Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Aarne Hällstén

Jatkuva yksityiskohtien taso peligrafii-kassa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

10.11.2020

Tekijä Otsikko Sivumäärä Aika	Aarne Hällstén Jatkuva yksityiskohtien taso peligrafiikassa 31 sivua 10.11.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli perehtyä jatkuvaan yksityiskohtien tasoon peligrafiikassa. Työssä tutustuttiin keskeiseen termistöön, siihen, mitä jatkuva yksityiskohtien taso pitää sisällään, ja siihen, missä sitä voidaan hyödyntää.

Insinöörityössä saatujen tietojen avulla luotiin Unity-pelimoottorilla useita testiympäristöjä, joissa hyödynnettiin jatkuvaa yksityiskohtien tasoa. Näillä testiympäristöillä suoritettiin mittauksia Fraps-ohjelmalla ja niiden tuloksia dokumentoitiin. Tärkeimpiä mittaustuloksia olivat kuvataajuuden eri asteet. Tuloksia vertailtiin vastaavista testeistä ilman jatkuvaa yksityiskohtien tasoa saatuihin tuloksiin.

Tuloksista voitiin päätellä, että mitä pienempi testiympäristö oli kyseessä, sitä paremmin ilman jatkuvaa yksityiskohtien tasoa tehdyt testit suoriutuivat. Kun mittauksia tehtiin suuremmilla testiympäristöillä, jatkuvalla yksityiskohtien tasolla tehdyt mittaukset suoriutuivat paremmin.

Avainsanat	LOD, CLOD, DLOD, Unity

Author Title Number of Pages Date	Aarne Hällstén Continuous Level Of Detail in game graphics 31 pages 10 November 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communcations Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The goal of this Bachelor's thesis was to become acquainted with Continuous Level Of Detail or abbreviated CLOD. The work introduces the essential terminology, what CLOD contains and where it can be utilized.

Using the information obtained during the work process, several test environments were created with Unity game engine, which utilized CLOD. Benchmarks were performed on these test environments with the Fraps program and these results were documented. The most important benchmarking results were the differences in frame rates. These results were compared with those obtained from similar benchmarks without the use of CLOD.

From the results, it could be concluded that the smaller the test environment, the better the tests performed without CLOD. When benchmarking was done in larger test environments, CLOD performed better.

Keywords	LOD, CLOD, DLOD, Unity

Sisällys

Lyhenteet ja käsitteet

CLOD	Continuous Level Of Detail. Jatkuva yksityiskohtien taso, objektin yksityiskohtien taso muuttuu jatkuvasti.
DLOD	Discrete Level Of Detail. Erillinen yksityiskohtien taso, objektista on tehty muutama erillinen versio eritasoisilla yksityiskohdilla ja sitä muutetaan tarvittaessa.
FPS	Frames Per Second. Kuvia sekunnissa, eli näytölle piirrettyjen kuvien määrä jokaisena sekuntina.
LOD	Level Of Detail. Yksityiskohtien taso, eli kuinka yksityiskohtainen jokin objekti on.
Polygoni	Vertekseistä, reunoista ja tasoista koostuva monikulmio.
Renderöinti	Kuvan luominen tietokoneella kaksi- tai kolmiulotteisesta mallista.
Verteksi	Tietokonegrafiikasta puhuttaessa verteksi tarkoittaa pientä pistettä tai tietorakennetta. Verteksiin voidaan tallentaa esimerkiksi sen sijainti, väri tai normaalivektori. Kappaleet koostuvat yleensä useista vertekseistä. Esimerkiksi kuutiossa on kahdeksan verteksiä, yksi kussakin kulmassa.

1 Johdanto

Insinöörityössä perehdytään käsitteeseen CLOD: mitä kaikkea se sisältää, mitä sen ympäriltä löytyy ja mihin sitä voidaan hyödyntää. CLOD-lyhenne tulee englanninkielisestä termistä Continuous Level Of Detail eli jatkuva yksityiskohtien taso. Käytännössä se tarkoittaa sitä, että esimerkiksi peleissä olevat objektit näytetään eri määrällä yksityiskohtia riippuen siitä, kuinka kaukana katsoja niistä on. CLOD ja yleisesti LOD ovat tärkeitä aiheita, koska tietokoneet ja laitteistot eivät välttämättä pysty suorittamaan ohjelmia tai pelejä, joissa on paljon yksityiskohtaisia objekteja.

Työssä selvitettyjen asioiden pohjalta rakennetaan Unity-pelimoottorilla useita testimaailmoja hyödyntäen jatkuvaa yksityiskohtien tasoa. Maailmoille tehdään suorituskyvyn mittaustestejä käyttäen Fraps-ohjelmaa. Näitä tuloksia vertaillaan ilman jatkuvaa yksityiskohtien tasoa toteutettujen maailmojen testituloksien kanssa ja analysoidaan sitä, kumpi toteutus on tehokkaampi.

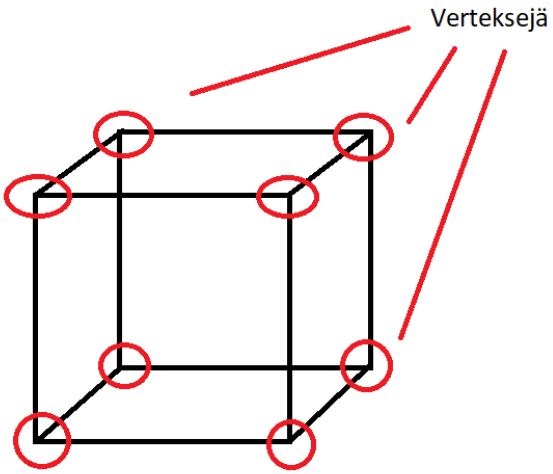
Aluksi käydään läpi tärkeitä käsitteitä, kuten LOD, CLOD ja DLOD. Luvussa 3 tutustutaan siihen, mihin edellisessä luvussa läpikäytyjä asioita voidaan hyödyntää. Luvuissa 4 ja 5 tutustutaan pintapuolin Unity-pelimoottorin LOD-järjestelmään ja polygonien generointiin. Viimeisessä luvussa käydään tarkasti läpi testimaailman rakennusprosessi, mitä kaikkea sitä tehdessä tuli ottaa huomioon, ja näytetään valmista maailmaa. Luvun lopuksi käydään läpi maailmoille tehdyt suorituskykytestit ja analysoidaan niiden tuloksia.

2 Yksityiskohtien taso

LOD eli level of detail tarkoittaa objektin yksityiskohtaisuutta. Peleissä objektien yksityiskohtia säädellään tarkemmiksi tai epätarkemmiksi, riippuen siitä, kuinka kaukana ne ovat katsojasta tai kuinka tärkeitä ne ovat pelin kannalta. Kaukana olevien objektien yksityiskohdat säädetään minimiin, ja näin saadaan pelin suorituskykyä parannettua, samalla kuitenkaan pelaaja ei välttämättä edes huomaa sitä. [1; 2; 3.] Kuva 1 havainnollistaa eritasoisia yksityiskohtia.

Kuva 1. Kani eritasoisine yksityiskohtineen [4].

Yksityiskohtia vähennetään yleisimmin muuntamalla ja vähentämällä objektin verteksejä, erilaisia polygonimallin generointialgoritmejä apuna käyttäen. Verteksi tarkoittaa pientä pistettä tai tietorakennetta, joista polygonit koostuvat. Verteksiin voidaan tallentaa muun muassa sen sijainti tai väri. Esimerkiksi kuvassa 2 on ympyröitynä kaikki kuution verteksit. Verteksien väheneminen parantaa objektin renderöintinopeutta. [1; 2; 3.]



