LAPIN AMK
Lapland University of Applied Sciences

UNITY-OPTIMOINTI

Aleksi Narkilahti

Opinnäytetyö

Tieto- ja viestintätekniikka
Insinööri (AMK)

2021

LAPIN AMK
Lapland University of Applied Sciences

Opinnäytetyön tiivistelmä

Tieto- ja viestintätekniikka
Insinööri (AMK)

Tekijä	Aleksi Narkilahti	Vuosi	2021
Ohjaaja	Juhani Angelva, Toni Westerlund		
Toimeksiantaja	FrostBit-ohjelmistolaboratorio		
Työn nimi	Unity-optimointi		
Sivumäärä	49		

Tässä opinnäytetyössä käsitellään Unity-pelimoottoriin liittyviä eri optimointitekniikoita ja menetelmiä C#-ohjelmakoodien, peligrafiikan sekä suorituskykyongelmien etsimisen kannalta. Optimoinnilla tässä opinnäytetyössä tarkoitetaan pelien tai muiden Unity-sovelluksien suorituskyvyn parantamista.

Opinnäytetyö lähti liikkeelle kaveriporukasta lähteneestä peli-ideasta. Opinnäytetyön rinnalla kehitettiin eteenpäin omaa peliprojektia, josta aiheeseen sai erinomaista käytännön kokemusta.

Unity-pelimoottorissa on useita optimointiin tarkoitettuja työkaluja, kuten Unity-profiloija sekä muita sisäänrakennettuja menetelmiä ja tekniikoita, joita pelien tai muiden Unity-sovelluksien optimoinnissa voidaan hyödyntää. Työssä pyrittiin käyttämään ajankohtaisia ja kehittäjäläheisiä lähteitä, mikä käytännössä tarkoitti, että iso osa lähteistä oli Unityn omasta dokumentaatiosta sekä sen blogikirjoituksista.

Avainsanat optimointi, peliohjelmointi, pelioptimointi, Unity, Unity-optimointi

LAPIN AMK
Lapland University of Applied Sciences

Abstract of Thesis

Degree Programme in Information and Communication Technology
Bachelor of Engineering

Author	Aleksi Narkilahti	Year	2021
Supervisor	Juhani Angelva, Toni Westerlund		
Commissioned by	FrostBit Software Lab		
Subject of thesis	Unity Optimization		
Number of pages	49		

The aim of this thesis was to study the different optimization techniques and methods related to the Unity game engine in terms of C# programming, Unity game graphics and finding out performance problems. Word optimization in this thesis refers to improving the performance of games or other Unity applications.

The Unity game engine has several built-in optimization tools, techniques, and methods, such as the Unity profiler, which were covered in detail with the help of small examples. The aim was to use up to date and developer-friendly sources, which in practice meant that a large part of the sources were from the Unity's own documentation and its blog posts.

Optimization in Unity games or other applications is a very large and time-consuming task because applications often have multiple separate sections, such as game graphics and program code. The own game project, which was developed alongside the thesis, provided excellent practical experience on the subject because performance problems were witnessed firsthand. Witnessing and solving these problems helped to deal with the subject and as a result, the aim of this thesis was well met.

Key words: game optimization, Unity, Unity optimization, game programming, optimization

SISÄLLYS

KÄYTETYT LYHENTEET JA TERMIT

DOTS	Data-Oriented Technology Stack
ECS	Entity Component System
FPS	frames per second, kuvaa sekunnissa
Frame	kehys, kuva
Frame rate	ruudunpäivitysnopeus
GC	Garbage Collector, roskankerääjä
GPU	Graphics Processing Unit, näytönohjain
LOD	Level of Detail
Main Thread	ohjelman pääsäie
Prefab	tallennettu peliobjekti komponentteineen
Pullonkaula	hitauttava osa-alue sovelluksessa
Render Thread	ohjelman renderöintisäie
Scene	Skene, Unity-sovelluksen kenttä
Script	Script, ohjelmakoodi
Shader	Shader-ohjelma, varjostin
UI	User Interface, käyttöliittymä
VR	Virtual Reality, virtuaalitodellisuus

1 JOHDANTO

Pelien ja muiden reaaliaikaisten sovelluksien optimointi on haasteellista, sillä yhden framen (kehyksen) aikana tapahtuu paljon asioita ja niiden suorittamiseen ei ole paljon prosessointiaikaa. Jotta pelattavuus tuntuu käyttäjille sulavalta, pitää sovelluksen toimia vähintään 30 FPS (frames per second). Nykyään kuitenkin peleissä tähdätään 60 FPS ja joillain alustoilla tämä vaatii vieläkin enemmän. Tämä tarkoittaa käytännössä, että jokainen yksittäinen frame pitää keskimäärin suorittaa 16,6 millisekunissa, jos sovelluksen halutaan toimivan 60 FPS.

Sovelluksien optimoinnista tekee haasteellisen myös se, että jokainen alusta, kuten VR tai mobiiliympäristöt eroavat muun muassa laitteiston tehokkuuksiltaan ja tavoitteiltaan. Esimerkiksi VR-sovelluksissa tähdätään vähintään 90 FPS, sillä muutoin sen käyttö voi aiheuttaa pahoinvointia osalla käyttäjistä. Mobiiliympäristöllä taas laitteiston suorituskyky on huomattavasti alhaisempi kuin esimerkiksi PC-alustalla. Mobiiliympäristöllä huomioon täytyy myös ottaa muun muassa akun kesto ja sovelluksen tiedostokoko. Optimoinnissa täytyy siis ymmärtää kohdealusta ja sen tuomat vahvuudet, heikkoudet ja tavoitteet.

Työ on tehty toimeksiantona Lapin ammattikorkeakoulun FrostBit-ohjelmistolaboratoriolle. FrostBit luo ja kehittää pelejä ja pelillistettyjä ympäristöjä monenlaisiin eri hankkeisiin. FrostBitin hankkeet usein hyödyntävät VR tai mobiiliympäristöjä, jotka ovat haasteellisia ympäristöjä optimoinnin kannalta. Työn tavoitteena oli tutkia ja selvittää nykyaikaisia optimointitekniikoita ja menetelmiä, minkä avulla FrostBit voisi luoda suorituskykyisempiä sovelluksia ja/tai yksityiskohtaisempia pelimaailmoja Unity-pelimoottorilla.

Pelien ja muiden sovelluksien optimointi on kokonaisuudessaan erittäin laaja aihe, minkä takia aihe rajattiin käsittelemään Unity-profiloijatyökalun käyttö suorituskykyongelmien etsimisessä, Unity-pelimoottorin ohjelmakoodien sekä Unity-peligrafiikan eri optimointitekniikoita ja menetelmiä. Opinnäytetyön rinnalla on myös kehitetty omaa peliprojektia, jossa eri optimointitekniikoita on käytännössä päässyt ja joutunut käyttämään, jotta pelin pelattavuus on pysynyt sulavana.

2 UNITY PROFILER

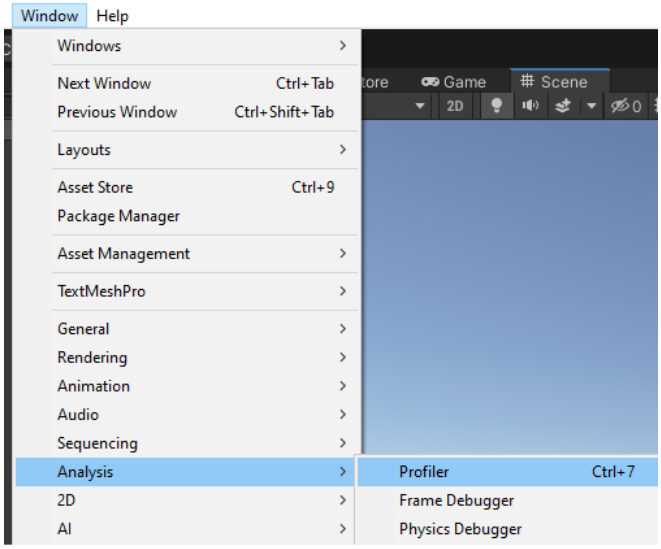
2.1 Unity Profiler -työkalun esittely

Unity-pelimoottorissa on sisäänrakennettu suorituskykyanalyysiin tarkoitettu profiloijatyökalu, jonka avulla kehittäjä pystyy tutkimaan sovelluksen suorituskykyä sovelluksen ollessa käynnissä Unity-editorissa tai suoritettavassa sovelluksessa. Profiloijan avulla pystytään tutkimaan esimerkiksi, kuinka kauan prosessorilla menee aikaa suorittaa tietty toimenpide, kuten Unity-pelimoottorin fysiikkalaskelmat tai tietyn ohjelmakoodin suorittaminen.

Profiloija on erittäin tärkeä ja hyödyllinen työkalu, koska sen avulla kehittäjä voi tutkia ja löytää suorituskykyongelmien lähtösyyt. Ilman työkalua, kuten Unity-profiloija, joutuisi kehittäjä arvioimaan, arvaamaan tai manuaalisesti ohjelmallisesti mittaamaan paljonko prosessointiaikaa mikäkin osio sovelluksesta vie, siten mahdollisesti käyttämään ylimääräistä aikaa suorituskykyongelmien etsimiseen tai optimoimaan osioita sovelluksesta, jotka eivät välttämättä vaatisi optimointia ollenkaan.

2.2 Unity Profiler -työkalun käyttö

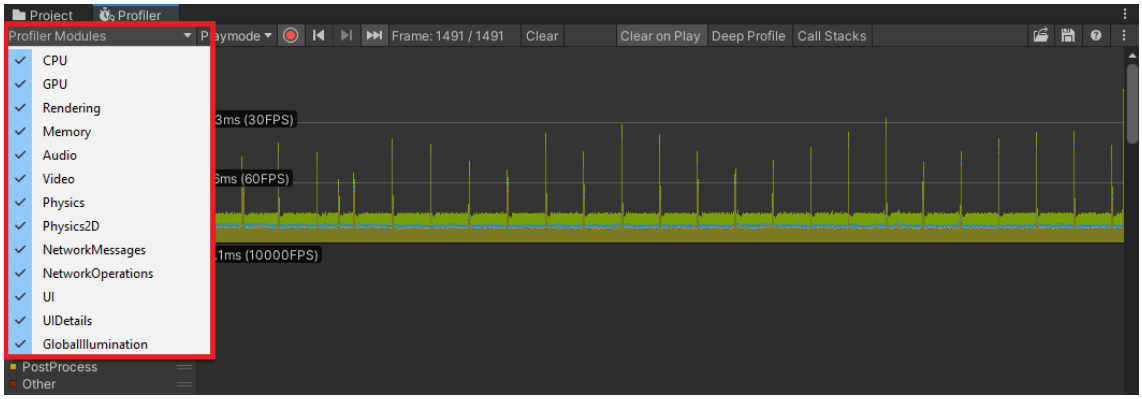
Unity-profiloijatyökalun saa käyttöön avaamalla Profiler-kohdan kuvion 1. mukaisesti tai oletuspikanäppäimellä painamalla Ctrl + 7 (Unity 2020a). Unity avaa uuden Profiler-ikkunan mikäli ikkuna ei ole jo auki. Profiler-ikkunaa voi siirrellä, suurentaa tai pienentää, kuten mitä tahansa muutakin ikkunaa Unity-editorissa.



Kuvio 1. Profiloijan käyttöönotto

Unity-profiloija tallentaa useita sovelluksen eri osa-alueita ja näyttää näiden suorituskykytiedot Profiler-ikkunassa. Näiden tietojen perusteella saadaan kattava tieto, mihin prosessointiaika on kulunut eri framejen välillä. Oletuksena profiloija tallentaa viimeisen 300 framen suorituskykytiedot, mutta tämän arvon pystyy nostamaan 2000 frameen Profiler-ikkunan asetuksista. (Unity 2020a.)

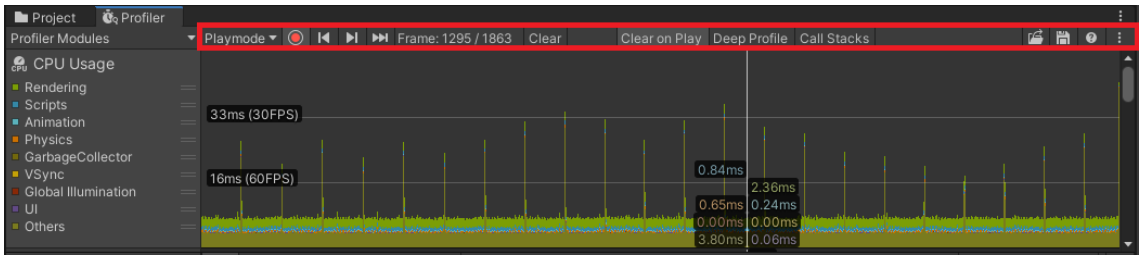
Profiler-ikkunan Profiler Modules -alasvetovalikosta voidaan valita mitä moduuleita profiloija näyttää ja mittaa sovelluksen aikana (Kuvio 2). Jotta profiloija alkaa mittaamaan tietoja halutuilta moduuleilta, pitää ne olla valittuna alasvetovalikosta. (Unity 2020a.)



Kuvio 2. Profiler Modules -alasvetovalikko

Kuviossa 3. nähdään profiloijan kontrollit, joilla voidaan navigoida profiloitujen framejen välillä ja kontrolloida minkä tasoista profilointia Unityn halutaan suorittavan. Esille tästä paneelista nousevat seuraavat valinnat.

