Tampereen ammattikorkeakoulu

Peliprototyypin luominen Unity DOTSia käyttäen

Eetu Maunuksela

OPINNÄYTETYÖ
Joulukuu 2019

Tietojenkäsittely
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

MAUNUKSELA, EETU:
Peliprototyypin luominen Unity DOTSia käyttäen

Opinnäytetyö 50 sivua
Joulukuu 2019

Tämän opinnäytetyön tavoitteena oli oppia ymmärtämään ja käyttämään Unity-pelinkehitysalustan uutta kehitysvaiheessa olevaa teknologiapakettia ja sen kolmea osaa. Opinnäytetyössä tutkittiin, miten uusien teknologioiden käyttäminen eroaa perinteisestä Unitystä ohjelmoinnin ja työnkulun kannalta sekä mitä hyötyjä tai haittoja teknologiapaketti mahdollisesti tuo tullessaan. Opinnäytetyössä käytettiin Unityn 2019.1.10f1 -versiota. Koodin kirjoittamiseen käytettiin Microsoft Visual Studiota ja C#-ohjelmointikieltä.

Uusi teknologiapaketti on suuri kokonaisuus ja sen ominaisuuksien ja menetelmien oppiminen ja sisäistäminen lyhyessä ajassa koitui haastavaksi. Opinnäytetyön tuloksena syntyi 2D-pelin prototyyppi, joka on tehty Unityn uusia teknologioita käyttäen. Uusien teknologioiden avulla voidaan parantaa huomattavasti pelin suorituskykyä. Lisäksi ne auttavat kirjoittamaan helppolukuista koodia ja vähentämään bugeja eli ohjelmointivirheitä. Opinnäytetyön aikana kävi ilmi, että paketista puuttuu vielä paljon oleellisia ominaisuuksia.

Johtopäätöksinä todettiin, että uuden teknologiapaketin kokonaisuuden sisäistämiseen vaaditaan pidempi aika kuin tämän opinnäytetyön tekemiseen käytetty aika. Lisäksi teknologiapaketin osia on tällä hetkellä järkevin käyttää yhdessä perinteisen Unityn kanssa, jos haluaa hyötyä uudesta teknologiapaketista. Uusi teknologiapaketti on kuitenkin viemässä Unityä oikeaan suuntaan, ja se on tulevaisuudessa varmasti osa Unityä.

Asiasanat: unity, unity dots, ohjelmointi, c#, pelinkehitys

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Game Production

EETU MAUNUKSELA:
Creating a Game Prototype Using Unity DOTS

Bachelor's thesis 50 pages
December 2019

The aim of this thesis was to learn to understand and use Unity's new technology stack and its three main features. This thesis focuses on the new technologies' pros and cons, and how their usage differs from the classic Unity in terms of programming and workflow. Unity 2019.1.10f1 along with Microsoft Visual Studio and C#, were used for the practical part of the thesis.

The new technology stack is a large set and learning its features in a short time proved to be quite difficult. A prototype of a 2D game was made using Unity's new technologies. With these new technologies the game's performance can be increased considerably. They also help in writing easy-to-read code and decrease the number of bugs. During this thesis it became clear that the technology stack is still missing many essential features.

A conclusion was made that internalizing the whole technology stack would take a much longer time than what was used for making this thesis. The second conclusion is that it is recommended to use the new features in conjunction with the classic Unity if there is a desire to benefit from the new technologies. However, the new technology stack is a good direction for Unity, and it will be most likely a part of Unity in the future.

Key words: unity, unity dots, programming, c#, game development

SISÄLLYS

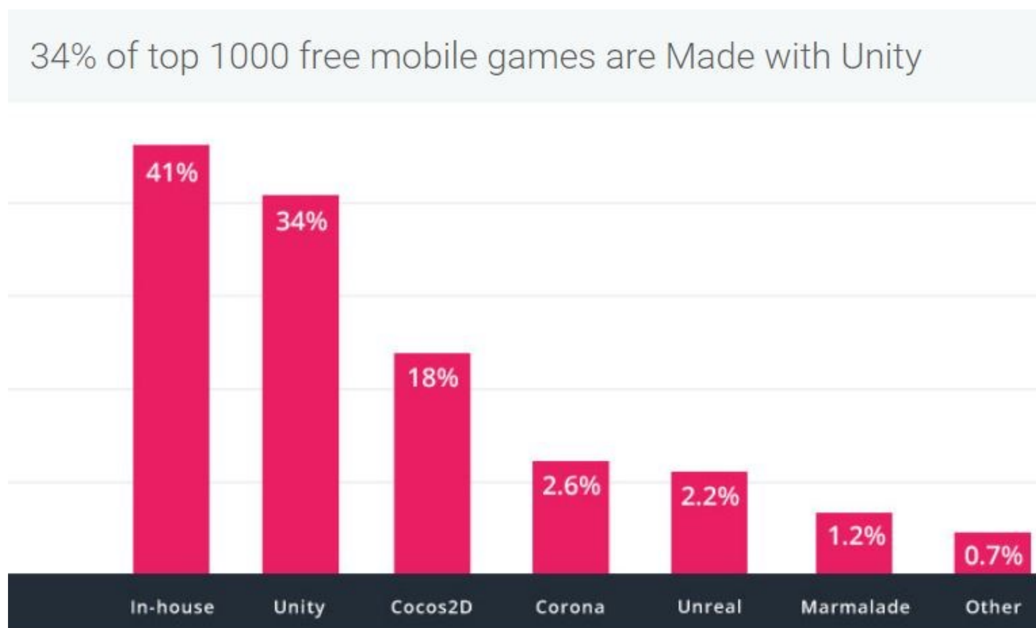
1 JOHDANTO 6
2 UNITY 8
3 Olio-ohjelmointi 10
3.1 Luokat ja objektit 10
3.2 Kapselointi 10
3.3 Periytyminen 11
3.4 Polymorfismi 12
3.5 Abstraktio 12
3.6 OOP Unityssä 12
4 DOTS 14
4.1 Dataorientoitunut malli 14
4.2 Entity Component System 15
4.2.1 Entiteetit 16
4.2.2 Komponentit 16
4.2.3 Systeemit 17
4.3 C# Job System 18
4.4 Burst Compiler 19
5 Toteutus 20
5.1 DOTS-ominaisuuksien käyttöönotto 20
5.2 Pelin tilat 21
5.2.1 Päävalikko 21
5.2.2 Pelitila 21
5.2.3 Peli ohi -tila 22
5.3 Koodi 23
5.3.1 Renderointi 24
5.3.2 Animointi 25
5.3.3 UI 26
5.3.4 Kamera 32
5.3.5 Äänet 33
5.3.6 Pelin tilat 34
5.3.7 Pelaajan hahmo 36
5.3.8 Viholliset 42
5.3.9 Poimittavat aseet ja parannukset 46
5.3.10 Job System ja Burst Compiler 47
6 POHDINTA 50
LÄHTEET 51

LYHENTEET JA TERMIT

CLR	Common Language Runtime
DOTS	Data-oriented technology stack
ECS	Entity Component System
OOP	Object-oriented Programming
DOD	Data-oriented design
Job	Toiminnallisuus, joka ajetaan työläis-säikeessä
API	Application Programming Interface
Scene	Tiedosto, joka sisältää kokoelman pelin elementtejä
Sprite	Kaksiulotteinen kuva
Spritesheet	Kuva, jossa on monta spriteä
Frame-by-frame	Animointitekniikka, jossa animaation jokainen vaihe on oma spritensä
Poolaus	Menetelmä, jossa tarvittavia objekteja luodaan valmiiksi listaan ja niitä aktivoidaan ja deaktivoidaan sitä mukaan, kun niitä tarvitaan.

1 JOHDANTO

Peliala kukoistaa ja uusia pelejä kehitetään ja julkaistaan jatkuvasti. Videopeliteollisuus tuotti 131 miljardia dollaria vuonna 2018, ja sen on arvioitu tuottavan jopa 300 miljardia dollaria vuodessa vuoteen 2025 mennessä. (Lanier. 2019.) Videopelit eivät kuitenkaan synny tyhjästä, vaan on olemassa työkaluja, joita käytetään pelien luomiseen. Videopelit käyvät niin sanotulla pelimoottorilla, jonka päälle peli rakennetaan. Yksi maailman suosituimmista pelimoottoreista/pelinkehitys alustoista on Unity Technologies:in Unity. Unity omisti arvion mukaan yli 45% pelimoottorimarkkinoista vuonna 2017. Vuonna 2016 tehdyn tutkimuksen mukaan 1000:nnesta suosituimmasta ilmaisesta mobiilipelistä 34% oli tehty Unityllä (kuvio 1). (Nanalyze 2017.) Unityssä käytetty ohjelmointi- ja suunnittelumalli on tähän asti ollut objektiorientoitunutta, mutta tässä lähestymistavassa on huomattu puutteita ja ongelmia. Unity on päättänyt luoda uuden teknologiapaketin, Data Oriented Technology Stack:in (DOTS), jonka tarkoitus on ratkaista kyseiset ongelmat siirtymällä perinteisestä olio-ohjelmoinnin mallista dataorientoituneeseen malliin.



Kuvio 1. Pelimoottoreiden käyttö prosentteina 1000:nnessa suosituimmassa mobiilipelissä vuonna 2016 (Nanalyze 2017.)

Tässä opinnäytetyössä käsitellään Unity DOTSia ja sen tämänhetkistä tilannetta. Opinnäytetyön tavoitteena on oppia käyttämään ja ymmärtämään Unityn uutta

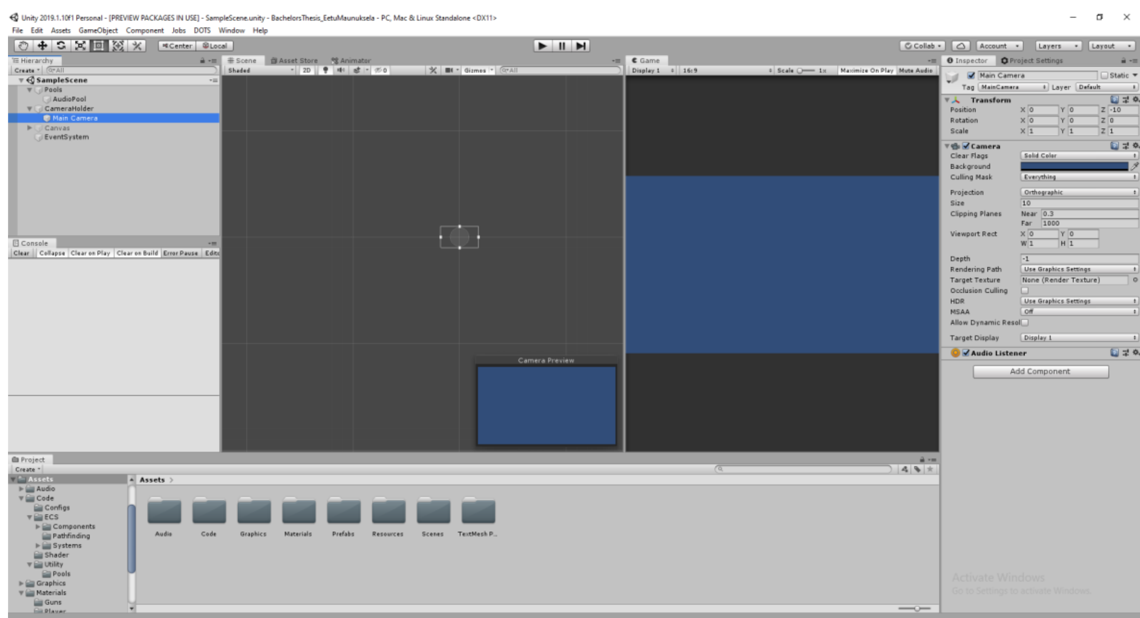
DOTS-järjestelmää, jotta opittua tietoa voidaan käyttää hyödyksi tulevaisuudessa. Lisäksi tavoitteena on tutkia tarkemmin, mitä hyötyjä tai haittoja uusi järjestelmä tällä hetkellä pitää sisällään. Työn yhteydessä syntyy 2D-pelin prototyyppi, joka on luotu käyttäen Unity-pelimoottoria ja Unity DOTSia.

Työllä ei ole toimeksiantajaa, vaan se tehdään omaksi eduksi. Opinnäytetyötä voi hyödyntää myös kuka tahansa muu aiheesta kiinnostunut.

Työn teoriaosuudessa käydään läpi Unityn perinteistä järjestelmää ja menetelmiä, jonka jälkeen siirrytään uuteen teknologiapakettiin ja sen kolmeen pää ominaisuuteen. Toteutusosuudessa keskitytään tarkemmin siihen, miten uuden teknologiapaketin ominaisuudet saa käyttöönsä ja miten niitä käytetään. Lopuksi pohdintaosuudessa käydään läpi työn tuloksia ja siitä syntyneitä johtopäätöksiä.

2 UNITY

Unity on pelimoottori ja pelinkehitysalusta, jonka on kehittänyt San Fransiscolainen Unity Technologies vuonna 2005. Unityn pelinkehitysalustaa voi käyttää niin aloittelijat kuin ammattilaiset ja suuret pelistudiotkin. Unityllä voidaan luoda pelejä lähes jokaiselle laitteelle ja alustalle. Kaksi ja kolmiulotteisten pelien lisäksi Unityllä voidaan luoda myös virtuaalitodellisuus sekä sulautettutodellisuus pelejä. Unity on maailman suosituin pelimoottori, ja sitä voidaan käyttää pelialan lisäksi myös esimerkiksi elokuva-alalla. Unity on saatavilla Microsoft Windowsille, macOS:lle sekä Linuxille (kuva 1). (Nanalyze 2017.)



Kuva 1. Unity editori.

Unityn Play Mode on erittäin kätevä työkalu pelinkehittäjälle. Sen avulla peli voidaan käynnistää yhdellä napin painalluksella ja peli tulee näkyville editorin peliruutuun. Pelin arvoja ja muita osia voidaan myös muuttaa ajon aikana, jolloin muutosten vaikutukset voi nähdä välittömästi. Yhdessä Microsoft Visual Studion kanssa kehittäjä voi käydä koodia läpi rivi kerrallaan, mikä auttaa huomattavasti koodin virheiden etsintää. Unityn APIen avulla editoria voidaan muokata käyttäjän haluamalla tavalla. Kehittäjän tarpeisiin räätälöidyt kustomoinnit voivat helpottaa pelinkehitysprosessia huomattavasti. Editoriin voi myös ladata muiden käyttäjien luomia plugineita eli liitännäisiä Unityn Asset Storesta. (Unity Technologies 2019a.)

Unityn pääasiallinen koodikieli on C# ja siinä on natiivi Visual Studio -integraatio. Muita työkaluja ja ominaisuuksia, joita Unity sisältää: animaatiot, grafiikat, optimointi, äänet, fysiikat ja partikkelijärjestelmä. Unity tukee lukuisia eri formaatteja kuva, ääni, video ja teksti tiedostoissa, ja niitä on helppo tuoda projektiin esimerkiksi raahaamalla ja pudottamalla. (Unity Technologies 2019a.)

Unitystä on olemassa kolme eri versiota. Personal-versio on ilmainen ja se on tarkoitettu aloittelijoille sekä kenelle tahansa, joka haluaa kehittää pelejä pienessä skaalassa. Personal-versiota saa käyttää vapaasti, kunhan tuotot tai rahoitusten määrä eivät ylitä 100 000 dollaria vuodessa. Plus-versio on tarkoitettu harrastelijoille ja se maksaa noin 25 dollaria kuukaudessa. Pro-versio taas on tarkoitettu ammattilaisille ja pelistudioille ja se maksaa 125 dollaria kuukaudessa. Maksulliset Unity-versiot sisältävät lisäominaisuuksia, joita ilmaisversioon ei kuulu. (Unity Technologies 2019a.)

