Rami Pasanen

MonoGame- ja Unity-ympäristöjen vertailu pelinkehityksessä

Tietotekniikan kandidaatintutkielma

29. kesäkuuta 2017

Jyväskylän yliopisto

Tietotekniikka

Tekijä: Rami Pasanen

Yhteystiedot: ramipasa@student.jyu.fi

Ohjaaja: Antti-Jussi Lakanen

Työn nimi: MonoGame- ja Unity-ympäristöjen vertailu pelinkehityksessä

Title in English: Comparison of MonoGame and Unity as game development platforms

Työ: Kandidaatintutkielma

Sivumäärä: 40+0

Tiivistelmä: Tutkielmassa verrataan MonoGamea ja Unitya pelinkehitysalustoina C#-kielellä. Tarkoituksena on selvittää MonoGamen ja Unityn eroja ja potentiaalisia vahvuuksia toisiinsa verrattuna, ensisijaisesti 2D-peleissä ja aloittelevan pelinkehittäjän näkökulmasta. Tutkimusmenetelmänä on kirjallisuuskatsaus ja vertailu molemmilla sovelluksilla toteutetun bullet hell -genreen sijoittuvan esimerkkipelin avulla. Sekä MonoGamen että Unityn toimintaan tutustutaan erillisissä luvuissa, ja tutustumisen jälkeen toteutetaan esimerkkipeli molemmilla sovelluksilla. Johtopäätöksenä todetaan Unityn tarjoavan matalamman kynnyksen pelinkehityksen aloittamiseen ja yleensä myös säästävän aikaa MonoGameen verrattuna, mutta MonoGamella voi joissain tapauksissa saada sulavamman kehityskokemuksen ja suorituskykyisemmän lopputuloksen.

Avainsanat: MonoGame, XNA, Unity, Microsoft, videopeli, pelinkehitys, 2D, C#

Abstract: The study compares MonoGame and Unity as game development platforms using C#. The purpose is to figure out differences between MonoGame and Unity and their potential strong points compared to each other, especially in 2D game development from a beginner's viewpoint. The study was performed by researching existing literature and by comparing the creation of an example game belonging to the bullet hell genre with both applications. First the reader is made familiar with both MonoGame and Unity in separate chapters, and then the example game is implemented with both applications. The conclusion of the study is that Unity offers a lower barrier to entry for game development and it usually also saves time compared to MonoGame, but in some projects MonoGame might give a

smoother development experience and better performance in the final game.

Keywords: MonoGame, XNA, Unity, Microsoft, video game, game development, 2D, C#

Kuviot

Taulukot

Sisältö

1 Johdanto

Pelien ohjelmoiminen vaatii aikaa ja ohjelmointitaitoa. Usein pelitalot rakentavat omat pelimoottorinsa tuotteitaan varten, mutta ajan säästämiseksi käytetään usein myös markkinoilla jo olemassa olevia pelimoottoreita tai sovelluksia, jotka helpottavat pelimoottorin luontia. Etenkin pelinkehityksen kallistuessa ja monimutkaistuessa kehittäjät ovat alkaneet hyödyntämään etenevissä määrin kolmannen osapuolen luomia pelimoottoreita ja muita komponentteja (Kanode ja Haddad 2009; Wang ja Nordmark 2015). Koska pelin toteutus riippuu vahvasti käytetystä pelimoottorista ja sen komponenteista, kehittäjät joutuvat jo kehityksen alkuvaiheessa päättämään, mitä komponentteja tai pelimoottoria heidän kannattaa käyttää projektiaan varten.

C#-kielellä tunnettuja pelien tekemistä helpottavia ratkaisuja ovat Unity ja Microsoftin XNA Game Studio (myöhemmin XNA). Unity on valmis pelimoottori ja työkalu pelimoottoria käyttävien pelien luomista varten. XNA vuorostaan on ohjelmointiympäristö, jonka tarkoituksena on helpottaa pelien kehittämistä ensisijaisesti C#-kielellä. Microsoft lopetti XNA:n kehittämisen vuonna 2013 (Corriea 2013), mutta edelleen aktiivinen avoimen lähdekoodin projekti MonoGame toteuttaa XNA:n tarjoamat rajapinnat (MonoGame 2017a).

MonoGame ja Unity soveltuvat kumpikin monenlaisten pelien tekemiseen. Sovelluksista molemmat myös tukevat paljon erilaisia alustoja tietokoneista konsoleihin ja mobiililaitteisiin (MonoGame 2017a; Unity Technologies 2017a). Tutkimuksen tarkoituksena on selvittää, miten pelinkehitysprosessi MonoGamella ja Unitylla eroaa toisistaan Windows-alustalla, ja mitkä ovat näiden sovellusten potentiaaliset vahvuudet toisiinsa verrattuna. Näkökulman rajaamiseksi vertailussa tullaan keskittymään ensisijaisesti 2D-pelien luontiin aloittelevan pelikehittäjän näkökulmasta. Koska pelin tekninen toteutus on vahvasti riippuvainen käytetystä pelimoottorista ja sen komponenteista, tutkimus voi antaa arvokasta tietoa uutta projektia C#-kielellä aloittaville pelinkehittäjille.

Työn tutkimusmenetelmä on kirjallisuuskatsaus. Lisäksi MonoGame- ja Unity-ympäristöjä verrataan bullet hell -peligenreen sijoittuvan kaksiulotteisen esimerkkipelin kautta. Kirjallisuuden avulla pyritään esittelemään sekä MonoGamen että Unityn toimintaa ja selvittämään

näiden sovellusten eroja. Koska MonoGame toteuttaa XNA:n tarjoamat rajapinnat, XNA:ta varten tuotettu kirjallisuus pätee yleensä myös MonoGameen, joten tutkimuksessa hyödynnetään myös XNA:ta koskevaa kirjallisuutta. Asian lähestyminen vain kirjallisuuden kautta voisi jättää erot abstrakteiksi, joten esimerkkisovelluksen kautta MonoGamen ja Unityn eroja pyritään konkretisoimaan.

Valmiilla pelimoottoreilla pyritään ratkaisemaan useita pelinkehityksen teknisiä haasteita. Näitä ovat Kanode ja Haddadin (2009) mukaan sisällön hallinta, erilaisten alustojen tukeminen, ja ajan sekä rahan säästäminen käyttämällä pelimoottoreiden valmista toiminnallisuutta oman koodin kirjoittamisen sijaan. Pelinkehittäjät haluavat käyttämiltään sovelluksilta myös korkeaa suorituskykyä ja mukautuvuutta pelin kehitysprosessin aikana mahdollisesti tehtyihin muutoksiin (Wang ja Nordmark 2015; Kasurinen, Strandén ja Smolander 2013). Näiden haasteiden ratkaiseminen ja odotusten täyttäminen muodostavat kriteerit tutkimuksessa tehdylle MonoGamen ja Unityn vertailulle.

Tutkimuksen toisessa ja kolmannessa luvussa tutustutaan MonoGamen ja Unityn toimintaan. Neljännessä luvussa esitellään tutkimusta varten toteutettu esimerkkisovellus ja vertaillaan MonoGamen ja Unityn keskeisiä eroja esimerkkisovelluksen toteutuksen kautta. Viidennessä luvussa esitetään tutkimuksen yhteenveto ja jatkotutkimusideioita.

2 MonoGame

Tässä luvussa esitellään MonoGame, ja tarkastellaan sen toimintaa sekä historiaa. Lisäksi kuvataan MonoGame-pelin arkkitehtuuria sekä kehitysprosessia.

2.1 Yleistä

MonoGame on avoimen lähdekoodin toteutus Microsoft XNA Frameworkin version 4 ohjelmointirajapinnasta (MonoGame 2017a). Projekti alkoi vuonna 2009 nimellä XNA Touch, ja sen alkuperäisenä tavoitteena oli saada XNA:lla tehdyt pelit toimimaan mobiililaitteilla. Ajan myötä projekti kehittyi kattamaan myös tietokoneet ja konsolit. MonoGamea kehitetään edelleen aktiivisesti, ja se on nykyään osin kehittyneempi kuin XNA. Esimerkiksi MonoGame tukee useampia alustoja kuin XNA, ja lisäksi MonoGame tukee DirectX11-teknologiaa grafiikan piirtämiseen Windowsilla, kun XNA:n viimeinen julkaistu versio käyttää vanhempaa DirectX9-teknologiaa (Microsoft 2011). Microsoftin lopetettua XNA:n kehittämisen vuonna 2013 MonoGame on pitkälti korvannut XNA:n. Myös Microsoft itse on tukenut MonoGamea XNA:n korvikkeena kehittäjille, jotka haluavat julkaista pelinsä uusimmille alustoille (Visual Studio Team 2016; Chris Charla 2016). Vastaavasti kuin XNA:n kanssa, MonoGamella tehdyn pelin julkaiseminen ja myynti on ilmaista.

2.2 XNA

Koska MonoGame toteuttaa XNA Frameworkin rajapinnan, pelinkehitysprosessi MonoGamella ei huomattavasti eroa pelinkehityksestä XNA:lla. Ymmärtääksemme MonoGamea paremmin tutustummekin ensin lyhyesti XNA:han.

Microsoft julkaisi XNA Game Studion vuonna 2006. Se korvasi Microsoftin aiemman Managed DirectX -ohjelmointirajapinnan, joka mahdollisti DirectX:n käyttämisen .NET Frameworkin kautta ja sen myötä pelien kehittämisen C#- ja Visual Basic .NET -kielillä. XNA Game Studio oli ohjelmointiympäristö, joka mahdollisti pelien tekemisen Windows Phone-, Xbox 360- ja Windows-alustoille Visual Studio -kehitysympäristön avulla (MSDN 2011b).

Kuten aiempi Managed DirectX, myös XNA tuki C#-kieltä, ja viimeinen XNA:sta julkaistu versio, 4.0 Refresh, lisäsi virallisen tuen myös Visual Basicille (MSDN 2011a). Sen lisäksi, että XNA mahdollisti pelinkehityksen C#-kielellä, XNA:n tavoitteena oli myös ohjelmoijien ajan säästäminen; XNA:n kanssa pelinkehittäjien ei tarvitsisi kirjoittaa yhtä paljon toisteista koodia eri laitteistojen tukemiseen (Microsoft 2004).

XNA Game Studion tärkein komponentti oli XNA Framework, joka on kokoelma kirjastoja, jotka helpottavat DirectX:n käyttöä ja pelin ohjelmointia yhdessä .NET Frameworkin kanssa. XNA Game Studion muut komponentit liittyvät XNA Frameworkin käytön helpottamiseen Visual Studio -ohjelmointiympäristössä, ja pelin sisällön, kuten grafiikan ja äänen muokkaamiseen lopulliseen peliin sopivaan muotoon. Tutkielmassa ‘XNA’ viittaa yleensä XNA Framework -kirjastoihin, mutta voi viitata myös XNA Game Studioon.

