LAB University of Applied Sciences

# 3D-pelien visuaalinen optimointi

LAB-ammattikorkeakoulu
Insinööri (AMK), Tieto- ja viestintätekniikka
2022
Eero Nojosaho

**Tiivistelmä**

| Tekijä(t) | Julkaisun laji | Valmistumisaika |
|---|---|---|
| Nojosaho, Eero | Opinnäytetyö, AMK | 2022 |
| | Sivumäärä<br>65 | |

Työn nimi

**3D-pelien visuaalinen optimointi**

Tutkinto ja koulutusala

Insinööri (AMK), Tieto- ja viestintätekniikka


Tiivistelmä

Työn tavoitteena on tutkia 3D-peleissä käytettäviä yleisimpiä optimoinnin keinoja ja niiden tehokkuutta. Työssä toteutetaan demonstraatio, jossa optimoidaan 3D-peli. Työn tarkoitus on näyttää, kuinka voidaan optimoida 3D-videopeli mahdollisimman hyvin yksinkertaisia visuaalisia keinoja käyttäen, kuitenkaan menettämättä liikaa visuaalisia yksityiskohtia.

3D-mallinnukseen käytetään ilmaista Blender-mallinnusohjelmaa. Joihinkin tekstuureihin liittyviin menetelmiin käytetään lisäksi ilmaista GIMP-kuvankäsittelyohjelmaa. Demonstraatio tehtiin Unity-pelimoottorilla.

Jokaisen optimointimenetelmän hyötyjä testattiin ja analysoitiin peliprojektissa käyttämällä Unityn Profiler-työkalua. Profilerin antamien tietojen avulla vertailtiin optimointimenetelmien käyttämisen aiheuttamaa kuormitusta pelissä kuormitukseen, joka aiheutui, kun pelissä ei ollut käytetty optimointimenetelmiä. Joidenkin optimointimenetelmien käyttämisestä oli hyötyä, mutta osasta oli jopa haittaa peliprojektissa ja osan toteutuksessa kesti liian kauan saavutettuihin hyötyihin nähden.

Lopputuloksena saatiin selville, kuinka tehokkaasti visuaaliset optimointimenetelmät toimivat peliprojektissa. Normal map -menetelmän ja läpinäkyvien tekstuurien käyttäminen paransivat suorituskykyä parhaiten. PVS-menetelmä paransi suorituskykyä sitä paremmin, mitä enemmän pintoja scenessä oli. Valaistuksen optimointi paransi suorituskykyä selvästi. LOD-, Mipmaps- ja Portal rendering -menetelmät vaikuttivat suorituskykyyn heikoiten. Kaikkia optimointimenetelmiä käytettäessä samaan aikaan suorituskyky parani erittäin paljon, ruudunpäivitysnopeus kasvoi kymmenkertaiseksi ja virrankulutus laski. Pelissä ei myöskään menetetty liikaa visuaalisia yksityiskohtia.

Joidenkin menetelmien hyödyt kuitenkin korostuivat liiallisesti menetelmien luonteen vuoksi. Tuloksista voidaan silti päätellä, että visuaalinen optimointi on kannattavaa, kunhan oikeita menetelmiä käytetään oikeassa paikassa.



Asiasanat

3D grafiikka, Blender, GIMP, optimointi, pelimoottori, pelisuunnittelu, Unity, videopeli

**Abstract**

| Author(s) | Type of Publication | Published |
|---|---|---|
| Nojosaho, Eero | Thesis, UAS | 2022 |
| | Number of Pages<br>65 | |

Title of Publication

**Visual optimization in videogames**

Degree and field of study

Bachelor of Engineering, Information and Communications Technology

Abstract

The objective of the thesis is to examine the most common optimization methods used in 3D videogames and their efficiency. A demonstration where a 3D videogame is optimized is implemented in the thesis. The purpose of the thesis is to point out how to optimize a 3D videogame as well as possible using graphical optimization methods without losing too many visual details at the same time.

A free 3D modelling program, Blender, was used for 3D modelling. GIMP, a free image processing program, was used for some methods related to texturing. The demonstration was implemented with Unity game engine.

Benefits of every optimization method were tested and analysed in a game project using Unity game engine's built-in Profiler tool. Profiler was used to get information data about performance when the optimization methods were being used and that information was then compared to the information that was got when no optimization methods were used in the game. Using some of the optimization methods was beneficial but some harmful for performance in the game project and implementing some of the methods took too long time compared to the gained benefits.

As a result, the efficiency of using the visual optimization methods in the game project was found out. The use of Normal map -method and transparent textures improved performance the most. PVS-method improved performance the better the more faces there were in a scene. Optimizing lighting improved performance considerably. LOD-Mipmaps- and Portal rendering -methods affected performance the most poorly. When every optimization method was used at the same time, performance was improved very greatly. Framerate increased by tenfold and power consumption decreased. Too many visual details were not lost either.

However, the benefits of some of the methods were overly exaggerated because of the nature of the methods. Still, a conclusion can be drawn from the result. Visual optimization is worthwhile if the right methods are used at the right place and time.

Keywords

3D graphics, Blender, game design, game engine, GIMP, optimization, Unity, videogame

Sisällys

1 Johdanto ... 1
2 Visuaalinen optimointi ... 3
3 Suorituskyvyn tarkkailu ... 4
4 Clipping Planes ... 6
5 Culling ... 8
5.1 Occlusion Culling ... 8
5.2 Potentially Visible Set ... 8
5.3 Portal rendering ... 11
6 3D-mallit ... 13
6.1 3D-mallien optimoinnin perusteet ... 13
6.2 Normal map ... 14
6.3 Mipmaps ... 15
6.4 Läpinäkyvät tekstuurit ... 16
6.5 LOD ... 18
6.6 Esineen sisältämien komponenttien määrä ... 19
7 Valaistus ... 21
7.1 Yleistä tietoa valaistuksen optimoinnista ... 21
7.2 Lightmapping ... 22
7.3 Light Probes ... 22
8 Case: Optimointimenetelmien käyttäminen Unityssä ... 24
8.1 Projektin taustatiedot ... 24
8.2 Clipping Planes -menetelmän käyttö ... 25
8.3 Occlusion Culling -menetelmien käyttäminen ... 26
8.3.1 Portal rendering -menetelmän toteutus ... 27
8.3.2 Potentially Visible Set -menetelmän toteutus ... 28
8.4 3D-malleihin liittyvien menetelmien toteutus ja käyttöönotto ... 32
8.4.1 Normal mapin luominen ja käyttöönotto ... 32
8.4.2 Läpinäkyvien tekstuurien käyttäminen 3D-mallissa ... 35
8.4.3 Mipmaps-menetelmän käyttäminen ... 40
8.4.4 LOD-mallin luominen ja käyttöönotto ... 44
8.5 Valaistuksen optimointi Unityssä ... 47
8.5.1 Lightmapping-menetelmän toteutus ... 47
8.5.2 Light Probes -menetelmän toteutus ... 50
8.6 Optimointimenetelmien hyötyjen tarkasteleminen ... 53

9 Yhteenveto ........................................................................................................64
Lähteet ..................................................................................................................66

**Termit**

Ajonaikainen. Pelin tai sovelluksen suorittamisen aikana tapahtuva.

Alpha Channel. 2D-kuvan läpinäkyvyyden säätelyyn tarvittava värikanava, jonka olemassaolo vaaditaan, jotta kuvassa olisi mahdollista olla läpinäkyvyyttä.

Bake. Tiedosto, johon on tallennettu esikäsitelty informaatio, jota luetaan ajonaikana.

Baking. Informaation esikäsittely, jossa informaatio tallennetaan tiedostoon, josta informaatiota luetaan ajonaikana.

Batches. Ryhmiä draw calleja, jotka CPU kokoaa yhteen ja lähettää GPU:lle.

Blender. Ilmainen 3D-mallinnusohjelma, jota voidaan käyttää esimerkiksi 3D-mallintamiseen ja 3D-mallien teksturointiin.

CPU. Laitteen, esimerkiksi tietokoneen, suoritin.

Draw calls. Renderöintipyyntöjä, joita CPU lähettää GPU:lle, jotta se tietäisi mitä renderöidään ja millä tavoin.

Dynaaminen. Pelin tai sovelluksen ajonaikana muuttuva ominaisuus.

First Person Shooter. Peli, jota pelataan ensimmäisestä persoonasta, eli pelaajan näkökulmasta tarkoittaen, että pelin kamera toimii pelihahmon ”silminä”.

Forward Rendering Path. Unityn tapa renderöidä objekti pelissä yhtä monta kertaa, kuin siihen osuu eri valonlähde.

Frame. Yksittäinen kaksiulotteinen kuva, joka piirtyy yleensä kymmeniä kertoja sekunnissa laitteen, esimerkiksi tietokoneen, ruudulle.

Frames Per Second. Peräkkäin piirtyvien kuvien määrä laitteen ruudulle sekunnissa.

Frustum. Kameran näkökenttä eli kartion muotoinen alue, jonka sisällä olevat esineet renderöityvät pelissä.

Generointi. Minkä tahansa, kuten esimerkiksi informaation tai esineen luominen tiettyjen parametrien perusteella.

GIMP. Ilmainen kuvankäsittelyohjelma, jota käytetään pääasiassa rasterigrafiikassa 2D-kuvien käsittelemiseen.

GPU. Laitteen, esimerkiksi tietokoneen, grafiikkaprosessori.

Layer. Kuvankäsittelyohjelmissa käytettävä nimitys 2D-kuvan osista tai kerroksista, joita voidaan muokata erillään toisistaan vaikuttamatta samalla muihin osiin kuvasta.

Layer mask. Layeriin lisättävä komponentti kuvankäsittelyohjelmissa, jonka avulla layerin läpinäkyvyysinformaatiota voidaan hallita mustavalkoisen 2D-kuvan avulla.

Lightmap. Tekstuuri, joka sisältää 3D-objektin valaistusinformaation, jota käytetään ajonaikana 3D-objektien valaistuksen laskemiseen.

Mesh Renderer. Unityssä objektiin liitettävä komponentti, joka hallitsee objektin renderöitymistä ja jonka avulla objekti renderöityy.

Modifier. Blenderissä käytettävä automaattinen operaatio, joka vaikuttaa 3D-mallin geometriaan, ja jonka avulla voidaan säästää aikaa, kun operaatioita ei tarvitse tehdä manuaalisesti.

Moiré-kuviot. 3D-mallien pinnoille ilmestyvät häiriöt silloin, kun 3D-malli nähdään kaukaa ja sen käyttämällä tekstuurilla on liian suuri resoluutio.

Node. Blenderin Node Editorissa käytettävät, materiaalin informaatiota muokkaavat ja erilaisia ominaisuuksia sisältävät osat, joista materiaali koostuu, ja joita voidaan yhdistää toisiinsa viivoilla, jolloin ne vaikuttavat toisiinsa, ja siten materiaaliin, tietyllä tavalla.

Node Editor. Blenderissä materiaalien luomiseen käytettävä järjestelmä, jonka toiminta perustuu nodeihin.

Normal map. 2D-tekstuuri, joka sisältää 3D-mallin pintojen normaalivektori-informaation, jota käytetään 3D-mallin pintojen varjostuksen ja valaistuksen laskemiseen.

Object Pooling. Unityssä käytetty tehokas ohjelmointiin liittyvä optimointikeino, jonka avulla on mahdollista vähentää merkittävästi suorittimen kuormitusta.

Occludee. Piilotettava objekti, joka piilotetaan, kun se ei ole pelaajan näkyvissä Unityn Occlusion Culling -menetelmää käytettäessä.

Occluder. Piilottava objekti, joka voi estää pelaajan näköyhteyden alueelle, jolta piilotettavat objektit piilotetaan Unityn Occlusion Culling -menetelmää käytettäessä.

Overdraw. Monen läpinäkyvän objektin piirtyminen päällekkäin, joka aiheuttaa saman kuvan renderöimisen useaan kertaan.

Profiler. Unityn sisäinen työkalu, jonka avulla voidaan tarkkailla Unityllä luotujen pelien ja sovellusten suorituskykyä ja löytää helpommin syy suorituskykyongelmiin.

Renderöinti. Kaksiulotteisen kuvan piirtyminen laitteen, esimerkiksi tietokoneen, ruudulle.

ScriptableObject. Unityssä käytetty tehokas ohjelmointiin liittyvä optimointikeino, jonka avulla on mahdollista vähentää muistin käyttöä.

Shader. Ohjelma, joka laskee esimerkiksi 3D-objektin materiaalin renderöintiä varten oikeanlaisen värin ja valaistuksen tason.

Smooth Shading. 3D-mallinnuksessa käytetty keino, illuusio, jonka tarkoitus on heijastaa 3D-mallin pinnoista valoa niin, että 3D-mallin pinta saadaan näyttämään sileältä.

Source Engine. Valve Corporationin kehittämä kuuluisa pelimoottori, jota on käytetty esimerkiksi tunnetuissa peleissä Half Life 2, Left 4 Dead 2 ja Team Fortress 2 sekä myös uudemmissa peleissä kuten Half Life: Alyx.

Staattinen. Pelin tai sovelluksen ajonaikana muuttumaton ominaisuus.

Teksturointi. Väri-informaation tai muun pintojen yksityiskohtien informaation, kuten valaistusinformaation, lisääminen 3D-mallin pinnoille tekstuureja eli 2D-kuvia käyttämällä.

Tekstuuri. 3D-mallin pintojen väri-informaation tai muun pintojen yksityiskohtien informaation, kuten valaistusinformaation, sisältävä 2D-kuva.

Topologia. 3D-mallin pintojen järjestyksen avulla muodostettu rakenne, jota pidetään täydellisenä silloin, kun se on mahdollisimman sujuva, symmetrinen ja helposti muokattavissa.

Triangles. 3D-objektien pinnat, jotka on muutettu pelimoottorissa kolmioiksi.

Unity. Unity Technologiesin kehittämä tunnettu, useiden pelialan yritysten käyttämä pelimoottori, jolla voidaan kehittää esimerkiksi 3D- ja 2D-pelejä ja simulaatioita.

Unity Dots. Unity Technologiesin Unityyn kehittämä uusi, erittäin suorituskykyinen keino kehittää ja ohjelmoida pelejä.

UV map. 3D-objektin pintojen kaksiulotteinen informaatio, jota käytetään hyödyksi teksturoinnissa.

UV Unwrap. Operaatio, jossa 3D-objektin pinta levitetään kaksiulotteiseksi informaatioksi.

View Volume. Pelialue, jolla pelaaja tulee oletettavasti olemaan tai jonka pelaaja tulee oletettavasti näkemään pelin aikana.

# 1 Johdanto

Tämän tutkimustyön tavoitteena on esitellä visuaalisia keinoja optimoida pelejä ja sovelluksia ja näyttää, miten näitä keinoja voidaan käyttää Unity-pelimoottorilla. Usean eri optimointimenetelmän käyttöä demonstroidaan Unity-pelimoottorilla ja 3D-mallien osalta osin myös ilmaisella Blender-mallinnusohjelmalla. Tekstuureihin liittyvissä menetelmissä käytetään lisäksi ilmaista GIMP-kuvankäsittelyohjelmaa.

Pelien ja sovellusten optimointi on tärkeää pelin toimivuuden ja sujuvuuden kannalta, mutta myös siksi, että peli toimisi useammalla laitteella, jotta saataisiin mahdollisimman iso määrä potentiaalisia pelaajia tai käyttäjiä. Potentiaalisilla pelaajilla tarkoitetaan pelin kohderyhmää, eli pelaajia, joille peli on suunnattu. Mitä suurempi kohderyhmä pelillä on, sitä enemmän pelaajia sillä voi olla. Monet ihmiset eivät osta kalliita älypuhelimia tai isoja pelitietokoneita pelatakseen pelejä, joten he jättävät usein uudet, laitteistolle vaativat pelit ja sovellukset ostamatta. Jos peli suunnitellaan vain uusimpia laitteita ajatellen ja optimointi unohdetaan täysin, pelin pelaajamäärät eivät voi olla suuremmat kuin uusien, kalliiden laitteiden omistajien määrä. Jos peli optimoidaan kunnolla, voidaan saada jopa kymmeniä miljoonia uusia pelaajia. Optimoimalla peli voidaan lisäksi vähentää sen aiheuttamaa virrankulutusta, joka on tärkeää varsinkin mobiililaitteilla. Jos mobiililaitteen akku tyhjenee liian nopeasti, saattaa pelaaja lopettaa pelin pelaamisen kokonaan.

Unityssä on monia sisäänrakennettuja ja helppoja keinoja optimoida sen avulla tehtyjä videopelejä, joiden käyttöön ottaminen on järkevä tapa saada lisää pelaajia ja peli toimimaan nopeammalla ruudunpäivitysnopeudella (framerate). Tämä tutkimustyö auttaa ymmärtämään yleisimpien visuaalisten optimointimenetelmien toimintaa ja sitä, missä tilanteissa niitä on järkevintä käyttää, jolloin niistä voidaan hyötyä parhaiten. Visuaaliset keinot optimoida pelejä ovat todella tärkeitä jokaisessa 3D-pelissä, elleivät jopa tärkeimpiä, sillä renderöinti on yksi raskaimmista suorituskykyyn vaikuttavista seikoista videopeleissä. Visuaaliset optimointimenetelmät keskittyvät pääasiassa 3D-objektien pintojen lukumäärään ja valaistukseen eli toisin sanoen renderöintiin.

Jotta visuaalisten optimointimenetelmien hyödyt saadaan selville tässä tutkimustyössä, ne otetaan käyttöön Unity-pelimoottorissa luodussa yksinkertaisessa peliprojektissa, ja niiden vaikutusta suorituskykyyn tarkkaillaan Unity-pelimoottorin sisäänrakennetun Profiler-työkalun avulla. Optimointimenetelmän käyttämisen aikana suorituskykyä tarkkaillessa saatuja tuloksia verrataan tuloksiin, jotka saadaan tarkkailemalla suorituskykyä, kun testattava optimointimenetelmä ei ole käytössä. Tuloksia vertaillessa analysoidaan, milloin optimointimenetelmän käyttäminen on hyödyllistä peliprojektin kannalta ja milloin sen käyttäminen on esimerkiksi liian hidasta, haitallista tai hyödytöntä peliprojektissa.

Tästä tutkimustyöstä hyötyvät eniten pelinkehittäjät, joiden tarkoituksena on luoda tavallisille kuluttajille 3D-peli, jossa on suuri määrä 3D-objekteja. Tuloksista voidaan päätellä, millaisessa tilanteessa tiettyjä visuaalisia optimointikeinoja kannattaa käyttää, ja milloin niistä on enemmän haittaa optimoinnin kannalta kuin hyötyä. Tämän tutkimustyön avulla pelin visuaalinen optimointi voidaan suunnitella helpommin etukäteen, jolloin optimoinnista on maksimaalisin hyöty.

## 2 Visuaalinen optimointi

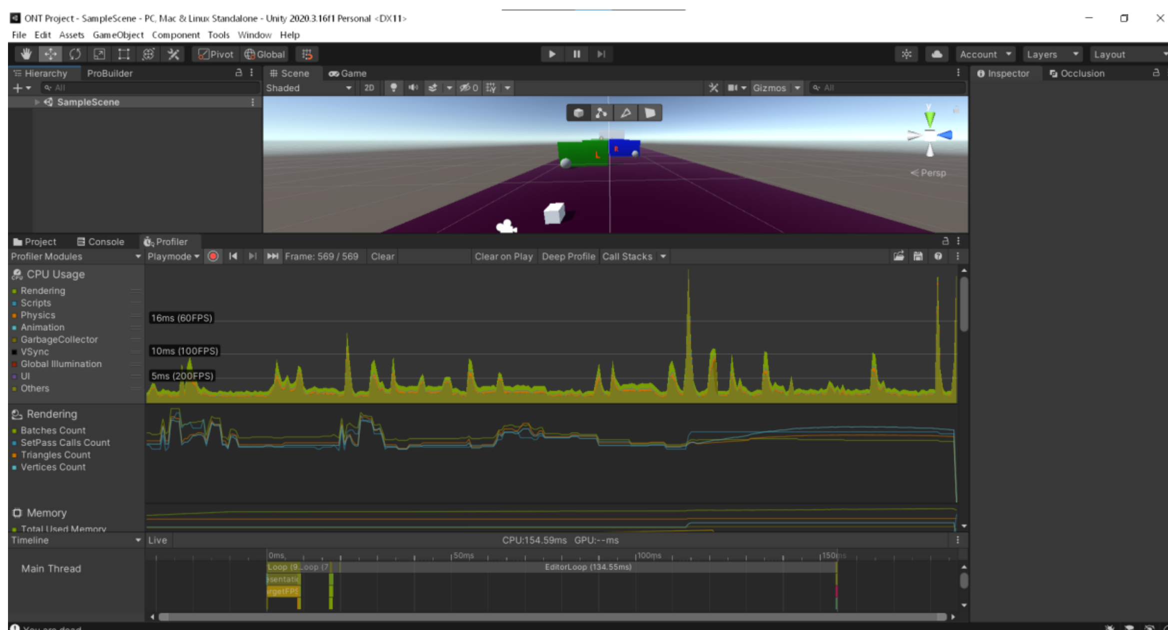
Visuaalisella optimoinnilla tarkoitetaan videopelien optimointikeinoja, jotka liittyvät pelissä näkyviin visuaalisiin ominaisuuksiin eli esimerkiksi pelissä näkyviin 3D-malleihin ja pelin valaistukseen. Toisin sanoen siihen kuuluu peliobjektien renderöintiin liittyvät optimointikeinot. Pintojen määrän tulee olla mahdollisimman pieni ja valaistus eli valon kirkkaus, sen heijastuminen pinnoilta muille pinnoille ja sen aiheuttamat varjot tulisi laskea enimmäkseen etukäteen, ettei valaistusta tarvitsisi laskea jatkuvasti ajonaikaisesti. Optimoinnilla pyritään siis pohjimmiltaan ajonaikaisten laskutoimitusten minimointiin. Tärkeimmät ja yksinkertaisimmat visuaaliset optimointimenetelmät ovat objektien piilottamiseen, 3D-malleihin ja valaistukseen liittyvät menetelmät. Visuaaliseen optimointiin ei siis kuulu optimointikeinot, joiden käyttöä pelaaja ei voi huomata, sillä ne eivät liity pelin ulkonäköön. Näitä keinoja ovat ohjelmointiin liittyvät optimointikeinot, joita ovat esimerkiksi useissa pelimoottoreissa yleisesti käytettävä Object Pooling ja Unityn omat keinot, joita ovat esimerkiksi ScriptableObjectien käyttäminen ohjelmoinnissa ja Unity DOTS -systeemin hyödyntäminen. Pelaaja ei voi tietää milloin ei-visuaalisia optimointikeinoja on käytetty, vaikka niiden avulla voidaan vaikuttaa myönteisesti pelin suorituskykyyn. Kun visuaalisia optimointikeinoja käytetään pelissä, niiden käyttämisen voi useimmiten huomata, mutta tarkoituksena on kuitenkin, että ne ovat pelaajalle lähes huomaamattomia. Usein visuaalisten keinojen käyttämisen voi kuitenkin huomata pelin aikana. (Unity Technologies 2021t.)

Visuaaliset optimointikeinot ovat tärkeitä peleissä, koska niitä käyttämällä voidaan parantaa huomattavasti suorituskykyä ja käyttämättä niitä peli saattaa olla pelaamiskelvoton, koska sitä pelatakseen pelaaja tarvitsisi äärimmäisen tehokkaan pelaamiseen tarkoitetun laitteen, jotta pelaaminen olisi mielekästä. Suurin osa näistä keinoista ovat helppokäyttöisiä, mutta niiden käyttämiseen tarvitaan tietoa niiden toimintatavasta ja toteutuksesta. Kun optimointikeinon toimintatapa tiedetään, voidaan päätellä, missä tapauksessa sitä kannattaa käyttää. Jos visuaalisia optimointikeinoja käytetään väärässä paikassa, niiden käyttämisestä saattaa aiheutua suorituskyvyn kannalta enemmän haittaa kuin hyötyä. Optimointikeinojen hyödyllisyydestä voidaan saada tietoa Unityssä helpoiten Profiler-työkalua käyttämällä. (Unity Technologies 2021t.)

# 3 Suorituskyvyn tarkkailu

Videopelien optimointi on niin tärkeä asia videopeleissä, että useissa pelimoottoreissa on erilaisia työkaluja, joiden avulla voidaan tarkastella, kuinka hyvin optimoitu peli on. Tätä kutsutaan usein profiloinniksi. Esimerkiksi huonosti optimoitu alue voidaan huomata helpommin käyttämällä tällaista työkalua. Unityn profilointityökalu Profilerin, ja sen useiden eri moduulien avulla voidaan tarkastella laajasti useita eri optimoinnin osa-alueita kuten esimerkiksi ääntä, valaistusta tai GPU:n eli grafiikkaprosessorin kuormitusta. (Unity Technologies 2021o.)

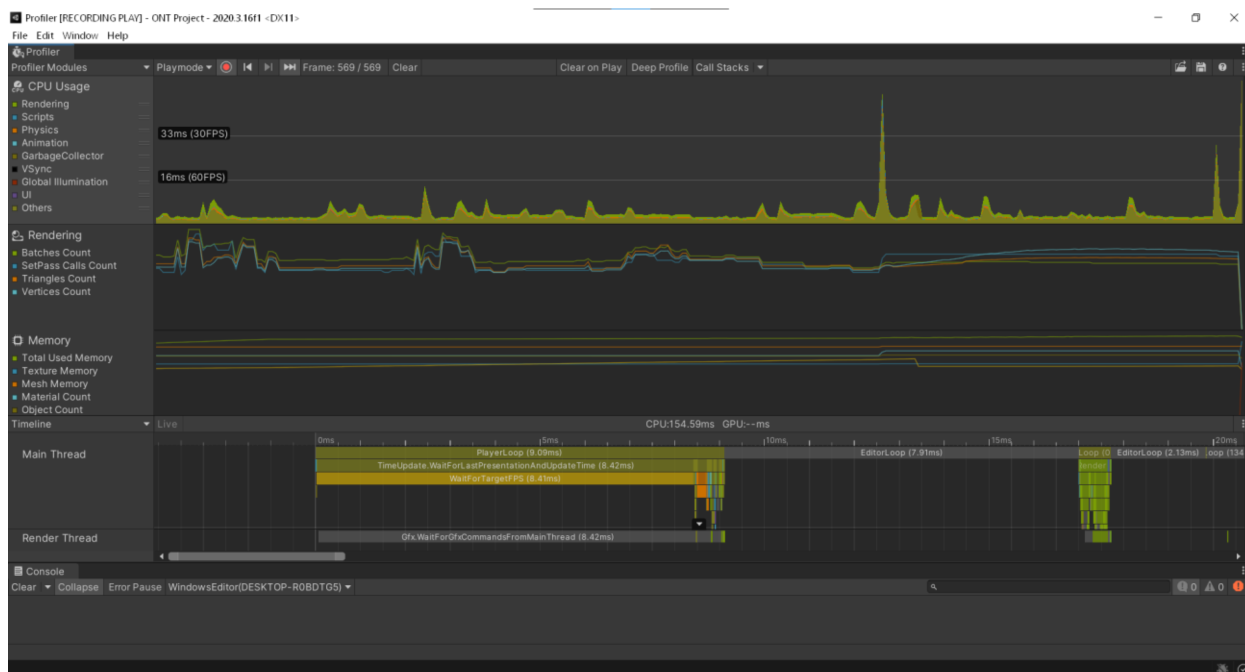
Profiler tallentaa yksityiskohtaista tietoa jokaisesta framesta oletuksena viimeisen 300 framen ajalta. Profilerin näyttämiä tietoja tarkastelemalla saadaan selville, mikä kuormitukseen on syypää. Keskittymällä suurimpiin piikkeihin Profilerin luomassa kaaviossa, saadaan selville, mikä aiheuttaa suurimmat suorituskykyongelmat pelissä. Klikkaamalla piikin kohdalta, saadaan tietoa prosesseista, jotka aiheuttivat piikin. Kuvassa 1 esimerkki Profilerin näkymästä Unity Editorin välilehdellä. (Unity Technologies 2021p.)




Kuva 1. Profiler näkymä Unity Editorin välilehdellä

Profilerin avulla eri optimoinnin osa-alueita voidaan tarkastella myös erikseen. Useita osa-alueita varten on olemassa moduuli. Jokaisella moduulilla on oma kaavionsa, joista pystyy tarkastelemaan tietyn optimoinnin osa-alueen suorituskykyä. Jotkut moduulit ovat oletuksena poissa päältä, sillä ne vaikuttavat haitallisesti pelin suorituskykyyn testauksen aikana. Esimerkiksi GPU-moduuli on poissa päältä, eikä se edes toimi useilla laitteilla. (Unity Technologies 2021p.)

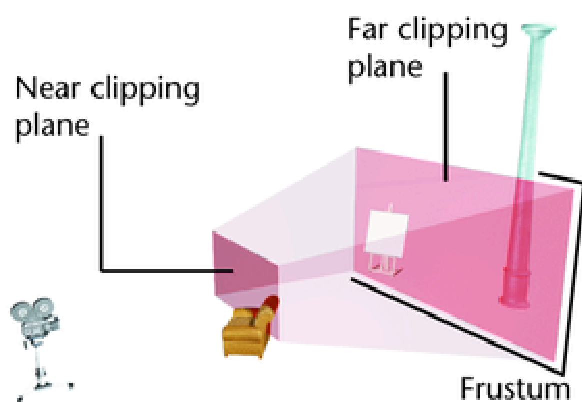
Profilerin voi käynnistää erillisenä prosessina, jolloin se aukeaa omaan ikkunaansa. Tämä tapa on hyödyksi, sillä Profilerin omat prosessit vaikuttavat vähemmän pelistä kerättyyn optimointidataan, jolloin data on täten ”puhtaampaa” eli vastaa paremmin todellisuutta, koska pelin ulkopuoliset prosessit eivät vaikuta yhtä paljoa kerättyyn dataan. Kuten kuvassa 2 nähdään, Profiler näyttää kuitenkin täysin samalta kuin Unity Editorissa, mutta näkymä on suurempi. Ikkuna avautuu kuitenkin hitaammin eikä prosessia voi yhdistää missään vaiheessa Unity Editorin ikkunaan. (Unity Technologies 2021p.)




Kuva 2. Profiler avattuna erilliseen ikkunaan

## 4 Clipping Planes

Frustum Culling on Unityssä automaattisesti toimiva renderöintiin liittyvä toiminto, jonka tarkoituksena on olla renderöimättä esineitä, jotka eivät ole kameran muodostaman näkökentän eli frustumin sisällä. Se tarkoittaa kameran näkökenttää, joka on leikatun kartion muotoinen alue, joka on Near Clipping Planen ja Far Clipping Planen välissä. Clipping Planes -optimointimenetelmän toiminta perustuu näiden kahden Clipping Planen manipuloimiseen, eli kameran näkymän koon muuttamiseen. Near Clipping Plane määrittää, kuinka lähellä kameraa esineet voivat näkyä. Jos kamera on liian lähellä esinettä, se ei näy kameran näkymässä, eikä siis renderöidy. Far Clipping Plane määrittää, kuinka kaukana kameraa esineet voivat näkyä. Jos esine on liian kaukana kamerasta, vaikka pelaaja katsoisikin esinettä kohti, se ei renderöidy. Ainoastaan esineet, jotka ovat näiden kahden Clipping Planen välille jäävän tilan eli frustumin sisällä renderöityvät ja näkyvät pelaajalle. Kuvassa 3 voidaan havannoida visuaalisesti, miten kameran näkymä eli frustum muodostuu ja miten Clipping Planet toimivat. Tämä on menetelmän yleisin muoto, mutta menetelmä voi olla joissain tapauksissa myös erilainen – Clipping Planeja voi olla esimerkiksi enemmän ja ne voivat olla myös suhteessa 3D-malliin sen sijaan, että ne olisivat suhteessa kameraan. (Unity Technologies 2021f; PTC; Autodesk 2020.)