3 Olio-ohjelmointi

Olio-ohjelmoinnilla (OOP) tarkoitetaan tietynlaista ohjelmointimallia, joka viittaa ohjelmointiin, jossa ohjelmoija määrittää tietorakenteiden tyypin sekä funktiot, joita tietorakenteisiin sovelletaan. Tietorakenteesta luodaan objekteja, jotka sisältävät sekä dataa että funktioita. OOP:ssä objektien/luokkien välille luodaan usein yhteyksiä ja ne ovat vuorovaikutuksessa keskenään. OOP:n avulla ohjelmoija voi luoda modulaarista koodia. (Beal n.d.)

3.1 Luokat ja objektit

Objektilla tarkoitetaan yleensä yksittäistä asiaa, jota voidaan manipuloida. Objektit koostuvat datasta, sekä toiminnallisuuksista jotka muokkaavat tätä dataa. (Beal n.d.)

Luokat toimivat ikään kuin objektien pohjapiirustuksina. Luokan sisällä voidaan määritellä dataa sisältäviä muuttujia. Muuttujien lisäksi luokkaan voidaan määritellä metodeja, jotka sisältävät toiminnallisuutta ja muokkaavat dataa. Luokista luodut instanssit, eli objektit, saavat käyttöönsä luokassa määritellyt ominaisuudet ja toiminnallisuudet. (Caceres 2016.)

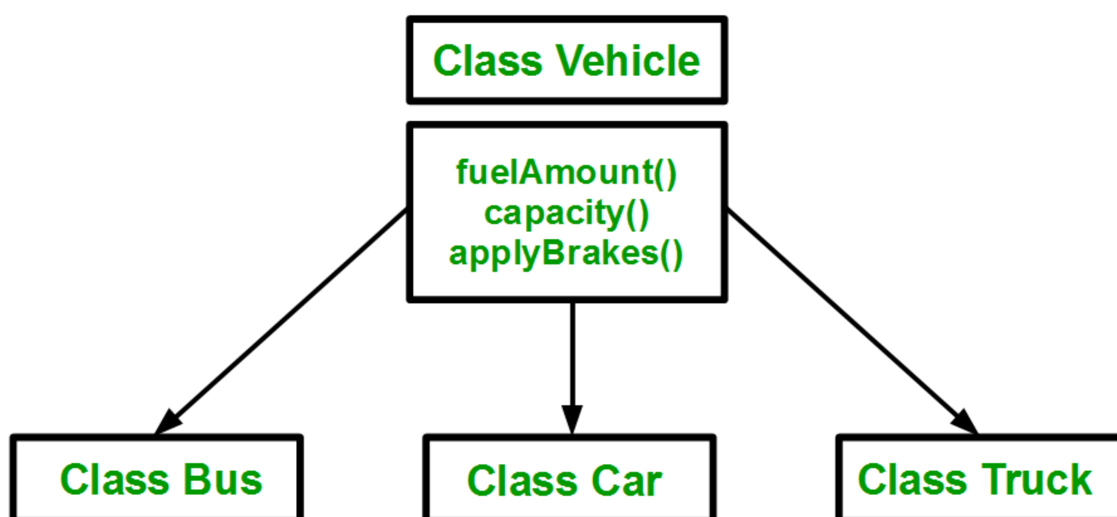
3.2 Kapselointi

Ohjelmoinnissa luokat ovat usein riippuvaisia toisistaan. Toisin sanoen joku luokka saattaa tarvita tietoa tai toiminnallisuuksia toisesta luokasta toimiakseen. Varsinkin isommissa projekteissa näistä riippuvaisuuksista saattaa syntyä odottamattomia ongelmia, kun luokkiin tehdään muutoksia. Kapseloinnilla pyritään välttämään riippuvaisuuksista syntyviä ongelmia. Kapselointi viittaa koodin suunnittelemiseen modulaarisesti niin, että kaikki osat ovat mahdollisimman erotettu toisistaan. Kapselointi yksinkertaisimmillaan tapahtuu käyttämällä termejä: ’public’, ’protected’ ja ’private’. Public -termillä merkityt muuttujat ja metodit ovat julki-

sia, jolloin niihin päästään käsiksi myös objektin/luokan ulkopuolelta. Private elementit ovat yksityisiä, eli niitä ei voi käsitellä luokan ulkopuolelta. 'Protected' elementteihin pääsee käsiksi luokan sisällä, mutta myös luokasta periytetyissä luokissa. (Luke 2016.)

3.3 Periytyminen

Olio-ohjelmoinnissa periytymisellä tarkoitetaan jonkin luokan ominaisuuksien tai metodien periyttämistä niin, että niihin päästään käsiksi ja niitä voidaan käyttää perivässä luokassa. Periyttämällä voidaan siis luoda uusia luokkia, jotka pohjautuvat toiseen olemassa olevaan luokkaan (kuva 2). Tämän ansiosta vältytään esimerkiksi kirjoittamasta samankaltaisia metodeja useaan kertaan. Yläluokkaa, jonka ominaisuuksia periytetään, kutsutaan base classiksi. Alaluokka, joka perii ominaisuuksia base classilta kutsutaan sub classiksi. Sub classit voivat olla johdettu joko yhdestä tai useammasta base classista. Mikäli luokka on johdettu useasta base classista, se saa käyttöönsä kaikkien base classien ominaisuudet. Luokka voi myös periä toisen luokan, joka taas on perinyt kolmannen luokan jne. Base classeilta perittyjä toiminnallisuuksia voidaan muokata tai toteuttaa kokonaan uudelleen sub classeissa. (Educba n.d.)



KUVA 2. Periytymistä kuvaava diagrammi (Agarwal n.d.)

3.4 Polymorfismi

Polymorfismi on kreikkalainen sana, joka tarkoittaa monimuotoista. Käytännössä polymorfismi on sitä, kun peritystä luokasta luotua objektia käsitellään kuin se olisi samantyyppinen kuin sen perimä luokka. Tätä voidaan käyttää hyödyksi esimerkiksi metodien parametreissä tai taulukoissa. Kun polymorfismia tapahtuu, objektille annettu tyyppi muuttuu ajon aikana, jolloin objektin ajonaikainen tyyppi on eri kuin sen luomisen yhteydessä julistettu tyyppi. Kun ajon aikana kutsutaan perittyjen luokkien metodeja, CLR tarkistaa kyseisten objektien ajonaikaisen tyypin ja suorittaa metodin tämän mukaan. Jos siis ajonaikana kutsutaan jonkin base classin metodia, tämän base classin toteutuksen sijaan suoritetaan sub classin toteutus, mikäli metodi on yliajettu sub classissa. (Microsoft 2015.)

3.5 Abstraktio

Margaret Rousen (2014) mukaan abstraktiolla tarkoitetaan prosessia, jossa otetaan pois ominaisuuksia joltain niin, että jäljelle jää olennaisista ominaisuuksista koostuva setti. Abstraktion avulla voidaan vähentää monimutkaisuutta piilottamalla yksityiskohtia, kuten metodeja, jotka ovat käyttäjälle tarpeettomia. Abstraktion päälle on helpompi luoda monimutkaistakin logiikkaa, kun piilotettuja yksityiskohtia ei tarvitse ymmärtää tai välttämättä edes ajatella. Lisäksi abstraktio vähentää muutosten vaikutusta, kun sisäisiin metodeihin tehdään muutoksia, sillä niihin ei vaikuteta luokan ulkopuolelta. Abstraktiosta syntyvä objekti on representaatio alkuperäisestä objektista, jossa ei-halutut yksityiskohdat ja tiedot on piilotettu. (Janssen 2019.)

3.6 OOP Unityssä

Vaikka Unity käyttääkin OOP-mallia, työnkulku Unityssä on hieman erilaista, sillä se rakentuu hyvin pitkälti komponenttien ympärille. Komponenttien voidaan ajatella olevan yksittäisiä koneiston osia, joilla on omat sisäisesti hoidetut toiminnallisuudet ja tehtävät. Komponentteja voi hyödyntää useakin eri ”koneisto” ja niitä

voidaan liittää erilaisiin järjestelmiin, jotta ne voivat toteuttaa oman tehtävänsä. (Porter 2013.)

Unityssä kaikki pelimaailmaan luotavat objektit periytyvät GameObject -luokasta. Näin ollen ne saavat käyttöönsä kaikki GameObject -luokan toiminnallisuudet ja ominaisuudet, kuten transformin, joka sisältää tiedon objektin sijainnista, rotaatiosta ja skaalasta. Transformin lisäksi objekteihin voidaan liittää useita erilaisia komponentteja. Unity tarjoaa lukuisia valmiita komponentteja, mutta oleellinen osa pelinkehitystä Unityllä on omien komponenttien toteuttaminen. Unityn valmiisiin komponentteihin kuuluu esimerkiksi Collider -komponentti, joka hoitaa objektin törmäystentarkastelun. Kaikki objektit, joihin on liitetty kyseinen komponentti voivat törmäillä keskenään. Komponentit ovat luokkia, jotka periytyvät Unityn MonoBehaviour -luokasta. Luotuja objekteja ja niiden komponentteja voi tarkastella ja muokata Unityn editorin Inspector -ikkunassa. (Learn to create games 2017.)

4 DOTS

Unityn pohjaa uudelleenrakennetaan uudella DOTS-teknologiapaketilla (Data-oriented technology stack). Se tulee muuttamaan Unityn ydin perustan ja sen, miten Unityllä tehdään pelejä. DOTSin avulla pelit pystyvät hyödyntämään nykyaikaisten prosessorien useita ytimiä ilman, että pelinkehittäjän tarvitsee itse kirjoittaa monimutkaista koodia. Se koostuu kolmesta pääominaisuudesta: Entity Component Systemistä, C# Job Systemistä sekä Burst Compilerista. Näiden ominaisuuksien avulla pelinkehittäjä saa käyttöönsä ympäristön, jolla kirjoittaa monisäikeistä koodia ja parantaa pelin suorituskykyä huomattavasti. (Unity Technologies 2019b.)

DOTSin monisäikeisten järjestelmien avulla voidaan kehittää pelejä, jotka pyörivät tehokkaasti useilla erilaisilla laitteistoilla. Se mahdollistaa laajempien ja rikkaampien maailmojen sekä yhä monimutkaisempien ja kehittyneempien simulaatioiden luomisen. DOTSin avulla voidaan myös esimerkiksi parantaa mobiililaitteiden akun kestoa ja vähentää laitteen kuumentumista. DOTSin myötä Unity siirtyy olio-ohjelmoinnin mallista dataorientoituneeseen malliin, jolloin koodi on helpommin kierrätettävää ja helppolukuisempaa. (Unity Technologies 2019b.)

4.1 Dataorientoitunut malli

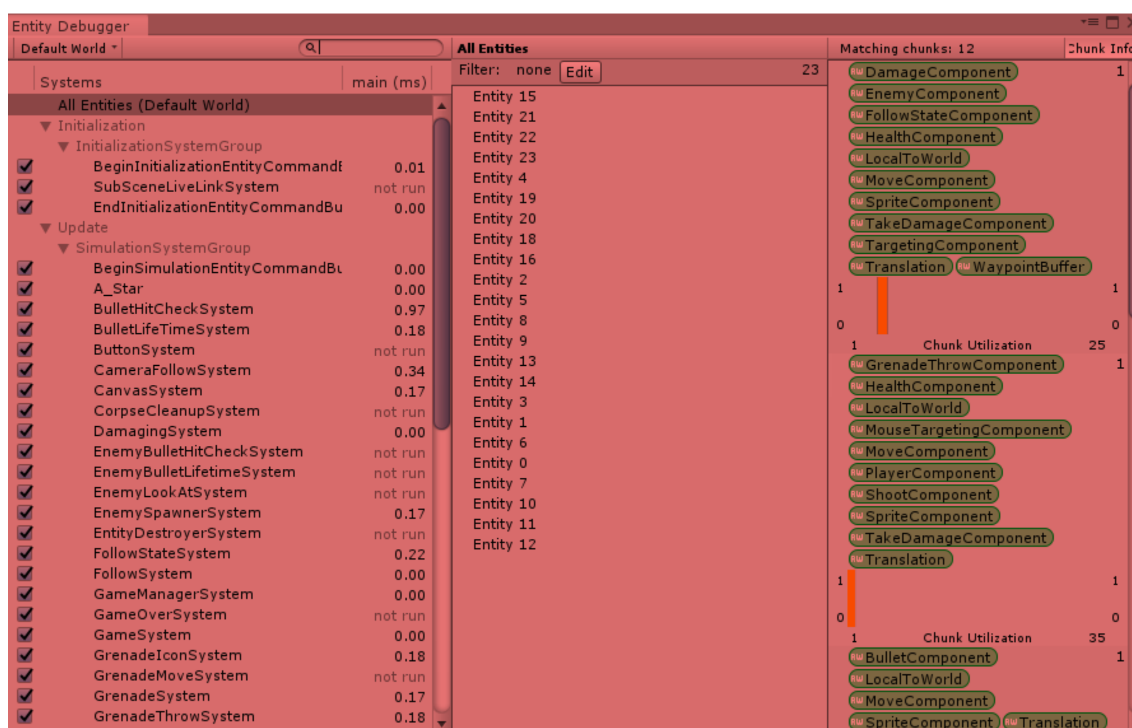
Kun puhutaan tietojenkäsittelystä, dataorientoitunut malli (DOD tai DOP) on lähestymistapa, jossa objektien sijaan keskitytään dataan itsessään. Dataorientoituneessa mallissa keskitytään erityisesti datan asetteluun ja siihen, kuinka tätä dataa muokataan ja käsitellään. Kentät erotellaan ja järjestellään muistiin sen mukaan, milloin niitä tarvitaan. (Noel 2009.)

Tietokoneet käsittelevät parhaiten homogeenistä ja peräkkäistä dataa, joka on lohkoissa. Jos data on aseteltu muistiin sinne tänne, muistiin viittaaminen ja datan hakeminen on huomattavasti hitaampaa. Dataorientoitunut suunnittelumalli erottelee datan toiminnallisuuksista. Sen avulla prosessorit toimivat paremmin, sillä data on järjestelty muistiin optimaalisesti. (Sefton 2016.)

4.2 Entity Component System

Entity Component System (ECS) on Unity DOTSin ydin, ja se koostuu kolmesta osasta: entiteetit, komponentit ja systeemit. Entity Component System ei ole varsinaisesti mikään uusi teknologia, vaan se on rakenteellinen malli. ECS erottelee identiteetin, datan ja toiminnallisuuden toisistaan. Identiteettejä kutsutaan entiteeteiksi, data on komponenteissa ja toiminnallisuudet ovat systeemeissä. ECS-malli keskittyy ensisijaisesti dataan. Systeemit käsittelevät ja muuntavat dataa hakemalla komponentteja, joita entiteetit indeksoivat. (Unity Technologies 2019c.)

Unityn editorissa entiteettejä, systeemejä ja komponentteja voidaan visualisoida käyttämällä Entity Debuggeria (kuva 3). Entity Debugger avataan navigoimalla editorin valikossa: Window -> Analysis -> Entity Debugger. Systeemit listassa näkyvät kaikki projektin systeemit. Systeemejä voidaan sammuttaa tai käynnistää listan avulla, ja siinä näkyy muun muassa kuinka paljon aikaa minkäkin systeemin ajamiseen käytetään. Yksityiskohdat kohdasta voi tarkastella mitä komponentteja mikäkin systeemi käsittelee, sekä mihin entiteetteihin näitä komponentteja on liitetty. Entiteettien komponenttien dataa voidaan tarkastella Inspector-ikkunassa valitsemalla entiteetti listasta. (Unity Technologies 2019c.)



KUVA 3. Entity Debugger.

