Opinnäytetyö (AMK)

Tietojenkäsittelyn koulutusohjelma

Tietojärjestelmät

2014

Petrus Lehto

ANDROID-MOBIILIPELIN TOTEUTUS GAMEMAKER-PELIMOOTTORILLA

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely | Yrityksen Tietojärjestelmät

2014 | 40

Anne Jumppanen

Petrus Lehto

ANDROID-MOBIILIPELIN TOTEUTUS GAMEMAKER-PELIMOOTTORILLA

Opinnäytetyön tavoitteena oli selvittää, mitä vaaditaan toimivan Android-mobiilipelin toteuttamiseen pelimoottoria apuna käyttäen. Ohjelmoitu peli perustuu teoriaosuudessa esitettyihin huomioihin hyvistä mobiilipeleistä.

Teoriaosuudessa pohditaan kirjallisuuteen perehtyen hyvän mobiilipelin ominaisuuksia ja tarkastellaan pelin ohjelmointiin käytettyä GameMaker-ohjelmaa. Lisäksi tarkastellaan eroavaisuuksia tietokonepelien ja mobiilipelien välillä. Teoriaosuus auttaa ymmärtämään, mitä edellytyksiä menestyksekkään mobiilipelin tekeminen vaatii.

Opinnäytetyön tuloksena syntyi mobiilipeli Android-käyttöjärjestelmälle. Pelissä väistellään eteen tulevia esteitä säätelemällä hahmon hypyn korkeutta. Peli on toimiva ja täyttää hyvin mobiilipelille asetettuja vaatimuksia pelattavuuden sekä kiinnostavuuden kannalta.

ASIASANAT:

Android, mobiilipelit, peliohjelmointi

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Business Information Technology | Business Information Systems

2014 | 40

Anne Jumppanen

Petrus Lehto

DEVELOPING AN ANDROID MOBILE GAME WITH GAMEMAKER SOFTWARE

The aim of this thesis was to discover what is required to create a functional Android mobile game and, after that, to program such a game.

The theoretical part of this thesis includes a literature review of the characteristics of a good mobile game and introduces the GameMaker program which is used for the game programming. It also examines the differences between computer games and mobile games. The theoretical part helps the reader to understand what is required to make a successful mobile game.

The game that was programmed for this thesis was a functional Android mobile game. The game is a simple platform game where the player is dodging obstacles by controlling the jump height of the main character. The programmed mobile game fulfills the observations made in the theoretical part. It is working well and meets the basic requirements of an addicting mobile game. This means that the game has a proper learning curve, achievements, enough of content and entertaining graphics.

KEYWORDS:

Android, Mobile games, Game programming

SISÄLTÖ

1 JOHDANTO **6**

2 MOBIILIPELIN EDELLYTYKSET JA GAMEMAKER **8**

2.1 Mobiilipelaamisen historia 8

2.2 Hyvän mobiilipelin edellytykset 8

2.3 Mobiilipelien tulokeinot 10

2.4 Mobiililaitteiden rajoitukset 11

2.5 Pelimoottorit 13

2.6 GameMaker 14

3 ANDROID-MOBIILIPELIN TOTEUTUS **17**

3.1 Törmäystunnistus 17

3.2 Kengurun hyppy 19

3.3 Pelin häviäminen 21

3.4 Hahmon kuvan valinta 22

3.5 Muistin käytön optimointi 25

3.6 Pelin tallennus 28

3.6.1 Pelin lataaminen avattaessa 29

3.6.2 Kolikoiden ja ennätysten tallennus 30

3.7 Kenttävalikko 32

4 YHTEENVETO **38**

LÄHTEET **40**

KUVAT

Kuva 1. GameMakerin käyttöliittymä 15
Kuva 2. Koodi esimerkki 16
Kuva 3. Sprite-esimerkki 24
Kuva 4. Kengurun animaatio 25
Kuva 5. Kenttänappulat 34
Kuva 6. Valmis valikko 37

1 JOHDANTO

Mobiilipelejä pelataan ympäri maailmaa koko ajan kasvavissa määrin. Yhä nuorempia voi nähdä kadulla älypuhelin tai tabletti kädessään. Mobiilipelit tuntuvatkin olevan jo hyvin arkinen asia monen elämässä eikä pelaamista enää luokitella vain perinteiseksi nörttien touhuksi. Tulevaisuus näyttää myös valoisalta mobiilipelien suosion kannalta.

Mobiilipelit ovat kätevä ajanviete lyhyille tauoille esimerkiksi bussissa matkustaessa tai ystävää odotellessa. Näiden pelien suosio perustuukin helppoon pelattavuuteen ja hyvin lyhyeen ajalliseen panostukseen. Usein minuutti on jo riittävä aika, jotta mobiilipeliä ei tarvitse jättää kesken, vaan esimerkiksi kenttä saadaan pelattua loppuun. Myös pelien suuri tarjonta nykymarkkinoilla takaa, että jokaiselle löytyy varmasti mieluista pelattavaa omalle bussimatkalleen.

Tässä opinnäytetyössä tavoiteenani on selvittää, mitä vaaditaan toimivan Android mobiilipelin luomiseen pelimoottorin avulla. Peli toteutetaan GameMaker ohjelmalla, jota olen itse harrastuspohjalta hieman harjoitellut jo ennen tämän opinnäytetyön kirjoittamista ja esimerkkipelin ohjelmointia.

Tarkoituksena olisi saada valmiiksi toimiva ja omalla tavallaan koukuttava mobiilipeli, joka soveltuu hyvin lyhytaikaisiin pelisessioihin. Opinnäytetyön aikana yritän saada valmiiksi mobiilipelin, joka pienellä lisähiomisella olisi mahdollista julkaista jopa Google Playhin ladattavaksi. Pelimekaniikkojen tulisi siis olla työn lopussa valmiita ja peliä pitäisi pystyä pelaamaan Android-käyttöjärjestelmällä varustetussa kosketusnäytöllisessä laitteessa.

Ohjelmoitavassa pelissä ideana on päästä kulkemaan hahmolla mahdollisimman pitkä matka ilman kuolemista. Pelissä hahmo liikkuu automaattisesti tasaista vauhtia kenttää oikealla ja pelaajan vastuulle jää esteiden yli hyppiminen näyttöä koskettamalla. Hypyn korkeutta on mahdollista kasvattaa painamalla näyttöä kauemmin. Kenttä vaikeutuu mitä pidemmälle

pelaaja pääsee. Pelin koukuttavuus piilee pelaajan halussa rikkoa oma kenttäennätys, joka tallennetaan muistiin.

Opinnäytetyö etenee siten, että aluksi tutustutaan mobiilipelaamisen historiaan lyhyesti ja pohditaan mobiilipelien menestyksen syitä. Myös mobiilipelien tulokeinot tarkastellaan pääpiirteittäin. Seuraavaksi selvitetään yleisesti hieman pelimoottoreiden tarkoitusta ja paneudutaan tarkemmin esimerkkipelin ohjelmoinnissa käytettävään GameMaker-pelimoottoriin. Tämän jälkeen siirrytään itse pelin ohjelmointiin ja tapoihin toteuttaa pelin eri ominaisuudet. Viimeiseksi tarkastellaan, miten pelin ohjelmointi onnistui ja minkälaisia ongelmia työssä kohdattiin.

2 MOBIILIPELIN EDELLYTYKSET JA GAMEMAKER

2.1 Mobiilipelaamisen historia

Mobiilipelaaminen sai alkunsa vuonna 1997, jolloin Nokia sisällytti ensimmäisenä matkapuhelinvalmistajana Snake-pelin eli matopelin yhteen matkapuhelinmalliinsa. Yritykset huomasivat matopelin suosion ja tästä innoittuneena alkoivat kehittää WAP-teknologiaa, joka mahdollistaisi pelidatan siirron servereiltä matkapuhelimeen (MGF 2015).

Suurin mullistus mobiilipelaamisessa nähtiin vuonna 2007 Applen iPhonen julkaisun yhteydessä. iPhone toi mukanaan App Store-sovelluksen, jossa kuka tahansa pystyi julkaisemaan omia ohjelmiaan ja pelejään muiden käyttäjien ladattavaksi. App Store antoi kehittäjille todella helpon tavan tavoittaa asiakkaita ilman suuria kustannuksia. App Storen menestyksen myötä Google julkaisi hetkeä myöhemmin Android-käyttöjärjestelmää käyttäville matkapuhelimille Android Marketin, jonka toimintaperiaate on täysin sama kuin App Storella (MGF 2015). Mobiilipelien markkinat ovat kasvaneet vauhdilla todella suuriksi ja alalta löytyy jo useita miljoonayrityksiä, kuten Rovio, SuperCell ja King.

2.2 Hyvän mobiilipelin edellytykset

Menestyvän mobiilipelin suunnittelu ei ole itsestäänselvyys ja kilpailua on nykyään markkinoilla todella paljon. Tämän vuoksi ennen pelin suunnittelua on suositeltavaa tarkastella hieman, mikä saa ihmiset palaamaan mobiilipelien pariin uudestaan ja uudestaan.

Toisin kuin suuren budjetin konsoli- tai pc-pelejä mobiilipelejä pelataan huomattavasti lyhyempiä aikoja kerralla. Tämä asettaa peleille tietyt vaatimukset, jotta käyttäjä saadaan aluksi innostumaan pelistä ja tämän jälkeen vielä palaamaan pelin pariin myöhemmin. Lyhyitä aikoja pelattaessa pelin

oppimiskäyrä ei saa alkaa liian korkealta, vaan uuden pelaajan pitäisi oppia pelaamaan peliä perustasolla hyvin nopeasti. Helposta alusta huolimatta pelin vaikeusastetta on hyvä nostaa tasaiseen tahtiin, sillä liian helposti läpäistävän pelin kiinnostus laskee usein nopeasti.

