Renderoinnin toteutus Unreal Engine 5 - ympäristössä

Syväkatsaus nanite tuotantoputkeen

Ammattikorkeakoulun opinnäytetyö

Hämeen ammattikorkeakoulu, Tieto- ja viestintätekniikan koulutsohjelma

Kevät 2022

Mikael Kinnunen

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto, ja viestintätekniikka, insinööri | Tiivistelmä
Tekijä Mikael Kinnunen | Vuosi 2022
Työn nimi Renderoinnin toteutus Unreal Engine 5 - ympäristössä
Ohjaaja Mika Virolainen

Opinnäytetyön tavoitteena on tarkastella Nanite tuotantoputken toimintaa Unreal Engine 5 early access -kehitysympäristössä ja toimia suomenkielisenä tietolähteenä Unreal Engine viiteen liittyen. Työn teoriaosuudessa käydään läpi Naniten tuotantoputken vaiheet, jonka yhteydessä myös käsitellään pelisuunnittelua yleisellä tasolla.

Opinnäytetyön käytännön osuudessa perustetaan testausympäristö, minkä avulla toteutetaan myös työn testausosuus. Testaukseen kuuluvat kuvataajuus-, sekä kolmiomäärä testit. Testien avulla tutkitaan Nanite tuotantoputken suorituskykyä.

Mittaustulokset analysoidaan ja ne esitellään johtopäätökset osiossa.

Avainsanat renderointi, Unreal Engine 5, pelikehitys, Nanite
Sivut 24 sivua ja liitteitä 5 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and communications technology, engineer
Author Mikael Kinnunen
Subject Rendering in Unreal Engine 5
Supervisors Mikael Kinnunen

Abstract
Year 2022

The goal of this thesis is to examine the workflow of Nanite's pipeline in Unreal Engine 5 and work as a source of information in finnish language regarding Unreal Engine 5. The theoretical part of this thesis will include different stages of Nanite's pipeline and also cover game designing on a general level.

The practical part consists of test environment building and testing. The tests will include a frames-per-second and triangle count tests in the environment. With these two tests we will study Nanite's performance during build.

Results will be analyzed and presented in the conclusion part of the thesis.

Keywords rendering, Unreal Engine 5, game development, Nanite
Pages 24 pages and appendices 5 pages

Sisällys

1 Johdanto 1
2 Käsitteet 2
3 Renderoinnin historiaa 3
4 Unreal engine ja Nanite tietoperusta 4
4.1 Historia 4
4.2 Mikä on Nanite? 5
4.3 Rasterointi 6
4.4 Yksityiskohtien taso (LOD) 8
4.5 Klusterointi ja buildaaminen 10
4.6 Tiedostokoko 12
4.7 Laitteistopohjainen vai ohjelmistopohjainen rasterointi 13
5 Testiympäristön perustaminen 15
5.1 Saariympäristö 15
5.2 Testausmenetelmät 15
5.2.1 Kuvataajuus testi 16
5.2.2 Kolmiomäärä testi 17
6 Johtopäätökset 21
7 Pohdinta 22
Lähteet 24
Liitteet 26

1 Johdanto

Tämän opinnäytetyön tavoitteena on tarkastella uuden Unreal Engine 5.0.0 – version tuomaa uudistusta pelimoottorissa käytettävään renderointi teknologiaan. Valitsin kyseisen aiheen, koska minulla on kokemusta edellistä Unreal Engine 4.27 – versiosta ja sen lisäksi uuden version toiminnallisuudet sekä ominaisuudet peligrafiikan käsittelyssä vaikuttavat mielestäni hyvin mielenkiintoisilta. Aiemmat toteutukseni ovat keskittyneet pääsääntöisesti animoinnin sekä niiden toiminallisuuksien kehittämiseen blueprint ohjelmakoodilla Unreal peliympäristössä. Työ on suunnattu aiheesta kiinnostuneille opiskelijoille sekä myös heille, joilla on jo aikaisempia kokemuksia Unreal pelimoottorista.

Opinnäytetyö keskittyy tarkastelemaan Nanite renderoinnin tuotantoputkea koska tämä teknologia tulee muuttamaan toimintatapoja peligrafiikan kehitystyössä karsimalla aikaisemmin käytettyjä tarpeettomia menetelmiä mm. 3D-kappaleiden renderointi prosessissa. Tarkoitus on selvittää Nanite tuotantoputken eri vaiheet, niissä tapahtuvat tekniset ratkaisut ja miten uudet prosessit vaikuttavat Unreal kehittäjien työhön.

Opinnäytetyöni koostuu kolmesta pääosiosta: teoria, testaus ja analyysi. Teoriaosuudessa käsitellään Nanite tuotantoputken tietoperustaa sekä teknisten ratkaisujen toiminnallisuuksia. Testaus-osiossa selvitetään kahdella eri menetelmällä Nanite teknologian hyötyjä sekä rajallisuuksia. Näiden tietojen pohjalta tehdään tulokset-osio, jossa analysoidaan materiaali.

Suomenkielistä tietoa uudesta Unreal Engine (early access) 5.0.0 – versiosta ei juurikaan ole, joten opinnäytetyöni toimii myös muille aiheesta kiinnostuneille suomenkielisenä tietolähteenä. Tietoperusta koostuu virallisesta englanninkielisestä Unreal Engine 5.0 dokumentaatiosta, koska aiheen tuoreuden vuoksi muita aineistoja ei ole.

Tutkimusmenetelmäksi valitsin toimintatutkimuksen, koska sen avulla voidaan hakea ratkaisua teknisiin ongelmiin. Työssäni pyrin toimintatutkimuksen keinoin tuottamaan omanlaista tietoa pelimoottorin käytöstä. (Saaranen-Kauppinen & Puusniekka 2006).

2 Käsitteet

- Mikropolygoni
  monikulmio, joka on hyvin pieni suhteessa renderoitavaan kuvaan. Mikropolygonien avulla voidaan luoda erittäin yksityiskohtainen kuva näytölle.

- Gpu tuotantoputki
  malli, minkä avulla kuvataan eri työvaiheet, jotka grafiikkajärjestelmän on tehtävä renderoidakseen 3D-mallin näytölle.

- Mesh objekti
  kokoelma erilaisia pintoja, pisteitä ja reunoja, jotka yhdessä muodostavat 3D-mallin

- Klusterointi
  kolmioiden ryhmittely eri lohkoihin

- Rasterointi
  vektorigrafiikka muodossa kuvattu kuva muunnetaan rasterikuvaksi, joka on sarja pikseleitä, viivoja tai pisteitä. Muodostaa kolmiulotteisen kuvan kaksiulotteiselle näytölle.

- Instanssi
  menetelmä, jonka avulla renderoidaan useita kopioita samasta mesh objektista esimerkiksi pelimaailmaan.

- LOD, level of detail
  Yksityiskohtien taso 3D-mallissa.

3 Renderoinnin historiaa

Tässä opinnäytetyössä käsitellään runsaasti renderointiin kuuluvia aihepiirejä, joten on syytä avata käsitettä tarkemmin. Renderoinnin prosessiin kuuluu kaksiulotteisen tai kolmiulotteisen mallin muuntaminen fotorealistiseen tai epärealistiseen muotoon. Kolmiot on tietokone grafiikan datarakenteen perusta. On olemassa muitakin data rakenteita, joista näytöllä oleva grafiikka koostuu, mutta yleisesti kehittäessä visuaalista ympäristöä koostuu kaikki pohjimmiltaan kolmioista.

Miksi? Syy piilee kolmioiden monikäyttöisyydessä. Esimerkiksi 3D-mallinnuksissa pohjatason kolmiot soveltuvat sekä ns. ”taipuvat” eri muotoihin, jos verrataan esim. neliöön. Tämä osaksi helpottaa myös suorituskyvyn taakkaa renderoida tarkkalaatuista kuvaa. Jos otetaan tarkasteluun vaikkapa neliö 3D-mallinnuksessa, käy hyvin nopeasti ilmi että mikäli neliön muotoa muuttaa yhdestä pisteestä vaakatason alle sitä ei pysty renderöimään. Jos kyseinen neliö muutetaankin kahdeksi kolmioksi, joista neliö muodostuu pystyy tietokone sen silti piirtämään ruudulle. Nykypäivänä on monia eri ratkaisuja miten 3D-mallinnus toteuttaa data rakenteen. (Brewster, 2018, s. 1)

Tunnetuin tietokone grafiikan tutkija lienee Martin Newell, joka kehitti kuuluisan Utahin teekannu-mallin vuonna 1975. Martin aikaisemmat kokeet olivat tietyssä määrin epäonnistuneet jäljiteltävien muotojen yksinkertaisuuden takia. Mallit kuten shakkinappula sekä donitsi olivat muodoiltaan kelvottomia, koska ne eivät taipuneet esittämään Martinin luomia tehosteita esimerkiksi varjoja tai heijastuksia. Newell tarvitsi monimutkaisemman mallin ja päätyi vaimonsa ehdotuksesta mallintamaan teekannun, jolla hän pystyisi testaamaan tietokone grafiikalle kehitettyä algoritmiaan. Objektin muoto erilaisine kurveineen mm. kahvan muoto ja siinä oleva reikä mahdollisti myös varjon heijastumisen teekannun Bezier-käyrien ansiosta. Myöhemmin Newell julkaisi tutkimuksensa matemaattisen datan julkiseen tietoon, jolloin muut aihepiirin tieteilijät pystyivät tekemään omia testejään. Pian Utahin teekannu-malli saavutti suuren yleisön suosion ja sitä alettiin käyttää suorituskykytestinä renderoinnissa (Dunietz, 2016, s.1) (Liite 1.).

Nykypäivänä teekannu löytyy erinäisten 3D-mallinnusohjelmien vakiobjektien joukosta ja se onkin aloittelevalle mallintajalle hyvä objekti testata erilaisia valaistuksia, varjoja sekä pintamateriaaleja. Martin Newellin teekannu on myös päässyt kulttimaineen tasolle ja sitä näkeekin satunnaisesti vitsinä piilotettuna tehtyihin mallinnuksiin. Ennen Martin Newellin 1970-luvun teekannu-mallinnusta tehtiin myös toinen merkittävä kehitystyö. Tämä oli maailman ensimmäinen 3D-mallinnukseen tarkoitettu ohjelmisto Sketchpad, jonka kehittäjänä toimi Ivan Edward Sutherland. (Sutherland, 2003, s.31)

Paljoa ei tapahtunut 1980-luvulla renderöintiin liittyen mutta 1990-luvulta lähtien nähtiin suuria harppauksia teknologian kehityksen seurauksena mm. tehokkaammat tietokoneprosessorit sekä näytönohjaimet sallivat monimutkaisempia laskelmia. Elokuvissa oli mahdollista upottaa 3D-malleja kohtauksiin, tehdä erikoisefektejä ja animaatioita. Videopeleissä nähtiin myös vastaavanlaisia harppauksia.

4 Unreal engine ja Nanite tietoperusta

4.1 Historia

Unreal Engine on Epic Gamesin kehittämä alun perin vuonna 1998 julkaistu pelimoottori PC:lle. Ensimmäisen sukupolven Unreal pelimoottoria lähti kehittämään Tim Sweeney, joka on myöskin yksi Epic Gamesin perustajista. Vuonna 1995 aloitettu kehitystyö johti kolmen vuoden jälkeen Unreal pelin julkaisuun, jonka ohessa esiteltiin myös sitä pyörittävä pelimoottori. Ensimmäisen sukupolven versiot nojautuivat vahvasti ohjelmistopohjaiseen prosessori painotteiseen hahmontamiseen. Matkan varrella pelimoottori sai myös tuen käyttää näytönohjainta graafisiin laskelmiin tietokoneprosessorin lisäksi. Tämän ominaisuuden ansioista kehittäjät saivat lisää liikkumavaraa pelien tuotantoon. Ero muihin pelimoottoreihin tuohon aikaan oli merkittävä sillä Unreal pystyi hyödyntämään 16-bittistä väriavaruutta kun kilpailijat pysyttelivät vielä 8-bittisessä muodossa.

