdulatsor több képkockán van megrajzolva, és azokat váltogatva látszódik a karakter mozgása. Előnye, hogy könnyebb implementálni és kisebb erőforrást igényel megjelenítése, viszont ha a karakteren egy apró részletet meg szeretnénk változtatni, azt minden képkockán meg kell tennünk. A *Hiero* segítségével lehet a programban *BitmapFont* típusú betűket készíteni, és megjeleníteni állítható mérettel, színnel és egyéb típussal. A bővítmény magába foglal még két- és háromdimenziós részecske szerkesztő eszközt, mellyel többek között tüzet, füstöt és ködöt lehet készíteni.
- **Controllers**: Joystick, gamepad és egyéb kontrollerekkel történő inputbevitelt támogatja. Működése megegyezik az alapból támogatott beviteli eszközök működésével.

- **Box2D** és **Box2Dlights:** A Box2D egy kifejezetten elterjedt és népszerű kétdimenziós fizikai könyvtár libGDX-re portolása. Alapvetően C++-ban íródott, libGDX-en belül Java nyelvvel használható. Létre lehet benne hozni különböző típusú testeket - statikust, dinamikust, kinematikust – melyekre máshogy hat a környezetük. Tudja szimulálni a tehetetlenséget, tömeget és gravitációt. Különlegessége, hogy a távolságra és hosszra használt mértékegysége nem a programozásban megszokott pixel, hanem a méter, ami tovább gyűrűdzik a sebesség és a gyorsulás mértékegységeibe is. A Box2Dlights a Box2D világán belüli sugárkövetést támogató rendszer. A fényeken és árnyékokon kívül lehet szimulálni például robbanást is.
- **Ashley:** Egy Java nyelven írt kisebb entitás kezelő keretrendszer. Olyan keretrendszerek ihlették, mint az Ash (innen a név) és az Artemis. Segítségével a játékban létrehozott entitások nem leszármazás alapján, hanem a C nyelvben is létező struktúrákhoz hasonló összeállításban kapják meg tulajdonságaikat.
- **AI:** Mesterséges intelligencia megvalósítását segítő bővítmény. Magába foglalja többek között az élethű csapatban történő mozgást, különböző útvonalkeresési technikákat, mint például az A* keresést, emellett döntéshozási struktúra támogatásaként állapotgépet is lehet vele készíteni.

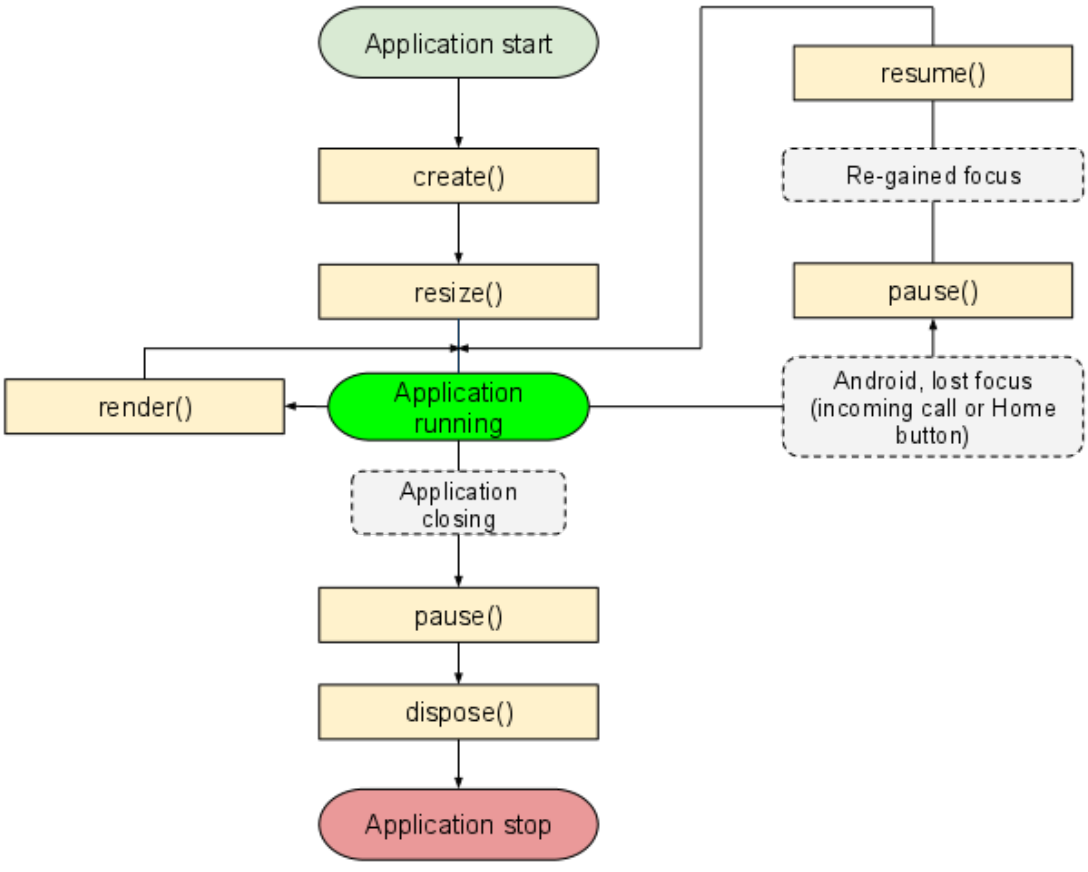
3.1.3 Életciklus metódusok

A libGDX az android alkalmazásokhoz hasonló életciklus metódusokkal rendelkezik. Ezeknek a felülírásával lehet formálni a program működését, és időzíteni függvények végrehajtását. Meghívásuk automatikus, egyetlen feltétele, hogy az ezt használó osztály implementálja az *ApplicationListener* osztályt, vagy legyen példányosítva benne egy *Application* objektum. Hasonló az Android alkalmazások ciklusához.

Az életciklus metódusok:

- `create()`: Egyszer hívódik meg az alkalmazás létrehozásakor.
- `resize(int width, int height`: Ez a metódus minden alkalommal meghívásra kerül, amikor a játék képernyőjét átméretezik, és a játék nincs *paused* állapotban. Egyszer közvetlenül a `create()` metódus után is lefut. A paraméterek a képernyő új szélessége és magassága pixelben.

- `render()`: A játékciklus által az alkalmazásból meghívott függvény minden rendereléskor. A játéklogikai frissítéseket általában szintén itt hajtják végre.
- `pause()`: Androidon ugyanakkor hívódik meg, mint az ott lévő, vele azonos nevű metódus (például ha megnyomják a Kezdőlap gombot, vagy bejövő hívás érkezik). Asztali számítógépen ugyanez akkor történik meg, amikor az ablak kicsinyítve van, emellett közvetlenül a `dispose()` előtt, amikor kilép a felhasználó az alkalmazásból. Alkalmas hely a játék állapotának elmentésére.
- `resume()`: Androidon létező párjához hasonlóan az szüneteltetett állapotból való kilépés után hívódik meg.
- `dispose()`: Akkor fut le, ha az alkalmazás megsemmisül. Megelőzi még ilyenkor a `pause()` függvény.



4. Ábra: libGDX életciklus metódusainak futási sorrendje.

3.1.4 Kirajzolás

A kirajzolás folyamatának megértése szükséges lesz a keretrendszer használatához. Ennek legfontosabb elemei a következőek.

Az eredeti formátumból (pl. PNG) dekódolt és a GPU-ra feltöltött képet textúrának nevezzük. A textúra megrajzolásához szükséges annak geometriai alakjának leírása, mint hogy hol vannak a kép sarkai. Geometriai alak lehet egy téglalap, és a textúra leírható úgy, hogy a téglalap minden sarka megfeleljen a textúra egy-egy sarkának. Azt a téglalapot, amely egy textúra részhalmaza, textúra régiónak nevezzük.

A tényleges rajzoláshoz először a textúrát össze kell csomagolni (ezt beépített osztályok elvégzik nekünk, ki is gyűjtik a geometriai alakjának leírását), majd azt tovább kell adni az OpenGL-nek, hogy az megrajzolja. A textúra méretét és helyét a képernyőn a geometriai leírás, és az OpenGL viewportja (ez adja meg, hogy a képernyőn éppen mi szerepel, ami azért fontos, mert azt nem kell lerajzolni, ami nem látszik a képernyőn) határozza meg. Sok kétdimenziós játék úgy konfigurálja a viewport-ját, hogy megfelelő legyen a képernyő felbontás számára.

Általános szokott az lenni, hogy ugyanazt a textúrát vagy textúrarégiót több különböző helyen rajzolják meg, ezzel memóriát takarítva meg. Nem lenne hatékony viszont ilyen esetben minden egyes ugyanolyan alakzatot egyesével küldözgetni a GPU-nak. Ezt segíti a *SpriteBatch* osztály. Összegyűjti az azonos elemeket, és lejegyzi azoknak a geometriai leírását. Amikor már nem kap több ilyet, egyszerre elküldi a GPU-nak az egy darab képet a több különböző koordinátával, ami kirajzolja azokat.

A libGDX-hez tartozó további kisebb technikai részleteket (nagyrészt hogy milyen segédosztályai vannak, és azokat hogyan kell használni) az önálló munkán belül fejtem ki, ott ahol először alkalmazva van, a könnyebb érthetőség kedvéért.

3.2 Tiled térképkészítő

A Tiled egy kétdimenziós pályaszerkesztő program. Elsődleges funkciója a különböző formájú csempe alapú térképek szerkesztése, de támogatja a szabad képelhelyezést is, valamint biztosít eszközt arra, hogy a készítendő program által használt extra információval jelöljön meg elemeket. Támogatja a klasszikus téglalap alakú csemperétegeket, de a vetített izometrikus, lépcsőzetesen eltolt izometrikus és lépcsőzetesen eltolt hatszögletű rétegeket is. [12]

A térképek elkészítéséhez csempekészletet (ami különböző textúrájú négyzeteknek az összefoglaló fájlja) használ, amely lehet egyetlen kép, amely több csempét tartalmaz, vagy lehet egyedi képek gyűjteménye. Bizonyos mélységhamisítási technikák támogatása érdekében a csempék és rétegek egyéni távolsággal eltolhatók, és a megjelenítési sorrendjük változtatható. A csemperétegek szerkesztésének menete hasonlóan működik egy nagyon egyszerű rajzolóprograméhoz. Elsődleges eszköze egy ecset, amely lehetővé teszi a csempeterületek festését és másolását, emellett támogatja a vonalak és körök rajzolását is. Ezeken kívül is számos egyéb szerkesztő eszköze létezik (például van olyan, amely automatikus terepátmeneteket készít). Létezik benne mintaillesztésen alapuló festés is, amivel könnyen lehet automatizálni a pályakészítést.

A Tiled támogatja az objektumrétegeket létrehozását, amelyek hagyományosan csak a térkép információkkal való kiegészítésére szolgálnak, de képek elhelyezésére is használhatók. Hozzáadható téglalap, pont, ellipszis, sokszög, vonallánc és csempe objektumok. Az objektumok elhelyezése nem korlátozódik a csemperácsra, szabadon méretezhetők vagy forgathatók is. Ezek hozzáférhetőek a készítendő programból, így a későbbiekben felhasználhatóak egyes algoritmusukhoz;

3.3 Pinta

A Pinta egy ingyenes, nyílt forráskódú program rajzoláshoz és képszerkesztéshez. A klasszikus Paint-nél komplexebb, de a PhotoShop-nál jóval egyszerűbb alkalmazás. A szakdolgozaton belül alapvetően karakterek pixelgrafikájú megrajzolására lett alkalmazva, kihasználva a rétégkezelő funkcióját, mely segítségével egy mozdulatsor képkockáit egymásra helyezve lehet átrajzolni. [13]

3.4 GDX Texture Packer

A libGDX keretrendszer támogatásához létrehozott hasznos kis segédprogram. Tulajdonképpen a libGDX-ben már létező *TexturePacker* osztályokhoz készített vizuális felhasználói felület. Célja a készítendő programban felhasznált textúrák, köztük térképelemek, képek, karakteranimáció képkockáknak egy képbe való tömörítése. Ezt a textúrákat tömörítő képet nevezik textúra atlasznak. Az atlasz készítésénél egyes képeket régiókhoz lehet rendelni, amit fel lehet használni később például animációk lejátszásához, ahol a régió képeit sorban kirajzolva látszódik a mozgás. [14]

A textúra atlasz alapvető célja, hogy megspórolja a képek egyesével történő betöltésének futási idejét, illetve memóriában elfoglalt helyét. Az atlasz későbbi feldolgozásához és használatához létezik a *TextureAtlas* osztály, amely segítségével sokkal egyszerűbb ilyen formában kezelni a képeket (főként animációkat), mint a hagyományos formákban.

