Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Tietotekniikan koulutusohjelma

PELIN SUORITUSKYVYN OPTIMOINTI UNITY TINYN AVULLA

Työn tarkastaja(t): Professori Jussi Kasurinen

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Tietotekniikan koulutusohjelma
Karri Pyykkö

Pelin suorituskyvyn optimointi Unity Tinyn avulla
Kandidaatintyö 2021

49 sivua, 25 kuvaa, 6 taulukkoa
Työn tarkastajat: Professori Jussi Kasurinen

Hakusanat: Unity, Tiny, suorituskyky, optimointi, ECS, DOTS

Unity Tiny on Unityn 2018 loppupuolella julkaisema Unityn paketti, jonka tarkoituksena on ottaa täysi hyöty irti Unityn ECS (Entity-Component-System) arkkitehtuurista ja tarjota kehittäjälle työkaluja kyseisen arkkitehtuurin ja sen teknologioiden hyödyntämiseksi. Mainospuheessaan Unity lupaa Tinyn tuovan merkittäviä parannuksia pelimoottorin suorituskykyyn perinteiseen Unityyn ja sen oliokeskeisen arkkitehtuuriin verrattuna. Tämän työn tavoitteena on tutkia näiden väittämien paikkaansa pitävyyttä vertailukykyisten teknologiademojen ja suorituskyvyn mittauksen kautta. Työssä ei kuitenkaan suoriteta optimointia ohjelmakoodin kautta, vaan työssä keskitytään täysin itse paketin tuomiin muutoksiin suorituskyvyssä. Suorituskykyä arvioidaan kolmen suorituskyvyn kannalta oleellisen metriikan kautta, jotka ovat CPU:n (Central Processing Unit) vasteaika, GPU:n (Graphics Processing Unit) vasteaika ja muistin käyttöaste. Tutkimuksen tuloksien perusteella Unity Tiny ja DOTS (Data-Oriented Technology Stack) tuovat merkittäviä parannuksia suorituskykyyn erityisesti suurilla peliobjektien lukumäärillä. Kuitenkin ainakin toistaiseksi perinteinen Unity suoriutuu Tinya paremmin pienemmillä peliobjektien lukumäärillä. Selvistä tapauskohtaisista suorituskyvyn eduista huolimatta, Tiny on yhä hieman keskeneräinen ja kehittäjän tulisikin punnita paketin soveltuvuutta projektikohtaisesti käyttötarkoituksen mukaan.

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Engineering Science
Degree Programme in Software Engineering
Karri Pyykkö

Game performance optimization using Unity Tiny
Bachelor's Thesis 2021

49 pages, 25 figures, 6 tables
Examiners: Professor Jussi Kasurinen

Keywords: Unity, Tiny, performance, optimization, ECS, DOTS

Unity Tiny is a package released by Unity in late 2018. The package's purpose is to take full advantage of Unitys ECS (Entity-Component-System) architecture and to provide the developer with tools to take advantage of said architecture and technology. In their sales pitch Unity promises that Tiny will bring significant performance benefits for the game engine in comparison to the traditional Unity game engine. The purpose of this work is to compare the validity of these claims through building two comparable artefacts and gathering data on their performance. Instead of optimization of the program code, this work will be focusing on the direct performance benefits of the package itself. The performance is measured through three different core metrics, which are CPU (Central Processing Unit) response time, GPU (Graphics Processing Unit) response time, and memory utilization. Based on the results, Unity Tiny and DOTS (Data-Oriented Technology Stack) see a significant performance boost, especially with a high number of active game objects. On the contrary, at least for now, the traditional Unity performs better with a lower number of active game objects. Despite clear performance benefits, Tiny is still a little rough around the edges, which is why the developer should consider Tiny on a project to project basis.

SISÄLLYSLUETTELO

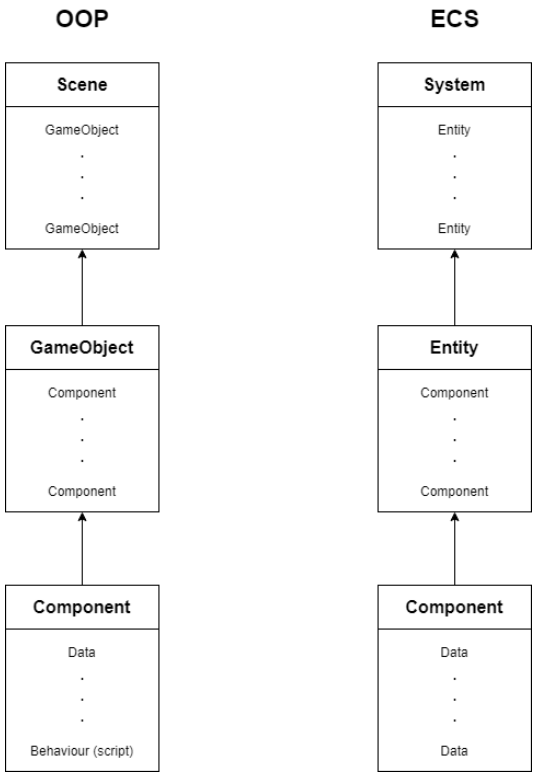
SYMBOLI- JA LYHENNELUETTELO

2D	Two-Dimensional
3D	Three-Dimensional
AR	Augmented Reality
CPU	Central Processing Unit
DOTS	Data-Oriented Technology Stack
EC	Entity Component
ECS	Entity-Component-System
FPS	Frames Per Second
GC	Garbage Collection
GPU	Graphical Processing Unit
HPC#	High-Performance C#
ID	Identifier
LOD	Level of Detail
LTS	Long Term Support
OO	Object-Oriented
OOP	Object-Oriented Programming
SIMD	Single Instruction Multiple Data
UI	User Interface

1 JOHDANTO

1.1 Tausta

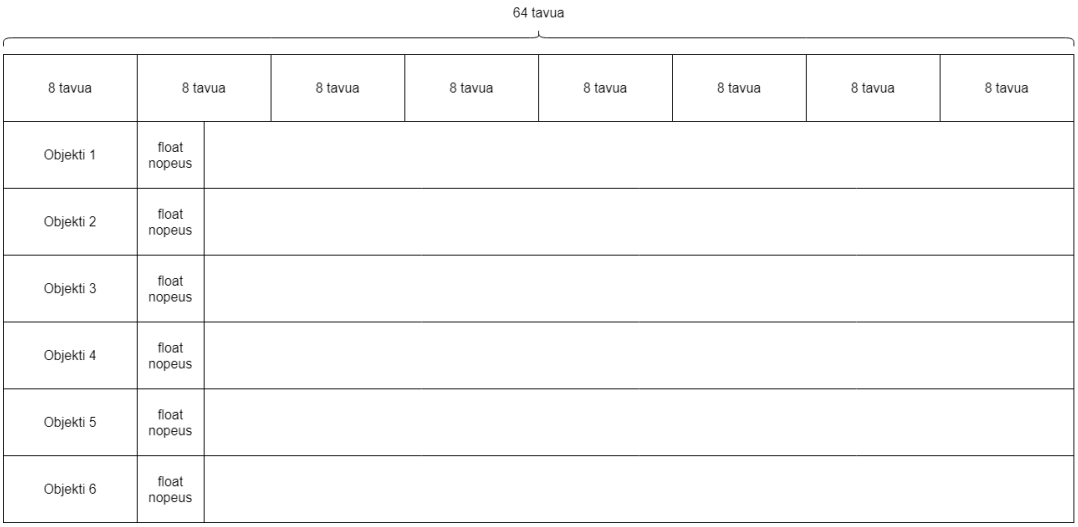
Unity julkaisi 2018 joulukuussa uuden pelikehitystyökalun nimeltä Project Tiny, joka rakentuu pakettina Unity-pelimoottorin ja editorin päälle. Tiny on suunniteltu erityisesti mobiilipelien ja muiden tiedostokoolta kevyiden pelien kuten pelattavien mainosten tai sovelluskaupan sisäisten pelidemojen rakentamiseen. Tiny mahdollistaa pelien pelaamisen ilman erillistä asennusta, ja lupaa tämän lisäksi lähes viiveettömiä latausaikoja. Tiny käyttää modulaarista datakeskeistä ECS (Entity-Component-System) arkkitehtuuria, jonka pääasiallinen etu on sen tuoma parantunut suorituskyky ja pakettikoko. Ballard ja Best (2018) kirjoittavatkin blogissaan, että IPhone 6S:llä Tiny kykenee näyttämään 3-4 kertaa enemmän liikkuvia ja animoituja sprite-grafiikoita kuin muut johtavat verkkoalustaiset 2D pelimoottorit säilyttäen 60 FPS kuvataajuuden. Vaikka ECS arkkitehtuurin periaatteet ovat hyödynnettävissä perinteisessäkin Unityn pelimoottorissa, Tinyn pääasiallinen ero on se, että se pakottaa käyttäjän noudattamaan ECS arkkitehtuuria. Vaikka ECS ja perinteinen Unityn oliokeskeinen OOP-malli (Object-Oriented Programming) ovat monella tapaa samankaltaiset kuten osittain nähtävissä kuvassa Kuva 1, on arkkitehtuurien välillä yksi suuri ero: toisin kuin perinteisen Unityn komponentit, eivät ECS komponentit voi sisältää muuta kuin dataa. Tämä tarkoittaa käytännössä sitä, että kaikki skriptit eli *systeemit* ja *behaviourit* määritellään Tinyssa irrallisina kokonaisuuksina. Tämän lisäksi Tiny ei tue enempää kuin yhtä saman tyypin komponenttia yhdessä entiteetissä. (Unity Technologies, 2020a)



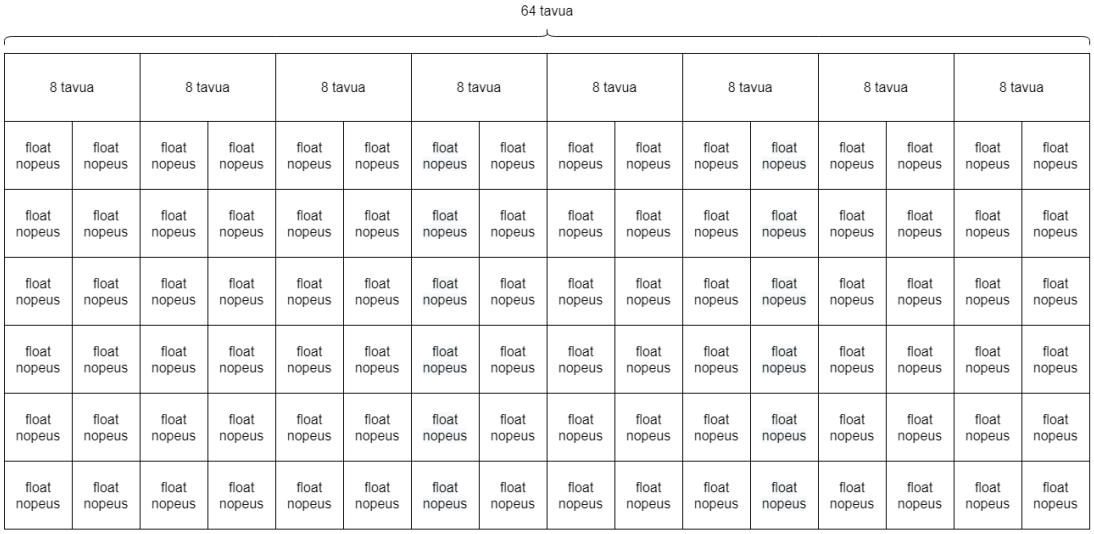
Kuva 1. Unity OOP ja ECS arkkitehtuuri

Kuten aikaisemmin mainittu, Tinylla pystytään kehittämään erityisesti mobiilialustalle soveltuvia pienen pakettikoon omaavia pelejä, joita ei tarvitse asentaa erikseen. Vaikka Mooren laki, jonka mukaan transistorien määrä mikropiireissä kaksinkertaistuu joka toinen vuosi, uskotaan päättyvän lähitulevaisuudessa ilman merkittävää tieteellistä läpimurtoa, on prosessorien kehitys yhä huomattavasti nopeampaa kuin muistin kehitys. (Kumar, 2015; Unity, 2019; Waldrop, 2016) Siitä johtuen, muistista tuleekin helposti monien ohjelmistojen pullonkaula. Ratkaisuna tähän ongelmaan voidaan hyödyntää välimuistia, josta tiedon hakeminen on huomattavasti nopeampaa kuin päämuistista hakeminen. Tämä tuo kuitenkin mukanaan uuden ongelman, joka on välimuistin tilan optimointi. Kuvan 2 esittämä OOP-mallin mukainen välimuistin muistinvaraus, jossa jokaiselle objektille on varattu kiinteä määrä muistia, tuhlaakin jopa 80% välimuistin kapasiteetista. Tinyn ECS arkkitehtuurin hyödyntämä datakeskeinen malli pyrkii ratkaisemaan tämän ongelmaa datan lokalisaation kautta. Sen sijaan, että jokaiselle objektille varataan kiinteä määrä välimuistia, pakataan samankaltainen data Kuvan 3 mukaisesti tiiviiksi paketiksi. Tehokkaamman välimuistin hyödyntämisen kautta

säästetään aikaa muistioperaatioissa, ja sitä myötä myös energiaa, jolloin puhelimen akku tyhjenee hitaammin. Luonnollisesti tiiviisti pakattu data myös vapauttaa muistia järjestelmän muihin toimintoihin, mikä on erityisen hyödyllistä raskaiden ja/tai rinnakkaisten operaatioiden yhteydessä. Esimerkkinä, jos pelin objektit ovat tiiviisti pakattuna muistiin, mahdollistaa se isompien alueiden tai useiden alueiden latauksen samanaikaisesti. Datan lokalisaation ja rinnakkaisten prosessien kautta pystytään myös minimoimaan välimuistihuteja eli tiedon hakemista hitaammasta muistista välimuistiin käsiteltäväksi. Edellä mainittu välimuistihutien minimoiminen ilmenee pienentyneinä datan prosessointiaikoina, mikä vaikuttaakin siihen miltä pelikokemus itsessään tuntuu. (Unity, 2019; Unity Technologies, 2020b) Kaikki nämä edellä mainitut tekijät tekevät Tinysta hyvin soveltuvan erityisesti mobiilikehitykseen; vaikka älypuhelimet kehittyvät nopeaa tahtia, on niiden suorituskyky yhä pöytäkoneita merkittävästi jäljessä – osittain johtuen virrankulutuksen ja valmistuskustannusten asettamista rajoituksista (Koushanfar ym., 2000). Rajoitetuista resursseista johtuen onkin siis selvää, että ECS arkkitehtuurin kaltaiselle ratkaisulle on luonnollisesti kysyntää.



