

Samuli Eskelinen

PELIMAAILMAN GRAFIIKAN OPTI- MOINTI

Opinnäytetyö
Tietojenkäsittely

2020



**Kaakkois-Suomen
ammattikorkeakoulu**

Tekijä	Tutkinto	Aika
Samuli Eskelinen	Tradenomi (AMK)	Toukokuu 2020
Opinnäytetyön nimi Pelimaailman grafiikan optimointi		46 sivua
Toimeksiantaja Xamk virtuaalinen rakentaminen -hanke		
Ohjaaja Jukka Selin		
Tiivistelmä <p>Opinnäytetyön tavoitteena oli selvittää, mitä eri vaiheita kuuluu Unity-pelinkehitysympäristössä tehdyn pelin optimointiin sekä minkälaisia optimointitekniikoita pelinkehitysympäristö tarjoaa. Työn toimeksiantajana toimi Virtuaalinen Rakentaminen -hanke, joka on osa XAMKin TKI - hankkeita. Optimointitekniikoiden testaamiseen sekä opinnäytetyön käytännön osuudessa käytettiin Unity:llä tehtyä pelimaailmaa.</p> <p>Työn teoriaosuudessa käsitellään yleisesti, mitä optimointi on ja mitä sillä yritetään saavuttaa. Lisäksi käsitellään yleisimpiä ongelmakohtia ja pullonkauloja, joihin pelinkehittäjä voi törmätä optimoidessa. Käymme myös vaihe vaiheelta lävitse, kuinka näitä ongelmakohtia ratkotaan Unityn tarjoamalla optimointitekniikoilla.</p> <p>Käytännön osuus keskittyy optimointitekniikoiden hyödyntämiseen käytännön projektissa. Työn vaiheet on esitetty ruutukaappausten ja selitteiden avulla. Optimointitekniikoita lisättiin projektiin kumulatiivisesti ja käytännön osuuden lopussa käymme lävitse optimoinnin onnistumista projektissa.</p> <p>Unityn tarjoamat optimointitekniikat todettiin helppokäyttöisiksi ja monimuotoisiksi työkaluiksi. Kokonaisuudessaan optimointi tuotti hyvää tulosta projektin suorituskyvyn kannalta ja antoi projektille lisää kasvuvaraa tulevaisuutta varten.</p>		
Asiasanat Peligrafiikka, peliohjelmointi, pelisuunnittelu, optimointi, tietokonegrafiikka, Unity		

Author	Degree	Time
Samuli Eskelinen	Bachelor of Business Administration	May 2020
Thesis title		46 pages
3D Graphics optimization		
Commissioned by		
Xamk virtual construction project		
Supervisor		
Jukka Selin		
Abstract		
<p>The objective of the thesis was to discover what different kinds of optimization techniques Unity had to offer, and what steps a game developer must take when optimizing games. The work was commissioned by the virtual construction project which was a part of South-Eastern Finland University of Applied Sciences' RDI project.</p> <p>The work was divided into two parts: theoretical and practical. The theoretical part covered optimization in general; what optimization is and what it sets out to accomplish, usual problems which a game developer might run into when optimizing games and how to solve these problems using Unity's optimization tools.</p> <p>The practical part of the thesis concentrated on using these optimization tools within a practical project. The steps taken during the optimization were presented with screen captures and captions. The optimization techniques added to the practical project were added cumulatively and the results of the optimization were covered in the latter part of the chapter.</p> <p>The optimization tools which Unity offers were found to be very easy to use and diverse tools for developers to add to their arsenal. In its entirety, the optimization of the project yielded good results, and gave the game more room for continuing development.</p>		
Keywords		
Game graphics, game programming, game design, game optimization, optimization, Unity		

SISÄLLYS

1	JOHDANTO	5
2	GRAFIIKAN OPTIMOINNISTA	6
2.1	Renderointikanava	7
2.2	Varjostimet	9
2.3	Yleisimmät ongelmakohdat ja pullonkaulat	11
3	YLEISIMPIÄ OPTIMOINTITEKNIIKOITA	13
3.1	Profilointi	13
3.2	Karsinta	17
3.3	LOD – Level of Detail	23
3.4	Draw call batching	26
4	KÄYTÄNTÖ	29
4.1	Unity-projektin optimointi	30
4.2	Optimoinnin tulokset	37
5	VIRTUAALINEN RAKENTAMINEN -HANKE	39
6	PÄÄTÄNTÖ	42
	LÄHTEET	44

1 JOHDANTO

Opinnäytetyössä perehdytään Unity-pelinkehitysympäristöön ja sen tarjoamiin optimointityökaluihin ja menetelmiin. Työ on tehty toimeksiantona Virtuaalinen rakentaminen -hankkeelle, joka on osa Kaakkois-Suomen ammattikorkeakoulun TKI-hankkeita. Työn tavoitteena on selvittää toimeksiantajalle, minkälaisia optimointityökaluja nykyaikaisessa pelinkehitysympäristössä on tarjolla ja kuinka niitä voidaan hyödyntää.

Opinnäytetyö on jaoteltu teoria- ja käytännön osuuteen. Teoriaosiossa selvitetään aluksi mitä optimointi on ja tutustutaan tarkemmin pelikehitysympäristöjen tarjoamiin renderointi rajapintoihin ja varjostimiin. Jälkimmäisessä luvussa perehdytään yleisimpiin optimointiongelmiiin ja kuinka ne tunnistetaan eri suorituskäytännöihin erikoistuneilla työkaluilla, eli profiloijilla (engl. Profiler). Kappaleessa käydään vaihe vaiheelta lävitse Unityn tarjoamien optimointityökalujen käyttöä, minkälaisissa tilanteissa kyseisiä työkaluja olisi järkevintä käyttää sekä minkälaisia mahdollisia haittavaikutuksia työkalujen liiallisella käytöllä on sovelluksen suorituskyvyn kannalta.

Käytännön osuudessa esittelen vuoden 2018 syksyn peliohjelmointikurssilla luodun, toimeksiantajalle tehdyn pelin, jota käytin teoriaosuudessa käytyjen optimointitekniikoiden testaamiseen. Kappaleessa kartoitetaan aluksi pelin lähtökohta optimoinnin kannalta ja saadut tulokset kirjataan ylös. Kartoituksen tarkoituksena on saada lähtökohta optimoitavalle sovellukselle, jotta voimme myöhemmin vertailla optimoinnin onnistumista. Kappaleessa hyödynnetään teoriaosuudessa käytyjä tekniikoita ja työkaluja. Kappaleen lopussa käsitellään optimoinnin tuloksia ja niiden vaikutusta pelin suorituskykyyn.

Lopussa esittelen tarkemmin työn toimeksiantajan sekä käsittelen opinnäytetyön lähtökohtaa; miten toimeksiantaja hyödyntää opinnäytetyössä saatuja tuloksia. Viimeisessä kappaleessa teen yhteenvedon opinnäytetyössä käydyistä ja opituista asioista sekä pohdin, kuinka hyvin tuloksiin päästiin. Lisäksi käsitelen työn tuloksia tulevaisuutta ajatellen.

Tietokonegrafiikka on grafiikkaa, jota pystytään luomaan tietokoneella. Tietokonegrafiikkaa käytetään kuvien esittämiseen ja muokkaamiseen digitaalisessa muodossa. Tietokonegrafiikalla ja sen kehityksellä on ollut mullistavia vaikutuksia erityyppisiin medioihin, kuten elokuvaan, peliteollisuuteen, mainontaan ja yleisesti graafiseen suunnitteluun. Tietokonegrafiikan osa-alueisiin kuuluvat esimerkiksi sprite-grafiikka, vektorigrafiikka, 3D-mallinnus, käyttöliittymäsuunnittelu, renderointi ja animointi. Nämä tietokonegrafiikan osa-alueet ovat hyvin läheisesti sidoksissa myös pelinkehitykseen (Davis 2011, 7). Laajempaan käsitteeseen tietokonegrafiikasta on sanottu sen olevan ”lähes kaikki tietokoneella mikä ei ole ääntä tai tekstiä” (What is Computer Graphics? 1998).

Videopelit hyödyntävät erityyppisiä tietokonegrafiikan tekniikoita sisällön esittämiseen. Kyseiset tekniikat ovat ajan saatossa kehittyneet suuresti, pääasiassa grafiikkasuorittimien ja muiden laitteistojen kehityksen myötä. Grafiikan suunnittelu ja visuaalisen ilmeen kehittäminen ovat erittäin tärkeä osa videopelisuunnittelua. Videopelijulkaisijoiden mukaan tietokonepelien grafiikka on yksi tärkein osa niiden markkinoinnissa, sillä ensimmäinen asia mitä videopelissä loppukäyttäjä näkee, on sen visuaalinen osa, eli grafiikka (Masuch & Röber 2005, 2). Grafiikan optimoinnissa tulee siis ongelma; kuinka luoda visuaalisesti miellyttävää grafiikkaa siten, että se ei vie liikaa resursseja tietokoneelta.

Pelimoottoreissa optimoinnin yhtenä päämääränä on säästää mahdollisimman paljon tietokoneen resursseja laskennallisesti kuormittavilla hetkillä. Tähän päästään esimerkiksi vähentämällä tarvittavaa laskentaa tärkeissä osissa koodia, jotta laskentatehoa voitaisiin hyödyntää esimerkiksi fysiikkamoottorin reaaliaikaiseen simulaatioon. Mutta ennen kuin optimointia voidaan tehdä, ohjelmoijan tai graafikon on määritettävä, mitkä osat koodista tai grafiikasta luokitellaan tärkeiksi ja kustannustehokkaaksi optimoida.

2 GRAFIIKAN OPTIMOINNISTA

Optimoinnin peruseräpäätteenä on parhaimman vaihtoehdon valitseminen ennalta määritellyistä optimiarvoista tai -määristä. Ennen kuin näihin optimiar-

voihin päästään, joudutaan ohjelmasta ottamaan ylös vertailuarvoja, joita vasten optimoinnin onnistumista mitataan. Huonosti optimoidussa pelissä tavanomaisimpia ongelmakohtia ovat muiden muassa alhainen kehysnopeus (engl. Frames per second, FPS), pitkät latausajat, mahdollinen pelin kaatuilu, lopullisen pelin suuri tiedostokoko sekä nykyisin myös laitteen akun kulutus mobiilipelien suosion myötä. Kaikilla näillä ongelmilla on suuria vaikutuksia käyttäjäkokemukseen, joka on tietotekniikan alalla yksi tärkeimmistä vaikutustekijöistä siihen, että loppukäyttäjä tekee lopullisen ostopäätöksen tai että jatkaako käyttäjä tuotteen käyttämistä.

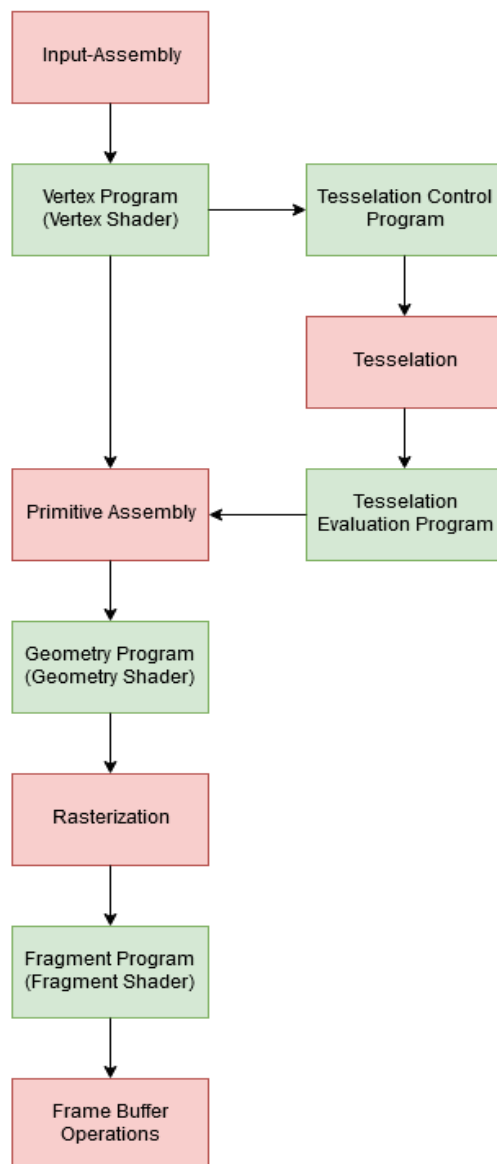
Peliohjelmioijilla on käytössään erittäin kehittyneet työkalut pelien luomiseen. Monimuotoisen ja kehittyneen käyttöliittymänsä ansiosta Unity-pelimoottori on alentanut kynnystä kehittäjille, jotka haluavat aloittaa peliohjelmoinnin. Esimerkiksi vuonna 2012 haastatellun Unity Technologiesin toimitusjohtajan David Helgasonin mukaan Unitya käytti 1,3 miljoonaa kehittäjää. Tämän suosion ansiosta videopeliateollisuudesta on kehittymässä erittäin kilpailuhenkinen ala, jossa lopullisen tuotteen laadulla on hyvin suuri merkitys myynnin ja pelijulkaisijoiden suosion kannalta. Kuluttajat ovat tottuneet saamaan yhä enemmän ja enemmän loppuun hiottuja ja entistä näyttävämpiä pelejä. (Dickinson 2017, 8.)

2.1 Renderointikanava

Renderointikanava on 3D-mallien renderointiin erikoistunut vaiheistettu ohjelma, jota tietokoneen näytönohjain suorittaa. Renderointikanava yleisesti kuvataan kokoonpanolinjaksi, jossa jokainen vaihe linjassa lisää ominaisuuksia aikaisempien päälle. Näytönohjain tarvitsee näitä vaiheita, jotta se pystyy renderoimaan kolmiulotteisen mallin kaksiulotteiselle näytölle. Eri laitteistojen ja ohjelmistojen moninaisuuden takia ei kuitenkaan ole olemassa universaalia renderointikanavaa, joka mahdollistaisi grafiikan piirtymisen halutulla tavalla. Grafiikkasuorittimet hyödyntävät tästä syystä ennalta ohjelmoituja grafiikka rajapintoja (engl. Application Programming Interface, API) kuten Direct3D ja OpenGL.

Nykyaikaisen grafiikkasuorittimen renderointikanavassa on useita eri vaiheita tai ohjelmia. Osa ohjelmista, kuten kärkipisteiden määrittely, kärkipisteiden jälkiprosessointi, primitiivien kasaaminen, rasterointi sekä kehyksen puskurointi (engl.

Frame buffering) operaatiot, ovat kiinteitä osia tässä kanavassa. Nämä kiinteät ohjelmanpätkät ovat yleisesti pakollisia vaiheita, jotta tietokone pystyy esittämään kuvan oikein näyttöpäätteessä. Loppuosat kanavan suoritettavista vaiheista ovat valinnaisia, joista suurin osa on ohjelmoitavissa. Näitä ohjelmanpätkiä kutsutaan varjostimiksi (engl. Shader). (OpenGL Rendering Pipeline An Overview s.a.)