4 Tervezés

Ebben a fejezetben a készítendő program tervezési elvei és szempontjai lesznek prezentálva. Elsőként a feladat teljesítésének kitűzött céljai lesznek meghatározva, illetve hogy hogyan nézzen ki maga a végeredmény, később pedig a felsőszintű architektúra bemutatása történik meg.

4.1 Feladat bemutatása

A sikeres RPG játék megtervezéséhez három irányból érkező feltételeknek kell eleget tenni. Egyrészt az irodalomkutatásban identifikált műfaji sajátosságok közül néhányat implementálnia kell. Másodrészt a piackutatásból megismert sikerjátékok közös tulajdonságaiból párat fel kell használnia. Harmadrészt a libGDX keretrendszer képességeihez hozzá kell igazítani a célokat.

4.1.1 Műfaji sajátosságok

Több különböző alap- és hibridműfaj közül személyes preferencia alapján választottam, mivel az bizonyítottan nem befolyásolja nagy mértékben az eredményességet. Az akció szerepjáték mellett döntöttem, mivel véleményem szerint izgalmasabb és dinamikusabb játékélményt kínál társainál, illetve a fejlesztési folyamata is érdekesebbnek tűnik, tekintve a valósidejű harcrendszer összetettségét a menüalapúval szemben.

Az ARPG elemei, melyek alapjait fogják képezik a játéknak:

Egy karaktert lehet irányítani valós időben. Mivel központi szerepet játszik, a játéktérben mindenképpen a képernyő közepén kell elhelyezkednie, így azon van a fókusz. Ennek a karakternek legyenek a teljesítményét befolyásoló tulajdonságai, melyeknek fejleszthetőnek kell lenniük.

Az irányítása dinamikus és valós idejű. Mindkettőt szolgálja az ilyen játékoknál népszerű joystick irányítás. Ez a kontrollereken fizikailag is létező társához hasonlóan működik, csak kétdimenziós megjelenítésben. Ez annyit tesz, hogy a képernyőn egy nagyobb körben található egy kisebb kör, és az utóbbi, a nagyobb kör középpontjához viszonyított eltolásával megegyező irányába mozog a karakter.

5. Ábra: Telefonos Harmageddon című ARPG felülete.

Szerepeljenek ellenfelek, melyek ugyanúgy valós időben cselekszenek. Ezekkel az ellenfelekkel legyen lehetőség harcolni. Ez kimerülhet abban, hogy megfelelő távolságon belül tartózkodva a két fél tudja sebezni a másikat tulajdonságaiktól függő mértékben, de mindenképpen bővíthetőnek kell lennie.

A pálya, mely a játékteret képezi, felülnézetes vagy izometrikus legyen. Legyenek rajta olyan elemek, amelyek befolyásolni tudják a mozgást vagy a harcot (például egy fa, amin nem lehet keresztülmenni);

Mivel telefonos játékról beszélünk, az asztali számítógépekhez képest kis képernyő áll rendelkezésre, emiatt intuitívnak és letisztultnak kell lennie. Ez megvalósításban annyit eredményez, hogy az irányítást befolyásoló felhasználói felület (később UI, mint User Interface) elemeknek a jobb és bal alsó sarokban kell lenniük, mivel ott éri el a felhasználó kényelmesen őket, emellett nem szabad feleslegesen sok gombnak vagy egyéb UI elemnek megjelennie, amelyek sokat takarnának a játéktérből.

4.1.2 Népszerű játékelemek

A piackutatás fejezet végén összegzett három legfontosabb eredmény (tartalmi sokszínűség, bővíthetőség, szociális interakció biztosítása) közül egyet választottam. A bemutatóban megfogalmazott elsődleges célom alapján, amely szerint egy olyan játékkeretet szeretnék megvalósítani, melyet később már csak tartalommal kelljen

feltölteni, a bővíthetőség határozható meg, mint alaptétel, mely később elősegíti a sokszínűség megteremtését is.

A bővíthetőség figyelembevétele sokkal inkább a kód felépítésében fog nagy szerepet játszani. Az eredménynek lehetőséget kell biztosítania ahhoz, hogy több különböző karaktert lehessen irányítani, azoknak különböző képességeik lehessenek, ezekkel szemben bővíthető típusú, különböző viselkedésű és kinézetű ellenségek létezhessenek, és különböző felépítésű pályákon lehessen velük összecsapni.

4.1.3 LibGDX lehetőségei

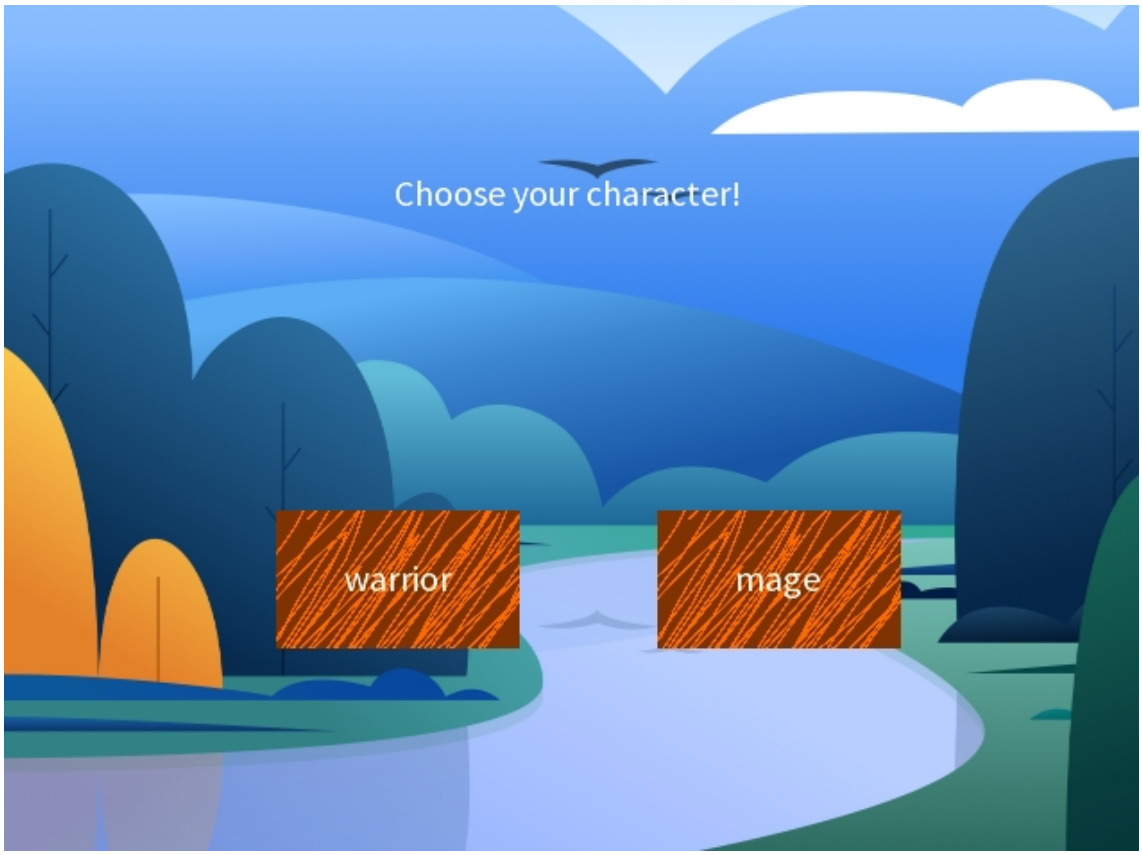
A libGDX keretrendszerben rejlő lehetőségek többsége inkább az architekturális felépítésben játszanak majd szerepet. Mivel alapvetően játékfejlesztésre tervezték, a fentebb megfogalmazott célokat tudja támogatni, egyedüli szűk keresztmetszete, hogy kétdimenziós megjelenítéshez nyújt elsősorban eszközöket, háromdimenziós kihasználása nehezebb, ezért az előbbit választottam.

4.1.4 Játékmenet

A játékmenet nem túl bonyolult. Az ezt alkotó részek a már kész alkalmazásról készült képekkel lesznek bemutatva.

6. Ábra: Az új játék indításának felületet adó képernyő

Első állomásként az alkalmazás megnyitása után lehetőség van új játékot indítani.



7. Ábra: Karakter típus választását felkínáló képernyő

Új játék kezdése után ajánlja fel lehetőségként a kezdő karakterosztályokat egy külön ablakban. Kiválasztás után jelenik meg a játéktér, azon az első szint.

8. Ábra: A játéktér, amin szerepel a karakter és annak ellenségei is

A játéktérben lesz látható a pálya, azon a felhasználó karaktere és az ellenségek. A képernyőn lévő joystick-kel lehet a karaktert irányítani, amely ha az ellenségek közelébe ér, azok elkezdik megközelíteni és megtámadni. A felhasználónak van ugyanúgy lehetősége a képernyőn szereplő gombok segítségével sebzést ejteni az ellenfelein.

9. Ábra: Karakterfejlesztő képernyő, rajta a tulajdonságokkal és értékükkel

Miután mindegyiket legyőzte, megjelenik a karakterfejlesztő képernyő, amin ki lehet választani, hogy a karakter mely tulajdonságát erősíti a játékos. Miután az megtörtént, megjelenik újra a játéktér a következő pályával.

Abban az esetben, ha a játékosnak elfogy az életpontja az ellenségek támadásának következtében, a játék véget ér.

4.2 Architektúra

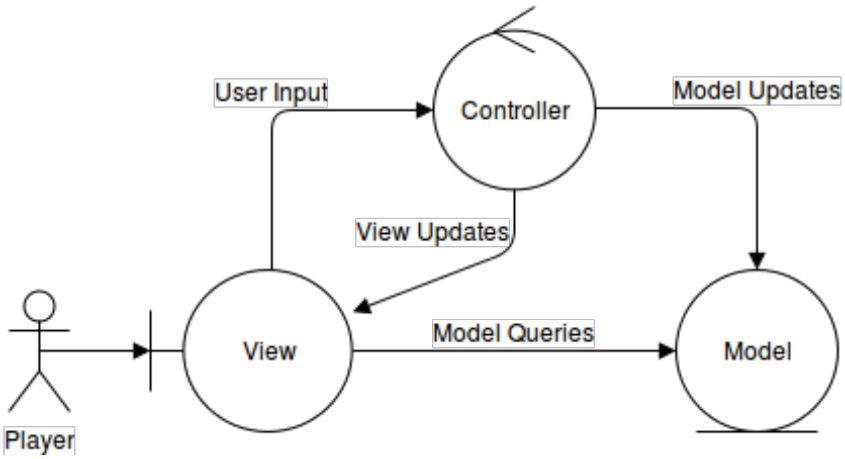
Mindenekelőtt mivel még nem foglalkoztam komolyabban sem játékfejlesztéssel, sem a libGDX keretrendszerrel, tapasztalatszerzés céljából készítettem két kisebb játékot (egy Breakout klónt, ahol labdával kell eltalálni csempéket, hogy leessenek, illetve egy mászkálós játékot, ahol térképen lehet animált karakterrel mozogni), célul kitűzve a megjelenítésért felelős osztályok megismerését, illetve ahhoz hasznos programozási minták társítását. Legfontosabb tanulsága az volt, hogy a megjelenítést és a játéklogikát mindenképpen érdemes két külön részbe

szervezni, illetve a megjelenítéssel foglalkozó komponenst több kisebb részre feldarabolni. Utóbbira leginkább azért lesz szükség, mivel egyes *libGDX* elemek létrehozásához és tartalommal való feltöltéséhez több kódsor szükséges, emellett hamar bele lehet zavarodni, hogy melyiknek mire van szüksége, ha egymás alatt szerepelnek.