4.2.1 Entiteetit

Entiteetit edustavat yksittäisiä asioita pelissä. Entiteeteillä itsessään ei ole minkäänlaista toiminnallisuutta tai dataa, vaan ne kertovat mitkä osat datasta kuuluvat yhteen. Entiteetit ovat ikään kuin id:itä. Niiden voi myös kuvitella olevan erittäin kevyitä GameObjecteja joilla ei ole tyyppiä tai edes nimeä. Entiteettien avulla voidaan säilyttää viittauksia komponentteihin ja toisiin entiteetteihin. Entiteettien iterointi niiden komponenttien mukaan on oleellinen osa ECS-mallia. (Unity Technologies 2019c.)

Entiteettejä voidaan kategorisoida ja jakaa ryhmiin niihin liitettyjen komponenttien avulla, vaikka entiteeteillä ei olekaan tyyppiä. EntityManager pitää kirjaa kaikista uniikeista yhdistelmistä komponentteja, joita entiteetteihin liitetään. Tällaista komponentti yhdistelmää kutsutaan arkkityypiksi. EntityManager luo automaattisesti EntityArchetypen aina kun entiteetille annetaan komponentteja. Näitä EntityArchetypejä voidaan luoda myös manuaalisesti ja niitä voidaan käyttää uusien entiteettien luomiseen, jolloin uudet entiteetit ovat yhdenmukaisia arkkityypin kanssa. EntityManager hallitsee kaikkia entiteettejä. Se pitää yllä listaa kaikista entiteeteistä ja järjestelee niihin liitettyä dataa niin, että saavutetaan mahdollisimman korkea suorituskyky. (Unity Technologies 2019c.)

4.2.2 Komponentit

Komponentit ovat dataa, eivätkä ne sisällä mitään toiminnallisuuksia. Käytännössä komponentit ovat tietueita (struct), jotka toteuttavat jonkun komponentteihin vaadittavista rajapinnoista. Erityyppisillä komponenteilla on kaikilla oma käyttötarkoituksensa. Komponentti tietueet eivät voi pitää sisällään viittauksia hallittuihin objekteihin, sillä kaikki komponenttidata tallennetaan Chunk-muistiin, jossa ei ole käytössä roskan keruuta. (Unity Technologies 2019c.)

Yleiskomponentit (General Purpose Component) ovat yleisimmin käytetty komponenttityyppi. Yleiskomponentit pitävät sisällään ainoastaan instanssidataa entiteeteille. Jaetut komponentit (Shared Component) ovat erityinen komponentti tyyppi ja niitä käytetään, kun entiteeteillä on jotain yhteistä. Jaetut komponentit

jakavat entiteettejä alaluokkiin. Entiteetit jaetaan alaluokkiin niiden jaetuissa komponenteissa olevien arvojen mukaan, ottaen huomioon myös entiteettien arkkityypit. EntityManager asettaa kaikki entiteetit, joilla on samat jaetut data-arvot samaan lohkoon Chunk-muistissa. Jaettujen komponenttien dataa on mahdollista prosessoida yhtä-aikaisesti systeemeissä. Mikäli entiteeteiltä poistetaan tai niille lisätään komponentteja, tai niille annetun jaetun komponentin arvoja muutetaan, EntityManager siirtää entiteetin toiseen lohkoon muistissa. EntityManager voi luoda uuden lohkon tarpeen vaatiessa. Systeemintilakomponentti (System State Component) ja jaettu systeemintilakomponentti (System State Shared Component) toimivat kuten yleiskomponentti ja jaettu komponentti, mutta ne mahdollistavat systeemin sisäisten resurssien seuraamisen. Systeemintilakomponentin sisäisiä resursseja voidaan luoda ja tuhota tarpeiden mukaan käyttämättä erillisiä takaisinkutsuja. Dynaamisten puskurikomponenttien (Dynamic Buffer Component) avulla entiteeteille voidaan antaa erikokoisia muistipuskureita. Ne ovat komponentteja, jotka sisältävät sisäisen kapasiteetin tietylle määrälle elementtejä. Mikäli sisäinen kapasiteetti ylittyy, dynaaminen puskurikomponentti voi varata itselleen lohkon heap-muistista. EntityManager hallitsee dynaamisia puskurikomponentteja niin, että kaikki heap-muistista varattu muisti vapautetaan automaattisesti, mikäli komponentti poistetaan. Lohkokomponentit (Chunk Component) pitävät sisällään dataa, joka pätee kaikkiin entiteetteihin tietyssä lohkossa. Mikäli lohkokomponentissa olevaa arvoa muokataan käyttämällä entiteettiä, joka kuuluu tiettyyn lohkoon, lohkon muidenkin entiteettien saman lohkokomponentin arvot muuttuvat. (Unity Technologies 2019c.)

4.2.3 Systeemit

Systeemit sisältävät kaiken pelin toiminnallisuuden. Toisin sanoen systeemeissä on kaikki logiikka, joka muuttaa ja käsittelee dataa. Systeemit käyvät läpi kaikkia entiteettejä ja hakevat niistä tiettyjä komponentteja, joita ne tarvitsevat toimiakseen. Entiteetit, tai niiden komponentit, joita haetaan, määritellään EntityQuery-tietueessa. Kun entiteetit tietyillä komponenteilla on löydetty, niitä iteroidaan ja komponenttien sisältämää dataa käsitellään. Yksittäinen systeemi on aina vas-

tuussa kaikkien sen hakemien entiteettien komponenttien datan käsittelystä. Entiteettien hakeminen ja iterointi systeemeissä on todella nopeaa datan muistiin järjestelyn vuoksi.

Unityn Entity Component System tarjoaa useita erilaisia systeemeitä. Kaksi yleisintä systeemiä ovat ComponentSystem sekä JobComponentSystem. Systeemit tarjoavat takaisinkutsufunktioita, kuten OnCreate ja OnUpdate, samaan tapaan kuin Unityn perinteinen MonoBehaviour-luokka. Näihin funktioihin voidaan tehdä implementaatioita, jotka ajetaan koodissa tiettyihin aikoihin. Syteemien takaisinkutsufunktioita kutsutaan ja ajetaan pääsäikeessä. JobComponentSystemeissä tehtäviä aikataulutetaan tyypillisesti OnUpdate-funktiossa. Tehtävät, tai Jobit, joita luodaan ja aikataulutetaan ajetaan työläissäikeissä. JobComponentSystemeillä saavutetaan tyypillisesti paras suorituskyky, sillä ne käyttävät hyväksi prosessorien useita ytimiä ja ne voidaan kääntää käyttämällä Burst Compileria. (Unity Technologies 2019c.)

Unity tunnistaa automaattisesti projektiin luodut systeemit ja instantioi ne, kun koodi ajetaan, ellei automaattista luontia ole erikseen kielletty attribuutilla. Systeemit järjestetään ryhmittäin ja kehittäjä voi hallita tätä ryhmittämistä. Oletuksena kaikki systeemit lisätään samaan SimulationSystemGroup-ryhmään. Systeemin päivityssilmukan ohjaamisesta vastaa ComponentSystemGroup, johon systeemi kuuluu. Nämä ryhmät ovat itsekin eräänlaisia systeemeitä. Vaikka systeemien päivityksestä vastaavatkin ComponentSystemGroupit, pystyy kehittäjä silti vaikuttamaan systeemien päivitysjärjestykseen käyttämällä erilaisia attribuutteja. (Unity Technologies 2019c.)

4.3 C# Job System

C# Job System mahdollistaa nykyaikaisten prosessorien ydinten hyödyntämisen niin, että useaa tehtävää voidaan ajaa samanaikaisesti. Yksi Unityn perinteisen mallin suurimpia ongelmia suorituskyvyn kannalta on se, että Unityn Update-silmukka pyörii ainoastaan yhdessä säikeessä. Monisäikeisen koodin kirjoittaminen on tähän asti ollut hankalaa, sillä siinä saattaa syntyä paljon odottamattomia ongelmia, joita voi olla vaikea ratkaista. Yksi suurimpia ongelmia monisäikeisen

koodin kirjoittamisessa on kilpailutilanteet, eli missä järjestyksessä säikeitä ajetaan. Mikäli kirjoittaa monisäikeistä koodia itse, täytyy myös itse hallita säikeiden määrää ja sitä, mitä mikin säikeistä tekee. C# Job Systemin avulla kehittäjä voi säikeiden sijaan kirjoittaa tehtäviä ja C# Job System hoitaa tehtävien ajamisen sekä kaikki säikeiden hallintaan liittyvät asiat automaattisesti. (Code Monkey 2019.)

4.4 Burst Compiler

Code Monkey (2019) kuvailee Burst Compilerin olevan uskomaton, melkein taianomainen pala teknologiaa. Se kääntää kirjoitetun C#-koodin todella optimoiduksi konekieleksi. Se toimii erityisen hyvin yhdessä Jobien kanssa. Burst Compiler myös hyödyntää tiettyjä optimisaatioita riippuen siitä, mille alustalle peliä kehitetään. (Code Monkey 2019.)

5 Toteutus

Tämän opinnäytetyön yhteydessä tuotettu peliprototyyppi toteutettiin Unityn 2019.1.10f1 -versiolla. Unity DOTSia on mahdollista käyttää yhdessä perinteisen Unityn kanssa (Hybrid ECS), mutta tämä projekti haluttiin toteuttaa puhdasta Entity Component Systemiä käyttäen (Pure ECS). Vain hyvin pieni osa projektista tehtiin käyttämällä perinteisiä GameObjecteja ja MonoBehaviour-luokkia. Peli tehtiin Microsoft Windows -alustalle ja se on Top-down Shooter -tyylinen 2D-peli. Projektin ideana oli päästä testaamaan käytännössä Unity DOTSia ja sen kolmea pääkomponenttia.

5.1 DOTS-ominaisuuksien käyttöönotto

Unity DOTSin ominaisuudet eivät ole käytettävissä oletuksena, vaan ne täytyy ottaa käyttöön manuaalisesti. Tämä onnistuu kuitenkin vaivattomasti Unity-editorin sisällä. DOTS-teknologiat saa käyttöönsä Package Managerin kautta: Windows -> Package Manager. Package Managerin ylävalikon Advanced-kohdasta täytyy valita kohta ”Show preview packages”, jotta DOTSiin vaadittavat paketit saadaan näkyville listaan. Tämän jälkeen listasta valitaan paketit, jotka projektiin halutaan tuoda. Saadakseen täyden hyödyn DOTSista on asennettava jokainen relevantti paketti, mutta osaa niistä voi käyttää myös yksittäin. DOTSiin kuuluvat paketit ovat: Burst, Collections, Entities, Hybrid Renderer, Jobs sekä Mathematics.

Entities-paketti tuo projektiin Entity Component System -implementaation. Burst-paketin avulla käyttöön saadaan Burst Compiler ja Jobs-paketti tarjoaa lisää Job-tyyppejä. Hybrid Renderer -paketti tarjoaa systeemeitä ja komponentteja, joiden avulla voidaan piirtää meshejä käyttäen DOTSia. Collections-paketti sisältää natiivikokoelmia, kuten NativeList, ja Mathematics-paketin mukana tulee erittäin optimoitu matematiikkakirjasto, joka tarjoaa esimerkiksi uuden float3-tyypin, joka vastaa perinteistä Vector3:sta.

5.2 Pelin tilat

Pelit koostuvat yleensä useista eri tiloista. Tyypillisesti pelissä on ainakin Päävalikko ja pelitila, ja nämä ovat useimmiten erillisiä Scenejä. Sceneä voidaan vaihtaa käyttämällä Unityn SceneManager-luokkaa, jolloin kaikki edellisessä scenessä olleet objektit tuhotaan ja uuden Scenen objektit ladataan automaattisesti. Tässä projektissa kaikki tilat ovat samassa Scenessä ja tiloja vaihdetaan tuhoamalla ja luomalla entiteettejä manuaalisesti.

5.2.1 Päävalikko

Päävalikko (kuva 4) on ensimmäinen tila, joka tulee näkyville, kun pelin käynnistää. Päävalikossa on kolme nappia, joista voi käynnistää pelin, eli siirtyä pelitilaan, avata pelin ohjeet, tai sulkea pelin.

KUVA 4. Pelin päävalikko.

Tavallisesti päävalikko toteutettaisiin hyödyntämällä Unityn valmiita UI-työkaluja ja komponentteja. Tässä projektissa se ei kuitenkaan ollut mahdollista, sillä peli haluttiin toteuttaa käyttäen puhdasta ECS:ää. Päävalikon kaikki ominaisuudet, eli napit ja elementtien renderointi ja sijoittelu on siis toteutettu itse koodissa.

5.2.2 Pelitila

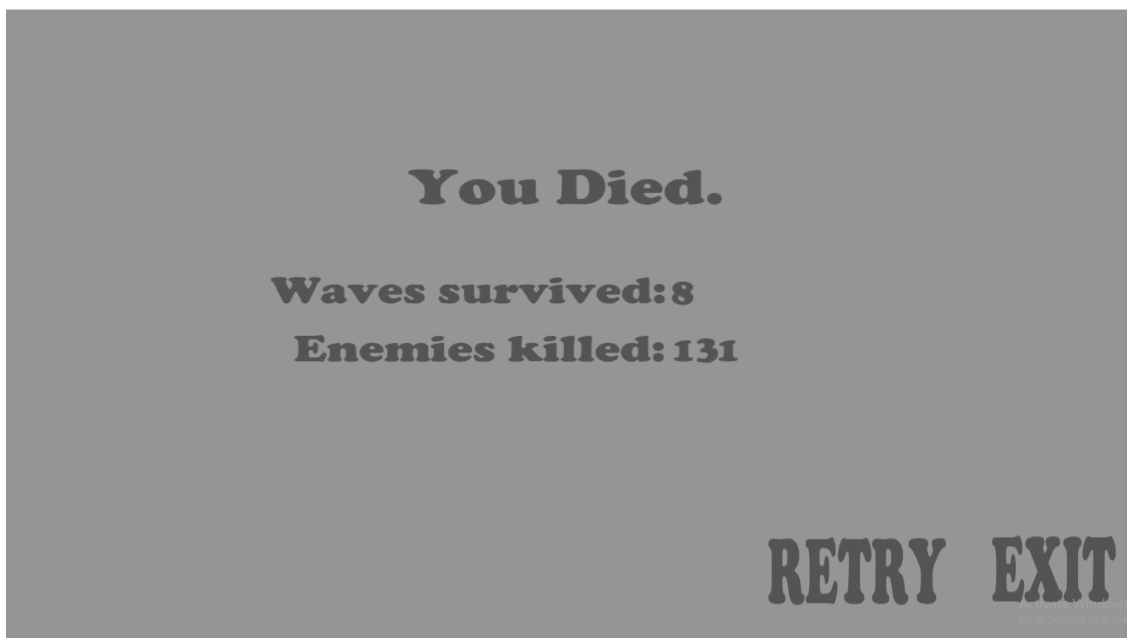
Pelissä pelaajan on tarkoitus selviytyä vihollisaalloista mahdollisimman pitkään. Pelaaja ohjaa yksittäistä hahmoa, joka voi liikkua, ampua ja heittää kranaatteja. Pelin edetessä pelikentälle luodaan vihollisia, jotka pyrkivät vahingoittamaan pelaajaa (kuva 5). Mikäli pelaajan terveyspisteet laskevat nollaan, pelaajan hahmo kuolee ja peli on ohi. Pelaaja voi tuhota vihollisia ampumalla tai käyttämällä kranaatteja. Pelikentälle luodaan satunnaisesta aseita ja parannuksia, joita pelaaja voi poimia. Mitä pidemmälle peli etenee, sitä enemmän ja haastavampia vihollisia syntyy.

KUVA 5. Pelin pelitila.

UI-työkalujen tapaan Unity tarjoaa valmiita komponentteja ja ominaisuuksia myös itse pelin rakentamiseen. Valmiita ominaisuuksia, joita usein käytetään hyödyksi ovat esimerkiksi fysiikat ja törmäysten tarkastelu sekä animointi. Tässä projektissa nämä ovat implementoitu itse, sillä Entity Component System ei niitä tarjoa. Myös renderointi tehtiin itse, vaikka se onkin mahdollista hoitaa käyttämällä hyödyksi Hybrid Renderer -pakettia, sillä ECS:n tarjoaman renderoinnin kanssa ei ole mahdollista käyttää spritesheettejä ja frame-by-frame -animointia.