Kuva 3. Near ja Far Clipping Plane (Autodesk 2020)

Unityssä on käytössä Clipping Planen perinteinen muoto, joten kameralle täytyy määritellä Near Clipping Plane ja Far Clipping Plane. Unityssä valitaan siis kamera, jolle Clipping Planet halutaan asettaa. Inspectorissa Clipping Planes -kohdasta valitaan arvot Near- ja Far-kohtiin. Near Clipping Planea valitessa kannattaa miettiä, millä etäisyydellä esineet saattavat häiritä näkyvyyttä ja kuinka läheltä esineet on tarpeellista nähdä. Ylhäältä päin kuvatuissa peleissä Near Clipping Plane on yleensä kauempana kuin esimerkiksi First Person Shooter -tyyppisissä peleissä. Far Clipping Planen etäisyys on usein suuri, jotta pelikenttä

ei yhtäkkiä leikkautuisi kahtia jossain kohtaa. Ideaalinen etäisyys pelissä, jossa ei ole paljon todella suuria avoimia tiloja, on maksimietäisyys pelikentän suurimman kohdan laidasta laitaan. Far Clipping Plane ei ole kuitenkaan esimerkiksi vanhoissa Valven Source Enginellä tehdyissä peleissä kuten Left 4 Dead 2:ssa erityisen suuri, vaikka siinä voisikin periaatteessa nähdä melko kauas. Pelikenttä leikkautuu useissa Valven peleissä yhtäkkiä Far Clipping Planen mukaan, mutta kukaan ei huomaa tätä, sillä ongelma on hoidettu kiinnostavalla ja toimivalla tavalla. Hieman ennen Far Clipping Planea pelikenttään ilmestyy sumua, jonka taakse ei voi nähdä. Sumu näyttää luonnolliselta ja sen avulla voidaan helposti vähentää Far Clipping Planen etäisyyttä ja näin optimoida peliä tehokkaasti. Huonona puolena on joissain tapauksissa visuaalisen ilmeen huonontuminen, esimerkiksi silloin, jos pelaajalle halutaan näyttää kaukana sijaitsevaa maisemaa. (Unity Technologies 2021k; Valve Developer Community e.)

Unityssä eri layereille voidaan lisäksi määrittää eri Clipping Planes -arvot, joka mahdollistaa sen, että jotkut objektit voidaan piilottaa näkymästä Clipping Planesia käyttämällä ennen kuin toiset objektit. Tämä on hyödyllisintä pienten objektien kanssa, jotka kannattaa piilottaa ennen suuria objekteja, jotka pelaaja pystyy näkemään helposti kaukaakin. Pienet objektit täytyy siis asettaa eri layerille ja tämän jälkeen koodissa käytetään funktiota ”Camera.layerCullDistances”, jotta layerille saadaan asetettua eri Clipping Plane -arvot. (Unity Technologies 2021k.)

Clipping Planes -menetelmä perustuu Frustum Culling -toiminnon manipuloimiseen, mutta sen avulla ei voida korvata Occlusion Culling -menetelmää, sillä se piilottaa ainoastaan objektit, jotka eivät ole kameran muodostaman potentiaalisen näkökentän sisällä. Menetelmä on myös hyödytön niissä sovelluksissa ja peleissä, joissa kaikki objektit ovat jatkuvasti kameran potentiaalisessa näkökentässä eli frustumissa. (Unity Technologies 2021b; Unity Technologies 2021f.)

# 5 Culling

## 5.1 Occlusion Culling

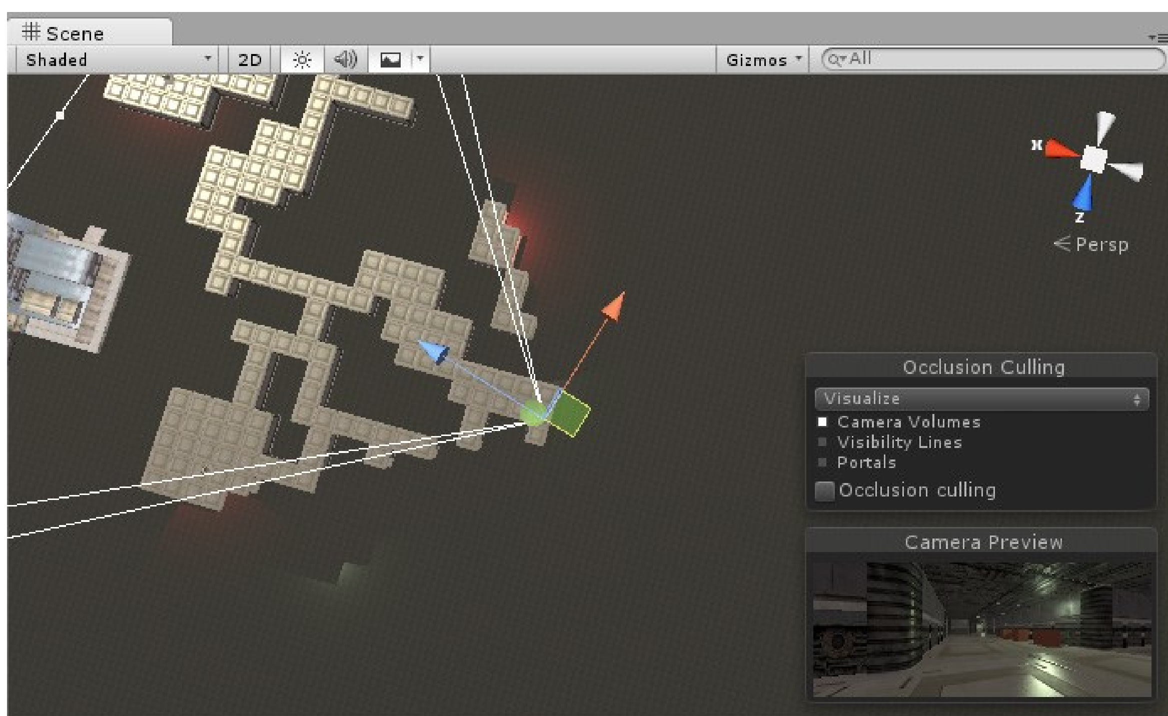
Occlusion Culling -menetelmän ideana on lyhyesti sanottuna olla renderöimättä objekteja, joita ei tällä hetkellä näy kamerassa — eli pelaajan ruudulla — jonkin toisen objektin peittäessä ne näkyvistä. Menetelmän toimivuus nojaa siis suurelta osin näköesteisiin. Tämä on yksi yleisimmistä peleissä käytettävistä optimointimenetelmistä. Menetelmästä on kaksi erilaista muotoa, Potentially Visible Set eli PVS ja Portal rendering. (Unity Technologies 2021b.)

Menetelmä on erittäin tehokas esimerkiksi sellaisissa peleissä tai niiden osissa, joissa on paljon käytäviä ja suljettuja tiloja. Esimerkiksi tunnetut pelit Doom 3 ja Half-life 2 käyttävät paljon kyseistä menetelmää, joskin pelien tekemiseen on käytetty eri pelimoottoreita. Jos pelissä on paljon aukeita paikkoja, joista voi nähdä koko pelimaailman tai erittäin suuren osan pelimaailmasta, tämä menetelmä ei hyödytä optimoinnissa lähes millään tavalla. Esimerkkinä tästä voidaan ottaa Elder Scrolls -pelisarjan pelit, joissa voi paikoin nähdä suuria ja näyttäviä maisemia todella kaukana pelaajasta. Tällaisessa pelissä ongelmana on siis menetelmän lähtökohtana olevien näköesteiden vähyys. Kannattaa myös huomata Potentially Visible Set -muotoa käytettäessä, että Unityn sisäänrakennettu menetelmä säästää grafiikkaprosessori (Graphics Processing Unit eli GPU) aikaa suoritin (Central Processing Unit eli CPU) ajan kustannuksella, sillä CPU laskee ajonaikaisesti näkyvyyttä näkyvyysdatan avulla, jonka takia CPU:lle hyödyt ovat pienemmät. Mikäli pelin tiedetään olevan raskas GPU:lle, kannattaa ottaa menetelmä käyttöön. CPU hyötyy menetelmästä vähemmän. (Unity Technologies 2017; Unity Technologies 2021b.)

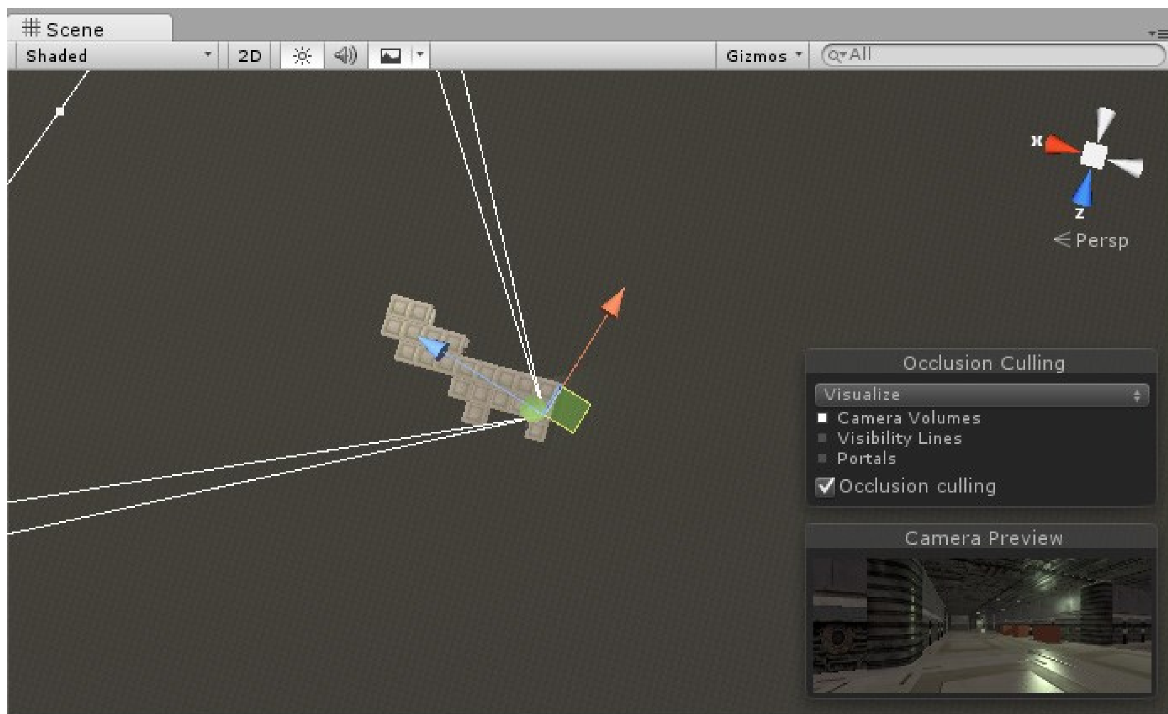
## 5.2 Potentially Visible Set

PVS toimii useissa pelimoottoreissa siten, että pelimoottori jakaa pelikentän osiin ja laskee etukäteen, millä alueilla ollessa pelaaja voi nähdä minkäkin alueen ja millä alueilla tiettyjä alueita ja tietysti niillä olevia objekteja ei voi nähdä näköesteen takaa. Pelimoottori käyttää tätä dataa pelin aikana piilottaakseen geometrian, jota pelaaja ei voi nähdä, säästäen näin GPU- ja CPU-aikaa, jota kuluisi turhaan näkymässä näkymättömien esineiden renderöintiin. Laskeminen suoritetaan etukäteen, sillä sen laskeminen reaaliajassa olisi niin raskasta laitteistoille, että siitä ei olisi tarpeeksi hyötyä. Pelimoottorin täytyy laskeakseen tietää, mitkä objektit ovat näköesteitä ja mitkä eivät. Koska pelimoottorit laskevat näkyvyyslaskelmat etukäteen, ne eivät voi käyttää näköesteinä esimerkiksi objekteja, jotka voivat liikkua tai muuttua jollain tavoin pelin aikana. Näköesteinä kannattaa siis käyttää mieluiten esimerkiksi rakennusten lattioita, seiniä ja kattoja, sillä nämä ovat yleensä staattisia objekteja peleissä.

Tärkeää on myös, että objektien läpi ei voi nähdä. Myöskään objektit, jotka käyttävät myöhemmin työssä käsiteltävää LOD-menetelmää, eivät ole järkeviä kohteita näköesteiksi, sillä mallin ääriviivat voivat usein muuttua hieman pelin aikana. Jos ääriviivat eivät muutu, LOD-menetelmää käyttäviä objekteja voi myös käyttää huoletta. Objektin kuuluisi kuitenkin pysytellä lähes muuttumattomana pelin aikana. Kuvissa 4 ja 5 nähdään esimerkki optimointimenetelmän käyttämisen hyödyistä. Optimointimenetelmää on käytetty tässä tapauksessa pelikentässä, jossa on paljon käytäviä ja suljettuja tiloja. Kuvissa pelikenttä on kuvattu ylhäältäpäin. Kuvassa 4 näkyvässä Unityn scenessä menetelmä ei ole päällä, jolloin huomataan, että myös objektit, jotka ovat piilossa pelaajalta renderöityvät, täysin turhaan. Kuvassa 5 näkyvässä Unityn scenessä, jossa menetelmä on päällä, voidaan huomata, kuinka ainoastaan tarvittavat eli pelaajan näkökentässä olevat objektit renderöityvät. (Courrèges, A. 2016; Unity Technologies 2017; Unity Technologies 2021b.)




Kuva 4. Ylhäältä päin kuvattu Unityn scene, jossa PVS ei ole käytössä (Unity Technologies 2021b)



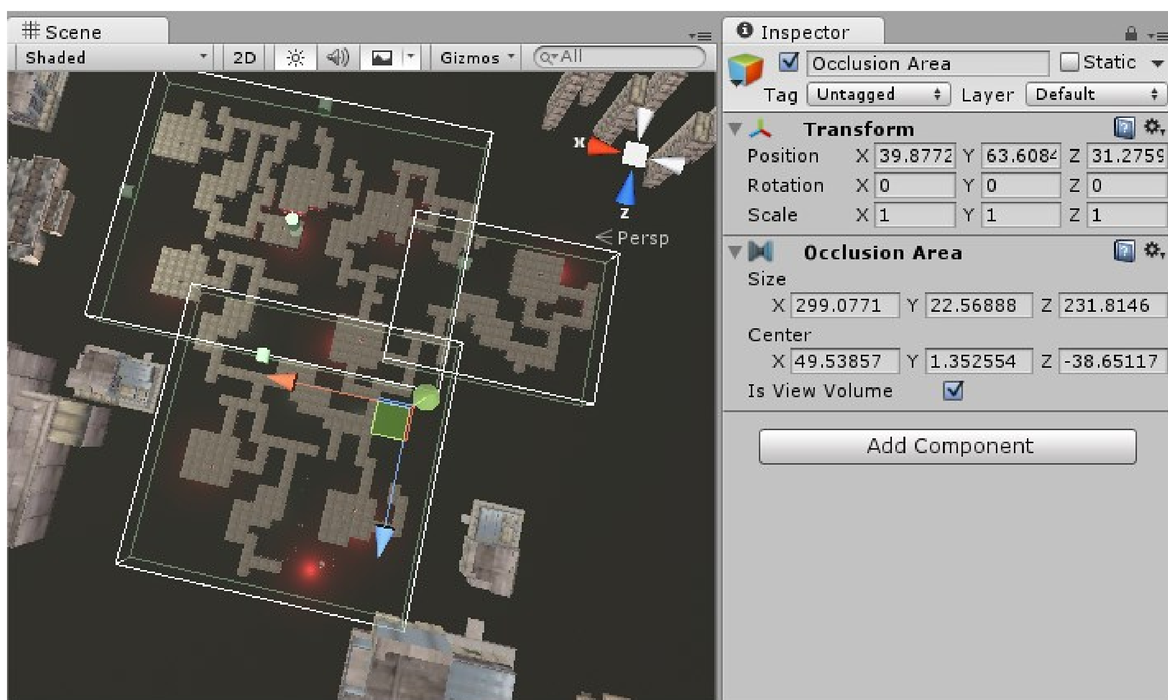

Kuva 5. Ylhäältä päin kuvattu Unityn scene, jossa PVS on käytössä (Unity Technologies 2021b)

Unityssä näköeste on nimeltään Occluder ja piilotettava esine on nimeltään Occludee. Eräistä muista pelimoottoreista poiketen jopa piilotettavat esineet täytyy merkitä Unityssä manuaalisesti. Staattinen eli koko pelin ajan muuttumattomana ja liikkumattomana pysyttelevä objekti voi olla sekä Occluder Static, että Occludee Static samanaikaisesti, mutta dynaaminen eli peliaikana jotenkin muuttuva tai liikkuva esine voi olla ainoastaan Occludee, koska dynaamiset objektit eivät voi piilottaa muita esineitä. Unity myös yhdistää automaattisesti joitakin näkyvyysalueita toisiinsa, kun se on mahdollista, jotta näkyvyysdatan tiedostokoko ei kasvaisi liian suureksi (Unity Technologies 2021b). Kun tarkastellaan muita pelimoottoreita, huomataan, että muun muassa Valve Corporationin kehittämä Source Engine käyttää samaa tekniikkaa peleissä kuten Left 4 Dead 2, Half-Life 2 ja Team Fortress 2. Esimerkiksi ainakin joillakin Source Enginen versioilla alueiden yhdistäminen täytyy määritellä manuaalisesti, mikä on melko hidasta ja kömpelöä. (Unity Technologies 2017; Unity Technologies 2021b; Epic Games a; Valve Developer Community a.)

On tärkeää huomata, että menetelmä ei toimi Unityssä esimerkiksi peleissä, joissa pelikenttä generoidaan sattumanvaraisesti pelin alkaessa, sillä menetelmän toimintaperiaatteena on näkyvyyslaskelmien laskeminen etukäteen. Menetelmää voidaan simuloida tällaisessa pelissä manuaalisesti ohjelmointikeinoin, esimerkiksi poistamalla käytöstä tiettyjen objektien Mesh Renderer pelaajan ollessa alueella, jolla esinettä ei voi nähdä, mutta tämä voi olla työläs prosessi manuaalisesti hoidettavaksi. Erityisesti tässä tapauksessa on

järkevää käyttää menetelmän Portal rendering -muotoa, vaikkakin sen käyttäminen on myös muulloin kannattavaa tietyissä tapauksissa. (Unity Technologies 2021b; Unity Technologies 2021g.)

Menetelmän toimintaa voidaan parantaa huomattavasti lisäämällä pelialueelle etukäteen suoritettavia laskutoimituksia eli näkyvyysdatan luomista varten Occlusion Area -komponentti, jonka avulla voidaan määritellä, millä alueilla pelaaja tulee oletettavasti olemaan pelin aikana. Komponentteja voi olla myös useita samaan aikaan, kuten esimerkiksi kuvassa 6, jossa kolme Occlusion Area -komponenttia ympäröi pelialueen. Occlusion Area -komponenttien muodostamaa aluetta kutsutaan nimellä View Volume. Jos oletettavaa pelialuetta eli View Volumea ei ole määritetty, Unity olettaa pelialueen koon automaattisesti Occludee ja Occluder Static -merkittyjen objektien perusteella. Tämä saattaa aiheuttaa paljon turhia laskutoimituksia, jotka voivat johtaa datan luomisen hitauteen, aiheuttaa enemmän kuormaa CPU:lle ajonaikana ja datan tiedostokoko saattaa kasvaa tarpeettoman suureksi. (Unity Technologies 2021h.)

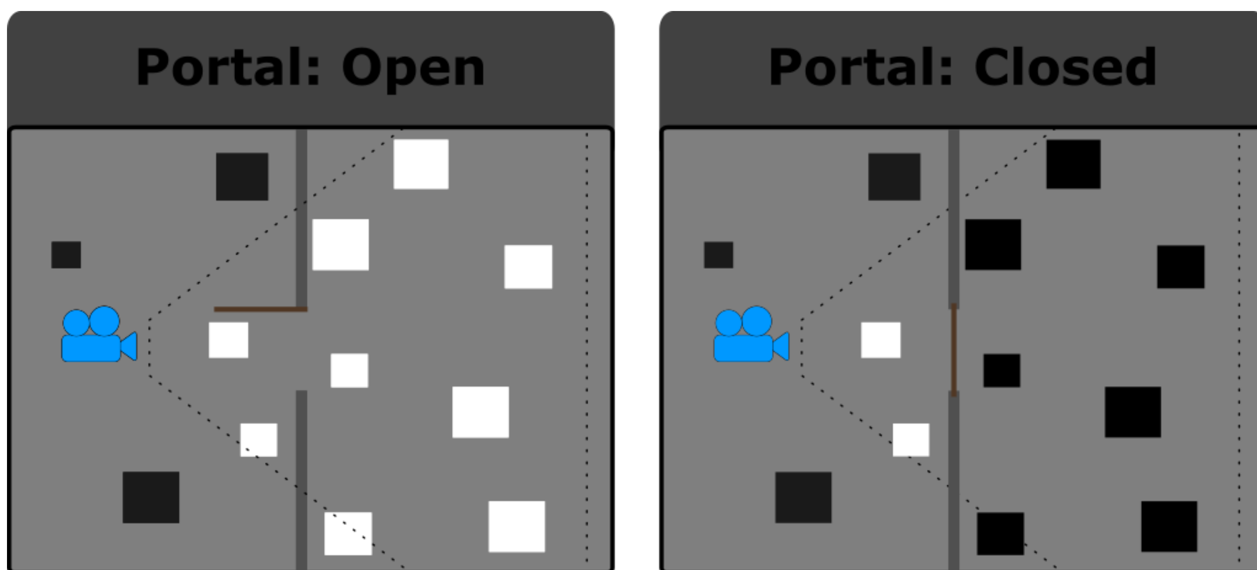



Kuva 6. Kolme Occlusion Area -komponenttia pelikentän ympärillä (Unity Technologies 2017)

## 5.3 Portal rendering

Tämä Occlusion Culling -menetelmän muoto on täysin erilainen kuin Potentially Visible Set. Sen toiminta perustuu ”portaaleihin”, joiden ollessa kiinni, objekteja ei renderöidä, mutta portaalin ollessa auki, objektit renderöidään. Portaaleja käytetään usein ovien yhteydessä.

Oven ollessa auki, pelaaja voi nähdä alueelle, joten alue renderöidään, mutta oven ollessa kiinni, pelaaja ei voi nähdä alueelle, joten aluetta ei renderöidä. Kun pelaaja astuu alueelle, jolla on aiemmin huomattu suorituskykyongelmia, voidaan esimeriksi sulkea ovi, jolloin pelaaja ei näe edelliselle alueelle ja portaali piilottaa tämän alueen objektit parantaen siten suorituskykyä uudella alueella. Oven pitäisi olla läpinäkymätön, rikkoutumaton ja sen täytyisi sulkeutua automaattisesti, sillä pelaaja saattaisi jättää oven auki ja tällöin menetelmää ei ikinä edes välttämättä otettaisi käyttöön. Kuviossa 1 havainnollistetaan portaalin toimintaa ylhäältäpäin kuvattuna. Nelikulmiot edustavat piilotettavia eli pelissä renderöitäviä objekteja. Valkoinen väri tarkoittaa, että objekti on renderöity ja musta väri tarkoittaa, että objekteja ei ole renderöity. Sininen objekti on kamera ja mustilla katkoviivoilla rajattu alue edustaa kameran näkymää eli view frustumia. Vasemmanpuoleisessa skenaariossa ovi ja portaali ovat auki, jolloin kaikki huoneessa olevat esineet, jotka ovat kameran näkymän sisällä, on renderöity. Oikeanpuoleisessa skenaariossa ovi ja portaali ovat kiinni, jolloin huoneessa, jonka portaali sulkee sisäänsä, ei renderöidy mikään objekti, vaikka ne olisivat kameran näkymän sisäpuolella. (Unity Technologies 2021g.)




Kuvio 1. Portaalin toiminta, kun se on liitetty oveen

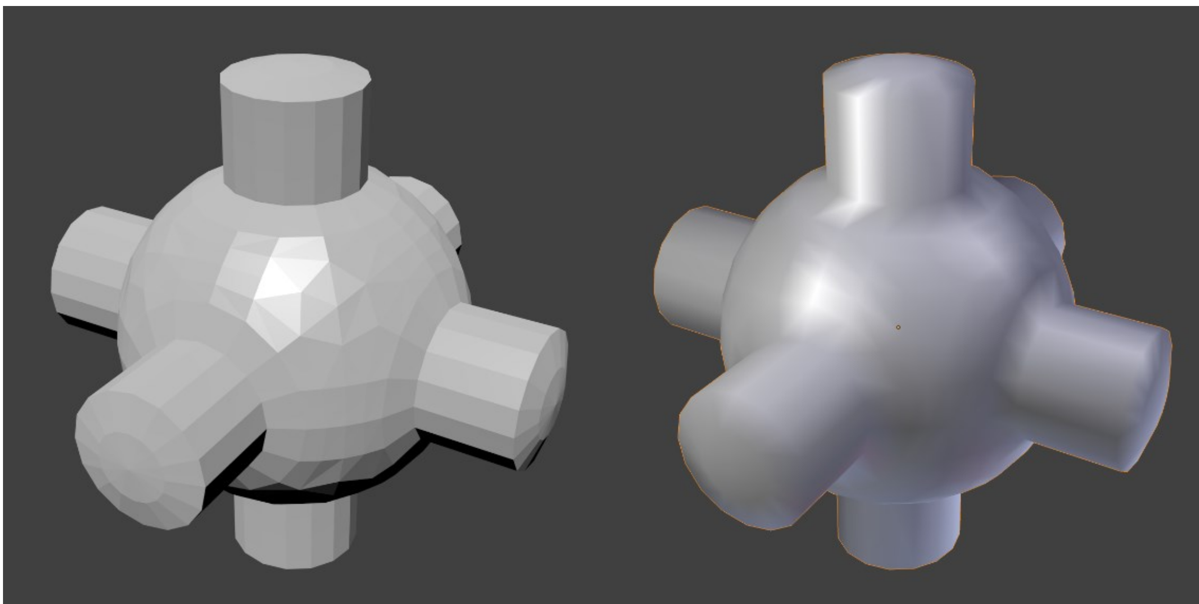
Portaaleja voi asettaa pelikenttään myös ilman minkäänlaista ovea, mutta portaali täytyisi sulkea jonkun muun tapahtuman aikana ja varmistaa, ettei pelaaja näe portaalia ja sen piilottamia objekteja. Unity-pelimoottorin manuaalissa portaaleja kutsutaan nimellä Occluder Portals. (Unity Technologies 2021g.)

# 6 3D-mallit

## 6.1 3D-mallien optimoinnin perusteet

3D-mallien pintojen määrä on tärkeässä roolissa pelien optimoinnin kannalta. Mitä enemmän pintoja kuvaruudulla on, sitä raskaampaa se on laitteistolle. Pintojen määrän minimoiminen on yksi tärkeimmistä optimointiin liittyvistä asioista videopeleissä. 3D-mallien suunnittelussa täytyy siis ottaa huomioon pintojen määrä ajatellen samalla kuitenkin mallin ulkonäköä. Mitä vähemmän mallissa on pintoja, sitä enemmän ulkonäkö kärsii. Mallista ei saa kuitenkaan tehdä liian yksityiskohtaista, jotta se ei vaikuttaisi haitallisesti pelin pelattavuuteen. (Unity Technologies 2021a; Unity Technologies 2021r; Arm Limited a.)

Usein 3D-mallit tai niiden kulmat ovat pyöreitä. Jotta kappaleen kulmista saadaan oikeasti pyöreitä, tarvitaan erittäin suuri määrä pintoja. 3D-mallien käyttämisestä tulisi kuitenkin mahdotonta, jos jokaista pyöreää kohtaa kohti tarvittaisiin tuhansia pintoja. Tämän vuoksi on olemassa keino, jonka avulla voidaan luoda illuusio kulmien tai kappaleiden pyöreydestä. Smooth Shading, toiselta nimeltään Smoothing, joka tarkoittaa tasoittamista, on yleinen 3D-malleissa käytetty temppu, jossa objektin pintojen kulmat saadaan näyttämään tasaisilta, vaikka todellisuudessa objektin kulmat ovat huomattavasti epätasaisempia. Illuusio saadaan luotua heijastamalla valoa 3D-mallin pinnoista tietyllä tavalla, jolloin kulmat näyttävät katoavan ja objektin pinnasta (surface) tulee sileän näköinen. Objektin siluetti eli ääriviivat eivät kuitenkaan muutu millään tavalla, joten esimerkiksi siluetin sileää kulmaa sivusta katsottuna voidaan huomata selvästi sen pintojen määrä. Smooth Shading on vanha 3D-mallintamisen perusteisiin kuuluva keino vähentää objektin pintoja ja sitä käytetään lähes poikkeuksetta kaikissa 3D-malleissa, joissa on pyöreää pintaa. Kuvassa 7 nähdään esimerkkinä vierekkäin kaksi 3D-mallia, jotka ovat muuten täysin samanlaisia, mutta oikeanpuoleisessa kuvassa Smooth Shading on käytössä. (Blender Foundation 2020c; Autodesk 2017.)



Kuva 7. Smooth Shading (Blender Foundation 2020c)

## 6.2 Normal map

Joidenkin 3D-mallien yksityiskohtien luontiin voidaan käyttää menetelmää, joka saa esineen näyttämään siltä, että siinä olisi paljon yksityiskohtia, vaikka todellisuudessa pintojen määrä onkin alhainen. 3D-mallille voidaan luoda normal map. Normal Map -menetelmää käytetään normaalivektorien laskemiseen ja tallentamiseen yksityiskohtaisesta mallista yksinkertaisen 3D-mallin normal map -tekstuuriin, josta saatua tietoa käytetään esimerkiksi pelimoottoreissa valaistuksen ja varjostuksen laskemiseen (Blender Foundation 2017). Näin saadaan luotua illuusio, jossa 3D-malli, jolla on vähän pintoja, saadaan näyttämään yksityiskohtaiselta, vaikka näin ei oikeasti ole. Kuvassa 8 nähdään esimerkki Normal Map -menetelmän hyödyistä. Yksinkertaiselle mallille voidaan usein luoda normal map yksityiskohtaisen mallin avulla. (Unity Technologies 2021j; Blender Foundation 2020b.)

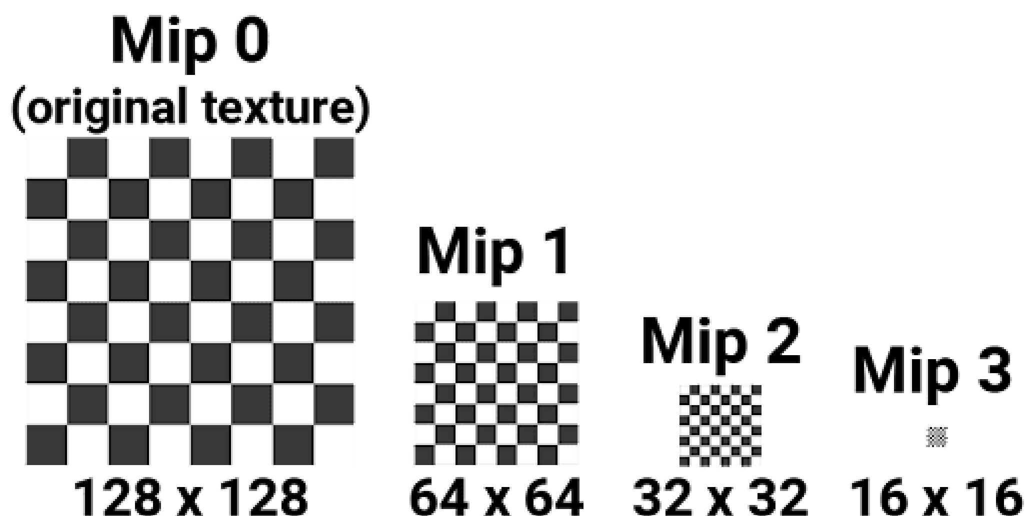
Kuva 8. Yksityiskohtainen malli, yksinkertainen malli ja yksinkertainen malli normal mapin kanssa (Blender Foundation 2020b)

## 6.3 Mipmaps

Tekstuurien koko vaikuttaa renderöintinopeuteen ja siten myös GPU:n kuormitukseen. Suuren tekstuurin yksityiskohtia ei kuitenkaan huomaa kaukaa katsottuna. Sen ansiosta olisi järkevää käyttää resoluutioltaan pienempiä tekstuureja pelaajan ollessa kaukana objektista. Näin mipmapit toimivat. Mipmap on tekstuuri, johon sisältyy monta eri resoluutioista versiota tekstuurista eli mipiä. Näitä eri resoluutioisia tekstuurin versioita kutsutaan nimellä mip ja menetelmää kutsutaan tämän vuoksi nimellä Mipmaps. (Unity Technologies 2021q; Epic Games c.)

Mipmapeja käytetään yleisesti useissa eri pelimoottoreissa ja niistä on muutakin kuin suorituskykyyn liittyvää hyötyä. Suuret tekstuurit voivat kaukaa tietyssä kuvakulmassa katsottuna aiheuttaa objektien pinnoille epämuodostumia, Moiré-kuvioiksi kutsuttuja eräänlaisia häiriöitä. Kun suuri tekstuuri vaihdetaan pienempiresoluutioiseen tekstuuriin, nämä häiriöt katoavat. Käyttämällä mipmapeja oikealla tavalla, voidaan siis lisäksi myös kohentaa pelin visuaalista puolta. (Unity Technologies 2021q; Valve Developer Community g; Epic Games c.)