4.2.1 MVC

A program a klasszikus Modell-Nézet-Vezérlő (Model-View-Controller, avagy MVC) tervezési mintát tekinti irányadónak, ahol a Nézet felel a megjelenítésért és a bemeneti események összegyűjtéséért, a Modell az üzleti- és játéklogikáért, a Vezérlő pedig a kettő közötti közvetítésért, ami szerint a megfelelő bemenetekre megfelelő függvényeket hív meg a modellben. E három rész meghatározott módon kapcsolódik egymáshoz [15]: Viszonyuk egymással a következő:

- A Modellt a Nézet leolvassa, a Vezérlő pedig frissíti a benne lévő adatokat.
- A Nézet leolvassa a Modellt, és a bemeneten történt eseményeket továbbítja a Vezérlőnek.
- A Vezérlő frissíti a Nézetet, és ő frissíti a Modellt *is.*

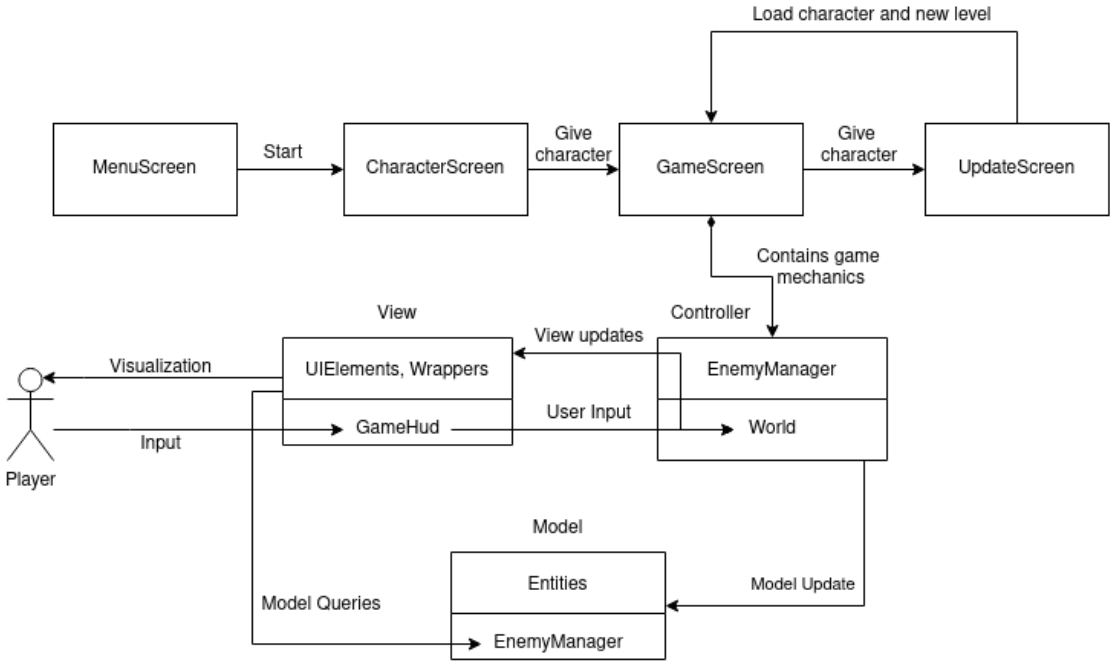


10. Ábra: MVC architektúra elméleti működésének bemutatása.

4.2.2 Tervek

A játékmenet alapján a négy részre lesz tagolható a program a futása alatt, amelyeket a saját *Screen* interfészt implementáló osztályok foglalnak magukba. Ezek a játéknyitó-, karakterválasztás-, játék- és karakterfejlesztés-képernyők.

A nyitó képernyő a legegyszerűbb rész, ennek létrehozása előtt fog megtörténni a játékhoz szükséges fájlok memóriába töltése, mint a pálya, karakteranimációt tartalmazó textúraatlaszok és egyéb textúrák. A következő karakterválasztó képernyő sem túl komplex. Az adott karaktertípus gombjára rákattintva létrehozza azt a karaktert, és továbbadja a következő képernyőnek.



11. Ábra: Program vázlatos működése és komponensek közötti kommunikáció. A játékképernyő tartalmazza az MVC mintájú részeket.

A programozásban a legizgalmasabb helyzet a *GameScreen* esetében következik el, mivel azon szerepel a játéktér, illetve maga a „játék rész” is. Itt fog aktív szerepet játszani az MVC architektúra.

A Nézet rész két részre lesz osztható elméletben. Az egyik a felhasználói bemenetek elemeinek megjelenítését (joystick és képesség gombok), illetve a bemeneti eseményeket kezelő *GameHud* osztály alkotja, a másik része pedig a tisztán csak a megjelenítést szolgáló kisebb osztályokból áll. Ezek között szerepel a pálya megjelenítését szolgáló *MapContainer* osztály, az egyes entitások (karakter és ellenségek) animációját kezelő *AnimationWrapper*, illetve az ezekben is szereplő kisebb csomagoló osztályok, mint a *TexturedDrawable és SpriteDrawable,* melyek egy saját *Drawable* interfészt valósítanak meg.

A Modell rész felosztása hasonlóképpen két részre tagolható. Az egyik fele a külső fájlokból betöltött adatok tárolója és kezelője, a singleton *ResourceManager* osztály. A második pedig a játéklogikát tartalmazó entitások, melyek az *Entity* interfészt valósítják meg.

A kettőt összekötő Vezérlő komponenst két osztály kezeli. A *World* osztály frissíti folyamatosan a Nézetet, illetve fogadja a a *GameHud* eseményeit, mint a joystick mozgatását és gombok megnyomását. A bemenetek függvényében meghívja a Modellt alkotó *Entity* interfészt megvalósító osztályokat (karakter és ellenségek) számon tartó és azokat kezelő *EntityManager* függvényeit.

Amint a *Modellben* a győzelmi feltétel teljesül, és elfogy minden ellenség, az *UpgradeScreen* jelenik meg, és a *GameScreen* továbbadja neki a *Character* osztályt. Az új képernyő kirajzolja a karakter tulajdonságpontjait, és gombok segítségével enged választani azok közül, Azok megnyomásával végrehajtódik a karakter fejlődése, és megjelenik újra a *GameScreen*-re vált. Minden újból indítás esetén frissíti az adott szint függvényében a térképet, az ellenfeleket és azoknak elhelyezkedésüket, illetve a hős elhelyezkedését, mindezt a szinteket kezelő *LevelManager* osztály segítségével, .

4.2.3 Egyéb tervezési minták

Több kisebb volumenű tervezési minta is alkalmazva lett egyes komponensek létrehozásában. Ezek elsősorban a fejlesztés könnyítését, illetve az előző pontokban kihangsúlyozott bővíthetőség lehetőségének kiterjesztését szolgálják. Itt egy kisebb összefoglaló következik belőlük, az hogy melyik osztály ezeket hogyan implementálja, az következő fejezetben, a bemutatásuknál lesz kifejtve.

- Menedzser tervezési minta lett alkalmazva azokban az esetekben, ahol több azonos típusú (vagy azonos szülőosztályú) entitást kell kezelni. Ilyen például a különböző erőforrásokat kezelő *ResourceManager* vagy a különböző pályák tartalmát tároló *LevelManager*. [16]
- Gyakran társulnak Singleton tervezési mintával a menedzser osztályok, mely biztosítja, hogy csak egy példány létezzen ezekből a program működése alatt.
- Csomagoló tervezési minta (más néven wrapper vagy decorator minta), amely lehetővé teszi új viselkedések hozzárendelését az objektumokhoz azáltal, hogy

ezeket viselkedést tartalmazó speciális burkoló objektumokba helyezi. Többek között a MapContainer vagy az AnimationWrapper fogja alkalmazni. [17]

- Gyártó metódus tervezési minta is alkalmazva lett, amely egy interfészt biztosít egy szuper-osztályban lévő objektumok létrehozásához, de lehetővé teszi az alosztályok számára, hogy módosítsák a létrehozandó objektumok típusát. Példa lesz rá a különböző *Enemy* típusú entitások gyártását kínáló *EnemyFactory*.

5 A megvalósítás

Ebben a részben lesznek kifejtve a megvalósítás alacsonyabb szintű részletei. A bemutatás elsősorban a futási sorrend alapján fog szerepelni, egy-két kivételtől eltekintve, a könnyebb megértés érdekében.

5.1 *Game* és alapok

A *Game* osztály az, amelyet a különböző platformokról történő indításnál azok példányosítanak (Androidnál egy *AndroidApplication*, asztali gépnél egy *Java Application*). Örököse a libGDX *ApplicationAdapter* osztályának, amely lehetővé teszi a már bemutatott libGDX futási ciklus metódusainak futtatását, így azt futtathatóvá teszi a keretrendszer számára.

Alapvető felelősségei közé tartozik többek között a létrehozásánál a textúrák és egyéb fájlok beolvasása a memóriába, a képernyő méret leolvasása és elmentése, illetve teljes kijelző megjelenítését és játékmenetet tagoló képernyők, a *Screen* interfészt implementáló osztályok kirajzolása, és az esetenkénti visszalépések kezelése. Arról, hogy a képernyőkön belül mi zajlik, nem kell tudnia. Mindezek mellett neki kell delegálnia a libGDX futási ciklus metódusainak meghívódásai közül a képernyő átméretezését, a renderelést, a képernyő frissítésének megállítását, amikor háttérbe kerül a program (például telefonhívás esetén), és újraindítását.

A fájlok betöltését, későbbi tárolását illetve lekérdezését a statikusan létező *ResourceManager* biztosítja, mely a fentebb már bemutatott menedzser és singleton tervezési mintákat alkalmazza. Gyakorlatban feladatát úgy teljesíti, hogy tartalmaz több, különböző típust magába foglaló *HashMap*-et, többek között textúrákhoz, betűtípusokhoz illetve textúraatlaszokhoz, melyeket módosítani függvényeken keresztül enged. A feltöltésnél egy azonosító *String*-et kell rendelni a fájlok mellé, amivel később a lekérdezésnél lehet hivatkozni rájuk azt.

```
//játékos animáció képkockáit tartalmazó textúra
atlasz //betöltése az alkalmazás elindításánál
ResourceManager.getInstance().loadTextureAtlas("player",
    "bob.atlas");
```

```
//annak későbbi lekérése egy új karakter konstruktorában
character =new Character(new AnimationWrapper(

ResourceManager.GetInstance().GetTextureAtlas("player")),
    mapContainer. getCollisionLayer());
```

A `getInstance()` metódussal lehet bárhonnan elérni, így nem kell tartalmaznia az azt használó osztályoknak.

A képernyőméretek leolvasása és tárolása a *Settings* osztály feladata. Működési elve megegyezik teljesen a *ResourceManager*-ével. Lehetőséget biztosít későbbi egyéb beállítások tárolására.

A *Game* a *ScreenContainer* osztályt használja fel az éppen megjelenített képernyő eléréséhez. Ő tárolja el a pillanatnyilag futó *Screen*-t implementáló osztályt, illetve megjegyzi az előzőt is,

A *Screen* interfész célja, hogy az azt öröklő osztályok átvegyék a megjelenítés feladatát a fő osztálytól. Ezt úgy éri el, hogy megvalósítja a a libGDX életciklus metódusait, így azokat a *Game* osztály meg tudja hívni amikor sorra kerülnek, azok pedig hasonlóképpen továbbdelegálják azt az egyes képernyőn szereplő elemek felé, amik a megadott pozícióban kirajzolják magukat. Az egyes *Screen* gyermekosztályok működése később lesz részletesen leírva.

Egy utolsó, de annál fontosabb feladata hátra van még a *Game* osztálynak. Létrehozza a grafikus megjelenítéshez szükséges libGDX *SpriteBatch* osztályát. Ő felelős a libGDX leírásában is bemutatott kirajzolásért (pontosabban a kommunikációért a program és a videokártyát kezelő OpenGL között). Fontos, hogy minden `render()` metódus hívásnál ugyanezt az objektumot érdemes átadni, mert menedzselése erőforrásigényes, így kontraproduktívvá tenné használatát. Ha mégis többet kell alkalmazni (két ilyen kivétel lesz, annak okai helyben lesznek kifejtve), akkor oda kell figyelni, hogy használatuk között ne legyen átlapolódás, mivel az konfliktus helyzetet okozna közöttük, és megjelenési anomáliákat eredményezne.

Amint megtörténik a *Game* inicializálása, létrehozza a *MenuScreen*-t, és beállítja azt a futó képernyőnek.

5.2 *MenuScreen* és a kirajzolható osztályok

A *MenuScreen* a tervezésben bemutatott első állomása a játéknak. Célja pofonegyszerű, működése kicsit bonyolultabb. Feladata kirajzolni a hátteret és a játékindító gombot, illetve a bővíthetőség jegyében később hozzáadott gombokat. Mindeközben figyeli, hogy azokat megnyomják-e, annak bekövetkezésekor a megfelelő következő képernyőnek adja át a teret.