5.2.3 Peli ohi -tila

Peli ohi -tilaan siirrytään pelitilasta, kun pelaaja kuolee. Peli ohi -tilassa pelaaja voi nähdä kuinka monta vihollisaaltoa hän selvisi, sekä kuinka monta vihollista pelaaja tuhosi pelin aikana (kuva 6). Peli ohi -tilasta pelaaja voi nappeja painamalla joko aloittaa pelin uudelleen, tai palata päävalikkoon. Peli ohi -tila on toteutettu samalla tavalla kuin päävalikko.



KUVA 6. Pelin peli ohi -tila.

5.3 Koodi

Koska puhtaassa ECS:ssä ei käytetä MonoBehaviour-komponentteja, alustus arvojen antaminen komponenteille ei onnistu tavalliseen tapaan editorin kautta. Projektin assetteihin ei myöskään voida antaa viittauksia editorissa normaalisti, joten projektissa käytettiin hyödyksi ScriptableObjecteja, sekä staattisia apuluokkia. Apuluokkien ja ScriptableObjectien avulla alustus arvoja voitiin antaa komponenteille editorin kautta. Kaikkien apuluokkien ja ScriptableObjectien toteutukset ovat lähestulkoon identtiset. Apuluokan GetConfig-metodia (kuva 7) kutsutaan lähettämällä parametrinä enum-arvo, joka kertoo mikä tietorakenne (kuva 8) halutaan hakea.

```
public class EnemyHelper
{
    private static EnemyConfig _enemyConfig;

    public static void Init()
    {
        _enemyConfig = Resources.Load<EnemyConfig>("EnemyConfig");
    }

    public static EnemyConfig.EnemyData GetConfig(EnemyConfig.EnemyType type)
    {
        for (int i = 0; i < _enemyConfig.enemyDatas.Count; i++)
        {
            if (_enemyConfig.enemyDatas[i].enemyType == type)
            {
                return _enemyConfig.enemyDatas[i];
            }
        }

        Debug.LogError("EnemyHelper: Enemy config of type: " + type + " was not found.");
        return new EnemyConfig.EnemyData();
    }
}
```

KUVA 7. Esimerkki apuluokasta (EnemyHelper)

```
[CreateAssetMenu(fileName = "EnemyConfig", menuName = "Create/EnemyConfig", order = 4)]
public class EnemyConfig : ScriptableObject
{
    public enum EnemyType...

    public enum EnemyAttackType...

    public enum EnemyGunType...

    public List<EnemyData> enemyDatas;

    [Serializable]
    public struct EnemyData
    {
        public string name;
        public EnemyType enemyType;
        public EnemyAttackType enemyAttackType;

        public int health;
        public int damage;
        public float attackCooldown;
        public float movementSpeed;
        public float attackRange;

        [ConditionalHideAttribute("enemyAttackType", EnemyAttackType.Ranged)]
        public EnemyGunType gunType;
        [ConditionalHideAttribute("enemyAttackType", EnemyAttackType.Ranged)]
        public AudioConfig.AudioType audioType;
        [ConditionalHideAttribute("enemyAttackType", EnemyAttackType.Ranged)]
        public int projectileCount;
        [ConditionalHideAttribute("enemyAttackType", EnemyAttackType.Ranged)]
        public float projectileSpeed;
        [ConditionalHideAttribute("enemyAttackType", EnemyAttackType.Ranged)]
        public float projectileLifeTime;
    }
}
```

KUVA 8. Esimerkki ScriptableObjectista (EnemyConfig)

5.3.1 Renderointi

Jokainen pelissä näkyvä asia renderoidaan samaa renderointisysteemiä käyttäen. Kuten mainittu, renderointi olisi voitu hoitaa hyödyntämällä Hybrid Renderer -pakettia, mutta jotta pelin elementtejä saatiin animoitua, jouduttiin renderointi implementoimaan itse (kuva 9).

```
protected override void OnUpdate()
{

    Entities.ForEach(( ref Translation translation, ref SpriteComponent spriteComponent) =>
    {
        AnimationConfig.AnimationData animationData = AnimationHelper.GetConfig(spriteComponent.spriteType);
        MaterialPropertyBlock materialPropertyBlock = new MaterialPropertyBlock();

        materialPropertyBlock.SetVectorArray("_MainTexx_UV", new Vector4[] { spriteComponent.uv });
        materialPropertyBlock.SetColor("_Color", spriteComponent.color);

        Graphics.DrawMesh(
            quadMesh,
            spriteComponent.matrix,
            animationData.animations[(int)spriteComponent.currentAnimation],
            0,
            Camera.main,
            0,
            materialPropertyBlock);

    });
}
```

KUVA 9. SpriteSheetRenderer.

SpriteSheetRenderer-luokan OnUpdate-metodi hakee entiteettejä, joilta löytyy Translation- ja SpriteComponent-komponentit. Se luo materiaaliobjektin SpriteComponent-komponentin (kuva 10) arvojen pohjalta ja kutsuu metodia, joka piirtää entiteetin ruudulle. Translation on valmis komponentti, joka löytyy Unityn.Transforms -kirjastosta. Se sisältää float3-muuttujan, joka kuvaa entiteetin sijaintia.

```
public struct SpriteComponent : IComponentData
{
    public AnimationConfig.SpriteType spriteType;
    public AnimationConfig.AnimationType currentAnimation;

    public int frameCount;
    public float currentFrame;
    public float frameTimer;
    public float frameTimerMax;
    public Vector3 scale;
    public Quaternion rotation;
    public bool LoopingAnimation;
    public bool shouldAnimate;

    public Vector4 uv;
    public Matrix4x4 matrix;
    public Color color;
}
```

KUVA 10. SpriteComponent.

5.3.2 Animointi

Pelin elementtien animoinnista vastaa SpriteAnimationSystem (kuva 11). Myös SpriteAnimationSystem hakee entiteettejä, joilta löytyy Translation- ja Sprite-Component-komponentit. Se muuttaa SpriteComponent-komponentin arvoja niin, että renderointi-systeemi tietää, mikä osa kuvasta milloinkin tulee piirtää.

```
public class SpriteAnimationSystem: ComponentSystem
{
    protected override void OnUpdate()
    {
        float time = Time.deltaTime;

        Entities.ForEach(( ref SpriteComponent spriteComponent, ref Translation translation ) =>
        {
            if(spriteComponent.shouldAnimate) {
            spriteComponent.frameTimer += time;
                if(spriteComponent.frameTimer >= spriteComponent.frameTimerMax)
                {
                    spriteComponent.frameTimer -= spriteComponent.frameTimerMax;

                    if(spriteComponent.LoopingAnimation)
                        spriteComponent.currentFrame = (spriteComponent.currentFrame + 1) % spriteComponent.frameCount;
                    else if(spriteComponent.currentFrame < spriteComponent.frameCount - 1)
                        spriteComponent.currentFrame += 1;
                }
            }

            float uvWidth = 1f / spriteComponent.frameCount;
            float uvHeight = 1;
            float uvOffsetX = uvWidth * spriteComponent.currentFrame;
            float uvOffsetY = 0;
            spriteComponent.uv = new Vector4(uvWidth, uvHeight, uvOffsetX, uvOffsetY);

            spriteComponent.matrix = Matrix4x4.TRS(translation.Value, spriteComponent.rotation, spriteComponent.scale);

        });
    }
}
```

KUVA 11. SpriteAnimationSystem.

5.3.3 UI

Projektissa kaikki UI-elementit ovat aivan kuten mikä tahansa muukin entiteetti, jota ruudulla näkyy. Kaikki teksti ja numerot jotka näkyvät pelissä ovat kuvan-muokkausohjelmalla tehtyjä kuvia, sillä Entity Component System ei tarjoa min-käänlaisia tekstityökaluja.

Peliin luotiin komponentteja ja systeemeitä, joiden avulla ruudulle sijoitellaan UI-elementtejä ja niiden sijaintia päivitetään niin, että ne eivät jää ruudun ulkopuo-lelle, vaikka kameraa liikutetaan. CanvasSystem (kuva 12) imitoi Unityn Canvas-komponenttia. Se hakee entiteettejä, joilla on CanvasComponent-komponentti, ja laskee niille positioita ruudun reunoilta. UIPositioningSystem (kuva 13) päivit-tää UI-entiteettien sijaintia hakemalla ja hyödyntämällä arvoja entiteetin Canvas-Component-komponentista.

```
public class CanvasSystem : ComponentSystem
{
    public enum Anchor...

    protected override void OnStartRunning()
    {
        World.Active.EntityManager.CreateEntity(typeof(CanvasComponent), typeof(Translation), typeof(LocalToWorld));
    }

    protected override void OnUpdate()
    {
        if (CameraHelper.cameraHolder == null) return;

        float3 camPos = CameraHelper.cameraHolder.position;

        Entities.ForEach((ref CanvasComponent canvasComponent) =>
        {
            if (CameraHelper.camera == null) return;

            float cameraHeight = CameraHelper.camera.orthographicSize * 2;
            float cameraWidth = cameraHeight * CameraHelper.camera.aspect;

            canvasComponent.height = cameraHeight;
            canvasComponent.width = cameraWidth;
            canvasComponent.middle = new float3(camPos.x, camPos.y, 0);
            canvasComponent.topRight = new float3(camPos.x + cameraWidth / 2, camPos.y + cameraHeight / 2, 0);
            canvasComponent.topLeft = new float3(camPos.x - cameraWidth / 2, camPos.y + cameraHeight / 2, 0);
            canvasComponent.bottomRight = new float3(camPos.x + cameraWidth / 2, camPos.y - cameraHeight / 2, 0);
            canvasComponent.bottomLeft = new float3(camPos.x - cameraWidth / 2, camPos.y - cameraHeight / 2, 0);
        });
    }
}
```

KUVA 12. CanvasSystem.

```
public class UIPositioningSystem : ComponentSystem
{
    protected override void OnUpdate()
    {
        CanvasComponent canvasComp = default;

        Entities.ForEach((ref CanvasComponent canvasComponent) =>
        {
            canvasComp = canvasComponent;
        });

        Entities.ForEach((ref UIComponent uiComponent, ref Translation translation, ref SpriteComponent spriteComponent) =>
        {
            switch (uiComponent.anchor)
            {
                case CanvasSystem.Anchor.Middle:
                    translation.Value = canvasComp.middle + uiComponent.position;
                    break;

                case CanvasSystem.Anchor.TopLeft:
                    translation.Value = canvasComp.topLeft + uiComponent.position;
                    break;

                case CanvasSystem.Anchor.TopRight:
                    translation.Value = canvasComp.topRight + uiComponent.position;
                    break;

                case CanvasSystem.Anchor.BottomLeft:
                    translation.Value = canvasComp.bottomLeft + uiComponent.position;
                    break;

                case CanvasSystem.Anchor.BottomRight:
                    translation.Value = canvasComp.bottomRight + uiComponent.position;
                    break;

                default:
                    translation.Value = canvasComp.middle + uiComponent.position;
                    break;
            }
        });
```

KUVA 13. UIPositioningSystem.

Pelin kaikki UI-elementit luodaan UICreatorSystem-luokassa. Luokassa on EntityArchetype-muuttujat kaikentyyppisille UI-entiteeteille. Lisäksi luokassa on natiivilistat kaikille UI-entiteeteille (kuva 14).

```
public class UICreatorSystem : ComponentSystem
{
    public EntityArchetype healthBarArchetype;
    public EntityArchetype genericUIArchetype;
    public EntityArchetype weaponIconArchetype;
    public EntityArchetype grenadeIconArchetype;
    public EntityArchetype mainMenuBackgroundArchetype;
    public EntityArchetype buttonArchetype;

    private EntityManager _entityManager;

    private NativeList<Entity> menuEntities;
    private NativeList<Entity> instructionsEntities;
    private NativeList<Entity> inGameEntities;
    private NativeList<Entity> inGameCurrentWaveEntities;
    private NativeList<Entity> inGameEnemiesRemainingEntities;
    private NativeList<Entity> gameOverScreenEntities;

    private EnemySpawnerSystem enemySpawnerSystem;
```

KUVA 14. UICreatorSystem-luokan muuttujat.

Kaikille UI-elementeille luodaan arkkityypit luokan OnCreate-metodissa (kuva 15), jotta niitä voidaan käyttää aina kun entiteettejä luodaan. CreateArchetype-metodille annettaan parametrinä komponentteja, joita arkkityypistä luotuihin entiteetteihin halutaan liittää.

```
protected override void OnCreate()
{
    _entityManager = World.Active.EntityManager;
    enemySpawnerSystem = World.Active.GetExistingSystem<EnemySpawnerSystem>() as EnemySpawnerSystem;

    genericUIArchetype = _entityManager.CreateArchetype
        (
        typeof(Translation),
        typeof(LocalToWorld),
        typeof(SpriteComponent),
        typeof(UIComponent)
        );

    healthBarArchetype = _entityManager.CreateArchetype
       (
       typeof(Translation),
       typeof(LocalToWorld),
       typeof(SpriteComponent),
       typeof(HealthBarComponent),
       typeof(UIComponent)
       );

    weaponIconArchetype = _entityManager.CreateArchetype
        (
        typeof(Translation),
        typeof(LocalToWorld),
        typeof(SpriteComponent),
        typeof(UIComponent),
        typeof(WeaponIconComponent)
        );

    grenadeIconArchetype = _entityManager.CreateArchetype
        (
        typeof(Translation),
        typeof(LocalToWorld),
        typeof(SpriteComponent),
        typeof(UIComponent),
        typeof(GrenadeIconComponent)
        );

    buttonArchetype = _entityManager.CreateArchetype
        (
```

KUVA 15. UICreatorSystem -luokan OnCreate -metodi.

Pelin jokaisen tilan UI-elementtien luomiseen on oma metodi, joka alustaa NativeListin, luo entiteetit, asettaa entiteettien komponentteihin arvot sekä lisää entiteetit lopuksi listoihin (kuva 16).

```
public void CreateMenu()
{
    menuEntities = new NativeList<Entity>(Allocator.Persistent);

    Entity newEntity;
    AnimationConfig.AnimationData animData;

    //Background
    newEntity = _entityManager.CreateEntity(genericUIArchetype);
    animData = AnimationHelper.GetConfig(AnimationConfig.SpriteType.MenuBackground);

    _entityManager.SetComponentData(newEntity, new UIComponent { position = float3.zero, anchor = CanvasSystem.Anchor.Middle });

    _entityManager.SetComponentData(newEntity, new SpriteComponent
    {
        spriteType = AnimationConfig.SpriteType.MenuBackground,
        frameCount = animData.idleAnimationFrameCount,
        frameTimerMax = animData.idleAnimationFrameTime,
        scale = animData.scale,
        rotation = Quaternion.identity,
        LoopingAnimation = false,
        shouldAnimate = false,
        color = animData.color
    });

    menuEntities.Add(newEntity);

    //Buttons
    //Play
    newEntity = _entityManager.CreateEntity(buttonArchetype);
    animData = AnimationHelper.GetConfig(AnimationConfig.SpriteType.PlayButton);

    _entityManager.SetComponentData(newEntity, new UIComponent { position = new float3(0, 5, 0), anchor = CanvasSystem.Anchor.Middle });

    _entityManager.SetComponentData(newEntity, new SpriteComponent
    {
        spriteType = AnimationConfig.SpriteType.PlayButton,
        frameCount = animData.idleAnimationFrameCount,
        frameTimerMax = animData.idleAnimationFrameTime,
        scale = animData.scale,
        rotation = Quaternion.identity,
        LoopingAnimation = false,
        shouldAnimate = false,
```

KUVA 16. UICreatorSystem -luokan CreateMenu -metodi.

