KYMENLAAKSON AMMATTIKORKEAKOULU
Tietotekniikka / ohjelmistotekniikka

Sami Husso

2D-PELIOHJELMOINTI UNITYÄ KÄYTTÄEN

Opinnäytetyö 2013

TIIVISTELMÄ

KYMENLAAKSON AMMATTIKORKEAKOULU
Tietotekniikka

HUSSO, SAMI	2d-peliohjelmointi Unityllä
Opinnäytetyö	34 sivua
Työn ohjaaja	lehtori Teemu Saarelainen
Toimeksiantaja	Kymenlaakson ammattikorkeakoulu, Gamelab
Toukokuu 2013	
Avainsanat	peliohjelmointi, pelisuunnittelu, Unity, c#, iOS

Unity on peliohjelmointiympäristö, jonka avulla pelintekijä pystyy kehittämään pelejä monelle eri alustalle. Unity on saatavilla monelle eri käyttöjärjestelmälle. Unityn ohjelmointi perustuu script-tiedostoihin, joita voidaan ohjelmoida kolmella eri ohjelmointikielellä.

Tässä opinnäytetyössä oli tarkoituksena suunnitella ja ohjelmoida toimiva peli Unityllä iOS-käyttöjärjestelmälle iPhone-puhelimelle käyttäen C#-ohjelmointikieltä. Opinnäytetyössä perehdytään aluksi pelisuunnitteluun sitten tarkemmin Unityyn ja lopuksi tutkitaan miten pelin toiminnot saadaan aikaiseksi käytännössä. Opinnäytetyö on tehty Kymenlaakson ammattikorkeakoulun Gamelabille.

Opinnäytetyön pelin ohjelmoinnin aikana selvisi Unityn helppokäyttöisyys aloittelevalle peliohjelmoijalle sen laajan internetissä olevan manuaalin ansiosta. Unityn suosio peliohjelmoijien keskuudessa on myös mahdollistanut aktiivisen keskustelupalstan syntymisen, jossa ongelmia ratkotaan. Näiden avulla projektin aikana saatiin aikaiseksi pelimekaniikaltaan toimiva peli.

Tulevaisuudessa peliä on mahdollista kehittää aina julkaisukelpoiseksi asti. Pelin suunnittelu- ja ohjelmointivaiheessa on otettu huomioon, että pelinkehitys voi jatkua saumattomasti lisäominaisuuksien ohjelmoinnin muodossa. Unity mahdollistaa myös helpon grafiikan ja äänien lisäämisen, joita ei tämän projektin aikana peliin tehdä.

ABSTRACT

KYMENLAAKSON AMMATTIKORKEAKOULU
University of Applied Sciences
Information Technology

HUSSO, SAMI	2d Programming Using the Unity Game Engine
Bachelor's Thesis	34 pages
Supervisor	Teemu Saarelainen, Senior Lecturer
Commissioned by	Kymenlaakso University of Applied Sciences, Gamelab
May 2013	
Keywords	game programming, game design, Unity, C#, iOS

Unity is a game development platform available for many operating systems. Unity can be used to program games for many different platforms including mobile devices, gaming consoles and home computers. Programming with Unity is script-based and three different programming languages can be used with Unity.

The thesis was commissioned by the Gamelab of Kymenlaakso University of Applied Sciences. The objective for this thesis was to program a working game with Unity using the C# programming language. The game was designed specifically to run on iOS devices and iPhone. In this study some game designing aspects used to design the game were considered first. After the design phase Unity and its uses were explained in more detail. Finally all the decisions made in the design phase were put to practice and explained.

During programming the game it was made clear how easy Unity was for an inexperienced game programmer. Unity provides a large documentation of its classes and functions in the internet. Also an active forum is hosted by Unity. This forum offers many different solutions to many problems. Using these resources in addition to the sources mentioned in the bibliography provided a good source of information.

The game was designed to be easily improved with a possible release in mind. This was necessary because some aspects, particularly the audio and graphics could not be finished during the making of this project because of lacking skills and time.

SISÄLLYS

TERMIT JA LYHENTEET

iOS	Käyttöjärjestelmä, joka on käytössä kaikissa Applen valmistamissa älypuhelimissa sekä tableteissa.
OSX	Käyttöjärjestelmä, joka on käytössä Applen valmistamissa Mac-tietokoneissa.
Unity	Pelinkehitysympäristö, jolla tässä opinnäytetyössä käsitelty peli on tehty.
C# (C Sharp)	Oliopohjainen ohjelmointikieli, jolla tässä opinnäytetyössä käsitelty peli on ohjelmoitu.
JavaScript	Yksi kolmesta ohjelmointikielestä, jolla ohjelmointi on mahdollista Unityssä. JavaScript on yleisessä käytössä selaimissa ja perustuu oliopohjaiseen Java-kieleen.
Boo	Yksi kolmesta ohjelmointikielestä, jolla ohjelmointi on mahdollista Unityssä. Perustuu Python-ohjelmointikieleen.
Xcode	Applen ohjelmistonkehitysympäristö, jota tarvitaan iOS-laitteelle oman ohjelman asentamisessa.
Prefab	Prefab, prefabricated eli esivalmistettu. Objekti Unityssä, joka on valmiiksi käytettävissä ja liitettävissä peliin.
Scene	Unityn nimitys alueelle, joka on kerralla näkyvissä ja ladattuna pelissä.
Script	Ohjelmoitu osa pelistä, joka mahdollistaa toimintoja jollekin objektille.
Rigid body	Unityn käyttämä termi fysiikkalaskennassa jäykälle objektille. Tarkoittaa objektia, joka ei muuta muotoaan, vaan liikkuu siihen kohdistuvien voimien vuoksi.
Collider	Rigid bodyyn liitetty komponentti, joka määrittelee muodon, johon jonkin toisen colliderin osuma huomataan.

1 JOHDANTO

Tämä opinnäytetyö esittelee 2d-peliohjelmointia iOS-käyttöjärjestelmille peliohjelmointialusta Unityä käyttäen. Opinnäytetyöhön liittyvän pelin ohjelmointi aloitettiin kesällä 2012, jolloin peli-idea syntyi ja Kymenlaakson ammattikorkeakoulun Gamelab hyväksyi työn opinnäytetyöksi. Unityn valinta ohjelmointialustaksi oli luonteva sen ollessa jo aikaisemmasta kenttäsuunnitteluprojektista tuttu. Samalla ohjelmointikieleksi valikoitui C# Javascriptin ja Boon sijasta. C# on lähempänä Javaa, joka on tekijälle tutumpi aikaisempien kurssien ja projektien takia. iOS-laitteen omistaminen vaikutti päätökseen tehdä peli juuri sille käyttöjärjestelmälle. iOS-käyttöjärjestelmä on myös suosituimpia käyttöjärjestelmiä älypuhelimissa ja tableteissa. Ensimmäinen vaihe kesällä 2012 opinnäytetyössä oli Unityyn tutustuminen syvemmin ja tähän projektiin tarvittavan tiedon kerääminen. 2d-peliohjelmoinnin erityisominaisuuksia esimerkiksi törmäyksentunnistuksen suhteen tuli tutuksi jo tässä vaiheessa. Alussa ohjelmointi suoritettiin Unityllä Windows-ympäristössä, koska iOS-kohtaisia ohjelmointityökaluja ei tässä vaiheessa vielä tarvittu. Ohjelmointi erityisesti iOS-laitteelle aloitettiin vuoden 2013 tammikuussa. Tällöin ohjelmoinnin käyttöjärjestelmäksi jouduttiin vaihtamaan OSX, jotta peli pystyttiin ajamaan testausta varten iOS-laitteella. Kaikki iOS-laitteella toimivat ohjelmat joudutaan käynnistämään Xcode-nimisen Applen kehitysympäristön kautta. Tämä kehitysympäristö on saatavilla ainoastaan OSX-laitteilla.

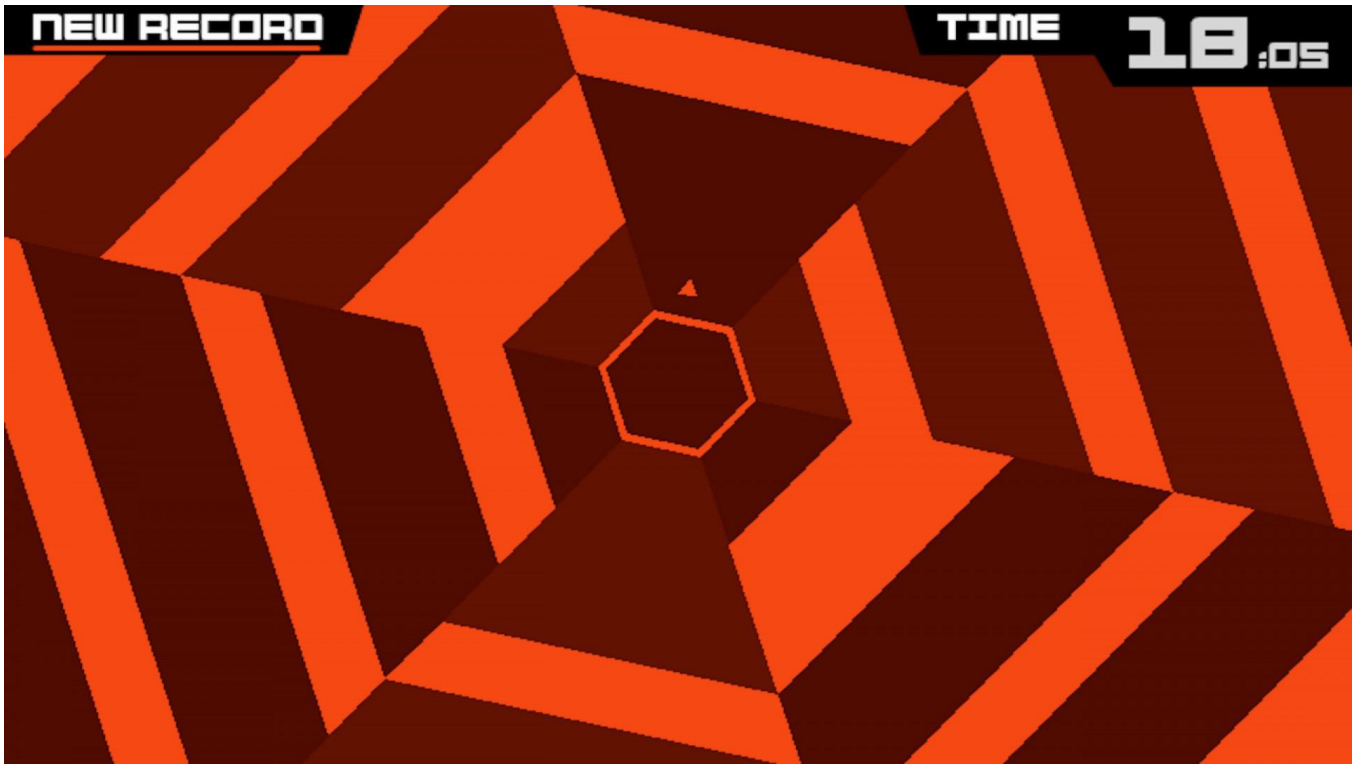
Unityn avulla peli tulee toimimaan iOS-käyttöjärjestelmissä, joka on käytössä Applen iPhonessa ja iPadissa. Peli ottaa syötteitä pelaajalta iOS-laitteen kiihtyvyysanturista. Kiihtyvyysanturin arvojen perusteella hahmo liikkuu ympäri paikallaan pysyvää kenttää. Kenttään ilmestyy satunnaisesti väistettäviä esteitä sekä pelaajan hahmon kykyjä parantavia esineitä. Näiden lisäksi kentässä on aina yksi kerättävä esine, josta saa pisteitä. Osuminen seinään tai esteeseen johtaa elämän menetykseen. Pelin perimmäisenä tarkoituksena on selvitä mahdollisimman pitkään ja saada mahdollisimman paljon pisteitä. Peli on tarkoitus suunnitella sellaisella tavalla, että pelikerran pituus on optimaalinen tutkimusten mukaan. Pelistä ei tule täysin valmis opinnäytetyön tekoaikana, vaan tarkoituksena on ainoastaan tehdä prototyyppiasteella oleva peli-idean toteutettavuuden osoitus. Myöskään grafiikka- eikä äänitiedostoja luoda tässä opinnäytetyössä, koska tekijällä ei ole tarvittavaa osaamista näillä aloilla.

