KYMENLAAKSON AMMATTIKORKEAKOULU
Tietotekniikka / Ohjelmistotekniikka

Mika Uurainen

UNITY MOBIILIPELIEN KEHITYKSESSÄ

Opinnäytetyö 2013

TIIVISTELMÄ

KYMENLAAKSON AMMATTIKORKEAKOULU
Tietotekniikka

Uurainen Mika	Unity mobiilipelien kehityksessä
Opinnäytetyö	50 sivua
Työn ohjaaja	Paula Posio, Yliopettaja
Toimeksiantaja	Kymenlaakson ammattikorkeakoulu
Huhtikuu 2013	
Avainsanat	Unity, mobiili, pelisuunnittelu, ohjelmakoodi esimerkit

Opinnäytetyön tarkoituksena oli toteuttaa mobiilipelidemo, jonka toteutuksessa käytettiin apuna Unity-pelimoottoria. Pelin suunnittelu ja ideointi on toteutettu täysin itse. Tarkoituksena oli tehdä nykymarkkinoille suurta suosiota nauttineiden mobiilipelien rinnalle oma toimiva ja yhtenäinen pelidemo.

Opinnäytetyössä käsitellään laajasti Unity-pelimoottoria ja mobiilipeleille yleisesti ominaisia asioita sekä niiden aiheuttamia haasteita ohjelmoijalle, verrattaessa normaaliin Windows-ympäristössä toteutettaviin peleihin. Opinnäytetyössä tulee ilmi Unityn komponenttimainen ohjelmointi. Yhtenä tavoitteena oli myös tehdä Kymenlaakson ammattikorkeakoulun Gamelabin käyttöön opas aloitteleville ohjelmoijille Unityn käyttöön. Opinnäytetyö sisältää sovellettavia ohjelmakoodiesimerkkejä.

Opinnäytetyössä käydään läpi myös mobiilipelin valmistusprosessi: mitä kaikkea tulee tehdä, jotta peli päätyy itse laitteille sekä mitä ohjelmia ja lisenssejä tarvitaan. Käydään lyhyesti läpi käytettävän pelimoottorin ominaisuuksia ja miten sen ominaisuuksia on hyödynnetty eri tilanteissa. Tämän lisäksi tarkastellaan myös pelin ideaa ja mitä tulee ottaa huomioon, kun peliä suunnittelee.

ABSTRACT

KYMENLAAKSON AMMATTIKORKEAKOULU
University of Applied Sciences
Information Technology

Uurainen Mika	Unity in Developing Mobile Games
Bachelor's Thesis	50 pages
Supervisor	Paula Posio, Senior Lecturer
Commissioned by	Kymenlaakso University of Applied Sciences
April 2013	
Keywords	Unity, mobile, design, code examples

The main focus of thesis was to produce a mobile game demo. Unity game engine was used to make some of the game's features. Design and ideas were self-made. The main goal was to make a demo, which would challenge the mainstream games with easy to use controls and likeable characters.

The study covered the Unity game engine as a whole and discussed the characteristics of mobile games as well as the challenges that they pose for a programmer, in comparison to games developed in the Windows-environment. The component-based programming of Unity was demonstrated in the study. In addition to the game demo, one of the goals was to make a beginner programming guide, for students at the Game lab of Kymenlaakso University of Applied Sciences. The study included code examples that could be put into practice.

The process of making a mobile game is reviewed, also the needed devices, licenses and programs are listed in the thesis. Some of the game features were also analysed and how is it useful to use those in some of the cases. The game's idea and general game design concepts were also mentioned.

SISÄLLYS

Termit ja Lyhenteet

Occlusion Culling	Kamera ei piirrä mitään peliobjekteja, mitä ei sillä hetkellä ole nähtävissä kameran näköpiirissä.
Profiler	Unityn oma optimointi työkalu. Se näyttää pelin sen hetkisen muistin käytön ja prosessorin käytön. Lisäksi voi yksityiskohtaisesti katsoa, mitkä ohjelmakoodit tai peliobjektit käyttävät eniten prosessoria tai muistia.
Muodostin	Käytetään olio-ohjelmoinnissa. Muodostinta kutsutaan aina olion syntymisen yhteydessä. Muodostimessa alustetaan tai asetetaan olion jäsenmuuttujat.
Luokka	Kokonaisuus, joka sisältää luokalle ominaisia muuttujia ja metodeja. Luokista luodaan olio-ohjelmoinnissa olioita.
Muuttuja	Tietotyyppi, joka voi olla numero, totuusarvo, teksti tai luokka.
Metodi	Luokan aliohjelma, jota pystytään kutsumaan luokan sisällä tai muista aliohjelmista.
Nollaviittaus	Kun ohjelma yrittää osoittaa muistiin, joka ei ole ohjelman käytössä, syntyy nollaviittaus, joka aina käsitellään ohjelmavirheeksi.
Renderöinti	Pelissä kaikki mikä näytetään pelaajalle renderöidään eli piirretään.

1 JOHDANTO

Opinnäytetyössä on tarkoitus kartoittaa nykyaikaisten pelien tietyt niksit, myyntivaltit, suunnitteluprosessit ja erilaiset tekniset rajoitteet. Apuna käytettiin Unity -pelimoottoria ja pelissä hyödynnetään sen tarjoamia ominaisuuksia hyvinkin paljon.

Kymenlaakson ammattikorkeakoulun ohjelmistotekniikan koulutusohjelma muutettiin peliohjelmoinniksi vuonna 2012. Tämän seurauksena ammattikorkeakoulussa aloitettiin erilaiset yhteistyöt, muun muassa Kotkan ja Kouvolan välillä. Tavoitteena oli muodostaa opiskelijaryhmiä, jonka seurauksena syntyisi mahdollisia uusia yrityksiä.

Bitter rolling-mobiilipelidemo on toteutettu yhteisyössä Einari Lavasteen kanssa. Hän opiskelee Kouvolassa viestintää ja on suuntautunut dgitaaliseen mediaan. Yhteistyöhön kuului peli-idean, markkinoinnin- ja prosessin suunnittelua. Einari Lavaste toteutti pelin graafiset ominaisuudet ja minulle jäi ohjelmistopuolen toteutus, suunnittelu ja testaus. Prosessi on kestänyt keväästä 2012 asti ja syksyllä 2013 päätin tehdä kyseisestä ideasta opinnäytetyön.

Pelin tekemiseen liittyy erilaisia rooleja. Tarvitaan ohjelmoijia, graafikoita, mallintajia, kentän tekijöitä, johtajia, markkinointia ja laaduntestausta. Ohjelmoijat yleisesti ohjelmoivat hahmot liikkumaan, ohjelmoivat pelin logiikan ja mahdollisesti tekevät pelimoottorin. Pelimoottori hoitaa piirtämisen ja sinne usein ohjelmoijaan fysiikat ja animaatiot. Graafikot tekevät grafiikka ja mallintajat mallintavat erilaisia hahmoja tai objekteja peliin. Yleensä mallinnuksen ja grafiikat mallille tekee sama henkilö. Kentän tekijät suunnittelevat ja toteuttavat pelin eri kenttiä. He lisäävät mallintajien tekemiä objekteja ja hahmoja kenttiin. Peliprojektiin tarvitaan myös johtaja, kuka johtaa ja opastaa tekijäryhmää. Tämä on erittäin haasteellinen, että peli valmistuu aikataulussa ja on oikein budjetoitu. Markkinointi on yksi tärkeimmistä, jotta peli saadaan asiakkaiden tietoon. Huono markkinointi yleensä johtaa huonoihin myyntilukuihin ja peli ei menesty odotetulla tavalla. Laaduntestauksella pyritään siihen, että peli toimii moitteettomasti. Pelin kaatumiset ja ohjelmavirheet pyritään havaitsemaan ja korjaamaan.

2 MOBIILIPELIT

Mobiilipelit kappaleessa käydään läpi mobiiliepelien historiaa ja niiden käsitteitä. Lisäksi kerrotaan yleisimmät pelityypit ja nostetaan esille muutama kuuluisa mobiilipeli.

2.1 Historia ja pelityypit

Ensimmäinen mobiilipeli oli Tetris. Hagenukin valmistamalle MT-2000 laitteelle tämä laite julkaistiin vuonna 1994. Suomalaisillekin tuttu mobiilipeli on Nokian tekemä Snake. Snake julkaistiin vuonna 1997 Nokian 6110 ja Nokian 2110 puhelimille. Kyseisiä laitteita myytiin arviolta noin 350 miljoona kappaletta. (1)(2)

Mobiilipelejä on tehty laidasta laitaan. Löytyy muun muassa autopelejä, pulmanratkonta pelejä, ristikoita, korttipelejä ja strategiapelejä. Nykyään suuressa suosiossa ovat mahdollisimman yksinkertaiset ja fysiikkavetoiset pelit, kuten voi todeta maailmalla menestyneestä suomalaisesta mobiilipelistä Angry Birds. Angry Birds on sinänsä ilmiö, sillä sen suuruiseen menestykseen ei ole moni pystynyt. Angry Birdsin suosiota on vaikea selittää, mutta varmaankin pelin pelattavuus ja hauskat hahmot ovat yksi syy suuren suosion takana. Angry Bridsin ollessa enempi ilmaislatausten kärjessä on taas Tetris maksullisten listan kärjessä. Tetriksen tapauksessa listalla ei ole otettu huomioon laitteita, joiden mukana Tetris on julkaistu, vaan ainoastaan digitaaliset ostot mobiililaitteille nykyisistä jakelukanavista.

3 UNITY-PELIMOOTTORI

Unity pelimoottorina on tehnyt pitkän matkan. Aluksi sitä ei käytetty juuri missään ja nykyään se on yksi käytetyimmistä pelimoottoreista mobiilipeleissä. Seuraavassa käydään läpi Unityn historiaa, eri versioiden eroja, Unity Editorin ominaisuuksia ja myös pelimoottorin ominaisuuksia.

3.1 Historia

Unity kehitys aloitettiin vuonna 2001. Se julkaistiin Applen kaupassa vuonna 2005 ja siitä oli saatavilla ainoastaan MAC OSX -versio. Windows-versio tuli vasta myöhemmin 2.0 päivityksen ohessa. (3)

Ajan myötä Unity on kasvanut erittäin suosituksi pelimoottoriksi. Se kattaa nykyään Windows-, MAC OSX- ja Linux-ympäristöt ja useat erityyppiset konsolilaitteet. Unityllä pystyy tekemään samanaikaisesti usealle alustalle pelejä, kunhan omistaa tarvittavat lisenssit. Unity oli vuonna 2012 käytetyin pelimoottori mobiilipelien tekemisessä. (4)

Seuraavassa listassa on esitetty Unity-pelimoottorin suurimmat päivitykset ja niiden mukana tulleet isoimmat ominaisuudet.

Versio 1.0

- Lisäsivät dokumentaatiota
- Unity Web Player sai sisäisen testausympäristön

Versio 2.0

- Maastoeditori
- Pystyy käyttämään videoita peleissä
- DirectX 9.0 renderointi
- Mahdollisuus tehdä verkkomoninpelejä Unityn omilla komponenteilla
- Reaaliaikaiset dynaamiset varjot
- Mahdollisuus luoda graafisia käyttöliittymiä
- Unity-versionhallinta
- Unity-profiler
- Animaatiokomponentteihin parannuksia
- PhysX-päivitys uudempaan versioon

Versio 3.0

- Mahdollisuus kääntää Android-alustoille
- Ohjelmoijille uusi työkalu Mono Develop
- Erilaisia suorituskyvyn parrannuksia iOS:lle käännettäviin peleihin
- Valmis reitinhaku
- PhysX-päivitys uudempaan versioon

Versio 4.0

- Uusi animaatiojärjestelmä
- DirectX 11 -renderöinti
- Mahdollisuus julkaista Linuxille
- Unity-editorille uusi käyttöliittymä
- Unityn uudet työkalut graafisten käyttöliittymien tekemistä varten

Lista 1: Listassa on Unityn suurimmat päivitykset. Listassa ei ole jokaista päivitystä. Tarkemman listan löytää Unityn sivuilta. (3)

Unitystä on saatavilla ilmainen ja maksullinen versio. Lisäksi jokaiselle alustalle on ostettava oma lisenssi, jos alustalle halutaan julkaista pelejä. Unity Pro-lisenssi maksaa 1500 dollaria. Mikäli haluaa kehittää iOS- tai Android-alustoille, tulee kehittäjän maksaa 400 dollaria lisenssistä. Näistä iOS- ja Android-lisensseistä on tarjolla myös Pro-lisenssi, ja jokainen Pro-lisenssi maksaa erikseen 1500 dollaria. Pro-lisenssin ominaisuuksia on listattu lista 2:ssa. Nämä ominaisuudet pätevät jokaiseen Unity Pro - , Android Pro - ja iOS Pro -lisensseihin. Joitakin alustakohtaisia eroja saattaa löytyä, mutta ne kannattaa tarkistaa Unityn kanssa asioidessa. Näiden kaikkien lisenssien ohella Unityllä on tarjota ryhmälisenssi, joka sisältää Unityn oman versionhallinnan. Tämä tuo lisäkustannuksia 500 dollarin edestä. (5)

Yleiset

- Reitinhaku
- Pystyy käyttämään videoita3D-objekteissa
- Jos yrityksen liikevaihto on enemmän kuin100 0004 $ vuodessa, yrityksen tulee ostaa Pro-lisenssi
- Animaatiosta löytyy muutama eroavaisuus
- Pelien koon ja muun optimointi Profilerin avulla

Graafiset

- 3D-grafiikan käyttö
- Reaaliaikaiset varjot peliobjekteille
- Valoanturit
- Occlusion culling

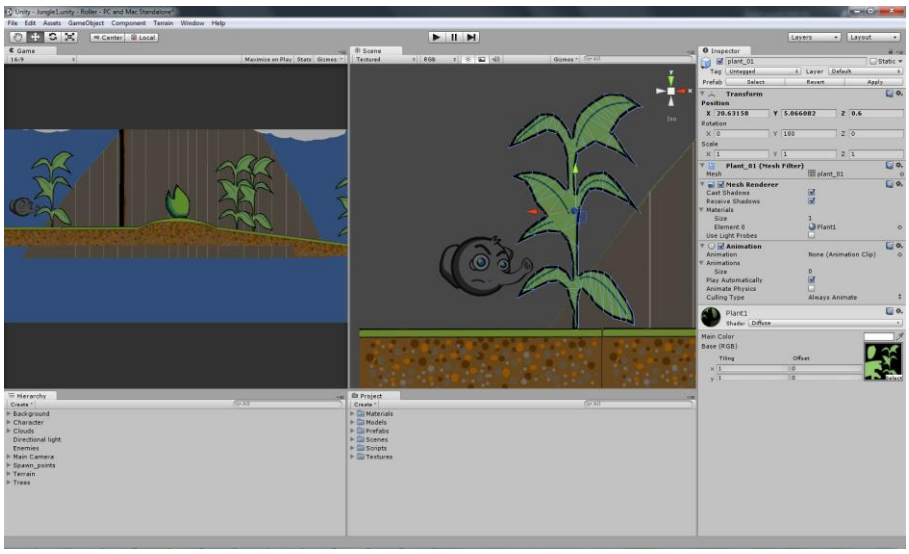
Editori

- Profiler
- Tummempi käyttöliittymä
- Mahdollisuus käyttää erillistä tietokonetta, joka hoitaa kääntämisen automaattisesti.