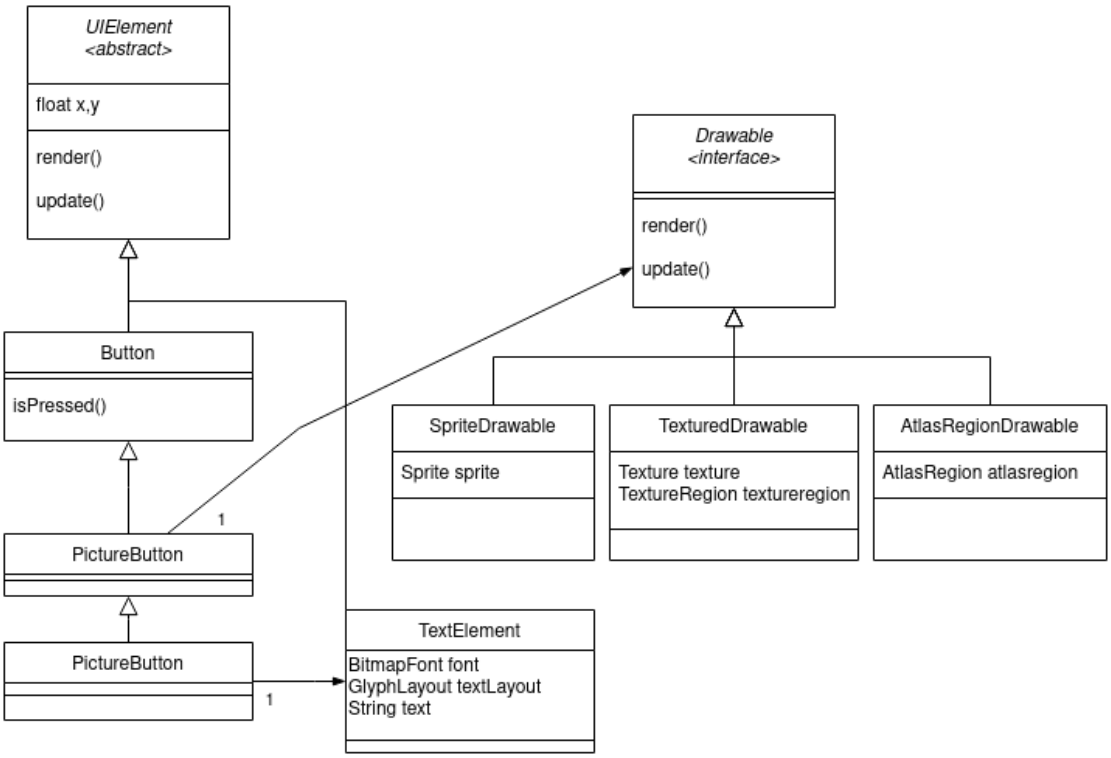
Működési folyamatának első lépése a háttér képének betöltése a *ResourceManager*-ből. Ez a kép egy beépített osztályú *Sprite*, ami a libGDX 3.1.4 fejezetben leírt kirajzolás működésében az adott textúrát, illetve annak geometriai alakzatának jellemzőit tartalmazza. Tudja kezelni a textúra pozícióját, forgatási szögét, színét és sok egyéb tulajdonságot, emellett a kirajzolását is egy függvényhívásra egyszerűsíti le.

Az izgalmasabb pillanat a gomboknál érkezik el. Ezek bemutatását egy fokkal távolabbról érdemes kezdeni.

5.2.1 A képernyőn megjelenő elemek

Ahogy fentebb említve lett, a *Screen* osztályok is továbbdelegálják a megjelenítés feladatát az egyes képernyőn megjelenő elemekre. Mivel ezeknek sok közös tulajdonsága és függvénye van, emellett mivel a végső cél egy könnyen bővíthető kódbázis létrehozása, a klasszikus objektum orientált elvek alapján célszerű megszervezni őket.

A kirajzolási függvényhívási lánc legvégén álló libGDX-ben natív osztályokat csomagoló osztályba szerveztem ki. Ezek a *Drawable* interfészt valósítják meg, mely biztosítja a különböző bemenetű renderelő, frissítő, méretlekérdező és egyéb függvények meglétét. Így biztosítja azt, hogy az egyes osztályok igényük szerint válasszák meg a nekik szükséges kirajzolást.



12. Ábra: A Screen osztályon megjelenő elemek egymáshoz kapcsolódó viszonyrendszere. Csak a legfontosabb függvények neve látható.

A *UIElement* egy absztrakt osztály, amely a képernyőn szereplő elemeket testesíti meg, és annak a pozícióját kezeli. Tartalmaz ugyanúgy renderelő és frissítő függvényeket, viszont ezeknek feladata az lesz, hogy meghívja a később benne tárolt kirajzolható osztályokat.

A *Button* a *UIElement* leszármazottjaként annyival bővíti ki azt, hogy megvizsgálja, hogy egy beérkező koordinátapár a saját területén történt-e. Neki a gyermekosztálya egy fokkal izgalmasabb, mivel a *PictureButton* már megjelenít a felületén egy *Drawable* objektumot is, aminek neki kell továbbítania a renderelés parancsát.

A *TextButton*, ahogy neve is utal rá, szöveget is képes megjeleníteni. Ennek kihívásait viszont a *TextElement* osztályba szerveztem ki, mely ugyanúgy egy *UIElement*. Ő tartalmaz egy natív *BitmapFont* típusú osztályt, mely a *ResourceManager*-ből betöltött adott betűtípusú ábécére kapott referenciát, illetve egy hasonlóan natív *GlyphLayout*-ot, ami a betűk megjelenítéséhez tárol extra információt,

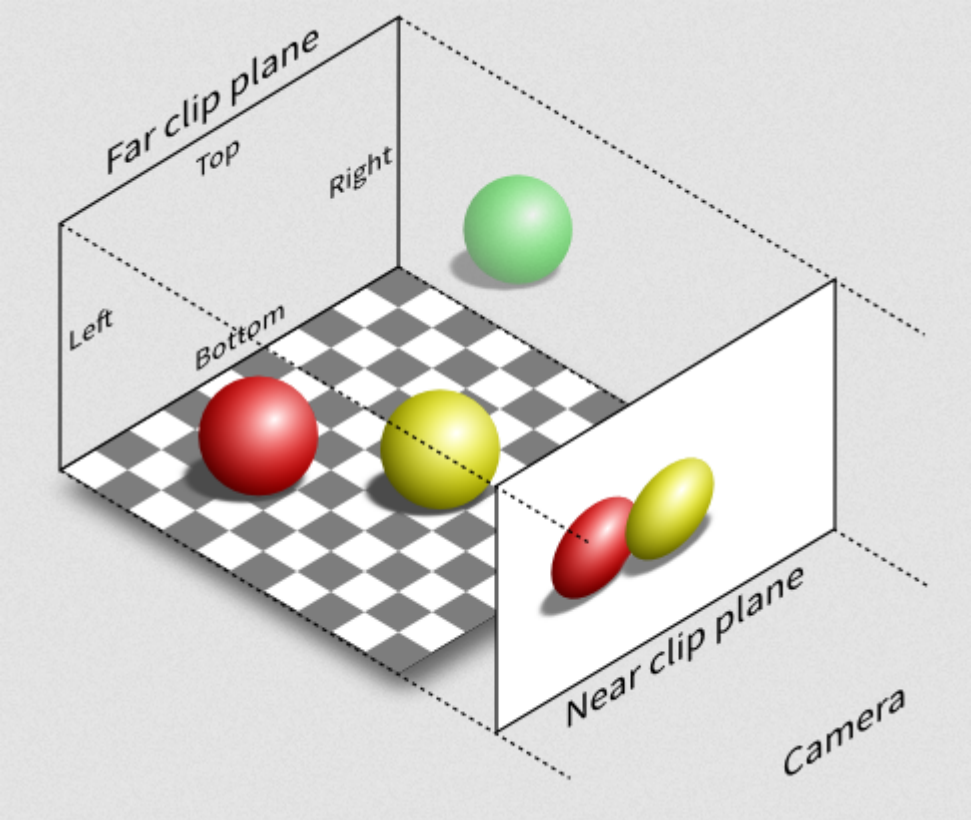
mint hogy milyen színű legyen, mennyivel legyen elforgatva stb. Ezek mellett természetesen a megjelenítendő szöveg egy *String* típusú változóban pihen.

Fog még szerepelni a játék stádiumban megjelenést szolgáló osztály, amik nem kaptak jelenlétet ebben a rendszerben, mivel a keretrendszer biztosít melléjük lényegesen bonyolultabb osztályokat, melyek a lehetőségeket kibővítik és a működést leegyszerűsítik, emellett vannak olyanok, melyek szorosan kötődnek egy-egy másik játékelemhez, és nem kapcsolódnak ezekhez az elemekhez. Természetesen ezek is be lesznek a későbbiekben mutatva.

5.2.2 A kamera és képernyőfelbontás

Egy fontos részlet, amiről nem esett még szó, viszont a megjelenítésnél kulcsfontosságú, a különböző felbontású képernyők kezelése.

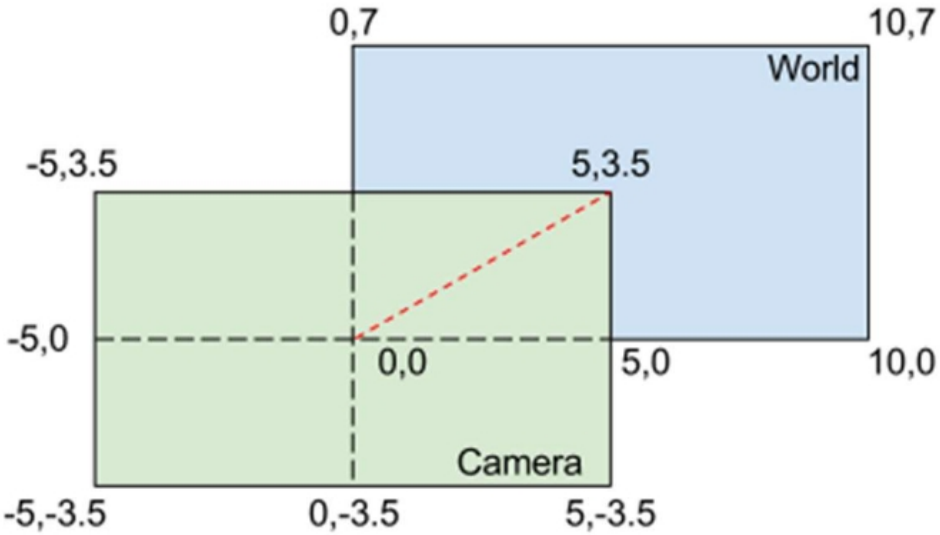
Természetesen játékfejlesztő keretrendszerhez méltóan, létezik natívan is ezzel a problémával foglalkozó osztályt. A *Camera* felhasználásával egyszerűbben kezelhető a különböző pixelsűrűség, mivel a *libGDX* felskálázza az eredményeket az eszköz tulajdonságinak megfelelően, ezenkívül megkönnyíti a látótér mozgatását, ha a jelenet nagyobb, mint egy képernyő.



13. Ábra: Ortografikus kamera működése.

A legtöbb kétdimenziós játékhoz hasonlóan ortografikus vetítésű kamerát használtam. Ezzel a módszerrel a kamerából párhuzamosan érkező sugarak érkeznek a képre, ennek következtében a megjelenített objektum mérete a renderelt képen állandó marad, függetlenül a kamerától lévő távolságától. [18]

Ezt is csomagoló osztályba szerveztem, ez az *OrthoCamera*, mely a natív *OrthographicCamera* örököse, illetve tartalmazza a V*irtualViewport* osztályomat. Feladata, a képernyőméret és a jelenetméret közti eltérések kezelése, illetve azok változása esetén a látótér hozzájuk igazítása.



14. Ábra: Kamera látószögének "elcsúszása" a képernyőhöz képest

Az alap *Camera* osztálynak van egy olyan „problémája”, hogy a játéktér koordináta rendszerének origóját a képernyő bal alsó sarkába teszi, emellett ezt pontot a látótér közepének veszi, így alapértelmezett esetben csak a játéktér egynegyede látszódik. Ennek megoldásához a csomagoló osztály a kamera pozícióját a játéktér közepére állítja, emellett megoldja azt egy beépített függvény meghívásával, hogy továbbra is úgy kezelje a koordinátákat, mintha a bal alsó sarokban lenne az origó.