Mipmapeja käytetään yleensä tekstuureissa 3D-objekteissa, joiden etäisyys kamerasta voi vaihdella pelin aikana paljon. Pelaaja voi siis nähdä esineen pelin aikana läheltä ja kaukaa. GPU laskee tekstuurin etäisyyden ja kameran kulman suhteessa tekstuuriin ja päättelee kuinka paljon yksityiskohtia pelaaja voi nähdä tekstuurista. Laskeminen perustuu pikseleihin, joten pelaajan laitteen ruudun resoluutio vaikuttaa laskutoimitukseen. Tämän perusteella GPU valitsee, mitä mip-tasoa se käyttää. Mitä lähempänä ja pienemmässä kulmassa kamera on esineeseen nähden, sitä suurempiresoluutioisen mip-tason GPU valitsee. Kun Trilinear filtering -asetus on päällä, Unity voi tehdä vaihdon eri mip-tasojen välillä huomaamattomammaksi sekoittamalla kahden eri mip-tekstuurin informaatiota toisiinsa. Alkuperäisellä eli suurimmalla mipillä on Unityssä taso 0. Pieni resoluutioisimmalla mipillä on suurin taso, kuten kuvasta 9 huomataan. Seuraavan mip-tason tekstuuri on tavallisesti resoluutioltaan puolet pienempi kuin edellisen mip-tason tekstuuri. (Unity Technologies 2021q; Valve Developer Community g.)



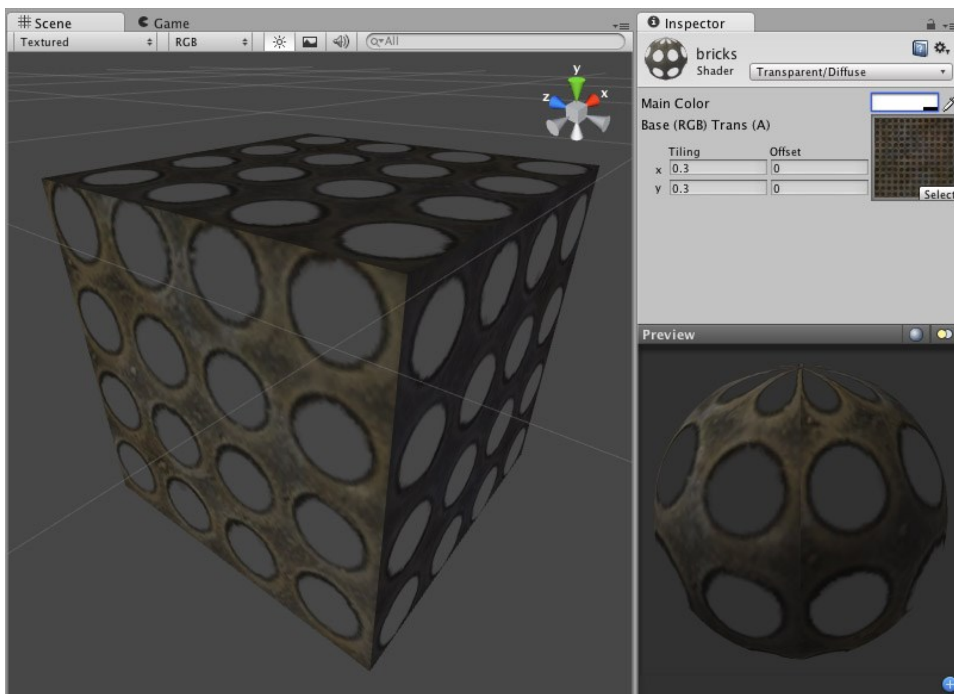

Kuva 9. Mipmap, alkuperäinen tekstuuri ja muut mip-tasot (Unity Technologies 2021q)

Mipmapeja ei kannata käyttää jokaisen tekstuurin kanssa, koska mipmapit lisäävät tekstuurin kokoa levyllä ja välimuistissa noin 33 prosenttia. Jos mipmapista ei ole tarpeeksi hyötyä suorituskyvyn kannalta, sen käyttäminen on haitallista. Jos tekstuuria käytetään esimerkiksi ainoastaan UI:ssa eli käyttöliittymässä, se on aina yhtä kaukana kamerasta, jolloin pelin aikana käytettäisiin ainoastaan yhtä mip-tasoa. Mipmapista ei olisi hyötyä tälläisen tekstuurin kanssa. Yleensä mipmapeja käytetäänkin 3D-objektien tekstuurien kanssa, sillä 3D-objektit voivat olla lähellä ja kaukana kamerasta. Myöskään pelaajahahmo ei ole usein kaukana kamerasta, joten pelaajahahmon 3D-mallin tekstuureista näytettäisiin useimmiten vain yhtä mip-tasoa. Tällaisessa tapauksessa mipmapin käyttäminen ei olisi järkevää ja aiheuttaisi enemmän haittaa kuin hyötyä pelin kannalta.

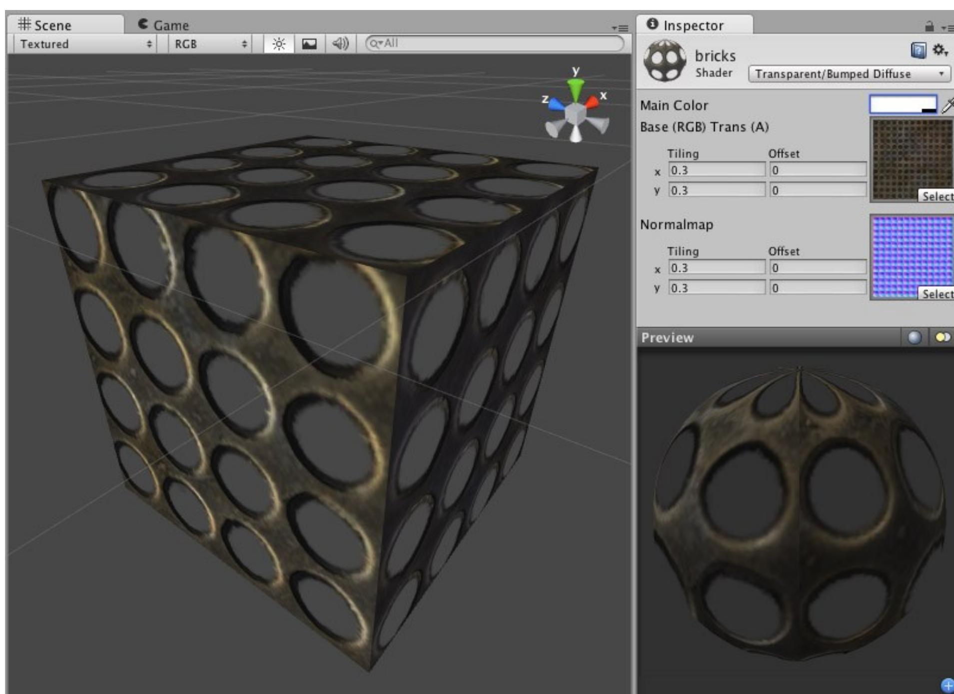
## 6.4 Läpinäkyvät tekstuurit

Joissakin esineissä on reikiä, joiden mallintaminen olisi hankalaa. Lisäksi 3D-mallissa olisi lopulta suuri määrä pintoja, jotka aiheuttaisivat paljon suorituskykyongelmia, sillä reikien mallintamiseen tarvitaan paljon pintoja. Esimerkiksi verkkoaidan mallintamiseen tarvittaisiin suuri määrä pintoja. Pintojen määrän vähentämiseksi voidaan käyttää läpinäkyvää tekstuuria, jolloin reiät 3D-malliin saataisiin tehtyä käyttäen vain yhtä pintaa. Rei’istä täytyy vain tehdä 2D-tekstuuri, jossa reiät ovat täysin läpinäkyviä. Tätä menetelmää käyttäessä pelaaja voi kuitenkin kiinnittää huomion siihen, että reiät näyttävät selvästi kaksiulotteisilta, eli niissä ei ole paksuutta, kuten esimerkiksi 3D-mallissa kuvassa 10. Käyttämällä normal mapia, reikiin voidaan saada vaikutelma paksuudesta tai kolmiulotteisuudesta. Kuvassa 11 esimerkki tällaisesta 3D-mallista. Kuvan 3D-mallissa käytetään tosin vanhentunutta shaderiä, jonka tilalla käytettäisiin nykyään Standard Shaderiä. Lopputulos näyttäisi kuitenkin lähes

samalta. Osittain läpinäkyviä tekstuureja käytettäessä kannattaa kuitenkin olla varovainen, sillä useampi osittain läpinäkyvä tekstuuri voi aiheuttaa ongelman, jota kutsutaan nimellä overdraw eli ylipiirtäminen. Se tarkoittaa saman pikselin renderöitymistä useaan kertaan. Mitä useammin tätä tapahtuu, sitä raskaampaa renderöimisestä tulee. (Arm Limited a; Epic Games d; Unity Technologies 2021a; Unity Technologies 2021r; Arm Limited c.)



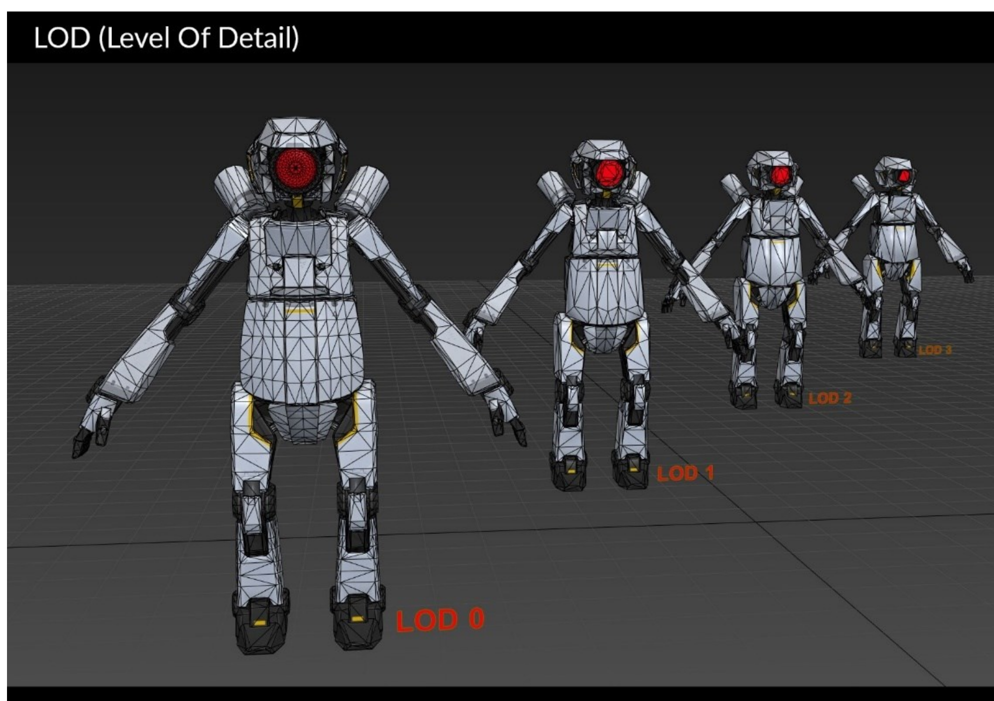

Kuva 10. Reikäinen 3D-malli, jossa on käytetty läpinäkyvää tekstuuria (Unity Technologies 2021v)



Kuva 11. Reikäinen 3D-malli, jossa on käytetty läpinäkyvää tekstuuria ja normal mapia (Unity Technologies 2021s)

## 6.5 LOD

LOD eli Level of Detail on useissa pelimoottoreissa käytettävä tunnettu optimointimenetelmä, jossa 3D-mallista tehdään yksi yksityiskohtainen versio ja muutama muu versio, joissa on vähemmän yksityiskohtia. Yksityiskohtaisin versio objektista näytetään pelaajalle, kun pelaaja on lähellä objektia. Pelaajan siirtyessä kauemmas objektista, se korvautuu huomaamatta versiolla, jossa on vähemmän yksityiskohtia. Tämä tapahtuu uudelleen vielä kauempana objektista vielä yksinkertaisemmalla versiolla ja niin edelleen, kunnes viimeinen LOD-versio on käytössä. Yksinkertaiset versiot korvautuvat myös yksityiskohtaisemmilla LOD-versioilla pelaajan lähestyessä objektia. Kuvassa 12 esimerkkinä robottia esittävän 3D-mallin neljä eri LOD-versiota, joiden on tarkoitus vaihtua tietyllä etäisyydellä kamerasta. (Arm Limited b; computer-graphics.se; Valve Developer Community b; Valve Developer Community d; Epic Games b; Unity Technologies 2021l.)



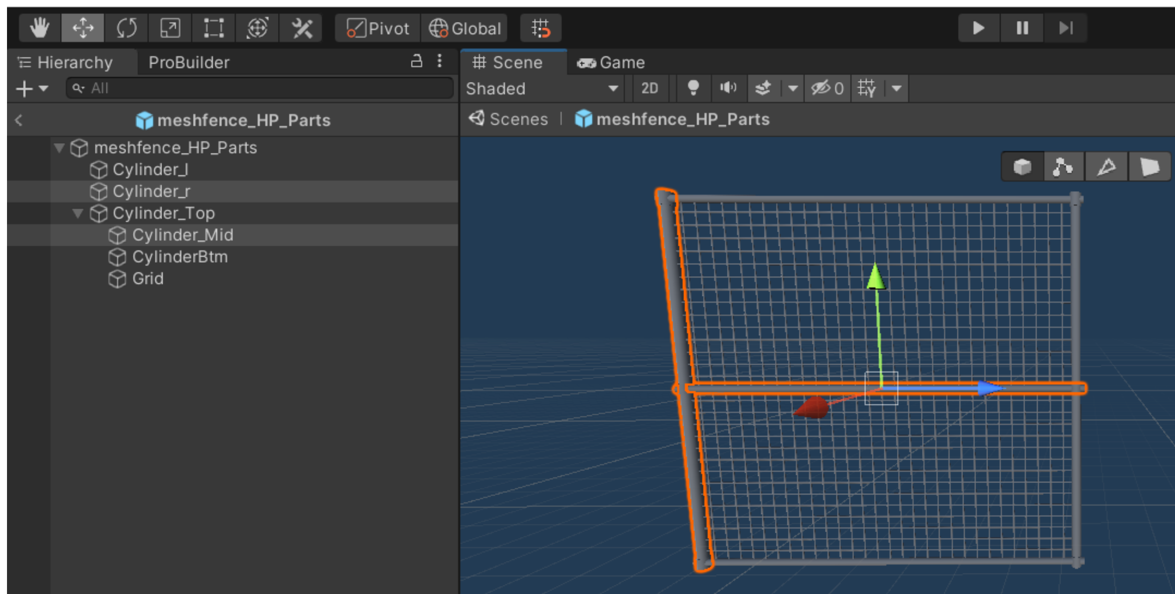

Kuva 12. Saman 3D-mallin eri LOD-versiot (Arm Limited b)

Tarkoituksena on, että muutokset LOD-versioissa ovat niin pieniä, ettei pelaaja huomaa LOD-version vaihtumista, muttei liian pieniä, jotta menetelmästä olisi hyötyä. Esimerkiksi Valve Corporationin Source Engineen keskittyneen Valve Developer Communityn mukaan LOD-versiota ei kannata tehdä, ellei yksityiskohtia eli pintojen määrää voida vähentää vähintään 30 prosenttia edellisestä versiosta. (Valve Developer Community c.)

LOD-menetelmää kannattaa käyttää silloin, kun pelaaja voi nähdä 3D-mallin läheltä ja kaukaa, varsinkin silloin, kun mallissa on paljon yksityiskohtia kuten pieniä komponentteja, joita ei voi nähdä kaukaa tai esimerkiksi pyöreitä kohtia. Jos pintojen määrä 3D-mallissa on suuri, menetelmän käyttämistä kannattaa harkita. Menetelmää käytetään erittäin paljon esimerkiksi Elder Scrolls -pelisarjan peleissä, joissa voidaan nähdä todella kauas, mutta menetelmä on niin yleinen, että sitä käytetään lähes jokaisessa videopelissä, jossa ylipäätään käytetään 3D-malleja. (Arm Limited b; Epic Games b; Valve Developer Community d; Unity Technologies 2021l.)

## 6.6 Esineen sisältämien komponenttien määrä

Esineen sisältämien komponenttien määrän vähentäminen on yksinkertainen ja Unitylle yksilöllinen optimointikeino, joka vaikuttaa CPU:lta vaadittuun suorituskykyyn. Esineen sisältämät komponentit tarkoittavat minkä tahansa Unityn scenessä olevan objektin hierarkiassa alempana olevia objekteja, joita kutsutaan myös objektin lapsiksi. Toisin sanoen 3D-objektin hierarkian koko kannattaa pitää mahdollisimman pienenä. Unity käyttää nimittäin CPU tehoa lähettääkseen erilaisia tietoja kuten valodataa ja esimerkiksi komentoja GPU:lle jokaisesta objektista erikseen. Tästä syystä objektien määrä kuormittaa CPU:ta, joten niiden yhdistäminen on useimmiten järkevää. Jos esimerkiksi kättä esittävällä pääobjektilla on viisi erillistä sormea lapsena, mutta ne eivät liiku missään vaiheessa erillään pääobjektista, sormet kannattaa liittää pääobjektiin, jolloin komponentit eli objektit vähenisivät kuudesta komponentista yhteen komponenttiin. Kuvassa 13 on esimerkki 3D-mallista, jossa sen kaikki komponentit ovat erillisiä, vaikka ne eivät oletettavasti ole liikkuvia osia. Koska tämänkaltainen 3D-malli liikkuu yhtenä kappaleena ja on luultavasti staattinen, kaikki sen hierarkiassa olevat kappaleet eli komponentit tulisi yhdistää toisiinsa 3D-grafiikkaohjelmassa. (Unity Technologies 2021a; Unity Technologies 2021u.)




Kuva 13. Verkkoaidan 3D-malli, joka koostuu turhaan useasta eri komponentista

Esineen sisältämiä komponentteja ei kuitenkaan kannata yhdistää silloin, jos ne sijaitsevat todella kaukana toisistaan, sillä eri paikassa sijaitsevat valot voivat osua niihin, joka tarkoittaa, että kaksi eri valoa valaisee samaa objektia. Unity renderöi jokaisen objektin, johon valo osuu, jokaiselle valolle yksilöllisesti. Jos objektiin osuu kaksi valoa, objekti renderöidään kaksi kertaa. Tämä aiheuttaa turhaa kuormitusta laitteistolle, jolloin komponenttien yhdistämisestä ei hyödytä millään lailla ja laitteisto saattaa kuormittua tästä jopa enemmän kuin ilman menetelmän käyttöä. Kannattaa ottaa lisäksi huomioon, että hierarkiassa olevat tyhjät komponentit eivät vaikuta suorituskykyyn, sillä niissä ei ole mitään renderöitävää. (Unity Technologies 2021a; Unity Technologies 2021u.)

# 7 Valaistus

## 7.1 Yleistä tietoa valaistuksen optimoinnista

Jos valaistusta ei optimoida, se voi viedä merkittävästi suoritin ja prosessoritehoa. Unityssä valaistus ei ole valmiiksi optimoitu, joten optimointi täytyy määritellä ja toteuttaa itse. Ongelma valaistuksessa optimoinnin kannalta eli eniten suoritin- ja prosessoritehoa vievä asia on dynaaminen valaistus. Dynaaminen valaistus tarkoittaa valaistusta, joka voi muuttua pelin aikana jollain tavalla. Tästä syystä pelimoottori laskee jatkuvasti eri pinnoille osuvaa valoa ja niiden aiheuttamia varjoja. Optimoinnin kannalta onkin ideaalista käyttää mahdollisimman paljon staattista valaistusta. Staattinen valaistus tarkoittaa valoa, joka on laskettu etukäteen pelimoottorissa ja tallennettu datana levylle. Pelimoottori käyttää tätä dataa luodakseen halutunlaisen valaistuksen peliin, eikä sen siten tarvitse laskea valaistusta jatkuvasti ajonaikana. Laitteiston kuormitus vähenee siis merkittävästi käytettäessä staattista valoa, mutta tällä on tietysti myös varjopuolensa. Staattinen valo ei voi muuttua ajonaikana, joten sitä ei voi kontrolloida esimerkiksi voimakkuudeltaan tai väriltään, valaistuksen esilaskemisessa voi kulua paljon aikaa, varsinkin jos pelikentän koko on suuri ja lisäksi valaistusdatan koko voi joissain tapauksissa kasvaa suureksi. (Unity Technologies 2021a; Valve Developer Community f.)

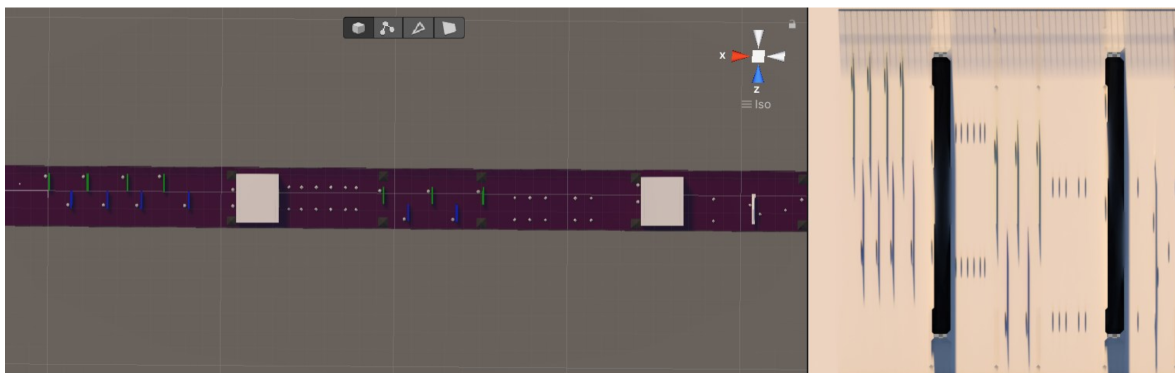
Unityssä dynaaminen valo tarkoittaa valoa, jonka Light-komponentin Mode-kohtaan on valittu vaihtoehto Realtime. Muita vaihtoehtoja ovat Mixed ja Baked. Baked tarkoittaa staattista, ajonaikana muuttumatonta valoa. Mixed tarkoittaa sitä, että valo on osittain dynaaminen ja osittain staattinen. Tällä tarkoitetaan sitä, että pelimoottori suorittaa osan valaistukseen liittyvistä työstä etukäteen ja osan ajonaikaisesti. Koska se suorittaa aina jonkin verran laskuja ajonaikaisesti, se on optimoinnin kannalta huonompi vaihtoehto kuin Baked eli staattinen valaistus. Mikään ei kuitenkaan estä käyttämästä eri vaihtoehtoja eri valoille, mutta Realtime- ja Mixed-tyyppisiä valoja ei kuitenkaan optimoinnin kannalta kannata olla pelissä liikaa, vaan staattisia valoja kannattaa käyttää aina, kun se on mahdollista. (Unity Technologies 2021a.)

Ajonaikaisessa valaistuksessa (dynaamiset valot) kannattaa ottaa huomioon myös Forward rendering path, joka on Unityn tapa renderöidä objekteja pelissä. Unityssä objektit renderöityvät yhtä monta kertaa kuin niihin vaikuttavia valonlähteitä pelissä on. Tämä tarkoittaa, että jos objektiin osuu monen dynaamisen valonlähteen valo yhtä aikaa, se renderöidään useasti, vaikka valo ei vaikuttaisi valaistukseen lähes mitenkään. Jos objektissa on esimerkiksi osia, jotka ovat etäällä toisistaan, niihin voivat vaikuttaa eri valonlähteiden valot, joka aiheuttaa sen, että koko objekti renderöityy uudelleen, vaikka valo vaikuttaa vain pieneen

osaan objektia. Siksi kannattaa useimmiten erottaa tällainen pieni komponentti suuremmasta kappaleesta erilliseksi objektiksi, jos se sijaitsee kaukana isommasta kappaleesta. (Unity Technologies 2021a.)

## 7.2 Lightmapping

Lightmapping on menetelmä, jolla lasketaan etukäteen objektin pintojen kirkkaus ja tallennetaan laskemisesta saatu data lightmap-tekstuuriin. Menetelmää voi käyttää suoran ja epäsuoran valon kanssa. Dynaamisia valoja voi käyttää lightmap-tekstuurien kanssa, mutta lightmap-tekstuuria ei voi muuttaa ajonaikana. Jotta Lightmapping-menetelmää voidaan käyttää 3D-mallin kanssa, sille täytyy luoda toinen UV map eli tekstuurikartta vain lightmap-tekstuuria varten. Se voidaan luoda joko 3D-grafiikkaohjelmistoa käyttäen tai generoida automaattisesti Unityssä. Kuvassa 14 esitellään Unityssä luotu lightmap-tekstuuri peliprojektin alustalle, plane-objektille. Vasemmalla kuvassa näkyy pelikenttä ylhäältä päin kuvattuna ja oikealla sille luotu lightmap-tekstuuri. Plane-objekti eli pelikentän alusta on violetin värinen. (Unity Technologies 2021c.)



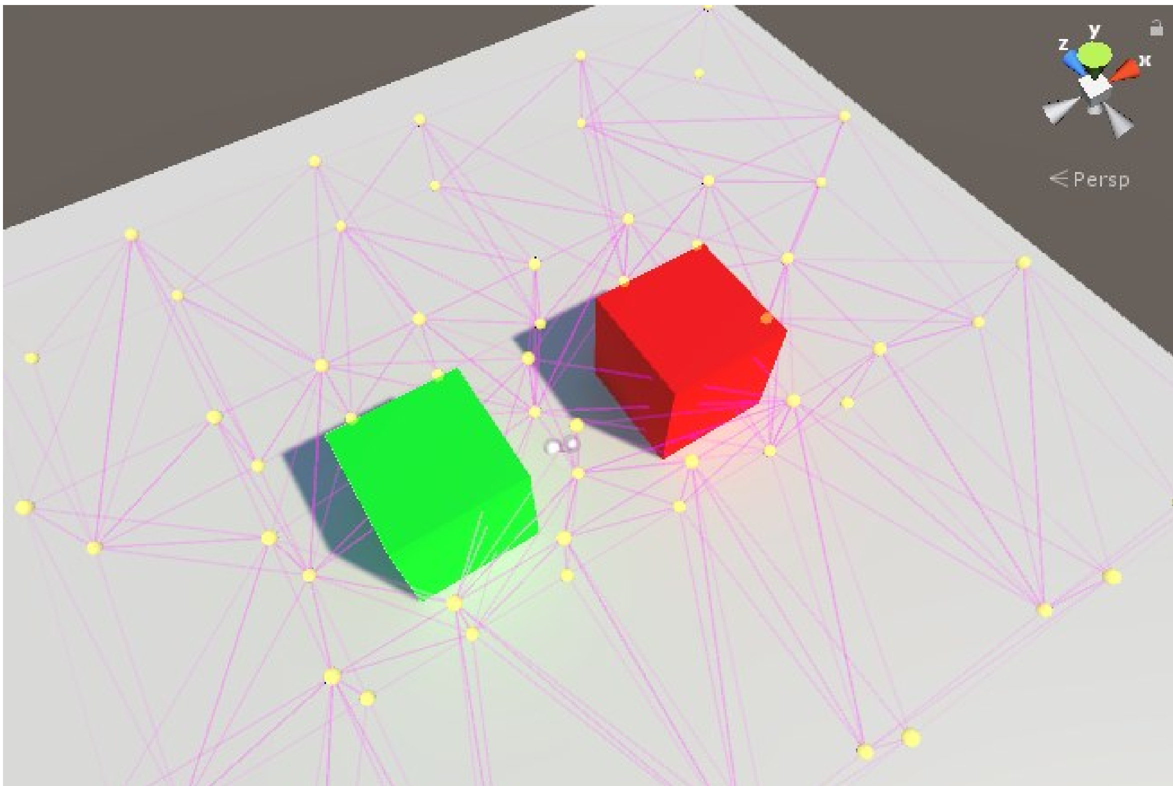
Kuva 14. Vasemmalla peliprojekti ylhäältä kuvattuna, oikealla pelikentän alustalle generoitu lightmap

Lightmap voidaan luoda vain staattisille eli muuttumattomille objekteille. Unityssä objektit, joille lightmap generoidaan, täytyy siis lisäksi merkitä staattiseksi, jotta Unity tietää mille objekteille lightmap luodaan. Myös kaikki valot, joita käytetään lightmapin generoimisessa täytyy merkitä staattiseksi. (Unity Technologies 2021c.)

## 7.3 Light Probes

Light Probes -menetelmän avulla tiedostoon voidaan tallentaa ajonaikana käytettävää informaatiota tyhjän tilan halki kulkevasta valosta. Light Probet ovat antureita (probe) eli kohtia, joita käytetään valaistuksen laskemiseen valaistusinformaation generoimisen aikana. Menetelmän avulla lasketaan siis, minkälainen valaistus esineillä on niiden ollessa tietyillä

alueilla, joten sitä käytetään pääasiassa valaistuksen laskemiseen dynaamisille objekteille, mutta valon heijastuminen alueelle lasketaan ainoastaan staattisiksi merkityistä objekteista. Menetelmää täytyy käyttää valaistuksen laskemiseen dynaamisille objekteille, sillä Lightmapping-menetelmä ei toimi dynaamisten objektien kanssa, mutta sen avulla voidaan myös parantaa valaistusta, sillä sen avulla pelimoottori voi laskea pinnoista toisille heijastuvaa valoa (light bounce), jota ei voida saavuttaa edes dynaamista (Realtime) valoa käytettäessä. Kuvassa 15 esimerkki light probeista asetettuna kahden objektin ympärille yksinkertaisessa Unityn scenessä. (Unity Technologies 2021d; Unity Technologies 2021e.)

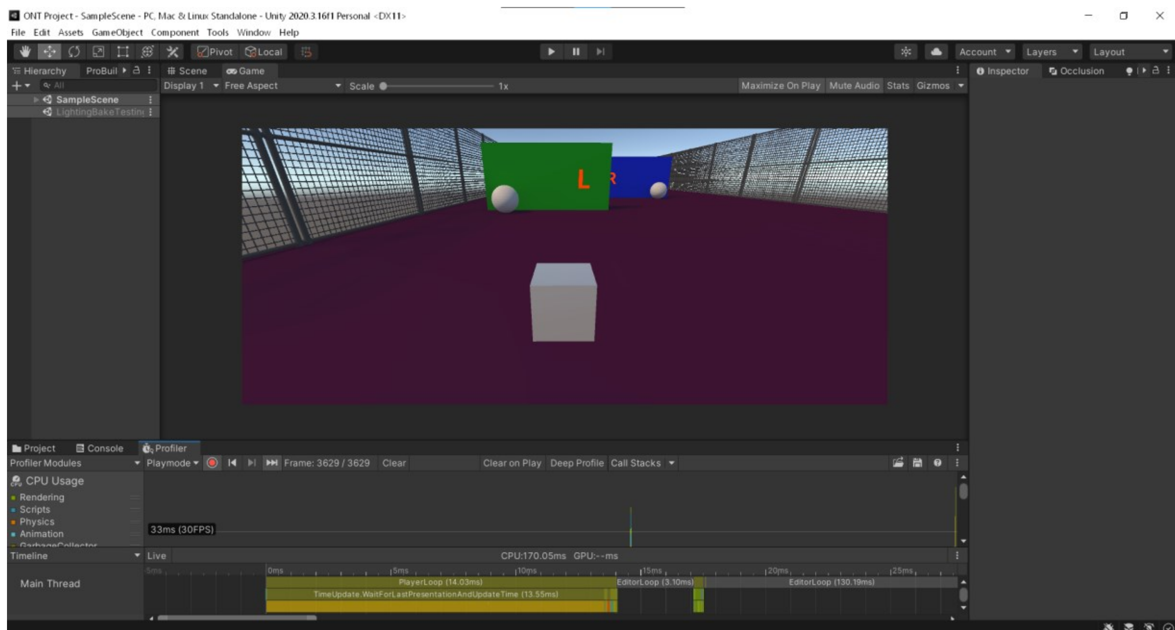



Kuva 15. Light probet asetettuna kahden kuution ympärille Unity Editorissa (Unity Technologies 2021d)

# 8 Case: Optimointimenetelmien käyttäminen Unityssä

## 8.1 Projektin taustatiedot

Optimointimenetelmien toiminnan esittelyä varten luotiin Unityä käyttäen yksinkertainen 3D-peliprojekti, jossa hyödynnetään aiemmin työssä käsiteltyjä visuaalisia optimointimenetelmiä. Kuvassa 16 nähdään kyseisen pelin aloitusruutu, peli ei ole käynnissä. 3D-malleihin liittyvien menetelmien toteuttamisessa hyödynnettiin Blender-mallinnusohjelmaa ja tekstuureihin liittyvien menetelmien toteuttamisessa GIMP-kuvankäsittelyohjelmaa.



