Eelis Nampajärvi

Rinnakkaislaskennan työkalut peli-moottorissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

3.5.2022

Tiivistelmä

Tekijä:	Eelis Nampajärvi
Otsikko:	Rinnakkaislaskennan työkalut pelimoottorissa
Sivumäärä:	58 sivua
Aika:	3.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Antti Laiho Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli selvittää Unity-pelimoottorin rinnakkaislaskentaa varten tarkoitettujen työkalujen soveltuvuus pelien monisäikeistämiseen ja muiden prosessoriominaisuuksien, kuten vektoriyksiköiden, hyödyntämiseen pelikehityksessä. Työn oli tarkoitus toimia käytännön esimerkkinä työkalujen käytöstä Unityllä kehitettävissä sovelluksissa ja niiden tarjoamista suorituskykyeduista.

Insinöörityössä kehitettiin testisovellus, joka optimoitiin käyttämällä Unityn tarjoamia rinnakkaislaskennan työkaluja. Testisovellus monisäikeistettiin käyttämällä Unityn Job Systemiä, jolloin sovellus pystyi hyödyntämään kaikkia käytössä olevia prosessoriytimiä. Lisäksi hyödynnettiin Unityn Burst-kääntäjää, joka onnistui vektorisoimaan osan testisovelluksesta ja tarjosi myös muita optimointeja.

Työssä selvisi, että työkalujen käyttö oli testisovelluksen tapauksessa helppoa ja tehokasta, ilman että aikaisempaa kokemusta pelien rinnakkaistamisesta juurikaan oli. Testisovelluksella tehtiin suorituskykymittauksia, joista selvisivät työkalujen tarjoamat, usein hyvinkin suuret suorituskykyedut. Lisäksi huomattiin Unityn kahden vakiona tarjoaman käännöstavan, Monon ja IL2CPP:n, joissain tapauksissa suuretkin suorituskykyerot.

Työn tulokset osoittivat, että käytetyt työkalut tarjoavat pelikehittäjille tehokkaan tavan optimoida ainakin joitakin osia peleistä. Työn tulosten yleiskelpoisuutta rajoitti kehitetyn testisovelluksen suppeus, ja työkalujen laajempi testaus käytännön peleissä todettiin tarpeelliseksi.

Avainsanat: rinnakkaislaskenta, Unity, pelikehitys, optimointi, monisäikeistys, vektorisointi, SIMD, Job System, Burst

Abstract

Author:	Eelis Nampajärvi
Title:	Parallel computing tools in a game engine
Number of Pages:	58 pages
Date:	3 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Supervisors:	Antti Laiho, Senior Lecturer Miikka Mäki-Uuro, Senior Lecturer

---

The purpose of the Bachelor's thesis was to investigate the suitability of parallel computing tools offered by Unity game engine, for multithreading and vectorization of game applications. The thesis was meant to act as a practical example on how to use these tools and what performance benefits they can bring to games.

A test application was developed and then optimized with parallel computing tools offered by Unity. The test application was multithreaded using Unity Job System which enabled the application to utilize all available cores. Additionally, Unity Burst compiler was used, which enabled vectorization and other optimizations.

It was found out that the usage of these tools was easy and efficient, even without much prior experience in writing multithreaded game code. Benchmarks were performed on the developed test application, and they showed often very large performance gains offered by the used tools. In addition, it was noticed that the performance difference between Mono and IL2CPP, the two scripting backends offered by default by Unity, can be significant. The thesis's results showed that the tools used, offer an efficient way for game developers to optimize at least some parts of developed games. General applicability of the thesis's results were thought to be limited, caused by the test applications narrow focus and further testing performed on actual games was recommended.

Keywords: parallel computing, Unity, game development, optimization, multithreading, vectorization, SIMD, Job System, Burst.

Sisällys

Lyhenteet ja käsitteet

.NET: Microsoftin kehittämä ilmainen, avoimen lähdekoodin ohjelmistokehys.

AVX: *Advanced Vector Extensions*. SIMD-käskykantalaajennus x86-arkkitehtuurin prosessoreille. Tuo prosessorille käyttöön muun muassa 256-bittiset rekisterit.

Burst: Unity-pelimoottorin Job Systemiä ja ECS:ää varten kehitetty kääntäjä.

DOTS: *Data oriented technology stack*. Unityn tarjoama kokoelma työkaluja suorituskykyisten pelien kehittämistä varten.

ECS: *Entity component system*. Unity-pelimoottorin dataorientoituneen pelikoodin kehittämiseen tarkoitettu järjestelmä.

HPC#: *High Performance C#*. Unityn kehittämä tehokkuutta tavoitteleva C#-kielen osajoukko. Käytetään Unityn Job Systemin ja Burst-kääntäjän kanssa.

IL2CPP: *Intermediate Language To C++*. Unity-pelimoottorin tarjoama vaihtoehtoinen käännöstapa, joka kääntää esimerkiksi C#-skripteistä tuotetun välikielen C++-koodiksi, joka käännetään lopulta kohdearkkitehtuurin konekieliseksi koodiksi. Tarjoaa ennenaikaisen käännöstavan Unityssä.

LLVM: *Low Level Virtual Machine*. Kokoelma ohjelmointikielen kääntäjätyökaluja.

Job System: Unity-pelimoottorin pelikoodin monisäikeistämistä helpottava työkalu.

Mono: Avoimen lähdekoodin kehitysympäristö. Unity käyttää Monon tarjoamaa skriptialustaa sen vakiokäännöstapana. Tarjoaa ajonaikaisen käännöksen Unityssä.

Prefab: Uudelleenkäytettävä versio peliobjektista Unity Editorissa.

SIMD: *Single instruction multiple data*. Vektoriprosessorien toimintatapa, jossa operoidaan yhdellä käskyllä useaa data-alkiota samanaikaisesti.

SISD: *Single instruction single data*. Prosessorin toimintatapa, jossa operoidaan yhdellä käskyllä yhtä data-alkiota.

SMT: *Simultaneous multithreading.* Symmetrinen monisäikeisyys, laitteistotuki useamman säikeen samanaikaiselle suoritukselle samalla prosessoriytimellä.

SSE: *Streaming SIMD Extensions*. SIMD-käskykantalaajennus x86-arkkitehtuurin prosessoreille liukulukulaskujen nopeuttamista varten. Tuo prosessorille käyttöön 128-bittiset rekisterit. SSE2 toi myöhemmin tuen myös muille muuttujatyypeille.

1 Johdanto

Pelialalla ja ohjelmistoalalla yleisesti vallitsi pitkään olettamus, että seuraavan sukupolven laitteisto toisi aina suuria suorituskykyparannuksia ja näin edellisvuoden pelit toimisivat aina huomattavasti paremmin uuden sukupolven prosessoreilla, ilman että peliä tarvitsi mitenkään muuttaa tai optimoida uuden sukupolven laitteita varten. Vaikka uusien näytönohjainten tarjoama suorituskyky onkin jatkanut suhteellisen tasaista kasvuaan, grafiikan renderöinnin rinnakkaisen luonteen takia sama kasvu ei ole jatkunut tavallisen pelikoodin suorituksen kohdalla, kun prosessorien yhden säikeen suorituskyvyn kasvu hidastui 2000-luvun alun jälkeen. Tämä muutos käänsi prosessorikehityksen moniydinprosessoreihin ja rinnakkaisuuden eri tasojen tehokkaampaan hyödyntämiseen.

Vaikka moniydinprosessorit ovat teoriassa tuoneet suuren määrän lisää laskentatehoa, ei tätä kaikkea ole voitu hyödyntää käytännössä tehokkaasti, koska useiden ohjelmien, kuten pelien, muuttaminen useita prosessoriytimiä hyödyntäväksi on hankalaa eikä aina edes mahdollista. Useiden ytimien lisäksi prosessoreihin oli tätä ennen alettu lisätä pääasiassa liukulukulaskuja nopeuttavia vektoriyksiköitä, jotka kasvattivat prosessorin laskentatehoa kellosykliä kohden, kun tehtiin suuria määriä peräkkäisiä laskutoimituksia. Usein näiden yksiköiden käyttö vaatii kuitenkin ohjelmoijalta tarkkaa alustakohtaista tietämystä, ja se jää sen takia monissa tapauksissa täysin hyödyntämättä, vaikka ne voivat tarjota suuriakin suorituskykyparannuksia useissa ohjelmissa.

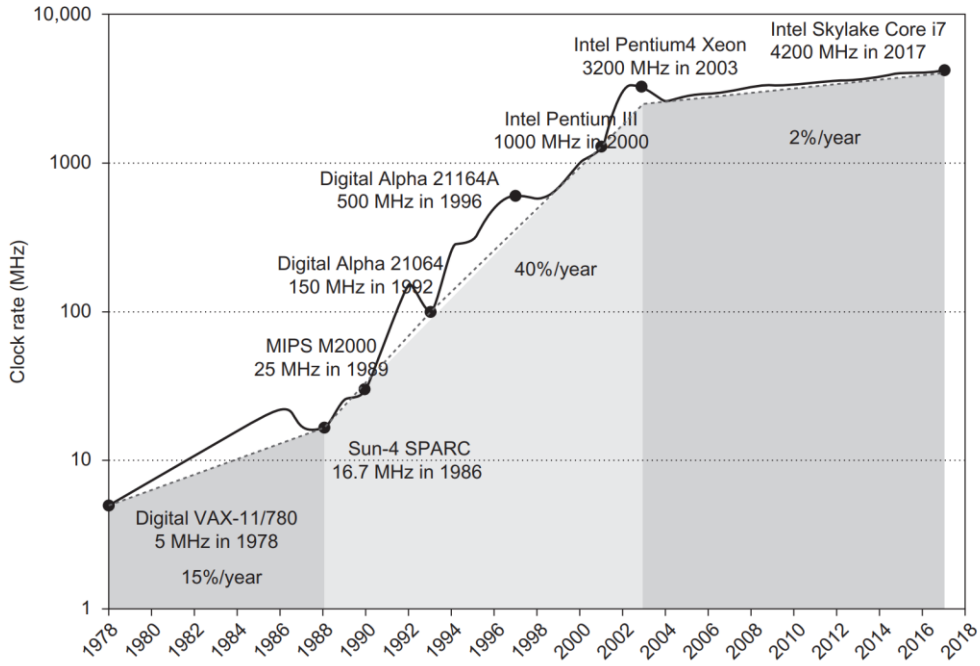
Yhä laajempien ja yksityiskohtaisempien pelien kehittämiseksi on tärkeää, että pelikehittäjät saavat laitteistosta kaiken mahdollisen hyödyn suorituskyvyn parantamiseksi. Tämän takia on tärkeää, että pelien monisäikeistäminen useiden prosessoriydinten hyödyntämiseksi ja muiden prosessoriominaisuuksien, kuten vektoriyksiköiden, käyttöönotto on mahdollisimman vaivatonta ja tehokasta.

Insinöörityön tarkoituksena oli selvittää, kuinka hyvin Unity-pelimoottorin tarjoamat työkalut soveltuvat pelien monisäikeistämiseen ja muiden prosessoriominaisuuksien, kuten vektoriyksiköiden, hyödyntämiseen. Insinöörityössä tutkittiin

Unityn monisäikeistystä helpottavan Job Systemin ja sitä tukevan Burst-kääntäjän toimintaa ja käyttöönottoa testisovelluksessa. Lisäksi testisovelluksella suoritettiin suorituskykymittauksia, joilla selvitettiin Job Systemin monisäikeistämisestä sekä Burst-kääntäjän vektorisoinnista ja muista optimoinneista saatua käytännön hyötyä.

2 Rinnakkaislaskenta

Yksi tärkeimpiä prosessorien suorituskykyä kasvattavia tekijöitä prosessorien alkuajoista lähtien, Mooren lain lisäksi, oli Dennard-skaalaus (engl. Dennard scaling). Dennard-skaalaus oli Robert Dennardin vuonna 1974 tekemä huomio ilmiöstä, jossa määrätynkokoisen prosessorin vaatima teho pysyi vakiona, vaikka transistorien määrää saatiin kasvatettua yhä pienevien transistorien ansiosta. Tämä skaalaus salli prosessorien transistorien toimimisen yhä korkeammilla kellotaajuuksilla ja käyttäen samalla vähemmän virtaa. Dennard-skaalaus oli suuressa osassa prosessorien nopeuden kehityksessä vuosikymmenten ajan, mutta tuli vihdoin päätökseen noin vuonna 2004. (1, s. ix, 4, 5.) Kuvassa 1 näkyy, miten prosessorien kellotaajuuksien kasvaminen hidastui merkittävästi vuoden 2003 jälkeen.



Kuva 1. Prosessorien kellotaajuus vuosina 1978–2018 (1, s. 26).

Dennard-skaalauksen loppuminen käänsi prosessorikehityksen suunnan moniydinprosessoreihin ja eri tasojen rinnakkaisuuksien hyödyntämiseen. Ennen tätä muutosta lähinnä vain käsky- ja bittitason rinnakkaisuuksia oli hyödynnetty prosessoreissa, mutta nyt suunta kääntyi hyödyntämään muita rinnakkaisuuden tasoja, kuten data-, tehtävä- ja säietason rinnakkaisuuksia. Rinnakkaisuus kaikilla tasoilla on tämän jälkeen ollut tärkein tietokoneiden laskentatehoa kasvattava tekijä ja määrää laajalti kehitettävien tietokoneiden arkkitehtuurien kehityssuuntaa. (1, s. 4, 5, 10; 2, s. 15.)

2.1 Rinnakkaisuuden tasot

Tietojenkäsittelytieteessä rinnakkaisuus jaetaan usein neljään eri luokkaan, bitti-, data-, käsky- (engl. instruction-level parallelism) ja tehtävätason (engl. task-level parallelism) rinnakkaisuuksiin. Joskus tähän luokitteluun sisällytetään säietason rinnakkaisuus (engl. thread-level parallelism), jota usein käytetään tehtävätason rinnakkaisuuden kanssa, ja lisäksi harvemmin pyyntötason rinnakkaisuus (engl. request-level parallelism). (1, s. 10; 3.)

Bittitason rinnakkaisuudella tarkoitetaan prosessorin käsittelemän datan pituuden kasvattamista, jotta voidaan operoida suurempikokoista dataa yhdellä operaatiolla usean operaation sijasta. Mikroprosessorin kehityksen alkuaikoina parannukset prosessoriarkkitehtuuriin keskittyivät suurissa määrin bittitason rinnakkaisuuden kasvattamiseen. Prosessorien rekisterit ja dataväylät kasvoivat 1970-luvulta lähtien 4 bitistä 8 bittiin, 8 bitistä 16 bittiin, kunnes kasvu pysähtyi pidemmäksi aikaa 32 bittiin 1980-luvun puolessa välissä, josta siirryttiin pidemmän tauon jälkeen viimein 64 bittiin, jossa suurin osa prosessoreista on nykypäivänä. (2, s. 15–16.)

Datatason rinnakkaisuudella tarkoitetaan datan hyödyntämistä siten, että voidaan käsitellä suurempi määrä dataa samanaikaisesti. Datatason rinnakkaisuutta hyödynnetään esimerkiksi SIMD- eli single instruction multiple data -prosessoinnissa, jossa operoidaan yhdellä käskyllä useaa dataelementtiä laskentatehon kasvattamiseksi. (1, s. 10–11.)

Käskytason rinnakkaisuudella tarkoitetaan sitä, miten prosessorilla voi olla suorituksessa useampi käsky samanaikaisesti. Tämä on mahdollista prosessorien liukuhihnarakenteen ansiosta, jossa yhden käskyn suoritus on jaettu useampaan vaiheeseen. Käskyjen liukuhihna mahdollistaa, että useiden käskyjen eri vaiheita voidaan suorittaa samanaikaisesti. Ideaalisessa tapauksessa liukuhihna mahdollistaa sen vaiheiden määrän mukaisen suorituskykyparannuksen liukuhihnattomaan prosessoriin verrattuna. (1, s. C-2, C-3; 3.) Prosessorien liukuhihnat on usein jaettu viiteen eri vaiheeseen, käskyn hakuun, käskyn purkuun, suoritukseen, muistihakuun ja takaisinkirjoitukseen (1, s. C-6). Kuvassa 2 nähdään, miten viisivaiheisessa käskykannan liukuhihnassa voi olla viisi eri vaiheissa olevaa käskyä samanaikaisessa suorituksessa.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				IF	ID	EX	MEM	WB	
Instruction *i*+4					IF	ID	EX	MEM	WB

Kuva 2. Viisivaiheinen käskykannan liukuhihna, jossa on viisi käskyä samanaikaisessa suorituksessa eri liukuhihnan vaiheissa (1, s. C-6).

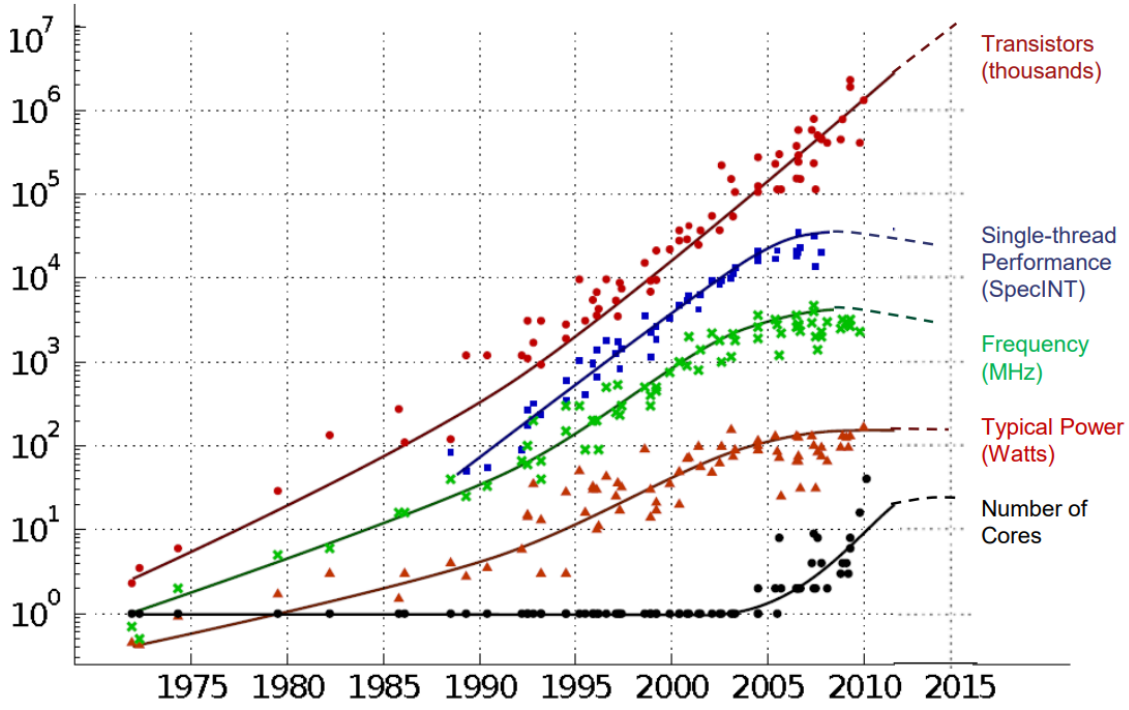
Useat modernit prosessorit yrittävät hyödyntää käskytason rinnakkaisuutta vielä tehokkaammin aloittamalla useampia käskyjä samalla kellosyklillä. Tällaisia prosessoreita kutsutaan superskalaarisiksi prosessoreiksi. Niissä on useita laskentaan tarkoitettuja yksiköitä useiden samanaikaisten käskyjen vaiheiden rinnakkaista suorittamista varten. Näitä yksiköitä ovat esimerkiksi aritmeettisloogiset yksiköt ja liukulukuyksiköt. Kääntäjät yrittävät parhaansa mukaan asettaa konekieliset käskyt järjestykseen, jossa jokainen prosessorin laskentayksikkö on koko ajan käytössä maksimaalisen suoritustehon saavuttamiseksi. (4.)