Mobiilipelit harvoin nousevat suosioon niiden hyvän juonen, realististen grafiikoiden tai mullistavien uusien pelimekaniikkojen vuoksi. Yksinkertaiset ja nopeasti pelattavat pelit, kuten Angry Birds, tuntuvat täyttävän mobiilipelaamiselle asetetut vaatimukset hyvinkin toimivasti (Chrome Coders 2013, 104). Tällaisissa peleissä pelaajan on mahdollista viedä peliä eteenpäin jopa alle minuutissa, joten suoritukseen ei tarvitse varata juuri mitään aikaa. Pitkien pelikertojen sijaan tavoitteena onkin saada käyttäjä käynnistämään peli mahdollisimman usein.

Kiinnostuksen ylläpitämiseksi pelaajalla pitää olla tavoitteita, joita on pelaamalla mahdollista saavuttaa (Chrome Coders 2013, 81). Yksinkertaisimmillaan saavutukseksi riittää esimerkiksi uusien kenttien avaaminen tai oman ennätyksen rikkominen. Joissain peleissä saavutuksia on mahdollista kerätä suorittamalla erilaisia haasteita, kuten pelaamalla peliä yhteensä kymmenen tuntia tai kirjautumalla peliin viitenä peräkkäisenä päivänä. Keinot, jolla pelistä saadaan koukuttava, riippuvat täysin pelin tyylistä ja toteutustavasta

Sosiaalinen media on viime vuosien aikana kasvanut räjähdysmäisesti muun muassa Facebookin ja Twitterin menestyksen myötä (Chrome Coders 2013, 93). Tämä on otettu huomioon myös mobiilipeleissä. Usein peleissä on esimerkiksi mahdollisuus jakaa tuloksia sosiaalisen median kautta muiden nähtäväksi tai pyytää kavereitaan liittymään mukaan peliin. Näin mobiilipeli saa ilmaista näkyvyyttä ja pelaajaa puolestaan voidaan palkita esimerkiksi pelin sisäisillä saavutuksilla.

2.3 Mobiilipelien tulokeinot

Kuten muidenkin pelien, on myös mobiilipelien tarkoituksena pyrkiä tuottamaan kehittäjälleen voittoa tehdystä työstä. Mobiilipeleille on tarjolla useita eri myyntistrategioita ja aina oikean valitseminen ei ole itsestäänselvyys. Pelinkehittäjän pitää löytää vaihtoehdoista omaan peliinsä sopivin tapa myydä tuotettaan asiakkaille, jotta hänen on mahdollista saada tuottoa työstään.

Perinteisin tapa myydä pelejä on aina ollut kertamaksun veloittaminen. Kertamaksussa asiakas maksaa pelin hinnan kerran ja saa tämän jälkeen pelin täysin käyttöönsä ilman lisämaksuja (Fields 2014, 139). Tällaisessa tavassa on tärkeää saada peli hinnoiteltua oikein, sillä liian suuri kertamaksu saattaa ajaa asiakkaat pois ja liian pieni puolestaan ei ole tuottavuudeltaan kannattava ratkaisu. Pelin hinta täytyy myös suhteuttaa pelattavan sisällön määrään sekä pelin laatuun. Pitkästä, huolella tehdystä pelistä voi pyytää selvästi enemmän rahaa, kuin parin tunnin pelikokemuksesta. Usein maksullisista peleistä löytyy demoversio, jossa käyttäjä pääsee testaamaan peliä ennen ostopäätöksen tekemistä.

Mikromaksut ovat viime vuosina nousseet yhdeksi suosituksi tavaksi rahoittaa mobiilipelien tuotantoa. Mikromaksullisissa peleissä peli itsessään on lähes aina maksuton, mutta peliin pystyy ostamaan pelin sisältä erilaisia apuja sen nopeuttamiseksi (Fields 2014, 146). Periaatteena tällaisissa peleissä on, että peli on mahdollista pelata läpi ilman mitään maksuja, mutta ilman maksuja aikaa ja vaivaa kuluu huomattavasti enemmän. Mikromaksut ovat yleensä pieniä muutaman kymmenen sentin tai parin euron ostoksia, joilla saa peliin helpotusta. Mikromaksuja maksamalla pelaaja saattaa käyttää huomattavasti suuremman määrän rahaa peliin kuin kertamaksulla. Vaikka osa pelaajista ei maksa pelistä mitään, tuovat he silti pelille näkyvyyttä ja suosiota.

Yksi tapa saada tuottavuutta on sallia mainosten näyttö pelin sisällä. Mainoksia käyttävät pelit ovat normaalisti ilmaisia ja pyrkivät saamaan suuren käyttäjämäärän pelin pariin (Fields 2014, 150). Mainoksista maksaminen voi tapahtua useilla eri tyyleillä. Mainoksista voidaan maksaa esimerkiksi

näyttökertojen, painallusten tai mainoksen kautta tulleiden asennusten mukaan. Mainosten tuotot ovat hyvin pieniä mainosta kohden ja tämän vuoksi mainoksilla saavutettu tuotto vaatii paljon käyttäjiä, joille niitä voidaan esittää. Mainoksien käytössä täytyy myös olla varovainen, sillä liian aggressiivinen mainosten esittäminen saattaa ärsyttää pelaajan pois pelin parista. Usein ilmaispeleistä on tarjolla myös mainokseton versio, jolloin peli ostetaan kertamaksulla.

2.4 Mobiililaitteiden rajoitukset

Mobiililaitteiden kehitys on viime vuosina ollut todella nopeaa ja parannuksia uusien mallien mukana tulee jatkuvasti. Tästä huolimatta älypuhelinten sekä tablettien tehot ovat huomattavasti perinteisiä tietokoneita pienemmät. Pelien suunnittelussa tämä asia on tärkeää ymmärtää ja ottaa huomioon. Mobiililaitteiden sisältä ei löydy tietokoneisiin verrattavia prosessoritehoja eikä näin ollen mobiilipelitkään voi tällaisia tehoja vaatia. Liian raskas peli saattaa pätkiä mobiililaitteilla ja estää pelin kunnollisen pelaamisen. Pahimmassa tapauksessa raskas mobiilipeli ei pyöri laitteessa ollenkaan vaan kaatuu suoritustehojen puutteeseen. Prosessorin liiallinen käyttö voi aiheuttaa myös laitteen kuumenemista, jolloin suorituskyky laskee nykylaitteissa automaattisesti (TechAdvisory 2013). Suunnittelussa on otettava huomioon myös vanhempien mobiililaitteiden heikompi suorituskyky. Vaikka peli pyörisi sujuvasti uusimmassa älypuhelinmallissa, ei tämä automaattisesti tarkoita, että vanhemmat mallit pystyisivät samaan. Liian suuret tehovaatimukset rajoittavat huomattavasti mahdollista asiakaskuntaa, jos peliä on mahdollista käyttää vain uusimmissa älylaitteissa.

Prosessorien suoritustehot eivät ole ainoa ongelma, jonka mobiililaitteet asettavat pelisuunnittelijoille. Mobiililaitteet ovat myös muistien osalta selviä altavastaajia perinteisiin tietokoneisiin nähden. Mobiililaitteista löytyy tietokoneiden tapaan käyttömuisti eli RAM-muisti sekä massamuisti. Muistien

koot ovat kuitenkin huomattavasti tietokoneita pienempiä. Käyttömuisti yleisimmissä älypuhelimissa vaihtelee 512 megabitin ja 3 gigabitin välillä (Phonegg 2015). Käytännössä tämä tarkoittaa käyttömuistin osalta sitä, että pelin vaatimaan muistiin tulee todella kiinnittää huomiota suunnitteluvaiheessa. Liiallisen käyttömuistin vaatiminen laitteelta saattaa hidastaa tai kaataa pelin. Lisäksi Android-käyttöjärjestelmissä käyttömuistia käytetään siten, että käyttäjän suosituimpia ohjelmia pidetään taustalla käynnissä, jotta niiden käyttö seuraavalla kerralla olisi nopeampaa (Developers 2015). Yhden ohjelman liian suuri käyttömuistin vaatiminen poistaa muita ohjelmia käyttömuistista ja voi näin ollen hidastaa muiden ohjelmien käynnistymistä ja niiden välillä siirtymistä.

Massamuistia puhelimissa puolestaan on 8 gigabitistä aina 128 gigabittiin (Phonegg 2015). Muistimäärät massamuistinkin osalta ovat siis melko vaatimattomia verrattuna tietokoneiden massamuistimääriin, joissa alle yhden terabitin muistit ovat jo melko pieniä.. Peliltä ei siis voi vaatia myöskään tallennustilalta järjettömiä liian suuriä määriä muistia asennusvaiheessa. Tietokoneilla pelit voivat viedä useita gigabittejä tilaa, mutta mobiililaitteille tämän kokoisilla peleillä ei ole asiaa. Mobiililaitteissa liian suuret vanhat ohjelmat saavat nopeasti väistyä, jos käyttäjä tarvitsee laitteeltaan tilaa uusille ohjelmille. Poistetun pelin pariin käyttäjä ei luultavasti enää koskaan palaa, joten peli olisi hyvä saada pysymään käyttäjän laitteessa, vaikka sitä ei juuri sillä hetkellä käytettäisikään aktiivisesti. Tämänhetkiset pelien koot Google Playssa pyörivät kymmenistä megabiteistä korkeimmillaan muutamaan sataan megabittiin.

