Unitylla kehitetty pelin suorituskykyongelmien ratkominen ja parantaminen

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tieto- ja viestintätekniikka, insinööri (AMK), Riihimäen kampus

Syksy, 2020

Johnny Tran

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintä tekniikka, insinööri (AMK) Tiivistelmä

Riihimäen kampus

Tekijä Johnny Tran Vuosi 2020

Työn nimi Unitylla kehitetty pelin suorituskykyongelmien ratkominen ja parantaminen.

Ohjaajat Petri Kuittinen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli selvitellä Unitylla kehitetyn pelin suorituskykyongelmien aiheuttajia ja löytää niille ratkaisuja, kun pelin suorituskykyongelman sattuessa ei oikein tiedetä sen todellista aiheuttajaa ja syytä. Pelin yleiset suorituskykyongelmien ilmiöt ovat dramaattinen hidastuminen ja hetkellisiä pysähtymisiä pelaamisen aikana. Parantamalla pelin suorituskykyä ei anna pelkästään pelaajille sulavan pelikokemuksen. Hyvä suorituskykyinen peli kykenee pyörimään heikoimmissa pelijärjestelmien laitteistoissakin, joka mahdollistaa pelaajayleisön kasvun pelillesi.

Tähän työhön luotiin raskaasti suoriutuvaa peliä törmätäkseen suorituskykyongelmaan. Peliä rasitettiin kaikkia mahdollisia osastoja, joita ovat fysiikat, äänet, erikoisefektit, tekoälyt ja animaatiot. Työssä tutustuttiin pelisuorituskyvyn profilointimenetelmiin helpottaakseen suorituskykyongelman aiheuttajan paikantamisen.

Suorituskykyongelmaan ratkaisuksi löydettiin kahta menetelmää, jotka ovat kohdistettu enimmäkseen pelifysiikkaan. Ensimmäinen suorituskykyongelman ratkaisumenetelmässä oli korvata suorituskykyongelmaista koodia vähäisellä kuormittavalla koodilla tai poistamalla muutamia toiminnan ominaisuuksia. Toisessa suorituskykyongelman ratkaisumenetelmässä oli käyttää DOTS(Data Oriented Technology Stack) kehitystapaa sijaan Unityn klassista Monobehaviouriä. DOTS on dataa suuntautuvaa ohjelmointia, joka keskittyy pelidatan järjestäytymisen keskusmuistiin (RAM). Huomattiin testissä DOTSin antoivan pelille merkittävän suorituskykyparannuksen verrattuna ensimmäisessä ratkaisumenetelmässä ja ei tarvittu ollenkaan uhrata toimintojen yksityiskohtia. DOTSin hyvän datan järjestelyn ansiosta aiheutettiin vähäisen määrän välimuistihuteja prosessorille (CPU), joka oli yksi suurista

suorituskykyongelmien syistä. Huomattiin Monobehaviourin kehitysmenetelmällä kärsitään useita välimuistihuteja oletuksena juuri sen huonon datamuistihallinnan takia.

DOTSilla pystyttiin antamaan pelille helposti suuren suorituskykyparannuksen verrattuna klassisella Monobehaviour kehitysmenetelmällä. DOTSin dataan suuntautunut ohjelmointityylisen vuoksi vaadittiin erilaisen kehitysajattelutavan kuin Monobehaviourissa, joka voi olla hankala aluksi olio-ohjelmointityyliin tottuneille koodajille. Testissä huomattiin myös DOTSin huonoja puolia. DOTSin olevan vielä kehitysvaiheessa Unityssa, joten sillä puuttuu paljon käteviä työkaluja, joita Monobehaviourissa on tarjolla. DOTSissa puuttuvat työkalujen tekeminen voi viedä paljon aikaa ja tuottaa hankaluuksia. DOTSiin liittyvät koodivirheilmoitukset ovat huomattavasti hankalampi löytää ratkaisuja verrattuna Monobehaviourillä.

Avainsanat Unity, suorituskykyongelma, optimointi, DOTS, peli
Sivut 49

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and Communication Technology
Riihimäki campus

Abstract

Author Johnny Tran
Year 2020

Subject Solving and improving Unity game performance.

Supervisors Petri Kuittinen

ABSTRACT

The aim of the project was to find out performance problems and solutions for games that have used Unity as a development platform. The goal was finding the sources and reasons for occurred performance problems in games. Common phenomena in game performance issues are dramatic slowdowns and abnormal momentary stops during gameplay. Improving game performance gives players a smooth gaming experience. Games with a good performance are capable to run on weaker gaming system hardware that allow the growth of the game player base.

A very high-performance game was created for this project to analyze performance problems. All crucial game elements were added to the game. The game had physics, sounds, special effects, artificial intelligence, and animations. The game's performance was analyzed by using profiling methods. Basic profiling methods were used for profile game performance. The collected data from profiling helped to find the source of the performance problem more easily.

There were two methods founded to improve game performance which both were mostly focused on game physics performance problems. The first method of solving a performance problem was to sacrifice some of the game details by removing some of the features. Another method of solving performance problems was to use the DOTS (Data Oriented Technology Stack) development method instead of Unity's classic Monobehavior development style. DOTS is data-oriented programming that focuses on organizing game data compactly to the main memory (RAM). Both methods improved game performance, but the DOTS development method gave significantly better performance. DOTS's great data

arrangement feature made less cache misses to the processor (CPU) which was one of the major causes of performance problems. Noticed in Unity's classic game development method suffers several cache misses by default exactly because of its poor data memory management.

DOTS can easily provide major performance improvement to games over the classic Monobehavior development method. DOTS being data-oriented programming style, it will require a different development mindset than Monobehavior. DOTS style can be tricky at first for coders that are accustomed to the object-oriented programming style. There are also downsides in using DOTS. It lacks a lot of the handy tools that are available in the Monobehavior development method. Creating those missing tools for DOTS may take a lot of time and may be difficult to implement. Code errors related to DOTS are much more difficult to find solutions to compared to Monobehavior.

Keywords Unity, performance problem, optimizing, DOTS, game
Pages 49

Sisällys

Käytetty terminologia tai käsitteistö 1

1 Johdanto 2

2 Pelisuorituskyvyn tärkeys 3

3 Suorituskykyongelmat 4

3.1 Välimuistihuti (Cache Miss) 4

3.2 Koodi tuottaa liikaa roskaa 5

4 Profilointi 5

4.1 Profiler-työkalu 6

4.2 Profiler-työkalun käyttö 7

4.3 Deep profile-moodi 8

4.4 Muut huomautukset 8

5 Suorituskykyongelmien etsintä ja ratkonta 9

5.1 Fysiikan suorituskyvyn ratkaiseminen 10

5.2 Unityn toinen pelikehitystapa DOTS 12

5.3 Monisäikeinen koodi 13

5.4 Monobehaviour olio-ohjelmointi class sijaan DOTS datakeskeinen ohjelmointi struct 13

6 DOTSin tavaton suorituskyvyn toiminnallisuus 14

6.1 Burst 15

6.1.1 Burst käyttöönotto 15

6.1.2 Burst-tarkastaja 15

6.2 Jobs 16

6.2.1 Jobs-systeemin käyttö 16

6.2.2 Burst käyttöönotto Jobille 18

6.2.3 Job-systeemin profilointi 19

6.3 Entity Component System 19

6.3.1 Entiteetti 20

6.3.2 Komponentti 20

6.3.3 Systeemi 20

6.3.4 ECS toiminnallisuus 20

6.3.5 Entity Debugger -välilehti 22

6.3.6 ECS kehitysympäristön pystytys 23

6.3.7 Entiteetin luonti 24

6.3.8 ECS komponentin luonti 25
6.3.9 ECS komponentin lisäys entiteettiin 26
6.3.10 ECS systeemin luonti 28
7 DOTS ja Monobehaviour välinen toteutus ja suorituskykyero 30
7.1 Simulaation toiminta 31
7.2 Monobehaviour-tavan simulointi toteutus 32
7.3 DOTS-tavan simulointi toteutus 33
8 Mittaustulokset 42
8.1 Monobehaviour version mittaustulokset 42
8.2 DOTS version mittaustulokset 43
9 Lopputulos 45
10 Johtopäätös 46
Lähteet 48

Kuvat, taulukot ja kaavat

Kuva 1. Ruudun piirustus 60 Hz:en ja 120 Hz:en välisen ero. 4
Kuva 2. Unityn Profiler-ikkuna. 6
Kuva 3. Unityn Profilerin ikkunan yläosa. 7
Kuva 4. Unity Profilerin ikkunan alaosa. 7
Kuva 5. Pelaajahahmo puolustaa tornia 300 vihollisilta. 9
Kuva 6. Tornipuolustuspelin CPU kuormitukset 30 FPS:ssä pylväs diagrammina. 10
Kuva 7. Vihollinen, jossa on räsynuken fysiikan törmäystunnistimet. 11
Kuva 8. Mittaustulokset fysiikan ja animaation väliset erot tuodakseen vihollisen kuolemaefektin. 11
Kuva 9. Unityn tekemä Megacity käyttäen DOTS. Kuva otettu Youtube videosta. (Unity, 2018) 12
Kuva 10. Koodin suoritusnopeus pelkällä pääsäikeella ja monisäikeisenä. 13
Kuva 11. Class ja struct-lista muistitallennus. 14
Kuva 12. Burst-tarkastaja ikkuna. 15
Kuva 13. IJob- ja IJobParallel-rajapinnan koodiesimerkki. 17
Kuva 14. Jobin luonti ja ajoitus koodiesimerkki. 18
Kuva 15. Jobille lisätty BurstCompile-attribuutti. 19

Kuva 16. Job-systeemin aikajana Profilerin alaikkunassa. 19
Kuva 17. Klassisen Monobehaviourin ja ECS:n datan ja logiikan toimintaerot. 21
Kuva 18. Klassisen ja ECS:n data haku allokoitumaa muistia RAMista. 21
Kuva 19. CPU:n välimuistitallennus ero klassisen ja ECS datahaussa RAMista. 22
Kuva 20. Entity Debugger-välilehti. 23
Kuva 21. Peliobjektilla ConvertToEntity-skripti liitettynä. 24
Kuva 22. Entiteetin luomisen koodi. 25
Kuva 23. ECS komponentin koodiesimerkki. 26
Kuva 24. IConvertGameObjectToEntity-rajapinnan entiteetti konversio funktio. 27
Kuva 25. Usean komponentin lisäys entiteettiin Authoring-komponentin pohjalla. 28
Kuva 26. ECS systeemin puhdas koodipohja. 29
Kuva 27. TestMovableSystem-systeemin koodi. 30
Kuva 28. Kuutiomuotoinen suljettu tila. 31
Kuva 29. Pallo-objekti. 31
Kuva 30. Pallo-objektin komponentit. 32
Kuva 31. ExplosiveBall-skriptin koodi. 33
Kuva 32. Pallon räjähdys koodi ExplosiveBall-skriptistä. 33
Kuva 33. Monobeviour versio räjähtävän pallon toimintalogiikka. 34
Kuva 34. DOTS versio räjähtävän pallon logiikka. 35
Kuva 35. DOTS räjähtäpallon komponentit. 36
Kuva 36. Räjähtävän pallon entiteetti konversio skripti Authoring tavalla. 37
Kuva 37. ExplosionData-skripti. 37
Kuva 38. ExplosiveIntervalComponent-skripti. 38
Kuva 39. ExplosiveIntervalComponentSystem-skripti. 39
Kuva 40. ExplodedAreaComponentSystem-systeemin OnUpdate-funktion koodi. 40
Kuva 41. ExplosionForcedComponent-komponentin datamuuttujat. 41
Kuva 42. ExplosionForcedComponentSystem-systeemin skripti. 41
Kuva 43. 7000 räjähtäviä palloja kuution sisällä. 42
Kuva 44. Räjähtävien pallojen profilointi tulos raskain suoritushetkellä. 43
Kuva 45. Simuloinnin Jobs-systeemin tiedot. 43
Kuva 46.Simulointi DOTSilla 16 000 räjähtäviä palloja kuutiosäiliössä. 44
Kuva 47. Entiteetti pallojen raskain kuormitushetken profilointi tulos. 44

Kuva 48. Jobs-systeemin profiili 16 00 räjähtävien pallojen simuloinnissa. 45
Kuva 49. Suorituskyvyn mittaustulos Monobehaviour ja DOTS välisen kehitystavan räjähtävien pallojen simuloinnissa. 46

Käytetty terminologia tai käsitteistö

AI – tekoäly

CPU – suoritin tai prosessori

FPS – ruutua per sekunti

PC – tietokone

RAM – luku- ja kirjoitusmuisti, päämuisti

SFX – äänitehosteet

VFX – kuvatehosteet

1 Johdanto

Opinnäytetyössä selvitellään Unitylla kehitetyn pelin suorituskykyongelman aiheuttajia ja etsiä sille mahdollisia ratkaisuja. Pelin suorituskyvyn parantamalla antaa pelaajille paremman pelikokemuksen ja mahdollistaa pelaajamäärän kasvun pelille. Pystytään julkaisemaan pelin monelle pelijärjestelmien laitteistoille, kun jokaisella on omat teknistä tietoa. Jotkut laitteistot ovat tehokkaampi tai heikompi kuin toiset. Tällä hetkellä tunnettuja pelijärjestelmiä, joita pelaajat käyttävät ovat Playstation, Xbox, Nintendo, Android, IOS ja tietokone. Pelin kykenemään pyörimään heikoimmilla laitteilla pystytään tähtäämään suuremmalle peliyleisölle. He ovat sellaisia pelaajia, jotka eivät useasti päivitä tai varaa parempaan laitteeseen. Hyvä suorituskykyinen peli antaa pelikehittäjille enemmän tilaa parantaa muita osastoja, kuten uuden pelimekaniikan lisäämisen, luoda realistisemmat 3D-hahmot, laajentaa pelimaailmaa jne.

Opinnäytetyöhön varten yritetään luoda todella raskasta suorittavaa peliä törmätäkseen erilaisiin suorituskykyongelmiin. Peliä rasitetaan, kunnes aiheutuu pahan suorituskykyongelman eli se suoriutuu huomattavasti hitaammin kuin normaalisti. Tutkitaan ongelman syitä ja etsitään sille ratkaisukeinoja. Suoritetaan pelin suorituskyvyn parannuksen löytämillä ratkaisukeinoilla ja otetaan mittaustulokset talteen. Katsotaan kukin keinon suorituskykyparannuksen suuruuttaa. Tutustutaan pelin suorituskyvyn profilointiin, kun se auttaa paikantamaan helpommin suorituskykyongelmaisen koodin. Saadaan sillä myös selville pelin kuormitukset ajetussa laitteistossa.

Kaikkia suorituskykyongelmien selvittämiseen ja ratkaisemiseen ei varmasti ehditä käymään lävitse tässä opinnäytetyössä, mutta aiheesta kiinnostunut löytää tästä hyödyllisiä tietoja ja suuntaa antavana omien peliprojektien suorituskykyongelmien ratkaisemisessa.

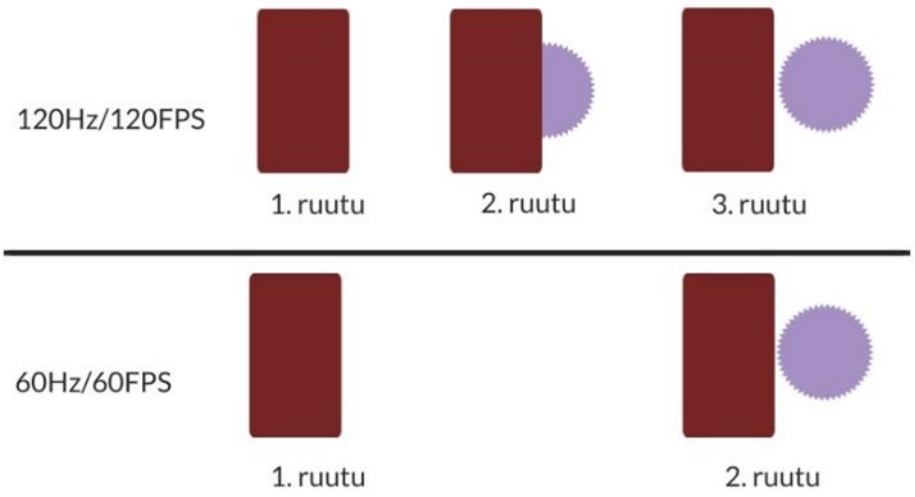
2 Pelisuorituskyvyn tärkeys

Pelisession aikana tapahtuessaan dramaattisia hidastumisia tai hetkellisiä jumiutumisia ovat juuri pelin suorituskykyongelmien oireita. Peli on silloin liian raskasta suoritettavaksi pelijärjestelmän laitteistolle. Pelin suorituskyky ajettavassa laitteistossa pystytään kattomaan, kun mitataan sen FPS arvo sen suorituksen aikana. Korkea arvoinen FPS voidaan antaa pelaajille saumattoman pelikokemuksen, joka on optimoinnin tavoitteena välttääkseen suorituskykyongelmia.

Pelisuorituskyvyn minimi FPS arvo kohde vaihtelee pelijärjestelmän laitteistosta. Nykyiset konsolit ja älypuhelimet ovat yleensä lukittuneet 30 tai 60 FPS:ään, mutta PC:ssä lukitusta voidaan poistaa ja sen FPS arvo voi nousta jopa yli 200. Näytöt ovat paljon kehittyneet, että ne pystyvät piirtämään ruudulleen noin korkeissa FPS arvoissa pyöriville peleille.

Linus Tech Tips-kanvassa todistettiin pelin kykenevän pyörittämään korkeassa FPS arvossa tuo parempia pelikokemuksia, vaikka näytön ruudunpäivitysnopeus on matalampi. He testasivat videossaan pelin pelaaminen korkeissa FPS:ssä ja näytön pystytahdistus lukittuneena ja lukittumattomana 60 Hz, 144 Hz ja 240 Hz ruudunpäivitysnopeuksilla. Testissään oli elektronisen ammattilaisurheilupelaajia testaajina. Testin tuloksena 144 Hz näyttö ja 144 FPS pyörittävä peli tuottivat suuria eroja 60 Hz:en näytön ja 60 FPS pyörivän pelin verrattuna. Korkeissa FPS arvoissa napin painalluksen syöttö viive lyhentyy huomattavasti, joka tekee pelistä enemmän reagoivaa. 144 Hz:n näyttö kykenee piirtämään enemmän ruutuja kuin 60 Hz:n näyttöä, joka antaa sen käyttäjille etulyönti aseman kilpailullisissa peleissä. Syynä huomataan peliruudun päivitys ilmiössä (Kuva 1), jossa pallo poistuu suojasta pelaajan tähdättäväksi. Pelin suorittaessaan 120 FPS:ssä, pallon sijaintia tiedetään 2 kertaa useammin kuin 60 FPS:llä ja 120 Hz:llä näytöllä päivitetään pallon sijaintia peliruutuun 2 kertaa useammin kuin 60 Hz näytöllä. Pelaaja pystyy reagoimaan nopeammin ylemmässä tilanteessa kuin alemmassa (Kuva 1), kun näkee pallon näyttäytyvän aikaisemmin. Näytöillä 144 Hz ja 240 Hz välillä videon mukaan, se ei tuottanut suurta eroja pelaamisessa verrattuna 60 Hz näyttöön. Korkeassa hertsisessä näytössä, peli näyttää enemmän terävämmältä kameran liikuttaessaan. (Linus Tech Tips, 2019)

Kuva 1. Ruudun piirustus 60 Hz:en ja 120 Hz:en välisen ero.