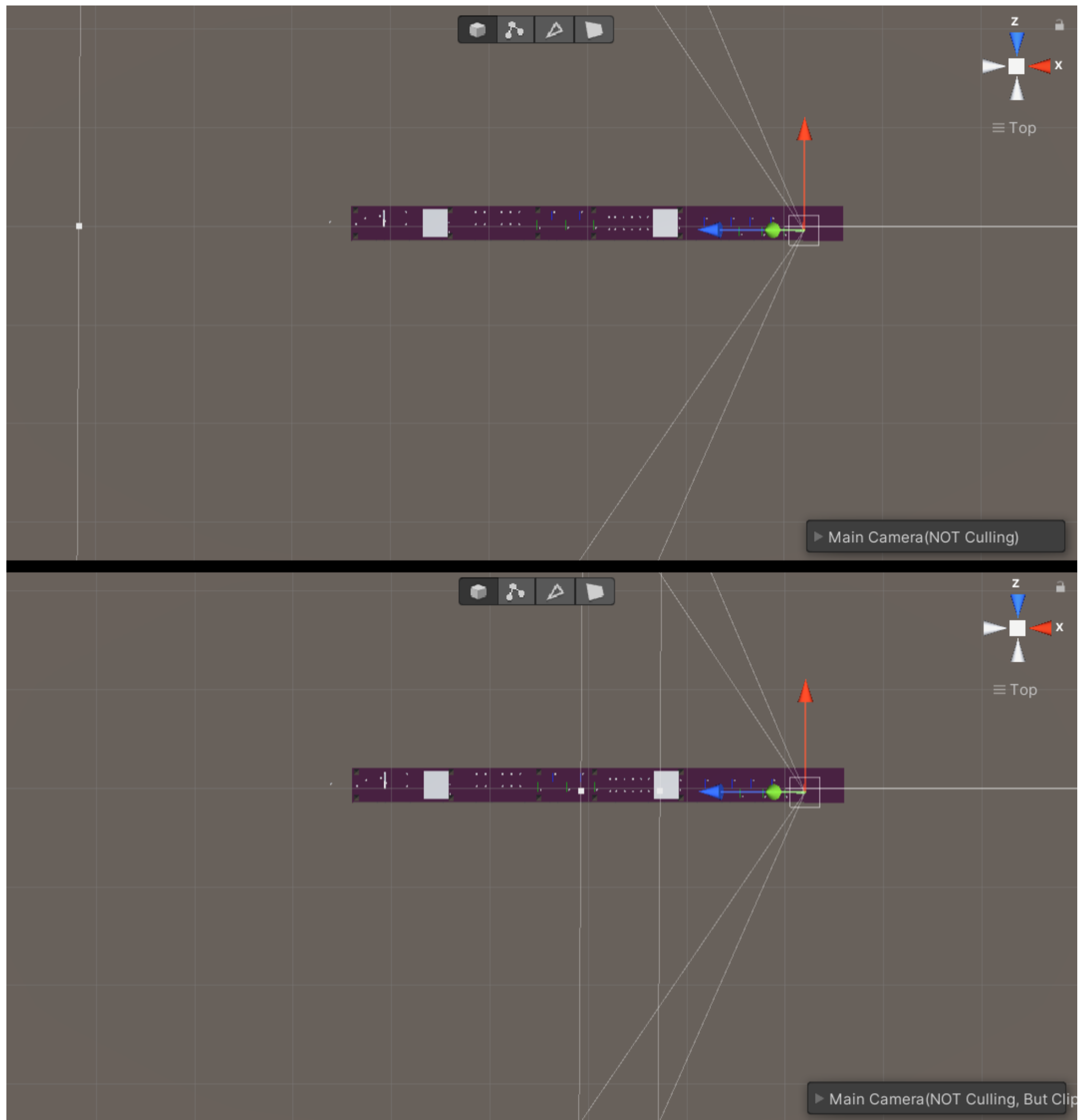

Kuva 16. Pelin aloitusruutu

Pelin idea on painaa näppäimistöltä seiniin ilmestyvää kirjainta ennen kuin pelaaja eli valkoinen kuutio törmää seinään. Jos näppäintä painetaan ajoissa, pelaaja ehtii väistää seinän ja peli jatkuu. Pelaaja voittaa pelin, kun hän saavuttaa maalialueen törmäämättä esteisiin ja vastaavasti häviää, jos hän törmää esteisiin. Pelaaja-objekti liikkuu jatkuvasti eteenpäin, mutta liikkuu myös oikealle tai vasemmalle pelaajan väistäessä estettä.

Projektin tarkoitus on esitellä optimointimenetelmien toimintaa, joten siinä on pyritty siihen, että eri optimointimenetelmien käyttäminen huomataan helpommin. Tästä syystä optimointimenetelmien olemassaoloa ei ole aina yritetty piilottaa yhtä selvästi kuin peleissä on tavallisesti tapana.

## 8.2 Clipping Planes -menetelmän käyttö

Clipping Planes -optimointimenetelmän käyttäminen on yksinkertaista. Sen toimintaa varten pelin kamerasta täytyy muuttaa vain kahta parametriä. Near-parametrin eli Near Clipping Planen arvo on säädetty peliprojektissa isommaksi kuin oletusarvo 0.3, sillä muussa tapauksessa pelaaja ei näkisi mitään kameran ollessa seinän ja pelaajan välissä, jos hän väistäisi seinän liian aikaisin. Arvoksi on asetettu 2, jotta pelaaja näkee seinän läpi, vaikka kameran edessä, lähietäisyydellä, olisikin seinä. Far-parametrin eli Far Clipping Planen arvoksi on asetettu pienempi arvo kuin oletusarvo, sillä pelaaja ei voi nähdä tätä pidemmälle kyseisessä pelissä. Oletusarvo 1000 on vähennetty arvoon 200. Kuvasta 17 nähdään, kuinka paljon tämä muuttaa kameran näkökentän kokoa. Kuvan ylemmässä scene-näkymässä nähdään, kuinka iso kameran näkökenttä on oletuksena ja kuvan alemmassa scene-näkymässä nähdään, kuinka iso se on optimoinnin jälkeen. Far Clipping Planen toimintaa ei huomaa pelin aikana, mutta pelaajan näkökentän ollessa paljon pienempi Unity käyttää esimerkiksi vähemmän prosessointitehoa Occlusion Culling -menetelmää käytettäessä, parantaen näin suorituskykyä.




Kuva 17. Kameran näkökentän koko ennen ja jälkeen optimointia

## 8.3 Occlusion Culling -menetelmien käyttäminen

Occlusion Culling -menetelmien käyttöönotto täytyy suunnitella hyvin, sillä muuten niistä saattaa aiheutua jopa haittaa suorituskyvylle. Profilerin käyttäminen on erityisen tärkeää Occlusion Culling -menetelmiä käytettäessä, jotta saadaan selville, onko menetelmien käyttämisestä optimoinnin kannalta enemmän haittaa vai hyötyä. Menetelmien käyttö täytyy suunnitella pelikentän mukaan ja koko pelialue tulisi testata, jotta saadaan selville pahimmat alueet, jotka aiheuttavat suurimmat piikit Profilerin luomassa kaaviossa. Siten saadaan helpoiten selville, milloin menetelmät kannattaa ottaa käyttöön ja kuinka tarkkoja niiden tulisi olla. PVS-menetelmä erityisesti saattaa aiheuttaa enemmän haittaa kuin hyötyä

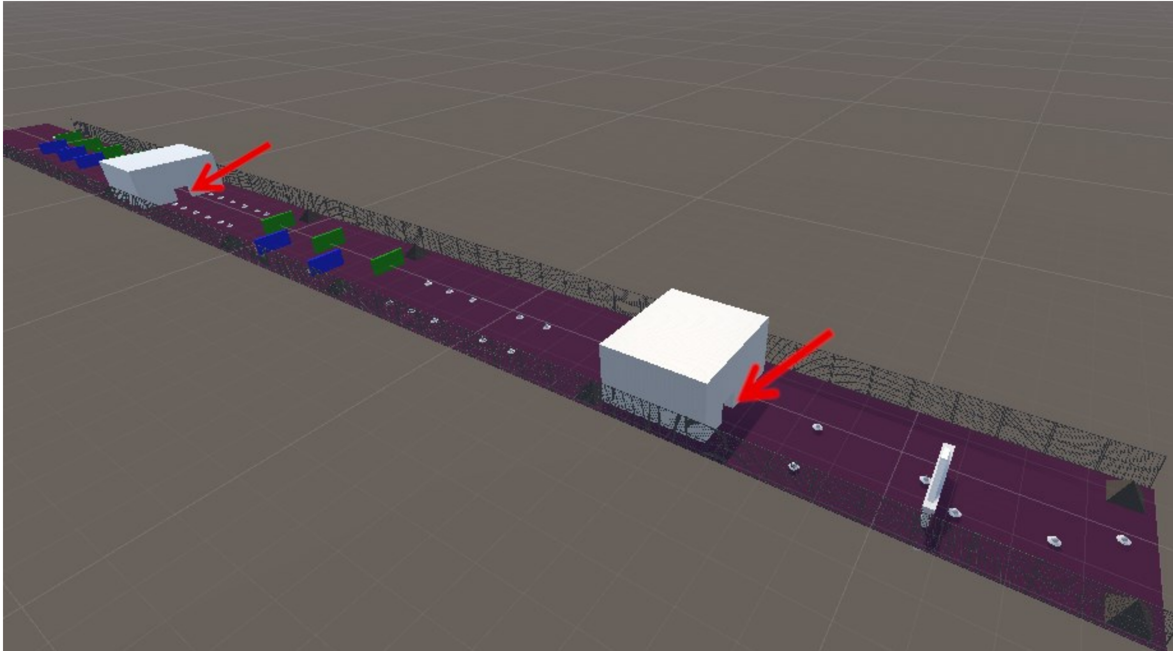
optimoinnin kannalta menetelmälle olennaisten ajonaikana suoritettavien laskutoimitusten takia.

### 8.3.1 Portal rendering -menetelmän toteutus

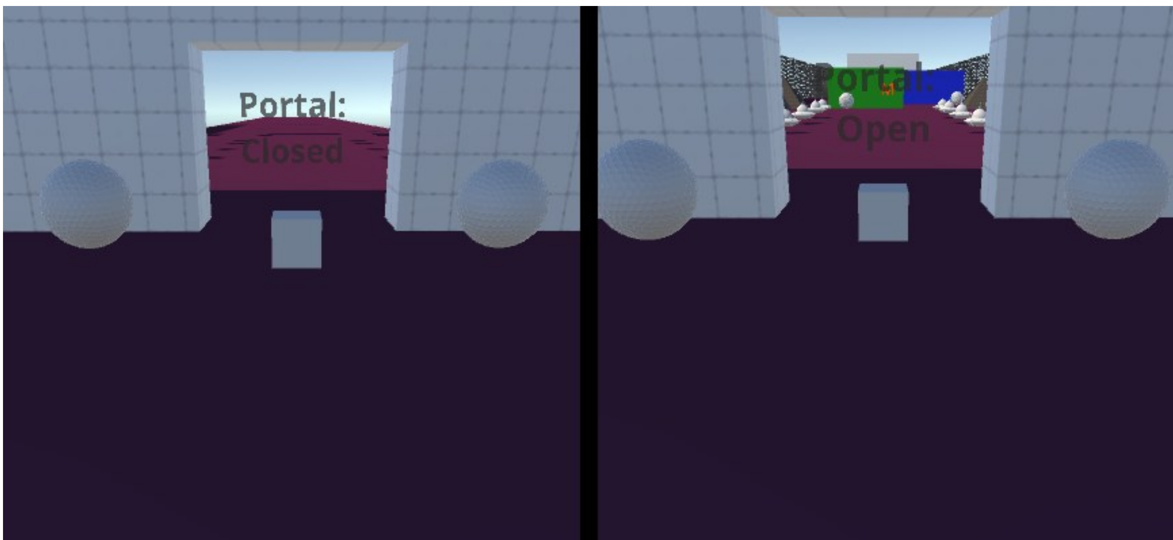
Portal rendering -optimointimenetelmässä käytettävän portaalin saa toimimaan Unityssä huomaamattomasti lisäämällä sopivaan GameObjectiin, useimmiten oveen, Occlusion Portal -komponentin. Objekti ei voi toimiakseen olla occluder eikä occludee. Portaalin avautuminen ja sulkeutuminen täytyy ohjelmoida esimerkiksi C#-ohjelmointikieltä käyttäen. (Unity Technologies 2021g.)

Ensin koodissa määritetään Occlusion Portal, jolle voidaan antaa nimeksi esimerkiksi ”myOcclusionPortal” ja portaalin kanssa toimivan oven avautuessa kirjoitetaan rivi ”myOcclusionPortal.open = true;”. Oven mennessä kiinni, Occlusion Portal täytyy myös sulkea, joten ohjelmakoodin tulee olla samassa paikassa kuin oven sulkeutumiseen liittyvä toiminnallisuus. Koodiin kirjoitetaan tällöin rivi ”myOcclusionPortal.open = false;”. Nyt oven avautuessa portaali avautuu ja oven mennessä kiinni portaali sulkeutuu. Portaaleja voi lisätä yhteen sceneen enemmän kuin vain yhden, mutta niiden käyttäminen kannattaa suunnitella järkevästi. (Unity Technologies 2021g.)

Peliprojektissa kaksi portaalia on asetettu kahden eri rakennuksen oviaukolle, jossa ei kuitenkaan ole ovea. Kuvasta 18 voidaan havainnoida, missä oviaukot ovat. Kaksi punaista nuolta osoittaa kahden eri rakennuksen oviaukolle, johon portaalit on sijoitettu. Oville on asetettu myös teksti, jossa lukee portaalin tila, jotta pelaaja huomaa helposti, milloin portaali on auki ja milloin kiinni. Kuvassa 19 näkyy kyseisen tekstin ja samalla myös portaalien toiminta. Tietyllä etäisyydellä oviaukosta portaali aukeaa ja kaikki objektit portaalin takana renderöityvät. Samalla teksti kertoo pelaajalle, että portaali on auennut. Tällä tavoin pelaaja pystyy tarkastelemaan optimointimenetelmän toimintaa helposti. Portaali ei kuitenkaan avautumisen jälkeen enää sulkeudu, sillä kyseisessä pelissä pelaaja ei voi kääntyä tai katsoa taaksepäin — mitään ei siis renderöidy pelaajan takana muutenkaan frustum culling -menetelmän ansiosta — eikä portaalin edessä ole edes ovea, joka estäisi pelaajaa näkemästä portaalin toimintaa.



Kuva 18. Portaalien sijainti pelikentällä osoitettuna punaisilla nuolilla

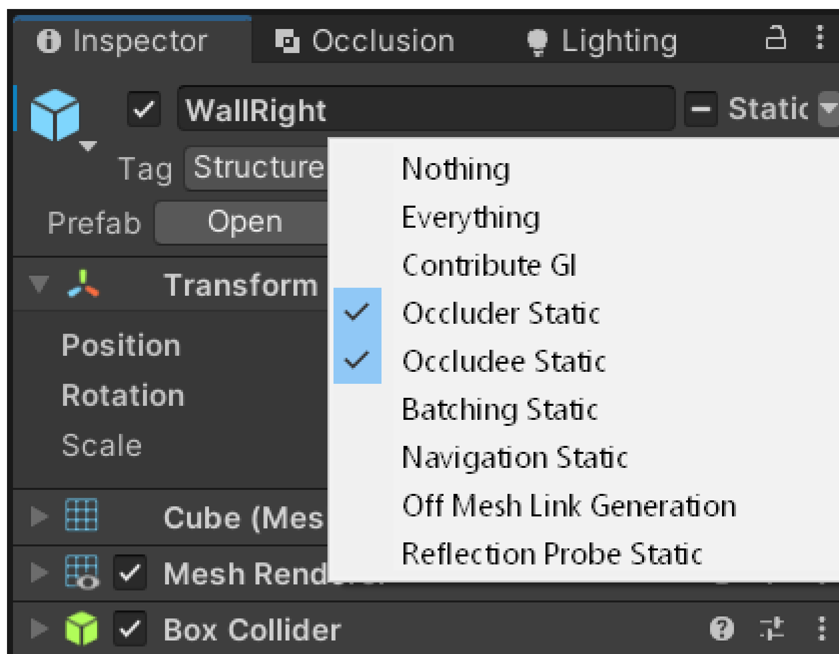



Kuva 19. Portaalien ja niiden tilan osoittavien tekstien toiminta peliprojektissa

### 8.3.2 Potentially Visible Set -menetelmän toteutus

Potentially Visible Set -muodon ottaminen käyttöön on hieman isompi tehtävä kuin Portal rendering -muodon. Ensimmäiseksi valitaan mahdolliset staattiset näköesteet, eli suuret läpinäkymättömät objektit, jotka voivat piilottaa paljon muita objekteja, kuten seinät, katto, lattia ja rakennukset. Ihanteellisesti objektien tulisi olla mahdollisimman yksinkertaisia ja niiden tulisi peittää suuri osa muita objekteja näkyvistä. Nämä objektit täytyy merkitä Static-merkinnällä Occluder Static, jotta Unity tietää, että objektit voivat toimia näköesteinä, joiden takana olevia objekteja ei renderöidä.

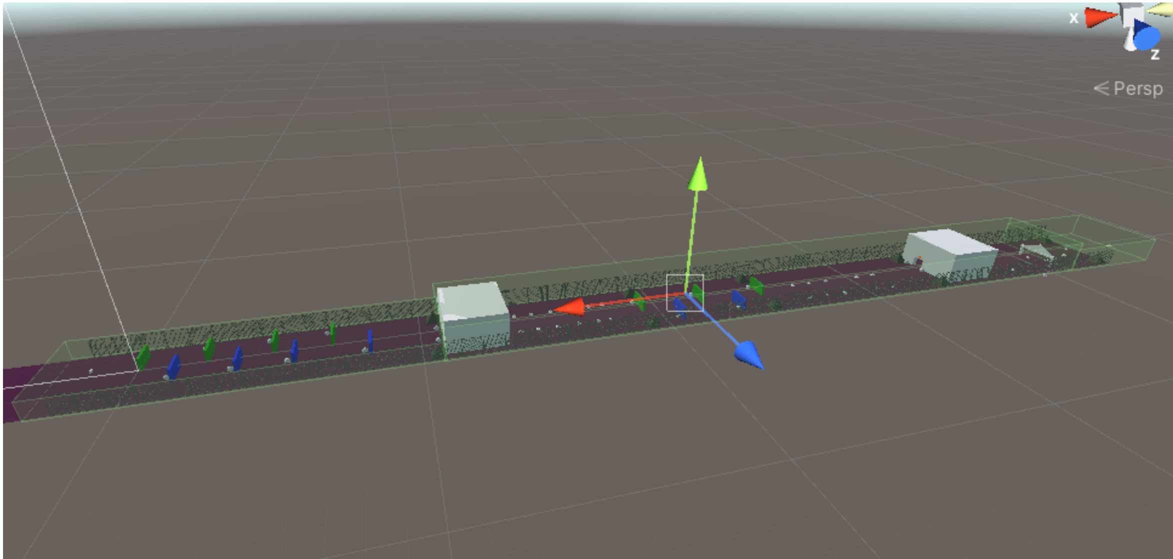
Mikä tahansa liikkumaton objekti pelissä, jolla on jonkinlainen Renderer-komponentti kannattaa merkitä Occludee Static -merkinnällä, jotta menetelmästä olisi mahdollisimman paljon hyötyä. Tämä merkintä objektissa kertoo Unitylle, että objektin voi piilottaa näköesteen takana pelaajalta. Jos objektilla ei ole tätä staattista merkintää, se renderöityy, vaikka se olisi Occluder Static -merkityn objektin takana. Objekteille voi merkitä enemmänkin staattisia merkintöjä kuin vain yhden, joten myös Occluder Static -merkityt objektit kannattaa useimmiten merkitä myös Occludee Static -merkinnällä. Esimerkiksi peliprojektin vihreät ja siniset seininä toimivat suorakulmiot on merkitty molemmilla merkinnöillä, kuten kuvassa 20 näkyy. Staattisia merkintöjä on lisäksi useita muitakin ja on mahdollista merkitä kaikki päälle kerralla, mutta ellei tiedä mitä on tekemässä, merkintöjä kannattaa tehdä vain yksi kerrallaan, sillä jokainen merkintä lisää objektin johonkin esilaskentasysteemiin (Unity Technologies 2021b; Unity Technologies 2017; Unity Technologies 2021i). Jos taas dynaamisen eli ei-staattisen kohteen halutaan toimivan piilotuksen kohteena, objektin Renderer-komponentista täytyy laittaa Dynamic Occlusion -vaihtoehto päälle. Tämän pitäisi tosin olla jo oletuksena päällä.




Kuva 20. Seinä merkittynä Occluder Static ja Occludee Static -merkinnöillä

Seuraavaksi luodaan Occlusion Area lisäämällä esimerkiksi tyhjään GameObjectiin Occlusion Area -komponentti. Is View Volume -vaihtoehdon kuuluu olla päällä ja kokoa voidaan säätää komponentista. Occlusion Area -komponentteja voi olla olemassa monta yhtä aikaa, joka on myös järkevää optimoinnin kannalta. Komponenttien muodostaman alueen sisällä pitäisi olla mahdollisimman tarkasti koko pelialue, eli kaikki paikat, jonne kameran eli pelaajan näkökentän on mahdollista päästä pelin aikana. Peliprojektissa Occlusion Area -

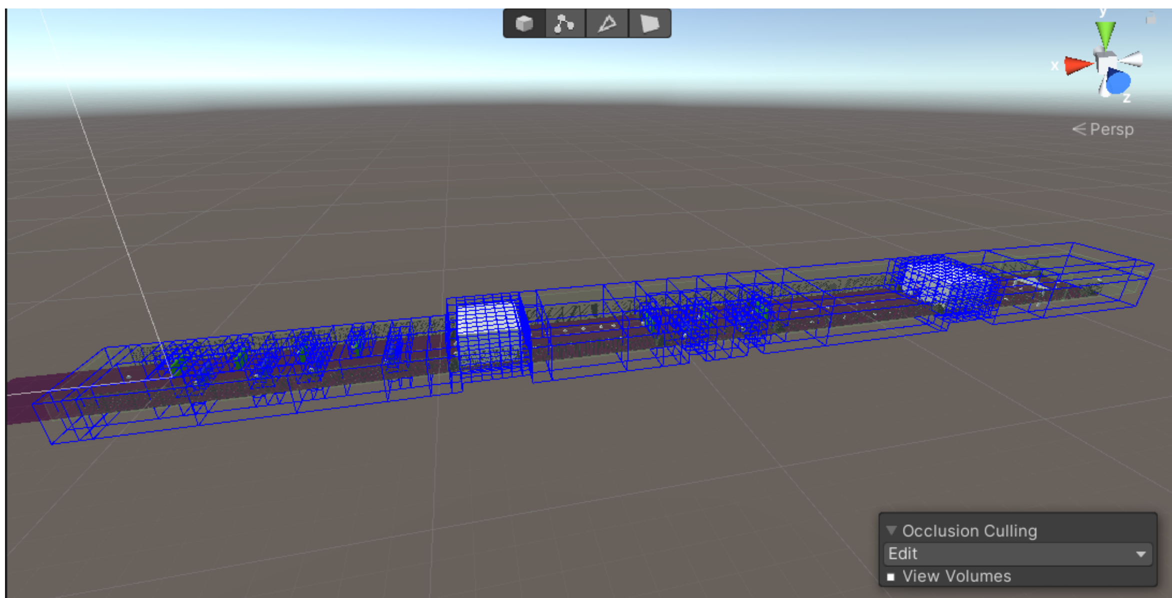
komponentteja on yhteensä kolme kappaletta. Kuvassa 21 nähdään koko pelikenttä, joka on kolmen läpinäkyvän vaaleanvihreän Occlusion Area -komponentin sisällä. Tämän jälkeen kannattaa varmistaa, että Occlusion Culling -vaihtoehto kamerassa on päällä. Tosin tämänkin pitäisi olla päällä oletuksena. (Unity Technologies 2021i.)




Kuva 21. Vaaleanvihreät Occlusion Area -komponentit peliprojektissa

Kun Occlusion Area -komponentit on lisätty sceneen ja halutut objektit merkitty oikeilla staattisilla merkinnöillä, optimointimenetelmää varten tarvittava näkyvyysdata voidaan generoida. Datan generoimiseksi täytyy avata Occlusion Culling -ikkuna päävalikosta. Unity aloittaa datan generoimisen painamalla ikkunan Bake-painiketta. Kun Unity on suorittanut laskutoimitukset ja luonut näkyvyysdatan, objektin näkyvyyttä voidaan testata Scene-välilehden näkymässä. Ikkunan oikeasta alalaidasta löytyy laatikko, jossa lukee ”Occlusion Culling”. Tekstin alla on pudotusvalikko, josta valitaan vaihtoehto Visualize. Nyt kameraa voidaan liikuttaa ruudulla, jotta saadaan testattua, kuinka tarkasti Occludee objektit katoavat, kun kameran ja objektin välissä on Occluder Static -objekti. Täytyy olla tarkkana, että kyseessä on Occlusion Culling, eikä Frustum Culling. Objektit katoavat Frustum Culling -menetelmän ansiosta aina, kun objektit ovat ulkona kameran muodostamasta kartiosta, joka on pelaajan potentiaalinen näkökenttä. Occlusion Culling -menetelmän toimimisen huomaa siitä, että esine on kameran ja näköesteen takana, mutta silti kameran muodostaman näkökentän sisällä. Jos Occludee objekti ei katoa kameran katsoessa sitä Occluder Static -objektin läpi, se johtuu luultavasti siitä, että nämä objektit ovat samalla alueella, eli Unity on jakanut alueen liian isoihin alueisiin. Tällöin Occlusion-ikkunassa täytyy pienentää asetusta Smallest Occluder. Sen jälkeen voidaan tyhjentää vanha näkyvyysdata oikeasta alakulmasta klikkaamalla Clear-painiketta ja generoida uusi näkyvyysdata klikkaamalla uudestaan Bake-painiketta. Sama toistetaan, kunnes menetelmä toimii tarpeeksi tehokkaasti.

Kannattaa ottaa huomioon, että mitä tarkemmat parametrit laskutoimituksille asetetaan, sitä kauemmin datan generoiminen kestää ja sitä suuremmaksi sen koko kasvaa. Parametrejä ei kannata asettaa liian tarkoiksi myöskään siksi, että optimointimenetelmästä olisi enemmän hyötyä, sillä Unity käyttää myös jonkin verran resursseja datan lukemiseen ajonaikaisesti. Jos data olisi jättimäinen, sen lukeminen hidastaisi peliä merkittävästi, jolloin optimointimenetelmästä ei olisi enää hyötyä (Unity Technologies 2021i). Kuvassa 22 nähdään, kuinka isoihin alueisiin Unity on jakanut pelikentän alueen peliprojektissa. Siniset viivat ovat renderöitävien alueiden ääriviivoja. Jos pelaajalla on näköyhteys tällaisen alueen sisälle pelin aikana, sen sisällä olevat objektit renderöidään, jos ei, objekteja ei renderöidä.




Kuva 22. Pelikenttä jaettuna renderöintialueisiin PVS-menetelmää varten peliprojektissa

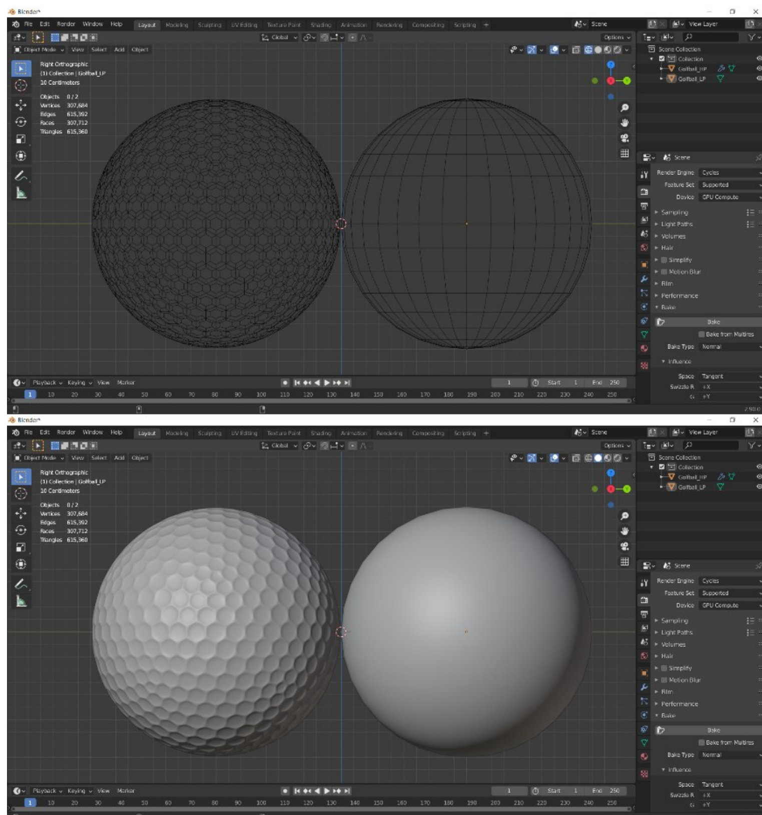
Peliprojektissa Occludee Static -merkintä on annettu kaikille näkyville objekteille paitsi lattialle, sillä pelaajalla on näköyhteys lattiaan koko pelin ajan. Kaikki seinät, lattia ja suuret, yksinkertaiset rakennukset ovat lisäksi Occluder Static -merkittyjä. Scenessä on kolme Occluder Area -komponenttia, jotka kattavat alueen, jolla pelaajan on mahdollista olla pelin aikana ja objektit, jotka pelaajan on mahdollista nähdä pelin aikana. Jotta PVS toimisi oikein, Bake-parametrit on asetettu pienemmiksi kuin niiden oletusarvo. Smallest Occluder -parametrin arvoksi on asetettu 1 ja Smallest Hole -parametrin arvoksi on asetettu 0.1. Pelaaja voi tarkastella optimointimenetelmän toimintaa pelissä törmäämällä seinään, jolloin seinä lähtee pois paikoiltaan. Silloin pelaaja huomaa, että seinän takana ei ole mitään, koska mitään mitä pelaaja ei näe seinän takaa, ei renderöidä. Tämä johtuu PVS-menetelmästä. Unity ei ota huomioon seinän olinpaikan vaihtumista ajonaikana optimointimenetelmässä, joten vaikka näköeste katoaa, mitään ei siitä huolimatta renderöidy. Tästä syystä liikkuvia objekteja ei kannata merkitä Occluder Static -merkinnällä.

## 8.4 3D-malleihin liittyvien menetelmien toteutus ja käyttöönotto

3D-mallien optimointi on lähes aina järkevää, mutta toisaalta optimointimenetelmien käyttöön ottaminen on usein aikaa vievää, joten menetelmien käyttöönoton hyödyllisyyttä kannattaa arvioida etukäteen. Jos pintojen määrää saadaan pienennettyä merkittäviä määriä ilman, että 3D-objektien ulkonäkö kärsii huomattavasti, myös käyttöön ottamisessa aikaa vievien optimointimenetelmien käyttöön ottaminen on järkevää. Kaikki toteutuksessa käytetyt 3D-mallit on tehty Blenderiä käyttäen, mutta kaikki menetelmät on kuitenkin lopuksi viimeistelty Unityssä.

### 8.4.1 Normal mapin luominen ja käyttöönotto

Normal map -menetelmän käyttöönottoa varten 3D-mallille täytyy luoda normal map -tekstuuri. Peliprojektiin normal map -tekstuuri luotiin golfpalloa esittävälle 3D-mallille Blender-mallinnusohjelman avulla. Golfpallo valittiin siksi, että sen pinnalla on paljon toistuvia yksityiskohtia, joiden mallintamiseen tarvittaisiin suuri määrä pintoja. Normal map -tekstuurin luomiseen käytettiin metodia, jossa yksityiskohtaista eli high-poly-mallia ja yksinkertaista eli low-poly-mallia vertaillaan toisiinsa. Metodin avulla yksityiskohtainen ja vähän polygoneja eli pintoja sisältävä golfpallon 3D-malli saadaan näyttämään lähes täysin toistensa kopioilta. Kuvasta 23 voidaan havainnoida yksityiskohtaisen ja yksinkertaisen 3D-mallin eroavaisuudet, kuten pintojen määrä.