Opinnäytetyön tavoitteena on valmistaa Kymenlaakson ammattikorkeakoulun Gamelabissä peli iOS-käyttöjärjestelmälle käyttäen Unity-ohjelmointialustaa. Pelin ei ole tarkoitus olla julkaisukelpoinen. Pelissä tulee kuitenkin olla ohjelmoituna erilaisia pelimekaanisia toimintoja, joiden avulla peli olisi prototyyppiasteella ja pienellä vaivalla julkaisukelpoiseksi päivitettävä. Myös lisäominaisuuksien, kuten uusien vihollisten ja powerupien ohjelmointi tulee olla helppoa. Suurin puutos tulee olemaan 3d-mallit, tekstuurit sekä äänet. On myös huomioitava, että kaikkien pelin vaikeuteen ja pelattavuuteen liittyvien muuttujien tulee olla helposti muokattavissa, jos jälkikehityksessä tällaiselle esiintyy tarve.

2 PELISUUNNITTELU

2.1 Yleistä pelisuunnittelusta

Lähtökohtana pelin suunnittelussa oli suunnitella peli, jonka käyttöikä olisi pitkä, mutta yksittäinen pelikerta olisi pituudeltaan hyvin lyhyt. Tämän tarkoitus olisi antaa pelaajalle mahdollisuus pelata yksi peli nopeasti missä tilanteessa tahansa. Pelikertojen lisäämiseksi tulisi pelin haasteen nousta nopeasti, jotta uutta sisältöä tulee eteen vain pienen etenemisen jälkeen. Esimerkki tämäntyylisestä pelistä on Super Hexagon. Siinä on tarkoitus väistellä esteitä, jotka tulevat eteen musiikin tahdissa.

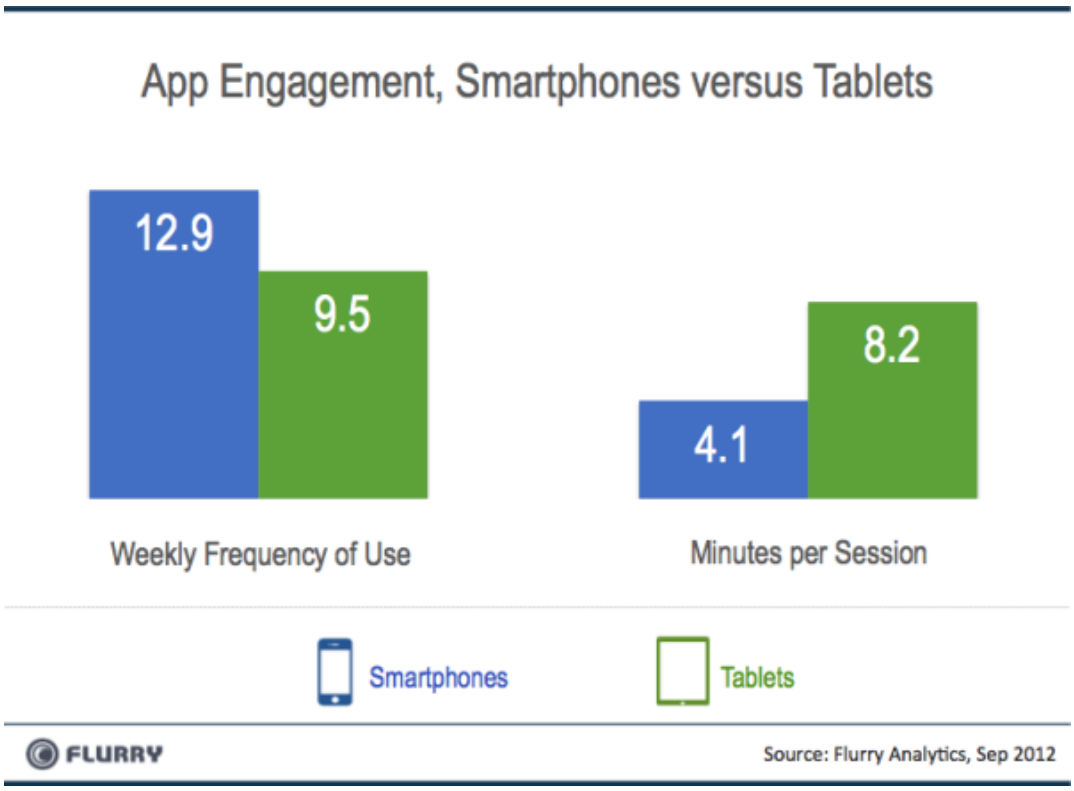


Kuva 1: Kuvakaappaus Super Hexagon-pelistä. Pelaaja on juuri tehnyt oman ennätyksensä 18,05 s.

Peli on hyvin korkeatempoinen ja yhden pelikerran maksimikesto on parhaimmilla pelaajilla noin 300 sekuntia. Aloittelijalla saattaa olla ongelmia kestää hengissä yli 10 sekuntia, mutta pelin kuitenkin oppii helposti ja edistystä tapahtuu nopeasti. Huipulle pääseminen on silti erittäin haastavaa ja vaatii harjoittelua. Super Hexagonissa siis pelaajien paremmuus mitataan ajalla, jonka pelaaja selviää hengissä. Tämän opinnäytetyön pelissä pelaajien paremmuus mitataan pistemäärällä. Tällaisella pelisuunnittelulla saadaan aikaiseksi tilanne, jossa lyhytkin pelisessio on tyydyttävä, koska pelaaja saavuttaa lyhyessä ajassa uuden maalin, hieman pidempään elossa pysymisen ja pisteiden keräämisen. Kuten Guy Lecky-Thompsonin Video Game Design Revealed –kirjassa mainitaan, jokaisessa pelissä on monia kenttiä tai vaiheita, joissa jokaisessa on oma maalinsa. (1, 210) Tämän opinnäytetyön peliprojektin kentät ovat nopeita vaiheita, joissa peli muuttuu uuden vihollisen ilmestyessä näkyviin ja maalina on siitä selviytyminen joko väistelemällä tai tuhoamalla vihollinen. Super Hexagonissa eri vaiheet on myös kuvattu väisteltävän monikulmion muodon muuttumisella esimerkiksi kuusikulmiosta viisikulmioon.

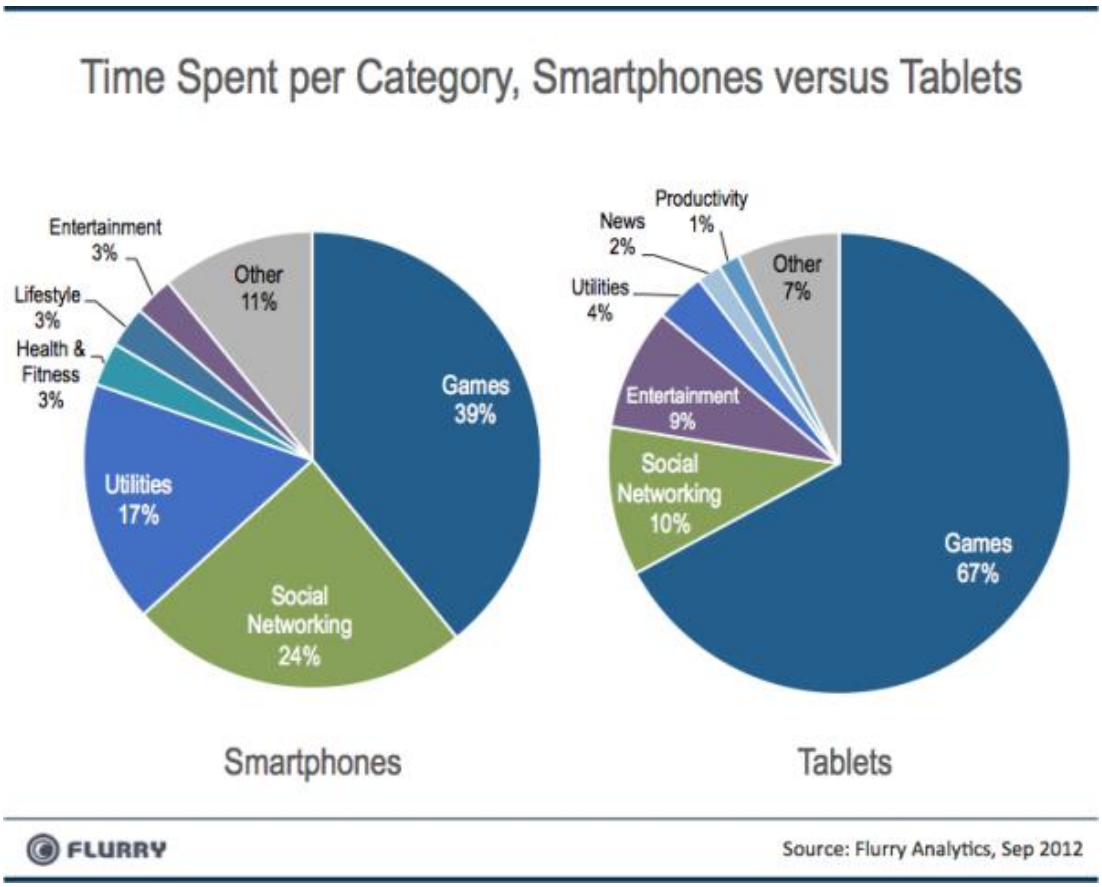
Flurry Analytics on maailmanlaajuinen tiedonkeräysyritys, joka kerää tietoa yli 100 000 yritykselle. Flurry Analyticsin tiedot kerätään yli miljardilta laitteelta, joiden käytön perusteella Flurry Analytics julkaisee erilaisia tietopaketteja omilla internetsi-

vuillaan. Mobiilipelissä lyhyt pelisessio on hyvä tavoite, koska Flurry Analyticsin tutkimusten perusteella älypuhelinta käytetään keskimäärin vain noin 4 minuuttia kerrallaan.