Mobiililaitteiden kosketusnäytöt asettavat myös omat haasteensa pelien suunnitteluun. Tietokoneilla pelaaminen tapahtuu yleensä joko hiirellä, näppäimistöllä tai näiden yhdistelmällä. Mobiililaitteissa pelien pelaaminen tapahtuu kuitenkin useimmiten täysin kosketusnäytön varassa. Kosketusnäytöllä pelattaessa on otettava huomioon, että painallukset sormella eivät ole läheskään yhtä tarkkoja kuin hiirellä tehtäessä. Sormi osuu näytössä moneen kohtaan yhtäaikaisesti, joten pelissä ei voi olla liian tarkkaa painallusta vaativaa sisältöä. Kosketusnäytöllä pelattaessa ei myöskään ole mahdollista

tehdä yhtä nopeita painalluksia kuin hiirtä käyttämällä. Myös näytön koko eroaa radikaalisti tietokoneiden näytöistä, joten pelin käyttöliittymä ja grafiikat tulee suunnitella mobiililaitteille sopiviksi. Jos peli puolestaan sisältää kirjoittamista, tulee suunnittelussa ottaa huomioon näytölle ilmestyvän näppäimistön vaatima tila ja huomattavasti näppäimistöä hitaampi kirjoitusnopeus.

2.5 Pelimoottorit

Pelien ohjelmointi alusta loppuun vie ohjelmoijalta todella paljon aikaa. Tämän vuoksi sitä on pyritty helpottamaan luomalla erilaisia pelimoottoreita ohjelmoijien käytettäviksi. Pelimoottorit ovat pelintekoon tarkoitettuja apuohjelmia, jotka sisältävät pelintekoon yleensä tarvittavia valmiiksi ohjelmoituja ominaisuuksia. Pelimoottorit muodostuvat useista eri osista, joista jokainen hoitaa omaa osaansa toimivan pelin aikaansaamisessa. Näitä osia ovat muun muassa fysiikkamoottori, piirtomoottori, tekoäly, muistin hallinta ja verkkokomponentit (Gregory 2014, 11.) Osia äänen tai grafiikan luontiin pelimoottoreissa harvemmin on, joten nämä tehdään lähes aina erikseen siihen tarkoitetuilla ohjelmilla.

Pelimoottoreita on markkinoilta nykyään todella paljon ja siksi onkin tärkeää löytää itselleen se sopivin. Pelimoottoria valittaessa on hyvä vertailla niiden ominaisuuksia, käytettävää koodikieltä, julkaisualustoja ja hintaa. Suosittuja pelimoottoreita ovat esimerkiksi Unreal Engine 4, Unity, CryENGINE 3 ja Source. Tämän opinnäytetyön tekemisessä käytän GameMaker: Studio pelimoottoria, joka soveltuu hyvin 2D mobiilipelien työstöön. Lisäksi omistan kyseisen ohjelman lisenssin ja ohjelman koodikieli muistuttaa Java-ohjelmointikieltä, josta itselläni on hieman kokemusta.

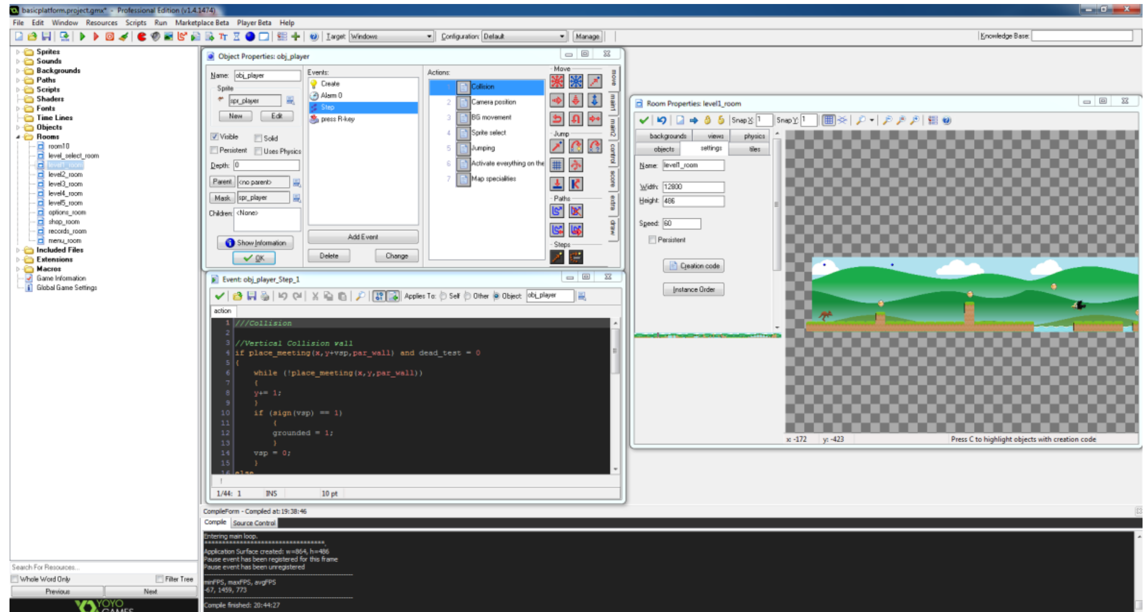
2.6 GameMaker

GameMaker on YoYo Gamesin vuonna 1999 luoma pelimoottori, jonka ideana oli mahdollistaa pelien tekeminen ilman aiempaa ohjelmointikokemusta (GMW 2014). GameMaker tarjosi käyttäjälleen Drag-and-Drop ominaisuuden, jolla pelejä oli mahdollista tehdä kirjoittamatta yhtään riviä koodia. Nykyään, yli 15 vuotta myöhemmin, GameMaker on kuitenkin kasvanut suosituksi ja toimivaksi pelinteko-ohjelmaksi tuoden mukanaan oman ohjelmointikielen GameMaker Languagen eli GML:n. GameMakerillä on tehty jo useita hyvin menestyneitä ja tuottaneita pelejä, kuten Hotline Miami, Risk of Rain ja Gunpoint. Kuvassa 1 on kuvakaappaus Game Makerin käyttöliittymästä.

GameMaker on pelimoottorina ensisijaisesti suunniteltu 2D-pelien tekoon, mutta myös työkaluja 3D-pelien toteuttamiseen lisäillään jatkuvasti. GameMaker sisältää myös yksinkertaisen grafiikaohjelman, jolla pystyy tekemään uutta grafiikkaa tai muokkaamaan muualta tuotuja kuvia. GameMakerillä toteutettu yksinkertainen peli sisältää aina seuraavat elementit:

- Sprite – Spritet antavat pelille visuaalisen ilmeen eli ne toimivat pelin grafiikkana. Sprite voi olla vain yksi kuva tai vaihtoehtoisesti useamman kuvan luoma animaatio. Game Maker tukee yleisimpiä kuvien tiedostotyyppejä, kuten .gif-, .bmp-, .png- ja .jpg-tiedostoja (Habgood ym. 2010, 3.)
- Objekti – Objektit sisältävät toisiin objekteihin liittyvää tietoja ja toiminnallisuutta. Objektit pystyvät kommunikoimaan keskenään lähettämällä ja vastaanottamalla tietoja toisilta objekteilta (Habgood ym. 2010, 4.) Jokaisella objektilla on oma roolinsa pelin toiminnassa.
- Huone – Huone tarkoittaa GameMakerissä paikkaa, jossa itse peli tapahtuu (Habgood ym. 2010, 4). Todella yksinkertaisessa pelissä voi olla vain yksi huone, mutta käytännössä huoneita löytyy kunnollisesta pelistä aina useampia. Objektit sijoitetaan huoneisiin.

Edellä mainitut elementit ovat edellytyksenä hyvin yksinkertaisen pelin luontiin GameMakerissä. Näiden lisäksi käytössä on myös lähes aina ääniä, fontteja, taustoja ja skriptejä.



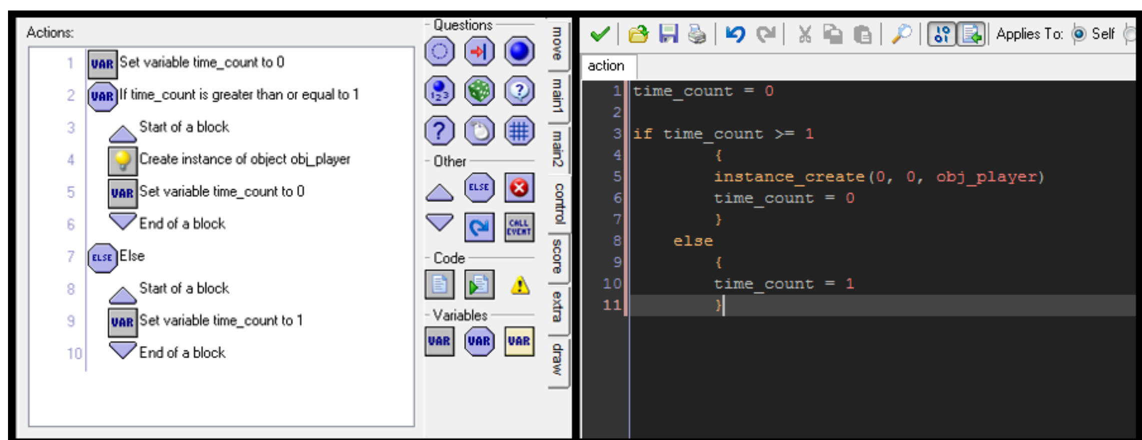
Kuva 1. GameMakerin käyttöliittymä.

GameMaker Language

GameMaker suunniteltiin aluksi lähinnä Drag-and-Drop ohjelmointia varten, jossa pelin teko tapahtui valitsemalla ja vetämällä eri funktioita sisältäviä ikoneita jonoon. Tämä tyyli on kuitenkin osittain rajallinen ohjelmoinnin kannalta eikä anna ohjelmoijalle täyttä vapautta toteuttaa monimutkaisempia asioita pelissä (Habgood & Overmars 2006, 225.) GameMaker Language tuotiin Drag-and-Drop ohjelmoinnin rinnalle täydentämään huomattuja puutteita. Nykyään Drag-and-Drop tekniikkaa ei enää ohjelmointiin tarvitse käyttää, vaan samat asiat sekä todella paljon muuta on mahdollista hoitaa huomattavasti joustavammin GameMaker Languagen avulla. Kuvassa 2 on esimerkki Drag-and-Drop tekniikasta sekä GameMaker Languagesta. Kuvan molemmissa ruuduissa toteutetaan sama asia.