Kuva 23. Yksityiskohtainen ja yksinkertainen 3D-malli golfpallosta Blenderissä

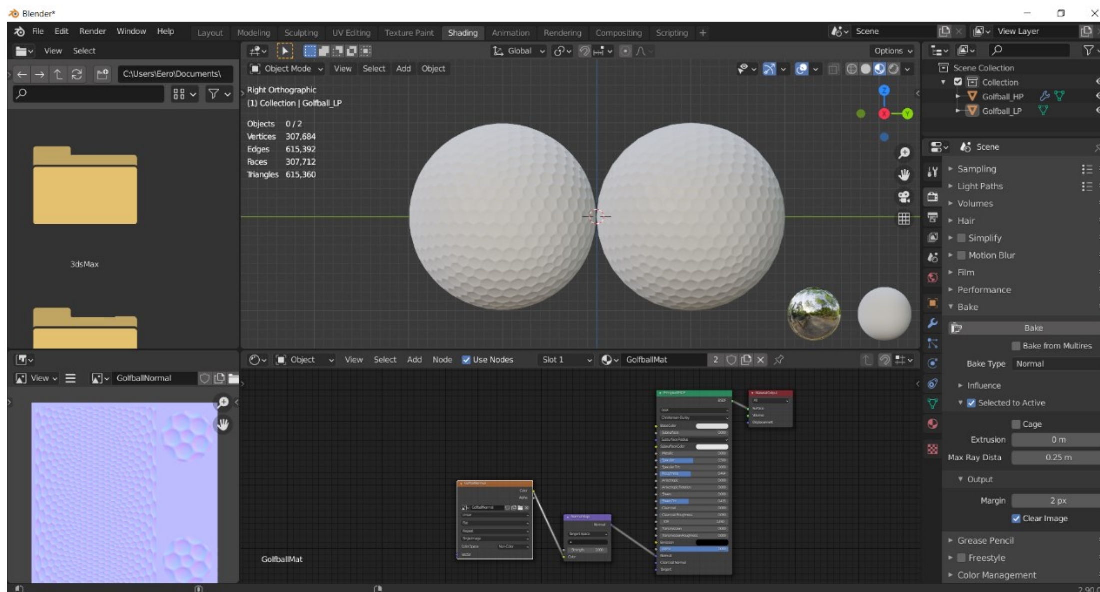
Yksinkertaisemmalle mallille täytyy suorittaa teksturointia varten unwrap-operaatio, jonka avulla voidaan luoda mallille UV map eli tekstuurikartta levittämällä mallin kolmiulotteiset pinnat niin, että ne voidaan esittää kaksiulotteisesti. Tämä voidaan tehdä Blenderillä helpoiten käyttämällä Smart UV Project -komentoa, joka luo objektille automaattisesti yksinkertaisen tekstuurikartan.

Kun UV map on luotu objektille, Node Editor -näkymässä luodaan uusi tekstuuri, johon normal map voidaan generoida. Tekstuurille annetaan nimi, koko, väri ja valitaan myös muutamia muita asetuksia. Koko kannattaa antaa kahden potensseina. Normal mapia luodessa alpha channeliä ei kannata luoda, mutta kannattaa valita vaihtoehto 32 bit Float normal mapin laadun parantamiseksi. Tekstuuri asetetaan Image Texture -nodeen, jonka asetuksista vaihdetaan vielä Color Space -asetus, jonka tulee olla normal mapin tapauksessa Non-color, sillä normal map ei sisällä väridataa – vaikka se ei olekaan väriltään mustavalkoinen – vaan se sisältää väridatan sijaan informaatiota normaalivektorien suunnasta, joka on tallennettu kuvan RGB-arvoihin (Blender Foundation 2020b; Blender Foundation 2017; Unity Technologies 2021j). Jos tätä asetusta ei muuteta, esikatselussa Blenderin sisällä 3D-mallissa saattaa näkyä saumoja.

Seuraavaksi valitaan molemmat 3D-mallit, mutta ensin yksityiskohtainen malli ja sitten yksinkertainen malli. Node Editor -näkymässä Image Texture -node täytyy olla valittuna, jotta Blender tietää luoda normal mapin oikeaan tekstuuriin.

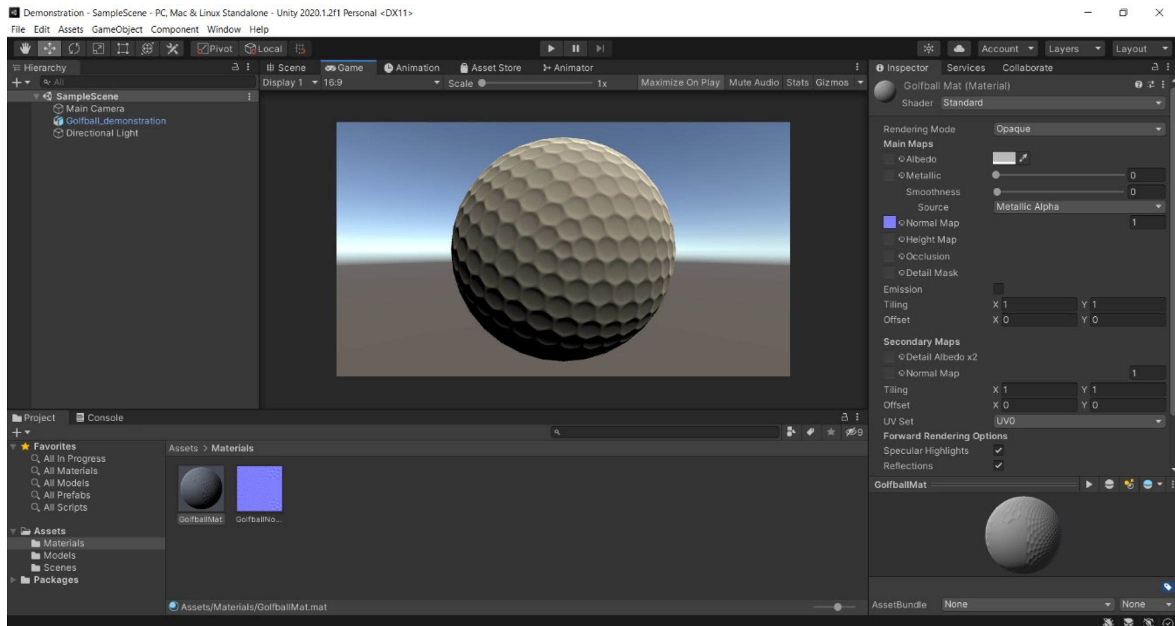
Lopuksi oikealla sijaitsevan työkalupalkin välilehdestä Render Properties tarkistetaan ensimmäisenä, että valittu Render Engine on Cycles. Sen jälkeen siirrytään välilehden kohtaan Bake, jossa Bake Type -kohtaan valitaan vaihtoehto Normal. Myös Selected to Active täytyy valita, sillä tämä tarkoittaa, että normal map luodaan ensin valitun kohteen perusteella toisen valitun kohteen UV mapiin. Asetusten ollessa valmiina normal mapin generoiminen voidaan aloittaa painamalla Bake-painiketta. Jos normal map ei onnistu täydellisesti, vaan siihen tulee pieniä virheitä, voidaan nostaa hieman Selected to Active -kohdan alta Max Ray Distance -parametrin arvoa virheiden eliminoimiseksi. Onnistuminen tarkistetaan lisäämällä Node Editoriin Normal map -node, johon Image Texture -node yhdistetään. Normal map -node yhdistetään yksinkertaisen golfpallon materiaaliin. 3D-näkymässä Viewport Shading, joka sijaitsee näkymän oikeassa yläkulmassa, täytyy olla päällä ja yksityiskohtainen esine piilottaa näkymästä, jotta nähdään lopputulos. Kun lopputulokseen ollaan tyytyväisiä, normal map -tekstuuri täytyy vielä tallentaa. Kun normal map on valmis, ja se on lisätty yksinkertaiseen 3D-malliin, 3D-mallien pitäisi näyttää lähes täsmälleen samalta,

vaikka toisen mallin pintojen määrä on huomattavasti pienempi kuin toisen, kuten kuvassa 24 näkyy.




Kuva 24. Vasemmalla yksityiskohtainen 3D-malli, oikealla yksinkertainen 3D-malli normal mapin kanssa

Viimeinen vaihe on normal mapin asettaminen paikoilleen Unityssä. 3D-malli täytyy ensiksi viedä Unityyn. Mallin voi joko muuttaa tiedostoksi, jota Unity pystyy käsitellä, esimerkiksi fbx-tiedostoksi, tai viedä blend-tiedosto suoraan pelimoottoriin, sillä Unity osaa käsitellä myös blend-tiedostoja. Peliprojektissa päädyttiin käyttämään fbx-tiedostoa. Käyttöönottoa varten fbx-tiedosto täytyy siirtää Unityyn raahaamalla tiedosto Unityn projekti-ikkunaan. Projektiin täytyy raahata tietysti myös luotu normal map -tekstuuri, jotta sitä voidaan käyttää objektin materiaalissa. 3D-mallin materiaali valitaan projekti-ikkunassa, jolloin Inspector-välilehdessä Main Maps -otsikon alta löydetään kaikki tekstuurikartat, joista yksi on Normal Map. Tähän raahataan Blenderissä luotu normal map. Jotta normal map toimisi oikein, se valitaan lopuksi vielä projekti-ikkunassa ja merkitään normal mapiksi valitsemalla Texture Type -kohdasta vaihtoehto Normal map. Jos tätä ei kuitenkaan tee, Unity ehdottaa tekstuurityypin vaihtamista automaattisesti. Kuvassa 25 nähdään lopputulos, eli yksinkertainen 3D-malli normal mapin kanssa Unityssä.




Kuva 25. Yksinkertainen 3D-malli, johon lisätty normal map Unity-pelimoottorissa

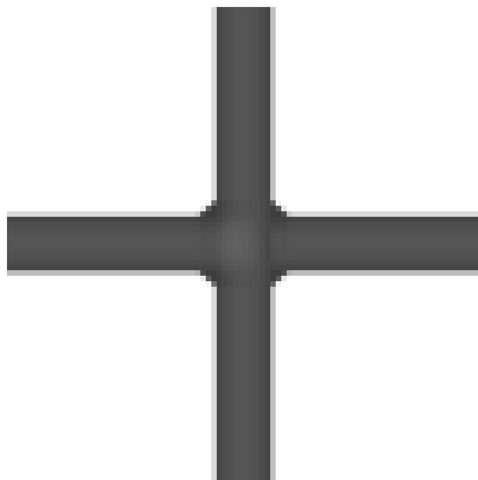
Suuria golfpallon 3D-malleja on sijoitettu peliprojektissa ympäri pelikenttää, jotta pelaaja voisi tarkastella Normal map -menetelmää käyttäviä 3D-malleja pelin aikana. Normal mapin käyttöönotossa ei tarvinnut tehdä Unityn kannalta muita toimenpiteitä kuin edellisessä kappaleessa kerrottu normal mapin lisääminen golfpallon materiaaliin sekä golfpallon 3D-mallin raahaaminen Unityn sceneen.

### 8.4.2 Läpinäkyvien tekstuurien käyttäminen 3D-mallissa

Läpinäkyvän tekstuurin käyttöön ottamiseksi se täytyy ensin luoda sopivalle 3D-mallille. Peliprojektissa läpinäkyvä tekstuuri luotiin verkkoaidan 3D-mallin verkkoa varten, sillä läpinäkyvän tekstuurin käyttäminen on ihanteellista tällaisessa tilanteessa, jossa 3D-mallissa on paljon reikiä, joiden mallintamiseen tarvittaisiin suuri määrä pintoja. Tekstuurin luominen vie kuitenkin enemmän aikaa kuin mallintaminen, ja se on lisäksi mallintamista monimutkaisempaa. Tekstuurin lopullisessa luomisessa tarvitaan myös 2D-grafiikkaohjelma GIMPiä, sillä Blenderin versiolla 2.91 tekstuuriin ei voi luoda läpinäkyvyyttä eli alpha channelia suoraan. Jotta GIMPin avulla saadaan viimeisteltyä haluttu läpinäkyvä tekstuuri, Blenderissä täytyy luoda kaksi eri tekstuuria. Toiseen tekstuuriin tulee lopullisen tekstuurin väri-informaatio ja toiseen, mustavalkoiseen tekstuuriin tulee lopullisen tekstuurin läpinäkyvyysinformaatio, jossa valkoinen väri tarkoittaa näkyvää ja musta läpinäkyvää.

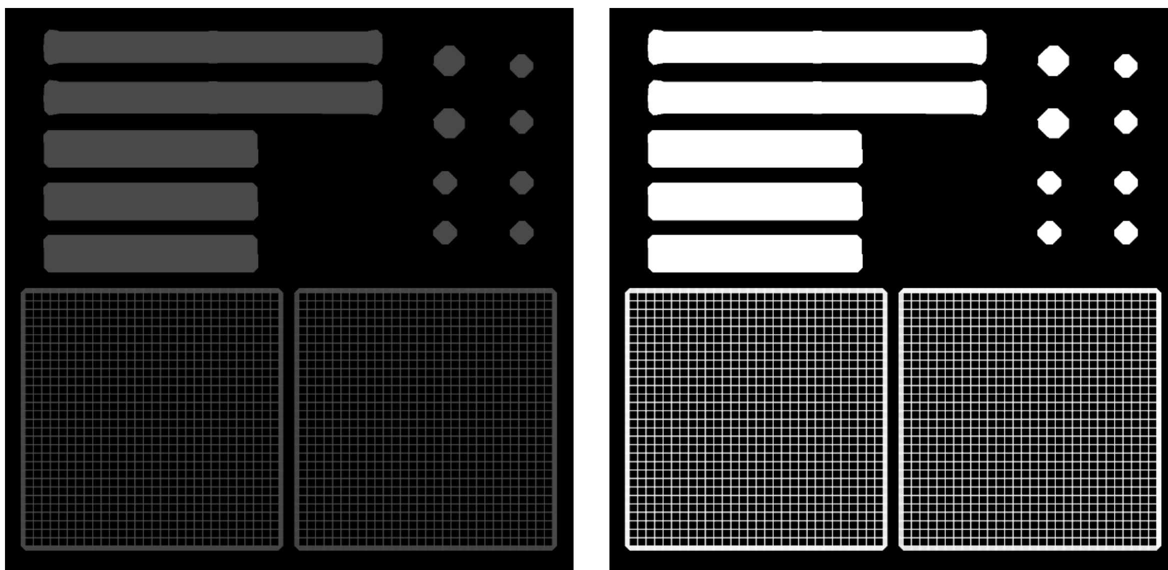
Aluksi 3D-mallille täytyy tehdä unwrap-operaatio, jolla levitetään objektin tekstuurikartta kaksiulotteisesti neliön muotoiselle tekstuurialueelle. Tätä kutsutaan myös UV mapiksi. Sen luomisessa tulee ottaa huomioon, ettei mikään alue ole toistensa päällä. Varsinkaan ne pinnat, joihin tulee läpinäkyvyyttä, eivät saa olla toisten alueiden päällä. Kun UV map on

luotu, voidaan aloittaa ensimmäisen tekstuurin luominen. Tähän voidaan käyttää mitä tahansa ilmaiseksi internetistä löydettyä oikean muotoista kuvaa. Verkkoaidan tekstuurien luomiseen käytettiin tässä tapauksessa kuvassa 26 esitettyä kuvaa, jota toistamalla tekstuuriin saatiin aikaan verkon muoto. Blenderin Node Editorissa kuva lisätään Image Texture -nodeen, jonka vaihtoehto Repeat on oletuksena valittuna. Siihen yhdistetään lisäksi Mapping-node, jonka avulla tekstuuria saadaan skaalattua, jotta pienet verkon reiät saadaan muokattua oikean kokoisiksi. Tässä tapauksessa skaalaus on luokkaa 15x15. Image Texture -noden Color-kohta yhdistetään Principled BSDF -shader noden Base Color -kohtaan.



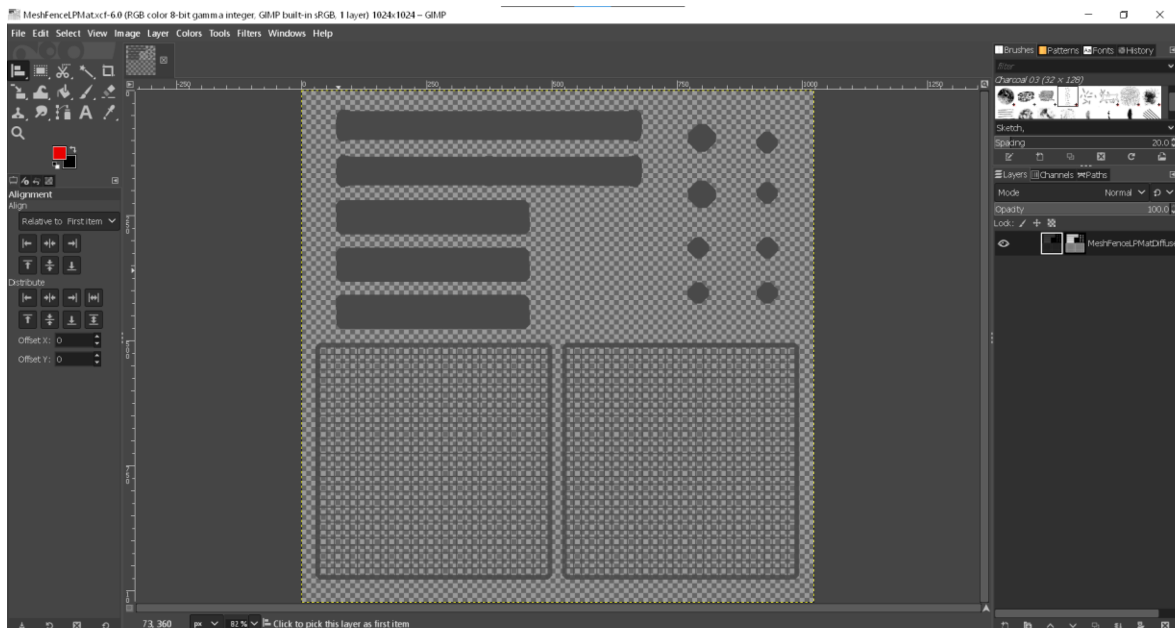
Kuva 26. Kuva, jota käytettiin verkkoaidan verkon reikien luomisessa tekstuuriin toistamalla muotoa tekstuurissa

Ensimmäistä kuvaa varten Blenderissä täytyy luoda tyhjä tekstuuri. Tekstuurin koon tulee olla melko suuri, jotta reikien reunat eivät jäisi pelissä liian rosoisen näköisiksi. Tekstuurille ei tarvitse luoda alpha channelia. Tekstuuriin saadaan siirrettyä 3D-mallin väri-informaatio navigoimalla Blenderin Properties-ikkunassa välilehteen Render Properties ja sen kohtaan Bake. Bake Type -vaihtoehdoksi valitaan Diffuse ja Contributions-vaihtoehdoista ainoastaan Color-vaihtoehdon tulee olla päällä, jotta ainoastaan 3D-mallin pintojen väri-informaatio siirretään tekstuuriin. Margin-arvoa eli marginaalia kannattaa myös lisätä muutama pikseli, jotta 3D-mallissa ei näkyisi saumoja tekstuuria käytettäessä. Kun Bake-asetukset on asetettu, painetaan Bake-nappia, jolloin Blender siirtää 3D-mallin pintojen väri-informaation tekstuuriin. Tekstuuri tallennetaan, jonka jälkeen toistetaan täsmälleen samat asiat uuden tekstuurin kanssa, mutta tällä kertaa Principled BSDF -shader noden Base Color -kohtaan yhdistetäänkin Image Texture -noden Alpha-kohta Color-kohdan sijaan. Suorittamalla Bake, saadaan luotua mustavalkoinen tekstuuri, johon on tallennettu tekstuurin läpinäkyvyysinformaatio. Kuvassa 27 nähdään näiden operaatioiden lopputulos, kaksi tekstuuria, jotka saadaan aikaiseksi tekemällä näin.



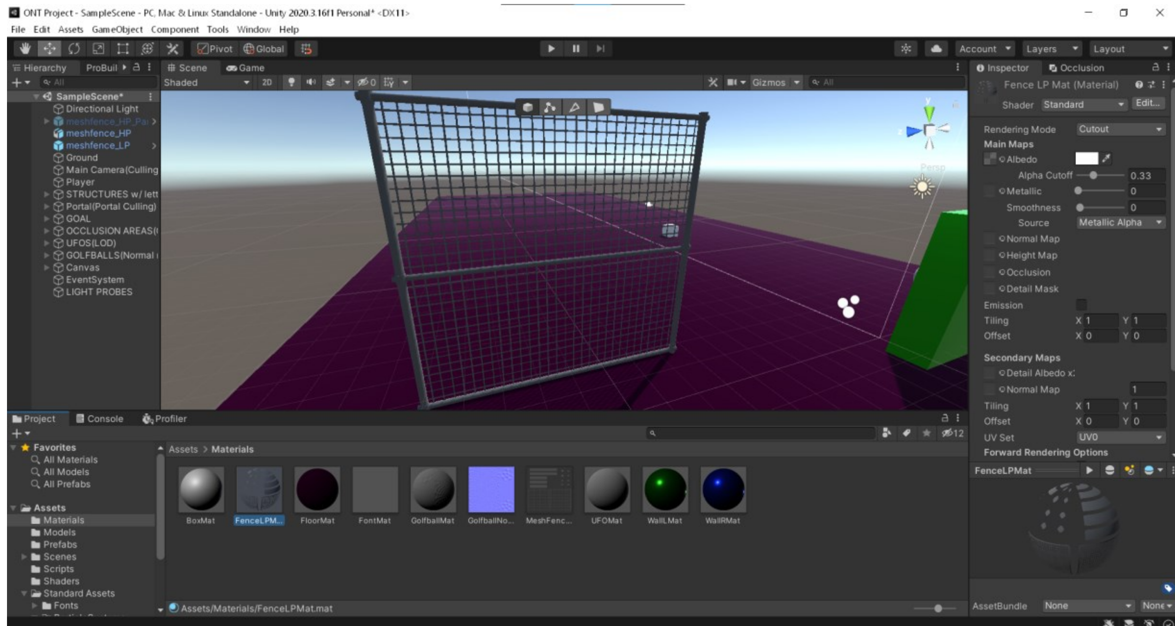
Kuva 27. Blenderissä luodut tekstuurit, joista vasemmanpuoleinen sisältää väri-informaation ja oikeanpuoleinen läpinäkyvyysinformaation

Koska Blenderin avulla ei pystytä luomaan suoraan läpinäkyvää tekstuuria, apuna täytyy käyttää GIMP-kuvankäsittelyohjelmaa. Sitä käyttämällä voidaan yhdistää kahden aiemmin luodun tekstuurin informaatiot — väri- ja läpinäkyvyysinformaatio — toisiinsa. Tämä tehdään viemällä GIMPiin aluksi vain tekstuuri, joka sisältää väri-informaation. Layerille, jolla tekstuuri sijaitsee, lisätään alpha channel, jotta kuvaan voidaan lisätä myös läpinäkyvyysinformaatiota. Layerille lisätään näiden vaiheiden jälkeen layer mask, jonka avulla voidaan säädellä layerin läpinäkyvyyttä. Sen toiminta perustuu siihen, että maskin valkoiset alueet ovat näkyviä ja mustat alueet läpinäkyviä. Layer maskina käytetään siis tekstuuria, johon on tallennettu lopullisen tekstuurin läpinäkyvyysinformaatio. Kuvassa 28 nähdään tekstuuri, joka saatiin lopputulokseksi, kun väri- ja läpinäkyvyysinformaatiot yhdistettiin GIMPissä toisiinsa. Kun kuva muutetaan png-tiedostoksi, kompressiota kannattaa käyttää mahdollisimman vähän, jotta rei'istä tulee ulkonäöltään parempilaatuisia ja reunoilta vähemmän rosoisia.



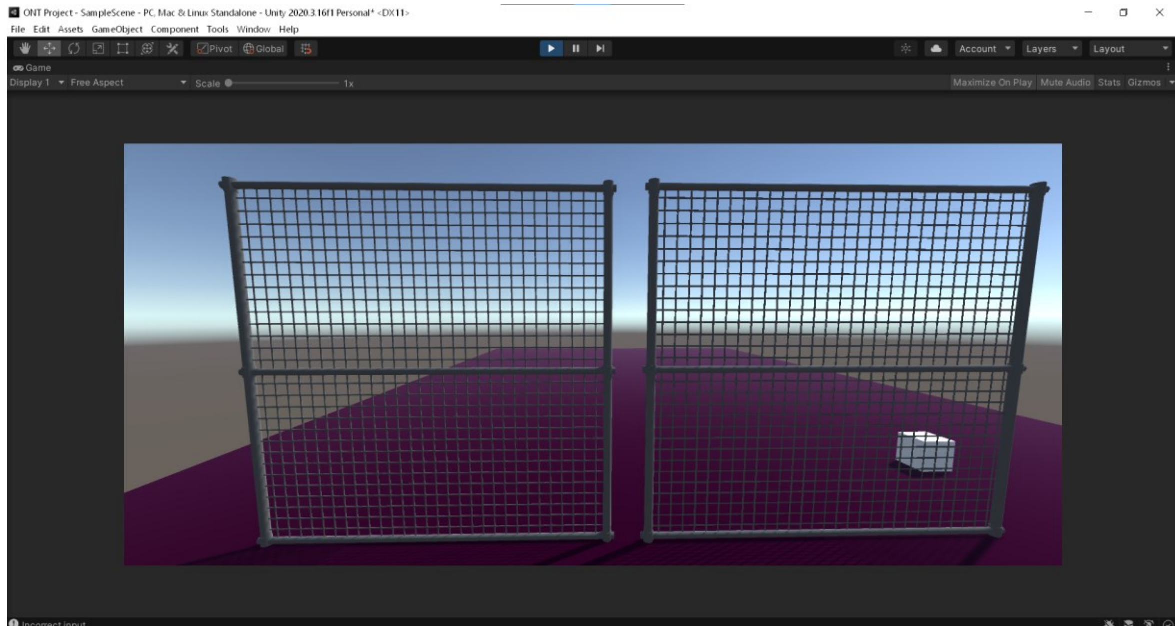

Kuva 28. GIMPissä luotu lopullinen tekstuuri, joka saadaan yhdistämällä väri- ja läpinäkyvyysinformaatiot toisiinsa kahden eri Blenderissä luodun tekstuurin avulla

Lopullinen tekstuuri ja 3D-malli lisätään lopuksi Unityyn, jossa tekstuuri lisätään 3D-mallin materiaaliin. Tekstuurin tuontiasetuksista täytyy laittaa päälle Alpha Is Transparency -kohta, jotta Unity ottaisi huomioon tekstuurin läpinäkyvyysinformaation eli ainakin tässä tapauksessa alpha channelin. Tekstuuria ei kannata myöskään kompressoida, mikäli haluaa parhaan näköisen lopputuloksen. Tekstuuri asetetaan lopuksi 3D-mallin materiaalin Albedo-tekstuuriksi ja Standard Shaderin Rendering Mode -vaihtoehdoksi valitaan Cutout. Alpha Cutoff -arvon tulee olla isompi kuin 0. Sitä säätämällä voidaan vaikuttaa tekstuurin reikien kokoon. Arvo 0 tarkoittaa, ettei reikiä ole tekstuurissa ollenkaan ja arvo 1 suurentaa reiät mahdollisimman isoiksi. Peliprojektissa arvoksi asetetaan 0.33, koska silloin reiät näyttävät mahdollisimman hyviltä. Kuvassa 29 nähdään valmis verkkoaidan 3D-malli Unityn scenessä läpinäkyvällä tekstuurilla.




Kuva 29. Verkkoaidan 3D-malli Unityssä läpinäkyvän tekstuurin kanssa

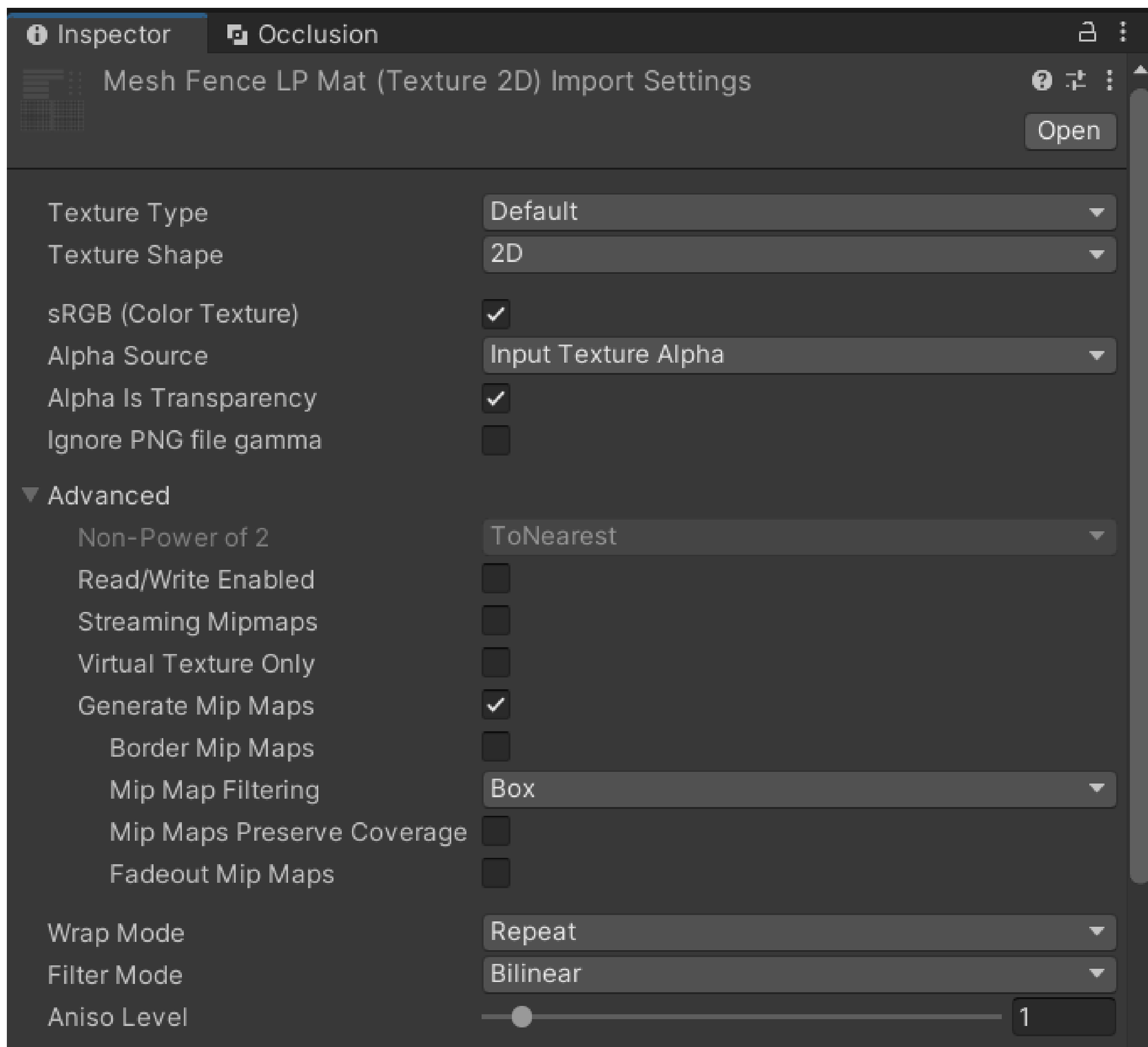
Jotta läpinäkyviä tekstuureja käyttävän verkkoaidan 3D-mallin ja niitä käyttämättömän verkkoaidan 3D-mallin aiheuttamaa vaikutusta suorituskykyyn voitaisiin verrata toisiinsa, verkkoaidan verkko täytyy luoda toiselle 3D-mallille mallintamalla. Koska 3D-malli on yksivärinen, sille ei edes luotu erillistä tekstuuria Blenderissä, sillä 3D-mallissa voidaan käyttää myös Unityn omaa materiaalia. Vaikka tekstuuri päätettäisiin silti luoda kyseiselle 3D-mallille, sen ei tarvitsisi olla läheskään yhtä iso kuin läpinäkyvää tekstuuria käytettäessä. 3D-mallin pintojen määrä tulisi kuitenkin olemaan paljon suurempi, kuin läpinäkyvää tekstuuria käytettäessä ja pintojen määrä on yksi tärkeimmistä asioista, jotka tulee huomioida 3D-mallien optimoinnin kannalta. Verkkoaidan 3D-mallissa — jossa läpinäkyvää tekstuuria on käytetty — pintoja on yhteensä 178, kun taas 3D-mallissa, jossa verkko on mallinnettu, pintoja on yhteensä 27170. Kuvassa 30 voidaan tarkastella kahden eri verkkoaidan 3D-mallin visuaalisia eroavaisuuksia Unityn scenessä. Tarkastelemalla mallien ulkonäköä ja pintojen määrää, voidaan todeta jo tässä vaiheessa, että läpinäkyvän tekstuurin käyttäminen on kannattavampaa tässä tilanteessa. 3D-mallien ulkonäössä ei ole merkittäviä eroja, mutta pintojen määrä on silti dramaattisesti — yli 150 kertaa — suurempi 3D-mallissa, jossa reiät on luotu mallintamalla. Pelaaja voi huomata reikien kolmiulotteisuuden tai paksuuden puuttumisen rei’istä sivulta katsottuna, mutta hyöty suorituskyvyn kannalta on visuaaliseen haittaan verrattuna niin suurta, että tästä voidaan olla välittämättä.