Kuva 1. Nykyaikainen OpenGL-renderointikanava

Videopelin pelimaailmassa saattaa yhden sekunnin aikana tapahtua suuria määriä asioita, kuten esimerkiksi rakennuksen sortuminen, jossa eri osat ra-

kennuksesta putoavat fysiikkamoottorin simuloiman fysiikan mukaisesti. Tällainen tuho luo peliympäristöön suuria määriä tuulen mukana liikkuvaa pölyä. Tällöin renderointikanava voi joissakin tapauksissa suorittaa miljardeja laskutoimituksia sekunnissa, ja kaikki reaaliajassa.

2.2 Varjostimet

Alun perin varjostimia käytettiin nimensä mukaisesti varjostukseen, eli kuvan valon, varjojen ja värien muutokseen. Nykyisin varjostimia käytetään tietokonegrafiikassa myös valotuksessa, erikoisefekteissä ja grafiikan jälkikäsittelyssä (engl. post-processing). Riippuen siitä, mitä toimenpiteitä renderointikanava joutuu tekemään renderoitavalle kappaleelle, saattaa kyseinen kappale joutua käymään neljän eri varjostin vaiheen lävitse, jotta se saadaan piirrettyä oikealla tavalla. Renderointikanavan neljä eri varjostin vaihetta ovat:

Verteksi varjostin (engl. vertex shader) on osa renderointikanavan verteksointi ohjelmaa. Jokainen kolmiulotteinen malli on rakennettu pienemmistä kappaleista, eli perusmuodoista (engl. primitive), jotka vuorostaan saavat muotonsa kärkipisteistä (engl. vertex). Verteksi varjostin käsittelee näiden perusmuotojen yksittäisiä kärkipisteitä. Jokaisessa kärkipisteessä on ennalta määriteltyjä syötearvoja, kuten sijainti 3D-avaruudessa, pinnan normaalit ja tekstuuri koordinaatit. (GLSL Tutorial – Vertex Shader 2015.)

Tesselaatio (engl. Tessellation) on vaihtoehtoinen prosessi verteksi varjostimessa. Ohjelma ottaa ennalta määriteltyjä kärkipisteitä, jotka on saatu verteksi varjostimelta, ja jakaa ne pienempiin perusmuotoihin. Tesselaatio mahdollistaa reaaliaikaisen 3D-mallin pintojen muutoksen, esimerkiksi pienentämällä vaadittavien renderoitavien pintojen kärkipisteiden määrää, kun pelattava hahmo on kaukana kyseisestä mallista. Verteksi varjostin määrittelee, tarvitseeko renderoitavalle mallille tehdä lisämuutoksia ja siirtää kappaleen tesselointiohjelmalle.

Tesselointiohjelmassa on kolme vaihetta; tesselaation-hallintaohjelma (engl. tessellation control program, TCS), tesselaatio ja tesselaation-arviointiohjelma (engl. tessellation evaluation program, TES). Ensimmäisessä vaiheessa (TCS)

ohjelma määrittää kuinka paljon perusmuotoa tarvitsee muokata. Tämän jälkeen ohjelma siirtyy kiinteään vaiheeseen eli tesselointiin. Tesselointi ottaa aikaisemman vaiheen keräämät tiedot perusmuodosta ja tekee sille kyseiset tesselointiin liittyvät muutokset. Suoritus siirtyy muutosten jälkeen tesselaation-arviointiohjelmaan. Mikäli tesselointivaiheessa luodaan uusia perusmuotoja, ohjelma ottaa talteen näiden luotujen perusmuotojen kärkipisteiden koordinaatit ja siirtää ne tesselaation-arviointiohjelmaan. TES-vaihe tarkistaa uusien luotujen perusmuotojen koordinaatit, sekä ensimmäisen vaiheen (TCS) syöteluvut ja laskee näiden avulla uusille kärkipisteille arvot. (OGLdev 2020.)

Geometria varjostin (engl. Geometry shader, GS) käsittelee verteksi varjostimelta tai tesselointiohjelmalta saatuja kärkipisteitä ja muuttaa saadun syötteen erilaisiksi perusmuodoiksi (esimerkiksi pisteiksi tai kolmioiksi). Saatua dataa tulee geometria varjostimelle järjestettynä taulukkona, jonka avulla varjostin rakentaa vaaditun muotoisen perusmuodon. Geometria varjostin on vaihtoehtoinen vaihe renderointikanavassa. Mikäli geometria varjostinta ei käytetä, verteksi varjostin lähettää järjestetyn taulukon kanavan seuraavaan vaiheeseen; *primitiivien kasa*us (engl. primitive assembly). (Geometry Shader 2020.)

Fragmentti varjostin (engl. Fragment shader, FS), joka tunnetaan myös nimellä pikseli varjostin, (engl. pixel shader) on vastuussa rasterointi ohjelmalta saatujen fragmenttien (*kokoelma rasterointi ohjelmalta tulleita tulosteita. Yksi fragmentti vastaa noin yhden pikselin verran piirrettävää dataa*) hallinnasta.

Rasterointi tapahtuu renderointikanavassa primitiivien kasaus vaiheen jälkeen. Rasterointi ohjelma ottaa aikaisemmalta vaiheelta saadun primitiivin ja sen mukana tulevan datan kuten koordinaatit, väri- ja tekstuuriominaisuudet ja läpinäkyvyys ja muuttaa kyseisen datan fragmentti kokoelmaksi.

Fragmentti varjostimen avulla pystytään tekemään monimutkaisia valo-, varjo- ja kohokuvioitehosteita, sekä erilaisia syvyysfektejä, joiden avulla saadaan esimerkiksi kappaleet näyttämään sumeilta niiden ollessa kauempana. FS pystyy myös yksinkertaisimmillaan lähettämään lävitse aikaisemmalta vaiheelta saadun fragmentin väriarvon ilman, että se tekee muutoksia siihen. (GLSL Tutorial – Fragment Shader 2020.)

2.3 Yleisimmät ongelmakohdat ja pullonkaulat

Renderointikanavan varjostimien optimoinnilla voi olla suuria vaikutuksia pelin suorituskykyyn niin positiivisessa kuin negatiivisessakin mielessä. Kun kyseessä on erittäin monimutkainen ja tärkeä osa tietokoneen prosessorin ja näytönohjaimen välisessä keskustelussa, on harvemmin itse renderointikanavan muokkauksella ja ohjelmoinnilla positiivisia vaikutuksia sen suorituskykyyn. Sen sijaan on hyvä opetella oikeaoppisia tapoja varjostimien käytössä itse pelimoottorissa. Esimerkiksi siirtämällä osia laskettavasta prosesseista näytönohjaimelta toisille komponenteille käyttämällä pelimoottoreissa valmiina olevia optimointityökaluja. Tällaisia ovat esimerkiksi erilaiset karsintaan (engl. culling), instansointiin (engl. instancing) ja erien (engl. batching) käsittelyyn erikoistuvat järjestelmät. Seuraavaksi kuvailen yleisimpiä grafiikan piirtämiseen liittyviä pullonkauloja ja ongelma-kohtia.

Piirtopyynnöt

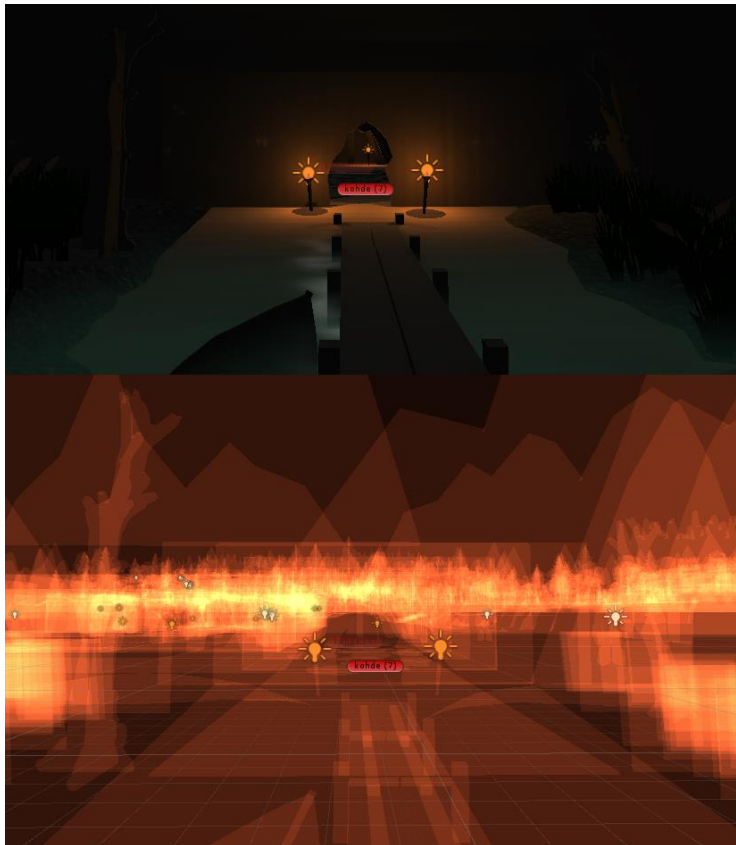
Jotta pelimoottori pystyy piirtämään halutun objektin näytölle, sen täytyy lähettää piirtopyyntö (engl. Draw Call) renderointikanavalle. Piirtopyyntö on kärjistettynä prosessorin käsky näytönohjaimelle, jossa se pyytää näytönohjainta renderoimaan objektin. Nämä pyynnot voivat olla resurssillisesti hyvin kalliita näytönohjaimelle, jos pelimaailmassa on paljon piirrettäviä objekteja.

Ongelmana ei itsessään ole piirtopyyntö, vaan niiden välissä tapahtuva renderointitilan (engl. render state) muutos. Tämä tila muuttuu joka kerta kun piirrettävän objektin tekstuurissa, materiaalissa, varjostimessa, koordinaateissa tapahtuu muutos. Tilan muuttuessa pelimoottori lähettää uuden piirtopyynnön renderointikanavalle. Tätä renderointitilan muutosta ja piirtopyynnön yhdistelmää kutsutaan eräksi (engl. batch). Jotta näytönohjaimelle ei tulisi jokaisesta piirrettävästä objektista tuhansia batcheja käsiteltäväksi, pelimoottoreihin on kehitetty useita työkaluja tämän tilanteen välttämiseksi. Unityssä on käytössä kolme erien käsittelyyn (engl. batching) erikoistunutta tekniikkaa (**Static batching, Dynamic batching ja GPU instancing**). Batchingillä tarkoitetaan useamman samaa renderointitilaa käyttävän objektin ryhmittämistä yksittäiseen batchiin. (Simonov 2017.)

Päällekkäispiirto

Päällekkäispiirto (engl. Overdraw) eli tilanne, jossa renderointikanava piirtää päällekkäisiä pikseleitä, tai saman pikselin useaan kertaan. Tämä tapahtuu, kun pelimaailmassa olevat objektit ovat päällekkäin tai kameran suuntaisesti peräkkäin. Suuri määrä overdrawia lisää kuvan renderointiin kuluvaan aikaa.

Kuvassa 2 havainnollistetaan, kuinka seinän takana olevat objektit renderoituvat huolimatta siitä, ovatko ne näkyvissä vai ei. Tämä johtuu siitä, kuinka Unityn oma renderointi tapahtuu. Unityn renderointikanava renderoi objektit alkaen kauimmaisesta lähimpään, eli lähimmäiset objektit piirtyvät kauempana olevien päälle.



Kuva 2. Unity projekti **Shaded** näkymässä ja **Overdraw** näkymässä

Overdrawin aiheuttaman resurssien kulutuksen vähentämiseen yksi tehokkaimmista optimointikeinoista on turhien tai päällekkäisten objektien piilotus, eli karsinta. Unity-pelimoottorissa on käytössä kaksi eri culling järjestelmää,

joiden tarkoitus on tehostaa pelin suorituskykyä piilottamalla reaaliajassa pelissä olevat mallit, joita pelihahmo/kamera ei sillä hetkellä näe. (Dickinson 2017, 195.)

3 YLEISIMPIÄ OPTIMOINTITEKNIIKOITA

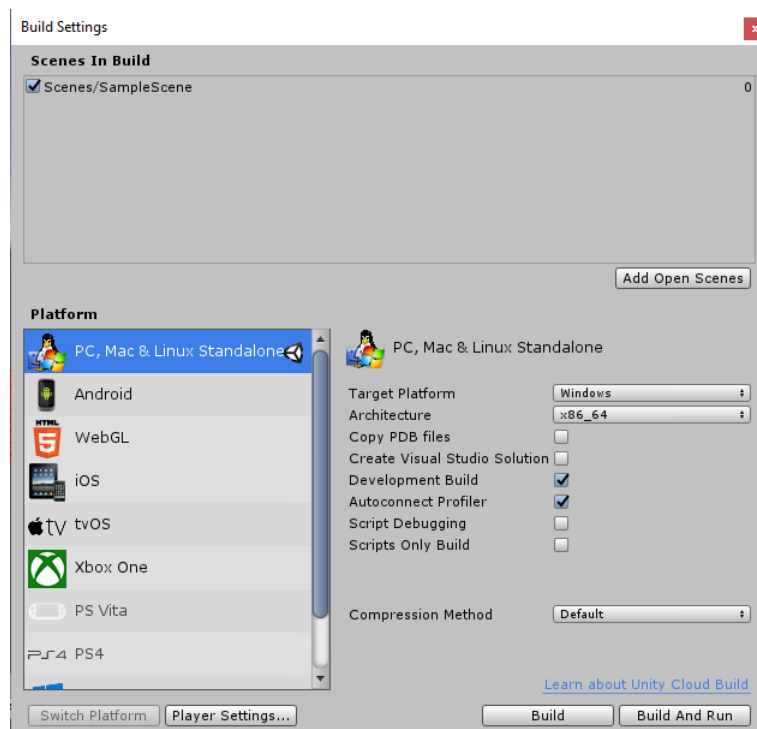
Kun peliä kehitetään, tulisi kehittäjällä olla tuotteen loppukäyttäjä tiedossa jatkuvasti. Sen avulla pystytään selvittämään, minkälaisessa olosuhteissa pelimme tulisi toimia ja minkälaiset laitekohtaiset rajoitukset ovat vastassa (Mobiililaitteet ja konsolit eritoten). Profilointityökalujen avulla pystytään mittaamaan laitteen eri komponenteista tietoa, sekä selvittämään nämä laitekohtaiset rajoitukset. Tämä antaa yleiskäsityksen tai puitteen sille, miten paljon esimerkiksi pelissä saa esiintyä eri animaatioita, miten aitoa fysiikan replikointia voidaan käyttää ja kuinka tarkkoja varjoja näytönohjain saa piirtää. Ennen kuin optimointia voidaan suorittaa, on oltava varmoja siitä, että pelissä on optimointia vaativa ongelma.

