OULUN AMMATTIKORKEAKOULU

Verner Holappa

UNITY PELIKEHITYSTYÖKALUNA

UNITY PELIKEHITYSTYÖKALUNA

Verner Holappa
Opinnäytetyö
Kevät 2017
Tietojenkäsittelyn koulutusohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma, web-sovelluskehitys

Tekijä: Verner Holappa
Opinnäytetyön nimi: Unity pelikehitystyökaluna
Työn ohjaaja: Teppo Räisänen
Työn valmistumislukukausi- ja vuosi: Kevät 2017 Sivumäärä: 42

Tämän opinnäytetyön aiheena on Unity kehitystyökaluna ja siinä on tarkoitus esitellä Unity-pelimoottori ja kertoa, kuinka sitä voi hyödyntää ammatillisessa käytössä pelialalla, erityisesti mobiilikehityksessä. Unity on laajasti pelialalla käytössä oleva pelimoottori, jolla voi kehittää pelejä usealle eri alustalle, ja sen suosio pelikehityksessä on kasvanut viime vuosina nopeasti. Työssä käydään läpi Unityllä toteutettavia hyödyllisiä pelikehitykseen liittyviä toimintoja sekä kuinka optimoida pelejä ja noudattaa hyviä käytäntöjä. Opinnäytetyön yhtenä päätavoitteena on tuoda esille ja kehittää omaa osaamistani ohjelmoijana ja pelikehittäjänä sekä tuoda esille kehitystyökalun valinnan tärkeyttä peliprojekteissa.

Aiheen taustalla on oma kiinnostus ja osaaminen pelialaa ja pelikehitystä kohtaan sekä myös työkokemus Koukoi Gamesilla. Koukoi Games on oululainen pelialan yritys, joka valmistaa korkealaatuisia mobiilipelejä iOS- ja Android-alustoille. Työskentelin kyseisessä yrityksessä pelikehittäjänä ja ohjelmoijana opinnäytetyön ajan, ja osallistuin aktiivisesti uuden elokuvaan perustuvan mobiilipelin kehitykseen projektin alusta alkaen. Koukoi tekee yhteistyötä suuren elokuvayhtiön ja sen peliliiketoimintayksikön kanssa tuottaen animaatioelokuvaan pohjautuvaa mobiilipeliä. Elokuvan taustalla on kuuluisa animaatiostudio, joka on monien hittielokuvien luoja.

Keskeisiin käsitteisiin työssä kuuluu Unity, ohjelmointi, pelikehitys, peliala, mobiilikehitys, projektityöskentely ja pelisuunnittelu. Työ sisältää paljon Unityyn ja ohjelmointiin liittyviä teknisiä termejä, joille ei välttämättä löytynyt selkeää suomenkielistä käännöstä, ja ovat siksi jätetty kääntämättä termin alkuperäisen merkityksen säilyttämiseksi. Työssä läpikäytäviä toimintoja ovat muun muassa proseduraalinen generointi, koneoppiminen, serialisointi, suunnittelumallit, komponentit, optimointi, hyvät käytännöt ja monia muita teknisiä käsitteitä. Työn tuloksia voi käyttää esimerkiksi Unity-pelimoottorin edistyneeseen opetteluun tai pelimoottorivalinnan perusteluihin.

Asiasanat: pelikehitys, ohjelmointi, ohjelmistokehitys, pelimoottorit, peliala, mobiilikehitys

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Systems

Author: Verner Holappa
Title of thesis: Unity as a game development tool
Supervisor: Teppo Räisänen
Term and year when the thesis was submitted: Spring 2017 Number of pages: 42

The subject of this thesis is Unity as a game development tool, with the intention of showing how the Unity game engine can be used professionally as a development tool for games. Unity is a widely-used game engine in the gaming industry and has greatly increased in popularity during the recent years. It can be used to develop games on multiple platforms including mobile, PC, consoles and web/social media platforms. In this thesis, we will explain topics concerning Unity in detail and go over many useful development patterns and techniques one can use during game development. Primary goal of this thesis is to explain the importance of development tool choices in game development and to learn more about game development with Unity.

The subject is based on my work at Koukoi Games and previous interest and usage of Unity game engine as a development tool for games. Most inspiration came from personal interest towards the game industry, as well as from the things I learned while working at Koukoi Games. Koukoi Games is a game company from Oulu which focuses on producing high quality mobile games for iOS and Android platforms. I have worked at Koukoi Games for about nine months as a full-time programmer and developer working on a mobile game project based on an animated film. The film is created by a famous animation studio which is also behind multiple hit movies. Koukoi Games has a partnership with a video game division owned by a large movie company. Much of the content in this thesis is based on my time at Koukoi Games and what I learned there.

Some key concepts in this thesis include Unity, programming, game development, mobile development, as well as many other subjects related to them. The work contains many advanced programming and Unity-related terms that were difficult to translate to Finnish directly without them losing their original meaning. Contents include subjects such as procedural generation, machine learning, serialization, design patterns, components, optimization, good practices, and many other technical concepts. The findings of this study can be used for advanced learning of Unity game engine or for the rationalization of engine choices for projects.

Keywords: programming, software engineering, game engines, game industry, game development, mobile development

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön aiheena on Unity kehitystyökaluna ja siinä on tarkoitus esitellä Unity-pelimoottori sekä kertoa, kuinka sitä voi hyödyntää ammatillisessa käytössä pelialalla, erityisesti mobiilikehityksessä. Unity on laajasti pelialalla käytössä oleva pelimoottori, jolla voi kehittää pelejä usealle eri alustalle, ja sen suosio pelikehityksessä on kasvanut viime vuosina nopeasti. Työssä käydään läpi Unityllä toteutettavia hyödyllisiä pelikehitykseen liittyviä toimintoja sekä kuinka optimoida pelejä ja hyviä käytäntöjä. Läpikäytävät asiat ovat hyvin olennaisia ammatillisessa pelikehityksessä. Peliteollisuus on maailmanlaajuisesti suuressa kasvussa oleva viihteenala, ja sillä on myös merkittävä noususuuntaus Suomessa. Pelialalla on yli 300 yritystä, ja alan koulutusta on tarjottu Suomessa kymmenen vuoden ajan 2000-luvun alusta saakka. Peliala vaatii hyvin monipuolista osaamista, ja vasta viime vuosina oppilaitokset ovat alkaneet panostaa enemmän pelialaan erikoistuneisiin koulutusohjelmiin. (Vähäkainu, Mononen & Neittaanmäki 2014, 4.)

Räjähdysmäisen kasvun myötä on helppo ymmärtää Unityn kaltaisten pelimoottorien tarjoaman hyödyn. Pelikehitys on vuosien varrella muovautunut helpommaksi ja nopeammaksi uusien työkalujen ja innovaatioiden myötä, ja useat nykypäivän pelialan yrityksen käyttävät Unityn kaltaisia kehitystyökaluja juuri tästä syystä. Unity on helppo, nopea ja joustaja, ja se soveltuu niin 2D- kuin 3D-pelikehitykseen. Syy Unityn helppouteen ja tehokkuuteen on sen graafinen käyttöliittymä ja suhteellisen helppo ohjelmoitavuus. Unity ei vaadi korkeatasoista ohjelmointiosaamista, vaan moni asia hoituu pelimoottorin ansiosta automaattisesti. Tarkemmin Unityn toiminnoista puhutaan myöhemmin työssä.

Kuluttajien laaja kiinnostus on saanut suuret yhtiöt kiinnostumaan alasta, mutta myös huomattavasti pienemmät, usein alle 20 henkilön tiimit, ovat alkaneet kehitellä omia projektejaan ja julkaisemaan niitä maailmanlaajuisesti. Jotkin pienten tiimien julkaisut ovat joskus yltäneet jopa suurten yritysten tuotoksien tasolle. Vaikka pienet yritykset eivät välttämättä pysty kilpailemaan suurten kanssa pelialalla, ne eivät kuitenkaan välttämättä jää niiden varjoon. Pienikokoiset pelikehitysyritykset voivat menestyä käyttämällä jo olemassa olevan pelimoottoria, kuten Unityä. Tällöin koko pelin infrastruk-

tuuria ja pohjaa ei tarvitse rakentaa alusta alkaen itse, vaan kehitystyössä voi keskittyä enemmän luovuuteen, innovaatioon ja toteutukseen, sillä suuri osa komponenteista ja toiminnoista joita pelikehitys vaatii, on jo olemassa. Pienistä pelialan yrityksistä voi siis nousta esiin hyvinkin omalaatuisia ja kekseliäitä pelejä, jotka erottuvat massasta.

Tässä opinnäytetyössä on teoriaa, käytännön esimerkkejä ja omakohtaisia kokemuksia Unityn käytöstä perustuen tekijän työhön Koukoi Games Oy:llä, joka on oululainen korkealaatuisia mobiilipelejä valmistava pelialan yritys. Koukoi Games tekee yhteistyötä suuren elokuvayhtiön peliliiketoimintayksikön kanssa tuottaen animaatioelokuvaan pohjautuvaa mobiilipeliä. Työssään tekijä osallistui kyseisen peliprojektin luovaan projektityöskentelyyn ohjelmoijana ja loi uusia prototyyppejä tuotantoa varten, joista projektin idea valittiinkin. Unityn analysointi ja tutkiminen on myös suuressa roolissa työssä. Työstä käy ilmi Unityn hyvät ja huonot puolet, kuinka sitä kannattaa käyttää pelialalla ammatillisessa kehitystyössä ja mikä tekee siitä suositeltavan.

Opinnäytetyön aiheet sisältävät ensin teoriaosuuden, jossa toiminnon taustaa, perusteluja, hyötyjä, haittoja ja käyttötarkoituksia selvitetään mahdollisimman perusteellisesti. Teoriaosuuksissa käytetään yksinomaan luotettavia alan ammattilaisten lähteitä. Lähteiden luotettavuus on todettu sisällön laadun tarkastelun ja taustatutkimusten perusteella. Teoriaosuuksien jälkeen työssä usein esitellään, kuinka haluttu toiminto on toteutettavissa Unityllä, ja miten toimintoa voi hyödyntää pelikehityksessä. Toiminnan esittelyn lopussa onkin usein esimerkkiehdotuksia erilaisiin käyttötarkoituksiin.

2 UNITYN SOVELTUVUUS JA ANALYYSI

Tässä luvussa tarkastellaan Unityn käyttöä ammatillisessa ympäristössä pelialalla. Tärkeimpiä kysymyksiä, joihin luvussa haetaan vastauksia ovat, miten, miksi ja mihin Unityä tulisi soveltaa. Oikean pelimoottorin valinta projektiin on tärkeä ensiaskel missä tahansa työssä, ja tässä luvussa on tarkoitus kertoa miksi. Suurin painotus on erityisesti mobiilipelikehityksessä ja siihen liittyvissä asioissa.

2.1 Käyttö pelialalla

Unity on ohjelmisto, jonka avulla on mahdollista luoda monimutkaisiakin videopelejä ilman, että käyttäjä välttämättä edes ymmärtää kaikkia taustalla olevia teknologioita. Tämän ansiosta Unityä käyttävät kaiken tasoiset kehittäjät voivat paremmin keskittyä vain pelimekaniikan toteutukseen käyttäen korkean tason toteutusmalleja C#- tai JavaScript-ohjelmointikielillä. Korkea tason toteutusmalli tässä tapauksessa viittaa siihen, että luodessaan pelin pelimoottorilla, kehityksessä ei tarvitse huomioida, kuinka ohjelmisto mallintaa pelin tai millä tavalla se on vuorovaikutuksessa näytönohjaimen tai prosessorin kanssa pelin optimoimiseksi. Pelimoottorin sisäänrakennetut kirjastot mahdollistavat hyvin ketterän kehityksen, sillä jokaiselle projektille ei tarvitse luoda uutta pohjaa alusta alkaen. Unity on melko joustava pelimoottori ja siinä on useita tärkeitä toimintoja luotu jo valmiiksi, kuten käyttöliittymä, tekoäly ja fysiikka, minkä ansiosta ohjelmoivat voivat siirtyä varsinaiseen kehitystyöhön suoraan. Unityn graafinen käyttöliittymä nopeuttaa kehitystyötä myöskin, sillä peliä ei tarvitse koota ja rakentaa jokaisen testauksen yhteydessä, vaan peliä voi pelata suoraan Unityssä. Käyttöliittymä tarjoaa myös graafisten elementtien esikatselun. (Felicia 2015, 31.)

Pelimoottorit mahdollistavat usein pelin rakentamisen usealle eri alustalle ilman, että lähdekoodiin tarvitsee tehdä muutoksia. Unity tukee tällä hetkellä 25:ttä eri alustaa, mukaan lukien eri PC-, konsoli-, VR-, Smart TV- ja mobiilialustat. Merkittävimmät alustat ovat Android, iOS, Windows, Mac, Linux, Playstation, Xbox, Wii U sekä WebGL. Erityisesti Unityn WebGL on hyödyllinen, sillä se antaa mahdollisuuden luoda selaimessa pelattavia pelejä, jotka siten toimivat lähes kaikilla laitteilla ja se-

laimilla. Esimerkiksi Facebook tukee Unityn WebGL-muotoa. (Unity Technologies 2016a, viitattu 3.8.2016.)

Markkinoilla on useita eri pelimoottoreita, mutta Unity on todistanut olevansa yksi menestyneimmistä. Sitä on käytetty useiden suosittujen 3D- ja 2D-pelien kehityksessä. Unityä voi käyttää lähes minkä tahansa tyyppisen pelin kehitykseen, rajoituksia ei juurikaan ole. Unity sisältää kaikki tarpeelliset työkalut pelikehitystä varten ja tarjoaa myös sen ohjelmointia helpottavan tekstinmuokkaimen nimeltään Mono Develop. Mono Develop tunnistaa ja tukee Unityn sisäänrakennettujen toimintojen muokkaamista. Mono Developia ei ole pakko käyttää, sillä vaihtoehtona tekstinmuokkaimeksi löytyy myös Microsoftin Visual Studio. Unityä voi ohjelmoida Boo-, C#- ja JavaScript-kielillä, mutta kolmannen osapuolen rajapinnat mahdollistavat myös muiden skriptikielien käytön. Kielien monipuolisuus helpottaa Unityyn siirtymiseen joltain muulta alustalta. Esimerkiksi aloittelijoille JavaScript saattaa olla kielenä helpompi kuin oliopohjainen C#. (Felicia 2015, 32.)

Unity on keveähkö ja joustava 3D- tai 2D-pelikehitykseen käytettävä monialustainen pelimoottori. Se julkaistiin alun perin vuonna 2005, mutta on huomattavasti kehittynyt vuosien varrella. Ohjelman käyttöliittymä on muuttunut huomattavan vähän, mutta pelimoottorin graafiset ja toiminnalliset ominaisuudet ovat kehittyneet huimaa tahtia. Unity oli alun perin pääasiassa 3D-kehitykseen tarkoitettu moottori, mutta 4.3-versiossa siihen lisättiin myös virallinen 2D-tuki.