Tehtävä- tai säietason rinnakkaisuudella tarkoitetaan sitä, miten yksi isompi tehtävä voidaan jakaa useampaan pieneen tehtävään ja suorittaa nämä yksittäiset tehtävät rinnakkain ja näin hyödyntää useaa suoritusyksikköä samanaikaisesti (3).

Ohjelmien ja ohjelmoijan kannalta hyväksikäytettävät rinnakkaisuuden tasot ovat lähinnä data- ja tehtävä- tai säietason rinnakkaisuudet. Ohjelmoija voi laatia sovelluksensa siten, että sen käsittelemä data on muodossa, joka on prosessorille suotuisa datatason rinnakkaisuuden kannalta. Samalla voidaan yrittää eksplisiittisesti vektorisoida koodia, jolloin saadaan paremmin hyödynnettyä prosessorien SIMD-yksiköitä. Tehtävä- tai säietason rinnakkaisuutta ohjelmoija voi hyödyntää monisäikestämällä jonkin osan ohjelmasta, jolloin monisäikeistetty osa voidaan ajaa rinnakkain usealla prosessoriytimellä.

2.2 Moniydinprosessorit ja monisäikeisyys

Vaikka Dennard-skaalaus loppuikin, Mooren laki jatkui vielä tästä eteenpäin, vaikka onkin hidastunut viime vuosina. Mooren laki on Gordon Mooren tekemä havainto mikroprosessorien transistorimäärien kaksinkertaistumisesta noin kahden vuoden välein. Tämä havainto on jo noin 50 vuoden ajan ollut Dennard-skaalauksen ohella tärkein prosessorien laskentatehoa kasvattava tekijä, kun kasvavia transistorimääriä voidaan käyttää lisäämään prosessorien kykyjä. Dennard-skaalauksen loppuessa prosessorikehitys kääntyi moniydinprosessoreiden kehittämiseen, jolloin Mooren lain siivittämät kasvavat transistoribudjetit käytettiin useiden prosessoriydinten lisäämiseen. (1, s. 5.) Kuvasta 3 nähdään, miten mikroprosessorien prosessoriydinten määrä pysyi yhdessä noin vuoteen 2005 asti ja on kasvanut siitä lähtien, kun taas kellotaajuuden kasvu on käytännössä pysähtynyt.



Kuva 3. Mikroprosessorien historialliset kehitystrendit vuosina 1975–2015 (5).

Moniydinprosessorissa on yhden suoritusyksikön sijaan samalla piirillä useampi suoritusyksikkö, joita kutsutaan ytimiksi. Jokainen ydin pystyy itsenäisesti

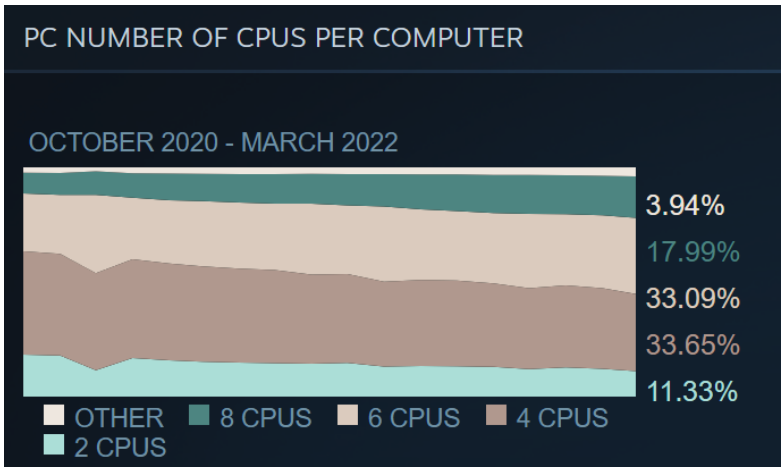
suorittamaan ohjelmien käskyjä, koska jokaisella ytimellä on omat yksikkönsä käskyjen suorittamiseen. (6.) Jokaisella ytimellä on yleensä myös omat L1- ja usein L2-tason välimuistit, joista dataa on nopea hakea keskusmuistin sijasta. Useilla moniydinprosessoreilla on vielä suuri L3-tason välimuisti, joka on jaettu kaikkien prosessoriydinten välillä. (1, s. 372.)

Koska jokainen moniydinprosessorin ydin voi suorittaa omaa säiettään, se voi hyödyntää monisäikeistettyä koodia, jossa ohjelman suoritus on jaettu useaan osaan säikeitä käyttäen (7). Näin saavutetaan säietason rinnakkaisuus. Prosessorin laskentayksiköiden maksimaaliseksi hyödyntämiseksi useat modernit superskalaariset prosessorit tukevat kahden säikeen samanaikaista suorittamista samalla prosessoriytimellä. Tällaista laitteistotukea monisäikeisyydelle kutsutaan symmetriseksi monisäikeistykseksi, josta käytetään englanninkielistä termiä SMT eli simultaneus multithreading. SMT:tä tukevat prosessorit voivat lukea käskyjä usealta säikeeltä samanaikaisesti, ja näin voidaan hyödyntää superskalaarisen prosessoriytimen resursseja tehokkaammin. (8.)

Nykypäivänä, kun lähes kaikki prosessorit ovat moniytimellisiä, on sovelluskehittämisessä tärkeää yrittää monisäikeistää ohjelma, koska monisäikeistetyn ohjelman suorittaminen moniydinprosessoreilla voi olla useita kertoja sarjallista suoritusta nopeampaa. Ideaalisessa tapauksessa suorituskyky voi kasvaa lineaarisesti käytettävien ytimien määrän mukaan (1, s. 440).

Monisäikeistetyn koodin kirjoittaminen on kuitenkin usein huomattavasti hankalampaa sarjalliseen koodiin verrattuna. Myöskään kaikkia ohjelmia tai osia ohjelmista ei voida rinnakkaistaa, sillä useissa ohjelmissa jotkin tehtävien vaiheet vaativat tiedon tehtävän edellisestä vaiheesta, eli on syntynyt riippuvuus. Ongelmana monisäikeistetyssä koodissa ovat myös kilpailutilanteet. Kilpailutilanne syntyy, kun eri säikeet tarvitsevat jotain samaa resurssia. Jos toinen säie kirjoittaa samaan muuttujaan, samalla kun toinen säie lukee sitä, voi syntyä virheitä. Tällaisten tilanteiden välttämiseksi vaaditaan oikeanlaista synkronointia säikeiden välillä, joka aiheuttaa viivettä ja vähentää rinnakkaisuudesta saatavaa hyötyä. (9.)

Maaliskuun 2022 Steam Hardware Surveyn (10) mukaan suurimmalla osalla Steamin kyselyyn osallistuneista käyttäjistä on käytössä 4–8-ytiminen moniydinprosessori ja korkeaytimisten prosessorien osa sen käyttäjillä on kasvussa (kuva 4).



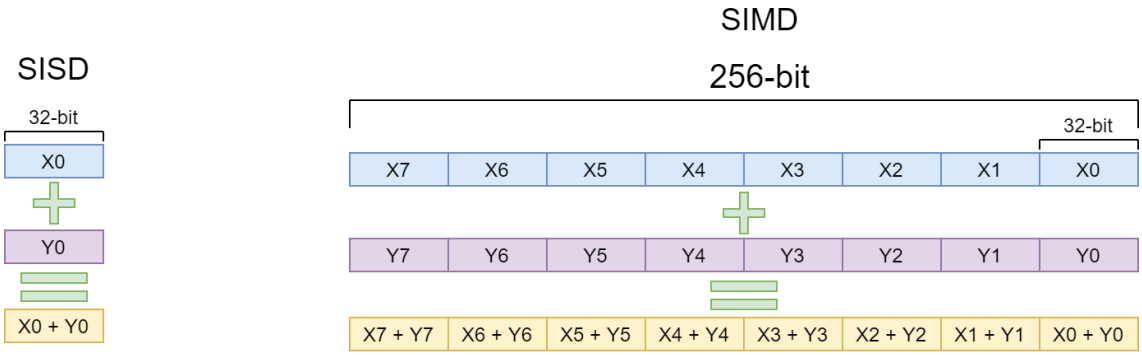
Kuva 4. Steamin käyttäjäkunnan prosessoritrendit (10).

Koska Steam on yksi suurimmista pelialustoista, voidaan sen käyttäjäkunnan arvioida antavan hyvän kuvan pelaamiseen käytettävistä tietokoneista. Koska suurella osalla pelejä pelaavista kuluttajista oletetaan olevan käytössään korkeaytimellinen moniydinprosessori, on pelikehityksessä tärkeää yrittää käyttää monisäikeistystä, jotta käyttäjien prosessoreita saadaan maksimaalisesti hyödynnettyä pelikokemuksen parantamiseksi.

2.3 SIMD-prosessointi

SIMD eli single instruction multiple data on prosessointimalli, joka hyödyntää datatason rinnakkaisuutta. Tavallisesti prosessorien ajatellaan toimivan SISD- eli single instruction single data -mallilla, jossa yhdellä käskyllä tehdään operaatio yhdelle data-alkiolle. SIMD-prosessoinnissa tehdään yksi käsky, joka operoi useaa data-alkiota samanaikaisesti. (1, s. 10–11.) Näin saadaan kasvatettua prosessorin laskentakapasiteettia yhtä kellosykliä kohden. SIMD-prosessoinnista käytetään usein myös termejä vektorisointi ja vektoriprosessointi.

Usein tavanomainen moderni prosessori käyttää 32- tai 64-bittisiä rekistereitä, joilla se voi operoida 32- tai 64-bittistä dataa. Modernit prosessorit, jotka toteuttavat SIMD-käskykantalaajennoksia, kuten SSE2 (Streaming SIMD Extensions 2) ja AVX (Advanced Vector Extensions), voivat käyttää vektorirekistereitä ja SIMD-käskyjä. SSE2-vektorirekisterit ovat 128-bittisiä, kun taas AVX-vektorirekisterit 256-bittisiä. Nämä leveät vektorirekisterit voidaan täyttää usealla pienemmällä data-alkiolla. Esimerkiksi yhteen AVX-rekisteriin voidaan sijoittaa kahdeksan 32-bittistä liukulukua ja näin voidaan operoida kokonaisilla vektoreilla data-alkioita yhden alkion sijaan. (11.) Kuvassa 5 nähdään havainnollistava esimerkki, miten SIMD-vektoriprosessoinnissa kaksi 256-bittistä vektorirekisteriä on täytetty 32-bittisillä data-alkiolla ja näin yhdellä yhteenlaskuoperaatiolla saadaan laskettua 16 data-alkiota toisiinsa.



Kuva 5. Havainnollistus SISD- ja SIMD-prosessoinnista.

Useat modernit kääntäjät yrittävät automaattisesti vektorisoida koodia, missä mahdollista (12). Vektorisoinnilla yleensä tarkoitetaan sitä, että ohjelma yrittää hyödyntää prosessorin SIMD-vektoriyksiköitä. Usein kääntäjät eivät kuitenkaan onnistu koodin vektorisoinnissa, jolloin se vaatii, että ohjelmoija käsin vektorisoi koodin. Tämä onnistuu käyttämällä esimerkiksi useita saatavilla olevia vektorikirjastoja tai kääntäjien intrinsics-standardifunktioita tai käsin kirjoittamalla kohdearkkitehtuurin assemblyä. Koska käsin tehtävä vektorisointi vaatii usein tarkkaa tietämystä ohjelman mahdollisista kohdeprosessoriarkkitehtuureista ja on muutenkin hankalampaa toteuttaa kuin tavallinen skalaarikoodi, jää vektorisointi usein tekemättä.

3 Rinnakkaislaskennan työkalut Unityssä

3.1 Unity Job System -työkalu

Unity on yksi pelialan käytetyimmistä pelimoottoreista, lähtökohtaisesti ilmainen ja sillä on mahdollista kehittää pelejä laajalle osalle eri alustoista. Unityn C# Job System on Unityn tarjoama työkalu monisäikeistetyn pelikoodin kirjoittamiseen. Se on osa Unityn DOTS- eli data oriented technology stack -kokelmaa, joka on Unityn yritys tarjota pelikehittäjille parempia työkaluja suorituskykyisten pelien kehittämiseksi. DOTS sisältää muitakin paketteja, kuten ECS:n eli entity component systemin ja Burst-kääntäjän. ECS on tarkoitettu dataorientoituneen koodin kirjoittamiseen, joka pystyy hyödyntämään prosessorin muistiarkkitehtuuria tehokkaammin kuin olio-orientoitunut koodi. Burst-kääntäjä on tarkoitettu kääntämään C#-koodia mahdollisimman tehokkaaksi konekieliseksi koodiksi käännettävän alustan ominaisuuksia hyödyntäen. DOTS on vielä kehitysvaiheessa, ja suurin osa sen ominaisuuksista on vielä merkitty esikatseluversioiksi. Unity toi Job Systemin saataville versiosta 2017.3 lähtien, ja se on myöhemmin tullut kiinteäksi osaksi Unityä. (13; 14; 15.)

Yhä laajempien ja korkealaatuisempien pelien kehittämiseksi on tärkeää, että pelikoodin suoritus on mahdollisimman tehokasta. Kuten luvussa 2.2 mainittiin, eräs tärkeimmistä tekijöistä suorituskyvyn kasvattamiseksi on hyödyntää modernien prosessorien useita ytimiä, jotka usein jäävät kokonaan käyttämättä, koska useat ohjelmat ovat täysin yksisäikeisiä. Tämän takia pelikehityksessä on tärkeää voida kirjoittaa helppoa ja tehokasta monisäikeistettyä koodia. Tavallisesti monisäikeisen koodin kirjoittaminen on hankalaa ja virhepitoista. Job System on tarkoitettu helpottamaan monisäikeisen koodin kirjoittamista ja samalla takaamaan sen virheettömyys, jolloin esimerkiksi kilpailutilanteita ei pääse syntymään.

Job Systemiä käytetään luomalla Jobeja eli kuvainnollisesti pieniä töitä, jotka annetaan Job Systemille suoritettavaksi. Job System kytkeytyy suoraan Unityn sisäisesti käyttämään natiiviin Job Systemiin, ja näin käyttäjän kirjoittamat Jobit

jakavat säikeet sen kanssa (16). Job System huolehtii säikeiden luomisesta ja Jobien ajoittamisesta (engl. scheduling) säikeille. Nämä ovat usein ongelmallisia monisäikeistettyä koodia kirjoittaessa. Job System hallitsee joukkoa työsäikeitä, jotka on jaettu prosessorin loogisille ytimille, usein yksi säie loogista ydintä kohden, jotta vältyttäisi prosessinvaihdoilta. Job System asettaa käyttäjän suoritettavaksi ajoitetut Jobit jonoon, josta työsäikeet ottavat ne suoritettavaksi. Samalla Job System pitää huolen Jobien välisistä riippuvuuksista ja varmistaa, että ne suoritetaan oikeassa järjestyksessä. (17.)

3.2 Job-työ

Job on Job Systemille suoritettavaksi annettava pienehkö työ, joka operoi sille annettavalla datalla (17). Koska Job Systemin tarkoitus on taata turvallinen kilpailutilanteeton koodi, sille ei voi antaa käsiteltäväksi C#-referenssejä, koska silloin ei voida varmistaa, käsitteleekö Unityn pääsäie Jobille annettua dataa samanaikaisesti. Tämän takia Jobeille luodaan sille annetusta datasta kopio, ja näin sille annettu data on eristetty pääsäikeeltä ja vältytään kilpailutilanteiden syntymiseltä. Jobeille voidaan antaa vain dataa, jonka muistirepresentaatio on sama hallitussa ja hallitsemattomassa koodissa. (18.) Tämä tarkoittaa sitä, että sille ei voida antaa käsiteltäväksi esimerkiksi luokkia, mutta suurin osa C#:n perustyypeistä taas, kuten esimerkiksi int, float ja double, ovat sallittuja. Myös perustyyppejä sisältävät yksiulotteiset taulukot on sallittu. Luokkien sijasta voidaan käyttää tietueita.

Koska jokaiselle Jobille kopioitu data on eristetty muusta sovelluksesta, on Job Systemissä saatavilla NativeContainer-tyyppejä, joilla dataa voidaan välittää Jobien ja muun sovelluksen välillä. NativeContainer on turvallinen rajapinta natiiviin muistihallitsemattomaan muistiin, jota käytetään osoittimen kautta. Vakiona saatava NativeContainer-tyyppi on NativeArray, mutta Unityn Entity Component System -pakkauksen mukana saa myös seuraavat tietorakenteet:

- NativeList
- NativeHashMap
- NativeMultiMap

- NativeQueue. (19.)