Optimointiongelmien ennenaikainen korjaaminen on harvoin kustannustehokasta ja useimmissa tapauksissa turhaa. Pelin eri skenaarioita tai tasoja kannattaa testata useilla eri laitteilla. Pelin testaustavat voivat olla esimerkiksi tason pelaamista osittain lävitse tai pitämällä peliä käynnissä pitempiaikaisesti ja seuraamalla laskeeko pelin suorituskyky merkittävästi. Suositeltavaa on kirjata ylös ne kohdat, joissa suorituskyvyn laskut tapahtuvat. Kun olemme varmoja siitä, että pelissämme on optimointiongelma, on seuraavana vaiheena selvittää mikä sen aiheuttaa. (Dickinson 2017.)

3.1 Profilointi

Nykyiset peliohjelmointiympäristöt tarjoavat käyttäjilleen suorituskykyanalyysiin tarkoitettuja työkaluja (Profiloijia), joiden avulla käyttäjä pystyy pelin kehityksen aikana, sekä julkaisun jälkeen seuraamaan tietokoneen yleisrasitusta (engl. overhead) pelin ollessa käynnissä. Profiloijan avulla voidaan esimerkiksi seurata, kuinka paljon prosessoritehoa 10 hahmon tekoälyn suorittamiseen kuluu yhden kehyksen (engl. frame) aikana. Näiden työkalujen avulla ohjelmasta etsitään ”pullonkauloja” ja pyritään estämään näiden pullonkaulojen syntymistä kehityksen myöhemmässä vaiheessa.

Profiloinnilla tarkoitetaan tietokoneohjelman suorituskyvyn mittaamista ja sen tutkimista, mihin tietokoneella kuluu eniten resursseja ohjelmaa suoritettaessa. Tietokoneen suorituskyvyn mittaamiseen on tarjolla hyvin monia eri testiovelluksia, jotka keräävät tietoa tietokoneen eri komponenteista samalla kun tietokone suorittaa esimerkiksi videon editointia, kuvankäsittelyä, 3D-sovelluksia tai tunnetuimpia toimistosovelluksia. Myös pelejä varten on kehitelty profilointityökaluja, jotka keräävät dataa ajonaikaisesti esimerkiksi tietokoneen prosessorilta, näytönohjaimelta, fysiikkamoottorilta ja renderointikanavalta.



Kuva 3. Profiloinnin käyttöönotto valmiiseen, käännettävään peliin

Pelin profilointiin Unity tarjoaa omaa profilointiohjelmaa "Unity Profiler". Unityn profiloijan avulla pystytään nopeuttamaan pelin suorituskyyä hidastavien pulonkaulojen etsintää sekä varmistamaan, että käytetyt optimointitekniikat tuottavat haluttuja tuloksia. Profiloija tallentaa pelin viimeisimmät 300 kehystä (engl. frame) ja luo käyttäjälle raportin, jossa ilmenee eri komponenttien ja osa-alueiden mittaustulokset. Profiloija mittaa seuraavia osia:

- prosessorin kulutus (fysiikkamoottori, ohjelmakoodit sekä animaatiot)
- näytönohjaimen kulutus (varjot ja grafiikan jälkikäsittely)
- renderointi (erät eli batchit sekä piirtopyynnöt)
- muistin kulutus ja kuinka sovellus varaa muistia ajon aikana
- audio järjestelmän kulutus

- käyttöliittymän (engl. User interface) kulutus
- 2D ja 3D fysiikkamoottorin kulutus
- valaisujärjestelmän (engl. Global Illumination) kulutus.

Mikäli pelin kehitysvaiheessa halutaan profiloida sovelluksen resurssien kulu-
tusta, kehittäjän on otettava huomioon Unity-editorin oma resurssien kulutus.
Unity-editori varaa tilaa muistista ja kuluttaa prosessorin tehoa, jotta editorin
käyttöliittymä pysyy ajan tasalla. Se renderoi ylimääräisiä näkymiä (Scene) ja
käsittelee erinäisiä taustaprosesseja. Projektien kasvaessa nämä resurssien
kulutukset saattavat aiheuttaa hyvin vaihtelevia profilointituloksia sekä mah-
dollisesti sovelluksen kaatumisen liiallisen muistin kulutuksen seurauksena.
(Unity Technologies 2020.)

Profiloija käynnistetään valikosta **Window -> Analysis -> Profiler** tai paina-
malla **Ctrl+7**. Tämä avaa Unityyn uuden ikkunan, jonka pystyy upottamaan
mihin tahansa raahaamalla Profiler välilehteä ja liikuttamalla sitä haluamaan
kohtaan. Profiloijan yläpalkissa on useita eri painikkeita, joiden avulla käyttäjä
pystyy säätämään mitä mittaustuloksia profiloija mittaa.

Add Profiler -alasvetovalikosta käyttäjä pystyy valitsemaan haluamansa mit-
tauskohteet, jotka esitetään profilointityökalun taulukossa. Oletusarvoisesti
profiloija mittaa kaikkia komponentteja näytönohjainta lukuun ottamatta.

Record-painikkeen tulee olla aktiivinen, kun profilointia halutaan suorittaa. Pro-
filoija pystyy mittaamaan ainoastaan dataa silloin, kun *Play Mode* on päällä
Unityn editorissa tai **Profile Editor** painike on aktiivinen.

Deep Profile (suom. Syväprofilointi) -säädin ei ole oletusarvoisesti aktiivinen
sen aiheuttaman yleisrasituksen takia. Syväprofilointia suorittaessa Unity lisää
ohjelman ja käyttäjän luomiin skripteihin mittaamiseen (instrumentaatioon)
vaadittavia lisämetodeja. Nämä metodit pitävät kirjaa kaikista funktiokutsuista
ja tulostavat mittaustulokset profiloijalle.

Syväprofilointi on hyödyllinen silloin, kun halutaan tietoa miten pitkä aika so-
velluksella kestää suorittaa tiettyjä skriptejä. Syväprofiloinnin aiheuttaman

muistin kulutuksen takia sitä suositellaan käyttämään pienemmissä projekteissa, jossa on käytetty yksinkertaista koodia. Jos syväprofilointia käytetään profiloimaan hyvin monimutkaista koodia, profiloija ei saata toimia ollenkaan, tai aiheuttaa sovelluksen kaatumisen.

Profile Editor pakottaa profiloijan mittaamaan Unity-editoria itseään. Tämän avulla pystytään mittaamaan esimerkiksi, kuinka paljon resursseja käyttäjän kehittämät mukautetut editor-skriptit käyttävät.

Editor-alasvetovalikosta käyttäjä pystyy valitsemaan mitä Unityn instanssia mitataan. Unity havaitsee automaattisesti laitteita, jotka ovat samassa verkossa tai liitettynä USB:lla tietokoneeseen ja listaa ne alasvetovalikkoon. Oletusarvoisesti profiloija mittaa editorissa auki olevaa projektia.

Clear on play -valinnan ollessa aktiivisena profiloija tyhjentää mittauksien taulukon joka kerta kun editorissa painetaan play-painiketta.

Clear-painikkeella käyttäjä voi tyhjentää haluttaessaan mittauksien taulukon.

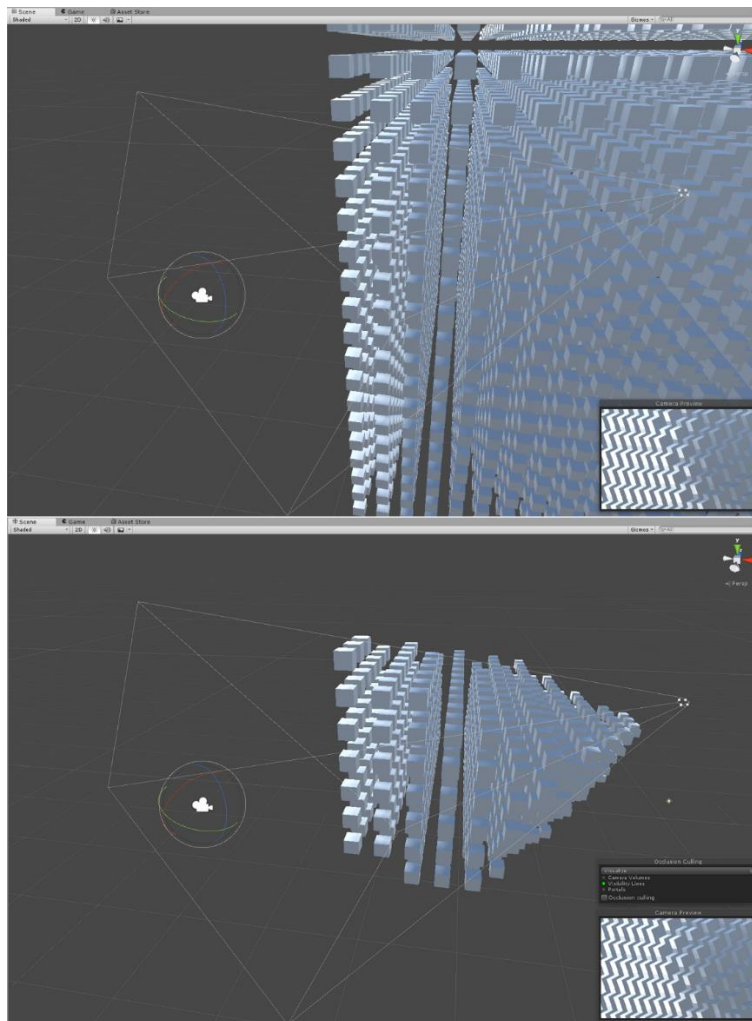
Save-painikkeella profiloija tallentaa mittauksien taulukon myöhempää katselua varten. Mittauksien taulukko tallentuu .data -päätteelliseksi tiedostoksi Unity-projektin *Project*-kansioon.

Load-toiminto lataa Save-painikkeella tallennetun mittauksien taulukon profiloijaan.

Profilointia suorittaessa ei ole resurssillisesti viisasta analysoida joka ikistä koodinpätkää, vaan on suositeltavampaa tehdä pintapuolisia mittauksia sovelluksesta. Pintapuolisella mittauksella tarkoitetaan sen tutkimista, ilmestyykö profiloijan taulukkoon suuria pudotuksia kehysnopeudessa tai varaako sovellus esimerkiksi liikaa tietokoneen muistia. Tärkeimpinä mittareina pelin suorituskyvyn tutkimisessa on, kuinka monta kehystä sovellus pystyy renderoimaan sekunnissa (engl. Frames-per-second, FPS) ja kuinka paljon sovellus kuluttaa muistia. Lisäksi tietoa saadaan tutkimalla prosessorin toimintaa etsimällä profiloijasta kohtia, joissa prosessorin kulutus hyppää hyvin korkeaksi. (Unity Documentation Profiler Window 2020.)

3.2 Karsinta

Frustum culling (engl. näkökenttäkarsinta) on Unityn oletusarvoinen culling-järjestelmä. Järjestelmä toimii karsimalla kaikki ne mallit, jotka jäävät kameran näkökentän ulkopuolelle. Ilman mitään culling-järjestelmää tietokoneen näytönohjain renderoisi kaikki objektit koko skenessä. Mitä enemmän geometriaa näytönohjain joutuu laskemaan, sitä hitaammin se renderoi ne. Unityssä tämä järjestelmä on automaattinen ja oletusarvoisesti aina käytössä.



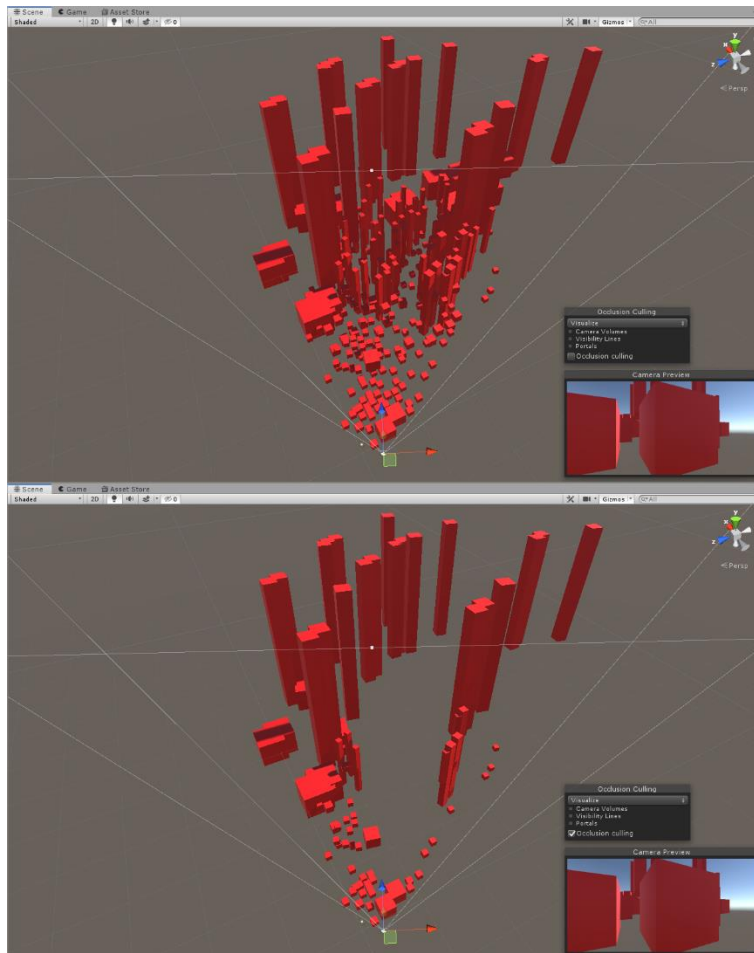
Kuva 4. Unityn Frustum Cullingin havainnollistus

Frustum cullingilla ei kuitenkaan pystytä piilottamaan muiden objektien taakse jääviä objekteja, minkä seurauksena tietokone saattaa piirtää turhaan jopa satoja objekteja. Tämä rasittaa tietokoneen prosessoria ja näytönohjainta.

Kuvassa 8000 kuutiosta rakennettu kuutio Unity-pelimoottorilla, jossa frustum culling -järjestelmällä karsitaan ne kappaleet, jotka jäävät kameran näköken-

tän ulkopuolelle. Koska Unityn renderointikanava piirtää objektit alkaen kauimmaisesta lähimpään, tulee näytönohjaimelle ylimääräisiä, päällekkäisiä piirto-pyyntöjä, vaikka taaimmaiset kappaleet voitaisiin jättää kokonaan piirtämättä. Kyseistä ongelmatilannetta varten Unity tarjoaa toisenlaisen tavan karsia objekteja (Occlusion Culling). (Unity Manual - Camera 2020.)

Occlusion culling -järjestelmä toimii paloittelemalla pelimaailman eri kokoi-siin soluihin ja lennättämällä virtuaalisen kameran pelimaailman tai käyttäjän määrittämän alueen lävitse. Järjestelmä kasaa hierarkkisen listan ympäris-tössä olevista objekteista. Tämän jälkeen pelimoottori piirtää objektit, jotka Occlusion culling -järjestelmä on kasannut listaan. Eli ne, jotka ovat näkyvissä kameralle.