3 Suorituskykyongelmat

Huonon suorituskyvyn aiheuttajia on monta. Yleisin pääaiheuttajat ovat heikko laitteisto tai koodia on huonosti optimoitu. Seurauksena laitteisto ei kykene prosessoimaan pelisimulaation koodia tarpeeksi nopeasti, joka tuottaa pelille matalan FPS arvon.

3.1 Välimuistihuti (Cache Miss)

Liiallinen määrä välimuistihuteja aiheuttaa hitaan koodi prosessoinnin ja aiheuttaa huonon suorituskyvyn. Huti tapahtuu, kun CPU ei löydä pyydettyä dataa välimuistista. Silloin se joutuu hakemaan datan RAMista, joka on todella hidasta. (techopedia, 2013)

CPU:lla on välimuisteja, johon se voi tallentaa väliaikaisesti RAMista poimittuja datoja tai komentoja. Datahakun löytäminen välimuistista antaa paremman suorituskyvyn, kun se on paljon nopeampaa kuin suorittaa datahaun RAMista. CPU:n välimuistissa on kolme erilaista muistitasoja. Niitä luokitellaan L1, L2 ja L3:ksi. Jokaisella CPU ytimillä on oma L1 tason välimuisti. L2 ja L3-tason välimuisteilla, ytimet saatavat jakaa keskenään. (Bitesize, n.d.)

Data pyynnössä, CPU aloittaa hakunsa matalimmasta välimuistitasosta ja jatkuu korkeampaan tasoon. Haku nopeus ja välimuistin tallennuskoko vaihtelee tasoittain. Data hakuaika on

nopeampi matalammassa tasossa kuin korkeimmissa tasoissa, mutta muistikoko on myöskin pienempi. (Teja, 2018)

Datan hakunopeus CPU:n välimuistista vaihtelee laitteistosta. Datan hakeminen sieltä voi nopeutua 25–100 kertaiseksi verrattuna RAMista hakeminen. (Dunstan, 2017)

3.2 Koodi tuottaa liikaa roskaa

Roska ohjelmoinnissa on RAMissa oleva data ilman viittaajaa. Ohjelma varaa ja tallentaa dataansa RAMiin suorituksen aikana. Koodissa usein nähdään new-operaation käytön sijoitellessaan muuttujaan. Se tarkoittaa ohjelma allokoi uutta dataa RAMiin ja asettaa sijoittamaa muuttujaa viittaamaan sitä. Datalle voi olla monta viittaajia. Datan viittaajamäärän tippuessaan nollaan eli muuttuessaan roskaksi, ohjelma suorittaa niiden keräyksen roskankerääjällä (GC) vapauttaakseen varattuja muisteja RAMista. Sen suorittaminen ohjelmassa antaa negatiivisen vaikutuksen ohjelman suorituskyvylle. Ohjelma silloin joutuu odottamaan GC:tä suorittamaan työnsä loppuun ennen kuin se pääsee jatkamaan eteenpäin. Suuren määrän roskan putsaamisen saattaa aiheuttaa pysähtymisiä ohjelmalle. (freeCodeCamp, 2020)

GC:llä on monta algoritmia. Yksi suosituista algoritmeista on sijoitetun datan viitemäärän katsominen. Datan viitemäärän tippuessaan nollaan, GC kerää sen heti pois ja vapauttaa sen varaamat muistit RAMista. Monimutkaisemmissa ja suorituskykyominaisuuksissa algoritmeissa ne eivät saata kerätä roskaa heti viittauksen määrän tippuessaan nollaan, vaan jättää myöhemmäksi kerättäväksi. Sillä tavalla saadaan parempi suorituskyky kuin viitemäärän katsonnan tavalla, mutta se voi johtaa hetkellisiä suuria suorituskyvyn pudotuksia. Syynä se on voinut kerryttää suuren määrän roskaa GC:lle, kun se suorittaa roskankeräämisen. (Wikipedia, 2020a)

4 Profilointi

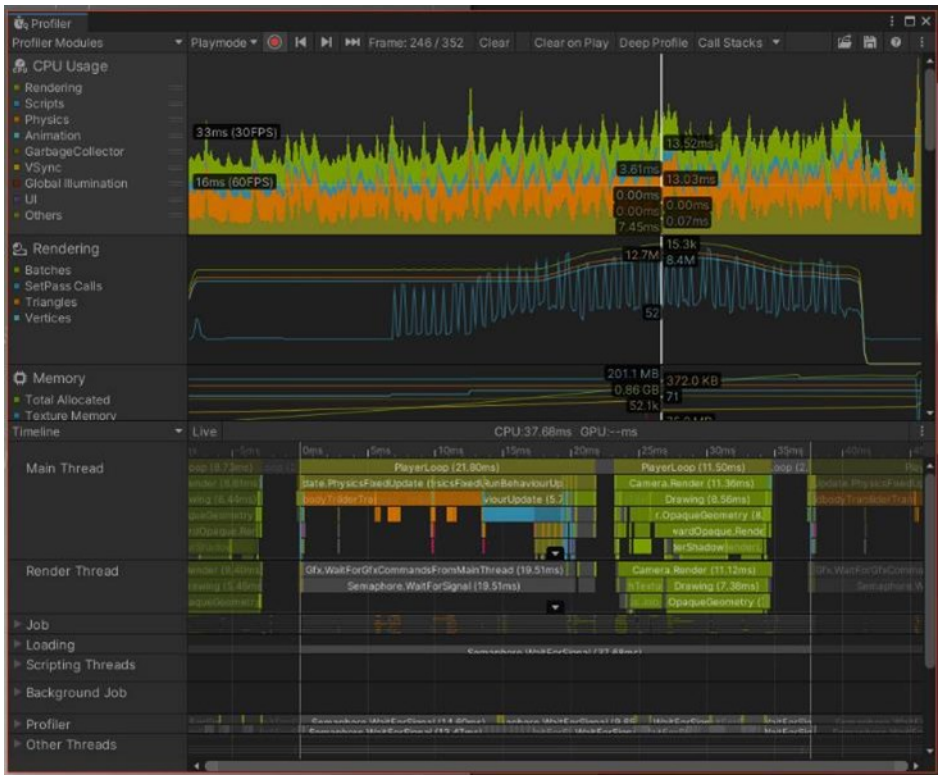
Profilointi on pelin suorituskyvyn valvominen ajetussa laitteistossa. Profiloimalla peliä pystytään analysoimaan pelikoodin käyttäytymisen laitteistossa. Saadaan kerättyä tietoja koodin funktioiden suoritusmäärän, suorituskeston ja niiden aiheuttamat kuormitukset laitteistossa. Näillä kerätyillä tiedoilla helpottavat ja nopeuttavat paikantamaan suorituskykyongelmaisen tai hitaasti

suorittavan koodin. Useasti peliä profiloidaan optimoidakseen sen suorituskykyä. (Wikipedia, 2020b)

4.1 Profiler-työkalu

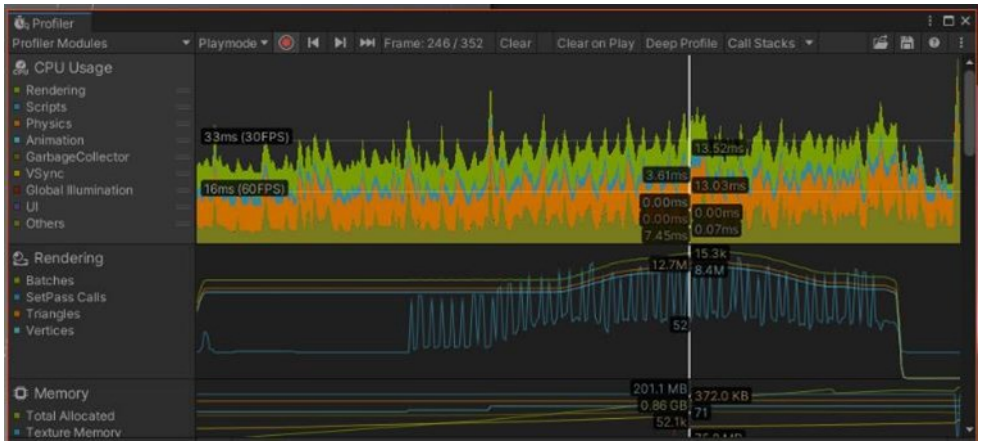
Profiler on Unityn sisäänrakennettu profilointi työkalu. Se kerää ja näyttää tietoja pelissä tapahtunutta kuormituksia laitteistolle, kuten CPU, RAM, renderöinti ja äänet (Kuva 2). Sen antamilla tiedoilla pystytään helposti paikantamaan huonoa suorituskykyistä koodia ja tehdä sille optimointeja. Se alkaa keräämään laitteiston suorituskyvyn tietoja ja piirtämään kaavioita, kun peliä ajetaan editorissa. Työkalu saadaan avattua valikosta Window > Analysis > Profiler. (Unity, 2020a)

Kuva 2. Unityn Profiler-ikkuna.



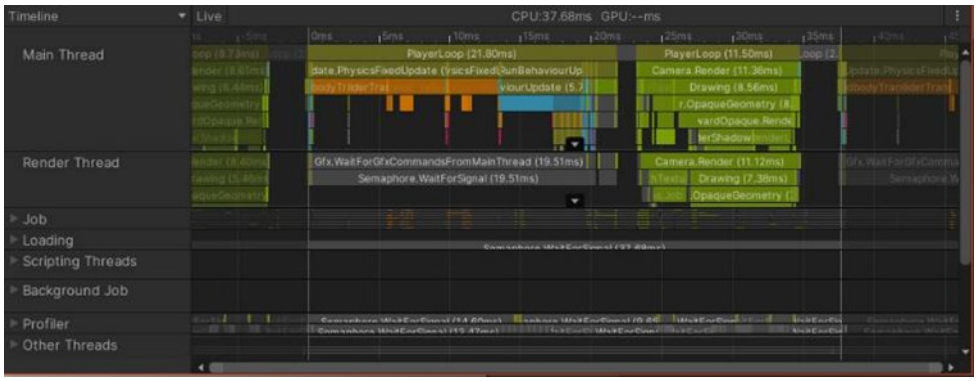
Profiler-ikkunan yläosassa (Kuva 3) näytetään tiedot graafisina kaavioina laitteiston suorituskyvystä. Sen vasemmalla olevaa palstaa näytetään moduuleja, moduulin kategorioita ja kategorian kaaviovärejä. Klikkaamalla kategoriaan voidaan kytketä pois tai päälle sen kaavion piirtämisen. Se helpottaa kaavioiden tarkkailua, kun halutaan vaan nähdä tiettyjä tietoja. (Unity, 2020b).

Kuva 3. Unityn Profilerin ikkunan yläosa.



Profiler-ikkunan alaosassa (Kuva 4) näytetään profilointitiedot yksityiskohtaisempana. Se ilmestyy aluksia tyhjänä, kunnes sen ikkunan yläosasta valitaan haluttua pelin simulointihetkeä. Se näyttää tietoja valitusta simulaatiohetkestä koodin funktiot ja laitteiston kuormitukset. (Unity, 2020b)

Kuva 4. Unity Profilerin ikkunan alaosa.



4.2 Profiler-työkalun käyttö

Suuren suorituskyky pudotuksen voidaan silmämääräisesti huomata katsomalla CPU Usage -moduulia. Äkillisiä jyrkkiä pylväitä tai täysin kattoon osunut kaavio tarkoittaa laitteistolle on tapahtunut suuria kuormituksia. Klikkaamalla jyrkkään kohtaan kaaviosta, päästetään tutkimaan yhden pelisimuloinnin toimintaa. Ikkunan alaosaan alkaa näyttää tai päivittää tietoja CPU:n kuormituksista. Sen näkymää voidaan vaihtaa kolmeen eri tilaan, jotka ovat Timeline-, Hierarchy- ja Raw Hierarchy -tila. Oletuksena se on asetettu Timeline-tilaan, jossa se näyttää suorittavat koodit aikajanoina. Väriä korostetut janat ovat koodissa suorittavia funktioita, jotka tapahtuivat

valitussa aikataajuudessa. Janan pituus kertoo sen koodin suoritusnopeuden ja suoritus hidastuu sen janan pidetessään. Raw Hierarchy- ja Hierarchy-tila näyttävät suorittaneet koodien funktioiden nimet hierarkkisena ja niiden kuormitustiedot. Sillä voidaan järjestää tiedot helposti laskevaan tai nousevaan järjestykseen funktion nimellä, CPU:n käyttöprosentilla, roskienkeräyksen määrällä tai kulunut aikaa funktiossa. Tämä ominaisuus nopeuttaa tietyn tiedon etsinnässä. (Unity, 2020a)

Profilerissa ilmestyy muutamia funktioita, joita voidaan jättää kokonaan huomiotta optimoinnin aikana. Niitä ovat Profiler.CollectEditorStats ja EditorLoop. Ne keräävät Unityn editori tuottamia kuormituksia CPU:lle, joita pyörivät taustalla pelin ajon aikana. Niiden mukaan ottaminen suorituskyvyn mittauksessa tuottaa vääriä mittaustuloksia todellisesta tuloksesta. Tärkein funktio optimoinnissa on PlayerLoop-funktio, kun halutaan seuraamaan ja tietämään pelin kuormittamia tietoja.

4.3 Deep profile-moodi

Profiler kerää yleensä Start, Update tai vastaavat funktiokutsuntoja profiloinnin aikana. Kaikkia funktiokutsuntoja profiloidaan Deep profile -moodin päässä ollessaan. Sitä käytetään, kun halutaan tarkemman paikannuksen suorituskyvyn ongelman aiheuttajaa. Se on hyvä käyttää pienempien ohjelmien profiloinnissa. Suurissa ohjelmissa moodia ei saata kyetä käyttämään, kun se käyttää paljon laitteiston resurssia ja muistia suorittaakseen profilointia ohjelmalle. (Unity, 2020b)

Moodin käyttöönotto onnistuu vaihtamalla Deep profile -painikkeen tilaa. Se sijaitsee Profiler-ikkunan yläosassa.

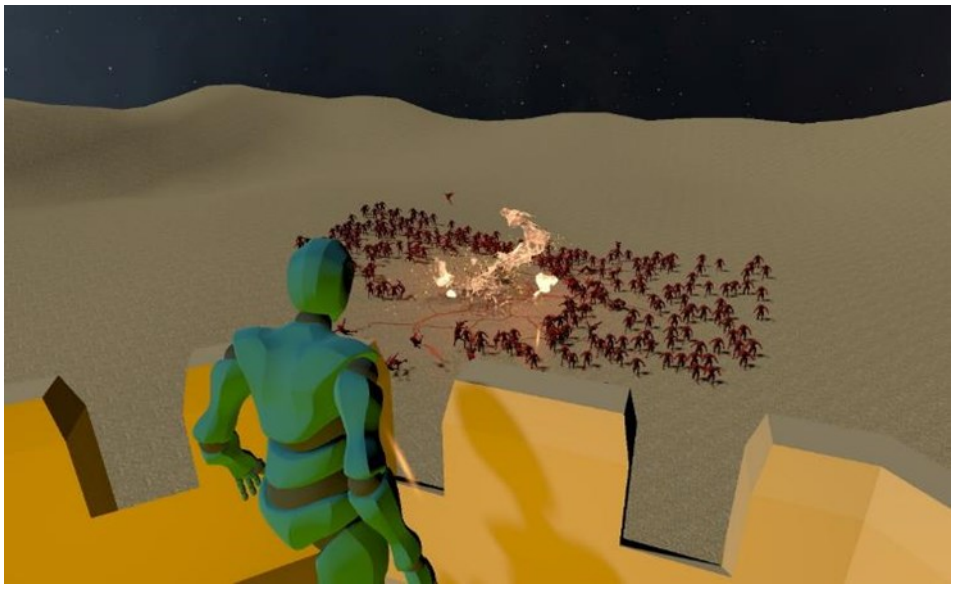
4.4 Muut huomautukset

Pystytahdistusta ei tarvita optimoinnissa, koska se antaa väärän FPS arvon. Se synkronisoi pelin FPS:ää samaan näytön virkistystaajuuteen. Esimerkiksi pelin suorituskyky ollessaan 120 FPS ja näytön virkistystaajuus on 60 Hz. Pelin suorituskyky muuttuu 60 FPS:ksi pystytahdistuksen ollessaan päällä. Pystytahdistuksen kaavion piirtäminen voidaan kytkeä pois päältä klikkaamalla VSync-kategoriaa, joka näkyy CPU Usage -moduulin alla.

5 Suorituskykyongelmien etsintä ja ratkonta

Kehitettiin raskaasti suorittavaa peliä törmätäkseen erilaisiin suorituskykyongelmiin. Pelille annettiin yleiset toiminnallisuudet, jotka ovat fysiikka, animaatio, SFX, VFX ja AI. Pelin pelimekaniikaksi päädyttiin tornipuolustuspelin tapaiseen peliin. Sen tyylinen peli pystyttiin helposti rasittamaan pelin suorituskykyä, kun vihollisille annettiin tekoälyt, äänet, animaatiot ja fysiikan liittyvät törmäystunnistimet. Kasvattamalla vihollisen kutsumismäärän pelikentälle (Kuva 5) saatiin rasitettua kaikkia toiminnallisuudet yhtäaikaisesti paitsi VFX:ää.

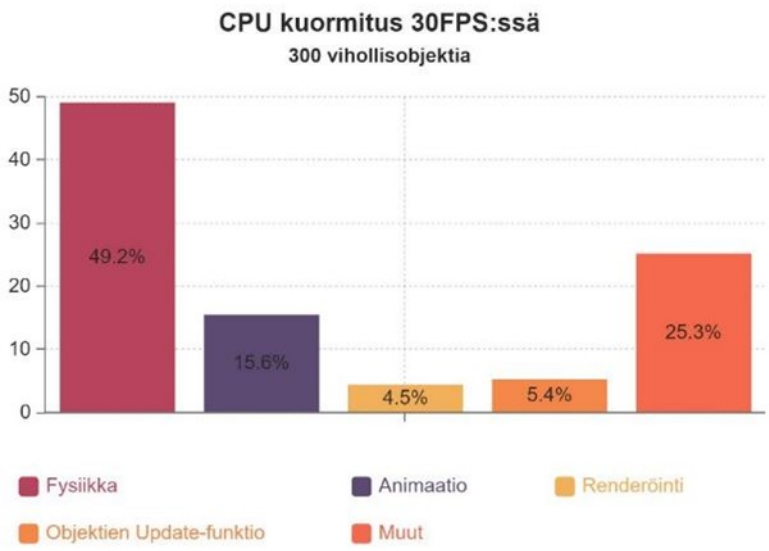
Kuva 5. Pelaajahahmo puolustaa tornia 300 vihollisilta.