Kaikille NativeContainer-tyypeille on rakennettu järjestelmä, joka varmistaa niiden turvallisen käytön. Se varmistaa esimerkiksi, ettei tietorakenteen ulkopuolella olevaa muistia yritetä hakea, tarkistaa ettei tietorakenteen muistia ole vapautettu sitä käytettäessä ja varmistaa ettei kilpailutilanteita pääse syntymään. Tämä turvajärjestelmä toimii vain Unityn editoritilassa ja antaa virheilmoituksen, jos jokin edellä mainituista virheistä tapahtuu. Jos esimerkiksi kaksi suoritettavaa Jobia yrittää kirjoittaa samaan NativeContaineriin, syntyy siitä virheilmoitus editorissa. Jos kahdelle eri Jobille halutaan kirjoitusoikeus samaan NativeContaineriin, ne on ajoitettava riippuvuudella, jolloin kun ensimmäinen Job on valmis, pääsee toinen Job käsiksi NativeContaineriin. Jos NativeContainer merkitään [ReadOnly]-attribuutilla, voivat Jobit lukea siitä samaan aikaan. (19.)

NativeContainer-tyyppejä luodessa pitää niiden muistivaraustyyppi valita sen mukaan, kuinka kauan sitä käyttävä Job kestää ja kuinka kauan muistia tarvitaan. Oikein valittu varaustyyppi varmistaa parhaan suorituskyvyn. Varaustyyppejä on kolme:

- Allocator.Temp
- Allocator.TempJob
- Allocator.Persistent. (19.)

Temp tarjoaa nopeimman muistivarauksen, ja sitä käytetään, kun muistivarausta tarvitaan alle yhden kehyksen ajan. Temp-varauksella varattua muistia ei voi kuitenkaan antaa Jobeille. TempJob on alle neljä kehystä kestäville Jobeille tarkoitettu varaustyyppi, joka on hitaampi kuin Temp ja pitää vapauttaa neljän kehyksen aikana. Persistent on hitain varaustyyppi, jota käytetään, kun varattavaa muistia tarvitaan pidempään. Ohjelmoijan pitää itse huolehtia kaikkien NativeContainer-tyyppien vapauttamisesta, kun niitä ei enää tarvita, kutsumalla niiden Dispose-metodia. (19.) Kuvassa 6 on esitetty, miten NativeContainer-tyyppejä varataan.

```
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);
```

Kuva 6. Esimerkki yhden alkion pituisen liukulukuja sisältävän NativeArray-taulukon varauksesta TempJob-varaustyypillä (19).

Itse Jobien luominen tehdään luomalla C#-tietue, joka toteuttaa IJob-rajapintaluokan. IJob on yksinäinen työ, joka suoritetaan rinnakkain muiden Jobien kanssa. IJob-tietueelle annetaan sen tarvitsemat jäsenmuuttujat NativeContainer tulosten välittämiseksi, ja sille toteutetaan Execute-metodi, joka ajetaan kerran yhdellä ytimellä. (20.) Kuvassa 7 on toteutettu yksinkertainen IJob-rajapintaluokan toteuttava Job.

```
// Job adding two floating point values together
public struct MyJob : IJob
{
    public float a;
    public float b;
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = a + b;
    }
}
```

Kuva 7. Yksinkertainen Job, joka sen Execute-metodissa laskee jäsenmuuttujat a ja b yhteen ja tallentaa vastauksen NativeArray-taulukkoon (20).

Jobien ajoittamiseksi suoritusta varten pitää ensiksi luoda Job-instanssi IJob-rajapintaluokan toteuttamasta tietueesta, antaa sille sen tarvitsema data ja kutsua sille Schedule-metodia. Schedule-metodi palauttaa JobHandle-muuttujan, jonka kautta Jobia hallitaan. Schedule-metodia kutsuttaessa Job laitetaan jonoon odottamaan suoritusta. Kun halutaan varmistua Jobin suorituksesta, voidaan

sen JobHandlelle kutsua Complete-metodia. Complete-metodi priorisoi Jobin suorituksen ja Jobin valmistuessa palauttaa Jobille annettujen NativeContainerien käyttöoikeuden takaisin pääsäikeelle.

Unityn mukaan Jobit kannattaa ajoittaa mahdollisimman aikaisin kehyksellä ja varmistaa niiden suoritus mahdollisimman myöhään esimerkiksi LateUpdate-metodissa, jotta Jobit saavat mahdollisimman paljon aikaa suorittamiseen. Jos luodaan samanaikaisesti useita Jobeja, kannattaa ne kaikki ensin ajoittaa ja sitten vasta kutsua niille Complete-metodia, koska työsäikeiden herättäminen Jobien suoritukseen voi olla raskasta. Jos halutaan varmistaa kaikkien jonossa olevien Jobien suorituksen aloittaminen, voidaan niiden suoritus aloittaa käyttämällä JobHandle.ScheduleBatchedJobs-metodia, joka aloittaa kaikkien jonoon asetettujen Jobien suorituksen. Jobin valmiutta voidaan tarkistaa JobHandlen isCompleted-muuttujalla, jonka tarkistuksen jälkeen kutsutaan Complete-metodia, jolla varmistetaan Jobin valmius. (21; 22; 23; 24.) Kuvassa 8 on esimerkki Jobien käyttämisestä.

```
// Create a native array of a single float to store the result. This example waits for the job to complete for illustration purposes
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

// Set up the job data
MyJob jobData = new MyJob();
jobData.a = 10;
jobData.b = 10;
jobData.result = result;

// Schedule the job
JobHandle handle = jobData.Schedule();

// Wait for the job to complete
handle.Complete();

// All copies of the NativeArray point to the same memory, you can access the result in "your" copy of the NativeArray
float aPlusB = result[0];

// Free the memory allocated by the result array
result.Dispose();
```

Kuva 8. Esimerkki Jobien luomisesta, ajoittamisesta, suorittamisen varmistamisesta, tulosten käyttämisestä ja varatun NativeArrayn vapauttamisesta (21).

Koska tavallinen Job tekee vain yhden tehtävän, se ei ole tehokas, kun halutaan tehdä suuri määrä samoja operaatiota isommalle määrälle dataa. Tätä varten on Job Systemissä tavallisen Jobin lisäksi myös ParallelFor- ja

ParallelForTransform-Jobeja. Nämä Jobit ovat tietueita, jotka toteuttavat IJobParallelFor- tai IJobParallelForTransform-luokkarajapinnan. ParallelFor-Jobit ajetaan usealla ytimellä rinnakkain, jolloin aloitetaan yksi Job prosessoriydintä kohden. ParallelForTransform-Job on tehty Transform-komponenttien operoimiseen, jotka toimitetaan sille TransformAccessArrayssä. ParallelFor-Jobien Execute-metodia kutsutaan kerran jokaiselle sille annetulle NativeArrayn tai TransformAccessArrayn elementille. (25; 26.) Execute-metodi saa käsiteltävän elementin indeksin parametrina kuvassa 9 esitetyllä tavalla.

```
// Job adding two floating point values together
public struct MyParallelJob : IJobParallelFor
{
    [ReadOnly]
    public NativeArray<float> a;
    [ReadOnly]
    public NativeArray<float> b;
    public NativeArray<float> result;

    public void Execute(int i)
    {
        result[i] = a[i] + b[i];
    }
}
```

Kuva 9. IJobParallelFor-Job (25).

ParallelFor-Jobeja ajoittaessa pitää niille antaa lähdedatana käytettävän taulukon koko ja yhden Jobin käsittelemän taulukon osan koko, niin sanottu eräkoko

(engl. batch count), jolloin työ voidaan jakaa tasaisesti Jobien välillä (25). Kuvassa 10 on esitetty ParallelFor-Jobia varten tarvittava pääsäikeen koodi.

```
NativeArray<float> a = new NativeArray<float>(2, Allocator.TempJob);

NativeArray<float> b = new NativeArray<float>(2, Allocator.TempJob);

NativeArray<float> result = new NativeArray<float>(2, Allocator.TempJob);

a[0] = 1.1;
b[0] = 2.2;
a[1] = 3.3;
b[1] = 4.4;

MyParallelJob jobData = new MyParallelJob();
jobData.a = a;
jobData.b = b;
jobData.result = result;

// Schedule the job with one Execute per index in the results array and only 1 item per processing batch
JobHandle handle = jobData.Schedule(result.Length, 1);

// Wait for the job to complete
handle.Complete();

// Free the memory allocated by the arrays
a.Dispose();
b.Dispose();
result.Dispose();
```

Kuva 10. Pääsäikeessä tehtävät valmistelut ja ParallelFor-Jobin suoritus (25).

Optimaaliseen eräkoon löytämiseksi Unity suosittelee aloittamaan asettamalla eräkoon ensiksi yhdeksi ja kasvattamalla sitä, kunnes suorituskyky ei enää parane (25).

3.3 Burst-kääntäjä

Unity tukee tavallisesti kahta eri skriptialustaa C#-skriptien kääntämiseksi valmista peliä koottaessa. Vakiona Unity käyttää Monon tarjoamaa skriptialustaa. Mono on avoimen lähdekoodin käyttöjärjestelmäriippumaton ohjelmistokehys. Unity tarjoaa myös vaihtoehtoisesti IL2CPP (Intermediate Language To C++) -skriptialustan alustoille, jotka eivät tue Monoa.

Mono käyttää ajonaikaista käännöstä, mikä tarkoittaa, että skriptit käännetään kohdearkkitehtuurin konekieliseksi koodiksi vasta sovellusta suoritettaessa skriptejä tarvittaessa. Tämän etuna ovat nopeat käännösajat ja

alustariippumattomuus, koska alustalle natiivi konekielinen koodi tuotetaan vasta ajonaikana. Jotkin alustat eivät kuitenkaan tue ajonaikaista kääntämistä, ja tätä varten Unity Tarjoaa IL2CPP-skriptialustan, joka käyttää ennenaikaista kääntämistä, jossa lähdekoodi käännetään sovelluksen koontiaikana kohdealustan konekieliseksi koodiksi. IL2CPP-kääntäjä kääntää esimerkiksi C#-skripteistä tuotetun välikielen C++-koodiksi, joka käännetään lopulliseksi kohdearkkitehtuurin konekieliseksi koodiksi. Tällä käännöstavalla saattaa olla suorituskykyetuja ajonaikaiseen kääntämiseen verrattuna, mutta sillä kääntäminen on myös usein hitaampaa. Samoin kuin ajonaikaisessa kääntämisessä, jotkin alustat eivät tue ennenaikaista kääntämistä. (27; 28.)

Burst on pääasiassa Job Systemiä ja ECS:ää varten kehitetty kääntäjä, jonka tarkoitus on kääntää C#-ohjelmointikielestä tuotettua välikieltä erittäin suorituskykyiseksi konekieliseksi koodiksi käyttäen LLVM:ää (29; 30). LLVM (Low Level Virtual Machine) on laajasti käytetty alustariippumaton ohjelmointikielen kääntäjä. Burst kehitettiin, koska Unity tarvitsi tehokkaan ja toimivan tavan päästä C++-ohjelmointikielen tasolle suorituskyvyssä, johon Mono-, IL2CPP- tai .NET-alustat eivät kyenneet (31; 32). .NET on Microsoftin kehittämä ohjelmistokehys, joka toimii pääasiassa Windows-käyttöjärjestelmällä. Unityn Alexandre Mutelin (32) mukaan Monon tuottama koodi voi olla 30–300 % hitaampaa kuin .NET-alustan tuottama koodi ja 1000–3000 % hitaampaa kuin C++-koodi.

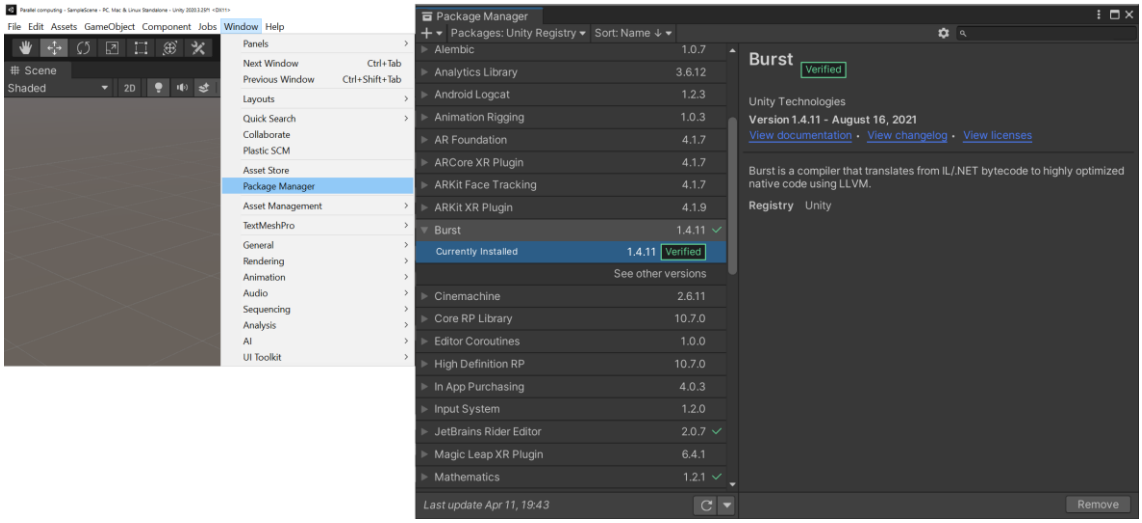
Unityn mukaan C++ ei ollut sopiva vaihtoehto sen käytön monimutkaisuuden takia, ja siksi päätettiin kehittää oma Unityyn erikoistettu kääntäjä, joka käyttäisi karsittua versiota C#-kielestä. C# valittiin, koska se oli ennestään tuttu Unity-kehittäjille ja sille löytyi valmiiksi hyvät kehitysympäristöt ja työkalut. (31.)

Unityn Andreas Fredrikssonin (33) mukaan yleiskäyttöisellä kääntäjällä on hankala saavuttaa korkeaa suorituskykyä erikoistuneeseen kääntäjään verrattuna, minkä takia Unityyn kehitettiin erikoistunut kääntäjä. Koska Burst on Unityä varten erikoistunut kääntäjä, sille on mahdollista asettaa rajoitteita, jotka parantavat koodin optimoitavuutta. Paremman suorituskyvyn saavuttamiseksi Burst käyttää C#-ohjelmointikielestä karsittua versiota, jota Unity kutsuu HPC#:ksi eli High

performance C#:ksi. HPC#:ssa ei ole esimerkiksi roskienkerääjää eikä luokkatyyppejä. Kielestä on myös karsittu monet C#-standardikirjaston ominaisuudet, kuten Linq, StringFormatter, List ja Dictionary. (31.) Fredrikssonin mukaan Burstin kontekstitietoisen analyysin ansiosta Burst pääsee eroon muistin lomittumisesta, joka on hyvin yleinen ongelma yleiskäyttöisissä kääntäjissä ja estää koodin vektorisoinnin. Tämä mahdollistaa automaattisen koodin vektorisoinnin useimmissa toistorakenteissa, jolloin voidaan usein saavuttaa yli 4-kertaiset suorituskykyparannukset. (33.)

Unity editorin Playmode-tilassa Burst-käännettävät Jobit käännetään vakiona asynkronisesti. Tämä tarkoittaa sitä, että Unity käyttää ensin Mono-käännettyä versiota Jobista, kunnes Burst-versio on saatu käännettyä ja se vaihdetaan lennossa. (29.) Useimmille alustoille koottua peliä varten Burst tuottaa dynaamisen kirjastotiedoston, joka sisältää Burstin tuottaman konekielisen koodin. iOS-alustalle koottaessa tuotetaan staattinen kirjastotiedosto. (34.)

Burst-kääntäjän saa käyttöön asentamalla sen Unityn Package Managerista (kuva 11).



Kuva 11. Burst-kääntäjän asennus Unityn Package Managerista.

Burst-kääntäjän käyttämiseksi pitää Jobin sisältämä skriptitiedosto merkitä käyttämään Unity.Burst-nimiavaruutta ja Burst-käännettävä Job merkitä BurstCompile-attribuutilla kuvan 12 näyttämällä tavalla.

```
using Unity.Burst;
[BurstCompile]
2 references
struct InfluenceMapJob : IJobParallelFor
{
```

Kuva 12. Burst-nimiavaruuden sisällyttäminen C#-skriptiin ja Jobin merkitseminen Burst-käännettäväksi.

Manuaalista vektorisointia varten Burst pystyy optimaalisesti käyttämään Unity.Mathematics-kirjaston SIMD-vektorityyppejä, kuten int4 ja float4 (35). Unity.Burst.Instrisics-kirjastosta löydetään myös alustakohtaiset käskyt tarkempaa alustakohtaista optimointia varten (36). Burst tarjoaa Job-kohtaisen liukulukujen tarkkuuden määrittämisen ja liukuluku laskutoimitusten järjestyksen uudelleenasettelun sisällyttämällä BurstCompile-attribuuttiin FloatPrecision- ja FloatMode-asetukset (37). Kuvassa 13 nähdään, miten InfluenceMapJob Burst-käännetään standardilla liukulukutarkkuudella ja FloatMode.Fast-asetuksella.

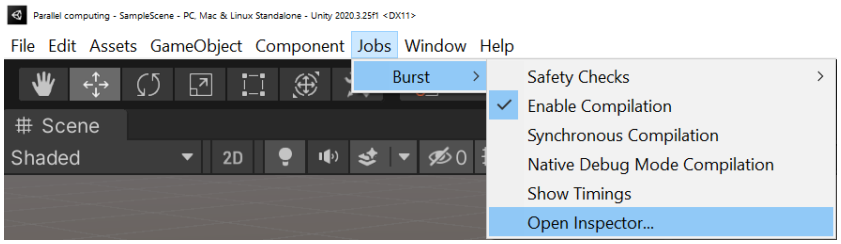
```
[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
2 references
struct InfluenceMapJob : IJobParallelFor
{
```

Kuva 13. Job merkitty käyttämään standardia liukulukutarkkuutta ja FloatMode.Fast-asetusta.

Alentamalla käytettävien liukulukujen tarkkuutta voidaan saada suorituskykyparannuksia, samoin kuin myös sallimalla Burst-kääntäjän uudelleen järjestää liukulukulaskutoimitusten järjestyksen. Tarkkuutta voidaan myös nostaa High-asetukselle ja asettaa Burst käyttämään deterministisiä liukulukulaskuja. Unityn Burst-kääntäjän dokumentaatiosivulta löydetään hyvin runsas määrä eri optimointitapoja, suosituksia ja asetuksia. (37.)