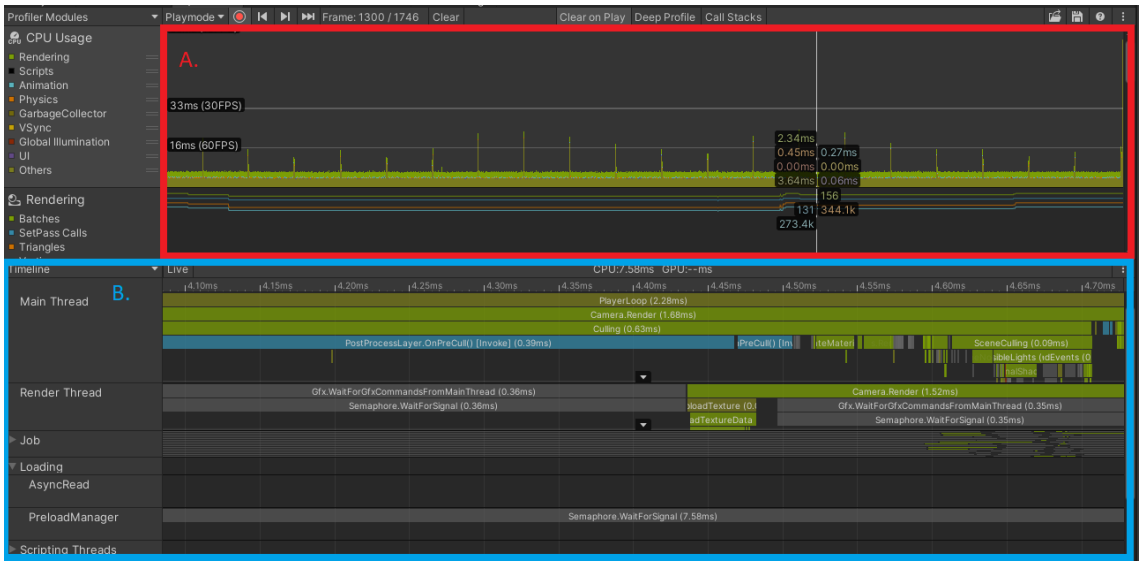
Record profiling information on punainen painike, jonka tulee olla aktiivisena, että profiloija voi aloittaa mittaamaan suorituskykyä. Tämä on automaattisesti päällä, kun profiloija avataan ensinmäisen kerran. **Navigation Arrows** -näppäimillä voidaan liikkua eri framejen välillä. Framejen välillä voi myös liikkua siirtämällä pystysuoraa valintaa. **Clear on play** on painike, joka tyhjentää Profiler-ikkunan suorituskykyanalyysi mittaukset, kun sovellus käynnistetään uudestaan. **Deep Profile** on syväprofilointinäppäin, joka ei ole automaattisesti päällä, koska sitä on huomattavasti raskaampaa suorittaa kuin normaalia suorituskykymittausta (Kuvio 3). Syväprofiloinnin ollessa päällä, Unity-pelimoottori luo ohjelmakoodeihin ja komponentteihin lisämetodeja, minkä avulla voidaan nähdä suorituskykytiedot melkein ohjelmakoodiriviä myöten. Syväprofiloinnilla näkee muun muassa ohjelmakoodien eri metodien ja funktioiden suorituskyvyn (Unity 2020a). Tämä on hyödyllinen, kun halutaan nähdä tarkasti mikä osio ohjelmakoodeissa tai komponenteissa vie eniten prosessointiaikaa.



Kuvio 3. Profiloijan kontrollit

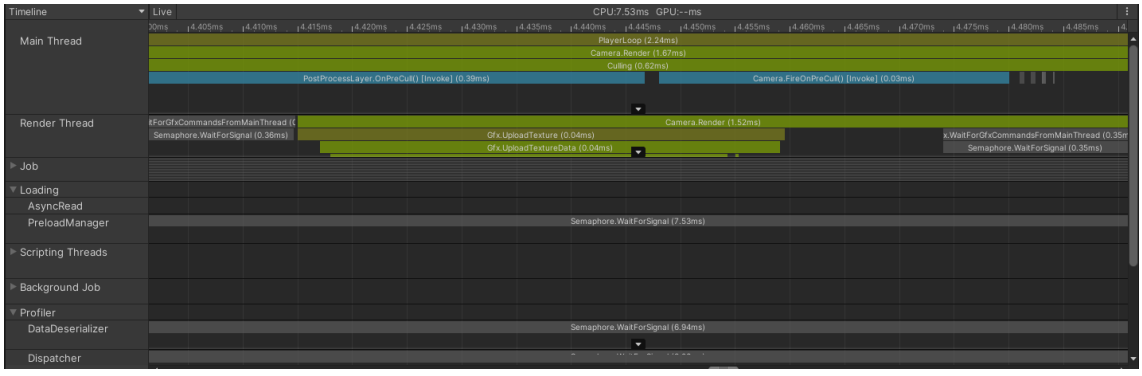
Kuvion 4. alueesta A. nähdään jokaisen moduulin kaaviot. Tähän alueeseen tulee informaatiota, kun sovellus käynnistetään. Tästä alueesta nähdään helposti mikäli profiloinnin aikana tapahtuu yksittäisiä piikkejä prosessoinnissa eli yhdellä tai useammalla framella tapahtuu enemmän prosessointia kuin muilla frameilla. Tämä voi tapahtua esimerkiksi silloin, kun suoritetaan jokin raskas ohjelmakoodin funktio tai metodi.

Kuvion 4. alueesta B. nähdään valittujen moduulien suorituskykymittaukset ja niiden profiloijamerkinnät. Tämän alueen tiedoista nähdään, paljonko mikäkin osio, komponentti tai ohjelmakoodi vie prosessointiaikaa.



Kuvio 4. Profiloijan kehyskaaviot ja moduulipaneelit

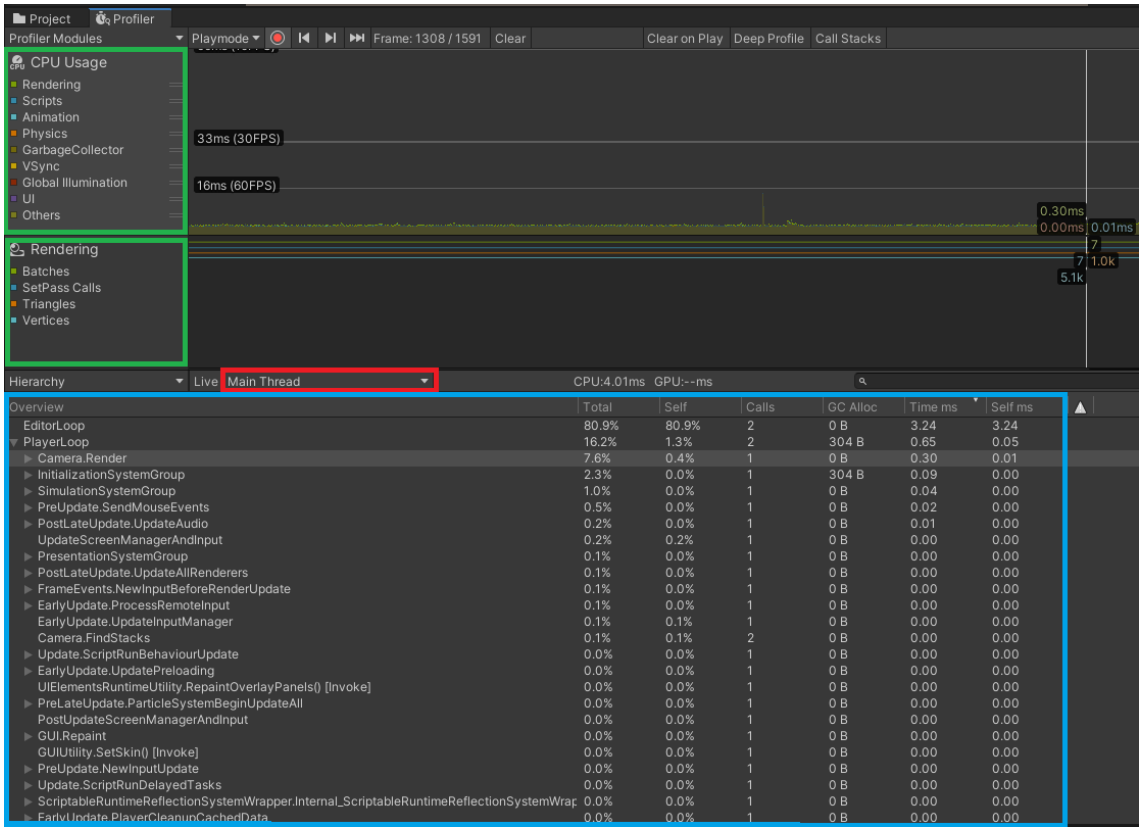
Unity-profiloijassa on kaksi erilaista päänäkymävaihtoehtoa, aikajana- sekä hierarkianäkymä. Näkymien erona on miten suorituskykytietoja esitetään Profiler-ikkunassa. Aikajananäkymä on kätevä, kun halutaan nähdä miltei kaikki profiloijan tiedot kerralla. Kuviossa 5. on valittuna aikajananäkymä.



Kuvio 5. Profiloijan aikajananäkymä

Hierarkianäkymästä nähdään kerrallaan vain yhden valitun moduulipaneelin tiedot. Kuviossa 6. on valittuna Main Thread eli ohjelman pääsäie ja prosessorimoduulipaneeli. Mikäli haluttaisiin tutkia muiden moduulien tietoja, niin silloin vihreällä olevista moduuleista klikkaamalla saadaan siniseen alueeseen näkyviin

valitun moduulin tiedot. Mikäli halutaan tutkia muiden säikeiden tietoja, niin silloin punaisesta alasvetovalikosta valitaan esimerkiksi Render Thread eli ohjelman renderöintisäie (Kuvio 6).

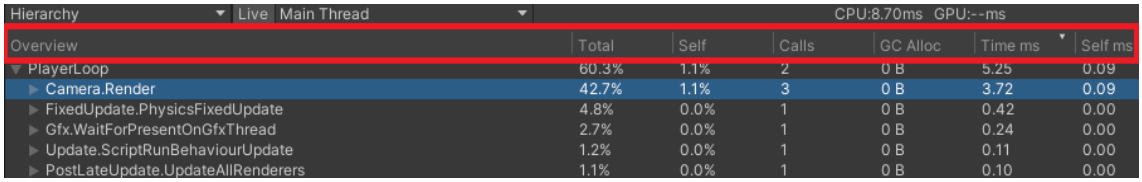


Kuvio 6. Profiloijan hierarkianäkymä

Hierarkianäkymän saraketiedoista nähdään esimerkiksi paljonko millekin profiloijamerkinnälle on kulunut prosessointiaikaa (Kuvio 7). Hierarkianäkymän saraketiedoista tärkeimmät kohdat ovat seuraavat.

Overview-sarakkeesta nähdään kaikki listatut profiloijamerkinnät. Eri moduulien mahdolliset alamerkinnät saa näkyviin klikkaamalla profiloijamerkinnän vasemmasta nuolesta. **Total**-sarakkeesta nähdään prosentuaalisesti käytetty aika profiloijamerkinnälle koko sovelluksen käyttämästä ajasta valitulla framella. **Calls**-sarakkeesta nähdään, kuinka monta kertaa prosessia on kutsuttu yhden framen aikana. Mikäli tässä kohdassa näkyy enemmän kuin yksi, niin prosessia tai ohjelmakoodia kutsutaan useammin kuin kerran yhden framen aikana. Tästä kohdasta näkee siis helposti, mikäli samaa ohjelmakoodia kutsutaan useasti (Kuvio 7).

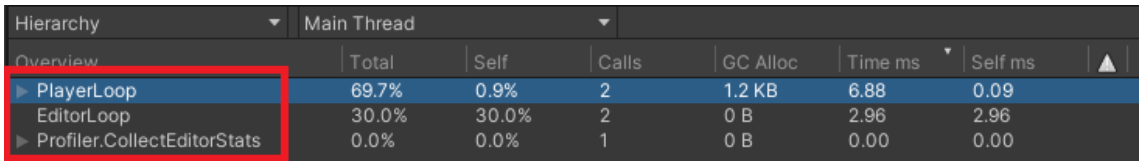
GC Alloc-sarakkeesta nähdään, kuinka paljon framella muistia varattiin. Tämä on hyvä pitää mahdollisimman alhaisena framea kohti, sillä muistin varaus ja poistaminen ovat suhteellisen raskaita operaatioita. **Time ms** -sarakkeesta nähdään millisekunteina käytetty aika profiloijamerkinnälle. Klikkaamalla tätä saraketta voi sen järjestystä muuttaa suurimmasta pienimpään tai toisinpäin (Kuvio 7).



Kuvio 7. Profiloijan hierarkianäkymän saraketiedot

2.3 Profiloijamerkinnät

Unity-profiloijatyökalu toimii siis käyttäen profiloijamerkintöjä, jotka on sisäänrakennettu kaikkiin Unity-pelimoottorin eri komponentteihin ja metodeihin (Unity 2020b). Esimerkiksi kuviosta 8. nähdään kolme eri profiloijamerkintää PlayerLoop, Editorloop sekä Profiler.CollectEditorStats. Nämä kolme profiloijamerkintää ovat ohjelman pääsäikeen base markers eli päämerkintöjä, jonka alaisuuteen kaikki muut pääsäikeen alamerkinnät kuuluvat. Nämä keskeiset päämerkinnät näyttävät eron sovellukseen käytetyn ajan, Unity-editorin sekä profiloijan omiin toimintoihin käytetyn ajan väliltä. (Unity 2020b.)



Kuvio 8. Profiloijamerkinnät

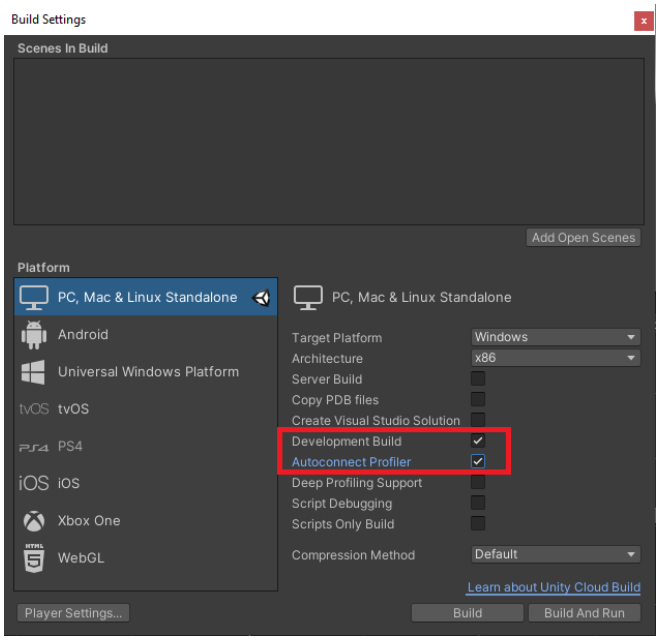
Unity-pelimoottorissa on todella paljon eri osioita ja komponentteja, joilla kaikilla on siis oma profiloijamerkintä. Useasti yhdellä profiloijamerkinällä on myös useita alamerkintöjä. Jos vastaan tulee profiloijamerkintöjä, joita ei täysin ymmärrä nimen perusteella, silloin kannattaa turvautua Unityn dokumentaatioon. Unity-pelimoottorin versiosta riippuen voivat profiloijamerkintöjen nimet hieman

vaihdella. Kirjoitushetkellä käytössä oli Unity-pelimoottorin versio 2019.4.16f1. Yleisimmät ohjelmakoodeihin sekä grafiikkaan liittyvät profiloijamerkinnät ovat seuraavat.