Kuva 2. Kuution verteksit ympyröitynä.

2.1 DLOD-järjestelmä

DLOD:ssa kirjain D vastaa englannin kielen sanaa discrete (suom. erillinen). Tämä viittaa siihen, että objektista, jonka LOD:a halutaan säädellä, on tehty valmiiksi useampi erillinen versio. Näitä versioita vaihdetaan portaittain tarpeen mukaan esimerkiksi riippuen siitä, kuinka kaukana objekti on katsojasta. [1; 2; 3.]

DLOD:ssa on laskettu LOD:n tasot jo valmiiksi ja ainut suoritusaikana vaadittava lasku on etäisyys katsojaan. Tästä syystä se on usein kevyempi suorituskykyvaatimuksiltaan verrattuna CLOD:iin. [1; 2; 3.]

Koska objektista on tehty kokonaan erillisiä versioita, ongelmaksi DLOD:ssa voi tulla niiden vaihtaminen. Objektin siirtyessä seuraavalle tasolle katsoja saattaa huomata sen, mikä vaikuttaa negatiivisesti peli- tai katsomiskokemukseen. [1; 2; 3.]

2.2 CLOD-järjestelmä

CLOD:ssa toisin kuin DLOD:ssa ei ole valmiiksi tehtyjä erillisiä versioita objektista. CLOD:n periaate on jatkuvuus (engl. continuous), eli objektille tehdään jatkuvasti laskuja hyödyntäen polygonimallin generointialgoritmeja, joilla määritetään, kuinka yksityiskohtainen se on. Objektin LOD vaihtelee tasaisesti riippuen sen etäisyydestä katsojaan. [3; 5.]

CLOD vaatii suoritusaikana paljon laskemista, siksi se voi joskus olla jopa raskaampi suorituskykyvaatimuksiltaan kuin ilman minkäänlaista LOD:a toteutettu ohjelma. Hyötynä CLOD:ssa on se, että objektin yksityiskohtien muuttuminen on tasaista eikä siinä ole hyppyä seuraavalle tasolle, minkä katsoja voi huomata. [3; 5.]

CLOD:n jatkeena voidaan käyttää katsomissuunnasta riippuvaa LOD:a. Siinä objektin eri puolet näytetään eritasoisilla yksityiskohdilla riippuen siitä, mistä suunnasta sitä katsotaan. Katsojan puolella yksityiskohdat ovat tarkkoja, kun taas objektin takana, mihin katsoja ei näe, yksityiskohdat ovat huomattavasti epätarkempia. [3.]

3 LOD-järjestelmän käyttö

LOD:n käyttö on yleistä erityisesti videopeleissä. Pelintekijät pyrkivät luomaan pelaajalle rikkaan ja immersiivisen pelikokemuksen. Tätä vaikeuttaa monesti laitteiston tehottomuus, minkä takia pelin suorituskykyvaatimuksia on pyrittävä pienentämään. Ennen LOD:n käytön yleistymistä peleissä kaukana olevat objektit jätettiin usein kokonaan renderöimättä, mikä rikkoi pelaajan immersiota, kun ne ilmestyivät näkyviin tyhjästä. [5.]

Toinen tapa parantaa suorituskykyä oli häivyttää kaukana olevia objekteja sumun avulla. Mitä kauempana objekti on katsojasta, sitä sakeampi sumu on ja lopulta objekti katoaa kokonaan näkyvistä. [5.]

Yksi ensimmäisistä LOD:a laajasti hyödyntäneistä peleistä on Spyro The Dragon. Siinä kustakin tasosta on luotu kaksi versiota. Ensimmäinen versio on pelaajalle läheltä näytettävä maailma, jossa on runsaasti yksityiskohtia ja eri tekstuureita. Toisessa versiossa tason polygonien määrä on huomattavasti matalampi kuin ensimmäisessä, ja siinä eri objektit on maalattu vain yhdellä värillä teksturoinnin sijaan. Kuvissa 3 ja 4 nähdään otokset Spyron pelimaailmasta, ensin pienellä määrällä polygoneja ja sitten suuremmalla määrällä. Peli on ohjelmoitu vaihtelemaan näiden kahden version välillä riippuen etäisyydestä pelaajaan. Tällä menetelmällä maailma saatiin yhtenäisen tuntuiseksi, kuitenkin samalla peli pyöri sulavasti PlayStation 1 -konsolilla. [7.]

Kuva 3. Spyro the Dragon -peli pienellä polygonimäärällä ja vähillä tekstuureilla [7].

Kuva 4. Spyro the Dragon -peli suurella polygonimäärällä ja kaikilla tekstuureilla [7].

3D-kaupunkimallit

3D-kaupunkimallit ovat digitaalisia kolmiulotteisia malleja kaupunkien maastosta, kasvillisuudesta ja infrastruktuurista. Malleja voidaan hyödyntää kaupunkisuunnittelussa. Kaupunkeja mallinnettaessa noudatetaan yleensä CityGLM-standardin mukaista LOD:a. Standardi koostuu viidestä eritasoisesta luonnoksesta eri yksityiskohdilla. [8.]

Ensimmäinen versio LOD0 on karkea luonnos maastosta ja sen korkeuseroista. Siinä rakennukset on kuvattu vain kaksiulotteisina pohjapiirroksina. Seuraavassa LOD1-versiossa rakennukset on kuvattu laatikoina. Laatikot pyritään mallintamaan oikeilla mittasuhteilla lukuun ottamatta kattoa, joka on tällä tasolla vielä tasainen. Kasvillisuutta mallinnettaessa näytetään vain kaikkein olennaisimmat kasvit. [8.]

LOD2 on mallin kolmas versio. Siinä taloihin lisätään alustavat katon ja seinien muodot. Tällä tasolla mittasuhteiden ja sijainnin tulee olla hyvin lähellä lopullisia arvoja. Kasvillisuutta mallinnetaan enemmän. [8.]

Neljännessä versiossa LOD3 kaupungista on mallinnettu tarkka malli. Taloihin on lisätty ovet ja ikkunat, katon muodot ovat tarkat ja mittasuhteet ja sijainti samat kuin lopullisessa versiossa. Alueen kasvillisuus on myös mallinnettu tarkasti. [8.]

Viimeiseen versioon LOD4 on LOD3:n yksityiskohtien lisäksi mallinnettu talojen sisätilat. Mallinnettujen talojen sisällä ovat huoneet, portaikot ja jopa huonekalut. Kuvasta 5 nähdään visuaalinen esitys LOD:n eri tasoista. [8.]

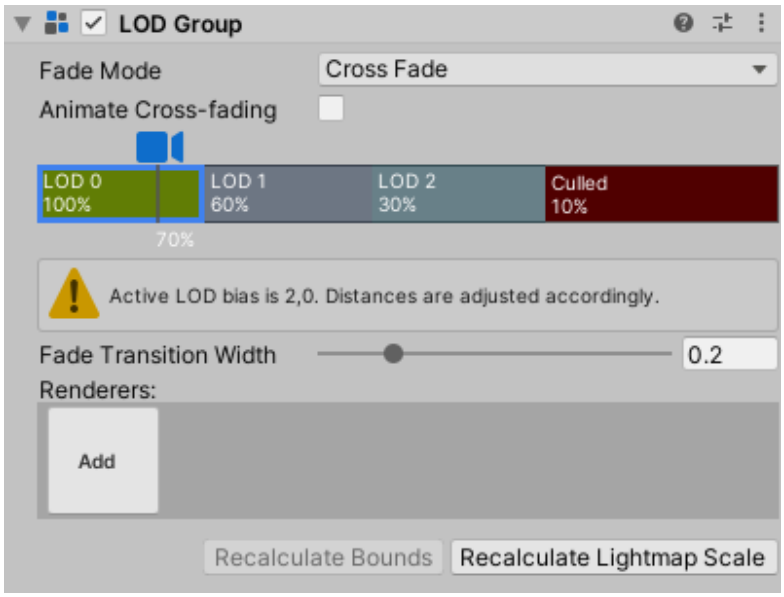
Kuva 5. CityGLM-standardin mukaiset LOD-tasot kaupunkimallinnuksessa [9].

