jamk.fi

Data-suuntautunut teknologia Unity-pelinkehitysmoottorissa

Tuomas Männistö

Opinnäytetyö
Elokuu 2020
Liiketalouden ala
Tradenomi (AMK), Tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Männistö, Tuomas	Opinnäytetyö, AMK	Elokuu 2020
	Sivumäärä 32	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Data-suuntautunut teknologia Unity-pelinkehitysmoottorissa

Tutkinto-ohjelma
Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja(t)
Tommi Tuikka

Toimeksiantaja(t)

Tiivistelmä

Muistin hidas kehitys ja prosessoreiden siirtyminen moniytimisiksi vaatii uusia ratkaisuja suorituskykyä vaativiin sovelluksiin, kuten videopeleihin. Tätä varten on kehitetty ohjelmointiparadigma data-suuntautunut suunnittelu, jonka perustarkoituksena on kehittää sovellus datan näkökulmasta, jotta muistin hallinta sovelluksessa olisi mahdollisimman tehokasta.

Tutkimuksen päätarkoituksena oli selvittää mikä on ohjelmointiparadigma, vertailla olio-ohjelmoinnin ja data-suuntautuneen suunnittelun eroja. Sekä tutustua Unity Technologies:n kehitysvaiheessa olevaan data-suuntautunutta suunnittelua hyödyntävään teknologiapakettiin nimeltä data-oriented technology stack eli DOTS.

Tutkimuksessa luotiin kaksi pelidemoa sekä normaalilla työnkululla että DOTS-työnkululla, joita vertailtiin keskenään toteutuksen ja suorituskyvyn näkökulmasta.

Tutkimusmenetelmänä käytettiin kvalitatiivista tutkimusta, jossa on hieman kehittämistutkimuksen ja kvantitatiivisen tutkimuksen piirteitä, sillä tutkimuksen aikana luotiin kaksi pelidemoa, joiden suorituskykyjä mitattiin ja vertailtiin.

Tuloksissa havaittiin DOTS:n tuottavan huomattavasti suoritustehokkaampaa ja uudelleenkäytettävämpää koodia, mutta vaatii enemmän koodirivejä ja tottumista uudenlaiseen tyyliin ohjelmoida. DOTS:n vajaa dokumentaatio, vaikea syntaksi ja keskeneräisyys vaikeuttivat toteutusta.

Vaikka DOTS vaikutti lupaavalta, ei sitä voitu vielä suositella suurempiin projekteihin sen keskeneräisyyden takia. Julkaistavan version tutkiminen olisi hyvä mahdollisuus jatkotutkimukselle.

Avainsanat (asiasanat)
Unity, ohjelmointiparadigma, data-suuntautunut suunnittelu, olio-ohjelmointi, muisti, prosessori, pelinkehitys, ECS, C#, DOTS

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Männistö, Tuomas	Bachelor's thesis	August 2020
		Language of publication: Finnish
	Number of pages: 32	Permission for web publication: x

Title of publication
Data-oriented technology in Unity game engine

Degree programme
Business Information Systems

Supervisor(s)
Tuikka, Tommi

Assigned by

Abstract

The slow progress of memory technology and the rise of multicore processors require alternative solutions to performance intensive software such as video games. A programming paradigm called data-oriented design has been developed to deal with this issue to program a software from the data perspective, so that the memory management of a software would be optimized.

The main of purpose of the research was to compare the differences between object-oriented programming and data-oriented design and to explore a new in development technology package developed by Unity Technologies called data-oriented technology stack, that utilizes data-oriented design.

The research was mainly implemented as a qualitative research with a hint of design research and quantitative research because the research resulted with two game demos which performances were measured and compared.

As a result of this study, DOTS was found to produce more performant and reusable code but needed more code lines to be written and a new style of coding to be learned. The lack of documentation, hard syntax and the incomplete state of the technology made the research more difficult.

Even though DOTS seemed promising, it could not yet be recommended to be used in bigger projects because it is still under development. Studying the released version could be a good opportunity for a new research.

Keywords/tags (subjects)
Unity, programming paradigm, Data-Oriented Design, object-oriented programming, memory, processor, game development, ECS, C#, DOTS

Miscellaneous (Confidential information)

Sisältö

Kuviot

Käsitteet

Unity	Unity Technologiesin kehittämä pelinkehitysmoottori
DOTS	Sanoista Data-Oriented Technology Stack. Data-suuntautunutta ohjelmointiparadigmaa hyödyntävä Unityn kehittämä teknologiapaketti.
ECS	Sanoista entity component system. Pelinkehityksessä käytetty sovellusarkkitehtuurillinen malli, jossa pelin logiikka on jaettu kolmeen eri osaan: entiteetteihin, komponentteihin ja järjestelmiin.
C#	Microsoftin kehittämä ohjelmointikieli, jota käytetään Unityssä ohjelmointikielenä.
IDE	Sanoista integrated development environment. Ohjelmointiympäristö eli ohjelma, joka sisältää työkaluja koodin tuottamiseen.
Job	Suomeksi työ. Laskennassa käytetty termi, joka kuvaa suoritettavan kokonaisuuden määrettä.
Pelinkehitysmoottori	Ohjelmisto, jolla tuotetaan pelejä.
Preview-paketti	Kehitysvaiheessa oleva paketti Unityn pakettijärjestelmässä.
tag	Kevyt merkintätyökalu entiteeteille

1 Johdanto

Videopelit ovat paljon dataa vaativia interaktiivisia ohjelmia, jotka viihdyttävät monen ikäisiä ihmisiä ympäri maailmaa. Pelit ja niiden käyttämät tekniikat kehittyvät, joka vaatii yhä optimaalisempien ratkaisujen keksimistä. Perinteisten ohjelmointimallien käyttö ei ole enää tarpeeksi tehokasta, joten vaihtoehdoksi täytyy löytää muu tapa toimia, jotta pelit vastaavat nykyteknologian vaatimuksia. Muistin hidas kehittyminen ja prosessoreiden kehitys moniytimisiksi vaatii muistin optimoimista ja prosessorin rinnakkaissuorittamista.

2 Tutkimusasetelma

Prosessoreiden suorituskyky on kehittynyt viime vuosikymmenten aikana huomattavasti nopeammin kuin muistin suorituskyky (Hennessy & Patterson 2012, 73). Tämän vuoksi muistin käytön optimointi tulee yhä tärkeämmäksi kehitettäessä prosessointitehoa vaativia ohjelmistoja, kuten esimerkiksi videopelejä. Tätä varten voidaan hyödyntää ohjelmointiparadigmaa nimeltä data-suuntautunut suunnittelu, jolla pyritään nostamaan ohjelmistojen suorituskykyä suunnittelemalla ohjelmisto datan näkökulmasta.

Unity Technologies on kehittämässä Unity-pelinkehitysmoottoriinsa data-suuntautunutta suunnittelua hyödyntävää teknologiapakettia data-oriented technology stack eli DOTS, joka on saatavilla preview-pakettina Unity:n pakettijärjestelmästä. Teknologian tutkiminen auttaa ymmärtämään sen hyötyjä, haasteita sekä käyttötarkoituksia.

2.1 Opinnäytetyön tavoitteet ja rajaukset

Opinnäytetyön tavoitteena on tutustua data-suuntautuneeseen suunnitteluun sekä vertailla sen eroja olio-ohjelmointiin ja kehittää yksinkertainen pelidemo hyödyntäen DOTS:a. Tutkimuksessa luodaan Unity:lla molempia tapoja hyödyntäen kaksi

yksinkertaista pelidemoa. Opinnäytetyö rajataan käsittelemään kehitysvaiheessa olevaa DOTS-teknologiapakettia.

2.2 Tutkimusmenetelmät

Tutkimusmenetelmänä käytetään pääosin kvalitatiivista tutkimusmenetelmää, jonka avulla voidaan määrittää ohjelmointiparadigmat sekä vertailla niiden eroja. Tutkimuksessa ilmenee myös kvatitatiivisen tutkimuksen piirteitä suorituskyvyn mittausten takia ja kehittämistutkimuksen piirteitä, koska tutkimuksen aikana kehitetään kaksi pelidemoa.

Aineistona käytetään alan kirjallisuutta, nettilähteitä sekä teknisiä dokumentaatioita. Tutkimukseen liittyvien pelidemojen toteuttamisessa hyödynnetään Unity:n virallisia dokumentaatioita, jotta teknologiaa käytettäisiin sille tarkoitetulla tavalla luotettavien tuloksien saavuttamiseksi.

2.3 Tutkimuskysymykset

Koska opinnäytetyössä vertaillaan kahta ohjelmointiparadigmaa keskenään sekä tutkitaan Unityn data-suuntautunutta teknologiapakettia, tutkimuskysymyksiksi muodostuivat:

- Mitä tarkoittaa Data-suuntautunut suunnittelu?
- Miten Data-suuntautunut suunnittelu eroaa olio-ohjelmoinnista?
- Miten kehitetään peli DOTS:n avulla ja mitkä ovat tämän teknologian edut?

3 Ohjelmointiparadigmat

Tässä osiossa on tarkoitus selvittää mitä ovat ohjelmointiparadigma, olio-ohjelmointi sekä data-suuntautunut ohjelmointi.