BehaviourUpdate-profiloijamerkintä sisältää kaikkien MonoBehaviour-ohjelmakoodien Update()-funktioiden näytteet. Tämän merkinnän alta nähdään kaikki kehittäjän ohjelmakoodit, joissa on toteutettu Update()-funktio. **LateBehaviourUpdate**-profiloijamerkintä sisältää kaikkien ohjelmakoodien LateUpdate()-funktioiden näytteet. **FixedBehaviourUpdate**-profiloijamerkintä sisältää kaikkien ohjelmakoodien FixedUpdate()-funktioiden näytteet. **Camera.Render** -profiloijamerkinnän alla on iso osa renderöintiin liittyvistä merkinnöistä. Tätä merkintää kannattaa siis avata syvemmälle tarkempien tietojen saamiseksi.

2.4 Suorituskyvyn mittaaminen profiloijalla

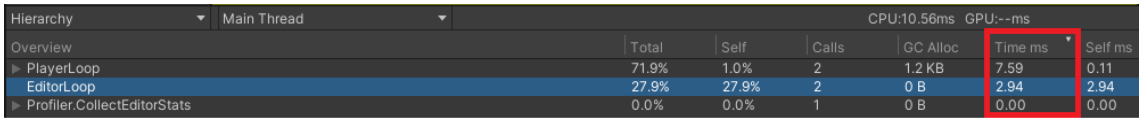
Unity-sovellusta voi profiloida suoraan Unity-editorissa, mutta Unity suosittelee aina profiloimaan sovellusta suoritettavassa ohjelmassa ja mieluiten myös kohdealustalla (Unity 2020c). Mikäli sovellusta kehitetään mobiiliympäristölle, niin sovelluksesta suositellaan luomaan mobiilille tarkoitettu suoritettava ohjelma ja profiloimaan sitä kohdepuhelimilla. Kuviossa 9. on korostettuna kohdat, mitkä täytyy olla valittuna, jotta profiloijan saa päälle suoritettavaan sovellukseen.



Kuvio 9. Profiloijan käyttöönotto suoritettavaan sovellukseen

Unity suosittelee vahvasti profilointia suoritettavassa ohjelmassa, sillä profilointi editorissa ei täysin vastaa profilointia suoritettavassa ohjelmassa, koska profiloinnin suorittaminen editorissa profiloi myös editoria itsessään (Unity 2020c). Tämä myös näkyy profiloijan tiedoissa Editorloop-profiloijamerkintänä, jos sovellusta profiloidaan editorissa (Kuvio 10).

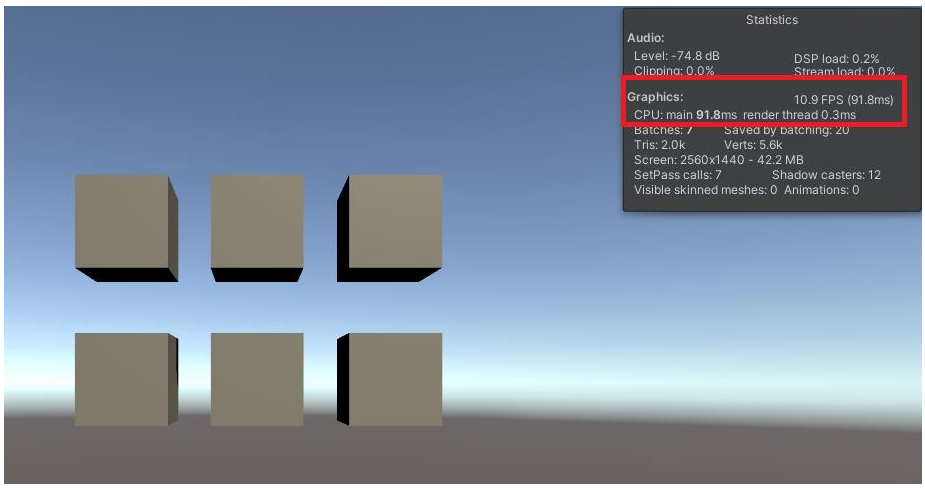
Kuviosta 10. nähdään suorituskyvyn merkitys, kun profilointia suoritetaan editorissa. Valitulla framella käytettiin 2,94 millisekuntia pelkästään editorin päivitykseen. Kun profilointia suoritetaan suoritettavassa ohjelmassa, silloin Unity-pelimoottorin ei tarvitse päivittää mitään editorin ikkunoita tai huolehtia monista eri editoriin liittyvistä tarkistuksista. Tämän lisäksi on hyvä huomioida, että kun sovelluksesta luodaan suoritettava ohjelma, voi Unity suorittaa monia optimointitekniikoita esimerkiksi ohjelmakoodeille, malleille ja muille Unity-pelimoottorin komponenteille (Unity 2020d). Näitä optimointitekniikoita ei siis suoriteta editorissa ollenkaan. Näiden takia on siis hyvä luoda suoritettava sovellus, johon profiloija on kytketty päälle, kun sovelluksesta halutaan tarkka suorituskykyanalyysi.



Kuvio 10. Profilointi Unity-editorissa

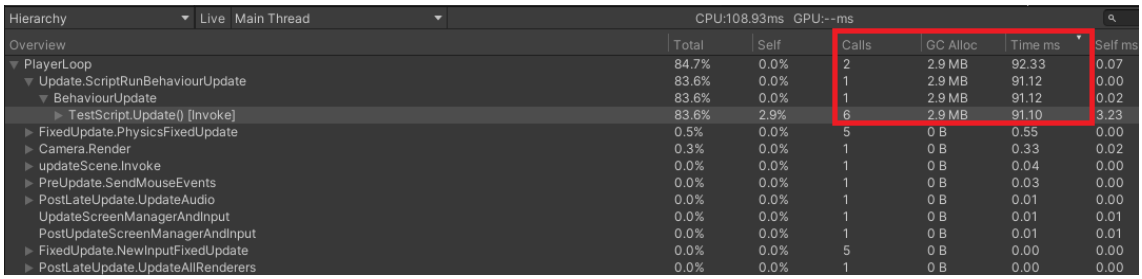
2.5 Esimerkki suorituskykyongelman löytämisestä

Esimerkkiä varten luotiin yksinkertaisen skene, jossa on kamerapeliobjekti, yksi valopeliobjekti sekä kuusi kuutiopeliobjektia. Skeneä suoritettaessa suorituskyky on heikko, kuten kuvion 11. Statistics-ikkunan tiedoista nähdään. Sovellus toimii on vain 11 FPS, mikä on erittäin alhainen frame rate (ruudunpäivitysnopeus) eikä se olisi riittävä sulavaan pelattavuuteen.



Kuvio 11. Skene ja Statistics-ikkuna

Kuviosta 12. nähdään skenen profiloija suorituskykytiedot, mistä nähdään, että iso osa suoritettavasta ajasta kului TestScript-nimisen ohjelmakoodin Update()-funktion suorittamisessa. Profiloija näyttää myös, että TestScript.Update() -funktiota on kutsuttu kuusi kertaa framen aikana sekä muistia varattiin 2,9 megatavua.



Kuvio 12. Skenen hierarkianäkymän tilastot

TestScript-ohjelmakoodista nähdään, että jokaisella framella Update()-funktiossa silmukka käydään läpi sata kertaa ja jokaisella silmukan pyörähdyksellä kutsutaan PrintMessage()-metodia, joka tulostaa Unityn konsoliin yhden viestin (Kuvio 13). Jokaisella framella tulostetaan siis sata viestiä ja koska profiloijan Calls-sarakkeessa näkyi, että ohjelmakoodia on kutsuttu kuusi kertaa yhden framen aikana, täytyy siis skenessä olla kuusi aktiivista peliobjektia, joihin ohjelmakoodi on liitetty. Jokaiseen kuutioon on siis liitetty TestScript-niminen ohjelmakoodi, joista jokainen käy silmukan läpi sata kertaa ja tulostaa yhden viestin konsoliin jokaisella silmukan pyörähdyksellä. Kokonaisuudessaan yhden framen aikana tulostetaan siis 600 viestiä konsoliin.

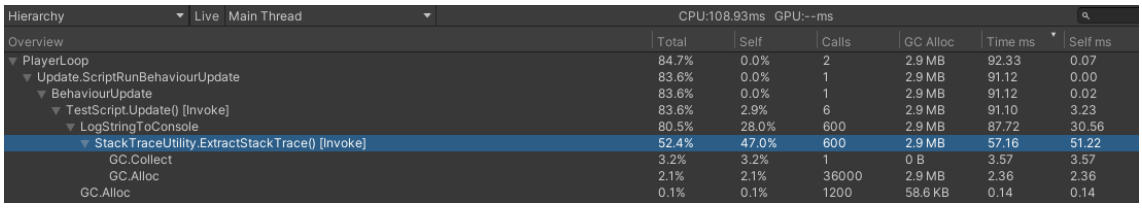
```
public class TestScript : MonoBehaviour {

    private void Update() {
        for (int i = 0; i < 100; i++) {
            PrintMessage();
        }
    }

    private void PrintMessage() {
        Debug.Log("Printing message.");
    }
}
```

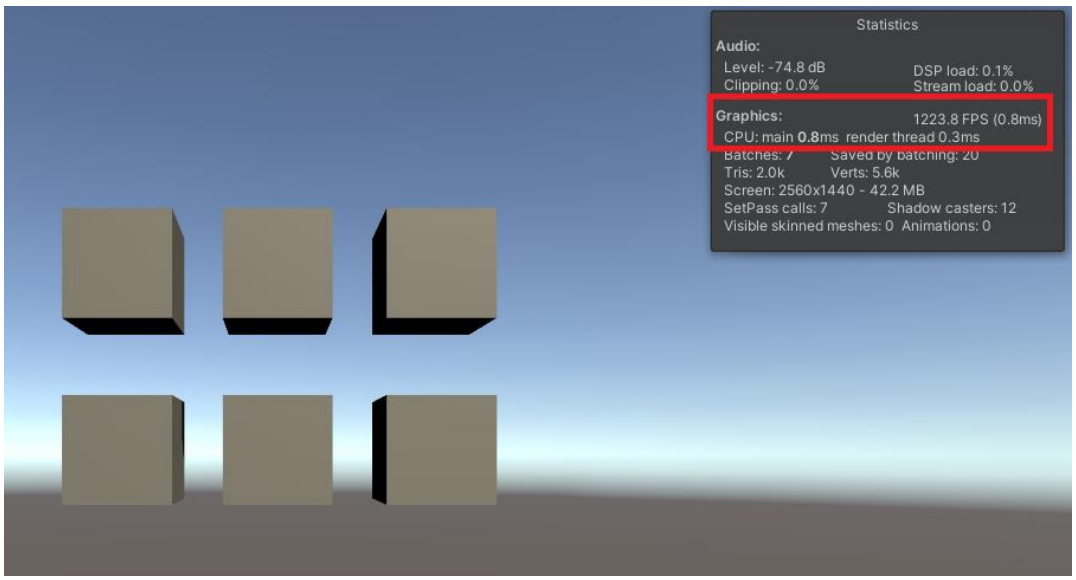
Kuvio 13. TestScript-ohjelmakoodi

Kuviossa 14. TestScript.Update() -profiloijamerkintää on avattu hieman syvemmälle, mistä nähdään tarkempia tietoja mihin prosessointiaika on kulunut. Jos profiloijan Deep Profile olisi kytketty päälle, olisi TestScript.Update() -profiloijamerkinnän alamerkinnät mennyt vieläkin syvemmälle. Tässä ei sille kuitenkaan ollut tarvetta, vaan näistä tiedoista voidaan jo päätellä, miksi skene toimi heikosti.



Kuvio 14. TestScript-ohjelmakoodin alamerkinnät

TestScript-ohjelmakoodi on lyhyt, mutta silti erittäin raskas, koska Debug.Log() -metodin käyttö on raskas operaatio. Tätä kannattaa välttää käyttämästä muuhun kuin erilaisiin ohjelmakoodien testaustarkoituksiin, minkä jälkeen ne kannattaa poistaa ohjelmakoodeista kokonaan (Unity 2020e). Poistamalla TestScript-ohjelmakoodi kaikista kuudesta peliobjekteista tai poistamalla pelkästään Debug.Log() -metodi ohjelmakoodista paransi suorituskykyä yli satakertaisesti. Frame rate nousi yli 1200 FPS, kuten kuvion 15. Statistics-ikkunan tiedoista nähdään.



Kuvio 15. Statistics-ikkunan tiedot optimoinnin jälkeen

Tämän esimerkin tarkoituksena oli havainnollistaa Unity-profiloijatyökalun hyödyllisyys. Mikäli profiloijaa ei olisi tässä katsottu ollenkaan, olisi kehittäjä voinut olla hieman ymmällään ja voinut lähteä etsimään ongelmaa täysin väärästä paikasta. Esimerkiksi mikäli kehittäjä olisi tässä lähtenyt optimoimaan grafiikkaa, ei sovelluksen suorituskykyyn olisi tullut mitään muutosta, koska sovelluksen pullonkaulana oli prosessorilla suoritettava ohjelmakoodi. Suorituskykyongelmien ilmetessä kannattaa siis aina turvautua profiloijatyökaluun. Isoista ja monimutkaisista skeneistä suorituskykyongelmien löytäminen ilman profiloijan apua voi olla täyttä arvailua kokeneemmalle kehittäjällekin.

3 OHJELMAKOODIEN OPTIMOINTI

3.1 Unity Monobehaviourin elinkaari

Unity-pelimoottorin toteuttama C#-ohjelmointikielen MonoBehaviour on perusluokka, josta jokaisen ohjelmakoodin pitää periä, mikäli sen haluaa kiinnittää Unity-pelimoottorin peliobjektiin (Unity 2021a). MonoBehaviour-perusluokkaan on sisäänrakennettu useita tapahtumapohjaisia funktioita, joita kehittäjät voivat käyttää. Tapahtumapohjaisia funktioita ei kutsuta, mikäli ohjelmakoodi ei ole aktiivinen tai peliobjekti, johon ohjelmakoodi on liitetty, ei ole aktiivisena. Seuraavana on listattuna yleisimpiä Unity-pelimoottorin MonoBehaviour-perusluokan tapahtumapohjaisia funktioita.

Awake()-funktio kutsutaan heti sovelluksen alussa, uuden skenen käynnistyessä tai kun peliobjekti tulee ensimmäistä kertaa aktiiviseksi. Tämä kutsutaan vain kerran koko ohjelmakoodin elinkaaren aikana. Awake()-funktio on ensinmäinen tapahtumapohjainen funktio, jota Unity-pelimoottori kutsuu ohjelmakoodin elinkaaren aikana. **OnEnable()-**funktiota kutsutaan joka kerta, kun ohjelmakoodi tulee aktiiviseksi. Sovelluksen tai uuden skenen käynnistyessä OnEnable()-funktiot kutsutaan vasta, kun kaikki Awake()-funktiot on kutsuttu. **Start()-**funktiota kutsutaan vain kerran, kun ohjelmakoodi ladataan ensinmäisen kerran pelimaailmassa. Start()-funktiota kutsutaan vasta, kun kaikki OnEnable()-funktiot on kutsuttu. (Unity 2021b.)