Unitystä on olemassa yksi ilmainen ja useita maksullisia versioita. Ilmaisversiota käytetään pääsääntöisesti opiskeluun ja harrasteluun. Se on kätevä prototypointiin ja pelikehityksen opetteluun. Unityn ilmaisversiolla on halutessa mahdollista julkaista myös kaupallisia pelejä, mutta Unity Personalin tulo- tai rahoituskatto on 100 tuhatta dollaria vuodessa. (Downie 2016, viitattu 31.7.2016.)

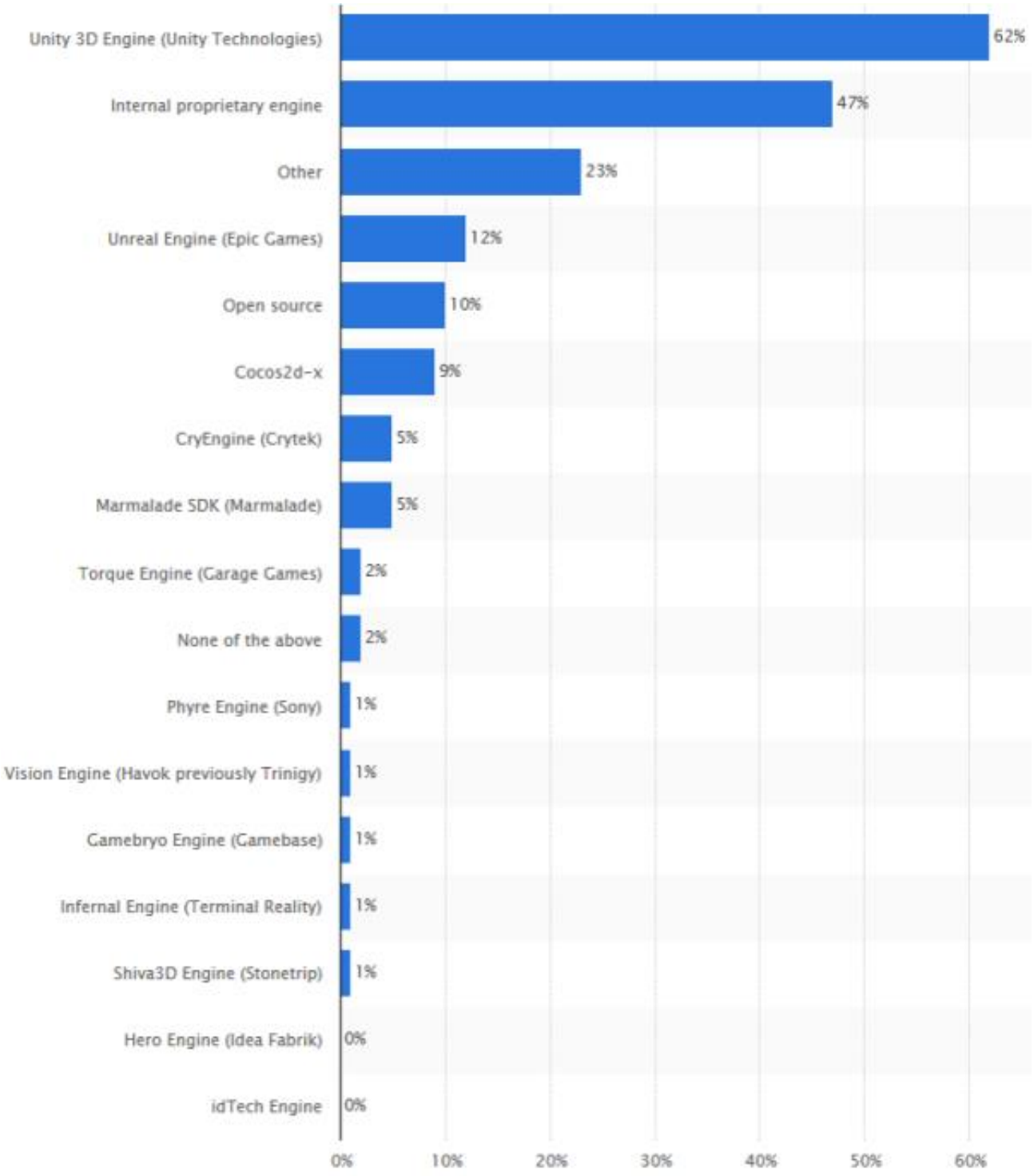
Kaupalliset versiot Unitystä ovat vuodesta 2016 lähtien olleet ainoastaan tilauspohjaisia. Sitä ennen Unity Pro -version lisenssin pystyi tilaamisen sijasta ostamaan tiettyyn hintaan. Käyttäjät voivat tilata maksullisen version vuodeksi ja maksaa joko kerran kuussa tai koko summan kerrallaan. Yrityskäyttöön ammattilaiskehittäjille voi tilata useita lisenssejä, ja Unity tarjoaakin mahdollisia räätälöityjä lisenssipaketteja suurille yrityksille. Unity mahdollistaa projektien tuonnin eri alustoille hyvin yksinker-

taisesti. Alustan vaihto tapahtuu helposti nappia painamalla, samoin kuin projektin rakentaminen pelattavaksi paketiksi. Pakettien koko on yleensä pieni, riippuen projektin koosta. (sama)

Unity on nopeasti suosiotaan kasvattava pelimoottori. Se on tällä hetkellä yksi suosituimmista mobiili-, konsoli- ja PC-pelimoottoreista, mutta se on nyt tähtäämässä myös AR (augmented reality/lisätty todellisuus) ja VR (virtual reality/virtuaalitodellisuus) -markkinoille. San Franciscossa sijaitsevaa Unity Technologies -yritystä arvioidaan 1.5 miljardin dollarin arvoiseksi. Se sai hiljattain 181 miljoonan dollarin lisärahoituksen, jonka yritys aikoo käyttää virtuaalisten ja lisättyjen todellisuuksien kehittämisratkaisuihin. Monet yritykset käyttävät jo Unityä pelien kehittämiseen markkinoiden suurimmille virtuaalitodellisuuslaseille. Maailman huippukehittäjät investoivat paljon tuodakseen uutta sisältöä markkinoille vuosina 2017 ja 2018. Unityn kehittyessä se antaa paremmat mahdollisuudet AR ja VR-applikaatioiden julkaisuun. Unityn alusta antaa kehittäjille tarvittavia rakennuspalikoita pelikehitykseen ja antaa mahdollisuuden julkaista monelle eri alustalle helposti. Unity Technologies on kasvanut lähes 1000 työntekijän yritykseksi. Sillä on 5.5 miljoonaa kehittäjää, mukaan lukien suuria ja pieniä yrityksiä. Sen suosion suuri kasvu sai alkunsa vuonna 2007, kun yritys julkaisi mahdollisuuden kehittää applikaatioita iPhonelle, hyödyntäen mobiilipelien kasvavaa suosiota tuolloin. (Haggin 2016, viitattu 23.8.2016.)

2.2 Kilpailevat pelimoottorit

Kilpailu pelialalla on kovaa, myös kehitystyössä käytettävien pelimoottorien kesken. Kehittäjien käyttämät työkalut muuttuvat ajan ja projektien luonteen mukaan jatkuvasti, mutta siitä huolimatta Unity-pelimoottori on selvästi muita edellä pelimoottorimarkkinoiden hallinnassa. Unityn osuus oli huikeat 45% pelimoottorimarkkinoista vuonna 2016, mikä on lähes kolme kertaa suurempi kuin lähin kilpailija. Unityn todennäköisin suurin kilpailija oli Unreal Engine. Alla olevassa kaaviossa (katso kuvio 1) on esitetty eri pelimoottorityyppien markkinaosuuksia vuodelta 2014. Vaikka osuudet eivät nykypäivä päde niin hyvin, ovat ne silti hyvin suuntaa antavia pelimoottorityyppien suosiosta ja monimuotoisuudesta. Suurin osa näistä pelimoottoreista ovat vielä nykyäänkin laajasti käytössä, ja tulevat lähivuosinakin olemaan. (TNW Deals 2016, viitattu 28.4.2017.)



KUVIO 1. Pelimoottorien markkinaosuuksia vuodelta 2014 (TNW Deals 2016, viitattu 28.4.2017)

Tarkkaa syytä Unityn suosioon on vaikea löytää, mutta osittain se johtuu hinnasta, laajasta alustojen tuesta, ohjelmoinnin helppoudesta ja valmiskomponenteista. Unityllä on toteutettu monia mobiilipeli-

hittejä, kuten Hearthstone: Heroes of Warcraft, Fallout Shelter ja Temple Run. Unity vetää puoleensa myös helpon opittavuutensa, aktiivisen resurssien ja lisäosien jakamisalustan (Unity Asset Store) ja nopean kehittämisnopeuden ansiosta. Vuonna 2016 Unityllä oli 4,5 miljoonaa rekisteröityä kehittäjää ja yli 15 tuhatta ilmaista tai maksullista lisäosaa tai muuta resurssia Asset Storessa. Unityn Asset Store on äärimmäisen hyödyllinen niille kehittäjille, joilla ei ole aikaa tai taitoa luoda joitain pelin osapuolia, kuten grafiikkaa, ääntä, musiikkia tai muuta, ja se nopeuttaa kehitystyötä entisestään. Kehittäjän mieltymyksen mukaan Unityä voi ohjelmoida käyttäen eri kieliä. Nämä kielet ovat C#, UnityScript ja Boo. Vuonna 2016 80,4% käyttäjiä käyttivät C#-kieltä ja 18.9% UnityScriptiä. UnityScript on JavaScriptiin perustuva kieli. C# on tehokas korkean tason ohjelmointikieli jonka hallinnan pelikehitykseen voi helposti oppia. Pelikehityksessä se on yleisesti ottaen parempi skriptikieli kuin C++, vaikka C++ on huomattavasti suorituskyvyltään tehokkaampi kieli. C++ saattaa olla hitaampi ja vaikeampi opittava. Esimerkiksi Unreal Engine käyttää C++-kieltä. C# hoitaa muistinhallinnan automaattisesti toisin kuin C++. Unityn modulaarinen komponenttijärjestelmä ja käytettävyys mahdollistavat ketterän ohjelmistokehityksen (agile, extreme programming, scrum). Virheenkorjaus Unityssä on helppoa ja nopeaa konsolin ja tarkastajan ansiosta, joissa voi tarkastella haluamiaan muuttujia, tulostuksia ja muuta informaatiota. (TNW Deals 2016, viitattu 28.4.2017.)

Suosiostaan huolimatta Unityssä kuitenkin on paljon haittapuoliakin, jotka hidastavat tai vaikeuttavat kehitystyötä. Unityyn luodut skriptit eivät toimi millään muulla alustalla kuin itsellään, sillä ne vaativat Unityn rajapintoja toimiakseen. Yleisesti ottaen Unity ei aivan yllä Unreal Enginen tasolle graafisilta toiminnoiltaan, vaikka Unityllä voikin kirjoittaa omia mukautettuja shadereita. Graafisten toimintojen takia Unreal Engine valitaan yleensä suuriin tuotantoihin, kun taas Unity pieniin nopeutensa ja helppoutensa takia. Unityn lähdekoodi on suojattu ja sitä ei voi itse muokata peliä kehittäessä, mikä estää kehittäjien pääsyn käsiksi joihinkin pelimoottorin sisäisiin ongelmiin ja virheisiin, jotka ovat korjattavissa. Unreal Enginessä ja muissa avoimen lähdekoodin pelimoottoreissa tätä ongelmaa ei ole. (sama)

2.3 Mobiilipelikehitys Unityllä

Tässä alaluvussa on tarkoitus kertoa mobiilikehityksestä Unityllä, erityisesti Android- ja iOS-alustoille. Pelien kehitys mobiilialustoille tapahtuu lähes samalla tavalla kuin mille tahansa muulle alustalle,

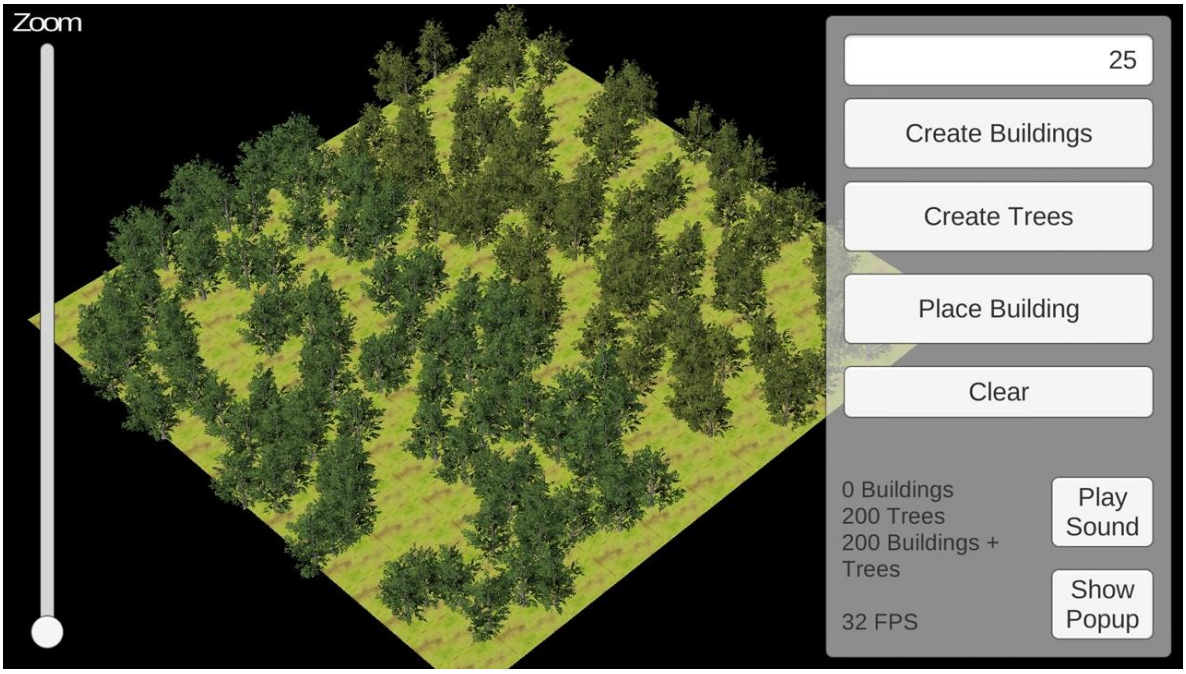
mutta kehitystyössä on monta seikkaa, jotka tulee ottaa huomioon. Mobiilikehityksessä vastaan saattaa tulla paljon yhteensopivuus- tai suorituskykyongelmia, jotka usein ovat laitekohtaisia. Sen vuoksi tässä luvussa sekä myös tulevissa käydään läpi juuri niihin liittyviä asioita. Tässä luvussa analysoidaan Unityä kehitystyökaluna mobiilialustoille ja sitä vertaillaan myös muihin kehitysmahdollisuuksiin.