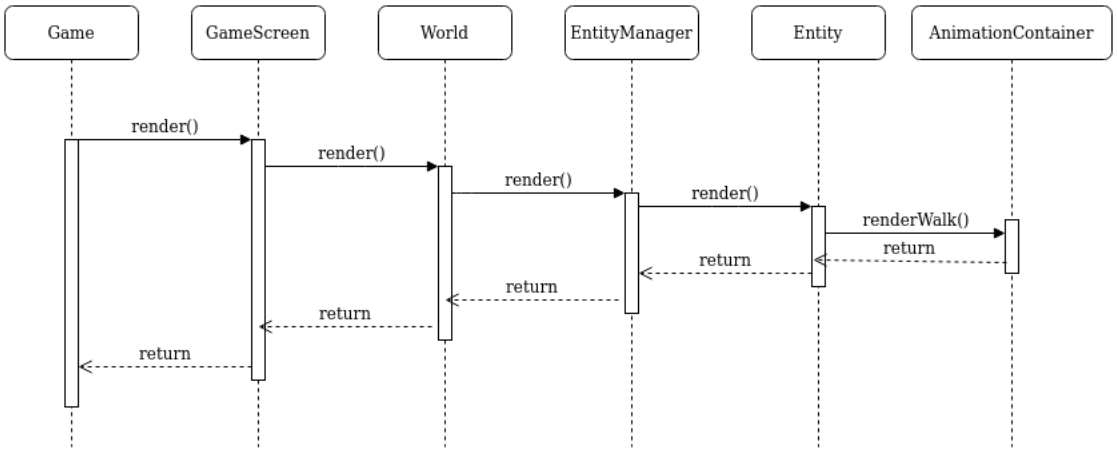
5.3 CharacterScreen

A *MenuScreen* ismeretében sok új dolog itt nem található. Hasonlóképpen elődjéhez a háttérképén kívül szöveges gombjai vannak, melyek a játékrészben irányítható karakter típusát írják le. A gomb megnyomásával a megfelelő karakter inicializálódik, és adja tovább a következő képernyőnek, amit egyúttal a *ScreenManager*-ben beállít főképernyőnek.

5.4 GameScreen

A program játék stádiumának teret adó képernyő. Feladata lesz megjeleníteni és kezelni a játékteret, emellett fogadnia a bemeneti eseményeket. Természetesen ezek a feladatok ki lettek szervezve. Két fontos osztályt tárol ezen célból. A *GameHud*, amely a bemenetet kezeli, és az annak felületet adó gombokat és joysticket jeleníti meg. Mellette van a *World*, amely minden más megjelenítését és a játéklogika menedzselését intézi.

A *GameScreen* az inicializálásánál hozza létre ezt a két osztályt. A *GameHud* konstruktorában megadja a *World* referenciáját, hogy tudjon neki jelezni a bemeneti eseményekről, ezek után minden egyes életciklus metódus meghívása során csak továbbítja a hívásokat.



15. Ábra: Példa a kirajzolás delegálásának hosszára. Jelen esetben egy entitás mozgásanimációjának megjelenítése zajlik.

5.4.1 GameHud és bemenetkezelés

A *GameHud*, ahogy már fentebb is le lett írva, a felhasználói felületet kezeli. Ez egy joystickból, illetve gombokból áll.

A *Joystick* osztály a natív *Touchpad* osztályt tartalmazza, és annak inicializálását szervezi ki, ugyanis az több részből áll. Két képet betölt a *ResourceManager*-ből, amelyekből egy *TouchpadStyle*-t hoz létre, amely tartalmazza a joystick hátterét és gombját. Ezt kapja meg végül az eredeti *Touchpad*, mellette a holttér kör sugarának az értéket. Utóbbi azért fontos telefonos játékoknál, mert az asztali gépektől eltérően nem egy precíz kattintásra képes egeret használ a felhasználó, hanem a sokkal pontatlanabbul operáló ujját, ami játékélmény szempontból nehézségeket okoz, ha apró mozgásokat szeretne végezni. A holttér adja meg azt a pixeltávolságot, amelyen belül ha esemény éri a joystickot, az nem fog reagálni. [19]

A joystickot egy natív *Stage* típusú objektum fogja actor-ként megkapni. A *Stage Actor* osztályokat kezel, köztük a megjelenítésüket, mind a rájuk érkező bemeneti események elkapását. Hátránya, hogy új *SpriteBatch* objektumot kell neki adni, és

figyelni kell arra, hogy ne legyen meghívva a kirajzoló függvénye, ameddig az eredeti *SpriteBatch* aktív.

A joystick mozgatása esetén szól a *World*-nek, és továbbítja neki a a joystick gomb x és y koordinátáinak százalékos elmozdulását a joystick hátterén belül.

A *GameHud* a gombokat a már bemutatott *PictureButton* típusként példányosítja. Amikor a libgdx input modulja bemeneti eseményt érzékel, annak a pontnak a koordinátáit az *OrthoCamera* segítségével átkonvertálja azt a megjelenített kép koordinátarendszerének megfelelő pontjára, Azt átadja a gomboknak, amelyek eldöntik, hogy az rajtuk történt-e meg. Ha igen, akkor a *GameHud* azt továbbítja a *World* számára.

```
if (joystick.getTouchpad().isTouched()) {
    world.moveUpdate(joystick.getTouchpad().
    getKnobPercentX(),joystick.getTouchpad().
    getKnobPercentY());
} else if (Gdx.input.isTouched(0)) {
    float
touchX=camera.unprojectXCoordinate(Gdx.input.GetX(),
    Gdx.input.getY());
    float
touchY=camera.unprojectYCoordinate(Gdx.input.GetX(),
    Gdx.input.getY());
}
if (attackBtn.isPressed(touchX,touchY))
    world.attackPressed();
else if (skill1Btn.isPressed(touchX,touchY))
    world.skill1Pressed();
```

Bár nem követelmény, de a felhasználói felületen az épp aktuális másodpercenkénti képkockasebességet (FPS) is meg lehet jeleníteni *TextElement*-ként, a libGDX graphics moduljának `getFramesPerSecond()` függvényével. Ennek a segítségével a játék futása közben ellenőrizni lehet azt, hogy nem éri-e aránytalanul nagy terhelés egy ilyen egyszerű játékban a megjelenítési folyamatokat, mert ha igen, akkor ott nagy valószínűséggel optimalizálási hiba található.

5.4.2 World és világépítés

A *World* a térkép és entitások megjelenítéséért, illetve a játéklogika eléréséért felel. Felépítésének magasabb szintű megfogalmazása a 4.2.2-es pontban már olvashatóak, így itt a hangsúly ennek a működésén, és a delegált feladatokat feldolgozó osztályokon lesz.

5.4.3 World működése

Három fontos változója van, melyeken keresztül mozgatja a szálakat.

- *EntityManager*, amely a képernyőn megjelenített entitásokat kezeli, mint a felhasználó által irányított karakter, annak ellenfelei, és esetlegesen később bővítéssel érkező nem játékos karakterek.
- *MapContainer*, ami egy térképet tud kezelni, amely a játékteret fogja képezni.
- *LevelManager*, ami a játékban létező szinteket kezeli, amik az új játék esetében az adott játéktér térképét, illetve az azon elhelyezkedő entitások pozícióját tárolja.

Ezen felül még lesz egy float típusú s*tateTime* változója, amely a játék elindulásától számított időt méri, ami az egyes animációk megjelenítéséhez lesz szükséges.

A *World* a konstruktorában példányosítja az osztályokat. Itt kéri le a *LevelManager*-ben tárolt adott szintnek az adatait, amit másik két osztálynak továbbít. Magában a konstruktorban a legelső szintet kéri le, későbbi esetekben (amikor már a karakterét fejleszti a felhasználó) lehet újat megadni neki

A szint adatai egy saját *Level* absztrakt osztályt megvalósító osztályban pihennek. Minden *Level* eltárolja a megjelenítendő térkép azonosító *String*-jét, illetve az irányítható karakter és egyéb entitások pozícióját.

A *World* jó vezérlő komponenshez híven tartalmaz a *GameHud* felől érkező felhasználói bemeneti eseményeket, az *EntityManager*-ben szereplő *Character* példányának továbbító függvényeket. Az entitásokat képező osztályok működéséről később lesz szó (az 5.4.6 fejezetben).

A legfontosabb metódusa a rajzolást és frissítést delegáló render függvénye.

```
public void render(SpriteBatch sb, OrthoCamera camera){
    //modellt (azaz az entitásokat) frissítő függvény
    update(camera);
    //térkép hátterének kirajzolása
    sb.end();
    mapContainer.renderBack(camera);
    sb.begin();
    //entitások kirajzolása
    entityManager.render(sb,stateTime);
    //entitások előtt lévő térképelemek (térkép előtere)
        //kirajzolása
    sb.end();
    mapContainer.renderFront(camera);
```

```
        sb.begin();
        //kamera frissítése
        camera.update(entityManager.getCharacter.getPosX(),
                entityManager.getCharacter.getPosY());
        //SpriteBatch méretek frissítése
        sb.setProjectionMatrix(camera.combined);
    }
```

Az update() metódus megnöveli a *stateTime*-ot az előző kirajzolás óta eltelt idővel, és továbbítja azt az *EntityManager*-nek, amely frissíti a tagjait.

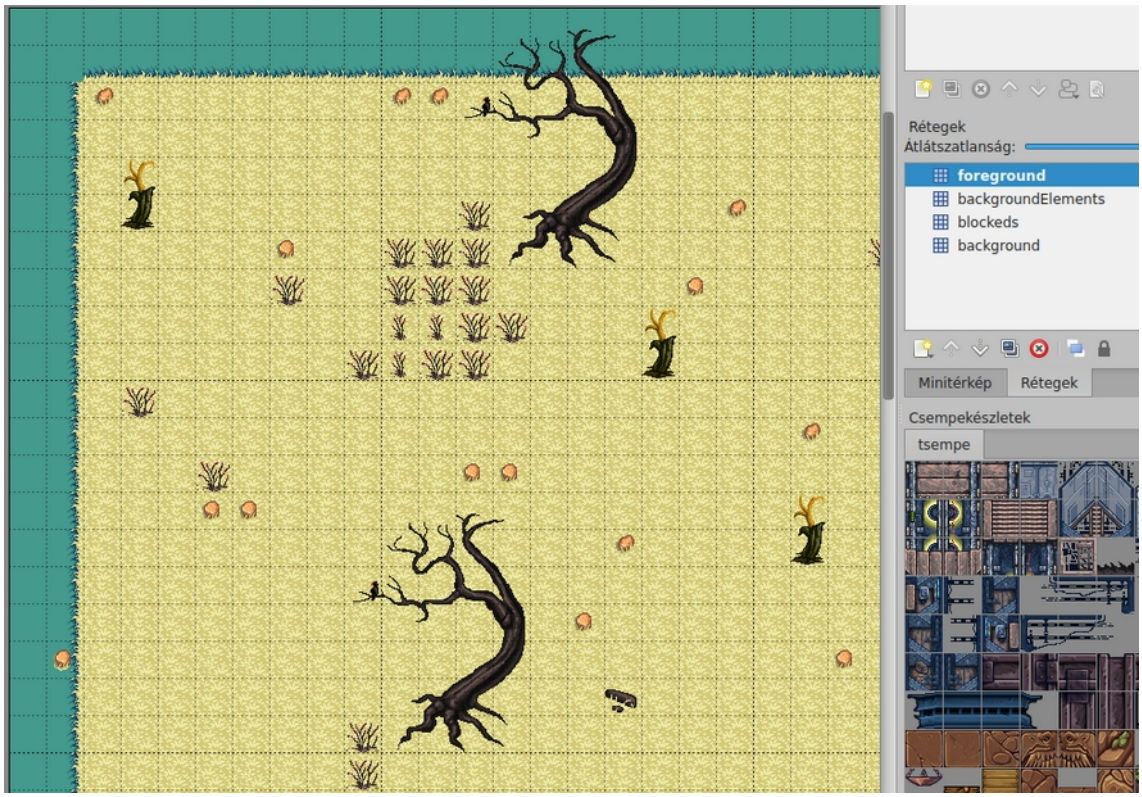
A térkép hátterének és előterének szétválasztása azért szükséges, mert a kirajzolt elemek sorrend szerint egymásra kerülnek rá. A játéktérnek vannak elemei, ami a térhatás érdekében az entitások előtt szerepelnek. Ilyen például egy fa, amelynek törzse és ágai mögé kerülhet a felhasználó karaktere.

A *SpriteBatch* zárása és nyitása azért van jelen a térképkirajzolás körül, mert ahogy a következő pontban be lesz mutatva, annak a kirajzolásához külön *SpriteBatch* lesz használva.

A kamera frissítése pedig a karakter elmozdulása függvényében történik, mivel annak kell a képernyő közepén szerepelnie, így a kamera középpontja oda lesz igazítva.

5.4.4 MapContainer és térképkészítés

Mielőtt a térképet kezelő osztályt megismernénk, érdemes egy pillantást vetni magára a térképre.