Kaikki NativeListit, joita käytetään entiteettien hallintaan, täytyy hävittää, jotta peliä suljettaessa ei synny muistivuotoja. Tämä hoidetaan aina, kun pelin tila muuttuu. Lisäksi esimerkiksi UICreatorSystem-luokassa on implementoitu OnDestroy-metodi (kuva 17), joka kutsuu muita metodeja, joissa tuhotaan entiteetit listasta ja hävitetään lista (kuva 18). Näin varmistettiin, että kaikki listat hävitetään, kun peli suljetaan.

```
protected override void OnDestroy()
{
    DisposeMenu();
    DisposeInGameUI();
    DisposeInstructionsMenu();
    DisposeGameOverScreen();
}
```

KUVA 17. UICreatorSystem-luokan OnDestroy-metodi.

```
public void DisposeMenu()
{
    if (!menuEntities.IsCreated) return;

    for(int i = 0; i < menuEntities.Length; i++)
    {
        _entityManager.DestroyEntity(menuEntities[i]);
    }

    menuEntities.Dispose();
}
```

KUVA 18. UICreatorSystem-luokan DisposeMenu-metodi.

Päävalikossa ja peli ohi -tilassa käytettävien nappien toiminnallisuus täytyi myös tehdä alusta asti itse. ButtonSystem-luokassa tarkastellaan, onko kursori napin päällä (kuva 20), luetaan käyttäjän syötettä ja kutsutaan metodia, jossa on on toteutettu napin toiminto (kuva 19).

```
protected override void OnUpdate()
{
    mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);

    Entities.ForEach((Entity entity, ref Translation translation, ref ButtonComponent buttonComponent, ref SpriteComponent spriteComponent) =>
    {
        if (IsHovered(ref buttonComponent, ref translation))
        {
            ChangeButtonState(entity, ref buttonComponent, ref spriteComponent, ButtonState.Hovered);

            if (Input.GetMouseButtonUp(0))
            {
                ButtonClicked(ref buttonComponent);
            }
        }
        else
        {
            ChangeButtonState(entity, ref buttonComponent, ref spriteComponent, ButtonState.Idle);
        }
    });
}
```

KUVA 19. ButtonSystem-luokan OnUpdate-metodi.

```
public bool IsHovered(ref ButtonComponent buttonComponent, ref Translation translation)
{
    bool result = false;

    if (mousePos.x > (translation.Value.x - buttonComponent.rectWidth / 2)
        && mousePos.x < (translation.Value.x + buttonComponent.rectWidth / 2)
        && mousePos.y > (translation.Value.y - buttonComponent.rectHeight / 2)
        && mousePos.y < (translation.Value.y + buttonComponent.rectHeight / 2))
    {
        result = true;
    }
    return result;
}
```

KUVA 20. ButtonSystem-luokan IsHovered-metodi.

Peli- ja peli ohi -tilassa näkyvien numeroelementtien päivitys ja luonti hoidetaan myös UICreatorSystem-luokassa. UpdateEnemiesRemaining-metodissa tuhotaan edelliset numeron näyttämiseen käytetyt entiteetit, kutsutaan metodia, joka

luo oikean määrän uusia entiteettejä, ja alustetaan sen jälkeen uusien entiteettien UIComponent-komponentit (kuva 21).

```
public void UpdateEnemiesRemaining(int enemyCount)
{
    if (inGameEnemiesRemainingEntities.IsCreated)
    {
        for(int i = 0; i < inGameEnemiesRemainingEntities.Length; i++)
        {
            _entityManager.DestroyEntity(inGameEnemiesRemainingEntities[i]);
        }
        inGameEnemiesRemainingEntities.Dispose();
    }

    inGameEnemiesRemainingEntities = new NativeList<Entity>(Allocator.Persistent);

    var numberEntities = CreateNumberSprites(enemyCount);

    for (int i = 0; i < numberEntities.Length; i++)
    {
        _entityManager.SetComponentData(numberEntities[i], new UIComponent { position = new float3(-2.1f + (i*0.7f), -2f, 0),
            anchor = CanvasSystem.Anchor.TopRight });
        inGameEnemiesRemainingEntities.Add(numberEntities[i]);
    }
    numberEntities.Dispose();
}
```

KUVA 21. UICreatorSystem-luokan UpdateEnemiesRemaining-metodi.

Koska ECS:ään ei kuulu minkäänlaisia tekstityökaluja, täytyi numeroelementit luoda hieman omalaatuisesti. CreateNumberSprites-metodi (kuva 22) ottaa parametrinä numeroarvon. Numeroarvon merkkien lukumäärä otetaan selville, jotta tiedetään, kuinka monta entiteettiä numeron näyttämiseen tarvitaan. Jokaista merkkiä kohtaan luodaan yksi entiteetti ja alustetaan sen SpriteComponent-komponentin arvot niin, että näytölle piirtyy oikea numero.

```
public NativeList<Entity> CreateNumberSprites(int number)
{
    NativeList<Entity> result = new NativeList<Entity>(Allocator.Temp);

    char[] charArray = number.ToString().ToCharArray();
    int figureCount = charArray.Length;

    Entity newEntity;
    var animData = AnimationHelper.GetConfig(AnimationConfig.SpriteType.NumberSheet);

    for (int i = 0; i < figureCount; i++)
    {
        newEntity = _entityManager.CreateEntity(genericUIArchetype);

        _entityManager.SetComponentData(newEntity, new SpriteComponent
        {
            spriteType = AnimationConfig.SpriteType.NumberSheet,
            frameCount = animData.idleAnimationFrameCount,
            frameTimerMax = animData.idleAnimationFrameTime,
            scale = animData.scale,
            rotation = Quaternion.identity,
            LoopingAnimation = false,
            shouldAnimate = false,
            currentFrame = int.Parse(charArray[i].ToString()),
            color = animData.color
        });

        result.Add(newEntity);
    }

    return result;
}
```

KUVA 22. UICreatorSystem-luokan CreateNumberSprites-metodi.

UI-elementit, jotka näyttävät pelaajan terveyspisteet sekä kranaatin latausajan toteutettiin samalla tyylillä. Kummallekin luotiin erilliset systeemit, jotka päivittävät kyseisten entiteettien SpriteComponent-komponentin currentSpriteFrame-arvoa (kuva 23). Kummankin elementin tekstuurit ovat puoliksi tyhjiä. Joten esimerkiksi terveyspistepalkin tapauksessa, kun tekstuurista piirrettävää aluetta liikutetaan pikkuhiljaa, näyttää kuin palkki tyhjenisi.

```
public class HealthBarSystem : ComponentSystem
{
    float lerpDuration = 1f;
    float lerpStartTime = 0;

    protected override void OnUpdate()
    {
        int playerHealth = 0;
        int playerMaxHealth = 0;

        float deltaTime = Time.deltaTime;
        float time = Time.time;
        float progress = time - lerpStartTime;

        Entities.ForEach((ref PlayerComponent playerComponent, ref HealthComponent healthComponent) =>
        {
            playerHealth = healthComponent.health;
            playerMaxHealth = healthComponent.maxHealth;
        });

        Entities.ForEach((ref HealthBarComponent healthBarComponent, ref SpriteComponent spriteComponent, ref Translation translation ) =>
        {
            healthBarComponent.targetSpriteFrame = 1 - (float)playerHealth / (float)playerMaxHealth;

            healthBarComponent.currentSpriteFrame = math.lerp(healthBarComponent.currentSpriteFrame, healthBarComponent.targetSpriteFrame, progress / lerpDuration);

            spriteComponent.currentFrame = healthBarComponent.currentSpriteFrame;

        });
    }
```

KUVA 23. HealthBarSystem.

5.3.4 Kamera

Pelin kamera on perinteinen GameObjecti, sillä ECS ei tarjoa työkaluja käyttää kameraa entiteettinä. Pelissä kamera seuraa pelaajan hahmoa CameraFollowSystem-luokan avulla (kuva 24). Kameran liikuttamisen lisäksi OnUpdate-luokassa varmistetaan, ettei kamera siirry liian kauas pelikentästä.

```
protected override void OnUpdate()
{
    float deltaTime = Time.deltaTime;

    Entities.ForEach((Entity entity, ref CameraShakeComponent cameraShakeComponent) =>
    {
        if (cameraShakeComponent.shakeDuration > 0)
        {
            CameraHelper.camera.transform.localPosition = cameraShakeComponent.startPos +
            new float3(UnityEngine.Random.insideUnitCircle.x, UnityEngine.Random.insideUnitCircle.y, 0) *
            cameraShakeComponent.shakeIntensity;

            cameraShakeComponent.shakeDuration -= deltaTime;
        }
        else
        {
            PostUpdateCommands.DestroyEntity(entity);
            CameraHelper.camera.transform.localPosition = cameraShakeComponent.startPos;
        }
    });

    Entities.ForEach((ref PlayerComponent playerComponent, ref Translation translation) =>
    {
        float3 newPos = new float3(translation.Value.x, translation.Value.y, CameraHelper.cameraHolder.position.z);

        float cameraHalfHeight = Camera.main.orthographicSize;
        float cameraHalfWidth = Camera.main.aspect * cameraHalfHeight;

        if (newPos.x >= (MapBounds.width / 2) - (cameraHalfWidth)
        || newPos.x <= (-MapBounds.width / 2) + (cameraHalfWidth))
        {
            newPos.x = CameraHelper.cameraHolder.transform.position.x;
        }

        if( newPos.y >= (MapBounds.height / 2) - (cameraHalfHeight)
        || newPos.y <= (-MapBounds.height / 2) + (cameraHalfHeight))
        {
            newPos.y = CameraHelper.cameraHolder.transform.position.y;
        }

        CameraHelper.cameraHolder.transform.position = newPos;
```

KUVA 24. CameraFollowSystem-luokan OnUpdate-metodi.

CameraFollowSystem-luokassa toteutetaan myös kameran ravisteluefekti. Ravistelu aloitetaan, kun systeemi löytää entiteetin, jolla on CameraShakeComponent-komponentti. Tämä entiteetti luodaan kutsumalla ShakeCamera-metodia, johon annetaan parametriksi ravistelun voimakkuus ja kesto (kuva 25).

```
public void ShakeCamera(float shakeIntensity, float shakeDuration)
{
    EntityManager entityManager = World.Active.EntityManager;
    Entity entity = entityManager.CreateEntity(cameraShakeArchetype);

    entityManager.SetComponentData(entity, new CameraShakeComponent
    {
        startPos = Camera.main.transform.localPosition,
        shakeIntensity = shakeIntensity,
        shakeDuration = shakeDuration
    });
}
```

KUVA 25. CameraFollowSystem-luokan ShakeCamera-metodi.

5.3.5 Äänet

Pelin äänentoistoon käytetään GameObjecteja, joilla on AudioSource-komponentti. GameObjectit poolataan MonoBehaviour-luokalla. Ääniä toistetaan hakemalla poolista GameObjecti ja kutsumalla apuluokka AudioHelperin

PlayOneShot-metodia (kuva 26), johon lähetetään parametrinä toistettavan äänen tyyppi ja viittaus juuri haetun GameObjectin AudioSource-komponenttiin. PlayOneShot-metodi hakee AudioConfig-ScriptableObjectista toistettavan äänitiedoston ja sen toistamiseen vaikuttavat arvot.

```
public static void PlayOneShot(AudioConfig.AudioType type, AudioSource aSource)
{
    if (aSource != null && type != AudioConfig.AudioType.Null)
    {
        for (int i = 0; i < _audioConfig.audioDatas.Count; i++)
        {
            if (_audioConfig.audioDatas[i].audioType == type && _audioConfig.audioDatas[i].clip != null)
            {
                if (_audioConfig.audioDatas[i].pitchVariation != AudioConfig.PitchVariationAmount.None)
                {
                    switch (_audioConfig.audioDatas[i].pitchVariation)
                    {
                        case AudioConfig.PitchVariationAmount.Low:
                            aSource.pitch = Random.Range(0.95f, 1.05f);
                            break;
                        case AudioConfig.PitchVariationAmount.Medium:
                            aSource.pitch = Random.Range(0.9f, 1.1f);
                            break;
                        case AudioConfig.PitchVariationAmount.High:
                            aSource.pitch = Random.Range(0.8f, 1.2f);
                            break;
                    }
                }
                else
                    aSource.pitch = 1;

                aSource.PlayOneShot(_audioConfig.audioDatas[i].clip);

                break;
            }
```

KUVA 26. AudioHelper-luokan PlayOneShot-metodi.

Äänen toistamiseen käytetty GameObjecti palautetaan pooliin automaattisesti äänen toistamisen jälkeen AudioSourceAutoDeactivate-luokan avulla (kuva 27).

```
public class AudioSourceAutoDeactivate : MonoBehaviour
{
    private AudioSource audioSource;

    private void Awake()
    {
        audioSource = GetComponent<AudioSource>();
    }

    private void OnEnable()
    {
        StartCoroutine(AutoDeactivateRoutine());
    }

    private IEnumerator AutoDeactivateRoutine()
    {
        yield return new WaitUntil(() => audioSource.isPlaying );

        yield return new WaitWhile( () => audioSource.isPlaying);

        Pools.Instance.AudioSourcePool.ReturnObjectToPool(audioSource);

    }
}
```

KUVA 27. AudioSourceAutoDeactivate.

5.3.6 Pelin tilat

Pelin tiloja hallitsee GameManagerSystem (kuva 28). Mainittakoon, että kyseisen systeemin toteutus on kehno ja hieman ristiriidassa Dataorientoituneen mallin ja ECS:n periaatteiden kanssa.

GameManagerSystemin OnCreate-metodissa kutsutaan kaikkien apuluokkien initialisointimetodeja, joissa ne hakevat viittaukset pelissä käytettäviin ScriptableObjecteihin. OnStartRunning-metodissa haetaan viittaukset pelin pelillisiä elementtejä hallitseviin systeemeihin, sekä asetetaan pelin tila Menu-tilaan.

```
public class GameManagerSystem : ComponentSystem
{
    public enum GameState...

    public GameState currentState = GameState.Null;

    private UICreatorSystem uiCreatorSystem;
    private EnemySpawnerSystem enemySpawnerSystem;
    private PlayerSpawningSystem playerSpawningSystem;
    private GunPickupSpawnerSystem gunPickupSpawnerSystem;
    private PowerUpPickupSpawnerSystem powerUpPickupSpawnerSystem;

    private Entity stateEntity;
    private EntityManager entityManager;

    protected override void OnCreate()
    {
        entityManager = World.Active.EntityManager;

        AudioHelper.Init();
        GunHelper.Init();
        AnimationHelper.Init();
        EnemyHelper.Init();
        PowerUpHelper.Init();
        PickupSpawnerHelper.Init();
    }

    protected override void OnStartRunning()
    {
        CameraHelper.Init();

        uiCreatorSystem = World.Active.GetExistingSystem(typeof(UICreatorSystem)) as UICreatorSystem;
        enemySpawnerSystem = World.Active.GetExistingSystem(typeof(EnemySpawnerSystem)) as EnemySpawnerSystem;
        playerSpawningSystem = World.Active.GetExistingSystem(typeof(PlayerSpawningSystem)) as PlayerSpawningSystem;
        gunPickupSpawnerSystem = World.Active.GetExistingSystem(typeof(GunPickupSpawnerSystem)) as GunPickupSpawnerSystem;
        powerUpPickupSpawnerSystem = World.Active.GetExistingSystem(typeof(PowerUpPickupSpawnerSystem)) as PowerUpPickupSpawnerSystem;

        ChangeState(GameState.Menu);
    }
```

KUVA 28. GameManagerSystem-luokan muuttujat ja alustusmetodit.