Pelin vihollisille annettiin räsynukkefysiikat eli käyttäytyy kuin tajuttomana oleva henkilö. Monessa peleissä on sellainen ominaisuus, kuten suosittu Counter-Strike pelissä myös käyttää sitä efektiä hahmon kuollessaan. Rasittaakseen suorituskykyä VFX:llä lisättiin pelaaja hahmo, joka osaa käyttää taikoja. Pelaaja voi käyttää rajattomasti taikoja tuhotakseen kentässä olevia vihollisia. Taiat tuottavat räjähdysfysiikkaa, joka lähettää räjähdyssäteessä olevat viholliset lentoon.

Aloitettiin rasittamaan peliä saadakseen mittaustuloksia, jotta pystyttäisi vertailemaan optimoidun version tuloksia. FPS kohteena valittiin 30, koska se oli yleisesti käytetty arvo suorituskyvyn profiloinnissa. Kentälle pystyttiin kutsumaan 300 vihollisobjektia, kunnes pelin suorituskyky iski FPS mittauskohteeseen. Peliä profilointiin ja huomattiin mittaustuloksesta (Kuva 6) fysiikan olevan pelin raskain CPU:n kuormittaja.

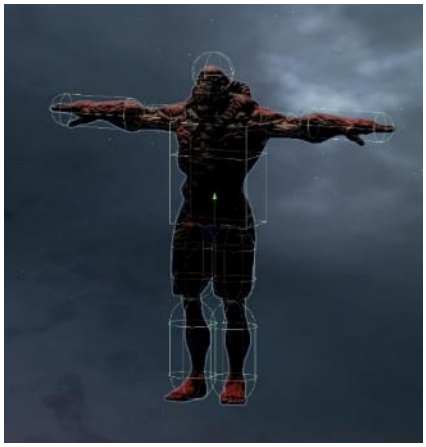
Kuva 6. Tornipuolustuspelin CPU kuormitukset 30 FPS:ssä pylväs diagrammina.



5.1 Fysiikan suorituskyvyn ratkaiseminen

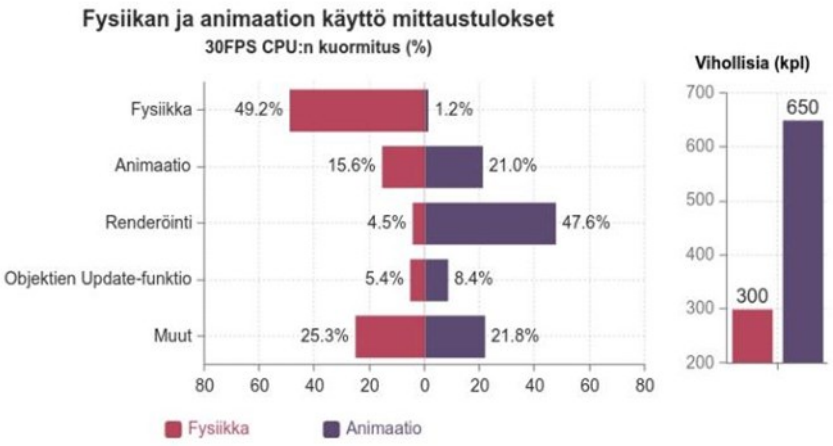
Tornipuolustuspelin fysiikan suuren kuormituksen syynä oli vihollisen räsynukke efekti. Siihen tarvittiin 11 törmäystunnistinta ja 11 Rigidbody-komponenttia (Kuva 7). Ne kaikki kuuluivat fysiikan kuormitukseen. Kokeiltiin poistaa räsynuken efektin törmäystunnistimet viholliselta ja lisää vasta sen kuoltuaan, mutta se toikin odottamattomia matalia FPS arvoja pelin aikana. Suuren määrän vihollisia kuollessaan yhtäaikaisesti aiheutti pelille suuren ja äkillisen suorituskyvyn pudotuksen. Räsynukke-efektin lisääminen kuolleille palautti saman CPU:lle pelifysiikan kuormitusongelman. Tällä keinolla pystyttiin nostamaan vihollisen kutsumismäärän, mutta oli todella epäluotettava ratkaisu. Se ei antanut tasaista FPS niin kuin vihollisilla räsynukke-efekti valmiina lisättynä.

Kuva 7. Vihollinen, jossa on räsynuken fysiikan törmäystunnistimet.



Vihollisen räsynukkefysiikka kuolemaefektiä kokeiltiin korvata animaatiolla vähentääkseen pelifysiikan kuormitusta. Se muutos antoikin suuren suorituskykyparannuksen, kun vertailtiin molempien mittaustuloksia (Kuva 8). Vihollisia pystyttiin kutsumaan tuplasti enemmän ja FPS pysyi varsin tasaisena. Tästä syystä yritettiin välttää fysiikkapohjaisien efektien käyttämisen ja korvattiin ne animaatioilla, kun ne olivat mahdollista.

Kuva 8. Mittaustulokset fysiikan ja animaation väliset erot tuodakseen vihollisen kuolemaefektin.



5.2 Unityn toinen pelikehitystapa DOTS

Paremman suorituskyvyn parannuskeinojen etsiessä, löydettiin DOTS(Data-Oriented Technology Stack) eli dataan suuntautunut teknologiapino. Se on Unityn toinen tapa kehittää peliä klassisen Monobehaviourin käytön sijasta. Pelin kehittäminen sillä sen mukaan antaa pelille oletuksena korkeata suorituskykyä. (Unity, n.d.-b)

Huomattiin DOTS-systeemin antavan tavattoman korkean suorituskyvyn pelille, kun katsottiin sen pyörittävän Megacityä (Kuva 9) Youtube videossa. Megacityssa oli 4,5 miljoona mesh-renderöijää, 100 000 äänilähdettä, 5 000 dynaamisia ajoneuvoja, 200 000 ainutlaatuisia objekteja per rakennuksessa ja se pyörii tasaisessa 60 FPS:ssä. Näytettiin vielä videon lopussa, että sillä on noin hyvä suorituskyky pyörittääkseen mobiililaitteessakin. (Unity, 2018)

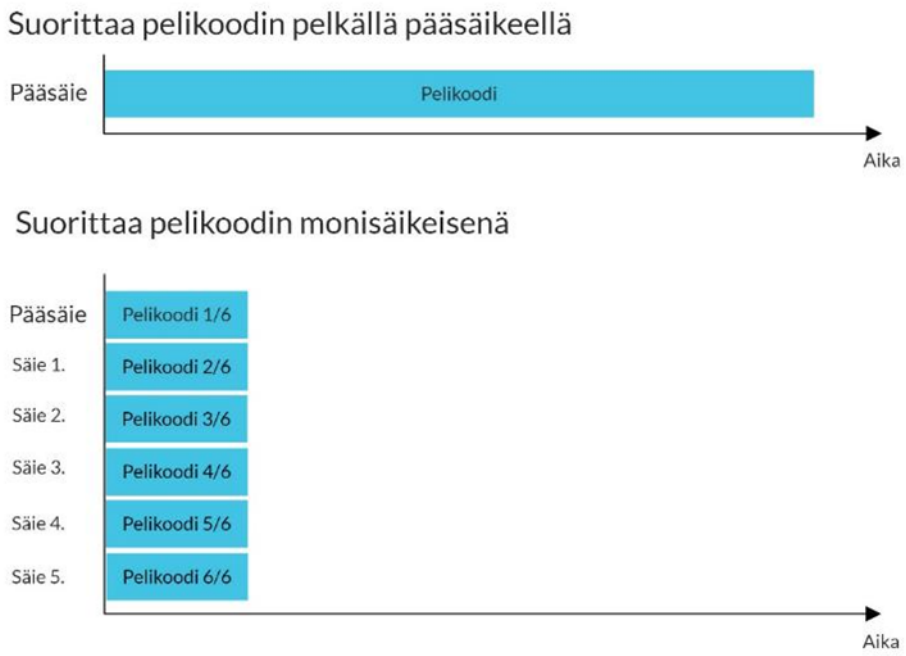
Kuva 9. Unityn tekemä Megacity käyttäen DOTS. Kuva otettu Youtube videosta. (Unity, 2018)

Tornipuolustuspelissä suoritetut suorituskykyparannukset ei edes päästy lähellekään Megacityn suorituskykytasoa. Tutkiessaan DOTS-systeemiä saatiin selville Monobehaviourin kehitystava ja olio-ohjelmointityyli aiheuttivat oletuksena liikaa välimuistihuteja. Monet Unityn valmiit työkalut ei myöskään hyödynnä CPU:n moniytimien ominaisuutta, jossa DOTSissa käytetään monisäikeistääkseen pelikoodia nostaakseen pelin suorituskykyä monikertaiseksi. Unityn työkalujen lähdekoodeihin ei pystytä tehdä muutoksiakaan korjatakseen tilannetta.

5.3 Monisäikeinen koodi

Ohjelman koodia ajetaan oletuksena yhdellä CPU:n ytimen säikeellä, joka on ohjelman pääsäie. Sen käyttö on turvallista, mutta hidasta. Se joutuu suorittamaan kaikkia ohjelman koodia yksin. Tähän ratkaisuun on kirjoittaa monisäikeistä koodia, jossa hyödynnetään CPU:n moniydin ominaisuutta. Jaetaan suorittavaa koodia moneen osaan muille CPU:n ytimen säkeille, jotta voidaan suorittaa samanaikaisena (Kuva 10). Ohjelman suoritusaika lyhentyy monikertaiseksi riippuen CPU:n ytimen määrästä. (Unity, 2019a)

Kuva 10. Koodin suoritusnopeus pelkällä pääsäikeella ja monisäikeisenä.

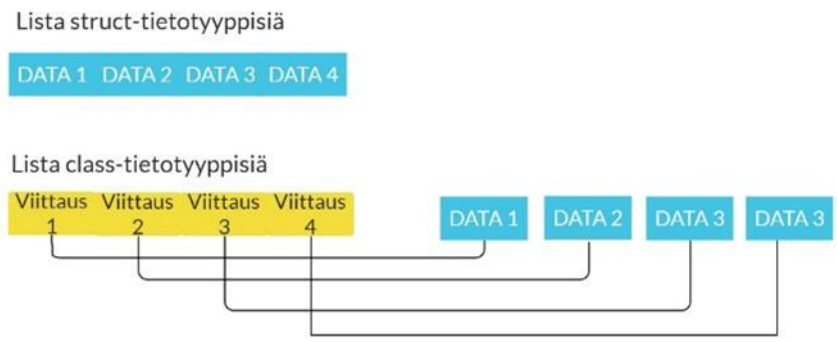


5.4 Monobehaviour olio-ohjelmointi class sijaan DOTS datakeskeinen ohjelmointi struct

Data merkitseminen class- tai struct-tietotyypiksi vaikuttaa suorituskykyyn. Class on viittaaja eli se viittaa johonkin RAMissa olevaan dataan ja struct on arvotyyppiä eli se on itse data. Struct-tietotyyppinen data on suorituskykyisempi kuin class-tietotyypillä. Syyn huomataan molempien tietotyypin datalistan tallennus eroista (Kuva 11). Struct-listassa olevat datat ovat tallennettu peräkkäisenä RAMiin ja class-listassa datat saattavat tallentua joka puolella RAMia classin polymorfismin ominaisuuden takia. Polymorfismi sallii classin dataa tulla erikokoisena, jonka takia

se on mahdotonta tallentaa datat vierekkäin kuin struct-listalla. Struct-tietotyyppinen data koko ei muutu. CPU datahaku on paljon nopeampi, kun datat ovat viereisenä tallennettuina RAMissa. Se vähentää välimuistihuteja CPU:lle. Tämän takia class-listalle saattaa aiheutua useita välimuistihuteja. (Held, 2019)

Kuva 11. Class ja struct-lista muistitallennus.



Monobehaviour olio-ohjelmointi kehitysmenetelmällä törmätään useisiin välimuistihuteihin juuri sen class-tietotyypin muistitallennustavan takia. Sen komponentit ovat class-tietotyyppisiä, niin niiden datat ovat voineet tallentua joka puolelle RAMiin. DOTS ehkäisee Monobehaviourin välimuistihutiongelman, kun se on datakeskeistä ohjelmointina ja käyttää struct-tietotyyppisiä datoja. DOTS keskittyy datojen järjestämisen ja tallentamisen vierekkäin RAMiin, joka tekee siitä välimuistiystävällinen CPU:lle.

6 DOTSin tavaton suorituskyvyn toiminnallisuus

DOTS koostuu kolmesta toiminnallisuudesta, jotka ovat Jobs, ECS ja Burst. Jobs hoitaa koodin monisäikeistämisessä. Se hyödyntää kaikkia CPU:n moniytimien ominaisuuksia ja pystyy jakamaan pelikoodia moneen osaan rinnakkaina suoritettavaksi. ECS helpottaa korkeata suorituskykyistä koodin kirjoittamista ja uudelleen käytettävyyttä. Burst kääntää koodia vielä korkeaan optimoivaan konekieleen, joka lisää suuren suorituskyvyn pelille.

6.1 Burst

Burst kääntää koodia korkeampi optimoitu konekieleen, joka parantaa laitteiston suorituskykyä. Se on suunniteltu pääosin Jobs-systeemille suorittamaan tehokasta monisäikeistä koodia. (Unity, n.d.-c)

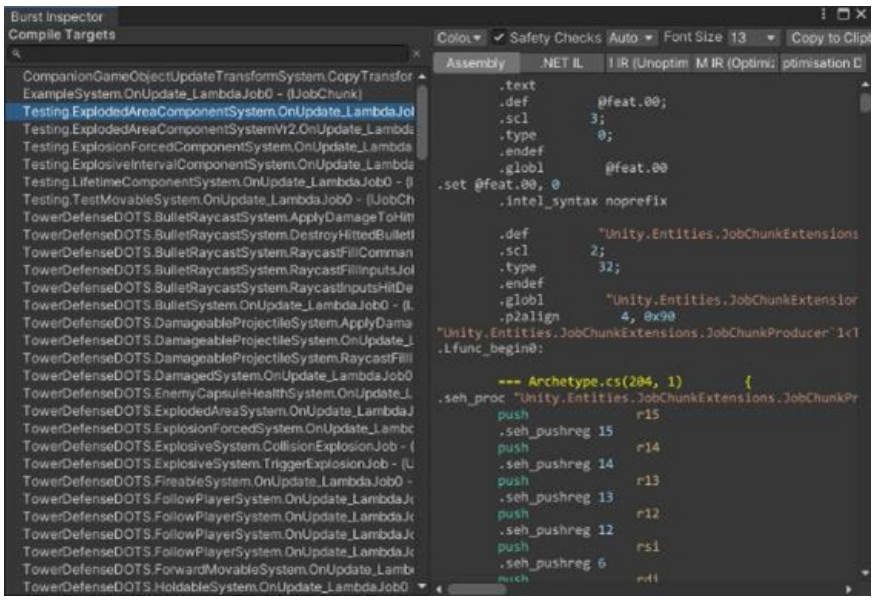
6.1.1 Burst käyttöönotto

Burst-kääntäjä tulee vakiona Unity pelimoottorin asennuksessa. Sen voidaan tarkistaa tai ladata tarvittaessa Unityn paketinhallinasta. Burst-kääntäjä toimii heti kytkettyään päälle Burst-valikosta, joka on Jobs-valikon sisällä. Silloin se kääntää kaikki Burst ominaisuutta hyödyntävät ja yhteensopivat Job-objektit korkeaan optimoivaan konekieleen. (Unity, n.d.-c)

6.1.2 Burst-tarkastaja

Burst-tarkastaja näyttää kaikkia luotuja Job-objekteja ohjelmassa. Sen ikkunaa (Kuva 12) voidaan avata navigoimalla Jobs > Burst > Open Inspector. Valkoisena tekstinä oleva Job-objektin nimi tarkoittaa pystyvän käyttää Burst-kääntäjää. Niiden nimen olevan harmaana eivät pysty hyödyntämään Burstia, kun ne eivät täytä ominaisuuden vaatimuksia. Klikkaamalla Jobin nimeen nähdään sen koodi natiivisena koodina, joka ilmestyy ikkunan oikealle puolelle. (Unity, n.d.-c)

Kuva 12. Burst-tarkastaja ikkuna.



6.2 Jobs

Paketti helpottaa monisäikeisen koodin kirjoittamista Unityssa. Se on turvallinen ja helppo tapa kirjoittaa monisäkeistä koodia. Burst-kääntäjä käytössä Jobsin kanssa antaa merkittävän suorituskyky parannuksen pelille. (Unity, 2019b)

Jobs huolehtii säikeiden kompleksisen käsittelyn, valvonnan ja monisäikeisen koodin kilpailuolosuhteiden estämisen. Kilpailuolosuhteella tarkoitetaan rinnakkaina suorittavat koodit yrittävät hakea ja muuntaa samaa dataa samanaikaisesti, joka johtaa koodin luku- tai kirjoitusvirheisiin.

Jobsia hyödyntämällä, kehittäjät voivat keskittyä vaan koodin logiikan tekemiseen ja syöttää valmiin sen Jobsille hoidettavaksi. Monisäikeisten koodien kuormitus on helppo profiloida, kun kaikki Job-systeemiä käyttävät koodit näkyvät Unityn Profilerissa.

6.2.1 Jobs-systeemin käyttö

Käyttääkseen Job-systeemiä Unityssä on ensin asennettava paketinhallinasta:

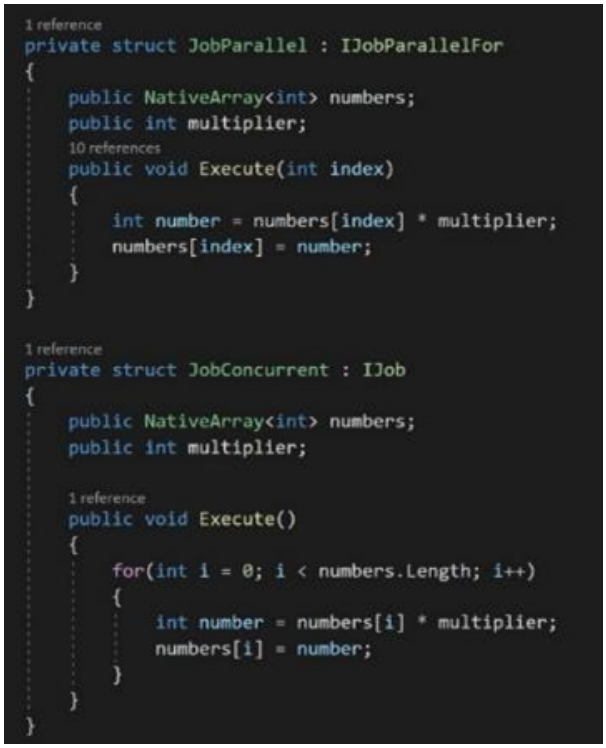
- Jobs
- Collections
- Mathematic

Jobsilla on kahdenlaista toiminnallisuutta. Koodin ajaminen samanaikaisena tai rinnakkaina. Samanaikaisesti koodin suorittaminen kestää yhtä kauan kuin ajaminen pääsäikeellä, mutta etuna on toinen säie hoitaa sen työn ja pääsäikeelle voi hoitaa muita koodeja sillä välin. Rinnakkaina suorittaminen hyödynnetään kaikkia saatavilla olevat säikeet. Säikeiden määrää vaihtelee CPU tuotteesta. Koodia pilkotaan käytettävissä oleviin säikeisiin rinnakkaina suoritettavasti. Suoritusaika vähentyy moninkertaisesti CPU:n säikeiden määrän mukaan.