4 LOD-järjestelmä Unity-pelimoottorissa

Unity on vuonna 2005 Unity Technologiesin julkaisema pelimoottori. Moottorilla pystytään kehittämään kaksi- ja kolmiulotteisia pelejä. Unityn käyttö on ilmaista yksityisille henkilöille ja yrityksille, joiden tulot ovat alle 100 000 € vuodessa. [10; 11; 12.]

Unityssa on valmiina sen oma LOD-järjestelmä. Sen saa otettua käyttöön lisäämällä Unityn GameObjectiin LOD group -nimisen komponentin. Komponentti toimii DLOD-periaatteella, eli siihen valitaan valmiit objektit eri määrillä yksityiskohtia, kuitenkin enintään kahdeksan eri objektia. [13; 14.]

Vaihdot objektien välillä tapahtuvat sillä perusteella, kuinka monta prosenttia objekti vie tilaa näytöltä. Vaihtojen sulavuutta on parannettu häivyttämällä objekti pois vähitellen ja samalla häivyttämällä uutta objektia sen tilalle. Kuvasta 6 nähdään, minkälainen on LOD Group -komponentin käyttöliittymä. [13; 14.]



Kuva 6. Unityn LOD Group -komponentin käyttöliittymä [14].

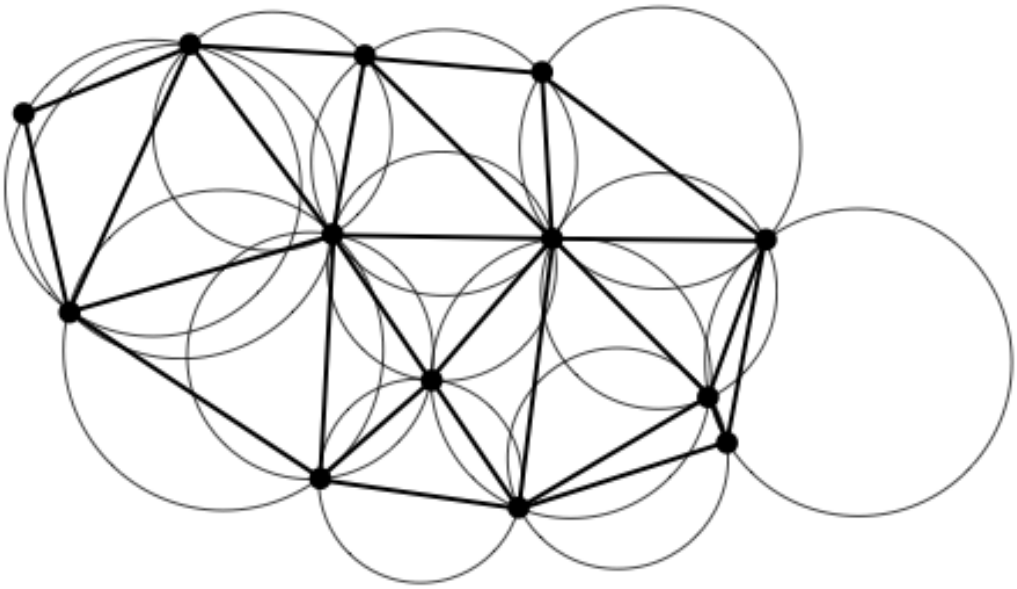
Ludvig Arlebrink ja Fredrik Linde Blekinge Institute of Technologystä tekivät vuonna 2018 tutkimuksen DLOD:sta Unity-pelimoottorissa. Tutkimuksessa luotiin maailma, jossa oli kamera, ja sinne generoitiin 8 000 kania. Kamera laitettiin kiertämään kaneja ympäri, ja samalla mitattiin suorituskykyä. Suorituskykymittauksia tehtiin useita kertoja Unityn LOD-järjestelmän ollessa käytössä, ja tällöin jokaisen kanin yksityiskohdat vaihtelivat viiden eri tason välillä. Yksityiskohtaisimmassa kanissa oli yhteensä 69 630 kolmiota, kun taas vähiten yksityiskohtia sisältävässä niitä oli vain 76.Toiset testit tehtiin ilman Unityn LOD-järjestelmää. Silloin jokainen kani pysyi koko testin ajan yksityiskohtaisimmassa versiossa. [15.]

Tehdyistä testeistä Unityn LOD-järjestelmä suoriutui huomattavasti paremmin. Sen keskimääräinen FPS-lukema (frames per second) testin aikana oli jopa yli 20 kertaa korkeampi kuin ilman sitä tehdyissä testeissä. Ilman LOD:a tehtyjen testien positiivinen puoli oli se, että se käytti tietokoneen keskusmuistia keskimäärin vain 3/4 siitä, mitä Unityn LOD-järjestelmä käytti. [15.]

5 Polygonimallin generointi

Polygonimallin generointi on oleellinen osa työprosessia, kun LOD-järjestelmää lähdetään toteuttamaan käytännössä. Mallit generoidaan valmiiksi eri määrillä yksityiskohtia, ennen kuin ohjelmaa käytetään ja malleja tarvitaan, jos käytössä on DLOD. Silloin kun käytössä ovat CLOD-mallit, joudutaan kuitenkin generoimaan ohjelman suorittamisen aikana. Polygonimalli tarkoittaa vertekseistä, reunoista ja pinnoista muodostuvaa monitahokasta. Polygonimallin pinnat koostuvat pienemmistä osista, jotka ovat yleensä kolmioita tai nelikulmioita. Mallin jokaisessa verteksissä on kyseisen verteksin x-, y- ja z-koordinaatit, ja samoin pinnoista löytyvillä tiedoilla saadaan kappaleesta renderöintivaiheessa laskettua valotukset ja varjostukset. [16; 17; 18.]

Polygonimalleja generoitaessa hyödynnetään yleensä Delaunayn kolmiomittausta (engl. Delaunay triangulation) ja jotain algoritmia verteksien lisäämiseen. Delaunayn kolmiomittaus on Boris Delaunayn kehittämä algoritmi. Siinä erillisille pisteille tasolla kolmiomitataan siten, että pisteistä piirrettyjen kolmioiden kehän sisälle ei tule ylimääräisiä pisteitä. Kuvasta 7 voidaan katsoa visuaalinen havainnollistus asiasta. [19.]



Kuva 7. Delaunayn kolmiomittaus [20].