3.1 Mikä on ohjelmointiparadigma?

Ohjelmointiparadigma on tiettyjä periaatteita noudattava tapa ohjelmoida (Van Roy 2009, 10; Nørmark 2014). Tunnetuimmat ohjelmointiparadigmat on jaettu kahteen pääosaan: imperatiivisiin ja deklaratiivisiin. Imperatiivisissa paradigmoissa ohjelmoija kertoo koneelle kuinka muuttaa sen tilaa, mutta deklaratiivisessa ohjelmoija esittää tarpeet haluttuun lopputulokseen eikä ota kantaa siihen kuinka lopputulos tuotetaan. (Microsoft Corporation 2015)

3.2 Olio-ohjelmointi

Olio-ohjelmoinnin ensimmäisiä piirteitä on nähty jo 60-luvulta alkaen Simula nimisessä ohjelmointikielessä (Gabbrielli & Martini 2010, 277; Weisfeld 2009, 5). Simula ohjelmointikielen tarkoituksena oli yhdistää järjestelmän kuvaus ja simulaatio-ohjelmointi (Nygaard & Dahl 1978, 246), tästä syntyi ensimmäisiä olio-ohjelmoinnille tyypillisiä ominaisuuksia. Olio-ohjelmointi kategorisoidaan imperatiivisiin ohjelmointikieliin (Microsoft Corporation 2015). Seuraavissa kappaleissa esitellään olio-ohjelmoinnin tyypillisimmät ominaisuudet.

Oliot

Oliot ovat olio-ohjelmoinnin pääosa (Oracle Corporation, What Is an Object?; Gabbrielli & Martini 2010, 282). Järjestelmän toiminta perustuu eri osien jakamiseen olioihin ja niiden välisiin kanssakäymisiin. Olioita voidaan verrata oikean elämän konsepteihin, kuten esimerkiksi autoon. Autolla voi olla tämän hetkisenä tilana nopeus, ratin asento ja renkaan ilmanpaine, joita voidaan muokata toiminnoilla nosta nopeutta, käännä rattia ja säädä ilmanpainetta.

Jotta jotain voi rakentaa, tarvitaan sitä varten jonkinlainen pohjapiirustus. Olio koostuu luokasta, joka on ikään kuin pohjapiirustus jostakin asiasta. Olioita voidaan ajatella pohjapiirrustuksista rakennettuina ilmeentyminä, joilla on omat ominaisuudet. (Oracle Corporation, What Is a Class?) Esimerkiksi luokka koira voi sisältää ominaisuudet nimi, rotu ja ikä sekä toiminnallisuuden hauku. Tästä luokasta

voidaan luoda olio, jolle voidaan asettaa haluttu nimi, rotu ja ikä. Tällöin luokasta on luotu ilmentymä eli olio, jolla on omat ominaisuudet.

Kapselointi

Kapselointi tarkoittaa samaan asiaan liittyvän datan ja toiminnallisuuden ryhmittämistä yhdeksi kokonaisuudeksi sekä datan piilottamista. C#- kielessä ja monessa muussa olio-ohjelmointipohjaisessa kielessä tämä saavutetaan luokalla, joka sisältää sekä ominaisuudet että toiminnallisuudet. Tämä helpottaa hahmottamaan koodikokonaisuuksia ja niiden toimintaa. (Microsoft Corporation 2020; Weisfeld 2009, 19.)

Datan piilottamisella pyritään suojaamaan olion sisäistä dataa kutsuvilta luokilta. Luokka voi sisältää olion ominaisuuksia muokkaavan metodin, jota ulkopuolinen luokka kutsuu, mutta ei näe mitä metodi tekee sisäisesti. Tällä tavoin pidetään huolta siitä, etteivät sovelluksen muut osat sekoita olion tilaa vahingossa. Monissa olio-ohjelmointikielissä datan ja metodien näkyvyydet määritellään näkyvyysmääreillä, joita ovat private, public ja protected. Private-määre näkyy vain luokan sisälle, public näkyy kaikkialle ja protected vain luokan sisäisesti ja perivälle luokalle. (mp.)

Periminen, rajapinnat ja abstraktit luokat

Perinnän avulla voidaan luoda uusi luokka, joka perii toiselta luokalta ominaisuuksia ja toimintoja (Microsoft Corporation 2018). Tämän avulla voidaan uudelleenkäyttää valmiita toteutuksia tuottamaan uusia kokonaisuuksia. Esimerkiksi luokka koira voi periä ominaisuudet luokalta eläin.

Rajapintojen avulla voidaan antaa luokalle jokin haluttu toiminnallisuus. Rajapinnan toteuttavan luokan täytyy sisältää rajapinnassa määritelty toiminto. Tämän avulla voidaan määritellä luokalle haluttu toiminnallisuus, jolloin rajapinnan metodeja voidaan kutsua rajapinnan kautta välittämättä toteuttavan luokan tyypistä. Abstraktissa luokassa on sama idea kuin rajapinnassa, mutta toimintojen lisäksi voidaan asettaa toteuttavalle luokalle myös halutut ominaisuudet. (Oracle Corporation, What Is an Interface?) Esimerkiksi luokka eläin voi toteuttaa rajapinnan

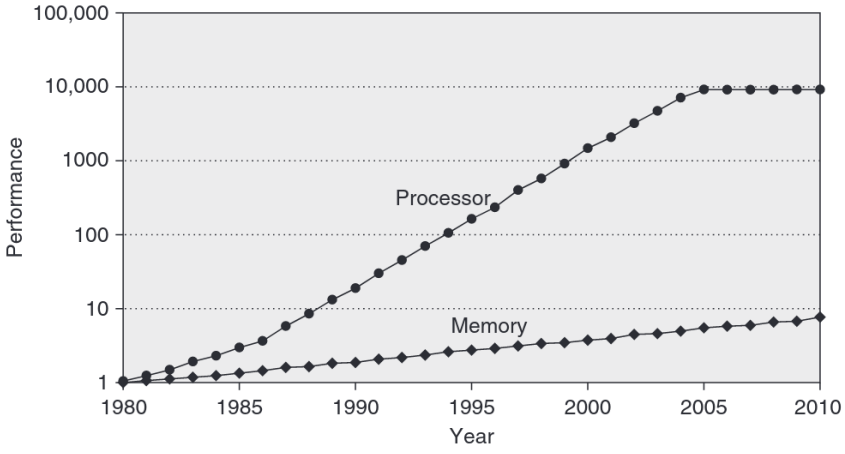
ääntelevä, joka sisältää metodin ääntele. Ohjelman ei tarvitse tietää onko kyse luokasta koira vai ihminen kutsuakseen ääntele-metodia.

3.3 Data-suuntautunut ohjelmointi

Data-suuntautuneen ohjelmoinnin käytäntöjä on hyödynnetty jo pitkään, mutta paradigma sai virallisesti nimensä vuonna 2009 Noel Llopisin kirjoittamassa artikkelissa Data-oriented design (Fabian 2018, luku 1). Llopisin (2009) mukaan data-suuntautunut suunnittelu voisi olla ratkaisu olio-ohjelmoinnin aiheuttamiin pelinkehityksessä yleisesti ilmeneviin ongelmiin, joista suurin on muistin epäoptimaalinen käyttö.

Muisti

Prosessoreiden nopeuden noustessa dataa voidaan prosessoida nopeammin, mutta prosessoitavan datan hakeminen ei ole nopeutunut suhteessa prosessointitehoon (Hennessy & Patterson 2015, 73). Tästä voidaan päätellä, että muistin käytön suunnitteleminen on yhä tärkeämpää paremman suorituskyvyn saavuttamiseksi.



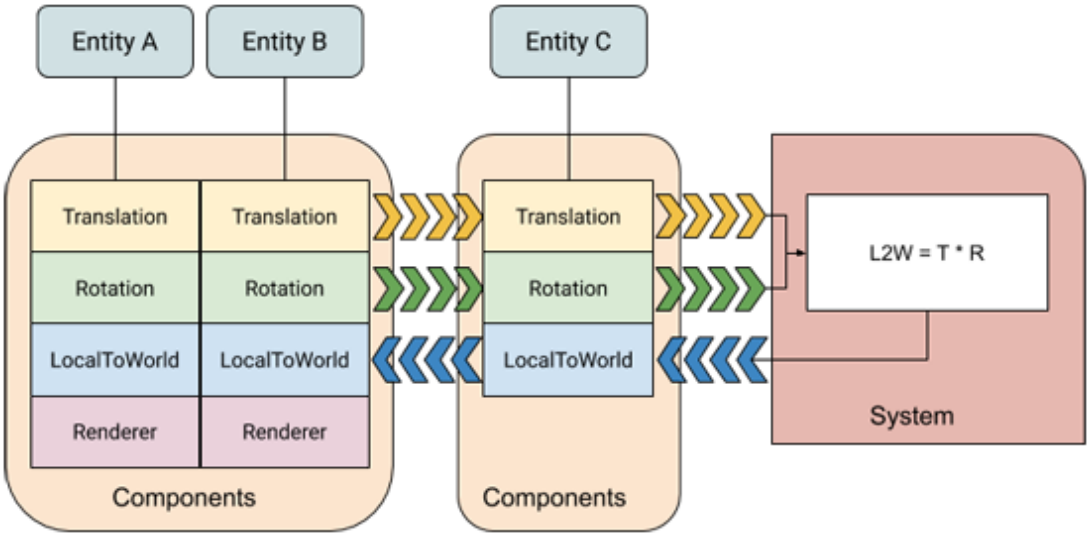
Kuvio 1. prosessorin ja muistin suorituskykyjen kehitys vuosien aikana (Hennessy & Patterson 2015, 73.)