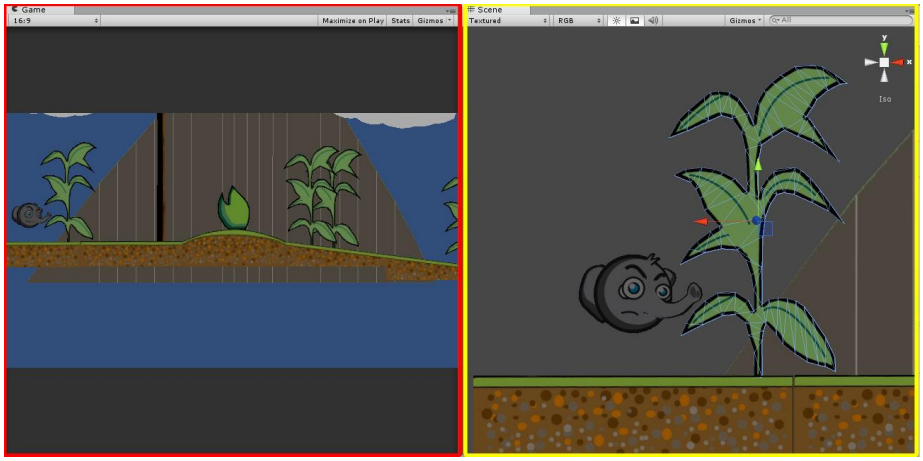
Lista 2: Unity Pro-lisenssi tarjoaa muun muassa seuraavia ominaisuuksia, joita ilmainen lisenssi ei tarjoa.

3.2 Unity-editori

Unity-editori on sekä ohjelmoijan, kentän tekijän että graafikon työkalu. Editorin käyttö on sinällään helppoa ja sen oppii nopeasti. Seuraavassa käydään läpi editorin eri näkymät ja ominaisuuksia. Kuvassa 1 nähdään alkutilanne kun Unity-editoriin on avattuna Bitter Rolling.



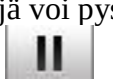
Kuva 1: Unity editori, johon on avattuna peliprojekti Bitter rolling.



Kuva 2: Unity-editorin Scene- ja Game-näkymä, näitä näkymiä käytetään peliä tehdessä.

Kuvassa 2 voi nähdä editorin kaksi näkymää: oikealla Scene ja vasen Game. Game-näkymä kuvastaa erittäin hyvin sitä, mitä itse pelissä tapahtuu. Scene on taas tekijälle tarkoitettu näkymä, jossa hän voi valita eri peliobjekteja, liikutella niitä, lisäillä uusia peliobjekteja, käyttää kameraa miten haluaa, liikkua mailmassa miten haluaa sekä muokata peliobjektien asetuksia. Yleensä työskennellään molemmat näkymät rinnakkain, jotta nähdään välittömästi, miltä jokin peliobjekti näyttää itse pelissä.

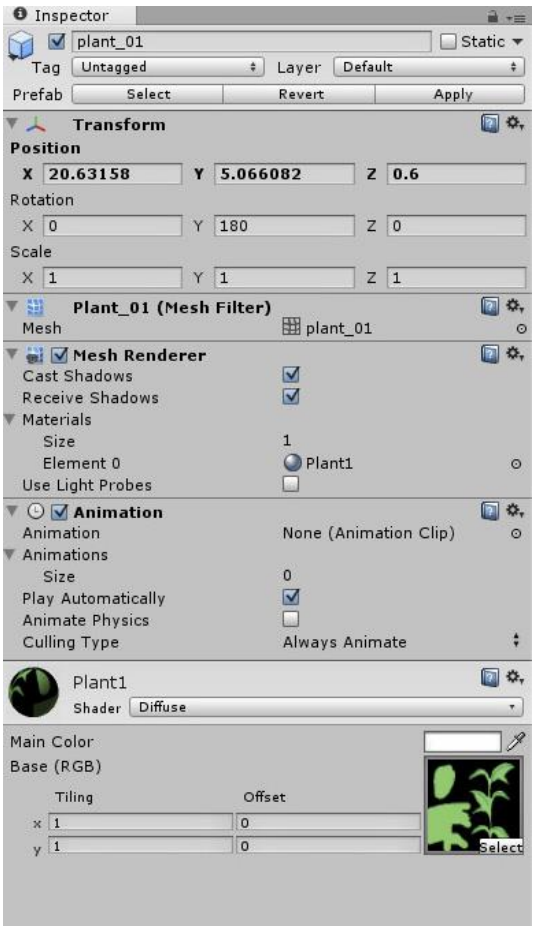
Valikoista pystyy muun muassa luomaan uuden projektin, lisäämään uusia peliobjekteja peliin, kytkemään objekteille eri komponenttejä ja muokkaamaan itse peliä eri työkalujen avulla. Seuraavassa listassa näkyy tärkeimmät työkalut.

- Käsityökalu, jonka avulla käyttäjä voi liikkua ympäri pelimaailmaa Scene-ruudussa. 
- Siirtotyökalu, jonka avulla käyttäjä pysty muuttamaan peliobjektin paikkaa itse pelimaailmassa. 
- Peliobjektin kääntötyökalu, jonka avulla käyttäjä voi esimerkiksi kääntää peliobjektia tietyn asteen verran. 
- Peliobjektin skaalaustyökalu, jonka avulla käyttäjä voi skaalata objektia isommaksi tai pienemmäksi 
- Käynnistä-nappi, jonka avulla peliä pystyy pelaamaan tai testaamaan editorissa 
- Pysäytysnappi, jonka avulla käyttäjä voi pysäyttää pelin editorissa, kunhan ensiksi on painettu Käynnistä-nappia 
- Kelausnappi, jonka avulla käyttäjä voi mennä kuva kerrallaan eteenpäin pelissä, kunhan peli on pysäytys tilassa. 

Peliobjektit on mahdollista laittaa eri tasoille (layers). Esimerkiksi kaikki maaobjektit kannattaa laittaa omalle maatasolle ja viholliset omalleen. Tasoja voidaan käyttää hyödyksi muun muassa fyysisissä törmäyksissä eri peliobjektien välillä tai tehtäessä sädeammuntaa. Näistä ominaisuuksista kerrotaan lisää Monobehaviour-osiossa. Näitä tasoja voidaan editorin valikon kautta käydä muuttamassa. Sinne voi luoda uusia tasoja tai muuttaa asetuksia, mitkä peliobjektit törmäävät keskenään. Jos halutaan päästä eroon vihollisten keskinäisistä törmäyksistä, näin voidaan määrittää asetuksissa. Ulkoasussa (layout) voi vaihtaa editorin näköä ja määrittää, kuinka monta asiaa editorista on yhtä aikaa näkyvillä. Kuvassa 3 näkyy Layers-valinta, joka määrittelee sen hetkiselle valitulle peliobjektille tason ja Layout-, joka vaikuttaa Editorin ulkoasuun.

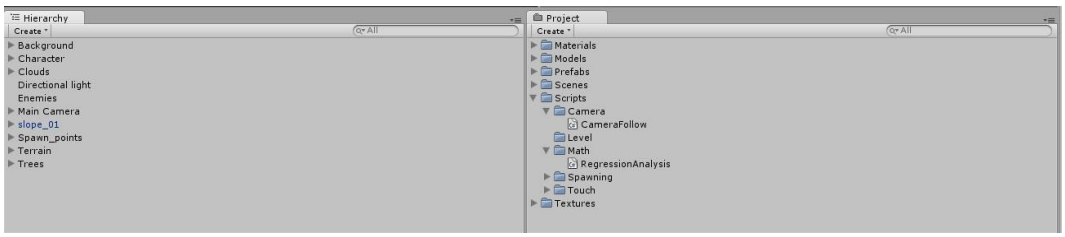
Kuva 3: Unity editorin tasot fysiikkaominaisuuksia varten ja editorin ulkoasun muuttamiseksi. Kuva on suurennos kuvan 1 oikeasta ylänurkasta.

Kuvassa 4 nähdään tarkastelunäkymä (inspector). Tämä näkymä näyttää sillä hetkellä valitun peliobjektin ominaisuudet, kuten nimen, paikan maailmassa ja kaikki sen omistamat komponentit



Kuva 4: Kuvassa näkyy tarkastelunäkymä, jostain valitusta peliobjektista. Tarkastelunäkymä näyttää peliobjektin sen hetkiset ominaisuudet.

Kuvassa 5 on hierarkianäkymä. Se näyttää kaikki projektissa olevat tiedostot, sekä sen hetkisen Scenen peliobjektit. Valitsemalla jonkin objektin ja tuplaklikkaamalla sitä, Unity Editori hyppää kätevästi Scene-näkymässä valittuun peliobjektiin. Näin on helppo etsiä jokin objekti ja katsoa, missä tai mitä se tekee itse pelissä. Tätä käytetään erilaisissa ongelma tilanteissa, jos jokin peliobjekti käyttäytyy omituisesti. Aktiiviset peliobjektit näkyvät mustalla fontilla hierarkiassa ja deaktivoidut näkyvät harmaalla fontilla. Kuvassa 5 nähdään BitterRolling -demon projekti ja yhden Scenen hierarkianäkymä.

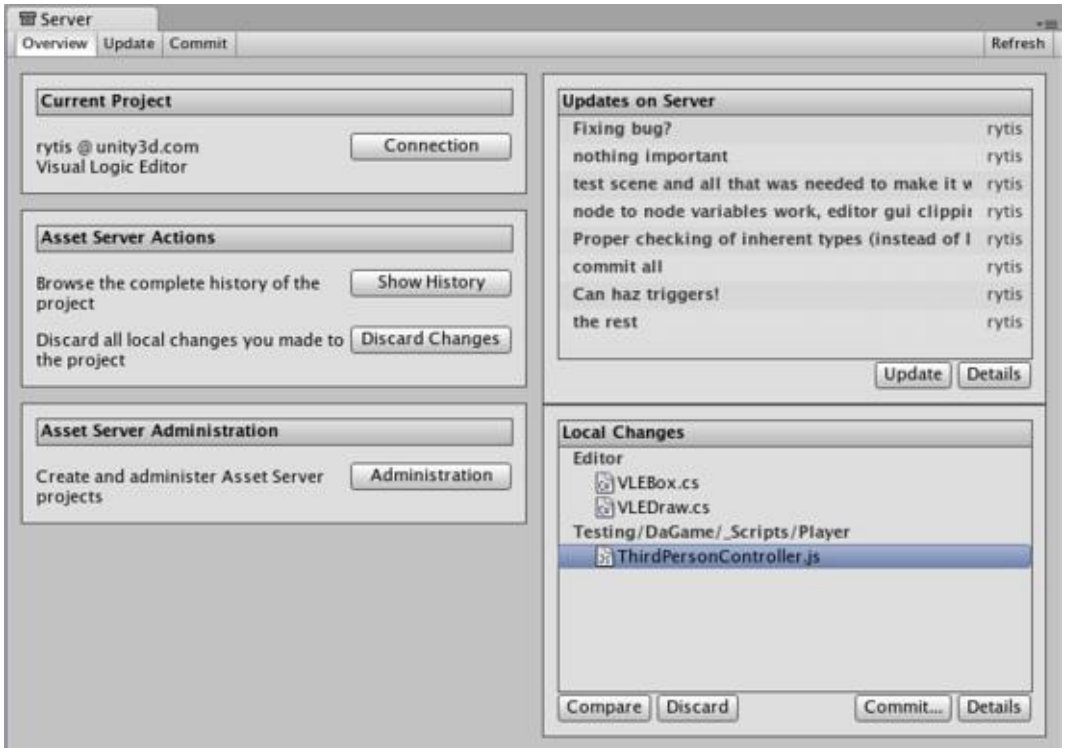


Kuva 5: Hierarkia näkymästä, vasemmalla on nähtävissä Scenen näkymä. Kuvassa oikealla on nähtävissä Bitter Rolling-projektin sen hetkiset tiedostot.

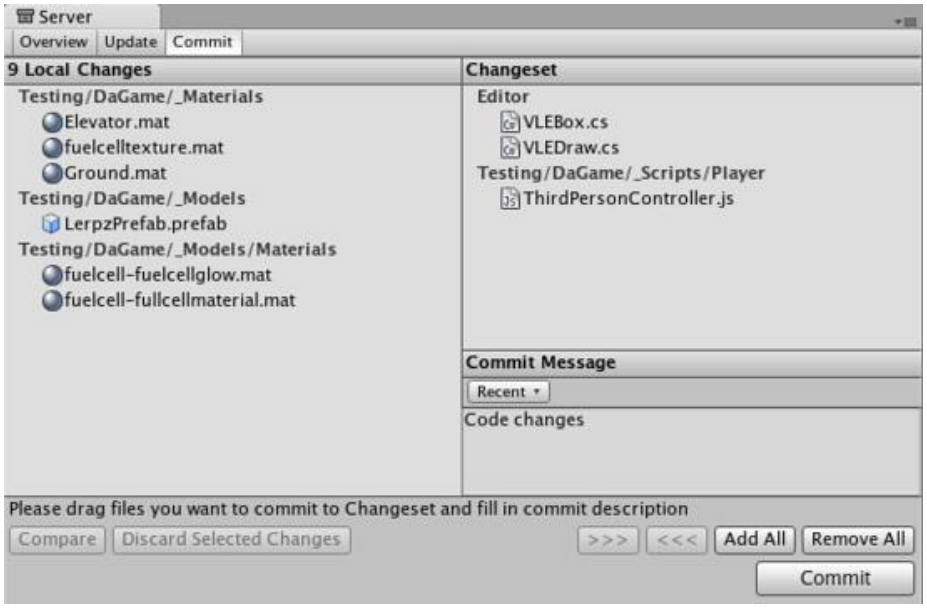
3.3 Unity versionhallinta

Versionhallinta on erittäin käytetty työkalu pelinkehityksessä ja Unity on rakentanut oman version siitä. Seuraavassa käydään läpi Unityn versionhallinta yleisesti ja tarkastellaan myös sen toiminallisuutta. Kappaleissa pyritään käymään oleelliset ja tärkeät asiat, mitä tulee muistaa kun tallettaa työnsä versionhallintaan.