Oman jobin luonti onnistuu lisäämällä sen struct-tietotyyppiset rajapinnat, jotka ovat IJob-, IJobFor-, IJobParallelFor- ja IJobParallelForTansform-rajapinta. Execute-funktio tulee rajapinnan mukana, jonka sen parametri vaihtelee rajapinnan toiminnallisuuden mukaan (Kuva 13). For-sana

rajapinnan nimen tarkoittaa sen Execute-funktio kutsutaan samalla lailla kuin for-silmukassa, jonka toiston määrää asetetaan sen jobin ajoittamisasetuksessa. IJobParallelForTransform-rajapinta on juuri tehty Transform-komponentti varten vähentääkseen koodin kirjoittamista.

Kuva 13. IJob- ja IJobParallel-rajapinnan koodiesimerkki.



```
1 reference
private struct JobParallel : IJobParallelFor
{
    public NativeArray<int> numbers;
    public int multiplier;
    10 references
    public void Execute(int index)
    {
        int number = numbers[index] * multiplier;
        numbers[index] = number;
    }
}

1 reference
private struct JobConcurrent : IJob
{
    public NativeArray<int> numbers;
    public int multiplier;

    1 reference
    public void Execute()
    {
        for(int i = 0; i < numbers.Length; i++)
        {
            int number = numbers[i] * multiplier;
            numbers[i] = number;
        }
    }
}
```

Jobin luominen onnistuu koodirivillä 132 new-operaattorilla (Kuva 14) niin kuin normaalisti tehdään muissakin class-tietotyypeille. Asetetaan sille tarvittavat arvot riveissä 133–136 sen aaltosulkeiden sisälle ja rivillä 138 ajoitetaan sitä Schedule-funktiolla. Luotiin tyhjän NativeArray-datalistan rivillä 124 jobille varten, jonka taulukon suuruudeksi asetettiin 1000. Jobille pystytään syöttää vain niitä listoja, jotka alkavat Native-nimellä. Listan luonnin parametrissä Allocator-enumluokan vakioarvo TempJob on yksi manuaalisen muistin varauksentavoista. Muistin varatut listat pitää silloin myös vapauttaa manuaalisesti Dispose-funktioilla, jossa nähdään rivillä 149. Ilman vapautusta aiheuttaa muistivuotoja, joka hidastaa peliä ja sen seurauksena aiheuttaa sen kaatumista. Syynä se jatkuvasti luo uutta dataa RAMiin ja ei koskaan vapauta vanhoja luotuja datoja muistista. Vanhat datat jäävät viittaamattomaksi ja GC eivät kerää viittaamatonta Native-listojen dataa automaattisesti. Datalistaa rivillä 127 täydennettiin for-silmukassa i-muuttujan arvolla. Tämä koodi on vain esimerkkinä. Oikeassa tilanteessa listaan täydennetään raskaita suorittavia koodien datoja, jotka tarvitseva Jobsin monisäikeen toiminnallisuutta. Jobsin

ajoittaminen palauttaa JobHandle-objektin. Sillä on Complete-metodi, jonka käyttäessään ohjelma odottaa jobin suorittamisen loppuun ennen kuin jatkaa koodin seuraavalle riville. Metodia usein käytetään ennen listan muistin vapautuksessa, kun halutaan varmasti tietää Jobin olevan valmis. Muistin vapautus kesken Jobia aiheuttaa koodivirheitä. JobHandle-objektia käytetään myös muiden jobin ajoittamisessa, joka näkyy rivillä 144 toisen jobin ajoittamisen asetuksessa. Näin tehdään välttääkseen kilpailutilanteita tai job tarvitsee toisen jobin ensin käsitellä tarvitsemansa datat ennen kuin pääsee suorittamaan.

Kuva 14. Jobin luonti ja ajoitus koodiesimerkki.

```
121  private void TestJobFunction()
122  {
123      int arraySize = 1000;
124      NativeArray<int> listNumbers = new NativeArray<int>(arraySize,
125          Allocator.TempJob);
126
127      for(int i = 0; i < arraySize; i++)
128      {
129          listNumbers[i] = i;
130      }
131
132      JobConcurrent jobConcurrent = new JobConcurrent
133      {
134          numbers = listNumbers,
135          multiplier = 2
136      };
137
138      JobHandle jobConcurrentHandle = jobConcurrent.Schedule();
139
140      int minAmountEaBatch = 32;
141      JobParallel jobParallel = new JobParallel {
142          numbers = listNumbers,
143          multiplier = 3
144      };
145      JobHandle jobParallelHandle = jobParallel.Schedule(arraySize,
146          minAmountEaBatch, jobConcurrentHandle);
147      jobParallelHandle.Complete();
148
149      listNumbers.Dispose();
150  }
```

6.2.2 Burst käyttöönotto Jobille

Job ei käytä Burstiä ellei siihen päälle ole lisätty BurstCompile-attribuuttia (Kuva 15). Sen attribuuttia löytyy Unity.Burst-nimitilasta. Sen toimivuus Burstin kanssa voidaan tarkistaa Burst-tarkastaja ikkunasta. Sen nimen olevan valkoisena tarkoittaa sen pystyvän käyttää Burstia.

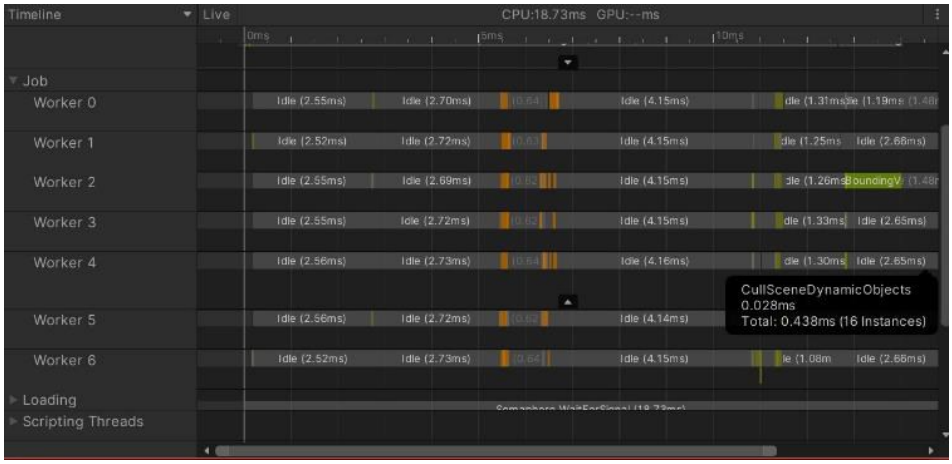
Kuva 15. Jobille lisätty BurstCompile-attribuutti.

```
[BurstCompile]
1 reference
private struct ParallelJob : IJobParallelFor
```

6.2.3 Job-systeemin profilointi

Kaikki ajoitetut jobien koodit pystytään seuraamaan Profilerin alaikkunassa, joka on Timeline-tilassa. Tämä helpottaa sen monisäkeisen koodin suorituskyvyn valvontaa ja samalla varmistaa sen toimivuudesta. Ajoitettujen jobien tiedot nähdään Job-nimisen kategorian alla, kun navigoidaan Profilerin alaikkunan vasenta palstaa. Klikkaamalla kategoriaan, se näyttää Worker-alakategorioita (Kuva 16). Worker vastaa CPU:n yhtä ydintä ja sen määrää riippuu CPU:n ydin määrän mukaan. Sen aikajanalle alkaa täyttyä, kun ajastetaan työtä Job-systeemin kautta. Workerin ollessaan pitkään aikaan idle-tilana tarkoittaa ohjelmassa ei hyödynnetä täysin Jobsin koodin monisäikeisyyttä. Janaan täyteen pakattu funktioita tuo optimaalisen suorituskyvyn pelille.

Kuva 16. Job-systeemin aikajana Profilerin alaikkunassa.



6.3 Entity Component System

ECS on toisenlainen tapa kirjoittaa pelikoodia Unityssä, joka antaa pelille oletuksena korkeata suorituskykyä. Se keskittyy struktuurimaiseen ohjelmointityyliin. Sen toiminnallisuus koostuu kolmesta elementistä, jotka ovat entiteetti, komponentti ja systeemi. Se suuntautuu data ohjelmointi tyyliin, jossa tietotyypit ovat structeja.

6.3.1 Entiteetti

Entiteetti on vastaavanlainen kuin Unityn tyhjä peliobjekti, jolla pystytään erottelemaan suuresta data joukosta. Sen itsenään ei tee mitään ilman data komponentteja liitettyinä.

6.3.2 Komponentti

ECS:n komponentti säilyttää pelkästään dataa, jolla voidaan liittää entiteettiin. Se on kuin klassisen komponentin jäsenmuuttuja. Se on systeemeille luettavaksi ja muunneltavaksi. Systeemi löytää sen datan, kun se on liitetty entiteettimaailmaan luodattuun entiteettiin.

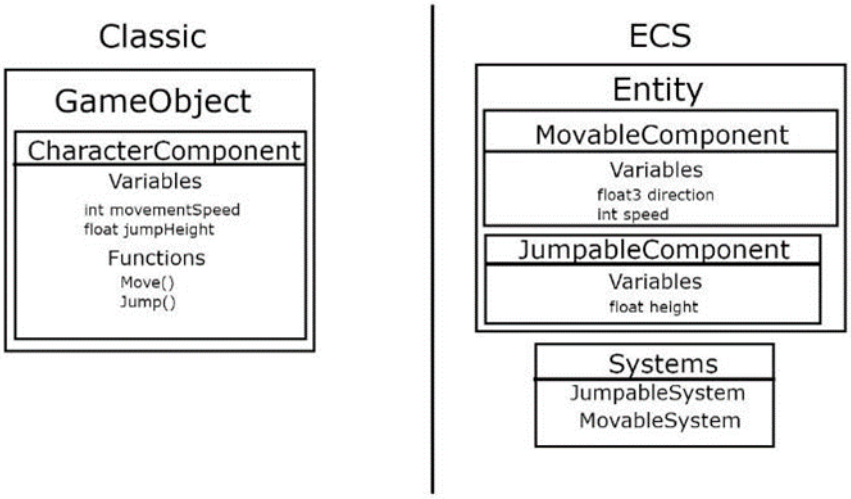
6.3.3 Systeemi

Systeemi toimii logiikkafunktiona, joka muuntaa komponenttien datat. Sillä on samankaltaisia funktioita kuin Monobehaviourillä, joista suosituin on Update-funktio. Se suorittaa logiikkansa komponenteille ja tarvittaessa tehdä datamuunnoksia. Se ei ainoasta suorita yhtä komponentin data niin kuin Monobehaviourissa vaan se suorittaa kaikkia komponenttia kerrallaan. Esimerkiksi klassisen peliobjektin komponentissa on Liiku-logiikkafunktio, joka suoritetaan Update-funktiossa. Sen suoritettuaan se jatkaa toiselle koodiriville suorittamaan toista logiikkafunktiota, kunnes se päättyy aaltosulkeisiin ja toinen komponentin vuoro aloittaa suorittamaan Update-funktiota. Systeemissä Liiku-logiikkafunktio suoritetaan kaikille Liiku-komponenteille. Tämä mahdollistaa Jobsille koodin monisäikeistämisen ja ECS:n muistihallinnan ansiosta se vähentää välimuistihuteja, kun komponenttien datat ovat tallennettu tiiviinä ja viereisenä asetettuina muistissa.

6.3.4 ECS toiminnallisuus

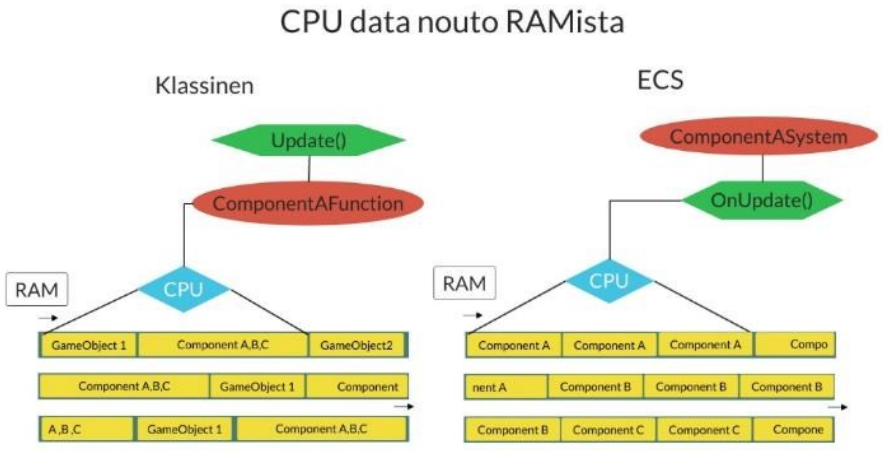
ECS:ssä data ja logiikka ovat eroteltu toisistaan (Kuva 17). Klassisella tavalla Monobehaviour-skriptin periytyneellä on jäsenmuuttujia eli dataa ja logiikkafunktioita, jotka käsittelevät sen dataa ovat yhdessä (Kuva 17).

Kuva 17. Klassisen Monobehaviourin ja ECS:n datan ja logiikan toimintaerot.



ECS:ssä erotellaan data ja logiikka tehdäkseen paremman datahallinta arkkitehtuurin. Se antaa huomattavasti paremman suorituskyvyn, kun pystytään pakkaamaan samankaltaiset datat samaan pakettiin sijoitellessaan muistiin. CPU datapyynnössä ei haeta pelkästään pyydettyä dataa. Se tuo myös mukanaan viereiset datat, jos sillä on vielä tilaa pyydetyn datan jälkeen (Kuva 18). ECS:n systeemien ideana on käsitellä dataa kimpaleina, jotta CPU välttyisi välimuistihuteja.

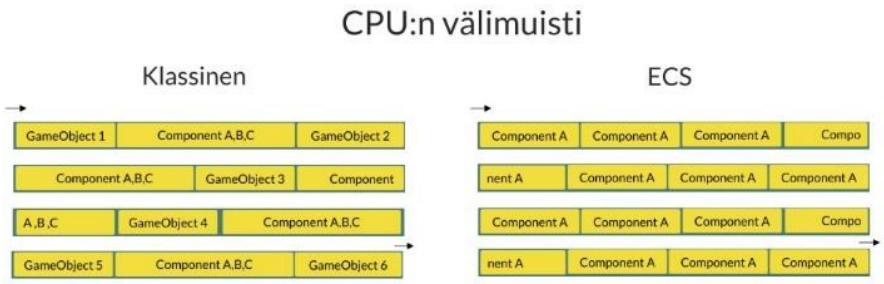
Kuva 18. Klassisen ja ECS:n data haku allokoitumaa muistia RAMista.



Klassisen eli Monobeviourin peliobjektien ja komponenttien kehitystavalla törmätään useisiin välimuistihuteihin sen muistinhallintatyylin takia. CPU datahaussa saattaa tallentua käyttämättömiä datoja välimuistiin. Esimerkiksi klassisen Monobehaviourin ja ECS:n

välimuistitallennuksen ero CPU haettaessaan komponentti A:n datoja (Kuva 19). ECS:ssä CPU löytää A-datoja välimuististaan paljon useamman ennen kuin sille iskee välimuistihudin verrattuna klassisella. CPU klassisessa on joutunut tallentamaan välimuistiinsa myös B- ja C-datan haettaessaan A-datan, jotka ovat turhia datoja. (Baumel, 2019)

Kuva 19. CPU:n välimuistitallennus ero klassisen ja ECS datahaussa RAMista.



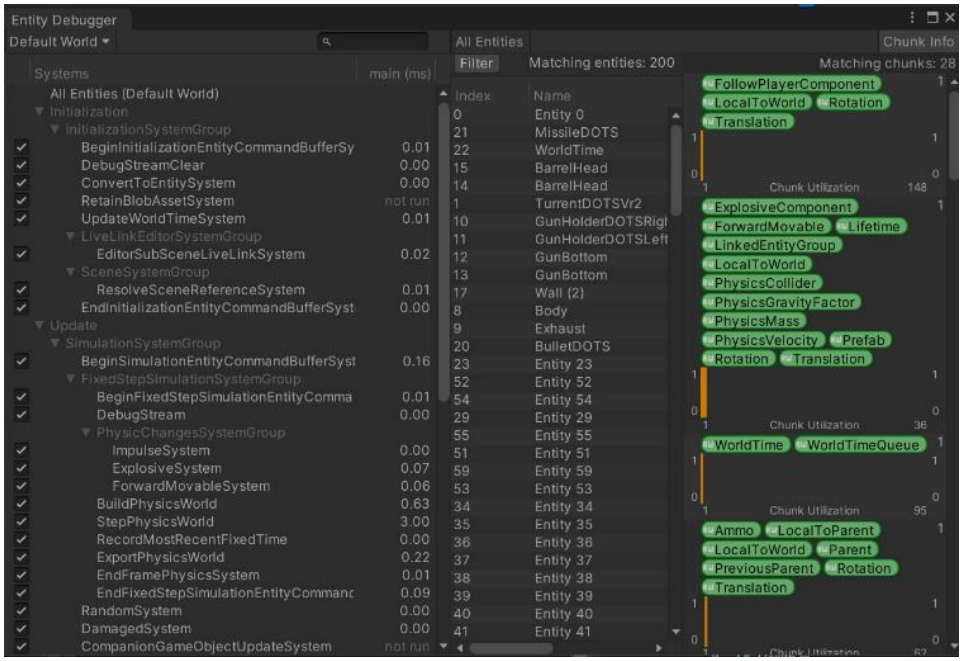
6.3.5 Entity Debugger -välilehti

Peliobjektit ilmestyvät hiearkiaan luodessaan niitä kentälle, mutta luodut entiteetit eivät ilmesty sinne. Entiteetit ilmestyvät Entity Debugger-välilehdessä (Kuva 20), josta voidaan avata navigoimalla Window > Analysis > Entity Debugger.