Kuva 2: Kuvaaja, joka esittää älypuhelimien ja tabletien käyttötapojen eroja. (2)

Tästä ajasta pelataan noin 40 %. Tabletteja käytetään keskimäärin pitempi aika, mutta käyttökertoja on vähemmän. Tabletteja käytetään myös suhteessa enemmän pelaamiseen kuin älypuhelimia. (2)



Kuva 3: Kuvaaja, joka esittää miten älypuhelimilla ja tableteilla käytetty aika jakaantuu eri tarkoituksiin. (2)

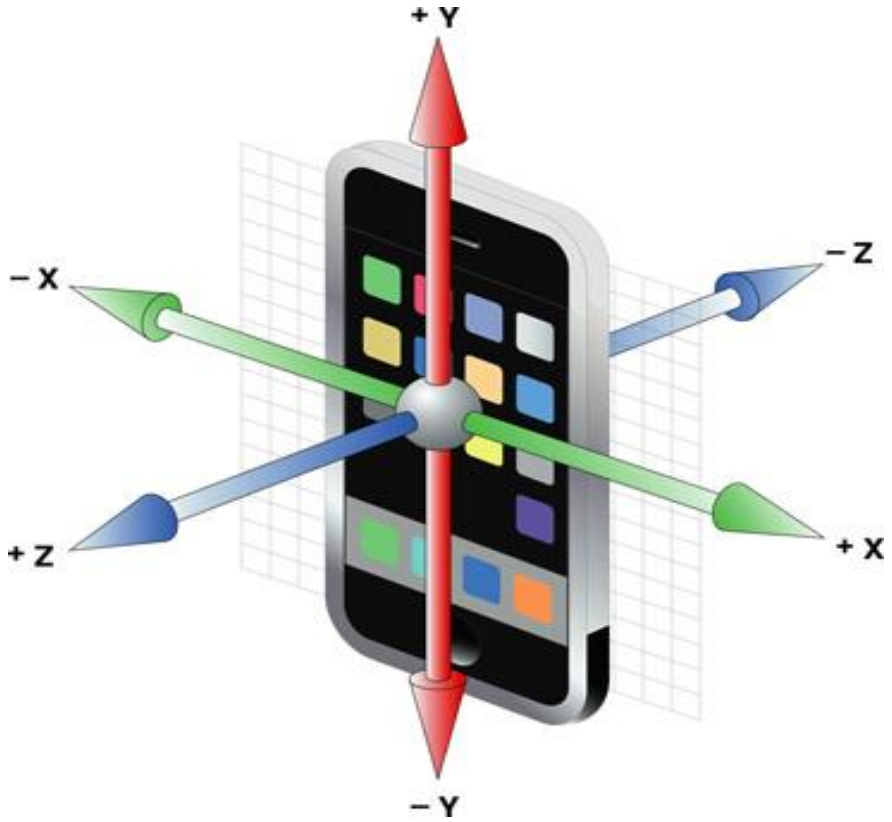
2.2 Esiteltävän pelin suunnittelu

Tämän opinnäytetyön peli kehitetään pääasiassa iOS-puhelimelle parempien testausmahdollisuuksien vuoksi. Paremmat testausmahdollisuudet johtuvat tekijän omistamasta iPhone 3GS-puhelimesta, jossa on iOS-käyttöjärjestelmä. Ohjelmoinnin testausvaiheessa peli on toiminnassa myös PC- ja Mac-tietokoneilla. Ohjelmointiin käytettävä alusta, Unity pystyy kääntämään ohjelman helposti mille tahansa iOS-laitteelle, mutta pelisuunnittelu tehdään älypuhelinta varten.

Nopean pelin aikaansaamiseksi pelissä pelaajan ohjaaman hahmon on tarkoitus väistellä ajan myötä vaikeampia vihollisia ja kerätä pisteitä, sekä powerupeja, eli hahmolle erilaisia päivityksiä antavia keräilyesineitä. Tämä hahmo liikkuu kentällä tarpeeksi nopeasti, jotta haastetta on riittävästi. Osuma viholliseen tai muuhun esteeseen aiheuttaa elämän menetyksen, ja pelaajalla on kolme elämää yhdellä pelikerralla. Elämien

määrää pystytään myöhemmin muuttamaan riippuen testauksen tuloksista. Kriittinen osa pelin vaikeudessa on myös vihollisen liike. Vihollisen tulisi yrittää liikkua mahdollisimman lähelle pelaajaa tarpeeksi vaikeasti ennakoitavissa olevalla liikeradalla. Vihollisia tulisi myös tulla pelikentälle enemmän pelin edetessä, jotta vaikeustaso pysyy haastavana.

Pelaajan apuna kentällä esiintyy satunnaisessa paikassa 10 sekunnin välein kerättävä powerup. Powerupiin osuessa pelaaja saa satunnaisesti jonkin positiivisen ominaisuuden. Tällä hetkellä erilaisia powerupeja on peliin ohjelmoituna lisäelämiä antava ja kolmen ammuksen ampumisen mahdollistava antava objekti. Ampuminen tapahtuu koskettamalla ruutua siitä kohdasta, johon pelaaja tahtoo ammuksen lähtevän liikkumaan. Jos ammus osuu viholliseen, vihollinen tuhoutuu. Ammus tuhoutuu itse osuessaan mihin tahansa objektiin paitsi pelaajaan. Pisteitä antava kerättävä esine ilmestyy uudelleen pelikentälle aina, kun pelaaja on siihen kerran koskenut. Osumalla objektiin pelaaja saa yhden pisteen.

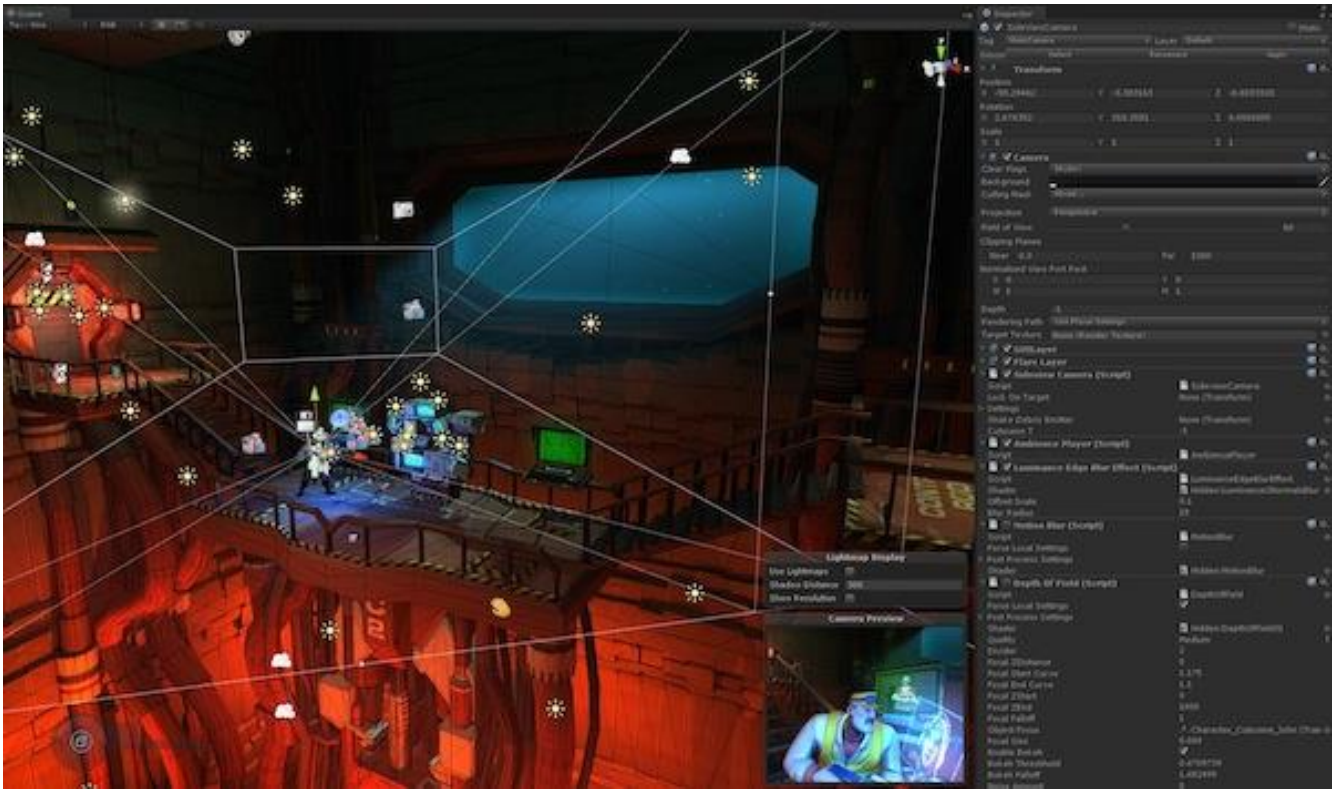


Kuva 4: Kuvassa iPhonen kiihtyvyysanturin akselit. (3)

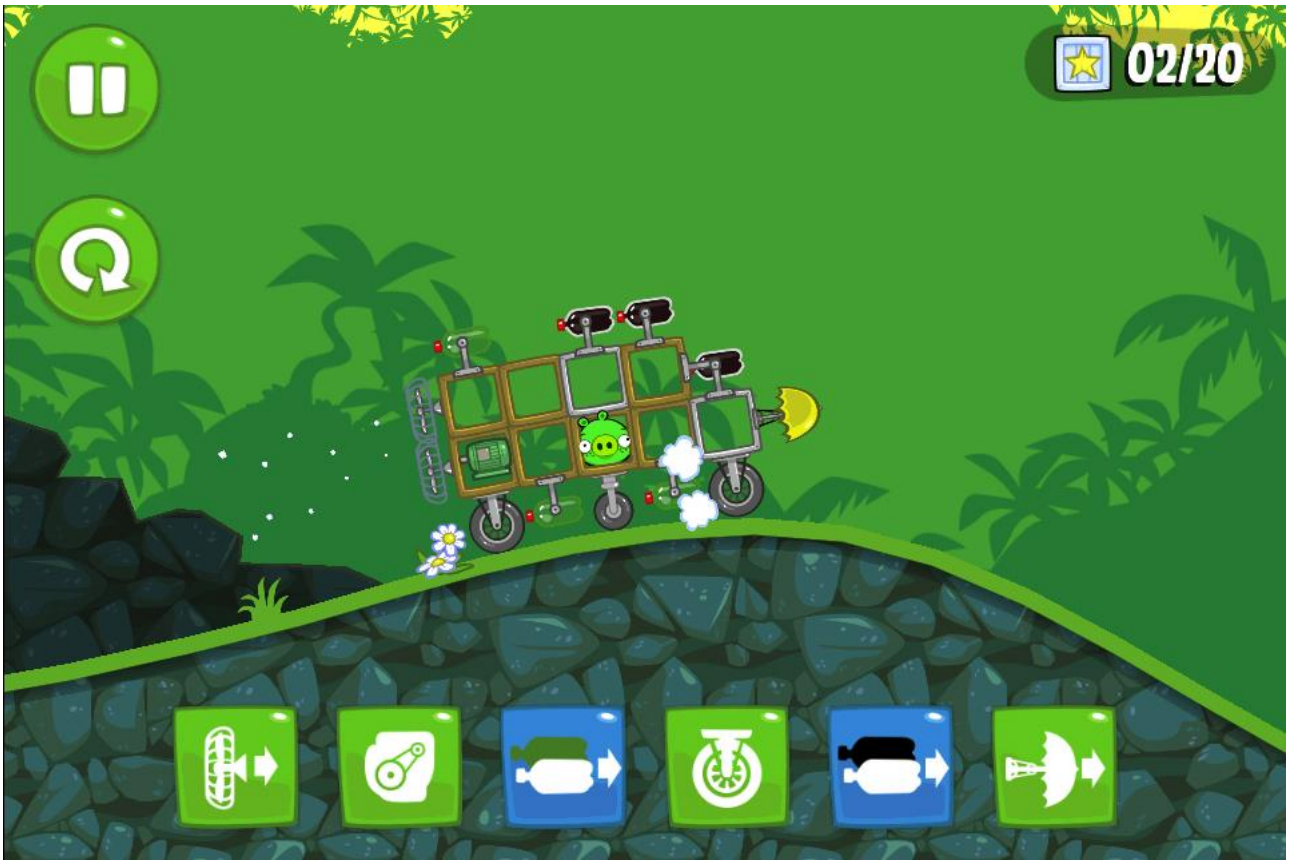
Pelissä ohjattavaa hahmoa ohjataan puhelimen tai tabletin kiihtyvyysanturilla, jonka pelille antamia syötteitä pelaaja pystyy hallitsemaan kallistelemalla puhelinta. iOS-laitteen kiihtyvyysanturin arvot vaihtelevat miinus yhden ja yhden välillä riippuen puhelimen asennosta maata kohti. Laitteen ollessa pöydällä näyttö ylöspäin on akselien arvot seuraavat; x = 0, y = 0 ja z = -1. Peli ottaa huomioon vain kuvassa näkyvät x- ja y-akselin lukemat ja näiden lukemien perusteella määritetään hahmon nopeus ja liikesuunta. Kallistettaessa laitetta oikealle, kasvaa x-akselin arvo suuremmaksi ja samalla pelissä hahmo liikkuu kallistuskulmasta riippuen tiettyä nopeutta oikealle. Esimerkiksi mitä suurempi x-akselin lukema on, sitä nopeammin hahmo liikkuisi kuvassa oikealle. (4) Pelaajan liikettä rajoittavat ainoastaan seinät ja vihollinen, joihin osuminen johtaa elämän menettämiseen. Seinät ovat pelikentällä neliön muotoisessa muodossa.