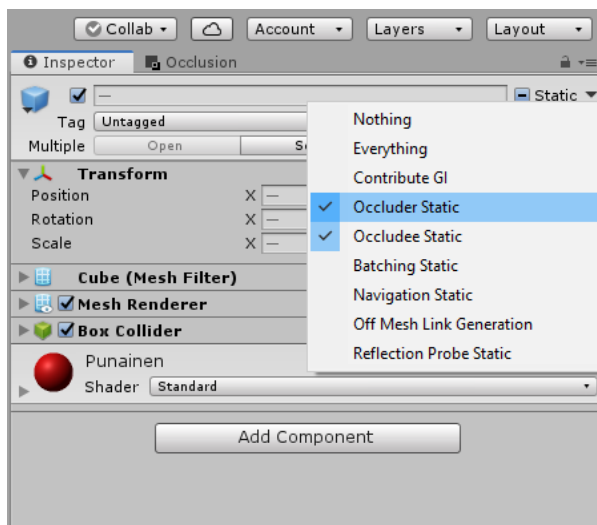


Kuva 5. Unityn Occlusion Culling -järjestelmän havainnollistus

Occlusion culling -järjestelmän käyttöönotto vaatii hieman esivalmisteluja, jotta se saadaan toimimaan halutulla tavalla. Ensimmäiseksi pelin skene kannattaa "paloitella" pienemmiksi kappaleiksi, jotta järjestelmällä ei ole niin suurta työtä

laskea kaikkia mahdollisia kappaleita, joita kamera näkee tai joutuu piilottaamaan. Tämä tapahtuu esimerkiksi lisäämällä pelikenttään suuria esteitä alueiden välille, jolloin suuret objektit peittävät alueet, joita ei haluta piirrettävän kyseisellä hetkellä.

Jotta järjestelmä pystyy suorittamaan karsintaa, täytyy skenen eri objektit merkata käyttötavasta riippuen **Occluder static** - tai **Occludee static** -määreellä. Tämä tapahtuu valitsemalla tarvittavat objektit ja menemällä **Inspector**-ikkunaan, sekä valitsemalla **static**-alasvetovalikosta halutut vaihtoehdot.



Kuva 6. Objektin merkkäminen occlusion cullingia varten

Occludee static vaihtoehto on hyvä käyttää niissä objekteissa, jotka ovat täysin läpinäkyviä tai läpikuultavia. Esimerkkinä ikkuna, jonka lävitse näkyvä objekti on renderoitava, mutta karsitaan, jos ikkunaa suurempi objekti peittää ikkunan ja kameran välisen näkyvyyden.

Oletusarvoisesti occlusion culling -järjestelmän asetusikkuna ei ole Unityssä esillä. Asetusikkuna saadaan esille menemällä Unityn päävalikkoon ja valitsemalla (**Window > Rendering > Occlusion Culling**). Occlusion-välilehti aukeaa Inspector-ikkunan päälle. Occlusion-ikkunassa on kolme eri välilehteä, jotka ovat **Object**, **Bake** ja **Visualization**.

Object-välilehdellä käyttäjä pystyy valitsemaan ne objektit skenestä, joille halutaan tehdä karsintaa. Kun käyttäjä valitsee objektihierarkiasta haluamansa

objektit, occlusion-ikkunaan ilmestyy valintaruudut, joilla käyttäjä pystyy muokkaamaan haluttuja staattisten objektien arvoja riippuen siitä, millä tavoin käyttäjä on merkannut etukäteen objektit (Occluder static tai occludee static). Koska objektit ovat merkattu staattisiksi, näille objekteille ei pystytä tekemään dynaamisesti toimivia karsintaan liittyviä toimenpiteitä.

Vaihtoehtoisesti käyttäjä pystyy rajaamaan itse alueen johon järjestelmä suorittaa karsintaa. Tällaista aluetta kutsutaan okkluusioalueeksi (engl. Occlusion Area). Occlusion area luodaan valitsemalla tyhjä peliobjekti (Create empty) ja valitsemalla ylävalikosta (**Component > Rendering > Occlusion area**) tai menemällä Occlusion välilehden *Object* sivulle ja valitsemalla *Occlusion areas*, sekä painamalla *Create new*. Luotu alue tulee siirtää skenessä kohtaan, jossa se kattaa ne objektit, joille halutaan tehdä karsintaa. Alueen luomisen jälkeen *Is View Volume* -valintaruutu tulee laittaa aktiiviseksi.

Mikäli luotu alue on niin suuri, että siihen jää paljon tyhjää tilaa ja tiedetään, että karsittavat objektit eivät missään vaiheessa tule liikkumaan kyseiseen kohtaan alueesta, kannattaa kehittäjän harkita occlusion area-komponentin paloittelemista pienemmiksi osiksi usealla komponentilla. On hyvä myös ottaa huomioon, kuinka suuren alueen komponentti kattaa, sillä mitä suurempi alue on, sitä enemmän resursseja okkluusio vie tietokoneelta. Okkluusiodata tallennuu tietokoneen kovalevylle, komponentin rakentama datarakenne joudutaan hakemaan RAM-muistista, sekä prosessori laskee joka kehyksellä, mitkä objektit tullaan piirtämään ja mitkä piilotetaan.

Kun halutut karsittavat objektit ovat valittu tai okkluusioalue on saatu rajattua, voidaan siirtyä Bake-välilehdelle. Tällä välilehdellä asetetaan karsittaville objekteille parametreja, minkä perusteella järjestelmä piilottaa objekteja. Välilehdellä on kolme eri säädettävää parametria; **Smallest Occluder**, **Smallest Hole** ja **Backface Threshold**.

Ennen arvojen muuttamista on otettava huomioon, että mitä pienempi luku asetetaan syötteenä, sitä pienempiä soluja järjestelmä joutuu luomaan ja laskemaan suurempia määriä dataa. Tämä lisää prosessorilta vaadittavaa laskemista ja prosessointiaikaa. Lopullinen Okkluusiodata tallennetaan tietokoneen RAM muistiin. Okkluusiodatan viemä RAM muistin määrä on ilmoitettu Bake – välilehden alaosassa tavuina.

Smallest occluder määrittää mikä on pienin objekti, joka piilottaa muita objekteja. Esimerkiksi, jos arvoksi asetetaan yksi (1), järjestelmä suorittaa karsintaa kaikilla objekteilla, jotka ovat korkeampia tai leveämpiä kuin 1 metri ja estävät näkyvyyden muille objekteille.

Smallest hole -arvo määrittelee mikä on pienin tarkoituksella tehty aukko geometriassa, josta pelaajan kamera tulisi nähdä lävitse. Esimerkiksi aidan rakojen lävitse näkyvä rakennus tai metsän takana oleva vuori. Parhaimmat tulokset saadaan asettamalla tämä arvo hieman pienemmäksi kuin kyseiset aukot. Oletusarvoisesti smallest hole -arvo on 25 senttimetriä eli 0,25 olettaen, että peli on mittakaavaltaan ”ihmismäinen” eli pelattava hahmo on noin 160 – 200 senttimetriä pitkä. Smallest hole -arvo on hyvä pitää arvojen 0.05 ja 0.5 välillä.

Backface threshold arvolla säädetään, kuinka herkästi ohjelma karsii culling -järjestelmän tekemästä datarakenteesta objekteja, joiden takatahkot ovat näkyvissä kameralle. Takatahkolla tarkoitetaan tahkon käänteistä puolta, joka objektin piirtämisen nopeuttamiseksi jätetään kokonaan renderoimatta. Arvo, joka syötetään järjestelmään, on prosentuaalinen. Eli arvo 50 tarkoittaa 50 %.

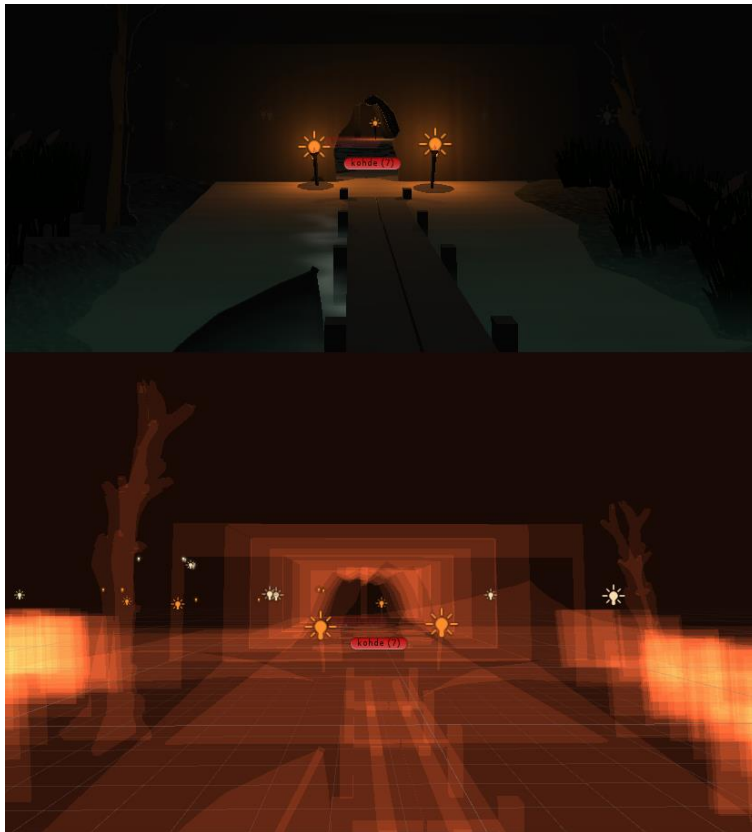
Backface threshold on esiasetettu siten, että järjestelmä ei käy takatahkoja lävitse ja ei poista datarakenteesta yhtäkään objektia riippumatta siitä, onko takatahko näkyvissä. Järjestelmä toimii lähettämällä säteitä occlusion-järjestelmän luomien solujen sisällä olevia objekteja päin ja tarkistaa, kuinka moneen kääntöpuoleiseen kolmioon se osuu. Mikäli saatu arvo ylittää tietyn kynnyksen (käyttäjän syöttämä arvo), järjestelmä karsii objektin datarakenteesta.

Kun halutut parametrit on asetettu käyttäjä voi painaa välilehden alakulmasta *Bake*-painiketta. Järjestelmä paloittelee alueen tai valitut objektit erikokoisiin soluihin (engl. cell). Solujen ei tulisi olla liian pieniä verrattuna karsittaviin objekteihin, eikä objektien niin suuria, että ne vievät usean solun verran tilaa.

Visualization välilehden avulla käyttäjä pystyy tarkastelemaan miten järjestelmä suorittaa okklusiota. Välilehdelle siirtyminen lisää Unityn Skene-näky-

män (engl. Scene view) oikeaan kulmaan *Occlusion Culling* paneelin. Asettamalla paneelin Occlusion Culling -valintaruudun aktiiviseksi järjestelmä simuloi kuinka se suorittaa okkluusiota asetetuilla parametreilla. Okkluusiota suositellaan testaamaan liikuttamalla kameraa eri objektien taakse. Jos parametrit ovat oikein asetettu, kameran ulkopuolella olevat ja muiden objektien taakse jäävät objektit tulisi hävitä näkyvistä scene view -ikkunassa. Jos objektit näkyvät virheellisesti, kuten ilmestyvät yhtäkkiä näkyviin liikuttaessa kameraa tai eivät karsiudu lainkaan, suositellaan parametrien vaihtoa.

Kuvassa havainnollistetaan Unityn Occlusion culling -järjestelmää ja kuinka se oikein asetettuna vähentää turhien piirrettävien objektien ja overdrawin määrää.



Kuva 7. Vähentynyt overdrawin määrä Unity projektissa Occlusion Culling -järjestelmän käytön jälkeen

Occlusion culling -järjestelmän käyttö projektissa kasvattaa sen käyttämää tilaa tietokoneen kovalevyllä, lisää RAM muistin käyttöä sekä prosessorin laskentatehoa. Mutta oikein asetettuna se vähentää suuresti näytönohjaimelle aiheutuvaa rasitusta vähentämällä päällekkäispiirron ja piirtopyyntöjen määrää vähentyneiden piirrettävien objektien seurauksena. (Hougaard 2013.)

3.3 LOD – Level of Detail

Level Of Detail (LOD) -termillä tarkoitetaan 3D-objektin yksityiskohtien tarkkuuden lisäämistä tai vähentämistä reaaliajassa riippuen siitä, kuinka kaukana objekti on kamerasta, tai kuinka paljon prosentuaalisesti objekti vie tilaa kameran näkymässä. Mitä suurempi välimatka kameran ja objektin välillä on, sitä vaikeampaa käyttäjän on erottaa pieniä yksityiskohtia objektista. Tästä syystä on turhaa kuluttaa tietokoneen resursseja ja rasittaa renderointikanavaa pakottamalla sitä piirtämään yksityiskohtaisia objekteja turhaan. Yleisimmin käytetty LOD-järjestelmä on polygoniverkkoihin perustuva LOD (engl. *mesh-based Level of Detail*), joka toimii vaihtamalla yksityiskohtaisen mallin vähemmän yksityiskohtaiseen.

Unityn sisäänrakennettua Level Of Detail (LOD) -järjestelmää käytetään asettamalla useita samaa objektia edustavia objekteja yhden *GameObjectin* lapsiobjekteiksi. Kyseiseen gameobjectiin, eli pääobjektiin (juuriobjekti) tulee olla yhdistettynä **LODGroup** -komponentti. Järjestelmä luo objektin ympärille alueen (engl. bounding box) ja vertaa sitä suhteessa kameran näkökenttään. Mikäli objektia ympäröivä alue vie suuren osan kameran näkökentästä, eli on lähellä kameraa, se asettaa objektille tarkemman mallin. Jos objekti vie pienen osan kameran näkökentästä, se asettaa sille vähemmän yksityiskohtaisen mallin.