Kuvio 1. XNA Game Studio -ohjelmointiympäristön rakenne (Vankaandreev 2011)

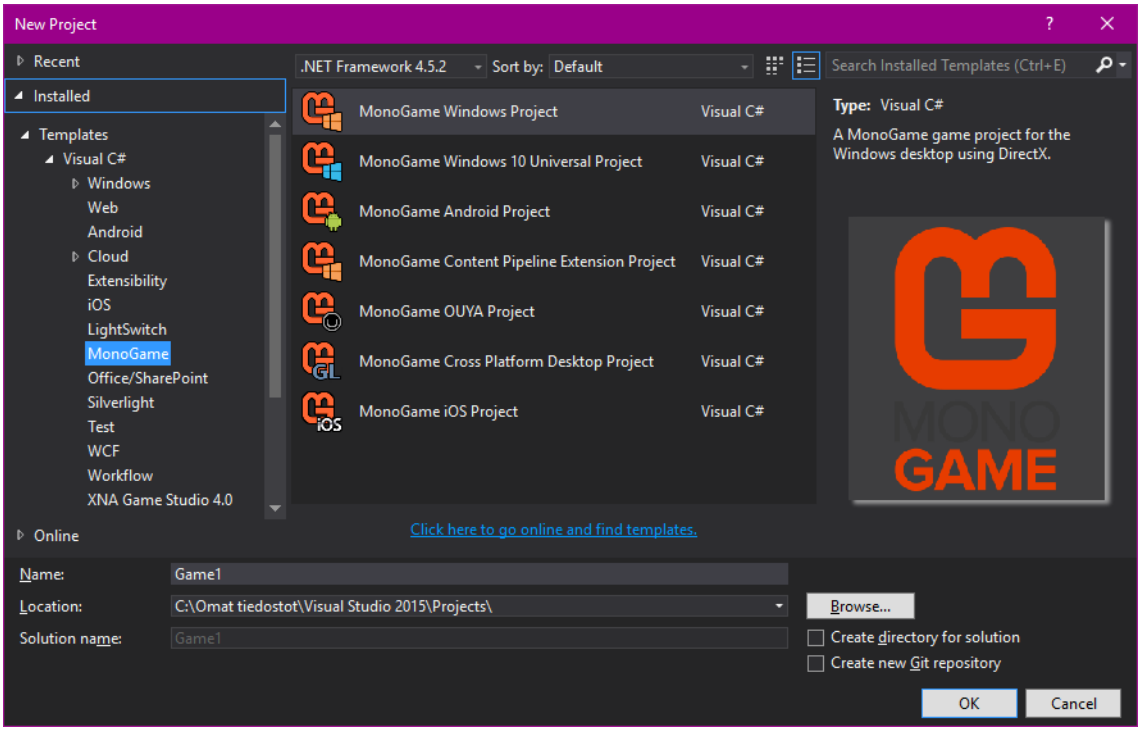
Ensimmäisen julkaisun jälkeen Microsoft päivitti XNA:ta aina vuoteen 2011 asti. Vuonna 2013 Microsoft virallisesti lakkautti XNA:n kehityksen. XNA:n on sittemmin useissa projekteissa korvannut MonoGame, joka toteuttaa XNA Framework -kirjastokokoelman viimeisimmän version, 4.0 Refreshin ohjelmointirajapinnan avoimena lähdekoodina (MonoGame

2017a). Projektin tavoitteena on mahdollistaa vanhojen XNA-kehittäjien siirtyminen nykyaikaisille alustoille, kuten uusimmille pelikonsoleille. Edelleen aktiivisena projektina MonoGame on myös teknisesti kehittyneempi. Kuten XNA, MonoGame myös säästää peliohjelmoijien aikaa; ohjelmoijan ei tarvitse suoraan kirjoittaa koodia grafiikkarajapinnoille kuten DirectX:lle ja OpenGL:lle huomioiden eri laitteistojen ominaisuudet ja niiden tukemisen, vaan MonoGame hoitaa grafiikkarajapintojen käsittelyn ohjelmoijan puolesta. Siten yhdelle alustalle kirjoitettu koodi esimerkiksi grafiikan piirtämistä varten toimii yleensä sellaisenaan kaikilla MonoGamen tukemilla alustoilla. Näin MonoGame vastaa pelinkehityksen teknisiin haasteisiin eri alustojen tukemisen osalta (Kanode ja Haddad 2009).

2.3 Kehitysympäristö

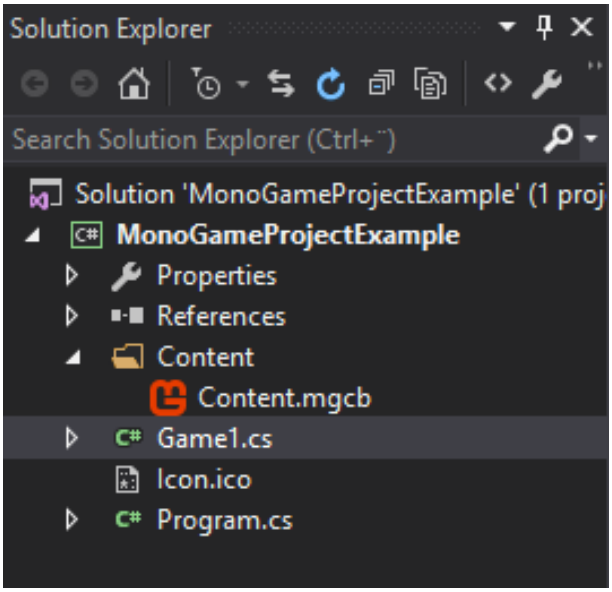
MonoGamella voi kehittää pelejä käyttäen useita eri ohjelmistoympäristöjä, kuten Microsoft Visual Studiota, Xamarin Studiota tai MonoDevelopia (MonoGame 2017c). Tässä tutkielmassa keskitytään aiheen rajaamiseksi MonoGame-pelin kehittämiseen Visual Studiolla. Tutkielmassa ei syvennytä Visual Studion käyttöön, vaan oletetaan, että lukija tuntee sovelluksen perustoiminnot.

MonoGame on ladattavissa Visual Studiolle MonoGamen kotisivuilta. Asentamisen jälkeen uuden MonoGame-pelin luonti onnistuu Visual Studion New Project -valikosta. Projektia luotaessa valitaan, millä alustalla peli oletuksena toimii.



Kuvio 2. Uuden MonoGame-projektin luominen

MonoGame-pelin kehitys tapahtuu Visual Studiolla vastaavasti kuin yleensäkin minkä tahansa C#-kielisen ohjelman kehitys. Projektissa on valmiina ainoastaan pääohjelman sisältävä tiedosto `Program.cs` ja myös `Game1.cs`, joka on luokka itse pelille. Valmiina on myös `Content`-kansio ja `Content.mgcb` -tiedosto, jonka avulla hallitaan peliin liittyvää sisältöä, kuten tekstuureita ja ääniä. Sisällön lisäämiseen ja käyttöön tutustutaan tarkemmin alaluvussa 2.5.



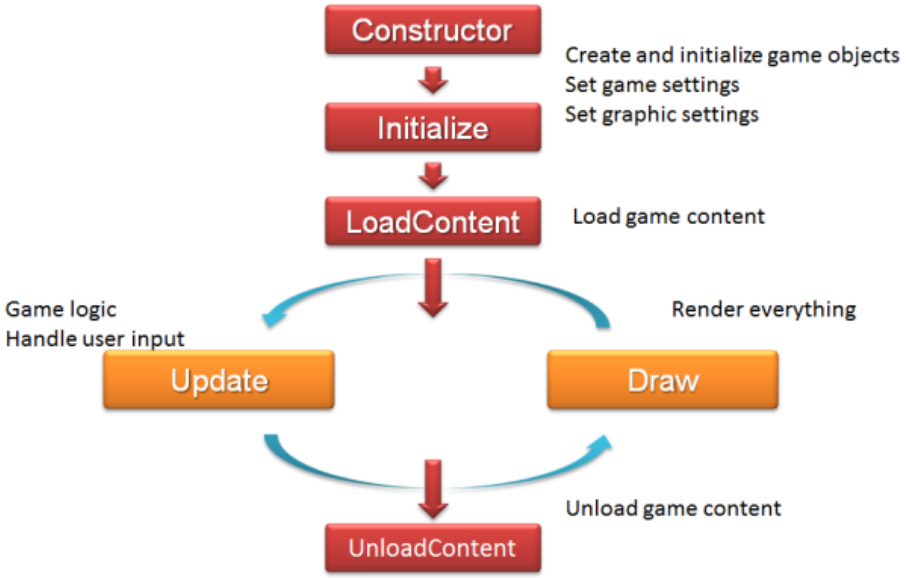
Kuvio 3. Uuden MonoGame-projektin tiedostot

2.4 Pelin arkkitehtuuri ja pelikomponentit

Pelaaja yleensä käsittää pelin tapahtumat samanaikaisina; jos pelissä voi esimerkiksi liikuttaa hahmoa, liikkuminen vaikuttaa tapahtuvan samanaikaisesti muun maailman toiminnan kanssa. Pelit eivät kuitenkaan yleensä käsittele pelaajan syötettä ja simuloi maailman toimintaa yhtäaikaa, vaan pelit tekevät nämä toiminnot peräkkäin niin sanotussa pelisilmukassa (game loop). Tätä silmukkaa suoritetaan pelin alkamisesta lähtien pelin päättymiseen asti, ja silmukan yksi suorituskerta pyritään pitämään niin lyhyenä, että peli toimii interaktiivisesti. Pelisilmukassa tehdyt toiminnot voidaan jakaa kolmeen osaan: käyttäjän syötteen käsittelyyn, pelimaailman päivittämiseen ja pelin piirtämiseen (Valente, Conci ja Feijó 2005).

MonoGamessa pelisilmukan vaiheet on toteutettu XNA:n tapaan pelin pääluokan (`Game`, oletuksena tiedostossa `Game1.cs`) `Update`- ja `Draw`-metodeissa. `Update`-metodi on tarkoitettu pelin logiikan päivittämiseen (esimerkiksi törmäystarkistusten tekemiseen) ja käyttäjän syötteen käsittelyyn, kun `Draw`-metodi vuorostaan on tarkoitettu pelin grafiikan piirtämiseen (Karppinen 2010). Oletuksena MonoGame pyrkii kutsumaan `Update`-metodia 60 kertaa sekunnissa ja `Draw`-metodia samaan tahtiin pelaajan käyttämän näytön virkistystaajuuden kanssa (Grootjans 2009). Molemmat aikavälit ovat säädettävissä.

Pelisilmukan metodien lisäksi pelin pääluokassa on `Initialize`-, `LoadContent`- ja `UnloadContent`-metodit. `Initialize`-metodissa alustetaan peli tilaan, jossa pelin on tarkoitus olla pelin alkaessa. `LoadContent`-metodia käytetään sisällön, kuten tekstuurien ja äänien lataamiseen, ja `UnloadContent`-metodissa vapautetaan tarvittaessa ladatun sisällön käyttämät resurssit (Karppinen 2010).



Kuvio 4. MonoGame / XNA -pelin elinkaari (Vankaandreev 2011)

Pelin kaikkea koodia ei kuitenkaan ole mielekästä kirjoittaa yhteen luokkaan, sillä luokasta tulisi projektin edetessä valtava ja mahdoton ylläpitää. Pelin logiikkaa ja erilaisia peliolioita varten MonoGamessa onkin luokka `GameComponent`, josta perityillä luokilla on `Draw`-metodia lukuunottamatta samat metodit kuin `Game`-luokallakin. Erikseen on myös `DrawableGameComponent`-luokka, jolla on myös `Draw`-metodi ja jonka piirtokoodia tarvitsevat luokat voivat periä. Komponentit voivat ohittaa näiden metodien alkuperäiset toteutukset määrittääkseen oman toiminnallisuutensa. Komponentteja voi sekä lisätä peliin että poistaa pelistä `Game.Components` -kokoelmaa muokkaamalla. Peliin lisättyjen komponenttien `Update` ja `Draw`-metodeja ei tarvitse kutsua manuaalisesti, vaan samoin kuin XNA, MonoGame kutsuu näitä metodeja automaattisesti pelisilmukkaa suorittaessaan (Reed 2010).