Kuva 2. Unity OOP välimuistin käyttö



Kuva 3. Unity ECS välimuistin käyttö

1.2 Tavoitteet ja rajaukset

Työn tavoitteena on toteuttaa minimalistinen teknologiademo sekä perinteisellä Unitylla että Tinylla, vertailla pelien eroja suorituskyvyssä, sekä arvioida Tinyn soveltuvuutta pelikehityksen työkaluna. Vertailtavuuden vuoksi demoista pyritään tekemään mahdollisimman samankaltaiset sekä toiminallisuudeltaan että tuotetun koodin ja käytettyjen algoritmien osalta. Tämä ei kuitenkaan realistisesti ole aina mahdollista johtuen eroista teknologioiden arkkitehtuurissa ja ohjelmointikielessä. Esimerkiksi Tiny mahdollistaa vain maksimissaan kahdeksan parametrin käyttämisen yksittäisessä For Each-rakenteen anonyymissa- eli ”lambda”-funktiossa ja myös käytettävissä olevat kirjastot ovat huomattavasti suppeammat kuin perinteisessä Unityssa, mikä puolestaan rajoittaa toteutuksien samankaltaisuutta (Unity Technologies, 2021a). Suorituskyvyn arviointi suoritetaan käyttäen Unityn *Profiler* työkalua ja sen laajennusta *Profile Analyzer*, jotka keräävät dataa useista suorituskyvyn kannalta oleellisista metriikoista ja muodostavat yhteenvetoja kerätystä datasta. Työssä käsitellään myös Tinyn etuja ja haittoja verrattuna Unityn oletusarvona käytettävään pelimoottoriin ja editoriin, ja pohditaan Tinyn tulevaisuutta sekä suunniteltujen muutosten vaikutusta suorituskykyyn.

Työn tarkoituksena ei ole optimoida suorituskykyä itsessään suoraan peliohjelmoinnin kautta vaan keskittyä ainoastaan käytetyn teknologian tuottamaan vaikutukseen pelin suorituskyvyssä. Työtä voidaan myös käyttää päätöksenteon tukena pelimoottorin valinnassa erityisesti, jos suorituskyky on pääasiallinen valintaperuste, kuten usein mobiilipeleissä tai muissa järjestelmissä, joiden suoritusteho ja muisti ovat rajallisia. Työ luo myös pohjaa tulevalle tutkimustyölle, aikaisemman tutkimusmateriaalin ollessa aiheeseen liittyen jokseenkin vähäistä.

1.3 Työn rakenne

Työ koostuu viidestä pääluvusta: Kirjallisuus, Toteutus, Tulokset, Pohdinta ja tulevaisuus, ja Yhteenveto. Luvussa 2 käsitellään tärkeimpiä aiheeseen liittyviä konsepteja aikaisemman kirjallisuuden kautta. Luku esittelee myös aikaisempia tutkimuksia työn aiheeseen liittyen, sekä pohtii metriikoiden valintaa olemassa olevaan kirjallisuuteen tukien. Luku 3 käsittää varsinaisen työn toteutuksen kuvauksen, mikä sisältää askeleet ympäristön valmistamiseksi ja vertailtavien artefaktien tuottamiseksi. Luvussa 4 esitetään mittaustulokset ja koejärjestelyt, joilla kyseiset tulokset saavutettiin. Luvussa 5 pohditaan tuloksia ja tehdään johtopäätöksiä saadun datan perusteella. Luvussa myös pohditaan mitä koejärjestelyissä olisi voitu tehdä toisin ja mihin asioihin tulevassa tutkimuksessa voidaan kiinnittää erityistä huomiota. Viimeisenä luku 6 sisältää yhteenvedon tuloksista sekä yleisen arvion Unity Tinyn valmiudesta pelikehityksen työkaluna.

2 KIRJALLISUUS

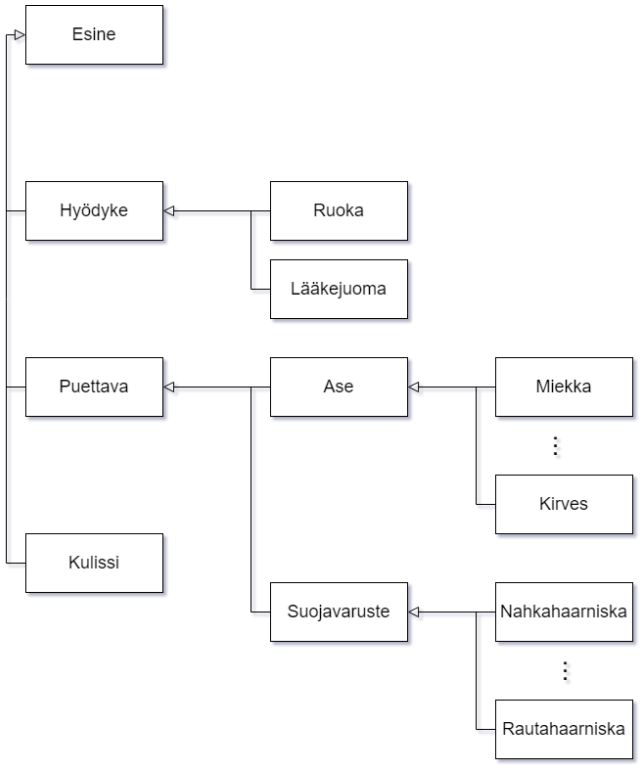
Unityn Tiny projektin ollessa vielä uusi, aiheeseen liittyvää kirjallisuus on vielä jokseenkin vähäistä. Tämän takia suuri osa taustatiedosta on lähtöisin suoraan Unityn omasta dokumentaatiosta ja muista hajanaisista verkkolähteistä. Aikaisemman tutkimuksen vähäisyydestä huolimatta, voidaan aikaisempia tieteellisiä kirjoituksia käyttää erityisesti ohjaamaan metriikoiden, analyysin, tutkimusmenetelmän, ja työkalujen valintaa. Tässä kappaleessa käsittelemme olioperustaisen ja dataperustaisen ohjelmoinnin taustaa ja pääasiallisia eroja, syitä dataperustaisen paradigman omaksumiselle, Unityn DOTS (Data-Oriented Technology Stack) rakennuspalikoita, suorituskyvyn mittaamisen metriikoiden valintaa, sekä esittelemme aikaisempia aiheeseen liittyviä tutkimuksia.

2.1 Object Oriented Programming (OOP)

Olioperustainen ohjelmointi (Object Oriented Programming, OOP) on jo vuosikymmeniä ollut ohjelmistokehityksen vallitseva ohjelmointiparadigma. Pelikehityksen suosiessa oliopohjaisia kieliä kuten C#, C++ ja Java, tilanne ei juurikaan eroa pelikehityksen osalta. (Garcia and de Almeida Neris, 2014) Suosion kautta OOP onkin vain vahvistanut asemaansa vallitsevana paradigmana, minkä takia erityisesti C++ ja Java ovatkin ensimmäisiä kieliä, mitä monet koulussa opettelevat. Lisäksi ne tarjoavat kattavan valikoiman sekä korkean ja matalan tason ominaisuuksia kuten bittimanipulaatiota ja polymorfismia, minkä takia ne soveltuvatkin hyvin opettamaan tärkeitä ohjelmoinnin aihealueita. Tämä ei kuitenkaan tarkoita, etteikö OOP kielissä olisi omia ongelmia ja puutteita. Erityisesti Polymorfismi sekä perintä ovat vaikeita konsepteja aloittelijoille, ja esimerkiksi Javan tiedostonkäsittely on huomattavasti monimutkaisempaa kuin useissa muissa kielissä. (Vujošević-Janičić and Tošić, 2008)

OOP:lle on tyypillistä tiedon kapselointi ja monimutkaiset perintäpuut. Kuvassa Kuva 4 on kuvattu yksinkertainen perintäpuu, jossa *Esine* on isäluokka ja lapsiluokat perivät isäluokan toiminnallisuuksia kuvan osoittamalla tavalla. Polymorfismin mukaisesti esimerkissä *Miekka* siis voidaan mieltää myös *Ase*, *Puettava*, ja *Esine* luokan jäseneksi. Vaikka kapseloinnissa ja

polymorfismissa on omat hyvät puolensa, tekevät ne nopeasti pelimoottorin ylläpidosta ja laajentamisesta vaikeaa. Ylläpidettävyyden lisäksi OOP Unityn kontekstissa tuo mukanaan myös monia suorituskyvyn kannalta oleellisia ongelmakohtia. Blogissaan Meijer (2019) tuo esille, kuinka Unityn perinteinen komponenttisysteemi on kirjoitettu OOP oppien mukaisesti; komponentit ja peliobjektit ovat raskaita C++ objekteja ja niiden luominen tai tuhoaminen vaatii mutex lukon, mikä luonnollisesti toimii esteenä rinnakkaisuudelle. Tämän lisäksi jokainen C# objekti osoittaa sitä vastaavaan C++ objektiin, eikä meillä ole tarkkaa tietoa niiden sijainnista muistissa, mikä puolestaan aiheuttaa runsaasti välimuistihuteja.



Kuva 4. OOP perintäpuu.

2.2 Unity Data-Oriented Technology Stack (DOTS)

Vaikka OOP on ollut jo pitkään hallitseva ohjelmointiparadigma, on siinä yhä puutteita erityisesti suorituskyvyn näkökulmasta kuten luvussa *2.1 Object Oriented Programming (OOP)* huomattiin. DOTS on Unityn vastaus tähän ongelmaan; OOP:n olioperustaisen

lähestymistavan sijasta Tiny on rakennettu hyödyntämään dataperustaista lähestymistapaa, joka on nimeltään Unity DOTS. Yksinkertaisuudessaan DOTS on yläkäsite, joka sisältää monia teknologioita suorituskyvyn parantamiseksi. Tässä kappaleessa keskitymmekin kolmeen DOTS:n alaluokkaan, jotka ovat *ECS, Job System* ja *Burst Compiler*.

ECS on konsepti, joka muodostaa Unityn dataperusteisen lähestymistavan ytimen. ECS itsessään ei ole uusi ilmiö – se sai alkunsa vastauksena pelikehityksen tarpeisiin. Tarkkaa synnyinhetkeä arkkitehtuurille on vaikea määrittää, mutta saman filosofian omaavia arkkitehtuureja löytyy jo useiden vuosien takaa. Leonard (1999) kirjoittaa Thief-pelin kehityksen jälkeisessä jälkipuinnissaan, että pelissä perinteisen OOP lähestymistavan sijasta pelissä ei ollut koodipohjaista peliobjektien hierarkiaa ollenkaan; hän myös kertoo dataperustaisen lähestymistavan olleen yksi kehityksen päätavoitteista (Gestwicki, 2012). Myös Bilas (2002) jakoi saman ajatusmaailman OOP:n kankeudesta pelikehityksen tiedon jäsentämistapana. Bilas ehdottaakin esityksessään, että vahvasti oliopohjaisen C++ sijasta siirryttäisiinkin käyttämään Komponentti Systeemiä (engl. Component System), jossa entiteetit kasattaisiin komponenteista vasta ajon yhteydessä. Yhtäläisyyksistä huolimatta, ei näitä kahta esimerkkiä tule sekoittaa ECS:n kanssa; ne edustavat enemmänkin EC (Entity Component) arkkitehtuuria, jossa komponentit sisältävät käyttäytymistä säätelevän logiikan – toisin kuin ECS:ssä, jossa systeemit ovat vastuussa logiikasta. Vaikka edellä mainitut EC arkkitehtuurit eivät ole täysin yhteneviä ECS arkkitehtuurin kanssa, toimivat ne pohjana Unityn perinteiselle arkkitehtuurille ja sittemmin myös ECS arkkitehtuurille.

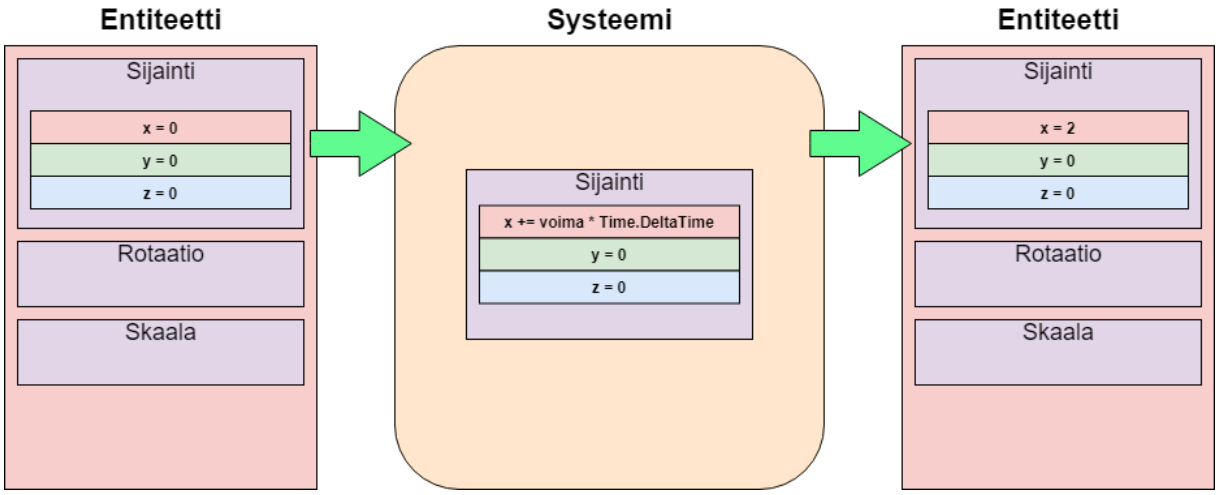
Nimensä mukaisesti ECS (Entity Component System) rakentuu kolmesta pääasiallisesta rakennuspalikasta: entiteeteistä, komponenteista ja systeemeistä.