FixedUpdate()-funktio on sovelluksen frame ratesta riippumaton funktio Unity-pelimoottorin fysiikkalaskelmia varten. Oletuksena Unity-pelimoottorissa fysiikkalaskelmat tapahtuvat 50 kertaa sekunnissa. Tätä arvoa kehittäjä voi vaihtaa projektin asetuksista. Koska tätä funktiota kutsutaan useasti sekunnissa, on tässä funktiossa olevat ohjelmakoodit hyvä olla mahdollisimman suorituskykyisiä. Unity-pelimoottori voi kutsua FixedUpdate()-funktiota useammin yhden framen aikana, mikäli sovelluksen frame rate on hitaampi kuin fysiikkalaskelman päivitysväli. (Unity 2021b.)

Update()-funktiota kutsutaan jokaisella sovelluksen framella (Unity 2021b). Mikäli ohjelmakoodi siis toteuttaa Update()-funktion ja sovelluksen frame rate on esimerkiksi 60 FPS, niin silloin Unity kutsuu kaikkia aktiivisia ohjelmakoodeja, 60

kertaa yhden sekunnin aikana. Tämän takia kaikki ohjelmakoodit tämän funktion sisällä on hyvä pitää suorituskykyisinä. **LateUpdate()-**funktio on samanlainen kuin Update()-funktio eli funktiota kutsutaan jokaisella sovelluksen framella mutta tätä kutsutaan vasta, kun kaikki Update()-funktiot on kutsuttu (Unity 2021b).

OnDisable()-funktiota kutsutaan aina, kun ohjelmakoodi menee pois päältä (Unity 2021b). OnDisable()-funktiota kutsutaan silloin, kun ohjelmakoodi laitetaan pois päältä tai kun peliobjekti, johon ohjelmakoodi on liitetty, laitetaan pois päältä. **OnDestroy()-**funktiota kutsutaan silloin, kun ohjelmakoodi tai peliobjekti, johon ohjelmakoodi on liitetty, poistetaan kokonaan pelimaailmasta. **OnGUI()-**funktio eroaa muihin tapahtumapohjaisiin funktioihin, siten että, se on ainoa tapahtumapohjainen funktio, mikä voi kutsua Unity-pelimoottorin GUI-järjestelmää. Unityn GUI-järjestelmä on täysin ohjelmakoodipohjainen UI eli käyttöliittymien tekoon tarkoitettu järjestelmä. Tätä metodia Unity voi mahdollisesti kutsua useasti yhtä framea kohti. Unity suosittelee välttämään tämän metodin käyttöä käyttöliittymien tekoon ja sen sijasta käyttämään peliobjektipohjaista UI-järjestelmää käyttöliittymien tekoon (Unity 2021c). OnGUI()-funktiota voi käyttää esimerkiksi erilaisiin testaustarkoituksiin.

Unity-pelimoottorin C#-ohjelmointikielen Monobehaviour-perusluokkaan sisäänrakennetut tapahtumapohjaiset funktiot ovat kehittäjille erittäin käteviä ja helppokäyttöisiä, mutta mikäli niiden kanssa ei ole tarkkana, voi suorituskykyongelmia ilmetä. On siis hyvä olla tarkkana, mitä tiettyihin tapahtumapohjaisiin funktioihin laittaa, varsinkin niihin funktioihin, joita kutsutaan useasti. On hyvä myös muistaa, että vaikka ohjelmakoodin tapahtumapohjaisessa funktiossa ei olisi riviäkään ohjelmakoodia, kutsutaan niitä silti. Tämän takia on hyvä poistaa kaikki tapahtumapohjaiset funktiot, joissa ei ole ohjelmakoodia ollenkaan. (Unity 2020e.)

3.2 Ohjelmakoodien optimointitekniikoita

3.2.1 Caching

Caching-tekniikalla tarkoitetaan ohjelmakoodien luokkamuuttujien välimuistiin referenssin tallentamista myöhempää käyttöä varten. Suorituskyvyn kannalta on

siis hyvä tapa tallentaa luokkamuuttujien välimuistiin kaikki useasti ohjelmakoodeissa käytettävät muuttujat ja Unity-komponenttien referenssit, varsinkin ne, joita käytetään niissä Monobehaviour-perusluokan tapahtumapohjaisissa funktioissa, joita kutsutaan usein.

Esimerkiksi kaikki Unity-komponentit on hyvä tallentaa ohjelmakoodien alussa GetComponent-metodi kutsuilla luokkamuuttujien välimuistiin talteen aina, kun se on mahdollista. Kuvion 16. ohjelmakoodin Start()-funktiossa tallennetaan Rigidbody-komponentti rb-nimiseen luokkamuuttujaan GetComponent-metodi kutsulla, jotta sitä ei tarvitse suorittaa jokaisella fysiikkalaskemalla FixedUpdate()-funktiossa. Yksittäinen GetComponent-metodi kutsu ei ole raskas operaatio, mutta peleissä ja muissa Unity-sovelluksissa, missä yksi frame pyritään suorittamaan mahdollisimman nopeasti, kannattaa pyrkiä eliminoimaan kaikki ylimääräiset kutsut, kuten komponenttien haut muualla kuin ohjelmakoodien alussa.

```
private Rigidbody rb;

void Start() {
    rb = GetComponent<Rigidbody>();
}

void FixedUpdate() {
    // Suorituskyvyltään hitaampi tapa,
    // koska jokaisella fysiikkalaskemalla kutsuttaisiin GetComponent kutsua.
    //GetComponent<Rigidbody>().AddForce(transform.forward * 1f);


    // Suorituskyvyltään parempi tapa,
    // käytetään rb referenssiä. Tällä vältetään GetComponent kutsun käyttö.
    rb.AddForce(transform.forward * 1f);
}
```

Kuvio 16. Caching-tekniikan käyttö ohjelmakoodeissa

3.2.2 Raskaat Unity-kutsut

Unity-pelimoottori on tehty kehittäjille helppokäyttöiseksi ja täten tarjoaa useita käteviä ja helppokäyttöisiä luokkia, metodeja ja funktioita. Osa näistä metodeista tai funktioista on kuitenkin suorituskyvyltään raskaita operaatioita eikä Unityn ohjelmointirajapinnat sitä useasti edes mainitse, vaikka muuten ovat hyvin doku-

mentoitu. Unity-ohjelmointirajapinnan tarjoamiin eri funktioihin ja metodeihin kannattaa siis suhtautua pienellä varauksella ja niiden suorituskykyä kannattaa tutkia profiloijan avulla. Joskus suorituskyvyn kannalta voi olla parempi luoda oma funktio tai metodi, kuin käyttää Unityn ohjelmointirajapinnan valmiiksi tarjoamaa funktiota tai metodia.

Esimerkiksi GameObject.Find() -funktio, jolla voidaan löytää peliobjekti skenestä nimen perusteella on hidas operaatio. Mikä tekee GameObject.Find() -funktiosta hitaan on se, että Unity-pelimoottori joutuu käymään läpi kaikki skenen peliobjektit, kunnes Unity löytää haetun peliobjektin nimen perusteella (Unity 2021d). Tämä voi olla erittäin hidas operaatio, jos skenessä on tuhansia peliobjekteja ja mikäli skenessä on useita samannimisiä peliobjekteja, voi funktio mahdollisesti palauttaa väärän peliobjektin. Seuraavana on listattuna muutamia raskaita Unityn ohjelmointirajapintakutsuja, joita kannattaa välttää käyttämästä niissä MonoBehaviour-tapahtumapohjaisissa funktioissa, joita kutsutaan useasti:

- GameObject.Find(string name)
- GameObject.FindWithTag(string tag)
- GameObject.FindGameObjectWithTag(string tag)
- GameObject.FindGameObjectsWithTag(string tag)
- Transform.Find(string name)
- Camera.main.

3.2.3 Object Pooling

Unity-pelimoottorissa peliobjekteja voidaan luoda ohjelmakoodeissa sovelluksen ajon aikana luomalla kokonaan uusi peliobjekti new GameObject() -kutsulla tai kopioimalla olemassa oleva peliobjekti komponentteineen Instantiate()-metodi kutsulla. Nämä operaatiot ovat kuitenkin suhteellisen raskaita, joten niiden käyttöä kannattaa pyrkiä välttämään sovelluksen ajon aikana, mikäli se on mahdollista.

Unity-sovelluksissa monesti kuitenkin tarvitaan peliobjekteja ajon aikana, esimerkiksi ammuntapeleissä luoteja tarvitaan vain silloin, kun pelaaja ampuu. Yksi tapa olisi siis luoda valmis luoti-prefab, joka Instantiate()-metodi kutsulla kopioidaan ja kun luotipeliobjektia ei enää tarvita, voidaan se poistaa Destroy()-metodi kutsulla. Tämä tapa on kuitenkin suhteellisen raskas prosessorin ja muistin käytön kannalta. Jos siis sovelluksessa on paljon tämäntyyppistä tarvetta, parempi tapa on rakentaa Object Pooling -järjestelmä, jossa peliobjekteja voidaan käyttää uudelleen ilman uusien peliobjektien luontia alusta lähtien. (Unity 2020e).

Object Pooling eli peliobjektiallas on siis tekniikka, jossa sovelluksen alussa luodaan kaikki sovelluksen ajon aikana tarvittavat peliobjektit talteen, esimerkiksi C#-ohjelmointikielen lista muuttujatyyppiin. Esimerkiksi ammuntapeleissä voidaan sovelluksen alussa valmiiksi luoda tietty määrä luotipeliobjekteja listaan talteen ja kun pelaaja ampuu, voidaan objektialtaan listasta hakea luotipeliobjekti, siirtää se oikeaan paikkaan ja laittaa peliobjekti päälle. Kun luotipeliobjektia ei enää tarvita, voidaan peliobjekti laittaa pois päältä ja tällä tavalla palauttaa se takaisin objektialtaaseen uutta käyttöä odottamaan. Tällä tekniikalla on mahdollista saada vältettyä Instantiate()- ja Destroy()-metodien käyttö sovelluksen ajon aikana.

3.2.4 Ohjelmakoodien jaksotus

Peleissä ja muissa sovelluksissa on usein asioita, joita pitää tarkistaa tiheästi, kuten vihollisen etäisyys pelaajaan tai muita tärkeitä suoritettavia toimintoja pelin tai sovelluksen kannalta. Näitä toimintoja ei kuitenkaan välttämättä kannata tai edes tarvitse suorittaa jokaisella sovelluksen framella, vaan näihin toimintoihin voisi mahdollisesti riittää joka toisella framella tarkistaminen, ellei väljemmin. Yksi tapa keventää ohjelmakoodia yhtä framea kohden on siis jaksottaa prosessointi usealle framelle tai suorittaa prosessointia vain tietyn aikavälein.

Esimerkiksi jos toiminnon tiedetään kestävän 15 millisekuntia, kannattaisi tämä pyrkiä suorittamaan pienemmissä osissa, esimerkiksi kolmessa viisi millisekuntia

kestävässä osassa, jolloin yhdellä framella ei tapahtuisi suurta piikkiä prosessoinnissa. Tälläinen ohjelmakoodin jaksotus voidaan toteuttaa esimerkiksi Coroutine-metodissa, kuten kuvion 17. ohjelmakoodissa on toteutettu.

```
// Tallennetaan WaitForEndOfFrame muuttujaan talteen.
WaitForEndOfFrame oneFrame = new WaitForEndOfFrame();

void Start() {
    // Kutsutaan startLongTask IEnumerator Coroutine-metodia.
    StartCoroutine(StartLongTask());
}

private IEnumerator StartLongTask() {
    // Odotetaan aina yksi frame (kehys), kunnes seuraavaa metodia kutsutaan.
    yield return oneFrame;
    LongTask1();

    yield return oneFrame;
    LongTask2();

    yield return oneFrame;
    LongTask3();
}

private void LongTask1() {
    // Paljon prosessointia vaativa metodi 1.
}

private void LongTask2() {
    // Paljon prosessointia vaativa metodi 2.
}

private void LongTask3() {
    // Paljon prosessointia vaativa metodi 3.
}
```

Kuvio 17. Ohjelmakoodin jaksotus Coroutine-metodissa

Ohjelmakoodin jaksotus muissa Monobehaviour-perusluokan tapahtumapohjaisissa funktioissa, kuten Update()-funktiossa, voidaan toteuttaa ajastimella tai frameja laskemalla. Kuvion 18. ohjelmakoodissa on kaksi metodia, joita kutsutaan eri aikavälein. CheckPlayerPosition()-metodia kutsutaan joka kymmenes frame ja CheckEnemyPosition()-metodia kutsutaan joka sekunti.

```
private float timer;
private int frames;

void Update() {
    ++frames;
    timer += Time.deltaTime;

    if (frames % 10 == 0)
        CheckPlayerPosition();

    if (timer > 1f)
        CheckEnemyPosition();
}

private void CheckPlayerPosition() {
    // Tätä metodia kutsutaan joka kymmenes frame (kehys).
    frames = 0;     // Nollataan frames laskuri.
}

private void CheckEnemyPosition() {
    // Tätä metodia kutsutaan joka sekunti.
    timer = 0f;     // Nollataan ajastin.
}
```

Kuvio 18. Ohjelmakoodin jaksotus Update()-funktiossa

3.3 Garbage Collector

3.3.1 Automaattinen muistinhallinta

C#-ohjelmointikielessä on automaattinen roskankerääjä (GC), joka hoitaa muistin varaamisen sekä ajallaan sen poiston. Monissa muissa ohjelmointikielissä, kuten C- tai C++-ohjelmointikielissä oheista ominaisuutta ei ole ja näissä ohjelmointikielissä ohjelmoija on itse vastuussa muistin varaamisesta sekä vapauttamisesta. Näillä kummallakin on omat hyvät puolensa. Esimerkiksi C#-ohjelmointikielen automaattinen roskankerääjä vapauttaa ohjelmoijan muistin ylläpitämisestä, millä voidaan välttyä muistivuodoilta melkein kokonaan, mutta toisaalta automaattinen roskankerääjä kuluttaa jonkin verran prosessointiaikaa.

Vaikka C#-ohjelmointikielessä on käytössä automaattinen roskankerääjä, ei se tarkoita etteikö muistinhallintaa kannata ottaa huomioon Unity-sovelluksien kehityksessä, sillä muistin varaukset ja poistot ovat suhteellisen raskaita operaatioita ja voivat johtaa suorituskykyongelmiin. Mitä enemmän roskaa eli muistinvarauksia GC:n täytyy puhdistaa, sitä kauemmin prosessi kestää.