5.1 Ruppertin algoritmi

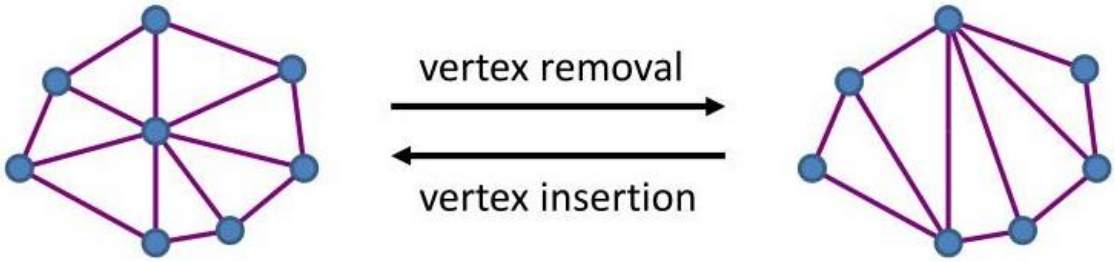
Ruppertin algoritmi on Jim Ruppertin 1990-luvun alussa kehittämä. Sen toimintaperiaate on kolmiomittausta hyödyntämällä generoida muoto, jossa on vain hyvänlaatuisia kolmioita. Kolmio luokitellaan algoritmin mukaan hyvänlaatuiseksi, jos sen kehän suhde lyhyimpään reunaan on pienempi kuin käyttäjän määrittelemä kynnysarvo. Kuvassa 8 olevasta Ruppertin algoritmin pseudokoodista nähdään sen toimintaa tarkemmin. [20; 21.]

```
function RuppertsAlgorithmPse(points, segments, threshold)
    X = Triangulation(points)
    Y = Poor quality triangles and encroached segments

    while Y is not empty
        if Y contains a segment s
            insert the midpoint of s into X
        else Y contains poor quality triangle t
            if the circumcenter of t encroaches a segment s
                add s to Y
            else
                insert the circumcenter of t into X
            end if
        end if
        update Y
    end while

    return X

    end RuppertsAlgorithmPse
```

Kuva 8. Pseudokoodi Ruppertin algoritmista [20; 21].

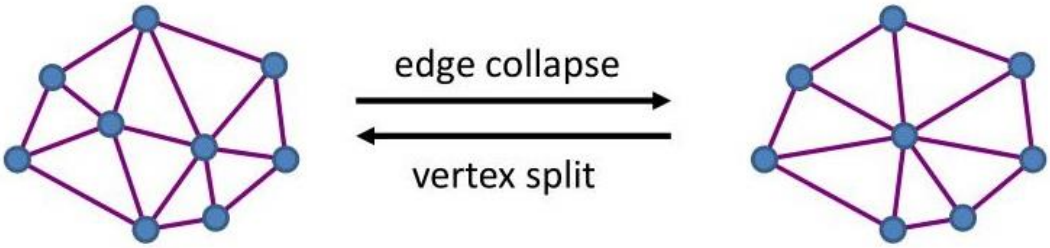
5.2 Polygonimallin yksinkertaistaminen

Polygonimallin yksinkertaistaminen tarkoittaa käytännössä sitä, että mallista poistetaan tai yhdistetään verteksejä, minkä seurauksena myös pinnat ja reunat vähenevät. Kun verteksi poistetaan, sen paikalle jäävä reikä täytetään uusilla kolmioilla kolmiomittauksen avulla. Uusien kolmioiden määrä on kaksi vähemmän kuin alkuperäinen kolmioiden määrä, eli yhtä verteksiä varten poistetaan kaksi kolmiota. Kuvassa 9 on visualisoituna verteksin poistaminen. [22; 23; 24; 25.]



Kuva 9. Verteksin poistaminen [25].

Verteksejä yhdistäessä kaksi verteksiä yhdistetään yhdeksi verteksiksi. Tätä operaatiota tehdessä pätee sama sääntö kuin verteksejä poistaessa. Jokaista verteksien yhdistymistä kohden kolmioiden määrä pienenee kahdella. Kuva 10 visualisoi verteksien yhdistämistä. [22; 23; 24; 25.]



Kuva 10. Verteksien yhdistäminen [25].

Näillä operaatiolla voidaan yksinkertaistaa kaikkia polygoneja. On kuitenkin olemassa tiettyjä sääntöjä, joita kannattaa noudattaa polygonia yksinkertaistettaessa. Jos verteksi on kaukana muista vertekseistä, sitä ei kannata poistaa, jotta polygonin yleinen muoto säilyisi mahdollisimman lähellä alkuperäistä. [22; 23; 24; 25.]

6 Testisovelluksen toteutus

6.1 Tavoitteet

Insinöörityöprojektin tavoitteena oli toteuttaa CLOD-järjestelmä Unity-pelimoottorilla, CLOD:n toimintaperiaatteen oppiminen ja ohjelmointitaitojen kehittäminen. Tavoitteena

oli saada CLOD toimimaan aluksi yksinkertaisilla muodoilla, kuten kuutiolla ja pallolla. Projektin lopussa tarkasteltiin suorituskykyä CLOD:a käytettäessä verrattuna siihen, kun CLOD ei ollut käytössä. Projektissa ohjelmointikielenä toimi C#.

Valmista projektia on tarkoitus pystyä hyödyntämään muissa projekteissa tai peleissä, joko suoraan sisällyttämällä valmiita skriptejä uuteen projektiin tai soveltamalla niiden toimintaa.

6.2 Lähtökohta

Projektiin valittiin toteutusalustaksi Unity-pelimoottori ja C#-ohjelmointikieli, koska ne olivat tekijälle ennestään tuttuja kehitystyökaluja. Unityssa on valmiina yksityiskohtien säätelyyn vain DLOD-menetelmä, joten CLOD:n toteuttaminen toimivasti on hyödyllinen päämäärä.

C# on Microsoftin vuonna 2000 julkaisema oliopohjainen ohjelmointikieli. Se on käytetyin ohjelmointikieli Unityssä ja maailmalla kokonaisuudessaan neljänneksi käytetyin. [12; 26; 27.]

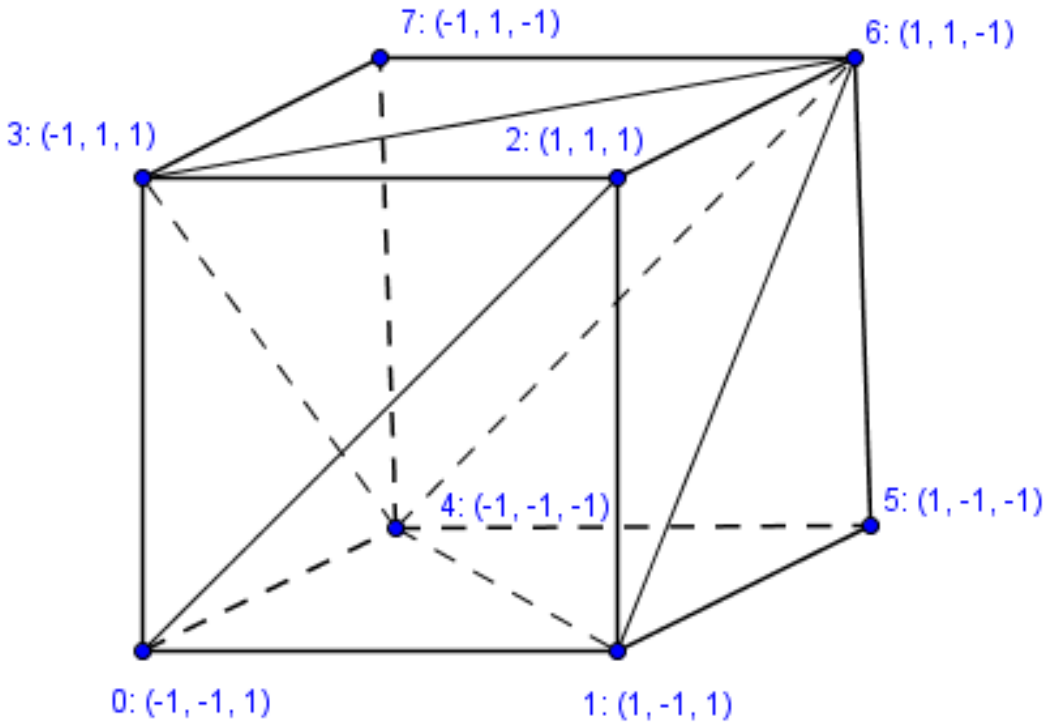
Projekti aloitettiin tutustumalla siihen, miten eri peleissä oli toteutettu CLOD- tai DLOD-menetelmiä ja miettimällä, miten niiden toteutusta voitaisiin hyödyntää tässä projektissa. Tutustumisen aikana nopeasti selvisi, että suurimmassa osassa peleistä ei ole hyödynnetty yhtään CLOD-menetelmää. Jos pelissä oli hyödynnetty LOD:a millään tasolla, niin siinä oli käytetty DLOD:a.