GameMaker Languagea on mahdollista käyttää neljällä eri tavalla:

- Skripteissä – Skriptit ovat GameMaker Languagella luotuja omia funktioita, joita voi kutsua koodissa (Habgood ym. 2010, 4).
- Tapahtumissa – Tapahtumat ovat objektien eri vaiheissa tapahtuvia asioita (Habgood & Overmars 2006, 14). Esimerkiksi tapahtumana voi olla hiiren oikea klikkaus tai kyseisen objektin luonti. Tapahtumat ovat yleisin tapa GameMaker Languagea.
- Huoneiden luontikoodeissa – Jokaisella huoneella on luontikoodi, joka ajetaan läpi aina, kun kyseinen huone luodaan.
- Objektien luontikoodeissa – Objekteihin on mahdollista lisätä luontikoodeja, kun niitä sijoitetaan huoneisiin. Näin objektit on mahdollista saada ajamaan eri luontikoodi huoneeseen mentäessä.



Kuva 2. Koodi esimerkki.

3 ANDROID-MOBIILIPELIN TOTEUTUS

Opinnäytetyön tiimoilta ohjelmoin Android-käyttöjärjestelmälle esimerkkipelin, jossa pyrin ottamaan huomioon teoriaosuudessa esiin tuotuja huomioita toimivan pelin ominaisuuksista. Ohjelmointi tapahtuu GameMaker: Studio-ohjelmalla. Peli on tarkoitus saada opinnäytetyötä varten pelattavaan kuntoon, mutta pelin lopullinen hiominen ja julkaisu Google Play -sisältöpalveluun tehdään vasta opinnäytetyön jälkeen. Tässä luvussa tarkastellaan eri ohjelmointiratkaisuja, joita olen pelin toiminnoissa päätynyt käyttämään.

Pelin työnimike on Jumparoo ja sen ideana on yksinkertaisuudessaan päästä hyppimään kengurulla mahdollisimman pitkälle eri kentissä. Kenguru liikkuu tasaista vauhtia vasemmalta oikealle ja hyppiminen tapahtuu painamalla näyttöä. Hypyn korkeus riippuu pelaajan painalluksen pituudesta ja seuraavaa hyppyä on jo mahdollista ladata kengurun ollessa vielä ilmassa edellisestä loikasta. Kentät vaikeutuvat mitä pidemmälle pelaaja pääsee ja pelin vaikeus muodostuu oikeankorkuisten ja -pituisten hyppyjen lataamisesta. Peli tallentaa pelaajan jokaisen kentän pituusennätyksen sekä kentistä kerättävien kolikoiden määrän. Kolikoilla on mahdollista avata uusia kenttiä ja kenties myöhemmin myös uusia ulkoasuja päähahmolle. Lisäksi kolikot antavat pelaajalle edistymisen tunteen, vaikka hän ei saisikaan tehtyä kenttään uutta ennätystä.

3.1 Törmäystunnistus

Jotta päähahmona toimiva kenguru ei putoaisi kentän maankamaran läpi, tarvitaan hahmolle keino tunnistaa kohtaaminen maan kanssa.

Event: obj_player_Step

```
if place_meeting(x,y+vsp,par_wall) and dead_test = 0
    {
    if (sign(vsp) == 1)
```

```
            {
            grounded = 1;
            }
        vsp = 0;
        }
else
        {
        grounded = 0;
        y += vsp;
        }

vsp += grav;
```

Ensimmäisessä rivissä määritellään missä tapahtumassa ja missä objektissa koodi tapahtuu. Tässä tapauksessa ensimmäinen rivi kertoo, että koodi suoritetaan obj_player -objektin step-tapahtumassa. Step-tapahtuma tarkoittaa, että koodia toistetaan objektissa jatkuvasti koko pelin ajan.

Itse koodin ideana on selvittää onko pelattava hahmo kosketuksissa maanpinnan kanssa. Tätä selvitetään ”place_meeting” -funktiolla, joka tarkistaa tässä tapauksessa onko ’obj_player’ kosketuksissa ’par_wall’ -objektin kanssa. ’par_wall’ on objekti, josta jokainen maanpinnaksi tarkoitettu objekti perii ominaisuudet käyttöönsä. Näin ollen kosketusta jokaisen erilaisen maanpintaobjektin kanssa ei tarvitse erikseen tarkastaa. ’dead_test’ -muuttujaa puolestaan käytetään myöhemmin tarkistamaan, että kenguru ei ole törmännyt seinään ja hävinnyt peliä.

Koodin ’vsp’ -muuttujaa puolestaan käytetään määrittelemään hahmon vertikaalista nopeutta pelissä. Käytännössä tämä tarkoittaa, että ’vsp’ -muuttujan ollessa negatiivinen luku hahmo liikkuu ylöspäin ja positiivisena lukuna alaspäin. Mitä suurempi luku on, sitä nopeammin kenguru liikkuu ylös tai alas. Tätä muuttujaa käytetään apuna myös kengurun hypyn aikaansaavassa koodissa. Tarkistamalla ”sign” -funktiolla muuttujan ’vsp’ etumerkkillä voidaan selvittää, onko hahmo maassa vai ilmassa. Mikäli hahmo on maassa, ’grounded’ -muuttujan arvoksi muutetaan 1 ja muussa tapauksessa 0.

Mikäli hahmo on testin mukaan ilmassa, kasvatetaan kengurun y-akselin sijaintia jatkuvasti ’vsp’ -muuttujan verran. Seuraavana tarkasteltava hyppykoodi antaa ’vsp’:n arvoksi negatiivisen luvun, joka näin ollen saa kengurun nousemaan huoneessa ylöspäin. Lopuksi ’vsp’ -muuttujaan lisätään jatkuvasti ’grav’ -muuttujan arvo, jolloin ’vsp’ lopulta muuttuu positiiviseksi luvuksi ja saa kengurun jälleen liikkumaan alaspäin. ’grav’ -muuttujalle on annettu positiivinen arvo heti kengurun luontivaiheessa ja sitä muuttamalla saadaan kenguru halutessa putoamaan nopeammin tai hitaammin.

3.2 Kengurun hyppy

Pelin perusidean toteuttamiseksi kengurun tulee hypätä pelaajan painaessa näyttöä. Lisäksi hyppyä tulee pystyä lataamaan korkeammaksi pitämällä näyttöä kauemmin painettuna.

Event: obj_player_Step_5

```
Key_Jump    = device_mouse_check_button_released(0, mb_left);
Key_Pressed = device_mouse_check_button(0, mb_left);

if (Key_Pressed) and jamount >= -13 and dead_test = 0
    {
    jamount -= 0.4
    }
```

Ylempi koodi alkaa määrittelemällä ’Key_Jump’ ja ’Key_Pressed’ -muuttujat. Näihin muuttujiin lisätään arvoksi ”device_mouse_check_button_released” ja ”device_mouse_check_button” -funktiot. Ensimmäinen funktio tarkistaa, jos hiiren vasen painike vapautetaan ja toinen funktio puolestaan tarkistaa painetaanko hiiren vasenta painiketta. Hiiren vasen painike tarkoittaa Androidilla työskennellessä normaalia näytön painallusta GameMakerissä. Käytännössä arvojen tallentaminen lyhyempiin muuttujiin säästä vain kirjoittamisen määrää ja selkeyttää koodia, sillä pitkiä funktioita ei tarvitse kirjoittaa uudestaan koodiin.

Seuraavaksi koodissa tarkistetaan if-lauseen avulla onko hiiren vasen painike, eli näyttö, painettuna ja onko muuttuja 'jamount' suurempi tai yhtä suuri kuin -13. Lisäksi tarkistetaan vielä onko hahmo hengissä eli 'dead_test' -muuttujan arvo on nolla. 'jamount' -muuttuja määrittelee määrän, jonka hahmo tulee hyppäämään pelaajan irroittaessa kosketuksen näytöstä. 'jamount' -muuttujaa siis pienennetään 0.4:llä niin kauan kunnes sen arvoksi tulee -13 tai pelaaja irroittaa kosketuksen näytöstä. -13 on testaamalla etsitty sopiva luku, joka määrittää maksimikorkeuden kengurun hypylle. Tämä luku saavutetaan muutaman sekunnin painalluksella, jonka jälkeen kengurun hyppy ei lennä enää korkeammalle, vaikka pelaaja lataisi hyppyä pidempään.

Event: obj_player_Step_5

```
if (Key_Jump)
        {
        if grounded = 1;
                {
                vsp = jamount;
                }
        jamount = 0
        }
```

Hypyn lataamisen jälkeen toteutetaan itse hyppy. Tämä tapahtuu myös if-lauseella tarkistamalla 'Key_Jump' -muuttuja eli onko kosketus näytöstä irronnut. Lisäksi tarkistetaan onko hahmo maassa 'grounded' -muuttujan avulla. Mikäli molemmat if-lauseet ovat tosia, muutetaan aiemmin käytetyn 'vsp' -muuttujan arvoksi hypyn latausmäärä eli 'jamount' arvo. 'vsp' -muuttujan muutos saa kengurun loikkaamaan halutun määrän. Lopuksi 'jamount' muutetaan takaisin nollaksi odottamaan seuraavaa latausta.

3.3 Pelin häviäminen

Jotta peliin saataisiin jotain ideaa ja haastetta, pitää peli olla myös mahdollista hävitä. Jumparoossa pelin häviäminen tapahtuu siten, että pelaaja törmää kentistä löytyviin erilaisiin esteisiin.

Event: obj_player_Step

```
if place_meeting(x,y,par_block)
	{
	dead_test = 1;
	}
```

Tällä kertaa ”place_meeting” -funktiota käytetään tarkistamaan törmääkö hahmo omassa sijainnissaan ’par_block’ -objektiin. Kaikki pelin esteet, jotka aiheuttavat törmäyksessä pelin häviämisen, perivät ’par_block’ -objektin. Mikäli törmäys tapahtuu, vaihdetaan ’dead_test’ -muuttujan arvo nollasta yhteen.