Tämä ensimmäisen sukupolven Unreal Enginellä oli myös rajallisuutensa esimerkiksi verkkopelaaminen oli muihin pelimoottoreihin verrattuna varsin kankeaa. Pelimoottorin korkeat laitevaatimukset toivat myös ongelmia monille käyttäjille. Nykypäivänä Unreal Engine on monipuolinen pelimoottori, jota käyttävät myös elokuva-alalla olevat kehittäjät. Virallinen julkaisupäivä Unreal Engine 5 tulee olemaan vuoden 2022 puolella, sillä käyttäjät voivat toistaiseksi hyödyntää vain beta-versiota. (Grove, 2019, s1)

4.2 Mikä on Nanite?

Nanite teknologia on Unreal Engine 5 pelimoottorissa toimiva virtualisoitu mikropolygoneihin perustuva GPU:lla ajettava tuotantoputki. Virtualisoitu tarkoittaa, että se lataa vain tarpeellisen tietomäärän joka vaaditaan näytöllä olevan grafiikan piirtämiseen. Nanite on tarkoitettu suurien kokonaisuuksien renderöintiin pelimaisemassa menettämättä laatua objekti tasolla. Nanite-tekniikka on suunniteltu analysoimaan mesh-objekti hierarkisesti, jonka aikana se myös pilkkoo objektin pienempiin osiin käyttäen klusterointia. Tämä tarkoittaa että esimerkiksi yksi kolmio mesh-objektista pilkotaan pienemmiksi kolmioiksi , joka taas pilkotaan yhä pienemmäksi.

Renderöidessä nanite pakkaa polygonit samalla klusterointi-tekniikalla vaihtaen tietoa niiden välillä. Käytännössä tämä tarkoittaa, että naniten avulla on mahdollista tuoda erittäin korkealaatuisia mallinnuksia käytettäväksi pelimoottoriin esimerkiksi photogrammetria tekniikalla kuvattuja objekteja tai ”elokuva-laatuisia” assetteja. Suorituskyvyn kannalta tämä on suuri harppaus aiempaan Unreal Engine 4.26 pelimoottoriin. Nanite mesh-objekti pystyy pitämään sisällään miljoonia kolmioita ja satoja tuhansia verteksejä kuin taas ilman sitä matala polygoni mesh-objekti pystyy käsittelemään vain murto-osan tästä. Jos tätä verrataan esimerkiksi korkealaatuiseen polygon mesh-objektiin, datan määrä naniten avulla pystytään pakata noin puolet pienemmäksi. Naniten hyöty tulee esille pyörittäessä suuria peliympäristöjä, jotka normaalisti vaatisivat rutkasti laskentatehoa tietokoneelta. Haittapuolena tässä pakkaamisessa on osittainen tarkkuuden menetys, jonka huomaa erityisesti etäisyyksiä muunnellessa. (Unreal Engine, 2021, Nanite)

4.3 Rasterointi

Nanite rasterointi tuotantoputken prosessi on seuraavanlainen. Käyttäjä tuo 3D-mallinnuksen Unreal Engine 5-ympäristöön, jolloin pelimoottori analysoi ja pilkkoo instanssit omiin ryhmiinsä. Nanite käyttää tässä vaiheessa instansseille läpäisyprosesseja, joiden tarkoituksena on eliminoida turhaa dataa tasojen sisällä. Kaikki Nanite tieto varastoidaan yhteen isoon resurssivarastoon, joka mahdollistaa datan haun missä vaiheessa tuotantoputkea tahansa.

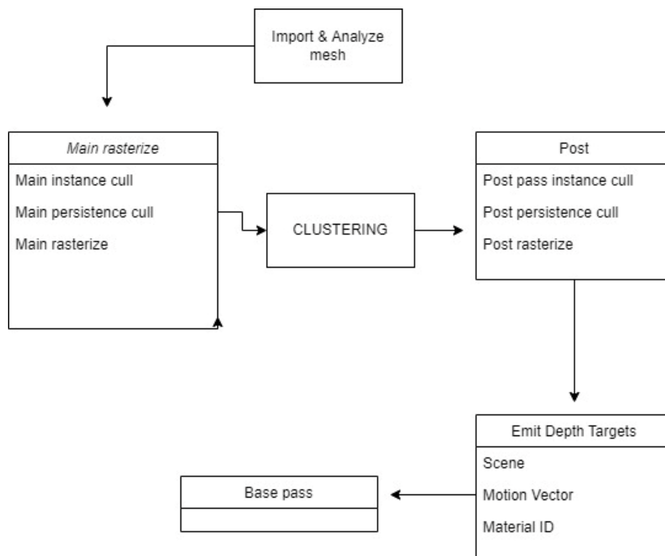
Rasteroinnin ensimmäinen osa on nimeltään ”Main rasterize” ja suurimman osan työstä Nanite tekee tässä vaiheessa eliminoiden instanssit ja kolmiot jotka eivät ole näkyviä. Prosessissa myös testataan onko näkymässä päällekkäisiä kuvia ”occlusion cull” hierarkisella z-puskurilla. Edellämainittua puskuria lyhennettynä HZB käytetään kiihdyttämään syvyyskyselyitä renderoinnissa. Käytännössä tässä läpäisyvaiheessa (instanssien eliminointi) laskennallinen kysely: laske suorakulmion rajoista, etsi alin arvo enimmäis intensiteetin projektiosta missä suorakulmio 4x4 pikseliä, testaa onko päällekkäisyyksiä. Naniten erikoisuus piilee kaksivaiheisessa okluusion läpäisytestissä. Se mitä oli renderöity edellisessa kuvassa on suurella todennäköisyydellä myös seuraavassa kuvassa. Toisinsanoen, edellisen kuvan data kutsutaan uuteen kuvaan, tehdään HZB-kysely, HZB:n avulla saadaan tieto onko kuvaan tullut uusia näkyviä objekteja, jos on niin vain nämä uudet objektit piirtyy näytölle. Tämä läpäisyvaihe vaikuttaisi olevan sidoksissa ”suoratoistoon”. (Siggraph, 2021, Advances in real-time rendering in games)

Ensimmäisen rasterointi-vaiheen jälkeen Nanite luonnollisesti on ryhmitellyt kolmiot sekä instanssit omille ryhmilleen jota myös voidaan kutsua klusteroinniksi. Tämä sama prosessi toistuu ”post rasterize” vaiheessa. Naniten viimeisimpänä vaiheen on erilaisten varjojen, syvyyksien sekä materiaali ID:n luokittelu. Syvyyksien laskentaan vaikuttaa objektien etäisyys toisistaan sekä myös käyttäjän näkökulmasta, eli kuinka lähellä tai kaukana objekti on ruudusta. Myös väliaikainen anti-aliasointi eli objektien reunojen pehmennykset ja heijastukset kuuluvat tähän kategoriaan. Prosessin materiaalien tekstuurisyvyydet

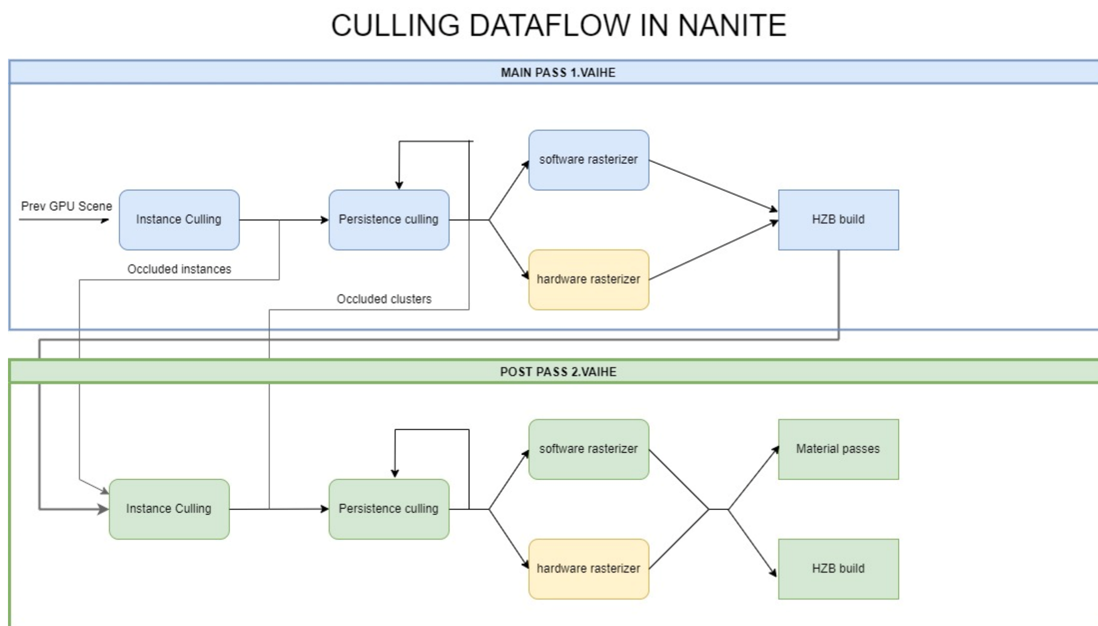
luokitellaan omiksi ID:ksi, joka käytännössä tarkoittaa, että jokaiselle materiaali ID:n omaavalle objektille on omakohtainen syvyysarvo muistissa ja sitä voi kutsua tarvittaessa. Tekninen termi tälle on ”visibility buffer” eli näkyvyyspuskuri. Esimerkiksi maisemaa skaalatessa arvo muuttuu suuremmaksi tai pienemmäksi toisinsanoen Nanite materiaali on yksityiskohtaisempi läheltä katsottuna kuin kaukaa, joka myös liittyy LOD parametreihin (Siggraph, 2021, Advances in real-time rendering in games).

Monesti pelituotannossa kehitysprosessin pullonkaulaksi osoittautuu normaalikarttojen baking-osuus eli objektin materiaalien päällystys. Normaalikartat ovat 3D-objektien pinnalle muodostuvia yksityiskohtia. Teoriassa Naniten rasterointi ei tarvitsisi geometriapuskurointia ollenkaan koska tarvittavan datan voisi aina hakea muistista. Käytännössä Nanite käyttää Gbufferia (geometrinen puskurointi) integroidusti oman varjostusdatansa kanssa jolloin lopputulos on optimoidumpi.

(kuva 2. Nanite rasterointi prosessi.)