6.3 Projektin eteneminen

Unityssä on valmiina käytettävissä useita yksinkertaisia muotoja. Näihin muotoihin sisältyvät kuutio ja pallo, joista projektin Unity-osio aloitettiin.

Ongelmaksi kuitenkin muodostui se, että valmiiden muotojen verteksien ja kolmioiden lukumäärää ei pystytä muokkaamaan. Tämä tarkoittaa, että muodot joudutaan generoimaan alusta alkaen koodin kautta.

Yksinkertainen kuutio luotiin tässä vaiheessa manuaalisesti kirjoittamalla kuution verteksien sijainnit Unityn Vector3[]-tyyppiseen listaan. Kun verteksien sijainnit ovat tiedossa, niiden perusteella on helppo piirtää niistä kolmiot. Unityssä kolmion näkyvä puoli on se, kummassa verteksit on kirjattu myötäpäivään. Jokainen kolmio siis koostuu kolmesta verteksistä, jotka ovat tallennettuna Unityn Int[]-tyyppiseen listaan. Kuvassa 11 näkyvät yksinkertaisen kuution verteksit ja niiden koordinaatit. Kuvissa 12 ja 13 kuution verteksit ja kolmiot on tallennettu Unityn Vector3[]- ja int[]-tyyppisiin listoihin.



Kuva 11. Yksinkertaisen kuution verteksit ja niiden koordinaatit [28].

```
Vector3[] vertices = new Vector3[]
{
    new Vector3(-1,1,1),//0
    new Vector3(1,1,1),//1
    new Vector3(-1,-1,1),//2
    new Vector3(1,-1,1),//3
    new Vector3(1,1,-1),//4
    new Vector3(-1,1,-1),//5
    new Vector3(1,-1,-1),//6
    new Vector3(-1,-1,-1),//7

};
```

Kuva 12. Yksinkertaisen kuution verteksit Vector3[]-listassa.

```
int[] triangles = new int[]
{
0,2,3, //etu
3,1,0,

4,6,7, //taka
7,5,4,

5,7,2, //vasen
2,0,5,

1,3,6, //oikee
6,4,1,

5,0,1, //ylä
1,4,5,

2,7,6, //ala
6,3,2,
};
```

Kuva 13. Yksinkertaisen kuution kolmiot int[]-listassa.

Tällä tavalla saatiin rakennettua niin yksinkertainen kuutio kuin mahdollista, eli kuutio sisältää kahdeksan verteksiä ja kaksitoista kolmiota.

Seuraavaksi projektissa alettiin pohtia sitä, miten tämän kuution verteksien ja kolmioiden määrää pystyttäisiin lisäämään. Suoraan valmiina oleviin listoihin uusien verteksien ja kolmioiden lisääminen ei tuottanut kuutiolle haluttua dynaamista muokattavuutta tai lopputulosta.

6.4 Seinä

Kuution rakentamista lähdettiin lähestymään uudelta kannalta. Tällä toteutustavalla kuution jokainen seinä tai tahko generoitiin erikseen. Lopuksi ne liitettiin yhteen muodostamaan kuutio.

Seinällä oli oma SideCreator()-konstruktorinsa. Siinä se sai tiedon kokonaislukuna siitä, kuinka monta verteksiä seinällä haluttiin olevan, ja Vector3-tyypin muuttujana tiedon siitä, mihin suuntaan seinä osoitti. Kuvassa 14 nähdään SideCreator()-konstruktori.

```
public SideCreator(Mesh mesh, int detLvl, Vector3 localUp)
{
    this.mesh = mesh;
    this.detLvl = detLvl;
    this.localUp = localUp;

    axisVertical = new Vector3(localUp.y, localUp.z, localUp.x);
    axisHorizontal = Vector3.Cross(localUp, axisVertical);
}
```

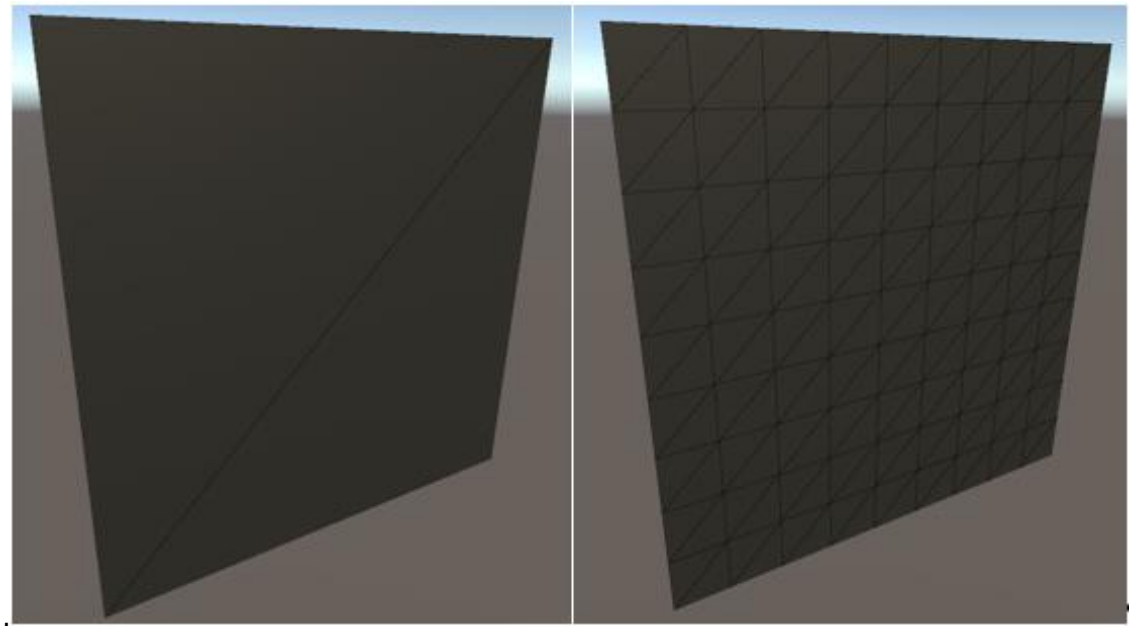
Kuva 14. SideCreator()-konstruktori.

Itse seinä generoitiin käyttämällä GenerateSide()-metodia, joka otti parametrinä kokonaisluvun siitä, kuinka monta verteksiä kyseiseen seinään haluttiin. Tästäkin metodista löytyi samat Unityn Vector3[]- ja int[]-tyyppiset listat seinän verteksien ja kolmioiden varastointia varten kuin ensimmäisen kuution generointiskriptistä. GenerateSide()-metodi asetteli silmukkalauseen avulla niin monta verteksiä ja kolmiota seinään, kuin parametrinä sille oli annettu. Kuvassa 15 näkyy osa GenerateSide()-metodia, jossa kolmiot asetetaan oikeille paikoille.

```
for (int i = 0; i < detailLevel; i++)
{
    for (int j = 0; j < detailLevel; j++)
    {
        int k = j + i * detailLevel;
        Vector2 amount = new Vector2(j, i) / (detailLevel - 1);
        Vector3 pointOnCube = localUp + (amount.x - 0.5f) * 2 *
            axisVertical + (amount.y - 0.5f) * 2 * axisHorizontal;
        vertices[k] = pointOnCube;

        if (j != detailLevel - 1 && i != detailLevel - 1)
        {
            triangles[triangleNum] = k;
            triangles[triangleNum + 1] = k + detailLevel + 1;
            triangles[triangleNum + 2] = k + detailLevel;
            triangles[triangleNum + 3] = k;
            triangles[triangleNum + 4] = k + 1;
            triangles[triangleNum + 5] = k + detailLevel + 1;
            triangleNum += 6;
        }
    }
}
```

Kuva 15. Osa GenerateSide()-metodia, jossa kolmiot asetetaan oikeille paikoilleen.