Välilehdessä näyttää kaikkia luotuja entiteettejä All Entities -palstassa, joka on oletuksena oikealla puolella ikkunaa. Entiteetin komponentti dataa voidaan seurailla Inspector-välilehdestä, kun entiteettiin klikataan. Entities-lehdestä näkyy myös entiteettien data ryhmiä. Samankaltaisia komponentteja entiteetissä sijoitetaan samaan data ryhmään. Klikkaamalla data ryhmään, välilehti näyttää pelkästään ryhmässä olevia entiteettejä. Näin voidaan suodattaa datahakua helpottaakseen profilointia.

Vasemmanpuoleisen ikkunasta näkyy kaikkia luotuja ECS:n systeemejä. Se näyttää systeemin suoritusjärjestyksen ja -keston. Systeemit suorittavat ylimmäisestä ensin ja jatkuu alaspäin. Suoritusjärjestystä voidaan vaihtaa lisäämällä UpdateInGroup-, UpdateBefore- ja UpdateAfter-attribuuttia systeemin skriptiin. Systeemejä voidaan kytkeä pois ja päälle ajon aikana vaihtamalla sen nimen vasemmalla puolella olevan laatikon tilaa.

Kuva 20. Entity Debugger-välilehti.



6.3.6 ECS kehitysympäristön pystytys

Työskennelläkseen ECS kehitysympäristössä vaaditaan joitakin paketteja ladattuina paketinhallinnassa, jotka ovat:

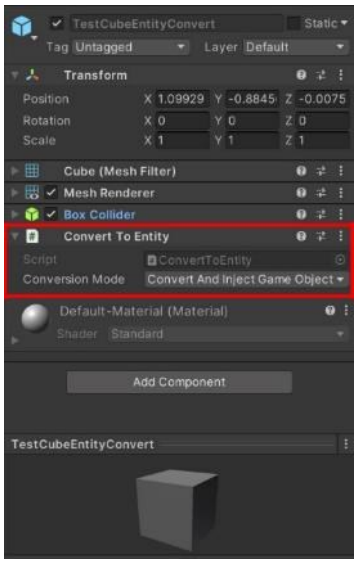
- Jobs
- Entities
- Mathemathics
- Hybrid Renderer
- Unity Physics
- Burst

Listassa on muutamia paketteja, joita ei ole vielä mainittu ennen. Mathemathics-paketissa sisältää struct-tietotyyppisiä data versioita Unityn matemaattiset class-tietotyyppiset objekteista, kuten Vector3 ja Vector2-luokka. Ne ovat tehty ECS:lle varten. Hybrid Renderer-paketti on entiteettien renderöijä. Entiteetti ei muuten renderöi peliin käyttäen Unityn sisäänrakennetulla renderöijällä. Unity Physics-paketti on ECS:stä tehty fysiikka, jossa sisältää fysiikan datakomponentteja ja systeemejä.

6.3.7 Entiteetin luonti

Entiteetin luonnissa on monta eri tapaa tehdä. Helpoin niistä on liittää ConverToEntity-komponenttia (Kuva 21) haluttuun entiteetiksi konvertoitavaan peliobjektiin. Komponentti ilmestyy komponenttilistaan, kun Entitities-paketti on asennettuna pakettihallinnassa. ConverToEntity-komponenttia liittäneet peliobjektit automaattisesti konvertoivat entiteetiksi, kun niitä luodaan peliin. Kaikkia peliobjektiin liitetyt komponentit eivät saata konvertoitua, jos niille ei ole tehty oma ECS konversiota. Unity on tehnyt suurin osa sen yleisistä komponenteista ECS:lle konvertoituvaksi.

Kuva 21. Peliobjektilla ConvertToEntity-skripti liitettynä.



ConvertToEntity-komponentissa on konversiomoodi, jonka sen asettama moodi vaikuttaa peliobjektiin entiteettikonversiossa. Konversiomoodia voidaan valita kahdesta eri moodista, jotka ovat Convert And Destroy- ja Convert And Inject-moodi. Convert And Destroy-moodissa peliobjektia tuhotaan konvertoituaan entiteetiksi. Convert and Inject Object-moodissa konvertoi peliobjektia entiteetiksi ja myös lisää sen pelikentälle.

Toinen tapa luoda entiteettiä on koodaten, jossa hyödynnetään EntityManager-objektia. Sen avulla voidaan hallita kaikkia entiteettejä. Sillä pystytään luomaan, poistamaan, lukemaan ja päivittämään entiteettejä. Oletuksena sen referenssi löytyy koodirivillä 17. koodilla (Kuva 22). Se on oletus entiteettimaailman manageri. Entiteettimaailmoja voidaan luoda lisää ja jokaisella maailmalla on oma entiteetti manageri. (Unity, n.d.-d)

Pystyäkseen käyttää Entity- ja EntityManager-objektia on lisättävä Unity.Entities-kirjasto, joka näkyy skriptin koodirivillä 2 (Kuva 22). Rivillä 17 haetaan oletuksena käytössä oleva entiteettimaailman manageria, jolla voidaan luoda entiteettejä siihen maailmaan. Rivillä 18. luodaan komponentittoman entiteetin ja riveillä 19.–23. luodaan komponentin entiteetille, jossa asetetaan datat valmiiksi ja liitetään luotuun entiteettiin. Tämä tapa eroaa hieman ConvertToEntity-komponentin lisäämistavalla. Peliobjektiin konvertoidessaan entiteetiksi ConvertToEntity-komponentilla, entiteettiin tulee mukanaan LocalToWorld-, Rotation- ja LocalToWorld-datakomponentteja. Nämä datakomponentit tulevat peliobjektin Transform-komponentin konversiosta. Jokaisella peliobjekteilla on aina Transform-komponentti liitettynä ja ei pystytä poistamaan peliobjektista. Tietyissä tilanteissa sen dataa ei tarvita ollenkaan, kuten esimerkiksi pelin managerit. Ne vaan valvovat ja hallinnoivat peliä. Pelielementit eivät tarvitse ollenkaan tietää niiden sijaintia pelin aikana, joten niiden sijainti- ja rotaatiodatat ovat turhia. Entiteetin luonti koodaten voidaan samalla välttää turhien datakomponenttien lisäämisen.

Kuva 22. Entiteetin luomisen koodi.

```
1  using UnityEngine;
2  using Unity.Entities;
3
4  namespace Testing
5  {
6
       Unity Script | 0 references
7      public class TestCreateEntityByScript : MonoBehaviour
8      {
           0 references
9          public struct TestEntityComponent : IComponentData
10         {
11             public int data1;
12             public float data2;
13         }
14
           Unity Message | 0 references
15         void Start()
16         {
17             EntityManager entityManager = World.DefaultGameObjectInjectionWorld.EntityManager;
18             Entity entity = entityManager.CreateEntity();
19             entityManager.AddComponentData(entity, new TestEntityComponent
20             {
21                 data1 = 1,
22                 data2 = 12.1f
23             });
24         }
25     }
26 }
27
```

6.3.8 ECS komponentin luonti

ECS komponentin koodipohjaa löytyy ECS-valikosta Runtime Component Type -nimenä, joka sijaitsee Create-valikossa. Oikean hiiren näppäimen klikkauksella Project-välilehden tyhjään tilaan löytääkseen Create-valikon. Pohjaan lisätään data muuttujia riippuen sen käyttötarkoituksesta. TestMovableComponent-komponentin datat (Kuva 23) on tehty systeemille, joka liikuttaa

entiteettiä. Tyhjä komponentti toimii myös ECS:ssä. Sitä yleensä käytetään entiteetin merkintänä erottaakseen entiteetin samoista entiteetti ryhmistä, kuten Tag-ominaisuus Monobehaviourissa.

Kuva 23. ECS komponentin koodiesimerkki.

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

namespace Testing
{
    [GenerateAuthoringComponent]
    [Serializable]
    0 references
    public struct TestMovableComponent : IComponentData
    {
        public float3 moveDirection ;
        public int moveSpeed;
    }
}
```

6.3.9 ECS komponentin lisäys entiteettiin

ECS:n komponentin lisäys onnistuu vetämällä sen skripti konvertoivaan peliobjektiin tai sen valitseminen komponenttilistasta, kun sen skriptiin on lisätty GenerateAuthoringComponent-attribuutin (Kuva 23). Se konvertoituu automaattisesti entiteetin komponentiksi peliobjektin entiteettikonversiossa.

Toinen tapa lisätä komponentteja entiteettiin on luoda auktorisoiva komponentti. Koodipohjamalli löytyy samasta paikasta kuin komponentin luomisessa, mutta nimellä Authoring Component Type -teksti. Authoring-komponentti on Monobehaviour-luokasta periytynyt ja sen lisäksi lisätty IConvertGameObjectToEntity-rajapinta. Rajapinnan mukana tulee Convert-funktio (Kuva 24). Se laukaisee vain kerran ja silloin, kun peliobjekti konvertoituu entiteetiksi ConvertToEntity-komponentilla. Convert-funktion sisälle voidaan koodata komponenttien lisäykset tai muutokset peliobjektin entiteetille.

Entity-parametri Convert-funktiossa on peliobjektista tuleva konvertoitunut entiteetti. Kaikki peliobjektissa olevat komponentit, jotka konvertoivat ECS komponenteiksi menevät samaan entiteettiin kuin entity-parametrissa.

Tehdäkseen ECS komponentti lisäyksiä tai muutoksia entiteettiin tarvitaan entiteetti managerin. Sen referenssin hakeminen ei toimi yleisellä tavalla World-objektista vaan siitä saa Convert-funktion dstManager-parametristä.

Kuva 24. IConvertGameObjectToEntity-rajapinnan entiteetti konversio funktio.

```
5 references
public void Convert(Entity entity, EntityManager dstManager, GameObjectConversionSystem conversionSystem)
{
```

GameObjectConversionSystem-objekti on kätevä systeemi, joka tarjoaa hyödyllisiä funktioita helpottaakseen hankalat komponentti konversiot. Se voi konvertoida peliobjektin ja sen liitetyt komponentit entiteetiksi yhdellä funktiolla käyttäen peliobjektin referenssiä. Se pitää myös tarpeellisia objektien referenssejä, jotka tarvitaan konvertoinnissa.

Authoring-komponentti pohja toimii kuin normaalit peliobjektin komponentit. Siihen voi laittaa jäsenmuuttujia tai referenssejä muista peliobjekteista. Pohjan tarkoitus on antaa kehittäjille enemmän valtaa entiteetin konversion menettelyssä. Se on hyvä sellaisissa tilanteissa, missä yritetään tehdä monta ECS komponentti lisäyksiä ja niiden datat ovat riippuvaisia toisistaan (Kuva 25).

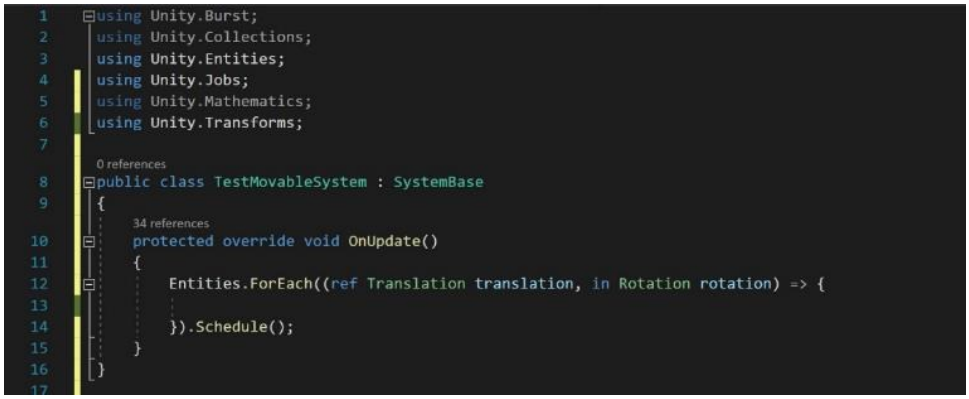
Kuva 25. Usean komponentin lisäys entiteettiin Authoring-komponentin pohjalla.

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

namespace Testing
{
    [DisallowMultipleComponent]
    [RequiresEntityConversion]
    Unity Script | 0 references
    public class TestECSComponentAuthoring : MonoBehaviour, IConvertGameObjectToEntity
    {
        // GameObject variables
        [SerializeField] int _moveSpeed;
        [SerializeField] float _jumpHeight;

        //Test ECS Components
        //-------------------------------------------------------------------------------------
        0 references
        public struct TestMovableComponent : IComponentData
        {
            public float3 moveDirection;
            public int moveSpeed;
        }

        0 references
        public struct TestJumpableComponent : IComponentData
        {
            public float jumpHeight;
        }
        //-------------------------------------------------------------------------------------

        5 references
        public void Convert(Entity entity, EntityManager dstManager, GameObjectConversionSystem conversionSystem)
        {
            dstManager.AddComponentData(entity, new TestMovableComponent {
                moveSpeed = _moveSpeed,
                moveDirection = transform.forward
            });

            dstManager.AddComponentData(entity, new TestJumpableComponent {
                jumpHeight = _jumpHeight
            });
        }
    }
}
```

6.3.10 ECS systeemin luonti

Systeemin pohjan löytyy nimellä ”System” ECS-valikosta, joka on Create-valikossa. Kooditiedosto (Kuva 26) on systeemin pohjasta luotu, jossa on poistettu kommentoitu käyttöohje. Systeemit perivät SystemBase-luokkaa ja se tuo mukanaan sen abstrakti funktion OnUpdate-funktio. Funktio toimii kuin Monobehaviour-luokan Update-funktiota, joka kutsutaan jokaisella simulaatiolla. Funktion sisällä oleva Entities.ForEach-lambda toimii kuin Jobs-systeemin koodin monisäikeistämisessä, joka on paljon yksinkertaisempi käyttää.

Kuva 26. ECS systeemin puhdas koodipohja.



Entities.ForEach-lambda käy lävitse kaikki entiteetit, jotka sisältävät samanlaiset komponentit kuin lambdan parametrissä. Entities.ForEach-lambdaan kirjoitetaan yhden entiteetin komponenttien data käsittelyn. Lambdan parametreissä ref- ja in-avainsanat määräävät komponentin datan käsittelyluvat. Ref-avainsanalla oleva komponentti voidaan lukea ja muuttaa sen dataa. In-avainsanalla oleva komponentti pystytään vain lukea sen dataa. Suositellaan laittamaan in-avainsanaa komponenteille, joita dataa luetaan pelkästään antaakseen paremman suorituskyvyn. (Unity, n.d.-e)

Entities.ForEach-lambda parametrijärjestys on tärkeä. Väärin laittaminen saadaan virheilmoituksia Unitylta. Parametri järjestys menee seuraavasti:

1. Entity entity
2. int entityInQueryIndex
3. int nativeThreadIndex
4. ref-avainsanalla komponentit
5. in-avainsanalla komponentit

Entity-parametri antaa nykyisen entiteetin. EntityInQueryIndex-niminen int-muuttuja on entiteetin data indeksi data kimpaleesta, jonka lambda käsittelee. Sitä käytetään monisäikeisen toimintojen kanssa, jotka tapahtuvat lambdan sisällä. NativeThreadIndex-niminen int-muuttuja on ainutlaatuisen säikeen indeksi, joka suorittaa nykyistä entiteettiä. (Unity, n.d.-e)

Entities.ForEach-lambda (Kuva 27) suorittaa koodinsa toisella säikeellä samanaikaisesti ohjelman pääsäikeen kanssa, kun se on asetettu Schedule-funktio koodirivissä 18. Sitä funktiota voidaan vaihtaa ScheduleParallel- tai Run-funktioniksi.

ScheduleParallel-funktiolla ajoittava lambda koodi suorittaa entiteettien datan käsittelyn kaikkiin mahdollisiin CPU:n säikeisiin ja rinnakkaina. Tällä funktiolla suoritetut lambdat eivät voi hakea systeemin ulkopuolelta dataa. Tarpeelliset datat tallennetaan paikalliseen muuttujaan, kuten skriptin rivillä 11:n (Kuva 27) delta aikaa tallennetaan paikalliseen muuttujaan dt. Silloin vasta dt-muuttujan data voidaan käyttää lambdassa, jossa näkyy koodirivillä 16.

Run-funktiolla suorittava lambda ajaa koodinsa pääsäikeessä. Lambdan sisään voidaan hakea referenssejä muista objekteista. Tätä käytetään sellaisissa tilanteissa, kun halutaan tehdä suoramuunnos data struktuuriin. Yleensä Run-funktiolla asetettu lambdaan lisätään myös WithOutBurst-funktion, kun se ei ole enää pelkkää dataa. Lambda käyttää oletuksena Burst-ominaisuutta.

Kuva 27. TestMovableSystem-systeemin koodi.

```
1  using Unity.Entities;
2  using Unity.Jobs;
3  using Unity.Transforms;
4
5  namespace Testing
6  {
       0 references
7      public class TestMovableSystem : SystemBase
8      {
           34 references
9          protected override void OnUpdate()
10         {
11             float dt = Time.DeltaTime;
12
13             Entities.ForEach((ref Translation translation, in TestMovableComponent movableComponent) =>
14             {
15
16                 translation.Value += movableComponent.moveDirection * movableComponent.moveSpeed * dt;
17
18             }).Schedule();
19         }
20     }
21 }
22
```

7 DOTS ja Monobehaviour välinen toteutus ja suorituskykyero

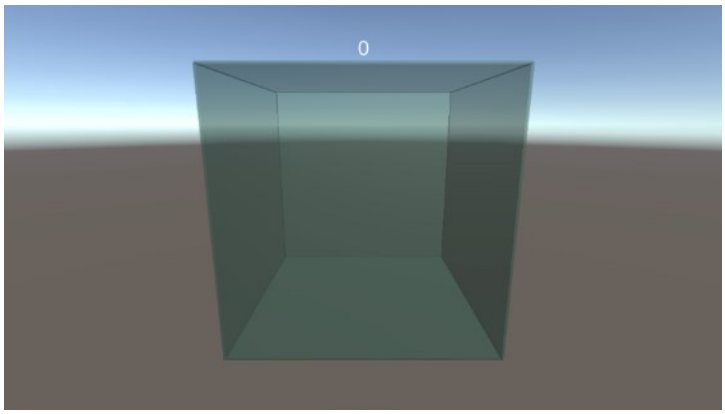
DOTS olevan vielä esikatseltavan työkaluna eli sillä ei ole ihan kaikkea valmiita käteviä ominaisuuksia kuin Monobehaviourilla. Valittiin kahden erojen vertailemiseksi fysiikan rasitus. Syynä se oli DOTSin ainoa ominaisuus, joka on kehitetty tarpeeksi pitkälle toimiakseen samalla lailla kuin Monobehaviourin fysiikka.

Vertaillakseen yleisen tavan eli Monobehaviour-objektin käyttöä ja DOTSin suorituskykyeroja ja kehitysmenetelmiä, niin molemmille kehitysympäristölle luotiin pieni raskaasti suorittava simulaatio. Tähtäyksenä on saada varsin identtiset toiminnallisuudet.