> **Entiteetit** ovat ECS arkkitehtuurin ylimmän tason rakennuspalikoita. Vaikka entiteetit rinnastetaan perinteisen Unityn GameObjecteihin, entiteetit itsessään eivät sisällä dataa vaan toimivat enemmänkin tunnisteena eri komponenteille ja systeemeille. Yksinkertaisinta on ajatella entiteettejä eräänlaisena tunnisteena (engl. identifier, ID), minkä avulla voidaan yhdistää toisiinsa liittyvät datat. Entiteettien hallinnointi tapahtuu EntityManagerin kautta, jonka tehtävänä on ylläpitää tietoa kaikista entiteeteistä ja jäsentää dataa suorituskyvyn parantamiseksi. Vaikka entiteeteillä ei ole tyyppiä,

voidaan ne luokitella niiden komponenttien perusteella. Tätä komponenttien yhdistelmää kutsutaan arkkityypiksi (engl. archetype). (Meijer, 2019; Unity Technologies, 2021b)

ECS/DOTS **Komponentit** voidaan suoraan rinnastaa perinteisen Unityn GameObject komponentteihin. Kuten aikaisemmin, komponentteja voidaan lisätä ja poistaa entiteeteistä vapaasti ja niitä käytetään yhä suurilta osin samoihin tarkoituksiin, kuten äänien, kameran ja partikkelien sitomiseen tiettyyn peliobjektiin tai entiteettiin. Entiteettien komponenttien ja peliobjektien komponenttien välillä on kuitenkin yksi merkittävä ero: entiteettien komponentit ovat puhtaasti tietovarastoja eivätkä siten sisällä mitään koodia. Useat ECS komponentit ovat myös perinteisen Unityn komponentteja rakeisempia. Esimerkiksi Transform-komponentti, joka perinteisessä Unityssa sisältää tiedon sijainnista, rotaatiosta ja skaalasta, on ECS:ssä paloiteltu kolmeen erilliseen osaan, jolloin kehittäjä saa vapaasti jättää pois osat, joita ei tarvita. (Unity Technologies, 2020c)

Systeemit toimivat ECS:n aivoina. Systeemit hoitavat kaiken pelin logiikan ja tilan muutokset. Systeemit voivat siis esimerkiksi päivittää tietyn entiteetin sijaintia kuvan Kuva 5 mukaisesti. Unityn ECS sisältää useita eri systeemityyppejä, joista oleellisimmat ovat: *SystemBase*, *EntityCommandBufferSystem*, *ComponentSystemGroup* ja *GameObjectConversionSystem*. Tämän lisäksi on olemassa kaksi systeemiluokkaa *The ComponentSystem* ja *JobComponentSystem*, joiden käyttöä ei kuitenkaan suositella. Näiden kahden systeemien käyttö on yhä mahdollista, eivätkä ne ole vielä virallisesti vanhentuneita, mutta niiden vaiheittaista poistoa DOTS rajapinnasta ajetaan hiljalleen eteenpäin.

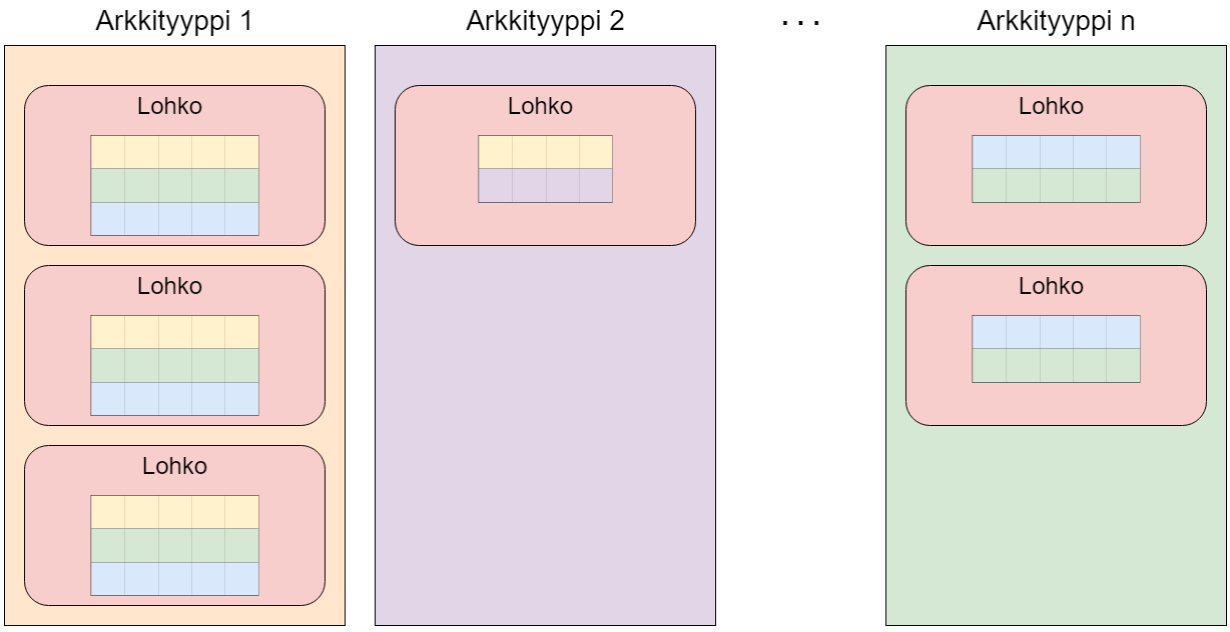


Kuva 5. ECS systeemi.

Vaikka ECS ja OOP ovat hyvin samankaltaisia kuten kappaleessa *1.1 Tausta* lyhyesti mainittiin, on niiden välillä kuitenkin oleellisia eroja. Sen sijaan, että objektit perisivät ominaisuuksia ja toiminnallisuuksia isäluokilta olio-ohjelmoinnille tyypillisellä tavalla, ECS pyrkii pitämään objektit (tai entiteetit), toisistaan irrallisina. Pitämällä objektien väliset suhteet yksinkertaisina tai olemattomina, OOP:n syvän perimispuun sijasta, vähentää se koko toteutuksen kompleksisuutta. Objektien välisten kytkentöjen irtauttaminen myös helpottaa pelin suorituksenaikasta objektien muokkaamista, sillä monet objektien käyttäytymistä määrittävät ominaisuudet (esim. näkyvyys, kinematiikka, vuorovaikutus) voidaan uudelleenmäärittää vapaasti. Esimerkkinä Hatledal ym. (2021) antaa *Breakable*-luokasta luodun olion, joka tulee aina olemaan tyyppiä Breakable, kun taas ECS mahdollistaa Breakable komponentin poistamisen tai korvaamisen toisella komponentilla. Myös Rafaillac ja Huot (2019) havainnollistavat edellä mainittua ECS:n ominaisuutta esimerkin kautta; kuvitellaan pelin näkymään *Instantiate*-metodin kautta luotu kukkaruukku, joka on nähtävissä, mutta ei estä pelaajan liikkumista, eikä kykene itse liikkumaan. ECS systeemin kautta voidaan kuitenkin suorituksen aikana antaa pelaajalle mahdollisuus olla vuorovaikutuksessa ruukun kanssa ja antaa ruukulle massa sekä geometriaa määrittävä kehikko.

Unityn sanojen mukaan ECS on dataperustainen lähestymistapa ohjelmien mallintamiseen, joka helpottaa monisäikeistämistä ja tarjoaa useita parannuksia suorituskykyyn (Unity

Technologies, 2020d). Parantunut suorituskyky tulee pääasiassa tehokkaamman muistinhallinnan muodossa. Kuten aikaisemmin mainittiin, ECS entiteetit luokitellaan niiden arkkityypin eli komponenttien yhdistelmän perusteella. ECS jakaa muistin 16 kilotavun kokoisiin lohkoihin (engl. Chunk) ja jokainen yksittäinen lohko muistia voi sisältää vain tietyn arkkityypin entiteettejä. Mikäli arkkityypin kaikki edustajat eivät mahdu yhteen 16 kilotavun kokoiseen lohkoon, varataan muistista uusi samankokoinen muistialue arkkityypille. Tämä tapa allokoida muistia arkkityypin mukaan on havainnollistettu kuvassa Kuva 6. Arkkityyppiperustainen tapa allokoida muistia mahdollistaa tehokkaamman muistin käytön datan tiiviin pakkaamisen ja datan lokalisaation kautta kuten Kappaleessa *1.1 Tausta* mainittiin. Välimuistin, ja erityisesti nopean L1 välimuistin, tilan ollessa hyvin rajallista on sen tilan tehokas hyödyntäminen luonnollisesti tärkeää. Data, joka ei mahdu nopeampaan välimuistiin vuotaa hitaampaan muistiin, mikä puolestaan hidastaa operaatioita, erityisesti välimuistihutien tapauksessa. Muistin hyödyntämisen lisäksi ECS parantaa suorituskykyä myös vektorisaation ja rinnakkaisten operaatioiden muodossa. (Meijer, 2019) Näistä tehtävistä vastuussa ovat Unityn *Job System* ja *Burst Compiler*.



Kuva 6. ECS muistin hyödyntäminen, perustuu Unityn dokumentaation kuvitukseen. (Unity Technologies, 2021c)

2.2.1 Unity Job System

Modernien prosessorien (engl. Central Processing Unit, CPU) ydinten määrä kasvaa kovaa tahtia. Esimerkiksi AMD:n 2017 kuluttajakäyttöön julkaisema Zen mikroarkkitehtuuriin perustuva Ryzen 7 2700X prosessori sisältää 8 ydintä, ja AMD:n myöhemmin 2019 julkaisema Zen 2 pohjainen Ryzen 9 3900X sisältää jopa 12 ydintä. Zen 2 mikroarkkitehtuuri tukee teoriassa jopa 64:ää ydintä, mutta tämä ei toteudu kuin malliston ääripään prosessoreissa. Myös Zen 3 mikroarkkitehtuuri julkaistiin 2020 loppupuolella, mutta malli ei nähnyt merkittäviä eroja ydinten lukumäärässä. (AMD, 2021; Suggs ym., 2020) Laitteistossa on siis selvästi reilusti varaa rinnakkaisprosessoinnille, mutta tästä huolimatta vain harvat sovellukset ja pelit osaavat ottaa hyödyn irti jokaisesta ytimestä. Tutkimuksessaan Gao ym. (2015) huomasivatkin, että heidän seitsemästä valitsemastaan mobiilisovelluksesta vain yksi pystyi hyödyntämään kaikkia laitteiston neljää ydintä. Myös esimerkiksi selainpeli nimeltä ”Jetpack” onnistui aktivoimaan kolme ydintä. Gao ym. myös huomauttaa, että ydinten aktivointi ei välttämättä tarkoita, että kaikkia ytimiä käytettäisiin, vaan että käyttöjärjestelmä ennustaa, että ytimelle saattaa löytyä tekemistä. Vähäinen ydinten käyttö saattaa osittain johtua siitä, että sille ei yksinkertaisesti ole tarvetta; Blake ym. (2010) kertovat tutkimuksessaan, että monisäikeistyksestä huolimatta, sovellusten rinnakkaisuus ei nähnyt merkittävää hyötyä ydinten lukumäärästä. Moniprosessoinnin hyödyntämiseksi kehittäjien tulisikin keskittyä rinnakkaisuuteen ohjelmoinnissaan. Muita syitä rinnakkaisuuden vähäisyydelle voivat olla muun muassa arkkitehtuurin asettamat rajoitukset ja/tai näytönohjaimen (engl. Graphical Processing Unit, GPU) sekä muun laitteiston suorittama rinnakkainen prosessointi, jolloin CPU:lle jää vähemmän prosessoitavaa.

Unityn huomasi edellä mainitun ongelman, ja lähtikin parantamaan rinnakkaista prosessointia C# Job Systemin kautta. Unityn sanojen mukaan ”Job System mahdollistaa yksinkertaisen ja turvallisen monisäikeisen koodin kirjoittamisen, joka on vuorovaikutuksessa Unityn pelimoottorin kanssa suuremman suorituskyvyn saavuttamiseksi.” (Unity Technologies, 2021d) Kokonaisuudessaan Unityn C# job system on moduuli, joka tarjoaa luokkia ja rajapintoja ”jobien” eli rinnakkaisesti suoritettavien tehtävien suoritukseen ja rakentamiseen. Rinnakkaisuus ei kuitenkaan aina ole helppoa saavuttaa; sen mukana ilmenee usein muita

haasteita kuten samaan dataan kohdistuvat kilpailutilanteet tai lukoista johtuvat umpikujat, joissa kaksi tai useampi säie (engl. thread) odottaa toisen lukon luovutusta. Tämän kaltaisissa kilpailutilanteissa systeemi varoittaa mahdollisesta ongelmakohdasta, ja pyrkii sitä kautta takaamaan turvallisen koodin kirjoittamisen riippumatta säikeiden lukumäärästä. Otetaan kuvan Kuva 7 esittämä koodi esimerkkinä; koodi sisältää yhden lähtötietotaulukon ja kaksi tulostaulukkoa. Jokaisella kuvan päivityksellä ohjelmakoodi pyrkii kopioimaan lähtötietotaulukon tiedot tulostaulukkoihin kerrottuna päivitysten välillä kuluneella ajalla. Toiminnon suorittamiseksi loimme kaksi jobia, jotka suoritetaan rinnakkain. Rinnakkaisuuden mahdollistamiseksi on lähtötietotaulukko *”in”* määritetty *ReadOnly*-tyypiksi. ReadOnly attribuutin lisäämisen kertoo kääntäjälle, että tähän dataan kohdistuu vain lukuoperaatioita ja siten useampi jobi voi lukea siitä tietoa aiheuttamatta kilpailutilanteita. (Meijer, 2019; Unity Technologies, 2021e)

Kuvassa Kuva 7 on nähtävissä myös toinen Unityn ECS arkkitehtuurille tyypillinen tietorakenne *NativeArray*. Koska on NativeArray on lähtöisin Unityn omasta kirjastosta pystyy kääntäjä tekemään oletuksia kyseisestä tietorakenteesta suorituskyvyn optimoimiseksi. Tämä onkin periaate johon Unityn burst compiler pohjautuu. Vektorisaatiolle esteenä on usein, että kääntäjä ei voi olla varma siitä, ettei kaksi osoitinta osoita samaan muistialueeseen ja sitä kautta vaaranna datan eheyttä. NativeArrayn ollessa Unityn käsialaa voidaan kuitenkin varmistua, ettei näin tule koskaan tapahtumaan ja tätä tietoa voidaan käyttää hyväksi burst compilerin toiminnassa. Sama pätee myös esimerkiksi Unityn Mathematics kirjastolle, jossa kääntäjä pystyy uhraamaan tarkkuutta suorituskyvyn parantamiseksi. (Meijer, 2019) Käsittelemme Unityn burst compileria tarkemmin luvussa *2.2.2 Unity Burst Compiler*.

```
public struct MyJob : IJobParallelFor
{
    public float dt;
    [ReadOnly]
    public NativeArray<int> in;
    public NativeArray<int> out;

    public void Execute(int i)
    {
        out[i] = in[i] * dt;
    }
}
public class ParallelReplicator : MonoBehaviour
{
    public void OnUpdate()
    {
        var src = new NativeArray<int>(10, Allocator.Persistent);
        var dst1 = new NativeArray<int>(10, Allocator.Persistent);
        var dst2 = new NativeArray<int>(10, Allocator.Persistent);

        for (var i = 0; i < src.Length; i++){
            src[i] = i;
        }
        var job1 = new MyJob
        {
            in = src,
            out = dst1,
            dt = Time.deltaTime
        };
        var job2 = new MyJob
        {
            in = src,
            out = dst2,
            dt = Time.deltaTime
        };
        JobHandle jobHandle1 = job1.Schedule(dst1.Length, 64);
        JobHandle jobHandle2 = job2.Schedule(dst2.Length, 64);

        jobHandle1.Complete();
        jobHandle2.Complete();
        src.Dispose();
        dst1.Dispose();
        dst2.Dispose();
    }
}
```