Kuva 30. Vasemmanpuoleisessa 3D-mallissa verkon reiät on mallinnettu, oikeanpuoleisessa 3D-mallissa ne on saatu aikaan läpinäkyvää tekstuuria käyttäen

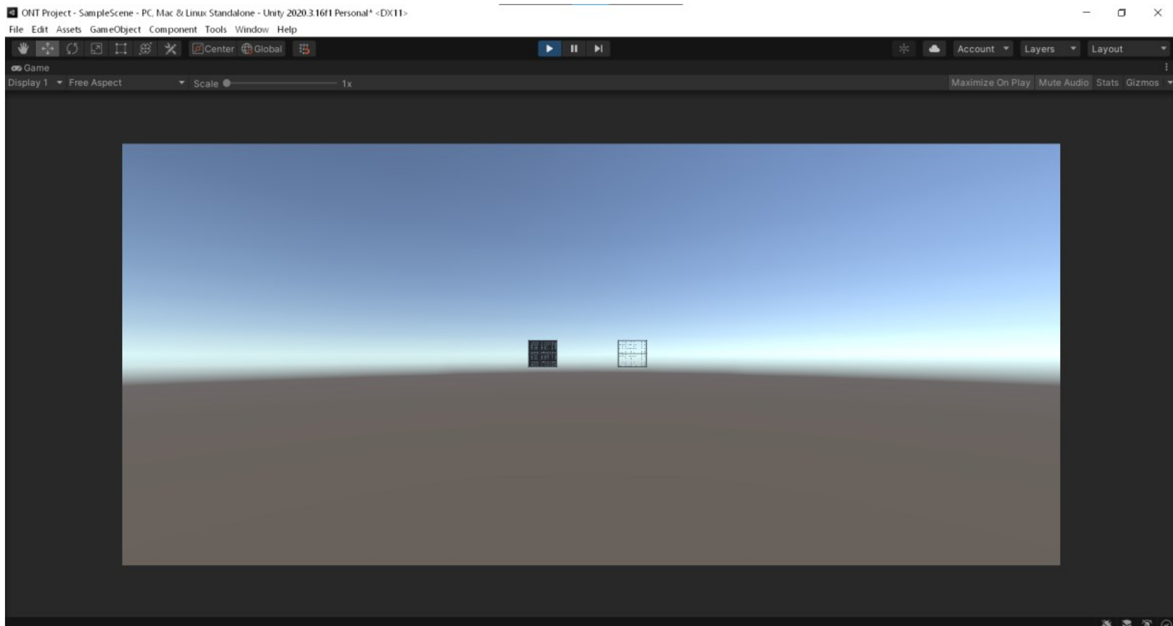
### 8.4.3 Mipmaps-menetelmän käyttäminen

Mipmaps-menetelmä on automaattisesti käytössä jokaisen 3D-mallin tekstuurilla. Tekstuurien tuontiasetuksissa vaihtoehto Generate Mip Maps on jo valmiiksi valittuna. Kuvassa 31 nähdään esimerkki verkkoaidan tekstuurin tuontiasetuksista, joita ei ole muokattu. Unity luo mipmapit automaattisesti kaikille tekstuureille, joilla kyseinen asetus on päällä. Menetelmän käyttöön ottamiseksi ei siis tarvitse välttämättä tehdä mitään. Tavallisten tekstuurien kanssa tuontiasetuksiin ei tarvitse tehdä mitään, mutta Generate Mip Maps -asetus kannattaa ottaa pois päältä silloin, jos tekstuuri ei tule koskaan olemaan kaukana pelaajan näkymästä, sillä se vie enemmän levytilaa — tekstuurin koosta noin kolmasosan — eikä menetelmästä ole mitään hyötyä tällaisessa tapauksessa.



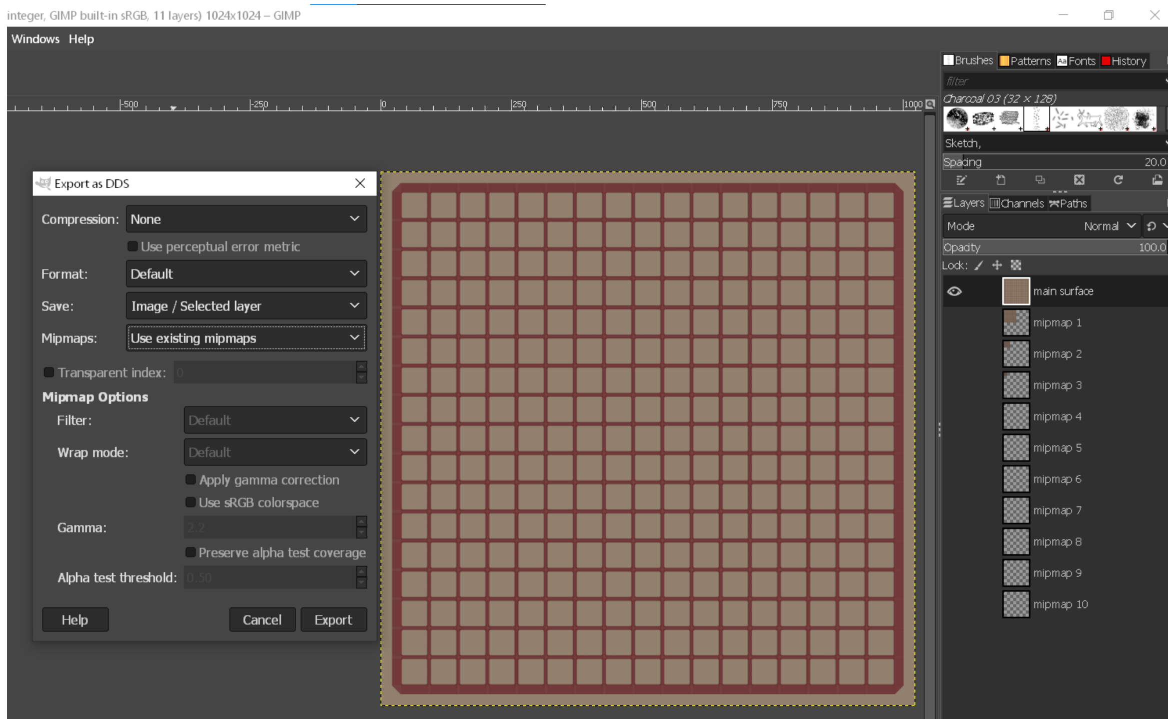

Kuva 31. Verkkoaidan tekstuurin tuontiasetukset, joissa mipmap-tasojen automaattinen luominen on asetettu päälle

Läpinäkyviä tekstuureja käytettäessä tekstuurien tuontiasetuksilla voidaan vaikuttaa siihen, millaisia mipmap-tasoista tulee. Esimerkiksi verkkoaidan 3D-mallin tekstuurin mipmapeista voidaan tehdä joko sellaisia, että kaukaa katsottuna tekstuurin läpi näkee kokonaan tai ei näe ollenkaan. Kuvassa 32 vierekkäin kaksi samanlaista tekstuuria käyttävää verkkoaidan 3D-mallia, joissa toisesta näkee läpi kaukaa katsottuna, mutta toisesta ei. Jos halutaan, että pelaaja ei näe kaukaa katsottuna verkkoaidan läpi, tekstuurin tuontiasetusten mipmaps-kohdasta voidaan valita vaihtoehto Mip Maps Preserve Coverage, jonka päälle laittaminen tuo esille Alpha Cutoff Value -muuttujan. Sen arvoksi tulee asettaa yli 0.5–0.99, jotta pelaaja ei näkisi verkkoaidasta läpi kaukaa. Unity luo tekstuurille näihin asetuksiin pohjautuvat mipmap-tasot, kun muutokset hyväksytään valitsemalla Apply-vaihtoehdon. Tämän jälkeen verkkoaidasta ei näe enää läpi kaukaa. Jos verkkoaidasta saa nähdä kaukaa katsottuna läpi, tuontiasetuksille ei tarvitse tehdä mitään.



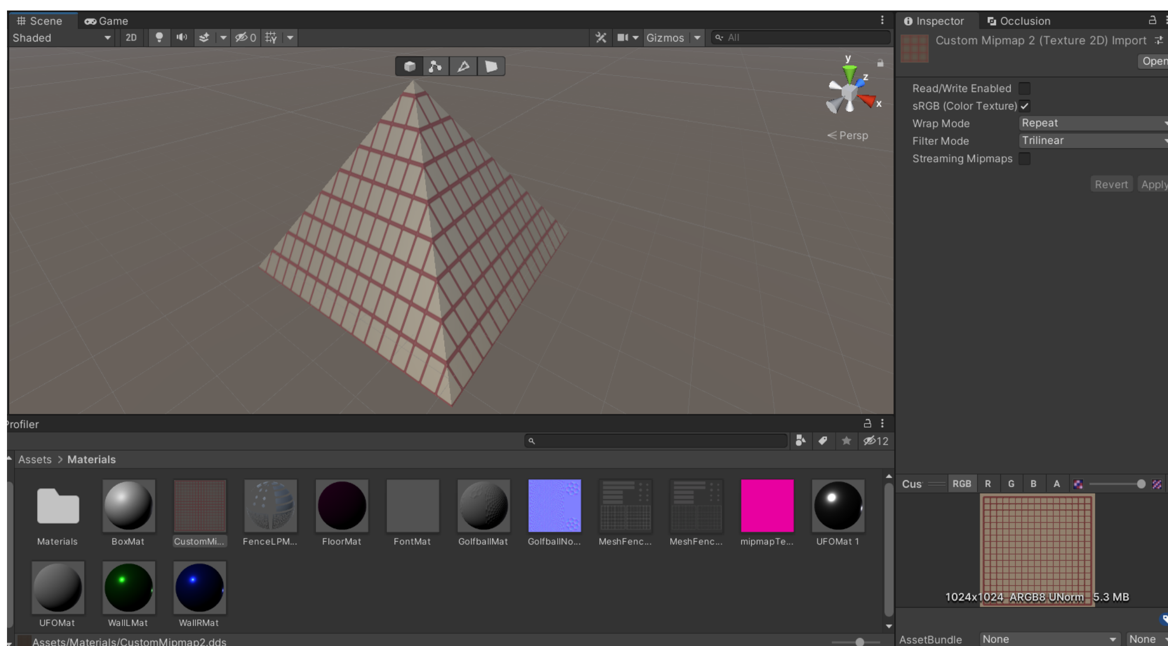

Kuva 32. Verkkoaidan 3D-mallit, joiden tekstuurien mipmaps-asetukset ovat erilaiset

Mipmapien automaattinen generointi on helppoa Unityssä, eikä niitä yleensä tarvitse luoda itse, mutta joissain tapauksissa mipmap-tasoja saatetaan haluta muokata itse, ehkä jonkin ominaisuuden vuoksi. Tätä varten täytyy luoda dds-tiedosto, joka sisältää alkuperäisen tekstuurin lisäksi eri mipmap-tasoja. Tiedoston luominen onnistuu luomalla ensin alkuperäinen tekstuuri esimerkiksi Blenderin avulla ja siirtämällä luotu png-tekstuuri sitten kuvankäsittelyohjelma GIMPiin. GIMPissä png-tiedosto muunnetaan dds-tiedostoksi tekemällä Export eli tiedoston vienti ja valitsemalla tiedostoksi DDS image. Vientiasetuksissa Mipmaps-kohtaan valitaan vaihtoehto Generate mipmaps ja Filter-kohtaan vaihtoehto Box. Kun dds-tiedosto luodaan, se generoi automaattisesti eri mipmap-tasot. Näin saadaan luotua tavallinen mipmap-tekstuuri. Jos tekstuuria halutaan muokata, dds-tiedosto täytyy avata, jolloin huomataan, että tiedosto sisältää monta eri mipmap-tasoa. Näitä mipmap-tasoja voidaan muokata, joten niitä voidaan esimerkiksi parannella tai voidaan saada aikaiseksi ainutlaatuisia efektejä tekstuurin muuntuessa erilaiseksi eri etäisyyksillä. Haluttujen muutosten jälkeen tiedosto viedään uudelleen, mutta tällä kertaa Mipmaps-kohtaan valitaan vaihtoehto Use existing mipmaps, jolloin dds-tiedosto tallentaa mipmap-tasoihin tehdyt muutokset. Kuvassa 33 esimerkki GIMPissä avatusta peliprojektiin lisätystä dds-tiedostosta ja vientiasetuksista. Mipmap-tasot avautuvat GIMPissä erillisinä layereinä.




Kuva 33. GIMPissä avattu, projektissa käytetty dds-tiedosto ja vientiasetus-ikkuna

Itsetehtyjä mipmapeja on helppo käyttää, sillä dds-tiedosto täytyy vain siirtää Unityyn ja käyttää sitä 3D-mallin tekstuurina. Tekstuurin vientiasetukset ovat suppeammat kuin tavanomaisella tekstuurilla, kuten esimerkiksi png-tiedostolla. Asetuksia ei kuitenkaan tarvitse muuttaa. Peliprojektissa käytetyn dds-tiedoston mipmap-tasot tummenevat vähitellen mipmap-tasojen vaihtuessa. Kuvassa 34 nähdään peliprojektiin lisätty pyramidia esittävä 3D-malli, jonka tekstuurina käytetään dds-tiedostoa itsetehtyjä mipmap-tasoja varten.



Kuva 34. Projektissa käytetty pyramidin 3D-malli, jonka tekstuurina on käytetty dds-tiedostoa

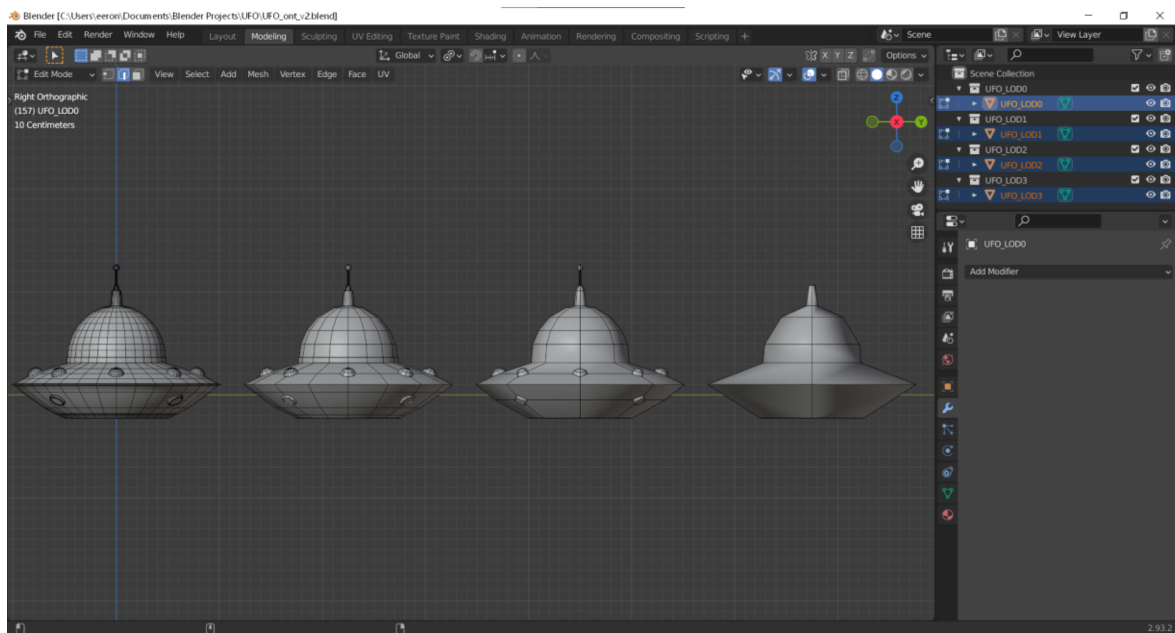
### 8.4.4 LOD-mallin luominen ja käyttöönotto

LOD-malli voidaan luoda Blenderiä käyttäen ainakin kahdella eri tavalla. Valmista mallia voidaan muokata manuaalisesti vähentämällä pintoja halutuista kohdista tai voidaan käyttää Decimate-modifieria, joka vähentää automaattisesti pintoja 3D-mallista. Manuaalista tapaa käytettäessä saadaan yleensä paremman näköinen lopputulos, mutta se on hidasta ja aikaa vievää työtä. Decimate-modifieria käytettäessä LOD-malli saadaan luotua ideaalisissa olosuhteissa saman tien, riippuen alkuperäisestä 3D-mallista. Huono puoli kyseistä modifieria käytettäessä on usein LOD-mallin huonompilaatuinen ulkonäkö. Modifieria käytettäessä tapahtuu myös jonkin verran virheitä, joita täytyy myöhemmin korjata manuaalisesti. Onkin järkevää käyttää molempia keinoja, jotta LOD-mallin luominen olisi vähemmän aikaa vievää. Decimate-modifier ei välttämättä luo täydellisiä LOD-malleja, mutta sen käyttäminen säästää aikaa LOD-versioiden luomisessa. Decimate-modifier vähentää pintojen määrää pyrkien samalla muuttamaan 3D-mallin muotoa mahdollisimman vähän. (Blender Foundation 2020a.)

Jos LOD-versioita ei haluta tehdä hitaasti käsin, valmiiseen 3D-malliin voidaan lisätä Blenderissä Decimate-modifier oikealla sijaitsevan valikon välilehdestä Modifiers. Sieltä löytyy valikko, josta valitaan modifier Decimate. Decimate-modifierissa on kolme vaihtoehtoa: Collapse, Un-Subdivide ja Planar. Ne ovat eri tapoja vähentää 3D-mallin pintoja, ja niitä tulee käyttää eri tapauksissa. Esimerkiksi Planar ei toimi kunnolla pyöreisiin 3D-malleihin. (Blender Foundation 2020a.)

LOD-malli voi sisältää LOD-versioita Unityssä maksimissaan kahdeksan, mutta yleensä niitä ei ole järkevää tehdä niin suurta määrää, vaan tyypillisesti yksi LOD-malli sisältää alle viisi kappaletta LOD-versioita. Peliprojektiin luotiin lentävää lautasta esittävä LOD-malli, johon sisältyy neljä eri LOD-versiota 3D-mallista. Ensimmäinen 3D-malli eli LOD-versio on alkuperäinen 3D-malli, LOD0, joten kolme muuta LOD-versiota luodaan sen pohjalta. Järkevintä kyseisen 3D-mallin kohdalla on käyttää Decimate-modifierin vaihtoehtoa Un-Subdivide, koska 3D-malli on pyöreä, symmetrinen ja topologia on lähes täydellistä. LOD1 on tehty kokonaan käsin, sillä siinä on vain vähän muutoksia alkuperäiseen verrattuna, mutta kuitenkin sen verran, että LOD-mallista on hyötyä. LOD2- ja LOD3-versio on tehty Decimate-modifierin Un-Subdivide-vaihtoehtoa käyttäen, mutta sen aiheuttamat virheet 3D-malleihin on korjattu manuaalisesti. LOD-versioita on myös yksinkertaistettu jonkin verran käsin. Esimerkiksi antennista on poistettu pala, sillä se on lähes täysin huomaamaton kaukaa katsottuna. LOD-versioita tehdessä kannattaa yleensä huomioida, miltä 3D-malli näyttää

kaukaa katsottuna ja kuinka selvästi pelaaja huomaa LOD-version vaihtumisen seuraavaan LOD-versioon. Useissa, jopa hyvin tunnetuissa peleissä, kuten Left 4 Dead -sarjan peleissä joidenkin 3D-mallien LOD-versioiden vaihtumisen toiseen LOD-versioon huomaa kuitenkin selvästi. Kyseessä on silti visuaalinen seikka, joka saattaa häiritä joitain pelaajia ja voi tehdä pelistä selvästi huonomman näköisen, joten mallien vaihtuminen kannattaa tehdä mahdollisimman hienovaraisen näköisesti. Kuvassa 35 kaikki peliprojektissa käytetyn LOD-mallin LOD-versiot on asetettu vierekkäin Blenderin näkymässä siten, että LOD-versioiden pintojen eri määrä huomataan mahdollisimman selvästi.

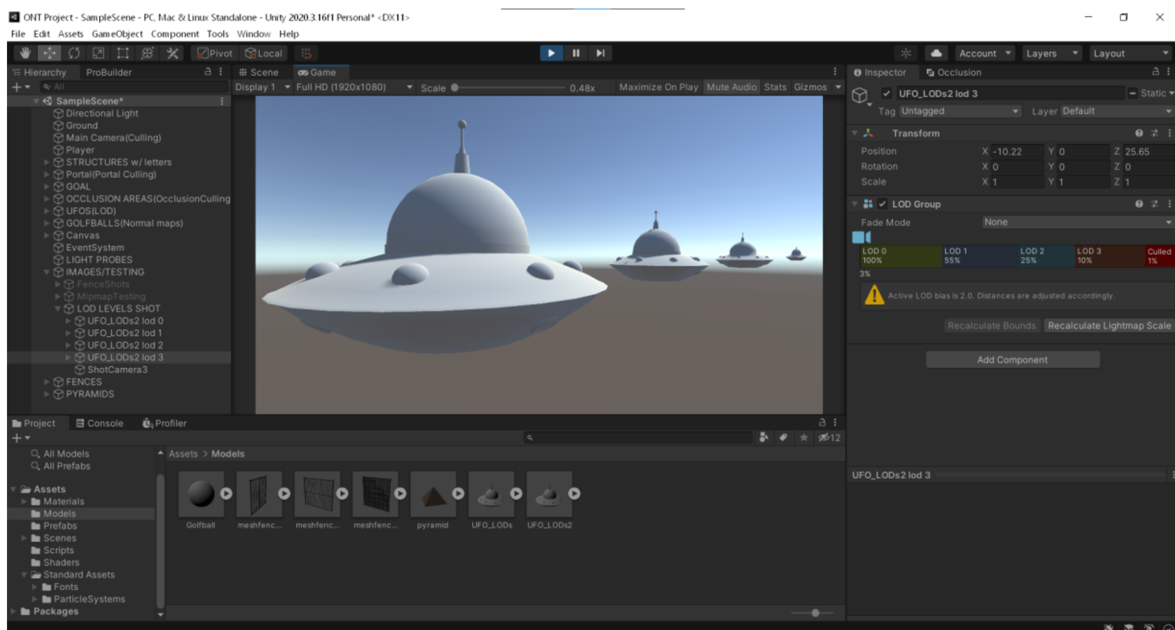



Kuva 35. Lentävää lautasta esittävän 3D-mallin LOD-versiot: LOD0: 2600 pintaa, LOD1: 630 pintaa, LOD2: 200 pintaa, LOD3: 25 pintaa

Unityssä 3D-mallille täytyy lisätä LOD-menetelmän käyttöönottoa varten LOD Group -komponentti. Sen avulla voidaan säädellä joitakin LOD-mallin ominaisuuksia, kuten sitä, millä etäisyyksillä LOD-mallit vaihtuvat. LOD-versioita voidaan lisätä, vaihtaa tai päivittää myöhemmin LOD-malliin komponentin avulla. LOD-malli voidaan luoda lisäämällä LOD-versioita yksitellen LOD Group -komponenttiin, mutta tähän on myös yksinkertaisempi keino. (Unity Technologies 2021m.)

Helpompi tapa ottaa LOD-menetelmä käyttöön Unityssä on melko yksinkertainen, eikä siihen vaadita mitään muita toimenpiteitä Unityssä kuin LOD-mallin lisääminen peliin eli Unityn sceneen. Jotta Unity osaisi päätellä LOD-mallin LOD-tasot, LOD-versioiden 3D-mallit täytyy nimetä Blenderissä nimeämiskäytännöllä xxx_LODX, jossa xxx on mallin nimi ja X on LOD-mallin numero. Jos malleja ei ole nimetty näin, Unity epäonnistuu LOD Group -komponentin automaattisessa luomisessa. LOD-versioiden täytyy sijaita Blenderissä

samassa kohdassa eli päällekkäin, eikä niissä saa lopuksi olla enää modifierejä. Seuraavaksi LOD-malli täytyy muuttaa Unityssä käytettäväksi fbx-tiedostoksi. Tätä varten Blenderissä valitaan kaikki LOD-versiot, mutta kuitenkin viimeisenä LOD0, jolloin LOD0 jää aktiiviseksi – tämän huomaa valinnan väristä. Tämän jälkeen Blenderissä valitaan tiedoston muuttaminen fbx-tiedostoksi, jolloin avautuu ikkuna, josta voidaan valita viemisen eli export-operaation asetukset. Täältä valitaan muunnettavaksi ainoastaan mesh ja laitetaan päälle lisäasetus !Experimental! Apply Transform. Kun objekti on muunnettu fbx-tiedostoksi, se voidaan raahata Unityyn. Kun se siirretään Unityn sceneen, sillä pitäisi olla jo valmiina täydellinen LOD Group -komponentti. Kuvassa 36 valmis LOD-malli, LOD Group -komponentti ja LOD-mallin eri LOD-tasot ja -versiot ovat näkyvissä Unityn scenessä. (Unity Technologies 2021l; Unity Technologies 2021m.)




Kuva 36. LOD-mallin eri LOD-tasot ja -versiot Unityssä

Peliprojektissa UFOa esittävät LOD-mallit on aseteltu peräkkäin niin, että pelaaja voi havainnoida LOD-menetelmän toimintaa. LOD-tasot vaihtuvat peliprojektissa pienemmällä etäisyydellä kuin oletuksena oli määritelty, joten UFOn LOD-tasojen vaihtumisen voi huomata melko selvästi. Tavallisesti pelinkehityksessä täytyy kuitenkin pyrkiä siihen, että LOD-tasojen vaihtuminen on mahdollisimman huomaamatonta, sillä pelaajan ei ole tarkoitus huomata LOD-tasojen vaihtumista peleissä. LOD-menetelmän kuuluisi ideaalisti olla hienovarainen ja pelaajalle lähes näkymätön optimointimenetelmä.

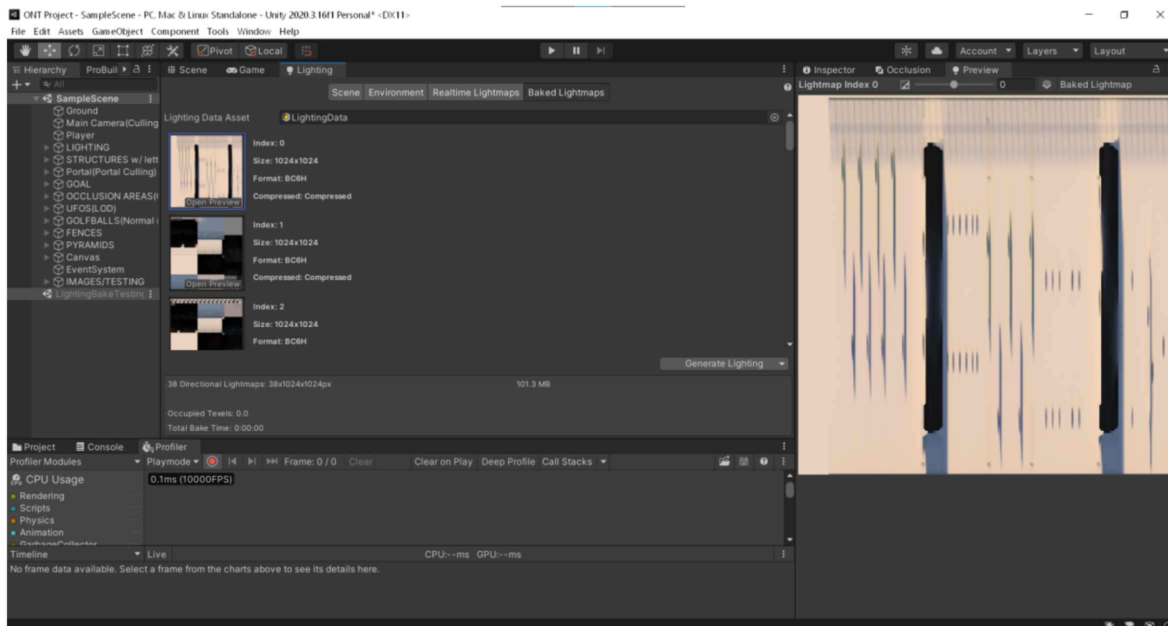
## 8.5 Valaistuksen optimointi Unityssä

Toteutuksessa käytetyt valaistukseen liittyvät optimointimenetelmät eli Lightmapping ja Light Probes perustuvat valaistusinformaation laskemiseen etukäteen ja menetelmien käyttöön ottaminen koostuu suurimmaksi osaksi tarvittavista toimenpiteistä, jotka täytyy suorittaa ennen valaistusinformaation laskemista. Käyttöön ottamisessa on tärkeää arvioida, että etukäteen laskettu valaistus parantaa suorituskykyä, mutta näyttää myös lähes yhtä hyvältä kuin ajonaikana laskettu Realtime-valaistus. Käyttöön ottamisessa kannattaa lisäksi ottaa huomioon, että valaistusinformaation laskemista ei kannata kuitenkaan tehdä liian tarkasti, sillä epätarkempi valaistusdata on hyödyllisempää suorituskyvyn kannalta ja sen koko levyllä on lisäksi pienempi. Profilerin avulla tarkastellaan lopuksi menetelmien hyötyjä ja verrataan pelin valaistusta Realtime-valaistukseen.

### 8.5.1 Lightmapping-menetelmän toteutus

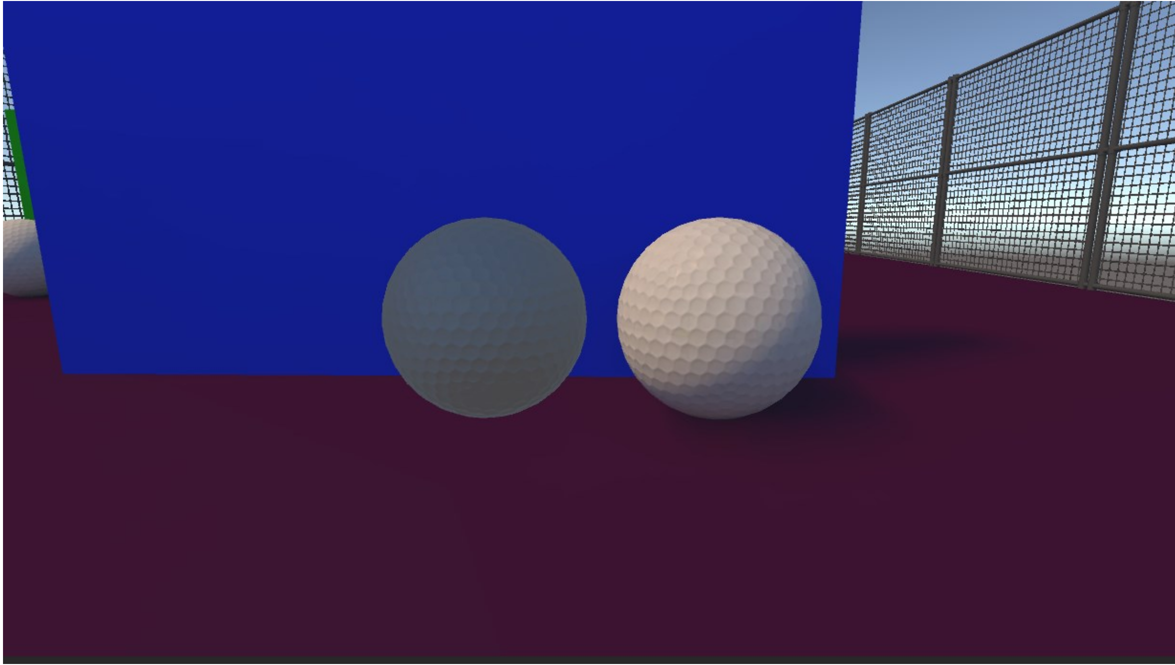
Lightmapping-menetelmää varten 3D-mallille luodaan UV map, jota käytetään ainoastaan lightmapin generoimiseen. Tämä voidaan tehdä joko Blenderissä Edit Modessa valitsemalla UV – Lightmap pack tai automaattisesti Unityssä. Blenderissä 3D-malleille on useimmiten jo luotu UV map tekstuureja varten, joten 3D-malliin täytyy lisätä lightmapia varten toinen UV-map. (Unity Technologies 2021c.)