Metodia tehdessä oli otettava huomioon Unityn vaatimus siitä, että verteksit tuli kirjata myötäpäivään, jotta niiden näkyvä osuus osoittaisi oikeaan suuntaan. Kustakin seinästä löytyi myös tieto siitä, mihin suuntaan se osoitti, jotta kuudesta seinästä saataisiin koottua kokonainen kuutio. Kuvassa 16 näkyy valmiina kaksi eri määrillä verteksejä generoitua seinää.



Kuva 16. Kaksi generoitua seinää eri määrillä verteksejä ja kolmioita.

6.5 Kuutio

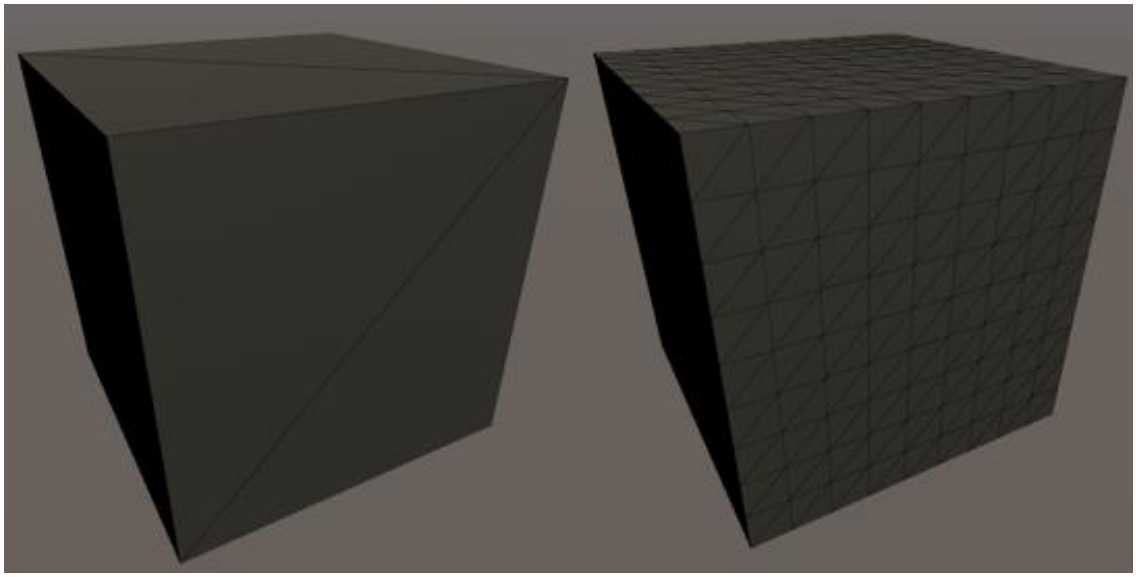
Kun yhden seinän generointi oli saatu toimimaan halutulla tavalla, oli aika koota useasta seinästä kuutio. Tämä tehtiin käyttämällä aikaisemmin tehtyä GenerateSide()-metodia niin monta kertaa, että haluttu määrä seiniä saatiin generoitua, eli kuution tapauksessa kuudesti.

Tässä vaiheessa tehdyllä Initialize()-metodilla valmisteltiin seinien generointia kertomalla niille, mihin suuntaan niiden haluttiin osoittavan, ja antamalla niille alkuarvo verteksien määrästä. Initialize()-metodia kutsuttiin niin monta kertaa kuin seiniä haluttiin luoda. Jokaiselle seinälle annettiin tässä myös Unityn MeshRenderer-komponentti, jolla ne renderöitiin. Nämä seinien alut varastoitiin SideCreator[]-tyyppiseen listaan. Kuvassa 17 nähdään osa Initialize()-metodia.

```
sideCreator = new SideCreator[6];
Vector3[] directions = { Vector3.up, Vector3.down,
Vector3.left, Vector3.right, Vector3.forward, Vector3.back };
sideCreator[i] = new SideCreator(meshFilters[i].sharedMesh,
detailAmount, directions[i]);
```

Kuva 17. Osa Initialize()-metodia.

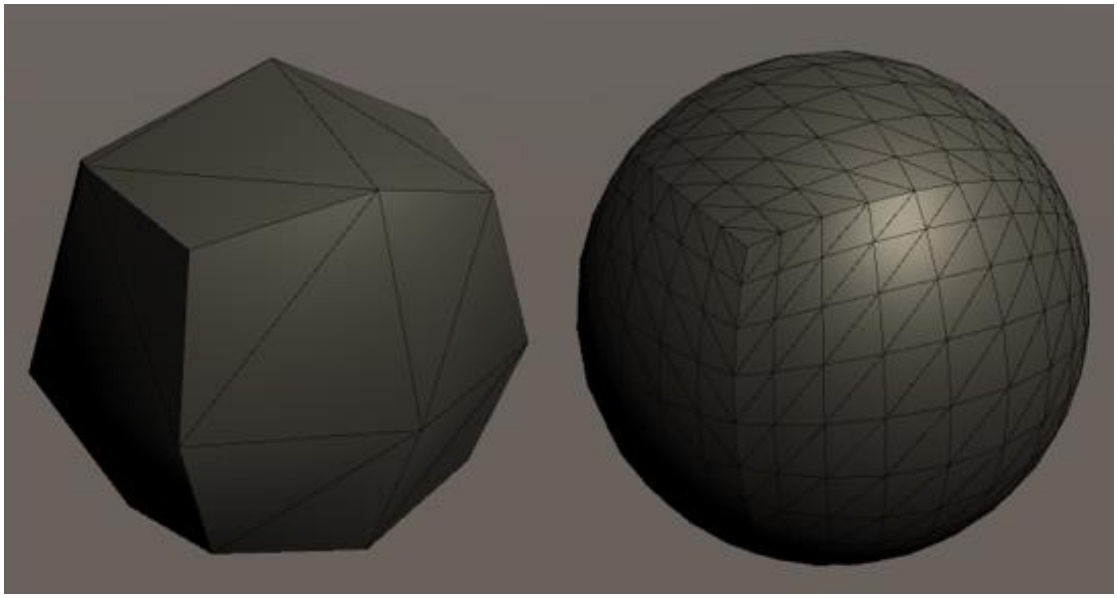
Valmistelujen jälkeen jokainen SideCreator[]-listan alkio kutsui GenerateSide()-metodia generoidakseen oman seinänsä, ja näin saatiin valmis kuutio. Kuvassa 18 näkyy kaksi valmista kuutiota, joille annettiin eri määrä verteksejä generoitaessa.



Kuva 18. Valmiit kuutiot eri määrällä verteksejä.