Ennen uuden projektin aloittamista, erityisesti kun kyse on mobiilialustasta, on tärkeää päättää projektin vaatimukset ennen kehitystyökalun valitsemista projektia varten. Vaihtoehtoja on monta, sillä valinnan voi tehdä natiivityökalujen lisäksi oman tai olemassa olevan pelimoottorin väliltä. Oman pelimoottorin kehittäminen takaa täyden ymmärryksen sen toiminnasta, joten korjauksia ja muutoksia on helppo tehdä. Oman pelimoottorin kehittäminen vie puolestaan paljon aikaa ja resursseja, joten projektista ja kehittäjistä riippuen kannattaa usein valita jokin muu vaihtoehto. Olemassa olevia pelimoottoreita, joilla voi kehittää pelejä mobiilialustoille, on muun muassa Stencyl, GameMaker, Cocos2D, Marmalade, Unreal Engine ja Unity. Vaihtoehtoja on paljon, ja suoraa vastausta ei ole olemassa, mikä niistä on todistettavasti paras. Jokaisella kehitystyökalulla on omat hyvät ja huonot puolensa, joiden perusteella ne kannattaa valita. (Weber, viitattu 28.1.2017.)

Kehitystyökalun valintaa mobiilikehitystä varten on hyvä perustella projektin ja kehittäjien vaatimusten mukaan. Tärkeitä seikkoja valinnassa ovat opittavuus, orientaatio (2D vai 3D), lähdekoodin avoimuus, suorituskyky sekä kehityksen ja testaamisen nopeus. Pelimoottorit, jotka soveltuvat ainoastaan 2D-peleihin voi karsia pois vaihtoehdoista, jos tavoitteena on luoda 3D-grafiikkaa sisältävä peli, mutta valinta vaikeutuu jälleen, jos kyse on esimerkiksi isometrisestä kuvakulmasta kuvattu peli, joka voi hyvinkin olla joko 2D tai 3D. Valinnassa tärkeänä tekijänä on myös suorituskyvyn tutkiminen. Suorituskyky, joka mobiilikehityksessä on elintärkeää laitteistorajoitusten vuoksi, on yksi ainoista asioista, josta voi todistettavasti vetää tutkimuspohjaisia johtopäätöksiä. Havainnollistavan suorituskykytestin voi esimerkiksi tehdä luomalla lähes identtisen prototyypin usealla eri alustalla, esimerkiksi Unreal Enginellä ja Unityllä sekä tarkastelemalla niiden muistin- ja prosessorinkäyttöä mobiililaitteilla. Prototyyppien luomisen aikana on käy usein nopeasti ilmi kyseisellä alustalla kehittämisen soveltuvuus projektiin sekä sen hyvät ja huonot puolet. (sama)

Testitapauksena mobiilisuorituskyvyn mittaamiseen voimme käyttää Unityllä ja Unreal Enginellä toteutettavaa prototyyppiä, jossa on useita animoituja rakennuksia ja puita. Prototyypissä on ruudukko,

johon mahtuu 176 rakennusta, ja jokaiseen ruutuun mahtuu kuusi puuta. Siinä tulee myös olemaan käyttöliittymä, jolla voi seurata ruudunpäivitysnopeutta ja päivittää näkymän malleja. Prototyypeissä käytetään identtisiä malleja ja tekstuureja, ja prototyyppien näkymien tulee vastata mahdollisimman tarkasti toisiaan. Puut käyttävät SpeedTree:n mobiililaitteille tarkoitettua mallia, jossa on noin tuhat monikulmiota (polygonia). Suorituskyvyn lisäksi testitapauksessa on otettu huomioon myös dokumentaation saatavuus, kokoamisajat, mobiilille rakentamisen vaikeus ja ohjelmistokoodin iterointi. Tarkastellaan ensin miltä testitapaus näyttää Unityllä toteutettuna (katso kuvio 2). (sama)



KUVIO 2. Kuvassa testitapauksen prototyyppi mallinnettuna Unity-pelimoottorilla (Weber, viitattu 28.1.2017)

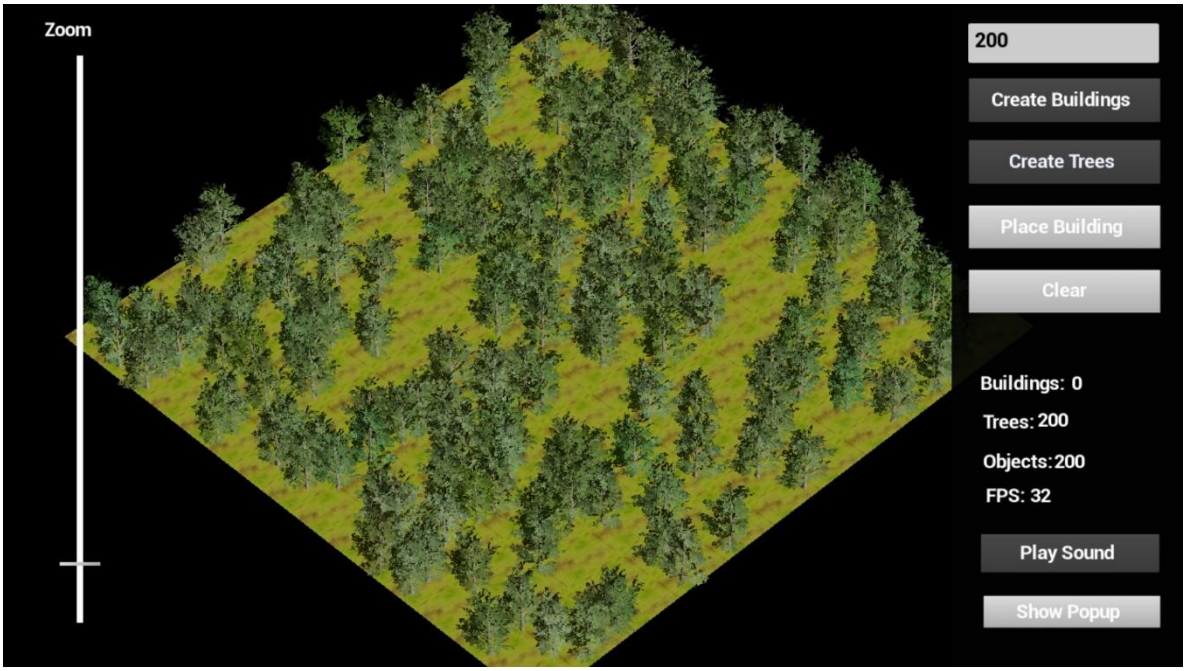
Unityssä oli useita hämmentäviä seikkoja, kuten sen statistiikat ja käyttöliittymän skaalautuminen. Sen yksi suurimmista heikkouksista on sen suljettu lähdekoodi, johon on mahdollista päästä käsiksi vain neuvottelemalla siitä hinnan. Kehitystyössä vastaan voi tulla yhteensopivuusongelmia joita on mahdoton korjata itse, ja kehittäjän on pakko odottaa, että ongelma korjaantuu seuraavan Unity-päivityksen myötä. Unityssä on myös muita ongelmia, kuten syötteenkäsittely kosketusnäytöllä. Sormenpainalluksesta ei saanut selvää osuiko se käyttöliittymäobjektiin vai ei, kuten testitapauksessa käyttäjä saattoi panoroida näkymää samalla, kun liikutti liukusäädintä. Yksi ratkaisu tähän olisi esi-

merkiksi GraphicsRaycaster-luokan laajennus, kuten alla olevassa kuvassa näkyy. Sen avulla voi tunnistaa osuuko painallus käyttöliittymään vai ei (katso kuvio 3). (Weber, viitattu 28.1.2017.)

```
public class CustomGraphicRaycaster : GraphicRaycaster
{
   public static CustomGraphicRaycaster Instance { get; private set; }

    int rayCastCount;
    public bool IsGuiBelowMouse { get { return rayCastCount > 0; } }

    protected override void Awake()
    {
        base.Awake();
        Instance = this;
    }

    public override void Raycast(PointerEventData eventData, List<RaycastResult> resultList)
    {
        base.Raycast(eventData, resultList);
        rayCastCount = resultList.Count;
    }
}
```

KUVIO 3. Kuvankaappaus laajennetusta GraphicsRaycaster-luokasta (sama)

Unity oli kuitenkin yksi ensimmäisistä julkisista pelimoottoreista, joihin implementoitiin tuki mobiilialustoille. Sen mobiilituki on hyvä ja visuaalisesti lopputulos muistuttaa hyvin paljon samankaltaista muokkaimessakin. Lähes kaikki Unityn toiminnallisuus perustuu olemuksiin (GameObject) ja komponentteihin (MonoBehaviour). Unityssä onkin mukana lähes kaikki tarpeelliset osatekijät, joita pelin kehittämiseen vaatii pelin varsinaista toiminnallisuutta lukuun ottamatta. Kehitysvaiheessa olevien prototyyppien testaus mobiililaitteilla on usein nopeaa ja vaivatonta Unityllä, millä on suuri painoarvo mobiilikehityksessä. Prototypointia nopeuttaa myös Unityn Asset Store, josta voi helposti ladata edullisia tai ilmaisia valmiita paketteja, jotka voivat sisältää muun muassa grafiikkaa, skriptejä tai lisäosia muokkaimeen. Unityllä on myös melko laaja dokumentaatio sen toiminnallisuudesta ja sen käyttäjäkunta on suuri. Suuren käyttäjämäärän ansiosta ongelmien kartoittaminen on usein helppoa. Unityllä on omien kanaviensa ja dokumentaation lisäksi myös Unity Issue Tracker, johon käyttäjät voivat lähettää kohtaamiaan ongelmia kehittäjien korjattavaksi. Seuraavaksi tarkastelemme samaa testitapausta Unreal Enginellä toteutettuna (katso kuvio 4). (Weber, viitattu 28.1.2017.)

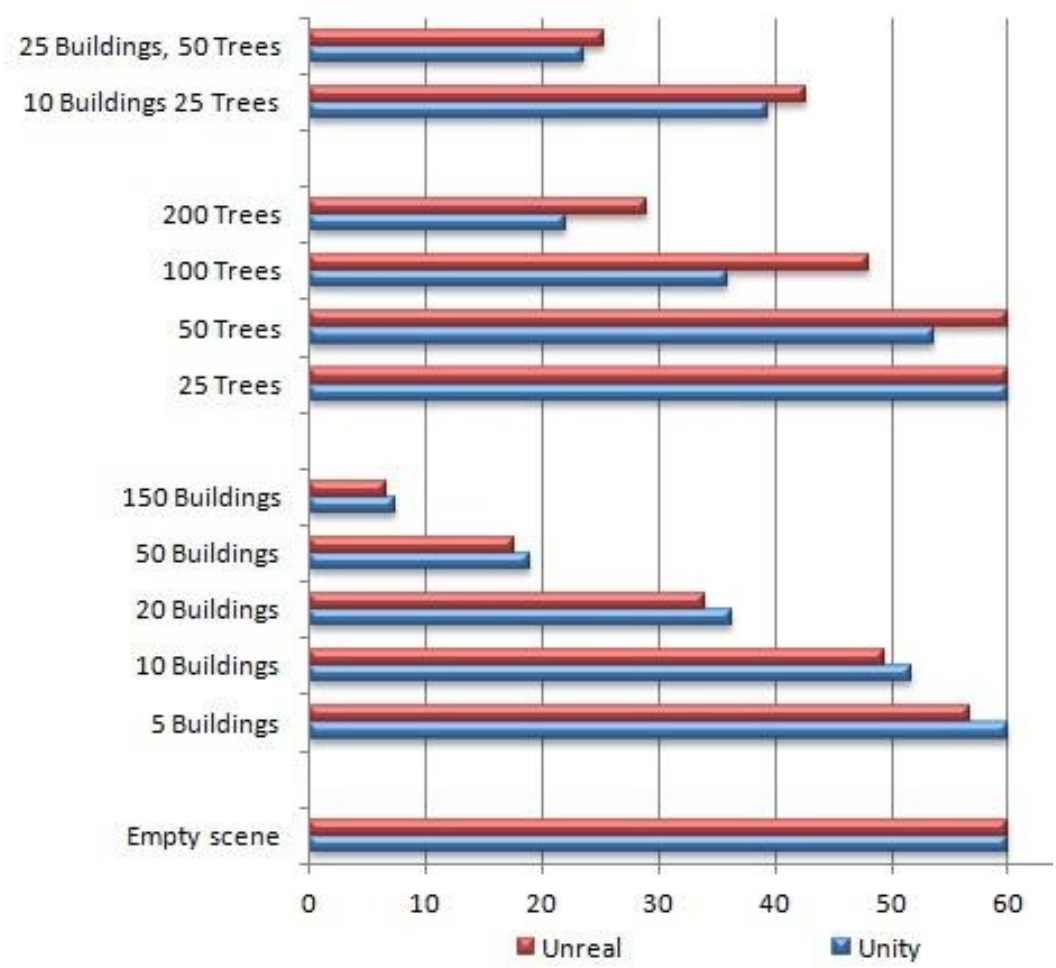


KUVIO 4. Kuvassa jälleen testitapauksen prototyyppi, tällä kertaa mallinnettu Unreal Engine-pelimoottorilla (sama)

Kahdella täysin eri työkalulla on mahdollista rakentaa yllättävän saman näköinen näkymä. Unreal Engineen totuttelu on kuitenkin melko hidasta taitavallekin ohjelmoijalle, sillä sen erinäisten makrojen ja toimintojen opetteluun kuluu paljon aikaa. Pelimoottoriin totuttelu hidastuu myös, jos käyttäjä yrittää samaan aikaan opetella Unreal Enginen käyttämää C++-ohjelmointikieltä. Unreal Enginen käyttämä visuaalinen skriptaus Blueprints saattaa myös toimintojen laajentuessa muuttua hyvin sekavaksi ja vaikeasti hallittavaksi. Blueprintsien käyttö on vaihtoehtoista, joten käyttäjä voi tahdonmukaisesti käyttää haluamaansa, mutta jotkin Unreal Enginen toiminnot, kuten sen käyttöliittymäjärjestelmä, vaatii niiden käyttämisen. Käyttöliittymäjärjestelmän Blueprintsien sotkuisuutta voi vähentää luomalla sitä varten apuluokkia. Unreal Enginen C++-dokumentaatio on hyvin puutteellista eikä siitä ole kovin paljoa apua ongelmatilanteen ilmentyessä, jättäen paljon parantamisen varaan. Mobiililaitteelle rakentaminen oli melko hidasta ja siinä saattoi ilmetä graafisia häiriöitä, kuten sumeita tekstuureja tai valaisemattomia malleja. Ongelmat olivat suuremmat iOS-laitteilla kuin Androidilla. Unreal Enginellä on kuitenkin hyvä ja erittäin laaja muokkain, ja se soveltuu graafiselta tasoltaan huippuluokan pelien kehitykseen. Unreal Enginen lähdekoodi on täysin avoin toisin kuin Unityn, mikä on tärkeä ominaisuus kehitystyökalulle. Pelimoottorin toiminnallisuutta on mahdollista muokata ja sen virheitä voi kor-

jata samalla, kun peliä kehittää. Blueprintsien kohtuullisella käytöllä on mahdollista luoda nopeasti yksinkertaista pelilogiikkaa. Niiden C++-integraatio on hyvää, mikä mahdollistaa aloittelijoiden ja ammattilaisten yhteistyön. (Weber, viitattu 28.1.2017.)