Toimintaperiaate on se, että projekti viedään jollekin tietylle palvelimelle, jonka jälkeen kaikki työntekijät ottavat yhteyden tähän palvelimeen. Kun yhteys on luotu, jokainen työntekijä työskentelee omien tehtäviensä parissa. Esimerkiksi työntekijä A saa tehdyksi jonkin tietyn asian. Hän siirtää työnsä palvelimelle, jolloin projektia päivitetään palvelimella uusilla lisäyksillä. Seuraavan kerran, kun käyttäjä B kirjautuu tai siirtää töitään, palvelin ilmoittaa päivityksestä, jonka työntekijä A on tehnyt ja kehottaa kaikkia käyttäjiä ottamaan päivitykset itselleen. Ennen kuin työntekijä B voi siirtää omat asiansa palvelimelle, hänen tulee ottaa päivitys omaan projektiinsa. Kuvassa 6 nähdään palvelimen näkymä, kun kirjaudutaan siirtämään omia töitä palvelimelle. Päivityksessä on mahdollista ohittaa tai kumota jonkun toisen tekemät muutokset, mikäli ne aiheuttavat ristiriitaa oman päivityksen kansa. (6)



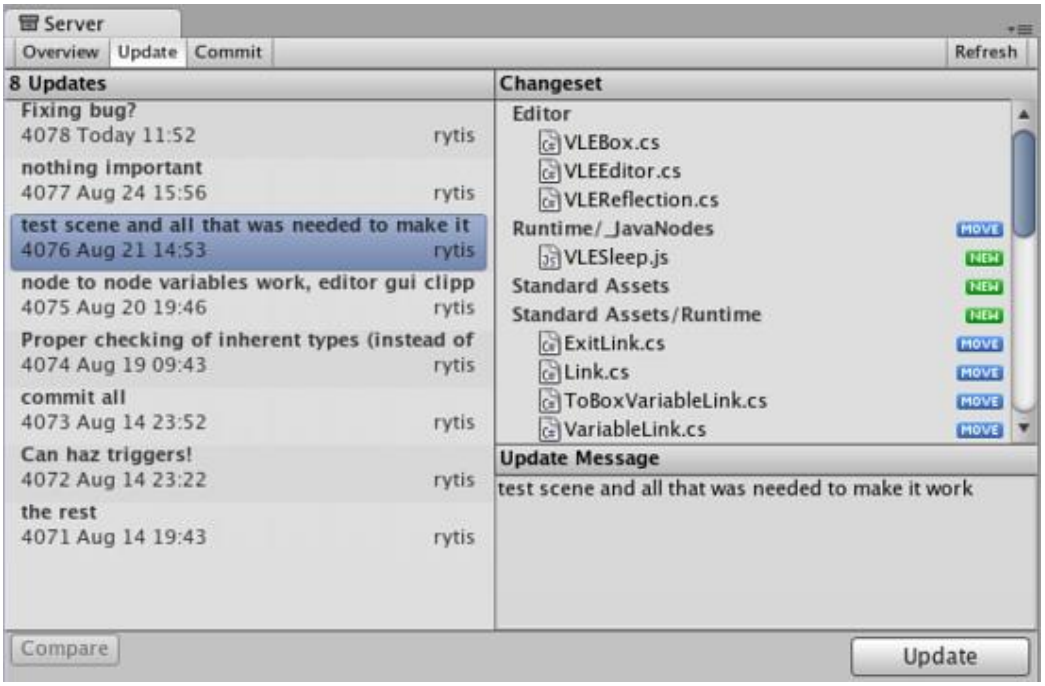
Kuva 6: Unity versionhallinnan näkymä, kun on luotu yhteys ja kirjauduttu version-hallintaan. Palvelin ilmoittaa saatavilla olevista päivityksistä ja mitä muutoksia itse on tehnyt.(6)



Kuva 7: Omien töiden siirtäminen versionhallintaan turvaan täytyy valita siirrettävät tiedostot ja kirjoittaa viesti mitä on tehty. (6)

Siirrettäessä muutoksia talteen palvelimelle määritellään siirrettävät tiedostot ja jokin lyhyt viesti siitä mitä on tehty. Kuvassa 7 nähdään palvelimen näkymä siirrosta. Jos työntekijä muistaa laittaa viestin tehdystä työstä, se helpottaa huomattavasti työskentelyä projektin parissa. Jos peliin tulee jokin paha ohjelmavirhe tai peli lakkaa toimimasta, on mahdollista selvittää tämän syy tutkimalla siirtohistoriaa. Tämän edellytyk-

senä on tietenkin se, että jokainen työntekijä on laittanut oleellisen viestin päivityksen yhteydessä. Viestin avulla päivitysten rajaaminen helpottuu ongelmatilanteissa. Kuvassa 7 näkyvä viesti ei kerro mitään oleellista tehdystä työstä, eli tämä on huono esimerkki viestistä. Viestissä tulee kertoa tarkasti, mitä ja minne on tehty.



Kuva 8: Versionhallinta ilmoittaa saatavilla olevista päivityksistä käyttäjälle ja suosittelee ottamaan päivitykset. (6)

Unity tarkastaa joka kerta kirjauduttaessa versionhallintaan, onko oma versio projektista ajan tasalla, sekä ilmoittaa päivitysten nimen, päivämäärän, milloin se on siirretty palvelimelle ja viestin, joka on siirron yhteydessä kirjoitettu. Kuvassa 8 nähdään palvelimen näkymä päivitysosiosta. Jokainen päivitys tulee ladata, jos itse haluaa siirtää jotakin palvelimelle. Kuitenkin päivityksen ottamisen yhteydessä käyttäjä voi olla piittaamatta jostain päivityksen osista tai hän voi korvata saman tien palvelimella olevan tiedoston omallansa, jos päivityksen versio on käyttäjän mielestä väärä. Paras keino on varmistaa päivityksen laittajalta mitä muutoksia hän on tehnyt, ennen kuin käy korvailemaan ja mahdollisesti tuhoamaan toisen työn.

Versionhallinnassa on tullut ilmi muutamia ongelmia. Ohjelmoijien ohjelmakoodit pystytään useimmiten yhdistämään hyvinkin näppärästi, jos on käynyt sellainen tilanne että kaksi ohjelmoijaa on muokannut samaa ohjelmakoodia. Jos Unityssä tehtyä Sceneä työstää samaan aikaan kaksi henkilöä, siitä seuraa konflikti. Unityssä ei ole mahdollisuutta yhdistää scenejä. Esimerkkinä tilanne, jossa henkilö A lisää Sceneen

talon ja henkilö B puun. Henkilö B siirtää muutoksen palvelimelle, jonka jälkeen henkilö A yrittää myös siirtää omaa muutostaan scenessä palvelimelle. Palvelin kuitenkin ilmoittaa, että hänen versionsa Scenestä on vanha ja hänen tulisi ennen siirtoa päivittää kyseinen Scene. Tällöin kuitenkin henkilö A:n muutokset kumotaan täysin sillä palvelin lataa henkilö B:n siirtämän Scenen, koska se on palvelimella olevista versioista ajan tasalla oleva versio. Tämän seurauksena henkilö A:n tekemä scene poistetaan täysin ja hänen tekemänsä muutokset poistetaan täysin. Tämä ongelma on ainoastaan sceneissä ja korjautuu pienellä kommunikaatiolla. Sceneä pystyy työstämään ainoastaan yksi henkilö kerrallaan, joten scene kannattaa jakaa pienempiin osiin, jos haluaa että useampi henkilö työstää sitä samanaikaisesti. Lopuksi nämä pienemmät palaset yhdistetään keskenään.

3.4 Monodevelop

Monedevelop on ohjelmistoympäristö, jota Unity tarjoaa ohjelmoijille. Monodevelopia käytetään C# ja muissa Dot Net -ohjelmointikielissä. Yleisesti sitä käytetään tehtäessä ohjelmistoja Windows-, Linux- ja Mac OSX -käyttöjärjestelmille. Unityn tapauksessa Monodeveloppia on muokattu, jotta sillä pystyy testaamaan itse Unity-editorissa tehtyä peliä. Monodevelop mahdollistaa ohjelmakoodin tekemisen ja ohjelmakoodin testauksen. Monodevelop kääntää ohjelmoijan ohjelmakoodin ja näyttää sen toiminnassa Game-näkymässä. Game-näkymässä peliä testataan erilaisten ohjelmavirheiden löytämiseksi.

Tavallisesti ohjelmoija pystyy luomaan luokkia, tekemään muuttujia sekä luomaan metodeja. Poikkeuksellista Unityssä on se, ettei muodostimen (constructor) käyttö ole mahdollista silloin, kun luokka perii Unityn oman Monobehaviour-luokan. Monobehaviour mahdollistaa kaikkien Unityn valmiiden komponenttien käytön. Luokka pystyy normaalisti toteuttamaan monta rajapintaa ja perimään yhden luokan. Kuitenkin pohjaluokan tulee periä Unity:n Monobehaviour, jos haluaa jossakin luokassa käyttää Unityn ominaisuuksia, ja tämä estää muodostimen käytön jokaisesta luokasta. Muuttujat ovat näkyvyydeltään public, private, protected tai static. Public näkyy luokan ulkopuolella ja sen arvoa pystytään muuttamaan Unity-editorista. Private näkyy vain luokan sisällä ja protected näkyy muille luokille, jotka perivät kyseisen luokan. Static on yhteinen kaikille luokkaa käyttäville peliobjekteille. Avoimessa muuttujassa on yksi pieni asia, jonka ohjelmoija tulisi ottaa huomioon.

```
public class EsimerkkiPublicMuuttujasta : MonoBehaviour {

public float voima = 100;

}
```

Koodiesimerkki 1: Public muuttujan esittely ja arvon asetus luokassa, joka perii Monbehaviour-luokan.

Kun koodiesimerkissä 1 luokan muuttuja alustetaan jollain arvolla, kuten ohjelmoinnissa yleensä on tapana ja kyseinen luokka liitetään johonkin peliobjektiin, tulevat kaikki public-muuttujat näkyville Unity-editoriin. Tällöin niitä on helppo muuttaa itse editorissa, ettei käyttäjän tarvitse joka kerta avata Monodeveloperia ja muuttaa arvoa tästä luokan koodissa. Eräs ongelma on mahdollinen, mikäli muuttuja on alustettu koodiesimerkki 1 tapaisesti sadaksi ja sille annetaan Unity-editorissa arvo 150. Tämän jälkeen pelatessa peliä muuttujan arvoksi tulee sadan sijaan 150. Tämän vuoksi olisi suositeltavaa alustaa kaikki public-muuttujat aina nollaksi ja laittaa niille tarvittava arvo itse editorissa, jotta vältytään sekaannuksilta ohjelmoijan yrittäessä etsiä virheitä ohjelmakoodista. Arvoa toki voi muuttaa kesken ohjelman-ajon ja myös ohjelmakoodissa. Public muuttujat voidaan piilottaa editoristakin käyttäen ”[HideInInspector]” määritystä muuttujan yläpuolella. Public muuttujien käyttö ja niiden piilottaminen on kuitenkin erikoista, on parempi käyttää private muuttujia.

Privaten tapauksessa muuttujaa ei voi asettaa editorissa, joten se on hyvä alustaa. Tätä arvoa ei pääse Unity Editorin kautta vahingossakaan muuttamaan, eikä siihen pääse käsiksi luokan ulkopuolelta, ellei tee sille erillisiä asettajia (set) ja hakijoita (get). Koodiesimerkissä 2 on yksinkertainen asettajasta ja hakijasta. Sama pätee protected muuttujaan. Protected muuttujiin pääsee käsiksi luokissa, jotka perivät kyseisen luokan. On olemassa keino, jolla muuttujat tulevat näkyville Unity-editorissa: tämä on niin sanottu testaustila (debug mode). Muutoin private- ja protected-muuttujat ovat piilossa ja ainoastaan ohjelmoijan käytettävissä. Testaustila asetetaan Unity-editorista käsin, oikeassa yläkulmassa on lukon kuva ja sen vieressä olevasta napista saa testaustilan päälle.

```
public class EsimerkkiAsettajastaHakijasta : MonoBehaviour {

//määritellään yksityinen muuttuja ja alustetaan se 0:ksi
```

```
int laskuri = 0;

void Start(){
  //kasvatetaan laskuria käynnistyksen yhteydessä yhdellä
  laskuri++;
}
void Update(){
  //jos käyttäjä painaa hiiren näppäimen alas, lisätään laskuria 10:llä
  //ja tulostetaan arvo
  if (Input.GetMouseButtonDown(0)){
    AsetaArvo += 10;
    Debug.Log(AsetaArvo);
  }
}
//asettaja ja hakija, mitä voidaan kutsua luokan ulkopuoleltakin
public int AsetaArvo{
  set{
    laskuri = value;
  }
  get{
    return laskuri;
  }
 }
}
```

Koodiesimerkki 2:Esimerkki yksinkertaisesta asettajasta, palauttajasta ja yksityisestä muuttujasta.

Staattiset muuttujat eivät koskaan tule näkyville Unity-editorissa, mutta ohjelmoija pystyy viittaamaan niihin kaikkialta, kunhan ne ovat public-tyyppiä. Muuttujan ollessa staattinen, on muuttuja kaikille luokan olioille yhteinen ja sama. Koodiesimerkissä 3 tehdään luokka, jolla on public static-muuttuja.

```
public class EsimerkkiStaattinenMuuttuja : MonoBehaviour {

public static float voima = 100;

}
```

Koodiesimerkki 3:Esimerkki staattisesta muuttujasta, sekä sen arvon asettamisesta jossain luokassa.

Kun luokka perii Unity MonoBehaviourin, luokassa voidaan käyttää kolmea Unityn metodia. Yleensä kun luodaan uusi luokka, Unity luo automaattisesti näistä kaksi Start- ja Awake-metodin. Koodiesimerkki 4:ssä tulee ilmi juuri luodun luokan alkutilanne, kun ohjelmoija on itse lisännyt Awake-metodin.

```
public class EsimerkkiUusiLuokka : MonoBehaviour {
```

```
void Awake(){
}

void Start(){
}

void Update(){
}
}
```

Koodiesimerkki 5: Esimerkissä näkyy Unity Monobehaviourin valmiit metodit, mitä ohjelmoija pystyy käyttämään. Sekä ohjelmoija itse lisäämä Awake-metodi.