Unity-pelimoottori käyttää Boehm–Demers–Weiser -roskankerääjää, joka on niin sanottu maailman pysäytys roskankerääjä. Tällöin kaikki muu suorittaminen lopetetaan kokonaan, kunnes GC on saanut työnsä valmiiksi (Unity 2020f). Unity julkisti kesäkuussa 2020 uuden ominaisuuden, jolla GC:n prosessointia voi jaksottaa usean framen ajalle, jotta pitkiltä yksittäisiltä maailman pysäytyksiltä vältyttäisiin (Unity 2021e). Pahimmillaan kaikki muu suorittaminen voi pysähtyä useaksi sadaksi millisekunniksi, mikäli roskaa on paljon mitä GC:n täytyy käydä läpi. Jos sovelluksen aikana siis tapahtuu satunnaisia yksittäisiä prosessointipiikkejä tai pysähdyksiä, voi tämä johtua siitä, että GC:llä on paljon roskaa, mitä käydä läpi. Tällöin kannattaa tutkia profiloijan avulla, mitkä ohjelmakoodit tai komponentit luovat roskaa. Tämän näkee selkeästi profiloijan hierarkianäkymän GC alloc -sarakkeesta.

3.3.2 Muistihallinnan huomioon ottaminen

Kun C#-ohjelmointikielessä luodaan uusi objekti, niin riippuen muuttujatyypistä, C#-ohjelmointikieli varaa tätä varten tarvittavan määrän muistia ja kun objektia ei enää tarvita, poistaa GC tämän automaattisesti. Esimerkiksi new string eli uuden merkkijonomuuttujan luominen ja ajallaan sen poisto, aiheuttaa työtä GC:lle. Tämän takia on hyvä pyrkiä välttämään käyttämästä new-sanaa tiettyjen C#-ohjelmointikielen muuttujatyyppien sekä tiettyjen Unity-ohjelmointirajapintojen kanssa. Tämä voidaan toteuttaa esimerkiksi alustamalla sovelluksen alussa kaikki tarvittavat objektit luokkamuuttujiin eli käyttämällä hyväksi Caching-tekniikkaa, jolloin vain sovelluksen alussa luodaan roskaa.

Kuvion 19. ohjelmakoodissa alustetaan Start()-funktiossa string-taulukkoon piste tekstit 0-30, jolloin vain sovelluksen alussa luodaan roskaa ja sovelluksen aikana PäivitäPisteet()-metodia voidaan kutsua luomatta uutta muistinvarausta tai roskaa. Lisäksi, että PäivitäPisteet()-metodi ei luo sovelluksen ajon aikana uutta muistinvarausta tai roskaa, on se myös suorituskyvyltään nopeampi, sillä ohjelmakoodissa viitataan taulukon indeksiin, joka on nopea operaatio.

```
public int maksimiPisteet = 30;
private string[] pisteTaulukko;

private void Start() {
    // Alustetaan string taulukon koko, 0-30.
    pisteTaulukko = new string[maksimiPisteet + 1];

    // Tallennetaan string taulukkoon numerot 0-30.
    for (int i = 0; i < pisteTaulukko.Length; ++i) {
        pisteTaulukko[i] = i.ToString();
    }
}

private void PäivitäPisteet(int pisteet) {
    // ToString() kutsu luo roskaa joka kerta kun sitä kutsutaan,
    // siksi sitä kannattaa välttää käyttämästä ajonaikana.
    //tekstiKomponentti.text = pisteet.ToString();

    // Näytetään string taulukosta pistettä vastaava numero ja koska,
    // string tekstit on jo alustettu alussa, niin se ei luo uutta roskaa.
    tekstiKomponentti.text = pisteTaulukko[pisteet];
}
```

Kuvio 19. Muistinhallinta ohjelmakoodeissa

C#-ohjelmointikielen muistihallinnan kannalta on siis hyvä välttää käyttämästä new-sanaa tiettyjen muuttujatyyppien sekä Unity-ohjelmointirajapintojen kanssa. Roskaa luovat muun muassa seuraavat C#-ohjelmointikielen muuttujatyypit tai Unity-pelimoottorin ohjelmointirajapinta kutsut:

- New Object()
- New GameObject()
- New String()
- New Dynamic()
- New List<T>()
- New Dictionary(Tkey, TValue)
- New WaitForSeconds(float seconds)
- New WaitForSecondsRealTime(float time)
- New WaitForEndOfFrame()
- GameObject.tag
- Object.Instantiate().

C#-ohjelmointikielessä on myös hyvä huomioida, että miltei jokainen string-muuttujatyypin operaatio luo aina uuden objektin ja näin myös uuden muistinvarauksen ja ajallaan sen poiston. String-muuttujatyyppiä kannattaa siis käyttää harkiten ja käyttää muita muuttujatyyppejä sen sijasta, mikäli se on mahdollista. Mikäli string-muuttujatyyppiä täytyy usein pilkkoa, rakentaa tai yhdistellä, niin kannattaa tähän käyttää C#-ohjelmointikielen StringBuilder-luokkaa. StringBuilder-luokka on tarkoitettu string-muuttujatyyppien käsittelyyn ilman, että se luo uusia muistinvarauksia. (Unity 2020e.)

Mikäli sovelluksen ajon aikana C#-ohjelmointikielen lista- tai sanakirja-tietorakenteeseen halutaan poistaa tai lisätä kohteita, niin tämä ei ole ongelma roskan luonnin kannalta. Pelkästään uuden listan tai sanakirjan luominen new-sanalla luo uuden muistinvarauksen ja ajallaan sen poiston. Tämän takia lista- ja sanakirja-tietorakenteet kannattaa alustaa sovelluksen alussa.

3.4 Unity Job System

3.4.1 Järjestelmän esittely

Unity Job Systemin avulla kehittäjä voi luoda helposti rinnakkaisajo ohjelmakoodeja Unity-pelimoottorissa tiettyjen rajojen puitteissa suorituskyvyn parantamiseksi (Unity 2021f). Oletuksena kaikki kehittäjän kirjoittamat ohjelmakoodit Unity-pelimoottorissa suoritetaan sovelluksen pääsäikeessä eli yhdellä prosessorisäikeellä. Nykyään prosessoreissa on kuitenkin useita säikeitä, joista osalla ei välttämättä ole ollenkaan prosessoitavaa, kun taas sovelluksen pääsäikeellä voi olla paljonkin prosessoitavaa. Job Systemin avulla osa prosessoinnista voidaan siis suorittaa rinnakkain MonoBehaviour-ohjelmakoodien kanssa. Job Systemin avulla voidaan esimerkiksi laskea kahden objektin etäisyys toisistaan samalla, kun jokin toinen ohjelmakoodi suorittaa jotain muuta toimintoa, jolloin toiminnot tapahtuvat rinnakkain yhtäaikaisesti.

Job System on pitkälti kehitetty osaksi DOTS-pakettia, mutta Job Systemiä voi käyttää myös MonoBehaviour-ohjelmakoodien kanssa. Tässä osiossa käydään läpi vain Job Systemin hyödyntäminen MonoBehaviour-ohjelmakoodien kanssa.

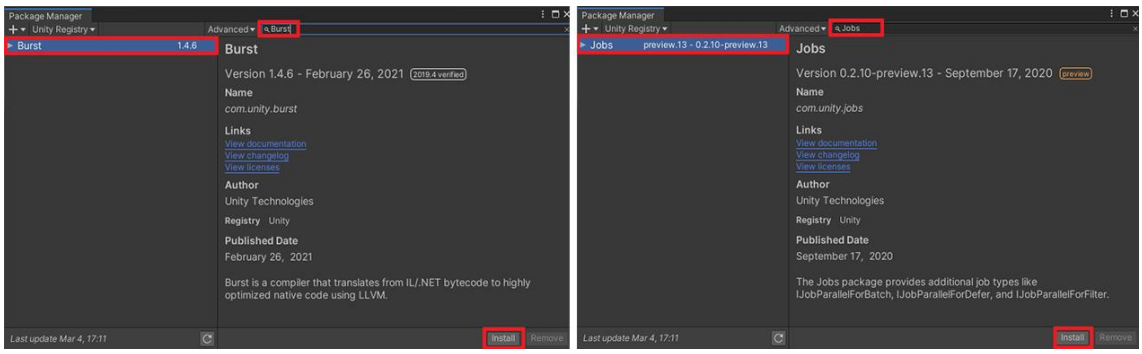
Huomiona tässä täytyy kuitenkin ottaa, että iso osa Unityn ohjelmointirajapinnoista eivät toimi muualla kuin Unityn pääsäikeessä eli isoa osaa komponenteista ei voi suoraan käyttää taustasäikeissä. Job System käsittelee rinnakkaisajo ohjelmakoodeja, luomalla ohjelmakoodeista niin sanottuja työtehtäviä, joita Unity-pelimoottorin taustasäikeet suorittavat työjonosta (Unity 2021g).

Job Systemin rinnalle Unity on myös kehittänyt oman ohjelmointikieli kääntäjän nimeltä Burst Compiler. Burst-kääntäjällä Unity-pelimoottori muuttaa kehittäjän ohjelmakoodit hyvin optimoituun natiivikoodiin käyttäen LLVM-ohjelmointikieli kääntäjä, jolla suorituskykyä saadaan parannettua entisestään (Unity 2021h). Burst-kääntäjä on suunniteltu vain Job System- ja ECS-ohjelmakoodien käyttöä varten. Burst-kääntäjä ei siis toimi MonoBehaviour-ohjelmakoodien kanssa.

Job System eroaa MonoBehaviour-ohjelmakoodeista, siten että, muistihallinnasta vastaa kehittäjä itse. Job System antaa kuitenkin selkeitä virheviestejä, mikäli muistia ei tyhjennetä tietyn ajan kuluessa. Job System on vielä jatkuvassa kehityksessä, joten uudemmissa versioissa asiat ovat voineet muuttua. Kirjoitushetkellä Jobs-paketista käytössä oli versio 0.2.10-preview.13, Burst-paketista versio 1.4.4 ja Unity-pelimoottorista versio 2019.4.16f1.

3.4.2 Järjestelmän käyttö ohjelmakoodeissa

Jotta Job Systemiä voi käyttää, täytyy se ensin olla asennettuna Package Managerista, jonka saa auki Window-alasvetovalikosta. Kun Package Manager on auki, voi sen hakukenttään kirjoittaa Jobs, minkä jälkeen Package Manager hakee Unityn rekisteristä Jobs-paketin. Valitsemalla Jobs-paketti vasemmasta laidasta, tulee Packager Managerin alalaitaan Install-näppäin, jolla paketin saa asennettua projektiin. Mikäli Job Systemin kanssa halutaan käyttää Burst-kääntäjää, pitää sekin asentaa Package Managerista samalla tavalla (Kuvio 20).



Kuvio 20. Pakettien asennus Package Managerista

Job System -ohjelmakoodeissa käytetään C#-ohjelmointikielen rakenteita, minkä täytyy toteuttaa jokin Job-rajapintaluokka. Jobs-paketissa on muutama eri sisäänrakennettu rajapintaluokka, joita kehittäjät voivat käyttää. Job System toimii vain C#-ohjelmointikielen rakenteiden kanssa, Job System ei siis tue luokkien käyttöä.

Kuvion 21. ohjelmakoodissa toteutetaan IJob-rajapintaluokka MyJob-nimiselle rakenteelle, joka ottaa vastaan kaksi float-muuttujatyyppiä sekä NativeArray-taulukon. NativeArray on Unity-pelimoottorin toteuttama muuttujatyyppi, jolla voidaan altistaa natiivimuisti puskuri kehittäjän ohjelmakoodeille. Tämän muuttujatyypin muistia ei poisteta automaattisesti, vaan kehittäjä on itse vastuussa tämän vapauttamisesta. Se tapahtuu kutsumalla NativeArray<T0>.Dispose() -metodia, kun muuttujatyyppiä ei enää tarvita. (Unity 2021i.) Viimeisenä MyJob-rakenteelle annetaan NativeArray-taulukko, johon laskutoimitukset tallennetaan.

Kun ohjelmakoodissa on luotu uusi rakenne ja annettu sille halutut arvot, voidaan Job käynnistää Schedule()-komennolla, jolloin MyJob-rakenteen Execute()-metodissa oleva laskutoimitus toteutetaan eri säikeellä kuin muu ohjelmakoodi. Kun Jobin laskutoimitus tarvitaan ohjelmakoodissa, täytyy kutsua Complete()-komentoa, joka on ainoa tapa varmistaa, että Job on saanut työnsä valmiiksi. Kun Complete()-komentoa kutsutaan, odotetaan siis niin kauan, kunnes Job on saanut työnsä valmiiksi. Complete()-komentoa ei suositella kutsuttavaksi heti, sillä muuten Job Systemin käytöstä ei saada käytännössä mitään hyötyä. Tämän takia Unity suosittelee alustamaan kaikki Jobit mahdollisimman aikaisin framen

alkupuolella esimerkiksi Update()-funktion alussa ja lukemaan arvot vasta myöhemmin framen loppupuolella, esimerkiksi LateUpdate()-funktiossa (Unity 2021j).

Lopuksi kuvion 21. ohjelmakoodin LateUpdate()-funktiossa tulostetaan MyJob-rakenteen laskutoimitus konsoliin. Ohjelmakoodi tulostaa konsoliin siis arvon 20, sillä MyJob-rakenteelle annettiin arvot 15 ja viisi, jotka MyJob-rakenteen Execute-metodissa lasketaan yhteen. Viimeisenä ohjelmakoodissa poistetaan NativeArray-taulukko kutsumalla Dispose()-metodia.

```
using Unity.Collections;
using UnityEngine;
using Unity.Burst;
using Unity.Jobs;

public class SimpleJob : MonoBehaviour {

    private NativeArray<float> resultArray;
    private JobHandle jobHandle;

    void Update() {
        // Alustetaan taulukko, joka annetaan MyJob rakenteelle.
        resultArray = new NativeArray<float>(1, Allocator.TempJob);

        MyJob job = new MyJob() {   // Luodaan uusi MyJob rakenne .
            a = 5f,                 // Annetaan rakenteelle a arvo.
            b = 15,                 // Annetaan rakenteelle b arvo.
            result = resultArray    // Annetaan rakenteelle NativeArray taulukko.
        };
        jobHandle = job.Schedule(); // Ajetaan Job Schedule() komennolla.
    }

    void LateUpdate() {
        // Complete() komennolla varmistetaan että Job on saanut työnsä valmiiksi.
        jobHandle.Complete();

        // Tulostetaan laskettu arvo testiä varten konsoliin. Tuloksen pitäisi olla 20.
        Debug.Log(resultArray[0]);

        // Poistetaan NativeArray. Tämä pitää tehdä muuten tulee error!
        resultArray.Dispose();
    }

    [BurstCompile]  // Job käännetään Burst kääntäjän läpi.
    public struct MyJob : IJob {
        public float a;
        public float b;
        public NativeArray<float> result;

        public void Execute() {     // Execute() on IJob rajapintaluokan toteuttama metodi
            result[0] = a + b;      // Lasketaan laskutoimitus
        }
    }
}
```

Kuvio 21. Job System -ohjelmakoodi

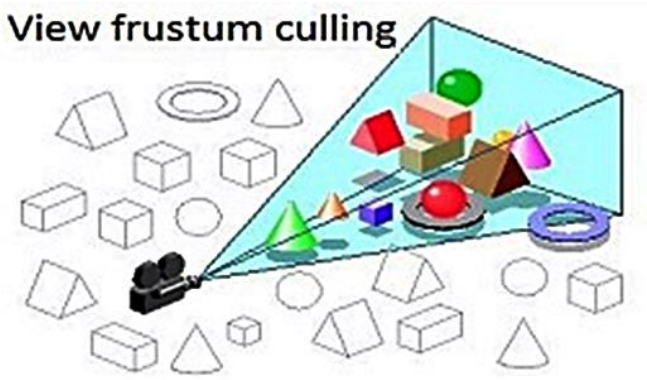
4 GRAFIIKAN OPTIMOINTI

4.1 Karsinta

4.1.1 Frustum Culling

Frustum Culling eli näkökenttäkarsinnalla tarkoitetaan prosessia, jolla kaikki piirrettävät objektit, jotka eivät näy kameran näkökentässä, poistetaan grafiikan piirtokanavasta kokonaan. Tämä vähentää näytönohjaimen piirtotaakkaa huomattavasti.