Kuva 7. Unityn job system koodinäyte. Perustuu Unityn ReadOnlyAttribute-luokan dokumentaatioon. (Unity Technologies, 2021e)

2.2.2 Unity Burst Compiler

Vaikka C ja C++ ovat tunnetusti korkean suorituskyvyn omaavia kieliä, eivät niiden kääntäjät silti aina toimi odotetulla tavalla, mikä puolestaan ilmenee epäoptimaalisena käännöksenä konekieleksi. C++ kääntäjän muovaamisen sijaan Unity päätti kehittää oman kääntäjänsä Unity burst compilerin, jonka tarkoituksena on soveltua erityisesti pelikehityksen ja Unity DOTS:n tarpeisiin, joita ovat muun muassa kääntäjälähtöinen korkean suorituskyvyn koodin

pakottaminen, alustariippumattomuus, koodigeneraation läpinäkyvyys ja turvallisen koodin kirjoittaminen. Burst compiler pohjautuu korkeamman suorituskyvyn omaavaan C# osajoukkoon, joka sisältää vain suorituskyvyn kannalta oleellisimmat kirjastot. Tätä osajoukkoa Unity kutsuu nimellä High-Performance C#, tai lyhyemmin HPC#. Burst compilerin päämääränä on ottaa maksimaalinen hyöty irti Unityn job systeemistä ja ECS arkkitehtuurista; esimerkkinä tästä ovat Unityn NativeArray-luokka ja Mathematics kirjasto, jotka mahdollistavat voimakkaasti optimoidun konekielisen koodin tuottamisen. Blogissaan Meijer huomauttaa, että suorituskyky ei ole ainut puoli, johon tulisi keskittyä. Esimerkkinä hän kertoo, että vaikka burst compilerille siirretty C++ renderöintisysteemi ei itsessään nähnyt parannusta suorituskyvyn osalta, sen lopullinen koko oli noin neljä kertaa pienempi johtuen C++ kääntäjän vastahakoisuudesta vektorisaatiota vastaan. (Meijer, 2019) Burst compilerin hyödyntäminen on onneksi tehty helpoksi. Samalla tavalla kuin käytimme ReadOnly-attribuuttia kuvassa Kuva 7, käyttäen *BurstCompile*-attribuuttia voimme kertoa kääntäjälle minkä osan koodista haluamme kääntää burst compileria hyödyntäen. BurstCompile-attribuutti ottaa kaksi valinnaista parametria *FloatPrecision* ja *FloatMode*. FloatPrecision (High/Low/Medium/Standard) määrittää liukulukujen tarkkuuden, missä korkeampi tarkkuus tulee korkeamman ajoajan kustannuksella. FloatMode (Default/Deterministic/Fast/Strict) sen sijaan määrittää liukulukujen optimoinnin moodin. Käytännössä tämä tarkoittaa, että optimoinnissa voidaan tehdä omia oletuksia, käyttää käänteislukuja, sekä sivuuttaa joitakin etumerkkejä kuten nollan tapauksessa. Alla on nähtävissä esimerkki BurstCompile attribuutin käytöstä. Esimerkissä kaikki tietueen metodit käännetään burstilla käyttäen matalaa tarkkuutta ja nopeaa moodia.

```
[BurstCompile(FloatPrecision.Low,FloatMode.Fast)]
struct tietue
{
    // tietueen koodi
}
```

ECS arkkitehtuurin ja job systeemin hyödyntämisen lisäksi burst compiler kykenee optimoimaan operaatioita laitteiston prosessorin perusteella. Esimerkiksi yksittäisen liukulukuoperaation sijasta voidaan suorittaa useampia rinnakkaisia operaatioita, mikäli prosessorilla on rekistereitä vapaana. Tällä hetkellä burst compiler tukee SIMD (Single Instruction Multiple Data) käsky- ja vektorilaajennuksia SSE:stä AVX2 asti. Tuetut SIMD

laajennukset on listattu tarkemmin alla. (Ferreira and Geig, 2018; Unity Technologies, 2021f)

```
using static Unity.Burst.Intrinsics.X86;
using static Unity.Burst.Intrinsics.X86.Sse;
using static Unity.Burst.Intrinsics.X86.Sse2;
using static Unity.Burst.Intrinsics.X86.Sse3;
using static Unity.Burst.Intrinsics.X86.Ssse3;
using static Unity.Burst.Intrinsics.X86.Sse4_1;
using static Unity.Burst.Intrinsics.X86.Sse4_2;
using static Unity.Burst.Intrinsics.X86.Popcnt;
using static Unity.Burst.Intrinsics.X86.Avx;
using static Unity.Burst.Intrinsics.X86.Avx2;
using static Unity.Burst.Intrinsics.X86.Fma;
using static Unity.Burst.Intrinsics.X86.F16C;
using static Unity.Burst.Intrinsics.X86.Bmi1;
using static Unity.Burst.Intrinsics.X86.Bmi2;
using static Unity.Burst.Intrinsics.Arm.Neon;
```

Kuten aikaisemmin mainittu, burst compiler pohjautuu C#:n osajoukkoon, jota kutsutaan nimellä HPC#. Tämä C# .NET:n osajoukko eroaakin perinteisestä C#:sta osittain tuettujen datatyyppien ja kontrollirakenteiden osalta. Esimerkiksi HPC# ei tue *char*, *string* ja *decimal* primitiivejä. Täysi lista eroista on saatavilla Unityn C#/.NET dokumentaatiosta. (Borufka, 2020; Unity Technologies, 2021g)

2.3 Aikasempi tutkimus

Kuten taustassa aikaisemmin mainittiin, Unityn DOTS ja ECS arkkitehtuuri ovat tuore ilmiö. Vaikka Unity demonstroi ECS konseptia Entitas C# ECS kehikon avulla jo vuoden 2016 puolessa välissä, Unity Tiny itsessään julkaistiin vasta vuoden 2018 lopulla (Unity, 2016). Teknologian ollessa uutta, siihen liittyvä tutkimuksen määrä on vielä hyvin rajallista. Kuitenkin tässä kappaleessa käymme läpi muutamia aikaisempia tutkimuksia DOTS:n suorituskyvyn kartoittamiseen liittyen. Tutkimusten ja tieteellisien julkaisujen lisäksi suorituskyvyn mittaamiseen liittyen löytyy myös useita verkkolähteitä. Emme kuitenkaan tässä kappaleessa käsittele näistä tarkemmin kuin Ferreiran ja Geigin Intelin verkkosivulla julkaisemaa artikkelia, jossa he vertaavat klassisen Unityn, job systeemin, burst compilerin ja ECS arkkitehtuurin suorituskyvyllisiä eroja. Kaiken kaikkiaan kirjallisuudessa tulokset ovat hyvin samankaltaisia: Unityn DOTS tuo mukanaan merkittäviä suorituskyvyllisiä parannuksia ja tulosten perusteella sen hyötyä on hankala kiistää.

Tutkimuksessaan Borufka (2020) kartoitti Unityn DOTS systeemin eri osien suorituskykyä. Suorituskyvyn arvioimiseen hän käytti yhden kuvan piirtämiseen käytetyn ajan mediaaniarvoa millisekunteina. Tutkimuksessaan hän erottelee tulokset kahteen luokkaan: piirtämiseen käytetty *aika 5 sekunnin jälkeen* ja *aika 180 sekunnin jälkeen.* Tulosten mukaan DOTS systeemi käyttää piirtämiseen jopa 30 kertaa vähemmän aikaa. Lisäksi hän huomasi, että optimoimalla koodia voidaan aikaa pudottaa vielä hieman enemmän. Kuitenkin aikasäästön ollessa millisekunnin kymmenesosan luokkaa, ei optimoinnin kautta säästetty aika ole merkittävää. Hän teki myös mielenkiintoisia havaintoja muihin DOTS ominaisuuksiin ja kirjastoihin liittyen. Hän kirjoittaa muun muassa, että *Unity.Mathematics* nimiavaruuden alla oleva pistetulon algoritmi antaa perinteisen unityn *UnityEngine* nimiavaruuden alla olevaa algoritmia parempia tuloksia, kun burst compileria ei käytetä. Kuitenkin algoritmien väliset erot katoavat, kun käytetään burstia. Matriisitulon tapauksessa tilanne on päinvastainen: vanhempi *UnityEnginen* alla oleva algoritmi suoriutuu *Unity.Mathematics* vastinetta paremmin. Kuten aikaisemmin, myös tässä tapauksessa erot katoavat, kun käytetään burst compileria. Hän kirjoittaa myös, että matriisituloa laskiessa kappaleen *2.2.2 Unity Burst Compiler* mukaiset liukulukujen tarkkuuden asetukset eivät tee huomattavaa eroaa. Kuitenkin esimerkiksi sinifunktion laskennan tapauksessa tarkkuuden vakioasetus (Standard) tuotti parempia tuloksia. Ferreiran ja Geigin (2018) havainnot eivät juurikaan eroa Borufkan tuloksista. Artikkelissaan he kirjoittavatkin, että pelkästään job systeemin integroiminen lähes kaksinkertaisti mahdollisten näytöllä olevien objektien määrän. Vastaavasti ECS arkkitehtuurin hyödyntäminen job systeemin lisäksi mahdollisti jo kuusinkertaisen objektien määrän. Viimeisenä testinä he käyttivät burst compileria tehtävien/jobien kääntämiseen. Pelkästään tämä pieni muutos nosti näytettävien objektien määrän 1.6-kertaiseksi aikasempaan ECS + job system kokoonpanoon nähden. Kaiken kaikkiaan tämä on siis hieman yli yhdeksän kertaa enemmän peliobjekteja samoilla FPS lukemilla.

Borufkan tavoin myös Turpeinen (2020) sai tutkimuksessaan samankaltaisia tuloksia, vaikkakin tutkimuksessaan Turpeinen keskittyykin mobiililaitteiden suorituskyvyn tutkimiseen pöytäkoneen sijasta. Tutkimuksessaan Turpeinen vertasi Unityn olioperustaisen OO (Object Oriented) systeemin ja DOTS systeemin suorituskykyä kolmen metriikan kautta: FPS (Frames Per Second) sekä CPU:n ja GPU:n vasteaika millisekunteina. Turpeinen huomasi, että DOTS

systeemi suoriutui OO systeemiä paremmin suurilla pelihahmojen lukumäärillä, kun taas OO systeemi pärjää paremmin, kun pelihahmojen variaatio eli eri arkkityyppien lukumäärä on suuri. Kaiken kaikkiaan DOTS systeemi pärjäsi OO systeemiä paremmin kuudessa kahdeksasta testistä, ja niissäkin testeissä, joissa DOTS pärjäsi OO systeemiä huonommin, oli ero marginaalinen. Turpeinen myös huomauttaa, että testien perusteella DOTS systeemi osaa korkean rasituksen alla tasapainoittaa CPU:n ja GPU:n työtaakkaa OO systeemiä paremmin. Tutkimuksen heikkoutena Turpeinen mainitsee sen, että jokainen testi suoritettiin vain kerran jokaista otantaa kohden. Myös käytetyn laitteiston olosuhteet eroavat hieman toisistaan; tulokset saattavat helposti vääristyä johtuen puhelimen akun latauksen tuottamasta lämmöstä, puhelimen jäähdytyksestä ja testien suorittamisjärjestyksestä. Testausolosuhteiden yhtenäistämisen lisäksi Turpeinen mainitsee, että tulevissa tutkimuksissa olisi hyvä tutkia GPU instantiaation (engl. GPU instancing) vaikutusta CPU:n taakan helpottamiseksi OO systeemissä. Hän myös mainitsee mallien tarkkuuden (LOD, Level of Detail) ja näkymän ulkopuolisten objektien poistamisen (engl. Frustum Culling) mahdollisiksi kehitysalueiksi suorituskyvyn optimoinnissa.

2.4 Metriikoiden valinta

Aikaisempi kirjallisuus tarjoaa useita eri kehyksiä pelimoottorien arviointiin ja vertailuun. Esimerkiksi Wu & Wang (2012) listaavat seitsemän kriteeriä ohjaamaan pelimoottorin/pelikehitys-alustan valintaa: (1) tekniset vaatimukset ja alhainen hankinta- sekä käyttöhinta, (2) riittävä dokumentaatio, (3) matala oppimiskynnys, (4) ristiriitaisuus käyttötarkoituksen ja käyttötavan kanssa, (5) tuetut ohjelmointikielet, (6) käyttöliittymän selkeys ja käyttäjäystävällisyys, (7) alustariippumattomuus. Kuitenkin, koska Tiny rakentuu suoraan Unityn ohjelmiston päälle, vaikeuttaa se edellä mainittujen valmiiden kehysten hyödyntämistä. Lisäksi ottaen huomioon paperin aiheen rajauksen, eivät valmiit kehykset tarjoa suoraan tarvittavia työkaluja tai lähestymistapaa suorituskyvyn arvioimiseksi, vaan keskittyvät enemmänkin geneerisiin käyttäjälähtöisiin kriteereihin.

Tutkimuksen kannalta on tärkeää asettaa selkeä määritelmä suorituskyvylle työn kontekstissa. Ilman selkeää määritelmää voidaan suorituskyky mieltää järjestelmäkeskeisen näkemyksen sijasta käyttäjäkeskeiseksi, jolloin tarkastelu kohdistuu järjestelmän resurssien sijasta ohjelmiston käyttäjään. Esimerkkejä edellä mainitun kaltaisista tutkimuksista ovat Wu & Wangin (2012) tutkimus, jossa kartoitettiin eri pelikehitysalustojen soveltuvuutta opetuskäyttöön, sekä Mercan & Durdun (2017) tutkimus, joka keskittyi Unityn käytettävyyden tutkimiseen eri kokemustason omaavilla käyttäjillä. Tämän paperin kannalta mainitut tutkimukset, sekä muut samankaltaiset tutkimukset eivät kuitenkaan omista suurta painoarvoa, vaan suorituskyky määritellään enemmänkin ISO/IEC 25010 standardin mukaisesti ominaisuutena, jota kuvataan pääasiassa kolmen aliluokan kautta: aikasidonnainen suorituskyky, resurssien käyttö, ja kapasiteetti (ISO/IEC 25010:2011, 2011).