Unity luo aina Start- ja Update-metodit, Awake-metodi tulee aina lisätä itse. Awake-metodia kutsutaan aina ensimmäiseksi kun scene ladataan. Unityn dokumentaatiossa sanotaan, että Awake vastaa olio-ohjelmoinnin muodostinta, jota kutsutaan olion luonnin yhteydessä. Näiden kahden välillä on se ero että muodostin, on samanniminen kuin itse luokka. Unityn Awake-metodi ei ole samanniminen kuin luokka, eikä Awakea pysty kuormittamaan, kuten olio-ohjelmoinnissa on. Eikä Awakelle pysty antamaan parametrien avulla arvoja. Kumpaakin Awakea ja muodostostinta kutsutaan aina kun peliobjekti ladataan Scenessä tai olio-ohjelmoinnissa luodaan uusi olio, tämä on yhteinen piirre molemmilla. Awakea ei tule verrata muodostimeen vaan tulisi verrata metodiin, jossa voidaan asettaa eri arvoja tai hakea peliobjektin muut tarvittavat komponentit myöhempää käyttöä varten. Start-metodi tehdään heti Awake-metodin jälkeen ja ne suoritetaan aina tässä järjestyksessä, kun peliobjekti aktivoidaan. Update-metodia päivitetään jokaisella näytönpäivityksellä, jos peliobjekti on aktiivinen.

```
public class EsimerkkiAwakenJaStartKäytöstä : MonoBehaviour {
//peliobjekti mikä luodaan kun tämä peliobjekti aktivoidaan, voidaan asettaa
editorista käsin
public GameObject pallo;
//pelaajan tallettamista varten
GameObject pelaaja;
//omaa jäykkäkappaletta varten
Rigidbody jaykkakappale;
//liikkumisnopeus
float nopeus = 0;

void Awake(){
  //haetaan pelaaja, otetaan oma jäykkäkappale, asetetaan oma nopeus ja lopuksi
  deaktivoidaan oma peliobjekti
  pelaaja = GameObject.Find("Pelaaja");
  jaykkakappale = GetComponent<Rigidbody>();
  nopeus = 105.0f;
  gameObject.active = false;
}

void Start(){
  //kun peliobjekti tulee aktiivisesti jostain kumman syystä
  //luodaan pallo siihen kohtaan missä peliobjekti on sillä hetkellä
```

```
  pelimaailmassa
  Instantiate(pallo,transform.position,Quaternion.identity);
  Debug.Log("Tulin aktiiviseksi");
  //lopuksi tuhotaan tämä peliobjekti, ettei se jää kummittelemaan
  Destroy(gameObject);
}

void Update(){
  //tätä ei tarvitse ollenkaan, koska objekti on toiminnassa pienen ajan. Eikä
  tarvitse tarkastella joka näytönpäivityksellä jotakin logiikkaa. Updaten
  voisi aivan hyvin poistaa
}
}
```

Koodiesimerkki 5: Esimerkissä on erään Awake- ja Start-metodien käyttötapaus. Esimerkissä tulee esille, mitä kannattaa Awakessa ja Startissa asettaa taikka hakea.

Koodiesimerkin 5 kaltaisesti Awake-metodissa yleensä haetaan tarvittavat komponentit, etsitään toisia peliobjekteja ja mahdollisesti asetetaan muuttujille arvoja. Jos kyseisen peliobjektin ei tarvitse olla heti pelin alussa aktiivisena, voidaan Awaken lopuksi deaktivoida kyseinen peliobjekti. Toinen vaihtoehto on se, ettei Awaken lopuksi tehdä deaktivointia, vaan jonkin toisen scriptin Start metodissa haetaan kaikki deaktivoitavat peliobjektit Scenestä ja deaktivoidaan kaikki peliobjektit. Jokaisen Scenen peliobjektin pitäisi ehtiä hakemaan talteen tarvitsemansa peliobjektit. Jotta vältyttäisiin tilanteelta, jossa toinen tekee omat Awake-metodin asiansa toista nopeammin. Erillinen deaktivointi on parempi ratkaisu. Jos peliobjekti on deaktivoitu, sitä ei pysty etsimään ohjelmakoodin avulla. Koodiesimerkki 6:ssä on esimerkki tästä deaktivoinnista.

```
public class EsimerkkiDeaktivoinnista : MonoBehaviour {

//taulukko kaikista deaktivoitavista peliobjekteista
GameObject[] etsittavat;
void Awake()
  {  //etsitään kaikki Scenessä olevat deaktivoitavat peliobjektit,  jokaisella
     on yhteinen tagi ”Deaktivoi”
     etsittavat = GameObject.FindGameObjectsWithTag("Deaktivoi");
  }

void Start()
  {
    //käydään taulukko läpi ja deaktivoidaan jokainen taulussa oleva peliobjek
    ti. Oletetaan että nämä on haettu talteen jo Awakessa, jotenka niitä ei
    tarvita enään
    foreach (GameObject ob in etsittavat)
    {
       ob.active = false;
     }
   }
```

Koodiesimerkki 6: Esimerkissä etsitään silmukan avulla tietyn tagin omaavia peliobjekteja.

Updatessa tehdään yleensä pelilogiikkaan liittyviä toimenpiteitä. Yleensä siellä toteutetaan laskentaa, tekoälyä, tarkistetaan käyttäjän antamia komentoja tai luodaan tietyn ajan kuluttua peliobjekteja sceneen.

```
public class EsimerkkiUpdatesta : MonoBehaviour {
void Update(){
  //jos käyttäjä painaa Q näppäintä, tulostetaan teksti
  if (Input.GetKey(KeyCode.Q)){
  Debug.Log("Painoit näppäintä Q!");
  }
  //jos käyttäjä painaa hiiren vasenta näppäintä, tulostetaan teksti
  if (Input.GetMouseButtonDown(0)){
  Debug.Log("Painoit hiiren vasenta näppäintä!");
  }
  //jos käyttäjä painaa hiiren oikeata näppäintä, tulostetaan teksti ja paikka
  koordinaatit
  else if (Input.GetMouseButtonDown(1)){
  Debug.Log("Painoit hiiren oikeata näppäintä!");
  Debug.Log("Hiiri on kohdassa " + Input.mousePosition + " ruutua!");
  }
 }
}
```

Koodiesimerkki 7: Esimerkki Update-metodista ja kuinka otetaan käyttäjän komentoja vastaan.

```
public class EsimerkkiAjastimenKäytöstä : MonoBehaviour {
public GameObject pallo;
float ajastin,aika;

void Start() {
  //asetetaan ajastimelle aika kun tapahtuu jotakin
  ajastin = 5.0f;
 }

void Update(){
  //päivitetään joka ruudunpäivityksellä aikaa, käyttäen Unityn Time
  aika += Time.deltaTime;
  //kun aika ylittää ajastimen
  if (aika > ajastin)
  {
  //luodaan pallo mikä on asetettu Editorissa, se ei saa olla jätetty tyhjäksi
  if (pallo != null)
  Instantiate(pallo, transform.position, Quaternion.identity);
  //nollataan aika
  aika = 0;
  //muutetaan omaa paikkaa, etteivät pallot tule päällekkäin
  transform.position += new Vector3(10.0f, 0.0f, 0.0f);
  //kierros alkaa alusta ja jatkaa loputtoomin pallojen luomista
  }
 }
}
```

Koodiesimerkki 8: Esimerkissä käsitellään ajastimen käyttöä Update-metodissa ja uusien peliobjektien luomista sceneen.

Jos ohjelman koodissa yritetään muuttaa tai käyttää jotain peliobjektia, joka on jostain syystä tyhjä tai on tuhottu, peli kaatuu ympäristöstä riippumatta. Tämä pätee myös mobiililaitteissa. Koodiesimerkissä 9 tulee ilmi mihin nollaviittaus johtaa.

```
public class EsimerkkiNollaViittauksesta : MonoBehaviour {
//määrittellään kaksi pelaajaa
public GameObject pelaaja1;
public GameObject pelaaja2;

float aika = 0;

void Start(){
  //etsitään pelaaja yksi automaattisesti
  pelaaja1 = GameObject.Find("Pelaaja1");
}

void Update(){

  aika += Time.deltaTime;
  //pelaaja yksi ei aiheuta ongelmia, koska se on haettu automaattisesti
  Debug.Log(pelaaja1.transform.position);
  //pelaaja kaksi taas aiheuttaa kaatumisen, jos se on unohdettu laittaa
  Editorissa
  Debug.Log(pelaaja2.transform.position);

  //tuhotaan pelaaja 1, 5 sekunnin jälkeen, jonka jälkeen molemmat
  aiheuttavat ongelmia
  if (aika > 5.0f)
    Destroy(pelaaja1);
  }
 }


}
```

Koodiesimerkki 9: Esimerkissä esitellään nollaviittaus ja mitä siitä seuraa jos pyritään käyttää tuhottua tai tyhjää peliobjektia.

```
public class EsimerkkiNollaViittauksenEstämisestä : MonoBehaviour {
//määrittellään kaksi pelaajaa
public GameObject pelaaja1;
public GameObject pelaaja2;

float aika = 0;

void Start(){
  //etsitään pelaaja yksi automaattisesti
  pelaaja1 = GameObject.Find("Pelaaja1");
}

void Update(){

  aika += Time.deltaTime;
```

```
  //estetään kaatuminen perinteisellä, jos tarkistuksella
  if(pelaaja1 != null)
    Debug.Log(pelaaja1.transform.position);
  //sama pelaaja kahdelle
  if(pelaaja2 != null)
    Debug.Log(pelaaja2.transform.position);

  //tuhotaan pelaaja yksi 5 sekunnin jälkeen, jonka jälkeen ei pelaaja1
  paikka tulostu
  //sekä varmistetaan vielä, että sitä ei ole jo valmiiksi tuhottu
  if (aika > 5.0f && pelaaja1 != null)
     Destroy(pelaaja1);
  //helpommalla pääsee kun laittaa kaikki yhden tarkistuksen sisään
  if (pelaaja1 != null){
    Debug.Log(pelaaja1.transform.position);

   if (aika > 5.0f)
     Destroy(pelaaja1);
  }
 }
}
```

Koodiesimerkki 10: Tässä esimerkissä estetään nollaviittaus, joka johtaisi pelin kaatumiseen.

Koodiesimerkissä 10 on estetty nollaviittaus. Tämän estäminen on aina ohjelmoijan vastuulla. Ohjelmoija voi tehdä myös tarkistuksen, jos hän vaikka unohtaa asettaa jonkin tärkeän peliobjektin Unity-editorissa. Koodiesimerkissä 11 käydään läpi tarkistuksen tekeminen ja viestin tulostus. Viestissä on kätevää antaa peliobjektin nimi. Kuitenkin jos samannimisiä objekteja on satoja, on hyvä muuttaa peliobjektin väri tilapäisesti punaiseksi, kuten koodiesimerkissä 12 on tehty.

```
//pyydetään antamaan maali editorissa
public GameObject maali;

void Start(){
//mutta jos se unohdetaan antaa, tulostetaan viesti.
  if (maali == null)
    Debug.Log("Peliobjektissa " + gameObject.name + " on unohdettu laittaa
    maali!");
 }
}
```

Koodiesimerkki 11: Esimerkki tarkistuksesta, jos ohjelmoija on unohtanut asettaa jonkin peliobjektin Unity-editorin kautta.

```
//pyydetään antamaan maali editorissa
public GameObject maali;

void Start(){
  //mutta jos se unohdetaan antaa, tulostetaan viesti ja vaihdetaan väri.
```

```
  if (maali == null){
    Debug.Log("Peliobjektissa " + gameObject.name + " on unohdettu laittaa
    maali!");
    gameObject.renderer.material.color = Color.red;
  }
 }
}
```

Koodiesimerkki 12: Esimerkissä muutetaan peliobjektin materiaalin väri punaiseksi, jotta se on helpompi tunnistaa muiden joukosta.

3.5 Reitinhaku

Reitinhakua käytetään yleensä vihollisilla ja pelaajan hahmossa, jotta ne osaavat navigoida 3D-maailmassa. Yksinkertaisin reitinhaku olisi siirtyä pisteestä A pisteeseen B välittämättä esteistä, muista vihollisista, rakennuksista tai korkeuseroista. Esteet tuovat reitinhaun toteuttamiseen haasteita ja ongelmia.

Unity Pro tarjoaa valmiin reitinhaun. Unityssä käytetään valmiiksi luotua navigointiverkkoa. Tämä verkko luodaan kaikkiin paikkoihin, jotka on määritelty navigoitavaksi. Jos jokin talo tai puu on määritelty ison alueen keskellä ei-navigoitavaksi, Unity luo näppärästi navigoitavan verkon sen ympärille.