Event: obj_player_Step

```
if dead_test = 1
	{
	dead_test = 2;
	sprite_index = spr_player_dead;
	image_index = image_index;
	image_speed = 0.8;
	direction = 270;
	speed = 0.4;
	alarm[0] = 60;
	}
```

‘dead_test’ -muuttujan saadessa arvon yksi, tapahtuu useita asioita. Aluksi ’dead_test’ -muuttujan arvoksi vaihdetaan 2, jotta tätä koodia ei toisteta useita kertoja eikä myöskään koodeja, jotka tapahtuvat arvolla 0. ’dead_test’ -muuttuja nollaantuu aina, kun kenguru luodaan eli esimerkiksi kentän alkaessa alusta. Myös kengurun sprite vaihdetaan ’sprite_index’ -muuttujaa vaihtamalla

kuvastamaan paremmin hävittyä peliä. ’image_speed’ puolestaan säätää nopeuden, jolla sprite käy läpi siitä löytyvää kuvasarjaa. ’direction’ on muuttuja, joka nimensä mukaisesti antaa suunnan hahmolle. Suunta 270 tarkoittaa GameMakerissä suoraan alaspäin. Näin kenguru lähtee putoamaan alaspäin pelaajan hävitessä pelin. ’speed’ puolestaan määrittää nopeuden, jolla objekti liikkuu ’direction’ -muuttujan antamaan suuntaan. Pelaajan hävitessä pelin käynnistyy myös hälytystapahtuma ’alarm[0]’. Hälytys toimii siten, että sille annetun arvon mukaisen ajan jälkeen tapahtuu jotain. Esimerkiksi tällä kertaa arvoksi annetaan 60, joka tarkoittaa tässä tapauksessa yhtä sekuntia, koska huoneen nopeutena on sama 60.

Event: obj_player_Alarm_0_1

```
room_restart();
```

Pelin häviämisen kannalta tärkein tapahtuma, joka hälytyksestä aiheutuu, on funktio ”room_restart”. Tämä funktio aloittaa nimensä mukaisesti pelatun huoneen alusta pelaajan törmättyä esteeseen ja pudottua alaspäin hälytyksen viiveen eli yhden sekunnin ajan.

3.4 Hahmon kuvan valinta

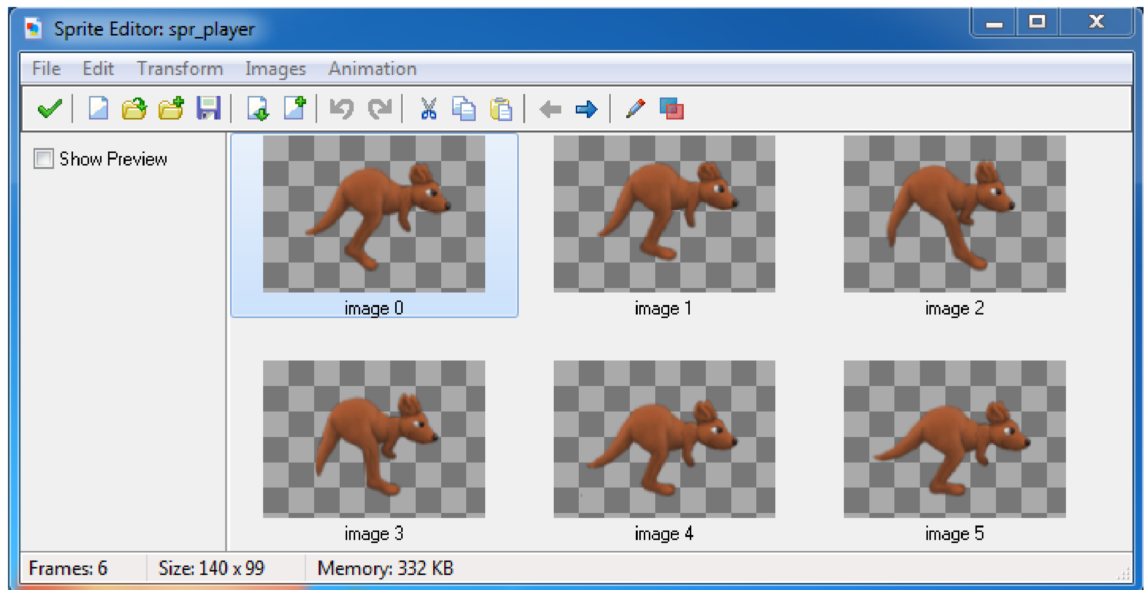
Yhdellä kuvalla luotu päähahmo ei ole kovinkaan kiinnostavaa katseltavaa. Viihdyttävyyden kannalta onkin tärkeää saada hahmosta elävämmän näköinen vaihtuvilla ja tilanteesta riippuvilla animaatioilla ja kuvilla.

Jumparoossa hahmolla on käytännössä neljä eri kuvavaihtoehtoa, jotka ovat tasaisella hyppely, ylöspäin hyppääminen, alaspäin hyppääminen sekä törmääminen. Näistä törmäyksen tarkistus hoidettiin jo edellisessä luvussa tarkistamalla ’dead_test’ -muuttujaa. Loput kuvat saadaan näkymään oikeaan aikaan tarkastelemalla ’grounded’ -muuttujaa sekä y-akselin arvoja.

Event: obj_player_Step_4

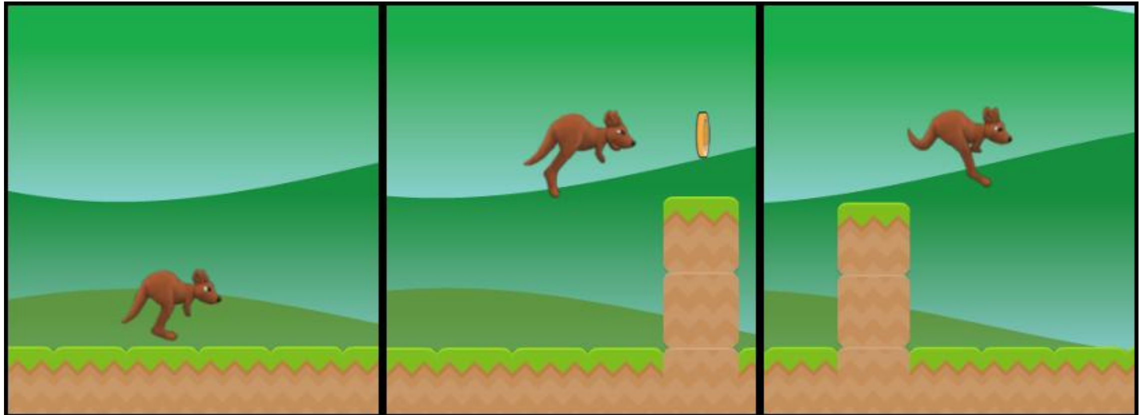
```
if grounded = 1 and dead_test = 0 and yprevious == y
    {
    sprite_index = spr_player;
    image_speed = 0.4;
    }
else
    {
    if grounded = 0 and dead_test = 0 and yprevious < y
        {
        sprite_index = spr_player_falling;
        }

    if grounded = 0 and dead_test = 0 and yprevious > y
        {
        sprite_index = spr_player_jump;
        }
    }
```

Ensimmäinen if-lause selvittää, onko hahmo maassa 'grounded' -muuttujan avulla sekä vertaamalla nykyistä y-akselia 'yprevious' -muuttujaan eli edelliseen y-akselin arvoon. Periaattessa pelkästään toinen näistä tarkistuksista riittäisi selvittämään onko hahmo maassa, mutta pelin testailussa tuli ilmi, että kahden muuttujan tarkistus toimi huomattavasti varmemmin. Mikäli hahmo on näiden tarkistusten perusteella maassa, vaihdetaan objektin kuvaksi 'spr_player'. Tämä sprite on animaatio, joten lisäksi tarvitaan 'image_speed' -muuttuja määrittelemään animaation nopeutta. Kuvassa 3 näkyy kengurun kävelyanimaation kuvia Game Makerin omassa kuvaeditorissa.



Kuva 3. Sprite-esimerkki.

Tämän jälkeen selvitetään onko hahmo ilmassa ja kumpaan suuntaan se on menossa. Mikäli hahmon 'yprevious' eli edellinen y-akselin arvo on pienempi kuin nykyinen, tarkoittaa se hahmon putoamista alaspäin. Tällöin 'sprite_index' eli kuvan määräävä muuttuja vaihtuu 'spr_player_falling' -kuvaksi. Jos taas 'yprevious' on suurempi kuin nykyinen y:n arvo, vaihdetaan kuvaksi 'spr_player_jumping', joka kuvastaa hahmon hyppäämistä. Kumpikin näistä kuvista on yksittäinen kuva, joten kuvien vaihtumisnopeutta ei tarvitse säätää 'image_speed' -muuttujalla. Kuvassa 4 näkyy vasemmalla yksi kengurun kävelyanimaation kuvista. Keskimmäisessä ruudussa puolestaan on kuva kengurun hypystä ja viimeisessä ruudussa laskeutumisesta.



Kuva 4. Kengurun animaatio.

3.5 Muistin käytön optimointi

Peleissä on tärkeää ottaa huomioon myös pelin vaatiman muistin määrä, jotta se pysyy järkevissä rajoissa. Varsinkin suuremmissa peleissä suorituskyvyn optimointi nousee yhä tärkeämpään roolin, koska näytöllä voi useaan aikaan olla todella suuri määrä grafiikkaa ja objekteja. Jumparoo on pelinä melko yksinkertainen eikä se juurikaan koettele laitteiden suorituskykyä millään osa-alueella. Siitä huolimatta pyrin käyttämään pelissä harjoituksen vuoksi muutamaa eri muistin optimointikeinoa.