3 UNITY

Unity on monelle käyttöjärjestelmälle tarkoitettu pelinkehitysympäristö. Unitystä on olemassa sekä ilmainen versio sekä ammattikäyttöön tarkoitettu Pro-versio. (5) Unityllä pelinkehitys tapahtuu joko C#-, JavaScript-, tai Python-pohjaisella Boo-kielellä. Alun perin Unity on kehitetty pelkästään OSX-pohjalle, mutta nykyisin sillä pystyy kehittämään pelejä monelle käyttöjärjestelmälle, kuten esimerkiksi iOS:lle, Androidille, Windowsille ja OSX:lle. Nykyisin Unity on yksi suosituimmista pelinkehitysympäristöistä ja vuonna 2012 se ylitti miljoonan rekisteröidyn käyttäjän merkkipaalun.(6)



Kuva 6: Kuvankaappaus Rochardin kehityksestä Unityllä (7)



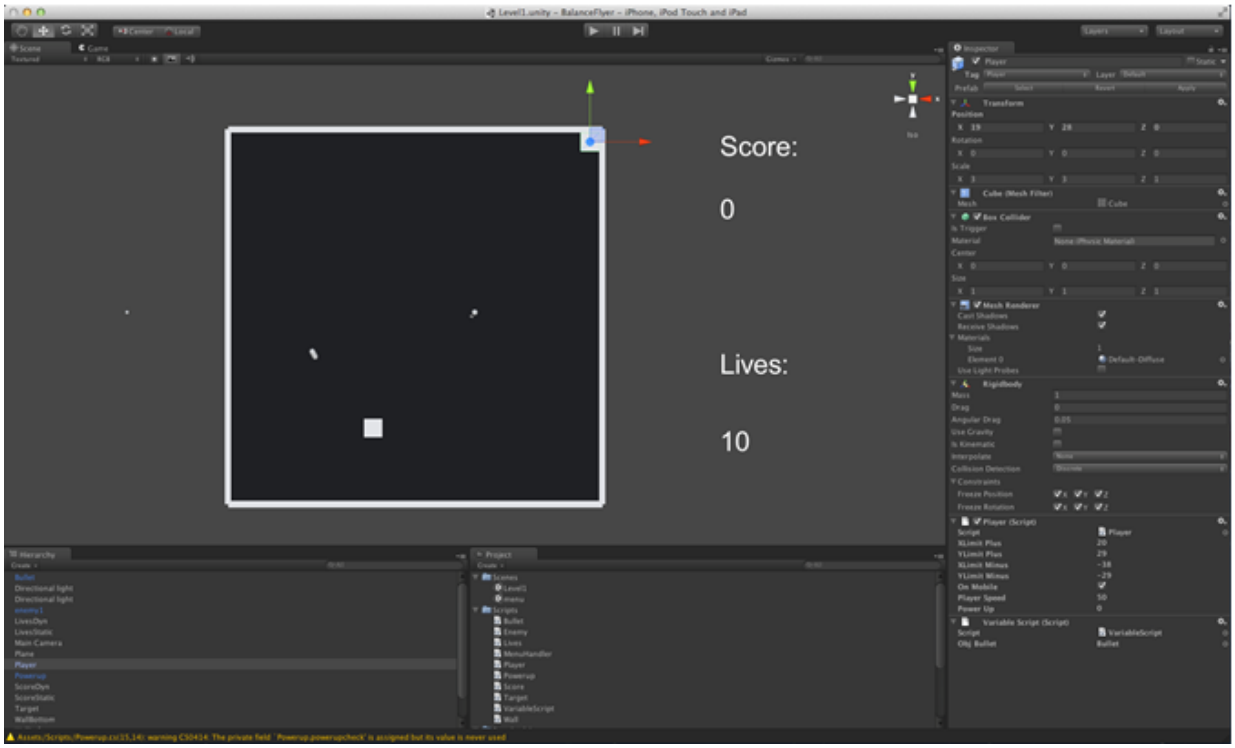
Kuva 5: Kuvankaappaus pelistä Bad Piggies, joka on kehitetty Unityllä. (8)

Unityllä kehitettyjä pelejä ovat muun muassa suomalaiset Rochard ja Bad Piggies, sekä suosittu kauhupeli Slender: The Eight Pages, avaruusraketin suunnittelu- ja rakentelupeli Kerbal Space Program ja avaruusstrategiapeli Endless Space. Rochardin tuottajan Kalle Kaivolan mukaan Unity oli paras vaihtoehto Rochardin kehittämiseen Unityn joustavuuden vuoksi (7). Bad Piggiesin kehittäjä, Rovion Jaakko Haapasalo mainitsee syitä Unityn käyttämiseksi esimerkiksi helpon siirtymisen prototyyppiasteelta julkaisukelpoiseksi peliksi (9).

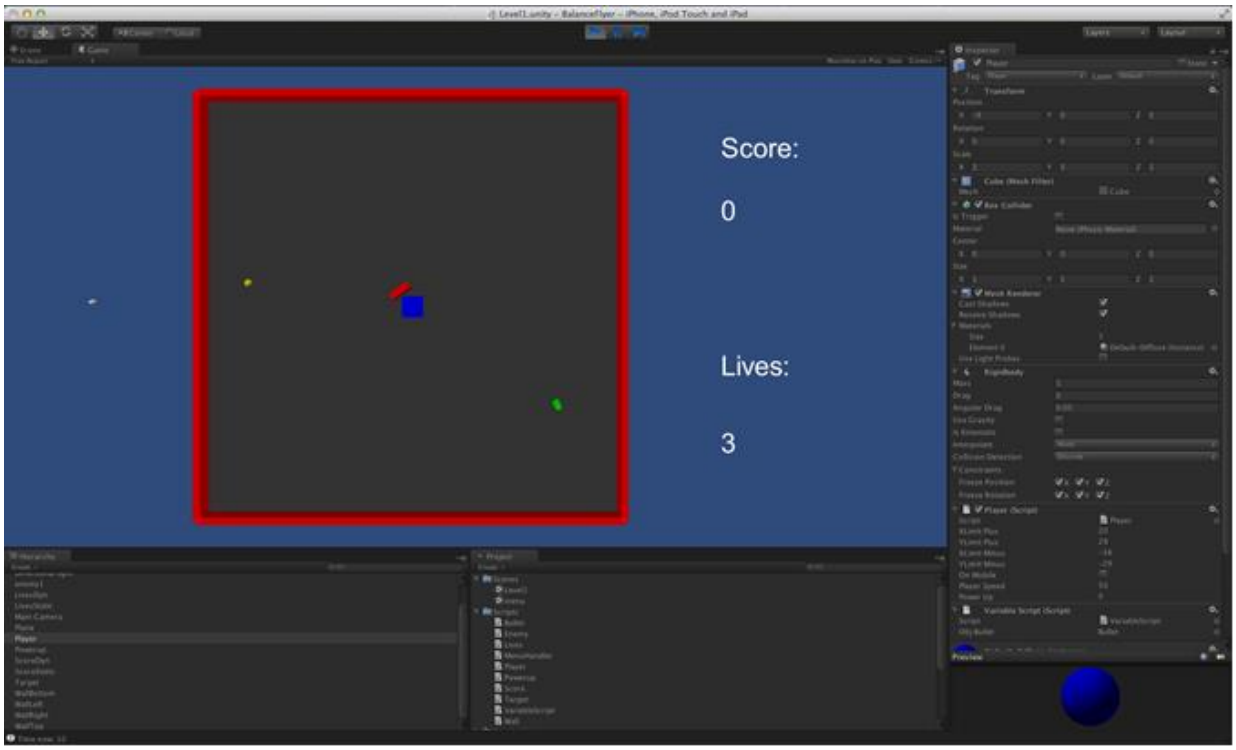
Unityllä ohjelmointi perustuu pitkälti script-tiedostoihin, joiden avulla pystyy luomaan erilaisia toimintoja pelissä oleville objekteille kuten prefabeille. Prefab on valmiiksi tehty peliin lisättävä objekti, jossa on yhdistettynä kaikki tarvittavat osat täydellisesti toimivaan peliobjektiin. Tämä vaatii, että prefabiin on liitetty jo muita Unityssä olevia komponentteja, kuten 3d-malli, tekstuurit ja script-tiedosto. Script-tiedostoissa varsinkin objektien monistukseen liittyvät funktiot vaativat, että monistettava objekti on jo valmis prefab. Lisää scripteistä ja prefabeista opinnäytetyön kappaleessa 3.4 ja 3.5.

3.1 Unityn käyttöliittymä

Unityn käyttöliittymä on modulaarinen, joten sen rakennetta pystyy muokkaamaan haluamakseen. Eri elementtejä on viisi erilaista. Ohjelmoitavan pelin näyttäminen on jaettu kahteen osaan, scene- ja game-elementteihin.



Kuva 7: Kuvankaappaus Unityn käyttöliittymästä, jossa näkyy keskellä Scene-elementti, alhaalla Hierarchy- ja Project-elementit ja oikeassa reunassa Inspector-elementti.



Kuva 8: Kuvankaappaus tilanteesta, jossa peli on käynnissä Unityn virheenilmoitustilassa. Keskellä näkyvissä Game-elementti, joka vastaa näkymää, jonka pelaaja näkisi pelissä.