Kaiken kaikkiaan varsinainen kirjallisuus liittyen pelimoottorien tai ylipäätään pelien suorituskyvyn arviointiin on vähäistä. Messaoudi ym. (2017) huomasivat tämän puutteen, minkä takia he tekivätkin omaa tutkimusta Unity 3D:n suorituskyvyn kartoittamiseksi. Tutkimuksessaan he mittasivat yhdeksän eri pelin suorituskykyä, kahdella eri grafiikkakokoonpanolla, ja usealla eri alustalla. Suorituskyvyn mittaamiseksi he käyttivät kahta eri metriikkaa: suoritusaika ja sisäinen tietovirta. Tässä suoritusajalla viitataan CPU/GPU:n käyttämään aikaa jokaisen eri näytölle piirretyn kuvan näyttämiseksi. Sisäinen tietovirta sen sijaan viittaa jokaisella eri yksittäisellä kuvalla suoritettuihin objekti/luokka- ja metodikutsuihin.

Vaikka erityisesti Messaoudin ym. tutkimuksessa käytetty suoritusaika on hyvä suorituskyvyn indikaattori ei se itsessään kerro koko tarinaa. Kuvitellaan tyypillinen *Hack and slash* -peli, jossa ruutu nopeasti täyttyy eri vihollisista ja partikkeliefekteistä; vaikka luonnollisesti jokainen kuvalla liikkuva tai jotenkin ympäristöönsä vuorovaikutuksessa oleva osa vaatii oman osansa prosessointitehoista, käy tämä nopeasti raskaaksi myös muistin osalta, sillä jokainen näistä vihollisista pitää olla ladattuna muistiin prosessoinnin nopeuttamiseksi. Unity Tinyn ja ECS:n pääasiallisten etujen ollessa tehokas välimuistin ja prosessorin hyödyntäminen tulisi metriikoiden ainakin osittain heijastaa näitä. Tästä syystä tutkimukseen valitut metriikat ovat

CPU:n vasteaika, GPU:n vasteaika ja muistin käyttöaste. Vaikka valitut metriikat eivät anna absoluuttista kuvaa suorituskyvystä, kattavat ne suurimman osan järjestelmän pääresursseista.

3 TOTEUTUS

Tässä kappaleessa esittelemme oleellisimmat yksityiskohdat testaus- sekä kehitysympäristön käyttöön ottamiseksi. Aluksi kappaleessa *3.1 Työkalut ja laitteisto* käymme läpi artefaktien kehitykseen ja suorituskykymetriikoiden mittaamiseen käytettyjä työkaluja, sekä listaamme käytetyn laitteiston teknisiä tietoja. Kappaleessa *3.2 Ympäristön käyttöönotto ja valmistelu* keskitymme Unityn Editorin ja projektin asetusten alustamiseen ja tarvittavien pakettien käyttöönottoon. Kappale *3.3 Perinteinen Unity ja MonoBehaviour* sisältää kuvauksen perinteisellä Unitylla kehitetystä teknologiademosta, ja vastaavasti kappaleessa *3.4 Unity Tiny* on kuvaus Unity Tinyn vertailtavasta teknologiademosta. Tämän kappaleen tarkoituksena on mahdollistaa tutkimuksessa tehtävien testien toistettavuus tulevaa tutkimusta ajatellen, sekä luoda pohjaa tulosten vertailtavuudelle.

3.1 Työkalut ja laitteisto

Suorituskyvyn arviointia lähestytään käytännön toteutuksen kautta, jossa toteutetaan minimalistinen teknologiademo Unityn vakio pelimoottoria ja editoria käyttäen, sekä jäljennetään demo hyödyntäen Unityn Tiny pakettia ja noudattaen ECS suunnitteluperiaatteita. Tuloksena on kaksi verrattavaa Unityn editorin sisällä ajettavaa demoa, joiden suorituskyvyn arviointi tapahtuu samaa laitteistoa käyttäen, samankaltaisissa olosuhteissa, tulosten vertailukelpoisuuden varmistamiseksi. Usein vertailussa käytetyt metriikat määräytyvät joko alustan tai käytetyn suorituskyvyn seurantaohjelmiston pohjalta (Peterson ym., 2014). Esimerkiksi useimmat käyttöjärjestelmät tarjoavat valmiita metriikoita kuten CPU:n -, GPU:n - ja välimuistin käyttöaste. Tämän työn tapauksessa suorituskyvyn arvioimisessa hyödynnetään Unityn sisäistä Unity profiler työkalua kuten Kappaleessa *2.4 Metriikoiden valinta* mainittiin. Unity profiler näyttää suorituksen aikaista tietoa monesta eri suorituskyvyn kannalta oleellisista metriikoista kuten jo aikaisemmin mainitut CPU/GPU:n käyttöaste ja välimuistin käyttöaste. Tämän lisäksi Unity profiler erottelee esimerkiksi äänien ja videon toistamiseen käytetyt resurssit, sekä tarjoaa tietoa verkon käyttöön liittyen. Lisänä Unity profilerin päällä käytetään Unityn Profile Analyzer pakettia, joka antaa ylimääräistä analytiikkaa kuten minimi-, maksimi-, mediaani-, keski- ja kvartiiliarvoja. Profile Analyzer mahdollistaa myös kahden eri datasetin

vertailun, mikä on erityisen hyödyllistä tutkimuksen luonteen takia. (Unity Technologies, 2020e)

Teknologiademojen kehittämiseen käytetään samaa Unityn versiota vertailukykyisten tulosten tuottamiseksi. Näin pyritään varmistamaan, että mahdolliset erot suorituskyvyssä eivät johdu versioiden välisistä eroista pelimoottorin ja kirjastojen toiminnassa. Valittu Unityn versio demojen toteutukseen on Unity 2020.3.8f1. Version valintaa ohjaa pääasiassa Unityn uudempien versioiden 2021.1 ja ylöspäin mukanaan tuomat ongelmat DOTS, ja erityisesti Entity-paketin, yhteensopivuudessa. Huhtikuussa 2021 Unity ilmoitti, että osa DOTS:n toiminnallisuudesta kuten Entities-paketti ei tule olemaan yhteensopiva Unity 2021 versioiden kanssa, eikä tukea paketille ole suunniteltu ennen vuoden loppua. (Gram, 2021) Teknologian asettamista rajoituksista johtuen käytettävissä oleva Unityn versio siis rajoittuu 2020 LTS (Long Term Support, v2020.3.8f1) julkaisuun. Rajoitukset koskevat myös Entities-pakettia, jonka uusin tuettu versio on versio 0.17. Koodieditorina työssä käytetään pääasiallisesti Visual studio Codea, jonka integraatio Unityyn tapahtuu Visual Studio Code Editor -paketin kautta. Täysi lista työssä käytetyistä työkaluista näkyy taulukossa Taulukko 1.

Työn toteutuksessa demojen vertailuympäristönä käytetään tutkijan omaa kannettavaa tietokonetta. Laitteiston tarkemmat tekniset tiedot ovat listattuna alla:

- **GPU:** NVIDIA GeForce GTX 1050, 2 GB memory
- **CPU:** Intel Core i5 8250U, 4-core @ 1.6 GHz
- **Keskusmuisti:** 8GB, Dual-Channel DDR4 @ 1333 MHz
- **Käyttöjärjestelmä:** Windows 10 Home, v1909

Taulukko 1. Käytetyt työkalut.

Työkalu	Versio	Kuvaus
Unity	2020.3.8f1	Pelimoottori ja pelien kehitysympäristö
Visual studio Code	1.56.2	Koodieditori
Unity profiler	2018.3.6	Unityyn sidottu suorituskyvyn seuranta -työkalu.
Profile Analyzer	0.6.0	Unityn paketti, joka tarjoaa lisää toiminnallisuutta Unity profileriin.

Taulukot 2 ja 3 listaavat työssä käytettyjä Unityn paketteja. Taulukko 2 sisältää Unityn vakiopaketit, jotka ovat esiasennettuna 3D (Three-Dimensional) projektin templaatissa. Taulukko 3 sisältää kaikki paketit, jotka asentuvat Project Tinyn asennuksen yhteydessä, ts. *Project Tiny Full* paketin ja sen riippuvuudet eli osapaketit. Vaikka pakettien lukumäärä on suuri, toteutuksen kannalta oleellisimmat paketit rajoittuvat pääasiassa Hybrid Renderer -pakettiin ja sen riippuvuuksiin. Taulukossa Taulukko 3 nämä paketit on korostettu lihavoituna.

Taulukko 2. Vakio paketit.

Paketti	**Versio**
Custom NUnit	1.0.6
JetBrains Rider Editor	2.0.7
Test Framework	1.1.24
Timeline	1.4.7
Unity Collaborate	1.3.9
Unity UI	1.0.0
Visual Studio Code Editor	1.2.3
Visual Studio Editor	2.0.7

Taulukko 3. Unity Tiny (com.unity.tiny.all) paketit.

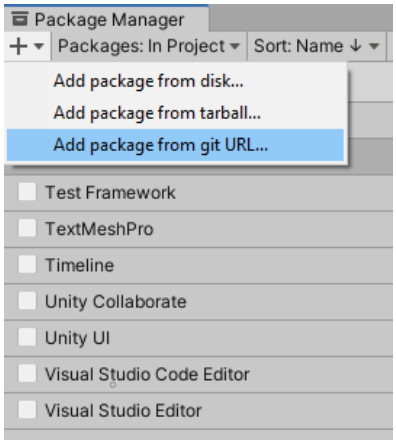
Paketti	Versio
2D Entities Physics	0.2.0-preview.5
Burst	1.4.1
Castle Core	1.0.1
Collections	0.15.0-preview.21
Core RP Library	10.5.0
DOTS Runtime	0.32.0-preview.54
Entities	0.17.0-preview.41
Hybrid Renderer	0.11.0-preview.42
Jobs	0.8.0-preview.23
Mathematics	1.2.1
Mono Cecil	0.16-preview.2
Moq	1.0.0
Newtonsoft Json	2.0.0-preview
Performance testing API	2.3.1-preview
Platforms	0.10.0-preview.10
Platforms Android	0.10.0-preview.10
Platforms Desktop	0.10.0-preview.10
Platforms iOS	0.10.0-preview.10
Platforms Linux	0.10.0-preview.10
Platforms macOS	0.10.0-preview.10
Platforms Web	0.10.0-preview.10
Platforms Windows	0.10.0-preview.10
Project Tiny	0.32.0-preview.54
Project Tiny for Desktop	0.32.0-preview.54
Project Tiny for Web	0.32.0-preview.54
Project Tiny Full	0.32.0-preview.55
Properties	1.6.0-preview
Properties UI	1.6.2-preview.1
Scriptable Build Pipeline	1.9.0
Searcher	4.3.2
Serialization	1.6.2-preview
Shader Graph	10.5.0
TextMeshPro	3.0.4
Unity Physics	0.6.0-preview.3
Universal RP	10.5.0

3.2 Ympäristön käyttöönotto ja valmistelu

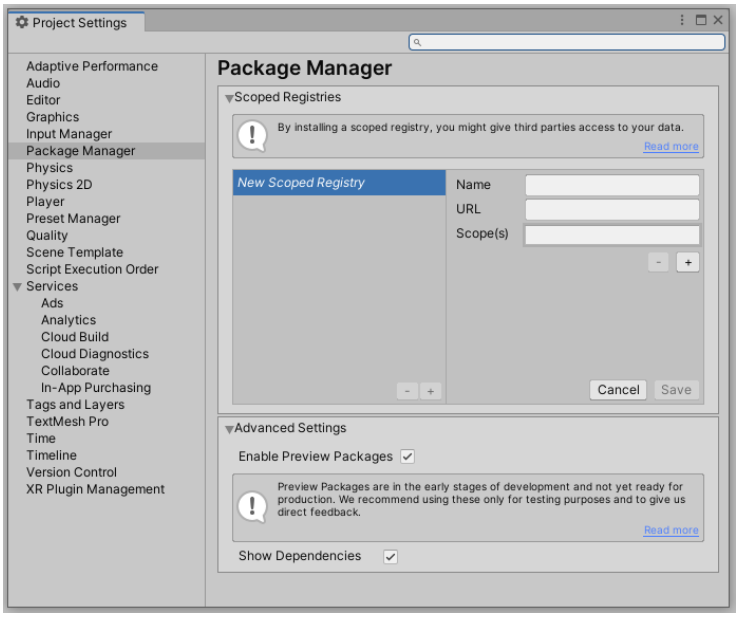
Tässä kappaleessa esittelemme lyhyesti tarvittavat askeleet testaus- ja kehitysympäristön käyttöönottamiseksi. Oletuksena on, että tarvittavat työkalut, jotka on listattu taulukossa Taulukko 1, löytyvät esiasennettuna lukuun ottamatta Unity Profileria ja Profile Analyzer -pakettia. Kappaleessa käymme läpi kolme ympäristön valmistelun kannalta oleellista askelta, jotka ovat (1) Projektin luominen, (2) pakettien asennus ja (3) Unityn asetusten hallinnointi.

Ensimmäinen askel on projektin luominen, mikä tapahtuu Unity Hubin kautta. Varmistetaan, että tarvittava Unityn versio on asennettuna. Mikäli tarvittavaa versiota, eli tässä tapauksessa Unity 2020 LTS (2020.3.8f1) ei löydy, asennetaan se navigoimalla *Installs > Add > Recommended Release > Unity 2020.3.8f1 (LTS)*. Seuraavaksi valitaan halutut Unity moduulit; tässä tapauksessa meille riittää vakiomoduulit *WebGL Build Support*, *Windows Build Support (IL2CPP)* ja *Documentation*. Unity version asennuttua voimme viimein alustaa projektin: Projects > New > 3D > Create. Tämän teknologiademon tapauksessa voidaan templaatiksi valita myös Universal Render Pipeline, jonka tarkoituksena on helpottaa erityisesti grafiikkojen optimointia. Kuitenkin tässä tapauksessa meille riittää hyvin perinteinen 3D templaatti.

