Lappeenrannan teknillinen yliopisto
Tuotantotalouden tiedekunta
Tietotekniikan koulutusohjelma

Kandidaatintyö

Joona Hasu

PELIOHJELMOINNIN ALKEET

Työn tarkastaja(t): Tutkijatohtori Jussi Kasurinen

Työn ohjaaja(t): Tutkijatohtori Jussi Kasurinen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tuotantotalouden tiedekunta
Tietotekniikan koulutusohjelma

Joona Hasu

Peliohjelmoinnin alkeet

Kandidaatintyö
2014
70 sivua, 11 kuvaa, 2 liitettä

Työn tarkastajat: Tutkijatohtori Jussi Kasurinen

Hakusanat: peliohjelmointi, alkeet, UnityScript, Unity3D
Keywords: game programming, basics, UnityScript, Unity3D

Tämä kandidaatintyö tutkii peliohjelmoinnin mahdollisuuksia ohjelmoinnin alkeiden opetuksessa sekä Unity3D-ohjelman toimivuutta 2D-pelinkehityksessä. Tutkimuksen tavoitteena oli luoda Peliohjelmoinnin alkeet -kurssille kurssirunko sekä todistaa Unity3D-ohjelman toimivuus 2D-pelinkehityksessä kehittämällä esimerkkipeli.

Tuotoksena kurssille kehitettiin kuusiosainen kurssirunko, johon myös suunniteltiin täysin aloittelijoille tarkoitettu vapaaehtoinen UnityScriptin alkeet osio. Ohjelmoinnin aloittelijoille suunniteltu kurssi käy läpi Unity3D-ohjelman perusteet sekä tutustuu skriptauksen alkeisiin. Kurssirunko käy 2D-pelinkehitykseen liittyviä konsepteja loogisesti läpi.

Työn toisena tavoitteena oli esimerkkipelin kehitys ja Unity3D-ohjelman 2D-pelinkehitystyökalujen testaus. Pelin tekeminen onnistui yli odotusten. Vaikka peli on erittäin yksinkertainen ja lyhyt, se toimii esimerkkipelinä mainiosti.

Johtopäätöksenä voidaan todeta, että Unity3D-ohjelman 2D-työkalut ovat toimivia laadukkaiden 2D-pelien kehittämiseen. Tuotoksena laadittu kurssirunko antaa suuntaa kurssien rakentamiseen peliohjelmoinnin aloittelijoille. Pelinkehitys kurssin suorittamisen jälkeen opiskelijalla on mahdollisuudet ja perustaidot jatkaa kehittymistä ja opiskelua tahtomallaan suunnalla.

ABSTRACT

Lappeenranta University of Technology
School of Industrial Engineering and Management
Degree Program in Computer Science

Joona Hasu

Game programming basics

Bachelor's Thesis

70 pages, 11 figures, 2 appendices

Examiners: Postdoctoral researcher Jussi Kasurinen
Keywords: game programming, basics, UnityScript, Unity3D

This bachelor's thesis researches possibilities of game programming to be used to teach programming basics and to test the 2D capabilities of Unity3D Game Engine. The aim of this thesis was to create a basic framework for Game programming basics course and to test the 2D capabilities of Unity3D Game Engine by creating an example game.

Output of the research is a six-part framework for the course to which was included a voluntary UnityScript part for complete beginners to programming. This course was designed for programming beginners and it goes through the basics of Unity3D and familiarizes students to basics of scripting. The course framework goes logically through 2D game development concepts.

The second goal was to develop of an example game and to test 2D capabilities of Unity3D Game Engine. Development of the game succeeded beyond expectations. Although the game is very simple and short it works as an example game perfectly.

It can be concluded that the 2D tools of the Unity3D Game Engine are functional for developing high-quality 2D games. The designed course framework provides a good base to build game programming courses to beginners. After completing the game programming course student has potential and basic understanding to continue improving and studying programming in a desired direction.

ALKUSANAT

Tämä tutkimus on tehty Lappeenrannan yliopistolle kandidaatintyönä. Tutkimuksen tekeminen on ollut erittäin mielenkiintoista ja haastavaa. Haluan esittää parhaimmat kiitoksen työn ohjaajalle Tutkijatohtori Jussi Kasuriselle kannustavasta, kriittisestä ja asiantuntevasta ohjauksesta sekä kärsivällisyydestä.

Työn tekeminen vei paljon aikaa ja voimia. Haluan myös kiittää läheisiäni kannustuksesta ja tuesta.

Lappeenranta 20.12.2014

Joona Hasu

SISÄLLYSLUETTELO

SYMBOLI- JA LYHENNELUETTELO

ACM DL	Association for Computing Machinery Digital Library
GDD	Game Design Document
GUI	Graphical User Interface
HTML5	HyperText Markup Language 5
HUD	Head-Up Display
IEEE Xplore	Institute of Electronical and Electronics Engineers
LUT	Lappeenrannan teknillinen yliopisto
MOOC	Massive Open Online Course
TMC	Test My Code
UI	User Interface

1. JOHDANTO

Tämän työn aihe on valittu mielenkiinnosta peliohjelmointiin ja sen opettamiseen sekä oppimiseen. Työn tavoitteena on luoda pohja peliohjelmoinnin alkeita opettavalle kurssille, joka sisältää myös esimerkkipelin suunnittelun ja toteutuksen. Valitsin aiheen juuri pelinkehityksen takia, koska uskoin sen tuovan työhön haastavuutta ja motivaatiota.

1.1. Tausta

Opinnäytetyö tehdään Lappeenrannan teknilliselle yliopistolle (LUT). Työn tarkoituksena on tutkia ja kehittää perusmateriaali Peliohjelmoinnin alkeet – kurssille. Tätä kurssia tarjotaan Lappeenrannan alueen lukioille. Kurssille kehitettyä aineistoa voidaan käyttää yliopiston erilaisissa peliaiheisissa kursseissa apuna.

Suomessa ja maailmalla peliala on hurjassa nosteessa. Peliala tarjoaa uramahdollisuuksia kehittäjille, artisteille, suunnittelijoille ja muusikoille sekä näyttelijöille. Pelejä hyödynnetään nykyään viihdekäytön lisäksi opetus-, kuntoutus- ja hoitoaloilla.

Suomessa on tunnistettu pelialan mahdollisuudet. Pelinkehityksen opetusta annetaan useassa koulussa monella eri koulutusasteella. LUT tarjoaa pelinkehityskursseja jo ohjelmointia osaaville opiskelijoille, mutta LUT:lta puuttuu alkeita opettava kurssi, johon tämä työ pyrkii luomaan perusmateriaalin.

Vaikka pelinkehitys tuntuu monimutkaiselta, on kurssin tarkoituksena tarjota konkreettiset välineet pelinkehityksestä tai ohjelmoinnista kiinnostuneille tulevaisuutta varten. Pelinkehityskurssin suorittamisen jälkeen opiskelijalla on mahdollisuudet ja perustaidot jatkaa kehittymistä ja opiskelua tahtomallaan suunnalla esimerkiksi www-ohjelmoinnissa.
Nykyään pelien kehittäminen ja rakentaminen alusta loppuun vaatii aikaa ja taitoa. Tästä syystä useat kehittäjät käyttävät hyväksi pelimoottoreita, jotka helpottavat pelinkehitystä huimasti. Pelimoottori on ohjelma, joka sisältää usein valmiin fysiikkamallinnuksen,

grafiikan piirtämisen ja komentojen syöttämisen [1]. Pelinkehittäjät luovat itse pelin käyttäen pelimoottorin tarjoamia työkaluja.

1.2. Tavoitteet ja rajaukset

Tämä kandidaatintyö tutkii peliohjelmoinnin mahdollisuuksia ohjelmoinnin alkeiden opetuksessa sekä Unity3D-ohjelman toimivuutta 2D-pelinkehityksessä. Työn päätavoitteena on luoda Peliohjelmoinnin alkeet -kurssi, joka on yhden periodin mittainen eli kuusiviikkoa pitkä. Työn toisena tuotoksena tehdään yksinkertainen esimerkkipeli, jota voidaan käyttää kurssin luento, harjoitus ja harjoitustyömateriaalien tekemiseen. Kurssi on alustavasti suunniteltu ohjelmointia aloitteleville opiskelijoille, minkä takia kurssin ja esimerkkipelin suunnittelussa on keskitytty ymmärrettävyyteen, yksinkertaisuuteen ja kokonaisuuksien esittämiseen luontevasti.

Suunnitellun kurssin tavoitteena on esitellä oppilaille kokonaiskuva, josta yksinkertaiset 2D pelit rakentuvat käyttäen Unity3D-pelimoottoria [2] ja JavaScript (UnityScript) ohjelmointikieltä [3]. Tarkoituksena on opettaa tarvittavat rakennuspalikat ja ajatustavat, jotta opittujen asioiden soveltaminen onnistuu käytännössä.

JavaScript ohjelmointikieli valittiin kurssilla käytettäväksi kieleksi, koska se on aloittelijaystävällisin Unity3D:n tukemista kielistä. Unity Technologies tarjoaa kattavat dokumentaatiot kaikista tukemistaan kielistä [4], mutta JavaScriptin suosio kehittäjien keskuudessa tarjoaa paljolti esimerkkimateriaalia ja ratkaisuja mahdollisissa ongelmatilanteissa [5]. Tarkoitus ei ole opettaa pelkästään käytetyn ohjelmointikielen syntaksia, eikä myöskään opettaa, että Unity3D, olisi ainut ratkaisu kehittää pelejä.

Työstä on rajattu pois vähemmän aloittelijaystävälliset teemat, kuten 3D pelienkehitys ja siihen liittyvät tekniikat. Työn tarkoituksena ei ole myöskään suunnitella kurssia, jossa oppilaat loisivat kaikki peliin tarvittavat materiaalit manuaalisesti. Tähän tarkoitukseen käytetään Unity Asset Storea [6], josta löytyy paljon hyviä materiaaleja 2D-projekteihin. Esimerkkipelin kehityksen aikana Unity Asset Storessa ei ollut teemallisesti sopivia

materiaaleja, joten käytin pelimaailman rakentamiseen OpenGameArt.org [7] sivuston tarjoamaa ilmaista materiaalia.

1.3. Työn rakenne

Luvussa kaksi käydään läpi työssä käytettyjä kirjallisuuslähteitä. Kolmannessa luvussa esitellään työssä käytetyt tutkimusmetodit. Luvussa neljä käydään läpi pelinkehityksen teoriaa ja opetusta. Luvussa viisi tehdään katsaus Unity3D-ohjelmaan. Luvussa kuusi avataan Peliohjelmoinnin alkeet -kurssin pohjaa sekä esitellään kurssirunkoa. Lisäksi luvussa seitsemän esitellään kurssia varten luotu esimerkkipeli ja analysoidaan käytettyjä ohjelmointitekniikoita. Kahdeksannessa luvussa pohdiskellaan työntuotoksia ja opittuja asioita. Yhdeksännessä luvussa käydään läpi jatkokehitysmahdollisuudet. Kymmenennessä luvussa on yhteenveto tutkimuksesta.

2. KIRJALLISUUSKATSAUS

Kirjallisuuskatsausta varten lähteitä haettiin lähinnä Google Scholar – hakukonetta käyttäen. Hakukoneen kautta kirjallisuutta löytyi mm. ACM DL (Association for Computing Machinery Digital Library) [8] ja IEEE Xplore (Institute of Electronical and Electronics Engineers) [9] tietokannoista. Hakusanoina käytettiin seuraavia: Unity3d, game development, 2d, 3d, game, teaching, education, learning, material, example, instructional, tutorial, CS0, CS1. Hakusanoilla kirjallisuutta löytyi todella paljon, joten käytettäviä lähteitä tuli karsia.

Työn kurssiosaa varten teoriapohjaa on alustettu tutustumalla erilaisiin tutkimusartikkeleihin. Näitä ovat mm. Caspersen & Bennedsenin Instructional Design of a Programming Course – A Learning Theoretic Approach [10] ja Giguettesin Pre-Games: Games to Introduce CS1 and CS2 Programming Assignments [11] sekä Panitz, Sung & Rosenberg The Journal of Computing Sciences in Colleges: Game Programming in CS0: A Scaffolded Approach [12]. Nämä artikkelit sisältävät katsauksen ihmisen oppimiseen, sen teoriaan ja opiskelun motivaatioon peliosa-alueita käyttämällä sekä peliohjelmointikurssin tarjoamiseen ensimmäisenä ohjelmointikurssina.

Työn peliosaa varten lähteenä käytetään Unity Technologiesin omaa dokumentaatiota Unity3d ohjelmalle sekä Javascript (UnityScript) kielelle. Lisämateriaalina käytetään Creightonin Unity 3.x Game Developement by Example [13] ja Pereiran Learning Unity 2D Game Development by Example [14] kirjoja.

3. RATKAISUMENETELMÄT

Tutkimusongelmien ratkaisu lähti liikkeelle tutustumalla Unity3D-ohjelmaan ja UnityScriptiin tarkoituksena luoda prototyyppi esimerkkipelistä. Prototyyppistä kehitettiin 4 kenttää sisältävä toimiva esimerkkipeli. Kehityksestä pidettiin dokumentaatiota, joka auttoi kurssirungon kehittämisessä. Kurssirunko saatiin kasattua tutkimalla kirjallisuutta ja muiden tarjoamia vastaavanlaisia kursseja sekä käyttämällä hyväksi omaa kokemusta esimerkkipelin kehityksestä.

Ongelmien ratkaisuun, etenkin kurssin suunnitteluun, käytettiin sovelletusti hyväksi Lukan konstruktiivisen tutkimusprosessin mallia [15]. Sen mukaan tutkimusprosessi voidaan jakaa seuraavasti (vapaasti suomennettu):

1. Etsi käytännöllisesti hyödyllinen ongelma, joka voi olla teoriallisesti hyödyllinen
2. Arvioi tutkimuskohteen pitkän aikavalin tutkimustyön potentiaalia
3. Kerää syvä ymmärrys aihealueesta
4. Kehitä ratkaisu ongelmaan ja kehitä ongelman ratkaiseva konstruktio
5. Kehitä konstruktiosta implementaatio ja testaa sen toimivuus
6. Arvioi kuinka laajalti ratkaisua voi hyödyntää
7. Tunnista ja analysoi tutkimuksen teoreettinen hyödyllisyys

Pelin suunnitteluun ja toteutukseen käytettiin hyväksi Unity3D:n tarjoamia 2D pelitutoriaaleja [16] ja dokumentaatioita sekä kirjallisuuskatsauksessa mainittuja Unity kirjoja. Alustava suunnitelma pelin tekemiseen oli seuraava:

1. Konseptin kehitys
2. Pelin testiversion kehitys
3. Dokumentointi
4. Pelin testaus ja viimeistely

4. PELINKEHITYS

4.1. Historia

Pelejä on kehitetty jo 1950-luvulta lähtien, mutta nämä olivat tutkimuslaitosten ja yliopistojen omia projekteja. Valtavirta pääsi nauttimaan pelaamisesta vasta 1970-luvulla, jolloin pelihalleissa päästiin nauttimaan peleistä ja ensimmäiset konsolit tuotiin myyntiin [17]. Nykyään pelaaminen tapahtuu kännyköillä, tietokoneilla ja konsoleilla. Peliala on myös nykyään yksi suurimmista teollisuusaloista ja tulee edelleen kasvamaan [18].