Jotta pelaaja tai viholliset pystyvät navigoimaan luodussa verkossa, tulee jokaisella olla oma ”agentti”. Agentin tehtävänä on laskea onko saavuttu määränpäähän, väistää muita agentteja, määrittää nopeus, jolla liikutaan sekä kiihtyvyys. Koodiesimerkissä 13 haetaan peliobjektin oma navigointiagentti, jonka avulla pystytään liikuttamaan peliobjektia pelimaailmassa. Agentille annetaan piste ja agentti suunnistaa kohti pistettä käyttämällä navigointiverkkoa. Jos pistettä ei löydy, agentti liikkuu siihen pisteeseen verkkoa, mikä on lähinnä kyseistä pistettä, mutta ei ikinä saavu määränpäähänsä.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

//luokassa on yksi void Liiku, jonka tarkoituksena on liikkua johonkin pistee-
seen. Luokka perii Unityn Monobehaviourin, jotta voidaan kutsua Unityn omia
komponentteja
//agentti hoitaa reitinlaskennan ja väistelyn
public class EsimerkkiReitinhausta : MonoBehaviour{

//NavMeshAgent määrittely
NavMeshAgent agentti;

//tehdään kun peli käynnistyy
```

```
void Start(){
  //haetaan objektin komponentti NavMeshAgent, jotta navigointi on mahdollista
  agentti = GetComponent<NavMeshAgent>();
 }
//Tapahtu joka ruudunpäivityksellä
void Update(){
  //Täällä ei tehdä vielä mitään
 }

//yksinkertainen liikkumis metodi, annetaan parametrinä kohde joka on tyyppiä
Vector3
//pystytään kutsumaan mistä tahansa koska on Public
public void Liiku(Vector3 kohde){
  agentti.destination = kohde;
  agentti.Move(kohde);
 }
}
```

Koodiesimerkki 13: Esimerkki Unityn tarjoaman reitinhaun käytöstä.

Unity-editorissa voi määrittää agentille liikkumisnopeuden, säteen joka määrittää kuinka iso agentti on, sekä agentin kiihtyvyyden. Ohjelmoija voi vaikuttaa näihin ominaisuuksiin myös itse ohjelmakoodissa. Mutta niiden ollessa public typpisiä, kannattaa ne aina asettaa Editorin kautta.

3.6 Animaatio

Unity tukee ma-, mb-, fbx-, ja blend-tiedostoformaatteja. Kyseisten tiedostotyyppien tuominen Unityyn on erittäin helppoa. Ensimmäisenä on luotava kansio projektiin, jonka jälkeen vain siirretään kyseiset tiedostot tähän kansioon. Joka kerta kun projekti avataan Unityssa, Unity automaattisesti tarkistaa, onko projektitiedostoihin tullut muutoksia. Sekä tuo kyseiset tiedostot kovalevyltä itse projektiin, jolloin ne ovat käytettävissä. Joissain ääritapauksissa voi tuomisessa tulla joitakin vaikeuksia. Toisinaan tuodun objektin rotaatio menee pieleen, jolloin tulee laittaa kyseinen objekti tyhjän peliobjektin alle, tätä peliobjektia kutsutaan yleensä emoksi. Tällä emo-objektilla on kaikki rotaatiot nollana, jolloin lapsiobjekti käyttää emon rotaatioita.

Animaation tarkoituksena on saada jonkin objektin tietyt osat liikkumaan, esimerkiksi hahmon jalat sen kävellessä. Hahmon nostaessa laatikkoa maasta käsivarsilleen, liikutellaan käsiä ja jalkoja simuloiden oikeaa ihmismäistä liikettä. Animaation tarkoituksena on tuoda elävyyttä ja sulavuutta pelin. Ilman tiettyjä animaatioita peli näyttäisi aika kankealta.

3.7 Peliobjekti

Unity engine käyttää peliobjekteja (game objects). Jokainen asia, joka peliin luodaan, on peliobjekti. Jokainen peliobjekti on mahdollista määrittää erilaiseksi, kun sitä verrataan toiseen peliobjektiin. Eroa voi olla esimerkiksi värissä ja muodossa. Peliobjektin ei välttämättä tarvitse näyttää miltään esimerkiksi voidaan luoda peliin joku peliobjekti, joka laskee kuinka monta palloa pelaaja on kerännyt. Samoin laskijaobjektin paikalla pelimaailmassa ei ole merkitystä, sillä laskennan voi suorittaa missä tahansa, kunhan tällä peliobjektilla on yhteys pelaajaan. Koodiesimerkissä 10 on haettu pelaaja ja muodostettu yhteys sitä kautta. Kun luodaan uusi peliobjekti, tällä tyhjällä peliobjektilla ei ole yhtään komponenttia, mutta komponentti tarvitsee aina peliobjektin. Voidaankin todeta, että peliobjekti säiliö, joka sisältää erilaisia komponentteja. (7)

3.8 Valmisolio

Valmisolio (prefab) voidaan luoda jostain peliobjektista. Valmisolio sisältää kaikki ominaisuudet ja komponentit mitä sillä peliobjektilla oli sillä hetkellä, kun se luotiin. Peliobjekteista pystyy luomaan erilaisia valmiita olioita, jotka voi tallentaa itse projektiin. Komponentteja voi myöhemmin lisätä miten haluaa ja voi myös ottaa valmiin olion jolle lisää tiettyjä komponentteja, joita ei halua toiselle oliolle laitettavan. Malliolion päivitys on helppoa, jos haluaa jonkin uuden komponentin kaikille. Se tapahtuu päivittämällä kyseinen malliolio Unity-editorin avulla. Ilman mallipohjaisia olioita työskentely olisi erittäin vaikeaa sillä jos jokainen objekti tehtäisiin alusta asti, kasvaisi projektin tekemiseen vaadittava työmäärä erittäin suureksi. Tämän lisäksi, jos haluaisi esimerkiksi muuttaa jokaisen vihollispeliobjektin väriä, tulisi kaikki viholliset käydä yksitellen läpi ja vaihtaa jokaisen väri manuaalisesti. Tätä eivät kuitenkaan kaikki pelinkehittäjät välttämättä tiedä ja he uhraavat paljon aikaa värien tai tekstuurien vaihtamiseen.

3.9 Komponentit

Seuraavassa käydään läpi yleisimpiä komponentteja. Komponentti tarkoittaa jotakin osaa, mikä on liitetty johonkin peliobjektiin. Sitä voi verrata siihen kun ihminen laittaa lippalakkinsa päähän, tällöin hän käyttää kyseistä lippalakkia. Sama pätee objekteihin ja komponentteihin. Jos objektiin liittää jonkin komponentin, objekti pystyy käyttää hyväkseen komponentin ominaisuuksia. Esimerkiksi peliobjektiin voidaan liittää Na-

vigointi-komponentti ja peliobjekti pystyy liikkumaan komponentin avulla. Ilman komponenttia peliobjekti ei liiku, eikä sitä myöskään liitetä niihin peliobjekteihin, joiden ei haluta liikkuvan.

3.9.1 Fysiikkakomponentit

Fysiikkakomponentit pitävät sisällään törmäykset, voimat, kitkat, kiihtyvyydet ja erityyppisiä räjähdyksiä. Kuten aiemmin on mainittu, törmäyksiin voidaan vaikuttaa Unity-editorin tasojen avulla. Seuraavassa käydään läpi erilaisia fysiikkakomponentteja läpi, sekä yhtä eritäin hyödyllistä ohjelmointiapuvälinettä sädeammuntaa.

Jäykkä kappale (rigidbody) on peliobjektiin liitettävä komponentti. Tämä mahdollistaa objektille fysikaaliset ominaisuudet ja toiminnot kuten muun muassa massan, painovoiman, kitkan tai muita rajoitteita. (8)

```
using UnityEngine;
using System.Collections;

public class EsimerkkiJäykkäkappale : MonoBehaviour {


//voima arvoina, alustetaan arvo 0 ja laitetaan Editorissa jokin arvo
public float voima = 0;
//itse jäykkäkappale
Rigidbody jaykkakappale;

 void Start(){
  //haetaan peliobjektin jäykkäkappale
  jaykkakappale = GetComponent<Rigidbody>();
 }

 void Update {
 }

 void FixedUpdate(){
  //jos painetaan näppäimistöltä Z-näppäintä käsketään liikkumaan
  if (Input.GetKey(KeyCode.Z))
    AnnaVoima();
  }

 void AnnaVoima(){
  //annetaan peliobjektille sen etenemis vektorin muikaisesti voimaa
  //peliobjekti liikkuu suoraviivaisesti eteenpäin.
  jaykkakappale.AddForce(Vector3.forward * voima);
 }
}
```

Koodiesimerkki 14: Esimerkki voiman antamisesta 3D-pelimailmassa, voiman suunta määritellään katsottuna eteenpäin peliobjektista.

Koodiesimerkki 14:ssa annetaan kappaleelle voima, voima saa peliobjektin liikkumaan suoraan etenpäin. Tätä voisi käyttää nuolessa tai jossain muussa etenevässä asiassa. Voiman voisi laskea erikseen esimerkiksi käyttäjän vetämästä matkasta, kuten Angry Birdsissä on tehty ja painovoima hoitaa loput.

```
//käytetään poikkeuksellisesti vasen tai oikeaa vektoria, riippuen mikä on sen
hetkisessä pelimailmassa eteenpäin oleva
void AnnaVoimaOikea()
{
  jaykkakappale.AddForce(Vector3.right * voima);
}
void AnnaVoimaVasen()
{
  jaykkakappale.AddForce(-Vector3.right * voima);
}
```

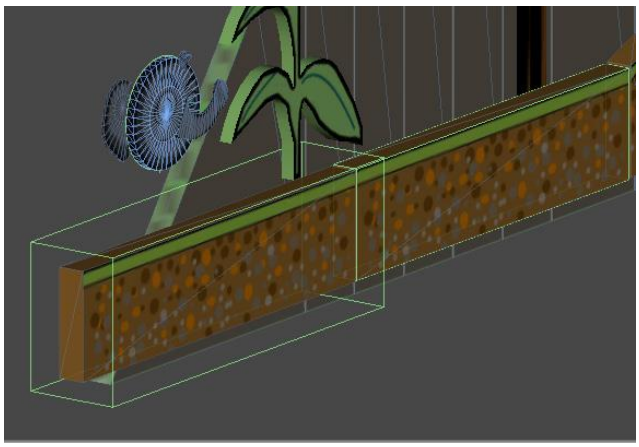
Koodiesimerkki 15: Esimerkissä käytetään 3D-maailman sijaan 2D-maailman oikeaa ja vasenta.

Ohjelmakoodi esimerkissä 15 annetaan voima kaksiulotteisessa maailmassa. Jostain syystä Unityssa määriteltäessä maailma kaksiulotteiseksi, ei ole mahdollista käyttää enää etuvektoria. Sen sijaan tulee käyttää oikeaa tai vasenta, jos halutaan suoraviivaista etenemistä.

Törmäimet (Colliders) ovat peliobjektiin liitettäviä komponentteja. Peliobjektit törmäävät toisiinsa ainoastaan, jos molemmilla on joku törmäin-komponentti. Mikäli objektiin halutaan liittää jokin fysikaalinen toiminto, kuten kimpoaminen törmätessä toiseen objektiin, sekä törmääjällä että törmätyllä tulee olla jokin törmäin. Tämän lisäksi toiseen niistä täytyy olla liitettynä jäykkäkappale-komponentti, jotta voidaan antaa jäykänkappaleen avulla törmäyksestä tuleva voima. Törmäimillä ei sinänsä ole muuta eroa toisiinsa nähden kuin muoto.

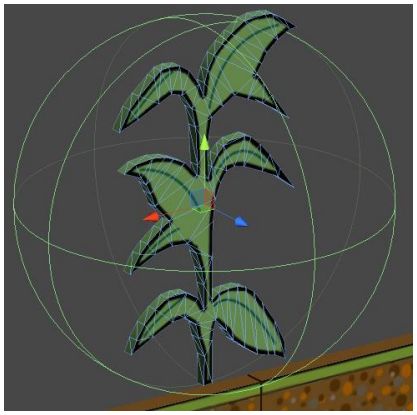
Törmäimillä on myös tiettyjä ominaisuuksia. Jokaisen kokoa pystyy säätämään, joko pituuden tai säteen avulla. Joidenkin suuntaa pystyy muuttamaan: meneekö se x- vai y-akselin suuntaisesti.

Laatikkotörmäin (Box collider) on muodoltaan kuutio. Sitä käytetään yleensä suoraviivaisten muotojen, kuten ovien, laatikoiden, lattioiden ja lavojen kanssa. Kuvassa 9 näkyy laatikkotörmäin liitettynä lattia peliobjektiin. Törmäin estää peliobjektia tippumasta lattiasta läpi.



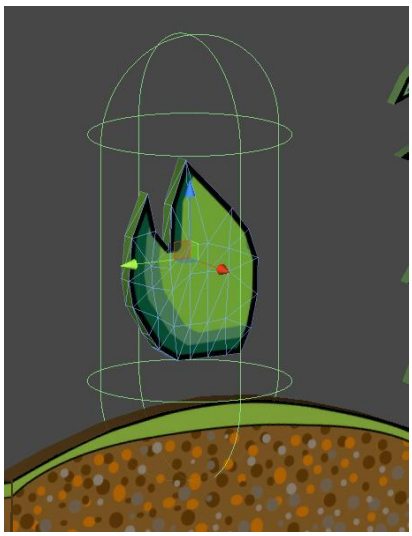
Kuva 9: Kuvassa näkyy laatikkotörmäin käytössä pelimaailman lattia peliobjektissa.

Pallotörmäimen (Sphere collider) koko määritellään säteen avulla. Tätä törmäintä käytetään yleensä pyöreissä muodoissa kuten kanuunan kuulassa sekä erilaiset pallot. Kuvassa 10 on pallotörmäin liitetynä kasvi-peliobjektiin.

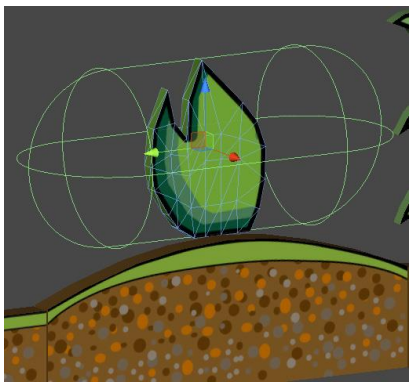


Kuva 10: Pallotörmäin käytössä pelimaailmassa olevassa kasvi-peliobjektissa..

Kapselitörmäimelle (Capsule collider) pystyy määrittämään säteen ja kapselin korkeuden. Kuvassa 11 on kapselitörmäin suunnattuna y-akselin suhteen ja kuvassa 12 kapselitörmäin on suunnattu x-akselin suhteen. Kapselitörmäintä käytetään eläimissä ja ihmisissä. Yleensä eläimet ovat nelijalkaisia, joten kapselin suunta tulee laittaa x-akselin suhteen. Ihmiset puolestaan ovat korkeita, jolloin kapselin suunta tulee laittaa y-akselin suhteen.



Kuva 11: Kapselitörmäin kun sen suunta on y-akselin suhteen.



Kuva 12: Kapselitörmäin kun sen suunta on x-akselin suhteen.

Pyörätörmäin (Wheel collider) eroaa muista törmäimistä siinä, että se sisältää valmiiksi tiettyjä komponentteja, kuten jäykänkappaleen, oman kiihtyvyyden, kitkan ja nopeuden säädön. Pyörä törmäintä käytetään yleensä maata pitkin etenevissä ajoneuvoissa.

Sädeammunta on ohjelmoijalle erittäin tärkeä työkalu. Ideana on ampua jokaista pistettä kohti säde ja tämän tarkoituksena on löytää ensimmäinen piste, joka pysäyttää säteen. Unityssä pysäyttäväksi pisteeksi voi määrittää minkä tahansa törmäin, ilman törmäintä säteet eivät pysähdy. Sädeammunnan avulla pystytään poimimaan pelimailmasta erilaisia objekteja, joille voi tehdä mitä haluaa. Esimerkiksi käyttöliittymässä on erilaisia nappeja kuten, valikko ja lopetus. Kummallakin napilla on törmäin. Käyttäjän painaessa näyttöä, ammutaan säde näytöstä katsottuna kohtisuorassa pelimaailmaan. Jos säde osuu valikko- tai lopetus-nappiin. Tehdään niille ominaiset toiminnot. Valikko avaa valikkonäkymän ja lopetuksesta suljetaan peli, jos käyttäjän painallus ei osu kumpaankaan, ei tehdä mitään. Sädeammunnan avulla voidaan etsiä

jokin piste ja käskeä pelaajaa liikkuman pisteeseen. Piste voi olla esimerkiksi jokin kohta lattiaobjektista. Sädeammunnan avulla on helppo poimia vihollisia. Jos säde osuu törmäimeen, voidaan helposti selvittää peliobjekti, johon törmäin kuuluu. Ohjelma koodi esimerkissä 12 on toteutus, tasojen käyttämisestä ja kaikkien sellaisten peliobjektien poimimisesta, joilta löytyy törmäin. (9)

Sädeammuntaa voidaan pitää käyttäjän ja pelimaailman välisenä vuorovaikutuksena. Peliobjektin ollessa hyvin pieni törmäintä suurentamalla voidaan objektiin osuminen tehdä huomattavasti helpommaksi. Unityn tasojen avulla voidaan eritellä eri peliobjektit ja esimerkiksi sellainen tilanne, kun halutaan säteen osuvan ainoastaan lattiaan. Vihollisille voidaan tehdä oma taso ja kun säde osuu vihollistason törmäimeen. Käsketään pelaajaa esimerkiksi hyökkäämään sitä vasten. Tämä pystytään määrittämään ohjelmakoodissa hyvinkin helposti. (10)