2.5 Sisällön lisääminen ja käyttö pelissä

Kun peliin luodaan sisältöä, kehittäjät joutuvat päättämään sisällön tiedostomuodon. Esimerkiksi tekstuurit voi sisällyttää JPEG- tai PNG-kuvina, ja myös äänille on useita eri formaatteja. Jokaisella formaatilla on omat lisenssiehtonsa, ja jokainen tuettu formaatti edellyttää joko kirjastoa tai omaa koodia, joka voi käsitellä kyseistä formaattia. Sisällön eri muotojen tukeminen ja sisällyttäminen peliin onkin yksi pelinkehityksen teknisistä haasteista, joihin pelimoottori joutuu vastaamaan (Kanode ja Haddad 2009).

MonoGamen ratkaisu sisällön muoto-ongelmaan on Content Pipeline -sisältöliukuhihna. Myös XNA:ssa on sisältöliukuhihna, ja alunperin MonoGame olikin riippuvainen XNA:n sisältöliukuhihnasta. Maaliskuussa 2015 julkaistusta 3.3 -versiosta lähtien MonoGame on tosin sisältänyt oman alustariippumattoman toteutuksensa sisältöliukuhihnalle, joten toisin kuin suurin osa XNA:n dokumentaatiosta, XNA:n sisältöliukuhihnan dokumentaatio ei sellaisenaan suoraan päde MonoGameen (MonoGame 2015).

MonoGamen sisältöliukuhihna ottaa erilaisissa tunnetuissa muodoissa olevaa sisältöä ja tuottaa niistä suoritettavaa peliä varten omassa formaatissaan olevia `.xnb`-tiedostoja, jotka MonoGame kykenee lataamaan peliä varten. Sisältöliukuhihna tukee oletuksena useita tiedostomuotoja, mutta tarvittaessa kehittäjän on mahdollista myös laajentaa sisältöliukuhihnaa tukemaan useampiakin formaatteja. Oletuksena MonoGame osaa muuttaa ainakin seuraavat tiedostotyypit `.xnb`-tiedostoiksi:

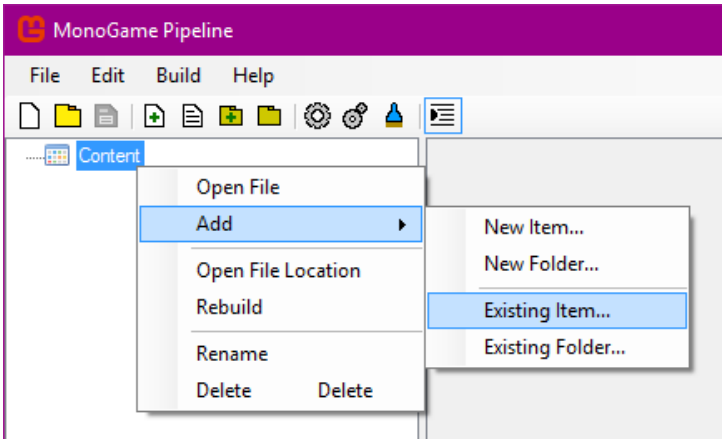
Taulukko 1. MonoGamen sisältöliukuhihnan tukemat tiedostotyypit (RB Whitaker; MonoGame 2017b)

Sisällön tyyppi	**Tuetut tiedostomuodot**
Ääni	.mp3, .wav, .wma., .ogg
Tekstuuri	.bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm, .tga
Fontti	.spritefont
3D-mallit	.x, .fbx, .obj

Visual Studiolla luodussa MonoGame-projektissa on valmiina kansio `Content`, ja tässä kansiossa tiedosto `Content.mgcb`. Sisällön hallinta MonoGame-pelissä tapahtuu avaa-

malla `Content.mgcb`-tiedosto MonoGameen sisältyvällä *MonoGame Pipeline* -sovelluksella. `Content.mgcb`-tiedostoa ei voi avata *MonoGame Pipeline* -sovelluksella suoraan Visual Studiosta, vaan kehittäjä joutuu navigoimaan kansioon resurssienhallinnan kautta ja avaamaan tiedoston manuaalisesti.

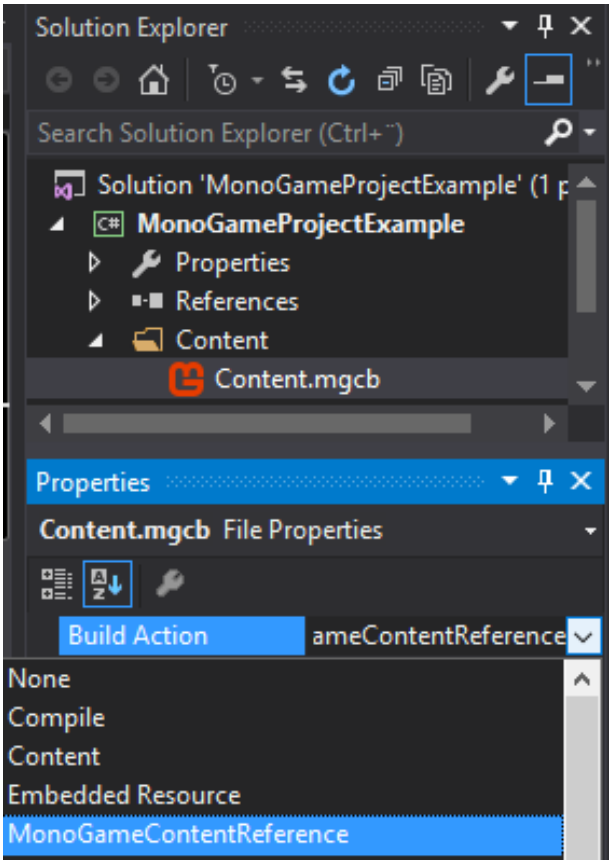
Tiedoston avaamisen jälkeen sisällön lisääminen onnistuu avaamalla kontekstivalikko vasemmalla olevan puunäkymän *Content*-kohdasta ja navigoimalla valintaan *Add -> Existing Item*.



Kuvio 5. Uuden sisältötiedoston lisääminen MonoGame-peliin

Lisäämisen jälkeen sisältö näkyy MonoGame Pipeline -sovelluksen puunäkymässä. Sisältö on mahdollista valita, jolloin sisällön tyypistä riippuen sovellus luettelee sisällölle useita ominaisuuksia, ja mahdollistaa myös näiden ominaisuuksien muokkaamisen. Näitä ominaisuuksia ei tarkastella tässä tutkielmassa tarkemmin.

Kun sisältö on lisätty, sisällön voi halutessaan kääntää manuaalisesti `.xnb`-tiedostoiksi MonoGame Pipeline -työkalun avulla. Nämä `.xnb`-tiedostot olisi sitten mahdollista lisätä Visual Studiossa pelin projektiin. Yksinkertaisempaa on kuitenkin antaa MonoGamen hoitaa kääntäminen ja `.xnb`-tiedostojen siirtäminen peliprojektiin itse. Tämä onnistuu valitsemalla Visual Studion Solution Explorer -valikosta `Content.mgcb`-tiedosto, ja antamalla tälle *Build Action* -ominaisuudeksi *MonoGameContentReference*. Uudessa MonoGame-projektissa tämä valinta on käytössä jo oletuksena.



Kuvio 6. MonoGameContentReference-asetus

Kun pelin kääntää Visual Studiossa `Content.mgcb`-tiedoston *Build Action* -ominaisuuden arvon ollessa *MonoGameContentReference*, MonoGame kääntää automaattisesti kaiken uuden tai muokatun `Content.mgcb`-tiedostoon linkitetyn sisällön ja kopioi `.xnb`-tiedostot käännetyn projektin `Content`-alikansioon (MonoGame 2017d). Siten kehittäjän ei tarvitse tehdä sisällön lisäämiseksi muuta, kuin lisätä tiedostot `Content.mgcb`-tiedostoon *MonoGame Pipeline* -sovelluksen avulla.

Sisällön käyttö itse pelin ohjelmakoodissa tapahtuu vastaavasti kuin XNA:ssa. Sisältö ladataan `ContentManager`-luokan instanssin `Load`-metodilla. Sisällön tyypistä riippuen se esimerkiksi joko piirretään pelikomponentin `Draw`-metodissa `SpriteBatch`-luokan instanssin avulla, tai ääniefektit soitetaan `SoundEffect`-luokan instanssin `Play`-metodin avulla.

3 Unity

Tässä luvussa tutustutaan yleisellä tasolla Unity-pelimoottoriin ja pelimoottoria käyttävän pelin luomiseen. Tavoitteena on antaa lukijalle yleiskuva pelin kehittämisestä Unityllä siten, että myöhempi vertailu MonoGameen on mahdollista.

3.1 Yleistä

Unity on Unity Technologiesin kehittämä, vuonna 2005 julkistettu pelimoottori. Unity tukee pelimoottorina useita eri alustoja, kuten eri käyttöjärjestelmiä ajavia tietokoneita, konsoleita ja älypuhelimia, ja siten auttaa kehittäjää ratkaisemaan yhden pelinkehityksen keskeisistä teknisistä haasteista (Kanode ja Haddad 2009; Unity Technologies 2017e). Alun perin Unity suunniteltiin lähinnä 3D-pelejä varten, mutta myöhemmin siihen on lisätty myös 2D-ominaisuuksia, erityisesti vuoden 2013 lopussa julkaistun 4.3 -version myötä (Unity Technologies 2013). Pelimoottoria ja siihen liittyviä työkaluja kehitetään edelleen aktiivisesti.

Unitya varten on luotu yksityiskohtainen, järjestelty dokumentaatio ja harjoituksia, jotka helpottavat pelimoottorin käyttöönottoa (Unity Technologies 2017b; Rogers 2012). Lisäksi Unityn kotisivuilla on keskustelupalsta ja wiki, jotka sisältävät runsaasti Unityn käyttäjäyhteisön luomaa tietoa pelimoottorin käytöstä (Rogers 2012). Vahvan dokumentaation lisäksi yksi Unityn vahvuuksista on Asset Store, jonne kehittäjät voivat ladata Unitya varten luomaansa sisältöä, kuten tekstuureja, 3d-malleja, animaatioita ja myös koodia eli skriptejä (näihin tutustutaan tarkemmin luvussa 3.4). Sisältö voi olla joko ilmaista tai maksullista. Toiset kehittäjät voivat ostaa ja ladata tätä sisältöä käyttääkseen sitä omissa peleissään, mikä säästää aikaa itse pelin kehittämiseen. Unity veloittaa myydyn sisällön hinnasta 30%, ja loput 70% saa sisällön julkaissut kehittäjä (Juslin 2015).

Unitysta on saatavilla neljä eri versiota: Personal, Plus, Pro ja Enterprise (Unity Technologies 2017f).

- Personal-versio on ilmainen, mutta sitä saa käyttää julkaistussa pelissä vain, jos pelin julkaisseen yrityksen liikevaihto on alle 100 000 dollaria. Lisäksi Personal-versiossa

on muita rajoitteita, kuten Made with Unity (MWU) -ruutu pelin käynnistyessä.