Scene-elementissä kaikkia objekteja pystyy liikuttamaan ja kuvakulmaa muuttamaan, kun taas game-elementti näyttää pelin peliin asetetun kameran näkökulmasta. Unity mahdollistaa ulkopuolisten 3d-objektien lisäämisen peliin, mutta tässä opinnäytetyössä graafisen ammattilaisen puutteen takia käytetään ainoastaan Unityn valmiita yksinkertaisia 3d-objekteja. Kaikki peliin lisätyt objektit ovat listattuna project-elementtiin. Project-elementin sisältö vastaa reaaliaikaisesti projektikansion sisältöä käyttöjärjestelmän resurssienhallinnassa, joten tiedostoja pystyy liittämään Unityyn kopioimalla ne haluttuun kansioon. Project-elementin sisällön pystyy myös organisoimaan haluamallaan tavalla eri kansioihin. Hierarchy-elementti toimii project-elementin tukena, mutta näyttää ainoastaan peliobjektit, jotka ovat lisätty näkyvissä olevaan sceneen. Objektit näkyvät listana ja ovat valittavissa, jolloin scene-näkymässä valittu objekti näkyy korostettuna ja muokattavana. Valitun objektin tiedot näkyvät myös inspector-elementissä. Inspector-elementti on yleistyökalu, jossa on listattu kaikki valittuun peliobjektiin liitetyt komponentit kuten script-tiedostot, fysiikkamoottorin törmäyksentunnistustyypit, tekstuurit sekä 3d-malli. Kaikki liitetyt komponentit ovat myös muokattavissa suoraan inspector-näkymästä. Myös script-tiedostojen julkisiksi asetetut muuttujat ovat muokattavissa tässä näkymässä.

3.2 Scenet

Unityn pelinkehitysympäristössä scene tarkoittaa sitä tilaa, jonka peliobjektit ovat ladattuna peliin yhtäaikaa. Esimerkkinä käyköön seikkailupelissä luolasto, johon pääsee sisälle ovesta. Pelaaja ei pysty liikkumaan pelissä scenestä toiseen ilman latausaikaa. Scenellä ei ole varsinaista maksimikokoa, mutta latausajat on otettava huomioon scenen kokoa mietittäessä. iOS-pelissä tämä korostuu resurssien vähyyden takia, vaikka tämän opinnäytetyön esimerkkipelissä asiaa helpottaa pelin rakenne sekä peliobjektien yksinkertaisuus. Pienet scenet mahdollistavat myös tarkemman testausvaiheen. Myös pelin valikot on helppo tehdä omaksi sceneksi ja pelaajan syötteiden perusteella ladata ruudulle seuraava scene.(10, 15)

3.3 Komponentit

Komponentti on yleistermi Unityssä erilaisille osille, joista peliobjekti koostuu. Tärkeimpiä komponentteja ovat transform-komponentti, joka määrittää peliobjektin sijainnin, asennon ja koon, script-tiedostot, jotka ohjaavat peliobjektin toimintoja, 3d-malli, joka määrittää peliobjektin ulkomuodon tekstuuritiedoston kanssa. sekä rigidbody, joka kertoo fysiikkamoottorille tarpeellista tietoa peliobjektista kuten massan ja voiman esimerkiksi törmäyksiä varten. Rigidbodyyn liittyvä komponentti, collider, kertoo tarkemmin, mitä törmäyksen jälkeen tulee tapahtua. Collideriin voi asettaa muuttujaksi physics materialin, joka määrittää törmäyksen tyypin. Rigidbodyn physics material voi olla esimerkiksi kimpoava, jolloin peliobjekti kimpoaa osuessaan toiseen rigidbodylla varustettuun peliobjektiin. Myös esimerkiksi erilaiset valo- ja partikkeliefektejä tuottavat osat liitetään peliobjekteihin komponentteina. Näiden lisäksi pelinäkymän tuottava pääkamera on komponentti. Myös script-tiedostoilla ohjattavat lisäkamerat ovat mahdollisia komponentteja.(10, 15)

3.4 Script-tiedostot

Script-tiedostot ovat Javascriptillä tai C#:lla tehtyjä koodinpätkiä, jotka antavat toimintoja jollekin objektille. Pelissä olevia objekteja voi olla esimerkiksi tekoälyn ohjaama vihollinen, seinä johon voi törmätä ja ehkäpä yleisimmin pelaajan ohjaama hahmo. Unityssä on valmiina iso kirjasto, joka sisältää monia hyödyllisiä funktioita ohjelmoijan käyttöön. Tärkein Unityn kirjastossa olevista luokista on MonoBehaviour-luokka, joka sisältää Start()-, Update()- ja Awake()-funktiot. Näiden avulla peliobjektiin pystyy liittämään toimintoja, jotka käydään läpi kun peliobjekti käy läpi ensimmäisen ruudunpäivityksen, jokaisen ruudunpäivityksen kohdalla tai kun peliobjekti ladataan ruudulle ensimmäistä kertaa.

Muita hyödyllisiä funktioita ovat fysiikkamoottorin törmäyksentunnistukseen liittyvät OnCollision-funktiot. Myös pelaajan hiirellä tekemiin toimintoihin pystyy liittämään monia funktioita, kuten OnMouseOver sekä OnMouseDown. Hiireen liittyvät funktiot tarvitsevat kuitenkin aina törmäystunnistuksen scriptin isäntäobjektille, eli niitä ei voi

käyttää esimerkiksi klikkauksen koordinaattien laskemiseen, jos klikkaus tapahtuu itse peliobjektin ulkopuolella. Unity tarjoaa myös moninpelattaviin peleihin valmiita funktioita, kuten OnPlayerDisconnect ja OnServerInitialized.

3.5 Prefabit

Prefab on peliobjekti, jonka sisältö on tallennettu Unityn kirjastoon yhdeksi paketiksi. Tämän ansiosta prefab on siirrettävässä jopa toiseen peliin. Tärkein ominaisuus ohjelmoijan kannalta on se, että uuden prefabin voi script-tiedostossa kloonata. Näin esimerkiksi ammukset ja muut objektit, joita pitää olla mahdollista monistaa, täytyy olla prefabeja.(10, 16)

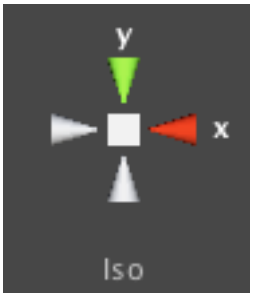
3.6 iOS-yhteensopivuus

Jotta ohjelmoidun pelin saa toimimaan Unityn oman virheenetsintätilan lisäksi iOS-laitteella, täytyy ensiksi pelaajan syötteiden lukua muuttaa niin, että ne ovat yhteensopivat iOS-laitteen kanssa. Tässä on eroja Android-laitteen kanssa siinä, että esimerkiksi Android-laite osaa lukea hiirelle ohjelmoituja toimintoja suoraan kosketuksina, kun taas iOS-laitteelle täytyy ohjelmoida erikseen kosketustoiminnot. Tämä johtuu iOS-laitteiden tavasta lukea kosketuksia ryppäinä. Kosketus on tavallaan siis vaihe, jonka aikana voi tapahtua monta kosketusta näytölle. Tässä opinnäytetyössä myös pelin hahmon liikkeet perustuvat kiihtyvyysanturin lukemiin, joten ne täytyi ohjelmoida kokonaan, ennen kuin peli oli edes testattavissa laitteella. Jotta peli pyörisi iOS-laitteella, täytyy Unity yhdistää samalla koneella asennettuna olevaan Xcode-ohjelmistoon, joka on Applen oma ohjelmointialusta iOS-laitteille. Tätä varten Unity vaatii erillisen lisäosan, joka on maksullinen yksityiskäyttäjälle. Lisäosan avulla Unity pystyy lähettämään Xcodelle pelin asennettavaksi puhelimeen. Ennen asennusta täytyy ohjelmoijalla kuitenkin olla Applen lisenssi, jonka avulla rekisteröidään ohjelmoijan käyttämä Xcode, itse peli, sekä kohdelaite. Lisenssin liittäminen laitteeseen ja Xcodeen onnistuu Applen tähän tarkoitukseen luodulla internetsivustolla. Tältä sivustolta ohjelmoijan täytyy ladata lisenssitiedosto, joka sitten liitetään Xcoden avulla sekä itse Xco-

deen, että haluttuun kohdelaitteeseen. Tämän jälkeen Unityyn kirjoitetaan nimi, jolla peli on lisensoitu Xcodeen ja lisäosan suoritustoiminnosta Unity käynnistää Xcoden, joka asentaa ja käynnistää pelin kohdelaitteeseen.

4 PELIN SUUNNITTELU UNITYLLE

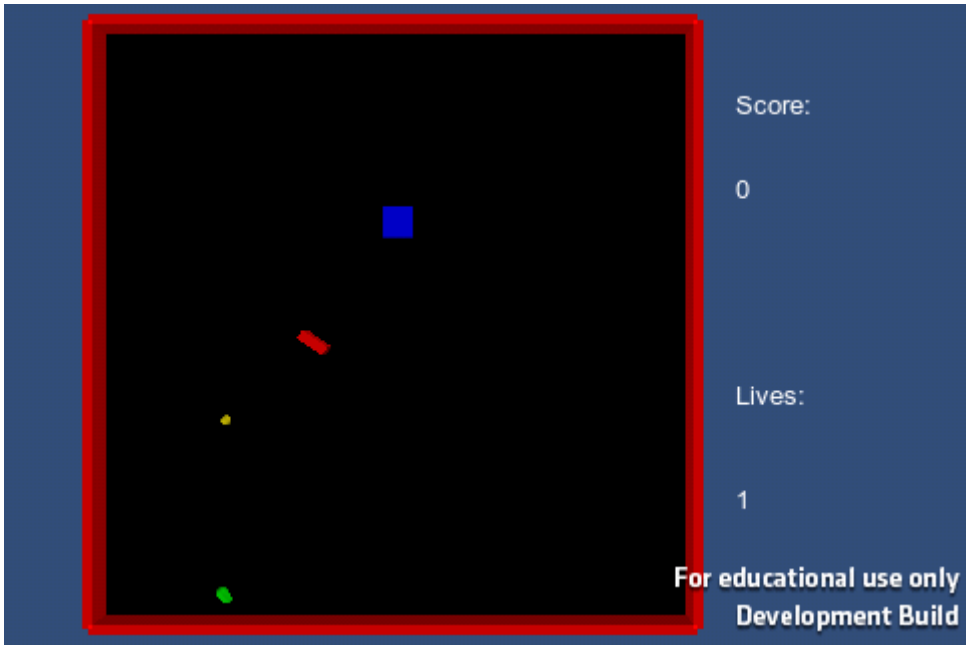
Koska Unity on lähtökohtaisesti kolmiulotteisessa ympäristössä toimiva pelinkehitysympäristö, tarjoaa se kaksiulotteiselle pelille monesti ohjelmoijalle yhden ylimääräisen ulottuvuuden. Esimerkiksi peliobjektien sijainnissa ja liikkeessä on aina otettava huomioon ainoastaan kaksi tarpeellista akselia. Ylimääräinen akseli mittaa syvyyssuuntaa, ja se joudutaan asettamaan nollaksi monessa toiminnossa. Esimerkiksi kaikki koordinaatit ovat Unityssä ilmoitettu kolmen pisteen avulla ja tästä johtuen liikettä laskettaessa vektoreilla on käytettävä kolmiulotteisia vektoreita, vaikka yksi vektorin arvo onkin aina nolla. Ohjelmoija voi kuitenkin päättää aloitusvaiheessa mikä Unityn akseli asetetaan kuvaamaan syvyyssuuntaa ja tässä projektissa pelin kuvakulma on valittu niin, että Unityn z-akseli kuvaa syvyyttä.



Kuva 9: Kuvankaappaus Unityn koordinaatteja kuvaavasta symbolista.