```
using UnityEngine;
using System.Collections;

public class EsimerkkiSädeammunta : MonoBehaviour {

//Tasonpeitto tämän maskin alla olevat peliobjektit otetaan huomioon
public LayerMask maa;

void Start()
{
}

void Update()
{
  //kun painetaan hiirtä ammutaan säde
  if (Input.GetMouseButtonDown(0))
  {
    //Säde mikä ammutaan, tulee kamerasta suoraan pelimaailmaan missä hiiri on
    sillä hetkellä
    Ray sade = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit osuma; //mihkä osutaan

  //jos osutaan maahan tulostetaan teksti "Osuit maahan"
   //mitään muuta ei huomioida
  if (Physics.Raycast(sade, out osuma, Mathf.Infinity, maa))
  {
     Debug.Log("Osuit maahan");
  }
   //tässä on käsin määritelty säde joka 100 pitkä, eikä ääretön kuten edelinen
  //tällä ei ole mitään tasonpeittoa jolloinka se ottaa ensimmäisen törmäimen
  mihin säde osu
  //osutun törmäimen avulla voidaan tehdä melkein mitä tahansa, eritellä
  peliobjektit vaikka nimen tai merkin avulla
  if (Physics.Raycast(sade, out osuma, 100))
  {
    //tulostetaan osutun objektin nimi
    Debug.Log(osuma.collider.gameObject.name);
    //tulostetaan objektin merkki
    Debug.Log(osuma.collider.gameObject.tag);
```

```
    //osuttu peliobjekti voidaan vaikka tuhota
    Destroy(osuma.collider.gameObject);
  }
 }
}
```

Koodiesimerkki 12: Esimerkissä käytetään hyväksi Unityn tarjoamia tasoja ja poimitaan peliobjekteja, kunhan säde osuu johonkin törmäimeen.

Koodiesimerkissä 12 käytetään hiirtä, mutta kyseinen ohjelmankoodi toimii myös kosketusnäytöille. Muuttamalla hieman kyseistä ohjelmaa voidaan vaivatta käyttää samaa tekniikkaa tablettilaitteissa. Koodiesimerkissä 13 on muunneltu sädeammuntaa toimivaksi kosketusnäytöllisessä laitteessa. Hiiren sijaan käytetään käyttäjän tekemiä kosketuksia näyttöön. Koodissa käytetään silmukkaa, sillä sormia voi olla 5 yhtä aikaa ruudulla ja jokaisesta sormesta ammutaan säde. (11)

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Esimerkki : MonoBehaviour {

//Tasonpeitto tämän tason alla olevat objektit otetaan huonioon
public LayerMask maa;

 void Start()
 {
 }

 void Update()
 {
 //jos ollaan Unity-editorissa
  if (Application.isEditor)
  {
     //kun painetaan hiirtä ammutaan säde
     if (Input.GetMouseButtonDown(0))
     {
      //Säde mikä ammutaan, tulee kamerasta suoraan pelimailmaan missä
     hiiri on sillä hetkellä
     Ray sade = Camera.main.ScreenPointToRay(Input.mousePosition);
     RaycastHit osuma; //mihkä osutaan

     //jos osutaan maahan tulostetaan teksti "Osuit maahan"
     //mitään muuta ei huomioida
     if (Physics.Raycast(sade, out osuma, Mathf.Infinity, maa))
     {
        Debug.Log("Osuit maahan");
     }
      //tässä on käsin määritelty säde joka 100 pitkä, eikä ääretönkuten
     edelinen
     //tällä ei ole mitään tasonpeittoa jolloinka se ottaa ensimmäisen törmäimen
     //mihin se osuu
     //osutun törmäimen avulla voidaan tehdä melkein mitä tahansa, eritellä
     objektit vaikkapa nimen tai merkin avulla
     if (Physics.Raycast(sade, out osuma, 100))
     {
```

```
    //tulostetaan osutun objektin nimi
      Debug.Log(osuma.collider.gameObject.name);
      //tulostetaan objektin merkki
      Debug.Log(osuma.collider.gameObject.tag);
      //osuttu peliobjekti voidaan vaikka tuhota
      Destroy(osuma.collider.gameObject);
    }
  }
}
//Tämä toimii tablettilaitteissa, koska oletetaan ollaan joko Unity-editorissa
tai itse laitteella
else
{
  //Katsotaan joka ikinen kosketus mikä tulee rudulle
  foreach(Touch kosketus in Input.touches)
  {
    //ja sormi on juuri laitettu ruudulle, ammutaan säde
    if(kosketus.phase == TouchPhase.Began){
      //käytetään samaa ohjelmakoodia kuin Unity editor osassa
      Ray sade = Camera.main.ScreenPointToRay(kosketus.position);
      RaycastHit osuma;
    if (Physics.Raycast(sade, out osuma, Mathf.Infinity, maa))
    {
      //tulostaa Macin Xcodessa nämä tekstit
      Debug.Log("Osuit maahan");
    }
    if (Physics.Raycast(sade, out osuma, 100))
    {
      //tulostaa Macin Xcodessa nämä tekstit jos laite on kytketty Maciin
      Debug.Log(osuma.collider.gameObject.name);
      Debug.Log(osuma.collider.gameObject.tag);
      Destroy(osuma.collider.gameObject);
    }
  }
 }
}
}
}
```

Koodiesimerkki 13: Koodi joka toimii sekä Unity-editorissa, että itse mobiililaitteessa. Ympäristön tunnistus on eroteltu helpolla if else vertauksella.

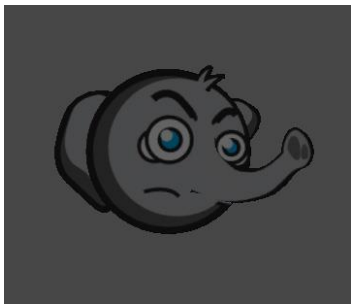
3.9.2 Malli

Jokaiselle peliobjektille voidaan määrittää malli. Mallin avulla peliobjektille saadaan jonkinlainen ulkomuoto. Mallit tuovat oman sävynsä peliin, jotta se ei olisi täynnä pelkkiä kuutioita tai ympyröitä, mikä olisi nykypäivän peleissä tylsän näköistä. Kuvassa 13 on Bitter Rolling päähenkilön malli.

Kuva 13: Malli pelattavasta hahmosta, mallilla ei ole tekstuuria tai minkäänlaista väriä.

3.9.3 Tekstuuri

Tekstuuri tuo mallille värin ja ulottuvuuden. Tekstuurin avulla hahmoista saadaan värikkäitä ja persoonallisen näköisiä.



Kuva 14:Pelattava hahmo tekstuurin kanssa. Tekstuuri tuo jo näyttävyyttä pelattavaan hahmoon.

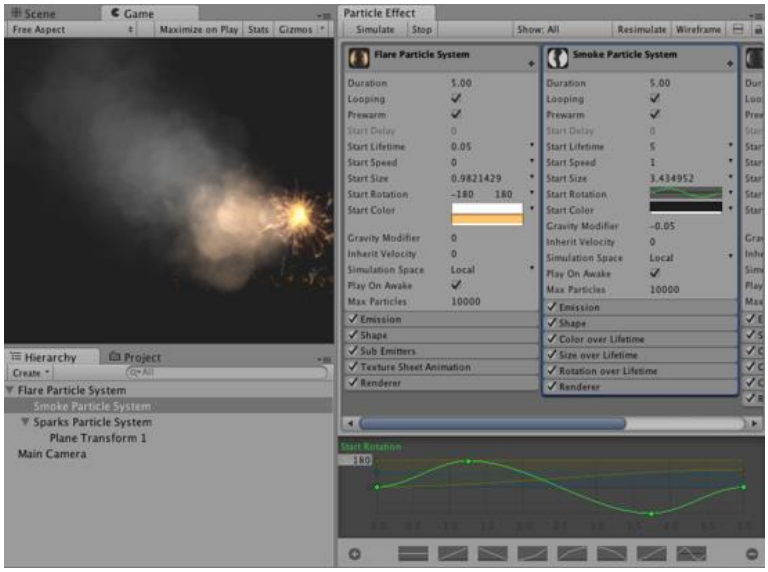
3.9.4 Materiaali

Materiaali on peliobjektin väri tai jokin tekstuuri. Muita vaihtoehtoja ei ole. Väri on yleensä jokin tietty väri ja tekstuuri voi sitten tehdä mallista yksityiskohtaisemman. Kuvassa 14 peliobjektiin on liitetty materiaali ja materiaalille on annettu tietty tekstuuri. Materiaalille voi antaa myös normaalin värin, tekstuurin käyttö ei ole välttämätöntä mutta tekstuuri on huomattavasti paremman ja yksityiskohtaisemman näköinen kuin pelkkä malli.

3.9.5 Efektit

Pelissä käytetään efektejä kuvaamaan muun muassa: savua, valoa, räjähdyksiä tai pyritään tekemään huomiota herättäviä tapahtumia. Efekti voi olla esimerkiksi avustaja, joka avustaa pelaajaa läpi kentän. Seuraamalla efektiä pelaaja löytää helposti kohti maalia.

Partikkeliefektiä käytetään savun, räjähdysten, tulen ja pilvien tekemiseen. Unityssä on kattava apuväline partikkeliefektien tekemiseen. Sillä pystyy muuttamaan efektin muotoa, kokoa, aluetta mihin se leviää, väriä sekä efektin nopeutta. Kuvassa 15 näkyy dynamiitin lanka palamassa, siitä syntyy savua ja palamisesta tulee partikkeleita, jotka kuvaavat itse kipinöitä.



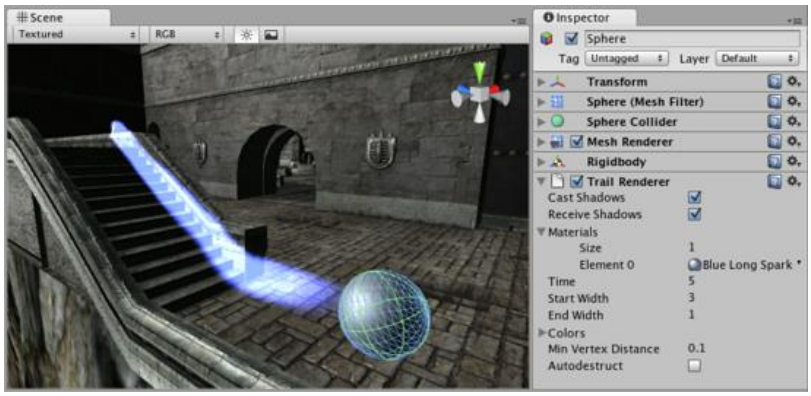
Kuva 15: Unity partikkelien tekotyökalu, missä voi tehdä erilaisia partikkeliefektejä. (12)

Linssiheijastus simuloi kameranlinssissä tapahtuvaa valon heijastumista. Tämän efektin avulla pyritään luomaan tunnelmaa. Jos voimakas valonlähde osuu kameran linssiin, siitä syntyy voimakkaita läiskiä ja juovia kameranlinssiin. Peleissä tämä nähdään tilanteessa, kun aurinko on tietyssä kulmassa pelaajaan nähden. Auringosta tulee läiskät ja juovat kohti pelaajaa. Kuvassa 16 tulee auringosta läiskä kohti pelaajahahmoa, koska ollaan tietyssä kulmassa aurinkoon nähden.(13)

Kuva 16: Linssiheijastus tulee kohti pelaajaan auringosta päin, efektin voi nähdä juovina ja läiskinä. (14)

Peliobjektit voivat jättää jälkeensä renderöintivanan. Renderöintivanaa voi käyttää opastukseen kentän läpi, jolloin pelaajan on helppo seurata kyseistä vanaa. Vanaa voi käyttää myös pelillisiin asioihin. Esimerkiksi pelaajahahmo kävelee pitkin tasoa ja jättää jälkeensä happoa, joka voi vahingoittaa vihollisia. Efektiä käytetään kuvaamaan si-

tä kohtaa, missä kyseinen happo on. Ohjelmakoodi hoitaa tarkistuksen, onko vihollisia kyseisen vanan päällä. Jos vihollisia löytyy, tehdään tarvittavat asiat vanhingon tai jonkin muun tuottamiseksi. Kuvassa 17 on pallo joka jättää jälkeensä renderöintivanan, jota pelaaja voi vaikka seurata.



Kuva 17: Kuvassa näkyy renderöintivanan minkä pallo jättää jälkeensä, kun se liikkuu pelissä. (15)

Unityssä on yksi renderointikomponentti ja se on kamera. Kamera näyttää pelimaailman pelaajalle. Kamera renderöi kaikki pelissä olevat peliobjektit. Kamera voi laittaa seuraamaan pelaajaa, laittamalla sen lapsiobjektiksi pelaajan alle. Tällöin kameran ollessa pelaajahahmon lapsiobjekti, kamera käyttää pelaajahahmon paikkaa ja kamera seuraa kätevästi pelaajahahmoa. Kamerasta voi tehdä ortografisen tai perspektiivisen. Ortografista käytetään 2D-peleissä, jolloin 3D-peliobjektit piirretään 2D:nä. Perspektiivisessä kamerassa voidaan kamera laittaa pelaajan ympärille, suoraan pelaajan taakse tai kuvaamaan pelaajan näkökenttää eli silmien korkeudelle.