7.1 Simulaation toiminta

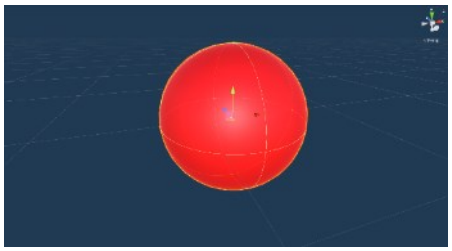
Simulaatiossa on suuri kuutio muotoinen suljettu tila (Kuva 28), johon sisään laitetaan rutkasti palloja. Palloihin lisätään fysiikan toiminnallisuudet ja törmäystunnistimia. Pallot tuottavat räjähdys fysiikkaa tietyllä intervallilla, joka puskee läheiset pallot pois. Pallojen räjähdys intervallin arvoja ovat satunnaisesti poimittu 3–10 sekunnin välistä.

Kuva 28. Kuutiomuotoinen suljettu tila.



Pallo (Kuva 29) luotiin Unity:n pallo-objektista, johon annettiin punaisen värisen materiaalin.

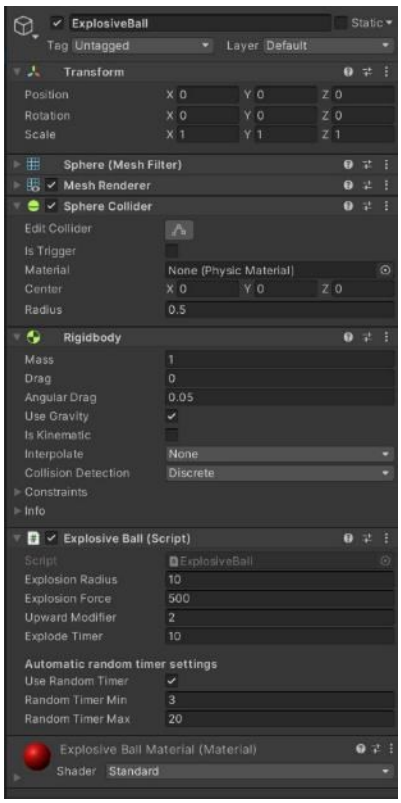
Kuva 29. Pallo-objekti.



7.2 Monobehaviour-tavan simulointi toteutus

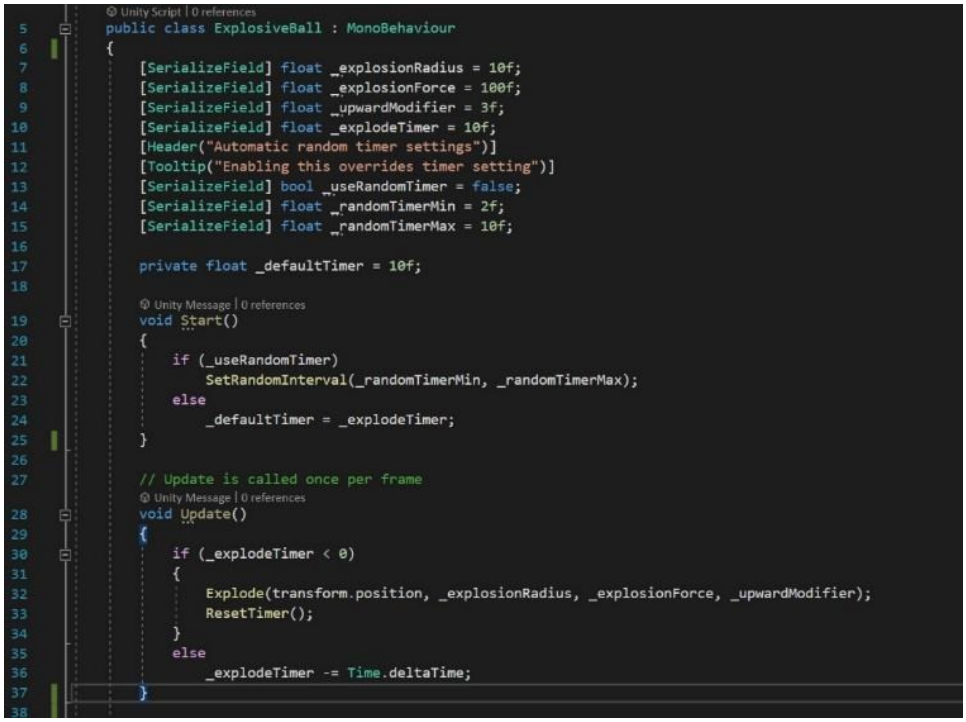
Pallolle lisättiin fysiikkaan liittyvät komponentit ja räjähdystoiminnalle luotiin ExplosiveBall-komponentin (Kuva 30). Komponentin tehtävänä on räjäyttää itsensä tietyllä intervallilla ja vaikuttaa muihin lähellä olevia palloja. Komponenttiin syötetään räjähdyksen sädettä, voimaa, ylöspäin lentokerroin ja intervalli. Siihen on lisätty satunnainen intervallin valinta ominaisuus, jotta pallot eivät räjähtäisi yhtä aikaa.

Kuva 30. Pallo-objektin komponentit.



ExplosiveBall-skriptin (Kuva 31) Update-funktio suorittaa _expodeTimer-muuttujan arvon tarkistuksen. Se toimii räjähdysajastimena. Sen arvon vähenee jokaisen ruudun päivityksellä ja sen tippuessaan alle nolla arvo, niin pallo räjäyttää itsensä Explode-funktiolla ja resetoi alkuperäiseen ajastinarvoon _defaultTimer-muuttujalla.

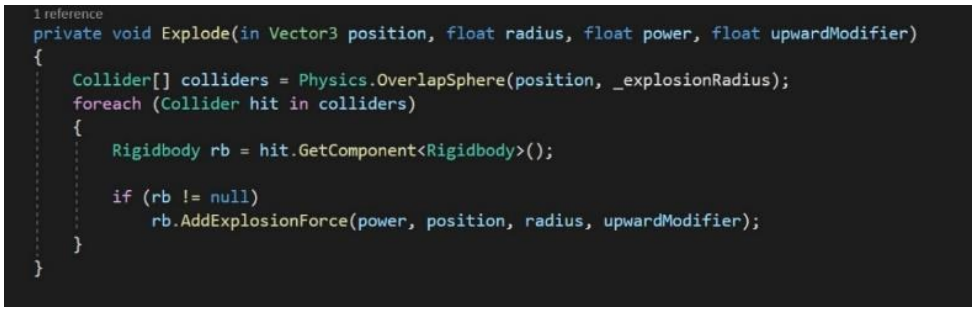
Kuva 31. ExplosiveBall-skriptin koodi.



```
Unity Script | 0 references
public class ExplosiveBall : MonoBehaviour
{
    [SerializeField] float _explosionRadius = 10f;
    [SerializeField] float _explosionForce = 100f;
    [SerializeField] float _upwardModifier = 3f;
    [SerializeField] float _explodeTimer = 10f;
    [Header("Automatic random timer settings")]
    [Tooltip("Enabling this overrides timer setting")]
    [SerializeField] bool _useRandomTimer = false;
    [SerializeField] float _randomTimerMin = 2f;
    [SerializeField] float _randomTimerMax = 10f;

    private float _defaultTimer = 10f;

    Unity Message | 0 references
    void Start()
    {
        if (_useRandomTimer)
            SetRandomInterval(_randomTimerMin, _randomTimerMax);
        else
            _defaultTimer = _explodeTimer;
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        if (_explodeTimer < 0)
        {
            Explode(transform.position, _explosionRadius, _explosionForce, _upwardModifier);
            ResetTimer();
        }
        else
            _explodeTimer -= Time.deltaTime;
    }
```

Pallon räjähdys koodi Explode-funktiossa (Kuva 32) etsitään räjähdykseen vaikuttaneet pallot Physics.OverlapSphere-funktiolla, joka palauttaa listan collider-komponentteja. Sillä listalla voidaan antaa räjähdysfysiikkaa niille palloille hakemalla Rigidbody-komponentin niiden Collider-komponentin kautta.

Kuva 32. Pallon räjähdys koodi ExplosiveBall-skriptistä.



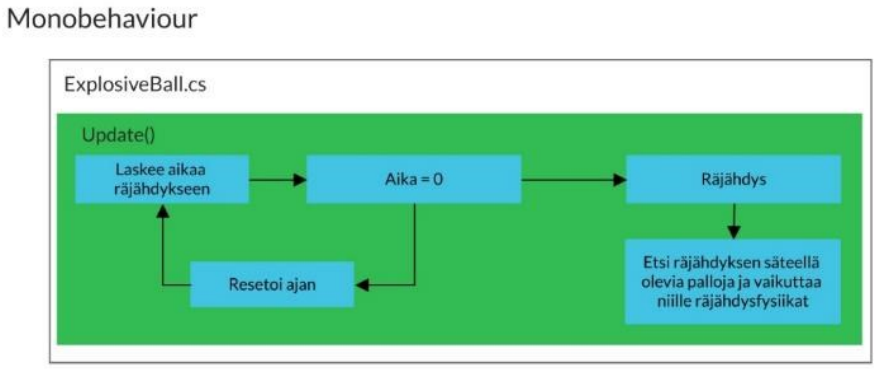
```
1 reference
private void Explode(in Vector3 position, float radius, float power, float upwardModifier)
{
    Collider[] colliders = Physics.OverlapSphere(position, _explosionRadius);
    foreach (Collider hit in colliders)
    {
        Rigidbody rb = hit.GetComponent<Rigidbody>();

        if (rb != null)
            rb.AddExplosionForce(power, position, radius, upwardModifier);
    }
}
```

7.3 DOTS-tavan simulointi toteutus

Räjähtävien pallojen DOTS version toteuttamiseen tutkittiin Monobehaviourin version simulointilogiikkaa (Kuva 33) luodakseen systeemejä. Tavoitteena on luoda sellaista systeemejä,

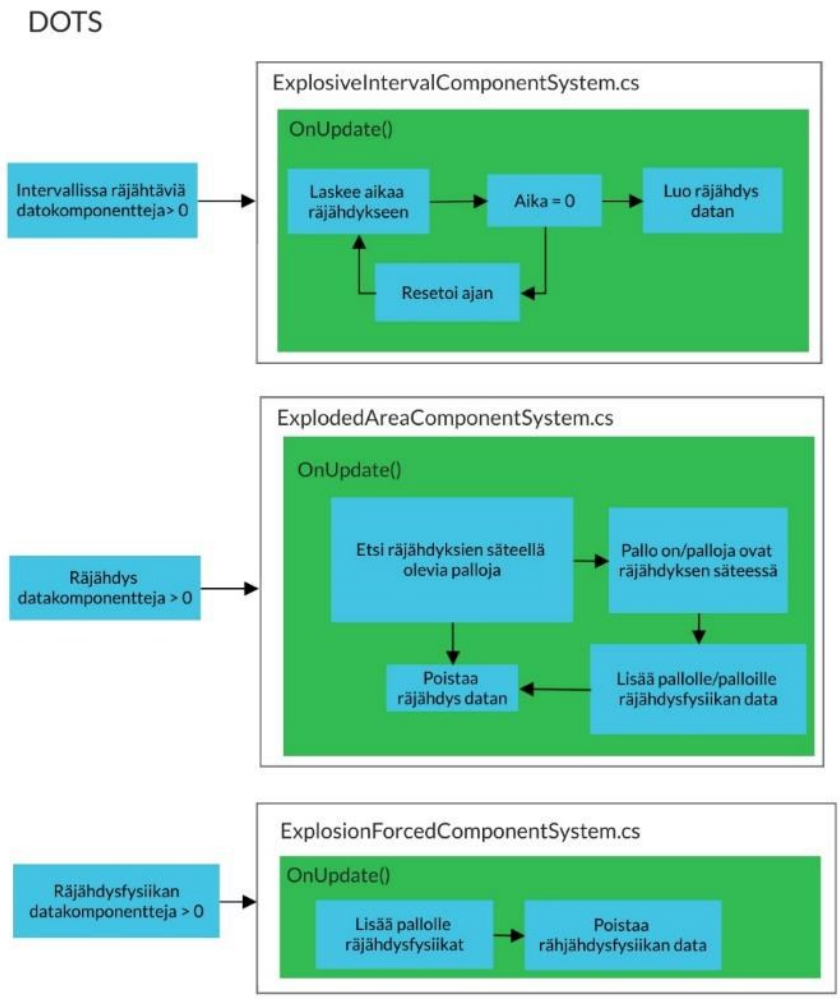
jotka hyödyntäisivät kaikkia annettuja dataa. Sillä tavalla vähennetään välimuistihuteja CPU:lle saadakseen optimaalisen suorituskyvyn.

Kuva 33. Monobeviour versio räjähtävän pallon toimintalogiikka.



DOTS version toimintalogiikka (Kuva 34) toimii samanlailla kuin Monobehaviourilla. DOTSissa räjähtävän pallon koodia pilkottiin kolmeen systeemiin. Systeemit ovat luotu hyödyntämään mahdollisimman kaikkia annettuja datakomponentteja. Räjähtävälle pallolle annettiin tilanteen mukaan vain tarvittavia datakomponentteja. Esimerkiksi räjähdyksen liittyvät datat olisivat turhia lisätä pallolle, joka ei ole vielä räjähtämäisillään. Se vaan aiheuttaisi CPU:lle turhia datatallennuksia sen pieneen välimuistiinsa, joka lisäisi välimuistihuteja. Systeemit eivät silloin myöskään kuormita CPU:ta, kun niiden käsiteltävät datakomponentit ei ole yhtäkään palloilla.

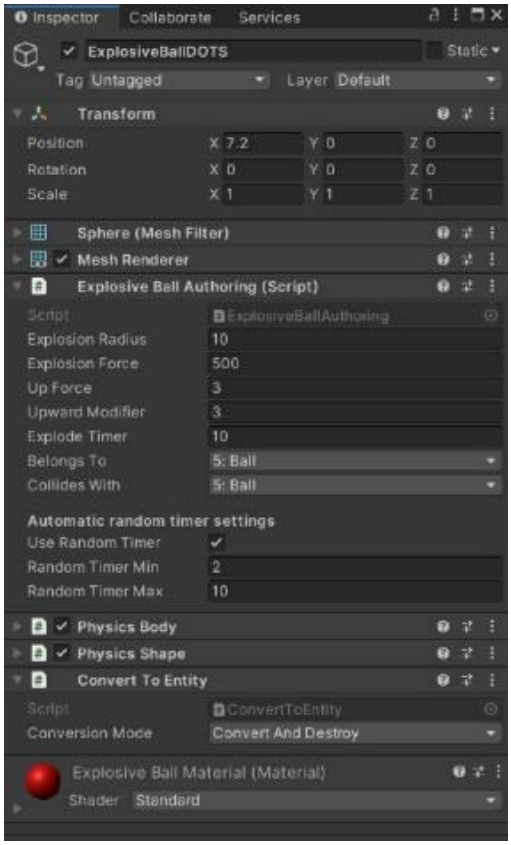
Kuva 34. DOTS versio räjähtävän pallon logiikka.



Räjähtävä pallolle ja kuutiosäiliölle jouduttiin muuttamaan niiden komponentteja, jotta ne toimisivat ECS-systeemissä. Kuutiosäiliölle lisättiin vaan ConvertToEntity-komponenttia toimiakseen ECS ympäristössä. Sillä oli käytössään pelkästään Unityn yleisiä komponentteja, joita Unity oli jo valmiiksi tehnyt ECS konversiotoimintamenetelmät. Räjähtävän pallon ECS versiolle jouduttiin luomaan uudet komponentit ja systeemit, kun sillä oli mukautettu toiminnallisuus.

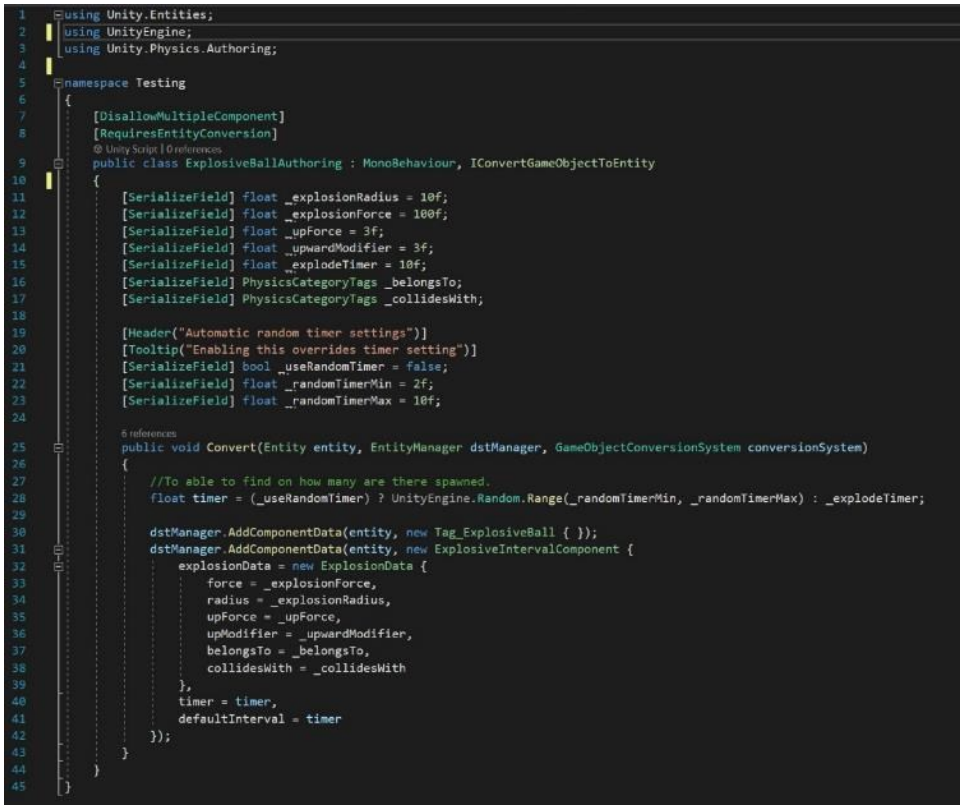
ECS räjähtävän pallon komponentti tehtiin Authoring-skriptillä pohjalla, kun tarvittiin sen Convert-funktiota. Sillä funktionalla pystyttiin tekemään monimutkaisia komponenttilisäyksiä entiteetille ja lisätä omia logiikkakoodeja ennen entiteettikonversiota. Authoring-skriptillä pystyttiin luomaan melkein identtisen näkymän Inpector-välilehdessä (Kuva 35) kuin Monobehaviour versiolla. Automatic random timer setting -ominaisuutta ei olisi pystytty lisäämään pelkällä datakomponentilla.

Kuva 35. DOTS räjähtäpallon komponentit.