Ympäristön latauduttua siirrymme asentamaan tarvittavat paketit: *Window > Package Manager > Add package from git URL…* Tässä tapauksessa haluamme asentaa Project Tiny Full -paketin, mikä onnistuu syöttämällä com.unity.tiny.all tekstikenttään ja painamalla enter-näppäintä. Paketin ja sen riippuvuuksien asennuttua pitäisi uudet paketit näkyä Package Managerin ikkunassa. Mikäli paketteja ei näy, varmista, että suodattimesta on valittuna *”Packages: In Project”* ja Package Managerin asetuksista on sallittuna preview packaget ja riippuuvuudet: *Project Settings > Package Manager > Advanced Settings > Enable Preview Packages / Show Dependencies*.



Kuva 8. Unity-pakettien lisääminen URL kautta.



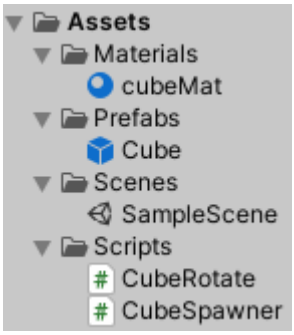
Kuva 9. Unity-pakettien näkyvyyden asetukset.

Viimeiseksi haluamme varmistaa, että DOTS:n toiminnan kannalta oleelliset asetukset ovat valittuna. Burst compilerin ja monisäikeisyyden hyödyntämiseksi haluamme, että *Jobs > Burst > Enable Compilation* ja *Jobs > Use Job Threads* ovat valittuna. Varmuuden vuoksi haluamme vielä viimeisenä varmistaa, että kääntäminen burstilla on sallittuna myös projektin asetuksissa *Project Settings > Burst AOT Settings > Enable Burst Compilation*.

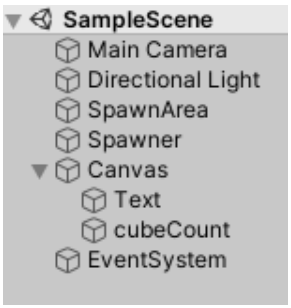
3.3 Perinteinen Unity ja MonoBehaviour

Ensimmäinen vertailtava teknologiademo rakennetaan perinteistä Unityä ja sen ohjelmointiperiaatteita hyödyntäen. Tämä tarkoittaa, että käytännössä kaikki pelin logiikka ja data ilmenevät objektien komponenttidatana. *MonoBehaviour* on perinteisen Unityn kantaluokka skripteille, jonka avulla on mahdollista liittää kyseistä luokkaa perivä ohjelmakoodi peliobjektiin (GameObject). Teknologiademon idea on luoda yksinkertainen spawneri, joka instantioi peliobjekteja, jotka ovat tässä tapauksessa kuutioita. Instantiaatio itsessään ei kuitenkaan ole suorituskyvyn testaamisessa oleellisin rasitusta luova tekijä, vaan tästä vastuussa on systeemi(t) / skriptit, jotka kiertävät kuutioita.

Kuva 10 havainnollistaa projektin kansiorakennetta. Kansiorakenne on tässä tapauksessa hyvin yksinkertainen; pelin assetit on jaettu neljään eri alikansioon: *Materials*, *Prefabs*, *Scenes* ja *Scripts*. Näistä oleellisin on pelin skriptit sisältävä kansio Scripts, joka sisältää kaksi C#-skriptiä *cubeRotate* ja c*ubeSpawner*, jotka ovat vastuussa kuutioiden instantiaatiosta ja kiertämisestä. Pelin näkymä eli scene löytyy Scenes kansion alta. Sekä kansio että sen alla oleva näkymä ovat molemmat esigeneroituja projektin luomisen yhteydessä. Näkymän hierarkia on kuvattuna kuvassa Kuva 11. *Main Camera* on pelin ensisijainen kamera, joka on vastuussa pelinäkymän esittämisestä eli siitä mitä pelaaja näkee. *Directional Light* on näkymään esigeneroitu valolähde, mikä vaikuttaa peliobjektien varjostukseen. *SpawnArea* on peliobjekti, joka ei sisällä muuta tietoa kuin sijainnin ja *Collider*-komponentin, jota käytetään kuutioiden spawnausalueen määritykseen. *SpawnArean* tavoin *Spawner*-peliobjekti on minimaalinen objekti, joka sisältää ainoastaan *CubeSpawner*-skriptin, jolloin skripti voidaan suorittaa heti pelinäkymän käynnistyessä. Hierarkiassa näkyvä *Canvas* on alue, johon piirretään UI (User Interface, käyttöliittymä) elementit eli tässä tapauksessa vain kuutioiden lukumäärää ylläpitävä teksti. Viimeisenä hierarkiassa on *EventSystem* esigeneroitu objekti, joka on vastuussa käyttäjän syötteen kuten hiiren tai näppäimistön painalluksista. Teknologiademon tapauksessa, sitä ei kuitenkaan käytetä ja se voidaan sivuuttaa tai poistaa, jos se nähdään tarpeelliseksi.

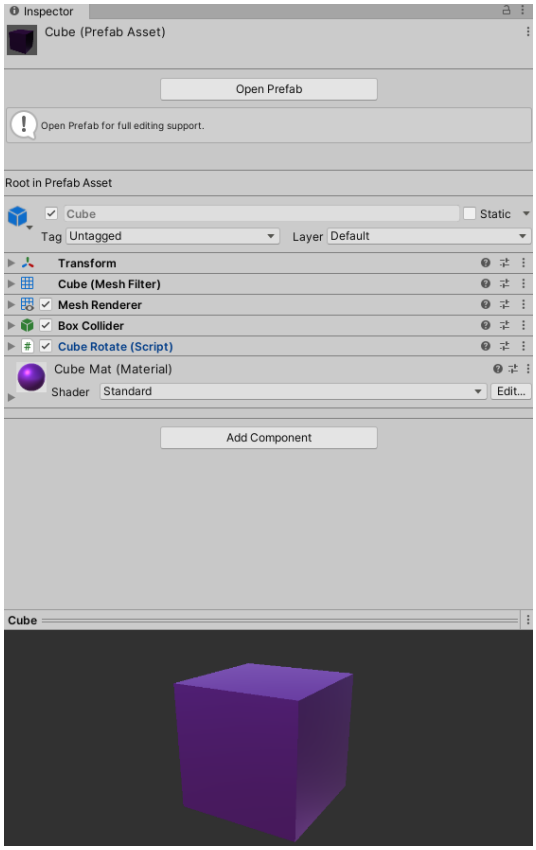


Kuva 10. Perinteinen Unity, teknologiademon kansiorakenne.

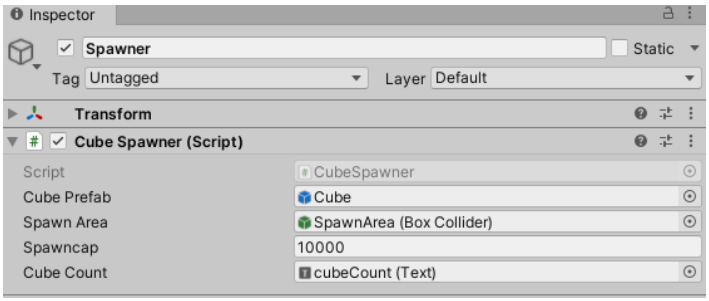


Kuva 11. Perinteinen Unity, näkymän hierarkia.

Materials kansio sisältää materiaalin, jota käytetään kuution pinnan määrityksessä. Kuution määritys löytyy *Prefabs* kansion alta nimellä *Cube*. Edellä mainittu *Cube* on Unityn termeissä *Prefab*, jota voidaan ajatella eräänlaisena mallina peliobjektille, jonka avulla voidaan luoda peliobjekteja skriptin sisällä. Kuvan Kuva 12 mukaisesti *Cube*-prefab sisältää myös sijainnin, fysiikan ja muodon lisäksi *CubeRotate* skriptin, mikä tarkoittaa, että kaikki tästä prefabista luodut peliobjektit sisältävät myös kiertämisestä vastuussa olevan koodin.



Kuva 12. Perinteinen Unity, kuutio- (Cube) prefab.



Kuva 13. Perinteinen Unity, Spawner-peliobjekti.

Alla olevassa kuvassa Kuva 14 on kuvattuna äärimmäisen yksinkertainen skripti *CubeRotate*, joka on vastuussa isäobjektinsa kiertämisestä. Tässä tapauksessa, kuten kuvassa Kuva 12 ilmenee, isäobjekti on *Cube*-Prefabista luotu peliobjekti. *CubeRotat*e-skripti sisältää *FixedUpdate*-metodin, joka on MonoBehaviour-luokan kuvan päivitysnopeudesta riippumaton metodi, joka soveltuu erityisesti fysiikoiden laskemiseen. Vaihtoehtoinen jokaisella kuvalla

kutsuttava metodi on nimeltään *Update*. Kiertämisen toteutus Update-metodilla on mahdollista, mutta tällöin, jotta kappaleen kiertämien on jatkuvaa ja päivitysnopeudesta riippumatonta, pitää meidän ottaa yksittäisten kuvien välinen aika huomioon *Time.deltaTime* avulla. Kappaleen kiertäminen tapahtuu *Rotate*-metodin avulla. Parametrina *Rotate* ottaa kolme arvoa, jotka ovat kiertämisen suuruus Eulerin kulmina.

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

public class CubeRotate : MonoBehaviour
{
    void FixedUpdate()
    {
        this.transform.Rotate(1, 1, 1);
    }
}
```

Kuva 14. CubeRotate C# skripti kuutioiden pyörittämiseen.

Kuvassa Kuva 15 on kuvattuna kuutioiden spawnaamiseen käytetty skripti. Jokaisella kuvapäivityksellä skripti instantioi uuden kuution pelin näkymään sekä päivittää UI:n cubeCount-tekstikentän arvon, kunnes saavutetaan spawncap-muuttujan osoittama kattoarvo. Kuution sijainti valitaan sattumanvaraisena koordinaattina *Collider*-komponentin osoittamien rajojen sisältä. Saavutettuaan kattoarvon skripti jatkaa yhä toimintaansa, vaikkakin if-lauseen ehto ei enää täyty eikä siten varsinaista laskentaa tai prosessointia tapahdu.

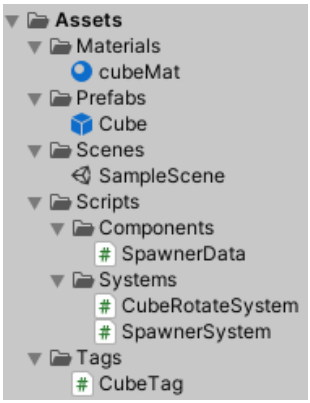
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class CubeSpawner : MonoBehaviour
{
    public GameObject cubePrefab;
    public Collider spawnArea;
    public int spawncap = 10;

    public Text cubeCount;

    private int cubecount = 0;
    private Bounds bounds;
    private Vector3 spawnCoordinates;

    void Start()
    {
        bounds = spawnArea.bounds;
    }
    void Update()
    {
        if (cubecount < spawncap)
        {
            Spawn();
            cubecount++;
            cubeCount.text = cubecount.ToString();
        }
    }

    void Spawn()
    {
        spawnCoordinates = new Vector3(Random.Range(bounds.min.x, bounds.max.x), Random.Range(bounds.min.y,
            bounds.max.y), Random.Range(bounds.min.z, bounds.max.z));

        Instantiate(cubePrefab, spawnCoordinates, Quaternion.identity);
    }
}
```

Kuva 15. CubeSpawner C#-skripti kuutioiden luomiseen.

3.4 Unity Tiny

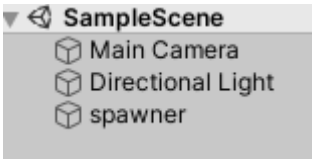
Toinen vertailtava teknologiademo rakennetaan Unityn Tiny-pakettia hyödyntäen. Tämä tarkoittaa käytännössä, että demon rakentamisessa käytetään ECS systeemin periaatteita ja Unityn DOTS systeemin kirjastoja ja ominaisuuksia. Kansiorakenteeltaan projektit ovat hyvin samankaltaiset. Kuten kuvassa Kuva 16 ilmenee, ainut varsinainen eroavaisuus on skriptien kansiorakenteessa; Tiny-projektissa skriptit on jaoteltu kahteen alaluokkaan: *Components* ja *Systems*. Jaottelu johtuu siitä, että unityn DOTS ei salli skriptin muuttujien arvojen muokkaamista editorin kautta – ei edes *public*-näkyvyysmääreen avulla. Components-kansion

alla olevat komponentit ovat siis puhtaasti tietovarastoja, kun taas Systems-kansion alla olevat C#-skriptit sisältävät logiikkaa.

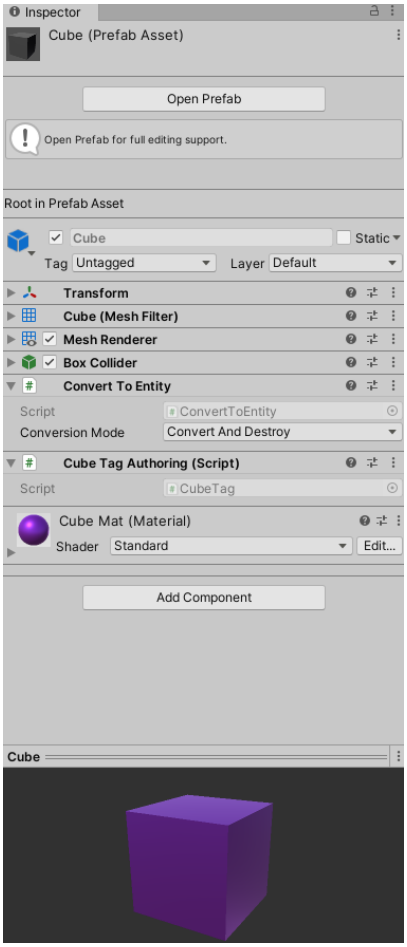


Kuva 16. Unity Tiny, teknologiademon kansiorakenne.