Jotta tiedon haku olisi mahdollisimman vaivatonta, muistin tila pitää täyttää niin, että se koostuisi jatkuvasta lohkosta, jossa samankaltainen data olisi ryhmitelty vierekkäin. (Llopis 2009)

Entity component system (entiteetti komponentti järjestelmä)

Entity component system eli ECS, suomeksi entiteetti komponentti järjestelmä, on pelinkehityksessä käytetty arkkitehtuurillinen malli, jossa luokan perimisen sijaan jaetaan koodi useampaan eri komponenttiin, joiden yhteistoiminta tuottaa halutun lopputuloksen. Tällä tavoin vähennetään syviä perimärakenteita ja epäselviä tapauksia, joissa objekti tarvitsisi toiminnallisuuksia useammasta perimähaarasta. (Fabian, R. 2018)

ECS noudattaa data-suuntautuneen ohjelmoinnin periaatteita, jossa data ja toiminnallisuus eriytetään toisistaan sekä järjestellään data muistiin tiiviiseen muotoon, jotta muistin tila käytettäisiin mahdollisimman optimaalisesti. Pelissä jokainen objekti on entiteetti, joka koostuu useammasta dataa sisältävästä komponentista, joita muokataan järjestelmillä. (adam 2007)



Kuvio 2. ECS-osien yhteistoiminta, jossa entiteettien LocalToWorld-komponentti saa uuden arvo, kun järjestelmälle syötetään Translation- ja Rotation-komponenttien arvot. (Unity Technologies 2020, ECS concepts.)

Muita ominaisuuksia

Data-suuntautuneessa ohjelmoinnissa ei ole kuitenkaan kyse pelkästään datan haun tehokkuudesta, vaan myös järjestelmän muutoksiin reagoimisesta. Järjestelmät muuttuvat usein alkuperäisestä suunnitelmasta, jolloin muutoksiin pitää pystyä

reagoimaan nopeasti. Tämän takia paradigmassa pyritään keskittymään käsiteltävään dataan ja tuottamaan haluttua dataa tietämättä oikeassa maailmassa käsiteltävistä konsepteista datan ympärillä. Tätä kautta muutoksia on helpompi tehdä, koska koodia ei ole tiukasti sidottu oikean elämän konsepteihin. (Fabian 2018, luku 1.)

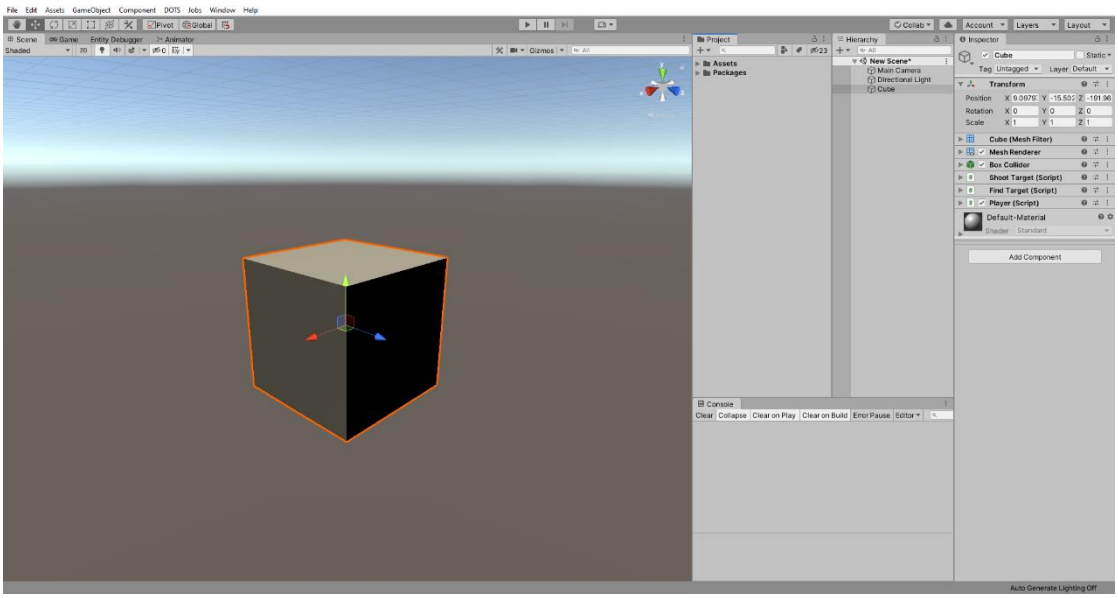
Koska modernit prosessorit ovat kehittyneet moniytimisiksi, täytyy sitä varten hyödyntää rinnakkaissuorittamista. Data-suuntautuneessa suunnittelussa rinnakkaisuus helpottuu, koska tämän paradigman menetelmillä tehdyt toiminnalisuudet on jaettu pieniin itsenäisiin funktioihin, joihin syötetään dataa, jonka jälkeen tulokseksi saadaan dataa. Tällaisia operaatioita on helppo jakaa useammalle säikeelle laskettavaksi. Samasta syystä toteutuksista tulee modulaarisempia ja testaaminen helpottuu. (Llopis 2009)

4 Unity

Unity on Unity Technologies:n kehittämä suureen suosioon noussut pelinkehitysmoottori, joka on julkaistu ensimmäisen kerran vuonna 2005. Moottorin avulla voidaan luoda sekä kaksiulotteisia että kolmiulotteisia pelejä usealle eri alustalle (Unity Technologies 2020, Unity Platform). Unityn normaali työnkulku perustuu scenejen, GameObjectien, prefabien ja MonoBehaviour-pohjaluokan perivien scriptien käsittelyyn.

Unity:ssa tehty peli koostuu yhdestä tai useammasta scenestä. Scene voidaan ajatella pelissä olevana tasona, joka sisältää tasolle ominaisen sisällön, kuten ryhmän peliobjekteja eli GameObjecteja. Normaali työnkulku perustuu vahvasti GameObjectien käsittelyyn, jossa haluttu toiminnallisuus asetetaan aina GameObjectille. Unity tarjoaa useita valmiita komponentteja, joiden avulla voidaan asettaa GameObjectille haluttu toiminnallisuus ja kun toiminnallisuuksia halutaan hallita, täytyy käyttää MonoBehaviour-pohjaluokkaa perivää scriptiä. Normaalisti GameObject on sidottu johonkin tiettyyn sceneen, mutta GameObjectista voidaan luoda uudelleenkäytettävä objekti eli prefab. (Unity Technologies 2017, Scenes; Unity Technologies 2017, GameObjects; Unity Technologies 2018, Prefabs)

Script on C#-ohjelmointikieltä käyttävä tiedosto, joka sisältää MonoBehaviour-pohjaluokan perivän luokan. MonoBehaviour tarjoaa useita pelilogiikan toteuttamiseen tarvittavia metodeja, joista yleisimmät ovat ajon alussa suoritettava Start-metodi ja ajonaikaisesti jokaisella päivityshetkellä suoritettava Update-metodi. Jotta näitä scriptejä voidaan suorittaa, ne on asetettava jollekin scenessä olevalle GameObjectille. (Unity Technologies 2018, Creating And Using Scripts; Unity Technologies 2020, MonoBehaviour)



Kuvio 3. Unityn editorin näkymä, joka sisältää sekä komponentteja että scriptejä sisältävän GameObjectin nimeltä Cube.

Data-oriented technology stack on Unityn kehittämä data-suuntautunutta suunnittelua hyödyntävä teknologiapaketti, joka koostuu kolmesta pääosasta: Entities, C# job system ja burst compiler.

Entities sisältää Unityn toteutuksen luvussa 3.3 kuvatusta entiteetti komponentti järjestelmästä. Paketti sisältää peruselementit entiteettien, järjestelmien ja komponenttien hallintaan. (Unity Technologies 2020, entities)

C# job system on monisäikeistystä hyödyntävä teknologia, jolla voidaan hyödyntää modernien prosessoreiden monisäikeisiä ominaisuuksia mahdollistaen useamman toiminnallisuuden ajamisen rinnakkain suorituskyvyn parantamiseksi. Normaalisti Unity suorittaa komentoja yksikerrallaan pääsäikeellä, mutta job system jakaa

komentoja prosessorien eri ytimille. (Unity Technologies 2018, What Is a Job System?)

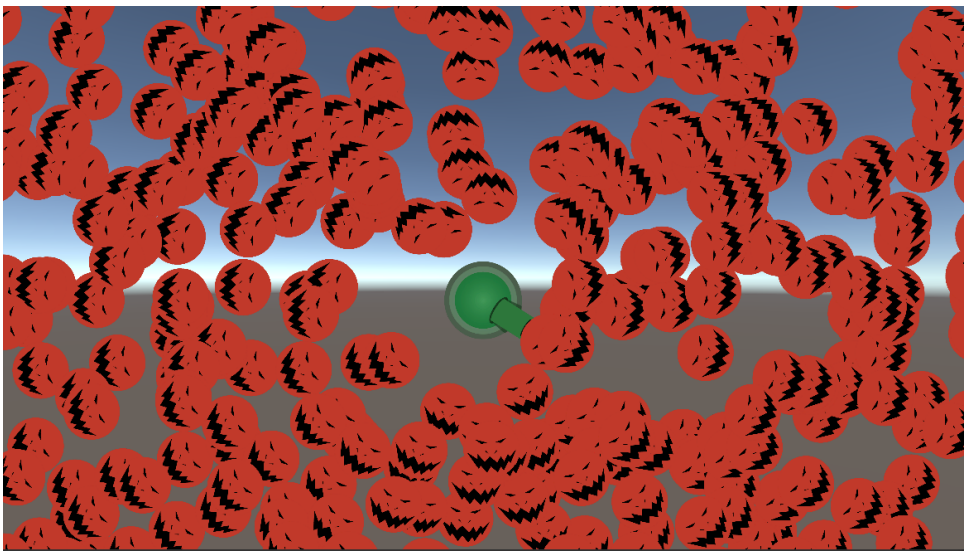
Monisäikeistetyssä koodissa voi ilmetä ongelmaksi niin sanotut kilpailutilanteet, joissa monet eri prosessit yrittävät muokata samaa tietoa yhtä aikaa aiheuttaen odottamattomia muutoksia. (Wheeler 2015.) Unityn C# job systemin safety system pitää huolen siitä, ettei kyseisiä tilanteita pääse syntymään. Havaituissa potentiaalisissa kilpailutilanteissa safety system kopioi käsiteltävän tiedon välttääkseen saman tiedon muokkaamisen. (Unity Technologies 2018, The safety system in the C# Job System)

Burst compiler on LLVM-projektia hyödyntävä teknologia, joka kääntää käyttäjän kirjoittaman C#-koodin hyvin optimoiduksi konekieleksi paremman suorituskyvyn saavuttamiseksi (Unity 2020, Burst User Guide). Tämän tutkimuksen DOTS-versiossa burst on automaattisesti käytössä, jonka voi myös halutessaan ottaa pois käytöstä.