4.2. Pelinkehittäjät suomessa

Suomessa peliteollisuus on 2000-luvulla ollut viihdeteollisuuden kovimmin kasvava haara. Neogames arvioi Suomen pelinkehityksen ja pelipalveluiden liikevaihdon olleen vuonna 2013 noin 900 miljoonaa euroa. Kasvua vuodesta 2012 vuoteen 2013 on tapahtunut noin 260 %. Toimialan kokonaisarvoksi Neogames arvioi noin 2,2 miljardia euroa. [18]

Suomalaisia kansallisesti ja kansainvälisesti menestyneitä pelinkehittäjiä on useita. Näistä hyviä esimerkkejä ovat Rovio Entertainment, Remedy Entertainment ja Supercell, jotka ovat maailmanlaajuisesti tunnettuja ja arvostettuja. Uusia pelitaloja perustetaan jatkuvasti lisää. Rovio on tunnettu Angry Birds sekä Bad Piggies peleistään. Etenkin Angry Birds on maailmanlaajuisesti menestynyt mobiilipeli. Remedy Entertainment tuli tunnetuksi Max Payne pelisarjastaan. Take-Two Interactive osti oikeudet pelisarjaan vuonna 2002. Remedyn valttikortteja ovat mielenkiintoinen ja elokuvamainen tarinankerronta. Tämä näkyy etenkin heidän viimeisimmässä Alan Wake -pelisarjassaan. Tällä hetkellä Remedy kehittää Quantum Leap nimistä peliä Xbox One -konsolille sekä Agents of Storm mobiilipeliä iOS-käyttöjärjestelmälle. Supercellin tunnetuimmat pelit ovat Clash of Clans, Hay Day ja Boom Beach mobiilipelit.

4.3. Pelimoottorit

Pelimoottorilla tarkoitetaan ohjelmaa, joka sisältää usein valmiin fysiikkamallinnuksen, grafiikan piirtämisen ja komentojen syöttämisen. Pelimoottoreiden käyttö pelejä tehdessä ei ole pakollista, mutta etenkin aloitteleville kehittäjille ne ovat mieluisa vaihtoehto. Pelimoottorit mahdollistavat kehittäjän keskittymisen itse pelin luomiseen ja sisällön tuottamiseen.

Markkinoilla on useita pelimoottoreita aina maksullisista ilmaisiin. Useasta maksullisesta pelimoottorista löytyy myös ilmaisversio, mutta näillä kehitettyjä pelejä ei yleensä voi kaupallistaa. CryTek-yhtiön CryEngine3 ja Epic Games-yhtiön Unreal Engine 4 ovat teknisesti ja graafisesti erinomaisia. Scirra-yhtiön Construct2- tai Unity Technologies-yhtiön Unity3D-pelimoottori ovat erityisen hyviä aloittelijoille aloittelijaystävällisyyden ja ilmaisversioiden tarjoamien mahdollisuuksien takia.

Tähän työhön pelimoottoriksi valitulla Unity3D-pelimoottorilla ilmaisversiolla kehitetyt pelit ovat vapaasti kaupallistettavissa. Ilmaisversio on luonnollisesti ominaisuuksiltaan karsittu verrattuna maksulliseen versioon. Unity3D:n viimeaikainen teknillinen kehitys ja aloittelijaystävällisyys ovat tehneet siitä yhden suosituimmista pelimoottoreista aloittelijoiden ja Indie-kehittäjien keskuudessa.

4.4. Peliohjelmointi ja opetus

Pelisuunnittelun opetusta ei voi lähestyä pelkästään teoreettisesti, mikä pätee mihin tahansa suunnittelutyöhön. Opetetut konseptit ja niiden hyväksikäyttö suunnittelussa ja kehityksessä toimivat hyvänä pohjana, mutta erityisen tehokas pelisuunnittelun opetuskeino on pelien luonti. [19]

Peliohjelmointi ensimmäisenä ohjelmointikurssina on mielenkiintoinen idea. Kurssista on mahdollista tehdä yksinkertainen ja alkeita läpikäyvä kurssi ohjelmoinnin ensikertalaisille. Peliohjelmointikurssista voidaan myös tehdä erittäin monimutkainen kurssi, jos sen tarkoituksena on opettaa pelinkehitystä ilman erilaisia pelimoottoreita.

Maailmalla on testattu peliohjelmointikurssien toimivuutta ensimmäisenä ohjelmointikurssina. Washingtonin yliopistossa tietotekniikan pääainetta kohtaan ei ollut tarpeeksi mielenkiintoa. Mielenkiinnon kasvattamista varten suunniteltiin peliohjelmointikurssi, koska useat koulun oppilaista piti pelejä mielenkiintoisina.

Kurssilla käytettiin ohjelmointikielenä C#-kieltä ja pelimoottorina/kehitystyökaluna GameMaker-ohjelmaa. Oppilaita pyydettiin arvioimaan tulivatko he kurssille, koska aiheena olivat pelit arvoasteikolla 0-5 (5 ollen vahva kyllä). Keskiarvoksi vastauksille saatiin 4, eli pelit vaikuttivat myönteisesti kurssille osallistumiseen.

Oppimista testattiin ennakkotestillä ja kurssin lopuksi lopputestillä, jolla saatiin selville kuinka hyvin kurssi oli onnistunut. Testit sisälsivät kysymyksiä lausekkeista, ehtolauseista, silmukoista ja taulukoista sekä funktioista. Esimerkiksi lausekkeiden osaaminen alussa oli 30 % ja lopputestissä 60.7 %. Funktioiden osaaminen nousi 26.7 %:sta 60.7 %:n. Kurssin tekijät olivat tyytyväisiä kurssin saamaan mielenkiintoon ja GameMaker-työkaluun tehokkuuteen sekä opetuksen toimivuuteen ensimmäisenä ohjelmointikurssina. [12]

4.5. Muita peliohjelmointikursseja

Maailmalla peliohjelmoinnin alkeita opettavia kursseja on useita. Peliohjelmointikursseja on tarjolla eri koulutusasteilla aina yläastetasoisista yliopistotasoisiin saakka. Internetistä löytyy useita ilmaisia ja maksullisia kursseja, joihin voi osallistua vapaasti. Kursseissa käytettävät ohjelmointikielet ja työkalut vaihtelevat laajasti. Kuitenkin yleisesti kursseille yhteistä ovat opetettavat asiat ja konseptit, jotka katsotaan tärkeiksi peliohjelmoinnissa.

Suomen eri yliopistojen kurssitarjontaan ei pääse käsiksi ilman, että olisi oppilaitoksessa kirjoilla, mutta esimerkiksi Helsingin ja Jyväskylän yliopistot tarjoavat kaikille ilmaiseksi ja vapaasti suoritettavia kursseja. Helsingin yliopiston etäopiskeluna suoritettavan Peliohjelmoinnin MOOC – kurssin (Massive Open Online Course) tarkoituksena on tutustua ohjelmointiin pelien kautta. Kurssi käyttää NetBeans-ohjelmointiympäristöä ja sen TMC-liitännäistä (Test My Code) sekä Java-ohjelmointikieltä [20]. Jyväskylän yliopiston Nuorten peliohjelmointi kurssi on tarkoitettu yläasteikäisille opettamaan ohjelmoinnin alkeita tietokonepelien kautta. Kurssilla käytetään Microsoft Visio-ohjelmointiympäristöä,

Microsoft XNA Game Studio - kirjastoa ja Jyväskylän yliopiston omaa Jypeli-kirjastoa sekä C#-ohjelmointikieltä. [21].

Peliohjelmointikursseja maailmalla ovat esimerkiksi University of Coloradon Coursera opetuspalvelussa Beginning Game Programming with C# - etäopiskelukurssi [22] ja Udacity opetusportaalin HTML5 Game Development -etäopiskelukurssi [23]. Beginning Game Programming with C# -kurssilla käytetään C#-ohjelmointikieltä ja Microsoftin XNA Game Studio -kirjastoa. HTML5 Game Development -kurssi on selvästi vaativampi kurssi kuin muut, mutta erinomainen ja kattava katsaus HTML5-ohjelmointikielen mahdollisuuksiin pelejä tehdessä.

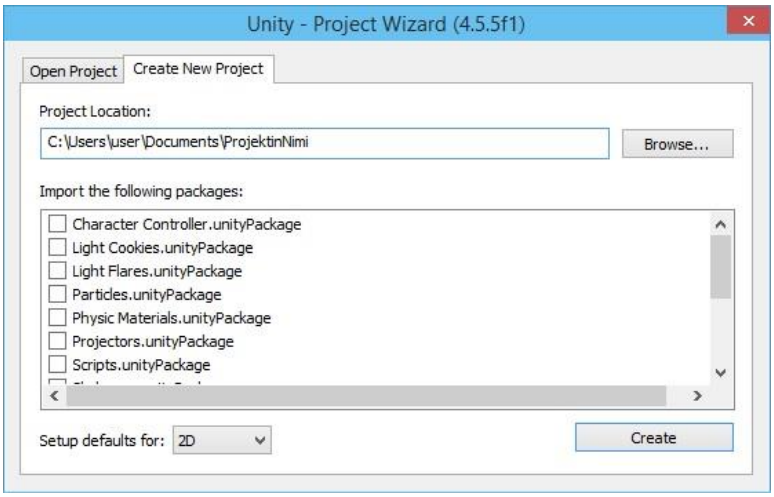
Edellä mainituilla kursseilla käsitellään paljon samankaltaisia teemoja. Jyväskylän yliopiston Nuorten peliohjelmointi kurssilla käydään läpi yksinkertaisimpia asioita Pong-peliä luodessa. Kurssilla liikutaan muuttujista ehtolauseiden ja aliohjelmien kautta olioihin ja luokkiin. Näiden keinojen avulla otetaan myös vastaan pelaajan käskyjä näppäimistön, peliohjaimen, hiiren tai puhelimen avulla. Helsingin yliopiston Peliohjelmoinnin MOOC -kurssilla käydään läpi samoja teemoja, mutta useammassa eri pelissä. Kurssi sisältää myös paljon monimutkaisempiakin asioita, kuten edistyneempää grafiikkaa ja käyttöliittymän luontia ja niiden käyttöä peleissä. Tosin pelaajan käskyjä otetaan kurssilla vastaan vain näppäimistöä hyväksikäyttäen. Beginning Game Programming with C# -kurssilla käydään erittäin tarkasti läpi myös pelien piirtämistä XNA-kirjastoa käyttäen sekä Xbox 360 -ohjaimen hyväksikäyttöä peleissä. Lisäksi kurssilla käsitellään myös äänien käyttöä peleissä. Udacityn HTML5 Game Developement -kurssi on vaativin edellä mainituista. Kurssilla käydään läpi samoja teemoja kuin muissa edellä mainituissa kursseissa, mutta siinä perehdytään lisäksi animointiin ja web-pelien materiaalien käyttöön tarkasti.

5. UNITY3D

Unity 3D ohjelmisto on tehokas pelimoottori, joka sisältää työkaluja kaikkiin pelinkehitys tarpeisiin [24]. Seuraavaksi tutustutaan Unity3D ohjelmistoon ja sen toimintaan. Tarkoituksena on katsastaa näkymiä ja ominaisuuksia, joita tarvitaan 2D-pelien kehittämisessä.

5.1. Asennus

Unity3D ohjelman saa ladattua Unityn internetsivuilta osoitteesta http://unity3d.com/unity/download. Asennuksen jälkeen, ensimmäisellä avauksella ohjelmaan kirjaudutaan Unityn tunnuksilla tai luodaan uudet ilmaiset tunnukset. Ensimmäisellä käynnistyksellä tehdään uusi projekti. Luonnin yhteydessä tulee valita 2D-moodi käyttöön. Tämä ottaa editoriin käyttöön oikeat asetukset, esimerkiksi kamera-moodiksi valitaan ortografinen-kamera ja materiaaleja tuodessa alustavaksi tekstuuri-tyypiksi Sprite. Käyttäjä voi myös valita nämä manuaalisesti projektin asetuksista. Tämän jälkeen Unity3D on valmiina käyttöä varten. Kuva 1 on esimerkki projektinluonnista.



Kuva 1. Unity3D 2D-projektin luonti.

5.2. Käyttöliittymä

Käyttöliittymä koostuu työkaluista (Toolbar), tarkastaja- (Inspector), projekti- (Project), hierarkia- (Hierarchy) ja kenttä- (Scene) sekä peli-näkymistä (Game). Peli-näkymä saadaan esille painamalla kenttä-näkymän Game-painiketta, jolloin kenttä-näkymä vaihtuu peli-näkymäksi.

Käyttöliittymä on täysin muokattavissa haluamakseen. Valmiita käyttöliittymämalleja voi valita Layout-pudotusvalikosta. Kuvassa 2 näkyy Unity3D-editorin perusnäkymä.

Kuva 2. Unity3D Käyttöliittymä. [25]

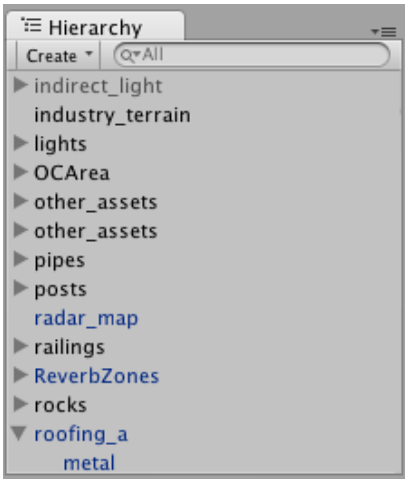
5.3. Projekti-näkymä

Projekti-näkymästä löytyvät kaikki projektin materiaalit (Kuvassa Assets). Materiaaleja voi tuoda erilaisista paketeista, Unity Asset Storesta tai manuaalisesti viemällä Assets kansioon. Esimerkiksi Blender-mallinnusohjelmalla rakennetut mallit ovat Unityn tukemia ja ne voidaan tallentaa suoraan Unity-projektin kansioon. Näkymästä voidaan lisätä halutut materiaalit peliobjekteiksi kenttään (Kenttä-näkymään), suoraan hierarkia paneeliin uusiksi peliobjekteiksi, tai peliobjekteihin komponenteiksi. Projekti-näkymä on esitetty kuvassa 3.

Kuva 3. Unity3D projekti-näkymä. [26]

5.4. Hierarkia-näkymä

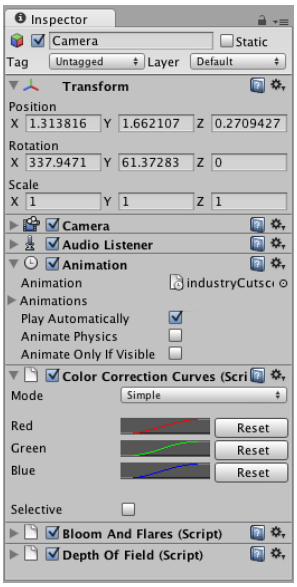
Hierarkia-näkymästä löytyvät kaikki kentässä olevat peliobjektit. Peliobjektille voidaan luoda lapsi-objekteja viemällä nämä haluttuun objektiin. Unity3D käyttää tähän Parenting-konseptia. Tämä tarkoittaa, että uusi lapsi-objekti perii vanhemmaltaan koko-, liikkumis- ja rotaatio-asetukset eli Transform-komponentin asetukset. Kuvassa 4 näkyy Hierarkia-näkymä.



Kuva 4. Hierarkia-näkymä. [27]

5.5. Tarkastaja-näkymä

Tarkastaja-näkymä näyttää pelin grafiikka- ja fysiikka-asetuksia tai valitun peliobjektin tiedot, komponentit ja komponenttien tiedot. Komponentteja voi lisätä objektiin painamalla Add Component - näppäintä, työkaluvalikosta tai tuomalla Projekti-näkymästä. Kuvassa 5 näkyy Tarkastaja-näkymä.