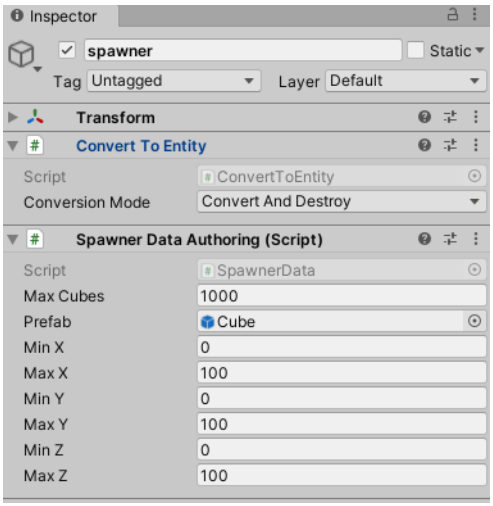
Myös pelin näkymä on hyvin samankaltainen, mutta hieman minimaalisempi. *SpawnArea*-objektin sijasta käytämme objektien instantiaation rajat manuaalisesti SpawnerData-skriptissä ja *Canvas* on tässä versiossa kadonnut kokonaan, sillä UI nikkarointi on Unityn DOTS:n kautta vielä toistaiseksi hyvin kankeaa ilman ulkoisia kirjastoja. Kuten kappaleessa *2.2 Unity Data-Oriented Technology Stack (DOTS)* mainittiin, komponentit eivät voi ECS arkkitehtuurissa sisältää skriptejä. Kuitenkin, jos katsomme kuvaa Kuva 18, niin Cube-prefab näennäisesti sisältää skriptin. Tämä johtuu siitä, että kyseinen *CubeTag*-skripti on *Authoring Component*, joka määritetään C#-skriptin sisällä kuvan Kuva 20 mukaisesti attribuutilla *GenerateAuthoringComponent*. Kyseinen komponentti ei voi sisältää muuta kuin puhdasta dataa – vaikkakin tässä tapauksessa se ei sisällä edes sitä, vaan sitä käytetään nimensä mukaisesti vain tagina. Toinen esimerkki samantyyppisestä komponentista on kuvan Kuva 21 *Spawner*-entiteetin alla oleva SpawnerData-komponentti. Tässä tapauksessa komponentti sisältää tiedon spawnattavien entiteettien lukumäärästä, viittauksen kuution Cube-prefabiin ja alueen minimi- ja maksimikoordinaatit.



Kuva 17. Unity Tiny, näkymän hierarkia.



Kuva 18. Unity Tiny, kuutio- (Cube) prefab.



Kuva 19. Unity Tiny, Spawner-entiteetti.

```
using Unity.Entities;

[GenerateAuthoringComponent]
public struct CubeTag : IComponentData
{

}
```

Kuva 20. CubeTag authoring component C#-skripti.

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

[Serializable]
[GenerateAuthoringComponent]
public struct SpawnerData : IComponentData
{
    public int maxCubes;
    public Entity prefab;
    public int minX, maxX;
    public int minY, maxY;
    public int minZ, maxZ;
}
```

Kuva 21. SpawnerData-komponentti.

Kuvan Kuva 22 CubeRotateSystem vastaa perinteisen Unityn CubeRotate-skriptiä. Skripti on muuten lähes sama, mutta *FixedUpdate* sijasta käytämme nyt *OnUpdate*-metodia, jonka takia joudumme ottamaan epäsäännölliset metodikutsut huomioon *Time.DeltaTime* kautta.

Käytämme myös *Entities.ForEach* entiteettien iteroimiseen ja kuvauksen mukaisten entiteettien löytämiseksi. Skriptissä siis etsitään entiteettejä, joilla on *CubeTag*- ja *Rotation*-komponentit ja kierretään kaikkia entiteettejä, jotka sopivat näihin kriteereihin.

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

public class CubeRotateSystem : SystemBase
{
    protected override void OnUpdate()
    {
        float dt = Time.DeltaTime;

        Entities.WithAll<CubeTag>().ForEach((ref Rotation rotation) => {
            rotation.Value = math.mul(rotation.Value, quaternion.EulerXYZ(2 * dt));
        }).Schedule();
    }
}
```

Kuva 22. CubeRotateSystem C#-skripti kuutioiden pyörittämiseen.

Kuvassa Kuva 23 on kuvattu perinteisen Unityn *CubeSpawner*-skriptiä vastaava *SpawnerSystem*-systeemi. Toisin kuin kuvan Kuva 22 CubeRotateSystem, ei tämän systeemin koodia suoriteta asynkronisesti, vaan kaikki kuutiot instantioidaan pääsäikeen toimesta. Tästä huolehtivat *AlwaysSynchronizeSystem*-attribuutti ja *Run*-metodi, joka suoritetaan välittömästi pääsäikeellä. Toiminnaltaan systeemi toimii kuten perinteisen Unityn vastine; systeemi arpoo koordinaatit ja instantioi koordinaatteihin uuden kuution, kunnes saavutetaan käyttäjän määrittämä maksimiarvo.

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

[AlwaysSynchronizeSystem]
public class SpawnerSystem : SystemBase
{
    // All spawning done on main thread.
    private bool running = true;

    private float3 spawnCoordinates;
    private int cubeCount = 0;
    protected override void OnUpdate()
    {
        if (running)
        {
            EntityManager entityManager = EntityManager;
            Random random = new Random();
            var seed = new System.Random();
            random.InitState((uint)seed.Next() * 1000);

            Entities.ForEach((ref Entity entity, in SpawnerData spawnerData) =>
            {
                Spawn(random, spawnerData, entityManager);
                cubeCount++;

                if (cubeCount > spawnerData.maxCubes)
                    running = false;
            }).WithStructuralChanges().Run();
        }
    }