5 Tutkimus

5.1 tutkimuksen kuvaus

Tutkimuksessa suunniteltiin molemmilla ohjelmointiparadigmoilla toteutettava yksinkertainen pelidemo, jonka ideana oli sisältää peleille tyypillisiä ominaisuuksia. Pelidemo sisältää ison määrän vihollisia, jotka hakeutuvat keskellä karttaa sijaitsevaan tykkiin. Tykki pyrkii puolustamaan itseään hakemalla lähimmän vihollisen ja ampumaan sitä räjähteellä, joka tuhoaa räjähdysalueella olevat viholliset. Näillä ominaisuuksilla projektista saatiin tarpeeksi laaja, jotta paradigmojen ominaisuuksia voitaisiin verrata keskenään.



Kuvio 4. Pelikuva tutkimuksessa kehitetystä pelidemosta.

5.2 Alkutoimenpiteet

Tutkimuksessa käytettiin Unityn versiota 2019.3.4f1. IDE:nä toimi microsoftin kehittämä visual studio 2019 community edition. Uutta projektia luodessa Unity sisältää valmiita projektimalleja, joista valittiin 3D-malliprojekti. Jotta DOTS-projektia pystyi kehittämään, Unity tarvitsi pakettijärjestelmästään DOTS-riippuvaiset paketit, joita olivat:

- Burst 1.3.0
- Mathematics 1.1.0
- Jobs 0.2.7
- Entities 0.8.0
- Hybrid Renderer 0.4.0

Pakettien asennusten jälkeen projekti oli valmis toteutettavaksi. Unity:n luotiin kaksi erikseen suoritettavaa sceneä, joista toinen on nimeltään main_oop ja toinen main_dots.

5.3 Toteutus

5.3.1 Olio-ohjelmointi

Tutkimuksessa toteutettiin kolme uudelleenkäytettävää game objectia eli prefabia: Enemy, Explosion sekä Player. Scriptejä tehtiin kokonaisuudessaan kuusi kappaletta, jotka ovat kooltaan pieniä ja perivät Unityn emoluokan MonoBehaviour, joka sisältää Unityn oleelliset toiminnot. Luokkien nimistä on helppo päätellä mitä ne tekevät esimerkiksi EnemySpawner, CameraController.

Unity:ssa olio-ohjelmointi on helppoa, koska paradigmaa on hyödynnetty moottorissa alusta asti. Uuden objektin luominen tapahtuu luomalla C#-scripti, joka perii Unityn toiminnallisuuksia sisältävän luokan MonoBehaviour. Unity luo scriptiin automaattisesti metodit Start ja Update. Startia kutsutaan silloin, kun objekti luodaan peliin ja Updatea kutsutaan jokaisella framella, kun objekti on pelimaailmassa olemassa. Jokainen MonoBehaviouria perivä luokka täytyy asettaa pelissä olevalle GameObjectille toimiakseen.

Peli sisältää kahden tyyppisiä tuhottavia objekteja, joita ovat Player ja Enemy. Nämä objektit sisältävät samoja toiminnallisuuksia, joita varten on hyvä käyttää olio-ohjelmoinnin perustoiminnallisuuksista perimistä ja rajapintaa. Sekä player että enemy voivat tuhoutua, joten luodaan yksi emoluokka nimeltä LivingEntity, jota molemmat luokat perivät. Lisäksi luodaan metodin TakeDamage sisältävä rajapinta IDamageable, jota voidaan toteuttaa LivingEntity-luokassa. Tämän avulla voidaan varmistaa että molemmat luokat sisältävät toiminnallisuuden TakeDamage, jota voidaan kutsua rajapinnan IDamageable kautta. Objektia vahingoittaessa ei siis ole väliä luokan varsinaisesta toteutuksesta, kunhan siltä vain löytyy IDamageable rajapinta. Näiden avulla sekä Player- että Enemy-luokka voivat hyödyntää ominaisuuksia health ja isDead.

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

1 reference
public class LivingEntity : MonoBehaviour, IDamageable{
    [SerializeField]
    protected float health;
    2 references
    public bool IsDead { get; protected set; }

    2 references
    public void TakeDamage(float amount) {
        if (health <= 0 && !IsDead) {
            IsDead = true;
            Destroy(gameObject);
        } else {
            health -= amount;
        }
    }
}
```

Kuvio 5. Player- sekä Enemy-luokan perimä luokka LivingEntity, joka toteuttaa rajapinnan IDamageable.

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

3 references
public interface IDamageable {
    3 references
    void TakeDamage(float amount);
}
```

Kuvio 6. IDamageable-rajapinta.

MonoBehaviourin metodilla GetComponent voidaan hakea tarvittavat riippuvuudet muihin scripteihin. MonoBehaviourin laajan kirjaston avulla pelidemon luominen on helppoa. Koodin määrä on pieni, joka auttaa selkeyttämään luokan lukemista.

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

0 references
public class Player : LivingEntity {
    private FindTarget findTarget;
    private ShootTarget shootTarget;

    0 references
    void Start() {
        findTarget = GetComponent<FindTarget>();
        shootTarget = GetComponent<ShootTarget>();

        List<GameObject> nearbyEnemies = EnemySpawner.nearbyEnemies;
        findTarget.NearbyEnemies = nearbyEnemies;
        shootTarget.NearbyEnemies = nearbyEnemies;
    }

    0 references
    void Update() {
        if(findTarget.Target == null) {
            findTarget.Find();
        } else {
            shootTarget.Shoot(findTarget.Target);
        }
    }
}
```

Kuvio 7. Player-luokka, joka käyttää luokkia FindTarget ja ShootTarget.

Player-luokka käyttää luokkia FindTarget löytääkseen lähimmän vihollisen ja ShootTarget ampuakseen sitä. MonoBehaviourin Start-metodia hyödyntämällä voidaan asettaa luokkien tarvitsemat riippuvuudet GetComponent-metodilla, kun player-objekti on luotu peliin. GetComponent hakee luokkaa GameObjectista, johon kutsuva scripti on liitetty. Tästä syystä Player-scriptin lisäksi GameObjectille on lisättävä Unityn käyttöliittymässä myös scriptit FindTarget ja ShootTarget, jotta ne löytyisivät. Enemy-luokan toteutus on samankaltainen, mutta eri tapaukseen sopivilla luokilla. Enemyn toteutuksen voi nähdä liitteistä 1-3.

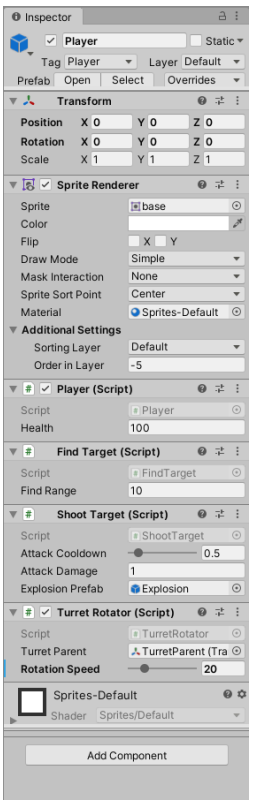
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

– references
public class FindTarget : MonoBehaviour {
    [SerializeField]
    private float findRange = 10f;
    – references
    public Transform Target { get; private set; } = null;
    – references
    public List<GameObject> NearbyEnemies { private get; set; } = null;

    – references
    public void Find() {
        if (NearbyEnemies.Count < 1)
            return;

        Transform closestTarget = null;
        float closestDistanceSqr = Mathf.Infinity;

        foreach (GameObject enemy in NearbyEnemies) {
            if (enemy == null)
                continue;

            float distanceSqrToTarget = (enemy.transform.position - transform.position).sqrMagnitude;
            float attackRangeSqr = findRange * findRange;

            if (distanceSqrToTarget <= attackRangeSqr && distanceSqrToTarget < closestDistanceSqr) {
                closestDistanceSqr = distanceSqrToTarget;
                closestTarget = enemy.transform;
            }
        }

        Target = closestTarget;
    }
}
```

Kuvio 8. FindTarget-luokka.