LODGroup-komponenttiin kuuluu useita eri parametreja, joita voidaan säätää; **Fade Mode, LOD Group selection bar, Fade Transition Width, Renderers, Recalculate bounds ja -Lightmap Scale.**

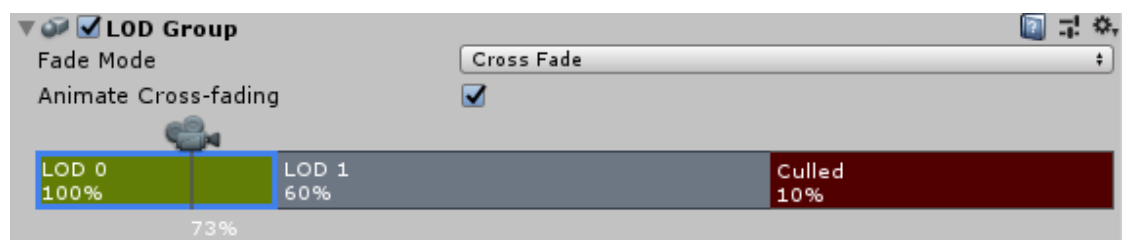
Fade Moden avulla käyttäjä pystyy säätämään, millä perusteella järjestelmä siirtyy käyttämään eri yksityiskohtaista objektia. Oletusarvoisesti fade mode on asetettu *noneksi*, eli siirtymää ei tehdä ja objekti vaihtuu hetkessä eri yksityiskohtaiseen malliin. Jotta välttyään tältä "ilmaantumiselta", voidaan fade mode asettaa ristihäivytykselle (engl. cross fade), jossa objektin eri LOD tasojen mallit tulevat yhtäaikaaisesti esille ja häivyttävät ne keskenään, jolloin ilmaantuminen ei ole niin huomattavaa. Valitsemalla Cross fade parametrin alapuolelle ilmestyy **Animate Cross-fading** valintaruutu. *Animate Cross-fading* arvoa

vaihtamalla voidaan päättää, tapahtuuko ristihiävytys prosentuaalisen arvon, vai ajan perusteella.

Jos käyttäjällä on käytössään Unityn *SpeedTree* liitännäinen ja järjestelmää halutaan käyttää liitännäisen luomien puiden tai kasvillisuuden säätämiseen, LODGroup – komponentti asettaa fade moden automaattisesti *SpeedTree*ksi.

LOD Group selection bar, eli LOD-komponentin eriväriset suorakulmiot, jotka kuvaavat eri LOD tasoa (engl. LOD level) prosentuaalisesti. Valintapalkissa on komponentin luontivaiheessa kolme eri LOD tasoa, jotka ovat *LOD0*, *LOD1* ja *LOD2*. Tasoa voidaan luoda ja poistaa painamalla hiiren oikeaa painiketta valintapalkin kohdalla ja valitsemalla *insert before* tai *delete*. Arvoja voidaan muokata valitsemalla LOD-tason vasen reuna ja raahaamalla sitä. Palkkien prosentuaaliset arvot vastaavat, kuinka monta prosenttia objekti vie kameran näkökentästä pystysuunnassa.

Esimerkiksi: Pelimaailmassa on kappale, joka vie yli 60 % kameran näkökentästä. Objektiin LOD-tasoksi asetetaan LOD0, eli tarkin ja yksityiskohtaisin malli valitaan. Mikäli arvo laskee alle 60 %, asetetaan LOD tasoksi LOD1 ja vähemmän yksityiskohtainen malli tilalle. Jos arvo laskee alle 10 % objekti karsitaan (engl. culled) kokonaan näkyvistä.



Kuva 8. LOD ryhmän parametrit

Fade Transition Width -arvolla pystytään muokkaamaan, missä vaiheessa tiettyä LOD-tasoa järjestelmä aloittaa siirtymän viereiseen tasoon. Esimerkiksi asettamalla arvo 0.5, järjestelmä aloittaa viereisen LOD tasoon asetetun objektiin renderoinnin, kun kameran etsin saavuttaa nykyisen tason puolivälin. Transition width tulee asettaa jokaiselle LOD tasolle erikseen. Arvon liukusäädin tulee aktiiviseksi, kun *Animate Cross-fading* ei ole valittu ja käyttäjällä on jokin LOD taso valittuna.

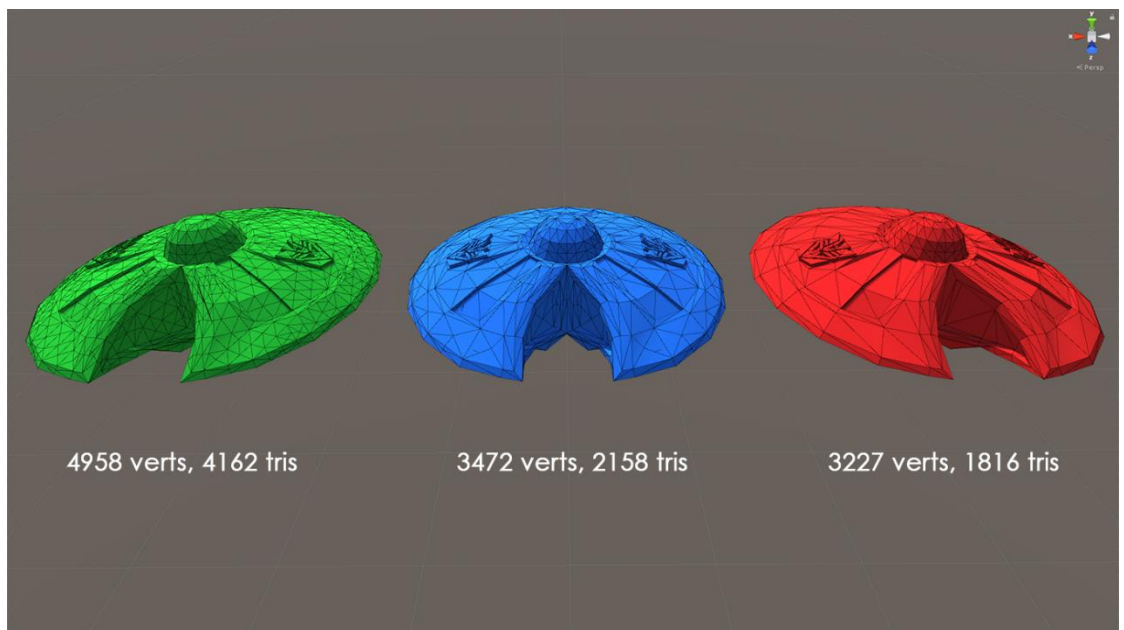


Kuva 9. Unity LOD-järjestelmän Fade Transition width -arvon havainnollistaminen

Renderers eli kyseisen LOD tason renderoitava objekti. Käyttäjä pystyy lisäämään renderoitavia objekteja painamalla *Add*-painiketta. Tämä avaa Unityn Object pickerin, josta käyttäjä voi valita haluamansa objektin. Mikäli objekti, jonka käyttäjä valitsee, ei ole pääobjektin lapsiobjekti, Unity kehottaa liittämään kyseisen objektin lapsiobjektiksi.

Recalculate bounds -painike tulee aktiiviseksi, jos käyttäjä on luonut uuden LOD-tason. Järjestelmä laskee objektia ympäröivän alueen uudelleen.

Oikein määritetty LOD-järjestelmä vähentää renderointikanavassa suoritettavien laskutoimituksien määrää. Yksinkertaisempien mallien piirtäminen tarkoittaa vähempien verteksien piirtämistä varjostimelle, jonka seurauksena renderointikanavan piirtopyyntöjen määrä vähenee.



Kuva 10. Esimerkki malleista, joita voidaan käyttää LOD järjestelmässä

Suurin hyöty LOD-järjestelmästä saadaan käyttämällä sitä skeneissä, joissa on laakeita näkymiä pelimaailmasta ja kamera liikkuu paljon, sillä suuren etäisyyden ja renderöitävien objektien määrän kasvaessa piirrettävien verteksien määrä kasvaa valtavasti. (Catlike Coding 2020.)

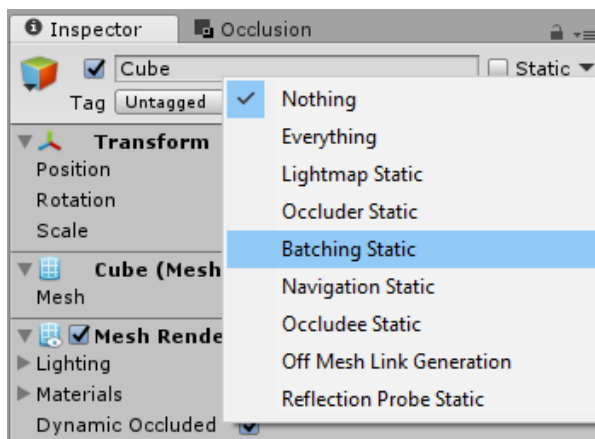
LOD-järjestelmän huonona puolena on sen vaatima kehitysaika. Graafikot joutuvat mallintamaan useita versioita malleista ja kehittäjät joutuvat testaamaan ja muokkaamaan LOD järjestelmän asetuksia, mikäli objektit piirtyvät väärillä tarkkuuksilla kameran liikkeessa. Järjestelmä vie tilaa kovalevyiltä ja RAM-muistia, sekä lisää prosessointiaikaa.

3.4 Draw call batching

Static batching pystyy vähentämään minkä tahansa kokoisen geometrian piirtämiseen tarvittavien piirtopyyntöjen määrää, mutta sen käyttö rajoittuu saman materiaalin omaaviin ja paikallaan oleviin, eli staattisiin objekteihin. Static batching toimii kopioimalla tiedot kaikista *batching static* merkatuista objekteista, yhdistämällä näiden objektien tahkoverkkojen tiedot yhdeksi isoksi datapuskuriksi (engl. data buffer) ja lähettämällä sen renderointikanavalle yhden piirtopyynnön aikana. Yhdistetty data tallennetaan tietokoneen muistiin. Muistin kulutuksen määrä vaihtelee riippuen siitä, kuinka paljon replikaatiota (kahdentamista) kopioitavien mallien välillä tapahtuu. Muistin kulutus perustuu kaavaan: ***jokaisen kopioidun tahkon määrä kerrottuna alkuperäisen tahkon koolla.***

Static batchingiä käyttäessä on otettava huomioon alkuperäisen kopioitavan mallin koko ja itse järjestelmän aiheuttama muistin kulutus. Esimerkiksi jos pelissäsi on 1000 kaksoiskappaleesta koostuva muuri, niin muurin renderointiin static batchingillä kuluu tässä tapauksessa 1000 kertaa enemmän muistia, kun saman kappaleen renderointi ilman static batchingiä. Static batching on siis muistin käytön ja renderointikanavan kulutuksen tasapainottelua.

Jotta static batchingiä voidaan käyttää, käyttäjän tulee merkata halutut objektit staattisiksi. Se tapahtuu menemällä Unityn inspector -ikkunaan ja valitsemalla **Static**-alasvetovalikosta **Batching Static**.



Kuva 11. Objektiin merkkaminen staattiseksi Static Batchingia varten

Unity suorittaa batchaus-prosessin vasta projektin rakennusvaiheessa, eli kun käyttäjä painaa editorissa play painiketta tai lopullisessa projektin rakennusvaiheessa. Mikäli käyttäjä haluaa, että objektit batchataan editorissa, voidaan objekteihin lisätä ohjelmakoodia, joka käyttää *StaticBatchingUtility*-luokkaa luodakseen batchejä reaaliajassa. Lisää tietoa aiheesta Unityn API sivuilla (ks. Unity Documentation 2020). <https://docs.unity3d.com/ScriptReference/StaticBatchingUtility.html>

Dynamic batching

Jotta Unity pystyy suorittamaan Dynamic batchingiä, täytyy käyttäjän asettaa Dynamic Batching aktiiviseksi asetuksista: **Player Settings > Player > Other settings**. Toisin kun sen työtoveri, dynamic batching suorittaa batchingiä liikkuviin objekteihin. Koska dynamic batching on automaattinen toiminto, jää käyttäjälle tehtäväksi "valmistella" objektit batchingiä varten. Unity voi suorittaa dynamic batchingiä objekteille, mikäli niissä täyttyvät tietyntyyiset kriteerit:

- Objektit jakavat keskenään saman materiaalin, eli objektiin on asetettu sama materiaalireferenssi.
- Polygonimallissa saa olla enimmillään 300 verteksiä ja 900 verteksiatribuuttia. (atribuuttien lukumäärää saatetaan kasvattaa tulevaisuudessa)
- Objekteissa **ei voi** hyödyntää materiaali-ilmentymiä (engl. material instance), vaikka materiaalit ovat periaatteessa identtiset.
- Unity projektissa **ei ole** käytössä Multi-pass renderointitekniikka.
- Dynamic batching **ei toimi** peilattuihin GameObjectteihin.

Dynamic batching toimii laskemalla ajonaikaisesti kaikki batchattavien GameObjecttien verteksit suhteessa pelimaailman koordinaatistoon. Resurssien

säästämisen seurauksena Unity on asettanut batchattavien polygonimallien verteksien enimmäismääräksi 300 verteksiä. Mikäli piirrettävä malli ylittää tämän 300 verteksin rajan, batchingiä ei pystytä suorittamaan. Esimerkiksi Unityssä automaattisesti luotu pallo (**GameObject > 3D Object > Sphere**) koostuu 515 verteksistä ja ylittää huomattavasti dynamic batchingiin vaadittavan rajan.

Dynamic batching on hyödyllinen työkalu, kun halutaan piirtää hyvin suuria määriä suhteellisen yksinkertaisia polygonimalleja, jotka ovat lähestulkoon identtisiä ulkonäöltään. Alla on listattuna mahdollisia tilanteita, joissa voitaisiin hyödyntää dynamic batchingiä:

- Pallomeri (otettava huomioon, että pallot eivät ole automaattisesti luotu).
- Metsä, jossa on paljon kiviä, puita ja kasvustoa.
- Rakennus, jossa on paljon huonekaluja ja muita paljon esiintyviä elementtejä, kuten putkia ja kaappeja.

Dynamic batchingiä suositellaan käytettäväksi silloin, kun sen aiheuttama kulutus on pienempi kuin yhden piirtopyynnön tekeminen. Muussa tapauksessa on kustannustehokkaampaa antaa tietokoneen suorittaa piirtopyyntö ilman mitään batchaus tekniikkaa. Yksittäisen piirtopyynnön kulutus riippuu siitä, mitä graafista ohjelmointirajapintaa käytetään. Esimerkiksi konsoleissa ja moderneissa rajapinnoissa yhden piirtopyynnön lähettäminen on usein tehokkaampaa kuin Dynamic batchingin käyttäminen. (Unity Documentation Draw Call Batching 2020.)

GPU Instancing

Geometria instansiointia (engl. GPU instancing tai geometry instancing) voidaan hyödyntää piirtopyyntöjen vähentämiseen. Sen käyttö perustuu saman renderointi tilan hyödyntämiseen renderointikanavan piirtäessä useita identtisiä polygonimalleja. GPU instancingin -toimintatapa ei eroa dynamic batchingistä juuri lainkaan. Ainoana erona on, että se ei ole automatisoitu prosessi ja tarjoaa enemmän vaihtoehtoja kehittäjälle muokattavuudensa ansiosta. GPU instansiointi rajoittuu pelkästään identtisten kopioiden piirtämiseen samasta polygonimallista, mutta jokaiseen instanssiin voidaan asettaa eri parametrit

(eri väri, eri asento, koko), jonka seurauksena kopioidut mallit eroavat toisistaan ja lisäävät variaatiota. Lisäksi GPU instancingissä ei ole rajoitusta sen suhteen, kuinka monta verteksiä piirrettävässä mallissa saa enimmillään olla.