    void Spawn(Random random, SpawnerData spawnerData, EntityManager entityManager)
    {
        spawnCoordinates = new float3(random.NextFloat(spawnerData.minX, spawnerData.maxX),
            random.NextFloat(spawnerData.minY, spawnerData.maxY), random.NextFloat(spawnerData.minZ,
            spawnerData.maxZ));
        Entity entity = entityManager.Instantiate(spawnerData.prefab);
        entityManager.SetComponentData(entity, new Translation {Value = spawnCoordinates });
    }
}
```

Kuva 23. SpawnerSystem C#-skripti kuutioiden luomiseen.

4 TULOKSET

Suorituskyvyn vertailu suoritettiin kahden teknologiademon välillä, joista toinen on rakennettu perinteisellä Unitylla ja toinen Unityn Tiny pakettia ja ECS kehitysperiaatteita noudattaen. Tarkemmat kuvaukset testiartefakteista löytyvät kappaleista *3.3 Perinteinen Unity ja MonoBehaviour* ja *3.4 Unity Tiny*. Tulosten mittaamiseksi käytettiin kolmea eri testijoukkoa, joista jokaisessa instantioidaan ja operoidaan eri määrä objekteja. Testijoukot ovat kooltaan 500 -, 2000 - ja 5000 objektia. Objektimäärän katon (5000 objektia) pääasiallisena valintaperusteena toimi perinteisen Unityn suorituskyvyn pullonkaula, jolloin objektien instantiaation ollessa FPS sidonnainen, pitäisi testiaikaa pidentää objektimäärän ja vertailukelpoisten tulosten saavuttamiseksi. Jokaiselle testille suoritetaan kolme otantaa, joista jokainen on kestoltaan viisi minuuttia ja otantojen välissä on viiden minuutin taukoperiodi.

Unityn Profilerin rajoitusten takia, otantojen mittausikkuna sisältää vain 300 viimeisintä suorituksen aikana tallennettua kuvaa. Vaikka mittausikkunan kokoa on mahdollista kasvattaa, se lisää Profilerin laitteistoresurssien - ja erityisesti muistin käyttöä, eikä siten sen perusteeton kasvattaminen ole erityisesti suorituskykyä mitattaessa kannattavaa. Koska demoa suoritetaan editorin sisäisesti *Play*-moodin kautta, vaikuttaa Unityn editorin käyttämät resurssit suorituskyvyn arviointiin. Unityn suositusten mukaisesti ja editorin vaikutusten minimoimiseksi on otantojen aikana kaikki muut Unityn ikkunat suljettu paitsi projekti- ja pelinäkymät. Samasta syystä myös Profiler käynnistetään omana erillisenä prosessinaan (engl. Standalone Process). Lisäksi, jotta sovelluksen resoluutio olisi lähempänä laitteen omaa resoluutiota, käynnistetään Play-moodi koko näytön tilassa, jolloin pystymme keräämään realistisempaa dataa erityisesti GPU:n piirtonopeuteen liittyen. (Unity Technologies, 2021h)

Mittausdataa kerättiin kolmen pääasiallisen metriikan kautta, jotka ovat CPU:n vasteaika, muistin käyttöaste ja GPU:n vasteaika. Tulokset laskettiin jokaiselle testijoukolle otantojen keskiarvoina. Mittaukset ovat kirjattuna taulukoissa Taulukko 4 ja Taulukko 5. Mittausten perusteella Perinteisen Unityn osalta muistin käyttöasteessa ei ole merkittäviä eroja ja muistinkäyttö pysyy lähes vakiona. GPU:n vasteaika kasvaa lievästi objektien määrän lisääntyessä. Kuitenkin CPU:n vasteajassa erot ovat huomattavasti suurempia; objektimäärän

kymmenkertaistuessa 500 objektista 5000 objektiin vasteaika on lähes kahdeksankertainen. Vastaavasti, jos verrataan tuloksia Unity Tinyn mittauksiin huomataan samanlainen trendi muistin- ja GPU:n käytössä, joskin GPU:n vasteaika pysyy huomattavasti matalampana suurilla objektien lukumäärillä. Isoin ero näkyy CPU:n vasteajassa, jonka mukaan objektimäärän kymmenkertaistuessa on Tinyn CPU:n vasteaika vain 1.5-kertainen.

Taulukko 4. Perinteinen Unity, otosjoukkojen keskiarvot.

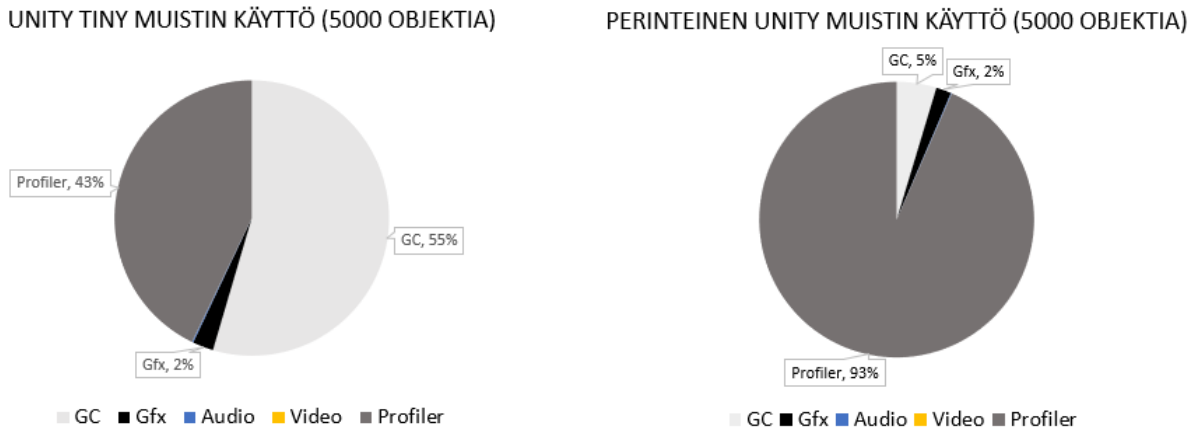
Metriikka	500 objektia	2000 objektia	5000 objektia
CPU vasteaika	7.169 ms	16.232 ms	55.341 ms
kokonais-muistinkäyttö	0.73 GB	1.02 GB	1.35 GB
varattu muisti	0.90 GB	1.09 GB	1.42 GB
järjestelmän muistinkäyttö	1.40 GB	1.62 GB	1.98 GB
GPU vasteaika	4.5 ms	6.5 ms	16 ms

Taulukko 5. Unity Tiny, otosjoukkojen keskiarvot.

Metriikka	500 objektia	2000 objektia	5000 objektia
CPU vasteaika	8.957 ms	10.847 ms	13.184 ms
kokonais-muistinkäyttö	1.07 GB	0.93 GB	1.07 GB
varattu muisti	1.47 GB	1.39 GB	1.42 GB
järjestelmän muistinkäyttö	2.14 GB	1.99 GB	2.03 GB
GPU vasteaika	4 ms	5 ms	7 ms

Muistin ja GPU:n profilointi suoritettiin CPU testauksesta erillisinä testeinä. GPU profiloinnin mahdollistamiseksi testien aikana Unityn *graphics jobs* on poissa käytöstä, sillä Unityn Profiler ei kykene keräämään GPU:n suorituskyvyn dataa asetuksen ollessa päällä (*Project Settings* > *Player* > *Other Settings* > *Graphics Jobs*). Grafiikka jobien lisäksi tuloksia tulkitessa on huomioitava, että GPU profilointi itsessään varaa enemmän järjestelmätehoja käyttöönsä ja sitä kautta heikentää testin aikaista suorituskykyä. Vaikka editorin ja profilerin tarkkaa vaikutusta mittaustuloksiin on hankala arvioida, voidaan tuloksien tulkinnassa niiden kokonaisvaikutusta spekuloida erityisesti CPU:n ja muistinkäytön perusteella; CPU profiloinnissa Unityn editorin toiminnot sisältyvät *EditorLoop*-prosessin alle, kun taas muistin profiloinnissa Profilerin käyttämä muisti näkyy *Profiler-* tunnuksen alla. Kuvassa Kuva 24 ja taulukossa

Taulukko 6 on esitetty Perinteisen Unityn ja Tinyn muistin käytön jakautuminen eri prosesseille. Verrattaessa Perinteisen Unityn ja Tinyn muistin käyttöä 5000 objektin koejoukolla on erityisesti Profilerin ja GC:n (muistinsiivous, engl. Garbage Collection) muistin käytössä eroavaisuuksia. Syy Tinyn moninkertaiselle GC:n muistin käytölle voi olla esimerkiksi Tinyn suorittama automaattinen optimointi eri tekniikoiden kuten *frustum culling* tai *occlusion culling* kautta, mikä ilmenee niiden objektien poistamisena, jotka eivät näy pelaajan ruudulla. Vaikka mittaukset näyttävät merkittävän määrän varattua muistia GC:n toimesta, on tämä todennäköisesti vapautettavissa, kun sitä tarvitaan. Profilerin muistin käyttö voidaan puolestaan sivuuttaa täysin, sillä se on olemassa vain mittausdatan keräämiseksi.



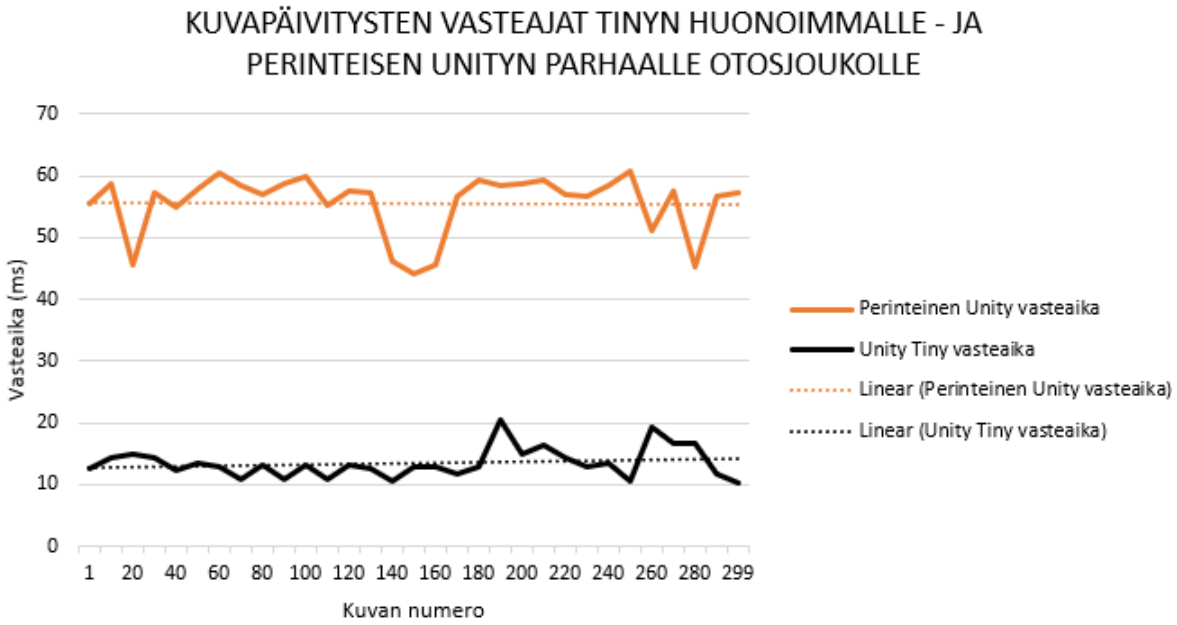
Kuva 24. Perinteisen Unityn ja Unity Tinyn muistin käyttö, 5000 objektia.

Taulukko 6. Unity Tinyn ja perinteisen Unityn prosessikohtainen muistinkäyttö.

Prosessi	Unity Tiny	Perinteinen Unity
GC	460.6 MB	56.2 MB
Gfx	20.8 MB	21.7 MB
Audio	1.2 MB	1.2 MB
Video	264.0 B	264.0 B
Profiler	361.4 MB	1123.3 MB

5 POHDINTA JA TULEVAISUUS

Kaiken kaikkiaan voidaan mittaustuloksista tehdä johtopäätös, että Perinteisen Unityn suorituskyky on Tinya parempi pienillä samanaikaisesti operoitavien objektien lukumäärillä, kun taas Tiny suoriutuu huomattavasti paremmin, mikäli objektimäärä on suuri. Tulosten vakauttamiseksi voimme vielä ottaa kärjistetyn kuvan Kuva 25 mukaisen esimerkin, jossa vertaamme perinteisen Unityn suorituskyvyltä parasta otantaa Unity Tinyn huonoimpaan otantaan. Epäreilusta asettelusta huolimatta tulosten perusteella Tiny on Perinteistä Unitya keskimäärin noin 43 ms edellä, kun vertaillaan CPU:n vasteaikoja 5000 objektilla. Kyseinen ero vasteajassa vastaa noin 59 yksikön eroa kuvataajuudessa. Myös GPU:n vasteaika on Tinylla huomattavasti parempi, kun eri peliobjektien lukumäärä on suuri. Esimerkkinä tästä on 5000 objektin testijoukko, millä Tinyn GPU:n vasteaika oli noin 129 prosenttia tai 80 FPS pienempi. Tulosten perusteella Perinteinen Unity on kuitenkin Tinya edellä pienemmillä objektien lukumäärillä sekä CPU:n suorituskyvyssä että muistin käytössä.



Kuva 25. Kuvapäivitysten vasteajat Unity Tinyn huonoimmalle - ja perinteisen Unityn parhaalle otosjoukolle, 5000 objektia.

Tutkimuksessa suorituskyvyn vertailussa käytettiin vain yhtä entiteettien arkkityyppiä ja siten entiteettien etsiminen muistista on väistämättä suoraviivaista. Tulevaisuudessa olisikin suotavaa, että suorituskyvyn vertailussa testataan myös eri arkkityyppien lukumäärän vaikutusta erityisesti Unity Tinyn suorituskykyyn. On myös huomioitava, että tässä tutkimuksessa painoarvo ei ollut itse ohjelmakoodin optimoimisessa; vaikka Unity Tiny ja ECS pakottavat kehittäjää noudattamaan tiettyjä käytäntöjä esimerkiksi rajoittamalla sallittuja tietotyyppejä, on tulevassa tutkimuksessa tilaa erityisesti ohjelmakoodin optimoinnille. Tämä voi olla esimerkiksi tyhjien Update-metodikutsujen välttämistä deaktivoimalla systeemeitä, kun ne eivät ole enää tarpeellisia, tai päivittämällä objekteja vain, jos ne näkyvät pelaajan ruudulla tai sen läheisyydessä. Ohjelmakoodin optimoinnin lisäksi suorituskyvyn tutkimuksessa voitaisiin tutkia eri optimointitekniikoiden kuten piilotetun pinnan poistamisen (engl. Occlusion culling) ja objektien ennenaikaisen instantiaation (engl. Object pooling) vaikutusta suorituskykyyn.

6 YHTEENVETO

Tutkimuksessa arvioimme Unityn Tiny-paketin ja DOTS:n vaikutusta pelin suorituskykyyn. Tutkimus toteutettiin kahden vertailukelpoisen teknologiademon kautta, joista toinen tehtiin Perinteisellä Unitylla ja toinen Tiny-pakettia hyödyntäen. Demojen vertailu suoritettiin Unityn sisäisen profilointityökalun Unity Profilerin avulla ja vertailussa keskityttiin kolmeen metriikkaan: CPU:n vasteaika, GPU:n vasteaika ja muistin käyttöaste. Tutkimuksen tulokset osoittivat, että Unity Tiny suoriutuu perinteistä Unitya huomattavasti paremmin, kun pelinäkymän samanaikaisten objektien määrä on suuri. Kuitenkin perinteinen Unity on ainakin toistaiseksi parempi valinta, kun objektien määrä on pieni. Kokonaisuudessaan Tinyn huomattavista suorituskyvyn eduista huolimatta Tiny ei sellaisenaan ole vielä täysin käyttövalmis, mikä näkyykin jo siitä, että sitä ei ole listattu Unityn pakettien alla Unityn uudemmissa versioissa (v2021.1 ja ylöspäin). Lisäksi osa toiminnoista kuten UI:n rakentaminen ja manipulaatio on tällä hetkellä hankalaa ilman suurempaa henkistä ponnistusta, eikä siten nykytilassaan sovellu kovinkaan hyvin Unityn kanssa aloittelevalle kehittäjälle. Kuitenkin, mikäli suorituskyky tai erikoisalueet kuten pelattavat mainokset ovat peliprojektin pääasiallisena tavoitteena, voi Unity Tiny -paketista ja Unityn DOTS systeemistä olla hyötyä. Toistaiseksi kuitenkin on suositeltavaa odottaa, että Unity saa ratkaistua DOTS:n suurimmat ongelmat, kuten yhteensopivuuden uudempien Unityn versioiden kanssa, ennen kuin aloittaa suurempia projekteja Tinylla.

LÄHTEET

AMD, 2021. AMD “Zen 3” Core Architecture [Verkkoaineisto]. Saatavissa: https://www.amd.com/en/technologies/zen-core-3. [Viitattu 12.5.21].

Bilas, S., 2002. A Data-Driven Game Object System 1–41.

Blake, G., Dreslinski, R.G., Mudge, T., 2010. Evolution of thread-level parallelism in desktop applications 302–312.

Borufka, R., 2020. Performance Testing Suite for Unity DOTS 15–57.

Ferreira, C., Geig, M., 2018. Get Started with the Unity* Entity Component System (ECS), C# Job... [Verkkoaineisto]. Intel. Saatavissa: https://www.intel.com/content/www/us/en/develop/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler.html. [Viitattu 19.5.21].

Gao, C., Gutierrez, A., Rajan, M., Dreslinski, R.G., Mudge, T., Wu, C.-J., 2015. A study of mobile device utilization. Teoksessa: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Esitetty: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 225–226. https://doi.org/10.1109/ISPASS.2015.7095808

Garcia, F.E., de Almeida Neris, V.P., 2014. A Data-Driven Entity-Component Approach to Develop Universally Accessible Games. Teoksessa: Stephanidis, C., Antona, M. (Eds.), Universal Access in Human-Computer Interaction. Universal Access to Information and Knowledge, Lecture Notes in Computer Science. Springer International Publishing, Cham, p. 540. https://doi.org/10.1007/978-3-319-07440-5_49

Gestwicki, P., 2012. The entity system architecture and its application in an undergraduate game development studio. Teoksessa: Proceedings of the International Conference on the Foundations of Digital Games - FDG ’12. Esitetty: the International Conference, ACM Press, Raleigh, North Carolina, p. 74. https://doi.org/10.1145/2282338.2282356

Gram, M., 2021. Unity - Notice on DOTS compatibility with Unity 2021.1 [Verkkoaineisto]. Unity Forum. Saatavissa: https://forum.unity.com/threads/notice-on-dots-compatibility-with-unity-2021-1.1091800/. [Viitattu 27.5.21].

ISO/IEC 25010:2011, 2011. ISO/IEC 25010:2011 [Verkkoaineisto]. ISO. Saatavissa: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/57/35733.html. [Viitattu 4.5.21].

Koushanfar, F., Prabhu, V., Potkonjak, M., Rabaey, J.M., 2000. Processors for mobile applications. Teoksessa: Proceedings 2000 International Conference on Computer Design. Esitetty: Proceedings 2000 International Conference on Computer Design, p. 603. https://doi.org/10.1109/ICCD.2000.878354

Kumar, S., 2015. Fundamental Limits to Moore’s Law 3.

Leonard, T., 1999. Postmortem: Thief: The Dark Project [Verkkoaineisto]. Saatavissa: https://www.gamasutra.com/view/feature/131762/postmortem_thief_the_dark_project.php. [Viitattu 5.5.21].

Meijer, L., 2019. On DOTS: Entity Component System - Unity Technologies Blog [Verkkoaineisto]. Saatavissa: https://blogs.unity3d.com/kr/2019/03/08/on-dots-entity-component-system/. [Viitattu 11.5.21].
Messaoudi, F., Ksentini, A., Simon, G., Bertin, P., 2017. Performance Analysis of Game Engines on Mobile and Fixed Devices. ACM Trans. Multimed. Comput. Commun. Appl. 13, 1–28. https://doi.org/10.1145/3115934
Peterson, K., Behunin, S., Graham, F., 2014. Performance metrics gathering from multiple video game platforms. US8788243B2.
Rachel Ballard, Martin Best, 2018. Project Tiny Preview Package is here! - Unity Technologies Blog [Verkkoaineisto]. Saatavissa: https://blogs.unity3d.com/cn/2018/12/05/project-tiny-preview-package-is-here/. [Viitattu 4.5.21].
Suggs, D., Subramony, M., Bouvier, D., 2020. The AMD "Zen 2" Processor. IEEE Micro 40, 49–50. https://doi.org/10.1109/MM.2020.2974217
Turpeinen, M., 2020. A Performance Comparison for 3D Crowd Rendering using an Object-Oriented system and Unity DOTS with GPU Instancing on Mobile Devices. 9–10.
Unity, 2019. Understanding data-oriented design for entity component systems - Unity at GDC 2019.
Unity, 2016. Unite Europe 2016 - ECS architecture with Unity by example.
Unity Technologies, 2021a. Using Entities.ForEach | Entities | 0.17.0-preview.41 [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_entities_foreach.html. [Viitattu 4.5.21].
Unity Technologies, 2021b. Entities | Entities | 0.17.0-preview.41 [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_entities.html. [Viitattu 10.5.21].
Unity Technologies, 2021c. Components | Entities | 0.17.0-preview.41 [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_components.html. [Viitattu 11.5.21].
Unity Technologies, 2021d. Unity - Manual: C# Job System [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/2021.2/Documentation/Manual/JobSystem.html. [Viitattu 14.5.21].
Unity Technologies, 2021e. Unity - Scripting API: ReadOnlyAttribute [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/2021.2/Documentation/ScriptReference/Unity.Collections.ReadOnlyAttribute.html. [Viitattu 17.5.21].
Unity Technologies, 2021f. Unity.Burst.Intrinsics | Burst | 1.6.0-pre.2 [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.burst@1.6/manual/docs/CSharpLanguageSupport_BurstIntrinsics.html. [Viitattu 19.5.21].
Unity Technologies, 2021g. C#/.NET Language Support | Burst | 1.6.0-pre.2 [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.burst@1.6/manual/docs/CSharpLanguageSupport_Types.html. [Viitattu 19.5.21].

Unity Technologies, 2021h. Unity - Manual: Profiling your application [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Manual/profiler-profiling-applications.html. [Viitattu 31.5.21].
Unity Technologies, 2020a. Intro for Unity developers | Package Manager UI website [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.tiny@0.13/manual/intro-for-unity-developers.html. [Viitattu 4.5.21].
Unity Technologies, 2020b. Accessing entity data | Entities | 0.2.0-preview.18 [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.entities@0.2/manual/chunk_iteration.html. [Viitattu 4.5.21].
Unity Technologies, 2020c. Intro to Project Tiny & DOTS Mode for Unity developers | Package Manager UI website [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.tiny@0.16/manual/intro-for-unity-developers.html. [Viitattu 8.5.21].
Unity Technologies, 2020d. Introduction to ECS | Package Manager UI website [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.tiny@0.16/manual/introduction-to-ecs.html. [Viitattu 10.5.21].
Unity Technologies, 2020e. Profile Analyzer Window | Package Manager UI website [Verkkoaineisto]. Saatavissa: https://docs.unity3d.com/Packages/com.unity.performance.profile-analyzer@0.4/manual/profiler-analyzer-window.html. [Viitattu 27.5.21].
Vujošević-Janičić, M., Tošić, D., 2008. The role of programming paradigms in the first programming courses. Teach. Math. 21 72–73.
Waldrop, M.M., 2016. The chips are down for Moore’s law. Nat. News 530, 144. https://doi.org/10.1038/530144a
Wu, B., Wang, A.I., 2012. A Guideline for Game Development-Based Learning: A Literature Review. Int. J. Comput. Games Technol. 2012, 1–20. https://doi.org/10.1155/2012/103710