(kuva 3. Siggraph, 2021, Culling dataflow in nanite)



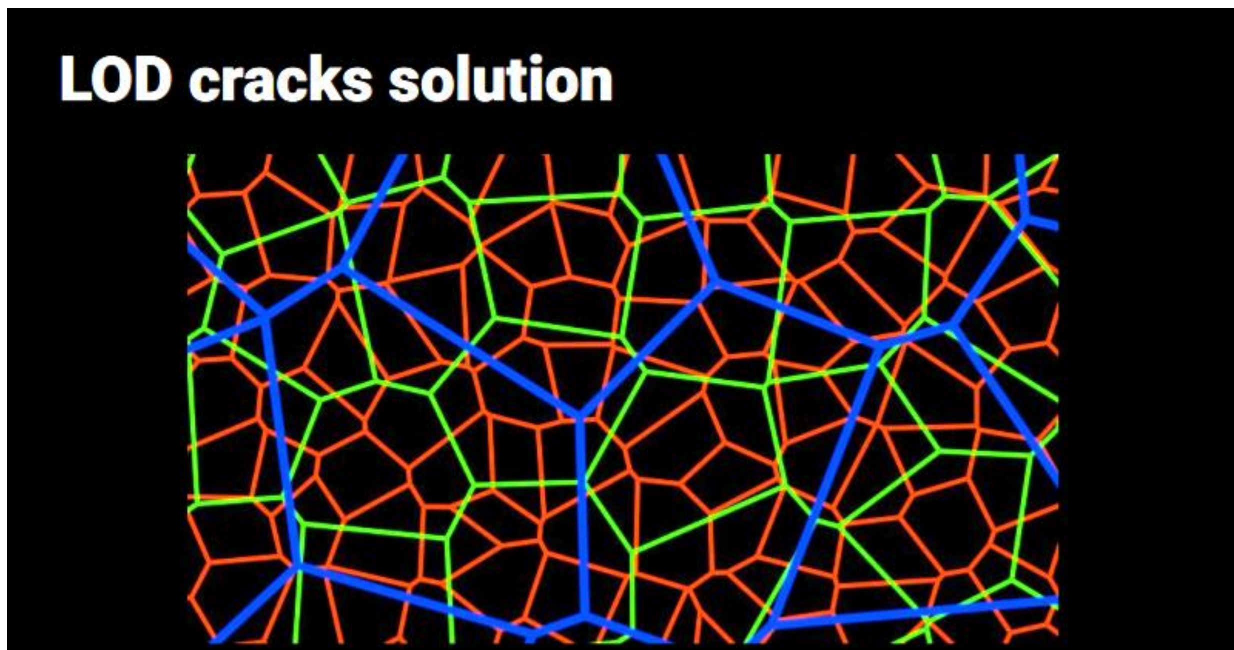
4.4 Yksityiskohtien taso (LOD)

LOD:it voidaan tarvittaessa ryhmitellä samanlaisiksi klustereiksi kuten kolmioiden tapauksessa Nanite tekee. Kokonaisuudessaan ”klusterointi-puu” on toteutettu yksinkertaisella parent-child rakenteella, mitä käytetään usein esimerkiksi ohjelmoinnissa. Jokaisen yksityiskohdan taso voidaan hakea klusterista vastaamaan ruudun perspektiivin mukaisesti. Toisinsanoen mikäli piirtoetäisyys objektiin on hyvin lähellä yksityiskohtien rakenne olisi parent – child – child – child muodossa, jolloin piirtotarkkuus olisi erittäin tarkka. Vastaavanlaisesti mikäli piirtoetäisyys objektista olisi todella kaukana vaikkapa peliympäristön vuoristot, ei niiden tarvitse käyttäjän näkökulmasta olla yksityiskohtaisia koska ruudun etualalla renderoitavat objektit ovat tärkeämpiä. Mielenkiintoista Naniten LOD:issa on, että koko klusterin ei tarvitse olla säilössä muistissa kokoajan. Samaan tapaan kuin rasteroinnissa, voidaan yksityiskohdat hakea datasta silloin kun niitä tarvitaan (on-demand). Tämä voi olla mikä tahansa yksityiskohdan osa ja samalla objektilla voi

yksityiskohdat olla eri laatuisia. Esimerkiksi jos 3D-mallin yläosa näkyy enimmäkseen käyttäjälle se voi olla hyvin tarkka ja vähän näkyvä tai näkymätön alaosa epätarkka.

Mitä tapahtuu repeämien suhteen? Repeämät tai halkeamat 3D-malleissa johtavat päällekkäisyyksiin, jonka seurauksena renderoitu malli voi olla epämuodostunut ja objektin varjostus epäonnistuu mikroskooppisten aukkojen takia verteksien tai tasoerojen välillä. Tässäkin tapauksessa LOD:it käyttävät klusterointia hyväkseen pakottamalla tasoerot omiin klustereihin jolloin myös halkeamat sulautuvat umpeen. Esimerkiksi 0-tasolla olevat halkeamat klusteroidaan 1-tasolla ja taso 1 klusteroidaan tasolle 2. Kuvassa punainen väri on taso 0, vihreä taso 1, sininen taso 2.

(kuva 4. Siggraph, 2021, Nanite LOD cracks solution)

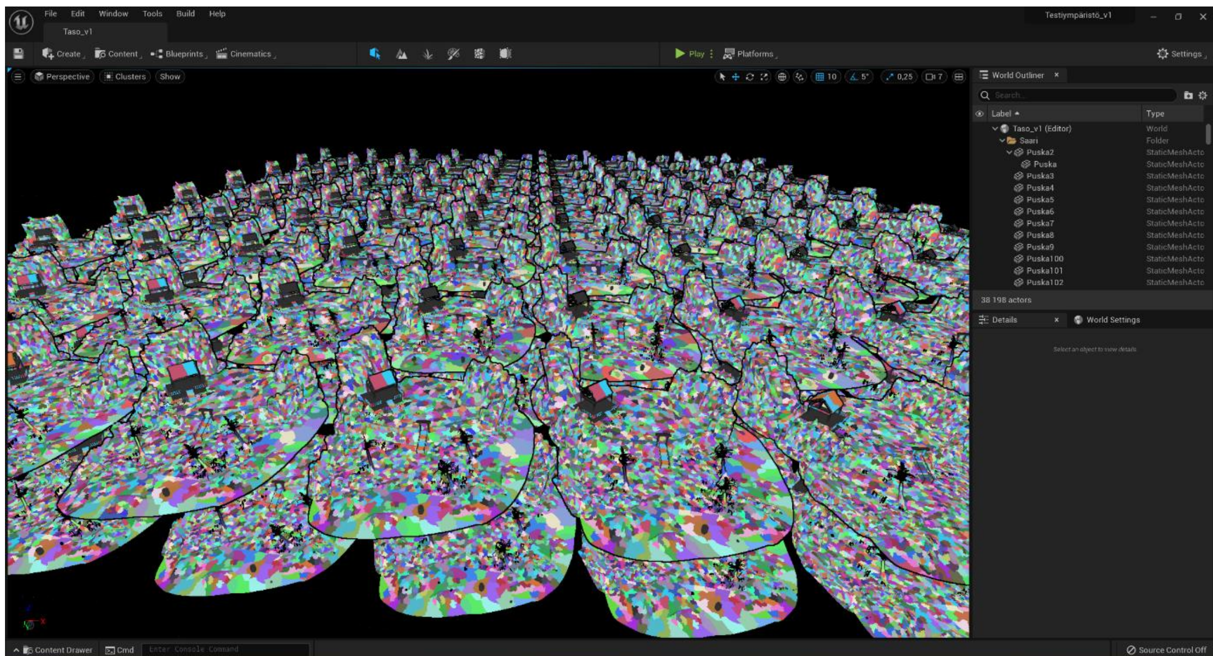


Tämän tekniikan avulla voidaan vaihtaa eri klustereiden rajoja tasolta toiselle ryhmittelemällä ja sulauttaa verteksien väliset lohkot. (Siggraph, 2021, advances in real-time rendering in video games, s.32).

Nanite hyödyntää METIS-ohjelmisto kirjastoa, joka laskee eri algoritmein graafiosion. Käytännössä METIS-kirjasto jakaa ensiksi tietyn määrän osioita graafiin, minimoi osioit jossa reunat ylittävät toisensa ja ylittävät osiot sulautuvat viereiseen klusteriin kuitenkin niin, että

ne eivät liity toisiinsa. Toisinsanoen sama kolmio ei voi olla samaan aikaan kahdessa eri klusterissa. (Karypis, George, 1997, PARMEmS: Parallel Graph Partitioning and Sparse Matrix ordering library, s. 3-9).

(kuva 5. Nanite LOD klusterointi testiympäristö, Unreal Engine 5)



4.5 Klusterointi ja buildaaminen

Nanite määrittelee ohjelman ajo-prosessissa kolmioiden määräksi maksimissaan 128 per klusteri ”while NumClusters > 1” parametrilla. Kompression aste alkuperäisestä kolmioiden määrästä on 50% ja klustereille määrätään oma värikoodi RGB-taulusta. Alkuperäinen klusteri puu muuntautuu tällä menetelmällä suunnatuksi sykliseksi verkoksi (DAG) koska tilanteissa, jossa ei ole mahdollista etsiä parent klusteria ylimättä kyseisen klusterin rajoja. Käytännössä se tarkoittaa, että repeämiä ei pitäisi löytyä LOD-tasolla. Prosessissa on kolme vaihetta:

1. Valitaan N-määrä klustereita. Jokaisella klusterilla on oma värikoodinsa, jonka perusteellaa ne voidaan tunnistaa. Esimerkiksi vihreä, sininen, punainen, keltainen

2. Nämä neljä klusteria yksinkertaistetaan ja sulautetaan toisiinsa. Vaiheen tarkoitus on puolittaa kolmioiden määrä. DAG:issa tämä tarkoittaa, että klustereille määrätään parent noodi, jonka alla kyseiset klusterit sijaitsevat.

3. Prosessin viimeisessä vaiheessa yksinkertaistettu noodi eli parent klusteri jaetaan takaisin kahdeksi klusteriksi jättäen molemmille parent klusterille alkuperäiset neljä child klusteria.

Vaihetta toistetaan niin kauan kunnes kaikki on pelkistetty yhdeksi klusteriksi DAG:n juuressa. (Siggraph, 2021, Advances in real time rendering in video games, s.46)

Nanite hyödyntää 3D-objektin klusterin yksinkertaistukseen Quadric error metric mittausta. Mittauksella selvitetään matemaattisen kaavan avulla kuinka kaukana verteksi on ideaalisesta pisteestä. Yleisesti ottaen algoritmi on seuraavanlainen:

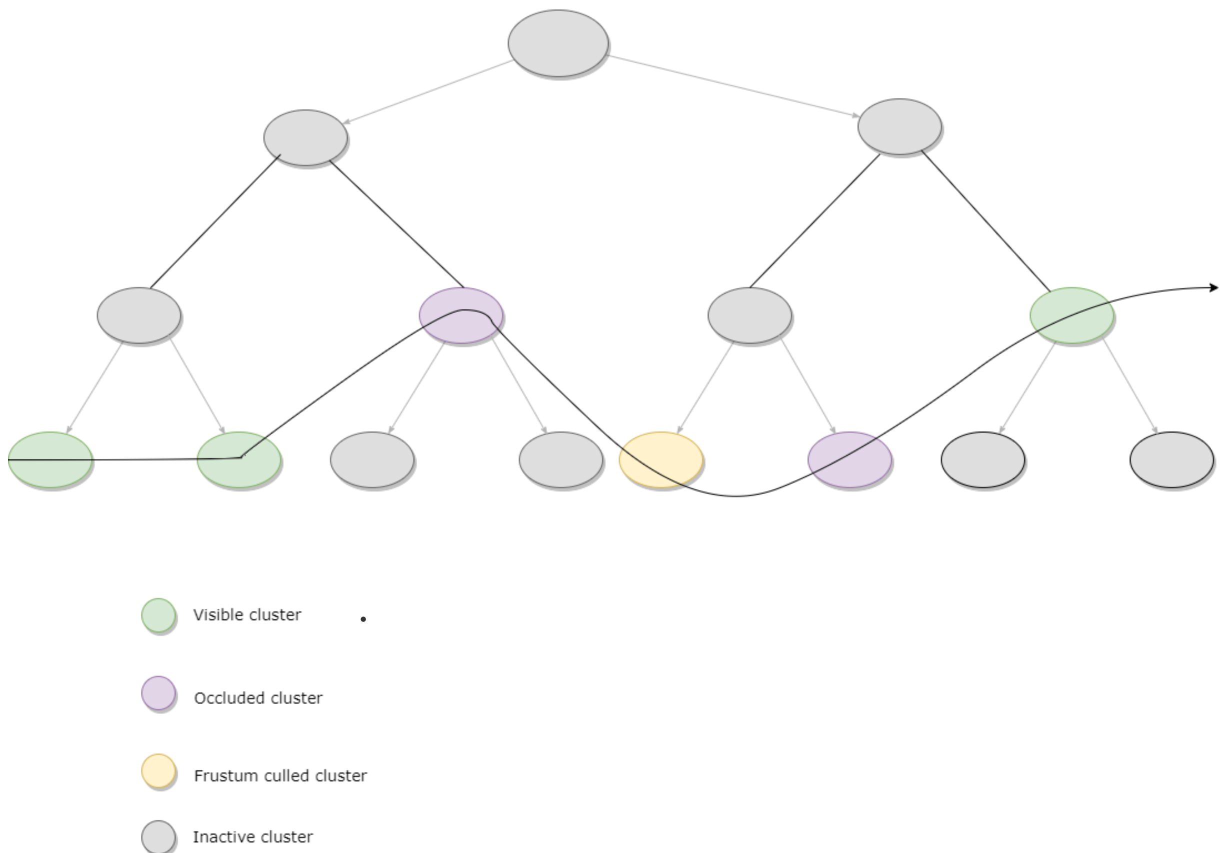
$$\bar{\mathbf{v}} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