GPU instancing toimii piirrettävän mallin materiaalitasolla ja se otetaan käyttöön avaamalla haluttu materiaali **Inspector**-ikkunaan ja asettamalla **Enable GPU Instancing** -valintaruutu aktiiviseksi. Muokkaamalla varjostimen koodia, voidaan yksittäisiin instansseihin lisätä poikkeuksia ja vaihtelua, kuten vaihtamalla jokaisen kopioidun mallin materiaali eri väriseksi. Lisää tietoa GPU-instanssoinnista voi lukea Unityn dokumentaatiosta (ks. Unity Manual – GPU Instancing 2020).

4 KÄYTÄNTÖ

”OnRailsShooter” on Unityllä suunniteltu ja kehitetty yksinkertainen 3D-pelimaailma, jonka ideana on kaataa erinäisillä esineillä ympäristöstä nousevia pahvisia maalitauluja samalla, kun pelaaja pelaa kelloa vastaan. Hahmo kulkee ”kiskoilla” suorituspisteelle. Maalitaulujen loppuessa hahmo liikkuu automaattisesti seuraavaan suorituspisteeseen. Peli suunniteltiin pelattavaksi kosketuksesta toimivaan interaktiiviseen seinään, jossa pelaaja heittää fyysisesti esineitä, kuten esimerkiksi tennispalloja seinälle projisoituun pelimaailmaan ja interaktiivinen seinä tunnistaa kosketusnäytön tavoin, mihin kohtaan heitetty esine osuu. Peli kehitettiin peliohjelmointikurssin lopputyönä toimeksiantona mikkeliäläiselle IT-palvelua tarjoavalle yritykselle. Pelille ei asetettu minkäänlaisia laitteistokohtaisia vaatimuksia tai rajoitteita



Kuva 12. "OnRailsShooter" -pelistä ruudunkaappauskuva

Pelin kehitysvaiheessa käytössämme oli Unityn versio 2018.2. Sovellus toteutettiin C#-ohjelmointikielellä. Projektissa 3D-mallien suunnitteluun ja toteutuksessa käytettiin 3ds Maxia. 3D-mallien tekstuurit tehtiin Adobe Photoshopilla sekä Illustratorilla. Lisäksi projektissa oli käytössä Unityn Standard Assets -paketti, jolla saimme lisättyä nopeasti vettä, sekä erilaisia partikkeli-efektejä. Versionhallintaan projektissa käytettiin GitHubia.

4.1 Unity-projektin optimointi

Kehitysvaiheessa projektin jäsenillä ei ollut aikaisempaa kokemusta tai tietämystä opinnäytetyössä käydyistä optimointitekniikoista, eikä kyseisiä tekniikoita ole hyödynnetty peliä kehittäessä. Jo pelin kehitysvaiheessa projektissa oli huomattavissa hyvin paljon alentunutta kehysnopeutta, ongelmia skriptien suorituksessa, sekä tekstuurien ja äänien häviämistä peliä testattaessa. Lisäksi ryhmän kokemattomuus versionhallinnan käytöstä aiheutti ongelmia kehitysvaiheen alkupuolella ja hidasti projektin kulkua huomattavasti. Versionhallinnan aiheuttaman ajan kulutuksen myötä myös pelin viimeistely (sekä optimointi) jäi toteuttamatta ja tämän seurauksena pelissä on huomattavissa suorituskykyyn liittyviä ongelmia.

Kartoitus

Ennen optimointia testausalustasta, eli tietokoneesta kartoitetaan sen komponentit ja mitataan pelin aiheuttaman yleisrasituksen kyseisille komponenteille. Lisäksi projektille on asetettava optimiarvot, joihin optimoidessa on päästävä. Komponenttien kulutusta seurataan tietokoneen tehtävienhallintapaneelista *Suorituskyky*-välilehdeltä. Pelin suorituskyvyn mittaamiseksi projektiin on lisätty Unityn Asset Storesta monitorointiin ja vianetsintään erikoistunut lisäosa **Graphy**.

Optimiarvot

Koska alkuperäiselle "OnRailsShooter"-pelille ei koskaan asetettu minkäänlaisia laitteisto- ja ohjelmistovaatimuksia, pelin optimiarvoina käytetään nykykaisten tietokonepelien standardia; eli **60 kehystä sekunnissa** (Garreffa 2015).

Graphy

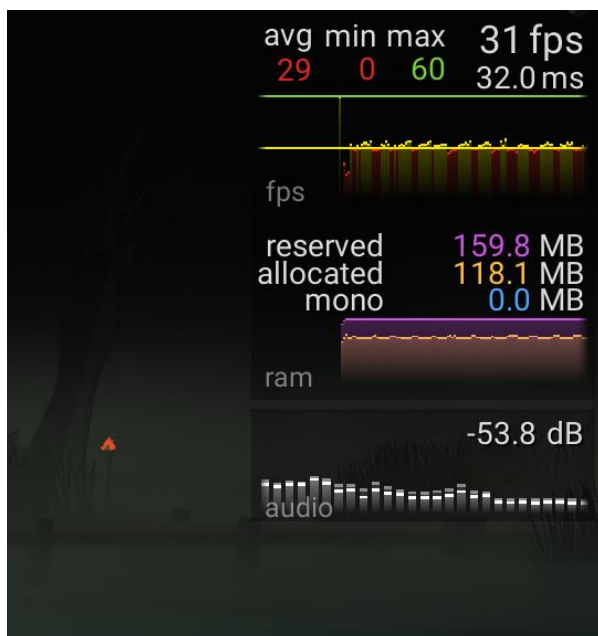
Lisäosan avulla käyttäjä pystyy tutkimaan ajonaikaisesti julkaistun sovelluksen kehysnopeutta sekä muistin kulutusta kuvaajan muodossa. Lisäosan tulostaa tarkat tiedot tietokoneen komponenteista, kuten esimerkiksi mitä grafiikkarajapintaa käytetään, VRAM (Virtual Random Access Memory) määrä ja tietokoneen käyttöjärjestelmä. Alla on listattuna Graphyn mittaustulokset testausalustan komponenteista ja pelin grafiikka-asetuksista.



```
Screen: 1920x1080@60Hz  
Window: 1920x1080@60Hz[96.0dpi]  
Graphics API: Direct3D 11.0 [level 11.1]  
GPU: NVIDIA GeForce GTX 1060 6GB  
VRAM: 6052MB. Max texture size: 16384px. Shader level: 50  
CPU: Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz [4 cores]  
RAM: 8149 MB  
OS: Windows 10 (10.0.0) 64bit [Desktop]
```

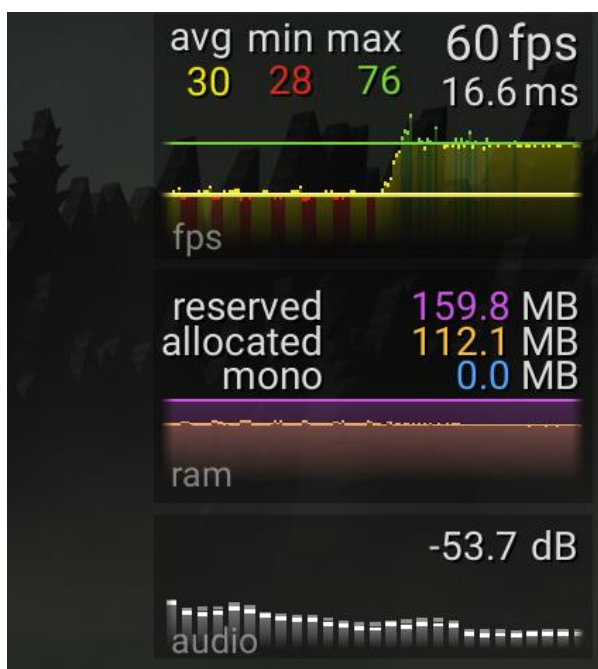
Kuva 13. Profiloijan mittaustulokset testausalustan komponenteista

Pelistä otettiin mittaustulos sen yleisestä kehysnopeudesta ajon aikana Graphy-lisäosan avulla ja tulokset mitattiin ylös. Pelin keskimääräinen kehysnopeus käynnistyessä oli 30 kehystä sekunnissa.



Kuva 14. Graphy-lisäosan mittaustulokset Start-painikkeen painamisen jälkeen

Pelaajan edetessä ensimmäisen suorituspisteen ohitse on pelin kehysnopeudessa huomattavissa suuri muutos. Mittaustuloksia tutkiessa pelin keskimääräinen kehysnopeus on noussut 60 FPS:ään.

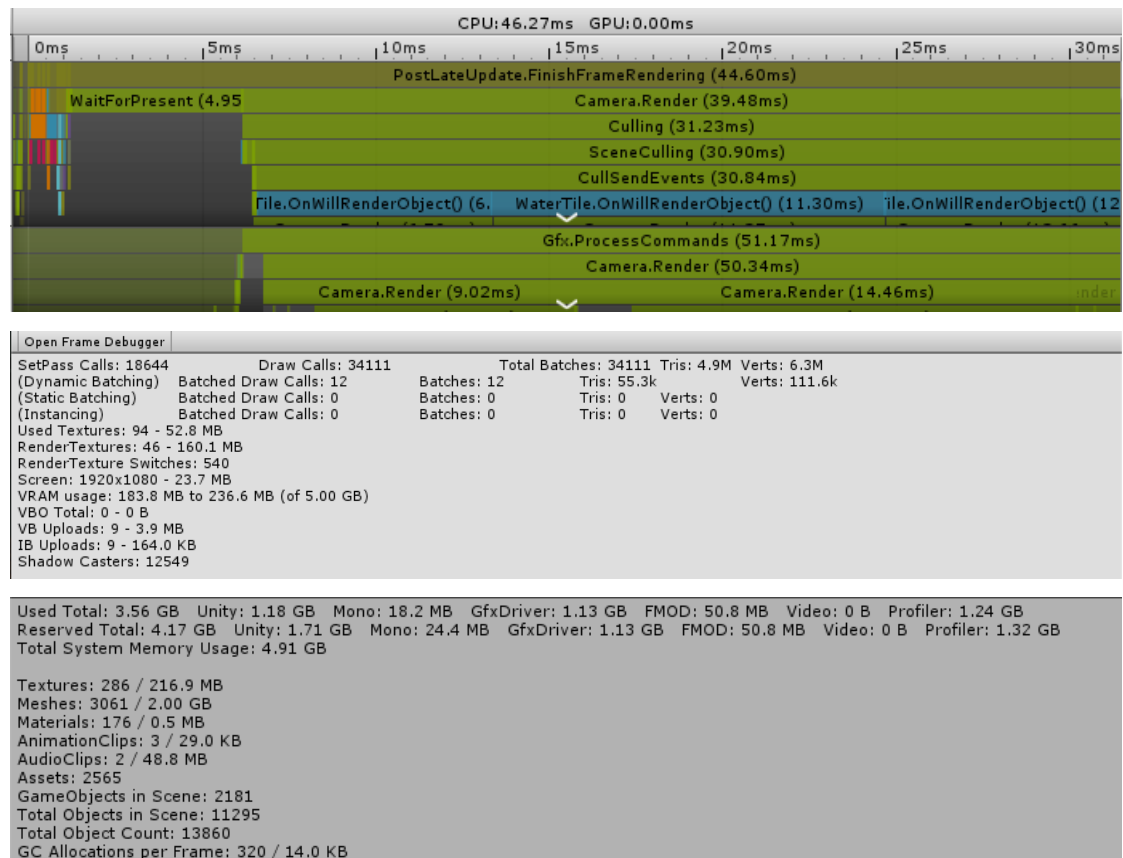


Kuva 15. Graphyn tulokset ensimmäisen suorituspisteen jälkeen

Pelissä on havaittavissa alhaista kehysnopeutta, kun pelattavan hahmon kamera katsoo Z-akselin suuntaisesti, eli samaan suuntaan minne pelikenttä jatkuu.

Unity Profiler

Pelistä tehdään myös mittavampi kartoitus käyttämällä Unityn profiloijaa, joka seuraa pelin aiheuttamaa kuormitusta CPU:lle ja muistille, sekä renderointikavan piirtopyyntöjen määrää. Seuraavaksi listattuna projektista otetut mittaustulokset kolmesta eri profilointimoduulista: **CPU Usage, Rendering ja Memory**.



Kuva 16. Profiloijan mittaustulokset eri profilointimoduuleista

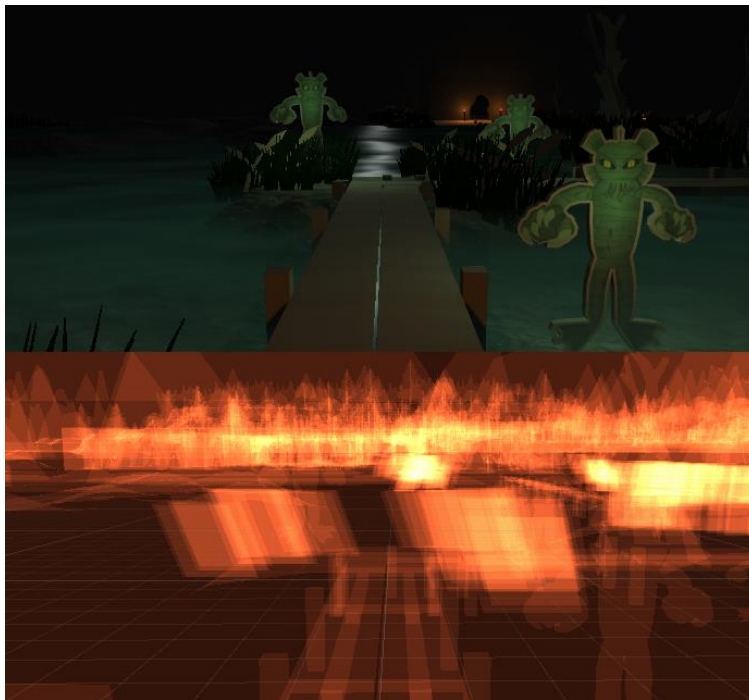
Koska optimointitekniikat toimivat yleisesti vähentämällä näytönohjaimen rasitusta siirtämällä osaa grafiikan laskemisesta tietokoneen prosessorille tai tallentamalla dataa renderoitavista objekteista tietokoneen muistiin, on hyvä pitää kirjaa siitä, kuinka paljon kumpaakin resurssia on hyödynnettävissä. Peli

profiloidaan joka optimointitekniikan käytön jälkeen. Mittaustulosten perusteella päätetään mitä optimointitekniikkaa päädytään käyttämään, mikäli vaihtoehtoja on useampia.

Occlusion Culling

profiloijan antamien, CPU kulutuksen mittaustuloksista käy ilmi, että peli aiheuttaa pullonkaulan prosessorille ja näytönohjaimelle (kuva 16). **PostLateUpdate.FinishFrameRendering** aiheuttaa renderointikanavalle 44 millisekunnin viiveen. "FinishFrameRendering" kärjistettynä tarkoittaa sitä, että prosessori joutuu odottamaan näytönohjainta, että näytönohjain saa piirrettyä nykyisen näkymän näyttöpäätteelle. Eli toisin sanoen, näytönohjaimelle on annettu liian paljon piirrettävää.