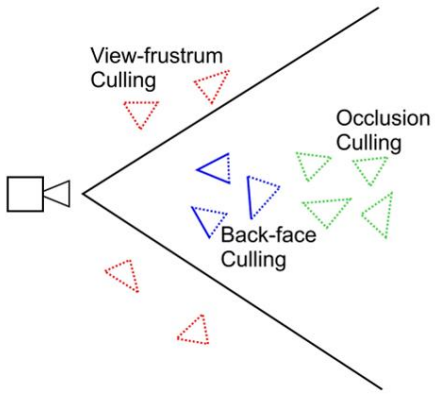
Unity-pelimoottori suorittaa näkökenttäkarsinnan automaattisesti kaikille aktiivisille kameroille, joten kehittäjän ei tarvitse tehdä mitään tämän käyttämiseksi (Unity 2020g). Kuviossa 22. on visualisoituna näkökenttäkarsinnan toiminta. Kaikki objektit, jotka ovat kameran näkökentän ulkopuolella, jätetään siis piirtämättä.



Kuvio 22. Näkökenttäkarsinta (Duarte 2020)

4.1.2 Occlusion Culling

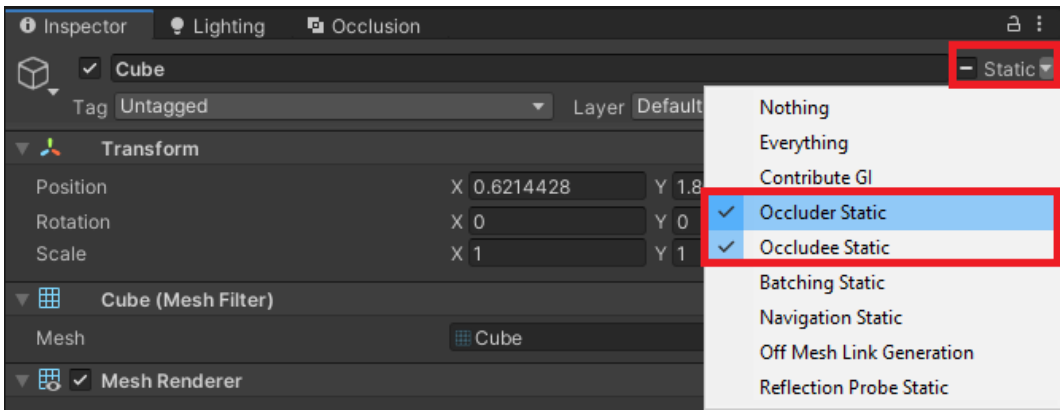
Occlusion Culling -tekniikalla tarkoitetaan prosessia, jolla kaikki objektit, jotka ovat kokonaan toisten objektien peitossa tai takana, poistetaan grafiikan piirtokanavasta. Kuviossa 23. vihreällä olevat objektit jätettäisiin siis piirtämättä, koska ne ovat sinisten objektien takana.



Kuvio 23. Occlusion Culling (Duarte 2020)

Unity käyttää Umbra Software -yrityksen luomaa Occlusion Culling -järjestelmää, joka on sisäänrakennettu Unity-pelimoottoriin (Unity 2020g). Järjestelmä on kaksivaiheinen. Ensinmäinen vaihe on rakentaa kevyt tietorakenne halutusta skenestä Unity-editorissa. Toinen vaihe tapahtuu sovelluksen ajon aikana, jolloin etukäteen rakennettua tietorakennetta lukemalla, voidaan piirtokanavasta poistaa kaikki objektit, jotka jokin toinen objekti tai objektit peittävät kokonaan.

Occlusion Culling -järjestelmä toimii merkitsemällä kaikki halutut peliobjektit, jotka halutaan ottaa huomioon Occlusion Culling -järjestelmässä, Occluder Static ja/tai Occludee Static Unity-editorin StaticEditorFlags-alasvetovalikosta. Alasveto-valikko löytyy peliobjektin Inspector-ikkunan oikeasta yläkulmasta (Kuvio 24).

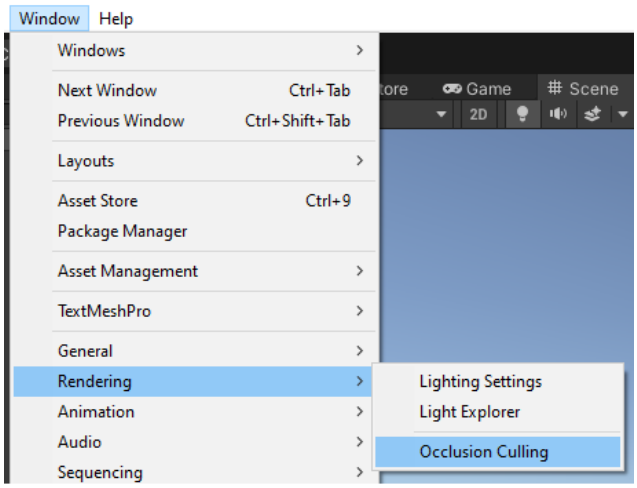


Kuvio 24. StaticEditorFlags-alasvetovalikko

Occluder Static -valinnalla merkitty objekti on siis peittävä objekti. Kaikki objektit, jotka ovat tämän objektin takana kokonaan piilossa kamerasta nähden,

poistetaan grafiikan piirtokanavasta. **Occludee Static** -valinnalla merkitty objekti on taas objekti, joka poistetaan piirtokanavasta, jos jokin Occluder Static -valinnalla merkitty objekti tai objektit peittävät sen kokonaan. (Unity 2021k.)

Kun kaikki halutut peliobjektit, jotka halutaan ottaa huomioon Occlusion Culling -järjestelmässä on merkitty StaticEditorFlags-alasvetovalikosta, voidaan Unity-editorissa suorittaa toiminto, joka luo tietorakenteen. Toiminto löytyy Occlusion-ikkunasta. Occlusion-ikkunan saa avattua Occlusion Culling -kohdasta (Kuvio 25).

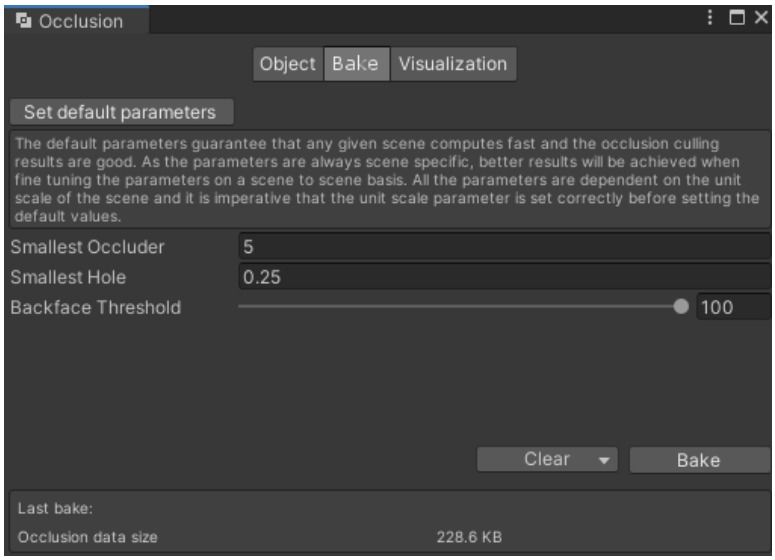


Kuvio 25. Occlusion-ikkunan avaaminen

Seuraavaksi Occlusion-ikkunasta valitaan Bake-välilehti, josta löytyy kolme eri asetusta sekä ikkunan oikeasta alalaidasta Bake-näppäin (Kuvio 26). Bake-näppäintä painamalla Unity-pelimoottori alkaa luomaan avatusta skenestä tietorakennetta, jota Occlusion Culling -järjestelmä käyttää sovelluksen ajon aikana. (Unity 2021k.) Riippuen skenen koosta, peliobjektien määrästä sekä Occlusion-ikkunan asetuksista, voi tässä kulua useampi minuutti. Tämä toiminto pitää siis suorittaa jokaiselle skenelle erikseen, mikäli skenen halutaan käyttävän Occlusion Culling -tekniikkaa.

Occlusion-ikkunan Bake-välilehdellä on kolme eri asetusta. Näillä kolmella eri asetuksilla voidaan säädellä Occlusion Culling -tietorakenteen parametrejä. Tässä on hyvä huomioida, että jokainen skene on erilainen ja jossain skeneissä tietyt asetukset voivat tuottaa paremman lopputuloksen. Näitä kolmea asetusta

kannattaa siis vaihdella katsoen, mitkä asetukset tuottavat parhaan lopputuloksen. Unityn oletusasetukset ovat kuitenkin hyvä lähtökohta. Occlusion-ikkunan alaosasta nähdään myös luodun tietorakenteen tiedostokoko (Kuvio 26).



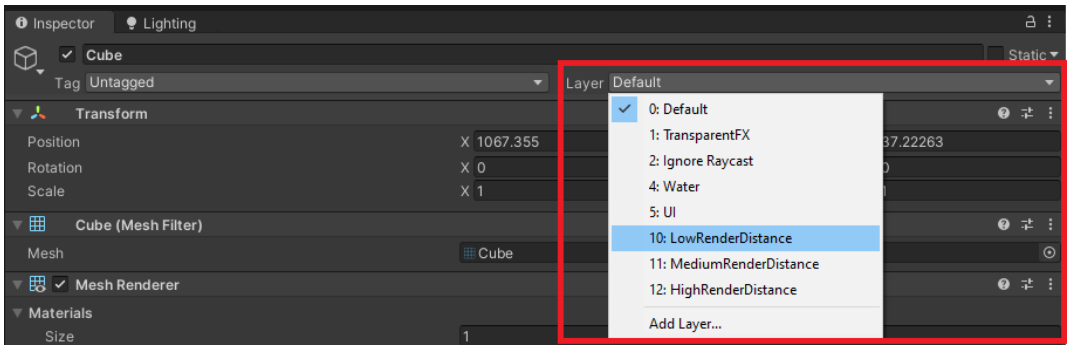
Kuvio 26. Occlusion-ikkuna

4.1.3 Per Layer Culling

Unity-sovelluksissa kerroksia käytetään usein tietyn logiikan tai järjestyksen ylläpitämisessä, mutta kerroksia voidaan käyttää myös grafiikan karsinnan käytössä. Tätä tekniikkaa kutsutaan Per Layer Culling eli kerroskohtaiseksi karsinnaksi. Esimerkiksi peliobjektit, jotka ovat tietyssä kerroksessa, voidaan karsia grafiikan piirtokanavasta etäisyyden perusteella, vaikka peliobjekti olisi muuten näkyvissä kameralla. Eri kerroksille voidaan siis asettaa kerroskohtaiset renderöintietäisyydet.

Tämä tekniikka on erittäin kätevä, sillä peleissä ja muissa Unity-sovelluksissa on usein erikokoisia objekteja, joille halutaan asettaa eri renderöintietäisyydet. Tätä tekniikkaa käyttämällä voidaan esimerkiksi isot objektit, kuten rakennukset ja vuoret, asettaa kerrokselle, jossa on pitkä renderöintietäisyys, kun taas pienet objektit, kuten lehdet tai pikkukivet, joita käyttäjä ei näe kuin vasta läheltä, asettaa kerrokselle, jossa on huomattavasti lyhyempi renderöintietäisyys. Tätä tekniikkaa käyttämällä voidaan siis vähentää näytönohjaimen piirtotaakkaa.

Unity-editorissa peliobjektien siirto eri kerroksille tai uusien kerroksien luonti tapahtuu peliobjektin Inspector-ikkunan oikean ylälaidan Layers-alasvetovalikosta (Kuvio 27). Kerroksia voi olla maksimissaan 32, joista osa on jo varattu Unity-pelimoottoria varten.



Kuvio 27. Layers-alasvetovalikko

Kerroskohtaiset renderöintietäisyydet voidaan asettaa vain ohjelmakoodillisesti käyttämällä Camera-komponentin ohjelmointirajapinnan Camera.layerCullDistances -ominaisuutta. Oletuksena kaikki kerrokset käyttävät Camera-komponentin farClipPlane-etäisyyttä eli kameralle asetettua maksimietäisyyttä (Unity 2020h).