, jossa v = on objektin reuna. Algoritmi laskee Q-matriisit kaikille alkupisteille (vertekseille), josta se valitsee kelvolliset parit. Tämän jälkeen lasketaan optimaalinen tavoite supistumiselle $\bar{\mathbf{v}}$ jokaiselle kelvolliselle parille (v1, v2). Parin supistumisen hinta saadaan arvioimalla virhe yhdistettyjen neliöiden parien kärjistä. Parien data rakenne järjestetään uudestaan lajittelemalla vähimmäiskustannusten mukaan. (Garland, Heckbert. Surface Simplification Using Quadric Error Metrics).

Esimerkiksi aikaisempi tapa renderoida moniresoluutioisia rakenteita, joka tukeutui vahvasti tietokoneen suorittimen sekä näytönohjaimen tehoon on nykyinen Naniten kaltainen ohjelmistopohjainen renderointi prosessi tehokkaampi kuin laitteiston tehoon perustuva renderointi. Laitteistopohjainen lähestymistapa lisää merkittävästi suorittimen työkuormaa koska LOD määritetään kolmioiden tai verteksien avulla. Huomioon tulee myös ottaa merkittävät tekniset harppaukset näytönohjaimissa, jonka takia pullonkaulaksi muodostuu nopeasti tietokoneprosessorin laskentakyky. Toisin sanoen laitteistopohjainen moniresoluutio renderointi prosessi ei pysty laskemaan mitä tarvitsee renderöidä ja kuinka

tarkasti riittävän nopeasti. Naniten tapauksessa tätä ongelmaa ei tule eteen koska klusteroinnilla mahdollistetaan mitä tarvitsee ja mitä ei tarvitse renderöidä.

(Kuva 6. esimerkki datan hausta klusteripuussa.)



4.6 Tiedostokoko

Nanite käyttää tiedostojen supistamiseen tai kompressointiin kahta eri kompressiorakennetta. Ne käsittelevät samaa dataa mutta ovat optimoitu eri tarkoituksia varten, toinen on tarkoitettu muistin ja toinen levytilan käyttöön. Muistiformaatin tarkoituksena on minimoida kokoa levytilassa kompression jälkeen. Ensiksi sijainnit sekä muut määritteet kvantisoidaan omiksi arvoikseen, jonka jälkeen klusterointi prosessilla nämä arvot tallennetaan koordinaatteina klusterin minimi/maksimi alueen sisään. Toisin sanoen

jokainen klusteri varastoi uniikin formaatin verteksistä eikä mitään muuta. (Siggraph, 2021, Advances in real-time rendering in games). Tarkastelussa huomataan kuinka suuri kompression määrä on kääntäessä korkealaatuinen assetti nanite assetiksi.

(kuva 7. 3D-tynnyri high poly ja nanite asetuksilla testiympäristössä.)

S_Dirty_Plastic_Drum_uiujbcxva_lod0_Var1 (Static Mesh)

Path: /Game/Megascans/3D_Assets/Dirty_Plastic_Drum_uiujbcxva
Asset Filepath Length: 158 / 210
Cooking Filepath Length: 215 / 260
Disk Size: 870,423 KiB
Vertices: 8 190
Triangles: 14 356
Materials: 1
Never Stream: False
LODGroup: None
Approx Size: 56 × 57 × 90
Nanite Percent: 0,0
Sections with Collision: 1
Nanite Enabled: False
Collision Complexity: CTF_UseSimpleAndComplex
Min LOD: 0
UVChannels: 1
Collision Prims: 0
Default Collision: BlockAll
LODs: 5
Source File: ../../../../../../Downloads/cache/uiujbcxva/tier1/uiujbcxva_LOD0.fbx

S_Dirty_Plastic_Drum_uiujbcxva_high_Var1 (Static Mesh)

Path: /Game/Megascans/3D_Assets/Dirty_Plastic_Drum_uiujbcxva
Asset Filepath Length: 158 / 210
Cooking Filepath Length: 215 / 260
Disk Size: 143,254 MiB
Vertices: 1 640
Triangles: 2 000
Materials: 1
Never Stream: False
LODGroup: None
Approx Size: 56 × 58 × 90
Nanite Percent: 0,0
Sections with Collision: 1
Nanite Enabled: True
Collision Complexity: CTF_UseSimpleAndComplex
Min LOD: 0
UVChannels: 1
Collision Prims: 0
Default Collision: BlockAll
LODs: 1
Source File: ../../../../../../Downloads/cache/uiujbcxva/tier0/uiujbcxva_High.fbx

Tiedoston koko levyllä high poly assetilla noin. 870 mb kun taas nanite muodossa n. 143 mb. Kolmioiden sekä verteksien määrä supistuu klusteroinnin ansioista jolloin 84 % muutos tynnyri mallissa on merkittävä. Laadullista eroa ei huomaa.

4.7 Laitteistopohjainen vai ohjelmistopohjainen rasterointi

Naniten tapauksessa suurin osa rasteroinnista tapahtuu ohjelmistopohjaisesti (software) ja isojen kolmioiden tapauksessa nanite hyödyntää edelleen laitteisto rasterointia (hardware) koska tähän tarkoitukseen se on edelleen nopeampi, toisinsanoen isot kolmiot jotka peittävät suuren alueen pikseleitä tietokoneen ruudulla. Laitteisto rasteroinnissa kompastuskiveksi muodustuu nopeasti syvyystestien teko. Toki on olemassa esimerkiksi DirectX shader 5 versio, jolla pystyy tehdä syvyystestin (early depth test) laitteistolla välttääkseen ylimääräisten pikselien prosessointia mutta Naniten tapauksessa se vain hidastaisi tuotantoputkea. (Microsoft, 2021, DirectX, HLSL, Shader 5)

Sen sijaan ohjelmistopohjainen syvyystesti hyödyntää 64bitin funktiota (atomic function), joka näyttää seuraavanlaiselta:

(Kuva 8. 64b syvyystesti)

```
for( uint y = MinPixel.y; y < MaxPixel.y; y++ )

 {
    float CX0 = CY0;
    float CX1 = CY1;
    float CX2 = CY2;
    float ZX = ZY;

for( uint x = MinPixel.x; x < MaxPixel.x; x++ )
   {

    if( min3( CX0, CX1, CX2 ) >= 0 )
    {

    WritePixel( PixelValue, uint2(x,y), ZX );
    }
        CX0 -= Edge01.y;
        CX1 -= Edge12.y;
        CX2 -= Edge20.y;
        ZX += GradZ.x;
   }

  CY0 += Edge01.x;
  CY1 += Edge12.x;
  CY2 += Edge20.x;
  ZY += GradZ.y;
 }
```

korkean bittimäärän ansiosta syvyystesti on mahdollinen koska suurempi määrä bittejä voidaan käyttää syvyysarvojen määrittämiseen tai tallentamiseen. Naniten software rasterointi erittelee bittien käytön seuraavanlaisesti: 30 bittiä syvyysarvoihin, 27 bittiä näkyvien klusterien indeksointiin ja 7 kolmio indeksiin. Nämä 27+7 bittiä muodostavat näkyvyyspuskurin joka lopulta määrittää renderoidaanko pikseli pakatussa muodossa näytölle vai ei.

5 Testiympäristön perustaminen

Testiympäristönä käytettiin Unreal Engine 5 early access – pelimoottoria. Ympäristö on julkaistu kuluttajakäyttöön toukokuussa 2021. Lopullinen julkaisu täysversiolle on suunnitteilla alkuvuodesta 2022. Tarkoituksena on testata Nanite toiminnallisuuksia sekä sen suorituskykyä erilaisin rasitustestein. Samalla käytännön osuus painottuu tehtävien testien lisäksi havainnollistamaan, kuinka opinnäytetyön teoria osuutta hyödyntämällä voidaan todeta suorituskyvyn muutoksia.

5.1 Saariympäristö

Opinnäytetyön testiympäristönä käytetään itse suunniteltua sekä rakennettua saarta Nanite prosessien testaukseen. Ympäristön luonti oli helppo toteuttaa Quixel bridge-työkalua käyttäen, jonka avulla saatiin suoraan tuotua Nanite-mesh 3D-mallit peliympäristöön. Mallien tuonti tapahtui napin painalluksella ja jäljelle jäi vain mallien sijoittelu loogisille paikoille.

Alkuvaiheessa ongelmaksi koitui lähinnä liikojen ideoiden karsinta, koska huomioon täytyi ottaa myös oman tietokoneen suorituskyky. Ohjelmallisesti liian raskaan ”maailman” luominen ei olisi käytännön osaltakaan järkevää, koska opinnäytetyön tarkoituksena on vain toteuttaa testejä ja tarkastella niiden tuloksia. Tarkoituksena oli luoda ympäristö, jossa voitaisiin testata Nanite teknologian mahdollisuuksia sekä rajallisuuksia.

5.2 Testausmenetelmät

Ennen kuin testaus aloitetaan on syytä selventää minkälaisella PC:lla testausympäristö pyörii. Tietokoneen käyttöjärjestelmä opinnäytetyön tekohetkellä Windows 11 Pro, versio 21H2.

Prosessorina AMD Ryzen 7 5800x 3,9 gigahertsiä, näytönohjaimena AMD Radeon 6700 XT, 32 GB DDR4 keskusmuistia ja Adata 2TB SSD kiintolevy.

Testaus toteutettiin pelimoottorin sisällä, keskittyen tiettyihin aihepiireihin. Näitä ovat kuvataajuus testi sekä kolmiomäärä testi. Tarkoituksena oli tarkastella näiden osa-alueiden ominaisuuksia sekä selvittää miten parametri-arvojen muutokset vaikuttivat pelimoottorin toimivuuteen. Testien tekemisen hetkellä tarkasteltiin myös tietokoneen suorituskyvyn muutoksia.