Kuva 5. Tarkastaja-näkymä. [28]

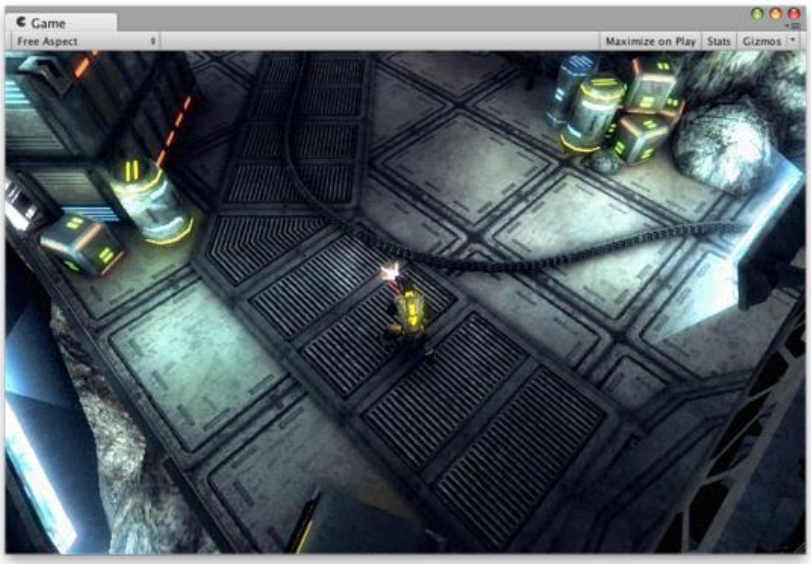
5.6. Kenttä-näkymä

Kenttä-näkymässä rakennetaan tuoduista materiaaleista kenttä ja hahmot. Näitä peliobjekteja voi liikuttaa ja sijoitella haluttuihin paikkoihin x-, y- ja z-akselissa. Näkymän ylälaidassa on useita erilaisia valikoita, joista voidaan hallita 2D/3D-moodia, valo- tai ääniefektien näkyvyyttä sekä muiden efektien näkyvyyttä. Nämä valinnat eivät vaikuta lopullisen pelin ulkonäköön vaan editointi vaiheen tekstuurien näkyvyyteen. Kuvassa 6 on esimerkki Kenttä-näkymästä.

Kuva 6. Kenttä-näkymä. [29]

5.7. Peli-näkymä

Peli-näkymästä näkyy miltä lopullinen pelattava versio näyttää. Play-painikkeella peli-ikkuna aktivoituu ja peliä voi pelata. Pelin voi myös kehitys ja testaus tarkoituksessa keskeyttää. Alla on esimerkki Peli-näkymästä.



Kuva 7. Peli-näkymä. [30]

5.8. Muut näkymät

Muita esiteltäviä näkymiä ovat esimerkiksi Konsoli-, Animaatio- ja Animaattori-näkymät. Konsoli-näkymästä nähdään virhe- ja varoitus-ilmoitukset ja muut kehittäjän haluamat tulostukset. Tämä on erittäin hyödyllinen testattaessa skriptien toimivuutta. Animaatio-näkymästä voidaan muokata ja editoida valittuja animaatioita. Animaattori-näkymässä voidaan manipuloida animaattori-kontrollerin (Animator controller) tilakoneita ja sen parametreja sekä siirtymiä animaatioiden välillä.

5.9. UnityScript

UnityScript on Unity:n kehittämä ohjelmointikieli, joka on mallinnettu JavaScriptin mukaan [31]. Molemmat kielet ovat dynaamisia. Tämä tarkoittaa, että ohjelmoinnissa käytettävien muuttujien tyypit voivat muuttua ajonaikaisesti. UnityScriptiä käytettäessä on suositeltavaa käyttää skriptin alussa #pragma strict -merkintää. Tämä pakottaa käyttämään staattista tyypitystä. Staattinen tyypitys tarkoittaa, että muuttujan tyyppi tulee antaa muuttujan esittelyn yhteydessä. Tämän etuna on sen nopeus, koska Unity3D:n ei tarvitse ”miettiä”, minkä tyyppinen mikäkin muuttuja on. JavaScriptillä ja UnityScriptillä ei ole juurikaan eroja syntaksillisesti.

UnityScriptin heikkous verrattaessa muihin Unity3D:n tukemiin kieliin on sen ominaisuuksien puutteellisuus. Tämä ei vaikuta pelien toimivuuteen, koska puuttuvat ominaisuudet voidaan lisätä peliin esimerkiksi C#-kielellä ohjelmoiduilla skripteillä, jotka toimivat normaalisti muiden skriptien kanssa. Puuttuvista ominaisuuksista esimerkkinä C#-kielessä tuettu delegate-funktio, joka on hyödyllinen luodessa monimutkaisia dynaamisia toimintoja. UnityScriptissä tämä puute tosin voidaan kiertää tekemällä manuaalisesti funktion, joka toimii samalla tavalla kuin delegate. Jos käytetään useata delegate-funktiota samanaikaisesti (ns. multicast), ohjelmoija joutuu tekemään tämän käyttäen C#-skriptiä. [32]

6. PELIOHJELMOINNIN ALKEET KURSSIRUNKO

Oikean kurssin rakentaminen ilman pedagogista koulutusta on hankalaa ja miltei mahdotonta. Tämän työn tarkoituksena ei ole siis vastata kysymykseen *miten.* Tässä työssä tarkastellaan *mitä* materiaaleja ja konsepteja pystyy ja olisi järkevää opettaa 2D-pelinkehityksestä käyttämällä Unity3D-ohjelmaa ja UnityScript-ohjelmointikieltä.

Kurssirunko on suunniteltu esimerkkipelin kehityksen aikana opittujen ja käytettyjen konseptien ja tekniikoiden pohjalta sekä tutkimalla jo olemassa olevien kurssien materiaalia unohtamatta kuitenkaan internetissä tarjolla olevia peliohjelmointikursseja. Runko on suunniteltu kuuden viikon mittaiseksi eli se on jaettu kuuteen viikon mittaiseen osaan. Kurssin osaksi on myös suunniteltu UnityScript-tutoriaali, jossa käydään läpi perusasiat UnityScript-ohjelmointikielestä, minkä on tarkoitus olla vapaaehtoinen. Liitteessä 1 on ehdotettu kurssirunko, jota käsitellään seuraavaksi tarkemmin.

6.1.1. UnityScript tutoriaali

Koska kurssi on suunniteltu aloitteleville ohjelmoijille, on syytä aloittaa kurssin materiaalien läpikäynti UnityScriptin alkeista. Vaihtoehtoisena ratkaisuna voisi tarjota internetistä löytyviä ilmaisia kursseja, näistä esimerkkinä toimii codecademy.com:n JavaScript-kurssi.

UnityScript tutoriaalissa tulee käydä läpi perusteita asioista, joita tullaan tarvitsemaan kurssirungon muissa osissa. Tämä ei tarkoita, että kaikki asiat tulee oppia hetkessä, mutta asioihin tutustuminen on tärkeää etenkin ohjelmoinnin ensikertalaisille. Seuraavaksi käydään läpi aiheita, joihin opiskelijan tulisi tutustua ennen itse kurssia.

UnityScript voidaan aloittaa perussyntaksista ja muuttujista. Skriptissä käytetään dataa useassa eri muodossa. Data tallennetaan muuttujiin, joita hyödynnytetään ja/tai manipuloidaan skripteissä. Vaikka voidaan käyttää dynaamisia muuttujia, kuten aikaisemmin mainittiin, on hyvä ohjelmointitapa käyttää **#pragma strict** -käskyä ja staattisia muuttujatyyppejä. Erilaisiin tietorakenteisiin voi tallentaa useamman muuttujan. Esimerkiksi listaan voidaan tallentaa useamman muuttujan kerrallaan ja voi myös sisältää erityyppisiä muuttujia.

Kommentointi on tärkeä osa ohjelmointia. Sen avulla voidaan dokumentoida tai tarkentaa haluttuja osia koodista. Kommentoimalla voidaan tehdä koodista muille lukijoille ymmärrettävää. UnityScriptissä kommentointi tapahtuu yhdellä rivillä aloittamalla rivi // merkeillä. Useamman rivin kommentointi aloitetaan **/*** merkinnällä ja lopetetaan ***/** merkinnällä.

Numeroiden käyttö ja aritmeettiset operaatiot ovat tärkeä osa UnityScriptin käyttöä. Nämä matemaattiset operaatiot toimivat normaalisti. Usein tarvitaan muuttujien välisiä vertailuja, kun halutaan vertailla näitä keskenään tai tarkistaa onko jokin asia totta. Vertailut palauttavat totuusarvon, sen mukaan onko väite tosi vai epätosi. Loogisilla operaattoreilla voidaan soveltaa totuusarvoja ja vertailla näitä toisiinsa. UnityScript tukee kolmea erilaista loogista operaatiota. Nämä ovat: ja, tai, ei.

Valintarakenteilla ohjataan koodin kulkua. Nämä tunnistavat koodin muuttuvia tekijöitä ja annettuja ehtoja, joiden mukaan rakenne valitsee oikean etenemistavan. Näitä ovat ehtolauseet (if-else-rakenne) ja valintalauseet (switch-lause).

Toistorakenteiden tarkoitus on suorittaa haluttua koodia niin kauan kuin tietty toistoehto on voimassa. While-lausetta suoritetaan niin kauan kuin sille annettu ehto täyttyy. Do-while silmukkaa suoritetaan ainakin kerran, ennen kuin tarkistetaan ehto, joka toistetaan niin kauan kuin ehto on tosi. For-silmukkaa suoritetaan tietty määrä. Siinä alustetaan muuttuja, joka toimii laskurina, jonka täyttymistä seurataan jokaisella silmukkakierroksella.

Funktiot ovat useasti käytettyjä paloja koodia. Funktioita ovat aliohjelmat, proseduurit, metodit tai esimerkiksi moduulit. Unityssä on useita sisäänrakennettuja funktioita, joita voi käyttää hyväksi omassa koodissa. Funktioita voidaan myös itse kirjoittaa. Myös olio-ohjelmoinnista voidaan mainita tutoriaalissa, vaikka tämä on hyvin monimutkainen aihe. Olio-ohjelmointia käytetään paljon UnityScriptissä ja se on vahvasti liitoksissa jokaiseen ohjelmassa käytettävään skriptiin.

6.1.2. Osa 1

Kurssin ensimmäisessä osassa käydään läpi, mikä Unity3D-ohjelma oikeastaan on ja mitä sillä on mahdollista saada aikaan. Ohjelman läpikäyminen voi olla hyvin nopea ja yleismaailmallinen katsaus. Tässä voitaisiin katsastaa editorin eri osa-alueet hyvin samalla tavalla, kuin tämän työn 5. luvussa. Ohjelman eri osa-alueisiin voidaan syventyä tarkemmin, kun niitä käytetään harjoituksissa tai demottaessa.

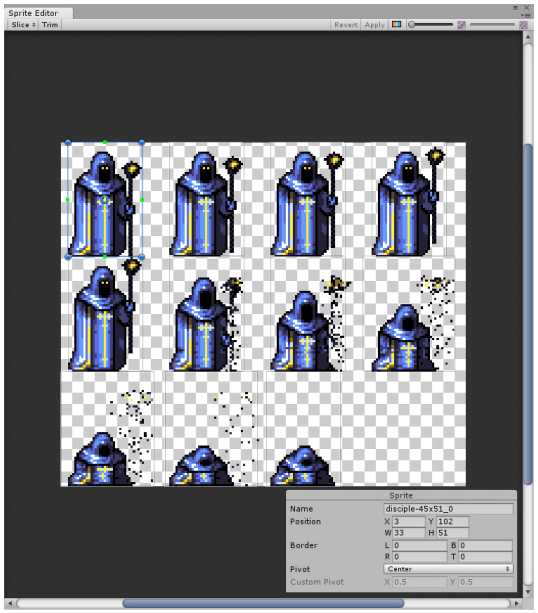
Unity3D:n kameran toiminnasta on hyvä mainita heti alkuun. Kameran tehtävänä on esittää pelaajalle pelimaailma kehittäjän haluamalla tavalla. Uudessa kentässä on Main Camera -objekti valmiina, mutta useampaa kameraa voidaan käyttää, tosin yksi tulee aina olla. Kameramoodeja on kaksi, joita voidaan käyttää 2D-moodissa, nämä ovat perspektiivinen ja ortograafinen kamera. Perspektiivinen kamera toimii, kuin ihmisen silmä, eli suunnat ja syvyydet näkyvät, toisin kuin ortograafisessa kamerassa, joka näyttää näkymän objektit täyskokoisina etäisyyksistä riippumatta.

Ensimmäisiä asioita, joita voidaan opettaa, on peliobjektin lisääminen kenttään. Erilaisia objekteja voidaan lisätä manuaalisesti valitsemalla **GameObject → Create Empty** tai tuomalla projekti-näkymästä uusia materiaaleja, joista Unity3D tekee peliobjekteja. Kenttä-näkymään lisätyt objektit näkyvät myös automaattisesti Hierarkia-näkymässä.

On myös tärkeää selventää mitä objekti sisältää. Kaikilla peliobjekteilla on Transform-komponentti, joka kertoo objektin sijainnin suhteessa kentän nollapisteeseen, tai jos kyseessä on lapsiobjekti, niin suhteessa objektin vanhempaan. Objekteille voidaan myös lisätä komponentteja skripteistä, animaattoreista aina ääniin asti. Komponenttien asetuksia ja arvoja voidaan muokata Tarkastus-näkymässä. Komponentteja lisätään objektiin Tarkastaja-näkymästä painamalla Add Component -näppäintä ja valitsemalla esiin tulevasta valikosta haluttu komponentti.

Valmiita objekteja voidaan tallentaa uudelleenkäyttöä varten Prefabs-kansioon. Tällaisia objekteja kutsutaan Prefab-objekteiksi. Nämä tallentavat myös komponentit ja niiden asetukset. Ne ovat erityisen käytännöllisiä, jos objektia tarvitsee usein. Prefab-objekteja voidaan myös kopioida kenttään skriptien avulla.

Tuotaessa materiaaleja 2D-peliin voidaan käyttää yksinkertaisia kuvia tekstuureina. Näitä kutsutaan nimellä Sprite. Kuvatiedosto voi sisältää useamman kuvan, josta Sprite-editorilla erotellaan erilliset Spritet omiksi materiaaleiksi, joita voidaan lisätä kenttään tai objektin animaatioihin. Tuettuja kuvaformaatteja ovat PSD, TIFF, JPG, TGA, PNG, GIF, BMP, IFF ja PICT. Kuvassa 8 nähdään Sprite-editori toiminnassa.



Kuva 8. Sprite-editori.

Seuraavaksi voidaan selittää miten kentät rakentuvat Unityssa. Kentät rakentuvat peliobjekteista, jotka näkyvät Hierarkia-näkymässä ja Kenttä-näkymässä. Pelin kokonaisuus rakentuu kehittäjän luomista elementeistä ja toiminnollisuuksista. Yleensä yksi kenttä ei kuitenkaan riitä koko pelin toteutukseen. Kenttiä voidaan tallentaa aivan kuten objekteja. Yksinkertaisimmillaan skriptien avulla voidaan luoda useasta kentästä yhtenäinen kokonaisuus eli peli.

Ensimmäisessä osassa skriptaus jää hieman vähemmälle huomiolle, koska ohjelma ja pelinkehitys itsessään vaativat paljon selitystä, mutta yksinkertaisia skriptejä esimerkiksi hahmon liikuttamiseen voidaan toki esitellä. Nämä sisältävät muuttujien arvojen muuttamista ja muuttujiin tallentamista, yksinkertaisia ehtolauseita ja funktioita sekä yksinkertaisia Unity3D:n sisäänrakennettuja funktioita.

Ohjelman perusteiden jälkeen voidaan käydä läpi mistä pelit rakentuvat ja mitä tulisi ottaa huomioon niitä tehdessä. Esimerkiksi kuinka peliyhtiöt rakentuvat ja mitä minkälaista henkilöstöä ne vaativat. Yleensä pelkät ohjelmointialan ammattilaiset eivät riitä vaan tarvitaan graafisia suunnittelijoita, kenttä- ja äänisuunnittelijoita, aina kirjoittajista tuottajiin asti.

Voitaisiin myös käsitellä minkälaisia työkaluja ja ohjelmia Unity3D:n lisäksi olisi hyvä käyttää kehittäessä pelejä. Adoben ohjelmistot ovat mainioita konsepteja suunniteltaessa. Blender Foundationin Blender-ohjelma on erinomainen mallien ja animaatioiden luomiseen. Äänien suunnitteluun ja muokkaukseen voidaan käyttää esimerkiksi Adoben Auditionia tai ilmaista Audacity-ohjelmaa. Mikäli on tarkoitus tehdä pelejä yksin tai pienessä ryhmässä, jossa ei ole mahdollisuutta suunnitella ja luoda omia materiaaleja, voidaan käyttää Unity:n omaa Unity Asset Storea materiaalien hankkimiseen. Voidaan myös käyttää OpenGameArt.org internetsivuston tarjoamia materiaaleja. OpenGameArt.org internetsivuston on oltava tarkkana mitä materiaaleja käyttää ja miten. Materiaalit ovat avoimesti jaossa, mutta niiden lisenssit määrittelevät miten niitä saa käyttää. Tämä tulee ottaa huomioon materiaaleja käytettäessä.