6.6 Pallo

Pallon generoinnissa käytettiin pohjana kuution generointiin käytettyä koodia. GenerateSide()-metodilla tehtiin kuusi seinää, niihin lisättiin Unityn MeshRenderer-komponentti ja ne yhdistettiin kuutioksi samoin kuin aikaisemminkin. Seinät saatiin tehtyä kaareviksi käyttämällä Unitystä valmiina löytyvää Vector3:n muuttujaa normalized. Se palautti Vector3-muotoisesta verteksistä yhden yksikön mittaisen vektorin, ja kun tämä tehtiin jokaiselle verteksille jokaisessa seinässä, saatiin lopputulokseksi pallo. Kuvassa 19 näkyy valmiina kaksi eri määrällä verteksejä generoitua palloa.



Kuva 19. Valmiit pallot eri määrällä verteksejä.

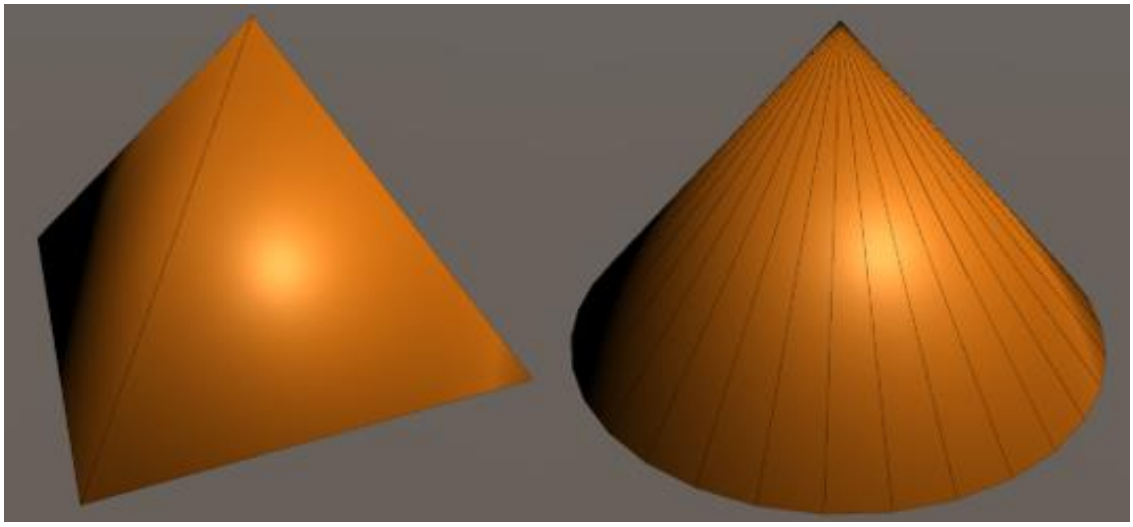
6.7 Kartio

Kartion luomista varten tehtiin GenerateCone()-metodi, joka otti parametrinä kokonaisluvun halutusta verteksien määrästä. Metodi laski kartion laskentakaavoilla ja halutulla verteksien määrällä, mihin kukin verteksi ja sitä kautta kolmio sijoitetaan. Verteksit ja kolmiot tallennettiin Unityn Vector3[]- ja int[]-tyyppisiin listoihin. Valmiille kappaleelle annettiin Unityn MeshRenderer-komponentti sen renderöintiä varten. Kuvassa 20 nähdään osa GenerateCone()-metodia, jossa lasketaan verteksit ja asetetaan kolmiot paikoilleen.

```
float alpha = Mathf.PI * 2 / detailAm;
float omega = alpha * 0.5f;
for (int i = 0; i < detailAm; i++)
{
    vertices[i + 2] = new Vector3(Mathf.Cos(i * alpha + omega),
    0, Mathf.Sin(i * alpha + omega));
    int k = i;

     if(i == 0)
     {
         k = detailAm + 1;
     }
     else
     {
        k = i + 1;
     }

    int triangleNum = i * 6;
    triangles[triangleNum + 0] = i + 2;
    triangles[triangleNum + 1] = k;
    triangles[triangleNum + 2] = 0;
    triangles[triangleNum + 3] = i + 2;
    triangles[triangleNum + 4] = 1;
    triangles[triangleNum + 5] = k;
}
```

Kuva 20. Osa GenerateCone()-metodia.

Näin saatiin generoitua ympyräkartio, ja jos GenerateCone()-metodille annettiin riittävän pieni määrä verteksejä, ympyräkartio saatiin muutettua särmäkartioksi eli pyramidiksi. Kuvassa 21 näkyy valmiina kaksi kartiota eri määrillä verteksejä generoituna.



Kuva 21. Valmiit kartoit eri määrillä verteksejä.

6.8 Pelaaja

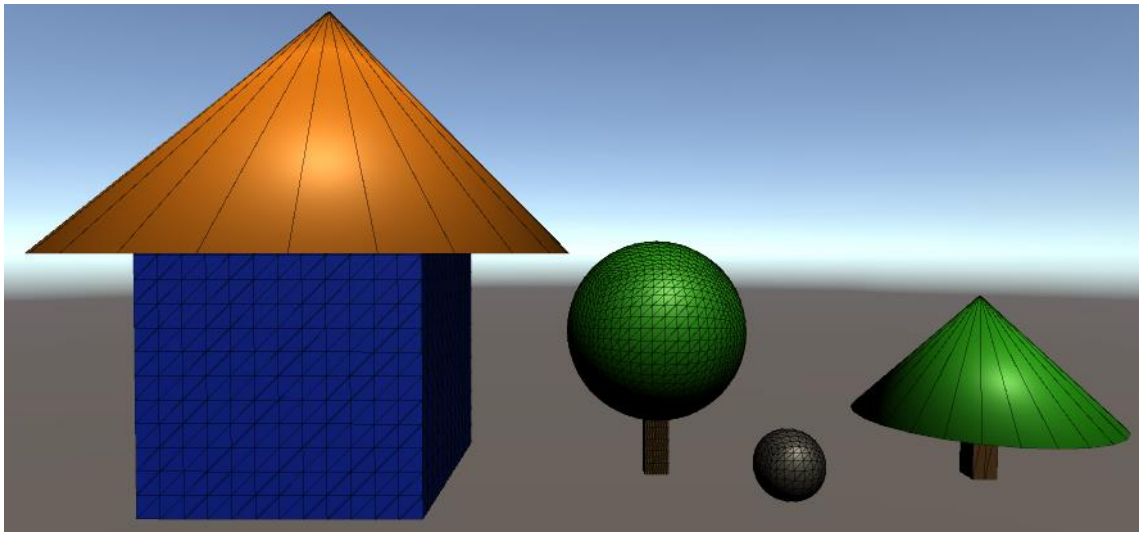
Pelaajaksi tehtiin tavallinen kuutio, jota pystyttiin liikuttelemaan nuolinäppäimillä. Jokaisesta tähän asti tehdystä muodosta mitattiin jatkuvasti etäisyyttä pelaajakuutioon. Etäisyyden perusteella kunkin muodon generointimetodi generoi muodon uudestaan uudella määrällä verteksejä.

6.9 Testimaailman rakentaminen

Suorituskyvyn mittausta varten rakennettiin kaksi maailmaa, jossa pelaajalla liikutaan. Ensimmäisessä maailmassa jokaisella objektilla oli käytössä CLOD, joka muunteli objekteja itse asetettujen minimi- ja maksimiverteksimäärien välillä.