5.2.1 Kuvataajuus testi

Kuvataajuus eli frames-per-second (fps) rasitustestin tarkoituksena on selvittää mitkä tekijät vaikuttavat kuvataajuuteen. Kuvataajuuteen vaikuttavat monet tekijät. Näistä yleisimpiä ovat esimerkiksi

- Antialiasointi, mikä pehmentää rosoisia reunoja kuvassa
- Field of view toisinsanoen kuinka lähellä näkökenttä on varsinaista ympäristöä. Määrittää kuinka paljon ruudulle renderoidaan ympäristöä
- Näytön resoluutio
- Tekstuurien yksityiskohdat
- Valo/varjostus

Tietokoneessa oleva laitteisto vaikuttaa myös kuvataajuuteen. Testaus toteutettiin pelaajakameraa liikuttamalla eri suunnista sekä kamerakulmista katsottuna. Testaus aloitettiin valitsemalla Unreal Engine 5 EA-editorista ”show fps stat”, mikä näyttää ruudun päivitysnopeuden per sekuntti ruudun oikeassa yläkulmassa. Nanite sekä Lumen laitettuna päälle FPS-arvo oli n. 90, kameran ollessa paikallaan ylhäältä alaspäin suunnattuna. Tästä kuvakulmasta oikealta vasemmalle liikkeessä arvo tippui 86 fps.

Samaa menetelmä toistettiin kamera horisonttiin suunnattuna. Paikaltaan fps-arvo on 86 ja oikea-vasen liikkeessä 76fps. Merkittävä havainto tehtiin asetuksia muuttamalla: Nanite enabled, Lumen disabled. Globaalin varjostuksen pois kytkentä nosti suorituskykyä 31.11% kamera ylhäältä alas sekä horisonttiin suunnattuna 16.27% kameran ollessa staattinen. Liikkeestä vastaavilla kuvakulmilla saadut arvot olivat 24.42% (ylhäältä alas) ja 11.84% (vasen-oikea).

(Kuva 9. FPS-testi tarkemmin kuvattuna.)

KAMERA NÄKYMÄ	KUVATAAJUUS (FPS)	Lumen ON	Lumen OFF
KAMERA YLHÄÄLTÄ ALAS, STAATTINEN	90	x	
KAMERA YLHÄÄLTÄ ALAS, LIIKKEESSÄ	86	x	
KAMERA YLHÄÄLTÄ ALAS, STAATTINEN	117,9		x
KAMERA YLHÄÄLTÄ ALAS, LIIKKEESSÄ	106,9		x
KAMERA SUORAAN ETEEN, STAATTINEN	86	x	
KAMERA SUORAAN ETEEN, LIIKKEESSÄ	76	x	
KAMERA SUORAAN ETEEN, STAATTINEN	99,9		x
KAMERA SUORAAN ETEEN, LIIKKEESSÄ	84,9		x

5.2.2 Kolmiomäärä testi

Tämän kokeen tarkoituksena oli pyrkiä selvittää Naniten kyky käsitellä maksimaalinen määrä kolmioita sekä tarkastella, miten määrä vaikuttaa suorituskykyyn sinä hetkenä.

Nanite tuotantoputken ensimmäinen osa, main cull rasterize osoittaa että pre-cull arvo instanssien määrälle on 20115 . Ensimmäisen läpäisyn jälkeen post-cull arvo on 3573. Jo ensimmäisellä kerralla suurin osa instansseista eliminoituu näkyvyys testin seurauksena.

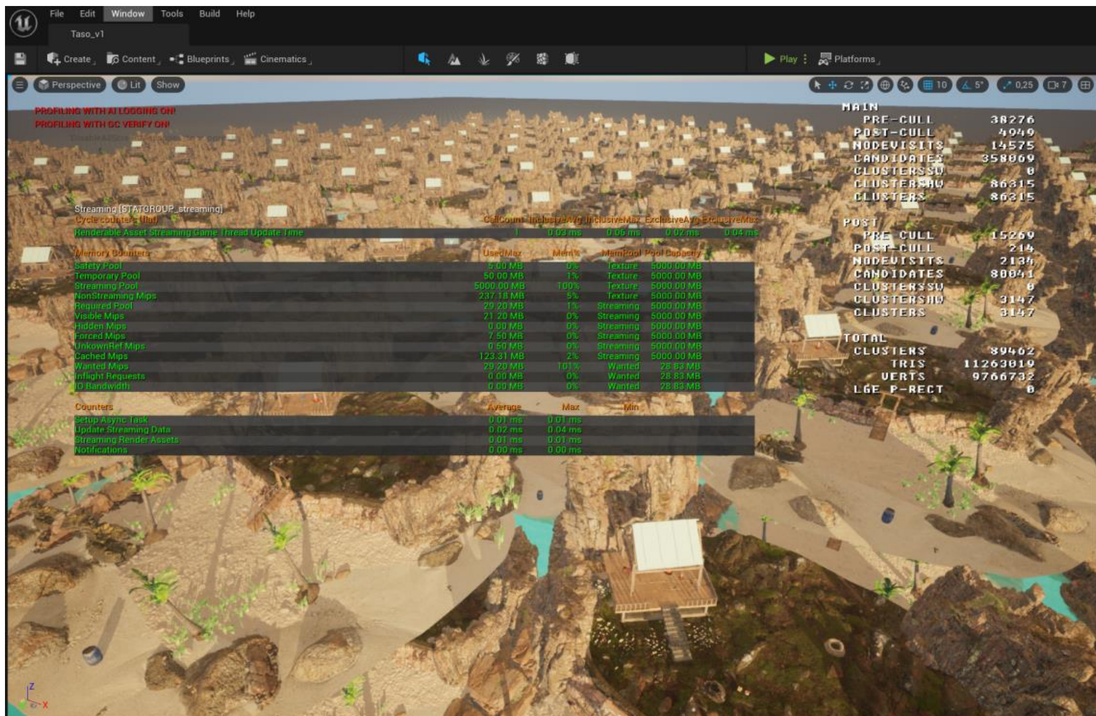
Klustereita ensimmäisellä kierroksella oli 76241. Naniten kompressoi vielä ensimmäisen osan jälkeen toisen kerran, tarkoituksenaan löytää lisää instansseja näkyvyystestien avulla. Myös klusterointiin kiinnitetään prosessissa erityistä huomiota. DAG-menetelmän avulla klusterit voidaan kompressoida laajempaan kokonaisuuteen jolloin vain tarvittavat näkyvät osat renderöidään näytölle. Alkuperäinen arvo 76241 pienentyy huomattavasti arvoon 2447.

Testi aloitettiin kloonaamalla saariympäristö skaalaus toimintoa käyttäen 1 -> 100 kpl. Ensimmäisen skaalauskierroksen jälkeen kolmioita peliympäristössä oli noin 11,1 miljoonaa kuvataajuuden ollessa 62 fps 4K-asetuksilla. Tämä ei juuri vaikuttanut ympäristölle raskaalle suorituskyvyn kannalta, mikä mahdollisti lisäskaalauksen. Saariympäristö kloonattiin uudelleen jolloin päädyttiin 22,2 miljoonaan kolmioon. Kuvataajuus laski 47fps 4K-asetuksilla. 22.2 miljoonaa kolmiota oli tuotettu ennen kuin se oli kulkenut nanite tuotantoputken läpi.

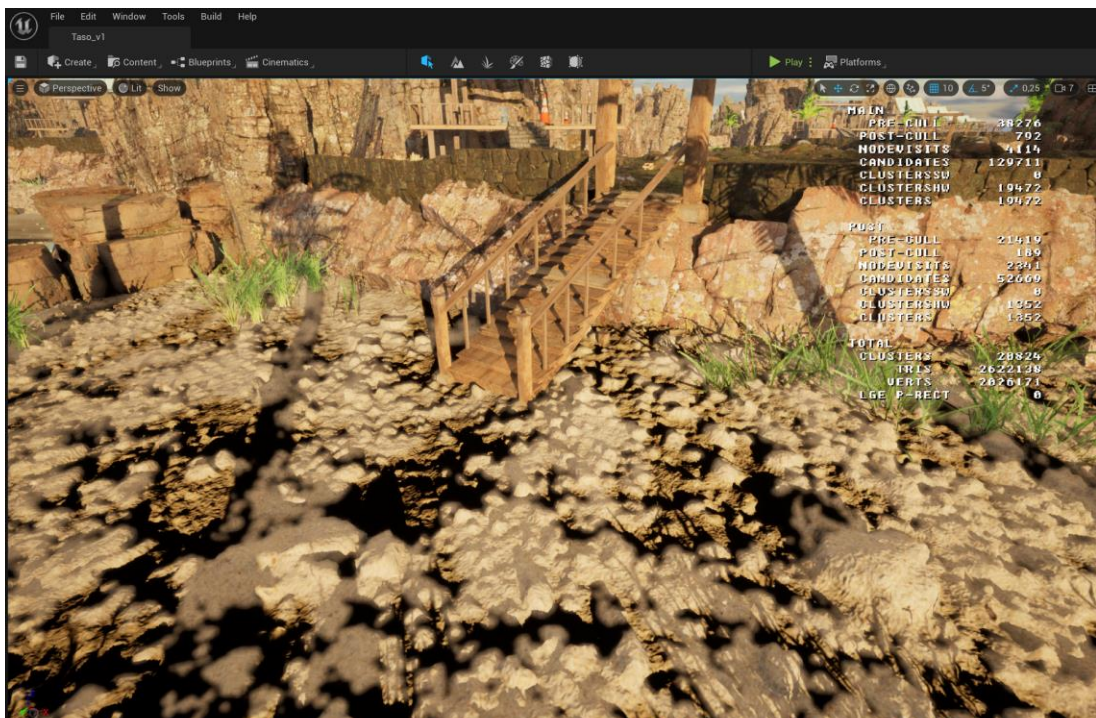
Mielenkiintoista oli, että Nanite päätti käyttää testaus hetkellä testiympäristössä hardware-pohjaista clustershw renderointia ympäristön piirtämiseen ruudulle. Tämä hidasti työskentelyä ja testausta merkittävästi skaalatessa sekä monistaessa saarta koska laitteistopohjainen renderointi vaatii tietokoneelta todella paljon suorituskykyä. Toiseksi esteeksi koitui hetkellisesti tekstuurien suoratoiston budjetti. Default asetuksilla se on 1000.00 MB, joka ylittyi kun saaria monistettiin useampi kappale testiympäristöön. Unreal Enginen komentorivi-työkalua hyödyntämällä tekstuuri budjettia voi lisätä r.Streaming.PoolSize – komennolla. Budjettia lisättiin viisinkertaiseksi 1000.00 MB -> 5000.00 MB, jolloin päästiin eroon ”Texture streaming pool over budget” – virhesanomasta.

Kuvakaappauksen ottamisen hetkellä nähdään että main-cull clustershw arvo on 86315 ja post-cull clustershw supistuu 3147. Naniten hienous käy ilmi zoomatessa näkymää lähemmäksi saarta, jolloin arvot tippuvat huomattavasti. Main cull clustershw arvo tippui 19472 klusteriin ja tuotantoputken post-cull loppuvaiheessa arvo on 1352. Kuvat 10 ja 11 havainnollistavat asiaa tarkemmin.