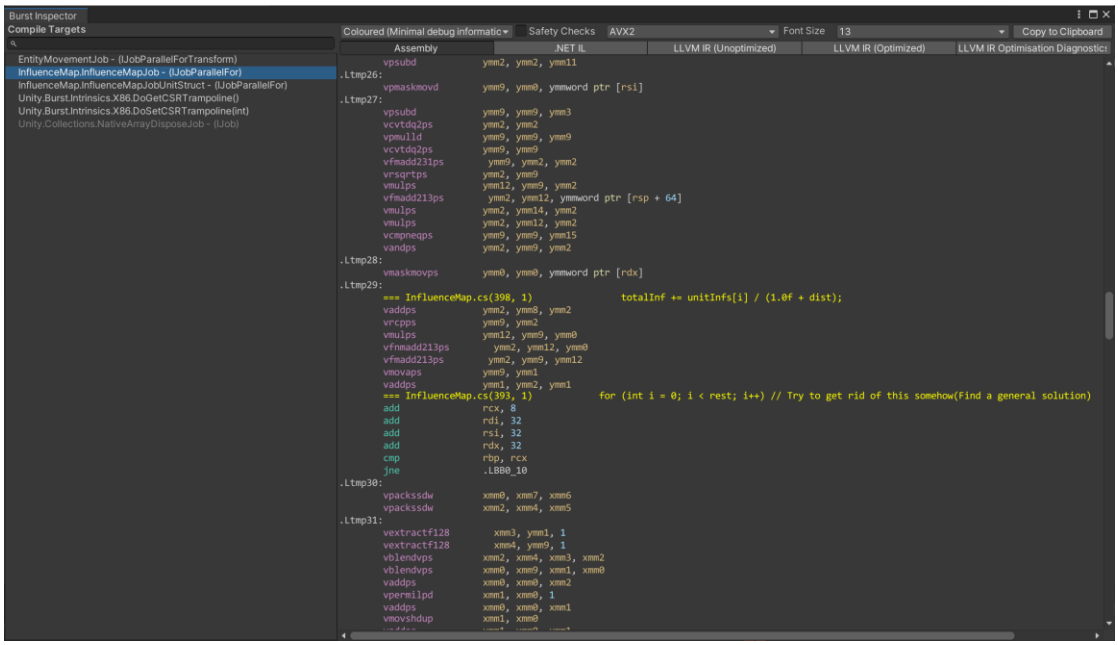
3.4 Burst Inspector -työkalu

Unity-editorissa, Jobs-valikon alta löydetään Burst-valikko, josta voidaan kontrolloida Burst-asetuksia. Valikosta voidaan esimerkiksi kytkeä Burst-käännetyille Jobeille turvatarkistukset päälle ja avata Burst Inspector -työkalu.



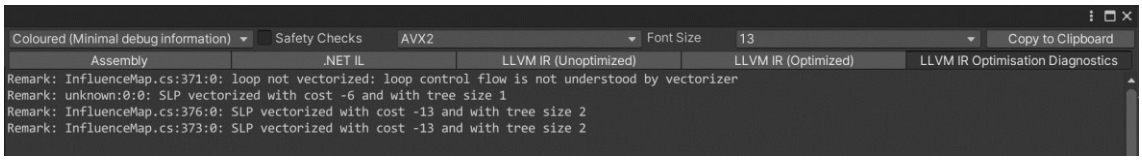
Kuva 14. Burst-valikko Job-valikon alla.

Burst Inspectorissa voidaan tutkia Burst-käännettyjä Jobeja ja niiden tuottamaa alustakohtaista konekielistä koodia. Inspectorissa voidaan tutkia, minkälaista konekielistä koodia Burst tuottaa eri käskykannoille. Virheiden välttämiseksi Burst Inspectorissa on vakiona turvatarkistukset päällä, jolloin Burst Inspector ei usein näytä parhaiten optimoitua versiota koodista, minkä takia lopullisen optimoidun koodin tutkimista varten turvatarkistukset kannattaa ottaa pois päältä. Kuvassa 15 on näkyvissä Burst-kääntäjän tuottama konekielinen koodi InfluenceMapJob-Jobille, kun käytetään AVX2-käskykantalaajennosta ja kun turvatarkistukset on kytketty pois päältä.



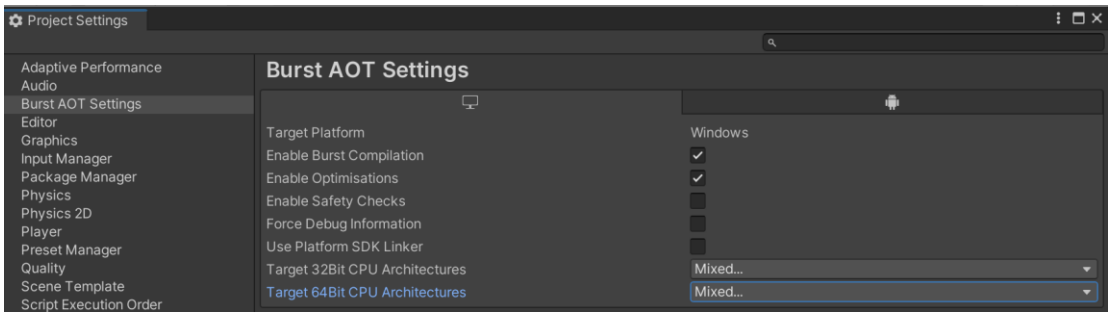
Kuva 15. Burst Inspector -ikkuna.

LLVM IR Optimization Diagnostics -sarakkeesta selviävät muun muassa kääntäjän vektorisointiyritykset. Kuvassa 16 nähdään, miten Burst antaa tiedon vektorisointistatuksista.



Kuva 16. Burst Inspectorin optimointidiagnostiikkasarake.

Unityn projektiasetuksista löydetään Burst AOT settings -valikko, josta voidaan valita, mille prosessoriarkkitehtuureille Burst-käännös tehdään ennenaikaista käännöstä käytettäessä.



Kuva 17. Burstin ennenaikaisen käännöksen asetukset projektiasetukset-valikossa.

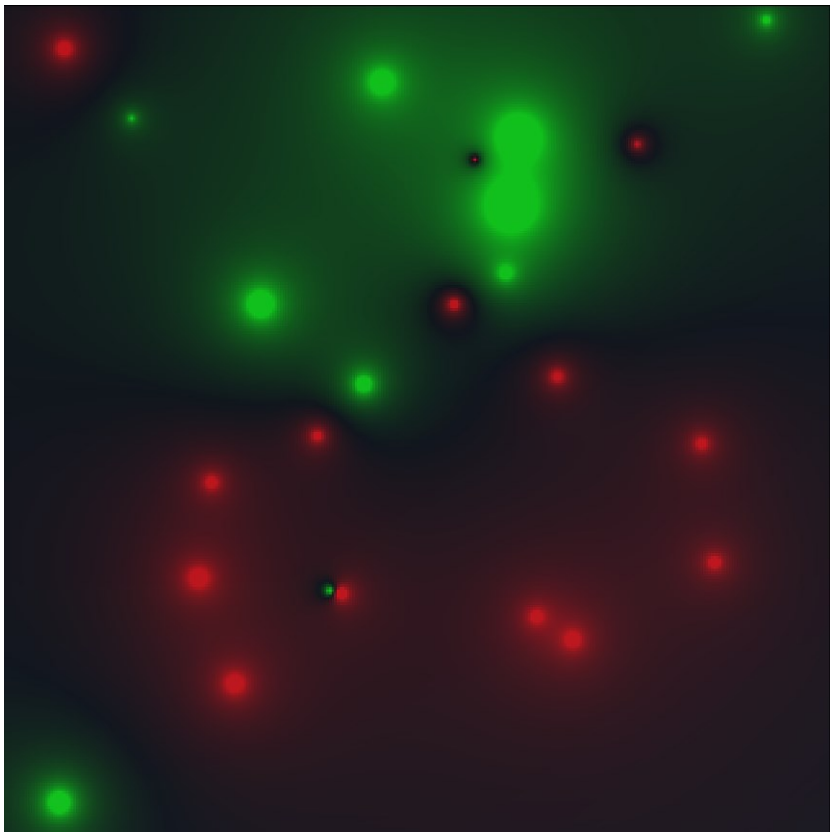
Kun useampi arkkitehtuuri on valittuna, Burst tekee käännökset jokaista valittua arkkitehtuuria kohden (34).

4 Rinnakkaislaskennan käyttö testisovelluksessa

Insinöörityössä kehitettiin Unityllä yksinkertainen testisovellus, joka oli tarkoitus optimoida käyttäen Unityn Job Systemiä ja Burst-kääntäjää. Tarkoituksena oli selvittää, kuinka hyvin nämä työkalut soveltuvat Unityllä kehitettävän sovelluksen optimoimiseen monisäikeistämisen ja muiden optimointien, kuten vektorisoimisen, kannalta.

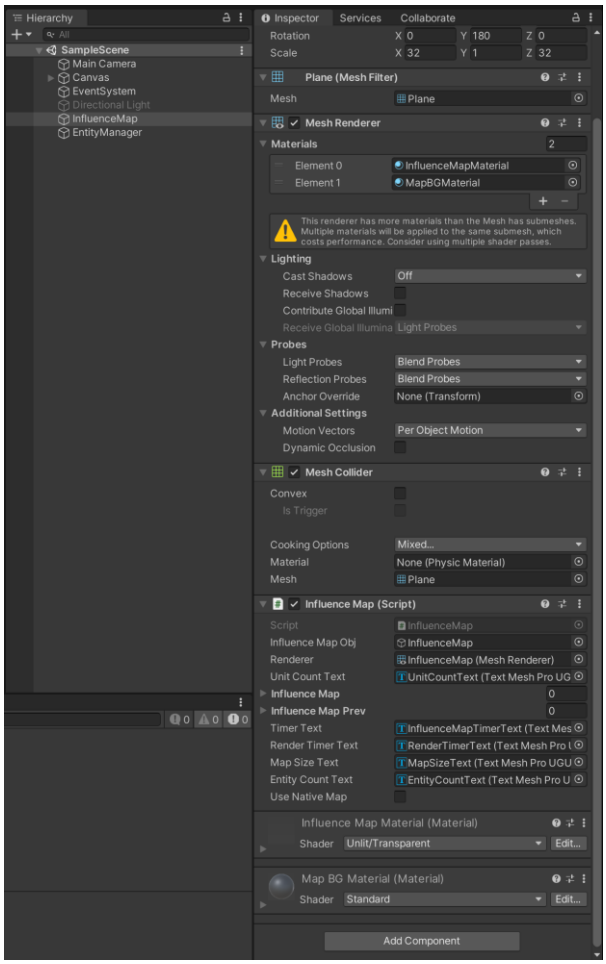
4.1 Vaikutuskarttasovellus

Toteutettavaksi sovellukseksi valittiin suhteellisen yksinkertainen vaikutuskarttasovellus. Vaikutuskartta on kaksiulotteinen ruudukko, jonka kaikkiin ruutuihin lasketaan kartalla olevien yksiköiden vaikutus. Tässä tapauksessa kartalle aseteltiin satunnaisesti joko positiivisia tai negatiivisia yksiköitä, kuten kuvassa 18 näkyy. Yksiköiden vaikutus kartan ruutuihin vähenee etäisyyden mukaan. Tällaista karttaa voidaan hyödyntää esimerkiksi jossain pelissä. Vaikutuskartan laskemisen lisäksi kartalle voitiin asettaa liikkuvia olioita, jotka hakeutuivat ystävällistä vaikutusta kohti.



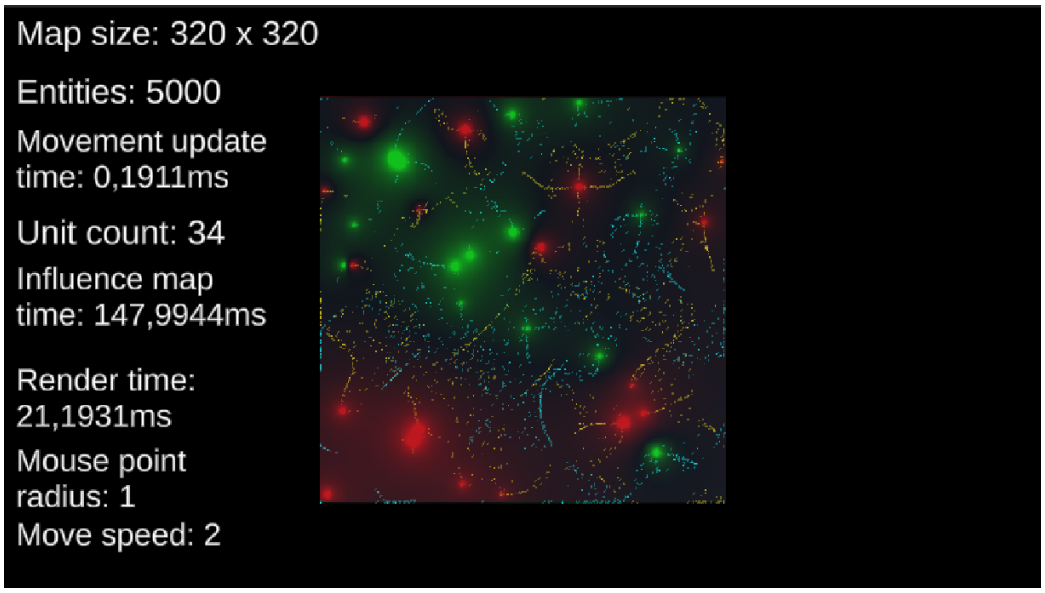
Kuva 18. Laskettu vaikutuskartta. Vihreät yksiköt kuvastavat positiivisia yksiköitä ja punaiset negatiivisia yksiköitä.

Testisovellusta varten Unity-projektiin rakennettiin yksinkertainen näkymä (engl. scene), johon luotiin peliobjekti, jonka komponenttina oli vaikutuskartan laskentaskripti. Tähän peliobjektiin lisättiin myös MeshRenderer-komponentti ja tarvittavat materiaalit kartan visualisoimiseksi (kuva 19). Lisäksi näkymään luotiin peliobjekti, joka hallitsi olioiden luomista ja liikkumista siihen komponentiksi asetetun EntityManager-skriptin kautta.



Kuva 19. Testisovelluksen näkymän hierarkia ja InfluenceMap-peliobjektin komponentit.

Kartan lisäksi sovelluksen yksinkertaiseen käyttöliittymään lisättiin tekstiobjekteja testausta varten (kuva 20).



Kuva 20. Vaikutuskarttasovelluksen käyttöliittymä.

Käyttöliittymästä nähdään muun muassa olioiden määrä kartalla ja niiden simuloimiseen kulunut aika, kartan yksiköiden määrä ja kartan laskemiseen kulunut aika.

4.2 Vaikutuskartan laskeminen

Vaikutuskartan laskemista varten luodaan ensiksi kartalle aseteltavat yksiköt. Yksiköt voidaan esittää kolmella muuttujalla, x- ja y-koordinaateilla sekä vaikutusarvolla, joka voi olla positiivinen tai negatiivinen riippuen siitä, kumman puolen yksikköä halutaan kuvata. Käytännössä yksiköt voidaan esittää koodissa kolmena taulukkona tai yhtenä taulukkona olioita tai tietueita, joilla on nämä kolme jäsenmuuttujaa (kuva 21).

```
public struct Unit
{
    public int x;
    public int y;
    public float influence;
}
List<Unit> units;
```

Kuva 21. Yksikön esittäminen listallisena tietueita.

Jos halutaan noudattaa dataorientoitunutta suunnittelua, voidaan yksiköt esittää kolmena taulukkona tai listana sitä kuvaavia muuttujia kuvan 22 mukaisesti. Yksiköitä läpikäydessä voidaan samalla indeksillä hakea kaikki yksikön data kolmesta eri taulukosta.

```
List<int> unitXCords;
List<int> unitYCords;
List<float> unitInfs;
```

Kuva 22. Yksiköiden esittäminen kolmena listana.

Yksikködatan luomista varten tehtiin kuvan 23 mukainen metodi. Metodiin voitiin antaa argumentiksi totuusarvo, jonka ollessa totta luotiin yksikködata ennalta päätetyn siemenluvun perusteella. Tämä vaihtoehto luotiin yhtenäistä testausta varten. Metodissa päätetään luotavien yksiköiden määrä, nollataan aiemmat listat ja täytetään listat uudella yksikködatalla.

```
void GenerateRandomUnitData(bool deterministic = false)
{
    // Satunnaislukugeneraattorille voidaan asettaa ennalta valittu siemenluku deterministisen kartan luomiseksi.
    if(deterministic)
    {
        Random.InitState(deterministicSeed);
    }

    // Valitaan luotavien yksiköiden määrä satunnaisesti.
    int unitsGreen = Random.Range(minUnitsPerSide, maxUnitsPerSide);
    int unitsRed = Random.Range(minUnitsPerSide, maxUnitsPerSide);

    int totalUnits = unitsGreen + unitsRed;

    unitXCords.Clear();
    unitYCords.Clear();
    unitInfs.Clear();

    // Luodaan yksiköt

    for (int i = 0; i < unitsGreen; i++)
    {
        int x = Random.Range(0, COLS - 1);
        int y = Random.Range(0, ROWS - 1);
        float influence = Random.Range(minInfluence, maxInfluence);

        unitXCords.Add(x);
        unitYCords.Add(y);
        unitInfs.Add(influence);
    }


    for (int i = unitsGreen; i < totalUnits; i++)
    {
        int x = Random.Range(0, COLS - 1);
        int y = Random.Range(0, ROWS - 1);
        // Punaisille yksiköille annetaan negatiivinen vaikutusarvo.
        float influence = Random.Range(minInfluence, maxInfluence) * -1.0f;

        unitXCords.Add(x);
        unitYCords.Add(y);
        unitInfs.Add(influence);
    }
}
```

Kuva 23. Yksikködatan luontimetodi.

Kun yksikködata on luotu, voidaan aloittaa vaikutuskartan laskeminen. Itse vaikutuskartta esitetään yksiulotteisena taulukkona liukulukuja, jotka voivat olla positiivisia tai negatiivisia. Vaikutuskartan dimensiot oli tässä tapauksessa ilmaistu vakioina ja taulukko varattu InfluenceMap-skriptin Start-metodissa. Vaikutuskartan laskeminen tehdään käymällä jokainen kartan ruutu läpi ja laskemalla kaikkien yksiköiden vaikutus käsiteltävään ruutuun (kuva 24).

```
void CalculateInfluenceMap()
{
    for (int x = 0; x < COLS; x++)
    {
        for (int y = 0; y < ROWS; y++)
        {
            int index = x * ROWS + y;
            influenceMap[index] = CalculatePointInfluence(x, y);
        }
    }
}
```

Kuva 24. Vaikutuskartan laskentasilmukka.