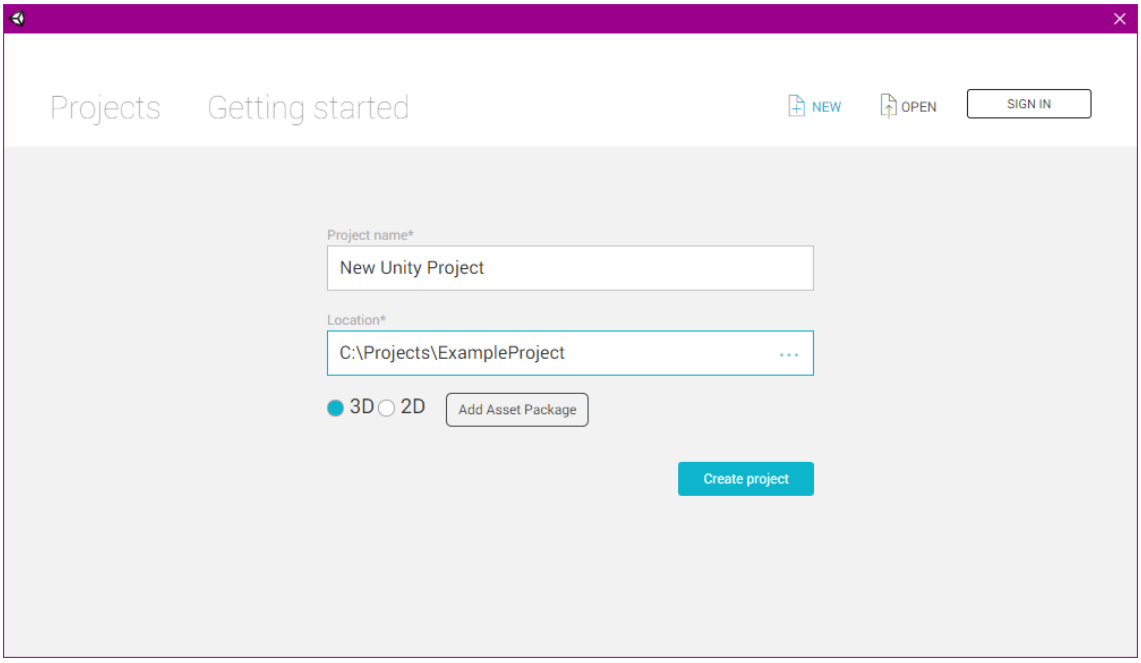
- Plus-versio, joka maksaa 35 dollaria kuukaudessa jokaista kehittäjää kohden. Plus-version liikevaihtoraja on 200 000 dollaria. Plus-versiossa ei ole pakollista MWU-ruutua, ja lisäksi se tarjoaa Personal-versiota enemmän ominaisuuksia, kuten työkaluja pelin suorituskyvyn tarkkailuun.
- Pro-versio maksaa 125 dollaria kuukaudessa kehittäjää kohden, mutta poistaa liikevaihtorajan kokonaan.
- Enterprise-versio on tarkoitettu suurille yrityksille, eikä sillä ole ennalta määrättyä hintaa. Enterprise-versio tarjoaa muun muassa Unityn ylläpitämät pelipalvelimet tuhansien samanaikaisten pelaajien moninpeliä varten, ja kehittyneet analytiikat pelaajien käyttäytymisen seuraamiseen.

Tätä tutkimusta varten käytettiin Unityn Personal-versiota.

3.2 Editori

Unity-pelin kehittäminen tapahtuu Unityn mukana tulevan editorin kautta. Tässä luvussa tutustumme lyhyesti editorin toimintaan ja käyttöön.

Unity-pelit toteutetaan projekteina, jotka sisältävät kaiken koodin, grafiikan, tasot, ja muun sisällön (Lavieri 2015). Unityn käynnistäessä editori mahdollistaa joko aiemman Unity-projektin avaamisen tai uuden Unity-projektin luomisen. Uutta projektia luotaessa Unity antaa määrittää projektin nimen, projektin kehityskansion polun ja sen, onko projektin tarkoitus olla 2D- vai 3D-peli. Unity kuitenkin mahdollistaa sekä 2D- että 3D-komponenttien käytön pelissä, joten alkuperäinen valinta ei rajoita kehittäjää (Lavieri 2015). Tätä tutkimusta varten luotiin 2D-projekti.

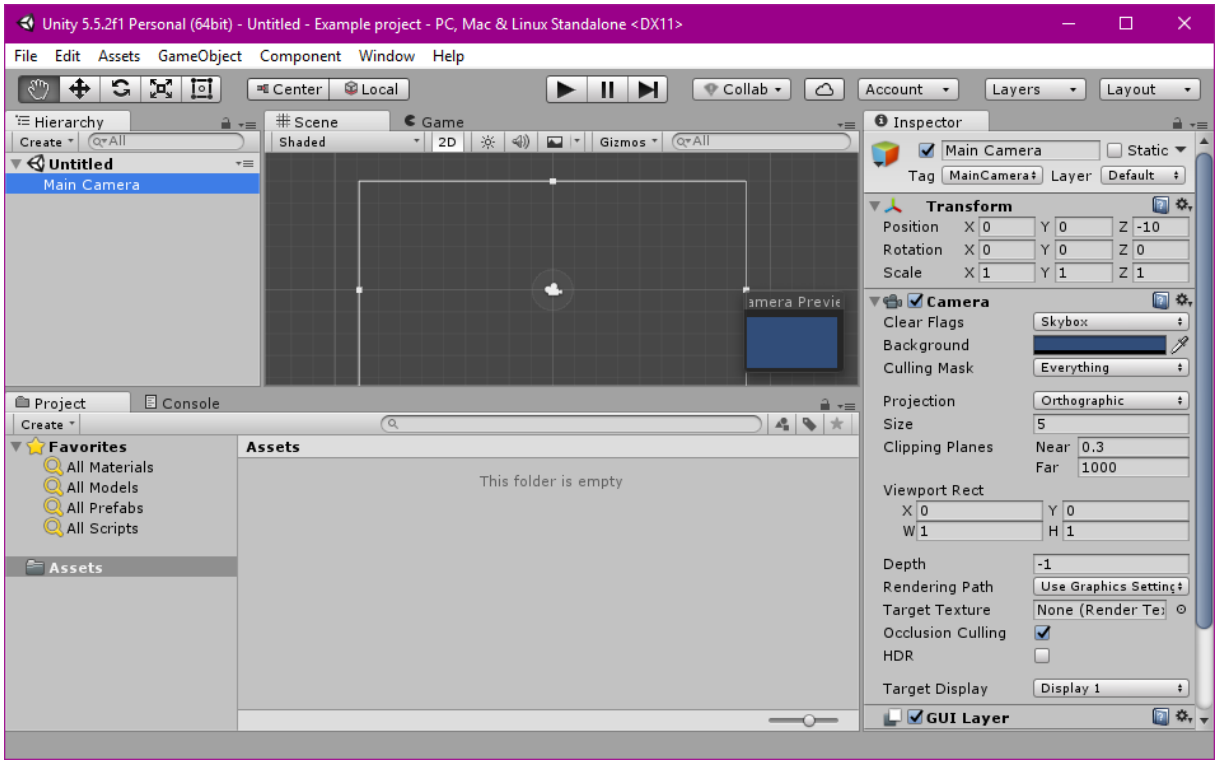


Kuvio 7. Uuden Unity-projektin luonti

Projektin luonnin jälkeen avautuu itse editorin pääkäyttöliittymä, josta käsin Unity-pelin kehittäminen tapahtuu. Editorin käyttöliittymässä on useita eri näkymiä eri tarkoituksia varten. Oletuksena ruudulla on seuraavat pelinkehityksen kannalta olennaiset näkymät (Lavieri 2015):

- *Scene*-näkymä, joka näyttää pelin "scenen"eli kehitettävän tason. Sceneen laitetaan kaikki tason oliot; kaikki, mikä pelissä on näkyvissä tai olemassa, on tasossa oleva olio. Unity-pelissä on aina yksi tai useampia tasoja, joista kehitetään editorissa kerrallaan vain yhtä.
- *Game*-näkymä. Yksi Unityn vahvuuksista on se, että peliä voi pelata suoraan editorista ilman erityistä kääntämistä. Kun peli laitetaan päälle, se näkyy ja toimii *Game*-näkymässä lähes kuin valmis, käännetty peli toimisi. *Game*-näkymän kokoa voi muuttaa ja siten voi myös helposti testata pelin toimivuutta erilaisilla kuvasuhteilla.
- *Hierarchy*-näkymä, joka näyttää hierarkian, joka sisältää jokaisen tason olion. Näkymän kautta olioita voi lisätä ja poistaa, sekä olioiden välille voi määrittää suhteita. Olioita voi myös valita, jolloin ne korostetaan *Scene*-näkymässä ja niiden ominaisuudet näytetään *Inspector*-näkymässä. Olioihin tutustutaan tarkemmin luvussa 3.3.

- *Project*-näkymä, joka näyttää puun Unity-projektin kehityskansiosta. *Project*-näkymän avulla projektin kehityskansiota voidaan selata ja projektiin lisätä tiedostoja, kuten tekstuureita, malleja, skriptejä ja ääniä. Yksinkertaisin keino sisällön lisäämiseen Windows-alustalla on sisältötiedostojen raahaaminen Windowsin resurssienhallinnasta suoraan Unityn *Project*-näkymään. Kuten MonoGame, myös Unity tukee laajasti erilaisia tiedostomuotoja, ja tarjoaa siten suoran ratkaisun eri sisältömuotojen tukemiseen liittyvään pelinkehityksen haasteeseen (Kanode ja Haddad 2009; Unity Technologies 2017d) *Project*-näkymästä on myös mahdollista raahata sopivia olioita suoraan *Scene*-näkymään, jolloin oliot lisätään suoraan pelattavaan tasoon.
- *Inspector*-näkymä, joka näyttää *Hierarchy*-näkymästä valitun olion komponentit ja ominaisuudet sekä mahdollistaa näiden muokkaamisen. Olioista ja komponenteista kerrotaan tarkemmin luvussa 3.3.



Kuvio 8. Unityn editorin käyttöliittymä

Eri näkymien sijainnit ja koot ovat vapaasti käyttäjän kustomoitavissa.

3.3 Pelin arkkitehtuuri, pelioliot ja komponentit

Unity noudattaa luokkahierarkiaa, jossa ylimpänä on `Object`-luokka, josta kaikki muut luokat peritään. Unityn kannalta tärkeimmät luokat ovat `GameObject`, `Component`, ja `MonoBehaviour` (Rogers 2012). `GameObject`-luokka on yksinkertainen säiliö `Component`-luokan instansseille, joita tästä lähtien kutsutaan komponenteiksi. Kaikki pelin yksittäisessä tasossa esiintyvät oliot ovat `GameObject`-luokan esiintymiä. Tästä lähtien `GameObject`-luokan instansseja kutsutaan peliolioiksi.

Komponentit määrittävät jokaisen peliolion varsinaisen toiminnallisuuden. Jokaiseen peliolioon voi lisätä komponentteja, ja komponentteja voi myös poistaa. Jokaisella komponentilla voi olla omia ominaisuuksiaan, joita muokkaamalla *Inspector*-näkymän kautta saadaan muutettua peliolion ulkoasua tai käyttäytymistä. Unityssa on mukana kymmenittäin erilaisia komponentteja, joilla jokaisella on oma tarkoituksensa. Esimerkiksi `Transform`-komponentti määrittää peliolion sijainnin, kierron sekä koon tasossa, ja `Rigidbody`-komponentin avulla olio saadaan noudattamaan mahdollista pelimaailman fysiikkaa (Rogers 2012).