Kuviossa 28. on lyhyt ohjelmakoodi, miten Camera.layerCullDistances -ohjelmointirajapintaa voidaan hyödyntää. Ohjelmakoodissa luodaan taulukko, jonka koko täytyy olla 32, koska Unity-pelimoottorissa on yhteensä näin monta kerrosta (Unity 2020h). Tämän jälkeen halutun kerroksen indeksille asetetaan haluttu renderöintietäisyys. Ohjelmakoodissa kerroksille 10, 11 ja 12 on asetettu halutut renderöintietäisyydet. Tämän jälkeen taulukko siirretään kameralle. Kun jokin peliobjekti asetetaan kerrokselle 10, 11 tai 12, poistetaan se grafiikan piirtokanavasta kokonaan, mikäli kameran etäisyys objektista on yli annetun arvon. Camera.layerCullDistances -ominaisuutta käytettäessä kannattaa huomioida, että kerroksiin joihin ei erikseen asetettu arvoa, käyttävät siis oletusarvo renderöintietäisyyttä eli ne kerrokset käyttävät kameran farClipPlane-arvoa. Mikäli sovelluksessa on käytössä useita kameroita, pitää kaikille kameroille asettaa kerroskohtaiset etäisyydet erikseen, mikäli niiden halutaan käytettävän kerroskohtaista karsintatekniikkaa.

```
// Alustetaan taulukko, jonka täytyy olla 32 kokoinen, koska Unityssä on 32 kerrosta/layer.
private float[] distances = new float[32];

void Start() {
    Camera camera = GetComponent<Camera>();  // Haetaan kamera komponentti
    camera.farClipPlane = 500;               // Muutetaan kameran renderöinti etäisyys 500 unit.

    distances[10] = 50;                      // Muutetaan kerroksien 10, 11 ja 12 renderöinti etäisyydet.
    distances[11] = 150;
    distances[12] = 250;
    camera.layerCullDistances = distances;   // Siirretään taulukko kameralle.
                                             // Kerroksiin joihin ei tehty muutoksia pysyvät siis ennallaan.
}
```

Kuvio 28. Per Layer Culling -ohjelmakoodi

4.2 Level of Detail

Level of Detail -tekniikalla (LOD) tarkoitetaan, kuinka yksityiskohtainen objekti näytetään riippuen siitä, kuinka kaukana objekti on kamerasta. Mitä kauempana objekti on kamerasta, sitä vähemmän yksityiskohtaisempi malli voidaan näyttää, sillä sitä vaikeampi käyttäjällä on erottaa tarkkoja yksityiskohtia objekteista (Unity 2020e). Tällä tekniikalla saadaan vähennettyä näytönohjaimen piirtotaakkaa, sillä vähemmän yksityiskohtaisien mallien piirtäminen on nopeampaa. Tätä tekniikkaa fiksusti käyttämällä, käyttäjä ei välttämättä edes huomaa, että malli on vaihdettu vähemmän yksityiskohtaisempaan malliin, joten visuaaliseen ilmeeseen ei siis tule suurta muutosta.

Unity-pelimoottorissa on sisäänrakennettu LOD-komponentti nimeltä LODGroup. LODGroup-komponenttien käyttö on yksi helpoimmista ja nopeimmista optimointitekniikoista Unity-pelimoottorissa, sillä LODGroup-komponenttiä voidaan käsitellä Unity-editorissa ilman, että kehittäjän täytyy kirjoittaa riviäkään ohjelmakoodia. Toisaalta tämän tekniikan käyttö lisää graafikkojen työtä, koska samasta mallista täytyy olla eri yksityiskohtaisia malleja. Mikäli malleista on luotu eri yksityiskohtaisia versioita, kannattaa LODGroup-komponenttia ehdottomasti käyttää.

Unity-pelimoottorin LODGroup-komponentti toimii siis laskien sovelluksen ajon aikana, kuinka paljon objekti vie prosentuaalisesti kameran näkymästä. Kun prosentuaalinen määrä ylittää tai alittaa käyttäjän asettamien LOD-tasojen arvon, niin silloin vaihdetaan seuraavaan LOD-tasoon ja malliin. (Unity 2021l.) Kun käyttäjä on riittävän kaukana objektista, voidaan malli vaihtaa kokonaan esimerkiksi

kuvaksi, näin vähentäen näytönohjaimen piirtotaakka entisestään. Kun käyttäjä on riittävän kaukana objektista, voidaan objekti poistaa grafiikan piirtokanavasta kokonaan.

4.3 Draw Call Batching

Draw Call Batching eli piirtopyyntöerien yhdistämisellä tarkoitetaan tekniikkaa, jolla piirtopyyntöjä pyritään lähettämään isommissa erissä, jotta prosessorilla ei tarvitse lähettää jokaista piirrettävää objektia yksitellen näytönohjaimelle. Esimerkiksi mikäli skenessä on näkyvissä useita samoja objekteja, voi nämä objektit lähettää yhdellä kertaa näytönohjaimelle piirrettäväksi. Jotta Unity-pelimoottori voi siis piirtää peliobjektin ruudulle, pitää pelimoottorin lähettää piirtopyyntö näytönohjaimelle grafiikkaohjelmointirajapinnan kautta. Näiden piirtopyyntöjen lähetys näytönohjaimelle on suhteellisen raskas operaatio ja tämä tapahtuu jokaisella sovelluksen framella prosessorin toimesta (Unity 2020i). Mitä enemmän piirtopyyntöjä, sitä kauemmin lähetyksessä kestää. Siksi on hyvä pyrkiä pitämään piirtopyyntöerien kokonaismäärä mahdollisimman alhaisena. Unity-pelimoottori siis hoitaa piirtopyyntöjen lähetyksen kehittäjän puolesta, mutta piirtopyyntöerien kokonaismäärä jää kehittäjän vastuulle.

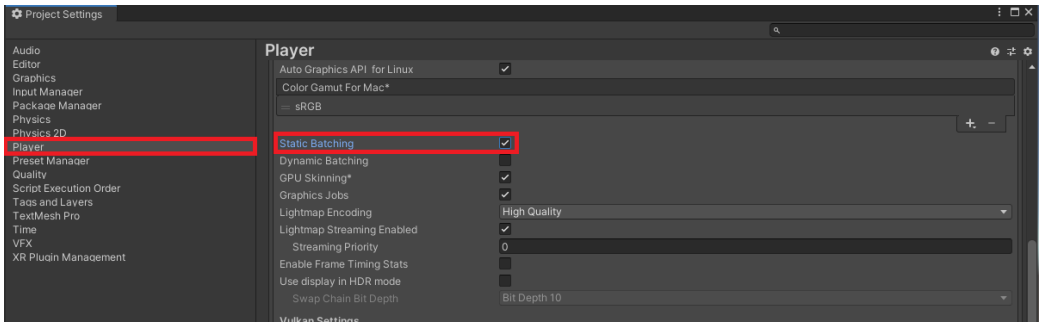
Unity-pelimoottori tarjoaa muutamia eri tekniikoita, jolla piirtopyyntöerien määrä saa vähennettyä. Näillä tekniikoilla on kuitenkin tarkat rajoitukset, joita peliobjektien täytyy noudattaa. Kaikki Unity-pelimoottorin renderöinti-komponentit eivät kuitenkaan tue tätä tekniikkaa, kuten Skinned Mesh Renderer- ja Cloth-komponentit. Seuraavat Unity-pelimoottorin renderöinti-komponentit tukevat piirtopyyntöerien yhdistämistekniikoita (Unity 2020i):

- Mesh Renderer
- Trail Renderer
- Line Renderer
- Particle System
- Sprite Renderer.

4.3.1 Static Batching

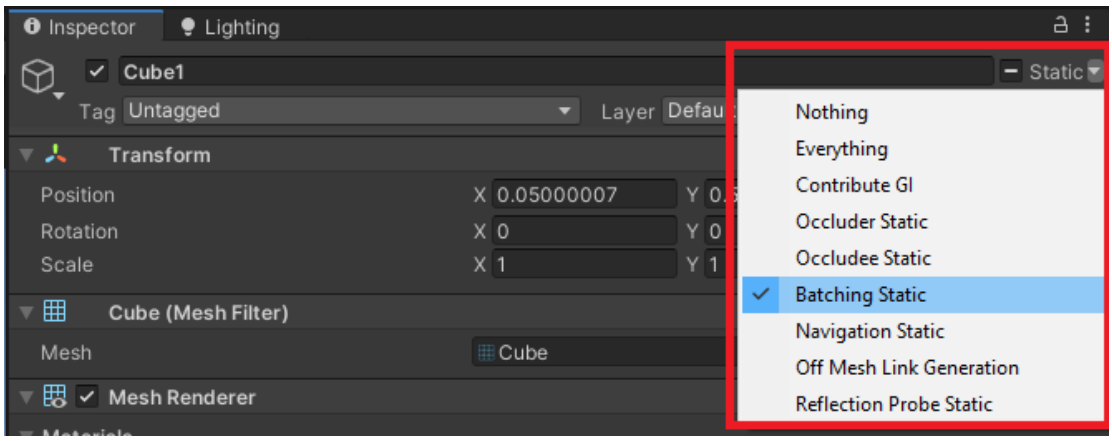
Static Batching eli staattisten objektien piirtopyyntöerien yhdistäminen on yksi tapa vähentää piirtopyyntöerien kokonaismäärää. Static Batching -tekniikalla piirtopyyntöeriä voi yhdistää yhdeksi kutsuksi, kun peliobjektit jakavat saman materiaalin ja ovat liikkumattomia (Unity 2020i).

Jotta Static Batching -tekniikkaa voi käyttää, täytyy sen ensin olla päällä projektin asetuksista. Projektin asetukset saa auki navigoimalla Edit-valikosta Project Settings -kohtaan. Kun projektin asetukset on auki, voi Static Batching -valinnan laittaa päälle Player-välilehdeltä (Kuvio 29).



Kuvio 29. Static Batching -valinta

Kun Static Batching -valinta on laitettu päälle projektin asetuksista, pitää kaikki peliobjektit, joiden halutaan käytettävän Static Batching -tekniikkaa, merkitä Batching Static -valinnalla StaticEditorFlags-alasvetovalikosta (Kuvio 30). Static Batching -tekniikkaa käytettäessä täytyy kuitenkin huomioida, että se käyttää enemmän muistia kuin muut piirtopyyntöerien yhdistämistekniikat, joten se ei välttämättä ole aina paras vaihtoehto. Esimerkiksi laitteilla, joissa muistia on rajallisesti, ei tiheää metsää kannata laittaa käyttämään Static Batching -tekniikkaa. (Unity 2020i.)

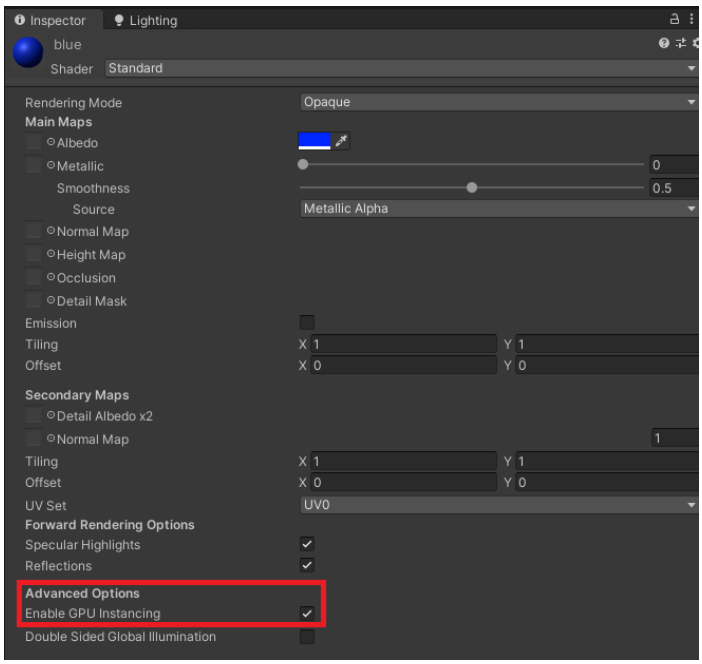


Kuvio 30. Batching Static -valinta

4.3.2 GPU Instancing

GPU Instancing on tekniikka, jolla voidaan piirtää useita kopioita samasta mallista yhdellä piirtopyyntökutsulla, kun objektit jakavat saman materiaalin. GPU Instancing -tekniikalla voidaan siis piirtää vain identtisia malleja jokaisella piirtopyyntöerällä, mutta jokaisella yksittäisellä objektilla piirtopyyntöerässä voi olla omia parametrejä, kuten objektin skaala tai materiaalin väri (Unity 2020j).

GPU Instancing -tekniikka pitää toteuttaa shaderiin (varjostimeen), että sitä voi hyödyntää. Shader on siis ohjelma, jota näytönohjain käsittelee, jotta se tietää miten objekti halutaan piirrettävän ruudulle. Monet Unity-pelimoottorin sisäänrakennetut shaderit toteuttavat GPU Instancing -tekniikan valmiiksi, mutta mikäli shaderin tekee itse, täytyy GPU Instancing toteuttaa kehittäjän itse (Unity 2020j). Mikäli käytössä on jokin Unity-pelimoottorin shader, joka toteuttaa GPU Instancing -tekniikan, kuten Standard Shader, voi GPU Instancing -tekniikan laittaa päälle materiaalin Inspector-ikkunan Advanced Options -kohdasta (Kuvio 31).



Kuvio 31. GPU Instancing -valinta

4.3.3 Dynamic Batching

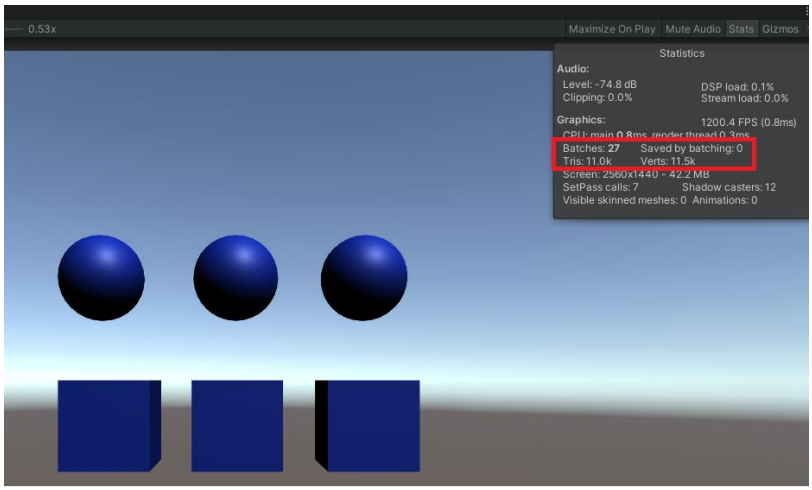
Dynamic Batching on automaattinen piirtopyyntöerien yhdistämistekniikka, mikä ei vaadi kehittäjältä mitään muita toimenpiteitä kuin sen päälle laittaminen projektin asetuksista. Dynamic Batching -tekniikan saa päälle projektin asetuksista samalla tavalla kuin Static Batching -tekniikan (Kuvio 29). Unity-pelimoottori voi hyödyntää Dynamic Batching -tekniikkaa seuraavien kriteerien täyttyessä (Unity 2020i):

- Objektien täytyy käyttää samaa materiaalia.
- Objektin mallissa on alle 300 verteksiä ja alle 900 verteksiatribuuttia.
- Objektien Transform-komponentit eivät ole peilattuja.
- Objektien materiaalin shader ei käytä Multi-pass -renderöintitekniikkaa.
- Yksittäisellä objektilla piirtopyyntöerässä ei ole materiaalissa eri arvoja, kuten väri.

Näiden rajoituksien takia Dynamic Batching -tekniikka, ei ole yksistään riittävä piirtopyyntöerien yhdistämistekniikka monimutkaisille ja isoille Unity-sovelluksille. Dynamic Batching -tekniikka on hyödyllinen, jos skenessä on paljon piirrettäviä objekteja, joiden mallit ovat suhteellisen yksinkertaisia.