Kuten kuvassa 25 näkyy, yksittäisen ruudun vaikutus lasketaan käymällä kaikki yksiköt läpi ja laskemalla niiden kokonaisvaikutus käsiteltävään ruutuun. Yksiköiden vaikutus ruutuun vähenee niiden etäisyyden mukaan.

```
float CalculatePointInfluence(int col, int row)
{
    float totalInfluence = 0.0f; // Kokonaisvaikutus.

    // Käydään läpi kaikki yksiköt ja summataan niiden vaikutus.
    for (int i = 0; i < unitInfs.Count; i++)
    {
        int x = unitXCords[i];
        int y = unitYCords[i];

        // Lasketaan yksikön etäisyys ruudusta.
        float distance = Distance(col, row, x, y);

        // Yksikön vaikutus ruutuun laskee etäisyyden mukaan.
        totalInfluence += unitInfs[i] / (1 + distance);
    }
    return totalInfluence;
}
```

Kuva 25. Yhden ruudun vaikutuksen laskeminen.

Kun vaikutuskartta on laskettu, sitä voidaan käyttää hyväksi jossain pelimekaniikassa ja se on mahdollista myös visualisoida esimerkiksi tekstuurina.

4.3 Vaikutuskartan visualisointi

Jos vaikutuskartta halutaan visualisoida, se voidaan esittää esimerkiksi tekstuurina. Insinöörityössä kartta visualisoitiin tekstuuriin käyttämällä Unityn tarjoamia tekstuurinkäsittelymetodeja. Kun vaikutuskartta on laskettu, kutsutaan kuvassa 26 näkyvää metodia, joka päivittää InfluenceMap-peliobjektin materiaalin

tekstuurin laskettua vaikutuskarttaa vastaavaksi. Tekstuuri oli valmiiksi varattu vaikutuskartan kokoiseksi skriptin Start-metodissa.

```
void RenderMapToTexture(float[] map)
{
    // Käydään kaikki kartan ruudut läpi.
    for (int x = 0; x < COLS; x++)
    {
        for (int y = 0; y < ROWS; y++)
        {
            int index = x * ROWS + y;

            // Muutetaan värin läpinäkyvyyttä vaikutuksen itseisarvon mukaan.
            float influence = map[index];
            float alpha = 255 * Mathf.Abs(influence);

            // Puristetaan läpinäkyvyys välille 0-255.
            if (alpha < 0.0f)
            {
                alpha = 0.0f;
            }
            else if (alpha > 255.0f)
            {
                alpha = 255.0f;
            }

            // Asetetaan tekstuuriin vaikutusta vastaava pikseli.
            if (influence >= 0)
            {
                greenColor.a = (byte)alpha;
                influenceMapTexture.SetPixel(x, y, greenColor);
            }
            else
            {
                redColor.a = (byte)alpha;
                influenceMapTexture.SetPixel(x, y, redColor);
            }
        }
    }

    // Päivitetään tekstuuri.
    influenceMapTexture.Apply();
}
```

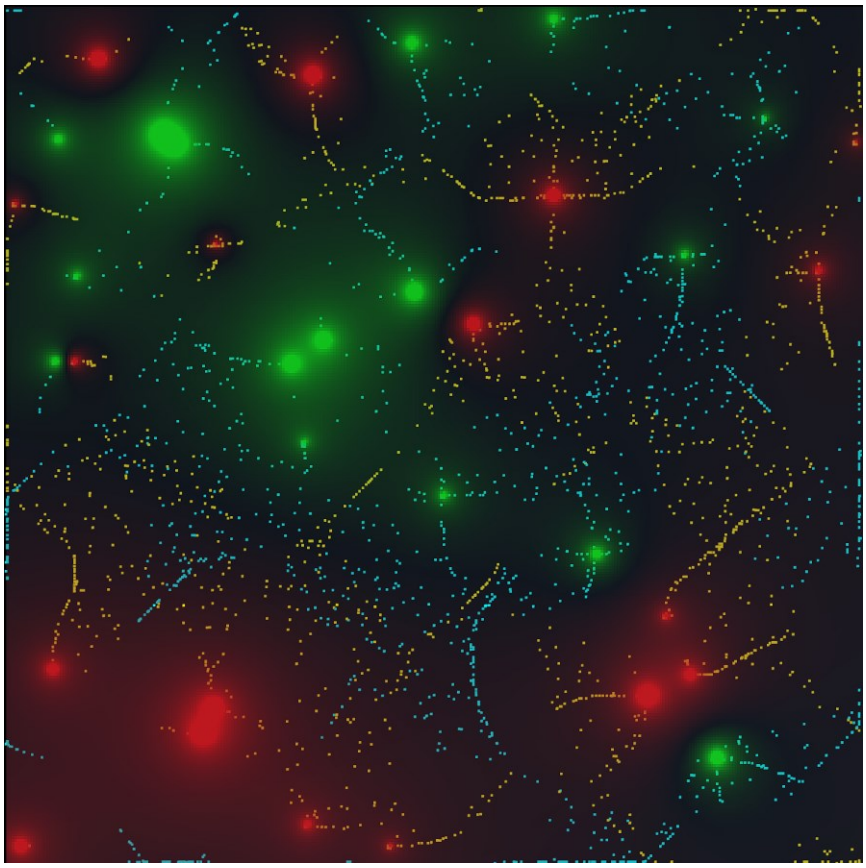
Kuva 26. Vaikutuskartan tekstuurin päivitysmetodi.

Metodissa käydään läpi kaikki vaikutuskartan elementit ja päivitetään elementtiä vastaavan pikselin väri ja läpinäkyvyysarvo elementin vaikutusarvon mukaan.

4.4 Kartalla liikkuvat oliot

Vaikutuskartan käyttöä haluttiin esitellä tavalla, joka saattaisi olla jossain pelissä ja joka olisi rinnakkaistettavissa ja optimoitavissa Job Systemillä ja Burst-

kääntäjällä. Tämän takia kartalle oli mahdollista luoda olioita, jotka liikkuivat positiivista tai negatiivista vaikutusta kohti. Kuvassa 27 nähdään, miten keltaiset yksiköt liikkuvat punaista vaikutusta kohti ja poispäin vihreästä vaikutuksesta ja siniset yksiköt vihreätä vaikutusta kohti ja poispäin punaisesta vaikutuksesta.



Kuva 27. Liikkuvia olioita vaikutuskartalla.

Olioita hallitsemaan luotiin testisovelluksen näkymään uusi peliobjekti, johon luotiin komponentiksi EntityManager-skripti. EntityManager vastasi olioiden luomisesta ja niiden liikkumisen simuloimisesta. Yksittäinen olio oli valmiiksi luotu prefab eli uudelleenkäytettävä versio peliobjektista, jolla oli Transform-komponentin lisäksi Mesh-Renderer-komponentti yksinkertaisella värimateriaalilla.

Olioita luotaessa tallennettiin niiden Transform-komponentit ja puolet omiin listoihinsa, joita käsiteltiin EntityMovement-metodissa, joka simuloi olioiden liikettä. Puoli ilmaistiin totuusarvona, jolloin tosi vastasi positiivista puolta ja

epätosi negatiivista puolta. EntityManager-skriptin käyttämä vaikutuskartta haettiin InfluenceMap-skriptistä LateUpdate-metodissa. Kuvissa 28, 29 ja 30 esitettyä EntityMovement-metodia kutsuttiin EntityManager-skriptin Update-metodissa. Kuvassa 28 on EntityMovement-metodin alku, jossa on silmukka, joka käy kaikki olioiden Transform-komponentit läpi. Silmukan alussa pyöristetään olion koordinaatit kokonaisluvuiksi ja haetaan olion puoli. Lisäksi asetetaan liikkumisen ja korkeimman löydetyn vaikutuksen vakioarvot.

```
void EntityMovement()
{
    timer.Restart();

    // Käydään läpi kaikkien olioiden Transform-komponentit.
    for (int i = 0; i < entityTransformList.Count; i++)
    {
        Transform t = entityTransformList[i];

        float x = t.position.x;
        float y = t.position.z;

        // Pyöristetään koordinaatit kokonaisluvuiksi vaikutuskartan indeksointia varten.
        int intX = Mathf.RoundToInt(x);
        int intY = Mathf.RoundToInt(y);

        // Haetaan olion puoli.
        bool side = entitySidesList[i];

        int moveIndex = intX * ROWS + intY; // Vakiona liikutaan nykyiseen ruutuun.

        // Asetetaan korkein vaikutus nykyisen ruudun mukaan.
        float highestInfluence = influenceMap[moveIndex];
```

Kuva 28. EntityMovement-metodin alku.

Kuvassa 29 näkyy, miten EntityMovement-metodissa käydään läpi kaikki olion välittömässä läheisyydessä olevat vaikutuskartan ruudut ja ruutu, johon liikutaan, päivitetään korkeimman löydetyn vaikutuksen mukaan.

```
// Käydään läpi kaikki ruudut välittömässä läheisyydessä
// ja valitaan ruutu jolla on korkein vaikutus olion puolta kohtaan.
for (int j = 0; j < offsets.Length; j += 2)
{
    int xPos = intX + offsets[j];
    int yPos = intY + offsets[j + 1];

    // Tarkistetaan että koordinaatit ovat kartan sisällä.
    if (xPos < 0 || xPos >= COLS
        || yPos < 0 || yPos >= ROWS)
    {
        continue;
    }
    else
    {
        // Lasketaan koordinaatteja vastaava taulukon indeksi
        // ja haetaan sitä vastaava vaikutusarvo vaikutuskartasta.
        int mapIndex = xPos * ROWS + yPos;

        float influence = influenceMap[mapIndex];

        // Päivitetään korkein vaikutusarvo ja liikkumis indeksi puolen mukaan.
        if (side)
        {
            if (influence > highestInfluence)
            {
                highestInfluence = influence;
                moveIndex = mapIndex;
            }
        }
        else
        {
            if (influence < highestInfluence)
            {
                highestInfluence = influence;
                moveIndex = mapIndex;
            }
        }

    }
}
```

Kuva 29. Välittömässä läheisyydessä olevien ruutujen läpikäynti.

Lopuksi valitun liikkumisindeksin mukaan lasketaan liikevektori ja olion sijaintia muutetaan liikkumisnopeuden ja liikevektorin mukaan kuvassa 30 esitetyllä tavalla.

```
        // Muunnetaan yksiulotteinen taulukon indeksi x- ja y-koordinaateiksi.
        float moveX = moveIndex / ROWS;
        float moveY = moveIndex % ROWS;

        // Lasketaan liikkumis suunta nykyisestä ruudusta.
        Vector3 moveVector = new Vector3(moveX - intX, 0.0f, moveY - intY);

        float modifiedMoveSpeed = 0;
        if (entityMoveSpeed > 0)
        {
            // Muunnetaan liikkumisnopeutta korkeimman vaikutuksen mukaan.
            modifiedMoveSpeed = entityMoveSpeed + Mathf.Abs(highestInfluence) * 5;
        }

        // Päivitetään Transform-komponentin sijaintia liikkumissuunnan ja nopeuden mukaan.
        t.position += moveVector * modifiedMoveSpeed * Time.deltaTime;
    }
    timer.Stop();
    movementUpdateTimeText.text = "Movement update time: " + timer.GetTimeStr();
}
```

Kuva 30. Olion sijainnin päivitys.

Myöhempiä suorituskykymittauksia varten metodin lopussa mitataan sen suorittamiseen kulunut aika ja asetetaan se käyttöliittymän tekstiobjektiin.

4.5 Monisäikeistäminen Job Systemillä

Insinöörityön tarkoituksena oli selvittää Unityn tarjoamien työkalujen soveltuvuus sovellusten optimointiin pääasiassa rinnakkaislaskennan kannalta. Koska vaikutuskartan laskeminen voi olla suhteellisen raskas operaatio, varsinkin isommilla kartoilla ja suurilla määrillä yksiköitä, siitä voi aiheutua suuriakin suorituskykyongelmia pelissä, jos sitä lasketaan useasti ja sen laskeminen tapahtuu pelin pääsäikeessä, jolloin muun koodin suoritus joutuu odottamaan. Jo 320 ruutua leveän ja 320 ruutua korkean kartan laskemiseen AMD Ryzen 5 2600-kuusiydinprosessorilla kesti keskimäärin noin 140 ms Mono-käännöstä käyttäen, kun kartalle oli aseteltu 16 vihreää eli positiivisen vaikutuksen yksikköä ja 18 punaista eli negatiivisen vaikutuksen yksikköä. Suorituskyky mitattiin käyttämällä C#:n Stopwatch-luokkaa, käynnistämällä kello juuri ennen CalculateInfluenceMap-metodin kutsumista ja pysäyttämällä välittömästi sen jälkeen, jolloin mukaan ei laskettu esimerkiksi yksiköiden generointia. Mukaan ei myöskään laskettu kartan visualisointia tekstuuriin.

Koska vaikutuskartan laskeminen on raskas operaatio, se on tärkeä optimointikohde. Koska vaikutuskartan ruudun laskeminen on suhteellisen riippumaton operaatio, se on hyvin rinnakkaistettavissa. Kartan ruutujen laskeminen voidaan jakaa useaan osaan, ja nämä osat voidaan ajaa useassa säikeessä usealla prosessoriytimellä rinnakkain säietason rinnakkaisuuden saavuttamiseksi. Koska vaikutuskartta koostuu suhteellisen yksinkertaisesta datasta, se ei myöskään tarvinnut Unityn luokkia sen toteuttamiseksi ja oli suhteellisen helppoa muuttaa Job Systemille soveltuvaan muotoon.

Vaikutuskartan laskemista varten luotiin kuvan 31 mukainen IJobParallelFor-rajapintaluokan toteuttava InfluenceMapJob.

```
[BurstCompile]
2 references
struct InfluenceMapJob : IJobParallelFor
{
    // Vaikutuskartta tallennetaan NativeArray-taulukkoon.
    public NativeArray<float> influenceMap;

    // Yksikködata välitetään kolmena NativeArray-taulukkona.
    [ReadOnly] public NativeArray<int> unitXCords;
    [ReadOnly] public NativeArray<int> unitYCords;
    [ReadOnly] public NativeArray<float> unitInfs;

    // Kutsutaan yhtä vaikutuskartan ruutua kohden.
    3 references
    public void Execute(int index)
    {
        // Muunnetaan yksiulotteinen taulukon indeksi x- ja y-koordinaateiksi.
        int x = index / ROWS;
        int y = index % ROWS;

        // Lasketaan ruudun vaikutus ja tallennetaan se vaikutuskarttaan käsiteltävään indeksiin.
        float influencePoint = CalculatePointInfluence(x, y);
        influenceMap[index] = influencePoint;
    }

    // Laskee ja palauttaa vaikutusarvon yhdelle kartan ruudulle.
    1 reference
    float CalculatePointInfluence(int col, int row)
    {
        float totalInfluence = 0.0f; // Kokonaisvaikutus.

        // Käydään läpi kaikki yksiköt ja summataan niiden vaikutus.
        for (int i = 0; i < unitInfs.Length; i++)
        {
            int x = unitXCords[i];
            int y = unitYCords[i];

            // Lasketaan yksikön etäisyys ruudusta.
            float dist = Distance(col, row, x, y);

            // Yksikön vaikutus ruutuun laskee etäisyyden mukaan.
            totalInfluence += unitInfs[i] / (1 + dist);
        }
        return totalInfluence;
    }
}
```

Kuva 31. Vaikutuskartan laskenta-Job.

Kuvan 31 Jobissa Execute-metodia kutsutaan jokaista vaikutuskartan alkiota kohden, kun Job ajoitetaan antamalla sille argumentiksi vaikutuskartan koko. Jobille annetaan myös argumentiksi yhden erän koko, joka määrää, kuinka ison osan kukin työsäie saa suoritettavaksi. Metodin saama taulukkoindeksi muutetaan laskettavan ruudun koordinaateiksi ja koordinaatit annetaan argumenteiksi metodille, joka laskee ruudun vaikutuksen. Ruudulle laskettu vaikutus tallennetaan samaan indeksiin vaikutuskarttataulukkoon.

Jobin käyttämä vaikutuskartta alustettiin valmiiksi oikeankokoiseksi InfluenceMap-skriptin Start-metodissa, ja yksikködatan luontia varten tehtiin uusi versio kuvassa 23 esitellystä metodista. Uusi metodi muokattiin käyttämään NativeArray-taulukkoja (kuva 32).

```
void GenerateRandomUnitDataNative(bool deterministic = false)
{
    Debug.Log("Generating random native unit data...");

    // Satunnaislukugeneraattorille voidaan asettaa ennalta valittu siemenluku deterministisen kartan luomiseksi.
    if (deterministic)
    {
        Random.InitState(deterministicSeed);
    }

    // Valitaan luotavien yksiköiden määrä satunnaisesti.
    int unitsGreen = Random.Range(minUnitsPerSide, maxUnitsPerSide);
    int unitsRed = Random.Range(minUnitsPerSide, maxUnitsPerSide);

    int totalUnits = unitsGreen + unitsRed;

    // Vapautetaan aikaisemmin varatut taulukot muistivuotojen välttämiseksi.
    unitXCordsNative.Dispose();
    unitYCordsNative.Dispose();
    unitInfsNative.Dispose();

    // Varataan uudet NativeArrayt yksiköiden määrän mukaan.
    NativeArray<int> xCords = new NativeArray<int>(totalUnits, Allocator.Persistent);
    NativeArray<int> yCords = new NativeArray<int>(totalUnits, Allocator.Persistent);
    NativeArray<float> infs = new NativeArray<float>(totalUnits, Allocator.Persistent);

    // Asetetaan vapautetut osoittimet osoittamaan uusiin taulukkoihin.
    unitXCordsNative = xCords;
    unitYCordsNative = yCords;
    unitInfsNative = infs;
```

Kuva 32. Yksikködatan luomiseen käytetyn metodin muokattu osio.

Koska uusi metodi käyttää NativeArrayita listojen sijasta ja NativeArrayt eivät ole muistihallittuja taulukoita, pitää aikaisemmin varatut taulukot vapauttaa muistista manuaalisesti muistivuotojen välttämiseksi. Tämän jälkeen varataan uudet taulukot yksiköiden määrän mukaan ja asetetaan vanhat vapautetut taulukko-osoittimet osoittamaan uusiin taulukoihin.