`GameObject`-luokka on suljettu, joten sitä ei voi periä ohjelmakoodin kautta. Unityssa uudelleenkäytettäville peliolioille ei ohjelmoidakaan omia luokkia niiden toiminnallisuuden määrittämiseen, vaan pelioliot tallennetaan projektiin *prefabeiksi*. Prefabit ovat projektin kansioon, levylle tallennettuja valmiita peliolioita, joita on helppo uudelleenkäyttää editorissa. Kun tason yksittäiselle pelioliolle on määritetty komponentit *Inspector*-näkymän kautta, olion voi ottaa editorin *Hierarchy*-näkymästä ja raahata *Project*-näkymässä olevaan kansioon, jolloin Unity luo automaattisesti prefabin peliolion pohjalta. Prefabiin tallennetun peliolion saa vastaavasti lisättyä joko alkuperäiseen tasoonsa tai johonkin toiseen tasoon raahaamalla prefabin *Project*-näkymästä joko *Scene*-näkymään tai *Hierarchy*-näkymään, jolloin Unity luo uuden peliolion prefabin pohjalta. Prefab jää aloilleen alkuperäiseen kansioonsa, jolloin siihen tallennettua pelioliota voidaan helposti monistaa ja käyttää uudelleen, jos peliin luodaan uusia tasoja (Rogers 2012).

`MonoBehaviour`-luokka on `Component`-luokan perivä luokka, joka mahdollistaa olion toiminnallisuutta määrittävän koodin eli skriptien liittämisen peliolioon. Skripteihin tutustutaan tarkemmin seuraavassa luvussa.

3.4 Skriptit

Unityn taustalla toimiva pelimoottori on toteutettu C- ja C++ -kielillä, mutta varsinainen pelilogiikka Unityssa toteutetaan skripteillä (Rogers 2012). Näitä skriptejä voi kirjoittaa joko UnityScriptillä, joka on Unitya varten muokattu versio JavaScriptistä, tai C#-kielellä. Tutkimuksen keskittyessä pelien tekemiseen C#-kielellä, tässä tutkimuksessa skriptit on toteutettu C#:lla.

Kuten aiemmassa luvussa todettiin, skriptit liitetään peliolioihin komponentteina. Jokaisen skriptin täytyy periä `MonoBehaviour`-luokka, joka itsessään on `Component`-luokasta peritty luokka. Siten yksittäinen skripti toimii komponenttina vastaavasti kuin muutkin komponentit. Kun skripti on lisätty peliolioon komponenttina, Inspector-näkymän kautta voi muokata peliolioon liitetyn skriptin ominaisuuksia eli julkisia muuttujia.

Skripteihin on mahdollista toteuttaa ennalta määrättyjä metodeja, joita Unity kutsuu eri tapahtumien yhteydessä ajaessaan pelilogiikkaa. Näiden metodien kautta voidaan vaikuttaa peliolion toimintaan ja yleensäkin pelimaailman tilaan. Koska skriptit ovat tavallista C#-koodia, näiden ennalta määrättyjen metodien kautta kehittäjän on mahdollista myös kutsua omia metodejaan. Seuraavassa taulussa luetellaan olennaisimmat metodit, joita Unity kutsuu skripteistä automaattisesti.

Taulukko 2. Keskeisiä metodeja, joita Unity kutsuu skripteistä (Unity Technologies 2017c)

Funktion nimi	**Tapahtuma, jonka yhteydessä funktiota kutsutaan**
Awake	Kutsutaan jokaiselle pelioliolle, kun taso (scene) ladataan.
Start	Kutsutaan ennen peliolion elämän ensimmäistä framea / päivitystä.
Update	Kutsutaan jokaisella framella ennen ruudun piirtämistä ja animaatioiden laskemista.
FixedUpdate	Kutsutaan aina ennen pelimoottorin fysiikkamallinnuksen päivittämistä. *Update*-funktion kutsunopeus riippuu peliä suorittavasta järjestelmästä, kun *FixedUpdate*-funktiota pyritään kutsumaan aina tietyin, ennalta määrätyin väliajoin. Esimerkiksi fysiikkaan vaikuttava koodi kannattaa siten kirjoittaa ennemmin *FixedUpdate*-metodiin kuin *Update*-metodiin.
LateUpdate	Kutsutaan jokaisella framella sen jälkeen, kun Update- ja FixedUpdate-metodit on ajettu kaikille peliolioille. Hyödyllinen, jos skriptin täytyy päivittää itseään peliolioiden liikkumisen jälkeen. Esimerkiksi jos pelin kameran on tarkoitus seurata jotain tiettyä pelioliota, kameran sijainnin päivitys kannattaa toteuttaa *LateUpdate*-metodin avulla.

Taulussa listattujen metodien lisäksi on metodeja, joiden käyttötarkoitus on erityisempi. Tällaisia ovat esimerkiksi `OnMouseOver` ja `OnMouseDown`-metodit pelaajan hiirellä antaman syötteen käsittelyyn, sekä `OnTriggerEnter` ja `OnCollisionEnter`-metodit fyysisten tapahtumien, kuten peliolioiden välisten törmäysten, käsittelyyn.

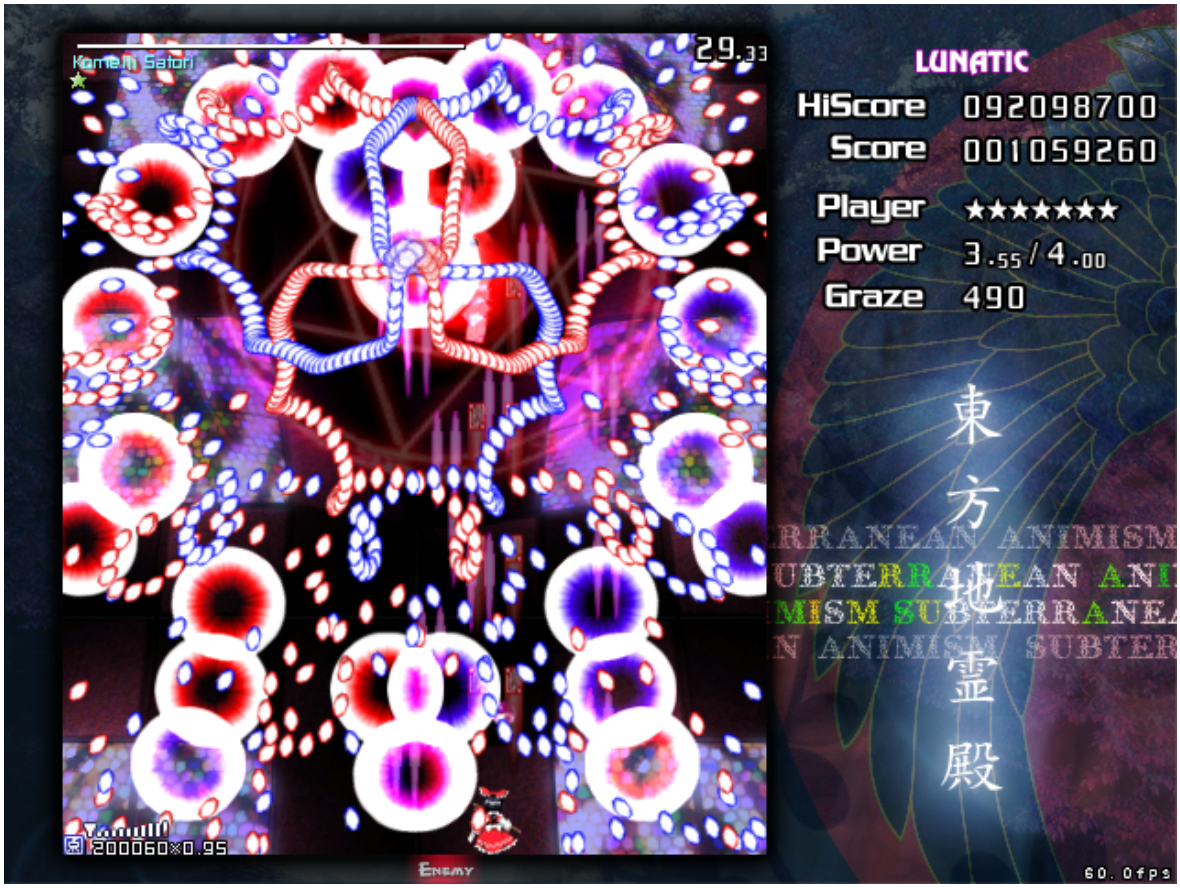
Unityn mukana tulee *MonoDevelop*-sovellus skriptien kirjoittamista ja testaamista (debugging) varten. MonoDevelopin lisäksi Unity tukee integraatiota myös Visual Studion kanssa. Joitain kehittämistä helpottavia lisäominaisuuksia saa myös erillisen Visual Studio Tools for Unity -laajennoksen kautta (Microsoft 2016). Tässä tutkimuksessa käytettiin Unityn kanssa Visual Studiota.

4 Esimerkkipeli: bullet hell

Tässä luvussa esitellään tutkimusta varten tehty esimerkkipeli ja verrataan pelin toteutusta sekä MonoGamella että Unityllä.

4.1 Pelin kuvaus

Toteutettu esimerkkipeli kuuluu bullet hell -genreen, joka on shoot em' up (shmup) -lajin alityyppi. Shmup-peleissä pelaajan tarkoituksena on ampua vihollista tai vihollisia samaan aikaan, kun pelaaja itse väistelee vihollisten ampumia ammuksia. Yksi varhaisimpia ja tunnetuimpia shmup-genren pelejä on *Space Invaders*. Bullet hell -genren erikoisuutena on nimensä mukaisesti vihollisten ampumien ammusten suuri määrä: ammukset usein peittävät suurimman osan ruudusta. Bullet hell -genre syntyi, kun 2D-pelien kehittäjien täytyi löytää uusia keinoja pelaajien houkuttelemiseen 3D-pelien grafiikoiden parantuessa (Collman 2014). Bullet hell -peleissä ammuksia onkin paitsi suuria määriä, ammukset usein muodostavat monimutkaisia ja visuaalisesti miellyttäviä kuvioita, joita on tarkoitus tulkita ammusten väistämiseksi. Liian moneen ammukseen törmääminen johtaa pelin päättymiseen, samoin kaikkien vihollisten tuhoaminen.



Kuvio 9. Esimerkki bullet hell -genrestä. Kuvakaappaus pelistä Touhou Chireiden ~ Subterranean Animism (Team Shanghai Alice 2008)

Syy bullet hell -pelin valintaan vertailusovellukseksi on pelin suhteellisessa yksinkertaisuudessa ja aiemmassa tutkimuksessa. MonoGamea ja Unitya on vertailtu aiemmin Oregon-lautapelin toteutuksen näkökulmasta (Päiveröinen 2014). Bullet hell -peli on nopeatempoisena toiminnallisena pelinä hyvin erilainen, ja antaa siten uudenlaisen näkökulman sovellusten vertaamiseen.

Bullet hell -peleissä on tyypillisesti useita tasoja, ja jokaisessa tasossa on kymmeniä ellei satoja vihollisia, jotka ampuvat kohti pelaajaa. Tason lopussa on usein tavallista vaikeampi vastus, jolla on paljon kestoa ja joka ampuu tavallisia vihollisia enemmän ammuksia monenlaisissa eri kuvioissa ja vaiheissa. Koska tätä tutkimusta varten tehdyn pelin tarkoituksena on puhtaasti MonoGamen ja Unityn vertaaminen, toteutetaan peliin vain sen verran sisältöä,

mikä on tarpeellista vertailua varten. Esimerkkipelissä on siten vain yksi taso, jossa on yksi monivaiheinen vihollinen. Pelaajan täytyy väistää vihollisen luomia ammuksia ja ampua itse vihollista samaan aikaan.