Unityllä ohjelmoiminen perustuu sceneihin, joissa tapahtuu kaikki mitä pelaajalle pelistä näkyy. Scenejen laajuus on tärkeä asia varsinkin mobiililaitteille ohjelmoitaessa resurssien pienuuden takia. Tässä opinnäytetyössä 3d-objektit ovat kuitenkin hyvin yksinkertaisia, eikä niitä tule olemaan ruudulla yhtä aikaa monia. Myös pelin rakenne nopeasti suoritettavana yhtenä kokonaisuutena puoltaa päätöstä tehdä koko pelin sisältö valikkoa lukuun ottamatta yhteen sceneen. Pelissä on siis kaksi sceneä, valikko, joka on pelaajalle ensimmäisenä näkyvä asia. Tästä valikosta voi käynnistää itse pelin tai lukea ohjeet ja tekijätiedot. Toinen scene on itse peli. Koska valikossa ei tarvitse

näkyä mitään itse pelissä olevia peliobjekteja, olisi turhaa yhdistää nämä scenet. Vaikka valikko on pieni kokonaisuus ja sisältää vain tekstiä, olisi turhaa ladata ne peliin, koska myöhemmässä vaiheessa näitä valikkoja ei tarvitse näkyä. Pelissä ei ole selkeitä vaiheita, joissa pelialue muuttuisi, joten monia scenejä ei olisi yksinkertaisesti toteutettavissa. Opinnäytetyön jälkeisessä mahdollisessa kehitystyössä voi kuitenkin tulla kysymykseen uusien kenttien luonti peliin, jolloin nämä maailmat on järkevin tehdä omaan sceneensä. Scenejen välillä on kuitenkin aina pieni latausaika, joten tämä on otettava huomioon lisäkenttiä suunniteltaessa.



Kuva 10: Kuvankaappaus pelistä iPhone 3GS-laitteella. Kuvassa näkyvät pelaaja sinisenä, vihollinen ja seinät punaisena, powerup keltaisena ja kerättävä piste vihreänä.

Peliobjekteja suunnitellessa ensimmäisenä tuli tehdä päätös pelialueen muodosta ja koosta. Pelialue olisi voinut myös olla pelinäkymää suurempi, jolloin kamera olisi seurannut pelaajaa, mutta vaikeuden lisäämiseksi ja ahtauden tunteen aikaansaamiseksi pelialue on neliö, joka näkyy kokonaan ruudulla. Samalla peliin liittyvät tilastot, kuten pisteet ja jäljellä olevat elämät näkyvät pelialueen ulkopuolella suorakaiteen muotoisen näytön ansiosta.

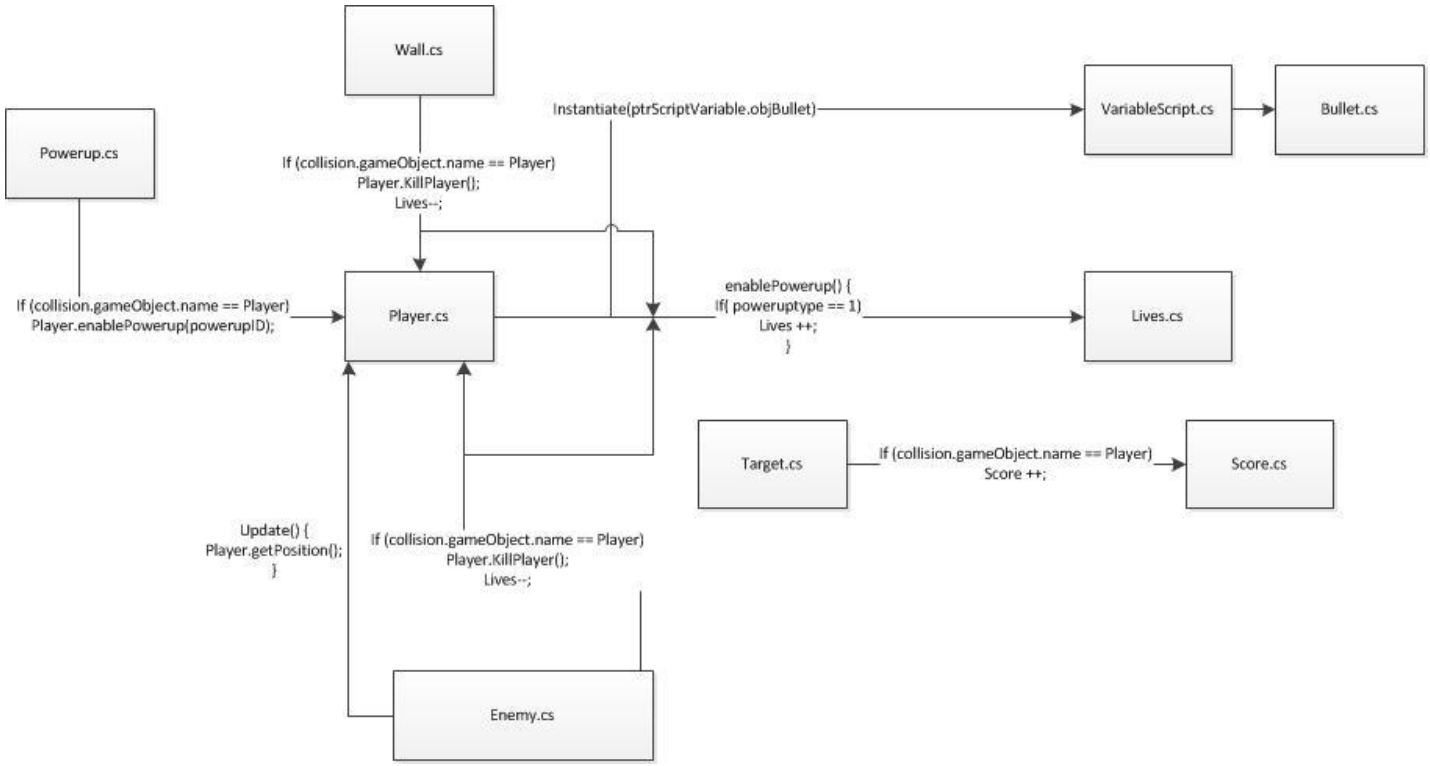
Erilaisilla script-tiedostoilla määritellään toiminnot jokaiselle pelissä esiintyvälle peliobjektille. Script-tiedostot ovat selvyyden vuoksi nimetty niiden peliobjektien mukaan, joihin ne ovat liitettyinä. Poikkeuksia tähän on MenuHandler.cs, joka hallitsee

ainoastaan pelin valikkonäkymää ja VariableScript.cs, jonka avulla monistetaan ammuttuja ammuksia tarpeen mukaan.

Peliobjekteille, joiden on reagoitava niihin osuttaessa jonkin toisen objektin toimesta, täytyy olla liitettynä collider-komponentti. Tällaisia peliobjekteja ovat pelaaja, vihollinen, powerup ja kerättävät pisteet, mutta ei pistelaskurit tai muut tekstiobjektit. Jokaisen objektin colliderin muoto voi olla yksinkertainen, koska peliobjektien koko ei ole niin iso näytöllä, että pelaaja huomaisi. jos colliderin muoto ei täsmää täydellisesti objektin 3d-mallin kanssa.

5 SCRIPTIEN KÄYTTÖ PELISSÄ

Pelin kaikki toiminnot on määritetty script-tiedostoissa, jotka ovat liitetty asianmukaisiin peliobjekteihin. Tässä projektissa tärkeimmät peliobjektit ovat pelaajan ohjaama hahmo, kerättävät pisteobjektit, kerättävät powerup-objektit sekä vihollishahmo. Jokaisella näillä on oma script-tiedosto liitettynä ja pelaajan hahmolla kaksi script-tiedostoa. Myös huomaamattomimmilla objekteilla, kuten seinillä on omat script-tiedostot.



Kuva 11: Kaavio script-tiedostojen välisestä toiminnasta.

5.1 Player.cs

Player-scripti toteuttaa toimintoja, jotka koskevat suoraan pelaajan ohjaamaa objektia. Player.cs:n rakenne on normaalin script-tiedoston kaltainen, eli siinä käytössä start- , awake- ja update-funktiot. Player-scriptin oleellisin tehtävä on liikuttaa pelaajan hallitsemaa peliobjektia pelaajan antamien syötteiden avulla. Tämä on toteutettu kahdella eri tavalla riippuen siitä, onko peli käynnissä iOS-pohjaisessa laitteessa vai PC- tai Mac-tietokoneessa. iOS-pohjaisessa laitteessa syötteet tulevat suoraan puhelimen kiihtyvyysanturista. Pelaaja pystyy hallitsemaan kiihtyvyysanturin antamia arvoja kallistamalla puhelinta ja näin liikuttaa peliobjektia pelialueella aivan kuten se liikkuisi todellisessa maailmassa painovoiman vaikutuksesta puhelimen pinnalla.

Ensiksi player-scriptissä määritellään muutamia globaaleja muuttujia. Asettamalla globaalit muuttujat public-tyyppisiksi on mahdollista muuttaa muuttujien arvoja Unityn käyttöliittymästä suoraan, koskematta itse scripti-tiedostoon. Näin on toimittu sellaisten muuttujien kohdalla, joiden muuttuminen on mahdollista jokaisella virheiden etsintäkerralla. Unityn fysiikkamoottorin toiminnan takia pelaajan hallitsemalle hahmolle pitää asettaa selkeät rajat joiden ulkopuolelle hahmo ei pääse koordinaatistossa. Koska kyseessä on 2d-peli, niin myös syvyyssuuntaa kuvaava z-akseli on lukittava nollaksi, jotta hahmo ei karkaa ulos pelikentältä jonkin odottamattoman törmäyksen vaikutuksesta. Muita globaaleja muuttujia ovat pelaajan nopeus, pointer-määrityksiä sekä laskentaan käytettäviä vektoreja.

5.1.1 Awake() ja Start()

Ensimmäinen funktio player-scriptissä on awake(), joka kutsutaan kun peliobjekti ladataan peliin. Awake():ssa määritetään peliobjektille tekstuuriksi pelkkä sininen väri, jotta se erottuu muista kentällä olevista objekteista. Awake():n jälkeen scriptissä on start()-funktio jota kutsutaan aina ennen kuin Update()-funktiota kutsutaan ensimmäisen kerran. Start()-funktiossa asetetaan Transform-tyyppinen _t-muuttuja Vector3-tyyppiseksi. Vector3 on vektori jolla on kolme attribuuttia, jonka vuoksi se toimii

kolmiulotteisessa koordinaatistossa. Transform-muuttuja on jokaisella peliobjektilla oleva muuttuja, joka pystyy kertomaan esimerkiksi peliobjektin sijainnin koordinaatistossa. Transformin arvoja muuttamalla peliobjektia pystyy myös liikuttamaan kentällä.

5.1.2 Update()

Update-funktiossa on kolme eri toimintoa, jotka suoritetaan jokaisen ruudunpäivityksen aikana. Ensimmäinen ja huomattavin on pelaajan hahmon liikuttaminen. Tämä tapahtuu kahdella eri tavalla riippuen pelaajan käyttöjärjestelmästä. _t Vector3-vektorimuuttujaa muutetaan pelaajan syötteiden perusteella ja tarkastetaan, ettei hahmo ole liikkunut pelialueen ulkopuolelle. Myös syvyysakseli z pidetään aina nollassa lukittuna. PC:llä tai Macilla pelatessa syötteet tulevat nuolinäppäimistä, joiden painallukset Unity havaitsee Input-luokan GetAxis-funktion avulla. GetAxis-funktiossa on horisontaalinen ja vertikaalinen arvo, joiden avulla Transform-muuttujan arvoja muutetaan ja näin liikutetaan hahmoa. GetAxis-funktion arvojen lisäksi Transform-muuttujan arvoihin kerrotaan pelaajan nopeus sekä ruudunpäivitykseen menevä aika, jotta ruudunpäivityksen hidastuminen ei vaikuta pelaajan liikkumisnopeuteen.