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

3 references
public class ShootTarget : MonoBehaviour {
    [Range(0, 5)]
    [SerializeField]
    private float attackCooldown = 0.5f;
    [SerializeField]
    private float attackDamage = 1f;
    private float timer = 0;

    [SerializeField]
    private GameObject explosionPrefab;
    2 references
    public List<GameObject> NearbyEnemies { private get; set; } = null;

    1 reference
    public void Shoot(Transform target) {
        float currentTime = Time.time;
        if (currentTime > timer) {
            SpawnExplosionEffect(target.position);

            foreach (GameObject enemy in NearbyEnemies) {
                if (enemy == null)
                    continue;

                if ((enemy.transform.position - target.position).magnitude <= 3f && enemy.TryGetComponent<IDamageable>(out var damageable)) {
                    damageable.TakeDamage(attackDamage);
                }
            }

            timer = currentTime + attackCooldown;
            target = null;
        }
    }

    1 reference
    private void SpawnExplosionEffect(Vector3 position) {
        GameObject effect = GameObject.Instantiate(explosionPrefab, position, Quaternion.Euler(0, 0, Random.Range(0, 360)));
        Destroy(effect, 1f);
    }
}
```

Kuvio 9. ShootTarget-luokka.

Objektien muuttujien arvoja voidaan normaalisti muokata Unityn inspector-ikkunan kautta, jos muuttujan näkyvyysmääre on public. SerializeField-attribuutin avulla ikkunassa voidaan näyttää arvo näkyvyysmääreestä välittämättä, jolloin näkyvyysmääre toimii ohjelmallisesti.



Kuvio 10. Unityn inspector-ikkuna Player GameObjectista.

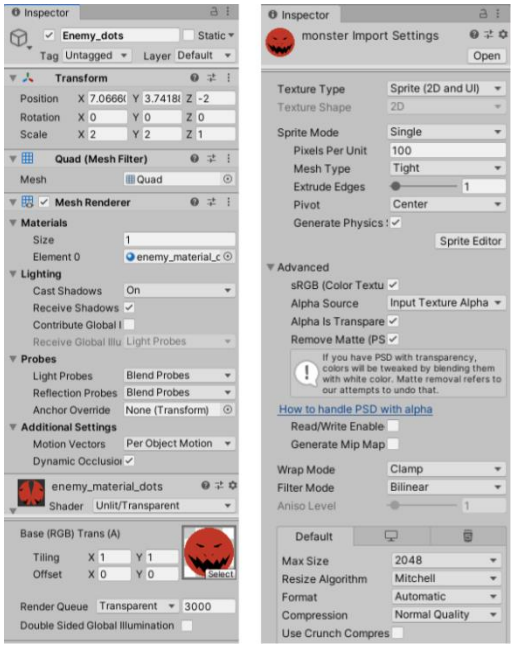
Pelissä luodaan haluttu määrä vihollisia pelaajan ympärille EnemySpawner-scriptillä, jonka jälkeen viholliset hakevat pelaajan ja suuntaavat sitä kohti. Vihollisten päästessä tarpeeksi lähelle ne alkavat hyökätä pelaajaa kahden sekunnin välein vähentäen pelaajan elämäpisteitä.

5.3.2 Data-suuntautunut suunnittelu

Data-suuntautuneessa suunnittelussa scriptien määrä on huomattavasti isompi. Samoihin toimintoihin tarvittavien scriptien määrä jakautui 14 komponenttiin ja 10 järjestelmään. Entiteetti komponentti järjestelmässä toiminnallisuuden toteuttaminen alkaa järjestelmän ja komponenttien yhteistoiminnan suunnittelemisesta. Entiteeteille voidaan asettaa komponentteja, joita järjestelmät muokkaavat saavuttaen halutun toiminnallisuuden.

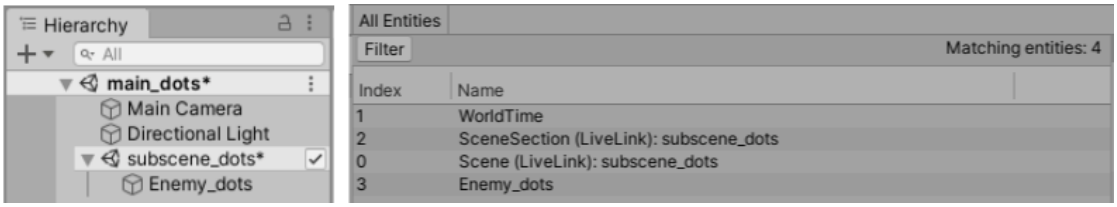
Vihollisen tarkoitus on hakea pelaaja kohteeksi, lähestyä sitä kohti ja etäisyydelle päästyään vahingoittaa sitä. Aloitetaan Enemy-entiteetin rakentaminen luomalla GameObject ja asettamalla sille mesh filter sekä mesh renderer. Koska peli on

kaksiulotteinen ja grafiikkana käytetään spritejä, asetetaan mesh filteriin meshin arvoksi quad ja mesh rendereriin oma materiaali, jonka shader on Unlit/transparent ja texture luomamme sprite.



Kuvio 11. Vasemmalla Inspector-ikkunan näkymä GameObjectista. Oikealla luodun spriten asetukset.

Jotta GameObject muutettaisiin entiteetiksi, pitää se asettaa subscenen alle. Luodaan uusi subscene Unityn Hierarchy-ikkunassa ja asetetaan GameObject sen alle. Tämän jälkeen sceneä ajettaessa GameObject muutetaan entiteetiksi ja entiteetti piirtyy näytölle. Entiteetin luomisen voi tarkistaa Entity Debugger-ikkunasta.



Kuvio 12. Vasemmalla scenen rakenne hierarchy-ikkunassa ja oikealla näkymä entity debugger-ikkunasta, joka sisältää luodun Enemy_dots-entiteetin

Seuraavaksi luodaan järjestelmä vihollisen liikkumiselle. Luodaan luokka nimeltä EnemyMovementJobSystem, joka perii luokan SystemBase. Järjestelmä etsii jokaisen

entiteetin, jolla on EnemyTag- ja HasTarget-komponentti ja muuttaa entiteetin Translation-komponenttia, joka määrittelee entiteetin sijainnin pelimaailmassa.

```
using Unity.Entities;
using Unity.Jobs;
using Unity.Transforms;
using Unity.Mathematics;

0 references
public class EnemyMovementJobSystem : SystemBase {
    9 references
    protected override void OnUpdate() {
        float deltaTime = Time.DeltaTime;

        Entities
            .WithAll<EnemyTag>()
            .ForEach((ref Translation translation, in HasTarget hasTarget) => {
                float3 direction = hasTarget.position - translation.Value;
                float distance = math.length(direction);

                if (distance > 2) {
                    translation.Value += (direction / distance) * deltaTime;
                }
            }).ScheduleParallel();
    }
}
```

Kuvio 13. EnemyMovementJobSystem-järjestelmä, jossa hyödynnetään rinnakkaissuorittamista ScheduleParallel-metodin avulla.

Luodulla entiteetillä ei ole vielä järjestelmän käyttämiä komponentteja. Luodaan struct nimeltä HasTarget, joka toteuttaa rajapinnan IComponentData ja sisältää kohteeseen liittyvää dataa, kuten mikä entiteetti on kyseessä ja mikä sen sijainti on. Komponentti on tarkoitus antaa entiteetille ajonaikaisesti silloin, kun entiteetille on löydetty kohde. Luodaan myös EnemyTag-komponentti, joka ei sisällä mitään dataa vaan toimii entiteetin tyypin tunnisteena. Komponentille pitää vielä lisätä attribuutti GenerateAuthoringComponent, jotta sen voi liittää GameObjectille Unityn inspector-ikkunassa.

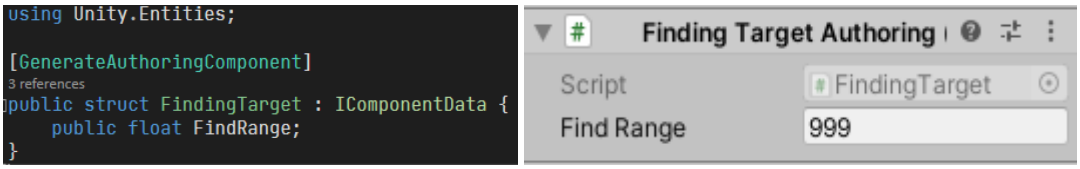
```
using Unity.Entities;
using Unity.Mathematics;
8 references
public struct HasTarget : IComponentData {
    public Entity target;
    public float3 position;
}
```

```
using Unity.Entities;

[GenerateAuthoringComponent]
4 references
public struct EnemyTag : IComponentData { }
```

Kuvio 14. Vasemmalla HasTarget-komponentti ja oikealla EnemyTag-komponentti

FindTargetSystem hoitaa lähimmän kohteen hakemisen ja HasTarget-komponentin lisäämisen. Kohteen haku on raskas operaatio, koska se vertailee jokaisen entiteetin välistä matkaa ja valitsee niistä lähimmän. Pääsäikeellä suorittaessa järjestelmän voisi toteuttaa niin, että tehdään kaksi ForEach-kutsua sisäkkäin, jossa jokaista entiteettiä vasten haetaan jokaisesta muusta entiteesti lähin kohde sekä lisätään EntityManager-luokan avulla entiteetille HasTarget-komponentti.

Operaatio on kuitenkin raskas, joten paremman suorituskyvyn saavuttamiseksi voisi käyttää rinnakkaissuorittamista. Rinnakkaissuorittamisessa ei kuitenkaan voi ainakaan vielä hyödyntää sisäkkäisten ForEach-metodien käyttöä eikä jobien sisällä voi hyödyntää EntityManageria. Tästä syystä koodista tulee hieman monimutkaisempi. Ensin pitää alustaa entity command buffer, sen jälkeen rakentaa kyselyt haluttuihin kohteisiin ja muuntaa kyselyt natiiveiksi taulukoiksi. Sen jälkeen kun taulukkoja ei enää käytetä, pitää kutsua niiden Dispose-metodia, joka poistaa taulukot muistivuotojen välttämiseksi. Järjestelmä vaatii vielä FindingTarget-komponentin, joten luodaan vielä kyseinen komponentti ja liitetään se entiteetille. Tämän jälkeen enemy-entiteetti hakee lähimmän kohteen ja alkaa liikkua sitä kohti. Liite 1 sisältää järjestelmän toteutuksen.



Kuvio 15. vasemmalla komponentti ja oikealla näkymä inspector-ikkunasta, johon entiteetille on asetettu haun kantamaksi 999.

Luodaan myös Pelaaja, joka on Enemy-entityn kohde ja lisätään sille vastaavat komponentit kuin Enemyllä, PlayerTag sekä FindingTarget. Pelaaja koostuu useammasta GameObjectista, joita ovat pohja ja tykki. Asetetaan PlayerTag molempiin objekteihin ja lapsiobjektiin FindTarget, jonka Find Range arvoksi asetetaan 8. Nyt pelaajakin pystyy hakemaan kohteen.

Kohteen haun lisäksi pelaajan pitäisi pystyä ampumaan kohdetta. Tätä varten luodaan järjestelmä CannonShootSystem, joka hakee Cannon- ja HasTarget-

komponenttien omaavat entiteetit ja ampuu kohteessa olevalle alueelle luoden ajonaikaisesti uuden entiteetin, joka sisältää komponentin AreaDamage.