16. Ábra: A térkép megrajzolt felülete, oldalt térkép rétegek, illetve rajzoláshoz szükséges csempekészlet

A térkép a már bemutatásra került Tiled programban készült. A használt csempekészletet egy ingyenesen felhasználható oldalról lett letöltve [20].

Alapvetően négy rétegből áll, amely a megfelelő sorrendű megjelenítés lehetőségének megteremtésében, illetve az entitásokra hatást gyakorló csempék szétválasztásában játszanak szerepet:

- A **background** azokat a csempéket tartalmazza, amely minden más mögött jelennek meg. Ilyen az ábrán is látható homok
- A **blockeds** réteg tartalmazza azokat a csempéket, melyeknek van entitásokra gyakorolt hatása. Példa rá a fa töve vagy a víz, amiken nem lehet átmenni. A background előtt helyezkedik el.
- A **backgroundElements** az a réteg, ami a background és blockeds réteg előtt, de az entitások mögött vannak. Funkcionalitása a vizuális megjelenés szebbé varázslása. Ezen szerepelnek például a kavicsok és fűcsomók.
- A **foreground** az, ami mind a többi réteg, mind az entitások előtt helyezkedik el. Többek között faágak és bokrok tartoznak hozzá.

A programban, ahogy már a tervezésben is utalva lett rá, egy saját készítésű *MapContainer* osztály kezeli mindezek elérését és renderelését. Biztosítja azt a *World* számára, hogy ha megkapja egy térképnek az azonosítóját, elkéri azt a *ResourceManager*-től, majd külön referenciát készít annak rétegeire, hogy azokat megfelelő sorrendben tudja megjeleníteni, illetve hogy az entitások mozgásához fontos blockeds réteget biztosítani tudja azok számára.

Megjelenítéshez a natív *OrthogonalTiledMapRenderer* osztályt használja, amely képes a rétegeket külön-külön is kirajzolni. Biztosítani kell számára viszont egy külön *SpriteBatch* osztályt, ami különösen hasznos a sok ugyanolyan négyzetet tartalmazó térképek erőforrás-takarékos megjelenítéséhez. Mindezek mellett minden kirajzoló függvény meghívásánál szüksége lesz az *OrthoCamera* osztályra, mivel tudnia kell azt, hogy a teljes térkép melyik szeletét készítse el a képernyőre, és ezáltal ne foglalkozzon fölöslegesen azokkal a részletekkel, amiket a felhasználó amúgy sem látna.

5.4.5 Entitásmenedzser és animációk

Ahogy már ezelőtt több helyen említve lett, az *EntityManager* osztály tárolja és kezeli a játékban szereplő entitásokat. Felépítése és működése megegyezik a már bemutatott többi menedzser osztályéval. Nagyon röviden minden *World* felől érkező függvényhívást továbbít a benne tárolt entitások fele (például joystick bemeneti esemény vagy a megjelenítés kérése), amik azt megvalósítják, illetve a szintek betöltésénél a megadott koordinátákon példányosítja azokat.

Mielőtt az entitások lennének bemutatva, érdemes megismerni az azok animációjának tárolását és kirajzolását végző *AnimationWrapper* csomagoló osztály, amely natív *Animation* osztály példányokat kezel. Az animációk számára a már bemutatott képkocka-alapú megvalósítását választottam. Ehhez saját, szerény művészi képességgel elkészített rajzokat használ fel.

17. Ábra: A Warrior osztály textúra atlasza. A képen a képkockák nem futási sorrendben helyezkednek el.

Létrehozásánál egy textúra atlaszt kap meg, melyből szétválasztja és eltárolja a megadott textúra régiókat. Háromféle textúra régiót különböztet meg enum állapotok alapján: mozgás, állás támadás. A textúra atlasz és régiói a 3.4-es pontban bemutatott GDX Packer program segítségével lettek elkészítve.

Ennek az osztálynak további feladata a megjelenítés. Ilyenkor az eltárolt animációnak meghatározza a következő képkockáját a *World*-ből kapott játékidő alapján, beállítja egy logikai igennel, hogy továbbra is ciklikus maradjon a képkockák lejátszása, és az x és y koordinátában kirajzolja azt a beállított szélességgel.

```
//állás animáció megjelenítése
sb.draw((TextureRegion) getAnimationStand().getKeyFrame(
    stateTime, true), x, y, textureWidth, textureHeight);
```

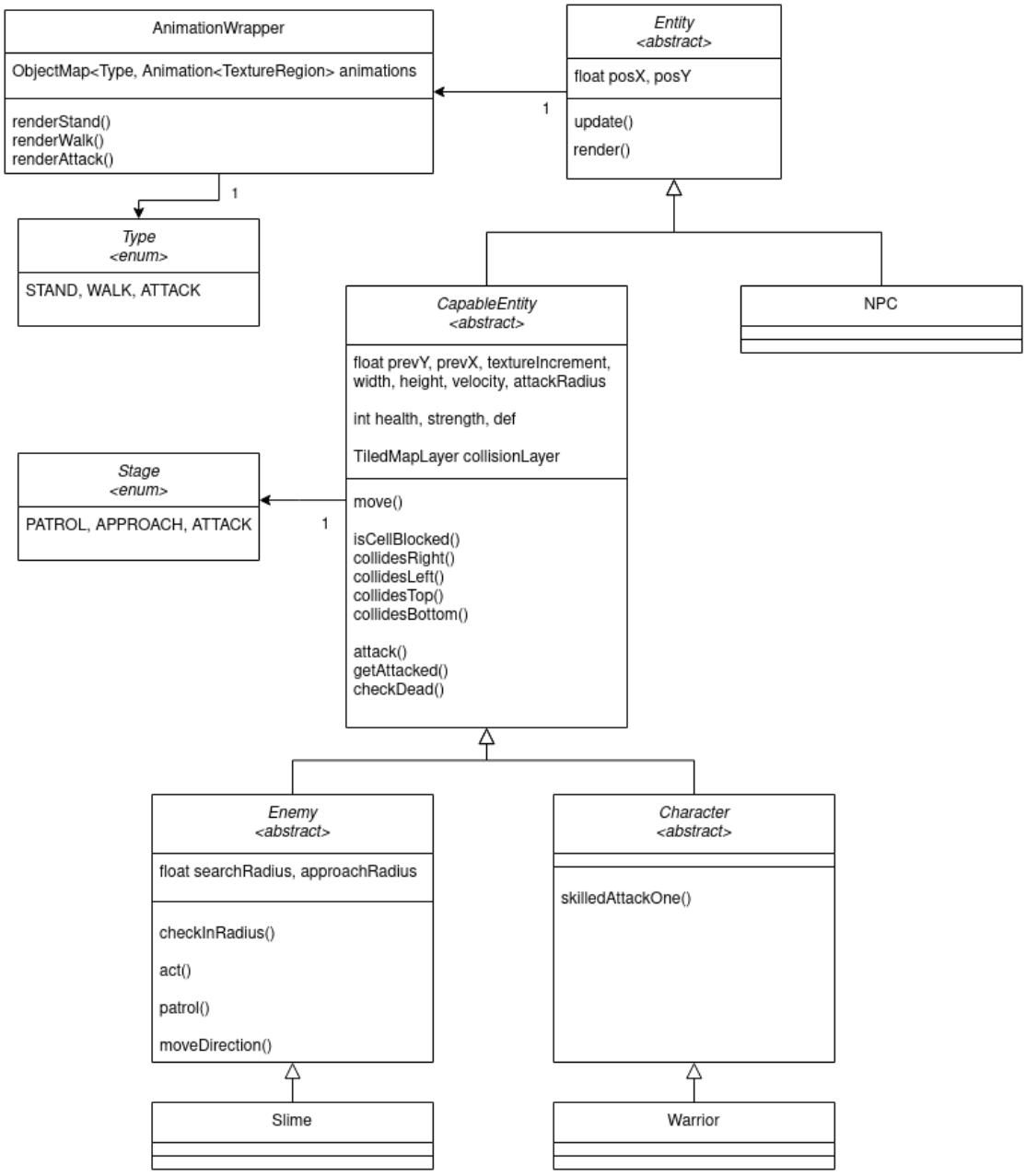
Egy fontos nehézség fog felmerülni a mozgás megjelenítése esetében. A program alapból a mozgásanimáció csak egyik irányba néző megvalósítását kapja meg, és amikor ezzel ellentétesen mozog az entitás, akkor annak az animációnak a képkockáit meg kell fordítani, amit biztosít is egy beépített flip() metódus. A veszélye ennek az,

hogy minden azonos típusú entitás ugyanazokra a képkockákra tárol referenciát az *AnimationWrapper*-en keresztül, így ha az egyik megfordítja azt, a többinek is ellenkező irányban lesz kirajzolva. Ez úgy lett megoldva végül, hogy a rajzolást végző *SpriteBatch* tükrözve kapja meg képkocka adatait, azaz annak a fordított irányba nézése esetén a kezdő kirajzolási pont (a textúra bal alsó sarka) el van tolva a textúra szélességének mértékével, emellett pedig a szélessége negatív értékként van megadva.

```
sb.draw((TextureRegion)getAnimationWalk().GetKeyFrame(
    stateTime, true), x + textureWidth, y, -textureWidth,
    textureHeight);
```

5.4.6 Entitások és mozgásuk

Mindezek ismeretében következzenek az entitások és azok felépítése.



18. Ábra: Osztálydiagram részlet az EntityManager által kezelt entitásokról, és azok viszonyáról

Minden entitás a saját készítésű *Entity* absztrakt osztályból származik le. Ez az adott entitás pozícióját és az animációt kezelő *AnimationWrapper* osztályt tárolja.

Ebből a közvetlenül leszármazó *NPC* osztály, amely a semleges nem játékos karaktereket testesíti meg, egyelőre csak a vizuális megjelenést kapja feladatként, egyéb felelősséggel nem rendelkezik.

Ennél érdekesebb a *CapableEntity* absztrakt osztály, amely az összes mozogni és harcra képes osztálynak a szülője. Tárolja a harchoz szükséges tulajdonságpontokat, és kezeli a mozgást

A mozgás alapvetően úgy működik, hogy a példány kap egy mozgásirányt mutató irányvektort, amelyet az entitás eltárolt sebességével megszorozva a pozíció koordinátáihoz adja.

```
//új x koordináta meghatározása
posX = posX + dirVector.x * velocity;
```

Fontos az, hogy a pozíciót tároló változó float típusú legyen, mivel akkor sokkal folyékonyabbnak látszódik a mozgás, mint egész szám esetében, ahol ugyanez lépcsőzetesnek tűnne.

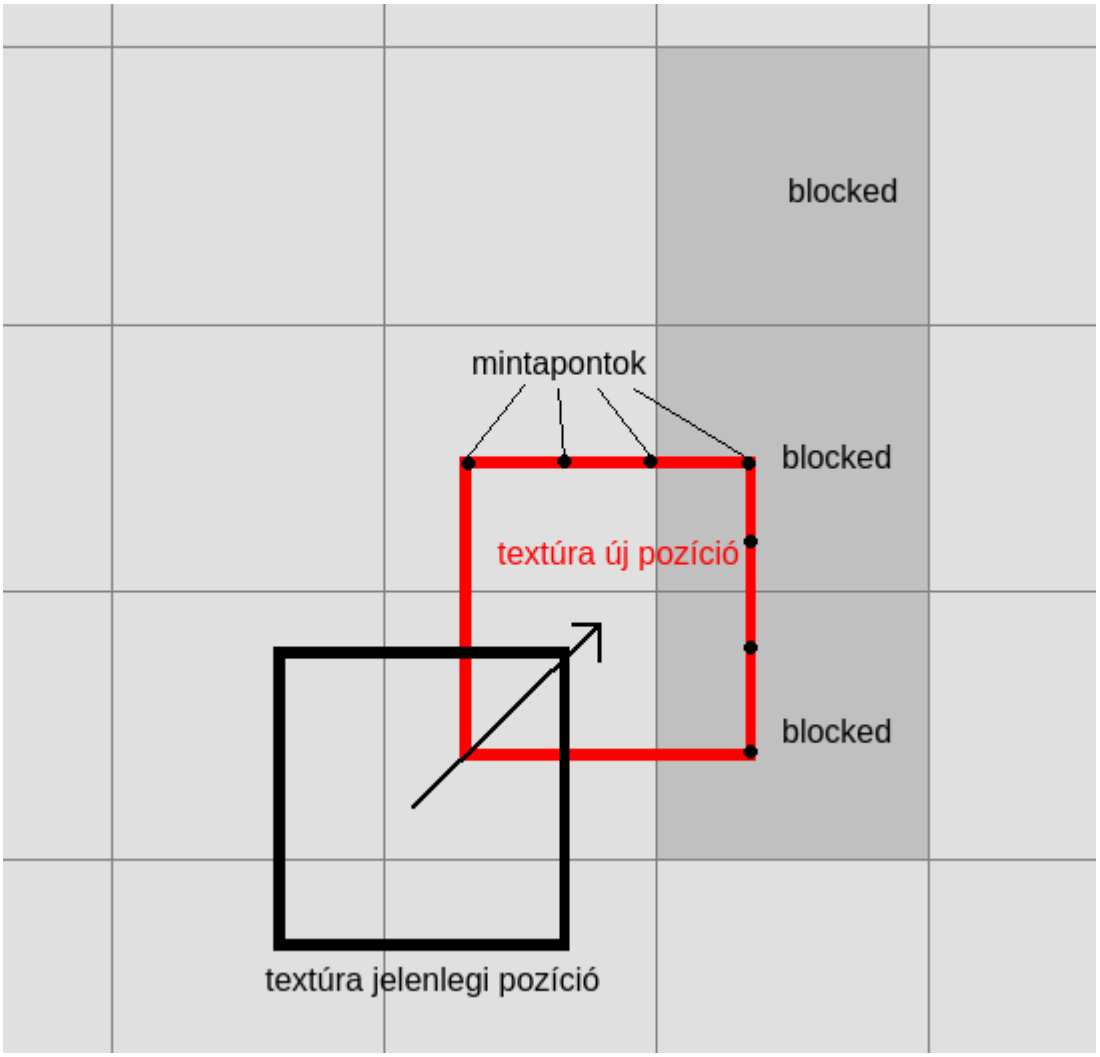
A bonyodalom ott fog kezdődni, hogy a térkép egyes elemein nem mehetnek keresztül az entitások. Erre kétféle megoldási lehetőséget tudok felvázolni.