Pelaajan pelatessa iOS-laitteella Transform-muuttujaa käsitellään puhelimen kiihtyvyysanturin lukemien perusteella. Eroavaisuuksia näppäimistöltä syötteen lukemiseen on ainoastaan funktio, josta syöte otetaan. iOS-laitteen kiihtyvyysanturin arvot löytyvät Input-luokan acceleration-funktiosta GetAxis-funktion sijaan. Koska iOS-laite on pelatessa vaaka-asennossa, joudutaan ottamaan huomioon koordinaatistojen muutokset, joten peliobjektin x-koordinaatiin lisätään iOS-laitteen y-akselin muutos miinusmerkkisenä ja peliobjektin y-koordinaattiin lisätään iOS-laitteen x-akselin muutos plusmerkkisenä. Varsinkin iOS-laitteilla on tärkeä lisätä jokaiseen kertolaskuun TimedeltaTime, joka on ruudunpäivitykseen kuluva aika, koska iOS-laitteilla ruudunpäivitys voi heitellä enemmän. Jos tätä ei ota huomioon pelaajan hahmo näyttää ajoittain liikkuvan erittäin hitaasti, kun ruudunpäivitys tapahtuu esimerkiksi puhelimen taustatoimintojen takia hieman hitaammin.

Update()-funktion aikana hoidetaan myös ampuminen käyttöjärjestelmätarkastuksen jälkeen. Jos käytössä on iOS-laite, kutsutaan erillistä playerTouch()-funktiota, joka hoitaa ampumisen ja muut mahdolliset Touch-tapahtumat. Hiirenpainalluksella am-

puminen tapahtuu Update-funktiossa ottamalla talteen hiiren sijainti Input.mousePosition-funktion avulla ja hiirenpainalluksen tapahtuessa kutsutaan ShootBullet-funktiota.

5.1.3 onCollisionEnter()

Törmäyksentunnistus player-scriptista toimii onCollisionEnter()-funktion avulla, jota kutsutaan kun peliobjektin törmäystunnistuksen alue osuu johonkin toisen peliobjektin vastaavaan alueeseen. Funktion sisältö on yksinkertainen yhden elämän vähennys jos osumakohde on ollut vihollinen ja pelaajan uudelleensynnyttäminen pelialueen keskelle.

5.1.4 enablePowerup()

Powerupin käyttöönottofunktiota kutsutaan powerup.cs-scriptitiedostosa, jossa törmäyksentunnistus tapahtuu. Player-scriptissä ainoastaan aktivoidaan oikeantyyppinen powerup powerup.cs:stä tulevan attribuutin mukaan. Tällä hetkellä powerupeja on kaksi erilaista, joista toinen lisää pelaajalle yhden elämän ja toinen mahdollistaa ampumisen.

5.1.5 playerTouch()

playerTouch-funktiota kutsutaan Update-funktiossa jokaisen ruudunpäivityksen aikana, mutta playerTouch ei tee mitään ellei Touch-tapahtumaa eli kosketusta näytölle ole tapahtunut. Touch on monimutkainen tapahtuma sen takia, että yksittäistä kosketusta ei ole mahdollista iOS-laitteella havainnoida suoraan. iOS-laitteet sen sijaan havainnoivat vaiheita, jotka laskevat kosketusten määrää ja nollautuvat tietyn ajan päästä. Koska tämän pelin ohjelmoimisessa ei ole hyötyä kosketusten laskemisesta, playerTouch-funktion toiminta käynnistää ainoastaan jos kosketusten määrä on

enemmän kuin nolla. Tämän jälkeen jokaisen kosketuksen koordinaatit saadaan Input.GetTouch(i)-funktion position-attribuutista, jossa i on kosketuksen indeksi. Tämän jälkeen luodaan näkymätön nappi kosketuksen koordinaatteihin ja tarkastetaan, onko pelaajalla ampumisen mahdollistava powerup aktiivisena. Koska äskettäin luodun napin koordinaatit ovat identtiset kosketuksen koordinaattien kanssa, on nappiin aina osuttu ja sen koordinaatit voidaan säilöä ShootBullet()-funktiolle, jota kutsutaan viimeiseksi ja jossa itse luoti luodaan.

5.1.6 ShootBullet() ja getPosition()

Tässä funktiossa luodaan luoti Instantiate-funktion avulla. Luodille lasketaan suunta kosketuksen koordinaattien perusteella, luoti asetetaan myös olemaan etupuoli liikesuuntaan päin Quaternionion avulla ja lopuksi luodaan Bullet-prefab kulkemaan pelaajan koordinaatista kosketuksen koordinaatteihin. Quaternioiden avulla pystytään laskemaan objektin kolmiulotteisessa koordinaatistossa kääntyminen ilman matriisilaskentaa. Bullet-prefabin luominen Player-prefabin toiminnon perusteella vaatii pointterin, joka on määritelty Start()-funktiossa ja osoittaa toiseen script-tiedostoon, joka on myös liitetty Player-prefabiin. Tässä toisessa script-tiedostossa on ainoastaan määritettynä peliobjektiksi Bullet. Bullet-prefabille annetaan myös fysiikkamoottoria varten voimaa, jotta se liikkuu oikealla nopeudella.

getPosition-funktio palauttaa kutsuttaessa pelaajan tämänhetkisen sijainnin, jos sitä tarvitsee jossakin ulkopuolisessa script-tiedostossa.

5.2 Powerup.cs

Powerup-scripti synnyttää peliin erilaisia kerättäviä esineitä tietyn ajan välein. Kerättävien esineiden merkitys muuttuu satunnaismuuttujan avulla muutamasta eri vaihtoehdosta. Tällä hetkellä vaihtoehdot koostuvat ampumismahdollisuudesta, lisäelämästä ja hetkellisestä kuolemattomuudesta. Powerup-scripti rakentuu niin, että siihen on mahdollista lisätä uusia kerättäviä esineitä myöhemminkin. Script-tiedostossa on mää-

riteltynä globaaleina muuttujina ääriarvot koordinaatteihin, joihin powerup pystyy syntymään sekä ajanlaskemiseen käytettävät muuttujat.

5.2.1 Start()

Start-funktiossa asetetaan objektille väriksi keltainen erottuvuuden takaamiseksi. Samalla kuitenkin asetetaan objekti alussa näkymättömäksi ja fysiikkamoottorin kannalta läpikuljettavaksi, jotta siihen ei voi vahingossa osua ennen kuin on tarkoitus. InvokeRepeating-funktion avulla lasketaan aikaa siihen aikaan jolloin spawnPowerup-funktiota tulee kutsua. InvokeRepeating kutsuu tämän jälkeen samaa funktiota aina tietyin väliajoin, tässä tapauksessa 20 sekunnin välein.

5.2.2 spawnPowerup()

Tämän funktion tarkoitus on ainoastaan muuttaa powerup-objekti näkyväksi satunnaiseen sijaintiin pelikentällä ja fysiikkamoottorin kannalta sellaiseksi, että siihen voi osua. Tässä funktiossa myös satunnaisesti päätetään minkä tyyppinen powerup tulee olemaan ja talletetaan tieto powerupID-muuttujaan.

5.2.3 OnCollisionEnter()

Törmäyksentunnistuksessa huomioidaan ainostaan ne törmäykset, jotka tapahtuvat pelaajan kanssa. Tällaisen törmäyksen jälkeen player.cs-tiedoston enablePowerup-funktiota kutsutaan ja sille annetaan attribuutiksi powerupID-muuttujan tieto, jotta tiedetään minkälaisia vaikutuksia powerupilla on pelaajan ohjaamaan hahmoon. Tämän jälkeen kutsutaan removePowerup-funktiota, joka poistaa powerupin näkymättömiin pelaajalta ja fysiikkamoottorilta kunnes Update-funktion sisällä InvokeRepeating kut-

suu jälleen spawnPowerup-funktiota. Jos pelaaja ei ehdi 20 sekunnin aikana kerätä powerupia, syntyy se uudelleen satunnaiseen paikkaan.

5.3 Enemy.cs

Enemy-scriptin tarkoitus on hyvin samanlainen kuin player-scriptin, mutta se ohjaa pelaajan sijasta erilaisten vihollisten toimintoja. Tällä hetkellä on olemassa ainoastaan yksi vihollistyyppi, joka yrittää jahdata pelaajaa.

Globaaleiksi muuttujiksi määritetään laskemiseen tarvittavia vektoreita ja koordinaatit, josta vihollinen aloittaa. Awake-funktion aikana asetetaan ainoastaan vihollisen väri punaiseksi. Start-funktiota sen sijaan ei tarvita ollenkaan. Updaten aikana hankitaan pelaajan sijainti getPosition-funktion avulla ja tämän jälkeen kutsutaan AIinput-nimistä funktiota, jonka avulla vihollinen liikkuu aina pelaajaa kohti. Vihollisen törmäyksentunnistuksessa otetaan huomioon kaksi eri mahdollisuutta. Jos viholliseen törmää pelaaja, kutsutaan player-scriptin KillPlayer-funktiota, joka johtaa pelaajan elämän menettämiseen. Jos taas pelaaja osuu ammuksella viholliseen, tuhotaan vihollinen kokonaan. Enemy-scriptiin on myös lisätty spawnEnemy-funktio, jonka avulla pystyy luomaan uuden vihollisen, mutta se ei ole tällä hetkellä vielä käytössä.

AIinput-funktiossa tapahtuu itse vihollisen liikuttaminen. Liikkuminen tapahtuu laskemalla vektori vihollisen nykyisestä sijainnista pelaajan sijaintiin ja liikuttamalla vihollista tämän vektorin suuntaan tietyllä voimalla, jonka suuruus on suoraan verrannollinen etäisyysvektorin suuruuteen. Vihollinen myös käännetään aina menosuuntaansa päin käyttämällä hyväksi Quaternion-luokan LookRotation-funktiota.

5.4 Muut C#-tiedostot

Muiden script-tiedostojen tarkoitus on hyvin kapea ja niillä ohjataan hyvin pienien kokonaisuuksien toimintoja. Koska Unity pohjautuu siihen, että jokaisella peliobjektilla on oma script-tiedosto, syntyy pieniä script-tiedostoja paljon. Tämän voisi kiertää tekemällä yhden ison script-tiedoston, joka liitettäisiin jokaiseen peliobjektiin ja tarkistettaisiin aina peliobjektin nimen perusteella oikea toiminto. Selkeästi nimetyt

script-tiedostot ovat kuitenkin paljon helpommin ymmärrettävissä, vaikka ongelmaksi voi muodostua tiedon liikuttaminen script-tiedostojen välillä.

Target.cs ohjaa kentällä esiintyvän kohteen toimintaa. Kohteesta saa aina pisteitä kun siihen onnistuu osumaan. Target on toiminnaltaan samankaltainen powerupin kanssa, mutta sillä on aina sama toiminto eikä se häviä missään vaiheessa. Target asetetaan väriltään vihreäksi ja se syntyy satunnaisesti johonkin kohtaan pelialuetta välittömästi pelin alkaessa sekä pelaajan kerätessä targetin. Törmäyksentunnistuksessa synnytetään siis uusi target sekä annetaan pistetaulua ohjaavalle script-tiedostolle score.cs:lle tiedoksi lisätä yksi piste.