Exlosive Ball Authoring -skriptin Convert-funktion (Kuva 36) avulla sisälle pystyttiin tehdä satunnaisen räjähdysintervallin arvon valinta. Siitä saadusta arvosta asetettiin ExplosiveIntervalComponent-komponenttiin ajastin liittyviin datamuuttujiin. Komponentin puuttuvat arvot löytyivät skriptin jäsenmuuttujista. Rivillä 30:ssä lisättiin Tag_ExplosiveBall-komponentti, joka on tyhjä data. Sen vastaavanlainen kuin peliobjektin tag-ominaisuus. Se auttaa suodattamaan entiteettien hakutulosta.

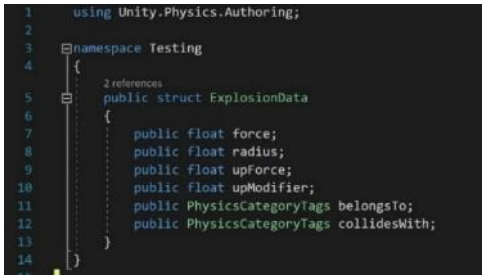
Kuva 36. Räjähtävän pallon entiteetti konversio skripti Authoring tavalla.



```
using Unity.Entities;
using UnityEngine;
using Unity.Physics.Authoring;

namespace Testing
{
    [DisallowMultipleComponent]
    [RequiresEntityConversion]
    @ Unity Script | 0 references
    public class ExplosiveBallAuthoring : MonoBehaviour, IConvertGameObjectToEntity
    {
        [SerializeField] float _explosionRadius = 10f;
        [SerializeField] float _explosionForce = 100f;
        [SerializeField] float _upForce = 3f;
        [SerializeField] float _upwardModifier = 3f;
        [SerializeField] float _explodeTimer = 10f;
        [SerializeField] PhysicsCategoryTags _belongsTo;
        [SerializeField] PhysicsCategoryTags _collidesWith;

        [Header("Automatic random timer settings")]
        [Tooltip("Enabling this overrides timer setting")]
        [SerializeField] bool _useRandomTimer = false;
        [SerializeField] float _randomTimerMin = 2f;
        [SerializeField] float _randomTimerMax = 10f;

        6 references
        public void Convert(Entity entity, EntityManager dstManager, GameObjectConversionSystem conversionSystem)
        {
            //To able to find on how many are there spawned.
            float timer = (_useRandomTimer) ? UnityEngine.Random.Range(_randomTimerMin, _randomTimerMax) : _explodeTimer;

            dstManager.AddComponentData(entity, new Tag_ExplosiveBall { });
            dstManager.AddComponentData(entity, new ExplosiveIntervalComponent {
                explosionData = new ExplosionData {
                    force = _explosionForce,
                    radius = _explosionRadius,
                    upForce = _upForce,
                    upModifier = _upwardModifier,
                    belongsTo = _belongsTo,
                    collidesWith = _collidesWith
                },
                timer = timer,
                defaultInterval = timer
            });
        }
    }
}
```

ExplosionData-datastruktuuri (Kuva 37) on räjähdysasetuksen data. Sen dataa tarvitaan Explosion pallon suorittaessaan räjähdysfysiikkaa.

Kuva 37. ExplosionData-skripti.



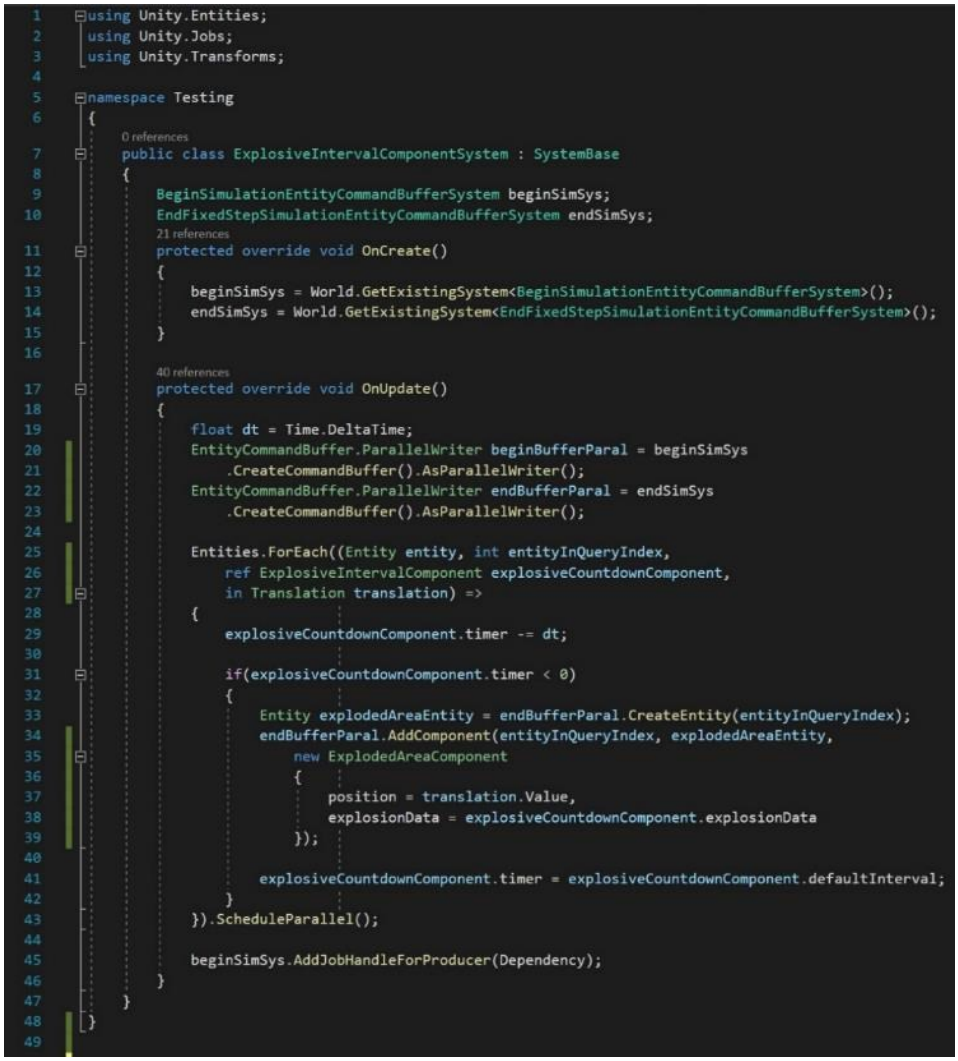
```
using Unity.Physics.Authoring;

namespace Testing
{
    2 references
    public struct ExplosionData
    {
        public float force;
        public float radius;
        public float upForce;
        public float upModifier;
        public PhysicsCategoryTags belongsTo;
        public PhysicsCategoryTags collidesWith;
    }
}
```

Pallon räjähtäminen tietyllä intervallilla luotiin ExplosionIntervalComponent-komponentin (Kuva 38). Se pitää hallussaan intervallin ja räjähdysfysiikan datoja, joita tarvitaan ExplosiveIntervalComponentSystem-systeemissa.

Kuva 38. ExplosiveIntervalComponent-skripti.

ExplosiveIntervalComponentSystem-systeemi on yksi kolmesta räjähtävän pallon systeemeistä. Sen tehtävänä on valvoa pallojen entiteettien räjähdysajastimen, ilmoittaa niiden räjähdykset ja resetoida kellon. Se lukee kaikkia entiteettiä, joissa on ExplosiveIntervalComponent- ja Translation-komponentti datastruktuuria ja käyvät ne lävitse Entities.Foreach-silmukkafunktiossa. Entiteettien ExplosiveIntervalComponent-komponentissa oleva timer-datamuuttujan arvo vähentyy jokaisella ruudunpäivityksellä. Systeemi luo entiteetin timer arvon tippuessaan alle nollan ja lisää entiteetille ExplodedAreaComponent-komponentin. Komponentti kerää räjähdyksen sijaintia ja räjähdysasetuksen dataa. Sijainnin dataa saatiin Translation-komponentin Value-datamuuttujasta ja räjähdysasetuksen data saatiin explosionCountdownComponent-muuttujasta, joka on silmukan parametrina koodirivillä 26 (Kuva 39). Systeemi resetoi komponentin ajastimen defaultInterval-datamuuttujalla, joka on myös komponentissa ja aloittaa laskennan alusta. Systeemi on todella suorituskyistä, kun asetettiin lambdaa toimimaan monisäikeisenä ScheduleParallel-funktiolla, joka näkyy koodirivillä 43.

Kuva 39. ExplosiveIntervalComponentSystem-skripti.



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

namespace Testing
{
    0 references
    public class ExplosiveIntervalComponentSystem : SystemBase
    {
        BeginSimulationEntityCommandBufferSystem beginSimSys;
        EndFixedStepSimulationEntityCommandBufferSystem endSimSys;
        21 references
        protected override void OnCreate()
        {
            beginSimSys = World.GetExistingSystem<BeginSimulationEntityCommandBufferSystem>();
            endSimSys = World.GetExistingSystem<EndFixedStepSimulationEntityCommandBufferSystem>();
        }

        40 references
        protected override void OnUpdate()
        {
            float dt = Time.DeltaTime;
            EntityCommandBuffer.ParallelWriter beginBufferParal = beginSimSys
                .CreateCommandBuffer().AsParallelWriter();
            EntityCommandBuffer.ParallelWriter endBufferParal = endSimSys
                .CreateCommandBuffer().AsParallelWriter();

            Entities.ForEach((Entity entity, int entityInQueryIndex,
                ref ExplosiveIntervalComponent explosiveCountdownComponent,
                in Translation translation) =>
            {
                explosiveCountdownComponent.timer -= dt;

                if(explosiveCountdownComponent.timer < 0)
                {
                    Entity explodedAreaEntity = endBufferParal.CreateEntity(entityInQueryIndex);
                    endBufferParal.AddComponent(entityInQueryIndex, explodedAreaEntity,
                        new ExplodedAreaComponent
                        {
                            position = translation.Value,
                            explosionData = explosiveCountdownComponent.explosionData
                        });

                    explosiveCountdownComponent.timer = explosiveCountdownComponent.defaultInterval;
                }
            }).ScheduleParallel();

            beginSimSys.AddJobHandleForProducer(Dependency);
        }
    }
}
```

Toinen systeemi on ExplodedAreaComponentSystem-systeemi (Kuva 40). Se hoitaa ExplosiveIntervalComponentSystem-systeemistä luotuja entiteettejä, joissa on ExplodedAreaComponent-komponentti liitettyinä. Sen tehtävä on selvitellä räjähdykset, räjähdyksen säteellä vaikuttaneet pallot ja asettaa räjähdyksestä tullutta fysiikkaa vaikuttaneille palloille ExplodedAreaComponent-komponenttien datan avulla.

Kuva 40. ExplodedAreaComponentSystem-systeemin OnUpdate-funktion koodi.

```
40 references
protected unsafe override void OnUpdate()
{
    CollisionWorld collisionWorld = buildPhysicsWorld.PhysicsWorld.CollisionWorld;
    EntityCommandBuffer endBuffer = endSimSys.CreateCommandBuffer();
    EntityCommandBuffer beginBuffer = beginSimSys.CreateCommandBuffer();


    NativeList<ColliderCastHit> hits = new NativeList<ColliderCastHit>(100, Allocator.TempJob);
     JobHandle handle = Entities
        .WithReadOnly(collisionWorld)
        .ForEach((Entity entity, in ExplodedAreaComponent explodedAreaComponent) =>
        {
            CollisionFilter filter = new CollisionFilter()
            {
                BelongsTo = explodedAreaComponent.explosionData.belongsTo.Value,
                CollidesWith = explodedAreaComponent.explosionData.collidesWith.Value
            };

            BlobAssetReference<Collider> collider = SphereCollider.Create(new SphereGeometry
            {
                Center = float3.zero,
                Radius = explodedAreaComponent.explosionData.radius
            }, filter);

            ColliderCastInput input = new ColliderCastInput
            {
                Collider = (Collider*)collider.GetUnsafePtr(),
                Orientation = quaternion.identity,
                Start = explodedAreaComponent.position,
                End = explodedAreaComponent.position
            };

            if (collisionWorld.CastCollider(input, ref hits))
            {
                for (int i = 0; i < hits.Length; i++)
                {
                    endBuffer.AddComponent(hits[i].Entity, new ExplosionForcedComponent
                    {
                        force = explodedAreaComponent.explosionData.force,
                        point = explodedAreaComponent.position,
                        radius = explodedAreaComponent.explosionData.radius,
                        upForce = explodedAreaComponent.explosionData.upForce,
                        upwardsModifier = explodedAreaComponent.explosionData.upModifier
                    });
                }
            }
            hits.Clear();

            beginBuffer.DestroyEntity(entity);
        }).WithDisposeOnCompletion(hits).Schedule(buildPhysicsWorld.GetOutputDependency());

    Dependency = handle;
    endSimSys.AddJobHandleForProducer(Dependency);
    beginSimSys.AddJobHandleForProducer(Dependency);
}
```

Selvittääkseen räjähdyssäteellä olevia palloentiteettejä, niin piti etsiä vastaavanlainen kuin Monobehaviourin Physics.OverSphere-funktion toiminnallisuus. Tähän ongelmaan löytyi CollisionWorld-objektin CastCollider-metodi. Se tarvitsee tietää havaitun alueen suuruuden, sijainnin ja antaa sille referoitua listaa, johon se voi tallentaa osuttujen dataa. Tällä tavalla saatiin datalistan osuneista palloentiteeteistä, johon asetettiin niille ExplosionForcedComponent-komponentin (Kuva 41) eli räjähdysfysiikan datan. Systeemin koodia suoritettiin samanaikaisesti pääsäikeen kanssa. Ei valitettavasti löydetty ratkaisua koodia toimimaan kaikkiin saatavilla oleviin säikeisiin, joka olisi huomattavasti nopeammin.

Kuva 41. ExplosionForcedComponent-komponentin datamuuttujat.

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

namespace Testing
{
    [GenerateAuthoringComponent]
    [Serializable]
    3 references
    public struct ExplosionForcedComponent : IComponentData
    {
        public float3 point;
        public float force;
        public float radius;
        public float upForce;
        public float upwardsModifier;
    }
}
```

ExplosionForcedComponent-komponentin dataa hoitaa ExplosionForceComponentSystem-systeemi, joka on räjähtävän pallon kolmas systeemi. Se etsii kaikkia palloja, jotka ovat osuneet räjäytyksessä ja antaa niille räjäytyksen fysiikkaa. Pallo entiteetille räjähdys fysiikan antaminen käytettiin koodirivillä 28. ApplyExplosionForce-metodia (Kuva 42). Metodin käyttö vaatii monta fysiikan liittyviä komponenttia toimiakseen, joka näkyy samasta skriptistä koodirivillä 24:stä–26:een. Räjäytys fysiikan annettuaan, systeemi poistaa ExplosionForcedComponent-komponetin entiteetiltä, jotta se varmasti tekee fysiikkaa vain kerran per räjähdys.

Kuva 42. ExplosionForcedComponentSystem-systeemin skripti.

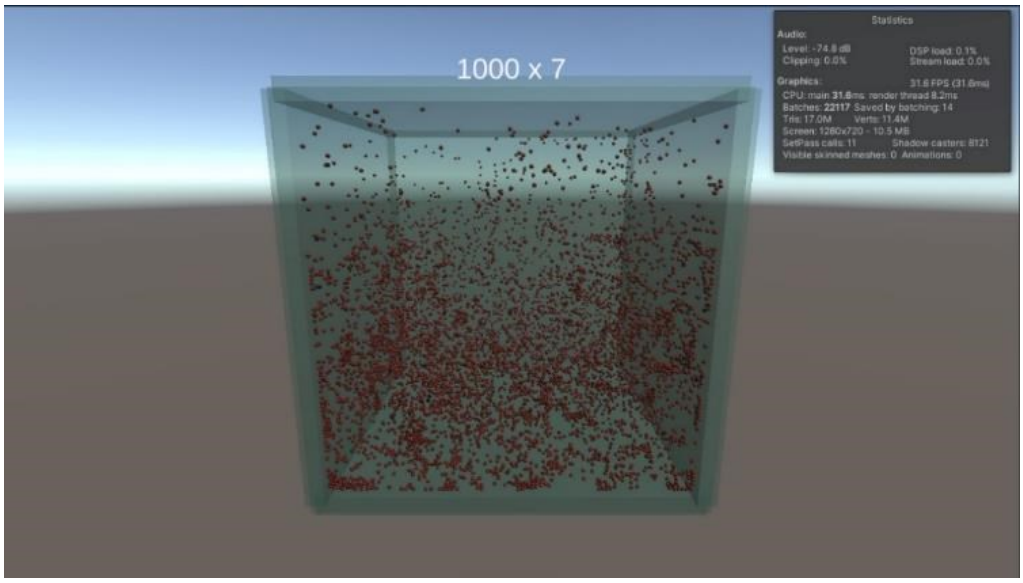
```
using Unity.Entities;
using Unity.Jobs;
using Unity.Mathematics;
using Unity.Transforms;
using Unity.Physics.Extensions;
using Unity.Physics;

namespace Testing
{
    0 references
    public class ExplosionForcedComponentSystem : SystemBase
    {
        BeginSimulationEntityCommandBufferSystem beginCommandBufferSys;
        22 references
        protected override void OnCreate()
        {
            beginCommandBufferSys = World.GetExistingSystem<BeginSimulationEntityCommandBufferSystem>();
        }
        40 references
        protected override void OnUpdate()
        {
            EntityCommandBuffer.ParallelWriter beginCommandParal = beginCommandBufferSys.CreateCommandBuffer()
                .AsParallelWriter();

            float dt = Time.DeltaTime;

            Entities.ForEach((Entity entity, int entityInQueryIndex, ref PhysicsVelocity velocity,
                in PhysicsMass mass, in PhysicsCollider collider, in Translation translation,
                in Rotation rotation, in ExplosionForcedComponent forceData) =>
                {
                    velocity.ApplyExplosionForce(in mass, in collider, in translation,
                        in rotation, forceData.force,
                        forceData.point - math.up(), forceData.radius, in dt, forceData.upForce,
                        forceData.upwardsModifier, ForceMode.Force);

                    beginCommandParal.RemoveComponent<ExplosionForcedComponent>(entityInQueryIndex, entity);
                }).ScheduleParallel();

            beginCommandBufferSys.AddJobHandleForProducer(Dependency);
        }
    }
}
```

8 Mittaustulokset

8.1 Monobehaviour version mittaustulokset

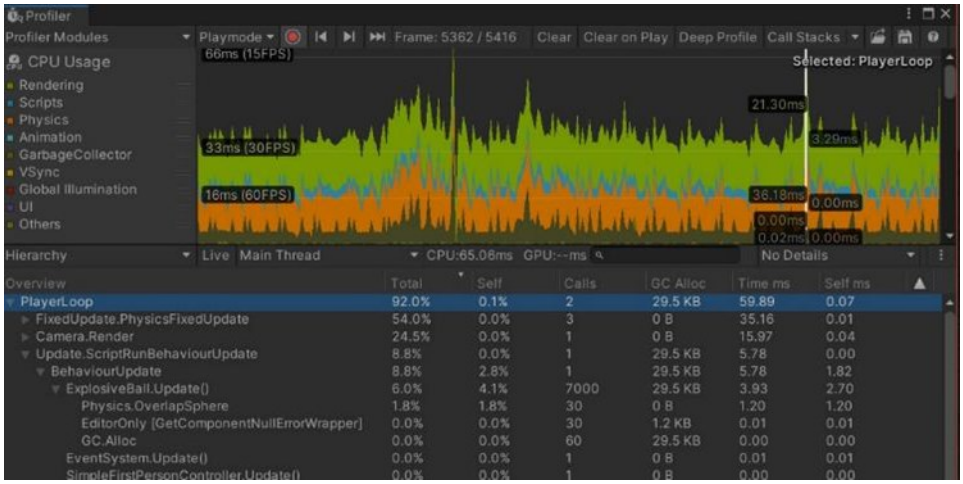
Monobehaviorin tavalla saatiin simuloitua 7000 räjähtävää palloja 29-31 FPS:än (Kuva 43).