Tutkimalla tarkemmin peliä Unityn editorissa ja asettamalla Scene-välilehden piirtoasetukseksi *Overdraw* on havaittavissa, että grafiikkasuoritin joutuu piirtämään hyvin paljon päällekkäisiä tekstuureja. Pelissä tapahtuu siis paljon Overdrawia ja sitä on syytä vähentää.



Kuva 17. Pelissä esiintyvän overdrawin määrä

Overdrawin vähentämiseen projektissa käytetään Unityn Occlusion culling -järjestelmää. Pelissä on järjestelmän käyttöönoton kannalta erittäin hyviä suuria objekteja, jotka peittävät suuria osia pelikentästä ja jaottelevat kentän eri osiin. Occlusion järjestelmän käyttö aloitetaan merkkamalla pelistä staattisiksi objektit, jotka oletetaan peittävän osia pelikentästä, kuten korkeat vuoret ja seinät, joiden läheltä pelaaja kulkee. Projektiin lisätään kaksi okkluusioaluetta ja okkluusiota varten parametreiksi asetettiin:

- **Smallest Occluder: 1**
- **Smallest Hole: 0.2**
- **Backface Threshold: 100**

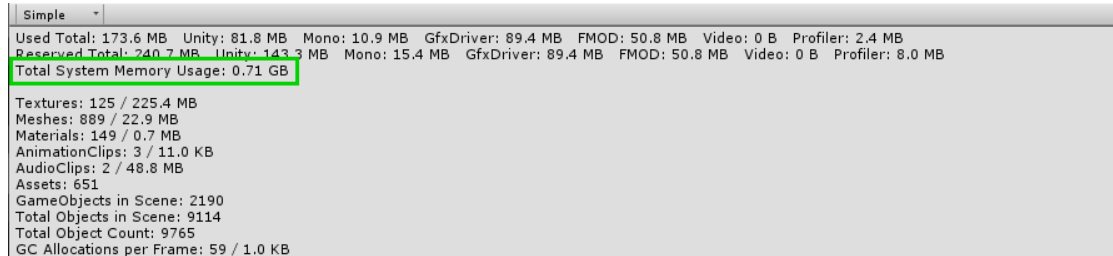
Pelimaailmaa peittävien objektien suuren koon ja testausalustan rajallisen muistin takia Smallest Occluder - ja Smallest Hole -parametrit asetetaan suhteellisen korkeisiin arvoihin, jotta projektin muistin kulutus pysyy vähäisenä.

Batching

Projektin suunnitteluvaiheessa asetimme vaatimukseksi, että OnRailsShooterin pelimaailma tulee koostumaan yksinkertaisista ns. pahvilaatikoista tehdyistä malleista, jotta saamme pidettyä 3D-mallintajan työn vähäisenä ja mallien yksinkertaisuuden ansiosta peli ei liikaa rasittaisi tietokoneen näytönohjainta. Optimoinnin kannalta polygonimallien yksinkertaisuus helpottaa batchingin käyttöä projektissa huomattavasti, sillä suurin osa objekteista, jotka esiintyvät pelimaailmassa, pysyvät dynamic batchingin asettaman 300 verteksin rajan alapuolella. Lisäksi pelissä käytetyt materiaalit toistuvat mallista toiseen, joten projektissa pystytään hyödyntämään GPU Instancingia.

Kartoitusvaiheessa saatujen Rendering profilointialueen tulosten perusteella renderointikanava saa käsiteltäväkseen yli 34000 piirtopyyntöä. Vähentämällä pelimaailman piirtämiseen vaadittavien piirtopyyntöjen määrää batchaamalla ne ja siirtämällä osan grafiikan laskemisesta prosessorille tai tallentamalla osan grafiikasta väliaikaisesti tietokoneen muistiin, vähennämme näytönohjaimelle aiheutuvaa räsitystä. Occlusion culling -järjestelmän käytön jälkeen

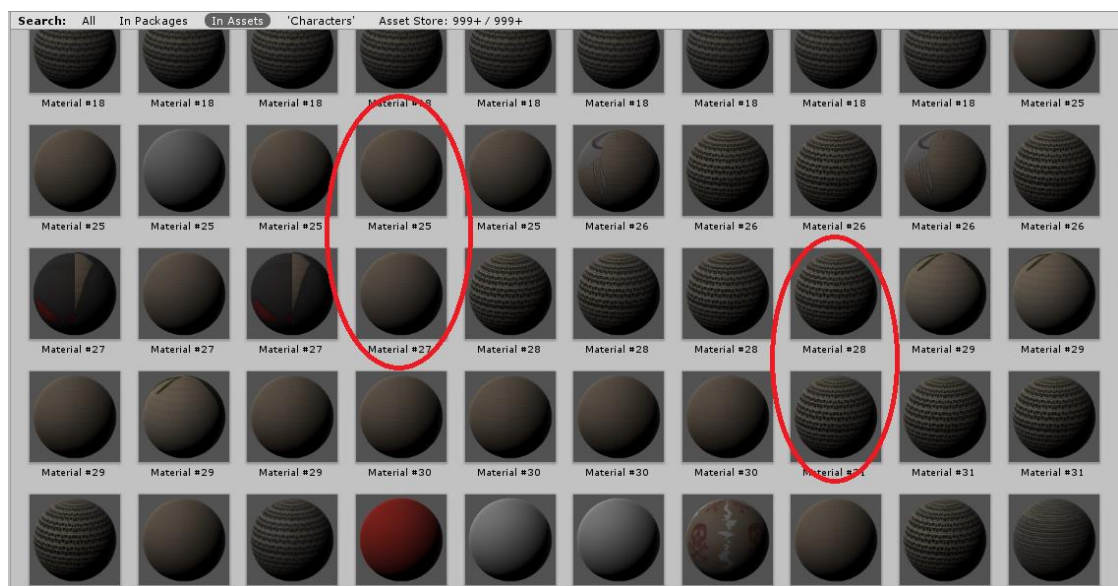
tuloksista voidaan havaita, että pelin muistin kulutus on vähentynyt huomattavasti. Tämän seurauksena projektissa on hyvät puitteet GPU Instancigin käytölle.



Kuva 18. Pelin muistin kulutus occlusion culling -järjestelmän käytön jälkeen

Batchaus aloitetaan merkkamalla staattisiksi objektit, jotka tiedetään pysyvän paikallaan pelin aikana ja käyttävän samaa materiaalia. Näihin objekteihin luokituvat puut, vuoret, luolan eri osat sekä linna ja linnan muurit. Batchingia varten objektit merkataan *batching Static* Unityn Inspector -ikkunan Static-alasvalikosta.

Projektissa esiintyy myös hyvin paljon identtisiä materiaaleja, joita peliobjektit käyttävät. Tämä johtuu projektin kehitysvaiheessa epähuomiossa väärin tuoduista FBX malleista. Joka kerta kun projektiin tuodaan uusi 3D-malli, Unity luo automaattisesti uuden materiaalin, jota malli käyttää.

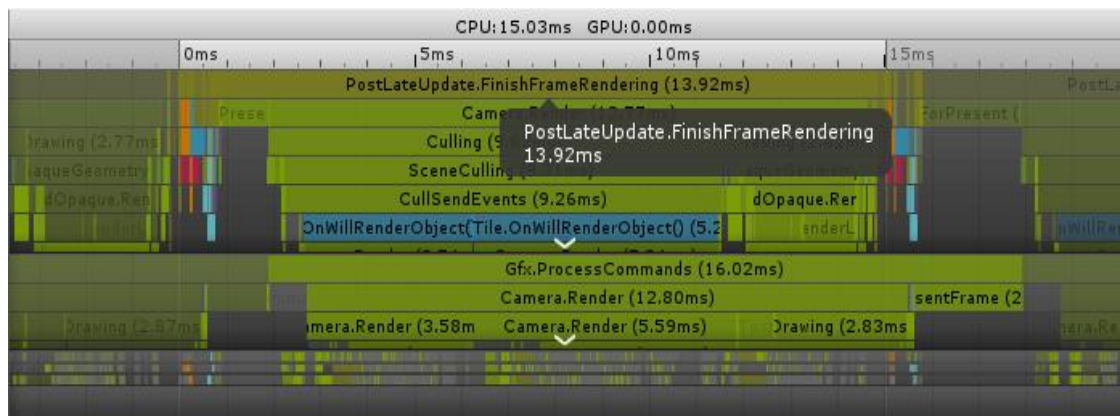


Kuva 19. Projektissa esiintyviä identtisiä materiaaleja

Koska suurin osa projektissa näkyvistä objekteista käyttävät täysin identtistä pahvista materiaalia, on turhaa luoda uusi materiaali jokaiselle erikseen. Ylimääräiset identtiset materiaalit poistetaan ja jäljelle jääneet materiaalit nimitään ”päämateriaaleiksi”. Kyseisille materiaaleille asetetaan GPU Instancing aktiiviseksi inspector-ikkunasta.

4.2 Optimoinnin tulokset

Occlusion culling -järjestelmän käytön jälkeen overdrawin määrä on runsaasti vähentynyt ja tämän seurauksena pelin kehysnopeuteen on tullut huomattava muutos. Järjestelmän varaaman muistin määrä onnistuttiin pitämään alhaisena: 2.2 MB. Tutkimalla Unity profiloijan *CPU Usage* mittaustuloksia aikaisemmin 44 millisekunnin viiveen aiheuttanut FinishFrameRendering on pienentynyt 14 millisekuntiin.



Kuva 20. Occlusion culling -järjestelmän käytön jälkeen prosessorin viive on vähentynyt 30 millisekunnilla

Rendering-moduulin mittaustuloksia tutkimalla, culling-järjestelmän käyttö on vähentänyt piirtopyyntöjen määrän neljäsosaan alkuperäisestä ja tämän seurauksena kehysnopeudessa on huomattavissa 30 FPS parannus.



Kuva 21. Vähentynyt piirtopyyntöjen ja overdrawin määrä occlusion culling-järjestelmän käytön jälkeen, sekä Graphyn mittaustulos pelistä ensimmäisessä suorituspisteessä

Hyödynsin projektissa piirtopyyntöjen erien käsittelyyn Dynamic batchingia, GPU Instancingia, sekä Static Batchingia. Dynamic batchingin käyttöönotto oli batchattavien mallien yksinkertaisuuden ansiosta hyvin helppoa, sekä sen käyttäminen vähensi piirtopyyntöjen määrää 982 kappaleella. GPU Instancingin käyttöönotto vaati aluksi materiaalien yhdistämistä, sekä ylimääräisten materiaalien poistamista. Pelistä karsittiin yli 40 identtistä materiaalia. GPU Instancingin käyttäminen vähensi pelin piirtopyyntöjen määrää yli 2500 kappaleella. Törmäsin kuitenkin ongelmaan joidenkin GameObjectien materiaalien vaihdossa. Vaihtaessani aikaisemmin nimettyä GPU Instancingia hyödyntävää

”päämateriaalia” tietyille GameObjecteille, aiheutti se pelissä joidenkin GameObjectien häviämistä tai väärin skaalautumista. Ratkaisuna merkkasin loput batchaamattomat GameObjectit staattisiksi, jotta niihin voidaan suorittaa Static Batchingia. Static batchingin avulla pelistä vähennettiin 400 piirtopyyntöä. Kaiken kaikkiaan Batchingin käyttö vähensi piirtopyyntöjen määrää 3964 kappaleella.

Open Frame Debugger					
SetPass Calls: 3665 (Dynamic Batching)	Draw Calls: 4887 Batched Draw Calls: 982	Total Batches: 4706 Batches: 198	Tris: 1.1M Verts: 1.3M		
(Static Batching)	Batched Draw Calls: 403	Batches: 166	Tris: 108.5k	Verts: 145.4k	
(Instancing)	Batched Draw Calls: 2579	Batches: 1350	Tris: 55.3k	Verts: 86.0k	
Used Textures: 89 - 51.5 MB			Tris: 172.0k	Verts: 269.3k	
RenderTextures: 38 - 153.8 MB					
RenderTexture Switches: 225					
Screen: 1920x1080 - 23.7 MB					
VRAM usage: 177.6 MB to 229.1 MB (of 5.00 GB)					
VBO Total: 0 - 0 B					
VB Uploads: 201 - 4.2 MB					
IB Uploads: 201 - 0.5 MB					
Shadow Casters: 3576					

Kuva 22. Pelin piirtopyyntöjen määrä Batchaus-työkalujen käytön jälkeen

Optimoinnin yhteenveto

Jo yhden optimointitekniikan lisääminen projektiin vähensi pelin aiheuttamaa viivettä prosessorille ja tämän seurauksena pelin kehysnopeus saatiin nostettua 60 kehykseen sekunnissa. Batching järjestelmien käyttö pelin objekteihin ja materiaaleihin antaa jatkokehityksen kannalta pelille lisää kasvuvaraa vähentyneiden piirtopyyntöjen ansiosta.

Pelissä käytettyjen 3D-mallien yksinkertaisuuden takia tulin siihen tulokseen, että LOD-järjestelmän hyödyntäminen projektissa olisi turhaa johtuen siitä, että 80 % pelistä koostuu malleista, joiden verteksimäärä pysyttelee alle tuhannessa. LOD järjestelmän käyttäminen olisi myös vaatinut hyvin paljon kehitysaikaa, sillä jokaiselle 3D mallille olisi pitänyt tehdä uudet yksityiskohtaisemmat versiot, jota en katsonut kustannustehokkaaksi toteuttaa.

5 VIRTUAALINEN RAKENTAMINEN -HANKE

Virtuaalinen rakentaminen -hanke on osa XAMKin TKI-hankkeita, joka keskittyy kehittämään tekniikoita ja menetelmiä rakennussuunnitelmien pelillistämiseen ja valmiiksi pelillistettyjen rakennussuunnitelmien hyödyntämiseen. Virtuaalinen rakentaminen -hankkeen tavoitteena on kehittää uusia konsepteja rakennusalalle digitaalisten prosessien ja liiketoimintakonseptien avulla. Hanke

hyödyntää nykyaikaisia pelimoottoreita, sekä uusimpia näyttöteknologioita rakennusten suunnitteluun, ylläpitoon ja visualisointiin. Hankkeen yhtenä tavoitteena on kehittää monimuotoinen alusta virtuaalisten rakennusmallien esittämiseen, joka toimii mahdollisimman monella eri laitetypillä, aina puhelimista VR-laseihin. (Xamk Virtuaalinen rakentaminen s.a.)