Ruudunpäivitysnopeuden mittauksissa ilmeni suuria eroja laitteistosta riippuen. Joillakin laitteilla Unreal Engine suoriutui Unityä paremmin millä tahansa kokoonpanolla, kun taas joillain Unity suoriutui paremmin paljon rakennuksia sisältävissä skenaarioissa. Yleisesti ottaen Unreal Enginellä oli parempi suorituskyky kuin Unityllä, mutta Unityn soveltuvuus mobiililaitteille ilmentyi sen visuaalisessa yhtäpitävyydessä ja laadussa. Unityn lopputulos mobiililaitteella vaikutti lähes samalta kuin muokkaimessa. Sama ei pätenyt Unreal Engineen, mikä voi osittain selittää sen graafisten vikojen syyn, erityisesti valaistuksen saamisen oikeanlaiseksi. Tulokset molemmille pelimoottoreille olivat oikein hyviä ottaen huomioon mallien kolmioiden määrän (katso kuvio 5). (sama)



KUVIO 5. Palkkikaaviossa ruudunpäivitysnopeuden mittaustulokset eri kokoonpanoissa (Weber, viitattu 28.1.2017)

Loppujen lopuksi testitapaus osoitti, kuinka lähellä nämä kaksi pelimoottoria ovat suorituskyvyssä. Se toi myös esille Unityn nopeuden ja yhtenäisyyden mobiilialustoille kehittäessä. Suorituskyvyn läheisyyden vuoksi päätös kehitystyökalun käytöstä riippuukin enemmän pelimoottorien ominaisuuksista ja kehitysnopeudesta sekä niiden hyvistä ja huonoista puolista. Unity on noin kolme kertaa nopeampi mobiililaitteille rakentamisessa kuin Unreal Engine. Molemmat pelimoottorit suoriutuivat yhtä hyvin iOS-laitteilla, kun kyse oli animoitujen mallien esityksessä. Muistinkäyttö molemmissa moottoreissa oli hyvin saman tasoista. Kahden tunnin pituisessa testissä käyttäen animoituja malleja Unreal Enginellä toteutettu versio kulutti vähemmän akkukapasiteettia kuin Unity. Testitapaus antoi hyvän yleiskuvan pelimoottorien soveltuvuudesta mobiilikehitykseen, mutta jotkin testin tulokset saattavat vanhentua muutamassa kuukaudessa tai vuodessa jatkuvan kehityksen seurauksena. Tämän testin tuloksena Unity kuitenkin soveltuu todennäköisesti parhaiten mobiilikehitykseen. Syy tähän on Unityn yhtenäinen visuaalisuus jokaisella alustalla, sen pienempi jalanjälki laitteilla, pienempi pakettikoko, helppous ja opittavuus sekä sen kehitysnopeus. On kuitenkin tärkeää ymmärtää, että paras mahdollinen kehitystyökalu saattaa olla joku muu riippuen projektin vaatimuksista. Vaatimusmäärittelyä ja tutkimusta kannattaakin tehdä ennen päätöksentekoa. (sama)

3 UNITYN TOIMINNALLISUUS

Tässä luvussa tarkastellaan Unityn tärkeimpiä toimintoja ja komponentteja, joita käyttämällä pelejä pääasiallisesti kehitetään. Toiminnallisuutta ja ominaisuuksia on tarkoitus käydä läpi pintaa syvemmin ottaen huomioon tärkeät seikat suorituskyvyn ja hyvän käytännön kannalta. Suorituskyvyn optimointi on erityisen tärkeää etenkin mobiilikehityksen kannalta.

3.1 Näkymät ja näkymähallinta

Projektit (project), näkymät (scene) ja kamerat ovat Unityssä tärkeitä peruskäsitettä, joihin koko sovelluksen ja pelin toiminnallisuus perustuu. Projekti sisältää kaikki pelissä näkyvät ja toimivat komponentit, kuten kuvat, 3D-mallit, skriptitiedostot, valmiselementit, musiikit ja äänet. Kun projektiin tuo tiedostoja, Unity automaattisesti pakkaa ne haluamaansa muotoon. Näkymät puolestaan viittaavat Unityssä tiedostoihin, jotka sisältävät grafiikkaa, kameroita, peliobjekteja ja muita tärkeitä peliin liittyviä ominaisuuksia. Unity lataa näkymät yksi kerrallaan riippuen siitä, mikä näkymä on aktiivisena. Näkymien hallinnalla on suuri merkitys erityisesti suurissa projekteissa muistihallinnan ja suorituskyvyn kannalta. Näkymähallinta (SceneManagement) on rajapintaominaisuus, joka esiteltiin Unityn versiossa 5.3. (Zucconi 2016, viitattu 23.12.2016.)

Näkymää voi ajatella yhtenä tasona pelissä, kuten niitä alun perin Unityn dokumentoinnissa referoitiinkin. Näkymiä ei kuitenkaan välttämättä käytetä pelkkinä tasoina peleissä, vaan niillä voi olla muitakin toimintoja, esimerkiksi latausruutu, päävalikko tai kehittäjän testialue. Lisäksi näkymiä voi olla ladattuna useita kerralla, mikä sekoittaa myös taso-käsitteen merkitystä. Näkymiä version 5.3 jälkeen ladata kolmella eri tavalla, sen indeksin, nimen tai tiedostopolun mukaan. Nykyisen aktiivisen näkymän haku tapahtuu Unityssä helposti kutsumalla SceneManagerin GetActiveScene-metodia. Näkymien uudelleenlatausta Unity ei tällä hetkellä tue, vaan se tapahtuu kutsumalla SceneManagerin LoadSceneä GetActiveScene parametrilla. Sekin toimii parhaiten, jos ladattuna on vain yksi näkymä kerrallaan. Siinäkin on omat ongelmansa. Aina uuden näkymän ladattaessa Unityssä, se puhdistaa kaikki muut aiemmin ladatut näkymät. Ainoastaan DontDestroyOnLoad-metodia kutsuneet objektit

säilyvät näkymästä näkymään. Se tarkoittaa, että käyttäjä voi kutsua LoadScene-metodia, vaikka muut näkymät ovat vielä ladattuina, eikä Unity pysty havaitsemaan sitä. (sama)

3.2 Peliobjektit

Peliobjekti on Unityssä hyvin tärkeä perustason käsite. Se on yläluokka kaikille objekteille Unityn näkymissä. Tyypillisiä peliobjektin arvoja ovat sijainti (x, y, z), rotaatio ja skaala. Peliobjekteihin voi kiinnittää komponentteja. Skriptejä voi Unityn käyttöliittymässä luoda helposti. Unityn terminologiassa skripti tarkoittaa usein samaa asiaa kuin komponentti. Komponentit vaikuttavat siihen, miten objekti käyttäytyy, kun ohjelmaa suorittaa. Jokaisella skriptillä tulisi ideaalisti olla vain yksi tehtävä tai toiminto, jota sen on tarkoitus suorittaa, ja sen tulisi olla loogisesti nimetty myös sen mukaisesti. Skriptin tulisi sisältää vain sen tiettyyn toimintoon liittyviä muuttujia ja metodeja. Käyttäjän luomat skriptitiedostot eivät tee mitään ennen kuin ne on lisätty Unityn näkymään ja vaativat peliobjektin toimiakseen. Optimaalista skriptauskäytäntöä on luoda skriptejä, joita voi uudelleen käyttää helposti eri objekteihin, ja skriptit, jotka ovat enimmäkseen riippumattomia muista skripteistä. Unityssä luokat toimivat samalla tavalla kuin normaalisti C#-syntaksin mukaisesti, mutta komponenttiluokat perivät Unityn MonoBehaviour-luokan. (Thorn 2015, 12.)

Valmiselementtien (prefabien) käyttö on Unityssä lähes välttämätöntä hyvän pelikehityskäytännön kannalta. Sillä tarkoitetaan peliobjektia, johon on valmiiksi asetettu tiettyjä parametreja ja komponentteja. Niitä voi luoda Unityn käyttöliittymässä helposti raahaamalla ja pudottamalla peliobjekti näkymähierarkiasta johonkin projektikansioon.

3.3 Fysiikka

Fysiikka toimii Unityssä lisäämällä siihen liittyviä komponentteja haluamiin objekteihin. Tyypillisiä komponentteja ovat osuma-alueet (collider), jäykät kappaleet (rigidbody) ja nivelet (joint). Kaikki peliobjektit muokkaamattomina luodaan oletuskerrokselle (default layer). Oletuskerroksella kaikki fysiikkaobjektit voivat törmätä toisiinsa, mikä voi olla suorituskyvyn kannalta raskasta, sillä silloin Unity

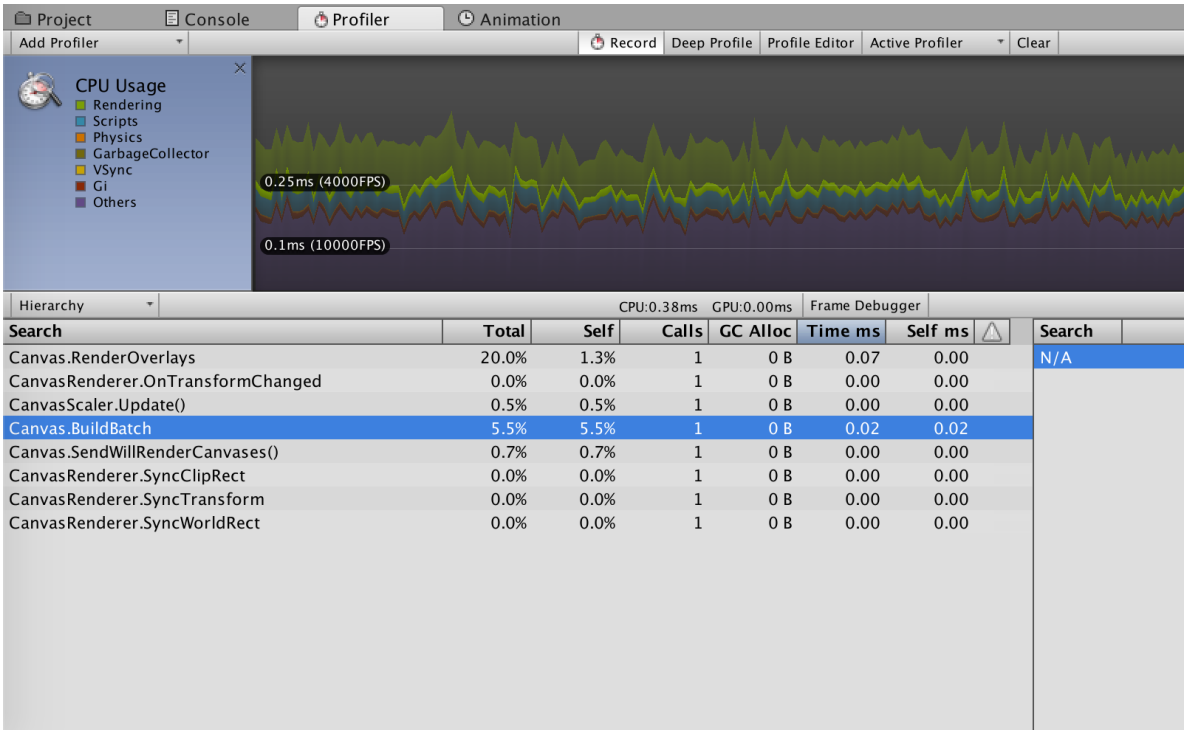
tarkistaa jokaisen objektin osuma-alueen keskenään. Hyvää käytäntöä on siis eritellä mahdollisimman tarkasti, mitkä objektit voi törmätä minkäkin kanssa. Hyvän törmäysmatriisin (layer collision matrix) määrittely voi vähentää törmäystarkistuksia jopa puolella, mikä näkyy tuntuvasti suorituskyvyssä erityisesti matalan tason laitteilla. (Unity Technologies 2015, viitattu 23.12.2016.)

3.4 Suorituskyvyn optimointi

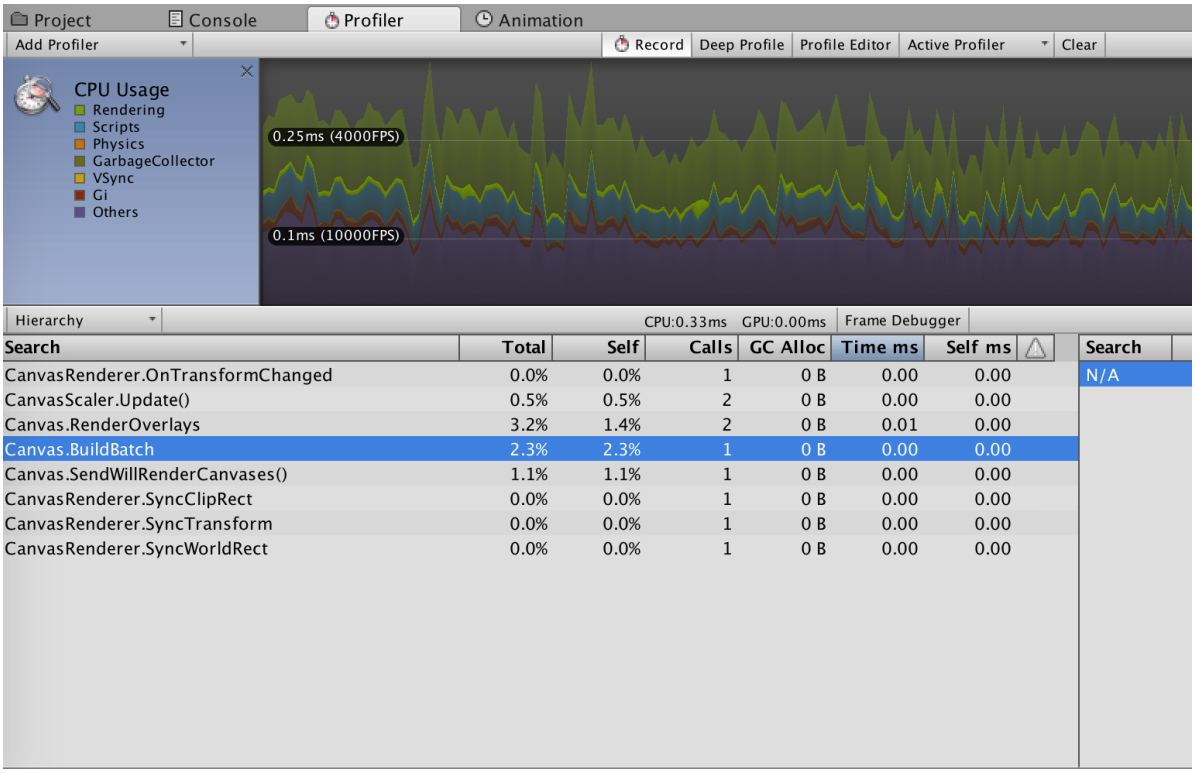
Tässä alaluvussa käydään läpi muutama hyödyllinen tekniikka jolla Unityllä toteutettuja pelejä voi optimoida. Suorituskyvyn optimointi on tärkeää erityisesti mobiilikehityksessä, sillä mobiililaitteissa on huomattavasti rajoittuneempi laitteistotehokkuus esimerkiksi pöytätietokoneisiin verrattuna. Laitteiden suhteellisen alhainen muistikapasiteetti ja prosessoritehokkuus tulee ottaa jo alusta alkaen huomioon mobiilipelejä kehittäessä, jotta kehitetyistä peleistä tulee mahdollisimman optimoituja. Hyvin optimoiduissa peleissä on pienempi tehon, muistin ja levytilan kulutus, ja ne suoriutuvat mobiililaitteella paremmin tehden pelikokemuksesta miellyttävämmän.