6.1.3. Osa 2

Osassa kaksi keskitytään tarkemmin materiaalien tuontiin ja kenttien rakentaminen. Perehdytään myös animointiin ja esitellään lisää Unityn sisäänrakennettuja funktioita ja monimutkaistetaan hieman skriptausta.

Toinen osa aloitetaan kertaamalla hieman edellisen osan asioita samalla vieden niitä hieman pidemmälle. Kentän suunnittelu on erittäin tärkeässä asemassa, koska siinä tulee ottaa huomioon minkälaisia materiaaleja käyttää kentän rakentamiseen ja millaisen lopputuloksen halutaan saada aikaan. Myös kamera-moodin valinta vaikuttaa kentän rakentaminen. 2D-maailmassa perspektiivistä kameraa käyttämällä ja objektien Transform-komponentin sijainnin z-arvoa muuttamalla saadaan syvyysvaikutelma ja ns. parallax-efekti. Tällä efektillä tarkoitetaan kentän taustojen liikkumista hitaammin kuin etualalla olevat objektit. Pelin taustat viedään kauemmas kamerasta ja hahmot sekä liikkumisalueet tuodaan

lähemmäksi. Jos objektit ovat samassa kohtaa (sama z-arvo), voidaan niiden piirtojärjestystä muokata Sprite Renderer -komponentin Layer-asetuksia vaihtamalla.

Animaatioihin päästään käsiksi pelkillä spriteillä ja parametreilla. Muutamasta spritestä rakennetusta animaatiosta päästään liikkeelle ja käsittelemään animaatioiden perusteita. 2D-peleissä animaatiot ovat yksinkertaisia iteraatioita, jotka käyvät läpi usean spriten sisältävän ketjun. Objektille lisätään Animaattori-komponentti, johon lisätään Animaatio-kontrolleri. Uuden animaation voi luoda Animaatio-näkymästä painamalla Create New Clip -painiketta. Tähän voidaan lisätä haluttuja spritejä, joita halutaan animaatioon. Tämän suoritusnopeutta voidaan muuttaa vaihtamalla Sample-asetuksen arvoa. Animaatiossa on paljon erilaisia muokkausvaihtoehtoja aina objektin värin muutoksista koon muutoksiin saakka. Animaattori-näkymässä voidaan lisätä Animaatio-kontrollerille uusia tiloja. Animaatio lisätään näihin linkittämällä se Tarkastaja-näkymän Motion-valikosta. Tilojen välille voidaan lisätä siirtymiä, jotka linkittävät animaatiot toisiinsa. Siirtymät aktivoidaan parametreilla, joita hallitaan myös Animaattori-näkymässä. Pelinaikaisia parametrien arvoja voidaan muuttaa skripteissä.

Toisessa osassa skriptausta syvennetään hieman. Tarkoituksena on tuoda esille uusia sisäänrakennettuja funktioita ja yksinkertaista komponenttien käsittelyä skriptien kautta. Esimerkiksi Vector-muuttujat voidaan esitellä tarkemmin tässä vaiheessa, koska niitä käyttäen Transform-komponentin arvojen manipulointi on mahdollista. Myös valintarakenteet voidaan sisällyttää skriptaukseen, sillä ne ovat hyvä keino esimerkiksi animaatioiden hallintaan.

Kuitenkin tärkein asia, joka opetetaan tässä vaiheessa skriptauksesta, on ohjelmavirheiden etsintä eli debuggaus ja Print-funktio. Unity3D sisältää konsolin, jota voidaan käyttää debuggaus tarkoitukseen. Print-funktiolla tulostetaan haluttuja tietoja suorituksen aikana konsoliin, mikä tekee siitä erinomaisen työkalun skriptien testaukseen.

6.1.4. Osa 3

Osan kolme tarkoituksena on tutustua tarkemmin aikaisemmissa osissa esitettyihin komponentteihin ja keskittyä etenkin fysiikkamallinnukseen ja sen tuomiin haasteisiin. Myös käyttäjän syöttötietojen käsittelyyn keskitytään tässä osassa.

Erilaisia komponentteja on hyvä käydä tarkemmin läpi, koska toimivan pelin rakentamiseen tarvitaan esimerkiksi niin fysiikka- kuin äänikomponenttejakin. Peliobjektin luonnin yhteydessä objektille lisätään Transform-komponentti ja 2D-peleissä käytettyihin Sprite objekteihin lisätään automaattisesti Sprite Renderer -komponentti. Fysiikkamallinnusta ja törmäyksiä varten vaadittavia komponentteja ovat Rigidbody2D- ja erilaiset Collider2D-komponentit. Animaatioita tehtäessä ja hallittaessa Animaattori-komponentti ja kontrolleri ovat tärkeitä. 2D-pelejä käsiteltäessä fysiikkamallinnus on mainio lisä pelin realismille. Unity3D käyttää NVIDIA PhysX -fysiikkamoottoria 3D-fysiikoiden hallitsemiseen ja Box 2D -fysiikkamoottoria 2D-fysiikkaan [33]. Objektin voi alistaa fysiikkamallinnukselle lisäämällä siihen RigidBody2D-komponentin. Tällä hallitaan mm. objektin painoa ja vetovoimaa.

Objektille on annettava törmäyskomponentti, jotta objektit saadaan kävelemään pintojen päällä ja estetään seinien läpi käveleminen. Näitä eli Collider 2D -komponentteja on useita erilaisia eri käyttötarkoituksiin. Pyöreällä Circle Collider 2D -komponentilla voidaan toteuttaa esimerkiksi hahmon kävely lisäämällä se objektin alaosaan simuloimaan jalkojen sijaintia ja toimintaa. Projektiilien ja seinien osumisen tunnistamiseen voidaan käyttää esimerkiksi Box Collider 2D -komponenttia. Nämä komponentit liitetään haluttuun objektiin ja siirretään Kenttä-näkymässä haluttuun sijaintiin.

Törmäys-komponenteista voidaan tehdä laukaisimia skripteille laittamalla komponentin asetuksista **Is Trigger** -valinta päälle. Tämä tarkoittaa, että kyseinen törmäys-komponentti ei törmää vaan laukaisee/aktivoi halutun tapahtuman samalla päästäen muut objektit läpi.

Kolmannen osan skriptauksessa tärkeään asemaan nousee Vector2- ja Vector3-muuttujat sekä fysiikkamallinnuksen ja törmäyksien mukanaan tuomat OnCollisionEnter2D- ja OnTriggerEnter2D-funktiot sekä niiden Exit-funktiot. OnCollisionEnter2D-funktio

aktivoituu, kun objekti, johon skripti ja törmäys-komponentti on liitetty, törmää toiseen törmäys-komponentin sisältämään objektiin. OnTriggerEnter2D-funktio aktivoituu, jos **Is Trigger** -valinta on valittuna komponentin asetuksissa. Nämä funktiot ovat erittäin monikäyttöisiä pelejä tehdessä. Niitä voidaan käyttää tietyn projektiilin osumien tunnistukseen tai esimerkiksi pelialueella pysymisen tarkastukseen.

Ensimmäisessä osassa esitetty liikkuminen voidaan päivittää käyttämään fysiikkamallinnusta. Tässä voidaan käyttää sisäänrakennettuja rigidbody2D- ja Vector3-funktioita. Esimerkiksi hyppääminen fysiikkamoottoria hyväksikäyttäen on helppoa toteuttaa jo opituilla taidoilla. Käyttäjän syöttötietojen hyväksikäyttöön tulee tutustua tarkemmin skriptauksessa. Esimerkiksi erilaiset Input-funktion käyttövaihtoehdot ovat hyödyllisiä. Funktioiden tekeminen ja hyväksikäyttö on myös mahdollista käydä tässä vaiheessa läpi. Itsetehtyjen funktioiden käyttö törmäysfunktioiden yhteydessä on hyvä keino tutustua tarkemmin funktioiden toimintaan. Funktioihin voidaan myös tutustua osassa 2, mutta animaatioita käsiteltäessä ne eivät ole yhtä tärkeitä kuin käyttäjän syöttötietoja tai törmäyksiä käsiteltäessä.

6.1.5. Osa 4

Osassa neljä kerrataan edellisen osan asioita ja tuodaan uusia työkaluja käyttöön. Näitä ovat esimerkiksi Invoke- ja Instantiate-funktiot sekä Random-luokka. Tarkoituksena on tuoda mukaan lisää pelimäisiä elementtejä ja vaihtoehtoja mielikuvituksen käyttöön pelejä kehittäessä.

Invoke-funktio suorittaa sille annetun metodin ennalta määrätyn ajan kuluttua. Tällä funktiolla voidaan siis suorittaa ajastettuja tehtäviä pelissä. InvokeRepeating-funktio ottaa vastaan yhden parametrin lisää ja se suorittaa annetun metodin tietyn ajan kuluttua tietyin aikavälein. Instantiate-funktiolla kopioidaan objekteja. Tämä on erittäin käytännöllinen funktio esimerkiksi Prefab-objekteja kopioidessa. Lerp-funktiolla voidaan interpoloida esimerkiksi siirtymä kahden pisteen välillä.

Muita tekniikoita, joita voidaan käyttää yksinkertaisia pelejä tehdessä, ovat Raycast2D- ja Linecast2D-funktiot. Nämä ovat fysiikkamoottorin funktioita. Raycast-funktio on kuin

lasersäde, joka voidaan lähettää tarkistamaan onko törmäys-komponentteja annetun matkan päässä tietyssä suunnassa. Linecast2D-funktio lähettää kuvitteellisen viivan, mutta molemmat palauttavat RaycastHit2D-objektin.

Erilaisten metodien ja tekniikoiden lisääntyminen tuo mukanaan useita skriptejä, joita voidaan lisätä yhteen objektiin. Tässä vaiheessa on hyvä opettaa oikeanlainen skriptien suunnittelu ja hallinta laajoissa projekteissa. Skriptejä voidaan suunnitella alusta lähtien uudelleenkäytettäväksi, jottei tarvitse kirjoittaa useata hyvin samanlaista skriptiä turhaan.

6.1.6. Osa 5

Osaan viisi mennessä on läpikäytynä paljon perusteita 2D peleihin liittyen. Projektiin voi tässä vaiheessa tuoda mukaan keinot rakentaa projektista pelimäisempi. Tietenkin peleihin tulee saada erilaisia erikoisefektejä. Ääni- ja valoefektit ovat oivia lisiä opetettaviin asioihin.

Peliin voidaan lisätä ääniä komponenteiksi peliobjekteille. Esimerkiksi taustamusiikin toteutus voidaan tehdä lisäämällä Main Camera -objektille Audio Source -komponentti, joka asetetaan toistamaan komponenttiin lisättyä äänitiedostoa halutulla voimakkuudella uudelleentoistona.

Valoefektien käyttö 2D-peleissä on mahdollista, mutta Sprite-tekstuurit eivät tue näitä suoraan. Kaikkiin objekteihin, joihin valoefektit vaikuttaisivat, tulisi lisätä uusi materiaali, jonka Shader-asetus on Sprites/Diffuse. Tämän jälkeen valoefektien käyttö on mahdollista lisäämällä haluttuja valoefekti-objekteja kenttään.

Pelimäisiä elementtejä, joita voidaan vielä lisätä yksinkertaiseen 2D-peliin, on useita. Päähahmolle voidaan luoda elämien hallinta tai elämäpisteet. Peliin voidaan myös luoda pisteytyssysteemi. Pelilliset elementit ovat erinomaisia keinoja käydä läpi monimutkaiset tietorakenteet ja toistorakenteet UnityScriptissä. Samalla voidaan kerrata edellisissä osissa läpikäytyjä tekniikoita.

Tässä osassa tuodaan mukaan keinot, joilla projektin useat kentät yhdistetään. Tämä voidaan tehdä yksinkertaisin skriptein käyttämällä Application.LoadLevel-funktiota. Tähän lisätään

haluttu kenttä tekstinä. Projektin valmistuessa tulee tietää kuinka pelistä tehdään sellainen, että muut pystyvät pelaamaan sitä omassa puhelimessaan tai tietokoneellaan. Unity3D tukee useita erilaisia järjestelmiä, mutta on tunnettava oikeat asetukset, joilla peli toimii halutussa järjestelmässä.

6.1.7. Osa 6

Ennen osaa kuusi suurin osa tärkeimmistä asioista on jo opetettu, joten tässä osassa kerrataan vaikeimpia asioita sekä käydään läpi harjoitustyötä ja sen viimeistelyä. Uutena asiana tässä osassa opetetaan graafinen käyttöliittymä ja valikkorakenteet.

Termillä Graphical User Interface (GUI) tarkoitetaan pelin graafista käyttöliittymää, jossa toiminnot kuvataan pääasiallisesti graafisesti symbolein tai muun informaation avulla. Peliin voidaan tehdä Canvas-objektilla monimutkaisia valikkorakenteita aina aloitusvalikoista pysäytysvalikkoihin asti. Tälläisistä UI-elementeistä (User Interface) voidaan myös tehdä Head-Up Display (HUD), jossa esitetään pelin aikana pelaajille tärkeätä tietoa, kuten hahmon elämäpisteiden määrä. Canvas-objekteihin voidaan liittää esimerkiksi kuvia ja näppäimiä. Canvas-objekti vaaditaan muita UI-elementtejä tehdessä ja muut UI-elementit asetetaan canvas-objektin lapsiobjekteiksi.

Erinomaisena lisänä kurssilla käytäviin asioihin, etenkin jos harjoitustyön tarkoituksena on tehdä yksinkertainenkin peli, on Game Design Document (GDD). Dokumentti sisältää kuvauksen suunnitellusta pelistä. Tämä tehdään yleensä yhteistyössä suunnittelijoiden, taiteilijoiden ja ohjelmoijien kanssa. GDD on hyödyllinen pienissäkin ryhmissä, jotta suunnittelun ja kehityksen aikana on hyvä kuva halutusta kokonaisuudesta.

7. ESIMERKKIPELI

”Where am I?” on yksinkertainen esimerkkipeli. Pelissä seikkaillaan velholla, joka on revitty omasta maailmastaan tuntemattomiin ulottuvuuksiin. Päähenkilö yrittää selvitä kentistä läpi elossa. Päävastuksena toimii pimeä velho, joka on luonut ulottuvuudet päähenkilön voimien heikentämistä varten, jotta voisi varastaa voimat itselleen omassa ulottuvuudessaan.

7.1. Tarkoitus

Pelin tarkoituksena on toimia todisteena Unity3D:n 2D-pelin kehitysominaisuuksista. Testauksessa keskitytään Unity3D:n editorin 2D-moodiin ja sen tarjoamiin työkaluihin. Tarkoituksena on myös testata opetettavien asioiden reaalimaailmallinen toteutus ja toimivuus.

7.2. Suunnittelu

Peli on suunniteltu aloittelevan pelinkehittäjän lähtökohdista. Kentät ovat lyhyitä, mutta sisältävät paljon erilaisia tekniikoita liikkumisen toteutuksesta aina ääniefekteihin asti.

Jokainen kenttä on suunniteltu niin, että ne sisältävät uusia ominaisuuksia ja tekniikoita edelliseen verrattuna. Pelinkehityksen loppuvaiheessa jokaiseen kenttään on lisätty taustamusiikki ja erilaisia ääniefektejä.