Unreal Engine

Hankkeen kehitystyökaluna on suurimmalta osalta käytetty Unreal Engineä sen tarjoamien arkkitehtuurilliseen suunnitteluun tarkoitettujen työkalujen, lisätyt todellisuuden lasien tuen ja fotorealististen visualisointien vuoksi. Lisäksi hankkeen pääyhteistyökumppanina toimiva rakennusliike U. Lipsanen on tehnyt omaa kehitystyötään Unreal Enginellä jo entuudestaan. Näiden asioiden vuoksi Unreal Engine valittiin pääkehitystyökaluksi hankkeelle. Lisäksi Xamk:lta löytyi entuudestaan Unreal Engine -osaamista. Hankkeen yhtenä keskeisimpinä tavoitteina on toteuttaa tietokoneella ja VR-laseilla toimiva, moninpleiä tukeva sovellusalusta.

Tietomallit

Virtuaalisen rakentamisen (VirRake) alusta hyödyntää rakennuksen tietomalleja rakennusten esittämiseen virtuaalisesti. Rakennuksen tietomallilla tarkoitetaan digitaalisessa muodossa olevan rakennelman kolmiulotteista esittämistä. Tietomalli sisältää tietoa rakennuksesta, sen prosessien ja rakennusosien eri ominaisuuksista. Tietomallien suunnittelussa rakennusalan toimijat käyttävät erilaisia BIM-ohjelmia (engl. Building Information Model). Suunnitteluohjelmat kuten Rhinoceros, BIM360 ja Revit hyödyntävät matemaattista mallia (NURBS) käyrien ja pintojen luomiseen, sekä esittämiseen. Matemaattisten mallien hyödyntämisen etuna on sen tarjoama tarkkuus ja muokattavuus. Tästä syystä niitä käytetään yleisesti teollisuuteen tarkoitetussa tuotesuunnittelussa. Näitä malleja yleisesti kutsutaan CAD malleiksi (BIMobject 2019).

Nykyaikaiset pelimoottorit erikoistuvat lähinnä polygoniverkkoihin perustuvan geometrian piirtämiseen ja harvemmin tukevat CAD tiedostomuotoja sellaiseen. Tämän ongelmatilanteen ratkaisemiseksi pelimoottoreihin on lisätty erinäisiä lisäosia CAD -mallien tuontiin projekteihin.

Datasmith

Tietomallien lisääminen Unreal-projekteihin tapahtuu yleensä Datasmith-lisäosan avulla. Datasmith kääntää saadun CAD-mallin FBX-malliksi. Lisäosa hyödyntää renderointikanavan tesselointivaihetta pintojen muuttamisessa polygoniverkoiksi. Tesselointi ei kuitenkaan pysty muuttamaan mallia polygoniverkoksi siten, että se vastaa täysin alkuperäistä matemaattista pintaa, johtuen matemaattisten pintojen skaalautuvuudesta. Tesselaatio kuitenkin yrittää approksimoida matemaattista mallia lisäämällä muunnettuun malliin polygoneja. Mitä enemmän mallissa on polygoneja, sitä realistisempi malli on visuaalisesti, mutta polygonimäärän kasvaessa tietokoneen näytönohjaimelle aiheutuu lisää raskautta. Vähentämällä polygonimallien tarkkuutta muuntamalla Datasmithin parametreja voidaan vähentää aikaa, joka näytönohjaimelta kuluu mallin renderointiin. Tämä saattaa aiheuttaa ei-toivottuja muutoksia, kuten että malli näyttää kulmikkaalta tai rosoiselta. (Unreal Engine Datasmith 2020.)

Ongelmakohta

Hankkeessa törmätään erilaisiin ongelmatilanteisiin, kun VirRake-alustalle lisätään FBX muotoon käännettyjä rakennusalan toimijoilta saatuja rakennusmalleja. Jokaista alustalle lisättyä mallia joudutaan muokkaamaan runsaasti ja pienentämään niiden tarkkuutta, jotta ne saadaan näkymään halutulla tavalla ja ne eivät vie liikaa resursseja laitteelta. Ilman minkäänlaisia muutoksia käännetyt rakennusmallit voivat pahimmassa tapauksessa ylittää ideaalisen polygonimäärän skenessä moninkertaisesti. Ongelmana on, että mallien laadussa on hyvin vähän pienentämisen varaa, sillä rakennusten eri elementit, kuten huonekalut ja LVI-ratkaisut on oltava tarpeeksi yksityiskohtaisia hahmotamisen vuoksi.

Ratkaisu

Kun mallien laadussa ei ole paljoa tinkimisvaraa, on keksittävä ratkaisu, jotta alustalle käännettävät rakennusmallit eivät aiheuttaisi alhaista kehysnopeutta

ja silti olisivat tarpeeksi yksityiskohtaisia. Ratkaisuna itse VirRake-alustaa alettiin optimoimaan Unreal Enginen tarjoamalla optimointityökaluilla. Tämä optimointitekniikoihin perehtyvä opinnäytetyö auttaa Virrake-alustan kehitystyössä tuoden menetelmiä ja tekniikoita alustan ja sen suorituskyvyn optimoimiseksi. Vaikkakin Virtuaalinen rakentaminen -hanke keskittyy kehitystyössään käyttämään Unreal Engineä, opinnäytetyössä käydyt Unityn optimointitekniikat ja -menetelmät ovat helposti hyödynnettävissä myös toisissa pelinkehitysympäristöissä.

6 PÄÄTÄNTÖ

Opinnäytetyön tavoitteena oli selvittää, mitä optimointitekniikoita ja -tapoja Unity-pelimootori tarjoaa, kuinka ne toimivat ja kuinka niitä voitaisiin hyödyntää VirRake-hankkeessa. Unity tarjoaa kehittäjän (ja optimoijan) onneksi hyvin laajan ja kattavan dokumentaation, jonka seurauksena optimointitekniikoiden, kuten Occlusion-järjestelmän käyttöönotto esimerkkiprojektissa onnistui erittäin hyvin. Pelin yksinkertaisten 3D-mallien ansiosta batchaus-tekniikoiden hyödyntäminen oli hyvin mutkatonta ja tuotti hyvää tulosta. Projektin optimoinnin onnistumisen kannalta pääsimme asettamiimme optimiarvoihin.

Näiden tekniikoiden lisääminen esimerkkiprojektiin ei kuitenkaan ollut niin mustavalkoista, kun alun perin opetin. Aluksi oli selvitettävä mitä optimointitekniikoita on tarjolla ja mitä tekniikoita on järkevintä projektissa hyödyntää. Oikeiden optimointitekniikoiden valitsemiseen tarvittiin Unityn profiloijaa, jonka avulla pelistä saatiin etsittyä suorituskykyä haittaavia ongelmakohtia. Osa profiloijan moduuleista aiheutti päänsäiväviä niiden antamien tulosten runsauden vuoksi. Esimerkiksi prosessorin kulutusta mittaavan moduulin tulosten tulkitseminen osoittautui erittäin hankalaksi, kun moduuli tulostaa jokaisen prosessorin säikeessä tapahtuvan kutsun ja skriptien päivityksen millisekunnin tarkkuudella.

Lisäksi osa Unityn tarjoamista optimointitekniikoista ovat mielestäni jääneet hieman vanhanaikaisiksi. Esimerkkinä Dynamic batchingin asettama 300 verteksin rajoitus on nykyisten 3D-mallien monimutkaisuuden rinnalla hyvin rajoitettava tekijä ja tästä syystä Dynamic Batchingiä käytettiin vasta viimeisimpänä batching-vaihtoehtona projektin optimoinnissa.

Jatkokehityksen kannalta projektin optimointi tuotti erittäin lupaavia tuloksia. Optimoitu peli kuluttaa paljon vähemmän tietokoneen resursseja. Tämä mahdollistaa uusien pelillisten elementtien, sekä mahdollisesti yksityiskohtaisempien mallien lisäämisen. Optimointivaiheessa LOD-järjestelmän käyttöä projektissa ei katsottu kustannustehokkaaksi 3D-mallien yksinkertaisuuden takia. Tulevaisuutta ajatellen seuraava askel pelin jatkokehityksessä olisi lisätä yksityiskohtaisempia malleja, jotta LOD järjestelmää voitaisiin hyödyntää.

Opinnäytetyötä tehdessäni opin optimointitekniikoiden lisäksi hyvin paljon oikeaoppisia tapoja, joita pelinkehittäjä tulisi noudattaa peliä tehdessään. Esimerkiksi peliprojektin tiedostohierarkian ylläpito. OnRailsShooterin identtisten materiaalien määrä olisi myös voitu estää oikeilla tuontiasetuksilla.

Lisäksi opin, kuinka tärkeää pelinkehitys projekteissa on välillä ottaa askel taaksepäin ja tutkia mittaustyökalujen antamia tuloksia. Kehitystyössä useasti keskitytään lopulliseen tulokseen, ilman että mietitään, minkälaista räsitusta esimerkiksi liian kompleksiset 3D-mallit aiheuttavat projektissa. Ongelmakohtia aletaan ratkomaan vasta projektin loppupuolella. Säännöllisten mittausten ja profiloinnin avulla kehittäjä pystyy ennaltaehkäisemään jopa usean päivän optimointityön.

LÄHTEET

Damon, W. 2012. Achieving Performance: An Approach to Optimizing a Game Engine. WWW-dokumentti. Saatavissa: <https://software.intel.com/en-us/articles/achieving-performance-an-approach-to-optimizing-a-game-engine> [viitattu 20.10.2019].

Davis, M. 2001. Computer Graphics. E-kirja. Nova Science Publishers. Saatavissa: <https://kaakkuri.finna.fi/> [viitattu 10.10.2019].

Dickinson, C. 2017. Unity 2017 Game Optimization – Second Edition. E-kirja. Packt Publishing. Saatavissa: <https://kaakkuri.finna.fi/> [viitattu 20.10.2019].

Flick, J. 2020. Level of Detail Cross-Fading Geometry. WWW-dokumentti. Saatavissa: <https://catlikecoding.com/unity/tutorials/scriptable-render-pipeline/level-of-detail/> [viitattu 8.1.2020].

Garreffa, A. 2015. id Software aiming for 1080p 60FPS on Doom. WWW-dokumentti. Päivitetty: 26.7.2015. Saatavissa: <https://www.tweaktown.com/news/46644/id-software-aiming-1080p-60fps-doom/index.html> [viitattu 20.3.2020].

Geometry Shader. 2020. Learn OpenGL. WWW-dokumentti. Saatavissa: <https://learnopengl.com/Advanced-OpenGL/Geometry-Shader> [viitattu 7.5.2020].

GLSL Tutorial – Fragment Shader. 2015. Lighthouse3D. WWW-dokumentti. Saatavissa: <https://www.lighthouse3d.com/tutorials/glsl-tutorial/fragment-shader/> [viitattu 10.11.2019].

GLSL Tutorial – Vertex Shader. 2015. Lighthouse3D. WWW-dokumentti. Saatavissa: <https://www.lighthouse3d.com/tutorials/glsl-tutorial/vertex-shader/> [viitattu 9.11.2019].

GPU Instancer: Terminology. 2020. Gurbu Technologies. WWW-dokumentti. Saatavissa: https://wiki.gurbu.com/index.php?title=GPU_Instancer:Terminology [viitattu 17.11.2019].

Hougaard, K. 2013. Occlusion Culling in Unity 4.3: The Basics. Blogi. Päivitetty: 2.12.2013. Saatavissa: <https://blogs.unity3d.com/2013/12/02/occlusion-culling-in-unity-4-3-the-basics/> [viitattu 18.11.2019].

Masuch, M., Röber, N. 2005. Game Graphics Beyond Realism: Then, Now, and Tomorrow. PDF-dokumentti. Saatavissa: <http://www.digra.org/wp-content/uploads/digital-library/05150.48223.pdf> [viitattu 7.5.2020].

NURBS vs Polygon Mesh - Why Both Models Are Needed in BIMscript. 2019. BIMobject. WWW-dokumentti. Päivitetty: 29.7.2019. Saatavissa: <https://academy.bimobject.com/en/bimscript/nurbs-vs-polygon-mesh-why-both-models-are-needed-in-bimscript> [viitattu 29.4.2020].

Profiling Applications Made with Unity. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://learn.unity.com/tutorial/profiling-applications-made-with-unity#5c7f8528edbc2a002053b5b6> [viitattu 4.1.2020].

Simonov, V. 2017. How to see why your draw calls are not batched in 5.6. WWW-dokumentti. Päivitetty: 3.4.2017. Saatavissa: <https://blogs.unity3d.com/2017/04/03/how-to-see-why-your-draw-calls-are-not-batched-in-5-6/> [viitattu 15.11.2019].

Takahashi, D. 2012. Game developers start your Unity 3D engines (interview) WWW-dokumentti. Päivitetty: 2.11.2012. Saatavissa: <https://venturebeat.com/2012/11/02/game-developers-start-your-unity-3d-engines-interview/> [viitattu 15.10.2019].

Tiwari, J. s.a. OpenGL Rendering Pipeline | An Overview. WWW-dokumentti. Saatavissa: <https://www.geeksforgeeks.org/opengl-rendering-pipeline-over-view/> [viitattu 7.11.2019].

Tonči, J. 2015. Draw Calls in a nutshell. WWW-dokumentti. Päivitetty: 26.6.2015. Saatavissa: <https://medium.com/@toncijukic/draw-calls-in-a-nutshell-597330a85381> [viitattu 16.11.2019].

Tutorial 30: Basic Tessellation. 2020. OGLdev. WWW-dokumentti. Saatavissa: <http://ogldev.atSPACE.co.uk/www/tutorial30/tutorial30.html> [viitattu 7.5.2020].

Unity Documentation. 2020. StaticBatchingUtility. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/ScriptReference/StaticBatchingUtility.html> [viitattu 14.1.2020].

Unity Manual - Camera. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/Manual/class-Camera.html> [viitattu 17.11.2019].

Unity Manual – Draw Call Batching. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/Manual/DrawCallBatching.html> [viitattu 12.1.2020].

Unity Manual - Getting started with the Profiler window. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/Manual/Profiler-Window.html#modules> [viitattu 4.1.2020].

Unity Manual – GPU Instancing. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/Manual/GPUInstancing.html> [viitattu 14.1.2020].

Unity Manual – Level of Detail. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/Manual/LevelOfDetail.html> [viitattu 8.1.2020].

Unity Manual – Occlusion Culling. 2020. Unity Technologies. WWW-dokumentti. Saatavissa: <https://docs.unity3d.com/Manual/OcclusionCulling.html> [viitattu 19.11.2019].

Unreal Engine 4 Documentation – Unreal Datasmith. 2020. Unreal Engine. WWW-dokumentti. Saatavissa: <https://docs.unrealengine.com/en-US/Engine/Content/Importing/Datasmith/index.html> [viitattu 29.4.2020].

Virtuaalinen Rakentaminen. s.a. XAMK. WWW-dokumentti. Saatavissa: <https://www.xamk.fi/tutkimus-ja-kehitys/virtuaalinen-rakentaminen/> [viitattu 13.4.2020].

What is Computer Graphics? 1998. Cornell University Program of Computer Graphics. WWW-dokumentti. Päivitetty 15.4.1998. Saatavissa: <http://www.graphics.cornell.edu/online/tutorial/> [viitattu 11.10.2019].