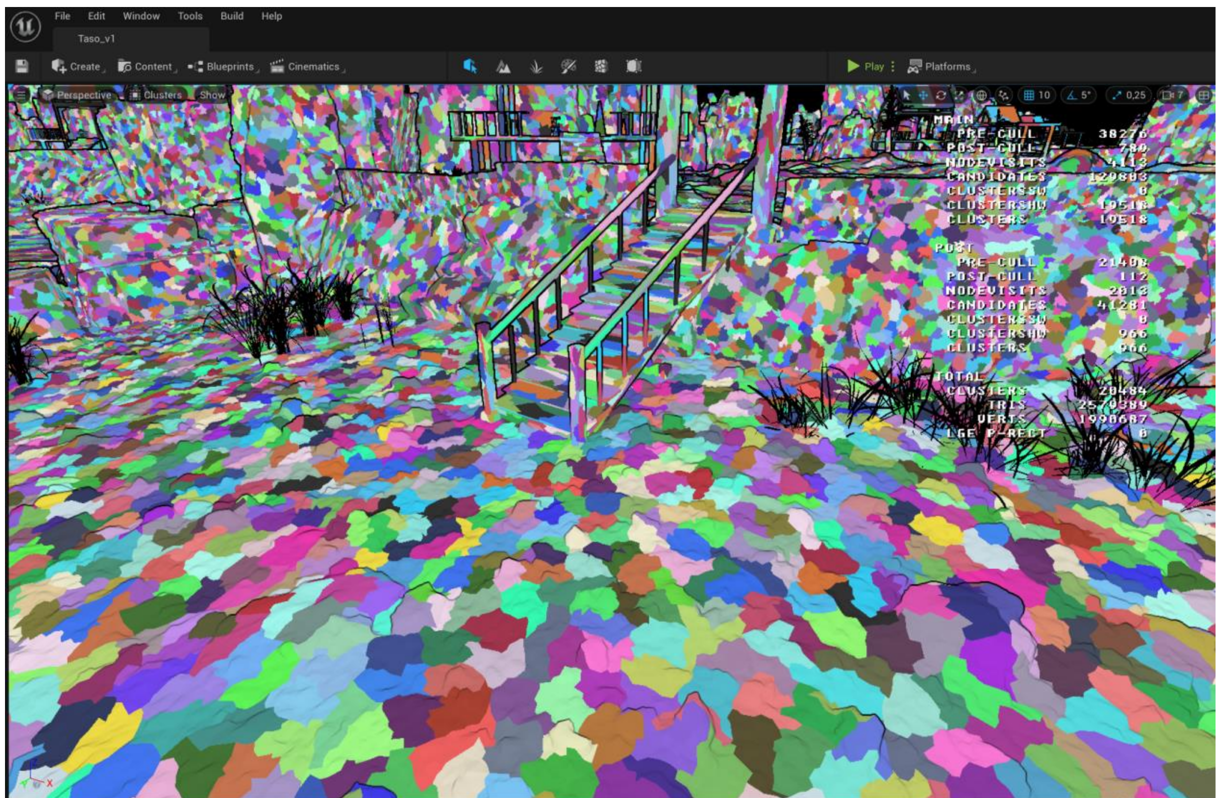
(kuva 10. Nanite statistiikkaa peliympäristöstä ennen toista kloonausta.)



(Kuva 11. Nanite statistiikkaa, zoomattu saari.)



(Kuva 12. Nanite klusteri visualisaatio.)



Miksi Nanite päätti hyödyntää mielummin laitteistoa renderointiin kuin ohjelmistoa? Yksi mahdollinen syy voi olla itse pelimoottorin versiossa. Koska tähän opinnäytetyöhön käytettiin Unreal Engine early access 5.0.0 versiota kaikki ominaisuudet eivät välttämättä toimi. Kuitenkin Epicin omasta kaupasta ladattavalla Unreal Engine Valley of the Ancients demo projektissa ohjelmistopohjainen renderointi toimi. Kyseiseen demoon on käytetty UE 5.0 pelimoottoria, joka on eri kuin kuluttajille julkaistu early access – versio. Tämä saattaa selittää eri mekanismit renderoinnin toteutukseen versioiden välillä.

6 Johtopäätökset

Ensimmäinen tutkimuskysymys opinnäytetyössä oli miten Unreal Engine 5 Nanite tuotantoputki vaikuttaa sitä käyttävien pelikehittäjien työhön? Tähän tutkimuskysymykseen löytyi vastaus. Uusi Nanite prosessi mahdollistaa 3D-mallinnuksessa käytettävän baking-menetelmän pois jättämisen, minkä ansiosta 3D-malli voidaan tuoda mallinnukseen tarkoitetusta ohjelmasta suoraan Unreal Engine 5 ohjelmaan. Tämä osoittaa että Nanite hoitaa 3D-mallien teksturoinnin ohjelman sisällä.

Toiseen tutkimuskysymykseen saatiin myös vastaus. Nanitesta löytyi enemmin mahdollisuuksia kuin rajallisuuksia. Opinnäytön testimenetelmissä havaittiin, että Nanite pystyy käsittelemään todella suuria määriä kolmioita (11 miljoonaa) hidastamatta ohjelmaa tai tietokonetta sekä pyörittämään korkealaatuisia tekstuureja melko suurella kuvataajuudella. Rajallisuudeksi osoittautui laitteistopohjainen renderointi Nanite tuotantoputkessa. Testistä havaittu 22 miljoonaa kolmiota hidasti merkittävästi pelimoottoria, mikä pidensi renderointia testiympäristössä.

Kuvataajuus pysyi testien jälkeen 47-67 ruutua-per-sekunti. Kuvataajuus testistä havaittiin kamerakulman vaikuttavan kuvataajuuden arvoon positiivisesti, kun näytölle renderoitavia objekteja oli vähän. Näkymän ollessa laaja kamerassa havaittiin muutoksen kääntyvän negatiiviseen suuntaan kuvataajuuden pudotessa. Mitä enemmän on renderoitavaa oli ruudulla, sitä enemmin kuvataajuus putosi. Yleisesti suorituskykyyn testeissä enimmäkseen vaikutti tietokoneen RAM-muistin määrä, mikä oli tekohetkellä 32GB. Näytönohjaimen sekä prosessorin rasitus pysyi koko testien ajan alhaisena.

7 Pohdinta

Opinnäytetyö opetti minulle paljon peliohjelmoinnissa käytettävistä menetelmistä, teknisistä toteutuksista sekä Nanite tuotantoputken eri vaiheista. Aikaisempaa tietoa minulla näistä aiheista oli vain rajallisesti, joten opinnäytetyön kautta tietoperustani laajentui rutkasti. Opinnäytetyö prosessina oli mielenkiintoinen sekä omalta osaltaan myös melko haastava koska tieteellinen kirjoittaminen oli varsin uusi kokemus itselleni. Toisaalta tutkimuksien luku oli tuttua muilta aloilta hankitun osaamisen myötä. Myöskin hyvä englanninkielen taitoni helpotti dokumenttien lukemista sillä kaikki opinnäytetyössä käytetyt materiaalit olivat kirjoitettu tällä kielellä. Oma kiinnostukseni myös peliohjelmointia sekä alaa kohtaan helpottivat työskentelyä koska tekeminen oli mielekästä. Suurten kokonaisuuksien ja vaiheiden hahmotus peliohjelmoinnin tuotantoputkessa aiheutti ajoittain haasteita sillä käytettäviä menetelmiä ja peliohjelmoinnin tapoja on monenlaisia. Opinnäytetyö herätti myös kiinnostuksen työelämäkohtaisesti ja tulevaisuudessa tästä opinnäytetyöstä opittuja taitoja voisi hyödyntää myös ammatillisesti.

Opinnäytetyön tavoitteet ja tutkimuskysymyksiin vastaaminen toteutuivat suurimmalta osin. Unreal Engine viiden uusi Nanite tuotantoputki karsii pelikehityksessä aikaisemmin käytettyjen toimintatapojen, kuten tekstuurien baking-prosessin sekä alhaisempien kolmiomäärien käytön 3D-malleissa. Opinnäytetyössä harmiksi koitui laitteistopohjainen renderointi, kun päätarkoitus oli tehdä ympäristö ohjelmapohjaisesti eli software rendering koska tämä on juuri Naniten vahvuuksia. Tulevaisuudessa ehkä tätä opinnäytetyötä voisi käyttää pohjana sekä jalostaa paremmaksi juurikin tuota aihetta tarkastellen ja testaten. Koska kyseessä oli vasta early access versio Unreal Engine viidestä ja varsinainen uuden pelimoottorin julkaisu on vuoden 2022 aikana, minkä takia sitä ei pystytty hyödyntämään täysin.

Tehdyissä testeissä havaittiin erikoinen ominaisuus Unreal Engine pelimoottorissa, joka oli RAM-muistin korkea rasitus testiympäristöä renderoidessa. Aikaisemmin oletin pelimoottorin olevan näytönohjaimelle raskas. Tämä muistin käyttöprosentti selittyy osaksi Naniten uudistuksista, jotka liittyvät tekstuurien suoratoistoon sekä muistiin säilöttävän datan sekä datan noutamisesta muistista. Vanhemmat SSD-levyt sekä matalataajuuksiset

RAM-muistit osoittautuvat pullonkaulaksi niiden kirjoitinnopeuksien takia. Kuitenkin Nanite tuotantoputken tapa kompressoida datan määrää menettämättä juurikaan visuaalista laatua yksityiskohtien tasolla on todella tehokasta vaikka kyseessä on vasta beta-versio tulevasta pelimoottorista.

Mielenkiintoista on miten uuden sukupolven pelikonsolit kuten Playstation 5 tai Xbox Series X hyödyntävät Unreal Engine viiden ominaisuuksia. Molemmat konsolit hyötyisivät suuresti suorituskyvyn kannalta koska Nanite keskittyy ohjelmistopohjaiseen renderointiin sekä mahdollistaa erittäinkin yksityiskohtaisen pelimaailman pyörittämisen pienellä vaivalla. Uuden sukupolven konsolit ovat erityisesti investoineet kiintolevyjen kirjoituslevyn nopeuksiin, mikä mahdollistaa esimerkiksi reaaliaikaisten muutosten teon pelimaailmaan. Esimerkiksi hiljattain julkaistu ”The Matrix Awakens, an Unreal Engine 5 experience” demo on todella vakuuttava renderoinnin tarkkuuden osalta koska oikean realistisen hahmon ja siitä tehdyn Nanite 3D-mallin erottaminen on lähes mahdotonta pysäyttämättä videota.

Pelikehittäjien näkökulmasta tarkastellen Nanite tuotantoputken uudet menetelemät mahdollistavat monimutkaisten sekä luovien ratkaisujen toteuttamisen, sillä Naniten ansioista esimerkiksi polygonien määrää ei käytännössä tarvitse rajoittaa kuten aikaisemmin on tehty. Myöskin Naniten mahdollistama 3D-mallien ja peliympäristöjen suoratoisto vapauttaa kiintolevyiltä muistia muihin tarkoituksiin. Tämän ansioista pelisuunnittelijat voivat kohdentaa vapaita resursseja esimerkiksi tarkempiin yksityiskohtiin pikselitasolla, minkä ansioista uuden sukupolven pelit tuntuvat mukaansatempaavimmilta ja aidommilta kuin aikaisemmin tai kehittää peliympäristöjä luovemmin. Perinteisesti peliympäristöjen tapauksessa pelikehittäjät ovat joutuneet ottamaan huomioon kuinka paljon ruudulle voi renderoida dataa, jonka takia esimerkiksi yksinpelien ympäristöt on jouduttu tekemään lineaarisin menetelmin sekä ujuttamalla niihin piilotettuja latausruutuja. Näitä ovat esimerkiksi hahmon kulkeminen pienistä raoista tai eteneminen pitkän käytävän läpi seuraavaan alueeseen. Koska uuden sukupolven konsoleissa on panostettu SSD-levyn kirjoitinnopeuteen sekä pelimoottorien kuten Unreal Enginen uudet versiot hyödyntävät suoratoisto menetelmiä, pääsevät peliohjelmoijat rajoittavista aspekteista eroon. Mielestäni tämä opinnäytetyö voisi myös toimia tulevaisuudessa toisille opinnäytetyön tekijöille tietopohjana koska suomenkielistä dokumentointia Unreal Engine viidestä ei löydy.

LÄHTEET

Ricky Grove. (2019). Focus: Unreal Engine – A brief history of Unreal

https://magazine.renderosity.com/article/5330/focus-unreal-engine-a-brief-history-of-unreal

Siggraph. (2021). Advances in real-time rendering in games.

https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf Luettu 9.12.2021

Brewster, Kahle (2018). Wayback Machine Internet Archive. https://web.archive.org/web/20180329092515/http://softimage.wiki.softimage.com:80/xsidocs/poly_basic_PolygonMeshes.html Luettu 9.12.2021.