4.2 Tekniset vaatimukset ja suunnittelu

Tässä alaluvussa määrittelemme esimerkkipelin tekniset vaatimukset ja pelilliset ominaisuudet.

Graafisesti pelissä täytyy olla saumattomasti jatkuva, vierivä tausta sekä grafiikat pelaajalle, viholliselle ja ammuksille. Merkittäville pelitapahtumille, esimerkiksi ammukseen törmäämiselle, täytyy olla äänet. Käyttöliittymässä tulee olla näkyvissä kestopalkki viholliselle sekä tekstit pelaajan elämien määrälle ja myös pelaajan tulivoimalle. Vihollisen ampumaan ammukseen törmääminen vähentää pelaajalta elämän sekä myös pelaajan tulivoimaa. Pelaajan tulivoiman väheneminen ilmenee pelillisesti pelaajan luomien ammusten määrän vähenemisenä. Pelaajan ammuksen osuminen viholliseen vähentää viholliselta kestoa.

Pelilogiikassa merkittävin haaste on pelilogiikan päivittäminen, kuten ammusten ja pelaajan liikuttaminen. Pelimoottorin täytyy kyetä päivittämään ja piirtämään sulavasti satoja ruudulla kerrallaan olevia ammuksia, mikä vaatii pelimoottorilta korkeaa suorituskykyä. Korkea suorituskyky on myös yksi keskeinen tavoite, johon pelinkehittäjät pyrkivät pelimoottoreissaan (Wang ja Nordmark 2015). Pelilogiikan päivittämiseen kuuluu myös törmäystarkistukset; pelin täytyy tunnistaa, milloin vihollisen ampuma ammus osuu pelaajaan, tai pelaajan ampuma ammus viholliseen.

Sisällöllisesti pelissä on yksi taso, jossa on pääosan ajasta yksi vihollinen. Vihollisella on kolme erilaista hyökkäystä: ensimmäinen hyökkäys on yksinkertainen kuvio, jossa ammutaan osa ammuksista kohti pelaajaa, ja osa ammuksista ympäristöön kiihtyvällä nopeudella. Toisessa hyökkäyksessä vihollinen ampuu satunnaisesti ympäristöön ammuksia, jotka kimpoilevat tason seinämistä. Kolmannessa hyökkäyksessä vihollinen luo ympäristöön kaareutuvasti liikkuvia ammuksia, jotka luovat lisää ammuksia liikkuessaan. Nämä uudet ammukset ovat aluksi paikoillaan, mutta alkuperäisten ammusten poistuessa ruudulta nämä uudet ammukset liikkuvat, rikkoen kaarensa. Vihollinen vaihtaa hyökkäyksestä toiseen kestopis-

teiden loputtua; jokaisella hyökkäyksellä on omat kestopisteensä. Kolmannen hyökkäyksen jälkeen tai pelaajan elämien loputtua peli päättyy. Näiden kolmen hyökkäyksen avulla pyritään mallintamaan ammusten tyypillistä käyttäytymistä bullet hell -peleissä.

Pelaajan kontrolleina on hahmon liikuttaminen ja ampuminen. Liikuttaminen tapahtuu nuolinäppäimillä. Vasenta Shift-painiketta painamalla pelaaja voi halutessaan liikkua hitaampaa, mikä mahdollistaa hahmon tarkemman ohjauksen. Z-näppäintä painamalla pelaajan hahmo ampuu. Nämä kontrollit mahdollistavat hahmon liikuttamisen oikealla kädellä samalla, kun vasemmalla kädellä lähinnä säädetään liikkumanopeutta.

Yleensä bullet hell -peleissä on monta eri vaikeustasoa, joista valita. Vaikeustasojen toteuttaminen ei kuitenkaan yksinkertaisuudessaan toisi tutkimukselle lisäarvoa, joten tutkimusta varten tehdyssä pelissä on vain yksi vaikeustaso.

Tutkimuksen johdannossa määritelyjen pelinkehityksen teknisten haasteiden ja pelimoottorilta yleisesti haluttujen ominaisuuksien osalta esimerkkipelin kehityksessä testataan sisällön hallintaa, pelimoottorin pelin kehityksessä säästämää aikaa, sekä pelimoottorin suorituskykyä. Yhdeksi pelinkehityksen merkittäväksi haasteeksi todettiin myös useiden eri alustojen tukeminen (Kanode ja Haddad 2009); tutkimuksen rajaamiseksi tätä ei testata esimerkkipelillä, vaan peli toteutetaan vain tietokoneelle ja Windows-käyttöjärjestelmälle.

4.3 Toteutus MonoGamella

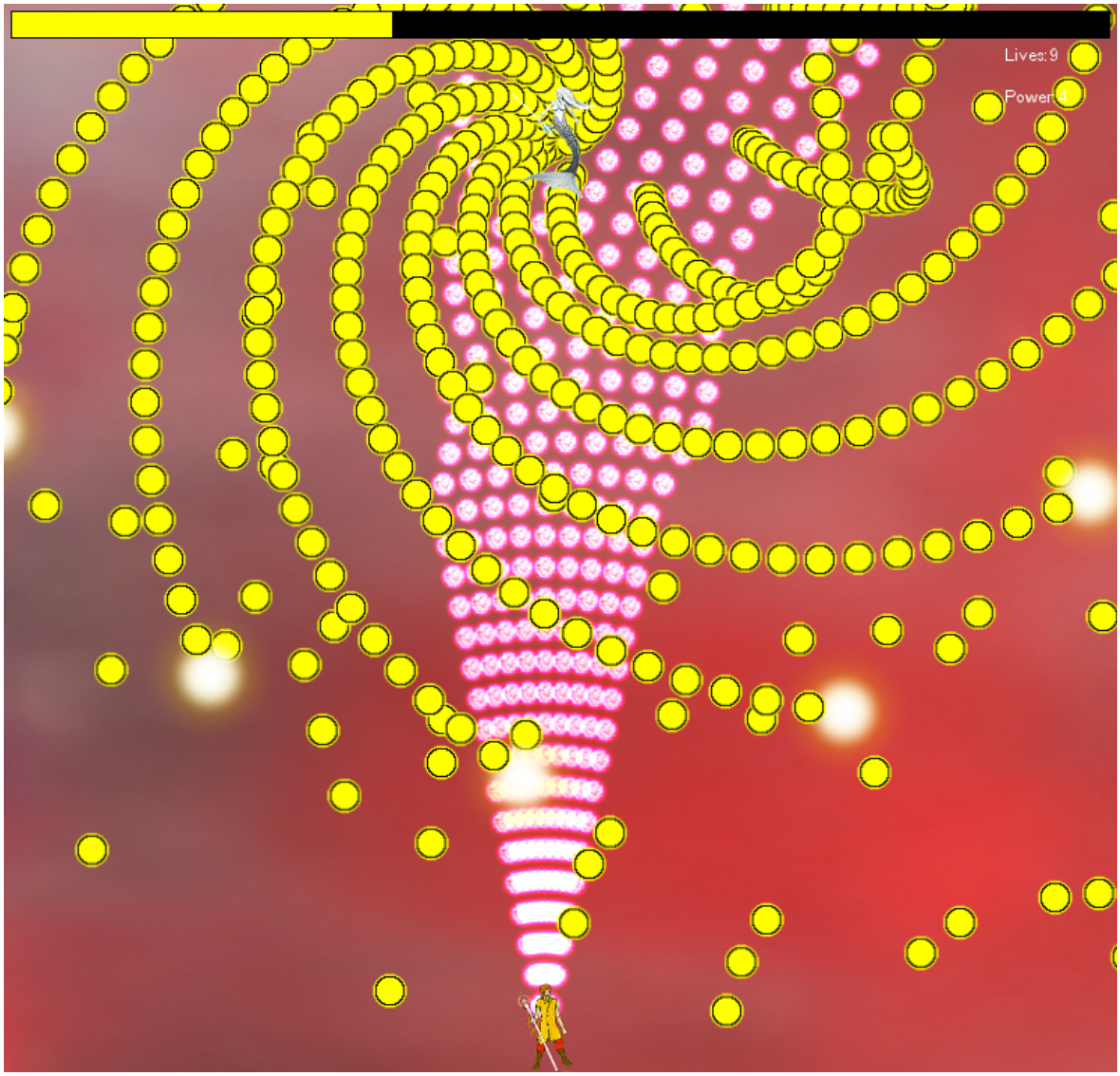
MonoGamessa ei ole sisäänrakennettua fysiikkamoottoria esimerkiksi törmäystarkistuksia varten. MonoGamen kanssa voi kuitenkin käyttää yleisesti C#-kielelle tehtyjä fysiikkamoottoreita, kuten Velcro Physics -moottoria (Velcro Physics 2017). Bullet hell -pelin tapahtumat, kuten pelaajan keston vähentäminen ammuksen osuessa pelaajaan, ovat kuitenkin varsin yksinkertaisia, eikä pelissä siten tarvita kovin monimutkaisia fysiikkaoperaatioita, kuten kappaleiden kimpoamista toisistaan. Siten esimerkkipelin MonoGame-toteutuksessa ei käytetä kokonaista valmista fysiikkamoottoria, vaan törmäystarkistukset kirjoitetaan pelimoottoriin itse.

Pelin eri komponentit toteutetaan luomalla `GameComponent`-luokasta periviä uusia luok-

kia. Tällaisia komponentteja ovat vierivä tausta, vihollisen kestopalkki, tekstikenttä pelaajan elämien määrälle ja tulivoimalle, sekä tärkeimpänä itse pelilogiikkaa päivittävä sekä pelioliot, kuten pelaajan, vihollisen ja ammukset, piirtävä komponentti. Pelaaja, vihollinen, ammukset ja vihollisen hyökkäykset toteutetaan myös omina luokkinaan, mutta nämä eivät peri `GameComponent`-luokkaa. Pelilogiikkaa päivittävällä komponentilla on jäseninä pelaajan ja vihollisen sekä ammuslistojen instanssit.

Vihollisen hyökkäykset toteutetaan omana luokkanaan, jonka instansseilla on lista erillisistä ammusten luojista. Jokainen erilainen ammusten luoja, joita voi olla hyökkäyksessä useampia, toteutetaan omana luokkanaan, joka perii erillisen `BulletSpawner`-luokan, joka sisältää yleistä logiikkaa ammusten luomista varten. Vihollinen päivittää pelilogiikassaan hyökkäysluokan instanssia, joka vuorostaan päivittää ammusten luojia.

Sisällön hallinnan aiheuttama pelinkehitykseen liittyvä haaste (Kanode ja Haddad 2009) ratkaistaan lisäämällä taustan, pelaajan, vihollisen ja ammusten tekstuurit sekä äänet peliin *MonoGame Pipeline* -työkalulla, ja nämä otetaan koodissa käyttöön `ContentManager`-luokan instanssin avulla.