Unity luo automaattisesti uuden UV-mapin 3D-mallille lightmapia varten. Tämä voidaan tehdä 3D-mallin tuontiasetuksista (Import settings). Siirrytään Model-välilehden Geometry-osioon, jossa valitaan Generate Lightmap UVs. Lightmap UV:n luomisen asetuksia voidaan säätää alle avautuvasta Lightmap UV settings -kohdasta. Jotta objektille voidaan luoda lightmap, se täytyy lisäksi merkitä Static-merkinnällä Contribute GI, sillä lightmapia ei voida luoda liikkuville objekteille. Lisäksi haluttujen valojen asetus Mode täytyy muuttaa Inspectorista vaihtoehdoksi Baked. Seuraavaksi avataan valaistusasetukset valitsemalla päävalikosta Window – Rendering – Lighting. Scene-välilehden alareunasta valitaan painike Generate Lighting, jolloin Unity aloittaa lightmapien generoimisen. Painiketta ei tarvitse painaa jokaisella kerralla, kun valoja lisätään peliin, jos Auto Generate -vaihtoehto laitetaan päälle. Tällöin Unity generoi lightmapin aina, kun peliin lisätään uusi valo. Kuvassa 37 on esimerkki yhdestä lightmapista, jonka Unity generoi pelin ”lattiana” toimivalle alustalle tai plane-objektille, jonka päällä pelaajahahmo liikkuu.



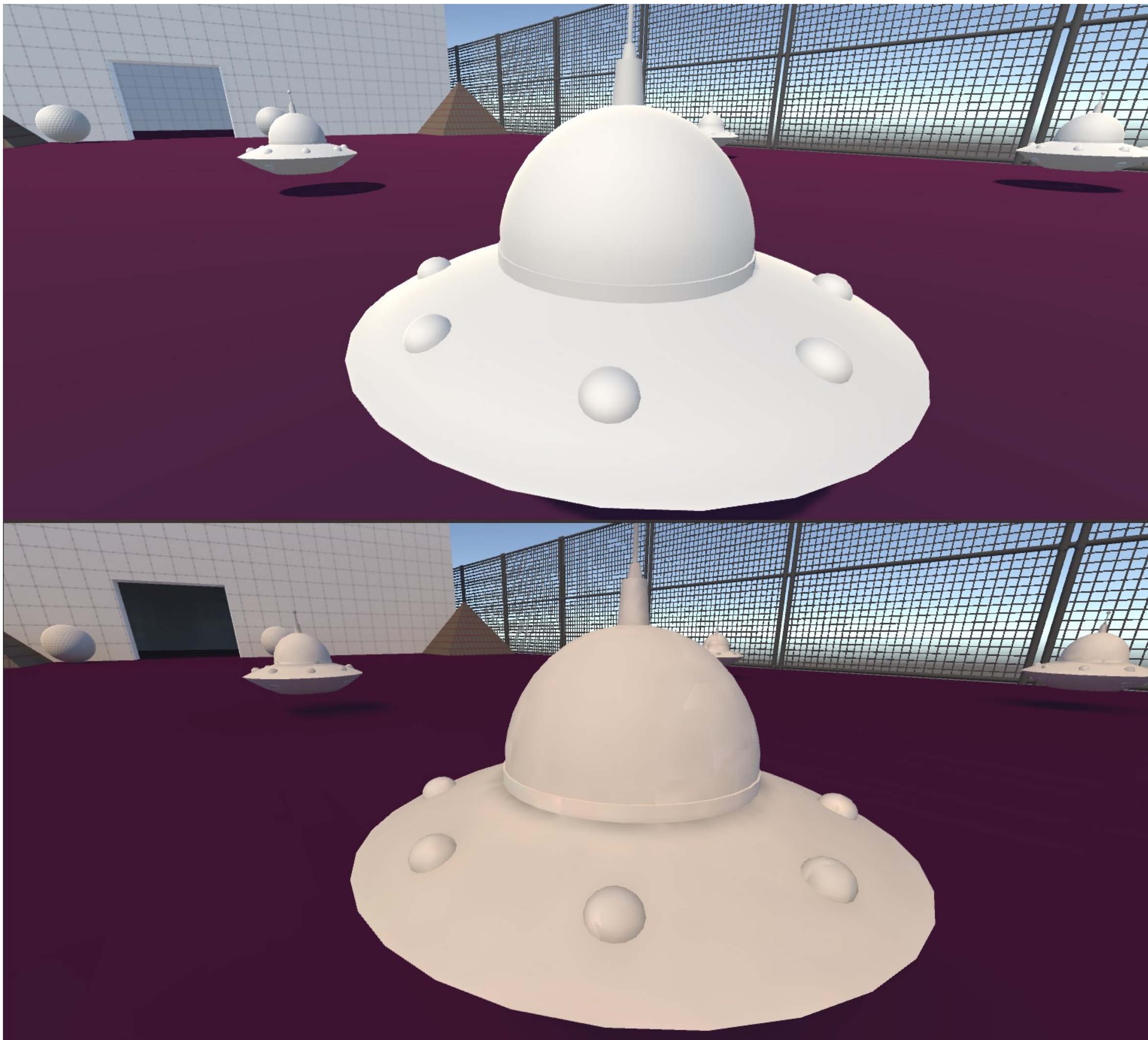

Kuva 37. Oikealla Unityn generoima lightmap pelin alustalle tai ”lattiana” toimivalle plane-objektille

Kannattaa ottaa huomioon, että objektien materiaalit vaikuttavat jonkin verran lightmapeihin, kun pelkkää Baked-valoa on käytetty valaistukseen. Jos tietyt objektit ovat huonosti valaistuja, tummempia kuin pitäisi, syy voi olla siinä, että objektin materiaalin Metallic-arvo on suurempi kuin 0. Jos esimerkiksi objektin materiaalin Metallic-arvo on 1, objekti on täysin tumma, näyttäen siltä, ettei sitä olisi valaistu millään tavalla. Toisin sanoen Baked-valoja ei kannata käyttää objektien kanssa, joiden halutaan näyttävän metallisilta. Sen sijaan kannattaa käyttää Mixed-valoja. Kuvassa 38 sama peliprojektiin lisätty golfpallon 3D-malli, joista vasemmanpuoleisen materiaalissa Metallic-arvo on 1, oikeanpuoleisessa arvo on 0.



Kuva 38. Metallic-arvon käyttäminen materiaalissa Baked-valojen ollessa käytössä

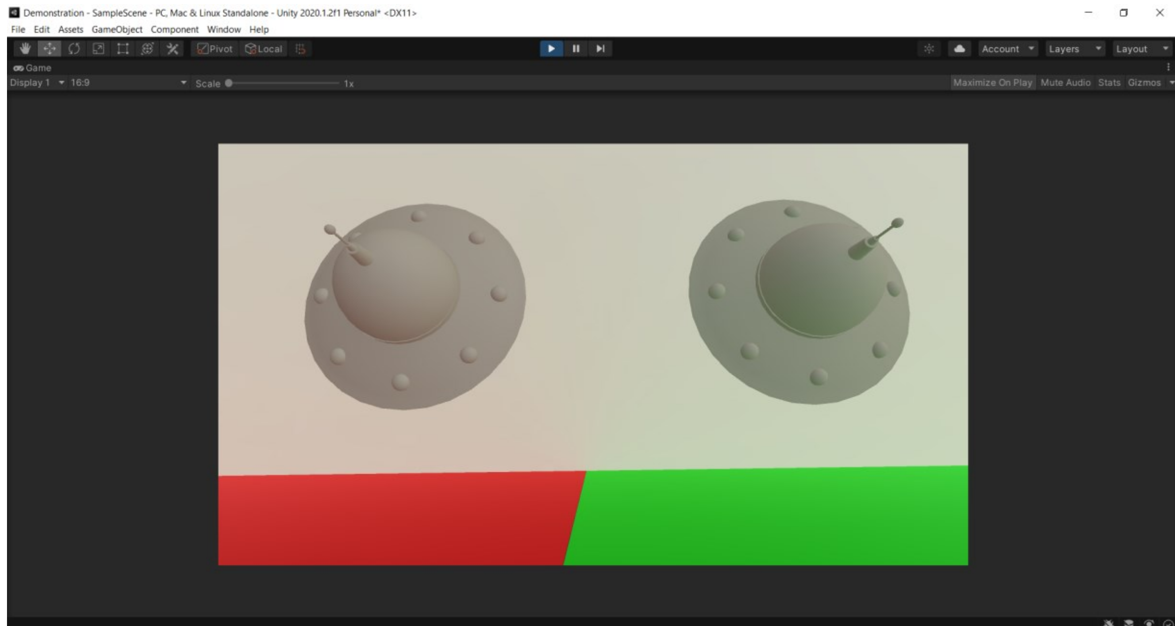
Peliprojektissa käytetään vain yhtä Baked-valoa. Lightmapien generoimisessa kesti yli tunti ja Unity generoi yhteensä 38 resoluutioiltaan 1024x1024 lightmapia, joiden koko on yhteensä noin 100 megatavua. Kuvassa 39 esitellään Realtime- ja baked-valaistusten eroja. Ylemmässä osassa kuvaa peliprojektissa on käytetty Realtime-valaistusta ja alemmassa osassa Baked-valaistusta. Baked-valaistuksessa — varsinkin lähellä olevan UFOa esittävän 3D-mallin pinnalla — voidaan huomata joitakin virheitä, mutta niitä voi korjata muuttamalla Bake-asetuksia, jolloin lightmap-tekstuureista tulee tarkempia. Tämä tarkoittaa kuitenkin myös sitä, että lightmapien generointi kestäisi kauemmin ja niiden koko levyllä olisi suurempi. Lightmapeista ei kannata tehdä tämän takia liian tarkkoja, vaan vain niin tarkkoja, kuin on tarpeellista. Pieniä virheitä lightmapeissa ei esimerkiksi huomaa kaukaa katsottuna.



Kuva 39. Realtime- ja Baked-valaistus peliprojektissa

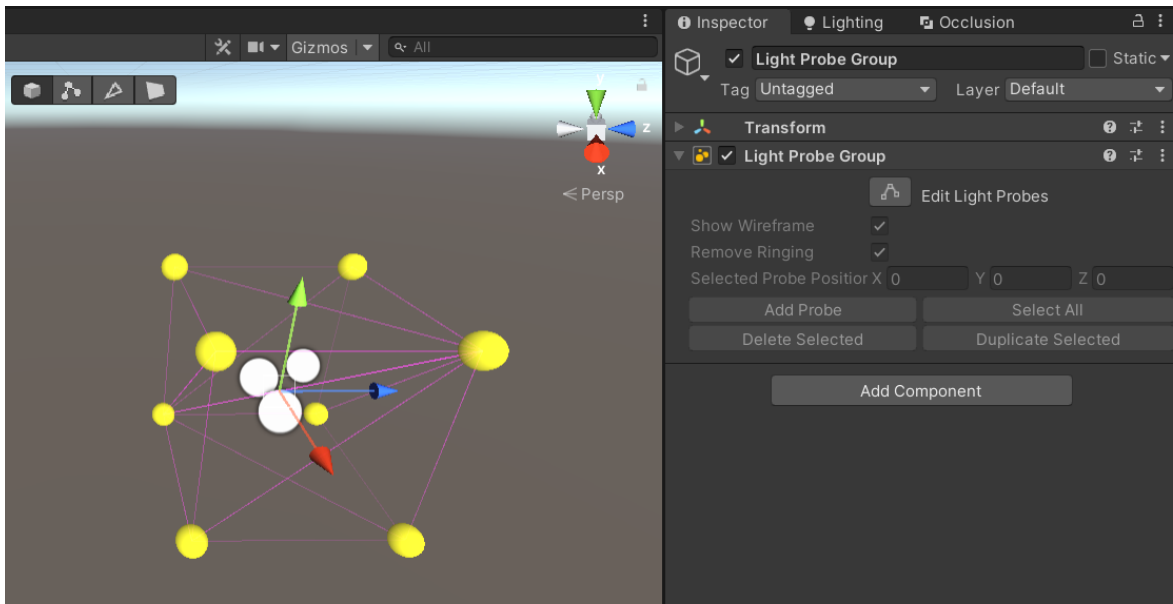
### 8.5.2 Light Probes -menetelmän toteutus

Light Probes -menetelmä tarvitsee toimiakseen valonlähteen, joka on staattinen, eli Unityn Light-komponentin, jonka Mode-vaihtoehdoksi on valittu Baked sekä light probeja. Useimmiten Light Probes -menetelmällä halutaan myös heijastaa valoa objekteista muiden objektien pinnoille, joten tarvitaan myös yksinkertainen staattinen objekti, josta valo voi heijastua, kuten seinä, joka on merkitty staattisella merkinnällä Contribute GI. Kuvassa 40 nähdään, kuinka kahden lentävää lautasta esittävän 3D-mallin pintaan heijastuu hieman valoa vihreästä ja punaisesta objektista. Valo ei heijastu tällä tavoin muiden objektien pinnoille, ellei Light Probes -menetelmä ole käytössä. Samaa ilmiötä ei saada aikaiseksi edes Realtime-valaistusta käyttämällä.



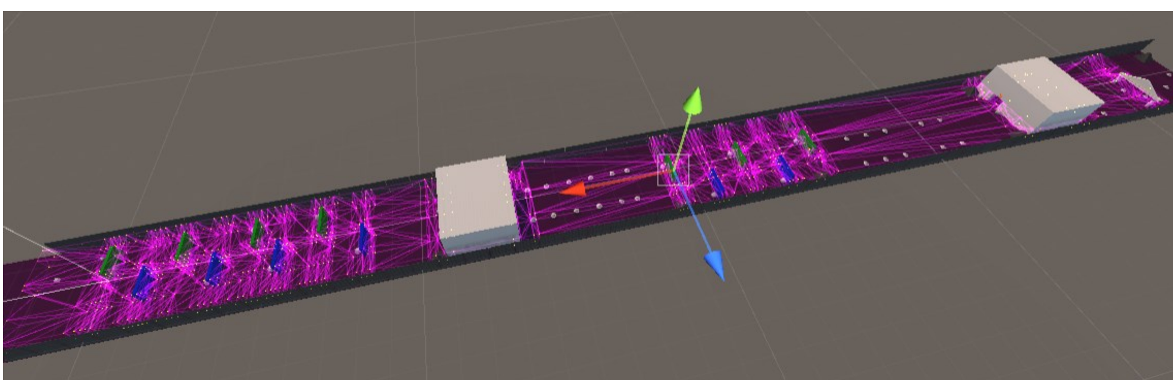

Kuva 40. Valo heijastuu erivärisistä seinistä objektien pinnalle Light Probes -menetelmän ansiosta

Light probeja voidaan luoda Unityssä lisäämällä sceneen valmiiksi määritellyn esineen Light Probe Group, jolloin Unity luo peliin objektin, joka sisältää Light Probe Group -komponentin. Kuvassa 41 on esimerkki juuri luodusta Light Probe Group -objektista, jossa light probet on asetettu simppelin kuution muotoon. Light Probe Group -komponentin avulla voidaan muokata light probeja. Light probeja voidaan kopioida, lisätä, poistaa ja liikuttaa. Light probet täytyy sijoittaa hieman valaistuksessa käytettävän staattisen objektin pinnan eteen, jotta valaistus generoituu oikealla tavalla. Light probeja ei saa sijoittaa esimerkiksi 3D-objektin sisäpuolelle, mikäli halutaan oikeanlainen valaistus. Light Probe Group -komponentteja voidaan lisätä monta yhteen Unityn sceneen, mutta tässä ei ole järkeä, sillä Light Probe Group -komponenttien light probet yhdistyvät aina toisiinsa. Toisin sanoen light probeja voi olla scenessä ainoastaan yksi yhtenäinen kokonaisuus eli mesh. Toinen merkittävä asia on, että light probe täytyy lisätä manuaalisesti koko pelikenttään — mikäli ei haluta jättää joidenkin alueiden valaistusta erilaiseksi — ja suurella pelikentällä tämä voi olla työläs ja aikaa vievä prosessi. (Unity Technologies 2021n.)




Kuva 41. Light Probe Group -objekti vasemmalla ja Light Probe Group -komponentti oikealla

Light probe -meshin luominen peliprojektiin oli yllättävän monimutkaista ja joillain alueilla mesh ei ole täydellinen, sillä pelaaja tai muu dynaaminen objekti ei koskaan liiku tietyille alueille pelissä. Kuvasta 42 nähdään tarkemmin, millainen light probe -mesh peliprojektiin lopulta luotiin. Vaaleanpunaiset viivat ovat light probejen välille piirtyviä valaistavien alueiden ääriviivoja. Light probeja on aseteltu enemmän sinisten ja vihreiden seinien lähelle, sillä siellä menetelmä vaikuttaa valaistukseen eniten. Kun light probe -mesh oli valmis, valaistusdata täytyi lopuksi vielä generoida samalla tavoin kuin Lightmapping-menetelmässä. Jotta bake voidaan luoda, Lighting-ikkuna täytyy avata. Scene-välilehden alareunasta valitaan painike Generate Lighting, joka on sama kuin Lightmapping-menetelmässä. Tämä tehtiin samaan aikaan kuin Lightmapping-menetelmän lightmapien generoiminen, joten valaistusdatan generoiminen kesti yli tunnin. Bake-asetuksista voitaisiin vielä muuttaa joitakin Light Probes -menetelmälle olennaisia arvoja, jotta valaistus näyttäisi paremmalta, mutta tässä tapauksessa käytettiin oletusarvoja.



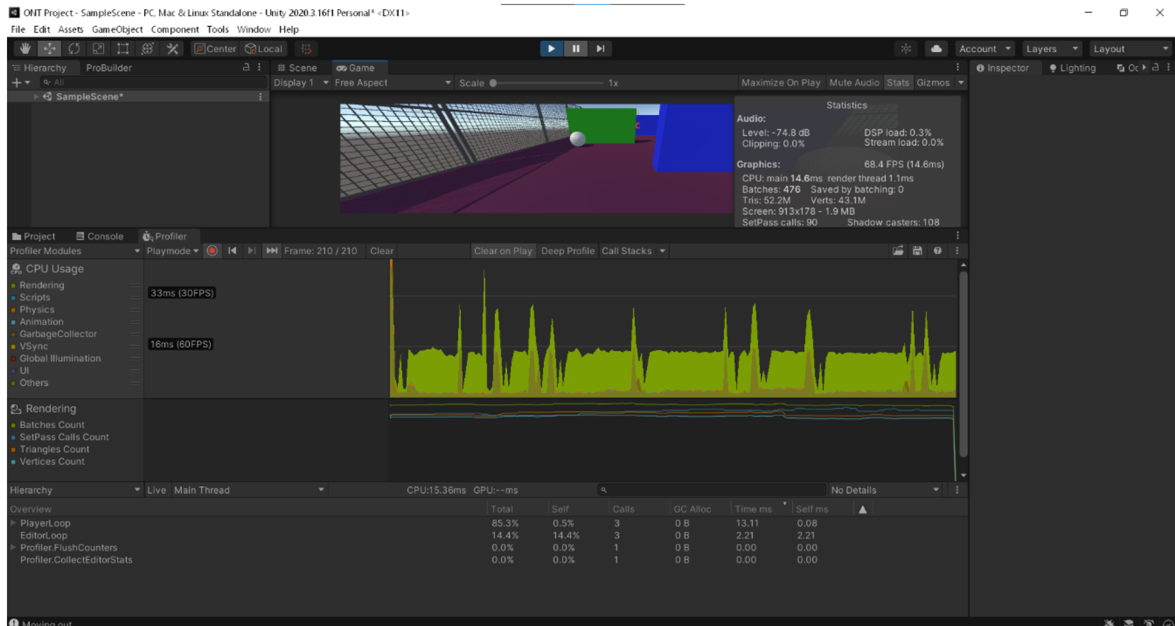
Kuva 42. Light Probe -mesh peliprojektissa

## 8.6 Optimointimenetelmien hyötyjen tarkasteleminen

Optimointimenetelmien hyödyt ja haitat suorituskyvyn kannalta saatiin selville käyttämällä Unityn Profiler-työkalua, jonka avulla suorituskykyä ja siihen vaikuttavia eri osa-alueita voidaan tarkkailla erikseen. Peliprojektissa optimointimenetelmien vaikutusta suorituskykyyn tarkasteltiin yksi optimointimenetelmä kerrallaan silloin, kun se oli järkevää. Esimerkiksi joissain tilanteissa 3D-mallien pintojen määrä kasvoi ainoastaan yhden tai kaksi menetelmää käyttämättä jättämällä niin suureksi, että se olisi vääristänyt joistakin optimointimenetelmistä saatuja tuloksia. Optimointimenetelmien hyötyjen testaaminen toteutettiin pysäyttämällä peli automaattisesti tiettyihin kohtiin pelin aikana, jotta Profilerilla saatuun dataan vaikuttaisi mahdollisimman vähän mikään muu ulkoinen tekijä. Testauksen aikana pelihahmo myös liikkui automaattisesti tiettyä rataa pitkin, joten pelaaja ei voinut vaikuttaa peliin, eikä siten myöskään testaukseen, millään tavalla. Profilerilla kerättiin ensin dataa, kun mikään optimointimenetelmä ei ollut käytössä, ja sen jälkeen tätä dataa verrattiin jokaisesta optimointimenetelmästä kerättyihin tietoihin. Testauksessa otettiin huomioon jokaisen optimointimenetelmän käyttöön ottamiseen kulunut aika sekä niiden käyttämisestä saavutetut hyödyt ja aiheutuneet haitat. Näiden seikkojen pohjalta pääteltiin ja analysoitiin optimointimenetelmien käyttöön ottamisen järkevyys kyseisessä peliprojektissa.

Tärkeimmät neljä asiaa, joita Profilerissa tulee tarkkailla ovat FPS, DrawCalls, Batches ja Triangles. FPS eli Frames Per Second tarkoittaa sitä, kuinka monta kertaa sekunnissa kuva päivittyy. DrawCalls tarkoittaa renderöintipyyntöjä, joita CPU lähettää GPU:lle, jotta se tietäisi mitä renderöidään ja millä tavoin. Batches tarkoittaa ryhmiä draw calleja, jotka CPU kokoaa ja lähettää GPU:lle. Triangles tarkoittaa 3D-objektien pintojen määrää kolmioiksi muutettuna. 3D-objektien pinnat jaetaan pelimoottoreissa kolmioiksi, vaikka 3D-mallintamisessa pyritään käyttämään nelikulmioita. Esimerkiksi 3D-objektin — jossa pintoina on käytetty vain nelikulmioita — Trianglejen määrä on tuplasti suurempi kuin pintojen määrä, sillä nelikulmion muotoiset pinnat jaetaan pelimoottoreissa yleensä kahteen kolmion muotoiseen osaan.

Profilerin moduulit, joiden tarkkailu on tärkeintä visuaalisen optimoinnin kannalta ovat CPU Usage, GPU Usage ja Rendering. Näitä moduuleja tarkkailtiin erityisesti testauksen aikana. Niiden avulla saadaan tietoa CPU:n ja GPU:n kuormituksesta ja esimerkiksi kumpi näistä kahdesta on huonommin optimoitu. Se onnistuu tarkastelemalla ja vertailemalla CPU:n ja GPU:n tarvitsemia millisekunteja kunkin framen renderöimiseen. Moduuleista saadaan selville myös draw callien, batchien ja Trianglejen eli pintojen määrä. FPS saadaan selville Game-ikkunaan avattavasta Statistics-infolaatikosta. Kuvassa 43 nähdään esimerkki Unityn näkymästä, kun Profileria käytettiin peliprojektin suorituskyvyn tarkkailuun.



Kuva 43. Unityn näkymä peliprojektin suorituskykyä tarkkaillessa

Optimoimaton versio peliprojektista on yllättävän raskas, sillä näkymätöntä tekstuuria ja normal mapia käyttämättömissä yksityiskohtaisissa 3D-malleissa on erittäin suuri määrä pintoja. Se saattaa osittain vääristää tuloksia, joten nämä optimointimenetelmät otetaan joissain tapauksissa lisäksi käyttöön muitakin menetelmiä testatessa.

Täysin optimoimattoman peliprojektin ensimmäisellä tarkastuspisteellä FPS-arvoksi saatiin 63, DrawCalls ja Batches määriksi saatiin 488, kolmioiden eli Trianglejen määräksi saatiin 52 miljoonaa ja CPU:n ja GPU:n yhden framen aikana käyttämät millisekunnit olivat keskimäärin 15 ja 14. Toisella tarkastuspisteellä luvut olivat 82, 430, 40 miljoonaa, 12 ja 11 ja kolmannella 140, 287, 20 miljoonaa, 7 ja 6. Arvot paranivat sitä mukaa, mitä lähemmäs maalia pelaaja pääsi, sillä mitä kauemmas kamera etenee, sitä enemmän renderöitäviä objekteja jää kameran taakse, jolloin renderöitävien objektien määrä automaattisesti vähenee.

**Clipping Planes**

Clipping Planes -menetelmän testauksesta saatuja arvoja tarkastelemalla ja vertailemalla optimoimattomaan versioon huomataan, että arvot ovat optimoimattomaan versioon verrattuna parempia kaikilla muilla paitsi viimeisellä tarkastuspisteellä, jolla arvot ovat lähes samat. Koska optimointimenetelmän toimintaperiaate, eli renderöintialueen rajaaminen tietylle alueelle tiedetään, voidaan todeta, että tämä johtuu siitä, että kameran näkökenttä luultavasti ylettyy viimeisellä tarkastuspisteellä pelin loppuun saakka. Tästä syystä optimointimenetelmä ei enää pelin lopussa vaikuta pelin suorituskykyyn. Optimointimenetelmän käyttöön ottaminen on kuitenkin erittäin helppoa ja nopeaa, siitä on pelin alussa merkittävästi hyötyä,

eikä siitä aiheudu minkäänlaista haittavaikutusta visuaaliselta kannalta eikä optimoinnin kannalta, joten sen käyttöön ottaminen on järkevää.

**Potentially Visible Set**

PVS-menetelmän toimintaa tarkkailtiin silloin, kun pelissä oli paljon pintoja ja vähän pintoja. PVS-menetelmää testattiin myös silloin, kun Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä, jotta menetelmän käyttämisestä saataisiin mahdollisimman monipuoliset tulokset. Oletuksena oli, että pelissä on niin vähän pintoja, että PVS-menetelmän käyttäminen peliprojektissa olisi haitallista peliprojektissa, ja että varsinkin CPU:n laskutoimituksiin tarvitsemat millisekunnit nousisivat korkeiksi.

Silloin, kun peliprojektissa oli paljon pintoja, saatiin loistavia tuloksia suorituskyvyn kannalta. PVS vähensi renderöitävien pintojen määrää lähes puolella, DrawCalls- ja Batches-arvot vähenivät puolella, FPS-arvo oli noin kaksi kertaa suurempi ja CPU ja GPU tarvitsivat noin puolet vähemmän millisekunteja käyttöönsä yhden framen aikana. PVS-menetelmän käyttäminen osoittautui erittäin hyödylliseksi peliprojektissa tällaisessa tapauksessa. Menetelmän käyttöä ei huomaa pelin aikana muutoin kuin silloin, kun kamera liikkuu seinän läpi, jolloin sen taakse näkee hetkellisesti. Silloin, kun pelissä on miljoonia pintoja, PVS-menetelmän käyttäminen on hyödyllistä, eikä siitä aiheudu mainittavia haittoja.

Menetelmää testattiin myös silloin, kun Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä ja pintoja oli pelissä vain noin 500 tuhatta, joka on vähän verrattuna alkuperäiseen noin 50 miljoonaan. Oletuksena oli, että CPU olisi käyttänyt enemmän millisekunteja yhden framen aikana CPU:n ajonaikana tekemien näkyvyyslaskelmien takia. Tätä ei yllättäen kuitenkaan tapahtunut, vaan CPU käytti lähes saman verran millisekunteja yhden framen aikana kuin silloin, kun PVS-menetelmä ei ollut käytössä. Myöskään GPU:n käyttämät millisekunnit eivät muuttuneet merkittävästi. Pintojen määrä laski kuitenkin noin puolella, kuten myös DrawCalls- ja Batches-arvot. FPS-arvo ei sen sijaan muuttunut ratkaisevasti paremmaksi. Arvo pysyi joko samana, tai laski hieman, mutta testauksessa arvo heitteli niin paljon, että sen perusteella oli hankala sanoa, kumpi vaihtoehto oli kyseessä. Ehkä arvot eivät huonontuneet siksi, koska näkyvyyslaskelmia eli bakea varten annetut parametrit eivät olleet liian tarkkoja tai siksi, että Occluder-objektit olivat geometrialtaan eli muodoltaan todella yksinkertaisia. Johtopäätöksenä voidaan sanoa, että tällaisessa peliprojektissa voidaan käyttää PVS-menetelmää myös silloin, kun renderöitäviä pintoja on vähän, kunhan Bake-parametrit ovat vähemmän tarkkoja, jolloin CPU:n ei tarvitse käyttää piilotuslaskelmiin liian montaa millisekuntia suorituskyvyn kannalta. FPS-arvo ei parantunut, mutta draw callit vähentyivät puolella. Se tarkoittaa sitä, että laitteen virran käyttö vähenee,

ja se on hyödyllistä erityisesti akkukäyttöisille mobiililaitteille suunnitelluissa peleissä (Unity Technologies 2021u).

**Portal rendering**

Portal rendering -menetelmän testausta varten Occlusion Culling -bake tehtiin oletusparametreja käyttäen, jolloin menetelmä ei ole vain PVS-menetelmän lisäys. PVS-menetelmä toimi samaan aikaan, mutta se oli paljon epätarkempi ja Portal rendering -menetelmä toimi sitä huomattavasti voimakkaammin peliprojektissa.

Menetelmän toiminnan vuoksi sen toiminta oli parempaa tietyillä tarkastuspisteillä. Viimeisellä tarkastuspisteellä toimintaa ei huomannut suorituskyvyssä lainkaan. Ensimmäisellä tarkastuspisteellä kaikki arvot paranivat hieman, arvot olivat noin neljäsosan parempia. Toisella tarkastuspisteellä kaikki arvot olivat huomattavasti parempia, jolloin arvot paranivat yli puolella. Arvot olivat myös PVS-menetelmään verrattuna parempia toisella tarkastuspisteellä, mutta kaikilla muilla tarkastuspisteillä merkittävästi huonompia kuin PVS-menetelmän arvot. Visuaalisesti Portal rendering -menetelmän käyttöä ei huomaisi, jos oviaukkoihin sijoittaisi jonkinlaisen oven, joka avautuisi samaan aikaan kuin portaali, joten menetelmän käyttämisen pystyisi piilottamaan kokonaan. Portal rendering -menetelmän toiminta peliprojektissa on kuitenkin epätasaista. Sen käyttämisestä saattaisi olla hyötyä, jos sitä käyttäisi tarkan PVS-menetelmän kanssa tai jos peliin suunnittelisi alun perin optimoinnin kannalta lisää jonkinlaisia käytäviä tai oviaukkoja, joihin voisi sijoittaa lisää portaaleja. Yksistään Portal rendering -menetelmästä ei ole kuitenkaan mainittavaa hyötyä suorituskyvyn kannalta kyseisessä peliprojektissa.

**Normal map**

Normal map -menetelmää tarkkaillessa huomataan helposti, kuinka suuri vaikutus 3D-objektien pintojen määrällä on pelin suorituskykyyn. Pintojen määrän vähetessä merkittävästi saavutettiin huomattavasti parempi suorituskyky. DrawCalls- ja Batches-arvot pysyivät samoina, mutta muut arvot paranivat merkittävästi. FPS-arvo suureni 90:llä ja Triangles eli pintojen määrä oli noin 2–3 kertaa pienempi jokaisella tarkastuspisteellä. CPU ja GPU käyttivät myös puolet vähemmän millisekunteja. Optimointimenetelmän käyttäminen on erittäin järkevää tällaisessa tapauksessa, sillä pintojen määrän dramaattinen väheneminen vaikuttaa suorituskykyyn erittäin positiivisesti. Menetelmän toteuttamisessa on haasteensa, mutta lopputulos on sen arvoinen suorituskyvyn kannalta. Visuaalisessa mielessä ei myöskään menetetä mitään, sillä yksinkertaiset 3D-mallit näyttävät normal map -tekstuurin kanssa lähes täysin samannäköisiltä kuin yksityiskohtaiset 3D-mallit.

### Mipmaps

Mipmaps-menetelmän käyttämisen vaikutusta suorituskykyyn verrattiin skenaarioon, jossa mikään optimointimenetelmä ei ollut käytössä sekä skenaarioon, jossa Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä. Kummassakaan vertailussa ei huomattu kuitenkaan minkäänlaista selvää vaikutusta suorituskykyyn. Tämä johtuu luultavasti siitä, että peliprojektissa on käytetty vain muutamaa tekstuuria. Mipmaps-optimointimenetelmän vaikutuksen suorituskykyyn huomaisi oletettavasti helpommin silloin, kun eri tekstuureja esiintyisi ruudulla esimerkiksi satoja samanaikaisesti. Visuaaliset virheet tekstuureissa eli Moiré-kuviot näkyivät kuitenkin joissain 3D-objekteissa selvästi kaukaa katsottuna silloin, kun Mipmaps-menetelmä ei ollut käytössä. Menetelmän ollessa käytössä, Moiré-kuvioita ei esiintynyt 3D-objektien pinnalla. Menetelmän käyttöön ottamisessa ei tarvita minkäänlaisia toimenpiteitä eli aikaa ja työtä, jos käytetään Unityn oletuksena generoimia mipmapeja, eikä itsetehtyjä mipmapeja. Menetelmän käyttämättä jättäminen säästäisi levytilaa noin yhden kolmasosan tekstuurien koosta, mutta tämä on melko pieni määrä, sillä tekstuureja on vain muutama. Menetelmän käyttäminen puolestaan korjaa melko vakavia visuaalisia haittoja pelissä, joten sen käyttäminen on suositeltavaa kyseisessä peliprojektissa ainakin visuaaliselta kannalta silloin, kun käytetään Unityn oletuksena generoimia mipmapeja, sillä tällöin menetelmän käyttöön ottaminen on täysin vaivatonta.