Ensimmäisessä kentässä esitellään yksinkertainen liikkuminen, hyppääminen, fysiikka ja skriptaus. Toisessa kentässä tuodaan mukaan vastustajat, ampuminen projektiileilla ja suojan aktivointi sekä portaalien käyttö. Kolmannessa kentässä esitellään ajan hyväksikäyttö ja satunnaisesti syntyvät objektit. Viimeisessä kentässä hyödynnetään aikaisemmissa kentissä käytettyjä ominaisuuksia ja esitellään uusia monimutkaisempia mekaniikkoja päävastuksen käyttäytymisessä, joita pelaajan tulee seurata.

7.3. Toteutus

7.3.1. Hierarkia

Kentissä käytetyt peliobjektit on sijoitettu hierarkiassa loogisesti omiin peliobjekteihin. Näillä peliobjekteilla ei ole muuta tarkoitusta, kuin pitää sisällään useita samaan kategoriaan kuuluvia objekteja. Esimerkiksi ensimmäisessä kentässä Main Camera -objekti on ainut peliobjekti, joka ei sisällä muita objekteja. Kaikki objektit, joista pelin tausta rakentuu, on sijoitettu Background-objektiin. Pelaajan kanssa samalla syvyydellä olevat kenttäobjektit ovat Foreground-objektissa ja päähahmoon liittyvät objektit on sijoitettu Hero-objektiin.

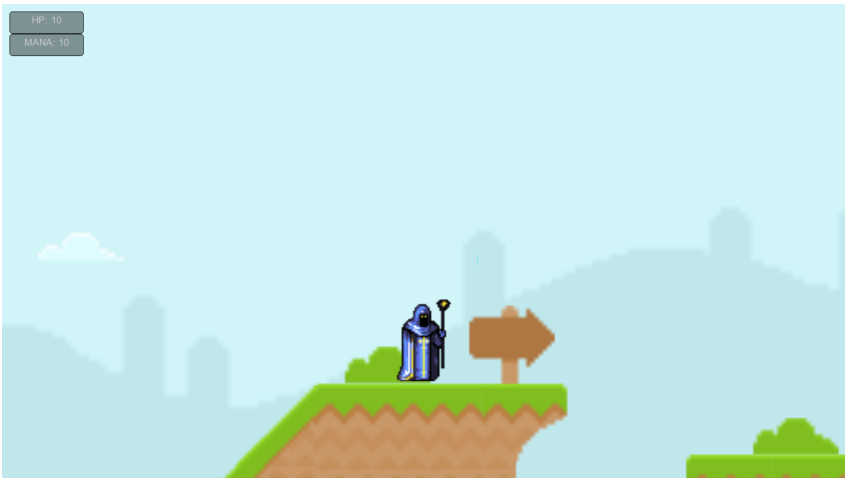
7.3.2. Kamera

Vaikka Unity3D tukee ortografista-kameraa, pelissä käytetään yleisesti 3D kehityksessä käytettävää perspektiivistä-kameraa. Ortografinen-kamera on Unityn kameramoodi 2D-kehitystä varten.

Perspektiivisellä kameralla parallax-efektin luominen onnistui erittäin yksinkertaisesti. Efekti toteutettiin ottamalla 3D-moodi editorista käyttöön ja viemällä taustaobjektit z-akselissa kauemmas pelaajasta. Tämä yhdistettynä perspektiiviseen-kameraan sai aikaan toimivan efektin. Parallaksin tekeminen ortografisella-kameralla on myös mahdollista.

7.3.3. Kenttä

Kenttien ulkonäkö on suunniteltu saatavilla olevien materiaalien mukaan. Unity Asset Storessa ei työn aloittamisen aikaan ollut ilmaisia ja esimerkkipelin tarkoitukseen sopivia materiaaleja. OpenGameArt.org internetpalvelusta löytyi pelin teemaan ja ulkonäköön sopivia materiaaleja. Näitä materiaaleja on tuotu peliin spriteinä. Useasta spritestä on rakennettu kenttiin tasoja ja seiniä sekä erilaisia efektejä. Ovilla on tarkoitus toimia portaaleina eripisteiden välillä sekä siirtymistyökaluina kenttien välillä. Materiaaleihin, joiden päällä hahmojen on tarkoitus liikkua, on lisätty Box Collider 2D – komponentti. Kentät sisältävät useita erilaisia apukeinoja hahmojen ja materiaalien hallintaan. Kuvassa 9 on kuvakaappaus pelin ensimmäisestä kentästä.



Kuva 9. Pelin ensimmäinen kenttä.

7.3.4. Hahmosuunnittelu

Hahmoissa on pyritty valitsemaan samantyylisiä materiaaleja, jotta pelimaailma on uskottava. Hahmot sisältävät erilaisia animaatioita liikkumiseen ja taisteluun sekä esimerkiksi hyppäämiseen. Päähahmo on vanha velho. Vastustajat ovat päävastuksen luomia olioita. Päävastus on ylimielinen pimeävelho. Myös hahmojen materiaaleissa on käytetty hyväksi OpenGameArt-palvelua.

7.3.5. Liikkuminen

Päähahmo pystyy liikkumaan vasemmalle ja oikealle sekä hyppäämään pelaajan käskyjen mukaan. Pelaaja liikuttaa hahmoa joko nuolinäppäimillä tai a-, d- ja w-näppäimillä. Liikkuminen ei ole rajoitettua, mutta hyppääminen tarkistaa onko hahmo tarpeeksi lähellä maata. Jos hahmo on tarpeeksi lähellä maata, w-näppäimen aktivointi rekisteröidään ja hahmolle annetaan voimaa y-akselissa, jolloin hahmo hyppää.

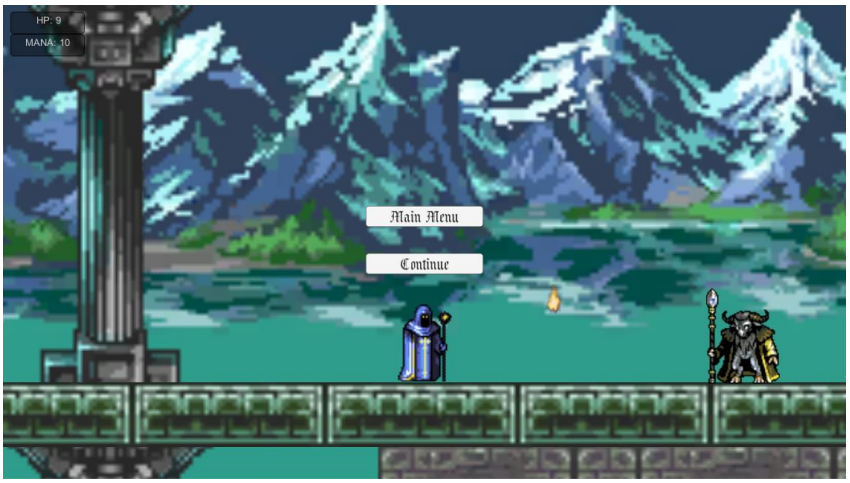
7.3.6. Taistelu

Päähahmolla on kymmenen elämäpistettä sekä kymmenen manapistettä. Pelaaja pystyy ampumaan tulisia projektiileja painamalla Ctrl-näppäintä tai suojautua vastustajilta jääkilvellä, joka aktivoituu painamalla välinäppäintä. Tulen ampuminen on pelaajalle ilmaista. Jääkilven käyttö kuluttaa päähahmon manapisteitä (yksi manapiste kuluu aktivointiin ja tämän jälkeen kuluu yksi piste joka toinen sekunti). Jos pelaaja ei käytä kilpeä, hahmolle regeneroidaan yksi manapiste joka viides sekunti. Päähahmo menettää elämäpisteitä jokaisesta viholliskosketuksesta tai heidän ampumistaan tuliprojektiileista. Normaalit viholliset tuhoutuvat joko tuli-iskusta tai kilven osumasta. Päävihollinen hallitsee tulipalloa, joka seuraa pelaajaa ja yrittää tuhota tämän. Päävihollisella on myös oma kilpensä, jonka se laskee vain kutsuakseen meteorimyrskyn. Päävihollinen on meteorimyrskyn kutsumisen ajan suojaamaton, jonka aikana pelaaja voi tuhota sen. Päävihollisella on kymmenen elämäpistettä. Alla olevassa kuvassa 10 on kuvakaappaus pelin viimeisestä kentästä, jossa on päävihollistaistelu.

Kuva 10. Päävihollistaistelu.

7.3.7. Valikkorakenteet

Peliin on luotu aloitusvalikko, keskeytysvalikko ja pelin päättyessä esitettävä lopetusvalikko. Aloitusvalikosta voidaan aloittaa, painamalla Start Game -näppäintä tai lopettaa peli Exit Game -näppäimellä sekä lukea pelissä käytetyistä materiaaleista painamalla Credits-näppäintä. Pelin aikana pelaaja voi painaa Esc-näppäintä, jolla saadaan esiin pelin keskeytysvalikko. Tämän valikon aktivoiminen keskeyttää pelin. Valikosta voidaan jatkaa peliä tai siirtyä alkuvalikkoon. Pelin lopetusvalikko sisältää lopputekstin ja Continue-näppäimen, josta pääsee takaisin aloitusvalikkoon. Kuvassa 11 on kuvakaappaus kentästä kaksi, jossa on aktivoitu pysäytysvalikko.



Kuva 11. Pysäytysvalikko.

7.4. Skriptiesimerkit

Seuraavaksi tarkastellaan lähemmin peliin tehtyjä skriptejä erilaisten esimerkkien kautta. Näiden tarkoituksena on analysoida ja avata miten tietyt pelimekaniikat ovat luotu ja miten erilaiset ongelmat ovat saatu ratkaistua. Esimerkit käydään läpi seuraavasti: Ensin kerrotaan skriptin toiminnoista tai ongelmien ratkaisuista lyhyesti, jonka jälkeen esitetään skripti-blokki, ja lopulta analysoidaan lyhyesti skriptiä. Esimerkkeinä toimivat skripti-blokit ovat osia isommista kokonaisuuksista, jotka löytyvät liitteestä 2.

7.4.1. Kamera

Pelin kamera on toteutettu perspektiivisellä kameralla, joka mahdollistaa yksinkertaisen parallaksi-efektin luomisen. Tämän lisäksi kamera-objektiin on lisätty skripti, jonka avulla kamera seuraa pelaajan hahmoa. Liitteessä 2 olevasta HeroCameraFollow-skriptistä on alla ote sen Update-funktiosta, jota tarkastellaan seuraavaksi hieman tarkemmin.

Skriptin Update-funktio:

```
var target:Transform;
var distance:float = 5;
var height:float = 1;
var damping:float = 5.0;

function Update () {
		var wantedPosition : Vector3 = target.TransformPoint(0, height, -distance);
		transform.position = Vector3.Lerp(transform.position, wantedPosition, Time.deltaTime * damping);
}
```

Kamera on saatu seuraamaan pelaajan hahmoa käyttämällä Unityn sisäänrakennettua Update-funktiota. Tämä yksinkertaisesti tarkoittaa sitä, että sen sisällä olevat koodirivit suoritetaan jokaisella päivityksellä. Funktiossa kerrotaan kameralle mihin pisteeseen sen tulisi liikkua kyseisellä päivityksellä. Tarkoituksena on liikkua kohti pelaajaa. Pelaajan hahmon sijainti kentän x-akselilla tallennetaan wantedPosition-muuttujaan TransformPoint-funktiolla. Samassa funktiossa annetaan muuttujalle manuaalisesti korkeus (height) ja etäisyyys (distance). Seuraavaksi päivitetään objektin sijaintia muuttamalla sen Transform-komponentin arvoja. Vector3.Lerp-funktiolla saadaan toteutettua pehmeä siirtyminen kahden vektorin/pisteen välillä. Funktion suorituksen nopeutta voidaan säätää muuttamalla sen viimeistä parametria. Nyt skripti kertoo viimeisen päivityksen ajan (Time.deltaTime) damping-muuttujalla, joka sisältää arvon 5. Näillä arvoilla saadaan kamera liikkumaan tasaisesti pelaajan mukana.

7.4.2. Kenttä

Peli sisältää kolme kenttää ja päävastustaistelun. Kentät sisältävät erilaisia skriptejä, joilla luodaan interaktiivisuutta ja monipuolisempaa pelattavuutta. Hyvänä esimerkkinä näistä toimii kolmas kenttä, jossa luodaan meteoriitteja puolen sekunnin välein satunnaiseen paikkaa pelaajan yläpuolelle. Seuraavaksi tutustutaan otteeseen MeteorSpawn-skriptistä.

Ote MeteorSpawn-skriptistä:

```
var meteor:Rigidbody2D;
var speed:float = -3.0;
var target:Transform;

InvokeRepeating("SpawnMeteor", .5, .5);

function SpawnMeteor()
{
		var meteorInstance:Rigidbody2D;
		meteorInstance = Instantiate(
				meteor,
				Vector3(Random.Range(target.position.x-8, target.position.x+8), 7, 0),
				Quaternion.identity)
				);
		meteorInstance.name = "meteorInstance(clone)";
		meteorInstance.velocity = new Vector2(0, speed);
}
```

Skriptissä kutsutaan InvokeRepeating-funktiolla SpawnMeteor-funktiota ensimmäisen kerran 0.5 sekunnin kuluttua skriptin aktivoinnista ja tämän jälkeen 0.5 sekunnin välein. SpawnMeteor-funktio lisää uuden meteor-objektin käyttämällä Instantiate-funktiota. Tämä kopioi vanhan objektin haluttuun sijaintiin halutussa rotaatiossa. Kopioitava objekti annetaan funktiolle meteor-muuttujassa. Uudelle objektille annetaan haluttu sijainti Vector3-muuttujassa ja rotaatio Quaternion-muuttujassa. Muuttujan y-arvoksi asetetaan 7 ja z-arvoksi 0. Muuttujan x-arvo asetetaan Random.Range-funktiolla, joka arpoo satunnaisesti arvon sille annettujen minimi- ja maksimiarvojen mukaan. Target.position.x-muuttuja sisältää päähahmon tämänhetkisen sijainnin x-arvon, josta lasketaan minimiarvo poistamalla 8 ja maksimiarvo lisäämällä 8. Quaternion-muuttujalle ei anneta uutta rotaatiota. Quaternion.identity tarkoittaa yksinkertaisesti ”ei rotaatiota” [34]. Lopuksi uuden kopion nimeksi asetetaan ”meteorInstance(clone)” ja sille annetaan voimaa alaspäin speed-muuttujan verran velocity-funktiolla.

7.4.3. Hahmo

Päähahmoon eli Player-objektiin on liitetty useita skripti-komponentteja. Nämä suorittavat erilaisia tehtäviä pelaajan liikkumisesta aina pelaajan elämä- ja manapisteiden hallintaan. Player-objektin elämäpisteiden hallinnasta vastaavassa Hero-skriptissä OnCollisionEnter2D-funktio aktivoituu, kun pelaaja törmää johonkin objektiin, joka myös sisältää törmäys-komponentin. Skriptissä tarkkaillaan useata pelaajaa vahingoittavaa objektia. Seuraavassa esimerkissä esitetään vastustajan tulipallon osuman hallinta.

Hero-skriptin OnCollisionEnter2D-funktion esimerkki:

```
#pragma strict
var health:int = 10;
var mana:int = 10;
function OnCollisionEnter2D(other:Collision2D)
{
		if(other.gameObject.name == "EnemyFireball(clone)"){
				health--;
				animator.SetInteger("health", health);
				playSoundHit();
				Destroy(other.gameObject);
		}
		if(health <= 0){
				Die();
				yield WaitForSeconds(0.50);
				Spawn();
		}
}
```

Funktio OnCollisionEnter2D aktivoituu, kun päähahmon törmäys-komponentti osuu toiseen törmäys-komponenttiin. Pelaajan elämäpisteisiin vaikuttavien objektien tarkkailu toimii ehtolauseilla. Mikäli EnemyFireball(clone)-objekti osuu pelaajaan, niin päähahmo menettää yhden elämäpisteen (health--). Pisteiden määrä päivitetään myös animaattorille SetInteger-funktiolla. Osumasta soitetaan pelaajalle osumaääni playSoundHit-funktiolla. Lopuksi päähahmoon osunut objekti tuhotaan Destroy-funktiolla. Skripti myös tarkistaa päähahmon elämäpisteiden määrän. Jos pisteet ovat nollassa, niin päähahmo kuolee ja suoritetaan kuolema-animaatio sekä hahmo siirretään takaisin kentän aloituspisteeseen. Die-funktio hoitaa kuolema-animaation, jonka suoritus odotetaan yield-funktiolla ennen kuin Spawn-funktio siirtää hahmon.