Kuva 43. 7000 räjähtäviä palloja kuution sisällä.



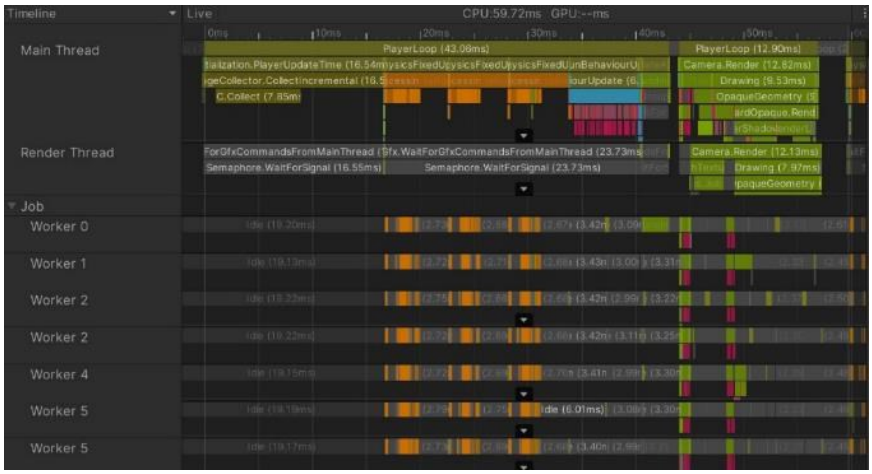
Simulaation raskain suoritus hetki oli objektien fysiikan päivittäminen, josta huomattiin profilointi mittaustuloksesta (Kuva 44). Se kuormitti CPU:ta 54% ja simulointi kesti 59,80ms. Toisena oli kuvan renderöinti, joka on 24,5%. Räjähtävien pallojen koodi kuormittivat CPU:ta 6% ja tuottivat yhteensä 29.5KB roskaa.

Kuva 44. Räjähtävien pallojen profilointi tulos raskain suoritushetkellä.



Monobehaviour oli hyödyntänyt Jobs-systeemiä fysiikan simuloinnissa, joka näkyi Profiler-ikkunassa Job-kategorian kohdalla (Kuva 45). Huomattiin suurin osa Monobehaviourin komponentit eivät ole käyttäneet Jobs-systeemiä, kun Jobsin työntekijät viettivät suurin osa ajoistaan toimettomana.

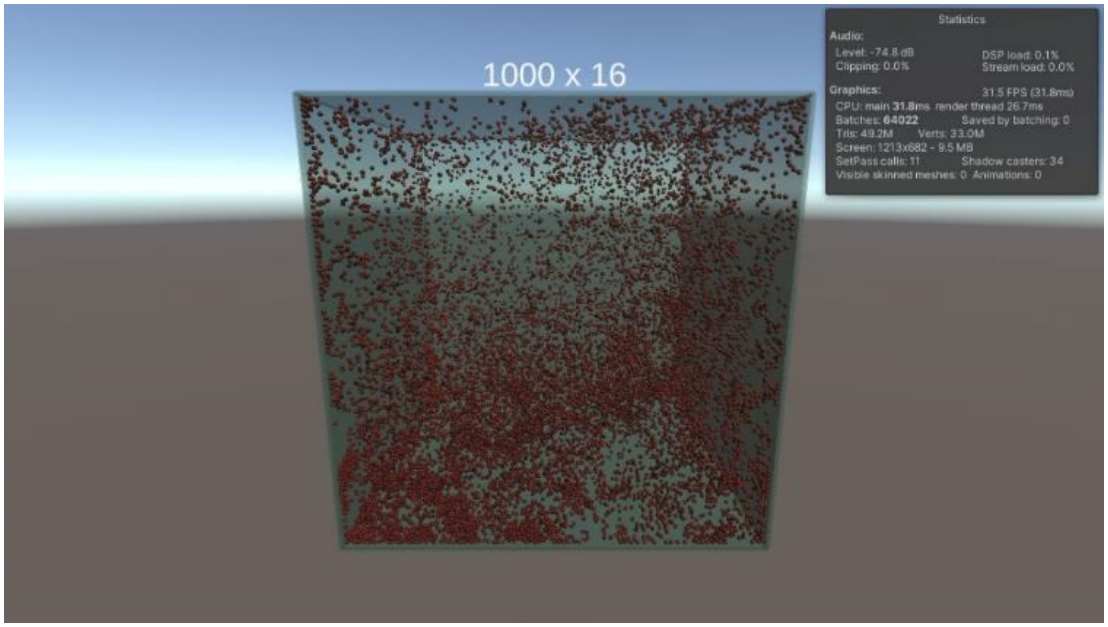
Kuva 45. Simuloinnin Jobs-systeemin tiedot.



8.2 DOTS version mittaustulokset

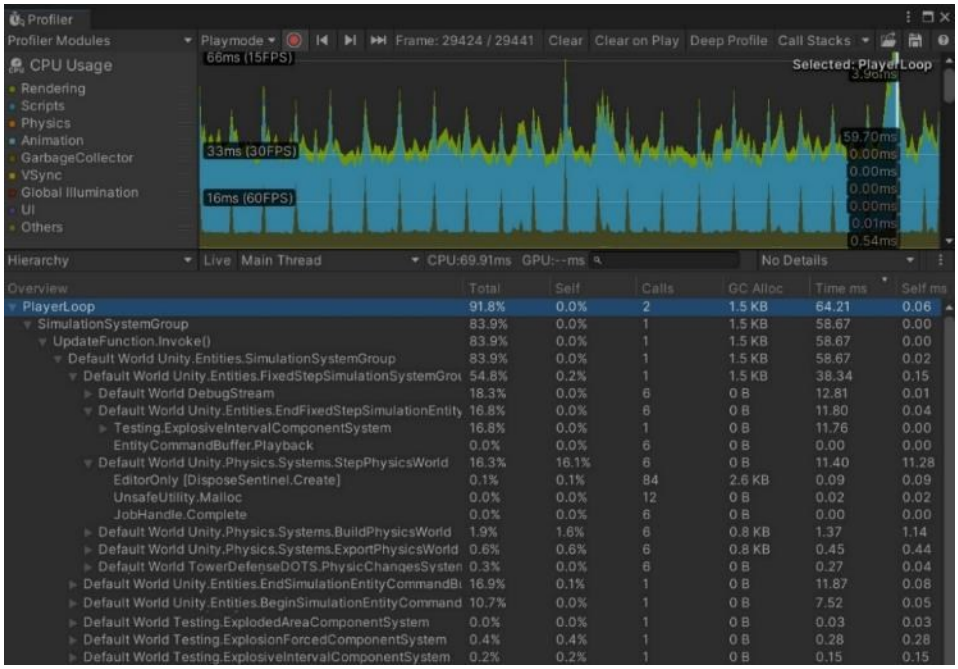
DOTSin tavalla kentälle (Kuva 46) pystyi simuloimaan 16 000 räjähtäviä palloja kentälle 27-34 FPS:än välillä.

Kuva 46.Simulointi DOTSilla 16 000 räjähtäviä palloja kuutiosäiliössä.



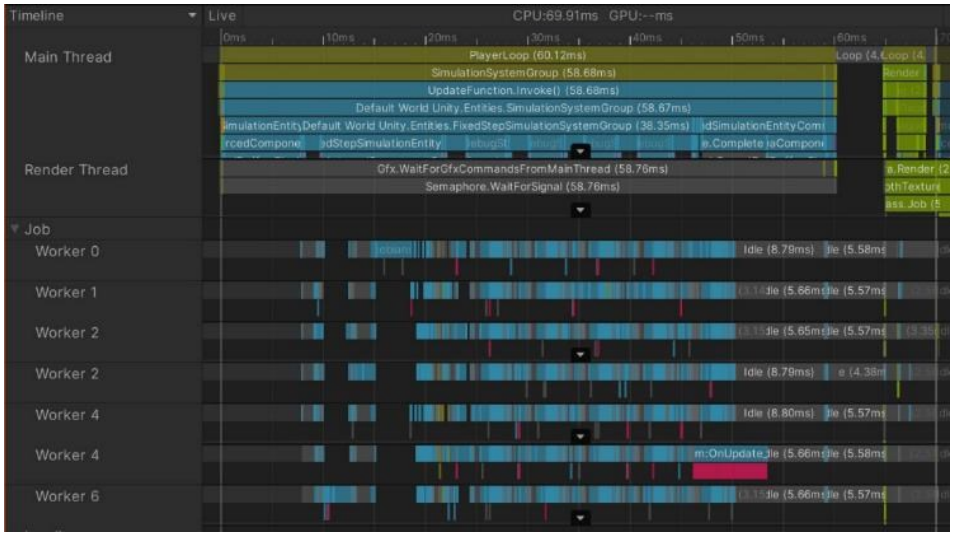
SimulationSystemGroup-suoritusryhmä Profilerissa (Kuva 47) sisältävät kaikki tutkimuksen koodit. FixedStepSimulationGroup-suoritusryhmässä on entiteettien fysiikan systeemejä, jotka kuormittavat yhteensä 83% CPU:ta, 64,21ms kestävä simulaatio ja tuotti 1,5KB roskaa. Luodut systeemit räjähtäville palloille kuormittivat yhteensä 44,9% CPU:ta ja simulointi kesto oli 31,65ms.

Kuva 47. Entiteetti pallojen raskain kuormitushetken profilointi tulos.



Jobsin ansiosta systeemit ajoivat koodinsa hyvin tiiviinä monessa säikeissä rinnakkaisena pääsäikeen kanssa (Kuva 48). ExplodedAreaComponentSystem-systeemi oli ainoa systeemi, joka ajoi koodinsa samanaikaisena pääsäikeen kanssa. Sen takia Profiler-ikkunassa nähtiin Jobsin 4. työntekijällä oli hieman pitempi työ kuin muilla (Kuva 48).

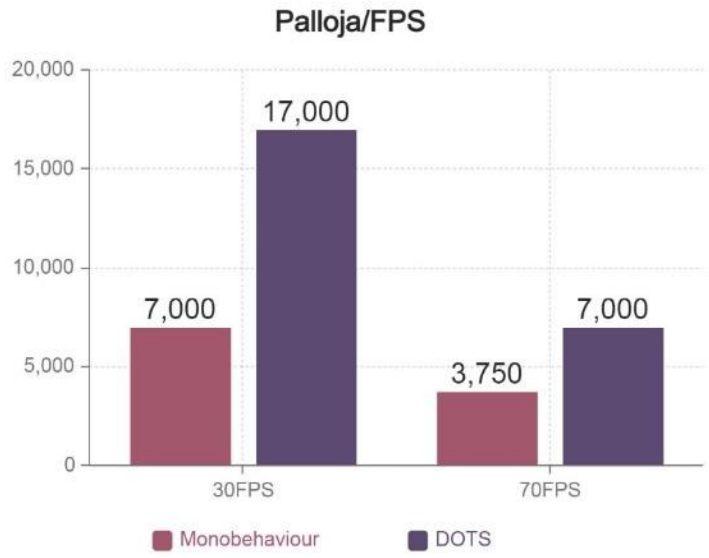
Kuva 48. Jobs-systeemin profiili 16 00 räjähtävien pallojen simuloinnissa.



9 Lopputulos

DOTSilla 30 FPS:ssä pystyi simuloimaan räjähtäviä palloja yli tupla määrän verrattuna Monobehaviour kehitystavalla (Kuva 49). Monobeaviour simuloidessaan 7000 palloja 30 FPS:ssä, niin DOTS pystyi simuloimaan saman määrän 70 FPS:ssä. DOTS antoi suorituskyvylle 86–140 %:n parannuksen.

Kuva 49. Suorituskyvyn mittaustulos Monobehaviour ja DOTS välisen kehitystavan räjähtävien pallojen simuloinnissa.



10 Johtopäätös

Pelin suorituskyvyn parantamisen ja ongelmien etsimisessä saatiin selville, kuinka välimuistihudit voivat aiheuttaa suurta suorituskykyongelmaa pelille. Unity ECS:än kehitysmenetelmän avulla annettiin pelille hurjan suorituskyvyn, kun se juuri ehkäisee Monobehaviourin pelikehitystavan välimuistihutien ongelman. Siinä ei tarvittu kirjoittaa kompleksista koodia saadakseen pelille hurjan suorituskykyparannuksen verrattuna Monobehavourillä, jossa huomattiin räjähtävien pallojen simulointi testissä. Raskaasti suorittava ominaisuutta ei olisi tarvittu myöskään korvata ECS:ssä, joka tehtiin tornipuolustuspelissä. Lisäksi ECS:ssä tuottamat koodit ovat oletuksena monisäikeiskoodi ystävällisiä, jotka mahdollistavat pelin suorituskyvyn parantamisen monikerroin.

Tässä tutkimuksessa ymmärrettiin DOTSin kyvyistä antaakseen pelille korkeata suorituskykyä. Se tulee olemaan Unityn tulevaisuus, joka hiljalleen korvaa Monobehaviouria. Se voi olla olio-ohjelmoinnin tottuneille aluksi hankala oppia ja käyttää, kun se on datakeskeistä ohjelmointia ja siinä vaaditaan uuden kehitysajattelutavan. ECS:än olevan vielä esikatseltavana ominaisuutena, joten sillä puuttuu paljon käteviä Monobehaviourin työkaluja, kuten äänilähteet, animaatiot ja AI. Saadakseen ne toimimaan DOTS ympäristössä, niin joudutaan itse tekemään niille DOTS

komponentit ja systeemit. Niiden toteuttaminen ovat hyvin hankalia, ellei täysin ymmärrä niiden työkalujen toimivuudesta ja datan käsittelyssä.

Vaikka DOTS antaa hurjan suorituskyvyn parannuksen, niin koko pelin kehittäminen pelkästään sillä ei ole vielä suositeltavaa. Syynä on juuri sen liiallinen määrä puuttuvia työkalujen takia. Niiden tekemiseen vievät paljon aikaa. DOTSin ECS ominaisuus olevan vielä esikatselutilassa, niin saatetaan törmätä hankalasti ratkaiseviin virhekoodeihin. Niiden ratkaisujakin ovat vaikeampi löytää verrattuna klassisella Monobehaviour pelikehitystavalla, koska ECS ei ole vielä yleinen pelikehitystapa Unityssa. DOTSissa voidaan törmätä sellaiseen virhekoodiin, jota on yritetty tuntikausiin ratkaista saadakseen myöhemmin selville sen virheen tulleen itse työkalusta. DOTSissa käytetyt Burst ja Jobs työkalut ovat hyviä antamaan pelille paremman suorituskyvyn, niin niitä voidaan myös käyttää ja hyödyntää klassisessa Monobehaviour-kehitysympäristössä.

Lähteet

Baumel, E. (2019). *Understanding data-oriented design for entity component systems - Unity at GDC 2019.* https://www.youtube.com/watch?v=0_Byw9UMn9g&list=PLvpHyQkRNoEbVVsfv5DYGeSUsfaH6bmo-&index=4&t=495s&ab_channel=Unity

Bitesize. (n.d.). *CPU and memory*. https://www.bbc.co.uk/bitesize/guides/zmb9mp3/revision/1

Dunstan, J. (2017). *How to Write Faster Code Than 90% of Programmers*. https://www.jacksondunstan.com/articles/3860

freeCodeCamp. (2020). *A Guide to Garbage Collection in Programming*. https://www.freecodecamp.org/news/a-guide-to-garbage-collection-in-programming/

Held, A. (2019). *Unity`s "Performance by Default" under the hood.* *https://tech.innogames.com/unitys-performance-by-default-under-the-hood/*

Linus Tech Tips. (2019). *Does High FPS make you a better gamer? Ft. Shroud - FINAL ANSWER*. https://www.youtube.com/watch?v=OX31kZbAXsA&ab_channel=LinusTechTips

techopedia. (2013). *Cache Miss*. https://www.techopedia.com/definition/6308/cache-miss#:~:text=Cache%20miss%20is%20a%20state,levels%20or%20the%20main%20memory.

Teja, K. (2018). *2. Hit, Hit Ratio and Miss Penalty - Computer Organization - Gate.* https://www.youtube.com/watch?v=cgaZYSnTc04&ab_channel=PacketPrep

Unity. (2018). *Entity Component System (ECS) "Megacity" walkthrough - Unite LA 2018 Keynote*. https://www.youtube.com/watch?v=j4rWfPyf-hk&list=PLvpHyQkRNoEZpjoqzSowaU1WxRWrvL7rD&index=31&ab_channel=Unity

Unity. (2019a). *What is multithreading?* https://docs.unity3d.com/2019.1/Documentation/Manual/JobSystemMultithreading.html

Unity. (2019b). *C# Job System Overview.* https://docs.unity3d.com/2019.1/Documentation/Manual/JobSystemOverview.html

Unity. (2020a). *Profiler overview.* https://docs.unity3d.com/2020.1/Documentation/Manual/Profiler.html

Unity. (2020b). *The Profiler window.* https://docs.unity3d.com/2020.2/Documentation/Manual/ProfilerWindow.html

Unity. (n.d.-b). *Performance by default.* https://unity.com/dots

Unity. (n.d.-c). *Burst User Guide.* https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html

Unity. (n.d.-d). *Class EntityManager.* https://docs.unity3d.com/Packages/com.unity.entities@0.0/api/Unity.Entities.EntityManager.html

Unity. (n.d.-e). *Using Entities.ForEach.* https://docs.unity3d.com/Packages/com.unity.entities@0.9/manual/ecs_entities_foreach.html

Wikipedia. (2020a). *Garbage collection (computer science).* https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)#:~:text=In%20computer%20science%2C%20garbage%20collection,in%20use%20by%20the%20program

Wikipedia. (2020b). *Profiling (computer programming).* https://en.wikipedia.org/wiki/Profiling_(computer_programming)#:~:text=In%20software%20engineering%2C%20profiling%20(%22,and%20duration%20of%20function%20calls.