```
using Unity.Entities;
using Unity.Mathematics;

0 references
public class CannonShootSystem : SystemBase {
    EndSimulationEntityCommandBufferSystem m_EndSimulationEcbSystem;

    7 references
    protected override void OnCreate() {
        base.OnCreate();
        m_EndSimulationEcbSystem = World.GetOrCreateSystem<EndSimulationEntityCommandBufferSystem>();
    }

    9 references
    protected override void OnUpdate() {
        var ecb = m_EndSimulationEcbSystem.CreateCommandBuffer().ToConcurrent();
        float deltaTime = Time.DeltaTime;

        Entities
            .ForEach((int entityInQueryIndex, Entity entity, ref Cannon cannon, in HasTarget hasTarget) => {
                cannon.timer -= deltaTime;

                if (cannon.timer < 0) {
                    float3 targetPosition = hasTarget.position;

                    Entity explosion = ecb.CreateEntity(entityInQueryIndex);
                    ecb.AddComponent(entityInQueryIndex, explosion, new AreaDamage { position = targetPosition, range = 3f, damage = 1f });

                    cannon.timer = cannon.cooldown;
                }
            }).ScheduleParallel();
    }
}
```

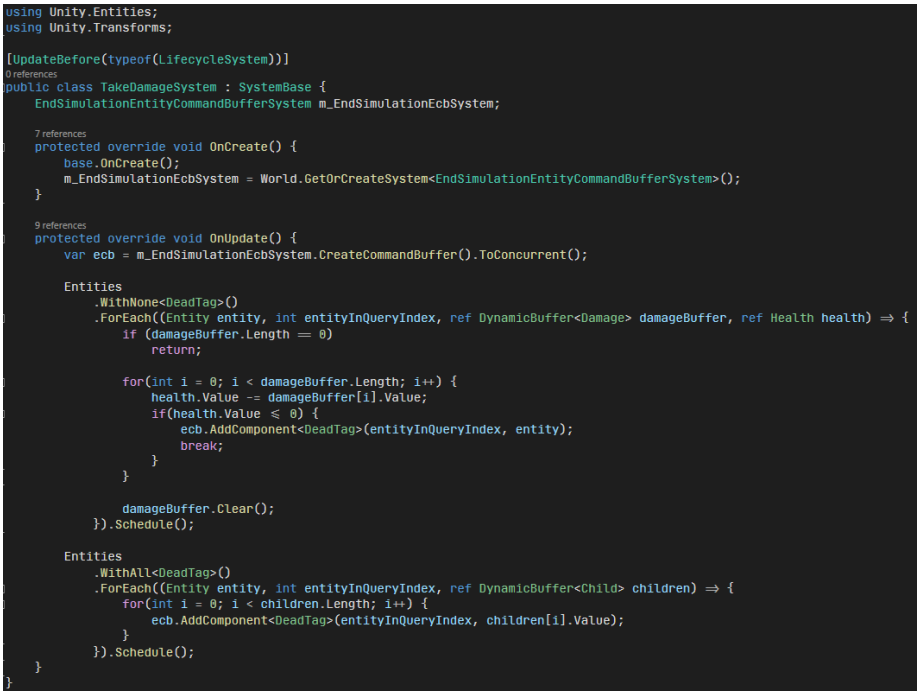
Kuvio 16. Järjestelmä ampumista varten.

AreaDamage-komponenttia varten tarvitaan AreaDamageSystem, jonka avulla vahingoitetaan vihollisia tietyltä alueelta. Järjestelmässä käydään läpi kaikki vihollisentiteetit ja asetetaan niille vahinkoa hyödyntämällä DynamicBufferia. AreaDamage-komponentti sisältää arvot vahingon määrälle ja etäisyydelle. Kun kaikkiin vihollisiin on tehty vahinkoa, tuhotaan AreaDamage-komponentin sisältävät entiteetit. Viholliset tekevät vahinkoa tietyin aikavälein päästessään pelaajan kantamalle hyödyntämällä DamageTargetSystemiä.

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Transforms;
using Unity.Jobs;
using Unity.Collections;

0 references
public class AreaDamageSystem : SystemBase {
    private EndSimulationEntityCommandBufferSystem m_EndSimulationEcbSystem;
    private EntityQuery query;

    7 references
    protected override void OnCreate() {
        base.OnCreate();
        m_EndSimulationEcbSystem = World.GetOrCreateSystem<EndSimulationEntityCommandBufferSystem>();
        query = GetEntityQuery(typeof(AreaDamage));
    }

    9 references
    protected override void OnUpdate() {
        var ecb = m_EndSimulationEcbSystem.CreateCommandBuffer().ToConcurrent();
        NativeArray<AreaDamage> areaDamages = query.ToComponentDataArray<AreaDamage>(Allocator.TempJob);

        Entities
            .WithAll<EnemyTag>()
            .WithAll<Health>()
            .ForEach((Entity entity, int entityInQueryIndex, ref DynamicBuffer<Damage> buffer, in Translation translation) => {
                for (int i = 0; i < areaDamages.Length; i++) {
                    if (math.distance(areaDamages[i].position, translation.Value) <= areaDamages[i].range) {
                        buffer.Add(new Damage { Value = areaDamages[i].damage });
                    }
                }
            })
            .WithDeallocateOnJobCompletion(areaDamages)
            .ScheduleParallel();

        Entities
            .WithAll<AreaDamage>()
            .ForEach((Entity entity, int entityInQueryIndex) => {
                ecb.DestroyEntity(entityInQueryIndex, entity);
            })
            .Schedule();
    }
}
```

Kuvio 17. Järjestelmä AreaDamage-komponenttien hallintaan.

Kun entiteeteille on lisätty vahinkoa, täytyy järjestelmän käsitellä ne ja muuntaa entiteettien Health-komponenttien arvoja. Jos vahinko ylittää komponenttiin määritetyn arvon, asetetaan entiteetille DeadTag merkitsemään entiteetti tuhottavaksi.



```
using Unity.Entities;
using Unity.Transforms;

[UpdateBefore(typeof(LifecycleSystem))]
0 references
public class TakeDamageSystem : SystemBase {
    EndSimulationEntityCommandBufferSystem m_EndSimulationEcbSystem;

    7 references
    protected override void OnCreate() {
        base.OnCreate();
        m_EndSimulationEcbSystem = World.GetOrCreateSystem<EndSimulationEntityCommandBufferSystem>();
    }

    9 references
    protected override void OnUpdate() {
        var ecb = m_EndSimulationEcbSystem.CreateCommandBuffer().ToConcurrent();

        Entities
            .WithNone<DeadTag>()
            .ForEach((Entity entity, int entityInQueryIndex, ref DynamicBuffer<Damage> damageBuffer, ref Health health) => {
                if (damageBuffer.Length == 0)
                    return;

                for(int i = 0; i < damageBuffer.Length; i++) {
                    health.Value -= damageBuffer[i].Value;
                    if(health.Value <= 0) {
                        ecb.AddComponent<DeadTag>(entityInQueryIndex, entity);
                        break;
                    }
                }

                damageBuffer.Clear();
            }).Schedule();

        Entities
            .WithAll<DeadTag>()
            .ForEach((Entity entity, int entityInQueryIndex, ref DynamicBuffer<Child> children) => {
                for(int i = 0; i < children.Length; i++) {
                    ecb.AddComponent<DeadTag>(entityInQueryIndex, children[i].Value);
                }
            }).Schedule();
    }
}
```