7.4.4. Liikkuminen

Pelaajan liikkuminen on mahdollista toteuttaa usealla tavalla. Kaikkia tapoja kuitenkin yhdistää pelaajan antama suunta, joka otetaan vastaan esimerkiksi ehtolauseella. Tässä pelaajan antamat käskyt muutetaan voimiksi, joilla liikutetaan peliobjektia. Liikkuminen tapahtuu lisäämällä voimia rigidbody2D-fysiikkakomponentille. Itse liikkumisen ja matemaattiset laskelmat hoitaa Unityn fysiikkamoottori. Seuraavassa esimerkissä liikutetaan hahmoa x-akselissa.

Movement-skriptin MoveCharacter-funktio:

```
var speed:float = 2.0;
var moveRight:boolean = true;
function MoveCharacter()
{
		var move:float = Input.GetAxis("Horizontal");
		rigidbody2D.velocity = new Vector2(move*speed, rigidbody2D.velocity.y);
		animator.SetFloat("speed", Mathf.Abs(move));
		if (move > 0 && !moveRight)
		{
			Flip();
		}
		else if(move < 0 && moveRight)
		{
			Flip();
		}
}
```

Move-muuttujaan tallennetaan pelaajan haluama liikkuminen x-akselissa. Input.GetAxis-funktio ottaa vastaan pelaajan a- ja d-näppäinten painallukset. Nämä kerrotaan speed-kertoimella, jolloin saadaan hahmolle haluttu suunta ja nopeus. Nopeus annetaan rigidbody2D-komponentille velocity-funktiolla, joka määrittelee siis hahmon nopeuden antamalla sille voimaa vektorin määrittelemien arvojen mukaisesti [35]. Vektorin y-arvona on sen tämänhetkinen lineaarinen y-akselin nopeus. Peliobjektin nopeus informoidaan myös animaattori-komponentille, jotta se voi toistaa oikean animaation riippuen objektin sen hetkisestä nopeudesta. Hahmon suuntaa hallitaan skriptissä olevilla ehtolauseilla. Move-muuttujan arvo määrittelee hahmon suunnan. Flip-funktiolla vaihdetaan tarvittaessa suunta. Seuraavaksi katsastetaan kyseistä funktiota tarkemmin.

Skriptin Flip-funktio:

```
function Flip()
{
		moveRight = !moveRight;
		var scale:Vector3 = transform.localScale;
		scale.x *= -1;
		transform.localScale = scale;
}
```

Flip-funktio on yksinkertainen keino vaihtaa pelaajaan spriten ja animaatioiden suunta. Suunnanvaihto tehdään kertomalla Transform-komponentin skaalaus-vektorin x-arvo luvulla -1. Tällä saadaan vaihdettua skaalauksen etumerkki, joka siis vaikuttaa spriten suuntaan.

7.4.5. Hyppääminen

Hyppääminen on toteutettu ehtolauseella Movement-skriptin MoveCharacter-funktiossa. Se aktivoidaan painamalla w-näppäintä. Ennen kuin hyppääminen on mahdollista, tarkistetaan koskettaako hahmo maata. Tämä testataan GroundedCheck-funktiolla. Funktiokutsu tapahtuu skriptin FixedUpdate-funktiossa, joka suoritetaan kiintein väliajoin. Seuraavaksi tarkastellaan hyppäämisen ehtolausetta sekä GroundCheck-funktiota ja niiden yhteistoimintaa.

MoveCharacter-funktion hyppääminen:

```
var jumpForce:float = 150;

if(Input.GetKey(KeyCode.W) && grounded){
		animator.SetBool("jump", true);
		rigidbody2D.AddForce(new Vector2(0,jumpForce));
}
```

Ehtolause tarkistaa onko päähahmo maassa ja painetaanko w-näppäintä. Mikäli molemmat ehdot täyttyvät animaattori-komponentille ilmoitetaan, että hahmo hyppää, jotta toistetaan oikea animaatio. Tämä tapahtuu asettamalla animaattorin jump-totuusarvo todeksi. AddForce-funktio antaa rigidbody2D-komponentille y-akselille jumpForce-muuttujan verran voimaa, jolla saadaan päähahmo hyppäämään ylöspäin. Ehtolauseessa oleva grounded-totuusarvo muutetaan GroundCheck-funktiossa. Seuraavaksi tarkastellaan tätä funktiota tarkemmin.

Movement-skriptin GroundCheck-funktio:

```
var groundEnd:Transform;
var grounded:boolean;

function GroundedCheck()
{
		Debug.DrawLine(this.transform.position, groundEnd.position, Color.green);
		grounded = Physics2D.Linecast(this.transform.position, groundEnd.position, 1 << 9);
}
```

GroundCheck-funktio piirtää näkymättömän viivan kahden pisteen välille. Tässä tapauksessa päähahmon eli Player-objektin keskipisteen ja groundEnd-objektin välille (groundEnd-objekti on liitetty päähahmoon ja sijoitettu sen alapuolelle maantasolle). Jos viiva kohtaa peliobjektin, jonka tunniste on 9, se palautta RaycastHit2D-objektin, jonka avulla päätellään koskettaako hahmo maata. Mikäli peliobjekti koskettaa maata, grounded-totuusarvoksi asetetaan tosi.

7.4.6. Taistelu

Taistelua varten päähahmolle on annettu kaksi kykyä, joita pelaaja voi käyttää. Pelaaja voi ampua tulipalloja tai suojata itseään jääkilvellä vastustajilta. Tulipallon ampuminen tapahtuu FireballAttack-skriptin Fire-funktiossa. Ampuminen aktivoidaan pelaajan toimesta joko hiiren vasemmalla näppäimellä tai Ctrl-näppäimellä. Tarkistus ampumiselle tehdään skriptin Update-funktiossa ehtolauseella. Seuraavaksi tarkastellaan Fire-funktiota ja kuinka uusi tulipallo luodaan peliin.

FireballAttack-skriptin Fire-funktio:

```
var fingerOfDeath:Transform;
var movementScript:Movement;
var right:boolean;
var fireball:Rigidbody2D;
var fireballSpeedRight:float = 6;
var fireballSpeedLeft:float = -6;
```

```
function Fire(){
		var fireballInstance:Rigidbody2D;
		right = movementScript.moveRight;
		fireballInstance = Instantiate(
				fireball,
				fingerOfDeath.position,
				Quaternion.identity
				);
		fireballInstance.name = "Fireball(clone)";
		if(right){
				fireballInstance.velocity = new Vector2(fireballSpeedRight, 0);
		} else if(!right)
		{
				fireballInstance.velocity = new Vector2(fireballSpeedLeft, 0);
		}
}
```

Funktiossa käytetään hyväksi Fireball-prefabia. Tätä kopioidaan skriptissä uudeksi ilmentymäksi eli instanssiksi. Tälle annetaan Rigidbody2D-komponentti, jotta fysiikkamoottori vaikuttaa siihen oikein. Seuraavaksi funktiossa tarkastetaan päähahmon tämänhetkinen suunta, jotta voidaan ampua tulipallo oikein funktion lopussa olevassa ehtolauseessa. Tämä tallennetaan right-muuttujaan. Ennen ampumista kuitenkin luodaan uusi instanssi Instantiate-funktiolla sekä nimettään se klooniksi. Lopuksi tulipallo ammutaan right-muuttujan mukaiseen suuntaan antamalla sille velocity-funktiolla nopeutta.

7.4.7. Animaatiot

Pelissä objektien animaatioita hallitaan animaattorin parametreilla. Näitä voidaan muuttaa skripteillä tarvittaessa. Esimerkiksi päähahmoon liitetty Hero-skriptin Die-funktiossa ilmoitetaan objektissa olevalle animaattorille, että pelaajan elämäpisteet ovat nollassa, jotta se voi toistaa oikean animaation. Ote Die-funktiosta on alla.

Hero-skriptin Die-funktio:

```
var animator:Animator;
function Die()
{
		animator.SetInteger("health", 0);
		playSoundDead();
}
```

Animaattorin parametreihin päästään käsiksi tallentamalla animaattori animator-muuttujaan. Unityn editorissa tulee lisätä tarkastus-näkymään oikea animaattori. SetInteger-funktio ottaa vastaan tekstinä parametrin nimen, jota halutaan muuttaa animaattorissa sekä sille asetettavan arvon numerona. Tässä tapauksessa animaattorin health-parametrin arvoksi asetetaan 0. Tämä aktivoi animaattorin siirtymän kuolema-animaatioon. Lopuksi funktio toistaa kuolemaäänen kutsumalla playSoundDead-funktiota.

7.4.8. Valikkorakenteet

Pelissä käytetään yksinkertaisia valikkorakenteita, jotka pääasiassa aktivoivat haluttuja kenttiä napinpainalluksella. Näistä parhaana esimerkkinä toimii pysäytysvalikko, jonka pelaaja voi aktivoida painamalla Esc-näppäintä. Seuraavaksi tutustutaan sen toimintaa GameManager-skriptin avulla.

GameManager-skriptin Update-, activate- ja deActivate-funktiot:

```
var pauseMenu:GameObject;
var paused:boolean = false;

function Update () {
		if(
		Application.loadedLevel != 0 &&
		Application.loadedLevel != 5 &&
		Application.loadedLevel != 6
		){
				if(!paused && Input.GetKey(KeyCode.Escape)){
						activate();
				}
		}
}
public function activate(){
		paused = true;
		pauseMenu.SetActive(true);
		Time.timeScale = 0;
}
public function deActivate(){
		pauseMenu.SetActive(false);
		paused =  false;
		Time.timeScale = 1;
}
```

Skriptin Update-funktio tarkistaa ehtolauseilla, että käynnissä oleva kenttä ei sisällä valikkorakenteita ja peliä ei ole aikaisemmin pysäytetty. Mikäli ehdot täyttyvät, on pelaajan mahdollista aktivoida pysäytysvalikko painamalla Esc-näppäintä. Painaminen suorittaa

activate-funktion. Tämä funktio muuttaa paused-totuusarvon todeksi, jotta muita Esc-näppäimen painalluksia ei rekisteröidä ja aktivoi pysäytysvalikon sisältämän Canvas-objektin SetActive-funktiolla sekä pysäyttää pelissä ajankulun muuttamalla Time.timeScale-muuttujan arvon nollaan. Esiin tulevassa valikossa pelaaja voi siirtyä aloitusvalikkoon tai jatkaa peliä. Painamalla Continue-näppäintä pelaaja aktivoi deActivate-funktion, joka poistaa Canvas-objektin käytöstä ja muuttaa paused-totuusarvon epätodeksi sekä jatkaa pelin ajankulkua muuttamalla Time.timeScale-muuttujan arvoon yksi.

8. TULOKSET

Työn tavoitteena oli luoda Peliohjelmoinnin alkeet kurssirunko ja tämä tavoite saavutettiin. Ohjelmoinnin aloittelijoille suunniteltu kurssirunko käy läpi Unity3D-ohjelman perusteet sekä tutustuu skriptauksen alkeisiin. Kurssirunko on rakennettu niin, että siinä käydään 2D-pelinkehitykseen liittyviä konsepteja loogisesti läpi.

Jälkeenpäin ajateltuna kurssin suunnittelua olisi auttanut pedagoginen tuntemus. Tässä työssä oli helppoa selvittää mitä olisi hyvä opettaa ohjelmoinnin aloittelijoille, mutta asioiden laittaminen oikeaan muotoon ja opetusmielessä optimaaliseen järjestykseen oli vaikeaa. Työssä tehty kurssirunko nykyisessä muodossaan toimii enemmänkin ehdotuksena ja pohjana tulevaa kehitystyötä varten.

Työn toisena tavoitteena oli esimerkkipelin kehitys ja Unity3D-ohjelman 2D-pelinkehitystyökalujen testaus. Pelin tekeminen onnistui yli odotusten ottaen huomioon, että aiempaa kokemusta pelinkehityksestä ei ollut. Vaikka peli on erittäin yksinkertainen ja lyhyt, se toimii esimerkkipelinä mainiosti. Voidaan myös todeta, että Unity3D-ohjelman 2D-työkalut ovat toimivia laadukkaiden 2D-pelien tekemiseen.

Jälkeenpäin ajatellen esimerkkipelin suunnittelussa ja toteutuksessa olisi ollut erittäin hyödyllinen GDD:n tekeminen. Skriptien parempi suunnittelu ja yhteistoiminta olisi myös pitänyt suorittaa huolellisemmin. Vaikka peli toimii hyvin, niin kenttäsuunnittelua ja skriptien toimintaa voisi optimoida toimivammaksi.

9. TYÖN JATKOKEHITYS

Ensimmäisenä jatkokehitysideana olisi työn jatkaminen ja kehittäminen Diplomityöksi. Tämä tarkoittaisi sitä, että tuloksena olisi realistinen ja toteuttamiskelpoinen kurssirakenne materiaaleineen. Kurssi olisi myös mahdollista pilotoida käytännössä, jolloin saadaan tutkimusmateriaalia ja tuloksia työtä varten.

Myös tämänhetkisen työn jalostaminen työn aikana opituilla asioilla ja tekniikoilla on mahdollista. Tällä tarkoitetaan tutkimuksellisia, kirjoituksellisia ja pelinkehitykseen liittyviä asioita. Kurssirungon eri osiin voisi kiinnittää enemmän huomioita syventymällä tarkemmin siihen, mitä kokonaisuuksia ja yksittäisempiä tekniikoita tulisi osata 2D-pelinkehityksessä. Nämä asiat voisi viedä pelinkehitykseen kehittämällä laadukkaamman testipedin ja kokonaisuuden. Tähän voisi lisätä myös muiden käyttöjärjestelmien tuen ja optimoinnin.

Kurssin asioiden vieminen monimutkaisempiin tekniikoihin ja kokonaisuuksiin, kuten 3D-pelinkehitykseen, on myös erittäin mielenkiintoinen ajatus. Myös vaihtoehtoiset kehitysympäristöt ovat mahdollisia vaihtoehtoja.

10. YHTEENVETO

Tämä kandidaatintyössä tutkittiin peliohjelmoinnin mahdollisuuksia ohjelmoinnin alkeiden opetuksessa sekä Unity3D-ohjelman toimivuutta 2D-pelinkehityksessä. Tutkimuksen tavoitteena oli luoda Peliohjelmoinnin alkeet -kurssille kurssirunko sekä todistaa Unity3D-ohjelman toimivuus 2D-pelinkehityksessä kehittämällä esimerkkipeli.

Tuotoksena kurssille kehitettiin kuusiosainen kurssirunko, johon myös suunniteltiin täysin aloittelijoille tarkoitettu vapaaehtoinen UnityScriptin alkeet osio. Ohjelmoinnin aloittelijoille suunniteltu kurssi käy läpi Unity3D-ohjelman perusteet sekä tutustuu skriptauksen alkeisiin. Kurssirunko käy 2D-pelinkehitykseen liittyviä konsepteja loogisesti läpi.

Työn toisena tavoitteena oli esimerkkipelin kehitys ja Unity3D-ohjelman 2D-pelinkehitystyökalujen testaus. Pelin tekeminen onnistui yli odotusten. Vaikka peli on erittäin yksinkertainen ja lyhyt, se toimii esimerkkipelinä mainiosti.

Johtopäätöksenä voidaan todeta, että Unity3D-ohjelman 2D-työkalut ovat toimivia laadukkaiden 2D-pelien kehittämiseen. Tuotoksena laadittu kurssirunko antaa suuntaa kurssien rakentamiseen peliohjelmoinnin aloittelijoille. Pelinkehitys kurssin suorittamisen jälkeen opiskelijalla on mahdollisuudet ja perustaidot jatkaa kehittymistä ja opiskelua tahtomallaan suunnalla.