GameManagerSystemin ChangeState-metodilla (kuva 29) voidaan vaihtaa pelin tilaa. Metodissa poistetaan kaikki nykyisen tilan entiteetit ja luodaan tilalle kaikki uuteen tilaan tarvittavat entiteetit.

```
public void ChangeState(GameState state)
{
    switch (currentState)
    {
        case GameState.Menu:
            uiCreatorSystem.DisposeMenu();
            break;
        case GameState.Game:
            uiCreatorSystem.DisposeInGameUI();
            enemySpawnerSystem.ClearEnemies();
            enemySpawnerSystem.DestroySpawner();
            gunPickupSpawnerSystem.ClearGunPickups();
            gunPickupSpawnerSystem.DestroySpawner();
            powerUpPickupSpawnerSystem.ClearPickups();
            powerUpPickupSpawnerSystem.DestroySpawner();
            AudioHelper.StopMusic();
            break;
        case GameState.GameOver:
            uiCreatorSystem.DisposeGameOverScreen();
            break;
    }

    currentState = state;

    switch (currentState)
    {
        case GameState.Menu:
            uiCreatorSystem.CreateMenu();
            break;
        case GameState.Game:
            playerSpawningSystem.SpawnPlayer();
            uiCreatorSystem.CreateInGameUI();
            enemySpawnerSystem.CreateSpawner();
            gunPickupSpawnerSystem.CreateSpawner();
            powerUpPickupSpawnerSystem.CreateSpawner();
            AudioHelper.StartMusic();
            break;
        case GameState.GameOver:
            uiCreatorSystem.CreateGameOverScreen();
            break;
    }
}
```

KUVA 29. GameManagerSystem-luokan ChangeState-metodi.

5.3.7 Pelaajan hahmo

Pelaajan syötteen lukemista varten luotiin yksi systeemi (kuva 30), joka lukee kaiken syötteen ja muuttaa pelaajaentiteetin komponenttien arvoja syötteen mukaan. Pelaajaa voi liikuttaa käyttämällä WASD- tai nuolinäppäimiä. Kääntyminen/tähtääminen ja ampuminen tehdään hiirellä, ja kranaatin voi heittää painamalla Q-näppäintä.

```
public class InputSystem : ComponentSystem
{
    protected override void OnUpdate()
    {
        Entities.ForEach((ref PlayerComponent playerComponent, ref MoveComponent moveComponent, ref MouseTargetingComponent mouseTargetingComponent,
            ref Translation translation, ref ShootComponent shootComponent, ref GrenadeThrowComponent grenadeThrowComponent ) =>
        {
            float inputX = Input.GetAxis("Horizontal");
            float inputY = Input.GetAxis("Vertical");
            moveComponent.moveX = inputX;
            moveComponent.moveY = inputY;

            if(inputX != 0 || inputY != 0)
                moveComponent.isMoving = true;
            else
                moveComponent.isMoving = false;

            float3 mousePosInWorld = Camera.main.ScreenToWorldPoint(Input.mousePosition);
            mousePosInWorld.z = 0;

            mouseTargetingComponent.mousePosInWorld = mousePosInWorld;
            float3 targetVector = (mousePosInWorld - translation.Value);
            mouseTargetingComponent.directionVector = new float2(targetVector.x, targetVector.y);
            mouseTargetingComponent.directionVectorNormalized = math.normalize(mouseTargetingComponent.directionVector);
            moveComponent.facingDirection = mouseTargetingComponent.directionVectorNormalized;

            if (Input.GetMouseButton(0))
                shootComponent.shooting = true;
            else
                shootComponent.shooting = false;

            if (Input.GetKeyDown(KeyCode.Q))
                grenadeThrowComponent.cooking = true;

            if (Input.GetKeyUp(KeyCode.Q))
                grenadeThrowComponent.throwing = true;
        });

    }
}
```

KUVA 30. InputSystem.

Pelaajan liikkumisesta vastaa MoverSystem-systeemi (kuva 32). MoverSystem hakee arvoja entiteetin MoveComponent-komponentista (kuva 31) ja päivittää pelaajaentiteetin sijaintia näiden arvojen perusteella. Liikkumistoteutuksen lisäksi MoverSystem laskee pelaajahahmon rotaation, vaihtaa animaatiota sen mukaan liikkuuko pelaaja vai ei (kuva 33) ja varmistaa, ettei pelaaja liiku ulos pelialueelta.

```
public class MoverSystem : ComponentSystem
{
    protected override void OnUpdate()
    {
        float deltaTime = Time.deltaTime;

        Entities.ForEach((ref MoveComponent moveComponent, ref Translation translation, ref SpriteComponent spriteComponent) =>
        {
            float rotZ = Mathf.Atan2(moveComponent.facingDirection.y, moveComponent.facingDirection.x) * Mathf.Rad2Deg;
            spriteComponent.rotation = Quaternion.Euler(0f, 0f, rotZ - 90);

            if (moveComponent.isMoving)
            {
                if (spriteComponent.currentAnimation != AnimationConfig.AnimationType.Moving)
                    SetAnimationData(ref spriteComponent, AnimationConfig.AnimationType.Moving, true, true);

                float3 newPosition = translation.Value + new float3(moveComponent.moveX, moveComponent.moveY, 0);
                float moveVectorMagnitude = math.distance(translation.Value, newPosition);

                if (moveVectorMagnitude >= 1)
                    newPosition = translation.Value + math.normalize(new float3(moveComponent.moveX, moveComponent.moveY, 0))
                    * deltaTime * (moveComponent.moveSpeed * moveComponent.moveSpeedMultiplier);
                else
                    newPosition = translation.Value + new float3(moveComponent.moveX, moveComponent.moveY, 0)
                    * deltaTime * (moveComponent.moveSpeed * moveComponent.moveSpeedMultiplier);

                if (moveComponent.useMapBounds)
                {
                    if (newPosition.x >= MapBounds.width / 2 || newPosition.x <= -MapBounds.width / 2)
                        newPosition.x = translation.Value.x;

                    if (newPosition.y >= MapBounds.height / 2 || newPosition.y <= -MapBounds.height / 2)
                        newPosition.y = translation.Value.y;
                }

                translation.Value = newPosition;
            }
            else
            {
                if (spriteComponent.currentAnimation == AnimationConfig.AnimationType.Moving)
                    SetAnimationData(ref spriteComponent, AnimationConfig.AnimationType.Idle, false, false);
            }
        });
```

KUVA 32. MoverSystem-luokan OnUpdate-metodi.

```
public struct MoveComponent : IComponentData
{
    public float moveSpeed;
    public float moveSpeedMultiplier;
    public float moveX;
    public float moveY;
    public bool isMoving;
    public float3 moveDirection;
    public float3 rotationTargetPos;
    public float2 facingDirection;

    public bool useMapBounds;
    public bool normalizeSpeed;

}
```

KUVA 31. MoveComponent.

```
private void SetAnimationData(ref SpriteComponent spriteComponent, AnimationConfig.AnimationType animType, bool shouldAnimate, bool loopAnimation)
{
    AnimationConfig.AnimationData animationData = AnimationHelper.GetConfig(spriteComponent.spriteType);

    spriteComponent.currentAnimation = animType;
    spriteComponent.currentFrame = 0;
    spriteComponent.frameCount = animType == AnimationConfig.AnimationType.Idle ? animationData.idleAnimationFrameCount
        : animationData.movingAnimationFrameCount;
    spriteComponent.frameTimer = 0;
    spriteComponent.frameTimerMax = animType == AnimationConfig.AnimationType.Idle ? animationData.idleAnimationFrameTime
        : animationData.movingAnimationFrameTime;
    spriteComponent.scale = animationData.scale;
    spriteComponent.LoopingAnimation = loopAnimation;
    spriteComponent.shouldAnimate = shouldAnimate;
}
```

KUVA 33. MoverSystem-luokan SetAnimationData-metodi.

Terveyspisteiden vähentäminen ja lopulta pelaajahahmon kuoleminen tapahtuu TakeDamageSystem-luokassa (kuva 34). Terveyspisteiden vähentämisen lisäksi

OnUpdate-metodissa kutsutaan metodeja, jotka toistavat ääntä, ravistelevat kameraa sekä päivittävät terveyspistepalkin, mikäli pelaaja ottaa vahinkoa. Vihollisen vahingoittaessa pelaajaa pelaaja muuttuu hetkeksi kuolemattomaksi, jolloin pelaajaan ei voida tehdä lisää vahinkoa.

```
protected override void OnUpdate()
{
    float time = Time.time;
    Entities.ForEach((Entity entity, ref HealthComponent healthComponent, ref TakeDamageComponent takeDamageComponent,
        ref SpriteComponent spriteComponent, ref PlayerComponent playerComponent) =>
    {
        if (!healthComponent.invulnerable)
            healthComponent.health -= takeDamageComponent.damageAmount;

        if (takeDamageComponent.damageAmount > 0)
        {
            var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
            aSource.gameObject.SetActive(true);
            AudioHelper.PlayOneShot(AudioConfig.AudioType.PlayerHit, aSource);
            (World.Active.GetExistingSystem<CameraFollowSystem>() as CameraFollowSystem).ShakeCamera(0.25f,0.25f);
            World.Active.GetExistingSystem<HealthBarSystem>().StartLerping(time);
            takeDamageComponent.damageAmount = 0;
        }

        if (healthComponent.health <= 0)
        {
            World.Active.GetExistingSystem<GameManagerSystem>().ChangeState(GameManagerSystem.GameState.GameOver);
            PostUpdateCommands.DestroyEntity(entity);
        }
    });
```

KUVA 34. TakeDamageSystem-luokan pelaajan toteutus.

Olennainen osa peliä on ampuminen. Ampumisen hoitaa ShootSystem-systeemi (kuva 35). ShootSystem luo OnCreate-metodissa EntityArchetypen, jota käytetään luotien luomiseen. OnUpdate-metodissa tarkastellaan ja muokataan ShootComponentin arvoja, joiden pohjalta ampuminen tapahtuu. Shoot-metodissa tapahtuu varsinainen luotien luominen (kuva 36).

```
public class ShootSystem : ComponentSystem
{
    private EntityArchetype bulletArchetype;
    private EntityManager entityManager;

    protected override void OnCreate()
    {
        entityManager = World.Active.EntityManager;

        bulletArchetype = entityManager.CreateArchetype
            (
            typeof(Translation),
            typeof(LocalToWorld),
            typeof(SpriteComponent),
            typeof(MoveComponent),
            typeof(BulletComponent)
            );
    }

    protected override void OnUpdate()
    {
        Entities.ForEach((Entity entity,ref Translation translation, ref ShootComponent shootComponent,
            ref MouseTargetingComponent mouseTargetingComponent) =>
        {
            if (shootComponent.shooting && shootComponent.cooldownTimer <= 0)
            {
                Shoot(shootComponent, translation.Value, mouseTargetingComponent.directionVectorNormalized);

                var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
                aSource.gameObject.SetActive(true);
                AudioHelper.PlayOneShot(shootComponent.audioType, aSource);

                shootComponent.cooldownTimer = shootComponent.cooldown;
            }
            else if (shootComponent.cooldownTimer > 0)
            {
                shootComponent.cooldownTimer -= Time.deltaTime;
            }
        });
    }
```

KUVA 35. ShootSystem-luokan OnCreate ja OnUpdate-metodit.

```
private void Shoot(ShootComponent shootComponent, float3 spawnPoint, float2 aimDirection)
{
    AnimationConfig.AnimationData bulletAnimationData = AnimationHelper.GetConfig(AnimationConfig.SpriteType.PlayerBullet);
    NativeArray<Entity> bulletEntities = new NativeArray<Entity>(shootComponent.projectileCount, Allocator.Temp);

    for (int i = 0; i < shootComponent.projectileCount; i++)
    {
        bulletEntities[i] = entityManager.CreateEntity(bulletArchetype);
        entityManager.SetComponentData(bulletEntities[i], new Translation { Value = spawnPoint + (new float3(aimDirection.x, aimDirection.y, 0) * 0.2f) });
        entityManager.SetComponentData(bulletEntities[i], new SpriteComponent
        {
            currentAnimation = AnimationConfig.AnimationType.Idle,
            spriteType = AnimationConfig.SpriteType.PlayerBullet,
            frameCount = bulletAnimationData.idleAnimationFrameCount,
            frameTimerMax = bulletAnimationData.idleAnimationFrameTime,
            rotation = Quaternion.identity,
            scale = bulletAnimationData.scale,
            color = bulletAnimationData.color
        });

        entityManager.SetComponentData(bulletEntities[i], new BulletComponent
        {
            lifeTime = shootComponent.projectileLifeTime,
            damage = (int)(shootComponent.damage * shootComponent.damageMultiplier),
            explosive = shootComponent.explosive,
            explosionRadius = shootComponent.explosionRadius,
            impactAudioType = shootComponent.impactAudioType
        });
    }

    float3 aimDir = new float3(aimDirection.x, aimDirection.y, 0);
    float3 projectileVector;
    float2[] projectileDirections = new float2[bulletEntities.Length];

    switch (shootComponent.projectileCount)...

    for(int i = 0; i < bulletEntities.Length; i++)
    {
        entityManager.SetComponentData(bulletEntities[i], new MoveComponent { isMoving = true, useMapBounds = false,
            moveSpeed = shootComponent.projectileSpeed, moveX = projectileDirections[i].x,
            moveY = projectileDirections[i].y , moveSpeedMultiplier = 1});
    }
}
```

KUVA 36. ShootSystem-luokan Shoot-metodi.

Luotien liikkuminen tapahtuu samalla MoverSystem-luokalla kuin pelaajan ja vihollisten liikkuminen. Törmäysten tarkastelu tehdään BulletHitCheckSystem-luokassa. Törmäysten tarkastelun implementaatio on hyvin alkukantainen ja kehno ratkaisu, mutta se toimii tässä tapauksessa (kuva 37).

```
protected override void OnUpdate()
{
    Entities.ForEach((Entity entity, ref Translation translation, ref BulletComponent bulletComponent) =>
    {
        bulletTranslation = translation;
        bool bulletHit = bulletComponent.hitEnemy;
        BulletComponent bullet = bulletComponent;

        Entities.ForEach((Entity enemyEntity, ref Translation enemyTranslation, ref EnemyComponent enemyComponent,
            ref HealthComponent healthComponent, ref TakeDamageComponent takeDamageComponent) =>
        {
            if (math.distance(bulletTranslation.Value, enemyTranslation.Value) <= 1f && !bulletHit)
            {
                if (bullet.impactAudioType != AudioConfig.AudioType.Null)
                {
                    var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
                    aSource.gameObject.SetActive(true);
                    AudioHelper.PlayOneShot(bullet.impactAudioType, aSource);
                }

                if (!bullet.explosive)
                {
                    takeDamageComponent.damageAmount += bullet.damage;
                }
                else
                {
                    SpawnExplosion(bulletTranslation.Value, AnimationConfig.SpriteType.ExplosionLarge);

                    Entities.ForEach((Entity enemyEntity2, ref Translation enemyTranslation2, ref EnemyComponent enemyComponent2,
                        ref HealthComponent healthComponent2, ref TakeDamageComponent takeDamageComponent2) =>
                    {
                        if (math.distance(bulletTranslation.Value, enemyTranslation2.Value) <= bullet.explosionRadius)
                        {
                            takeDamageComponent2.damageAmount += bullet.damage;
                        }
                    });
                }
                bulletHit = true;
            }
        });

        if (bulletHit)
            bulletComponent.hitEnemy = true;
```

KUVA 37. BulletHitCheckSystem.