3.9.6 Transform komponentti

Tämä komponentti sisältää peliobjektin paikan pelimaailmassa, sen kääntökomponentin sekä skaalauksen. Peliobjektin paikka pelimaailmassa on määritelty Vector3-tyyppiseksi, jolloin sillä on x-, y- ja z-koordinaatit. Objektin käännöstä voidaan muutella asteittain x-, y- ja z-akselin suuntaisesti. Skaalausta pystytään muuttamaan samojen akseleiden avulla. Mallit kannattaa jo projektiin tuodessa tehdä riittävän isoiksi ja skaalauksen avulla ainoastaan pienentää niitä, mikäli siihen on tarvetta. (16)

Aiemmin tuli esille että mallit kannattaa laittaa jonkin toisen peliobjektin lapseksi, jos on ongelma kääntymisen kanssa. Toisen lapsiobjektiksi laittamisesta on hyötyä myös muissa tapauksissa. Ajatellaan, että yhdessä Scenessä on sata vihollista. Hierar-

kianäkymä voi näyttää melko kaoottiselta, jos kaikki ovat siellä yhdessä pötkössä. Aloitettaessa scenen tekemistä, tulee ensimmäiseksi määrittää, missä nämä viholliset tulevat olemaan. Kannattaa luoda tyhjä peliobjekti 0-paikkakoordinaateilla, 0-käännös arvoilla ja skaalaus 1-suhteella. Sen jälkeen laitetaan jokainen vihollinen tämän objektin lapseksi, jolloin hiearkia näkymä ei täyty näistä kaikista. Luotu emo-objekti kannattaa nimetä viholliset nimiseksi ja jos haluaa deaktivoida kaikki viholliset scenestä, deaktivoidaan emo-objekti. Kun jokin emo-objekti deaktivoidaan, Unity kysyy halutaanko myös deaktivoida kaikki lapsiobjektit. Tämän avulla on helppo piilottaa näkyvistä, tiettyjä objekti ryhmiä Unity-editorissa.

3.10 Unity pelimoottorina mobiilipeleissä

Unity on erittäin suosittu pelimoottori mobiilikehityksessä tänä päivänä. Sillä on tiettyjä hyviä ja myös huonoja puolia. Seuraavassa käydään läpi sellaisia ominaisuuksia, joita kannattaa miettiä ohjelmoitaessa pelejä tai käytettäessä Unityä.

Unityn hyötyjä ovat sen helppo käyttöliittymä ja komponenttimaiset objektit. Lisäksi hyvä ominaisuus on se, että graafikot ja ohjelmoijat työskentelevät samoilla työkaluilla. Jos tekijöillä on käytössään Unityn oma versionhallinta, se helpottaa tiedostojen siirtämistä tekijältä toiselle. Unity on suhteellisen helposti opittavissa ainakin sellaisille henkilöille, jotka ovat aikaisemmin käyttäneet jotain pelimoottoria pelien tekemisessä. Unityn ominaisuus kääntää peli usealle alustalle tulee yritykselle huomattavasti halvemmaksi, kuin ostaa jokaiselle mahdolliselle ympäristölle oma kehitysympäristönsä. Unity-editorissa pystyy reaaliajassa muuttamaan arvoja, vaihtamaan malleja sekä lisäämään uutta koodia, ilman että peli on käännettävä uudelleen. Useissa pelimoottoreissa koko projekti on käännettävä uudelleen, jos sitä halutaan testata. Käännös vie yleensä minuutista kymmeniin, pelimoottorista riippuen. Unityssä käännösaika ei yleensä ole 30 sekuntia pidempi. Peliin pystyy tekemään muutoksia samanaikaisesti sitä pelatessa, mutta muutokset eivät kuitenkaan tule voimaan kesken pelin. Esimerkiksi testattaessa eri nopeuksia, voidaan lopuksi testin päätyttyä muuttaa arvo sopivaksi.

Haittoina voisi mainita komponenttimaisen ohjelmoinnin, sillä Unity ei noudata olio-ohjelmoinnin periaatteita. muodostimia ei käytetä ja luokkien periminen tuntuu tympeältä komponenttimaisuuden takia, koska useimmiten haetaan joku komponentti ja käytetään sen julkisia metodeja. Toisaalta se on myös hyödyllinen ominaisuus, sillä

kaikkien peliobjektien ei tarvitse toteuttaa fysiikkaa tai liikkua. Näiden ominaisuuksien lisääminen on kätevää niille objekteille, jotka niitä tarvitsevat. Se ei siis pakota jotakin tiettyä luokkaa perimään fysiikkaominaisuutta, mikäli joku sen ylempi luokka tarvitsee sen. Ominaisuus voi olla tälle luokalle täysin hyödytön. Tämä tekee Unityn luokkarakenteesta aika yksinkertaisen, olettaen että jokainen asia pidetään omana komponenttinaan. Unityssä piilee myös eräs oppimiseen liittyvä vaara pelimoottorin sisältäessä paljon valmiita komponentteja. Tarvitseeko ohjelmoijan itse ohjelmoida mitään vai käyttääkö hän ainoastaan valmiita Unityn komponentteja. Tämä voi johtaa siihen, että asioiden syvällisempi tutkiminen jää vähemmälle. Aloittelevalle ohjelmoijalle tämä on sinällään suuri apu. Oppimiskäyrä jää aika vaatimattomaksi, mikäli jatkossakin käyttää ainoastaan Unityn valmiita komponentteja.

4 MOBIILIALUSTAT

Mobiilialustat ovat nykyään jakautuneet Android, Apple iOS ja Microsoft Windows puhelimien kesken. Suurin ryhmä on Android, sillä laitevalikoima on laaja. Hyvänä toisena tulee Apple ja Windows:lla on oma lohkonsa markkinoista. Seuraavassa käydään läpi Apple- ja Android-alustat ja niille kehittäminen, tarvittavat laitteeet ja lisenssit. Windows puhelimia ei käydä läpi.

4.1 Apple

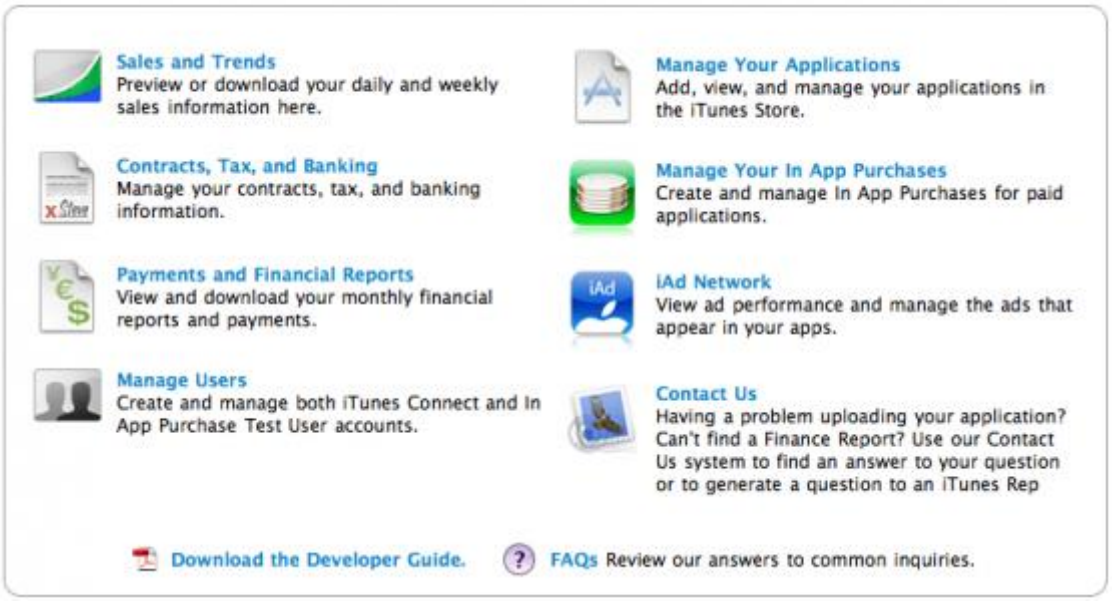
Jotta Applen alustoille pystyy tekemään pelejä tai ohjelmistoja, tulee rekisteröityä Apple kehittäjäksi. Rekisteröityminen on ilmaista ja sen myötä pystyy tekemään ohjelmistoja tai pelejä MAC OSX tietokoneille. Kuitenkin päästääkseen moobiilimaailmaan tulee maksaa 99$ vuodessa. Tämä antaa kaikki tarvittavat lisenssit ja oikeudet julkaista ohjelmia kaikille Applen kannettaville laitteille. Kehittäminen ei sinällään eroa Applen ja Androidin välillä, sillä samaan ohjelmistokoodiin pohjautuvan pelin tai ohjelmiston voi kääntää sekä mobiiliversioksi että Mac tietokoneelle. Kuitenkin kannettavissa laitteissa on omat rajoituksensa ja tavallisille tietokoneille suunnitellut pelit tai ohjelmistot eivät välttämättä pyöri mobiililaitteissa laitteiden heikon suorituskyvyn vuoksi.

4.1.1 Vaadittavat lisenssit ja laitteet

Pelin kehittämisprosessi on aika yksinkertainen, jos kehittäjä on valinnut itselleen työkaluksi Unityn. Kehittäjä tarvitsee itselleen Unity Pro -lisenssin ja Unity iOS -lisenssin, Mac OSX -tietokoneen, Xcode -ohjelmiston ja käyttäjän on oltava rekisteröitynyt Apple-kehittäjäksi. Unityssä itsessään on erittäin hyvät käännöstyökalut. Se hoitaa kääntämisen helposti Windows-ympäristöstä Apple ympäristöön. Unity luo automaattisesti Xcode -projektin, joka avataan Xcode -ohjelmistolla. Projektiin määritellään iOS lisenssitunnisteet ja tämän jälkeen pelin pystyy kääntämään mobiililaitteille. Prosessi on hieman hitaampi, jos yrityksessä on sekaisin Mac- ja Windows -ympäristöjä. Kun siirretään tietoja Windows -koneesta Mac -koneelle, Windows -ympäristöstä tulevat ominaisuudet on käännettävä Unityssä Max OSX -yhteensopiviksi ja tämä vie aikaa. Eniten aikaa vievät kaikki graafiset ominaisuudet, joten mahdollisuuksien mukaan grafiikat tulisi tehdä alun perin Mac -ympäristössä. Unityn versionhallinta helpottaa huomattavasti keskustelua näiden eri ympäristöjen välillä ja tekee tiedonsiirron ympäristöjen välillä erittäin helpoksi. Ilman versionhallintaa, projektista on luotava kaksi versiota: yksi Windows -pohjainen sekä toinen, joka on käännetty Mac -ympäristöön yhteensopivaksi.

Jos kehittäjä ei halua käyttää Unityä, voi hän ladata Applen oman kehitystyökalun. Jotta peli saadaan kauppaan tai peliä pystytään testaamaan itse laitteella, on joka tapauksessa ostettava kehittäjälisenssi ja tämä kehitystyökalu ei ole läheskään yhtä kattava kuin Unity. Applen omassa kehitystyökalussa on itse rakennettava monet Unityn tarjoamat valmiit palikat. Ohjelmointikieli on tässä tapauksessa objective C.

Kun rekisteröityy Apple -kehittäjäksi, saa tarvittavat oikeudet tehdä Apple ID:n. Jokaisella ohjelmistolla tai pelillä tulee olla Apple ID, jotta sen voi laittaa Applen kauppaan ladattavaksi. Tämä tapahtuu internetissä Applen iTunes-sivustolla. Täällä pystyy myös hallinnoimaan mahdollisia pelin sisäisiä ostoja, katsoa raportteja pelin myynnistä sekä hallinnoimaan käyttäjiä. Kuvassa 18 on iTunes -sivustolta näkymä. Sinne kirjaudutaan Apple -kehittäjän tunnuksilla



Kuva 18: Apple iTunes, missä luodaan Apple ID, tarkastellaan myyntejä, hallinnoidaan käyttäjiä ja määritellään pelin sisäisiä ostoja. (17)

Kun peli tai ohjelmisto on luotu iTunesiin ja on luotu käyttäjätunnukset, projektille pystytään luomaan niin sanottu Provisional profile. Sen avulla peli tai ohjelmisto pystytään kääntämään itse laitteelle. Tähän profiiliin määritetään pelin tai ohjelman Apple ID sekä niiden laitteiden ID:t, joihin kyseisen pelin tai ohjelman voi asentaa. Jokaisella Apple laitteella on oma uniikki ID. Sen voi katsoa laitteen asetuksista tai kun laite kytketään Xcodeen, se saadaan näkyviin. Rekisteröitymisessä Apple antaa rekisteröidä sata laitetta ja nämä laitteet voi poistaa vasta vuoden kuluttua, kun Apple -lisenssi on uusittava. Yleensä joitain laitteita ei enää tarvita kehityksessä ja ne vievät turhaan yhden paikan tästä sadasta, joten lisenssin uusimisen yhteydessä voidaan poistaa kaikki käyttämättömät laitteet pois, jotta tilaa vapautuu uusille. Sata kuulostaa paljolta, mutta ne saattavat kulua loppuun hyvinkin pian, etenkin, jos kehityksen edetessä tulee uusia laitteita, jotka halutaan rekisteröidä. Profiili on luotava uudestaan uusille laitetunnisteille. Käytettäessä vanhaa profiilia, jossa ei ole uusien laitteiden ID-numeroita, peliä tai ohjelmaa ei pysty asentamaan näille uusille laitteille. (18)

Kun profiili on ajan tasalla, avataan Unityn luoma projekti ja sen asetuksissa määritetään käytettävä profiili. Tämän jälkeen on mahdollista tehdä paketti, jonka voi jakaa kaikille, joiden laitteet on rekisteröity profiiliin. Vaihtoehtoisesti voi suoraan siirtää ohjelman tai pelin laitteeseen, joka on kytketty Mac -tietokoneeseen. Xcodesta voi valita, kääntääkö ohjelman suoraan laitteelle vai lähettääkö ohjelman käyttäjälle tai tes-

taajalle internetin välityksellä. Ohjelmien asentaminen onnistuu jokaisesta Mac -tietokoneesta, kunhan laitteen ID on provisional profiilissa.

4.1.2 Applen rajoitteet

Mobiilipelien kehityksessä on joitain yleisiä rajoitteita. Nykyisissä tietokoneissa on useita gigatavuja kovalevytilaa, muistia gigojen edestä ja prosessorin laskentateho on päätähuimaava. Mobiililaitteissa näin ei ole. Mobiilialustalle vältetään tekemästä kovinkaan massiivisia pelejä, vaan keskitytään yksinkertaisten ja koukuttavien pelien tekemiseen.