LÄHTEET

1. Unity Technologies, What is Unity?, Noudettu 2.6.2014 https://unity3d.com/pages/what-is-unity
2. Unity Technologies, Unity3d, Noudettu 2.6.2014 http://unity3d.com/
3. Unity Technologies, Scripting, Noudettu 2.6.2014 http://unity3d.com/learn/tutorials/modules/beginner/scripting
4. Unity Technologies, Documentation, Noudettu 2.6.2014 http://unity3d.com/learn/documentation
5. Unity Technologies, Forum, Noudettu 2.6.2014 http://forum.unity3d.com/
6. Unity Technologies, Asset Store, Noudettu 2.6.2014 https://www.assetstore.unity3d.com/en/
7. OpenGameArt.org, 2d Materials, Noudettu 2.6.2014 http://opengameart.org/
8. Association for Computing Machinery, ACM Digital Library, Noudettu 2.6.2014 https://dl.acm.org/
9. Institute of Electrical and Electronics Engineers, IEEE Xplore, Noudettu 2.6.2014 http://ieeexplore.ieee.org/Xplore/home.jsp
10. Caspersen, M., & Bennedsen, J. 2007. Instructional Design of a Programming Course – A Learning Theoretic Approach, pp. 111-122. ACM.
11. Giguette, R. Pre-Games: Games Designed to Introduce CS1 and CS2 Programming Assignments, 2003, pp. 288-292. ACM.
12. Panitz, M, Sung, K & Rosenberg, R. Game Programming in CS0: A Scaffolded Approach, pp. 126-132. The Journal of Computing Sciences in Colleges, 2010
13. Creighton, R, Unity 3.x Game Development by Example, 2nd edition, Packt Publishing, 2011.
14. Pereira, V. Learning Unity 2D Game Development by Example, Packt Publishing, 2014.

15. Lukka, K. The Constructive Research Approach, Metodix, Noudettu 2.6.2014 http://www.metodix.com/en/sisallys/01_menetelmat/02_metodiartikkelit/lukka_cost_research_app/kooste/

16. Unity Technologies, 2d tutorial, Noudettu 2.6.2014 http://unity3d.com/learn/tutorials/modules/beginner/2d

17. Wikipedia, History of video games, Noudettu 12.11.2014 http://en.wikipedia.org/wiki/History_of_video_games

18. Neogames, Tietoa toimialasta, Noudettu 13.11.2014 http://www.neogames.fi/tietoa-toimialasta/

19. Katie, S. & Zimmerman, E. Rules of Play: Game Design Fundementals, chapter 2, page 1. MIT Press, 2004

20. Helsingin yliopisto, Peliohjelmoinnin MOOC, Noudettu 20.11.2014 http://mooc.cs.helsinki.fi/peliohjelmointi

21. Jyväskylän yliopisto, Nuorten peliohjelmointikurssi, Noudettu 21.11.2014 https://trac.cc.jyu.fi/projects/npo

22. University of Colorado Coursera, Beginning Game Programming with C#, Noudettu 20.11.2014 https://class.coursera.org/gameprogramming-001

23. Udacity, HTML5 Game Development, Noudettu 21.11.2014 https://www.udacity.com/course/viewer#!/c-cs255/

24. Unity Technologies, Unity Overview, Noudettu 15.8.2014 http://docs.unity3d.com/Manual/UnityOverview.html

25. Unity Technologies, Learning the Interface, Noudettu 11.11.2014 http://docs.unity3d.com/Manual/LearningtheInterface.html

26. Unity Technologies, Project Browser, Noudettu 11.11.2014 http://docs.unity3d.com/Manual/ProjectView.html

27. Unity Techologies, Hierarchy, Noudettu 11.11.2014 http://docs.unity3d.com/Manual/Hierarchy.html

28. Unity Techologies, Inspector, Noudettu 11.11.2014
http://docs.unity3d.com/Manual/Inspector.html
29. Unity Techologies, Scene View, Noudettu 11.11.2014
http://docs.unity3d.com/Manual/SceneView.html
30. Unity Technologies, Game View, Noudettu 11.11.2014
http://docs.unity3d.com/Manual/GameView.html
31. Unity Technologies, Creating and Using Scripts, Noudettu 24.11.2014
http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html
32. Unity Technologies, Delegates, Noudettu 24.11.2014
http://unity3d.com/learn/tutorials/modules/intermediate/scripting/delegates
33. Unity Technologies, Unity Physics, Noudettu 23.11.2014
http://unity3d.com/unity/quality/physics
34. Unity Technologies, Quaternion.identity, Noudettu 23.12.2014
http://docs.unity3d.com/ScriptReference/Quaternion-identity.html
35. Unity Technologies, Rigidbody2D.velocity, Noudettu 24.11.2014
http://docs.unity3d.com/ScriptReference/Rigidbody2D-velocity.html

Liite 1. Ehdotettu kurssirunko

Kurssirunko on suunniteltu kuuden viikon mittaiseksi eli se on jaettu kuuteen viikon mittaiseen osaan. Runkoon on liitetty myös vapaaehtoinen UnityScript tutoriaali.

UnityScript tutoriaali

- Perussyntaksi
- Muuttujat
- Tietorakenteet
- Kommentointi
- Aritmetiikka
- Valintarakenteet
- Toistorakenteet
- Funktiot
- Olio-ohjelmointi

Osa 1

- Unity3D perusteet
    - Käyttöliittymä
    - Materiaalien tuonti
    - Kamera
    - Objektit
- Komponenttien perusteet
- Prefab
- Perusteet kentän rakennuksesta
- Pohja skriptaukselle
- Pelinkehityksen perusteet
- Muut työkalut

Osa 2

- Kertaus
    - Kentän suunnittelu tarkemmin
    - Kameran vaikutus
- Animaatiot
- Debug
- Skriptaus
    - Valintarakenteet
    - Print-funktio
    - Vector-muuttujat

(jatkuu)

LIITE 1. (jatkoa)

Osa 3

- Kertaus
    - Komponentit
- Fysiikkamallinus
    - Törmäykset
    - Laukaisimet
- Käyttäjän syöttötiedot tarkemmin
- Skriptaus
    - Input-funktio
    - Vector-muuttujat tarkemmin
    - rigidbody2D-funktiot
    - Enter/exit törmäysfunktiot
    - Funktioiden tekeminen

Osa 4

- Kertaus
    - Edelliset asiat
    - Funktiot
- Uusia työkaluja ja funktioita
- Skriptaus
    - Invoke-funktiot
    - Instantiante-funktio
    - Raycast2D- ja Linecast2D-funkiot
    - Lerp-funktio
    - Random-luokka

Osa 5

- Kertaus
- Äänet
- Valoefektit
- Pelimäiset elementit
- Skriptaus
    - Monimutkaiset tietorakenteet
    - Toistorakenteet
    - Edellisten tekniikoiden kertaus

Osa 6

- GUI/UI/HUD
- GDD
- Edellisten osien kertaus
- Harjoitustyö

LIITE 2. Esimerkkipelissä käytetyt skriptit

1. Boss.js

```
#pragma strict
// Variables
var bossAttack:Rigidbody2D;
var shield:GameObject;
var BFB:GameObject;
var pointOfAttack:Transform;
var attackSpeed:float = -6;
var level:String;
var timer:float = 0;
var waitingTime:int = 3;
var timer2:float = 0;
var waitingTime2:int = 12;
var timer3:float = 0;
var waitingTime3:int = 4;
var casting:boolean = false;
var bossHP:int = 10;
// SoundClip
var dieSound:AudioClip;
// Animator information
var animator:Animator;

function Update () {
		fire();
		castFireStorm();
		shieldToggle();
}

// Collision detection and boss health
function OnCollisionEnter2D(other:Collision2D)
{
		if(other.gameObject.name == "Fireball(clone)")
		{
				Destroy(other.gameObject);
				bossHP--;
				if(bossHP <= 0){

						animator.SetTrigger("IsAlive");
						Destroy(BFB);
						Destroy(shield);
						shield = null;
						BFB = null;
						audio.PlayOneShot(dieSound);
						stopMusic();
						yield WaitForSeconds(8);
						bossDie();
						end();
				}
		}
}

// Boss attack one
function fire(){
		if(BFB){
				timer += Time.deltaTime;
				if(timer > waitingTime){
						var bossAttackIntance:Rigidbody2D;
						bossAttackIntance = Instantiate(bossAttack, pointOfAttack.position,
Quaternion.Euler(new Vector3(-1,0,0)));
						bossAttackIntance.name = "EnemyFireball(clone)";
						bossAttackIntance.velocity = new Vector2(attackSpeed, 0);
						timer = 0;
				}
		}
}
```

(jatkuu)

LIITE 2. (jatkoa)

```
// Boss attack two
function castFireStorm(){
		timer2 += Time.deltaTime;
		if(timer2 > waitingTime2){
				casting = true;
		}
		if (casting){
				timer3 += Time.deltaTime;

				if(timer3 > waitingTime3){
						casting = false;
						timer2 = 0;
						timer3 = 0;
						gameObject.GetComponent(MeteorSpawn).SpawnMeteorStorm();

				}
		}
}

// Boss shield management
function shieldToggle(){
		if(shield){
				if(casting){
						shield.SetActive(false);
				} else if(!casting){
						shield.SetActive(true);
				}
		}
}

// Boss object cleaning
function bossDie(){
		Destroy(gameObject);
}

// End game
function end(){
		Application.LoadLevel(level);
}

function stopMusic(){
		var obj = GameObject.Find("Main Camera");
		obj.gameObject.GetComponent(AudioSource).audio.volume = Mathf.Lerp(0.13, 0, 1);
}
```

2. BossFireball.js

```
#pragma strict
var target:Transform;
var damping:float = 2;

function Update () {
		followTarget();
}

function OnCollisionEnter2D(other:Collision2D)
{
		if(other.gameObject.name == "Fireball(clone)")
		{
				Destroy(other.gameObject);
		}
}
```

(jatkuu)

LIITE 2. (jatkoa)

```
function followTarget(){
		var wantedPosition = new Vector3(this.transform.position.x, target.position.y, 0);
		// Commit position using Lerp (Smooth transition) Lerp(from, to, time);
		transform.position = Vector3.Lerp(transform.position, wantedPosition, Time.deltaTime * damping);


}

function kill(){
		// Wanted position takes in height and distance (negative)
		var wanted = target.position.y;
		var wantedPosition : Vector3 = target.TransformPoint(2.452579, wanted, 0);

		// Commit position using Lerp (Smooth transition) Lerp(from, to, time);
		transform.position = Vector3.Lerp(transform.position, wantedPosition, Time.deltaTime * damping);
}
```

3. Enemy1.js

```
#pragma strict
var hObj:GameObject;
var heroRight:boolean;
var heroLeft:boolean;
var enemySpeed:float;
var animator:Animator;
var moveRight:boolean = false;
var dieSound:AudioClip;

function Start () {
		enemySpeed = 0.5;
		animator = GetComponent(Animator);
}

function Update () {
		enemyMove();
		//visible();
}

function OnCollisionEnter2D(other:Collision2D)
{
		if(other.gameObject.name == "Fireball(clone)")
		{
				audio.PlayOneShot(dieSound);
				Destroy(other.gameObject);
				animator.SetTrigger("dead");
				yield WaitForSeconds(0.67);
				Destroy(gameObject);
		}
}


function visible()
{
		if(!renderer.isVisible){
				Destroy(gameObject);
		}

}
```

(jatkuu)

LIITE 2. (jatkoa)

```
function enemyMove(){
		hObj = GameObject.Find("Player");
		if(transform.position.x < hObj.transform.position.x)
		{
				animator.SetBool("heroLeft", false);
				animator.SetBool("heroRight", true );
				heroLeft = false;
				heroRight = true;
				transform.Translate(Vector3.right * enemySpeed * Time.deltaTime);
				if(!moveRight){
						Flip();
				}
		}
		else
		{
				animator.SetBool("heroLeft", true);
				animator.SetBool("heroRight", false );
				heroLeft = true;
				heroRight = false;
				transform.Translate(Vector3.left * enemySpeed * Time.deltaTime);
				if(moveRight){
						Flip();
				}
		}

}



function Flip()
{
		// Flip boolean
		moveRight = !moveRight;
		// Store current localScale information
		var scale:Vector3 = transform.localScale;
		// Flip x-axis
		scale.x *= -1;
		// Commit changes
		transform.localScale = scale;
}
```

4. Enemy2.js

```
#pragma strict
var enemyAttack:Rigidbody2D;
var pointOfAttack:Transform;
var attackSpeed:float = -6;
var timer:float = 0;
var waitingTime:int = 5;
var animator:Animator;
var dieSound:AudioClip;

function Start () {
		animator = GetComponent(Animator);
}
```

(jatkuu)

LIITE 2. (jatkoa)

```
// Attack
function Update () {
		timer += Time.deltaTime;
		if(timer > waitingTime){
			var enemyAttackIntance:Rigidbody2D;
			enemyAttackIntance = Instantiate(enemyAttack, pointOfAttack.position, Quaternion.Euler(new
			Vector3(-1,0,0)));
			enemyAttackIntance.name = "EnemyFireball(clone)";
			enemyAttackIntance.velocity = new Vector2(attackSpeed, 0);
			timer = 0;
		}
}

// Collision detection for dying
function OnCollisionEnter2D(other:Collision2D)
{
		if(other.gameObject.name == "Fireball(clone)")
		{
			audio.PlayOneShot(dieSound);
			Destroy(other.gameObject);
			animator.SetTrigger("dead");
			yield WaitForSeconds(0.54);
			Destroy(gameObject);
		}
}
```

5. EnemySpawner.js

```
#pragma strict
// Spawnpoints for enemies
private var spawnPointEnemy:Vector3;
var spawnPoint:Transform;
// Rigidbodies for enemies
var enemy:Rigidbody2D;
// Enemy information
var enemySpeed:float = 2.0;
var enemySpawnRate:int = 10;
var enemySpawnStart:int = 2;
var enemyName:String;

function Start () {
		// Store coordinates for spawnpoints
		spawnPointEnemy = spawnPoint.transform.position;
}

// Spawn information
InvokeRepeating("SpawnEnemy", enemySpawnStart, enemySpawnRate);

// Spawn functions
function SpawnEnemy()
{
		var enemyInstance:Rigidbody2D;
		enemyInstance = Instantiate(enemy, spawnPointEnemy, Quaternion.Euler(new Vector3(0,0,0)));
		enemyInstance.name = enemyName;
		enemyInstance.velocity = new Vector2(0, enemySpeed);
}
```

(jatkuu)

LIITE 2. (jatkoa)

6. Fireball.js

```
#pragma strict
var timer:float = 0;
var waitingTime:int = 3;

// Destroys unneeded objects after set time
function Update () {
		timer += Time.deltaTime;
		if(timer > waitingTime){
			Destroy(gameObject);
		}
}

function OnCollisionEnter2D(other:Collision2D){
		if(other.gameObject.name == "Fireball(clone)" || other.gameObject.name == "EnemyFireball(clone)" ||
		other.gameObject.name == "meteorInstance(clone)"){
			Destroy(other.gameObject);
			Destroy(gameObject);
		}
}

function OnTriggerEnter2D(other:Collider2D){
		if(other.gameObject.name == "ShieldOn"){
			Destroy(gameObject);
		}
}
```

7. FireballAttack.js

```
#pragma strict
// Fireball information
var fireball:Rigidbody2D;
var fireballSpeedRight:float = 6;
var fireballSpeedLeft:float = -6;
// Can attack timer
var canFireTime:float = 0;
var cooldown:float = 0.5;
// Position where fireballs are fired from
var fingerOfDeath:Transform;
// Check current direction from movement script
var movementScript:Movement;
var right:boolean;



function Update () {
		if(canFireTime <= Time.time){
			if(Input.GetButtonDown("Fire1"))
			{
				canFireTime = Time.time + cooldown;
				Fire();
			}
		}

}
```

(jatkuu)