Käyttöliittymäjärjestelmän optimointi

Unityssä käyttöliittymäjärjestelmä on kokonaisuus, jolla on mahdollista luoda käyttöliittymiä, sillä toteutettuihin peleihin. Se käyttää niin sanottuja canvaseja joiden aliobjekteina voi olla muun muassa tekstiä, kuvia tai painettavia nappeja. Käyttöliittymäjärjestelmää on kuitenkin helppo väärinkäyttää, mikä saattaa aiheuttaa paljon suorituskykyongelmia, kuten lisääntyneet piirtokutsut. Esimerkiksi käyttöliittymäkomponenttien muuntajan (transform) sijainnin, kierron tai skaalan muuttaminen vaatii koko käyttöliittymäerän uudistamisen, mikä saattaa käydä suorituskyvylle raskaaksi, jos sitä tapahtuu jokaisella piirtokutsulla, potentiaalisesti aiheuttaen piikkejä suorituskyvyssä. Ratkaisuna tähän on jakaa käyttöliittymän objektit kahteen osaan, staattisiin ja dynaamisiin. Dynaamiselle canvasille sijoitetaan objektit joiden muuntajat saattavat muuttua ajonaikaisesti. Käyttöliittymän aiheuttamia kutsuja voi seurata Unityn profiloijalla (profiler) kuten alla on esitetty (katso kuviot 6 ja 7). Profiloijalla voi seurata canvaksen uudelleenpiirtoja Canvas.BuildBatch ja Canvas.RenderOverlay nimikkeillä. Kuvissa esitetty optimoitu ja epäoptimoitu näkymä profiloijassa. (Lopez 2017, viitattu 18.3.2017.)



KUVIO 6. Epäoptimoidun käyttöliittymän profilointi (sama)



KUVIO 7. Optimoidun käyttöliittymän profilointi (Lopez 2017, viitattu 18.3.2017)

Ylläolevissa kuvissa on käytössä samat objektit mutta optimoidussa dynaamiset objektit on siirretty omalle canvasille. Optimoinnin kannalta tulee kuitenkin ottaa huomioon, että jokainen uusi canvas luo lisää piirtokutsuja, joten paras suorituskyky löytyy tasapainottamalla canvaksen uudelleenpiirtämisiä ja uusien canvaksen luomista. Parhaan tasapainon löytää profiloimalla käyttöliittymän suorituskykyä ja suunnittelemalla sen rakenteen hyvin. (sama)

4 EDISTYNEITÄ TOIMINTOJA UNITYLLÄ

Aiheina tässä luvussa ovat tekoäly, proseduraalinen generointi, koneoppiminen, objektivarastot ja serialisointi. Tarkoituksena on käydä läpi joitain edistyneempiä tekniikoita joita Unityllä voi tehdä esimerkkien ja teorian avulla. Lisäksi tarkastelemme, mitä hyötyä niistä voisi olla pelikehityksessä. Proseduraalinen generointi ja koneoppiminen valittiin esiteltäviksi tekijän kiinnostuksen myötä. Erityisesti proseduraalista generointia hyödynnetään nykyään paljon peleissä. Koneoppimisesta tulee varmasti tärkeä osa pelien tekoälyä tulevaisuudessa. Suunnittelumallit, objektivarastot ja serialisointi sisällytettiin tähän lukuun, sillä niillä oli tärkeä merkitys projekteissa, joiden parissa tekijä työskenteli.

4.1 Suunnittelumallit

Tässä alaluvussa käsittelemme pääasiassa yksiömallia (singleton pattern) ja yleisesti miksi ja miten suunnittelumalleja (design pattern) voi Unityssä hyödyntää. Yksiömallin käyttö Unityssä säästää kehitysaikaa, mikä on todennäköisesti tärkein syy miksi sitä kannattaa käyttää. Kehitystyön aikana on turha käyttää paljon aikaa sellaisten ongelmien ratkomiseen, jotka korjautuvat helposti käyttämällä yksiömallia. Yksiömallin käyttö nopeuttaa kehitystyötä ja sopii erittäin hyvin prototypointiin. Jos sitä ei käytä, tulee lähdekoodista helposti vaikeasti hallittavaa. Yksiömalli Unityssä eroaa tyypillisistä staattisista luokista sillä, että ne esiintyvät näkymissä objekteina ja pystyvät käyttämään Unityn sisäisiä toimintoja, kuten coroutineja. Yksiömalli on eräs yksinkertaisimmista ja helppokäyttöisimmistä suunnittelumalleista, ja se koostuu yhdestä luokasta, josta luodaan ilmentymä vain kerran. Objektin säilyminen näkymästä toiseen onnistuu kutsumalla DontDestroyOnLoad-metodia. Kyseistä metodia yleensä kutsutaan Unityssä Awake-metodissa, jota objekti kutsuu vain kerran käynnistyessään. Lisäksi yksiömalli vaatii staattisen referenssin itseensä. Alla esimerkki yksinkertaisesta yksiömallin toteutuksesta (kuvio 8). (Fernandez 2016, viitattu 26.12.2016.)

```
1   using UnityEngine;
2
3   public class Singleton : MonoBehaviour {
4
5       public static Singleton Instance {
6           get;
7           private set;
8       }
9
10      private void Awake() {
11          if (!Instance) {
12              Instance = this;
13              DontDestroyOnLoad(gameObject);
14          }
15          else {
16              Destroy(gameObject);
17          }
18      }
19
20  }
21
```

KUVIO 8. Yksinkertaisen yksiömallin toteutus C#-kielellä Unityssä

Luokkaa on mahdollista kutsua Singleton.Instance -kutsulla. Luokka on nimetty Singletoniksi yksinkertaisuuden vuoksi, mutta sitä voisi tässä tapauksessa nimittää miksi tahansa. Yksiömalli ei kuitenkaan ole valmis ilman ehtolausetta sen Awake-metodissa, sillä sen tulee tarkastaa, että siitä on olemassa vain yksi ilmentymä kerrallaan. (sama)

Yksiömallia voi parannella vielä paljon. Yläpuolella esitetty esimerkki vaatii sen, että samat vaatimukset kirjoitetaan jokaisen yksiön luokkaan. Toisin sanoen, jos projektissa on tarvetta monelle yksiölle, on myös pakko kirjoittaa samat asiat jokaiseen niistä. Jos yksiön toteutukseen haluaa tehdä muutoksia, täytyy samat muutokset tehdä useaan kertaan. Siksi yksiöstä kannattaa tehdä yksi yksittäinen luokka, jonka halutut luokat perivät. Alapuolella hyvä esimerkki hieman edistyksellisemmästä yksiömallista ja sen vaatimasta MonoBehaviourExtended-luokasta (katso kuviot 9 ja 10). (Unify Community 2015, viitattu 27.12.2016.)

```
using UnityEngine;

public class Singleton<T> : MonoBehaviour where T : MonoBehaviour {
    private static T _instance;

    private static object _lock = new object();

    public static T Instance {
        get {
            if (applicationIsQuitting) {
                return null;
            }
            lock (_lock) {
                if (_instance == null) {
                    _instance = (T)FindObjectOfType(typeof(T));

                    if (FindObjectsOfType(typeof(T)).Length > 1) {
                        return _instance;
                    }

                    if (_instance == null) {
                        GameObject singleton = new GameObject();
                        _instance = singleton.AddComponent<T>();
                        singleton.name = "(singleton) " + typeof(T).ToString();
                        DontDestroyOnLoad(singleton);
                    }
                }
                return _instance;
            }
        }
    }

    private static bool applicationIsQuitting = false;
    public void OnDestroy() {
        applicationIsQuitting = true;
    }
}
```

KUVIO 9. Singleton-luokka C#-kielellä toteutettuna Unityssä

```
using UnityEngine;

static public class MethodExtensionForMonoBehaviourTransform {

    static public T GetOrAddComponent<T>(this Component child) where T : Component {
        T result = child.GetComponent<T>();
        if (result == null) {
            result = child.gameObject.AddComponent<T>();
        }
        return result;
    }

}
```

KUVIO 10. MonoBehaviourExtended-luokka C#-kielellä toteutettuna Unityssä

Kyseessä on hieman korkeatasoisempi yksiömalli, jota on tarkoitus periyttää niille luokille, joiden on tarkoitus olla yksiöitä. Kyseinen malli tarkistaa myös sen, onko sovellus sulkeutumassa vai ei. Kun Unity sulkeutuu, se tuhoaa kaikki objektit satunnaisessa järjestyksessä. Yksiön tulisi periaatteessa tuhoutua vain silloin, kun peli suljetaan. Jos yksiö on jo tuhottu pelin sulkeutuessa ja jokin muu skripti yrittää kutsua sitä, yksiö luo siitä uuden olion, ja tuo uusi olio säilyy myös Unityn käyttöliittymässä. Tämän estämiseksi yksiön tulee tarkistaa, onko se sulkeutumassa vai ei. Tämä luokka tunnistaa myös, onko objektia jo valmiiksi olemassa näkymässä vai ei. Siinä tapauksessa, että yksiötä ei valmiiksi ole näkymässä, se luo sen automaattisesti, sillä jokaisesta yksiöstä tulee olla olemassa yksi ilmentymä. Muita hyviä puolia tämän yksiön käytössä on se, että sen laajentaminen on helppoa. Yksiöihin liittyviä muutoksia tarvitsee tehdä tasan yhteen luokkaan, ja ne toimivat kaikkialla. Singleton-luokka on helppo periyttää muihin luokkiin (katso kuvio 11). (sama)

```
1   using UnityEngine;
2
3   public class Example : Singleton<Example> { // class inherits Singleton.cs
4
5       protected Example() {
6           // preventing constructor from being used
7       }
8
9       public int exampleVariable = 1; // public variables can be accessed from anywhere
10
11  }
12
```

KUVIO 11. Esimerkki luokasta, joka perii Singleton-luokan

Esimerkkiluokka perii Singleton-luokan, jolloin siitä tulee yksiömallia käyttävä luokka. Konstruktorin käyttö on luokassa estetty lisäämällä siitä suojatun version. Julkisia muuttujia on mahdollista kutsua mistä tahansa Example.Instace –kutsulla. Tässä tapauksessa halutessamme kutsua esimerkkimuuttujaa, se tapahtuisi kutsumalla Example.Instance.exampleVariable.

Vaikka yllä oleva esimerkki hyvästä yksiömallista toimii, on siinä silti paljon parantamisen varaa. Sen globaali saatavuus, helppokäyttöisyys ja nopeus tekevät siitä hyvän vaihtoehdon nopealle pelikehitykselle, mutta se sisältää myös paljon pitkän tähtäimen ongelmia ja sitä on helppo väärinkäyttää. Väärinkäytön aiheuttamia ongelmia voi olla muun muassa:

- **Korkea kytkennäisyys.** Globaaleja muuttujia kutsutaan useissa eri luokissa, tehden niistä riippuvaisia toisistaan. Luokat eivät enää toimi modulaarisesti eikä niitä voi uudelleen käyttää helposti, mikä on huonoa ohjelmointikäytäntöä, varsinkin Unityssä joka toimii komponenttipohjaisesti.
- **Periytyvyys.** Yksiömallia käyttäviä luokkia on vaikea periyttää.
- **Liiallinen käyttö.** Liian monta yksiömallia projektissa saattaa tehdä lähdekoodista vaikeasti hallittavan.
- **Testaus.** Yksiömallia käyttäviä luokkia on vaikea yksikkötestata. (Rainsberger, viitattu 27.12.2016.)

Yksiöiden väärinkäyttöä voi välttää katsomalla ongelmaa eri kulmasta. Oletetaan, että sovellus tarvitsee yhden ilmentymän luokasta ja sovellus konfiguroi sen käynnistyessään. Loogisempaa olisi, että sovellus itsessään olisi vastuussa yksiöistä eikä luokat. Sovelluksen tulisi olla yksiö eikä komponenttien. Ratkaisuna tähän voi hyödyntää yksiömalliratkaisua nimeltään Toolbox. Toolbox itsessään on yksiö, ja se hallitsee sovelluksen kaikkien komponenttien yksiöitä. Komponentit luodaan joko sovelluksen käynnistyessä tai vaihtoehtoisesti ne voi luoda myös ajon aikana. (Rainsberger, viitattu 27.12.2016.)