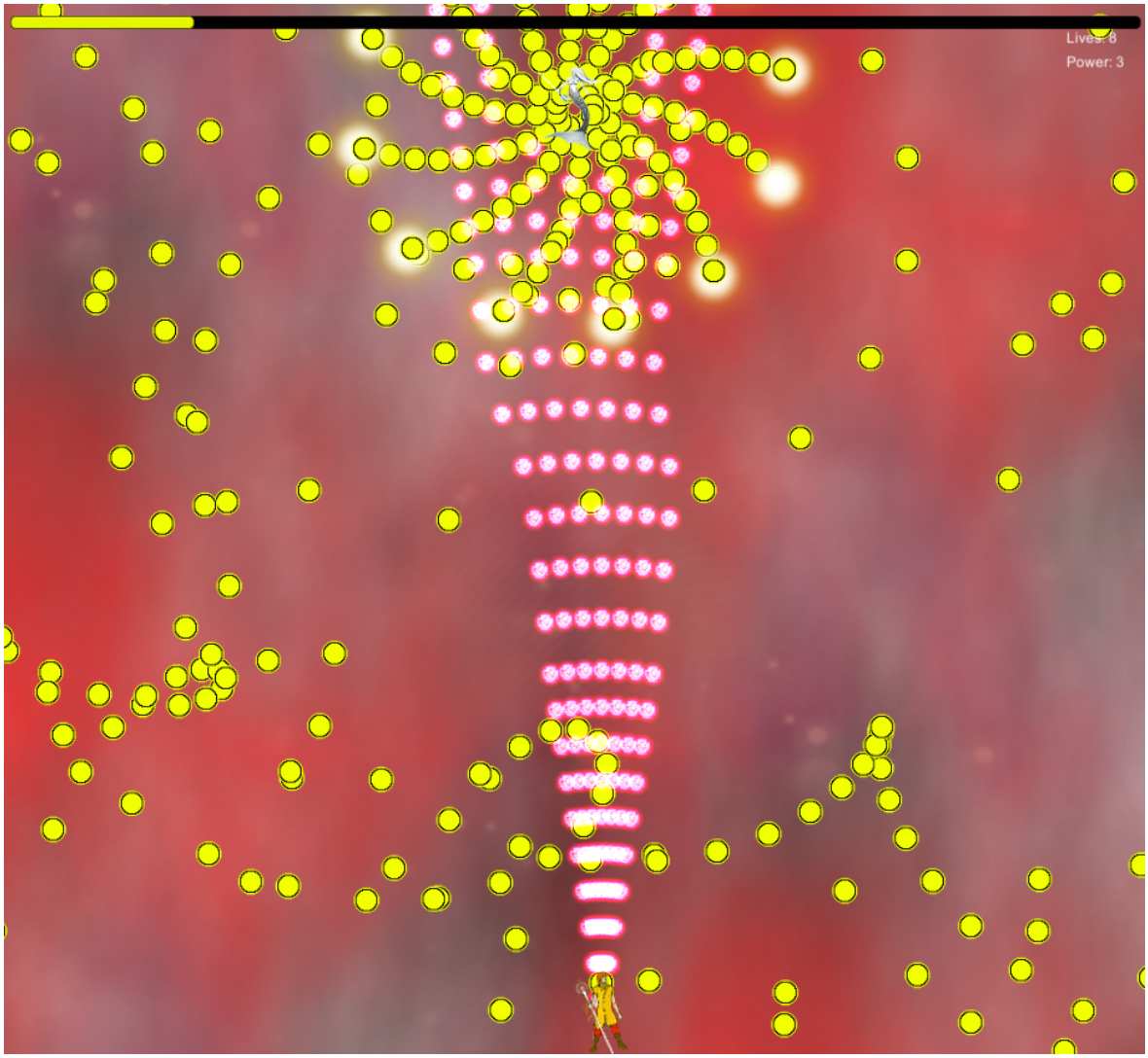
Kuvio 10. MonoGamella toteutettu esimerkkipeli

4.4 Toteutus Unitylla

Pelin toteutus Unitylla tapahtuu luomalla Unityn editorin avulla yksi taso, johon lisätään peliolioita, kuten tausta, vihollinen, pelaajan hahmo, ja käyttöliittymää varten oliot kestopalkille ja tekstit pelaajan elämien määrälle sekä tulivoimalle. Pelilogiikka toteutetaan skripteinä, jotka liitetään komponentteina peliolioihin. Esimerkiksi pelaajan toimintaa ohjaa skripti, joka liitetään pelaajan peliolioon. Vastaavasti viholliselle luodaan oma skriptinsä esimerkiksi

törmäysten käsittelyä varten. Myös ammusten käyttäytymistä varten luodaan skriptit, sekä vihollisen jokainen hyökkäys toteutetaan skriptinä, joka päivittyessään luo ammuksia pelisuunnittelun mukaisesti erillisten ammusten luojien avulla. Unityn pelimoottorin valmista logiikkaa hyödynnetään mahdollisimman paljon: esimerkiksi törmäystarkistuksia ei kirjoiteta erikseen, vaan Unityn fysiikkamoottorin annetaan hoitaa ne. Hieman vastaavasti kuin MonoGame-toteutuksessa, jokainen vihollisen hyökkäys toteutaan myös omana skriptinään, jolla on taulukko hyökkäyksen käyttämistä ammusten luojista. Erilaiset ammustyypit lisätään projektiin prefabeina, ja ammusten luojilla on viittaukset näihin prefabeihin.

Pelin sisällön hallinnan tuoman haasteen (Kanode ja Haddad 2009) Unity ratkaisee suoraviivaisesti: pelin käyttämät tekstuurit ja äänet lisätään raahaamalla sisältö Windowsin resurssienhallinnasta Unityn editorin *Project*-näkymään. Sisältö otetaan käyttöön yhdistämällä se peliolioiden komponentteihin editorin avulla.



Kuvio 11. Unitylla toteutettu esimerkkipeli

4.5 Toteutusten ja kehitysprosessien vertailu

Tekijällä oli jo runsaasti valmista kokemusta sekä MonoGamen että Unityn käytöstä. Tekijä ei kuitenkaan ollut tehnyt niillä aiemmin täysin vastaavaa projektia, joka olisi mahdollistanut vertailun.

Esimerkkipeli toteutettiin suunnitelman mukaisesti sekä MonoGamella että Unitylla. MonoGamella itse ohjelmakoodia kirjoitettiin noin puolet enemmän kuin Unitylla. Erityisesti

Unitya varten ei tarvinnut kirjoittaa törmäystarkistuksia, ja käyttöliittymäkomponentit eivät myöskään vaatineet riviäkään koodia. Tämä ei kuitenkaan tarkoittanut bullet hell -pelin yhteydessä kovin paljoa pienempää työmäärää, sillä käyttöliittymäkomponentit sekä törmäystarkistukset olivat yksinkertaisia eivätkä vaatineet kovin paljoa työtä. Unityssa työskenneltiin merkittävästi myös editorin kautta luodessa tasoa, mikä on työtä, joka ei näy koodin määrässä.

Työssä toteutetun esimerkkipelin perusteella voidaan väittää, että pelin varsinainen ohjelmointityö on yksinkertaisempaa ja siten hieman helpompaa Unityllä. Unityn arkkitehtuuri piilottaa itse pelimoottorin ohjelmakoodin kehittäjältä, kun MonoGamella joutuu pohtimaan enemmän esimerkiksi pelin komponenttien alustamista ja luokkien välistä hierarkiaa. Opetuksellisesta näkökulmasta tämä ei toisaalta ole yksiselitteisesti hyvä asia, sillä sekä MonoGame- että Unity-kehittäjän täytyy vähänkään monimutkaisempaa peliä tehdessään joka tapauksessa hallita olio-ohjelmoinnin käsitteet.

Jos pelistä olisi tehnyt laajemman, kokonaisen kaupallisen luokan bullet hell -pelin, siinä olisi ollut useita tasoja sekä enemmän vihollisia ja hyökkäyksiä, ja nämä olisivat yhdessä sisällön kanssa vieneet lähes kaiken pelin kehitysajasta. Esimerkkipelissä oleviin hyökkäyksiin käytetyn koodin sekä yleensäkin työn määrä oli sekä MonoGamella että Unitylla suunnilleen sama, mistä voidaan päätellä, että laajemmassa bullet hell -pelissä työn määrässä ei olisi juurikaan ollut eroa sovellusten välillä. Kolmannen osapuolen pelimoottorien ja muun teknologian käytön olennaisin tavoite on kehitysajan säästäminen työn vähentämisen kautta (Kanode ja Haddad 2009), ja tältä osin Unity ja MonoGame molemmat pärjäsivät bullet hell -pelin osalta yhtä vahvasti. Tekijällä oli toisaalta valmiiksi runsaasti kokemusta molemmista ympäristöistä, joten uuden ympäristön opetteluun ei mennyt aikaa.

Sisällön hallinta on yksi pelinkehityksen haasteista, jonka ratkaisemista kehittäjät myös odottavat pelimoottorilta (Kanode ja Haddad 2009; Wang ja Nordmark 2015). Sisällön, kuten tekstuurien ja äänien, lisääminen tapahtui Unitylla hieman MonoGamea helpommin. Unitylla riitti, että sisältötiedostot raahasi Unityn editoriin esimerkiksi Windowsin resurssienhallinnasta. MonoGamella sisällön joutui vuorostaan lisäämään *MonoGame Pipeline* -työkalun avulla. Toisaalta *MonoGame Pipeline* -työkalun käyttö oli myös nopeaa ja yksinkertaista, kun käyttöliittymään oli tottunut. Työkalun oppimiskynnystä ei myöskään koettu korkeaksi.

Siten sisällön hallinta ei ollut MonoGamella kovin merkittävästi vaikeampaa kuin Unitylla.

Sekä MonoGame että Unity lupaavat tuen lukuisille eri alustoille, Unityn tuen kattaessa laajemman määrän erilaisia alustoja (MonoGame 2017a; Unity Technologies 2017e). Tutkimuksen rajaamiseksi tätä esimerkkipeliä ei testattu useilla eri alustoilla, vaan peli toteutettiin ainoastaan Windowsin työpöytäversiolle.

Noin puolelle pelinkehittäjistä pelin arkkitehtuurin tärkein tavoite on suorituskyky (Wang ja Nordmark 2015). Toteuttaessa bullet hell -peliä merkittävin Unityn ja MonoGamen välinen ero huomattiin juuri suorituskyvyssä. Erityisesti testatessa peliä Unityn editorin kautta pelin sulavuus kärsi herkästi ammusmäärän noustessa korkeaksi, kun MonoGamella vastaavaa ongelmaa ei ollut. Testatessa käännettyä peliä Unityn editorin ulkopuolella suorituskykyongelma oli edelleen läsnä, mutta vain lievänä. Pelin kääntäminen vie kuitenkin huomattavasti aikaa, joten se voisi hidastaa testaamista verrattuna MonoGameen, pidentäen kehitysaikaa ja kasvattaen kustannuksia. Bullet hell -pelit vaativat usein pelaajahahmon tarkkaa liikuttamista ja nopeita refleksejä, joten pelin sulavuus on niissä tärkeää. Siten suorityskykyongelma tekisi pelin testaamisesta kehitysaikana vaikeampaa ja haittaisi kokonaisen bullet hell -pelin kehittämistä Unitylla.

Kokonaisuutena lähinnä suorituskyvyn vuoksi MonoGamen voi sanoa olevan hieman Unityä soveltuvampi kokonaisen, laajan bullet hell -pelin luontiin. Unitynkaan ei toisaalta voi sanoa olevan huono valinta uuden bullet hell -pelin tekijälle, varsinkaan aloittelevalle pelinkehittäjälle. Unityn suljettu pelimoottori ja graafinen editori tarjoavat MonoGamea matalamman oppimiskynnyksen pelin kehittämisen aloittamiseen, ja siten aloitteleva pelinkehittäjä olisi todennäköisesti säästänyt aikaa käyttämällä Unitya MonoGamen sijasta. Unityn matalampi oppimiskynnys helpottaa etenkin kehittäjiä, joilla ei ole aiempaa ohjelmointikokemusta. Toisaalta Unityn arkkitehtuurin perustuessa komponentteihin perinteisemmän oliohierarkian sijaan, helpotus voi tapahtua myös olio-ohjelmoinnin konseptien oppimisen kustannuksella, mikä voi vaikeuttaa pelien jatkokehittämistä. Jos projektissa olisi lisäksi tarvittu enemmän valmista pelilogiikkaa, kuten tarkasti mallinnettua tai monimutkaista fysiikkaa, Unity olisi valmiin pelilogiikkansa avulla kuitenkin säästänyt merkittävästi aikaa MonoGameen verrattuna, riippumatta kehittäjän aiemmasta kokemuksesta.