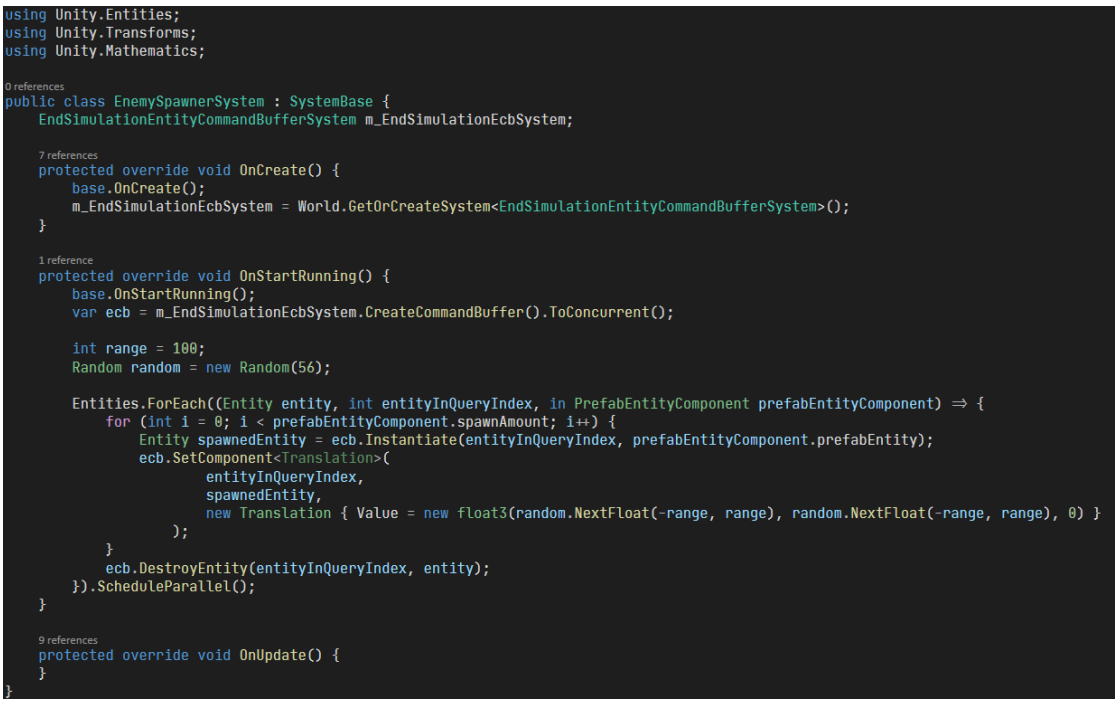
Kuvio 18. Vahinkoja käsittelevä järjestelmä

Pelissä oleva tykki sisältää lapsiobjekteja, jotka muutetaan itsenäisiksi entiteeteiksi pelin alussa. Vaikka parent-entiteetti tuhottaisiin, niin lapsientiteettejä ei tuhota. Tästä syystä jokaiselle Health-komponenttia sisältävälle entiteetille asetetaan ensin DeadTag, jonka jälkeen se asetetaan vielä niiden lapsille, ettei pelkästään yksi entiteetti tuhoutuisi. Kun DeadTagit on asetettu, LifeCycleSystem käy läpi jokaisen kyseisellä tagilla varustetun entiteetin ja tuhoaa sen. Tämän lisäksi LifeCycleSystem lisää myös jokaiselle health-komponentin omaavalle entiteetille Damage dynamic bufferin.

```
using Unity.Entities;

1 reference
public class LifecycleSystem : SystemBase {
    EndSimulationEntityCommandBufferSystem m_EndSimulationEcbSystem;

    7 references
    protected override void OnCreate() {
        base.OnCreate();

        m_EndSimulationEcbSystem = World.GetOrCreateSystem<EndSimulationEntityCommandBufferSystem>();
    }

    1 reference
    protected override void OnStartRunning() {
        base.OnStartRunning();

        var ecb = m_EndSimulationEcbSystem.CreateCommandBuffer().ToConcurrent();

        Entities.ForEach((Entity entity, int entityInQueryIndex, ref Health health) => {
            health.Value = health.max;
            ecb.AddBuffer<Damage>(entityInQueryIndex, entity);
        }).ScheduleParallel();

    }
    9 references
    protected override void OnUpdate() {
        var ecb = m_EndSimulationEcbSystem.CreateCommandBuffer().ToConcurrent();

        Entities
            .WithAll<DeadTag>()
            .ForEach((Entity entity, int entityInQueryIndex) => {
                ecb.DestroyEntity(entityInQueryIndex, entity);
            })
            .Schedule();

        this.CompleteDependency();
    }
}
```

Kuvio 19. Entiteettien elinkaarta käsittelevä järjestelmä.

Viimeiseksi tarvitaan järjestelmä, joka luo pelin alussa halutun määrän vihollisia pelaajan ympärille. Tätä varten tehdään EnemySpawnerSystem, joka käsittelee kaikki PrefabEntityComponentin sisältävät entiteetit ja luo komponenttiin asetettuja prefabeja halutun määrän.



```
using Unity.Entities;
using Unity.Transforms;
using Unity.Mathematics;

0 references
public class EnemySpawnerSystem : SystemBase {
    EndSimulationEntityCommandBufferSystem m_EndSimulationEcbSystem;

    7 references
    protected override void OnCreate() {
        base.OnCreate();
        m_EndSimulationEcbSystem = World.GetOrCreateSystem<EndSimulationEntityCommandBufferSystem>();
    }

    1 reference
    protected override void OnStartRunning() {
        base.OnStartRunning();
        var ecb = m_EndSimulationEcbSystem.CreateCommandBuffer().ToConcurrent();

        int range = 100;
        Random random = new Random(56);

        Entities.ForEach((Entity entity, int entityInQueryIndex, in PrefabEntityComponent prefabEntityComponent) => {
            for (int i = 0; i < prefabEntityComponent.spawnAmount; i++) {
                Entity spawnedEntity = ecb.Instantiate(entityInQueryIndex, prefabEntityComponent.prefabEntity);
                ecb.SetComponent<Translation>(
                        entityInQueryIndex,
                        spawnedEntity,
                        new Translation { Value = new float3(random.NextFloat(-range, range), random.NextFloat(-range, range), 0) }
                    );
            }
            ecb.DestroyEntity(entityInQueryIndex, entity);
        }).ScheduleParallel();
    }

    9 references
    protected override void OnUpdate() {
    }
}
```

Kuvio 20. vihollisten luomiseen käytetty järjestelmä

6 Tulokset

Tutkimuksen tavoitteena oli tutustua data-suuntautuneeseen suunnitteluun ja verrata sitä olio-ohjelmointiin. Data-suuntautuneen ajatusmallin opetteleminen vei aikaa, mutta tuotti lopulta huomattavasti suorituskykyisempää koodia. Koodin määrä oli kuitenkin suurempi olio-ohjelmointiin verrattuna. Olio-ohjelmoinnissa tuotettiin 280 koodiriviä, kun taas DOTS:ssa samoihin toiminnallisuuksiin tarvittiin 515 koodiriviä. Vaikka DOTS:n koodimäärä on suurempi, niin koodi on geneerisempää eli sitä ei ole sidottu johonkin tiettyyn konseptiin. Tästä syystä uskoisin koodin olevan uudelleenkäytettävämpää projektin laajentuessa.

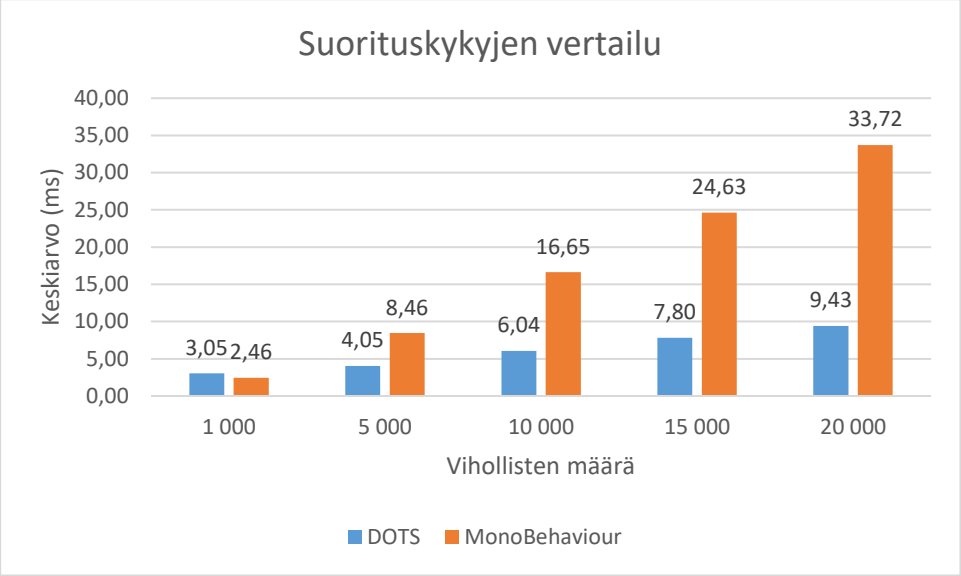
Data-suuntautuneen ajatusmallin opetteleminen tuotti paljon haasteita projektin aikana. Jokaisen toiminnallisuuden suunnittelussa piti unohtaa objektin konteksti ja miettiä komponentteja mahdollisimman pienissä osissa, mitä dataa minun täytyy muokata, jotta saavutan halutun toiminnallisuuden? Ajattelua helpottava periaate oli

se, että komponentti tuottaa saman lopputuloksen riippumatta siitä mihin entiteettiin komponentti asetetaan. Yksinkertaisesti olio-ohjelmointi on ylhäältä alaspäin ajateltava malli, kun taas data-suuntaunut suunnittelu on alhaalta ylöspäin ajateltava malli. Esimerkiksi olio-ohjelmoinnissa rakennetaan ensin auto, jolle annetaan ominaisuudet ja toiminnallisuudet, kun taas data-suuntautuneessa suunnittelussa rakennetaan ensin renkaat, moottori, kori ja ratti, joiden yhdistelmällä tuotetaan auto.

Kehitysvaiheessa olevan paketin nopeat muutokset aiheuttivat hankaluuksia DOTS:lla kehitettäessä. Dokumentaatio oli vähäistä ja suurin osa netistä löytyneistä ohjeista olivat kerenneet jo vanhentua. Metodien nimet ovat vaikeasti luettavia ja joitakin ominaisuuksia oli vaikea ymmärtää. Myös rinnakkaissuorittamiseen täytyi tutustua, jotta pakettia voitaisiin hyödyntää täysin, joka vaati lisää opettelua.