GameMakerissä on mahdollista aktivoida ja deaktivoida objekteja. Tämä tarkoittaa, että objekti on mahdollista kytkeä pois päältä ja takaisin päälle haluttaessa. Deaktivoitua objektia ei pysty käyttämään mitenkään ennen sen uudelleen aktivointia.

Event: obj_player_Create_2

```
instance_deactivate_all(true);
```

Funktio “instance_deactivate_all” poistaa kaikki huoneessa olevat objektit käytöstä. Kyseisellä funktiolla on yksi muuttuja ’notme’, jolla määritellään

funktion suorittavan objektin deaktivointi. Funktion perässä oleva ’true’ tarkoittaa, että ’notme’ muuttuja on tosi eikä tätä objektia näin ollen deaktivoida. ’false’ puolestaan aiheuttaisi myös tämän objektin eli itse kengurun deaktivoinnin, mutta tässä pelissä emme sitä halua.

Suorittamalla pelkästään tämä funktio, ei pelissä olisi mitään muita objekteja, kuin päähahmona toimiva kenguru. Tämä ei ole tarkoitus, joten tarvitaan tapa aktivoida vain tarvittavat objektit pelin edetessä. Yksi keino toteuttaa tämä on aktivoida käyttöön vain pelin näytöllä näkyvät objektit, jolloin kaikki muut, sillä hetkellä turhat objektit, pysyvät vielä poissa käytöstä.

Event: obj_player_Create_2

```
instance_activate_region(view_xview, view_yview, view_xview + 500,
view_yview, false);
```

Aktivoinnissa käytettävä funktio “instance_activate_region” aktivoi vain objektit määritellyltä alueelta. Funktiossa on viisi muuttujaa, jotka ovat ’left’, ’top’, ’width’, ’height’ ja ’outside’. Kaksi ensimmäistä muuttujaa määrittelevät alueen lähtöpaikan, kaksi seuraavaa antavat alueella leveyden sekä korkeuden ja viimeinen muuttua määrittelee aktivoidaanko alue sisältä ’false’ vai ulkoa ’true’.

Tässä tapauksessa neljään ensimmäiseen muuttujaan on annettu arvoksi ’view_xview’ ja ’view_yview’ -muuttujat, jotka sisältävät automaattisesti käytetyn näkymän x- ja y-akselin paikat sekä myös käytetyn näkymän leveyden ja korkeuden. Aktivointialueen leveyttä on lisätty funktiossa 500 pikselillä, jotta huoneen objektit aktivoituisivat hieman ennen näkymään ilmestymistä sekä jäisivät ruudulta poistuttuaan vielä hetkeksi aktiivisiksi. Aikaisemman aktivoinnin ideana on tuoda objektit näkyviin hieman sulavammin ja myöhäisemmällä deaktivoinnilla jätetään aikaa objektille poistaa itsensä kokonaan pelistä ohitettuaan pelaajan. Funktion viimeinen muuttuja ’false’ määrittelee sen, että aktivointi tapahtuu nimenomaan alueen sisällä oleville objekteille.

Toinen tapa keventää pelin aiheuttamaa kuormitusta laitteessa, on poistaa turhat objektit kokonaan pelistä. Deaktivoidut objektit ovat kevyitä, mutta vievät

silti pienen määrän muistia. Poistetut objektit puolestaan häviävät kokonaan pelistä eikä niitä voi enää aktivoida tai kutsua mitenkään.

Event: par_block_Step_1

```
if x < (obj_player.x - 250)
    {
    instance_destroy();
    }
```

Tämä kyseinen koodi löytyy niin ’par_block’ -objektista, kuin myös ’par_wall’ -objektista, joten käytännössä se sisältyy kaikkiin näkyviin objekteihin paitsi itse päähahmoon. Koodissa selvitetään if-lauseen avulla, onko objektin x-akselin arvo pienempi kuin päähahmon x-akselin arvo, josta on vähennetty vielä 250 pikseliä. Käytännössä siis tarkistetaan onko päähahmo jo ohittanut objektin ja mennyt vielä tästä 250 pikseliä eteenpäin. Tällöin objekti, jossa tämä koodi suoritetaan on jo ulkona pelin näkymästä. Jos tarkastus pitää paikkansa, tuhotaan tämä objekti pois huoneesta ”instance_destroy” -funktiolla. Deaktivoitu objekti ei pysty suorittamaan mitään komentoja ja siksi ylempänä aktivointialueen leveyttä kasvatettiin sen verran, että objekti ehtii tuhoamaan itsensä ennen sen uudelleen deaktivointia.

GameMakerissä if-lauseiden tarkistukset suoritetaan aina loppuun, vaikka yksi tai useampi ehto olisi jo todettu vääräksi. Monissa ohjelmointikielissä if-lauseen tarkistus keskeytyy heti ensimmäiden ehdon ollessa epätosi. Tämän vuoksi GameMakerissä if-lauseiden porrastaminen on tehokkaampi tapa suorituskyvyn kannalta, kuin useiden ehtojen perättäinen tarkistaminen.

```
if grounded = 1 and dead_test = 0 and yprevious == y
    {
    sprite_index = spr_player;
    image_speed = 0.4;
    }
```

Esimerkiksi ylempi lauseke olisi järkevämpää kirjoittaa GameMakerissä seuraavaan muotoon:

```
if grounded = 1
    {
    if dead_test = 0
        {
          if yprevious == y
            {
            sprite_index = spr_player;
            image_speed = 0.4;
            }
        }
    }
```

Ensimmäisessä esimerkissä GameMaker suorittaa aina kaikkien kolmen muuttujan tarkistuksen loppuun, vaikka ensimmäinen muuttuja ’grounded’ ei olisikaan haluttu arvo. Alemmassa esimerkissä tarkistus puolestaan loppuu aina sen if-lauseen kohdalle, jossa ehto ei toteudu. Jos ’grounded’ olisi arvoltaan 0, ei jälkimmäisiä muuttujia edes käytäisi tarkistamassa. Ensimmäiseksi testiksi on järkevintä laittaa ehto, joka on harvimmin tosi, jotta muiden if-lauseiden tarkistus jää mahdollisimman vähälle.

3.6 Pelin tallennus

Lähes kaikissa peleissä on mahdollisuus tallentaa peli tai vaihtoehtoisesti automaattinen tallennus itse pelin puolesta. Mobiilipeleissä varsinkin jälkimmäinen löytyy lähes jokaisesta pelistä. Tallentamista tarvitaan peleissä, jotta pelaajien edistys pelissä olisi mahdollista pitää muistissa. Kukaan ei jaksa pitkään pelata peliä, jossa peli aloitetaan aina täysin alusta.

Jumparoossa on muutamia eri asioita, jotka on pelin kannalta tärkeä saada säilymään muistissa. Ensimmäinen asia on eri kenttien ennätykset, joihin pelin koukuttavuus perustuu. Mikäli kenttien ennätykset eivät säilyisi muistissa ja näkyvissä pelaajalle, ei kentän uudelleen pelaamisen olisi minkäänlaista houkutusta. Toinen asia on kentistä kerättävien kolikoiden kokonaismäärä.

Kolikoiden määrää on tärkeää tallentaa, jotta pelaaja voi kerätä suuren määrän kolikoita useiden pelikertojen aikana kalliimpien kenttien avaamiseen. Kolmas tallennettava asia on jo avatut kentät. Ymmärrettävää on, että pelaaja ei halua avata samaa kenttää usean otteeseen, vaan kentän avaus pitää saada tallennettua.

3.6.1 Pelin lataaminen avattaessa

Pelissä käytetään tallennusmekanismina yksinkertaista ini-tiedostoa, joka soveltuu hyvin peleihin, joissa ei ole suurta määrää tallennettavaa. Tämä toimii siten, että pelissä tallennettavat muuttujat kirjoitetaan ulkoiselle ini-päätteiselle tiedostolle ja tarvittaessa arvot luetaan tältä ini-tiedostolta peliin mukaan.

Pelin alussa ini-tiedostolta halutaan lukea kaikki mahdolliset tallennukset, jotta peli saadaan viime tallennuksen tilaan aloitettaessa pelaaminen.

Event: obj_controller_Create_2

```
if file_exists("save.ini")
        {
        ini_open("save.ini");
        total_coins = ini_read_real("save","coins",0)
        distance_lvl1 = ini_read_real("save","d1",0)
        distance_lvl2 = ini_read_real("save","d2",0)
        lvl2_open = ini_read_real("save","lvl2",0)
        ini_close();
        }
```

Tämä koodi suoritetaan ’obj_controller’ nimistä objektia luotaessa. ’obj_controller’ on näkymätön koko pelin säilyvä objekti, jonka tehtävänä on hoitaa osaa pelin mekaniikoista. Koodin if-lause tarkistaa löytyykö ”save.ini” nimistä tiedostoa, joka käytännössä löytyy aina paitsi pelin ensimmäisellä käynnistyskerralla. Ini-tiedosto luodaan automaattisesti pelin tiedostoihin, kun peli tallentaa ensimmäisen muuttujan kyseiseen tiedostoon. Tämän jälkeen tiedosto löytyy aina, ellei peliä poisteta laitteesta täysin.

Jos tiedosto löytyy laitteesta, avataan se ”ini_open” -funktiolla, jonka ainoana attribuuttina on avattavan tiedoston nimi eli tässä pelissä ”save.ini”. Ini-tiedoston ollessa auki sieltä on mahdollista lukea sinne tallennettuja arvoja pelissä käytettäviin muuttujiin. Numeroiden lukeminen tapahtuu ”ini_read_real” -funktiolla, johon syötetään attribuuteiksi sektori, josta tiedot haetaan, arvon sisältävä avain sekä vakioarvo siltä varalta, että avainta ei löydy. Tässä tapauksessa tiedot haetaan aina ”save” -sektorista ja vakio-arvona on nolla. Haettu avain vaihtelee sen mukaan, mitä arvoa ollaan hakemassa.