Yksi pelinkehittäjälle potentiaalisesti merkittävä asia voisi olla myös Unityn maksullisuus: Unityn ilmaisversiossa on liikevaihtorajoite, ja lisäksi ilmaisversio näyttää pelin käynnistyessä Made with Unity -ruudun. Toisaalta aiemmassa tutkimuksessa on todettu, että pelinkehittäjät eivät koe käytettyjen kehitystyökalujen hintaa kovin merkittäväksi asiaksi edes startup-yrityksissä (Kasurinen, Strandén ja Smolander 2013).

Kehitystyökaluilta odotetaan myös mukautuvuutta pelinkehitysprosessin aikana tapahtuviin suunnitelman muutoksiin (Kasurinen, Strandén ja Smolander 2013). Pelin toteutuksessa ei tullut ilmi seikkoja, jotka tekisivät MonoGamesta tai Unitysta toisiinsa verrattuna vähemmän mukautuvia eri tilanteisiin.

5 Yhteenveto

Tutkimuksen tarkoituksena oli selvittää, miten pelinkehitys MonoGamella ja Unitylla eroaa toisistaan, ja mitkä ovat näiden sovellusten potentiaaliset vahvuudet toisiinsa verrattuna. Näkökulmassa painotettiin 2D-pelejä aloittelevan pelikehittäjän näkökulmasta.

Aiemmassa tutkimuksessa Unityn on todettu säästävän aikaa MonoGameen verrattuna toteuttaessa Oregon-lautapelin käännöstä tietokoneelle, toteutusten ollessa muuten yhtä päteviä (Päiveröinen 2014). Erityisesti Unityn on sanottu olevan aloittelevan pelinkehittäjän näkökulmasta MonoGamea parempi vaihtoehto käyttäjäystävällisen graafisen käyttöympäristönsä ansiosta (Päiveröinen 2014). Tämän tutkimuksen esimerkkiprojektissa Unity ei vuorostaan merkittävästi säästänyt aikaa MonoGameen verrattuna, mutta toisaalta Unity teki pelin kehittämisestä joiltain osin vähemmän haastavaa. Aloittelevan pelinkehittäjän tai varsinkin aloittelevan ohjelmoijan kannalta Unity olisi siten todennäköisesti säästänyt jonkin verran aikaa myös bullet hell -pelin toteuttamisessa. Tutkimuksen toteuttajalla oli ennestään jo runsaasti kokemusta sekä MonoGamesta että Unitysta, joten Unityn tarjoampi matalampi kynnys pelinkehityksen aloittamiseen ei juurikaan vaikuttanut esimerkkipelin toteutuksessa. Toisin kuin Oregon-lautapelin yhteydessä, MonoGame ylsi bullet hell -pelin toteutuksessa Unitya parempaan lopputulokseen pelin suorituskyvyn kannalta, ja myös laajan bullet hell -pelin kehittämiseen MonoGamen todettiin soveltuvan paremmin Unity-pelin editorissa testaamisen yhteydessä esiintyvän heikon suorituskyvyn vuoksi.

Sekä MonoGame että Unity tarjoavat pätevän ja kehittämistä helpottavan ympäristön pelien tekemiseksi C#-kielellä. Unity kuitenkin tarjoaa aloittelijaystävällisemmän käyttäjäkokemuksen graafisen editorinsa avulla, ja valmis pelilogiikka esimerkiksi fysiikkaa ja törmäystarkistuksia varten säästää pelin ohjelmoijien aikaa, mikä on olennainen syy valmiin pelimoottorin käyttöön (Kanode ja Haddad 2009). Siten Unity on yleensä MonoGamea parempi vaihtoehto, varsinkin aloitteleville pelinkehittäjille. MonoGamella on tosin avoimemman arkkitehtuurinsa puolesta omat vahvuutensa: jos peliin halutaan juuri jossain tietyssä asiassa tehokas pelimoottori, tämän tutkimuksen perusteella MonoGamella voi saada aikaan suorituskykyisemmän pelin kuin Unityn yleiskäyttöisemmällä pelimoottorilla. Suuri osa pelinkehittäjistä kokee suorityskyvyn pelimoottorin tärkeimmäksi tavoitteeksi (Wang ja Nord-

mark 2015). Kehittäjien kannattaakin tehdä valinta MonoGamen ja Unityn väliltä pohjautuen siihen, millaisen pelin he aikovat tehdä.

Jatkotutkimuksessa Unityn ja MonoGamen käyttöä olisi mahdollista verrata vielä jostain uudesta lähtökohdasta, kuten opetuksellisesta näkökulmasta. Vertailuun voisi ottaa mukaan myös muita pelimoottoreita, kuten Epicin Unreal Enginen.

Lähteet

Chris Charla. 2016. *A Letter from Chris Charla: ID@Xbox Updates and GDC.* Saatavilla WWW-muodossa, http://news.xbox.com/2016/03/14/letter-chris-charla-idxbox-updates-gdc/, viitattu 26.2.2017.

Collman, Nathan. 2014. "MAKU: A Code Generator for Bullet Hell Games". Tohtorinväitöskirja.

Corriea, Alexa. 2013. *Polygon - Microsoft has 'no plans for future versions' of XNA software, will not phase out DirectX.* Saatavilla WWW-muodossa, http://www.polygon.com/2013/1/31/3939230/microsoft-has-no-plans-for-future-versions-of-xna-software, viitattu 12.2.2017.

Grootjans, Riemer. 2009. *XNA 3.0 Game Programming Recipes: A Problem-Solution Approach.* Apress.

Juslin, Sonja-Maria. 2015. "2D-materiaalipaketin toteuttaminen Unity Asset Store-verkkopalveluun".

Kanode, Christopher M, ja Hisham M Haddad. 2009. "Software engineering challenges in game development". Teoksessa *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on,* 260–265. IEEE.

Karppinen, Tomi. 2010. "XNA-ohjelmointi".

Kasurinen, Jussi, Jukka-Pekka Strandén ja Kari Smolander. 2013. "What do game developers expect from development and design tools?" Teoksessa *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering,* 36–41. ACM.

Lavieri, Edward. 2015. *Getting Started with Unity 5.* Packt Publishing Ltd.

Microsoft. 2004. *Microsoft: Next Generation of Games Starts With XNA.* Saatavilla WWW-muodossa, https://news.microsoft.com/2004/03/24/microsoft-next-generation-of-games-starts-with-xna/, viitattu 26.2.2017.

Microsoft. 2011. *Hardware and Operating System Requirements.* Saatavilla WWW-muodossa, https://msdn.microsoft.com/en-us/library/bb203925.aspx, viitattu 26.2.2017.

———. 2016. *Visual Studio Tools for Unity | Microsoft Docs.* Saatavilla WWW-muodossa, https://docs.microsoft.com/en-us/visualstudio/cross-platform/visual-studio-tools-for-unity, viitattu 19.3.2017.

MonoGame. 2015. *MonoGame 3.3.* Saatavilla WWW-muodossa, http://www.monogame.net/2015/03/16/monogame-3-3-2/, viitattu 1.3.2017.

———. 2017a. *About | MonoGame.* Saatavilla WWW-muodossa, http://www.monogame.net/about/, viitattu 12.2.2017.

———. 2017b. *GitHub - MonoGame.* Saatavilla WWW-muodossa, https://github.com/MonoGame/MonoGame, viitattu 1.3.2017.

———. 2017c. *MonoGame | Write Once, Play Everywhere.* Saatavilla WWW-muodossa, http://www.monogame.net/, viitattu 26.2.2017.

———. 2017d. *Using the Pipeline Tool | MonoGame.* Saatavilla WWW-muodossa, http://www.monogame.net/documentation/?page=Using_The_Pipeline_Tool, viitattu 1.3.2017.

MSDN. 2011a. *What's New in XNA Game Studio 4.0 Refresh.* Saatavilla WWW-muodossa, https://msdn.microsoft.com/en-us/library/bb417503.aspx, viitattu 26.2.2017.

———. 2011b. *XNA Game Studio 4.0 Refresh.* Saatavilla WWW-muodossa, https://msdn.microsoft.com/en-us/library/bb200104.aspx, viitattu 26.2.2017.

Päiveröinen, Hannu. 2014. "Unity-ja XNA/MonoGame-pelimoottorien vertailu 2D-pelien toteutuksessa".

RB Whitaker. *Managing Content - RB Whitaker's Wiki.* Saatavilla WWW-muodossa, http://rbwhitaker.wikidot.com/monogame-managing-content, viitattu 1.3.2017.

Reed, Aaron. 2010. *Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7.* "O'Reilly Media, Inc.".

Rogers, Michael P. 2012. "Bringing unity to the classroom". *Journal of Computing Sciences in Colleges* 27 (5): 171–177.

Unity Technologies. 2013. *Unity Releases 2D Tools With 4.3 Update.* Saatavilla WWW-muodossa, `https://unity3d.com/company/public-relations/news/unity-releases-2d-tools-43-update`, viitattu 12.3.2017.

———. 2017a. *Unity - Game engine, tools and multiplatform.* Saatavilla WWW-muodossa, `https://unity3d.com/unity`, viitattu 12.2.2017.

———. 2017b. *Unity - Learn.* Saatavilla WWW-muodossa, `https://unity3d.com/learn`, viitattu 12.3.2017.

———. 2017c. *Unity - Manual: Event Functions.* Saatavilla WWW-muodossa, `https://docs.unity3d.com/Manual/EventFunctions.html`, viitattu 19.3.2017.

———. 2017d. *Unity - Manual: Importing Assets.* Saatavilla WWW-muodossa, `https://docs.unity3d.com/Manual/ImportingAssets.html`, viitattu 25.4.2017.

———. 2017e. *Unity | Multiplatform.* Saatavilla WWW-muodossa, `https://unity3d.com/unity/multiplatform/`, viitattu 12.3.2017.

———. 2017f. *Unity - Store.* Saatavilla WWW-muodossa, `https://store.unity.com/`, viitattu 12.3.2017.

Valente, Luis, Aura Conci ja Bruno Feijó. 2005. "Real time game loop models for single-player computer games". Teoksessa *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment,* 89:99.

Wang, Alf Inge, ja Njål Nordmark. 2015. "Software architectures and the creative processes in game development". Teoksessa *International Conference on Entertainment Computing,* 272–285. Springer.

Vankaandreev. 2011. *3D Graphics for Windows Phone 7 Using the XNA Framework.* Saatavilla WWW-muodossa, https://www.codeproject.com/articles/176162/d-graphics-for-windows-phone-using-the-xna-fram, viitattu 26.2.2017.

Velcro Physics. 2017. *GitHub - VelcroPhysics/VelcroPhysics: High performance 2D collion detection system with realistic physics responses.* Saatavilla WWW-muodossa, https://github.com/VelcroPhysics/VelcroPhysics, viitattu 12.4.2017.

Visual Studio Team. 2016. *XNA 5 - Visual Studio.* Saatavilla WWW-muodossa, https://visualstudio.uservoice.com/forums/121579-visual-studio-ide/suggestions/3725445-xna-5, viitattu 26.2.2017.