Apple rajoittaa pelejä myös sisällön suhteen: väkivaltaiset, rasistiset, poliittiset tai eroottiset pelit hylätään. Käyttäjien tietoja ei myöskään saa kerätä ilman heidän lupaansa. (19)

4.2 Android

Android -kehittäminen eroaa suuresti Applelle kehittämisestä. Android -alustalle julkaiseminen on täysin ilmaista. Jos omistaa tietokoneen ja Android laitteen, voi aloittaa kehittämisen. Android tarjoaa omat kehitystyökalunsa ja sen avoimuuden takia löytyy paljon valmiita ohjelmakehyksiä, joiden avulla voi helpottaa tai laajentaa kehitystä. Ohjelmointikielenä on Java ja ohjelmien laittaminen kauppaan ei vaadi mitään ihmeellistä. Android -pelien julkaisualustana toimii Google Apps. Sisällön suhteen on melkein samat rajoitukset kuin Applella. Googlen etu on se, ettei kehittäjän tarvitse maksaa mitään kehittäjämaksuja. Kunhan omistaa Google tilin, voi laittaa ohjelmansa muiden ladattavaksi.(20)(21)

Unityn tapauksessa riittää tietokone, jokin Android laite ja lisenssi Unityltä, jotta pelistä voi kääntää Android -version. Unity kääntää suoraan editorista pelin laitteelle ja kehittäjä pääsee välittömästi testaamaan peliä.

Android markkinoilla on omat haasteensa. Android markkinoilla on erittäin suuret valikoimat erilaisia laitteita. Pelin testaus yksittäisellä laitteella ei takaa sitä, että se toimii jokaisessa markkinoilla olevassa Android laitteessa. Google on kehittänytkin hyvät suodattimet, mitä ominaisuuksia laitteelta tulee löytyä, jotta pelin pystyisi asentamaan laitteelle. (22)

5 PELI-IDEA

Tässä kappaleessa käydään läpi oman pelidemon asiat: peli-idea, pelin ominaisuuksia ja graafinen tyyli. Pelin suunnittelu aloitettiin helmikuussa 2012 neljän opiskelijan muodostaman ryhmän voimin. Projektin edetessä kaksi henkilö lopettivat ja jäljelle jäi itseni lisäksi Einari Lavaste. Häneltä sain osan grafiikoista ja saimme kehitettyä pelidemoa eteenpäin.

5.1 Idea

Ideana oli tehdä fysiikkapohjainen tasohyppely. Ongelmanratkonnassa pyrittiin käyttämään hyödyksi kosketusnäytölle piirrettäviä hyökkäyksiä. Tarkoituksena oli tehdä helposti ohjattava pelihahmo, jotta mahdollisimman pienet lapset pystyisivät pelaamaan peliä ilman sen suurempaa opastusta. Graafisella tyylillä pyrittiin vetoamaan nuorempiin pelaajiin. Päähenkilöinä pelissä toimivat eläimet, joiden elämää ihminen uhkaa jokapäiväisellä häirinnällään.

Pelin eri kenttiin tulee kerättävää tavaraa ja salaisia paikkoja löydettäväksi. Pelistä löytyy tarinankerrontaa kuvien avulla. Puhuttua tai kirjoitettua dialogia pelistä ei löydy. Itse tarina kuvataan täysin ilmeiden avulla ja välillä tulevien kuvasarjojen avulla.

Tavoitteena on tehdä miellyttävät ja hauskat hahmot peliin. Mahdolliset lisähahmot olisivat mahdollisimman ainutlaatuisia, kun niitä verrataan keskenään. Ominaista niille on se, että ne olisivat aina eri elämiä. Omaisivat sellaisia erikoiskykyjä, mitä muilla ei ole ja eläinten luonteet olisivat mahdollisimman erilaisia. Päähahmona tässä pelidemossa on norsu.

5.2 Kontrollit

Yksi myyntivalteista on helppokäyttöiset kontrollit. Kun käyttäjä näpäyttää ruutua, hahmo liikkuu näpäytyksen suuntaan. Jos käyttäjä pitää sormea ruudussa kiinni ja tekee vetoliikkeen sormellaan sivuttaissuunnassa. Hahmo ampuu hahmolle ominaista kykyä. Pelidemon norsu ampuu vettä. Toinen vaihtoehto ampumiselle on aaltomainen liike. Jos käyttäjä vetää sormeaan ruudulla ja tekee tarpeeksi suuria aaltoja, norsun tapauksessa norsu hyökkää aaltomaisella vesisuihkulla. Painamalla itse hahmoa pidemmän aikaan, hahmo suorittaa sille ominaisen erikoiskyvyn.

5.3 Mobiilialustan haasteet

Useat pelit ovat liian monimutkaisia ja esimerkiksi pelin ohjauksen opettelemisessa menee kauan aikaa. Ja kun on oppinut kontrolloimaan hahmoa, huomaa, että kontrollit ovat erittäin huonot ja epäloogiset. Mobiililaitteissa ainoa tapa ohjata hahmoa on kosketusnäyttö, ei ole olemassa nappeja tai mitään muuta. Tämä on johtanut siihen, että jotkut pelintekijät täyttävät ruudun monilla erilaisilla napeilla, joista tapahtuu jotakin. Kannettavien puhelimien kohdalla näytöt ovat kooltaan rajalliset. Tämän takia tulee ongelmaksi joko koko tai tila. Usein nappeihin on joko liian vaikea osua omalla sormella tai napit peittävät ison osan ruudusta. (23)

6 PELISUUNNITTELU

Seuraavassa osiossa käydään läpi pelisuunnittelu yleisesti, sen muutamia haasteita ja ongelmia. Pelisuunnittelu saatetaan kokea mitättömänä, kuitenkin projektin edetessä usein huomataan, että hyvin suunniteltu on puoliksi tehty.

6.1 Yleistä

Pelin olennainen osa on sen suunnitteleminen. Ilman hyvää suunnitelmaa on miltei mahdotonta tehdä loogista peliä. Jos ideat hyppivät edestakaisin projektin edetessä ja ideoita muutetaan koko ajan, se voi johtaa loputtomaan kierteeseen ja tuloksena ei ole mitään valmista. Suunniteltaessa on hyvä miettiä pelin genre, ikäluokat, jolle se suunnataan, pelin ominaisuuksia, hahmoja ja jokin tarina.

Hyvään suunnitteluun kuulu hyvä dokumentointi. Usein näin ei ole. Internetissä liikkuu monta tarinaa siitä, kuinka projektit ovat epäonnistuneet huonon dokumentaation tai sen päivityksen johdosta. Jos dokumentaatiota ei päivitetä, kehitysryhmällä ei ole asiakirjaa, mistä tarkistaa jonkin asian ollessa epäselvä.

6.2 Ongelmat

Yleensä pelin alkutaipaleilla tehdyt suunnitelmat eivät välttämättä pysy sellaisenaan loppuun asti. Pelinkehitys on yleensä sitä, että idea kehittyy ja elää pelin edetessä. Kuitenkin jos aloitetaan liiallisella ideoinnilla, se ei edistä itse kehitystä lainkaan. Pitäisi miettiä mahdollisimman hyvin pelin ydin, jota ei ikinä tulisi muuttaa, ettei aleta

kesken prosessin tekemään aivan toisen genren peliä, kuin mitä alussa on mietitty. Tarinankin muuttaminen ja uudelleen kirjoittaminen voi johtaa kummallisuuksiin, jos pelimekaniikat on suunniteltu tarinan ympärille.

Toteuttamisessa yleensä törmätään resurssipulaan, jolloin joudutaan tekemään pitkiä päiviä pienen kehitysryhmän kanssa. Tämä sinällään ei palvele ketään, koska ylityöt sekä mahdollinen uusi työvoima maksavat. Ongelmana voi olla se, että pieni ryhmä palaa loppuun, eikä heillä välttämättä ole tarvittavaa energiaa projektin loppuun viemiseen. (23)

6.3 Kompastuskivet

Peleissä on yleensä vaikeata hahmottaa, paljonko aikaa mikin pelinosan toteuttaminen vaatii. Myös pelimoottorin hankinnassa on otettava huomioon monia asioita. Ostetaanko koko paketti vai tietyt osat kuten fysiikka ja renderöinti, ja muut työkalut tehdään itse kyseiseen pelimoottoriin. Tällöin jäljelle jää muun muassa animaatioiden, käyttöliittymän, kenttäeditorin ja tapahtumien toteutus. (23)

7 LOPPUSANAT JA YHTEENVETO

Mobiilipelin toteutuksessa minua on ärsyttänyt Unityn tapa käsitellä muistia. Peliobjekteja ei pysty itse luomaan keko-muistiin. Aina, kun uusi scene ladataan, tuhotaan kaikki muut paitsi uudessa scenessä olevat peliobjektit. Useilla muilla kehitysympäristöillä ohjelmoitaessa esimerkiksi pelaaja voidaan tallettaa keko-muistiin, jolloin se elää läpi pelin, ellei sitä tarkoituksellisesti tuhota jossain kohtaa. Unityssä objektien elinaikaan ei siis pääse suoraan vaikuttamaan. Peliobjektiin voidaan määrittää, ettei sitä tuhota, kun ladataan uusi scene. Tämä kuitenkin pätee vain niihin objekteihin, jotka perivät Unity -monobehaviourin. Kun peritään tämä monobehaviour -luokka, ei ole mahdollista ohjelmoida olio-ohjelmoinnin oppien mukaisesti.

Yleisesti ottaen Unity on ollut erittäin suuri apu opinnoissa, sen avulla olen päässyt hyvin ohjelmoinnin maailmaan. Tulevaisuus näyttää myös lupaavalta. Tämän hetkisen tilanteen mukaan mobiilipelejä kehitetään vielä useita vuosia ja opitut tiedot ja taidot eivät mene hukkaan. Kuitenkin voin todeta, että oma kiinnostukseni on isomman skaalan projekteissa. Mobiilipeleistä on hyvä aloittaa ja hankkia tarvittavaa kokemus-

ta. Pelien tekeminen ei ole missään tapauksessa helppoa. Se vaatii aikaa, resursseja ja erittäin paljon taitoa.

Itse pelidemo Bitter Rolling on saatu hyvin aluilleen, ja sitä on tästä hyvä lähteä jatkokehittämään. Idea on suhteellisen yksinkertainen ja perusominaisuuksista on tehty jo suurin osa. Pelisisältöä pitää laajentaa ja ominaisuuksien testaamista vaaditaan, joten tästä on vielä pitkä matka valmiiseen peliin.

8 LÄHTEET

1. **Handys,** Hagebuk MT-2000. [Online] [Viitattu: 15. huhtikuuta 2013.] http://www.handy-sammler.de/Handys/Hagenuk_MT-2000.htm

2. **Nokia,** The Nokia story. [Online] [Viitattu: 15.huhtikuuta 2013.] http://www.nokia.com/global/about-nokia/about-us/the-nokia-story/

3. **Unity,** Release Archive. [Online] [Viitattu: 11.maaliskuuta 2013.] http://unity3d.com/unity/whats-new/archive

4. **Unity,** Fast Facts. [Online] [Viitattu: 17.maaliskuuta 2013.] http://unity3d.com/company/public-relations/

5. **Unity,** Unity Licences. [Online] [Viitattu: 17.maaliskuuta 2013.] https://store.unity3d.com/

6. **Unity,** Asset Server. [Online] 2013. [Viitattu: 19.maaliskuuta 2013.] http://docs.unity3d.com/Documentation/Manual/AssetServer.html

7. **Unity,** The Game Object. [Online] 2009.[Viitattu: 11.maaliskuuta 2013.] http://docs.unity3d.com/Documentation/Components/comp-GameObjectGroup.html

8. **Unity,** Rigidbody.[Online] 2013. [Viitattu: 11.maaliskuuta 2013.] http://docs.unity3d.com/Documentation/Components/class-Rigidbody.html

9. **Macey, J**. Ray Tracing and other Rendering Approaches. [Online] [Viitattu: 21.maaliskuuta 2013.] http://nccastaff.bournemouth.ac.uk/jmacey/CGF/slides/RayTracing4up

10. **Unity,** Raycast. [Online] [Viitattu:21.maaliskuuta 2013.] http://docs.unity3d.com/Documentation/ScriptReference/Physics.Raycast.html

11. **Unity,** Input Touches. [Online] [Viitattu: 31.maaliskuuta 2013.] http://docs.unity3d.com/Documentation/ScriptReference/Input-touches.html

12. **Unity,** Particle System [Online] 2012. [Viitattu: 31.maaliskuuta 2013] http://docs.unity3d.com/Documentation/Manual/ParticleSystems.html

13. **Laitetekniikka.com,** Linssiheijastus. [Online] [Viitattu: 9.huhtikuuta 2013.] http://www.laitetekniikka.com/digikuvaus/linssiheijastus.html

14. **Unity,** Lens Flare [Online] 2008. [Viitattu 31.maaliskuuta 2013] http://docs.unity3d.com/Documentation/Components/class-LensFlare.html

15. **Unity,** Trail Renderer. [Online] 2011. [Viitattu: 31.maaliskuuta 2013.] http://docs.unity3d.com/Documentation/Components/class-TrailRenderer.html

16. **Unity,** Transform. [Online] [Viitattu: 4.huhtikuuta 2013.] http://docs.unity3d.com/Documentation/Components/class-Transform.html

17. **iPhone Developers,** iTunes Connect Still Open, but for How Long? [Online] 2010. [Viitattu 8.huhtikuuta 2013]

18. **Apple,** About iOS Development Team Administration. [Online] 2012. [Viitattu: 8.huhtikuuta.2013] http://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/DevPortalGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011159-CH1-SW1

19. **Apple,** App Store Review Guidelines. [Online] 2010. [Viitattu: 8.huhtikuuta 2013.] http://stadium.weblogsinc.com/engadget/files/app-store-guidelines.pdf

20. **Android,** Android Developer. [Online] [Viitattu: 8.huhtikuuta 2013.] http://developer.android.com/index.html

21. **Google,** Google Apps Marketplace Listing Approval. [Online] 2012. [Viitattu: 8.huhtikuuta 2013.] https://developers.google.com/google-apps/marketplace/listing_approval

22. **Google,** Filters on Google Play. [Online] [Viitattu: 9.huhtikuuta 2013] http://developer.android.com/google/play/filters

23. **Ian Fisch,** Game Design Process Pitfalls [Online] [Viitattu: 10.huhtikuuta 2013] http://www.gamasutra.com/view/feature/4017/10_game_design_process_pitfalls.php?print=1