Dunietz, Jesse. (2016). Nautil US.

https://nautil.us/blog/the-most-important-object-in-computer-graphics-history-is-this-teapot

Ivan E., Sutherland. (2003). Sketchpad: A man-machine graphical communication system.

https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf

Turizin, Mike (2020). Hierarchical Depth Buffers.

https://miketuritzin.com/post/hierarchical-depth-buffers/

Garland, Michael. Paul S. Heckbert, (????). Surface Simplification Using Quadric Error Metrics.

https://people.eecs.berkeley.edu/~jrs/meshpapers/GarlandHeckbert2.pdf

Karypis, George. Schloegel, Kirk. Kumar, Vipin (1997). PARMEmS: Parallel Graph Partitioning and Sparse Matrix ordering library.

https://conservancy.umn.edu/bitstream/handle/11299/215345/1997-060.pdf?sequence=1

Microsoft. (2021). DirectX, HLSL, Shader 5.

https://docs.microsoft.com/fi-fi/windows/win32/direct3dhlsl/sm5-attributes-earlydepthstencil

Unreal Engine. (2021). Nanite.

https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/Nanite/ Luettu 11.15.2021.

Unreal Engine. (2021). Virtual Shadow Maps.

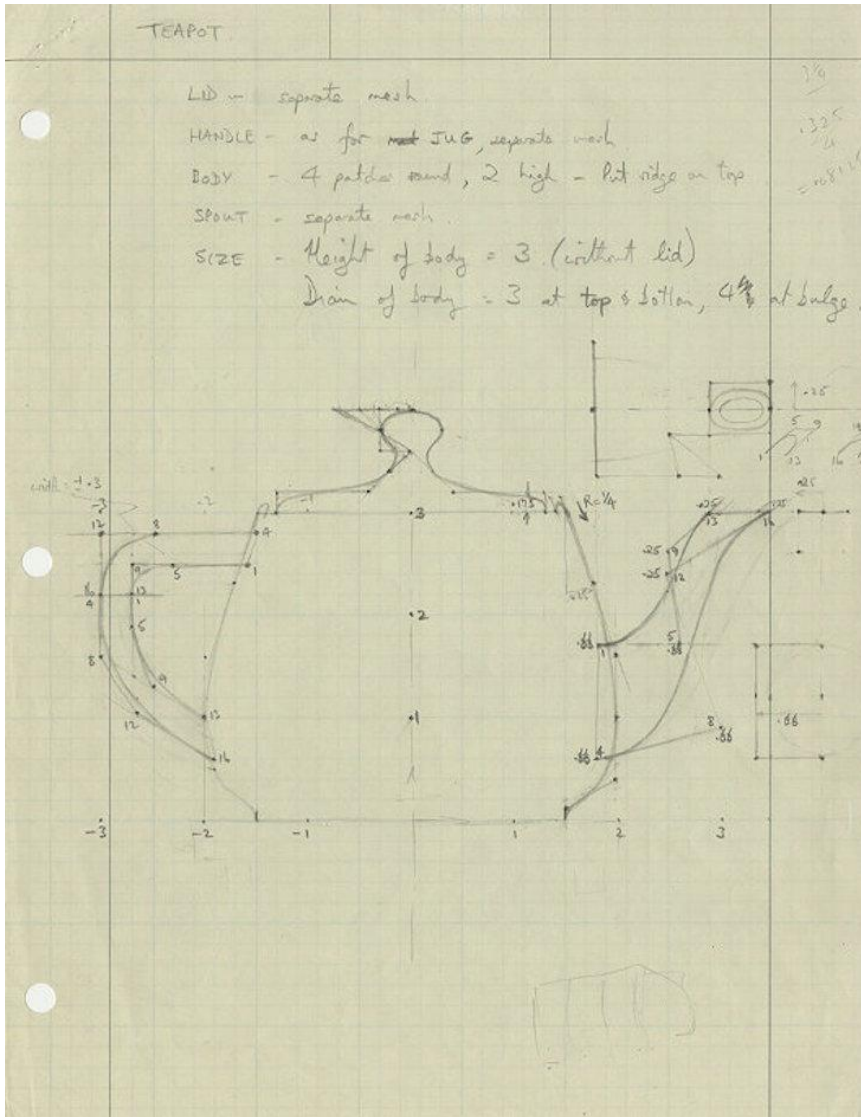
https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/VirtualShadowMaps/

Unreal Engine. (2021). Quixel Bridge Plugin for Unreal Engine

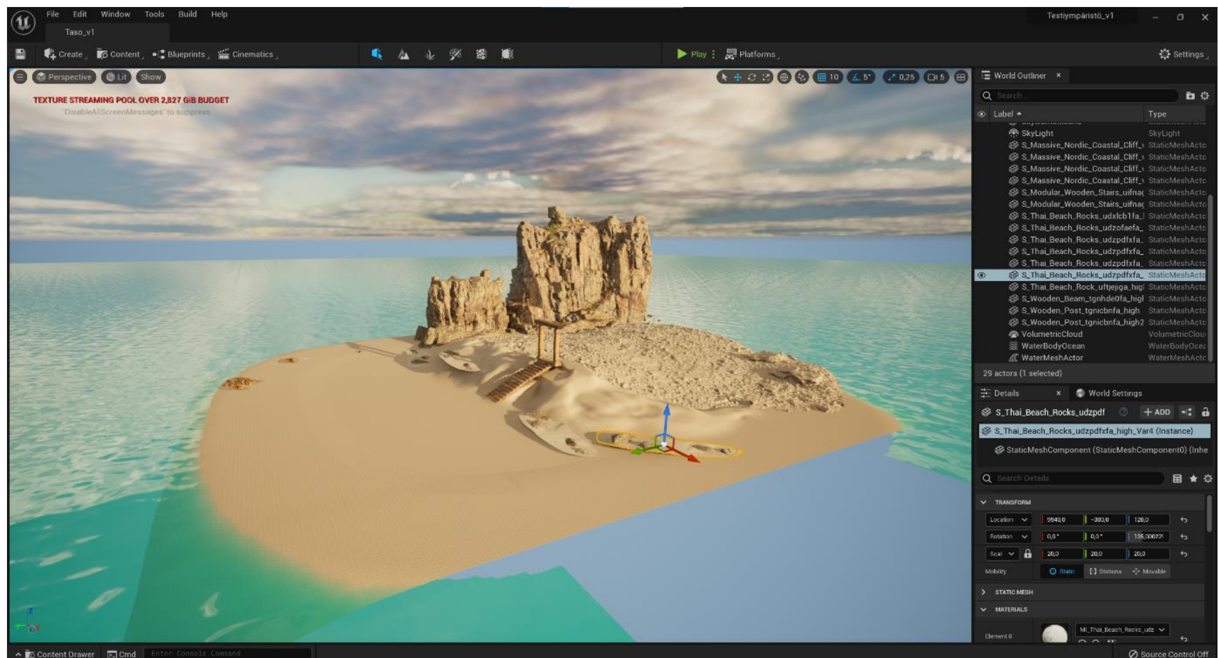
https://docs.unrealengine.com/5.0/en-US/WorldFeatures/QuixelBridgePluginForUnrealEngine/

LIITTEET

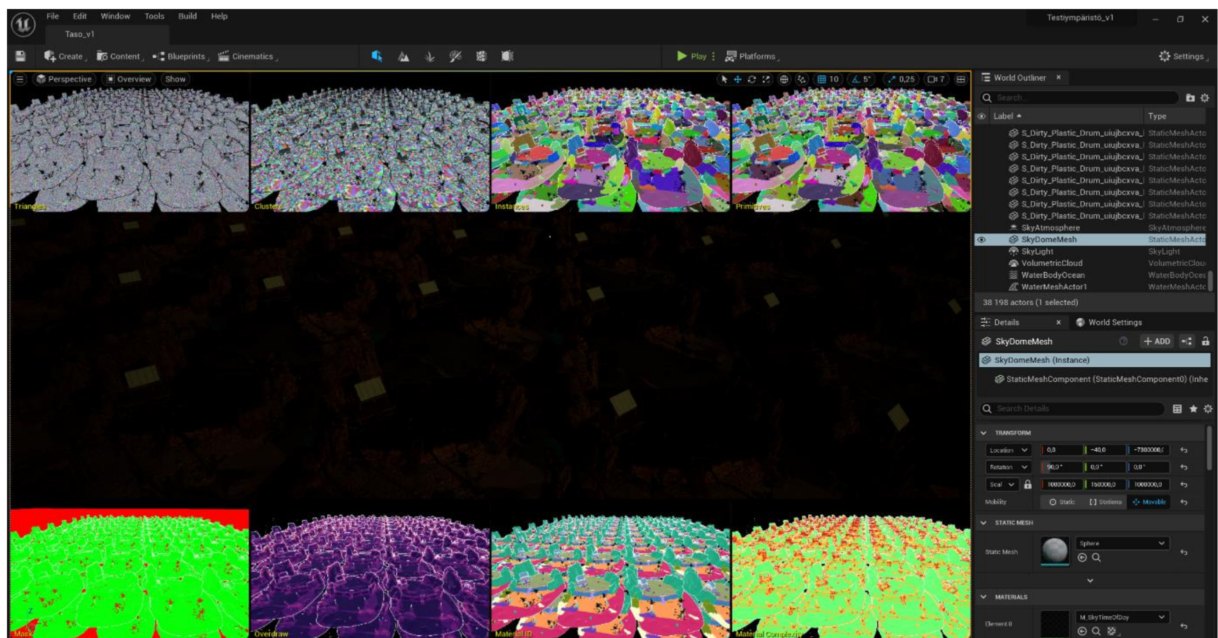
Liite 1. Utah Teapot. Martin Newell. (1975)