Koodissa haetaan neljä eri arvoa pelin muuttujiin. ’total_coins’ -muuttujaan haetaan ”coins” arvo, joka sisältää tiedon pelaajan kolikoiden määrästä. ’distance_lvl1’ ja ’distance_lvl2’ -muuttujiin haetaan ensimmäisen ja toisen kentän ennätyksiä tallentavat arvot ”d1” ja ”d2”. ’lvl2_open’ on puolestaan muuttuja, joka kertoo haetun ”lvl2” arvon perusteella onko toinen kenttä avattu vai lukittu. Viimeinen funktio ”ini_close” sulkee ini-tiedoston, jotta muistia ei kuluisi turhaan sen auki pitämiseen.

3.6.2 Kolikoiden ja ennätysten tallennus

Kolikoiden sekä ennätysten tallennus tapahtuu pelaajan hävitessä pelin. Tallennus on sijoitettu samaan häviöstä käynnistyvään hälytykseen, kuin aiemmin opinnäytetyössä mainittu kentän uudelleenaloitus.

Event: obj_player_Alarm_0_1

```
ini_open("save.ini");
ini_write_real("save","coins",obj_controller.total_coins)
ini_close();

switch (room)
    {
    case level1_room:
    if obj_controller.distance > obj_controller.distance_lvl1
        {
        obj_controller.distance_lvl1 = obj_controller.distance
```

```
        ini_open("save.ini");
        ini_write_real("save","d1", obj_controller.distance_lvl1)
        ini_close();
        }
        break;

    case level2_room:
    if obj_controller.distance > obj_controller.distance_lvl2
        {
        obj_controller.distance_lvl2 = obj_controller.distance
        ini_open("save.ini");
        ini_write_real("save","d2", obj_controller.distance_lvl2)
        ini_close();
        }
        break;
    }
```

Koodi alkaa kolikoiden määrän päivittämisellä ini-tiedostoon ”ini_write_real” -funktiota käyttäen. Funktioon syötetään attribuuteiks ini-tiedoston sektori, avain sekä muuttuja joka tallennetaan avaimen arvoksi. Tässä tapauksessa avaimen arvoksi tallennetaan muuttujan ’total_coins’ arvo. Tämä muuttuja on kuitenkin eri objektin muuttuja, kuin missä itse koodi suoritetaan. GameMakerissä toisten objektien muuttujiin on mahdollista päästä käsiksi ilmoittamalla muuttujan edessä sen objektin nimen, josta muuttuja haetaan. Koodissa esimerkiksi kaikki ini-tiedostoon tallennettevat muuttujat ovat ”obj_controller” -objektin muuttujia ja siksi tämä on mainittu jokaisen muuttujan edessä.

Kolikoiden määrä tallennetaan aina pelaajan hävitessä pelin, mutta ennätyksiä puolestaan tallennetaan vain, jos kentän vanha ennätys rikkoutuu. Koodissa suoritetaan ensin kentän tarkistus switch case -lauseella, jossa muuttujaksi syötetään GameMakerin valmis muuttuja ’room’. Tämä muuttuja sisältää aina käynnissä olevan huoneen nimen. Switch case -lauseella siis tarkistetaan missä kentässä pelaaja on ollut, jotta on mahdollista verrata pelaajan saavuttamaa tulosta sen hetkiseen ennätykseen. Esimerkiksi ”case level1_room” kohtaan siirrytään mikäli ’room’ muuttuja saa arvokseen ensimmäisen kentän eli ’level1_room’.

Kun oikea huone on saatu haettua, siirrytään vertailemaan pelaajan kulkemaa matkaa vanhaan ennätykseen. Tämä tapahtuu if-lauseen avulla verraten ”obj_controller” -objektin muuttujia ’distance’ ja ’distance_lvl1’ toisiinsa. ’distance’ pitää sisällään pelaajan uusimman kuljetun matkan, joten sen pitää olla suurempi kuin ’distance_lvl1’ -muuttujan, joka puolestaan sisältää vanhan ennätyksen. ’distance’ saadaan pyöristämällä hahmon x-akselin arvo jaettuna sadalla, jotta luku näyttäisi enemmän metreiltä:

Event: obj_controller_Step_1

```
if instance_exists(obj_player)
	{
	distance = floor(obj_player.x/100)
	}
```

Mikäli ’distance’ -muuttuja on tallennuskoodin vertailussa ennätystä säilövää ’distance_lvl1’ suurempi, siirrytään koodissa eteenpäin tallentamaan uutta ennätystä. Aluksi ’distance_lvl1’ arvo korvataan uudella ennätyksellä, jotta tallennusta ei toisteta montaa kertaa ennätystä pitävän muuttujan ollessa vieläkin pienempi kuin kuljetun matkan. Seuraavaksi avataan tallennustiedosto ”ini_open” -funktiolla, jotta ennäys saadaan noudettua seuraavallakin pelikerralla. Numeroarvojen kirjoittaminen tiedostoon tapahtuu ”ini_write_real” -funktiolla, jonka attribuutteina ovat tallennussektori, tallennuksen säilövä avain sekä avaimeen tallennettava arvo. Tässä tapauksessa ”obj_controller” -objektin muuttuja ’distance_lvl1’ tallennetaan ’d1’ avaimeen ’save’ sektoriin. Tallennuksen jälkeen ini-tiedosto suljetaan ja switch-lause lopetetaan break-komennolla.

3.7 Kenttävalikko

Jotta tällaisessa pelissä riittäisi tarpeeksi pelattavaa, täytyy sen tarjota pelaajalle useita erilaisia kenttiä, joiden ennätyksiä pelaaja voi rikkoa.

Lisäkentät puolestaan ovat aluksi lukittuja ja vaativat kolikoilla avaamista. Peliin tarvitaan siis valikko, jossa on mahdollista selata kenttiä, avata lukittuja kenttiä kolikoilla, tarkastella kenttien ennätyksiä sekä tietysti siirtyä haluttuun kenttään.

Kenttävalikko muodostuu ”obj_lvl_select_ctrl” -objektin ’selected_lvl’ -muuttujan ympärille, jonka arvo määrittelee mikä kenttä valikossa näkyy. ’selected_lvl’ -muuttujan arvo saadaan muuttumaan painikkeilla, joista toinen lisää ja toinen laskee kyseistä arvoa.

Event: obj_lvl_select_ctrl_Step_1

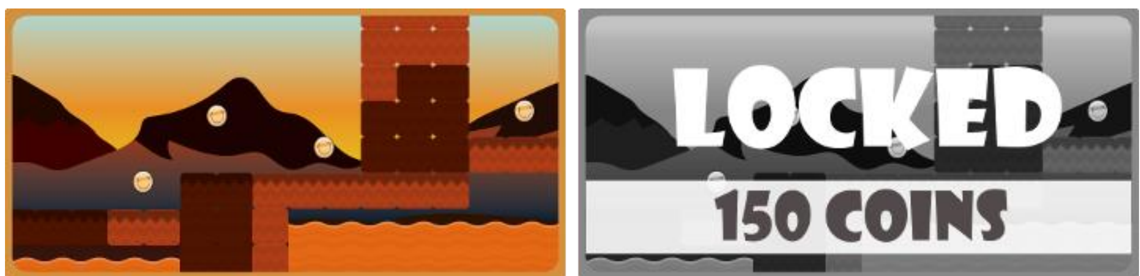
```
if room = level_select_room and test_created = 0
    {
    switch (selected_lvl)
        {
        case 1:
        instance_create(select_x, select_y, obj_level1_select)
        test_created = 1
        break;


        case 2:
        if obj_controller.lvl2_open = 0
            {
            instance_create(select_x, select_y, obj_level2_closed)
            }
            else
            {
            instance_create(select_x, select_y, obj_level2_select)
            }
        test_created = 1
        break;
        }
    }
```

Koodi lähtee liikkeelle tarkastamalla käytössä olevan huoneen sekä ’test_created’ -muuttujan arvon. Huone tarkastetaan siksi, että tämä objekti on pysyvä objekti. Tämä tarkoittaa, että objekti on aina olemassa joka huoneessa,

jolloin sen muuttujia on mahdollista käyttää tarvittaessa myös muualla kuin pelkästään kenttävalikko-huoneessa. ’test_created’ -muuttujalla puolestaan tarkistetaan onko tämän koodin objekti jo luotu. Näin vältytään luomasta haluttua objektia yhä uudestaan ja uudestaan tätä koodia ajettaessa.

Seuraavaksi siirrytään tarkistamaan switch case -lauseen avulla, mikä objekti on tarkoitus luoda. Tarkastettavana muuttujana käytetään ’selected_lvl’ -muuttujaa ja sen arvoa. Mikäli arvo on yksi, luodaan ”instance_create” -funktiolla objekti ’obj_level1_select’ ennalta määritettyihin ’select_x’ ja ’select_y’ koordinaatteihin. Tämä objekti on kuva ensimmäisestä kentästä ja sitä painamalla siirrytään kyseiseen kenttään. Koska ensimmäinen kenttä on pelissä aina avoinna, ei tässä tapauksessa ole vielä tarvetta tarkistaa erikseen, onko pelaaja avannut kentän.

Jos tarkistettavan ’selected_lvl’ -muuttujan arvo on kaksi, täytyy ennen halutun objektin luontia tarkistaa, onko kenttä lukittu vai auki. Tarkistus onnistuu ’obj_controller.lvl2_open’ -muuttujan arvolla. Jos kenttä on entuudestaan avattu, on arvo 1 ja muussa tapauksessa 0. Lukitulle kentälle luodaan objekti ’obj_level2_closed’ ja avatulle kentälle ’obj_level2_select’. Suljettua kenttää painammalla pelaajan on mahdollista avata kenttä kolikoilla ja avattua kenttää painammalla puolestaan siirtyä kenttään. Koodissa on näkyvillä vain kaksi kenttää, mutta muiden kenttien toimintaperiaate valikossa on täysin sama kuin ’select_lvl’ -muuttujan saadessa arvon 2. Kuvassa 5 näkyy esimerkki avastusta kentästä sekä vielä lukitusta kentästä.