Az egyik a Tiled program objektumréteg funkcióját kihasználó módszer. Ebben az esetben térképkészítésnél minden olyan pozícióba, ahol nem jelenhet meg az entitás, egy alakzat objektum kerül. Ezeket az *AnimationWrapper* példánya lekérdezi és eltárolja. Az entitás, a mozgás esetében, minden alkalommal megvizsgálja az összes alakzatot, hogy azokkal átfedésben van-e annak textúrája. Hátulütője ennek a rendszernek az, hogy létezhetnek olyan térképek, ahol nagyon sok ilyen objektum és számos entitás létezik. Mivel minden egyes entitásnak végig kell iterálnia az összes alakzaton, hogy pozíciókat hasonlítsanak össze, könnyen nagy erőforrás igényűvé tud válni. Mindemellett előfordulhat az is, hogy a térkép megrajzolásának feladata időigényesebbé válik, illetve könnyeben előfordulhatnak benne figyelmetlenségből elkövetett hibák.

A másik módszert, ami végül alkalmazva is lett, megvalósításának logikájában egy fokkal bonyolultabb, de az előzőnek a hibáit kiküszöböli. Ebben az esetben ugyanúgy a Tiled program egyik képessége lesz kihasználva, mégpedig az, amivel egyes csempéknek be tud állítani jellemzőket. Ahogy a térképrajzolásnál is említve lett (5.4.4-es bekezdésben), ezek a csempék külön rétegben lettek eltárolva, amit a

MapContainer külön tartalmaz is. Ezen réteg referenciáját továbbítja az entitások számára.

A módszer elméleti háttere az, hogy amikor az entitás mozog, megvizsgálja az abban az irányban elhelyezkedő csempéket, hogy tartalmazzák-e a blokkoló jellemzőt. Ennek több kihívása is lesz.



19. Ábra: Elméleti ábrázolása a mintapontos ütközés vizsgálatnak, ahol a karakter épp egy falnak ütközne. A jelölt mintapontokat vizsgálja meg a program, hogy blokkolt csempén tartózkodnak-e. (Megjegyzés: a karakter ennél jelentősen kisebb távolságokat mozog, a textúrája ennél közelebb fog megállni a falhoz)

Az entitás dinamikusan mozog, így egyszerre több csempén is tud állni, ezért nem csak egyetlen csempének a szomszédjait kell vizsgálni, hanem potenciálisan

többnek is. Ennek megoldása az lesz, hogy amelyik irányba éppen halad az x és y tengelyen, ott a leendő pozíciójában, az entitás textúrájának ugyanazon irányban levő oldalán mintapontokat kell felvenni. Ezen mintapontoknál meg kell határozni, hogy melyik csempén vannak, és azokat kell megvizsgálni, hogy blokkolt csempék-e.

A mozgás megvalósítása a következő.

```
//a moveXPercent és moveYPercent az irányvektor
koordinátái
public void move(float moveXPercent, float moveYPercent)
{
    //kiindulási pozíciók elmentése az esetleges
    //visszatéréshez
    float oldX = posX, oldY = posY;
    boolean collisionX = false, collisionY = false;

    //mozgás végrehajtása
    posX = posX + moveXPercent * velocity;
    posY = posY + moveYPercent * velocity

    //x tengelyen történő mozgás vizsgálata
    if(moveXPercent < 0)
        collisionX = collidesLeft();
    else if(moveXPercent > 0)
        collisionX = collidesRight();

    //reakció az ütközésre
    if(collisionX) {
        posX = oldX;
    }

    //y tengelyen történő mozgás vizsgálata
    if(moveYPercent < 0)
        collisionY = collidesBottom();
    else if(moveYPercent > 0)
        collisionY = collidesTop();

    ///reakció ütközésre
    if(collisionY) {
        posY = oldY;
    }
}
```

Az ütközés detektálásához először szükséges lesz a mintapontok meghatározásánál figyelembe venni azt, hogy az entitások textúrája eltérő méretű lehet. Ha nincsenek elég sűrűen elhelyezve, előfordulhat, hogy átcsusszan kettő között egy blokkolt csempe. Ennek elkerülése érdekében az ezek közötti távolság a csempeméret függvényében van meghatározva.

```
//a mintapontok távolsága a textúra és
csempeszélesség //közül a kisebbnek lesz a fele
```

```
increment = collisionLayer.getTileWidth();
increment = this.width < increment ? this.width / 2 :
    increment / 2;
```

Példaként a mintapontok létrehozása jobb oldali irányban a következő.

```
public boolean collidesRight() {
    for(float step = 0; step <= height; step +=
increment) {
        //csempe vizsgálata a mintapont alatt. A posX és
posY        //az entitás pozíciója, tehát a bal alsó
pontjának a         //koordinátái
        if (isCellBlocked(posX + width, posY + step))
            return true;
    }
f   return false;
}
```

Egy mintapont csempéjének megkeresése, és annak vizsgálata.

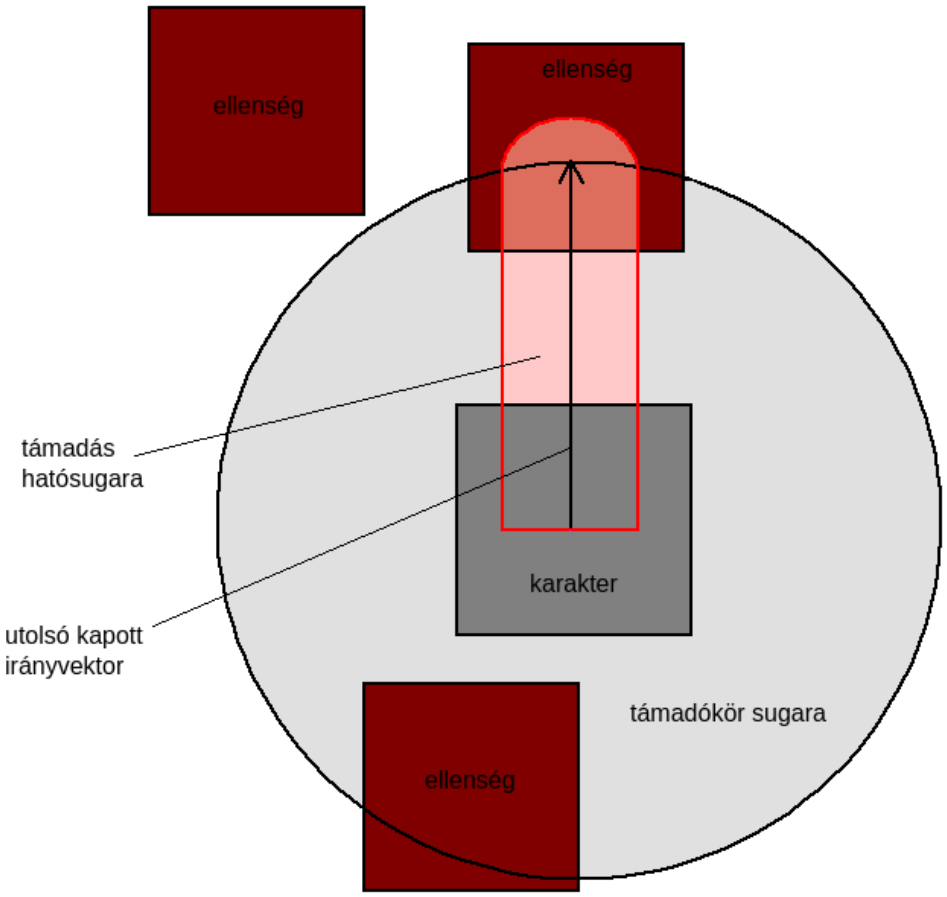
```
//x és y a mintapont koordinátája
private boolean isCellBlocked(float x, float y) {
    //csempe meghatározása koordináta alapján
    TiledMapTileLayer.Cell cell = collisionLayer.getCell(
                (int)                   (x                /
collisionLayer.getTileWidth()),                 (int)
(y / collisionLayer.getTileHeight()));
    //igazat   ad   vissza,   ha   a   csempe   létezik,   és
tartalmazza     //a blocked tulajdonságot
    return cell != null && cell.getTile() != null &&
            cell.getTile().getProperties().
                containsKey("blocked");
}
```

Mindezeken felül a *CapableEntity*-nek még egy felelőssége az, hogy a megfelelő animációt rajzoltassa ki az *AnimationWrapper*-rel. Ehhez az előző `render()` futási ciklusban meghatározott pozícióját hasonlítja össze a jelenlegivel. Ha a kettő megegyezik, akkor egy helyben áll, ha az előző kisebb volt, mint a mostani, akkor jobbra sétál, ellenkező esetben balra.

```
//a prevX az előző, a posX a jelenlegi koordináta
if ( prevX == posX)
    animationWrapper.renderStand(sb,stateTime,posX,
posY);
else if( prevX < posX)
    animationWrapper.renderWalk(sb,stateTime,posX, posY);
else if( prevX > posX)
    animationWrapper.renderWalkLeft(sb,stateTime,posX,
posY);
```

5.4.7 Karakterek, ellenségek és harcrendszer

A felhasználható által irányított karaktert megtestesítő *Character* absztrakt osztály közvetlenül a *CapableEntity*-ből származik le. Szülőjét a támadás típusokat megvalósító metódusokkal egészíti ki. Egyelőre kétféle támadásra van lehetőség, amelyek a különböző karakter fajtáknál eltérhetnek.



20. Ábra: A Warrior alaptámadásának koncepciója. Jelen esetben a három ellenség közül egyedül a karakter felettit találja el, mert az van egyedül a támadás hatósugarában.

A *Warrior* osztály az egyetlen működő gyermekosztálya egyelőre. Ennek a támadási képességei egyszerűek. Az alaptámadása egy adott sugarú körön belül működik, azon belül is a legutolsó megkapott irányvektor irányában. Végigiterál az *EntityManager*-től kapott ellenségeket tartalmazó listán, és amelyik textúrájának a középpontja a vektor megadott távolságában van, sebzést kap. A sebzés mértéke a

karakter erő tulajdonságával megegyezik. Másik képessége egy adott sugarú körben okoz sebzést az összes ott tartózkodó ellenségeknek, viszont a sebzés mértéke fele az erőpontjának.

Amikor sebzést kap a karakter, abból először levonja a védekezés tulajdonságának az értékét, és utána vonja le azt életerőpontjaiból. Abban az esetben ha az elfogy, értesíti az őt tároló *EntityMannager*-t, ami azt továbbadja a *World* oszálynak.

Az absztrakt *Enemy* osztály, ahogyan neve is utal rá, az ellenségek típusok szülőosztálya. Alapvetően három enum típusú állapota van, a járőrözés, a rohamozás és támadás. Az `update()` függvényének meghívásánál vizsgálja felül, hogy melyikben kell az adott pillanatban lennie. Ezek eldöntéséhez három kör sugarának az értékével veti össze a felhasználó karakterének a pozícióját. Ha az a kereső kör sugarán kívül van, akkor folytatja a járőrözést, ha a rohamozási kör sugarán belül van, rohamozás fázisba lép, ha pedig támadási kör sugarán belül van, akkor támadásba lendül. Az *Enemy* az `act()` metódusa meghívásánál az épp aktuális állapot alapján fog cselekedni.