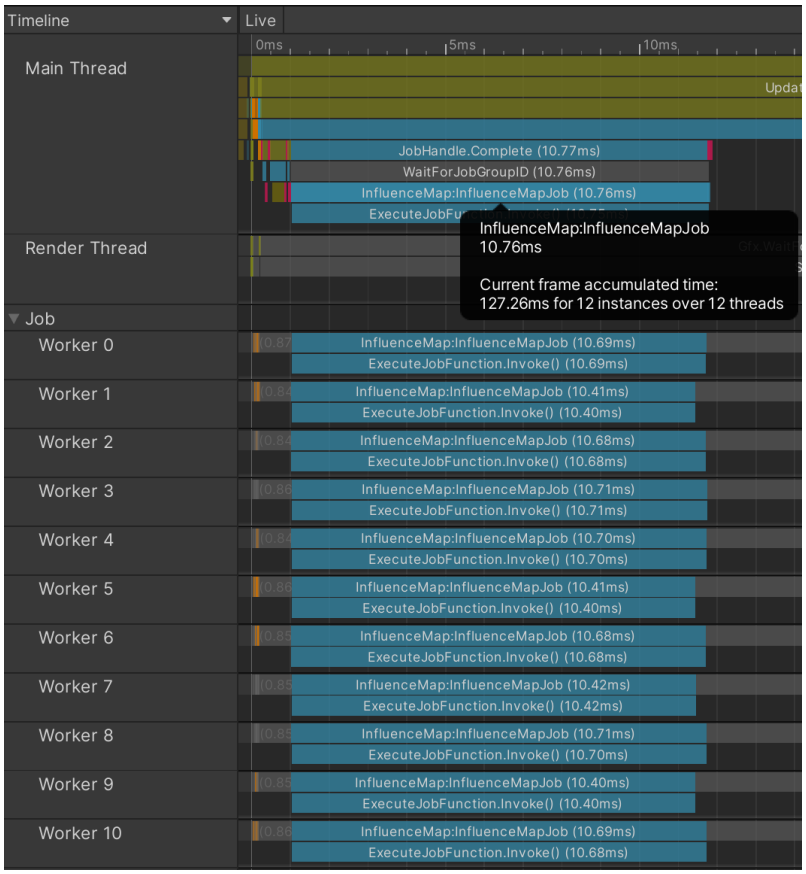
Jobin käyttämistä varten luotiin kuvassa 33 näkyvä metodi, jota voitiin kutsua InfluenceMap-skriptin Update-metodista näppäintä painamalla.

```
public void InfluenceMapJobTest()
{
    // Generoidaan satunnainen yksikködata.
    GenerateRandomUnitDataNative(false);

    Debug.Log("Starting a new influenceMapJob");
    influenceMapTimer.Restart();

    // Jobin luonti.
    influenceMapJob = new InfluenceMapJob
    {
        influenceMap = influenceMapNative,
        unitXCords = unitXCordsNative,
        unitYCords = unitYCordsNative,
        unitInfs = unitInfsNative
    };

    // Ajoitetaan Jobi suoritettavaksi.
    influenceMapJobHandle = influenceMapJob.Schedule(ROWS * COLS, ROWS);

    // Varmistetaan Jobin valmistuminen.
    influenceMapJobHandle.Complete();

    influenceMapTimer.Stop();
    timerText.text = "Influence map time: " + influenceMapTimer.GetTimeStr();

    // Merkitään minkä tyyppista dataa käytetään, EntityManageria varten.
    useNativeMap = true;

    // Kopioidaan juuri laskettu kartta toiseen taulukkoon,
    // jotta sitä voidaan käyttää mahdollisessa tilanteessa,
    // jossa uuden kartan laskeminen on kesken.
    influenceMapNative.CopyTo(influenceMapNativePrev);

    // Visualisoidaan kartta tekstuuriin.
    RenderNativeMapToTexture(influenceMapNativePrev);
}
```

Kuva 33. InfluenceMapJobin käyttömetodi.

Metodissa luodaan ensiksi yksikködata NativeArrayhin, jotka välitetään Jobille jäsenmuuttujina. InfluenceMapJob-tietue ja JobHandle oli valmiiksi luotu InfluenceMap-skriptin jäsenmuuttujiksi. Uuden Job-tietueen instanssin luonnin jälkeen se ajoitettiin suoritettavaksi ja argumenteiksi annettiin vaikutuskarttataulukon koko ja yhden säikeen suorittaman erän koko. Tässä tapauksessa Job haluttiin suorittaa välittömästi ja sen takia Complete-metodia kutsuttiin heti ajoittamisen jälkeen.

Unity Profilerista nähdään, miten InfluenceMapJob suoritetaan 12 säikeellä rinnakkain. Kuvassa 34 nähdään, miten pääsäie ja 11 työsäiettä jakavat InfluenceMapJobin suorituksen.



Kuva 34. Unity Profilerin näkymä InfluenceMapJobin suorituksesta 12 säikeellä.

Vaikutuskartan laskemisen lisäksi myös kartalla olevien olioiden liikkuminen monisäikeistettiin Job Systemillä. Olioiden liikkumista varten EntityManager-skriptiin luotiin IJobParallelForTransfom-luokkarajapinnan toteuttava kuvissa 35 ja 36 nähtävä EntityMovementJob. Jobille laitettiin jäsenmuuttujiksi taulukko olioiden puolista ja vaikutuskartta. Lisäksi luotiin muita apumuuttujia (kuva 35).

```
[BurstCompile]
2 references
struct EntityMovementJob : IJobParallelForTransform
{
    [ReadOnly] public NativeArray<bool> entitySides;
    [ReadOnly] public NativeArray<float> influenceMap;
    [ReadOnly] public NativeArray<int> offsets;
    [ReadOnly] public int ROWS;
    [ReadOnly] public int COLS;
    [ReadOnly] public float moveSpeed;
    [ReadOnly] public float deltaTime;
```

Kuva 35. EntityMovementJobin jäsenmuuttujat.

EntityMovementJobin Execute-metodi sai parametreina käsiteltävän olion indeksin ja sen Transformin TransformAccess-muodossa. Jobin Execute-metodi oli hieman muokattu versio aiemmin esitellystä EntityMovement-metodista. Kuvassa 36 on EntityMovementJobin Execute-metodin alku.

```
// Liikuttaa parametrina saatua Transformia.
0 references
public void Execute(int index, TransformAccess transform)
{
    float x = transform.position.x;
    float y = transform.position.z;

    // Pyöristetaan koordinaatit kokonaisluvuiksi vaikutuskartan indeksointia varten.
    int intX = Mathf.RoundToInt(x);
    int intY = Mathf.RoundToInt(y);

    // Haetaan olion puoli.
    bool side = entitySides[index];

    int moveIndex = intX * ROWS + intY; // Vakiona liikutaan nykyiseen ruutuun.

    // Asetetaan korkein vaikutus nykyisen ruudun mukaan.
    float highestInfluence = influenceMap[moveIndex];
```

Kuva 36. EntityMovementJobin Execute-metodin alku.

Transform-komponentteja ei voida suoraan antaa Jobille, vaan ne pitää muuntaa TransformAccess-taulukoksi. Kuvassa 37 on esitetty, miten edellä esitelty TransformAccessArray alustetaan listalla Transform-komponetteja.

```
transformAccessArray = new TransformAccessArray(entityTransformList.Count);
transformAccessArray.SetTransforms(entityTransformList.ToArray());
```

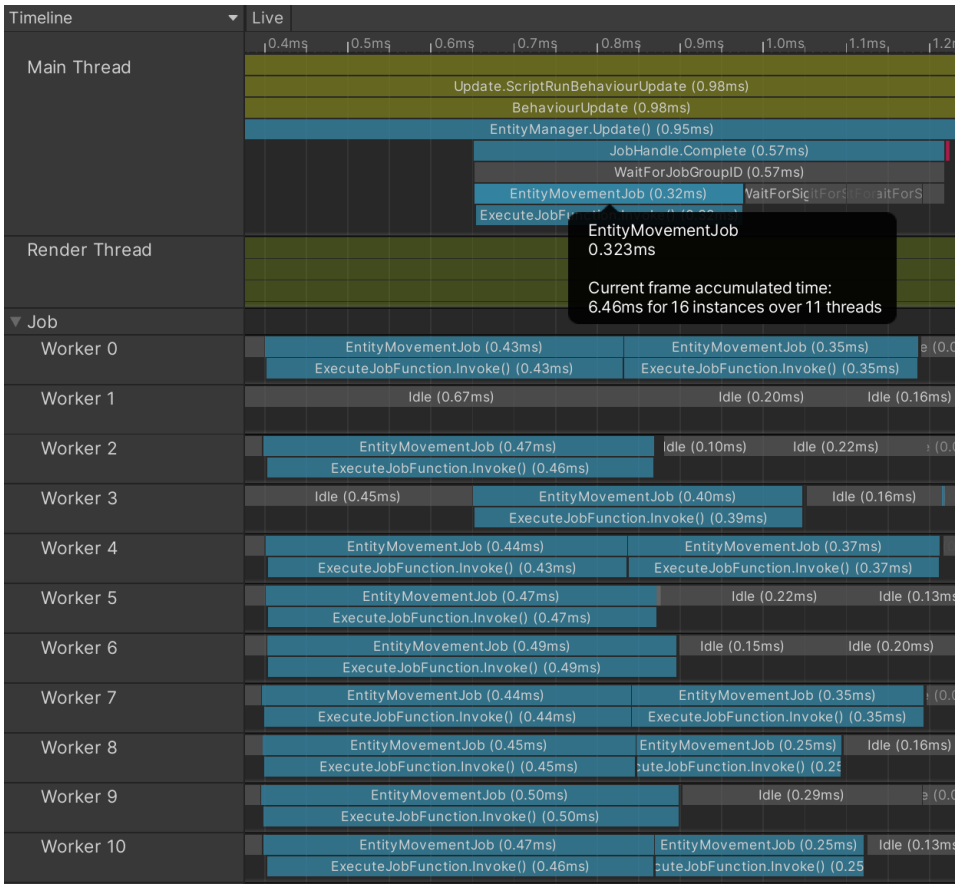
Kuva 37. TransformAccessArrayn alustaminen listallisella Transform-komponentteja.

Kuvassa 38 näkyy, miten IJobParallelTransform-rajapintaluokan toteuttava Job luodaan tavallisen Jobin tavoin mutta, TransformAccessArray annetaan Jobille argumenttina sitä ajoitettaessa.

```
entityMovementJob = new EntityMovementJob
{
    entitySides = entitySidesArray,
    influenceMap = this.influenceMap,
    COLS = this.COLS,
    ROWS = this.ROWS,
    offsets = offsetArray,
    moveSpeed = entityMoveSpeed,
    deltaTime = Time.deltaTime
};

// Jobin ajoittaminen suoritettavaksi.
movementJobHandle = entityMovementJob.Schedule(transformAccessArray);
```

Kuva 38. IJobParallelForTransform-Jobin luonti ja ajoitus.

Kuvan 39 Unity Profilerin näkymästä nähdään, että EntityMomeventJob suoritettiin 11 säikeellä rinnakkain.



Kuva 39. Unity Profilerin näkymä EntityMovementJobin suorituksesta.

TransformAccess tarjoaa turvallisen rajapinnan Tranform-komponenttien käsittelyyn suoraan Jobeista ja näin muuta käsittelyä ei tarvita (38).

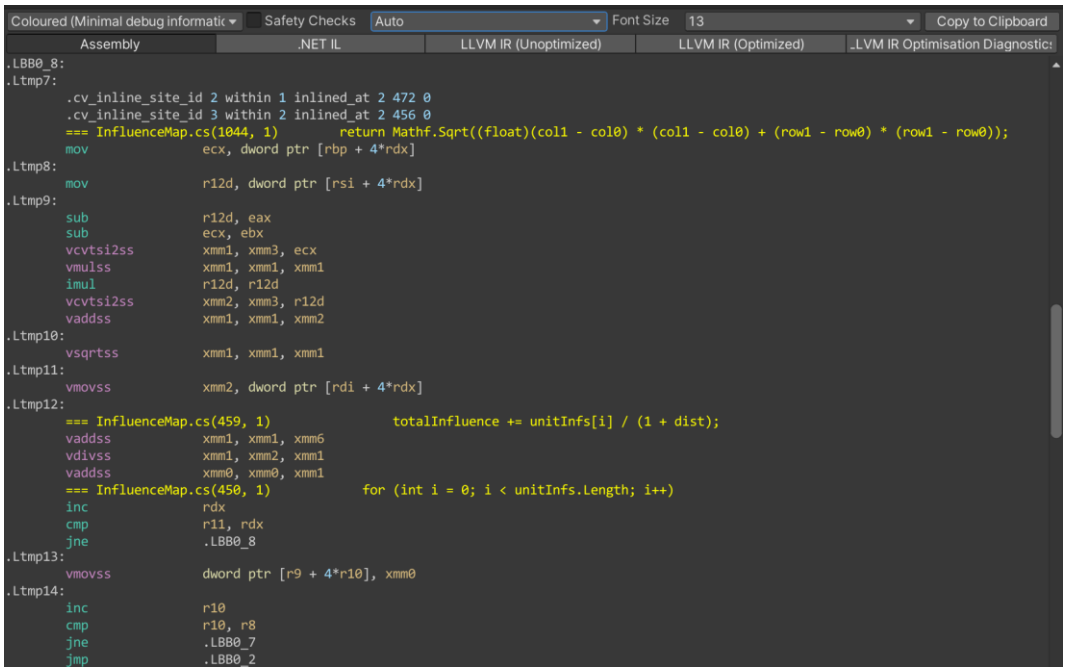
4.6 Burst-kääntäjän optimoinnit

Job Systemin monisäikeistämisen lisäksi testattiin Burst-kääntäjän mahdollistamia suorituskykyparannuksia Mono- ja IL2CPP-käännöksiin verrattuna. Vaikutuskartan laskemisessa tärkein optimointi, jota haluttiin testata, oli se, kuinka hyvin Burst pystyi käyttämään vektorisointia. Lisäksi testattiin liukulukuoptimointien vaikutukset. Burstin käyttöönotto oli yksinkertainen prosessi, sillä kun Burst oli asennettu Package Managerista ja Burstia käyttävät skriptit merkitty käyttämään Unity.Burst-nimiavaruutta, sitä käyttävät Jobit tarvitsi vain merkitä Burst-Compile-attribuutilla kuvassa 40 esitetyllä tavalla, sisällyttämällä halutessa liukulukuoptimoinnit.

```
[BurstCompile]
2 references
struct InfluenceMapJob : IJobParallelFor
{
```

Kuva 40. InfluenceMapJob merkitty käyttämään Burst-kääntäjää.

Kun InfluenceMapJob merkittiin BurstCompile-attribuutilla ja Burst Inspectorista tutkittiin sen tuottamaa konekielistä koodia, kun turvatarkistukset oli kytketty pois päältä, huomattiin, että koodi ei hyödyntänyt SIMD-käskyjä lainkaan. Kuten kuvassa 41 nähdään, kaikki konekieliset käskyt operoivat skalaariarvoilla. Tämän huomaa esimerkiksi siitä, miten useat Burst Inspectorissa näkyvät käskyt päättyvät ”-ss”-kirjaimiin, joka merkitsee ”scalar single”, eli käskyt operoivat skalaaria perustarkkuuden liukulukua. Burst ei myöskään antanut minkäänlaista ilmoitusta, että koodia ei ollut vektorisoitu.



Kuva 41. Vaikutuskartan laskennan vektorisoimaton konekielinen koodi.

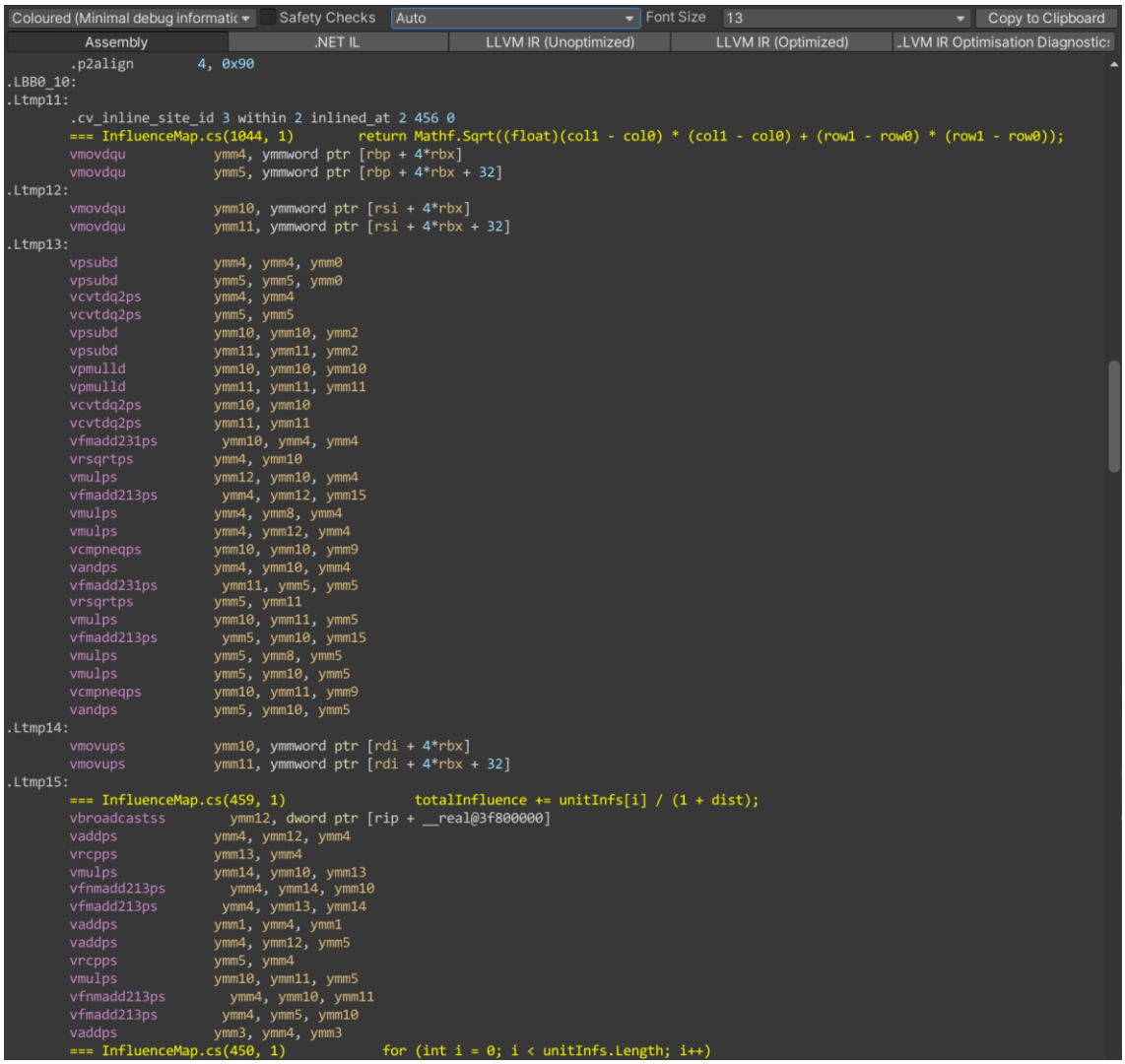
Koska kääntäjien automaattisesta vektorisoinnista oli aikaisempaa kokemusta, kokeiltiin InfluenceMapJobille lisäksi FloatMode.Fast-asetusta, koska joissain

tapauksissa automaattinen vektorisointi onnistuu kääntäjiltä paremmin, kun liukulukulaskujen järjestystä voidaan muuttaa. Kun Job merkittiin kuvan 42 mukaisesti, tuotti Burst vektorisoidun SIMD-käskyjä käyttävän koodin.

```
[BurstCompile(FloatPrecision.Standard, FloatMode.Fast)]
2 references
struct InfluenceMapJob : IJobParallelFor
{
```

Kuva 42. InfluenceMapJob merkitty Burst-käännettäväksi FloatMode.Fast-asetuksella.