Kranaattien heittämisestä vastaavan GrenadeThrowSystem-luokan OnCreate-metodissa luodaan EntityArchetype kranaateille ja OnUpdate-metodissa on toteutettu itse kranaatin heittäminen (kuva 38).

```
protected override void OnUpdate()
{
    float deltaTime = Time.deltaTime;

    Entities.ForEach((Entity entity, ref Translation translation, ref GrenadeThrowComponent grenadeThrowComponent,
        ref MouseTargetingComponent mouseTargetingComponent) =>
    {

        if (grenadeThrowComponent.cooldownTimer > 0)
        {
            grenadeThrowComponent.cooldownTimer -= deltaTime;

            if(grenadeThrowComponent.cooldownTimer <= 0)
            {
                grenadeThrowComponent.throwing = false;
                grenadeThrowComponent.cooking = false;
                grenadeThrowComponent.ready = true;
            }
        }

        if (grenadeThrowComponent.ready && grenadeThrowComponent.cooking)
        {
            if(_cookTimer == 0)
            {
                var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
                aSource.gameObject.SetActive(true);
                AudioHelper.PlayOneShot(AudioConfig.AudioType.GrenadePin, aSource);
            }
            _cookTimer += deltaTime;


            if (grenadeThrowComponent.throwing)
            {
                var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
                aSource.gameObject.SetActive(true);
                AudioHelper.PlayOneShot(AudioConfig.AudioType.GrenadeThrow, aSource);

                Entity grenade = _entityManager.CreateEntity(_grenadeArchetype);
                AnimationConfig.AnimationData grenadeAnimationData = AnimationHelper.GetConfig(AnimationConfig.SpriteType.Grenade);

                _entityManager.SetComponentData(grenade, new Translation { Value = translation.Value });
                _entityManager.SetComponentData(grenade, new SpriteComponent...);
```

KUVA 38. GrenadeThrowSystem-luokan OnUpdate-metodi.

GrenadeSystem sisältää itse kranaattien toiminnallisuuden. OnUpdate-metodissa (kuva 39) vähennetään kranaatin räjähtämisaikaa, ja kun aikaa on kulunut tarpeeksi, luodaan räjähdys entiteetti (kuva 40), toistetaan ääntä ja tehdään vahinkoa kaikkiin, jotka ovat räjähdyksen alueella.

```
protected override void OnUpdate()
{
    float deltaTime = Time.deltaTime;

    Entities.ForEach((Entity entity, ref Translation translation, ref GrenadeComponent grenadeComponent) =>
    {
        var grenadeTranslation = translation;
        var grenade = grenadeComponent;

        if(grenadeComponent.cookTime <= 0)
        {
            SpawnExplosion(translation.Value, AnimationConfig.SpriteType.ExplosionSmall);

            var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
            aSource.gameObject.SetActive(true);
            AudioHelper.PlayOneShot(grenadeComponent.impactAudioType, aSource);

            Entities.ForEach((Entity enemyEntity, ref Translation enemyTranslation, ref EnemyComponent enemyComponent,
                ref HealthComponent healthComponent, ref TakeDamageComponent takeDamageComponent) =>
            {
                if (math.distance(grenadeTranslation.Value, enemyTranslation.Value) <= grenade.explosionRadius)
                {
                    takeDamageComponent.damageAmount += grenade.damage;
                }
            });

            PostUpdateCommands.DestroyEntity(entity);
        }
        else
        {
            grenadeComponent.cookTime -= deltaTime;
        }

    });
}
```

KUVA 39. GrenadeSystem-luokan OnUpdate-metodi.

```
private void SpawnExplosion(float3 position, AnimationConfig.SpriteType explosionType)
{
    Entity explosionEntity = PostUpdateCommands.CreateEntity(explosionArchetype);

    AnimationConfig.AnimationData animationData = AnimationHelper.GetConfig(explosionType);

    PostUpdateCommands.SetComponent(explosionEntity, new Translation { Value = position });
    PostUpdateCommands.SetComponent(explosionEntity, new SpriteComponent
    {
        spriteType = animationData.animationType,
        frameCount = animationData.idleAnimationFrameCount,
        frameTimerMax = animationData.idleAnimationFrameTime,
        scale = animationData.scale,
        rotation = Quaternion.identity,
        LoopingAnimation = false,
        shouldAnimate = true,
        color = animationData.color
    });

    PostUpdateCommands.SetComponent(explosionEntity, new AutoDestroyComponent { autoDestroyTimer = animationData.idleAnimationFrameCount
        * animationData.idleAnimationFrameTime });

    (World.Active.GetExistingSystem<CameraFollowSystem>() as CameraFollowSystem).ShakeCamera(1,0.25f);
}
```

KUVA 40. GrenadeSystem-luokan SpawnExplosion-metodi.

5.3.8 Viholliset

Vihollisten luominen pelissä tehdään EnemySpawnerSystem-luokassa. EnemySpawnerSystem pitää listaa vihollisentiteeteistä, tuhottujen vihollisten määrästä sekä tämänhetkisestä vihollisaallosta (kuva 41). Jokaisen aallon alussa vihollisentiteettien lista tyhjennetään ja uusi lista alustetaan.

```
private void InitNewWave(ref EnemySpawnerComponent enemySpawnerComponent)
{
    var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
    aSource.gameObject.SetActive(true);
    AudioHelper.PlayOneShot(AudioConfig.AudioType.WaveCleared, aSource);

    enemySpawnerComponent.currentWaveEnemyCount = math.max(enemySpawnerComponent.firstWaveEnemyCount,
        (int)math.floor(enemySpawnerComponent.currentWaveEnemyCount * 1.35f));
    enemySpawnerComponent.currentWaveEnemiesSpawned = 0;
    enemySpawnerComponent.currentWaveKillCount = 0;

    if (enemyList.IsCreated)
        enemyList.Dispose();

    enemyList = new NativeList<Entity>(Allocator.Persistent);

    wave++;

    uiCreatorSystem.UpdateCurrentWave(wave);
    uiCreatorSystem.UpdateEnemiesRemaining(enemySpawnerComponent.currentWaveEnemyCount - enemySpawnerComponent.currentWaveKillCount);

}
```

KUVA 41. EnemySpawnerSystem-luokan InitNewWave-metodi.

Vihollisia luodaan tietty määrä jokaisessa aallossa. Vihollisia ei luoda kaikkia kerralla aallon alussa, vaan niiden määrä kasvaa pikkuhiljaa satunnaisesti (kuva 42).

```
protected override void OnUpdate()
{
    Entities.ForEach((ref EnemySpawnerComponent enemySpawnerComponent) =>
    {
        if (enemySpawnerComponent.currentWaveKillCount < enemySpawnerComponent.currentWaveEnemyCount)
        {
            if (enemySpawnerComponent.currentWaveEnemiesSpawned < enemySpawnerComponent.currentWaveEnemyCount)
            {
                if (enemySpawnerComponent.enemySpawnTimer <= 0)
                {
                    int spawnAmount = UnityEngine.Random.Range(1,
                        math.max(1, enemySpawnerComponent.currentWaveEnemyCount - enemySpawnerComponent.currentWaveEnemiesSpawned));

                    SpawnEnemies(ref enemySpawnerComponent, spawnAmount);
                    enemySpawnerComponent.enemySpawnTimer = UnityEngine.Random.Range(3, 10);

                }
                else
                    enemySpawnerComponent.enemySpawnTimer -= Time.deltaTime;
            }
        }
        else
        {
            if (enemySpawnerComponent.waveIntervalTimer <= 0)
                InitNewWave(ref enemySpawnerComponent);
            else
                enemySpawnerComponent.waveIntervalTimer -= Time.deltaTime;
        }
    });
```

KUVA 42. EnemySpawnerSystem-luokan OnUpdate-metodi.

SpawnEnemies-metodille annetaan parametrinä vihollisten määrä ja se arpoo vihollisen tyypin, luo uuden entiteetin, asettaa komponenttien arvot ja lisää entiteetin lopulta listaan (kuva 43). Pelin alussa luodaan ainoastaan yhden tyyppisiä vihollisia, mutta mitä pidemmälle peli etenee, sitä enemmän eri tyyppisiä vihollisia luodaan. Viholliset luodaan pelikentän ulkopuolelle, josta ne lähtevät liikkumaan pelaajaa kohti.

```
private void SpawnEnemies(ref EnemySpawnerComponent enemySpawnerComponent, int count)
{
    enemySpawnerComponent.currentWaveEnemiesSpawned += count;
    Entity enemyEntity;

    for (int i = 0; i < count; i++)
    {
        var enemyData = EnemyHelper.GetConfig(RandomizeEnemyType());
        var enemyAnimationData = AnimationHelper.GetConfigByEnemyType(enemyData.enemyType);

        enemyEntity = EntityManager.CreateEntity(enemyArchetype);
        EntityManager.SetComponentData(enemyEntity, new EnemyComponent { attackRange = enemyData.attackRange, attackType = enemyData.enemyAttackType });
        EntityManager.SetComponentData(enemyEntity, new Translation { Value = GetSpawnPosition() });
        EntityManager.SetComponentData(enemyEntity, new MoveComponent { moveSpeed = enemyData.movementSpeed });
        EntityManager.SetComponentData(enemyEntity, new HealthComponent { maxHealth = enemyData.health, health = enemyData.health });
        EntityManager.SetComponentData(enemyEntity, new DamageComponent { damage = enemyData.damage, cooldown = enemyData.attackCooldown });
        EntityManager.SetComponentData(enemyEntity, new TakeDamageComponent { leaveCorpse = true });
        entityManager.SetComponentData(enemyEntity, new SpriteComponent
        {
            currentAnimation = AnimationConfig.AnimationType.Idle,
            spriteType = AnimationConfig.SpriteType.Floater,
            frameCount = enemyAnimationData.idleAnimationFrameCount,
            frameTimerMax = enemyAnimationData.idleAnimationFrameTime,
            scale = enemyAnimationData.scale,
            rotation = Quaternion.identity,
            LoopingAnimation = true,
            shouldAnimate = true,
            color = enemyAnimationData.color
        });

        if(enemyData.enemyAttackType == EnemyConfig.EnemyAttackType.Ranged)
        {
            entityManager.AddComponent<EnemyShootComponent>(enemyEntity);
            EntityManager.SetComponentData(enemyEntity, new EnemyShootComponent {
            gunType = enemyData.gunType,
            audioType = enemyData.audioType,
            projectileCount = enemyData.projectileCount,
            projectileLifeTime = enemyData.projectileLifeTime,
            projectileSpeed = enemyData.projectileSpeed});
        }

        enemyList.Add(enemyEntity);
```

KUVA 43. EnemySpawnerSystem-luokan SpawnEnemies-metodi.

Vihollisten vahingonottaminen on toteutettu lähes samalla tavalla kuin pelaajan, mutta vihollisilla on kuolemisanimaatio ja ne jättävät hetkellisesti jälkeensä ruumiin (kuva 44). Viholliset eivät muutu kuolemattomaksi niiden ottaessa vahinkoa.

```
Entities.ForEach((Entity entity, ref HealthComponent healthComponent, ref TakeDamageComponent takeDamageComponent,
    ref SpriteComponent spriteComponent, ref EnemyComponent enemyComponent) =>
{
    if (!healthComponent.invulnerable)
        healthComponent.health -= takeDamageComponent.damageAmount;

    if (takeDamageComponent.damageAmount > 0)
    {
        var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
        aSource.gameObject.SetActive(true);
        AudioHelper.PlayOneShot(AudioConfig.AudioType.EnemyHit, aSource);

        takeDamageComponent.damageAmount = 0;
    }

    if (healthComponent.health <= 0)
    {
        enemySpawnerSystem.AddKill();

        var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
        aSource.gameObject.SetActive(true);
        AudioHelper.PlayOneShot(AudioConfig.AudioType.EnemyDie, aSource);

        if (takeDamageComponent.leaveCorpse)
        {
            PostUpdateCommands.RemoveComponent(entity, typeof(TakeDamageComponent));
            PostUpdateCommands.RemoveComponent(entity, typeof(MoveComponent));
            PostUpdateCommands.RemoveComponent(entity, typeof(EnemyComponent));
            PostUpdateCommands.RemoveComponent(entity, typeof(HealthComponent));
            PostUpdateCommands.RemoveComponent(entity, typeof(DamageComponent));
            PostUpdateCommands.RemoveComponent(entity, typeof(TargetingComponent));
            PostUpdateCommands.RemoveComponent(entity, typeof(DamageComponent));

            AnimationConfig.AnimationData animationData = AnimationHelper.GetConfig(spriteComponent.spriteType);
            spriteComponent.currentAnimation = AnimationConfig.AnimationType.Death;
            spriteComponent.currentFrame = 0;
            spriteComponent.frameCount = animationData.deathAnimationFrameCount;
            spriteComponent.frameTimer = 0;
            spriteComponent.frameTimerMax = animationData.deathAnimationFrameTime;
            spriteComponent.scale = animationData.scale;
            spriteComponent.LoopingAnimation = false;
            spriteComponent.shouldAnimate = true;
            spriteComponent.color = new Color(1, 1, 1, 1);

            PostUpdateCommands.AddComponent<CorpseComponent>(entity);
            PostUpdateCommands.SetComponent(entity, new CorpseComponent { corpseDecayTime = 2 });
```

KUVA 44. TakeDamageSystem-luokan vihollisten toteutus.

Vihollisten implementaatiota ei käydä läpi kokonaisuudessaan tässä opinnäytetyössä, sillä niistä vastasi projektissa toinen ohjelmoija. Lisäksi vihollisten implementaatio ei vielä tässä vaiheessa vastannut haluttua lopputulosta.

Vihollisten käyttäytymistä ohjataan useista eri systeemeistä. Se, millä systeemillä vihollista milloinkin ohjataan, määräytyy vihollisten eri tilojen määrittämiseen tarkoitettujen komponenttien avulla. Mikäli vihollisella on esimerkiksi FollowStateComponent-komponentti, sitä käsitellään FollowStateSystem-systeemissä. FollowStateSystem (kuva 45) tarkastelee, täytyykö viholliselle etsiä uutta reittiä. Mi-

käli vihollinen tarvitsee uuden reitin, A_Star-systeemi (kuva 46) hoitaa uuden reitin laskelmoinnin. Vihollisen sijainnin päivittäminen hoidetaan samalla systeemillä kuin pelaajan.

```
float deltaTime = Time.deltaTime;

Entities.ForEach((
    Entity entity,
    ref EnemyComponent enemyComponent,
    ref Translation translation,
    ref MoveComponent moveComponent,
    ref TargetingComponent targetingComponent,
    ref FollowStateComponent followStateComponent ) =>
{
    if(TargetReachedCheck(translation, targetingComponent, enemyComponent))
    {
        ChangeState(entity, enemyComponent);
    }
    else
    {
        if(targetingComponent.hasTarget)
        {
            DynamicBuffer<WaypointBuffer> waypoints = entityManager.GetBuffer<WaypointBuffer>(entity);

            if(!targetingComponent.target.Value.Equals(targetingComponent.targetLastPosition) || waypoints.Length <= 0)
            {
                targetingComponent.targetLastPosition = targetingComponent.target.Value;

                Node[] wayPoints = pathfinding.FindPath(translation.Value, targetingComponent.target.Value).ToArray();
                float3[] targetingWaypoints = new float3[wayPoints.Length];
                waypoints.Clear();
                for(int i = 0; i < wayPoints.Length; i++)
                {
                    Node n = wayPoints[i];
                    targetingWaypoints[i] = n.m_vPosition;
                    waypoints.Add(targetingWaypoints[i]);
                }
            }

            if(waypoints.Length > 0)
            {
                if(math.distance(waypoints[0].Value, translation.Value) <= 0.1f)
                {
                    waypoints.RemoveAt(0);
                }
                if(waypoints.Length > 0)
                {
                    float3 vector = waypoints[0].Value - translation.Value;
                    moveComponent.moveDirection = math.normalize(vector);
                    moveComponent.facingDirection = new float2(moveComponent.moveDirection.x, moveComponent.moveDirection.y);
                    translation.Value += moveComponent.moveDirection * moveComponent.moveSpeed * deltaTime;
                }
```

KUVA 45. FollowStateSystem.

```
public List<Node> FindPath (Vector3 start, Vector3 end)
{
    Node startNode = grid.NodeFromWorldPosition(start);
    Node endNode = grid.NodeFromWorldPosition(end);

    openSet.Add(startNode);
    while(openSet.Count > 0)
    {
        Node currentNode = openSet.RemoveFirst();
        currentNode.m_bProcessed = true;

        if(currentNode == endNode)
        {
            grid.ResetNodes();
            openSet.Clear();
            return RetracePath(startNode, endNode);
        }
        foreach(Node neighbour in currentNode.neighbours)
        {
            if(neighbour.m_bProcessed || neighbour.m_bIsBlocked)
                continue;

            int newMovementCost = currentNode.m_iGCost + GetDistance(currentNode, neighbour);

            if(newMovementCost < neighbour.m_iGCost || !openSet.Contains(neighbour))
            {
                neighbour.m_iGCost = newMovementCost;
                neighbour.m_iHCost = GetDistance(neighbour, endNode);
                neighbour.m_Parent = currentNode;
                if(!openSet.Contains(neighbour))
                {
                    openSet.Add(neighbour);
                }
                else
                {
                    //openSet.UpdateItem(neighbour);
                }
            }
        }
    }
    grid.path = null;
    grid.ResetNodes();
    openSet.Clear();
    return null;
}
```

KUVA 46. A_Star-luokan FindPath-metodi.