4.2 Proseduraalinen generointi

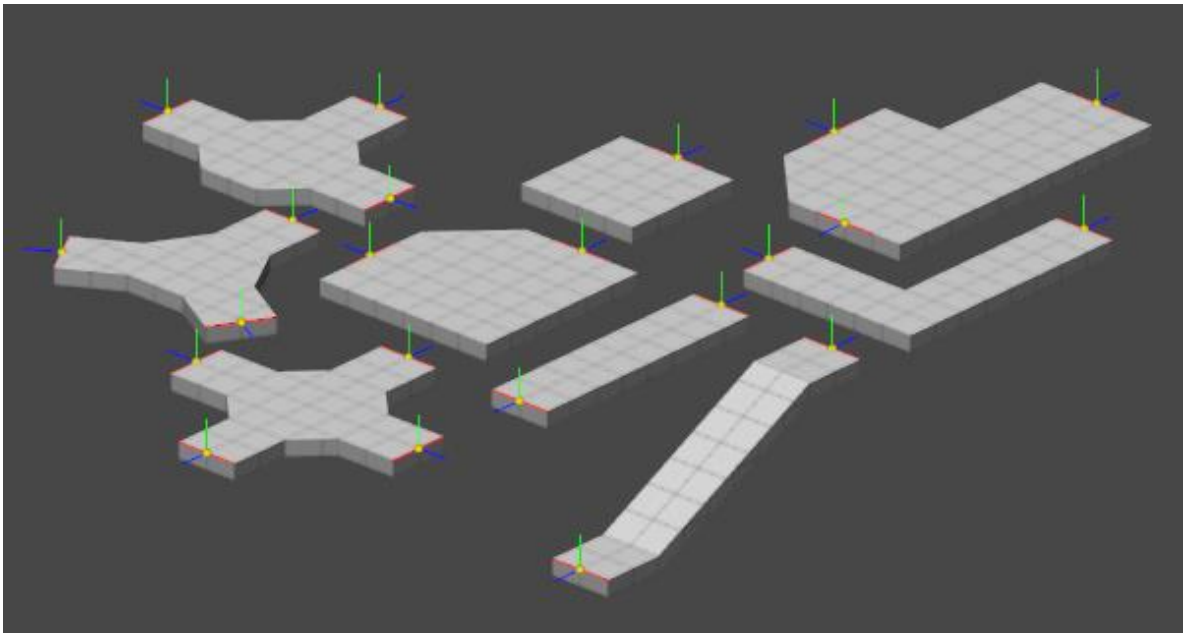
Proseduraalinen sisällön generointi (procedural content generation) on hyvin yleinen tapa luoda algoritmin avulla sisältöä peleihin. Sen avulla voi helposti luoda suuriakin määriä sisältöä peleihin, mikä puolestaan vähentää kenttäsuunnittelijoiden ja graafikoiden työmäärää. Jotkin proseduraalisen generoinnin muodot ovat yleistyneet pelialalla huomattavasti, mutta useimmat niistä soveltuvat vain tiettyihin konteksteihin tai pelielementteihin. Esimerkiksi SpeedTree-väliohjelmiston proseduraalisesti kehittämiä puita voi nähdä useissa eri peleissä. Proseduraalisen generoinnin hyötyjä arvostetaan nykyään enemmän kuin ennen. Proseduraalisen generoinnin tuomia hyötyjä ovat muun muassa:

- Sisällön nopea tuottaminen joka täyttää suunnittelijan vaatimukset
- Tuotetun sisällön monimuotoisuus, joka saattaa kasvattaa pelin uudelleenpelattavuusarvoa

- Kehitysvaiheessa suunnittelijan/yrityksen ajan ja rahan säästäminen
- Proseduraalisesti generoitu sisältö voi paremmin sopeutua pelaajan tarpeisiin (Linden, Lopez & Bidarra 2014, 1-3.)

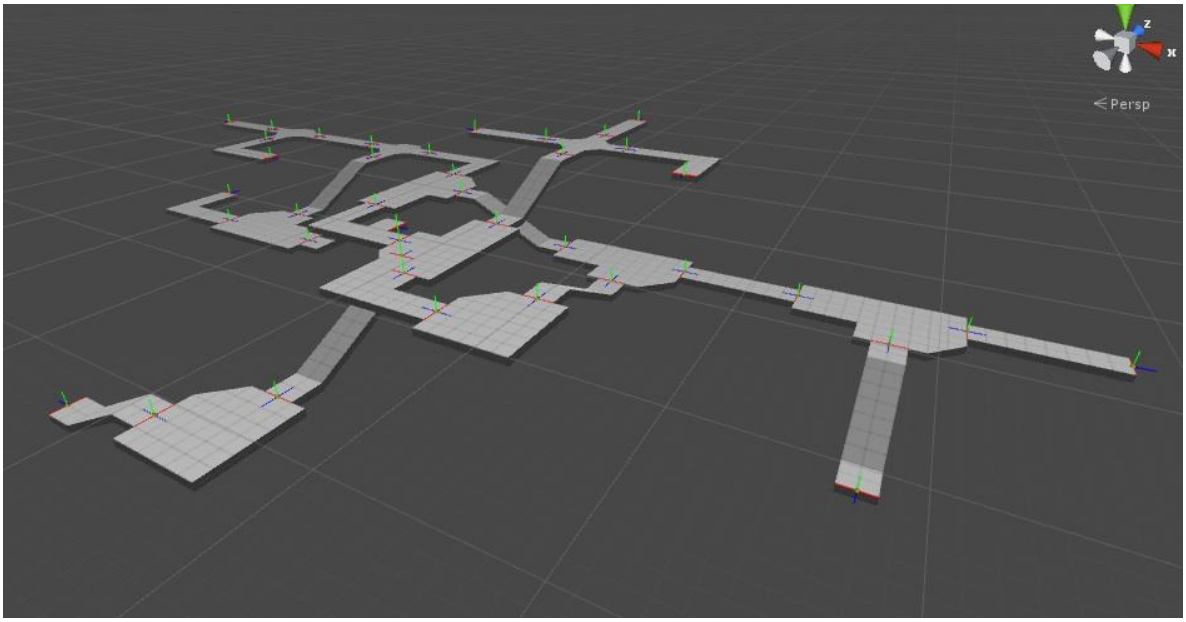
Hyvänä esimerkkinä sopivista pelityypeistä joihin proseduraalista generointia voi hyödyntää seikkailu- ja RPG-pelit, ja käytämme niiden kenttägenerointia esimerkkeinä tässä työssä, tarkemmin sanoen niiden luolastomaisia kenttätyyppejä. RPG-pelien kentät ovat usein labyrinttimaisia ja niiden ominaisuuksiin sisältyy sisäsidonnaisia haasteita, palkintoja, pulmia sekä hyvin tahdistettua ja etenemisen tunnetta tuovaa pelattavuutta. Koska proseduraalinen generointi on pohjimmiltaan kuin automaattista kenttäsuunnittelua, on siinä silti samat haasteet kuin manuaalisessakin. Suunnittelun kohteena on siis sisältöä luova algoritmi. (Linden, Lopez & Bidarra 2014, 1-3.)

Proseduraalista generointia voi toteuttaa monella eri tavalla Unityllä. Tässä työssä käytävä esimerkki on vain yksi mahdollinen toteutustapa. Proseduraalista generointia voi myös hyödyntää useaan eri käyttötarkoitukseen kenttien luomisen lisäksi. Kenttien luominen on kuitenkin yksi yleisimmistä käyttötarkoituksista proseduraaliselle generoinnille. Yksinkertaista kenttägenerointialgoritmia kehittäessä on hyvä pitää mielessä muutama sääntö kenttien luomiselle. Alueet asetetaan kentälle satunnaisesti, ja algoritmin tulee tarkistaa, että alueet ovat aseteltu kentälle oikein ilman päällekkäisyyksiä. Lisäksi kaikkien kentän alueiden tulisi olla jotenkin pelaajan tavoitettavissa. Tämän mahdollistamiseksi kenttien luomisessa voi käyttää esimerkiksi kolmea erilaista moduulia: huoneet, jotka ovat suuria alueita ja sisältävät yhden tai useamman uloskäynnin, liittymät, jotka ovat pieniä alueita ja sisältävät vähintään kolme uloskäyntiä ja käytävät, jotka yhdistävät huoneita. Moduuleja voi olla useita ja erilaisia riippuen siitä, kuinka monipuolisia kenttiä haluaa luoda. Kenttäluonnissa käytettävät moduulit voi luoda Unityssä käyttämällä Unityn valmiselementtejä, ja valmiselementit voivat esimerkiksi muistuttaa alla esitettyjä muotoja (katso kuvio 12). (Seredynski 2014, viitattu 27.5.2017.)



KUVIO 12. Esimerkkejä erilaisista moduulien muodoista, joita kenttägeneraatiossa voi käyttää (Seredynski 2014, viitattu 27.5.2017)

Moduulit toistaiseksi sisältävät vain huoneen lattian muodon. Jokaisella moduulilla on uloskäyntimerkinnät, joilla on tietty sijainti ja rotaatio. Jokainen moduuli on myös merkitty tietyn tyyppiseksi, ja sisältää listan tyypeistä, joihin se voi yhdistyä. Moduuleja yhdistetään keskenään kohdistamalla niiden uloskäynnit toisiinsa. Tämä tapahtuu siirtämällä ja kiertämällä niiden muuntajia, kunnes niiden sijainti on sama ja Z-akseli on käänteinen, ja Y-akselit ovat vastaavia. Yksinkertaisen toteutuksen voi toteuttaa Unityllä muutamaa skriptiä ja valmiselementtiä käyttäen, ja lopputulos saattaa näyttää kutakuinkin vastaavan alla esitettyä esimerkkiä (katso kuvio 13). (sama)



KUVIO 13. Esimerkki proseduraalisesti generoidusta kentästä Unityssä, joka on toteutettu yhdistämällä moduuleja (Seredynski 2014, viitattu 27.5.2017)

Yllä esitetyn esimerkin toimintoja voi parannella monella tapaa jatkokehityksellä. Moduuleihin voi lisätä algoritmien avulla generoida pelillistä sisältöä, kuten huonekaluja tai vihollisia. Esimerkissä käytetty algoritmi on hyvin yksinkertainen, sillä se ainoastaan vertailee uloskäyntien pisteitä ja asettaa moduuleja kohdilleen. Jotkin käytävät eivät johda mihinkään huoneisiin, ja joissain tapauksissa saattavat generoitua päällekkäin. Nämä ongelmakohdat tulisi ottaa huomioon kenttää luodessa. Uloskäyntien ja moduulien hallitseminen niiden määräen kasvaessa saattaa käydä hyvin työlääksi. Moduulien sijainneissa ja rotaatioissa olisi suotavaa käyttää kokonaislukuja desimaalilukujen sijaan, sillä desimaalilukujen epätarkkuus saattaa johtaa saumoihin moduulien välillä. Vaikka esimerkissä onkin paljon puutteita, antaa se kuitenkin kuvan siitä, miten kenttiä voi proseduraalisesti luoda Unityssä käyttäen valmiselementtejä. (sama)

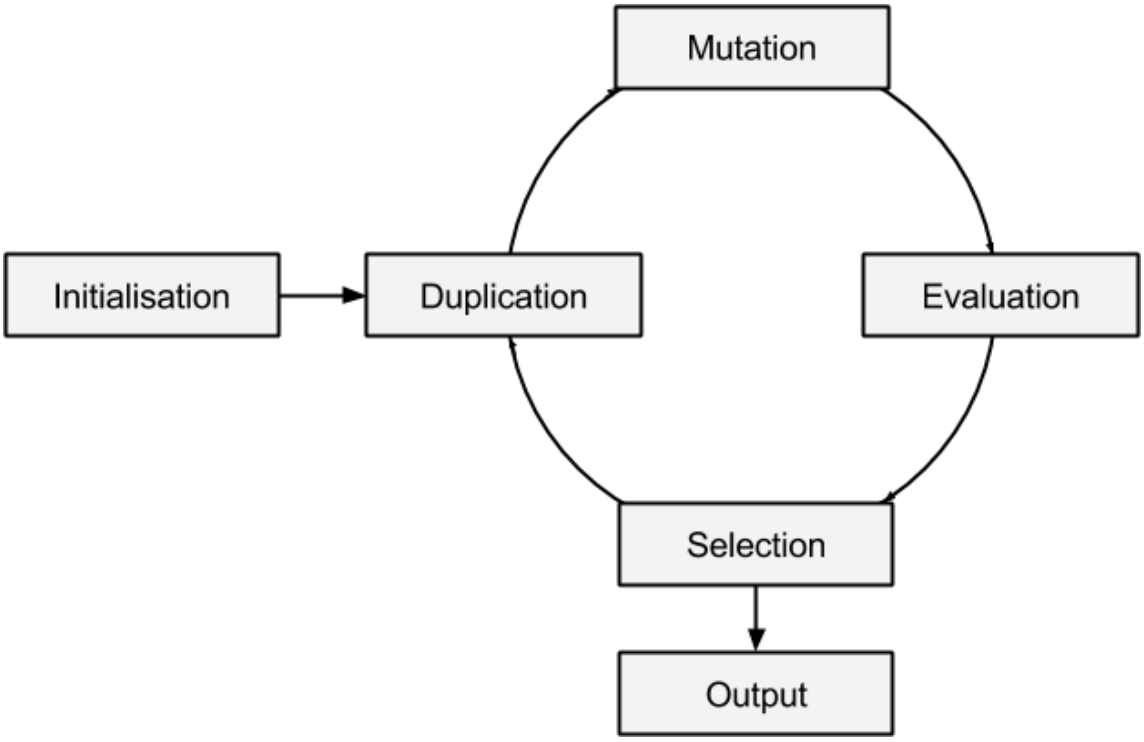
4.3 Koneoppiminen

Koneoppimistekniikoita on mahdollista toteuttaa Unityssä. Tässä alaluvussa käymme läpi, mitä koneoppiminen on, kuinka niitä voi toteuttaa Unityssä ja miksi ne ovat hyödyllisiä. Koneoppimisella (machine learning) tarkoitetaan usein tekoälytieteen alalajia. Siinä käsitellään usein Darwinin luonnonvalintaopin kaltaisia tietokonealgoritmeja. Algoritmit optimoivat itseään toistamalla samaa toimin-

toa kerta toisensa jälkeen, kunnes pystyvät suoriutumaan niistä lähes täydellisesti. Toiminnallisuuden optimointi perustuu toistojen aikana kerättyyn informaatioon. Algoritmi testaa eri mahdollisuuksia luomalla satunnaisesti uusia mutaatioita jokaista toistoa varten. Koneoppimisen tarjoamaa hyötyä voi käyttää monenlaiseen ongelmaan, jota tyypilliset algoritmit eivät helposti pysty ratkaisemaan. Esimerkiksi Google DeepMindin kehittämä AlphaGo, joka ensimmäisenä tekoälynä päihitti ammattilaisen go-lautapelissä, on koneoppimista hyödyntävä sovellus (Wikipedia 2016, viitattu 26.12.2016).

Jotta koneoppimista voisi ymmärtää paremmin, täytyy ymmärtää sen biologista toimintaa. Luonnossa jokaisella eliöllä on oma keho ja käytöksiä. Niitä kutsutaan eliön fenotyypiksi. Hiukset, silmät ja ihonväri perustuvat fenotyyppiin. Yksilön ulkomuoto määrittyy soluissa olevan informaation perusteella, jota kutsutaan genotyypiksi. Fenotyyppi määrittää miltä lopputulos tulee näyttämään valmiina, ja genotyyppi on kuin sen alkuperäinen pohjapiirustus. Ne ovat käsitteinä kaksi eri asiaa, sillä ympäristö vaikuttaa eliön lopulliseen ulkonäköön hyvin paljon. Ympäristö onkin usein se tekijä, joka määrittää genotyypin menestyksekkyyden. Biologisesti ajatellen menestys usein tarkoittaa selviytymistä. Pitkään selviytyneillä eliöillä on suurempi todennäköisyys lisääntyä, mikä tarkoittaa genotyypin periyttämistä jälkeläisille. Informaatio kehosta ja käytöksestä säilyy DNA:ssa, joka kopioituu jälkipolvelle. Satunnaisia mutaatioita voi tapahtua DNA:n kopioituessa, mitkä voi aiheuttaa muutoksia fenotyyppiin. (Zucconi 2016b, viitattu 26.12.2016.)