Kuvassa 43 näkyy, miten useat käskyistä päättyvät ”-ps”-kirjaimiin, joka merkitsee ”packed singles” eli pakattuja perustarkkuuden liukulukuja. Koska käskyt operoivat 256-bittisillä ymm-rekistereillä, tämä tarkoitti sitä, että käsiteltiin kahdeksaa 32-bittistä liukulukua yhdellä operaatiolla.



Kuva 43. Burst-käännetty vektorisoitu SIMD-käskyjä hyödyntävä InfluenceMapJobin konekielinen koodi.

SIMD-käskyjen hyödyntämisen lisäksi Burst onnistui osittain purkamaan silmukan (engl. loop unroll). Tämä tarkoittaa sitä, että yhdellä silmukan toistolla tehdään useampi silmukan operaatio. Tämä nopeuttaa koodin suoritusta, sillä silmukan toistoa hallitsevaa osaa tarvitsee kutsua harvemmin. Kuvassa 43 näkyy, miten yksi silmukan toisto käsittelee kahta erillistä operaatiota. Tämän huomaa selvimmin siitä, miten tietty käskysarja toistetaan kaksi kertaa peräkkäin eri rekistereillä. Burst onnistui myös pääsemään eroon hitaista jakolaskuista ja neliöjuurista korvaamalla ne käänteislukujen kertomisella ja käänteisneliöjuuren kertomisella.

4.7 Suorituskykymittaukset

Koska insinöörityön tarkoituksena oli selvittää, kuinka hyvin Unityn tarjoamat rinnakkaislaskennan työkalut soveltuvat sovellusten optimointiin käytännössä, oli tärkeää tehdä selvät suorituskykymittaukset siitä, kuinka paljon niiden käytöstä voidaan hyötyä. Suorituskykytesteissä testattiin sitä, kuinka paljon nopeammin työssä implementoidut ominaisuudet saatiin suoritettua, kun ne oli monisäikeistetty Job Systemillä, ja kuinka paljon hyötyä Burst-kääntäjästä oli. Vertauksissa testattiin myös suorituskykyero käytettäessä Monoa ja IL2CPP:tä. Käytännössä testattiin sitä, kuinka nopeasti vaikutuskartta saatiin laskettua vakion kokoisella 320 ruutua leveällä ja 320 ruutua korkealla kartalla ja kuinka kauan 10 000 olion liikkumisen simulointiin kului aikaa päivitystä kohden. Burst-kääntäjän tuottamaa versiota vaikutuskartan laskennasta verrattiin lisäksi käsin vektorisoituun versioon Unity.mathematics-kirjastoa käyttäen.

Testikokoonpanossa oli vakiokellotaajuudella toimiva AMD Ryzen 5 2600-kuusiydinprosessori, joka tuki SMT:tä ja AVX-käskykantalaajennosta. Kaikki suorituskykytestit tehtiin valmiiksi kootulla versiolla sovelluksesta, ja suoritusajat mitattiin C#:n Stopwatch-luokalla. Vaikutuskartan laskennasta tehtiin 20 peräkkäistä otosta, joista laskettiin keskiarvo. Olioiden liikkeen simuloiminen testattiin luomalla kartalle 10 000 oliota ja laskemalla seuraavasta 1000 simulaatioiteraatiosta yhteen simulaatioon keskimäärin kulunut aika.

Vaikutuskartan laskenta

Vaikutuskartan laskemisen testaus tehtiin siten, että InfluenceMap-skriptin Update-metodissa voitiin kutsua haluttua vaikutuskartan laskentametodia. Vaikutuskartan laskentaa verrattiin neljän eri tavan välillä. Tavallista kuvassa 44 esiintyvää CalculateInfluenceMap-metodia verrattiin Job Systemiä käyttävään monisäikeistettyyn IJobParallelFor-versioon, manuaalisesti vektorisoituun IJobParallelFor-versioon ja lisäksi yhdellä säikeellä ajettavaan IJob-versioon pelkän Burst-kääntäjän vaikutuksen testaamiseksi. Suoritusajan yhtenäistä testaamista

varten satunnaislukugeneraattori alustettiin ennalta määrätyllä siemenluvulla 118355416, jotta sama kartta luotaisiin joka testauskerralla.

Suoritusaika mitattiin käyttämällä C#:n Stopwatch-luokkaa laskien mukaan vain vaikutuskartan laskeminen taulukkoon kuvan 44 mukaisesti.

```
influenceMapTimer.Restart();
CalculateInfluenceMap();
influenceMapTimer.Stop();
```

Kuva 44. CalculateInfluenceMap-metodin suoritusajan mittaaminen.

Jobien tapauksessa mukaan laskettiin myös Jobien luonti ja ajoitus (kuva 45). Kaikille Jobeille asetettiin erän kooksi vaikutuskartan rivien määrä, joka testeissä käytetyn kartan tapauksessa oli 320.

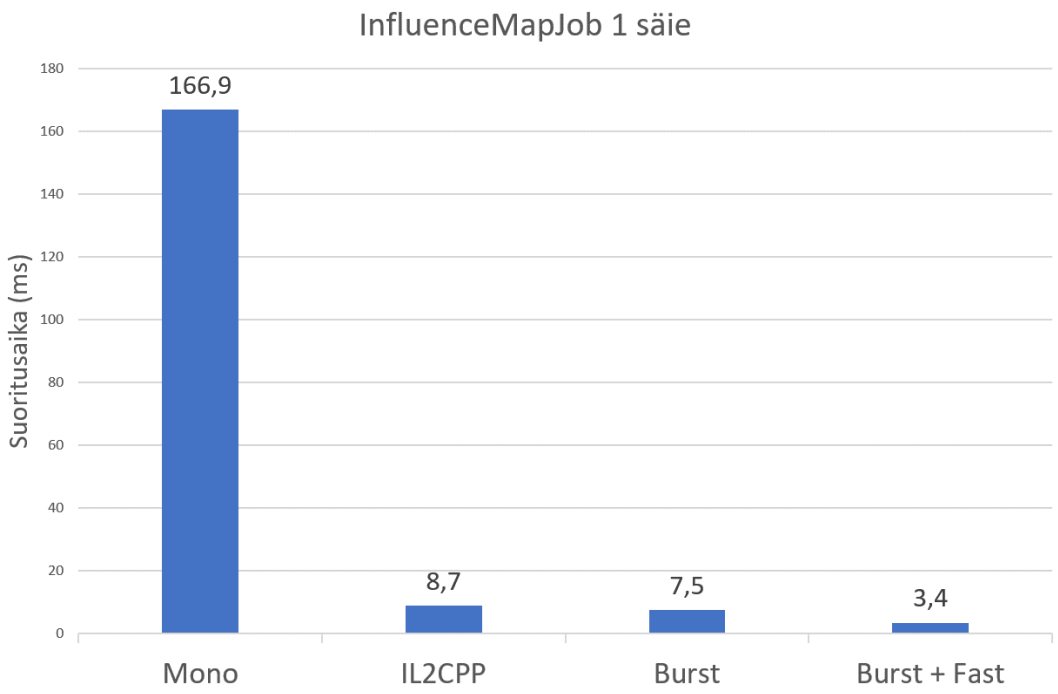
```
influenceMapTimer.Restart();

// Jobin luonti.
influenceMapJob = new InfluenceMapJob
{
    influenceMap = influenceMapNative,
    unitXCords = unitXCordsNative,
    unitYCords = unitYCordsNative,
    unitInfs = unitInfsNative
};

// Ajoitetaan Jobi suoritettavaksi.
influenceMapJobHandle = influenceMapJob.Schedule(ROWS * COLS, ROWS);

// Varmistetaan Jobin valmistuminen.
influenceMapJobHandle.Complete();

influenceMapTimer.Stop();
```

Kuva 45. Jobien suoritusajan mittaaminen.

Kaikki vaikutuskartan laskentatestit ajettiin 20 kertaa peräkkäin metodia kohden, ja niistä laskettiin keskiarvo.

Yhdellä säikeellä ajettavan CalculateInfluenceMap-metodin suoritusajaksi mitattiin 139,5 millisekuntia käytettäessä Mono-käännöstä. Suoritusaika laski 12,3

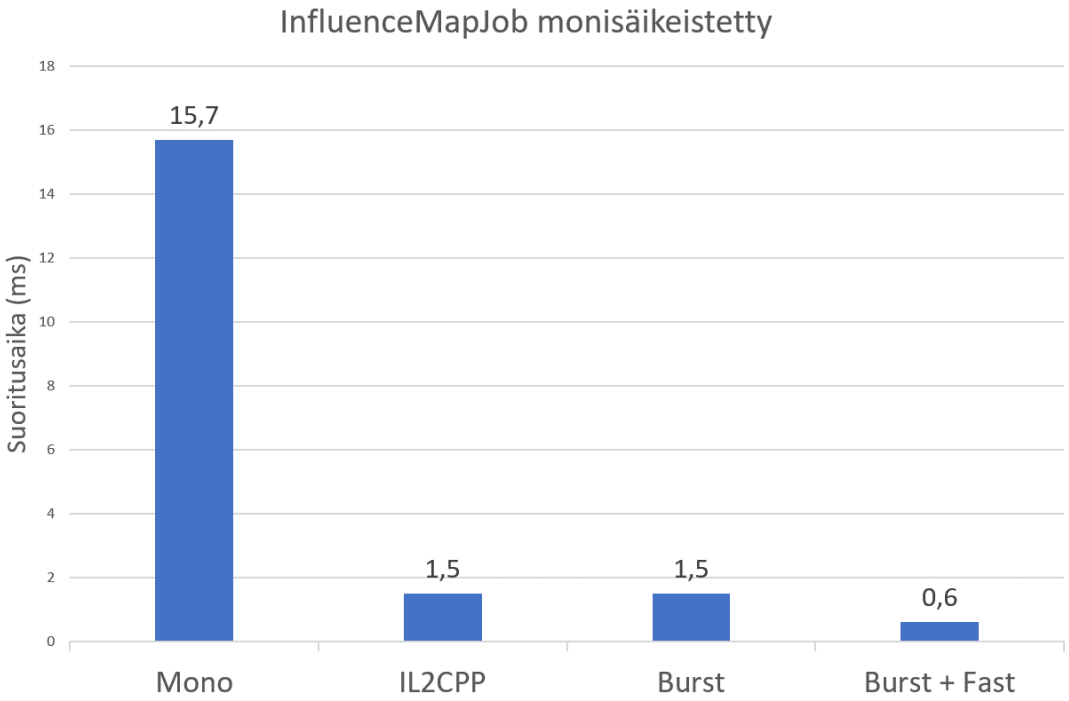
millisekuntiin, kun vaihdettiin IL2CPP-käännökseen, jolloin suorituskyky parani 11,3-kertaiseksi. Kuvan 46 kuvaajassa on vertailu eri käännösasetusten välillä, kun yksisäikeisestä algoritmista luotiin Job Systemiä käyttävä versio.



Kuva 46. Yksisäikeisen Job Systemiä käyttävän vaikutuskartan laskenta-algoritmin suorituskykyvertailut eri käännösasetusten välillä.

Job Systemiä käyttävä yhden säikeen versio Mono-käännöksellä oli noin 20 % hitaampi kuin ilman Job Systemiä. Suorituskyky parani IL2CPP-käännöstä käyttäen noin 19-kertaiseksi Mono-käännökseen verrattuna ja oli tällöin noin 31 % nopeampi kuin Job Systemitön IL2CPP-versio. Kun Burst otettiin käyttöön, suoritusaika parani noin 22,3-kertaiseksi Job Systemiä käyttävään Mono-käännökseen verrattuna ja 14 % Job Systemiä käyttävään IL2CPP-käännökseen verrattuna. Kuten aikaisemmin todettu, myöskään tämä Job ei vektorisoitunut, ennen kuin käytettiin FloatMode.Fast-asetusta, jolloin suorituskykyero Mono-käännökseen kasvoi noin 49-kertaiseksi ja vakio-Burst-käännökseen 2,2-kertaiseksi. Tällöin yhden säikeen Job Systemiä käyttävä versio oli tavallista yksisäikeistä versiota 41 kertaa suorituskykyisempi.

Monisäikeistetty Job Systemiä käyttävä versio testattiin samoilla käännösasetuksilla kuin yksisäikeinen versio (kuva 47). Monisäikeistetty Mono-käännös oli lähes 9 kertaa nopeampi yksisäikeiseen Mono versioon verrattuna, mutta 4,6 kertaa hitaampi kuin nopein yhden säikeen Job Systemiä käyttävä käännös. Monisäikeistetty versio oli tällöin myös 28 % yksisäikeistä IL2CPP-käännöstä hitaampi. Kun Burst otettiin käyttöön, parani suorituskyky 10,5-kertaiseksi Mono-käännökseen verrattuna. Myös käyttämällä IL2CPP:tä päästiin samaan tulokseen. Suorituskyky oli tällöin 5-kertainen vastaavaan yksisäikeiseen versioon verrattuna ja 2,3-kertainen nopeimpaan yhden säikeen versioon verrattuna.



Kuva 47. Monisäikeistetyn Job Systemiä käyttävän vaikutuskartan laskennan suorituskykyvertailut eri käännösasetuksilla.

Kuten luvussa 4.6 todettiin, monisäikeistetty Job vektorisoitui vasta, kun otettiin käyttöön FloatMode.Fast-asetus, jolloin suorituskyky kasvoi 2,5-kertaiseksi tavalliseen Burst-käännökseen verrattuna. Tämä oli nopein saavutettu suoritusaika vaikutuskartan laskemiselle, jolloin suoritusaika oli parantunut alkuperäisestä yksisäikeisestä Mono-version 139,5 ms:sta 0,6 ms:iin, jolloin suorituskyky

oli 232,5-kertainen. Ero yksisäikeiseen IL2CPP-käännökseen verrattuna oli 20,5-kertainen.

Manuaalisesti vektorisoidulla monisäikeistetyllä versiolla päästiin suunnilleen samoihin suorituskykylukemiin monisäikeistetyn version kanssa, mutta tällöin nopein suoritusaika ei vaatinut FloatMode.Fast-asetusta. FloatPrecision.Low-asetuksella ei ollut vaikutusta Burst-käännösten suorituskykyyn.

Olioiden liikkeen simulointi

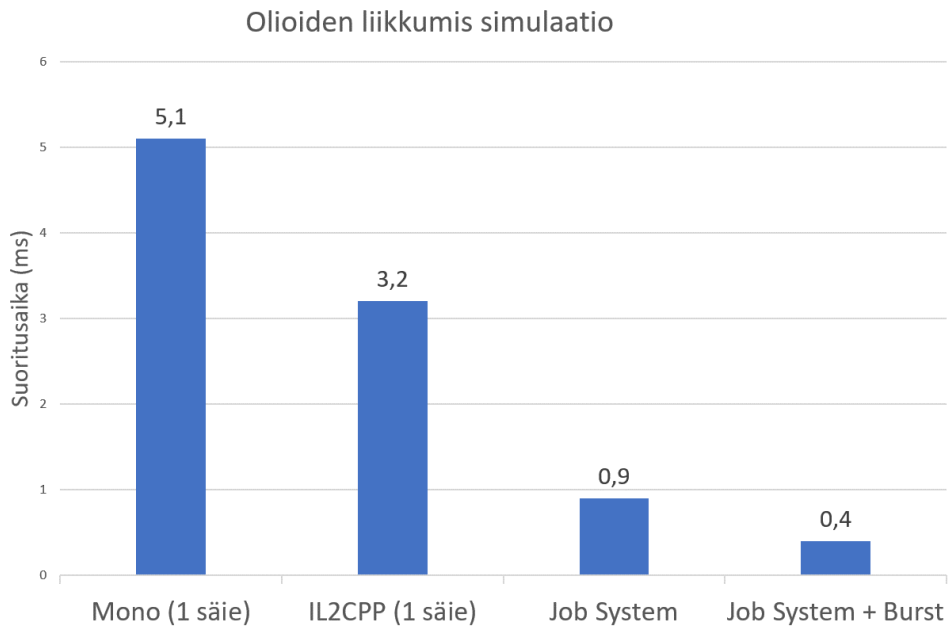
Olioiden liikettä simuloivaa metodia kutsuttiin EntityManagerin joka kehyksellä kutsuttavassa Update-funktiossa. Suorituskykymittauksia varten kutsuttiin joko tavallista tai Job Systemiä käyttävää metodia. Suoritusaikaan mitattiin koko metodissa kulunut aika. Job Systemiä käyttävän implementaation tapauksessa metodiin sisältyi tarkistus olioiden määrän muuttumisesta ja uuden TransformAccessArrayn luomisesta tarvittaessa (kuva 48).

```
timer.Restart();

if(entitiesChanged)
{
    if (entitySidesArray.IsCreated)
    {
        entitySidesArray.Dispose();
    }

    entitySidesArray = new NativeArray<bool>(entitySidesList.ToArray(), Allocator.Persistent);

    if (transformAccessArray.isCreated)
    {
        transformAccessArray.Dispose();
    }

    transformAccessArray = new TransformAccessArray(entityTransformList.Count);
    transformAccessArray.SetTransforms(entityTransformList.ToArray());

    entitiesChanged = false;
}
```

Kuva 48. Job Systemiä käyttävän oliosimulaatiometodin alku.

Suoritusaika mitattiin luomalla kartalle 10 000 oliota ja laskemalla keskiarvo seuraavasta tuhannesta simulaatioiteraatiosta. Myös tätä testiä varten satunnaislukugeneraattori oli alustettu samalla siemenluvulla 118355416.