### Läpinäkyvät tekstuurit

Läpinäkyvien tekstuurien käyttämisen tarkoitus on vähentää pintojen määrää. Pintojen määrä vähenikin huomattavasti, mutta kuitenkin vähemmän kuin Normal map -menetelmää käytettäessä. DrawCalls- ja Batches-arvot pysyivät samoina, mutta muut arvot paranivat jokaisella tarkastuspisteellä. Triangles-arvo oli huomattavasti pienempi, FPS-arvo oli huomattavasti suurempi ja CPU sekä GPU käyttivät laskemiseen aikaa lähes puolet vähemmän millisekunteja. Läpinäkyvien tekstuurien luominen ilmaisia ohjelmistoja käyttäen on hieman hankalaa ja aikaa vievää, mutta tulokset ovat niin hyviä, että läpinäkyvien tekstuurien käyttäminen on kannattavaa. Läheltä katsottuna voidaan huomata joitain eroja verrattuna verkkoaitaan, jossa reikiin ei ole käytetty läpinäkyvää tekstuuria, mutta erot ovat niin pieniä ja merkityksettömiä, että suorituskyvyn kannalta saavutetut hyödyt voittavat visuaaliset haitat selvästi.

### LOD

LOD-menetelmän tarkoituksena on vähentää kerralla renderöitävien pintojen määrää pelin aikana. Koska 3D-mallit, joita LOD-menetelmän toteuttamiseen käytettiin, sisältävät huomattavasti vähemmän pintoja kuin Normal map -menetelmässä käytetyissä ja läpinäkyviä tekstuureja käyttävissä 3D-malleissa, LOD-menetelmä ei vaikuttanut pelin suorituskykyyn

tarpeeksi näkyvästi, että sitä olisi voinut arvioida. Siksi samalla, kun LOD-menetelmää testattiin, Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä, jotta pintojen määrää saatiin laskettua. Ensin suorituskykyä tarkkailtiin silloin, kun vain Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä. Tästä saatuja tuloksia vertailtiin dataan, joka saatiin tarkkailemalla suorituskykyä silloin, kun lisäksi myös LOD-menetelmä oli otettu käyttöön. Triangles-arvo pieneni eli pintojen määrä väheni noin kolme neljäsosaa jokaisella tarkistuspisteellä. DrawCalls- ja Batches-arvot pienenivät kuitenkin vain hieman ja FPS nousi vain hieman suuremmaksi. CPU ja GPU käyttivät laskutoimituksiin saman ajan kuin ilman LOD-menetelmän käyttämistä. Tarkkailun perusteella tässä peliprojektissa LOD-menetelmän käyttäminen ei ole erityisen järkevää, sillä siitä oli vain hieman hyötyä, mutta sen toteuttaminen on melko työlästä ja aikaa vievää. Jos LOD-malleja olisi pelissä enemmän, menetelmän käyttäminen saattaisi olla järkevämpää. Menetelmästä olisi luultavasti saattanut olla enemmän hyötyä myös silloin, jos LOD-mallin yksityiskohtaisin versio olisi koostunut huomattavasti suuremmasta määrästä pintoja. LOD-menetelmän käyttämisestä on jonkin verran hyötyä, mutta tällaisessa tapauksessa, jossa LOD-malleja on käytössä melko vähän, sen käyttämisestä ei ole erityisen suurta hyötyä.

**Esineen sisältämien komponenttien määrä**

Peliprojektissa käytettiin verkkoaitaa esittäviä 3D-objekteja, jotka koostuivat yhden osan sijasta useammasta osasta eli komponentista, jotta saatiin testattua kuinka paljon esineen sisältämien komponenttien määrä vaikuttaa suorituskykyyn. Oletuksena oli, että suorituskyky on sitä parempi, mitä pienemmästä määrästä komponentteja 3D-objektit koostuvat. DrawCalls- ja Batches-arvot olivat tällä kertaa hieman eri lukuja, mutta molemmat nousivat noin tuhannella, eli kolminkertaistuivat, kun 3D-objektit koostuivat useasta komponentista. FPS-arvo kuitenkin nousi samalla hieman, joka on täysin päinvastoin odotuksia. Tämä testattiin uudelleen useita kertoja ja joka kerralla saatiin sama tulos. Myös CPU:n ja GPU:n tarvitsemat millisekunnit vähenivät, kun esine sisälsi useita eri komponentteja, mikä oli täysin päinvastoin odotuksia. Draw callien määrä kuitenkin moninkertaistui ja mitä enemmän draw calleja pelissä on, sitä enemmän sähköä, eli toisin sanoen myös akkua, laite kuluttaa (Unity Technologies 2021u). Esineen sisältämien komponenttien määrän vähentäminen vähentää draw callien määrää, mutta ilmeisesti vaikuttaa ainakin tässä tapauksessa myös haitallisesti FPS-arvoon, vaikka Unityn dokumentaatiossa väitetään toisin (Unity Technologies 2021u). Huomautuksena myös, että scenessä ei ole käytetty enempää kuin yhtä valonlähdettä. Jos sama, yhdestä komponentista koostunut objekti, olisi ollut usean eri valonlähteen valaisema, se olisi voinut vaikuttaa haitallisesti suorituskykyyn, mutta näin ei ollut.

Draw callien suuri määrä lisäisi virran käyttöä, joten esineen sisältämien komponenttien määrää kannattaisi rajoittaa varsinkin, jos kyseessä on mobiilipeli. Vähentämällä ylimääräisten draw callien määrää voidaan vähentää akun kulutusta ja välttää myös laitteiden ylikuumeneminen pelin aikana. Esineen sisältämien komponenttien määrää on helppo rajoittaa, joten esineen sisältämien komponenttien vähentäminen on siten kannattavaa. FPS-arvo, CPU- ja GPU-ajat huononevat vain hieman, mutta akkuun suuntautuva rasitus vähenee huomattavasti.

**Valaistus**

Valaistus eli Lightmapping- ja Light Probes -menetelmät testattiin samaan aikaan, sillä valaistuksen esilaskeminen laskee valaistuksen aina molemmille menetelmille samaan aikaan. Normal map -menetelmä ja läpinäkyvät tekstuurit olivat myös käytössä, koska valaistuksen esilaskeminen eli baking olisi kestänyt useita vuorokausia, mikäli se olisi tehty myös erittäin paljon pintoja sisältäviä objekteja varten.

Valaistuksen optimointi hieman yllättäen vähensi renderöitävien pintojen määrää (Triangles) yli puoleen pienemmäksi, kuin Realtime-valaistusta käytettäessä. FPS-arvot nousivat huomattavasti ja DrawCalls- ja Batches-arvot pienenivät yli puolella. Ainoastaan CPU- ja GPU-ajat eivät parantuneet merkittävästi, mutta tämä voi johtua siitä, että peliprojekti on liian kevyt Normal map -menetelmän ja läpinäkyvien tekstuurien ollessa poissa käytöstä, joten CPU- ja GPU-aikojen erot olivat liian pieniä, jotta ne olisi voinut huomata. Valaistuksen optimoinnista oli tulosten perusteella erittäin suuri hyöty lähes kaikilla osa-alueilla, mutta yksi valaistuksen optimoinnin haittapuolista on esilaskemisen pitkä kesto. Vaikka kyseessä onkin melko pieni Unityn scene, sen valaistuksen esilaskemisessa kesti oletusasetuksilla tunteja. Jotta suuria scenejä voisi optimoida, esilaskemista varten tarvittaisiin tehokas laitteisto, ettei se kestäisi vuorokausia. Lisäksi Light Probes -menetelmän light probejen asettaminen oli aikaa vievää ja joiltain osin myös hankalaa. Light probejen asettaminen suureen sceneen kestäisi melko kauan. Valaistuksen laatu heikkenee myös hieman, esimerkiksi varjojen laatu joidenkin 3D-mallien päällä on läheltä katsottuna melko huono. Valaistus näyttää toisaalta lähes samalta kuin Realtime-valaistus, ja siitä on suuri hyöty lähes kaikilla optimoinnin osa-alueilla, joten staattisen valaistuksen käyttäminen tässä tapauksessa on lopulta hyödyllistä.

**Kaikki optimointimenetelmät yhteensä**

Optimointimenetelmien hyötyjä tarkasteltiin myös silloin, kun kaikki optimointimenetelmät olivat käytössä yhtä aikaa. Saatuja tuloksia verrattiin täysin optimoimattomasta peliprojektista saatuihin tuloksiin ja tuloksiin, jotka saatiin peliprojektista silloin, kun vain Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä. Näin saatiin vähennettyä

vääristyneiden tulosten mahdollisuutta, joka aiheutuu massiivisesta määrästä pintoja, joita edellä mainitut kaksi optimointimenetelmää poistavat scenestä. Tuloksista saatiin siten monipuolisempia ja todenmukaisempia myös muiden optimointimenetelmien kannalta.

Skenaariossa, jossa mikään optimointimenetelmä ei ollut käytössä (tosin esineiden sisältämien komponenttien määrät oli minimoitu) tuloksiksi saatiin myös jo aiemmin mainitut arvot. Ensimmäisellä tarkastuspisteellä saatiin FPS-arvoksi 63, DrawCalls- ja Batches-arvoiksi 488, kolmioiden eli Trianglejen määriksi 52 miljoonaa ja CPU:n ja GPU:n yhden framen aikana käyttämät millisekunnit olivat keskimäärin 15 ja 14. Toisella tarkastuspisteellä arvoiksi saatiin 82, 430, 40 miljoonaa, 12 ja 11 ja kolmannella 140, 287, 20 miljoonaa, 7 ja 6. Skenaariossa, jossa Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä, saatiin paljon pienempiä arvoja. Tässä skenaariossa ensimmäisellä tarkastuspisteellä saatiin FPS-arvoksi 450, DrawCalls- ja Batches-arvoiksi 477, Triangles-määriksi 495 tuhatta ja CPU:n ja GPU:n yhden framen aikana käyttämät millisekunnit olivat keskimäärin 2 ja 1. Toisella tarkastuspisteellä arvoiksi saatiin 454, 419, 531 tuhatta, 2 ja 1 ja kolmannella 474, 280, 353 tuhatta, 2 ja 0.5. Mikäli esineiden sisältämien komponenttien määrää ei olisi minimoitu, DrawCalls- ja Batches-arvot olisivat olleet noin tuhannella suurempia molemmissa skenaarioissa.

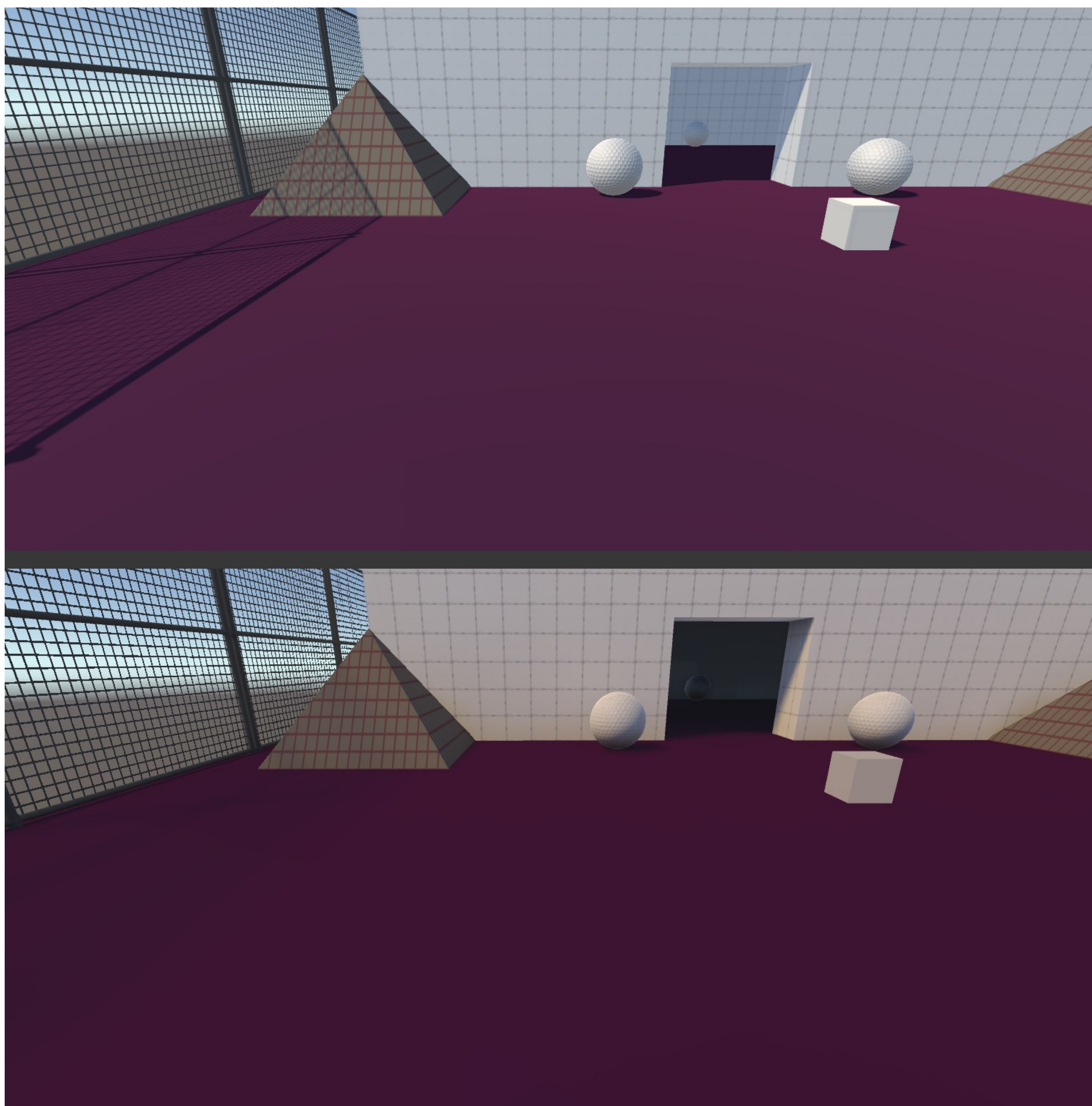
Kaikkien optimointimenetelmien ollessa käytössä, ensimmäisellä tarkastuspisteellä saatiin FPS-arvoksi 541, DrawCalls- ja Batches-arvoiksi 44, Triangles-määriksi 18 tuhatta ja CPU:n ja GPU:n yhden framen aikana käyttämät millisekunnit olivat keskimäärin 2 ja 1. Toisella tarkastuspisteellä arvoiksi saatiin 592, 26, 11 tuhatta, 2 ja 1 ja kolmannella 575, 32, 11 tuhatta, 2 ja 1.

Vertailemalla tuloksia, jotka saatiin kaikkien optimointimenetelmien ollessa käytössä tuloksiin, jotka saatiin silloin, kun mikään optimointimenetelmä ei ollut käytössä, voidaan todeta, että peliprojektin optimoinnista oli suorituskyvyn kannalta erittäin paljon hyötyä. Arvot ovat parempia kaikilla osa-alueilla. Tämä voidaan todeta helpoiten tarkastelemalla taulukkoa 1, johon on lisätty tulokset kaikilta tarkistuspisteiltä peliprojektin optimoimattomasta ja optimoidusta versiosta. Numeroiden perässä olevista kirjaimista ”m” tarkoittaa miljoonaa, ”k” tuhatta ja ”ms” millisekuntia. FPS-arvot ovat 5–10-kertaisia, DrawCalls- ja Batches-arvot 10 kertaa pienempiä, Triangles-määrät eli 3D-objektien kolmioiksi jaettujen pintojen määrät ovat tuhansia kertoja pienemmät ja CPU:n ja GPU:n yhteen frameen käyttämät millisekunnit ovat vähentyneet lähes minimiin. Optimointimenetelmien käyttämisen suurimpina haittapuolina olivat niiden käyttöönottoon kuluva aika ja pienet visuaaliset ongelmat, joita joistain optimointimenetelmistä aiheutui. Kuvasta 44 voidaan todeta, kuinka paljon optimointimenetelmien käyttäminen vaikuttaa pelin ulkonäköön tässä tapauksessa. Kuvan yläosassa

näytetään peli ennen optimointia ja alaosassa optimoinnin jälkeen. Optimoinnista koituvat haitat ovat kuitenkin niin vähäisiä verrattuna hyötyihin, että pelin optimointi on lopulta järkevää, vaikkakin joiltakin osin aikaa vievää.

| | **Optimoimaton** | | | **Optimoitu** | | |
|---|---|---|---|---|---|---|
| **FPS** | 64 | 84 | 142 | 541 | 592 | 575 |
| **Triangles** | 52,2 m | 39,1 m | 19,7 m | 18,1 k | 10,8 k | 10,9 k |
| **DrawCalls** | 1800 | 1500 | 1000 | 44 | 26 | 32 |
| **Batches** | 1780 | 1531 | 1037 | 44 | 26 | 32 |
| **CPU** | 15 ms | 12 ms | 7 ms | 2 ms | 1,5 ms | 2 ms |
| **GPU** | 14 ms | 11 ms | 6 ms | 0,5 ms | 0,5 ms | 0,5 ms |

Taulukko 1. Suorituskyvyn tarkkailusta saadut tulokset eri tarkistuspisteillä optimoimattomassa ja optimoidussa peliprojektissa



Kuva 44. Peliprojekti ennen optimointia ja optimoinnin jälkeen

Kun optimoidusta peliprojektista saatuja tuloksia verrataan tuloksiin, jotka saatiin peliprojektista silloin, kun ainoastaan Normal map -menetelmä ja läpinäkyvät tekstuurit olivat käytössä, huomataan, että tulokset ovat edelleen melko hyviä. Vaikka arvot ovat parantuneet vähemmän, ne ovat silti parantuneet huomattavasti. FPS-arvo nousee noin sadalla yksiköllä suuremmaksi, DrawCalls- ja Batches-arvot ovat tässäkin tapauksessa noin 10 kertaa pienempiä, pintojen kolmioiden määrä laskee kymmeniä kertoja pienemmäksi, mutta CPU:n ja GPU:n tarvitsema aika pysyy samana. Tosiasiassa arvo saattaa olla hieman pienempi, mutta aikaero on niin pieni, että sen erottaminen on Profilerissa on vaikeaa. Vertaillessa saadut arvot ovat yhä niin hyviä, että ainakin helpoiten käyttöönotettavien optimointimenetelmien käyttöön ottaminen on vielä kannattavaa visuaalisten haittojen varjolla. Vaikeammin

käyttöönotettavista optimointimenetelmistä saatetaan kuitenkin loppujen lopuksi hyötyä eniten, joten optimointimenetelmien käyttöön ottamista kannattaa harkita tarkkaan.

# 9 Yhteenveto

Tämän tutkimustyön tarkoituksena oli esitellä, tutkia ja analysoida erilaisia visuaalisia optimointimenetelmiä Unity-pelimoottorilla ja selvittää visuaalisten optimointimenetelmien toimintaperiaatteet sekä niiden vaikutukset pelin suorituskykyyn ja visuaaliseen ilmeeseen. Suorituskyvyn optimoinnin tarkoituksena on saada kasvatettua pelaajien määrää sujuvoittamalla pelin toimintaa, jolloin peli toimii suorituskyvyltään heikommillakin laitteilla sekä vähentää laitteen sähkön — ja siten myös akun — kulutusta. Lisäksi tavoitteena oli helpottaa 3D-pelien suunnittelua optimoinnin kannalta, jotta pelinkehittäjän olisi helpompi valita oikea visuaalinen optimointimenetelmä oikeaan tilanteeseen ja suunnitella peli alusta asti strategisemmin optimoinnin kannalta.

Opinnäytetyössä selvennettiin aluksi, mitä visuaalinen optimointi on, esiteltiin optimoinnin tarkkailuun tarvittava Profiler-työkalu ja tutkittiin sekä esiteltiin erilaisia visuaalisia optimointimenetelmiä. Aluksi esiteltiin piilottamiseen liittyviä optimointimenetelmiä, sitten 3D-malleihin liittyviä optimointimenetelmiä ja lopuksi valaistukseen liittyviä optimointimenetelmiä. Optimointimenetelmien toimintaperiaatteet käytiin läpi ja lisäksi otettiin kantaa siihen, missä tilanteessa optimointimenetelmän tulisi teoriassa toimia parhaiten ja missä huonoiten. Myös optimointimenetelmien Unity-pelimoottorille ominaisista toimintatavoista annettiin tietoa, sillä useita optimointimenetelmiä käytetään eri pelimoottoreissa usein hieman eri tavalla kuin Unity-pelimoottorissa, vaikka niissä onkin sama toimintaperiaate. Lisäksi optimointimenetelmien mahdolliset visuaaliset hyödyt tai haittavaikutukset selvitettiin, ja myös niitä analysoitiin.

Teoriaosuuden jälkeen Unity-pelimoottorissa luotiin yksinkertainen 3D-peli, jossa pelaajahahmo, eli valkoinen kuutio, liikkuu pelikentällä eteenpäin tasaista vauhtia ja väistelee seiniä. Jotta optimointimenetelmien testaamiseen ja tarkkailuun vaikuttaisi mahdollisimman vähän ulkoisia tekijöitä, pelaajahahmo liikkuu optimointimenetelmien testaamisen aikana täysin automaattisesti, ilman pelaajan kontrollia. Jokaisen optimointimenetelmän käyttöön ottaminen Unity-pelimoottorissa selitettiin melko yksityiskohtaisesti ja perusteellisesti.

Lopuksi optimointimenetelmiä testattiin ja tarkkailtiin pysäyttämällä peli automaattisesti kolmessa eri tarkistuspisteessä, jossa Profiler-työkalusta saadut tulokset laitettiin muistiin. Optimointimenetelmiä testattiin yksitellen ja saatuja tuloksia vertailtiin tuloksiin, jotka saatiin silloin, kun mikään optimointimenetelmä ei ollut käytössä. Vertailusta saatujen tulosten perusteella analysoitiin, kannattaako kyseinen optimointimenetelmä ottaa käyttöön peliprojektissa. Analysoinnissa otettiin huomioon myös optimointimenetelmästä aiheutuneet visuaaliset vaikutukset peliin. Joidenkin optimointimenetelmien kohdalla vertailu tehtiin myös toisen kerran, jolloin yritettiin eliminoida mahdolliset saatuihin tuloksiin vaikuttavat ei-toivotut

seikat, joiden epäiltiin vääristävän tulosta. Testaaminen ja vertailu tehtiin lopuksi myös silloin, kun kaikki optimointimenetelmät olivat käytössä samaan aikaan.

Suorituskyvyn tarkkailusta Profiler-työkalulla saatuja tietoja tutkimalla ja analysoimalla saatiin tuloksia, joiden avulla pystytään päättelemään optimointimenetelmien käyttämisen hyödyllisyys peliprojektissa. Kaikista tutkituista optimointimenetelmistä oli jonkinlaista hyötyä peliprojektissa, vaikka joissakin tapauksissa niiden käyttäminen ei vaikuttanut suorituskykyyn näkyvästi. Myös joitain haittavaikutuksia kuitenkin ilmeni.

Optimointimenetelmien toteuttaminen ja testaaminen peliprojektissa onnistui melko hyvin, mutta tuloksista olisi voitu saatu selkeämpiä, jos peliprojekti olisi ollut suurempi ja monipuolisempi. Jos peliprojektissa olisi esimerkiksi käytetty paljon enemmän tekstuureja, myös Mipmaps-menetelmän vaikutus suorituskykyyn olisi voitu huomata. Toisaalta silloin myös esimerkiksi valaistuksen optimointi olisi saattanut tulla hankalaksi pintojen suuresta määrästä johtuvan valaistusdatan laskemisen raskauden vuoksi. CPU- ja GPU-ajoissa oli lisäksi niin pientä vaihtelua, ettei eroja pystynyt näkemään kunnolla Profiler-työkalun avulla. Jos peliprojekti olisi ollut isompi, erot olisivat olleet luultavasti myös suurempia eli helpommin havaittavissa. Tuloksia ei voida yleistää täysin koskemaan kaikkia pelejä, koska on olemassa monia täysin erilaisia pelejä ja tilanteita, joissa esimerkiksi jotkin muut visuaaliset optimointimenetelmät toimisivat paljon paremmin, kuin tässä peliprojektissa hyvin toimineet optimointimenetelmät. Jokaista tilannetta ja peliä varten pitäisikin aina valita erikseen niille parhaiten soveltuvat visuaaliset optimointimenetelmät.

## Lähteet

Arm Limited a. Geometry Best Practices. Viitattu 24.11.2021. Saatavissa https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/game-artist-guides/geometry-best-practices/single-page

Arm Limited b. Level of Detail – LOD. Viitattu 28.11.2021. Saatavissa https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/game-artist-guides/geometry-best-practices/level-of-detail-lod

Arm Limited c. Transparency best practices. Viitattu 04.03.2022. Saatavissa https://developer.arm.com/documentation/102576/0100/Transparency-best-practise

Autodesk. 2017. Viewing and Changing Smoothing. Viitattu 28.03.2022. Saatavissa https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2017/ENU/3DSMax/files/GUID-1DB2E3C2-CE68-4AF7-899E-01B90F7EB320-htm.html

Autodesk 2020. Clipping Planes. Viitattu 01.11.2021. Saatavissa https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Maya-Rendering/files/GUID-D69C23DA-ECFB-4D95-82F5-81118ED41C95-htm.html

Blender Foundation. 2017. Bump & Normal Maps. Viitattu 23.11.2021. Saatavissa https://docs.blender.org/manual/en/2.79/render/blender_render/textures/properties/influence/bump_normal.html

Blender Foundation. 2020a. Decimate Modifier. Viitattu 12.11.2021. Saatavissa https://docs.blender.org/manual/en/2.91/modeling/modifiers/generate/decimate.html

Blender Foundation. 2020b. Normal Map Node. Viitattu 23.11.2021. Saatavissa https://docs.blender.org/manual/en/2.91/render/shader_nodes/vector/normal_map.html

Blender Foundation. 2020c. Shading. Viitattu 28.03.2022. Saatavissa https://docs.blender.org/manual/en/2.91/scene_layout/object/editing/shading.html

computer-graphics.se. Level of Detail LOD. Viitattu 28.11.2021. Saatavissa http://computer-graphics.se/TSBK07-files/pdf/PDF09/10%20LOD.pdf

Courrèges, A. 2016. DOOM (2016) - Graphics Study. Viitattu 25.11.2021. Saatavissa http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/

Epic Games a. Visibility and Occlusion Culling. Viitattu 27.10.2021. Saatavissa https://docs.unrealengine.com/en-US/RenderingAndGraphics/VisibilityCulling/index.html

Epic Games b. Creating and Using LODs. Viitattu 08.11.2021. Saatavissa https://docs.unrealengine.com/en-US/WorkingWithContent/Types/StaticMeshes/HowTo/LODs/index.html

Epic Games c. Texture Properties. Viitattu 03.03.2022. Saatavissa https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Textures/Properties/

Epic Games d. Using Transparency. Viitattu 04.03.2022. Saatavissa https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/HowTo/Transparency/

PTC. Working with clipping features. Viitattu 01.11.2021. Saatavissa https://support.ptc.com/help/creo/ced_modeling/r20.1.1.0/en/index.html#page/ced_modeling/OSDM_Main/Rendering_ClippingPlanesWorking.html

Unity Technologies. 2017. Occlusion Culling. Viitattu 24.10.2021. Saatavissa https://docs.unity3d.com/560/Documentation/Manual/OcclusionCulling.html

Unity Technologies. 2021a. Optimizing graphics performance. Viitattu 22.10.2021. Saatavissa https://docs.unity3d.com/2020.1/Documentation/Manual/OptimizingGraphicsPerformance.html

Unity Technologies. 2021b. Occlusion Culling. Viitattu 23.10.2021. Saatavissa https://docs.unity3d.com/2020.1/Documentation/Manual/OcclusionCulling.html

Unity Technologies. 2021c. Lightmapping. Viitattu 17.11.2021. Saatavissa https://docs.unity3d.com/Manual/Lightmappers.html

Unity Technologies. 2021d. Light Probes. Viitattu 20.11.2021. Saatavissa https://docs.unity3d.com/Manual/LightProbes.html

Unity Technologies. 2021e. Light Probes for moving objects. Viitattu 25.11.2021. Saatavissa https://docs.unity3d.com/Manual/LightProbes-MovingObjects.html

Unity Technologies. 2021f. Understanding the View Frustum. Viitattu 25.11.2021. Saatavissa https://docs.unity3d.com/Manual/UnderstandingFrustum.html

Unity Technologies. 2021g. Occlusion Portals. Viitattu 26.11.2021. Saatavissa https://docs.unity3d.com/Manual/class-OcclusionPortal.html

Unity Technologies. 2021h. Occlusion Areas. Viitattu 26.11.2021. Saatavissa https://docs.unity3d.com/Manual/class-OcclusionArea.html

Unity Technologies. 2021i. Getting started with occlusion culling. Viitattu 28.11.2021. Saatavissa https://docs.unity3d.com/Manual/occlusion-culling-getting-started.html

Unity Technologies. 2021j. Normal map (Bump mapping). Viitattu 31.11.2021. Saatavissa https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html

Unity Technologies. 2021k. Camera. Viitattu 31.11.2021. Saatavissa https://docs.unity3d.com/Manual/class-Camera.html

Unity Technologies. 2021l. Level of Detail (LOD) for meshes. Viitattu 05.12.2021. Saatavissa https://docs.unity3d.com/Manual/LevelOfDetail.html

Unity Technologies. 2021m. LOD Group. Viitattu 08.12.2021. Saatavissa https://docs.unity3d.com/Manual/class-LODGroup.html

Unity Technologies. 2021n. Light Probe Groups. Viitattu 10.12.2021. Saatavissa https://docs.unity3d.com/Manual/class-LightProbeGroup.html

Unity Technologies. 2021o. Profiler overview. Viitattu 21.02.2022. Saatavissa https://docs.unity3d.com/Manual/Profiler.html

Unity Technologies. 2021p. The Profiler window. Viitattu 22.02.2022. Saatavissa https://docs.unity3d.com/Manual/ProfilerWindow.html

Unity Technologies. 2021q. Mipmaps introduction. Viitattu 02.03.2022. Saatavissa https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html

Unity Technologies. 2021r. Creating models for optimal performance. Viitattu 04.03.2022. Saatavissa https://docs.unity3d.com/Manual/ModelingOptimizedCharacters.html

Unity Technologies. 2021s. Transparent Bumped Diffuse. Viitattu 04.03.2022. Saatavissa https://docs.unity3d.com/Manual/shader-TransBumpedDiffuse.html

Unity Technologies. 2021t. Graphics performance fundamentals. Viitattu 25.03.2022. Saatavissa https://docs.unity3d.com/2020.3/Documentation/Manual/OptimizingGraphicsPerformance.html

Unity Technologies. 2021u. Optimizing draw calls. Viitattu 11.05.2022. Saatavissa https://docs.unity3d.com/2020.3/Documentation/Manual/optimizing-draw-calls.html

Unity Technologies. 2021v. Transparent Diffuse. Viitattu 18.06.2022. Saatavissa https://docs.unity3d.com/Manual/shader-TransDiffuse.html

Valve Developer Community a. Visibility Optimization. Viitattu 23.10.2021. Saatavissa https://developer.valvesoftware.com/wiki/Visibility_optimization

Valve Developer Community b. LOD. Viitattu 30.10.2021. Saatavissa https://developer.valvesoftware.com/wiki/LOD

Valve Developer Community c. LOD Models. Viitattu 30.11.2021. Saatavissa https://developer.valvesoftware.com/wiki/LOD_Models

Valve Developer Community d. LOD system. Viitattu 30.10.2021. Saatavissa https://developer.valvesoftware.com/wiki/LOD_system

Valve Developer Community e. Fog Basics. Viitattu 08.11.2021. Saatavissa https://developer.valvesoftware.com/wiki/Fog_Basics

Valve Developer Community f. Lighting. Viitattu 05.12.2021. Saatavissa https://developer.valvesoftware.com/wiki/Lighting

Valve Developer Community g. MIP Mapping. Viitattu 03.03.2022. Saatavissa https://developer.valvesoftware.com/wiki/MIP_Mapping