5.3.9 Poimittavat aseet ja parannukset

Poimittavien aseiden luomisesta vastaa GunPickupSpawnerSystem ja parannusten luomisesta PowerUpPickupSpawnerSystem. Luokat ovat lähes identtisiä. Luokkien OnUpdate-metodissa pyöritetään yksinkertaista ajastinta, jonka mukaan kutsutaan SpawnPickup-metodia (kuva 47), joka hoitaa entiteettien luomisen peliin.

```
private void SpawnPickup(ref GunPickupSpawnerComponent gunPickupSpawnerComponent)
{
    Entity gunPickupEntity = _entityManager.CreateEntity(typeof(Translation), typeof(LocalToWorld), typeof(SpriteComponent), typeof(GunPickupComponent));

    _entityManager.SetComponentData(gunPickupEntity, new Translation
    {
        Value = new float3(UnityEngine.Random.Range(-MapBounds.width/2, MapBounds.width/2),
        UnityEngine.Random.Range(-MapBounds.height/2, MapBounds.height/2), 0)
    });

    var gunType = RandomizeGunType();
    var animationData = AnimationHelper.GetConfigByGunType(gunType);

    _entityManager.SetComponentData(gunPickupEntity, new GunPickupComponent { gunType = gunType, lifeTime = gunPickupSpawnerComponent.pickupLifeTime });

    _entityManager.SetComponentData(gunPickupEntity, new SpriteComponent
    {
        currentAnimation = AnimationConfig.AnimationType.Idle,
        spriteType = animationData.animationType,
        frameCount = animationData.idleAnimationFrameCount,
        frameTimerMax = animationData.idleAnimationFrameTime,
        rotation = Quaternion.identity,
        scale = animationData.scale,
        color = animationData.color
    });

    gunPickupEntities.Add(gunPickupEntity);
}
```

KUVA 47. GunPickupSpawnerSystem-luokan SpawnPickup-metodi.

Varsinaisesta poimimisesta vastaavat GunPickupSystem ja PowerUpPickupSystem. Nämäkin ovat toteutukseltaan hyvin samankaltaisia. Ne tarkkailevat, osuuko pelaajaentiteetti poimittavaan entiteettiin ja muuttavat pelaajaentiteetin komponenttien arvoja sen mukaan, mikä ase tai parannus on kyseessä (kuva 48). Aseiden ja parannusten arvot haetaan GunConfig- ja PowerUpConfig-ScriptableObjecteista. Mikäli ase- tai parannusentiteettiä ei poimita ajoissa, GunPickupSystem ja PowerUpPickupSystem tuhoavat entiteetin automaattisesti.

```
Entities.ForEach((Entity entity, ref Translation translation, ref GunPickupComponent gunPickupComponent) =>
{
    if(playerEntity == null)
        playerEntity = (World.Active.GetExistingSystem(typeof(PlayerSpawningSystem)) as PlayerSpawningSystem).playerEntity;

    float3 playerPosition = EntityManager.GetComponentData<Translation>(playerEntity).Value;

    if (math.distance(playerPosition, translation.Value) <= 1)
    {
        var pickupComponent = gunPickupComponent;

        Entities.ForEach((ref PlayerComponent playerComponent, ref ShootComponent shootComponent) =>
        {
            GunConfig.GunData gunData = GunHelper.GetGunConfig(pickupComponent.gunType);

            PostUpdateCommands.RemoveComponent(playerEntity, typeof(ShootComponent));
            PostUpdateCommands.AddComponent<ShootComponent>(playerEntity);
            PostUpdateCommands.SetComponent(playerEntity, new ShootComponent
            {
                gunType = gunData.gunType,
                cooldown = gunData.cooldown,
                damage = gunData.damage,
                projectileLifeTime = gunData.projectileLifeTime,
                projectileSpeed = gunData.projectileSpeed,
                projectileCount = gunData.projectileCount,
                audioType = gunData.audioType,
                impactAudioType = gunData.impactAudioType,
                explosive = gunData.explosive,
                explosionRadius = gunData.explosionRadius,
                damageMultiplier = World.Active.EntityManager.HasComponent(playerEntity, typeof(DoubleDamageComponent)) ? GetDamageMultiplier() : 1
                });

            var aSource = Pools.Instance.AudioSourcePool.GetPooledObject();
            aSource.gameObject.SetActive(true);
            AudioHelper.PlayOneShot(AudioConfig.AudioType.GunPickup, aSource);

            PostUpdateCommands.DestroyEntity(entity);
        });
    }

    if (gunPickupComponent.lifeTime > 0)
        gunPickupComponent.lifeTime -= deltaTime;
    else if (gunPickupComponent.lifeTime <= 0)
        PostUpdateCommands.DestroyEntity(entity);
```

KUVA 48. GunPickupSystem.

5.3.10 Job System ja Burst Compiler

Job Systemiä ja Burst Compileria ei käytetty projektissa ollenkaan. Tarkoituksena oli tehdä projekti ensin kokonaan ilman Job Systemiä, ja implementoida Job System vasta lopuksi, jotta nähdään selkeät erot suorituskyvyssä ja voidaan vertailla tilannetta ”kokonaisessa” pelissä. Job Systemin implementointi jo tehdyn koodin päälle osoittautui kuitenkin aikaa vieväksi ja aika loppui kesken.

Job Systemiä testattiin erillisessä projektissa, jotta voitiin demonstroida sen vaikutuksia. Projekti toteutettiin hybridinä, eli Job Systemiä ja Burst Compileria käytettiin yhdessä perinteisen Unityn kanssa. Entity Component Systemiä ei käytetty. Projektissa on yksinkertainen MonoBehaviour-luokka, joka sisältää kaksi samanlaista toteutusta, joissa simuloidaan erittäin raskasta operaatiota (kuva 49). Toinen toteutuksista on tavallinen metodi, ja toinen on Jobi, joka ajetaan Job Systemin avulla. Luokassa on boolean-muuttuja, jolla määritellään kumpaa toteutusta käytetään (kuva 50).

```
private void NonJobifiedTask()
{
    float value = 0f;
    for (int i = 0; i < 50000; i++)
    {
        value = math.exp10(math.sqrt(value));
    }
}

private JobHandle JobifiedTask()
{
    TestJob job = new TestJob();
    return job.Schedule();
}

[BurstCompile]
public struct TestJob : IJob
{
    public void Execute()
    {
        float value = 0f;
        for (int i = 0; i < 50000; i++)
        {
            value = math.exp10(math.sqrt(value));
        }
    }
}
```

KUVA 49. Testiluokan kaksi eri toteutusta.

```
private void Update()
{
    if (useJobs)
    {
        NativeList<JobHandle> jobHandleList = new NativeList<JobHandle>(Allocator.Temp);

        for (int i = 0; i < 10; i++)
        {
            JobHandle jobHandle = JobifiedTask();
            jobHandleList.Add(jobHandle);
        }
        JobHandle.CompleteAll(jobHandleList);
        jobHandleList.Dispose();
    }
    else
    {
        for(int i = 0; i < 10; i++)
        NonJobifiedTask();
    }
}
```

KUVA 50. Testiluokan Update-metodi.

Seuraavissa kuvissa näkyy tilastoja kolmesta eri tilanteesta. Ensimmäisessä käytetään perinteistä toteutusta (kuva 51), toisessa käytetään Job-toteutusta (kuva 52) ja viimeisessä Job-toteutuksen lisäksi käytetään Burst Compileria (kuva 53). Koodin ajamiseen käytetty aika tippuu 99% (67,1 ms) tavallisesta toteutuksesta.

Statistics

Audio:

Level: -74.8 dB DSP load: 0.3%
Clipping: 0.0% Stream load: 0.0%

Graphics: 14.8 FPS (67.5ms)

CPU: main **67.5**ms render thread 0.1ms
Batches: **0** Saved by batching: 0
Tris: 0 Verts: 0
Screen: 363x204 - 0.8 MB
SetPass calls: 0 Shadow casters: 0
Visible skinned meshes: 0 Animations: 0

KUVA 51. Tilastoja käyttämällä tavallista toteutusta.

Statistics

Audio:

Level: -74.8 dB DSP load: 0.1%
Clipping: 0.0% Stream load: 0.0%

Graphics: 71.3 FPS (14.0ms)

CPU: main **14.0**ms render thread 0.1ms
Batches: **0** Saved by batching: 0
Tris: 0 Verts: 0
Screen: 363x204 - 0.8 MB
SetPass calls: 0 Shadow casters: 0
Visible skinned meshes: 0 Animations: 0

KUVA 52. Tilastoja käyttämällä Job-toteutusta.

Statistics

Audio:

Level: -74.8 dB DSP load: 0.1%
Clipping: 0.0% Stream load: 0.0%

Graphics: 2710.1 FPS (0.4ms)

CPU: main **0.4**ms render thread 0.1ms
Batches: **0** Saved by batching: 0
Tris: 0 Verts: 0
Screen: 357x201 - 0.8 MB
SetPass calls: 0 Shadow casters: 0
Visible skinned meshes: 0 Animations: 0

KUVA 53. Tilastoja yhdistämällä Burst Compiler Job-toteutukseen.

6 POHDINTA

Unityn Data-oriented Technology Stack vaikuttaa siltä, että ollaan menossa oikeaan suuntaan ja sen hyödyt voi havaita jo nyt. Oikein käytettynä ECS koodin rakenne on erittäin selkeää ja bugeja syntyy hyvin vähän. Lisäksi DOTSin vaikutus suorituskykyyn ja sen mahdollistamat asiat ovat ällistyttäviä. Se on kuitenkin vielä erittäin keskeneräisessä vaiheessa, ja siitä puuttuu oleellisia ominaisuuksia kuten fysiikat ja äänet. Puhtaasti ECS:llä työskentely tuntui välillä vahvasti siltä, kuin keksisi pyörää uudelleen. Tällä hetkellä DOTSin yhdistäminen Unityn perinteiseen järjestelmään, ja vain yksittäisten osien muuntaminen ECS-malliin tuntuu kaikkein järkevimmältä vaihtoehdolta, mikäli DOTS ominaisuuksia halutaan käyttää.

Opinnäytetyön käytännön osuus sujui ilman suurempia ongelmia lukuun ottamatta Job Systemin käyttöä. DOTS on erittäin suuri kokonaisuus ja esimerkiksi ECS:n kaikkia toimintoja ja ominaisuuksia ei hyödynnetty projektissa. Projektin toteutukset eivät siis varmastikaan ole parhaita mahdollisia ratkaisuja. Oman näkökulman ja ajattelutavan muuttaminen dataorientoituneeksi osoittautui myös erittäin haastavaksi ja toteutuksissa esiintyy ristiriitoja kahden mallin välillä. Projektin aikana olisi pitänyt keskittyä enemmän koodin rakenteen suunnitteluun, eikä vain tehdä peliä eteenpäin ja lisätä ominaisuuksia. Koska DOTS on todella uusi paketti, netistä on vaikea löytää apua ja esimerkkejä eri ominaisuuksien käyttöön, mikä vaikeuttaa kokonaisuuden ymmärtämistä.

Koko DOTS-paketin oppimiseen vaaditaan pidempi aika, kuin tähän opinnäytetyöhön käytettiin. Lisäksi sen opetteleminen kannattaisi varmaankin aloittaa tekemällä esimerkiksi pieniä demoja, joissa testataan yksittäisiä ominaisuuksia ja luodaan pieniä järjestelmiä, eikä luoda heti suurempaa kokonaisuutta, jossa järjestelmät monimutkaistuvat nopeasti. Jatkotutkimuksena voisi perehtyä vielä syvemmin Unity DOTSiin ja tutkia esimerkiksi hybridi ECS:n käyttömahdollisuuksia.

LÄHTEET

Code Monkey. 2019. Unity DOTS Explained (ECS, Job System, Burst Compiler) Youtube-video. Julkaistu 24.4.2019. Viitattu 1.11.2019. https://www.youtube.com/watch?v=Z9-WkwdDoNY

Sefton, D. 2016. Developing a Data-Oriented Game Engine (Part 1). Julkaistu 7.5.2016. Luettu 31.10.2019. https://danielsefton.com/2016/05/developing-a-data-oriented-game-engine-part-1/

Educba. n.d. What is Inheritance in Programming? Luettu 30.10.2019. https://www.educba.com/what-is-inheritance-in-programming/

Learn to create games. 2017. Mapping Object Oriented Principles to C# with Unity & Syntax. Julkaistu 20.9.2017. Luettu 31.10.2019. http://learntocreategames.com/mapping-object-oriented-principles-to-c-with-unity-syntax/

Luke. 2016. Unity Software Design – Encapsulation. Julkaistu 29.3.2016. Luettu 30.10.2019. http://www.sigtrapgames.com/unity-software-design-encapsulation/

Lanier, L. 2019. Video Games Could Be a $300 Billion Industry by 2025 (Report). Julkaistu 1.5.2019. Luettu 2.12.2019. https://variety.com/2019/gaming/news/video-games-300-billion-industry-2025-report-1203202672/

Microsoft. 2015. Polymorphism (C# Programming Guide). Julkaistu 20.7.2015. Luettu 30.10.2019. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

Rouse, M. 2014. Abstraction. Päivitetty 30.6.2014. Luettu 30.10.2019. https://whatis.techtarget.com/definition/abstraction

Nanalyze. 2017. Unity Technologies – The World's Leading Game Engine. Julkaistu 18.10.2017. Luettu 29.10.2019. https://www.nanalyze.com/2017/10/unity-technologies-leading-game-engine/

Noel. 2009. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). Julkaistu 4.12.2009. Luettu 31.10.2019. http://gamesfromwithin.com/data-oriented-design

Porter. 2013. Unity: Now You're Thinking With Components. Julkaistu 31.10.2013. Luettu 31.10.2019. https://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

Janssen, T. 2017. OOP Concept for Beginners: What is Abstraction? Julkaistu 23.11.2017. Luettu 30.10.2019. https://stackify.com/oop-concept-abstraction/

Unity Technologies. 2019a. A feature-rich and highly flexible editor. Luettu 29.10.2019 https://unity3d.com/unity/editor

Unity Technologies. 2019b. Dots. Luettu 1.11.2019. https://unity.com/dots

Unity Technologies. 2019c. Entity Component System. Luettu 1.11.2019. https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/index.html

Agarwal, H. n.d. Inheritance in C++. Luettu 1.11.2019. https://www.geeksforgeeks.org/inheritance-in-c/

Beal, V. n.d. OOP – Object Oriented Programming. Luettu 30.10.2019. https://www.webopedia.com/TERM/O/object_oriented_programming_OOP.html

Caceres, Z. 2019. Notes for Noobs – Object-Oriented Basics for Unity3D. Julkaistu 19.5.2016. Luettu 30.10.2019. https://medium.com/@zachcaceres/notes-for-noobs-object-oriented-basics-for-unity3d-d98685235ebb