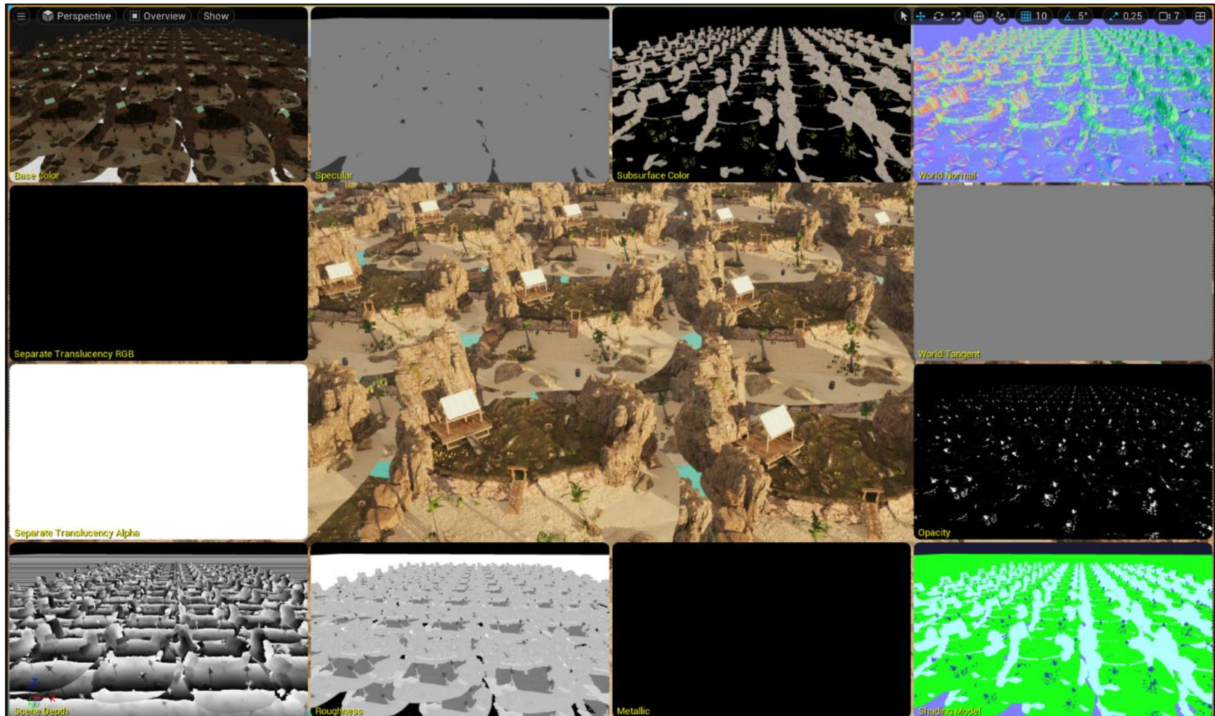
Liite 2. Testiympäristön alkuvaihe



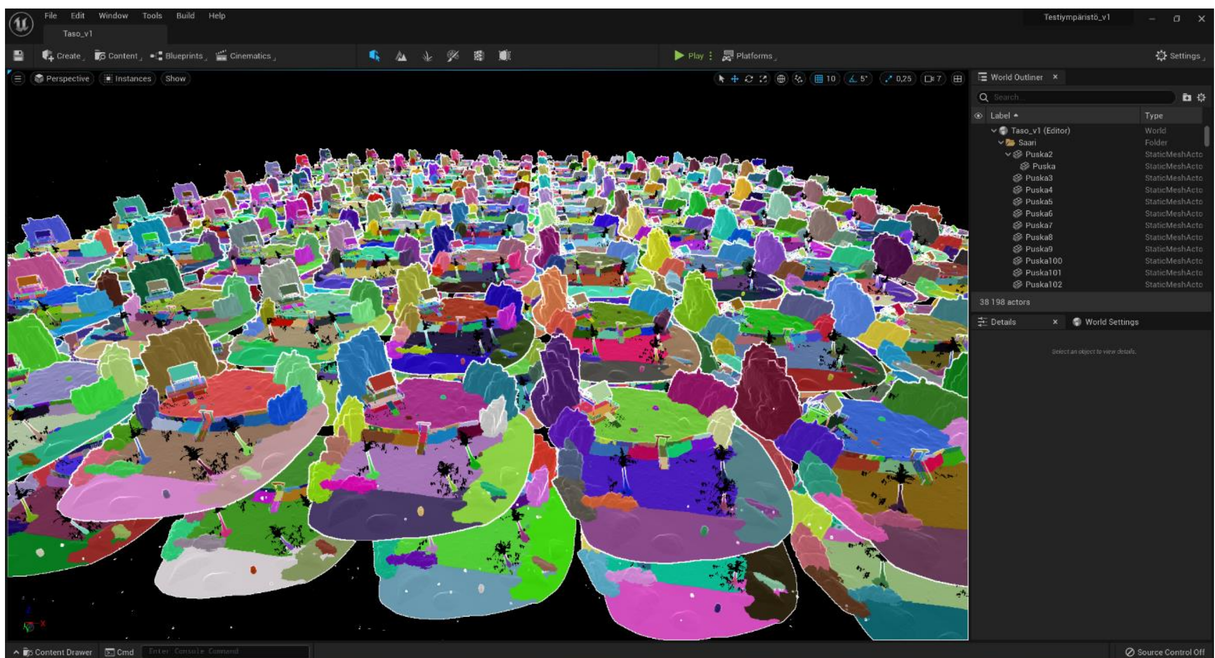
Liite3. Nanite visualisaatio saari monistettuna Unreal Engine 5.



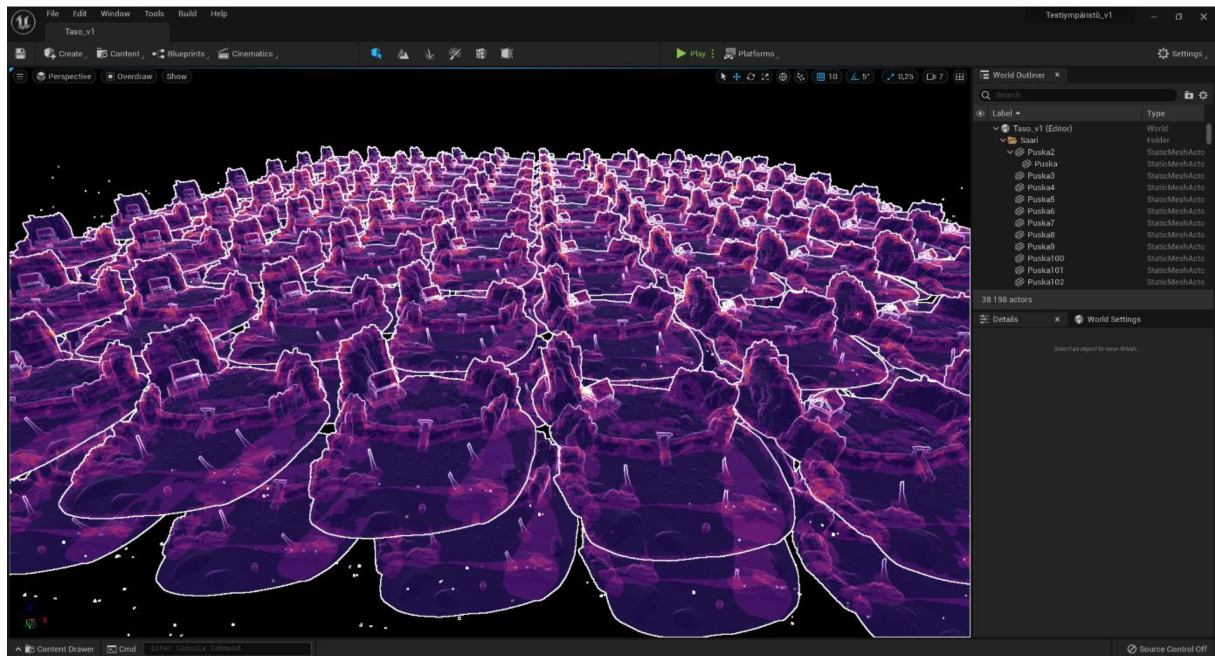
Liite 4. Varjostuskartat visualisaatio



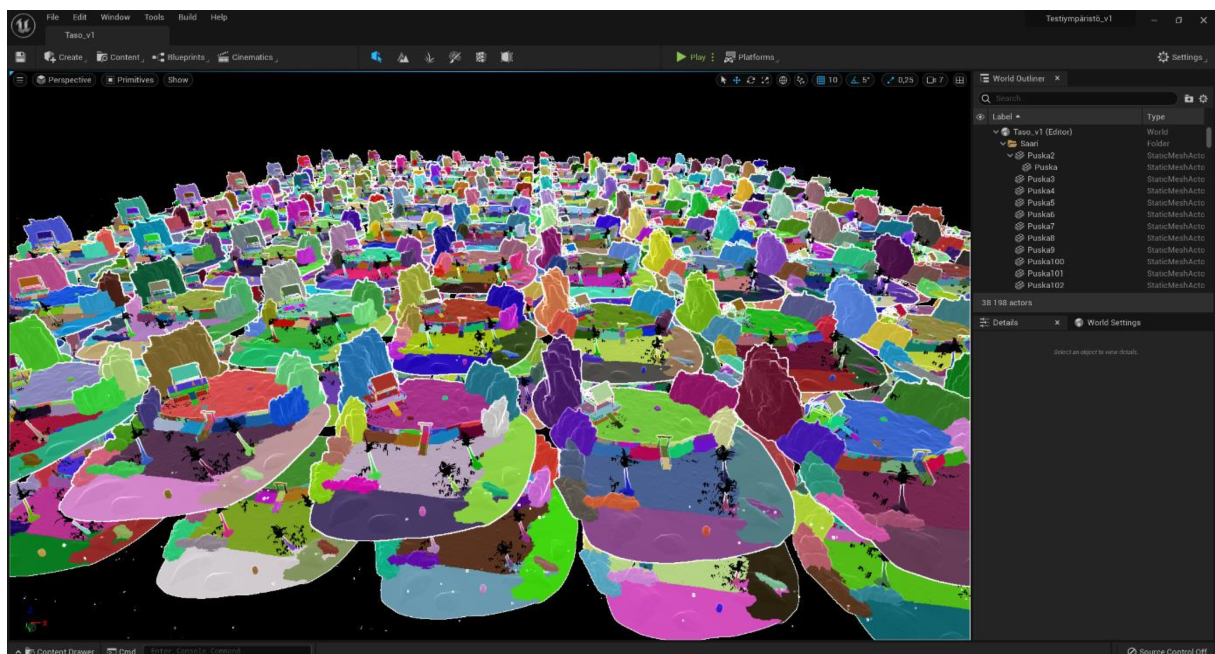
Liite 5. Instanssit



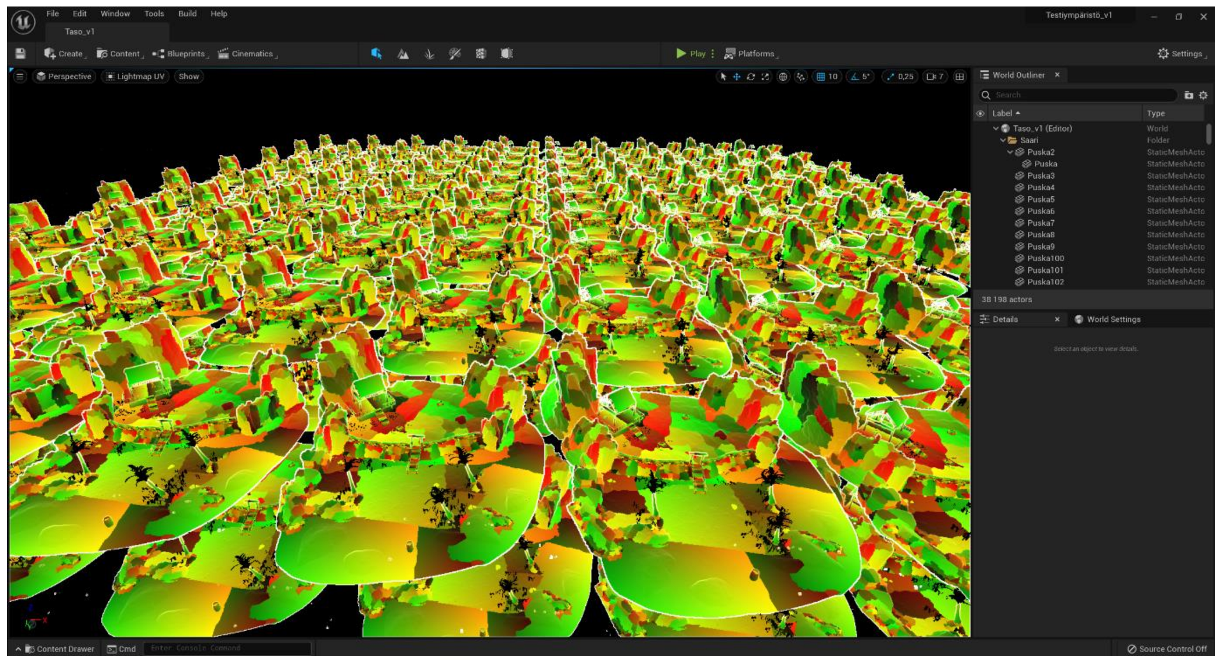
Liite 6. Rasterointi päällekkäisyydet



Liite 7. Primitiivit



Liite 8. UV kartta



Liite 9. Klusterit