Vahvasti iskostuneet olio-ohjelmoinnin periaatteet yrittivät tulla esiin data-suuntautunutta koodia kirjoittaessa. Tämä näkyy myös EnemyMovementSystemissä, joka on nimetty sitä käyttävän objektin mukaan, joka ei ole data-suuntautuneen suunnittelun periaatteiden mukaista. Parempi nimi kyseiselle järjestelmälle olisi geneerisemmin liikkumista kuvaava nimi, kuten esimerkiksi MovementSystem tai MoveToTargetSystem.

Suorituskyvyn keskiarvon laskemiseen käytettiin Github-käyttäjän Steve3003 kehittämää Unity Profiler Data Exporter -kirjastoa. Kuviossa 21 näkyy vertailu DOTS-paketin ja perinteisen MonoBehaviour tyylin suorituskykyjen keskiarvot vihollisten määrän suhteen.



Kuvio 21. Suorituskyky vihollisten määrän suhteen. Pienempi arvo on nopeampi.

Vertailusta näkyy kuinka MonoBehaviour- tyylillä toimintoihin käytetty aika kasvaa lähes tuplasti DOTS:n verrattuna 5 000 vihollisen kohdalla ja kasvaa entistä enemmän vihollisten määrän kasvaessa. DOTS:n suorituskyky huononee vihollisten määrän kasvaessa, mutta huomattavasti vähemmän kuin MonoBehaviourilla.

7 Pohdinta

Tutkijan vahva kokemus olio-ohjelmoinnista sekä teknologian vajaa dokumentointi voivat vaikuttaa tuloksiin niin, että data-suuntautuneen suunnittelun koodi ei välttämättä ole parhaiden käytänteiden mukaista. Muita data-suuntautuneen suunnittelun piirteitä ajatellaan myös olevan laajennettavuus, muutoksiin reagoiminen ja helppo testaus, joita ei pystytty tutkimaan projektin pienen koon takia.

Tuloksien käytettävyyteen voi vaikuttaa DOTS:n kehitysvaiheessa tapahtuvat muutokset ottaen huomioon aikaisemmat muutokset versioiden välillä. Data-suuntautuneen ohjelmoinnin periaatteet eivät kuitenkaan ole muuttuneet, joten paradigman tutkiminen osoittautui tutkijalle hyödylliseksi. DOTS:n yhteistoiminta

Unity:n editorin kanssa on vielä tällä hetkellä kesken, joten teknologian julkaistavan version tutkiminen olisi hyvä idea jatkotutkimukselle.

Tutkimuksessa käytetty kohteiden haku on myös epäoptimaalinen. Tällä hetkellä pelaajan ja räjähteen paikkoja verrataan jokaista vihollista kohden. Parempi ratkaisu olisi jakaa pelimaailma pienempiin alueisiin ja verrata alueiden sisällä olevia objekteja keskenään, jotta vältyttäisiin turhalta suorittamiselta.

Suurimmiksi ongelmiksi osoittautui vähäinen dokumentaatio, version muutokset ja koodin vaikea syntaksi. Myös järjestelmien kehittäminen rinnakkaisajoa varten osoittautui erittäin haastavaksi, koska sisäkkäisiä forEach-metodeja ei voinut käyttää. Tämä aiheutti hankaluuksia suunniteltaessa kahden entiteetin välisiä toimintoja. Vaikka suorituskyvyn näkökulmasta DOTS vaikuttaa erittäin lupaavalta, en suosittele teknologiaa vielä käytettäväksi sen keskeneräisyyden takia.

Lähteet

adam. 2007. Entity Systems are the future of MMOG development – Part 2. Viitattu 23.8.2020. http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/

Fabian, R. 2018. Data-Oriented Design. Viitattu 23.4.2020. https://www.dataorienteddesign.com/dodbook/node2.html

Fabian, R. 2018. Component Based Objects. Viitattu 23.8.2020. https://www.dataorienteddesign.com/dodbook/node5.html

Gabbrielli M. & Martini S. 2010. Programming languages: Principles and Paradigms. Springer.

Hennessy J. L. & Patterson D. A. 2012. Computer architecture: a quantitative approach 5th edition.

Llopis, N. 2009. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). Viitattu 3.5.2020. http://gamesfromwithin.com/data-oriented-design

Microsoft Corporation. 2015. Functional Programming vs. Imperative Programming (C#). Viitattu 23.8.2020. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/functional-programming-vs-imperative-programming

Microsoft Corporation. 2018. Inheritance in C# and .NET. Viitattu 23.8.2020. https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance

Microsoft Corporation. 2020. Object-Oriented programming (C#). Viitattu 11.8.2020. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/object-oriented-programming

Nygaard, K & Dahl O. J. 1978. The Development of the SIMULA Languages. Viitattu 23.8.2020. https://hannemyr.com/cache/knojd_acm78.pdf

Nørmark, K. 2014. Programming paradigms. Aalborg University. Viitattu 30.4.2020. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html

Oracle Corporation. What Is a Class? Viitattu 26.8.2020. https://docs.oracle.com/javase/tutorial/java/concepts/class.html

Oracle Corporation. What Is an Interface? Viitattu 23.8.2020. https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Oracle Corporation. What Is an Object? Viitattu 26.4.2020. https://docs.oracle.com/javase/tutorial/java/concepts/object.html

Steve3003. Unity Profiler Data Exporter. Viitattu 9.5.2020.
https://github.com/steve3003/unity-profiler-data-exporter

Unity Technologies. 2020. Burst User Guide. Viitattu 9.5.2020.
https://docs.unity3d.com/Packages/com.unity.burst@1.3/manual/index.html

Unity Technologies. 2018. Creating and Using Scripts.
https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

Unity Technologies. 2020. ECS concepts. Viitattu 25.4.2020.
https://docs.unity3d.com/Packages/com.unity.entities@0.8/manual/ecs_core.html

Unity Technologies. 2020. Entities. Viitattu 23.8.2020.
https://docs.unity3d.com/Manual/com.unity.entities.html

Unity Technologies. 2017. GameObjects. Viitattu 11.8.2020.
https://docs.unity3d.com/Manual/GameObjects.html

Unity Technologies. 2020. MonoBehaviour. Viitattu 11.8.2020.
https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Unity Technologies. 2018. Prefabs. Viitattu 11.8.2020.
https://docs.unity3d.com/Manual/Prefabs.html

Unity Technologies. 2017. Scenes. Viitattu 11.8.2020.
https://docs.unity3d.com/Manual/CreatingScenes.html

Unity Technologies. 2018. What Is a Job System? Viitattu 9.5.2020.
https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemJobSystems.html

Unity Technologies. 2018. The safety system in the C# Job System. Viitattu 9.5.2020.
https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemSafetySystem.html

Unity Technologies. 2020. Unity Platform. Viitattu 23.8.2020. https://unity.com/products/unity-platform

Van Roy, P. 2009. Programming Paradigms for Dummies: What Every Programmer Should Know. https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf

Weisfeld, M. 2009. The Object-Oriented Thought Process, Third Edition. Addison-Wesley.

Wheeler, D. A. 2015. Secure Programming HOWTO. Viitattu 9.5.2020.
https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/avoid-race.html

Liitteet

```
using UnityEngine;

[RequireComponent(typeof(EnemyMovement))]
[RequireComponent(typeof(EnemyAttack))]
0 references
public class Enemy : LivingEntity {
    3 references
    public Transform Target { get; private set; }

    private EnemyMovement movement;
    private EnemyAttack attack;

    0 references
    private void Start() {
        Target = GameObject.FindGameObjectWithTag("Player").transform;

        attack = GetComponent<EnemyAttack>();
        movement = GetComponent<EnemyMovement>();

        attack.target = Target;
        movement.target = Target;
        movement.transformToMove = transform;
    }

    0 references
    private void Update() {
        if(movement.DistanceToTarget ≤ attack.attackRange) {
            attack.Attack();
        } else {
            movement.Move();
        }
    }
}
```

Liite 1. Enemy-luokka

```
using UnityEngine;

3 references
public class EnemyMovement : MonoBehaviour {
    [HideInInspector]
    public Transform target;
    [HideInInspector]
    public Transform transformToMove;

    [Range(0, 100)]
    [SerializeField]
    private float rotationSpeed = 40f;
    [SerializeField]
    private float movementSpeed = 2f;

    2 references
    public float DistanceToTarget { get; private set; } = Mathf.Infinity;
    private Vector3 directionToTarget;

    1 reference
    public void Move() {
        if (target == null || transformToMove == null)
            return;

        CalculatePositions();
        RotateTowardsTarget();
        MoveTowardsTarget();
    }

    1 reference
    private void CalculatePositions() {
        directionToTarget = target.position - transformToMove.position;
        DistanceToTarget = directionToTarget.magnitude;
    }

    1 reference
    private void MoveTowardsTarget() {
        transformToMove.Translate(directionToTarget.normalized * movementSpeed * Time.deltaTime, Space.World);
    }

    1 reference
    private void RotateTowardsTarget() {
        Quaternion targetRotation = Quaternion.LookRotation(Vector3.forward, directionToTarget.normalized);
        transformToMove.rotation = Quaternion.Lerp(transformToMove.rotation, targetRotation, rotationSpeed * Time.deltaTime);
    }
}
```

Liite 2. EnemyMovement-luokka

```
using UnityEngine;

3 references
public class EnemyAttack : MonoBehaviour {
    [HideInInspector]
    public Transform target;

    [SerializeField]
    public float attackRange = 2f;
    private float nextAttackTime = 0f;

    1 reference
    public void Attack() {
        if (target == null)
            return;

        float currentTime = Time.time;
        if (currentTime >= nextAttackTime && target.TryGetComponent<IDamageable>(out var damageable)) {
            damageable.TakeDamage(1);
            nextAttackTime = currentTime + 2f;
        }
    }
}
```

Liite 3. EnemyAttack-luokka