Kehitysopillisessa ohjelmoinnissa (evolutionary programming) algoritmissa iteroitava kohde on usein muuttumaton, mutta sen parametrit ovat optimoitavissa. Se toteutuu käyttäen samankaltaista periaatetta kuin biologinen evoluutio (katso kuvio 14).

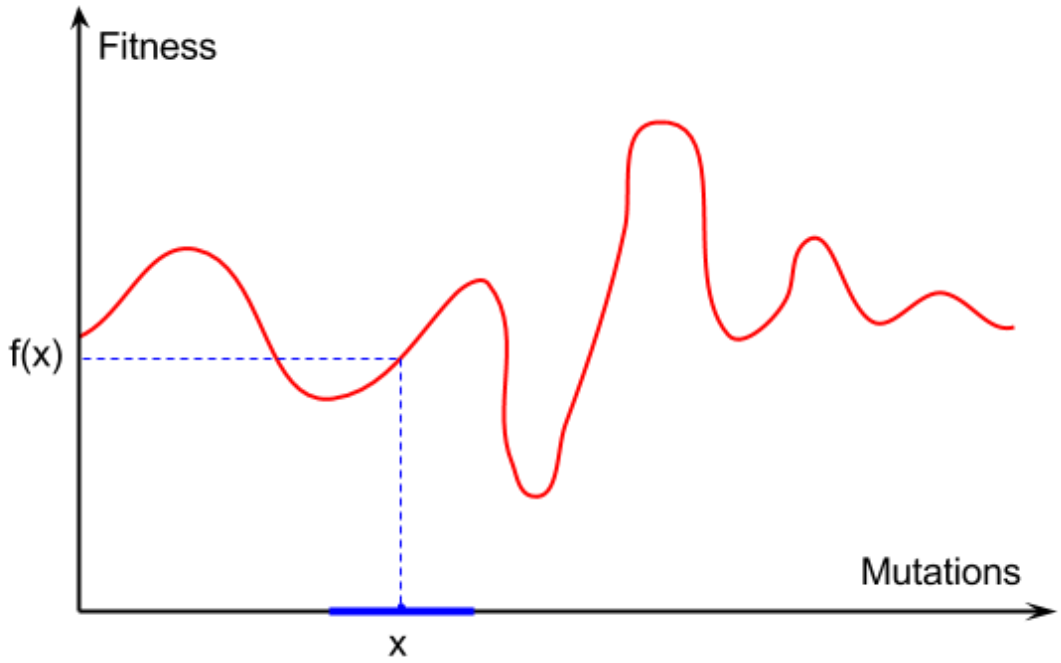


KUVIO 14. Kaaviossa esitetty iteratiivisen evoluution vaiheet (sama)

Iteratiivisen evoluution vaiheita ovat muun muassa:

- **Alustus (initialisation).** Kehitys vaatii aloitusarvoja, jotta se pystyy suorittamaan iteraatioita. Hyvien aloitusarvojen valitseminen on tärkeää, sillä eri arvot saattavat johtaa eri lopputuloksiin.
- **Monistaminen (duplication).** Kehitettävää kohdetta kopioidaan tarvittava määrä.
- **Mutaatio (mutation).** Jokaista kopiota mutatoidaan satunnaisesti. Mutaatioiden suuruus vaikuttaa kehityksen nopeuteen.
- **Arviointi (evaluation).** Kopioiden menestystä mitataan usein pistemäärällä. Pistemäärä määräytyy kopion suoriutumisen myötä, ja sen tuottaminen saattaa vaatia pitkän simulaatiovaiheen.
- **Valinta (selection).** Arvioinnin jälkeen ainoastaan parhaiten suoriutuneiden arvot kopioidaan seuraavaa sukupolvea varten.
- **Ulostulo (output).** Kehitys on iteratiivista, ja sen hetkisen parhaan ulostulon voi saada koska vain. (Zucconi 2016b, viitattu 26.12.2016.)

Kehityksen tarkoituksena on maksimoida genomin optimaalinen soveltuvuus annettuun ympäristöön. Periaate on sama kuin biologiassa. Tahdomme löytää tietylle funktiolle sen maksimaalisen pisteen. Monimutkaiselle kohteelle on lähes mahdoton löytää optimaalista ratkaisua ilman, että sitä kokeilee simulaatioilla useita kertoja peräkkäin. Siksi parhaan lopputuloksen saaminen vaatii useita kokeiluja (katso kuvio 15).



KUVIO 15. Viivadiagrammi joka esittää eri mutaatioiden synnyttämiä pistemääriä kehityksen aikana (sama)

Yllä oleva kaavio (kuvio 15) näyttää, kuinka eri mutaatiot X-akselilla johtavat eri pistemääriin Y-akselilla. Algoritmi ottaa näytteitä x-pisteen lähistöltä ja yrittää niiden perusteella kasvattaa seuraavan sukupolven pistemäärää. Maksimaalisen lopputuloksen saavuttamiseen tarvitsee lukuisia sukupolvia ottaen huomioon pisteen lähiympäristön koon. Tämä voi aiheuttaa ongelmia, sillä pisteen lähiympäristön ollessa liian pieni, sen kehitys saattaa jäädä jumiin paikalliseen maksimiin (local maxima). Ne ovat ratkaisuja, jotka ovat optimaalisia paikallisesti, mutta ei yleisesti. Paikallisten maksimien ilmentymä on hyvin yleistä soveltuvuuskäyrissä. Kehitys ei enää pysty löytämään uusia parempia ratkaisuja lähiympäristöstä ja se jää jumiin.

Tietäessämme paremmin koneoppimisen periaatteen, voimme siirtyä käytäntöön. Esimerkkinä käytämme kaksijalkaista hahmoa, jonka tarkoitus on oppia kävelemään algoritmin avulla. Hahmon keho ei tule muuttumaan, vaan ainoastaan sen tasapainotus ja kävelytoiminto. Kehitettävän hahmon valinta on tärkeää, sillä liian monimutkainen räsynukkemalli (ragdoll) vaatii enemmän aikaa opetteluun kuin yksinkertaistettu versio. Testausta varten on siis hyvä valita mahdollisimman alkeellinen räsynukke, ja siksi käytämmekin 2D-hahmoja. Räsynuken nivelissä voi käyttää Unityn joko SpringJoint2D- tai DistanceJoint2D-komponentteja. Huomioitavaa on, että DistanceJoint2D toimii ennustettavammin, joten on siten todennäköisesti parempi vaihtoehto. Kävely on jatkuvaa liikettä, joka vaatii tasapainoa molemmilta jaloilta. Alussa niveliä voi ohjata esimerkiksi käyttämällä sinimuotoisia aaltoja, jotka esittävät raajojen koukistusta ja ojennusta. Olion evoluution tarkoitus on saada raajoja ohjaavat siniaallot toimimaan siten, että se pystyy kävelemään. Aalloilla on neljä arvoa, jotka algoritmin tulee optimoida. Kun aaltojen toiminta on implementoitu, täytyy oliolle luoda jonkinlainen pisteenlaskentajärjestelmä. Teoriassa se kuulostaa helpolta, mutta se on hyvin tärkeää. Pisteenlaskennan täytyy olla mahdollisimman tarkkaa halutun lopputuloksen kannalta. (Zucconi 2016b, viitattu 26.12.2016.)

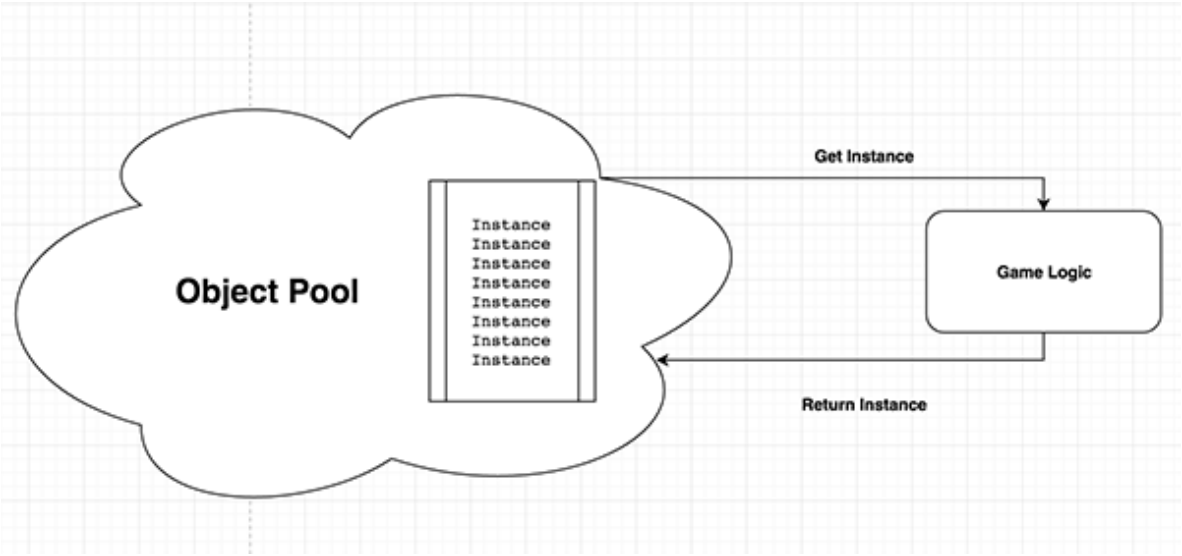
Huono pisteenlaskenta johtaa helposti heikkoihin lopputuloksiin, kun taas hyvä pisteenlaskenta tuottaa paljon parempia tuloksia. Jos kävelyn pisteitä mittaa vain matkan perusteella, algoritmi helposti löytää keinoja kiertää varsinainen kävely ja olio saattaa alkaa raahata itseään maata pitkin päästäkseen mahdollisimman pitkälle. Pistemäärän tulee siis skaalautua myös tasapainon perusteella. Liian suuri painoarvo tasapainossa saa olion pysymään paikoillaan, sillä kävellessä tasapainon menetyksen riski on korkea. Pituuden ja tasapainon pisteytyksen voikin eritellä parhaan lopputuloksen saamiseksi. Vaikka koneoppimista ei toistaiseksi ole hyödynnetty monissa moderneissa peleissä kovin laajasti, se tulee varmasti mullistamaan pelien tekoälyn toteutuksen tulevaisuudessa teknologian ja tekniikoiden kehittyessä. (Zucconi 2016b, viitattu 26.12.2016.)

4.4 Objektivarastot

Objektivarastointi (object pooling) on tärkeä muisti- ja resurssihallinan tekniikka Unityssä. Sille ei tällä hetkellä ole yleisesti käytössä olevaa tarkkaa suomennosta, mutta tässä työssä kuitenkin käytämme

nimitystä objektivarasto. Objektivarastointi tarkoittaa Unityn käyttämien peliobjektien säilyttämistä ja uusintakäyttöä mahdollisimman tehokkaasti ja suorituskyky-ystävällisesti. Varastoinnin ideana on, että objekteilla on esialustettu lista, josta objekteja alustetaan eri parametreilla tarpeen mukaan. Pelissä voi esimerkiksi olla kymmeniä tai satoja eri räjähdyksiä, efektejä, ammuksia tai muita moninaisia objekteja joita luodaan ja tuhotaan sekunnittain. Se on erittäin raskasta suorituskyvyn kannalta, ellei objektien luomista ja tuhoamista hallitse jollain tavalla. Objektivarastojen toiminnan ja tärkeyden ymmärtämiseksi täytyy ensin tutkia, kuinka objektien käsittely Unityssä tapahtuu. Tyypillisin tapa luoda uusia objekteja Unityssä on peliobjektien ilmentymien luonti. Ilmentymän luonti tapahtuu, kun ohjelmakoodi jossain vaiheessa kutsuu samannimistä metodia, jolloin haluttu peliobjekti tai valmiselementti (prefab) luodaan pelimaailmaan tiettyyn pisteeseen. Kun peliobjekti luodaan tällä tavalla, se varaa itselleen osan tyhjänä olevaa muistia. (Crook 2014, viitattu 26.12.2016.)

Myös objektien tuhoaminen vie muistia ja resursseja. Muisti pysyy varattuna, kunnes Unityn roskankerääjä (garbage collector) tyhjentää sen tiettyjen vaatimusten täyttyessä. Objektivaraston tehtävä on vähentää huomattavasti objektien luomis- ja tuohamiskutsuja. Varastossa objekteja ei koskaan tuhota, vaan ne asetetaan epäaktiivisiksi. Epäaktiivisina objektit säilyvät muistissa, eikä niiden asettaminen uudestaan aktiivisiksi aiheuta lisää muistinkulutusta. Objektivarasto-skriptit yleensä uudelleenkäyttävät vain listan epäaktiivisia jäseniä. Objekteja joita kannattaa varastoida on sellaiset, joita kontekstista riippuen tarvitsee usein (ammukset, efektit, viholliset). Varastointia suunnitellessa kannattaa miettiä, kuinka usein objekti oikeasti on aktiivisena. Jos jokin esine pelissä on aktiivisena vain kerran minuutissa tai harvemmin, sitä tuskin tarvitsee asettaa objektivarastoon. Objektivarastojen perusidea on siis kierrättää valmiiksi ilmennettyjä objekteja varastosta peliin tarpeen mukaan, kuten alla olevasta kaaviosta käy ilmi (katso kuvio 16). (sama)



KUVIO 16. Kaaviossa esitetty objektivaraston ilmentymien kierrätystä tarpeen mukaan (Banks 2016, viitattu 12.3.2017.)

Tekniikan ymmärrystä noudattaen otamme tarkasteltavaksi erään esimerkkitoteutuksen yksinkertaisesta objektivarastosta Unityssä. Objektivarastoja voi toteuttaa lukuisilla eri tavoilla ja niitä voi käyttää moneen eri tarkoitukseen, mutta tässä läpi käydään yksittäistä objektivarastoa, joka soveltuu esimerkiksi ammusten luomiseen pelissä. Ammukset ovat hyvä esimerkki objektivarastojen sovelluskohteeksi, sillä niistä tyypillisesti luodaan useita ilmentymiä lähes jokaisella ruudunpäivityksellä pelistä ja tilanteesta riippuen. Alla olevaa esimerkkivarastoa voi käyttää mihin tahansa valmiselementtiin (katso kuvio 17). (Banks 2016, viitattu 12.3.2017.)