Muita script-tiedostoja ovat Score.cs, Lives.cs ja Wall.cs. Score.cs ja Lives.cs script-tiedostojen tarkoitus on ainoastaan pitää huoli siitä, että näytöllä näkyvässä pistetaulussa näkyy oikea määrä pisteitä sekä elämiä. Wall.cs on liitetty jokaiseen seinään, jotka rajaavat pelialuetta. Tämä script-tiedosto asettaa seinien värin punaiseksi ja tarkastaa törmäykset pelaajan kanssa. Jos pelaaja osuu seinään, vähennetään pelaajalta yksi elämä ja kutsutaan Player.cs scriptin killPlayer-funktiota joka asettaa pelaajan uuteen sijaintiin kuoleman jälkeen.

6 ONGELMIA JA NIIDEN RATKAISUJA

Opinnäytetyötä aloittaessa ongelmia syntyi luonnollisesti Unityn käyttöliittymää opetellessa. Havaitsin hyväksi oppimiskeinoksi sellaisten YouTube-videoiden tarkastalun, joissa pelisuunnittelija ohjelmoi Unityllä jonkinlaisen yksinkertaisen pelin. Sain myös idean pelilleni tällaisesta videosarjasta, jossa BurgZergArcade-niminen käyttäjä ohjelmoi Unitylle Breakout-kopion (11). Samasta sarjasta opin, kuinka peliobjektin saa liikkumaan kentällä käyttäjän syötteiden perusteella ja kuinka syvyysakselia kannattaa hallita kaksiulotteisessa pelissä. Myös kirja nimeltä Unity Game Development Essentials oli apunani projektin alkuvaiheessa. Kirjasta sai hyvän kuvan Unityn mahdollisuuksista, mutta kirja keskittyi lähes kokonaan 3d-ohjelmointiin, eikä siitä projektin alkuvaiheetta lukuun ottamatta ollut paljoa hyötyä. Tämän takia jouduin turvautumaan enemmän internetistä saatavaan tietoon ohjelmointiongelmissa.

6.1 Törmäyksentunnistus

Törmäyksentunnistusta ohjelmoitaessa ongelmia alkaa muodostua siinä vaiheessa, kun kentällä on useita peliobjekteja yhtä aikaa. Tässä projektissa ongelma ilmeni ensimmäisen kerran, kun powerup-objektista oli saatava läpinäkyväksi ja läpikuljettavaksi. Unity tarjoaa kolme erilaista keinoa hallita törmäyksentunnistusta. Törmäystunnistuksen voi ottaa kokonaan pois päältä asettamalla objekti väliaikaisesti isTrigger-parametrillä triggeriksi, eli objektiksi, jonka tarkoitus on ainoastaan laukaista jokin toiminto. Tässä muodossa objektiin ei voi osua mikään muu objekti. Jos ohjelmoijan tarkoituksena on saada törmäyksentunnistus toimimaan vain tiettyjen objektien kesken, on käytettävä Unityn Physics.IgnoreLayerCollision- tai Physics.IgnoreCollision-funktioita, jotka estävät törmäykset kokonaisten kerroksien tai yksittäisten objektien kesken. Tässä projektissa päädyin käyttämään isTrigger-attribuuttia ja poistamaan powerup-objektin renderöinnin kokonaan, jolloin powerup-objekti häviää näkyvistä kokonaan, eikä siihen voi osua mikään.

6.2 iOS-yhteensopivuus

iOS:lle ohjelmoinnissa ajalla mitattuna isoimmat ongelmat ovat lisenssiasioiden kanssa, mutta myös iOS:n erilaisuus verrattuna muihin käyttöjärjestelmiin aiheuttaa omat ongelmansa. Tässä projektissa erityisen ohjelmointiongelman aiheutti iOS:n tapa ottaa huomioon käyttäjän kosketukset näytöllä. iOS-laitteet eivät laske yksittäisiä kosketuksia näytölle, vaan kosketukset otetaan huomioon vaiheittain. Kun kosketukset otetaan huomioon vaiheittain, pystytään helposti erottamaan kosketustyypit toisistaan. Esimerkiksi sormella vetäminen näytöllä, monet kosketukset ja yksinkertainen kosketus ovat erilaisia vaihetyyppejä. Tässä projektissa mielenkiinto kohdistuu yksittäisiin kosketuksiin ja niiden sijaintiin näytöllä. Yksittäisen kosketuksen koordinaattien selville saamiseksi joudutaan tekemään silmukka, jossa otetaan huomioon yhden vaiheen jokainen kosketus. Nämä kosketukset merkitään omalla indeksillä ja tämän indeksin avulla Input.GetTouch(indeksi).position-attribuutin arvo pystytään tallentamaan 3-ulotteiseen vektoriin. Tämän vektorin avulla luoti saadaan liikkumaan haluttuun suuntaan.

Ongelma tuotti aluksi vaikeuksia, koska iOS:n erilaisuus kosketusten laskemisessa ei ollut itsestäänselvyys. Android-laitteiden kosketustenhallinta Unityllä on helpompaa, koska Android-laitteet pystyvät tulkitsemaan hiirtä varten tarkoitetut Unityn funktiot kosketusten kanssa. Peliin oli alun perin ohjelmoitu kaikki kosketusta tarvitsevat toiminnot hiiren toimintoihin tarkoitettujen funktioiden avulla ja ongelma ilmeni vasta testausvaiheessa. iOS:n yleisyyden vuoksi ongelma oli kuitenkin tullut vastaan monilla muilla ja ratkaisuja oli tarjolla monia erilaisia Unityn kysymyspalstalla (12). Näiden ratkaisujen soveltaminen tähän peliin onnistui.

6.3 Tekoäly ja ampuminen

Tekoälyn ohjelmoiminen oli itselleni tuntematon asia ennen tämän projektin tekemistä. Tämän vuoksi yritin etsiä internetistä ohjelmoijia, jotka olivat dokumentoineet hieman samantyylisen pelin tekoälyn ohjelmoinnin ja löysinkin yhden dokumentin, jossa opetettiin 2d-räiskintäpelin ohjelmointi. Peli erosi tämän opinnäytetyön pelistä paljon pelimekaniikaltaan, mutta tekoälyn toiminta sekä ampumisen ohjelmointi olivat asioita, jotka tulisivat toimimaan myös omassa projektissani pienillä muutoksilla. Omassa projektissani jokaisen peliobjektin toimintaa ohjaa oma script-tiedostonsa, mutta David Lancasterin luomassa peliohjelmointiesimerkissä, Evac-Cityssä, kaikkien liikkuvien peliobjektien toiminta oli yhdessä script-tiedostossa. Tämä aiheutti hieman eroavaisuuksia tekoälyn ohjelmoinnissa. Esimerkissä tarkastettiin jokaisen ruudunpäivityksen aikana ohjattavan peliobjektin nimi, jonka perusteella liikutettiin joko pelaajan hahmoa pelaajan syötteillä, tai tekoälyä ohjelmallisesti. Omassa projektissani ei tarvita tarkastuksia objektien nimien suhteen, koska script-tiedostot ovat jo scene-elementissä liitetty omiin peliobjekteihinsa. (13)

Pelaajan hahmon suorittama ampuminen oli toteutettu David Lancasterin pelissä tavalla, joka sopi hyvin myös omaan projektiini. Evac-Cityssä ampuminen tapahtui kuitenkin hiiren painalluksella ja jouduin muuttamaan tämän toimimaan iOS:n kosketuksella. Tämä muutti huomattavasti tapaa, jolla koordinaatit saadaan pelaajan syötteestä, mutta tuloksena on kaksi vektoria, kuten Evac-Cityssäkin. Vektorit kuvaavat koordinaattia, jonka suuntaan ammuksen tulisi liikkua ja pelaajan sijaintia. Evac-Cityssä kamera seuraa aina pelaajaa, joten pelaajan sijainti on aina ruudun keskellä, mutta tämän muuttaminen muuttuvaksi vektoriksi, jonka arvot riippuvat pelaajan sijainnista ruudulla ei ollut vaikeaa. Luodin monistamiseksi pelaajan syötteen jälkeen tarvitaan

Instantiate-funktiota ja pointteria, joka viittaa pelaajan player.cs-script-tiedostosta luodin bullet.cs-tiedostoon ylimääräisen script-tiedoston, VariableScript.cs:n kautta. (13)

7 YHTEENVETO

Yhteenvetona projekti onnistui mielestäni hyvin, vaikka lähemmäs julkaisukelpoista tuotetta olisi pitänyt päästä. Varsinkin erilaisia vihollisia ja powerupeja olisi hyvä olla pelissä jo nyt enemmän. Pelin pääasia, eli pelimekaniikka on kuitenkin saatu ohjelmoitua toimimaan tarkoituksenmukaisella tavalla. Peli on myös ohjelmoitu alusta lähtien sillä tavalla, että uusien ominaisuuksien lisääminen olisi mahdollisimman helppoa. Myös Unity mahdollistaa helpon grafiikan ja äänen lisäämisen myöhemmässä vaiheessa.

Unity osoittautui helposti opittavaksi pelintekoalustaksi, vaikka kokemusta sen kanssa ohjelmoinnista ei ollut kuin yhden pienen projektin verran. Tähän vaikuttivat varsinkin Unityn laaja manuaali internetissä, jossa on selitetty jokaisen luokan toiminta hyvin tarkkaan. Myös Unityn keskustelupalstoille lähetettyjä ongelmia ja ratkaisuja oli hyvä soveltaa tähän projektiin. Unityn suosio mahdollisti myös monien valmistuneiden pelien raporttien lukemisen. Näistä oli helppo oppia uusia ohjelmointitapoja ja niiden soveltamista erilaisiin peleihin.

LÄHTEET

1. Lecky-Thompson, G W. 2007. Video Game Design Revealed, s. 210

2. Flurry Analytics blog. 2012. The Truth About Cats and Dogs: Smartphone vs Tablet Usage Differences. http://blog.flurry.com/bid/90987/The-Truth-About-Cats-and-Dogs-Smartphone-vs-Tablet-Usage-Differences [viitattu 19.3.2013]

3. Apple. 2012. iOS Development Library, UIAcceleration Class Reference. http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UIAcceleration_Class/Reference/UIAcceleration.html [viitattu 20.4.2013]

4. Unity Manual. 2013. Mobile Input. http://docs.unity3d.com/Documentation/Manual/Input.html [viitattu 20.3.2013]

5. Unity. 2013. License Comparisons. http://unity3d.com/unity/licenses [viitattu 20.4.2013]

6. Unity Fast Facts. 2013. Milestones. http://unity3d.com/company/public.relations/ [viitattu 20.3.2013]

7. Rochard blog. 2011. Rochard's Producer on the Unity engine. http://www.rochardthegame.com/tag/unity/ [viitattu 17.4.2013]

8. Appstore Arcade. 2013. Bad Biggies. http://www.appstorearcade.com/wp-content/uploads/2012/10/bad_piggies_2.png [viitattu 17.4.2013]

9. Develop. 2013. Unity Focus: Bad Piggies. http://www.develop-online.net/features/1798/Unity-Focus-Bad-Piggies [viitattu 17.4.2013]

10. Goldstone, W. 2009. Unity Game Development Essentials.

11. BurgZergArcade, Youtube.com. 2011. Community Challenge #1 – Breakout – Chapter 3 – Ball Script Part 1. http://www.youtube.com/watch?v=QBrYZMCfT0o [viitattu 19.3.2013]

12. Unity Answers. 2011. Simple Touch of a 3d Object in iOS. http://answers.unity3d.com/questions/47753/simple-touch-of-a-3d-object-in-ios.html [viitattu 10.2.2013]

13. Lancaster, David. A starters guide to making a game like Evac-City. http://www.rebelplanetcreations.com/downloads/Other/Tutorials/HowToMakeAGameInUnity3D.pdf [viitattu 10.2.2013]