Kuva 5. Kenttänappulat.

Avattuun kenttään siirtyminen kuvaa painamalla tapahtuu hyvin yksinkertaisesti ”room_go_to” -funktion avulla, johon annetaan attribuutiksi halutun huoneen nimi. Alla näkyvä koodi siirtyy esimerkiksi ‘level3_room’ -nimiseen huoneeseen hiiren vasemmalla painikkeella.

Event: obj_level3_select_Left Released_1

```
room_goto(level3_room)
```

Lisäksi sekä avoinna olevasta kenttäpainikkeesta, että lukitusta löytyy koodi, joka seuraa kentän vaihtumista.

Event: obj_level3_select_Step_1

```
if obj_lvl_select_ctrl.selected_lvl != 3
    {
    instance_destroy();
    }
```

Koodissa seurataan aiemmin mainittua ”obj_lvl_select_ctrl” -objektin ’selected_lvl’ -muuttujaa, joka määrää minkä kentän kuva tulisi olla näkyvillä valikossa. Esimerkiksi tässä tapauksessa ’selected_lvl’ -muuttujan saadessa eri arvon kuin kolme, tuhotaan kentän kuva pois. Näin tehdään tilaa toisen kentän kuvalle, joka luodaan samaan kohtaan.

Kenttien selaamiseen tarvitaan tietysti myös painikkeet, joilla valikossa on mahdollista navigoida. Jumparoossa kenttien selailu tapahtuu kahdella nuolipainikkeella, joista toinen vaihtaa kenttää vasemmalle ja toinen oikealle.

Event: obj_arrow_left_Left Released_1

```
if obj_lvl_select_ctrl.selected_lvl > 1
    {
    obj_lvl_select_ctrl.selected_lvl -= 1
    obj_lvl_select_ctrl.test_created = 0
    }
```

Yllä näkyvässä koodissa vasenta nuolta painettaessa tarkistetaan ”obj_lvl_select_ctrl” -objektin ’selected_lvl’ -muuttujan arvo. Mikäli muuttuja on yksi, valikossa näkyy jo ensimmäinen kenttä eikä vasemmalle vaihtava nuoli näin ollen tee enää mitään. Jos ’selected_lvl’ puolestaan on suurempi kuin yksi, tarkoittaa tämä sitä että kenttänä on jokin muu kuin ensimmäinen kenttä. Tällöin muuttujan arvosta vähennetään yksi aina klikkauksen tapahtuessa. Lisäksi ’test_created’ -muuttuja muutetaan nollaksi, jotta uusi kenttäkuvake voidaan luoda.

Event: obj_arrow_right_Left Released_1

```
if obj_lvl_select_ctrl.selected_lvl < obj_lvl_select_ctrl.max_lvl
	{
	obj_lvl_select_ctrl.selected_lvl += 1
	obj_lvl_select_ctrl.test_created = 0
	}
```

Edeltävän toisen selausnuolen koodin toimintaperiaate on muuten sama, mutta siinä tarkkaillaan ’selected_lvl’ -muuttujan vaihtumista pienemmäksi, kuin ’max_lvl’ -muuttuja, joka pitää sisällään kenttien kokonaismäärän. Jos ehto toteutuu, lisätään ’selected_lvl’ -muuttujan arvoon yksi ja ’test_created’ muutetaan nollaksi.

Jotta kenttiä selatessa kenttäkuvakkeet eivät jäisi toistensa päälle viemään turhaan muistia ja sekoittamaan valikkoa, tuhotaan vanha kenttäkuvake aina samalla, kun uusi luodaan sen tilalle. Tämä onnistuu hyvin helposti hyödyntämällä jälleen ’selected_lvl’ -muuttujaa.

Event: obj_level3_select_Step_1

```
if obj_lvl_select_ctrl.selected_lvl != 3
	{
	instance_destroy()
	}
```

Koodi on poimittu kolmannen kentän valikkokuvasta ja tämän vuoksi if-lauseen vertailtavana lukuna on kolme. Mikäli 'selected_lvl' -muuttuja saa arvokseen muun kuin kentän numeron, suoritetaan funktio "instance_destroy". Funktio tuhoaa objektin, jossa se suoritetaan, heti koodia ajettaessa. Kuvassa kuusi näkyy valmis valikko, jossa keskellä on kenttäpainike ja sivuilla kenttää vaihtavat nuolet.

Kuva 6. Valmis valikko.

4 YHTEENVETO

Tässä opinnäytetyössä pyrittiin selvittämään, mitä toimivan Android mobiilipelin kehityksessä pitää ottaa huomioon. Lisäksi teorian pohjalta toteutettiin toimiva Android mobiilipeli. Pelissä hypitään esteiden yli automaattisesti liikkuvalla kengurulla. Hyppiminen toimii kosketusnäytöllä ja hypyn korkeus riippuu painalluksen pituudesta. Lisäksi peli tallentaa pelaajan ennätykset jokaisesta kentästä.

Opinnäytetyön tekemisessä oli paljon haastetta, vaikka työllä ei ollutkaan erillistä toimeksiantajaa. En ollut ennen tätä työtä tehnyt peliä näin pitkälle vaan aiemmat kokemukseni rajoittuivat lähinnä yksinkertaisten harjoituspelien ohjelmointiin. Tässä pelissä lähdin kuitenkin siitä lähtökohdasta, että pelistä tulisi kilpailukykyinen nykymarkkinoiden peleille. Opinnäytetön suurimmat ongelmat tulivat vastaan tallennusmekaniikan koodauksessa sekä Androidin mukaan ottamisesta. Tietojen tallentamista en ollut ennen käyttänyt, joten sen toteuttaminen vaati paljon opettelemista. Ongelmaksi tuli tietojen tallennus oikeaan aikaan ilman jatkuvaa tallennusta, joka saattaisi viedä turhaan muistia. Androidin testauksessa vaikeinta oli GameMakerin Android-lisäosan käyttöönotto ja yhdistäminen omaan puhelimeeni. Suurimmat ongelmat aiheutuivat lopulta oman puhelimeni hieman viallisesta Micro-USB liitäntäpaikasta, jonka vuoksi GameMaker ei aluksi tunnistanut laitetta oikein.

Ohjelmointialustana käytämääni GameMaker-pelimoottoria olin ennestään jo jonkin verran käyttänyt. Tähän opinnäytetyöksi ohjelmoituun peliin oma osaamiseni ei kuitenkaan ollut suoraan riittävää, vaan opeteltavaa tuli vastaan todella paljon. Lisäksi Androidille en ollut ennen tehnyt kyseisellä pelimoottorilla mitään, joten koko Android osuus oli itselleni täysin uutta asiaa ja vaatikin yllättävän paljon aikaa saada toimimaan. Omasta mielestäni GameMaker sopii tämän tyyppisen 2D-mobiilipelin toteuttamiseen erinomaisesti, kunhan ohjelman opetteluun jaksaa hieman panostaa.

Ohjelmoimani peli onnistui odotuksiin nähden todella hyvin. Pelin mekaniikat toimivat halutulla tavalla ja peli on testattu toimivaksi kahdella eri mallisella älypuhelimella. Pelistä löytyy tällä hetkellä kolme erilaista pelattavaa kenttää. Peli olisi periaattessa jo lähes julkaisukelpoinen, mutta pelattavaa kolme kenttää ei tarjoa vielä tarpeeksi. Itseltäni löytyy kuitenkin monia ideoita uusille kentille, joten tarkoitus olisi saada peli vielä täysin valmiiksi ja tarjolle Google Playhin. Olen miettinyt esimerkiksi haasteosion lisäämistä peliin. Siinä jokaista kenttäteemaa kohden olisi noin kymmenen lyhyttä kenttää, jotka kaikki olisi mahdollista läpäistä. Tulokeinona pelissä olisi tarkoitus käyttää mainostuloja eli peli tulisi olemaan käyttäjälle ilmainen.

Seuraavaan peliprojektiin osaamiseni GameMakerissä, Androidissa ja yleisesti mobiilipelien toteuttamisessa on huomattavasti laajempaa, kuin ennen opinnäytetyötäni. Onnistuin siis mielestäni saavuttamaan kaikki tälle työlle alussa asettamani tavoitteet hyvin.

LÄHTEET

Chrome Coders 2013. Mobile Game Design. Viitattu 11.12.2014 http://www.indiegamepod.com/mobile-game-design-book.pdf.

Developers 2015. Switching Apps. Viitattu 14.2.2015 https://developer.android.com/training/articles/memory.html#SwitchingApps.

Fields, T. 2014. Mobile & Social Game Design: Monetization Methods and Mechanics. Florida: CRC Press.

GMW 2014. GameMaker Versions. Viitattu 1.12.2014 http://gamemaker.wiki/game-maker-versions.

Gregory, J. 2014. Game Engine Architecture. Florida: CRC Press.

Habgood, J.; Nielsen, N.; Rijks, M. & Crossley, K. 2010. The Game Maker's Companion. Game Development: The Journey Continues. New York: Apress.

Habgood, J. & Overmars, M. 2006. The Game Maker's Apprentice. Game Development for Beginners. New York: Apress.

MGF 2015. Always On the Move. A Histroy of Mobile Gaming. Viitattu 10.1.2015 http://www.globalmgf.com/wp-content/uploads/2015/01/The-History-of-Mobile-Gaming_ebook_MGF.pdf.

Phonegg 2015. Compare Phones. Viitattu 14.2.2015 www.phonegg.com > Compare Phones

TechAdvisory 2013. Processors: Computer vs mobile. Viitattu 14.2.2015 http://www.techadvisory.org/2013/12/processors-computer-vs-mobile.