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

public class ObjectPool {
    private GameObject prefab;
    private List<GameObject> pool;

    public ObjectPool(GameObject prefab, int initialSize) {
        this.prefab = prefab;

        this.pool = new List<GameObject>();
        for (int i = 0; i < initialSize; i++) {
            AllocateInstance();
        }
    }

    public GameObject GetInstance() {
        if (pool.Count < 1) {
            AllocateInstance();
        }

        int lastIndex = pool.Count - 1;
        GameObject instance = pool[lastIndex];
        pool.RemoveAt(lastIndex);

        instance.SetActive(true);
        return instance;
    }

    public void ReturnInstance(GameObject instance) {
        instance.SetActive(false);
        pool.Add(instance);
    }

    protected virtual GameObject AllocateInstance() {
        GameObject instance = (GameObject)GameObject.Instantiate(prefab);
        instance.SetActive(false);
        pool.Add(instance);

        return instance;
    }
}

```

KUVIO 17. Esimerkkitoteutus objektivarastosta (sama)

Objektivarasto alustetaan valmiselementillä ja halutulla aloituskoolla. Aloituskoko määrittää, kuinka monta ilmentymää luodaan alustuksen yhteydessä. Ilmentymänhaussa eli GetInstance-metodissa haetaan aina listan viimeinen elementti, sillä listasta poistaessa jonkin muun niin täytyy sen järjestellä uudelleen listan indeksit. Haun yhteydessä se poistetaan listasta ja sen objekti asetetaan aktiiviseksi. Ilmentymänpalautuksessa eli ReturnInstance-metodissa käyttötarpeensa toteuttanut ilmentymä palautetaan takaisin varastoon ja asetetaan epäaktiiviseksi. Ilmentymänluonnissa eli AllocateInstance-metodissa luodaan varsinaiset ilmentymät ja ne asetetaan heti epäaktiivisiksi, jotta ne ei ilmesty näkymään tai niitä ei päivitetä turhaan. Tämä metodi on merkitty virtuaaliseksi, jotta sen voi halutessaan uudelleen määritellä alaluokassa jotain erityistoimintaa varten. Ideana on tehdä objektivarastosta

tarpeeksi yleinen useimpiin käyttötarkoituksiin, mutta tarpeeksi joustava muokattavaksi tarpeen mukaan. On myös tärkeää muistaa pitää varaston aloituskoko mahdollisimman kohtuullisena. Ihanteellisesti objektivaraston tulisi kutsua ajonaikaisesti ilmentymänluomista mahdollisimman vähän, mutta samalla sen tulisi sisältää mahdollisimman vähän objekteja joilla on pitkä seisonta-aika. (Banks 2016, viitattu 12.3.2017.)

4.5 Serialisointi ja tietosuoja

Serialisointi (serialization) tarkoittaa Unityssä datan tallentamista johonkin tiedostomuotoon. Serialisoitua dataa voi käyttää alkuperäisen objektin uudelleenluomisessa, mikä on hyödyllistä esimerkiksi tallennuksessa pelisessioiden välillä. Serialisointi on äärimmäisen hyödyllinen tapa säilöä dataa, mutta siinä on useita huonoja puolia, kuten esimerkiksi helppo tietomurrettavuus. Tyypillisin tapa säilöä dataa Unityssä on joko PlayerPrefs tai ScriptableObject -tiedostot. PlayerPrefsit tallentuvat järjestelmärekisteriin Windows-alustoilla. Joillain ohjelmilla, kuten Regedit, joka on valmiiksi asennettu Windows-järjestelmille, pystyy tarkastelemaan ja muokkaamaan niitä. Järjestelmän rekisteriä ei pysty suojaamaan muokkaamiselta, joten käyttäjää ei pysty varsinaisesti estämään muokkaamasta tallennettuja arvoja. Koodiin pystyy kyllä lisäämään tarkistussumman (checksum), joka havaitsee arvojen muokkauksen. Tarkistussumman luomista varten täytyy luoda uusi tallennus, joka laskee yhteen kaikkien muiden tallennettujen arvojen summan. Kehittäjän tulee myös luoda jokin avainkoodi tarkistussummalle, jotta sitä ei voi väärentää. (Zucconi 2015, viitattu 22.12.2016.)

Pelidatan ja lähdekoodin suojelu on tärkeää, varsinkin suurissa kaupallisissa projekteissa. Hakkereiden estäminen online-peleissä on elintärkeää pelattavuuden säilyttämiseksi. Datan säilöminen palvelimelle on yksi tapa suojella dataa, mutta vaikeuttaa kehitystyötä huomattavasti. Kun projektin rakentaa Unityssä jollekin alustalle, se optimoi ja pakkaa lähdekoodit DLL-tiedostoiksi. Optimoituina koodit toimivat nopeammin, mutta ovat herkkiä tietomurroille. DLL-sisällön tarkastelu on helppoa, ja siihen löytyy monia ilmaisia sovelluksia, kuten ILSpy ja dotPeek. Se toimii erittäin hyvin juuri Unity-peleihin, sillä Unityn pakkaus ja optimointi on hyvin vähäistä. Unity ei myöskään piilota muuttujien nimiä, mikä tekee takaisinmallinnetun Unity-pelin lähdekoodin ymmärtämisestä hyvin helppoa. Lähdekoodia voi kuitenkin suojella joillain kolmannen osapuolen ohjelmilla, kuten Unity 3D Obfuscatorilla. (sama)

5 POHDINTA

Tässä opinnäytetyössä tarkoituksena oli tutkia ja esitellä Unityä pelikehitystyökaluna. Unity soveltuu pelikehitykseen hyvin, sillä kehitystyö on sen avulla melko nopeaa, sillä useat pelikehitykseen liittyvät toiminnot ovat pelimoottorissa jo valmiina. Lisäksi Unity on helppo oppia verrattuna useisiin muihin pelimoottoreihin tai kehitystyökaluratkaisuihin, kuten esimerkiksi Unreal Engine. Unity mahdollistaa kehityksen usealle eri alustalle samanaikaisesti, mikä on yksi sen hyödyllisimmistä ominaisuuksista pelimoottorina. Unitystä on olemassa ilmaisversio ja sen maksullisetkin versiot ovat melko edullisia joihinkin muihin vaihtoehtoihin verrattuna. Kehitysnopeus Unityllä maksaa takaisin osan hinnasta. Kehitysnopeutta edesauttaa Unityn asset store, josta voi joko maksullisesti tai ilmaiseksi saada valmista grafiikkaa tai ohjelmakoodia projekteihin. Muita pelikehitysmahdollisuuksia on hyvä harkita projektin luonteesta riippuen, mutta Unityn joustavuus tekee siitä hyvän ehdokkaan lähes jokaiseen peliprojektiin. Unityllä kuitenkin on useita haittapuolia, joista suurimpia ovat sen suljettu lähdekoodi ja heikko dokumentaatio, minkä takia jotkin kilpailijat kuten Unreal Engine saattavat olla parempia vaihtoehtoja joissain tapauksissa.

Koukoi Gamesilla tein tiivistä yhteistyötä muiden työntekijöiden kanssa uuden peliprojektin ensimmäisen version parissa ja erityisesti sen jatkokehitysvaiheessa. Projektissa oli Hollywood-tason lisensointi, joka perustui suuren animaatiostudion tuottamaan ja elokuvayhtiön julkaisemaan elokuvaan. Opinnäytetyön aiheeksi olisin voinut valita kyseisen projektin tuottamisen, mutta olin jo ehtinyt valita aiheen ja tuottaa siihen sisältöä niin paljon, että vaihtaminen ei ollut kannattavaa. Aihe kuitenkin silti liittyi vahvasti projektiin. Projektin kehittäminen oli aika ajoin hyvää tai huonoa riippuen monesta tekijästä, kuten projektin tiukasta aikataulusta. Projektin parissa työskennellessäni opin paljon, sillä siinä täytyi ottaa huomioon monta asiaa, joita omissa henkilökohtaisissa projekteissa en ollut hirveästi huomioinut, kuten lokalisointi, monetisaatio, tietosuoja, datan säilyminen, projektistruktuuri, content pipeline, serialisaatio ja suunnittelumallit. Nämä edellä mainitut asiat ovat äärimmäisen tärkeitä ammatillisissa tuotannoissa, mutta pienissä harrastelijaprojekteissa ei välttämättä yhtä paljon.

Opinnäytetyö lähti jo varhain työn alkuvaiheessa liikkeelle ajatustasolla, ja aiheen olin jo valinnut mielessäni kauan ennen työn aloittamista. Lähtötietoja minulla oli melko paljon, sillä olen kehittänyt pelejä harrastuspohjaisesti jo vuodesta 2008 lähtien, ja kokemusta Unityn käytöstä löytyy vuodesta 2010 lähtien. Asetin itse tietyt määränpäät ja vaatimukset työlle, ja mielestäni olen saavuttanut ne melko hyvin kaikki asiat huomioon ottaen. Työn aloitin etsimällä mahdollisimman paljon materiaalia sisältöä varten, ja varsinaista tuotosta alkoikin tulla vasta pitkän tutkinnan ja hyvien lähteiden etsimisen jälkeen. Aiheen laajuudesta huolimatta tarpeeksi laadukkaita lähteitä oli melko vaikea löytää noudattaen hyvää tiedonhakuprosessia, mikä on osittain ymmärrettävää aiheen suhteellisen nuoruuden ja jatkuvan kehityksen takia. Unity nimittäin nousi suureen suosioon vasta viime vuosien aikana.

Opinnäytetyötä tehdessäni ja erityisesti Koukoi Gamesilla työskennellessäni opin paljon uutta mobiilipelikehityksestä ja ohjelmoinnista. Oma-aloitteisesti uusien mobiilipeliprototyyppien luominen ja projektityöskentelyyn osallistuminen olivat hyvin mielenkiintoisia ja opettavaisia kokemuksia. Suurimpia ongelmia työtä tehdessä olivat ajan ja hyvien lähteiden puute. Joitain lukuja kirjoitusvaiheessa jouduinkin karsimaan lähteiden puutteen takia, mikä oli melko harmillista sillä ne olivat minulle kiinnostavia aiheita. Ongelmista huolimatta mielestäni työ kuitenkin kehittyi hyvin ajan mittaan, sillä tutkin ja työskentelin jatkuvasti Unityn ja pelikehityksen parissa. Jatkokehitystä työlle voisi harkita edellistäkin laajemman tutkimustyön kannalta, jossa tarkastelussa olisi uusia Unityllä toteutettavia pelikehitykseen liittyviä toimintoja. Voisin tutkia ja kirjoittaa enemmän aiheista, jotka täytyi työstä karsia, kuten Unityn muokkaimen toiminnot (editor skriptit, scriptable objektit) ja esimerkkejä Unityllä toteutetuista peleistä. Jatkotutkimuksessa voisi myös ottaa tarkemmin esille muita kilpailevia pelimoottoreita, jotka mainittiin vain lyhyesti nykyhetkellä.

LÄHTEET

Banks, K. 2016. Improve Game Performance by Implementing Object Pools in Unity3D. Viitattu 12.3.2017. https://kylewbanks.com/blog/tutorial-improve-game-performance-by-implementing-object-pools-in-unity3d

Crook, D. 2014. Object Pooling for Unity3D. Viitattu 26.12.2016. https://blogs.msdn.microsoft.com/dave_crooks_dev_blog/2014/07/21/object-pooling-for-unity3d/

Downie, C. 2016. New Unity products and prices launching soon. Viitattu 31.7.2016. http://blogs.unity3d.com/2016/05/31/new-products-and-prices/

Felicia, P. 2015. Unity 5 From Zero to Proficiency. Viitattu 23.12.2016.

Fernandez, E. C. 2016. Why You Should Start Using Singleton in Unity. Viitattu 26.12.2016. http://thedebuglog.com/2016/02/17/why-you-should-start-using-singletons-in-unity/

Haggin, P. 2016. Unity Reaches a $1.5 Billion Valuation. Viitattu 23.8.2016. http://www.wsj.com/articles/game-engine-unity-raises-181m-series-c-hits-1-5b-valuation-1468422602

Linden, R., Lopez R. & Bidarra R. 2014. Procedural generation of dungeons. Viitattu 28.12.2016. http://www.cg.its.tudelft.nl/Publications-new/2014/LLB14/Procedural.pdf

Lopez, A. 2017. Split Canvas for Dynamic Objects. Viitattu 18.3.2017. https://support.unity3d.com/hc/en-us/articles/115000355466-Split-canvas-for-dynamic-objects

Rainsberger, J. B. 2001. Use your singletons wisely. Viitattu 27.12.2016. https://www.ibm.com/developerworks/library/co-single/

Seredynski, M. 2014. Bake your own 3D dungeons with procedural recipes. Viitattu 27.5.2017. https://gamedevelopment.tutsplus.com/tutorials/bake-your-own-3d-dungeons-with-procedural-recipes--gamedev-14360

Thorn, A. 2015. Mastering Unity Scripting. Packt Publishing. Viitattu 21.12.2016.

TNW Deals 2016. This engine is dominating the gaming industry right now. Viitattu 28.4.2017. https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/

Unify Community 2015. Singleton. Viitattu 27.12.2016. http://wiki.unity3d.com/index.php/Singleton

Unity Technologies 2015. Physics Best Practices. Viitattu 23.12.2016. https://unity3d.com/learn/tutorials/topics/physics/physics-best-practices?playlist=30089

Unity Technologies 2016. Build once deploy anywhere. Viitattu 3.8.2016. https://unity3d.com/unity/multiplatform

Vähäkainu, P., Mononen, L., & Neittaanmäki, P. 2014. Suomen pelialan koulutuksen kartoitus. Jyväskylän yliopisto. Informaatioteknologian tiedekunta. Opinnäytetyö. https://www.jyu.fi/it/tutkimus/suomen_pelialan_koulutus

Weber, S. 2015. Unreal VS. Unity – Which Engine is better for Mobile Games? Viitattu 28.1.2017. http://www.makinggames.biz/feature/unreal-vs-unity-which-engine-is-better-for-mobile-games,8472.html

Wikipedia 2016. AlphaGo. Viitattu 23.12.2016. https://en.wikipedia.org/wiki/AlphaGo

Zucconi, A. 2015. A practical tutorial to hack (and protect) Unity games. Viitattu 22.12.2016. http://www.alanzucconi.com/2015/09/02/a-practical-tutorial-to-hack-and-protect-unity-games/

Zucconi, A. 2016a. Scene Management in Unity 5. Viitattu 23.12.2016. http://www.alanzucconi.com/2016/03/23/scene-management-unity-5/

Zucconi, A. 2016b. Evolutionary Computation. Viitattu 23.12.2016. http://www.alanzucconi.com/2016/04/06/evolutionary-coputation-1/