4.3.4 Batching-tekniikoiden vertailu

Eri piirtopyyntöerien yhdistämistekniikoiden vertailua varten luotiin yksinkertainen vertailuskene, jossa on kolme kuutiota ja kolme palloa, joista kaikki objektit käyttävät samaa materiaalia. Vertailussa piirtopyyntöerien määrä katsottiin Statistics-ikkunasta, josta nähdään piirtopyyntöerien määrä sekä määrä, jonka piirtopyyntöerien yhdistämisellä on säästetty. Kuviossa 32. näkyy vertailuskene sekä Statistics-ikkuna, josta vertailuarvot otettiin ylös.



Kuvio 32. Batching-tekniikoiden vertailuskene ja Statistics-ikkuna

Taulukosta 1. nähdään hyvin eri piirtopyyntöerien yhdistämistekniikoiden erot. Static Batching -tekniikka loi parhaan lopputuloksen piirtopyyntöerien määrän kannalta, mutta on hyvä muistaa, että Static Batching -tekniikka luo omia rajoituksia, esimerkiksi objekteja ei voi liikuttaa sovelluksen ajon aikana ollenkaan, eikä yksittäisille objekteille piirtopyyntöerässä voi antaa eri parametrejä, kuten materiaalin väriä.

Taulukko 1. Piirtopyyntöerien yhdistämistekniikoiden vertailu.

Batching-tekniikka	**Batches**	**Saved by Batching**	**Objekteja voi liikuttaa ajon aikana?**	**Yksittäisillä objekteilla voi olla esimerkiksi oma väri?**
Ei mikään	27	0	Kyllä	Kyllä
Static Batching	7	20	Ei	Ei
GPU Instancing	11	16	Kyllä	Kyllä
Dynamic Batching	20	7	Kyllä	Ei

Parhaan lopputuloksen oikeille skeneille ja sovelluksille saa hyödyntämällä kaikkia piirtopyyntöerien yhdistämistekniikoita yhdessä. On vaikea antaa ohjetta, milloin mitäkin tekniikkaa kannattaa käyttää, sillä pelejä ja muita Unity-sovelluksia on niin monta erityyppistä. Tärkeintä kuitenkin on muistaa, että kaikki piirrettävät peliobjektit, joita sovelluksessa esiintyy usein, on siis hyvä pyrkiä käyttämään jotain piirtopyyntöerien yhdistämistekniikkaa, jotta sovelluksen piirtopyyntöerien kokonaismäärä pysyisi mahdollimman alhaisena.

5 KÄYTÄNTÖ PELIPROJEKTISSA

5.1 Pelin esittely

Nation’s Man on Unity-pelimoottorilla luotu tarinapohjainen 3D-ensimmäisen persoonan ammuntapeli. Peli on niin sanottu semi-open world peli eli pelissä on vain yksi kartta, jossa pelaaja voi liikkua vapaasti muulloin kuin tehtävien aikana. Peli suunniteltiin alusta lähtien pelkästään PC-alustalle ja se on täysin tarinapohjainen yksinpeli, jossa dialogit, välianimaatiot sekä taistelukohtaukset vievät pelin tarinaa eteenpäin. Peliä kehitettiin ja jatkokehitetään täysin opinnäytetyön rinnalla ja pelin oikeudet pysyvät niiden tekijöillä.

Kuvio 33. Nation’s Man -peli

Pelin pääkartta on kokonaisuudessaan noin 3000 x 3800 Terrain-komponentti yksikköä, josta pelattavaa pelialuetta on hieman vähemmän. Kokonaisuudessaan kentässä on yli 20 000 peliobjektia, joista noin 15 000 on piirrettävää peliobjektia. Iso osa piirrettävistä peliobjekteista on puita, joita on tuhansittain ripoteltuna ympäri pelialuetta.

Pelissä suurin osa 3D-malleista sekä tekstuureista on hankittu Unity Asset Storesta, joko ilmaiseksi tai sieltä ostettuna. Miltei kaikki pelin logiikka ja ohjelmakoodit on itse ohjelmoitu. Pelikenttä, tarina, käyttöliittymä, dialogit sekä välianimaatiot on myös täysin itse suunniteltu ja luotu.

Kenelläkään peliprojektin jäsenillä ei ollut paljon kokemusta Unity-pelimoottorista tai optimoinnista ennen peliprojektin aloitusta. Alkuperäisenä ideana tarkoitus oli ensin laittaa peli alulle, minkä jälkeen opinnäytetyöaihetta oli tarkoitus alkaa tutkimaan tarkemmin mutta suorituskykyongelmien ilmetessä jo alkukehitysvaiheessa, otettiin optimointiin liittyvät asiat huomioon miltei heti. Tämä saattoi hieman hidastaa pelin kehitystä, mutta tällä varmistettiin, ettei peli käytännössä koskaan toiminut heikosti. Tämän takia pelistä on myös mahdoton antaa ennen ja optimoinnin jälkeen olevia tuloksia, koska optimointi otettiin huomioon projektin alusta lähtien.

5.2 Pelin ohjelmakoodien optimointi

Pelissä ohjelmakoodit eivät missään vaiheessa tuottaneet isoja suorituskykyongelmia, sillä miltei kaikki ohjelmakoodit kirjoitettiin alusta lähtien itse ja ne pyrittiin käyttämään mahdollisimman vähän suorituskykyä. Pelissä on noin 10-20 eri ohjelmakoodia, joita suoritetaan miltei joka pelin framella tai hieman väljemmin, riippuen mitä pelikentällä tapahtuu. Ohjelmakoodeissa käytettiin Caching-tekniikkaa hyväksi aina, kun se oli mahdollista. Tiettyjä ohjelmakoodeja laitetaan myös kokonaan pois päältä, kun niitä ei tarvita. Esimerkiksi kaikki pelaajan liitetyt ohjelmakoodit voidaan pysäyttää, kun pelaaja pysäyttää pelin painamalla ESC-näppäintä.

Myös C#-ohjelmointikielen muistinhallinta otettiin huomioon heti kehityksen alussa, sillä melkein kaikki optimointiin liittyvät artikkelit ja blogit korostivat, että se kannattaa ottaa huomioon alusta lähtien ohjelmakoodeissa. Esimerkiksi kaikki ohjelmakoodit, joissa käytetään string-muuttujatyyppiä, pyrittiin tallentamaan ohjelmakoodin alussa luokkamuuttujien välimuistiin talteen, jotta pelin aikana uusia string-muuttujatyyppejä ei tarvitse luoda. Tällä ja parilla muulla tekniikalla,

kuten Object Pooling -tekniikalla varmistettiin, että pitkiä GC:n aiheuttamia maailman pysähdyksiä ei tullut ollenkaan.

5.3 Pelin grafiikan optimointi

Jo pelin alkukehitysvaiheessa huomattiin grafiikkaan liittyviä suorituskykyongelmia, jotka aiheutuivat piirtopyyntöerien määrän takia. Esimerkiksi Unityn Terrain-komponentit loivat valtavan määrän piirtopyyntöjä näytönohjaimelle. Piirtopyynnöt saatiin kuriin muuttamalla Terrain-komponenttien Detail Resolution Per Patch- ja Detail Resolution -asetuksia. Toinen suorituskykyongelma Terrain-komponentteihin liittyen huomattiin, kun peliä profiloitiin tarkemmin. Terrain-komponentit käyttivät paljon aikaa profiloijan mukaan Terrain.UpdateMaterials() -metodissa. Tämä ratkaistiin piilottamalla pelimaailman sivuilla olevat Terrain-komponentit, kun pelaaja on niistä tarpeeksi kaukana.

Iso etu piirtopyyntöerien määrään saatiin käyttämällä eri piirtopyyntöerien yhdistämistekniikoita hyväksi yhdessä, kuten Static Batching- ja GPU Instancing -tekniikoita. Käytännössä melkein kaikki piirrettävät objektit, jotka pelissä esiintyy useasti, käyttävät Static Batching- tai GPU Instancing -tekniikkaa. Peliprojektissa on myös päällä Dynamic Batching -tekniikka, jonka hyötyä tai haittaa, ei ole vielä ehditty tutkimaan. Alustavasti Dynamic Batching on kuitenkin vähentänyt piirtopyyntöerien kokonaismäärää.

Pelissä hyödynnettiin myös Per Layer Culling -tekniikkaa, jolla peliobjekteille voitiin helposti asettaa eri renderöintietäisyydet asettamalla ne tietylle kerrokselle. Pelikentällä on kokonaisuudessaan paljon erilaisia piirrettäviä objekteja, joista iso osa on pieniä objekteja, kuten pikkukivet vesien rannoilla. Näitä kiviä pelaaja ei näe kuin vasta läheltä, joten kaikki nämä kivet asetettiin kerrokselle, jossa renderöintietäisyys on alhainen.

6 POHDINTA

Pelien tai muiden sovelluksien optimointi on aina iso työ, on kyse minkä kokoisesta sovelluksesta tahansa. Peleissä ja muissa sovelluksissa on useita eri osa-alueita, jotka optimoinnissa täytyy ottaa huomioon. Asiaa ei myöskään helpota, että eri alustoilla suorituskykyongelmat tai tavoitteet voivat vaihdella. Optimoinnista ei siis voi antaa yleistä yksinkertaista ohjelistaa, jotka kehittäjän täytyy toteuttaa, jotta sovelluksesta saadaan suorituskykyisempi. Jokainen sovellus on siis yksilöllinen.

Työssä käytiin läpi Unity-profiloijan käyttö ja sen hyödyllisyys suorituskykyongelmien etsimisessä ja kiteyttäisinkin tämän opinnäytetyön pääsanoman, että mikäli sovelluksessa esiintyy suorituskykyongelmia, niin kannattaa aina turvautua profiloijan tietoihin, sillä kehittäjän ajatus, mikä sovelluksesta tekee mahdollisesti hitaan on useasti väärä tai ei välttämättä kerro koko totuutta. Työssä käytiin myös läpi Unity-pelimoottorin C#-ohjelmointikielen ohjelmakoodeihin liittyviä vinkkejä ja konsepteja, joilla sovelluksien ohjelmakoodeja voi optimoida. Myös Unity-pelimoottorin peligrafiikkaan liittyviä optimointitekniikoita ja niiden peruskonsepteja käytiin läpi muutamin esimerkein.

Tiedonlähteinä pyrin suosimaan ajankohtaisia ja kehittäjäläheisiä lähteitä. Tämä käytännössä tarkoitti, että iso osa tiedonlähteistä oli Unityn omasta dokumentaatiosta ja sen blogikirjoituksista. Unity on dokumentoinut selkeästi suurimman osan pelimoottorin eri ominaisuuksista ja tarjoaa myös erinomaisia harjoitusprojekteja sekä muita blogeja Unity Learn -sivustolla.

Työ onnistui mielestäni hyvin, ottaen huomioon aiheen laajuuden ja sen, että aikaisempaa kokemusta pelien tai muiden reaaliaikaisten sovelluksien kehittämisestä tai optimoinnista ei ollut juuri ollenkaan. Ilman omaa peliprojektia, josta teoriaan sai erinomaista käytännön kokemusta, olisi aihe ollut paljon haastavampi käsitellä. Isoa osaa tässä työssä käsitellyistä eri optimointitekniikoista ja menetelmistä hyödynnettiin tai vähintään testattiin omassa peliprojektissa, joka helpotti työn aiheen ymmärtämistä huomattavasti.

LÄHTEET

Duarte, A. 2021. View Frustum Culling. Viitattu 1.12.2020 http://aduarte-games.blogspot.com/2016/03/view-frustum-culling.html.

Unity 2020a. Getting started with the Profiler Window. Viitattu 19.8.2020 https://docs.unity3d.com/Manual/ProfilerWindow.html.

Unity 2020b. Common Profiler Markets. Viitattu 30.11.2020 https://docs.unity3d.com/Manual/profiler-markers.html.

Unity 2020c. Profiling your application. Viitattu 19.8.2020 https://docs.unity3d.com/Manual/profiler-profiling-applications.html.

Unity 2020d. Standalone Player settings. Viitattu 31.11.2020 https://docs.unity3d.com/Manual/class-PlayerSettingsStandalone.html#Optimization.

Unity 2020e. Fixing Performance Problems. Viitattu 8.8.2020 https://learn.unity.com/tutorial/fixing-performance-problems#5c7f8528edbc2a002053b595.

Unity 2020f. Understanding Automatic Memory Management. Viitattu 8.8.2020 https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html.

Unity 2020g. Occlusion Culling. Viitattu 4.12.2020 https://docs.unity3d.com/Manual/OcclusionCulling.html.

Unity 2020h. Camera.layerCullDistances. Viitattu 13.1.2020 https://docs.unity3d.com/ScriptReference/Camera-layerCullDistances.html.

Unity 2020i. Draw Call Batching. Viitattu 12.1.2020 https://docs.unity3d.com/Manual/DrawCallBatching.html.

Unity 2020j. GPU Instancing. Viitattu 12.1.2020 https://docs.unity3d.com/Manual/GPUInstancing.html.

Unity 2021a. MonoBehaviour. Viitattu 15.1.2021 https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

Unity 2021b. Order of Execution for event functions. Viitattu 15.1.2021 https://docs.unity3d.com/Manual/ExecutionOrder.html.

Unity 2021c. Immediate Mode GUI. Viitattu 4.2.2021 https://docs.unity3d.com/Manual/GUIScriptingGuide.html.

Unity 2021d. GameObject.Find. Viitattu 6.2.2021 https://docs.unity3d.com/ScriptReference/GameObject.Find.html.

Unity 2021e. Feature Preview: Incremental Garbage Collection. Viitattu 4.2.2021 https://blogs.unity3d.com/2018/11/26/feature-preview-incremental-garbage-collection/.

Unity 2021f. C# Job System. Viitattu 6.2.2021 https://docs.unity3d.com/Manual/JobSystem.html.

Unity 2021g. What is a job system? Viitattu 6.2.2021 https://docs.unity3d.com/Manual/JobSystemJobSystems.html.

Unity 2021h. Burst User Guide. Viitattu 2.2.2021 https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html.

Unity 2021i. NativeContainer. Viitattu 6.2.2021 https://docs.unity3d.com/Manual/JobSystemNativeContainer.html.

Unity 2021j. Ijob. Viitattu 12.2.2021 https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Unity.Jobs.IJob.html.

Unity 2021k. Getting started with occlusion culling. Viitattu 16.2.2021 https://docs.unity3d.com/Manual/occlusion-culling-getting-started.html.

Unity 2021l. LOD Group. Viitattu 18.2.2021 https://docs.unity3d.com/Manual/class-LODGroup.html.