LIITE 2. (jatkoa)

```
function Fire(){
		// Rigidbody for fireball
		var fireballInstance:Rigidbody2D;
		right = movementScript.moveRight;

		// Instantiate new fireballs from fireball prefab to fingerOfDeath coordinates
		fireballInstance = Instantiate(fireball, fingerOfDeath.position, Quaternion.identity);

		// Change all names of instances to "Fireball(clone)"
		fireballInstance.name = "Fireball(clone)";

		// Direction and force/speed of the fireballs
		if(right){
				fireballInstance.velocity = new Vector2(fireballSpeedRight, 0);
		} else if(!right)
		{
				fireballInstance.velocity = new Vector2(fireballSpeedLeft, 0);
		}
}
```

8. GameManager.js

```
#pragma strict
var pauseMenu:GameObject;
var paused:boolean = false;

function Update () {
		if(Application.loadedLevel != 0 && Application.loadedLevel != 5 && Application.loadedLevel != 6){
				if(!paused && Input.GetKey(KeyCode.Escape)){
						activate();
				}
		}
}

// Startgame by loading level 1
public function startGame(){
		Application.LoadLevel("Level1");
		deActivate();
}

// Credits
public function startCredits(){
		Application.LoadLevel("Credits");
}

// Back to Menu
public function goToMenu(){
		Application.LoadLevel("Menu");
}

// Quit application
public function quit(){
		Application.Quit();
}
public function activate(){
		paused = true;
		pauseMenu.SetActive(true);
		Time.timeScale = 0;
}
public function deActivate(){
		pauseMenu.SetActive(false);
		paused =  false;
		Time.timeScale = 1;
}
```

(jatkuu)

LIITE 2. (jatkoa)

9. Hero.js

```
#pragma strict
// Sounds
var hitSound:AudioClip;
var dieSound:AudioClip;
var jumpSound:AudioClip;
// Hero information
var health:int = 10;
var mana:int = 10;
// Animator information
var animator:Animator;
// SpawnPoint coordinates for hero respawn
private var spawnPoint:Vector3;

function Start () {
		// Store animator component
		// to access parameters within the Animator
		animator = GetComponent(Animator);

		// Set information to animator
		animator.SetInteger("health", 10);
		animator.SetInteger("mana", 10);

		// Get spawnpoint
		spawnPoint = this.transform.position;
}

// Collisions
function OnCollisionEnter2D(other:Collision2D)
{
		// Character is killed by water or by falling out of game
		if(other.gameObject.name == "Water" || other.gameObject.name == "outOfBounds"){
				health = 0;
		}
		// Character loses 1 hp if he is hit by enemy
		if(other.gameObject.name == "Enemy1"){
				health--;
				animator.SetInteger("health", health);
				playSoundHit();
		}
		// Character loses 2 hp if he is hit by enemy
		if(other.gameObject.name == "Enemy2"){
				health = health - 2;
				animator.SetInteger("health", health);
				playSoundHit();
		}
		// Character loses 1 hp if he is hit by enemy fireball
		if(other.gameObject.name == "EnemyFireball(clone)"){
				health--;
				animator.SetInteger("health", health);
				playSoundHit();
				Destroy(other.gameObject);
		}
		// Character loses 2 hp if he is hit by meteor
		if(other.gameObject.name == "meteorInstance(clone)"){
				health = health - 2;
				animator.SetInteger("health", health);
				playSoundHit();
		}
```

(jatkuu)

LIITE 2. (jatkoa)

```
                // Character loses 5 hp if he is hit by Walls of Flame
                if(other.gameObject.name == "FlameLeft" || other.gameObject.name == "FlameRight" || other.gameObject.name ==
                "bossFireball"){
                        health = health - 5;
                        animator.SetInteger("health", health);
                        playSoundHit();
                }
                // Character super jumps when he touches jumpPlatform
                if(other.gameObject.name == "jumpPlatform"){
                        animator.SetBool("jump", true);
                        rigidbody2D.AddForce(new Vector2(0,600));
                        playSoundJump();
                }
                // If health reaches zero, hero dies and respawns
                if(health <= 0){
                        Die();
                        // yield is used to wait until animation is completed
                        yield WaitForSeconds(0.50);
                        Spawn();
                }
}

function playSoundHit(){
                audio.PlayOneShot(hitSound);
}

function playSoundDead(){
                audio.PlayOneShot(dieSound);
}

function playSoundJump(){
                audio.PlayOneShot(jumpSound);
}

function Die()
{
                animator.SetInteger("health", 0);
                playSoundDead();
}

function Spawn()
{
                health = 10;
                mana = 10;
                animator.SetInteger("health", 10);
                this.transform.position = spawnPoint;
}

function setHealth()
{
                health = 0;
                Die();
                Spawn();
}

// Hero health and mana are shown to player
function OnGUI()
{
                GUI.Box(Rect (10, 10, 100, 30), "HP: "+health);
                GUI.Box(Rect (10, 40, 100, 30), "MANA: "+mana);
}
```

(jatkuu)

LIITE 2. (jatkoa)

10. HeroCameraFollow.js

```
#pragma strict
// Object to follow
var target:Transform;
// Settings for camera
var distance:float = 5;
var height:float = 1;
var damping:float = 5.0;



function Update () {
		// Wanted position takes in height and distance (negative)
		var wantedPosition : Vector3 = target.TransformPoint(0, height, -distance);

		// Commit position using Lerp (Smooth transition) Lerp(from, to, time);
		transform.position = Vector3.Lerp(transform.position, wantedPosition, Time.deltaTime * damping);
}
```

11. Meteor.js

```
#pragma strict
var animator:Animator;

function Start () {
		animator.SetBool("dying", false);
}

function OnCollisionEnter2D(other:Collision2D){
		if(other.gameObject.name == "Ground" || other.gameObject.name == "Player" || other.gameObject.name ==
		"FlameRight" || other.gameObject.name == "FlameLeft" || other.gameObject.name == "Portal3"){
				animator.SetBool("dying", true);
				yield WaitForSeconds(0.30);
				Destroy(gameObject);
		}
}
```

12. MeteorSpawn.js

```
#pragma strict
// Meteor information
var meteor:Rigidbody2D;
var speed:float = -3.0;
// Hero/object coordinates
// used to target semirandom spawns
var target:Transform;

// Spawn information
InvokeRepeating("SpawnMeteor", .5, .5);

// Spawnfunctions
function SpawnMeteor()
{
		var meteorInstance:Rigidbody2D;
		meteorInstance = Instantiate(meteor, Vector3(Random.Range(target.position.x -8, target.position.x + 8), 7, 0),
		Quaternion.identity);
		meteorInstance.name = "meteorInstance(clone)";
		meteorInstance.velocity = new Vector2(0, speed);
}
```

(jatkuu)

LIITE 2. (jatkoa)

```
// Storm is used in bosslevel
function SpawnMeteorStorm()
{
		for(var i = 0; i < 5; i++){
			var meteorInstance:Rigidbody2D;
			meteorInstance = Instantiate(meteor, Vector3(Random.Range(target.position.x -8,
			target.position.x + 8), 7, 0), Quaternion.identity);
			meteorInstance.name = "meteorInstance(clone)";
			meteorInstance.velocity = new Vector2(0, -1.5);
		}
}
```

13. Movement.js

```
#pragma strict
// Hero information
var speed:float = 2.0;
var jumpForce:float = 150;
// Moving direction
var moveRight:boolean = true;
// Groundchecking
var groundEnd:Transform;
var grounded:boolean;
// Animator
var animator:Animator;

function Start () {
		// Store animator component
		// to access parameters within the Animator
		animator = GetComponent(Animator);
}

function FixedUpdate(){
		// Check if current level is Boss level
		// if true -> use movement functions made for bosslevel
		// if false -> use normal movement and groundchecking
		if(Application.loadedLevel == 4){
			MoveCharacterBoss();
		} else {
			GroundedCheck();
			MoveCharacter();
		}
}

// Move character by pressing A (left) or D (right)
function MoveCharacter()
{
		// Get movement info for animator
		var move:float = Input.GetAxis("Horizontal");
		animator.SetBool("jump", grounded);
		// Jump
		if(Input.GetKey(KeyCode.W) && grounded){
			animator.SetBool("jump", true);
			// Move by adding speed to rigidbody
			rigidbody2D.AddForce(new Vector2(0,jumpForce));
		}
		// Jump by adding speed to rigidbody
		rigidbody2D.velocity = new Vector2(move*speed, rigidbody2D.velocity.y);
		// Inform animator of current speed
		animator.SetFloat("speed", Mathf.Abs(move));
```

(jatkuu)

LIITE 2. (jatkoa)

```
                        // Flip animation when facing left/right
                        // (flip world)
                        // when moving to right and hero is not facing right
                        // flip to face right
                        if (move > 0 && !moveRight)
                        {
                                        Flip();
                        }
                        // Flip to face left
                        else if(move < 0 && moveRight)
                        {
                                        Flip();
                        }

}
function MoveCharacterBoss(){
                        // When in bosslevel there is no need for jumping
                        animator.SetBool("jump", false);
                        // Variable to store movement information
                        var move:Vector2;
                        // Get movement info for animator
                        move.x = Input.GetAxis("Horizontal");
                        move.y = Input.GetAxis("Vertical");

                        // Move
                        rigidbody2D.velocity = move * speed;
                        // Inform animator of current speed
                        animator.SetFloat("speed", Mathf.Abs(move.x));
                        // Flip animation when facing left/right
                        // (flip world)
                        // when moving to right and hero is not facing right
                        // flip to face right
                        if (move.x > 0 && !moveRight)
                        {
                                        Flip();
                        }
                        // Flip to face left
                        else if(move.x < 0 && moveRight)
                        {
                                        Flip();
                        }

}

function GroundedCheck(){
                        // Draw a line from character to groundEnd object
                        Debug.DrawLine(this.transform.position, groundEnd.position, Color.green);
                        // Check whether or not character is touching the ground
                        grounded = Physics2D.Linecast(this.transform.position, groundEnd.position, 1 << 9);
}

function Flip(){
                        // Flip boolean
                        moveRight = !moveRight;
                        // Store current localScale information
                        var scale:Vector3 = transform.localScale;
                        // Flip x-axis
                        scale.x *= -1;
                        // Commit changes
                        transform.localScale = scale;
}

// Function to inform other scripts of current direction
function GetMoveRight(){
                        return moveRight;
}
```

(jatkuu)

LIITE 2. (jatkoa)

14. NextLevel.js

```
#pragma strict
var level:String;

// Loads next level
function OnTriggerEnter2D(other:Collider2D){
	if(other.gameObject.name == "Player"){
		Application.LoadLevel(level);
	}
}

function OnCollisionEnter2D(other:Collision2D){
	if(other.gameObject.name == "Player"){
		Application.LoadLevel(level);
	}
}
```

15. OutOfBoundsHandler.js

```
#pragma strict
// Hero script reference
var handler:GameObject;

// Trigger detection
function OnTriggerEnter2D(other:Collider2D){
	// Destroy if other is tagged DestroyAble
	if(other.gameObject.tag == "DestroyAble"){
		Destroy(other.gameObject);
	}
	if(other.gameObject.name == "Player"){
		handler.GetComponent(Hero).setHealth();
	}
}
```

16. Portal.js

```
#pragma strict
var targetPortal:GameObject;
var target:GameObject;

// Sends player to another portal (Teleport)
function OnTriggerEnter2D(other:Collider2D){
	if(other.gameObject.name == "Player")
	{
		target.transform.position = targetPortal.transform.position;
	}
}
```

(jatkuu)

LIITE 2. (jatkoa)

17. SpriteFollow.js

```
#pragma strict
var target:Transform;
var damping:float = 5.0;
var distance:float = 1.0;
var current:Vector3;

function Start () {
		current = this.transform.position;
		//print(current);
}

function Update () {
		// Wanted position takes in height and distance (negative)
		var targetPosition : Vector3 = target.position;
		var wantedPosition = new Vector3(targetPosition.x, current.y, 0);

		// Commit position using Lerp (Smooth transition) Lerp(from, to, time);
		transform.position = Vector3.Lerp(transform.position, wantedPosition, Time.deltaTime * damping);

}
```

18. SurvivalLevel.js

```
#pragma strict
// Variable for starting time etc
private var startTime:float;
var textTime:String;
var timeLeft:float = 30;
var isItTime:boolean = false;
function Start () {
		// Check the time from start of the current level
		startTime = Time.timeSinceLevelLoad;
		timeLeft += Time.time;
}
function OnGUI(){
		// Timeleft shown to player
		// format information
		var guiTime = timeLeft - Time.time - startTime;
		var minutes:int = guiTime/60;
		var seconds:int = guiTime%60;
		var fraction:int = (guiTime*100)%100;
		// Timer shown when timeleft greater than zero
		if(guiTime > 0){
				textTime = String.Format("{0:00}:{1:00}:{2:00}", minutes, seconds, fraction);
				GUI.Box(Rect (500, 10, 150, 30), "Survive: " + textTime);
		}
		// Find the portal text shown to player when timer reaches zero
		// Portal spawns at zero
		if(guiTime < 0){
				GUI.Box(Rect (500, 10, 150, 30), "Go through the portal!");
				if(!isItTime){
						isItTime = true;
						removeRightFlame();
				}
		}
}
function removeRightFlame(){
		if(isItTime){
				Destroy(GameObject.FindWithTag("RemoveToContinue"));
		}
}
```

(jatkuu)

LIITE 2. (jatkoa)

19. UseShield.js

```
#pragma strict
// Shield coordinates and prefab
var shieldObject:GameObject;
var shieldPrefab:GameObject;
// Booleans to check if there is a shield
// if its in use
// and if sound is on
var isCreated:boolean = false;
var inUse:boolean = false;
var soundOn:boolean = false;
// Audioclips
var shieldSoundEffect:AudioClip;
var shieldSound:AudioClip;
// Timer information
var timer:float = 0;
var waitingTime:int = 2;
var regenTimer:float = 0;
var regenTick:int = 5;

function Update () {
		// Shield position
		var position = new
		Vector3(shieldObject.transform.position.x,shieldObject.transform.position.y,shieldObject.transform.position.z );

		// Shield activation and activation checks
		if(Input.GetKey("space") && !isCreated && GetComponent(Hero).mana > 0){
				var shieldPrefabInstance = Instantiate(shieldPrefab, position, Quaternion.identity);
				shieldPrefabInstance.transform.parent = shieldObject.transform;
				isCreated = true;
				audio.PlayOneShot(shieldSoundEffect);
		} else if(!Input.GetKey("space") && isCreated) {
				Destroy(GameObject.Find("ShieldOn(Clone)"));
				inUse = false;
				isCreated = false;
		} else if(GetComponent(Hero).mana == 0){
				Destroy(GameObject.Find("ShieldOn(Clone)"));
				inUse = false;
				isCreated = false;
		}
		manaConsumption();
		manaRegen();
}

// Shield protects hero from enemies when it is on
function OnTriggerEnter2D(other: Collider2D){
		if(other.gameObject.name == "EnemyFireball(clone)" || other.gameObject.name == "Enemy(clone)" ||
		other.gameObject.name == "Enemy2(clone)" || other.gameObject.name == "meteorInstance(clone)"){
				audio.PlayOneShot(shieldSoundEffect);
				Destroy(other.gameObject);
		}
}
```

(jatkuu)

LIITE 2. (jatkoa)

```
// Mana consumption
// 1 Mana to activate
// 1 mana/2sec after that
function manaConsumption(){
		var mana:int = GetComponent(Hero).mana;
		if(isCreated && mana > 0){
			if (!inUse){
				GetComponent(Hero).mana--;
				inUse = true;
			}
			if(inUse){
				timer += Time.deltaTime;
				if(timer > waitingTime){
					GetComponent(Hero).mana--;
					timer = 0;
				}
			}
		}
}

// Mana regenerates when shield is not used (1 mana every 5s)
function manaRegen(){
		if(!isCreated && GetComponent(Hero).mana < 10){
				regenTimer += Time.deltaTime;
				if(regenTimer > regenTick){
					GetComponent(Hero).mana++;
					regenTimer = 0;
				}
		}
}
```