Keskimääräinen 10 000 tuhannen olion liikkumisen simulaatioiteraation aika oli yksisäikeisellä Mono-käännöksellä 5,1 ms (kuva 49). IL2CPP-käännöstä käytettäessä suoritusaika parani 37 %.



Kuva 49. 10 000 olion liikkumissimulaation keskimääräinen suoritusaika eri versioiden välillä.

Kun simulaatio monisäikeistettiin Job Systemillä, oli suoritusaika 5,7 kertaa nopeampi yksisäikeiseen Mono-versioon verrattuna. Kun Burst otettiin käyttöön, suorituskykyero kasvoi 12,8-kertaiseksi. FloatMode.Fast-asetuksella ei ollut vaikutusta suoritusaikaan.

4.8 Suorituskykymittausten arviointi

Vaikutuskartan suorituskykytesteissä suurin yksittäinen suorituskykyparannus saavutettiin yllätyksellisesti pelkästään vaihtamalla Mono-käännöksestä Il2CPP-käännökseen. Yksisäikeisen CalculateInfluenceMap-metodin tapauksessa suorituskyky parani yli 11-kertaiseksi. Job Systemiä käyttävässä yksisäikeisessä versiossa suorituskyky ero Mono- ja IL2CPP-käännösten välillä oli vielä suurempi, noin 19-kertainen. Samankaltaisia eroja huomattiin myös Job Systemiä käyttävässä monisäikeistetyssä versiossa, jossa Mono- ja IL2CPP-käännösten suorituskykyero oli yli 10-kertainen. Insinöörityötä aloittaessa ei ollut tiedostettu, että näinkin suuria suorituskykyeroja näiden kahden käännöstavan välillä voisi olla, ja sen takia näin suuret suorituskykyerot olivat yllätyksellisiä.

Insinöörityössä ei paneuduttu tarkemmin siihen, miten IL2CPP toimii ja mistä sen tarjoamat suorituskykyparannukset johtuvat, mutta oletettavasti tiettyä kohdealustaa varten ennenaikaisesti käännettäessä voidaan optimoida ainakin insinöörityön testisovelluksen kohdalla huomattavasti paremmin, ajonaikaiseen Mono-käännökseen verrattuna. Näin suuria suorituskykyeroja ei nähdä kaikissa sovelluksissa, kuten huomattiin olioiden liikesimulaation suorituskykymittauksissa, mutta laajempi selvitys IL2CPP- ja Mono-käännösten eroista voisi olla mahdollinen lisäselvityskohde.

Burst-kääntäjää käytettäessä ilman lisäoptimointeja päästiin vähintään IL2CPP-käännöksen tasolle ja joissain tapauksissa hieman paremmaksikin. Vaikutuskartan laskennan koodi ei vektorisoitunut, ennen kuin otettiin käyttöön FloatMode.Fast-asetus, jolloin saavutettiin 2,2–2,5-kertainen suorituskykyparannus vakio-Burst-käännökseen verrattuna. Tämä vektorisoinnista saatu hyöty oli merkittävä, mutta ei päässyt lähelle odotettuja 4–8-kertaisia suorituskykyparannuksia, joita voidaan olettaa hyvin vektorisoituvalta sovellukselta, jota suoritetaan prosessorilla, joka tukee AVX-käskykantalaajennosta ja mahdollistaa näin neljän tai jopa kahdeksan liukuluvun käsittelemisen rinnakkain. Burst-käännöksellä ja FloatMode.Fast-asetuksella vaikutuskartan laskentaan kuluva aika oli 3,4

millisekuntia, jolloin suorituskyky oli 41-kertainen alkuperäiseen laskentatapaan verrattuna.

Jotta alkuperäinen 139,5 millisekuntia kestävä vaikutuskartan laskeminen saataisi tehtyä pelin aikana, pitäisi sen suoritus jakaa hyvin usealle kehykselle, jotta muulle koodille jäisi aikaa suoritettavaksi. Job Systemiä ja Burst-kääntäjää käyttävä yksisäikeinen versio saatiin tasolle, jossa sitä voitaisiin mahdollisesti kutsua jopa joka kehyksellä, muuta koodia hidastamatta. Tällöin jäisi myös vapaita työsäikeitä muiden mahdollisten Jobien suorittamiseen.

Monisäikeistämällä vaikutuskartan laskeminen saavutettiin 5–5,6-kertainen suorituskykyparannus Burst-käännettyjä versioita vertailtaessa. Tämä kertoo, että päästiin lähelle lineaarista suorituskykyparannusta testiprosessorilla käytettävissä olevien kuuden ytimen mukaan. Vaikutuskartan laskemiseen kulunut aika saatiin alennettua parhaimmillaan 0,6 millisekuntiin, joka mahdollistaisi useamman laskun tekemisen kehystä kohden tai laajemman kartan laskemisen. Tässä tapauksessa kaikki työsäikeet olivat kuitenkin käytössä ja näin veisivät suoritusaikaa muilta mahdollisesti samaan aikaan suoritettavilta Jobeilta. Kaiken kaikkiaan Job Systemiä ja Burst-kääntäjää käytettäessä vaikutuskartan laskennan suorituskykyä saatiin parannettua alkuperäisestä versiosta 232,5-kertaiseksi, joka kertoo siitä, miten paljon suorituskykyä voidaan saada kasvatettua, kun ohjelma on hyvin optimoitavissa ja se optimoidaan ottamalla hyödyksi kohdealustan ja laitteiston ominaisuudet.

Olioiden liikesimulaatiossa IL2CPP- ja Mono-käännösten välillä ei ollut yhtä suurta eroa kuin vaikutuskartan laskennassa. Ero yksisäikeisillä versioilla oli 37 %, joka on silti merkittävä parannus. Kun olioiden liikesimulaatio monisäikeistettiin, parani suorituskyky taas lähes lineaarisesti saatavilla olevien kuuden prosessoriytimen mukaan noin 5,7-kertaiseksi. Burst-kääntäjän käyttöönotto kasvatti suorituskyvyn vielä 2,3-kertaiseksi Burst-kääntämättömään versioon verrattuna. Vaikkei olioiden liikesimulaatio pystynytkään hyödyntämään SIMD-prosessointia, saatiin Burstia käytettäessä silti huomattava

suorituskykyparannus. Tämä kertoo siitä, miten Burst ei ole pelkästään parantamassa koodin vektorisointia, vaan pystyy tekemään muitakin merkittäviä optimointeja.

4.9 Työn tulosten arviointi

Unityn Job Systemin ja Burst-kääntäjän tutkiminen onnistui melko hyvin. Työssä saatiin hyvin selvitettyä ja dokumentoitua Job Systemin toimintatapa ja käyttöönotto tasolla, joka oli riittävä Unity-pelikehittäjälle Job Systemin tehokkaaksi hyödyntämiseksi ainakin joissain peleissä, vaikka aikaisempaa kokemusta monisäikeisen pelikoodin kirjoittamisesta ei juurikaan ollut. Job Systemiä saatiin tehokkaasti hyödynnettyä testisovelluksen monisäikeistämisessä, jolloin kaikki prosessoriytimet saatiin haluttaessa valjastettua sovelluksen suoritusta merkittävästi nopeuttamaan. Job Systemin tarjoama hyöty saatiin selvästi todistettua suorituskykytesteillä.

Vaikka Job Systemiä saatiin hyödynnettyä erittäin tehokkaasti insinöörityön testisovelluksessa, ei sovellus kuitenkaan vastannut käytännön peliä, ja tämän takia Job Systemin integroinnin helppous ja siitä saatu hyöty insinöörityössä voivat olla hieman harhaan johtavia, minkä takia käytännön hyöty ja käyttöönoton helppous voivat olla rajoittuneempia varsinaisissa Unityllä kehitettävissä peleissä. Job Systemin kattavampi arviointi vaatisi testaamista varsinaisissa peleissä.

Burst-kääntäjän toimintatapa selvisi pääpiirteittäin, ja sitä saatiin hyödynnettyä melko hyvin Job Systemin tukena. Burst-kääntäjän käyttöönotto oli erittäin helppoa ja yksinkertaista, ja sen optimointiominaisuuksia saatiin hyödynnettyä suorituskykyä usein merkittävästi parantavasti. Burstin muita ominaisuuksia saatiin myös hyödynnettyä. Esimerkiksi Burst Inspector osoittautui tärkeäksi työkaluksi, kun haluttiin tutkia Burstin luomaa konekielistä koodia optimointien tarkistamiseksi. Testisovelluksen suorituskykymittauksilla saatiin selvästi todistettua Burst-kääntäjän optimoinneista saatu, usein hyvinkin merkittävä hyöty. Suorituskykymittauksia tehdessä selvisi myös yllättäen Unityn kahden vakiona

tarjoaman skriptialustan, Monon ja IL2CPP:n, joissain tapauksissa suuretkin suorituskykyerot. Näiden kahden käännöstavan erojen tarkempi selvitys voisi olla aiheellinen.

Vaikka Burst-kääntäjän toiminta selvisi pääpiirteittäin, jäi sen ja muiden optimoivien kääntäjien toiminta vielä vaille syvempää ymmärrystä, joka voisi olla tärkeää Burstin tehokkaammaksi hyödyntämiseksi. Tosin Burst-kääntäjää yleensä esitetään ilmaisena suorituskykylisänä, jonka voi vain kytkeä päälle, ilman että sen toimintaa tarvitsee syvemmin ymmärtää. Burst tarjoaa käytettyjen optimointien lisäksi laajan määrän muita optimointitapoja ja asetuksia, jotka jäivät tutkimatta.

5 Yhteenveto

Koska uusien pelien odotetaan jatkuvasti tarjoavan yhä laajempia ja yksityiskohtaisempia kokemuksia, vaatii niiden suoritus yhä enemmän tehoa laitteistolta. Vaikka laitteiston tehon kasvaminen onkin jatkunut tähän päivään asti, eivät suorituskykyparannukset enää ole aivan niin yksiselitteisiä ja yleisiä, kuin ennen voitiin olettaa. Nykyään prosessorien tehokas käyttö vaatii, että sovellukset tarkoituksenmukaisesti hyödyntävät prosessorin ominaisuuksia, kuten useita ytimiä, muistiarkkitehtuuria ja vektoriyksiköitä. Usein prosessorien ominaisuuksien tehokas hyödyntäminen ei kuitenkaan ole helppoa, minkä takia se vaatii hyviä ohjelmointitaitoja ja aikaa.

Insinöörityössä tutkittiin Unity-pelimoottorin tarjoamia rinnakkaislaskennan työkaluja ja niiden soveltuvuutta pelien optimointiin. Unityn monisäikestämiseen tarkoitetun Job Systemin ja sitä tukevan Burst-kääntäjän testausta varten kehitettiin testisovellus, jossa työkaluja hyödynnettiin optimointitarkoituksessa. Job Systemillä saatiin tehokkaasti hyödynnettyä käytössä olevan prosessorin kuutta ydintä, ja Burst-kääntäjällä saatiin hyödynnettyä muun muassa prosessorin vektoriyksiköitä. Testisovelluksella suoritetuissa suorituskykymittauksissa selvisi, että kun prosessoriytimiä saadaan hyödynnettyä tehokkaasti, suorituskykyä voidaan saada kasvatettua moninkertaiseksi. Samalla selvisi, että myös

käytettävällä kääntäjällä ja kääntäjäasetuksilla voi olla suurikin merkitys saavutettavaan suorituskykyyn. Job Systemin ja Burst-kääntäjän tehokas käyttö osoittautui suhteellisen helpoksi, ilman että aiempaa kokemusta pelikoodin rinnakkaistamisesta juurikaan oli.

Koska Unityn rinnakkaislaskentaan tarjoamat työkalut osoittautuivat tehokkaiksi ja helpoiksi käyttää insinöörityön testisovelluksessa, kannattaa niiden käyttöä vahvasti harkita kehitettäessä pelejä Unityllä, vaikka aikaisempaa kokemusta samankaltaisten tekniikoiden käytöstä ei ennalta olisikaan. Toisaalta täytyy myös mainita, että insinöörityössä kehitetty testisovellus ei ollut kokonainen peli ja näin käytetyistä tekniikoista saatu käytännön hyöty ja niiden käytännöllisyys voivat olla rajoittuneempia varsinaisissa peleissä ja vaativat laajempaa tutkimusta.

Lähteet

1 Hennesy, John & Patterson David. 2017. Computer Architecture, A Quantative Approach. 6th ed. E-kirja. Morgan Kaufmann.

2 Culler, David E; Gupta, Anoop; Singh, J.P & Singh, Jaswinder Pal. 1998. Parallel Computer Architecture. E-kirja. Elsevier Science.

3 Introduction to Parallel Computing. 2021. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-to-parallel-computing/>. Päivitetty 4.6.2021. Luettu 4.4.2022.

4 Superscalar Architecture 2021. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/superscalar-architecture/>. Päivitetty 2.6.2021. Luettu 4.4.2022.

5 Moore, Chuck. 2011. Data processing in exascale-class computer systems. Verkkoaineisto. The Salishan Conference on High Speed Computing. <https://www.lanl.gov/conferences/salishan/salishan2011/3moore.pdf>. 27.4.2011. Luettu 3.4.2022.

6 Difference between MultiCore and MultiProcessor System. 2020. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/difference-between-multicore-and-multiprocessor-system/?ref=gcse>. Päivitetty 28.7.2020. Luettu 17.4.2022.

7 Benefits of Multithreading in Operating System. 2019. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/benefits-of-multithreading-in-operating-system/?ref=lbp>. 15.8.2019. Luettu 17.4.2022.

8 Simultaneous multithreading. 2021. Verkkoaineisto. IBM. <https://www.ibm.com/docs/en/sdse/6.4.0?topic=planning-simultaneous-multithreading>. Päivitetty 2.8.2021. Luettu 6.4.2022.

9 C++11 Multithreading – Part 4: Data Sharing and Race Conditions. Verkkoaineisto. thisPointer. <https://thispointer.com/c11-multithreading-part-4-data-sharing-and-race-conditions/>. Luettu 17.4.2022.

10 Steam Hardware & Software Survey: March 2022. 2022. Verkkoaineisto. Steam. <https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam>. Luettu 6.4.2022.

11 Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. 2021. Verkkoaineisto. Intel. <https://cdrdv2.intel.com/v1/dl/getContent/671200>. Luettu 17.4.2022.

12 Auto-Parallelization and Auto-Vectorization. 2021. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/cpp/parallel/auto-parallelization-and-auto-vectorization?view=msvc-170>. 3.8.2021. Luettu 17.4.2022.

13 What is DOTS and why is it important? Verkkoaineisto. Unity. <https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important/?tab=overview#>. Luettu 5.4.2022.

14 DOTS Packages. Verkkoaineisto. Unity. <https://unity.com/dots/packages>. Luettu 5.4.2022.

15 Johansson Tim. 2018. What is a Job System? Verkkoaineisto. Unity. <https://blog.unity.com/technology/what-is-a-job-system>. 22.10.2018. Luettu 5.4.2022.

16 C# Job System Overview. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemOverview.html>. 15.6.2018. Luettu 5.4.2022.

17 What is a job system? 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemJobSystems.html>. 15.6.2018. Luettu 5.4.2022.

18 The safety system in the C# Job System. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemSafetySystem.html>. 15.6.2018. Luettu 8.4.2022.

19 NativeContainer. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemNativeContainer.html>. Luettu 8.4.2022.

20 Creating jobs. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemCreatingJobs.html>. 15.6.2018. Luettu 9.4.2022.

21 Scheduling jobs. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemSchedulingJobs.html>. 15.6.2018. Luettu 9.4.2022.

22 JobHandle and dependencies. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemJobDependencies.html>. 15.6.2018. Luettu 9.4.2022.

23 JobHandle.Complete. Verkkoaineisto. Unity. <https://docs.unity3d.com/ScriptReference/Unity.Jobs.JobHandle.Complete.html>. Luettu 9.4.2022.

24 JobHandle.ScheduleBatchedJobs. Verkkoaineisto. Unity. <https://docs.unity3d.com/ScriptReference/Unity.Jobs.JobHandle.ScheduleBatchedJobs.html>. Luettu 9.4.2022.

25 ParallelFor jobs. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemParallelForJobs.html>. 15.6.2018. Luettu 9.4.2022.

26 ParallelForTransform jobs. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/JobSystemParallelForTransformJobs.html>. 15.6.2018. Luettu 9.4.2022.

27 Overview of .NET in Unity. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/overview-of-dot-net-in-unity.html>. Luettu 11.4.2022.

28 IL2CPP Overview. 2018. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/IL2CPP.html>. Päivitetty 15.5.2018. Luettu 11.4.2022.

29 Quick Start. Verkkoaineisto. Unity. <https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/QuickStart.html>. Luettu 11.4.2022.

30 Burst User Guide. Verkkoaineisto. Unity. https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/index.html. Luettu 11.4.2022.

31 Meijer, Lucas. 2019. On DOTS: C++ & C#. Verkkoaineisto. Unity. <https://blog.unity.com/technology/on-dots-c-c>. 26.2.2019. Luettu 11.4.2022.

32 Muttel, Alexandre. 2019. Behind the burst compiler, converting .NET IL to highly optimized native code. Verkkoaineisto. DotNext. <https://www.youtube.com/watch?v=LKpyaVrby04>. 23.9.2019. Luettu 11.4.2022.

33 Fredriksson, Andreas. 2018. Unity at GCD – C# to Machine Code. Verkkoaineisto. Unity. <https://www.youtube.com/watch?v=NF6kcNS6U80>. 30.3.2018. Luettu 11.4.2022.

34 Standalone Player Support. Verkkoaineisto. Unity. <https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/StandalonePlayerSupport.html>. Luettu 11.4.2022.

35 C#/.NET Language Support. Verkkoaineisto. Unity, <https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/CSharpLanguageSupport_Types.html#supported-net-types>. Luettu 11.4.2022.

36 Unity.Burst.Intrinsics. Verkkoaineisto. Unity. <https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/CSharpLanguageSupport_BurstIntrinsics.html#unityburstintrinsics>. Luettu 11.4.2022.

37 Optimization Guidelines. Verkkoaineisto. Unity. <https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/docs/OptimizationGuidelines.html>. Luettu 11.4.2022.

38 Ante, Joachim. 2020. How the TransformAccess actually transform game objects from jobs? Verkkoaineisto. Unity. <https://forum.unity.com/threads/how-the-transformaccess-actually-transform-game-objects-from-jobs.546040/>. 12.6.2020. Luettu 15.4.2022.