Egy gyermekosztálya van neki is, a *Slime*. Ő egy nagyon egyszerű ellenség, inkább példa célból jött létre. Járőrözésnél egy helyben üldögél. Rohamozásnál egyenesen a karakter irányába halad (egy okosabb ellenség nem közvetlenül közelítené meg a karaktert, hanem oldalirányból, hogy annak alaptámadása nagyobb eséllyel célt téveszthessen). Támadása sem különleges, ha a karakter az ő támadási sugarába kerül, akkor megsebzi azt a saját erő tulajdonságának értékével. Amikor sebzést kap, ugyanúgy viselkedik, mint a karakter, azaz a saját védekezési értékét levonja az okozott sebzésből.

Ha egy *Enemy*-nek elfogy az életpontja, jelez az *EntityManager*-nek, ami azt nem azonnal fogja megsemmisíteni, hanem csak kiszedi az aktív ellenségek listájából. Ennek az a célja, hogy a játékmenet végén a program egyszerre hívja meg mindegyikre a garbage collector-t, mivel mobiltelefonon annak meghívásakor mikroakadásokat okozhat használata, és az el tudna venni a játékélményből, ha az menet közben történne.

Amint mindegyik ellenségnek elfogyott az életpontja, és kiürül az aktív ellenségek listája, a *World* átadja a *Character*-t a következő *UpgradeScreen*-nek, amit egyúttal be is állít az aktuális képernyőnek a *ScreenManager*-en keresztül.

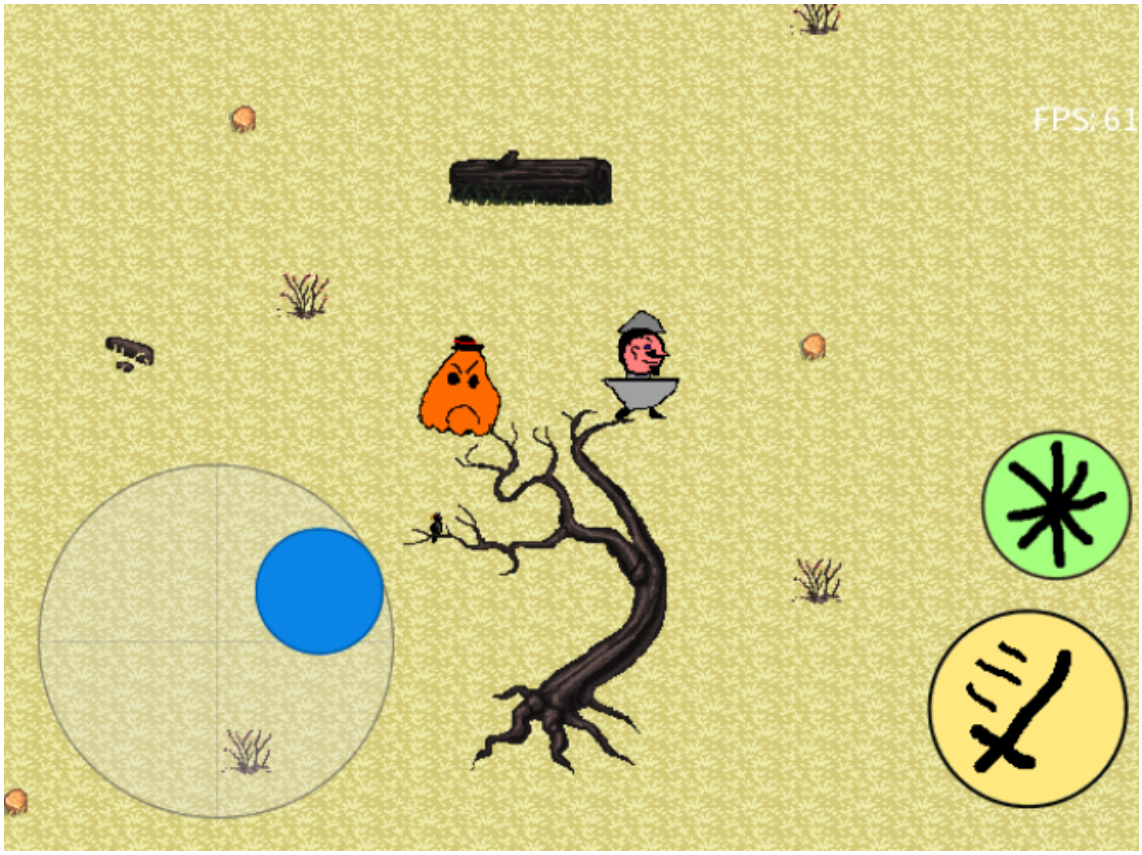
5.5 UpgradeScreen

Az *UpgradeScreen* a karakter tulajdonságai fejlesztésének biztosít felületet. Megjeleníti azoknak az értékét, és mellettük lévő gomb megnyomásával lehet azt növelni. Ezen tulajdonságok a sebesség, életerőpont, erő és védekezési érték. Amint a megfelelő kiválasztásra került, a karakternek azt a változóját módosítja, majd átadja a *GameScreen*-nek, amely a következő szinttel együtt betölti azt a *World* osztályába. Ezzel elindul a következő szint, és bezárul a játék futásának köre.

6 Önálló munka értékelése és továbbfejlesztési lehetőségek

Összességében sikerült megalkotni nagy részben a célként meghatározott játékot, mely működési és megjelenítési folyamatai elkészültek, így már csak a tartalommal való feltöltést várja, bár kisebb hiányosságok szerepelnek még benne, és egészen messze áll még a piackutatásnál megvizsgált játékok minőségétől.

A játékmenet egyelőre viszonylag egyszerű, viszont az egy pálya, egyfajta ellenség és választható karakter ellenére kellően szórakoztatóra sikeredett, emellett ezeknek számát nagyon könnyen lehet növelni a felépítésnek és a menedzserosztályoknak köszönhetően. A folyamatos fejlődés is kellően jutalmaz, viszont egyelőre még hiányzik a játék nehezedésének élménye. Ez később megvalósítható lenne fejlődő vagy alapból erősebb ellenféltípusokkal, illetve azok számának a növelésével.



21. Ábra: A karakter menekül az őt üldöző ellenség elől.

Legfőbb értéke a tervezésnek, hogy akár olyan személy is tudja a bővíthető tartalmakat hozzáadni, aki nem ért feltétlen a libGDX-hez, mivel a kirajzolással foglalkozó osztályok mind ki lettek szervezve csomagoló osztályokba, amiket a már létrehozott szülő- és az előbb említett menedzserosztályok kezelnek.

Felróható gyengesége a programnak a nem igazán ízléses megjelenése az indító-, karakterválasztó- és fejlesztőképernyőnek, viszont ezek is könnyen megváltoztathatóak, illetve a mögöttük álló logika mindezektől függetlenül megfelelően működik a próba futtatások alapján. Emellett másik hiányossága, hogy a bemutatott libGDX bővítményeket nem sikerült mélyrehatóan megismerni és felhasználni, pedig sok egyéb segítséget tudnának nyújtani a feladat megvalósításában.

A legidőigényesebb részt a libGDX megjelenítési folyamatának és osztályainak megismerése és használata jelentette, ami végül a már említett, azok külön osztályba szervezését is eredményezte. A játéklogika talán legnehezebb, de biztosan legizgalmasabb része, az entitások mozgásánál az egyes térképelemekkel való ütközés megtervezése és megvalósítása volt, ami végül teljesen megfelelően viselkedett. Ugyanez felhasználható lesz később olyan tulajdonságú térképelemek használatához, ami nem feltétlen blokkolja a karaktert, de egyéb más módon befolyásolhatja azt (például lassíthatja vagy sebezheti).

Ami komolyabb technikai fejlesztési lehetőség lenne, egy zsákmányrendszer, ami véletlenszerű tárgyat biztosítana egyes ellenségek legyőzésénél, amiket akár hordozni is lehetne, ezzel befolyásolva a karakter tulajdonságát és akár megjelenését is. Szükség lenne még narratívával felruházni a játékot, és emellé küldetésrendszert is biztosítani, ami célt adna a játékosnak az egyelőre indokolatlan harc mellé.

7 Irodalomjegyzék:

[1] 100 főre jutó mobiltelefon előfizetések száma, hozzáférés dátuma: 2022.05.02. https://data.worldbank.org/indicator/IT.CEL.SETS.P2end=2020&start=2000&view=chart

[2] Napi átlagos telefonhasználat, Statista statisztika adatbázis hozzáférés dátuma: 2022.05.02. https://www.statista.com/statistics/1224510/time-spent-per-day-on-smartphone-us/

[3] NewZoo 2021 videójáték bevételek platformok közötti megoszlása, hozzáférés dátuma: 2022.04.10. https://newzoo.com/key-numbers/

[4] Techopedia, IT Education Website, hozzáférés dátuma: 2022.05.20. https://www.techopedia.com/definition/27052/role-playing-game-rpg

[5] PocketGamer magazin újságcikk, Updated: Top 10 highest-grossing mobile games of all time hozzáférés dátuma: 2022.05.20. https://www.pocketgamer.biz/feature/77017/top-10-highest-grossing-mobile-games-of-all-time/

[6] Google Play toplista, hozzáférés dátuma: 2022.05.21. https://play.google.com/store/apps/collection/cluster?clp=0g4cChoKFHRvcHNlbGxpbmdfcGFpZF9HQU1FEAcYAw%3D%3D:S:ANO1ljLtt38&gsr=Ch_SDhwKGgoUdG9wc2VsbGluZ19wYWlkX0dBTUUQBxgD:S:ANO1ljJCqyI

[7] Statista statisztika adatbázis, Number of Pokémon GO app downloads worldwide from 3rd quarter 2016 to 4th quarter 2021 , hozzáférés dátuma: 2022.05.21. https://www.statista.com/statistics/641690/pokemon-go-number-of-downloads-worldwide/

[8] NewZoo átfogó RPG piackutatása 2020-ból, Pangle x Newzoo: Mobile Game Genre Report Role-Playing Games Comparing & Contrasting Eastern and Western Markets, hozzáférés dátuma: 2022.04.30. https://newzoo.com/insights/trend-reports/rpg-games-market-across-east-and-west/

[9] Facebook piackutatás telefonos játékműfajok alapján, Facebook-Gaming Genre and Great Games report, 2021, hozzáférés dátuma: 2022.04.30. https://www.facebook.com/fbgaminghome/marketers/build-great-games/game-design-hub/genre-great-games-report

[10] libGDX dokumentáció, The application framework, hozzáférés dátuma: 2022.05.22. https://libgdx.com/wiki/app/the-application-framework

[11] libGDX keretrendszert bemutató könyv, David Saltares, Alberto Cejas Sánchez - Libgdx Cross platform Game Development Cookbook, 2014,

[12] Tiled felhasználói leírás, hozzáférés dátuma: 2022.06.01. https://doc.mapeditor.org/en/stable/manual/introduction/#about-tiled

[13] Pinta szerkesztőprogram, hozzáférés dátuma: 2022.06.01. https://www.pinta-project.com/

[14] GDX Texture Packer github oldala, hozzáférés dátuma: 2022.05.25. https://github.com/crashinvaders/gdx-texture-packer-gui

[15] Otter tech libgdx fejlesztő blog oldal, hozzáférés dátuma: 2022.06.04. https://otter.tech/an-mvc-guide-for-libgdx/

[16] Menedzser tervezési minta leírása, hozzáférés dátuma: 2022.06.05, https://www.eventhelix.com/design-patterns/manager/

[17] Csomagoló tervezési minta, hozzáférés dátuma: 2022.06.05 https://refactoring.guru/design-patterns/decorator

[18] Ortografikus kamerakezelés, hozzáférés dátuma: 2022.05.20. https://www.geofx.com/graphics/nehe-three-js/lessons17-24/lesson21/lesson21.html

[19] LibGDX touchpad dokumentáció, hozzáférés dátuma: 2022.05.22. https://libgdx.badlogicgames.com/ci/nightlies/docs/api/com/badlogic/gdx/scenes/scene2d/ui/Touchpad.html

[20] Ingyen letölthet csempekészletek, hozzáférés dátuma: 2022.05.19. https://opengameart.org/content/lots-of-free-2d-tiles-and-sprites-by-hyptosis