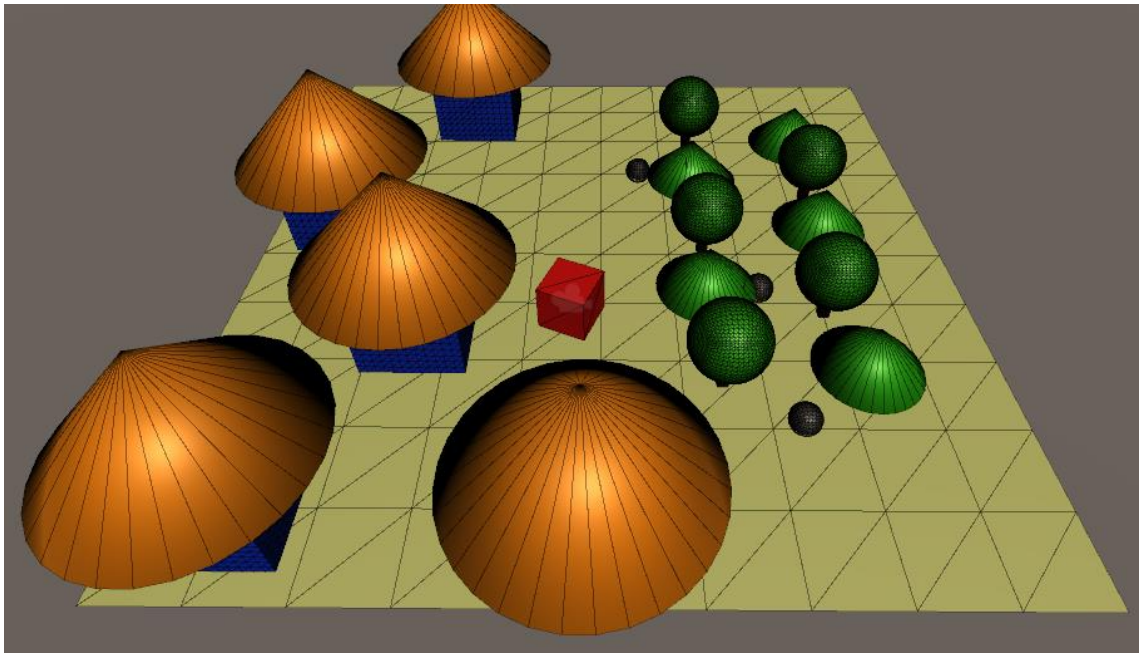
Toinen maailma oli identtinen kopio ensimmäisestä, mutta siinä CLOD ei ollut ollenkaan käytössä. Jokaisen objektin verteksien määrä oli rajoitettu aikaisemmin asetettujen minimi- ja maksimimäärien puoleenväliin.

Maailmat koostuivat taloista, kivistä ja kahdenlaisia puista, jotka oli koottu aikaisemmin tehdyistä kuutioista, palloista ja kartioista. Kuvassa 22 näkyy otos edellä mainituista objekteista.



Kuva 22. Talo, kivi ja kaksi eri puuta.

Kumpikin maailma sisälsi yhteensä kymmenen palloa, kymmenen kartiota ja viisitoista kuutiota. Kuvasta 23 nähdään, miltä maailma näytti testien aikana.



Kuva 23. Valmis maailma.

6.10 Suorituskyvyn mittaus

Suorituskykyä haluttiin mitata siksi, että voitaisiin vertailla, kuinka sulavasti maailma pyöri CLOD:n kanssa ja ilman sitä. Mittausta varten pelaajalle kirjoitettiin koodi, joka kierrätti sitä ympäri aluetta. Pelaaja kulki aina saman reitin, jotta testitulokset olisivat mahdollisimman tarkkoja.

Suorituskyvyn mittaukseen käytettiin Fraps-ohjelmaa. Se on suorituskyvyn testaukseen ja ruuduntallennukseen tehty ohjelma Windowsille. Se julkaistiin vuonna 1999, ja sen pääkehittäjä oli Beepa. [29.]

Frapsilla otettiin viiden minuutin mittaisia otoksia pelaajan liikkuessa ympäri aluetta. Mittauksissa tallennettiin kierroksen keskimääräinen, maksimi, minimi, alin 1 % ja alin 0,1 % FPS.

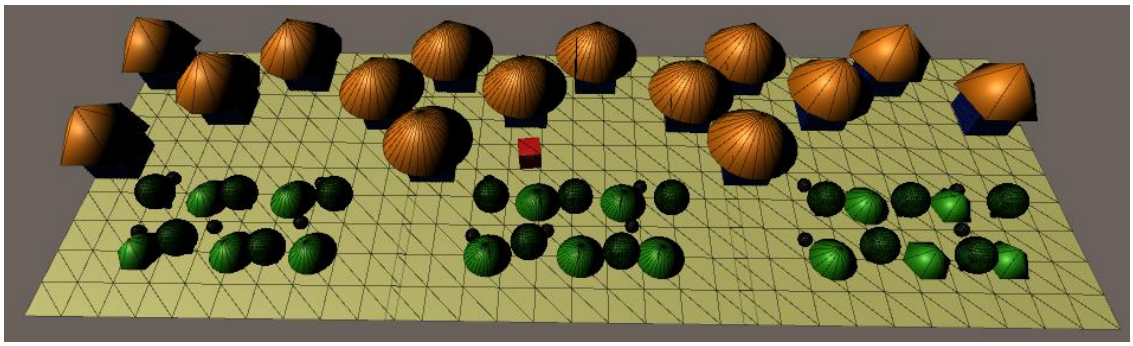
Pelin sulavuutta tarkasteltaessa keskimääräinen, maksimi tai minimi FPS eivät anna realistista tulosta. Maksimin ja minimin ongelmana on, että eri mittauskerroilla maksimin ja minimin variaatio saattaa olla todella huomattava eikä se anna tasaista tulosta. Se, että yksittäisen hetken FPS on korkea tai matala, ei välttämättä kerro pelin sulavuudesta juuri mitään.

Keskimääräinen FPS on maksimia ja minimiä parempi mittausarvo. Keskiarvo antaa paremmin suuntaa siitä, kuinka sulavasti peli yleisesti toimii. Keskiarvon ongelmana taas on, että se saattaa kaunistella ja tasoittaa isojakin FPS-notkahduksia. Näiden ongelmien takia Scott Wasson ja Steve Burke ovat tehneet tunnetuksi mitta-asteikot alin 1 % ja alin 0,1 % FPS. [30; 31.]

Frapsilla pystytään myös mittaamaan, kuinka kauan jokainen kuva pysyy näytöllä, ennen kuin seuraava kuva tulee sen tilalle. Kun peli pyörii tasaisesti 60 FPS, jokaisen kuvan ruutuaika (engl. Frametime) on noin 16,7 ms. Alimpaan 1 % ja 0,1 % sisällytetään 1 % tai 0,1 % kaikista kuvista, jotka pysyivät ruudulla pisimpään, ja nämä muutetaan takaisin FPS-lukemaksi. Jos alin 1 % tai alin 0,1 % eroavat suuresti toisistaan tai keskimääräi-

sestä FPS-luvusta eli jotkin kuvat pysyvät näytöllä huomattavasti kauemmin kuin keskimäärin, peli tuntuu tökkivältä ja käyttäjäkokemus kärsii. Siksi pelkkä keskimääräinen FPS ei riitä kertomaan pelin sulavuutta. [30; 31.]

Mittauksia tehtiin kaikkiaan kymmenellä maailmalla. Alkuperäinen alue kopioitiin kolme, viisi, yhdeksän ja kahdeksantoista kertaa, ja jokaisella tehtiin mittaus CLOD:na ja ilman sitä. Kuvassa 24 on käynnissä testi kolmella alueella. Siitä voidaan nähdä käytännössä, että mitä kauempana keskellä olevasta kuutiosta mikäkin objekti on, sitä vähemmän verteksejä kyseinen objekti sisältää.



Kuva 24. Kolme aluetta, kun CLOD oli käytössä.

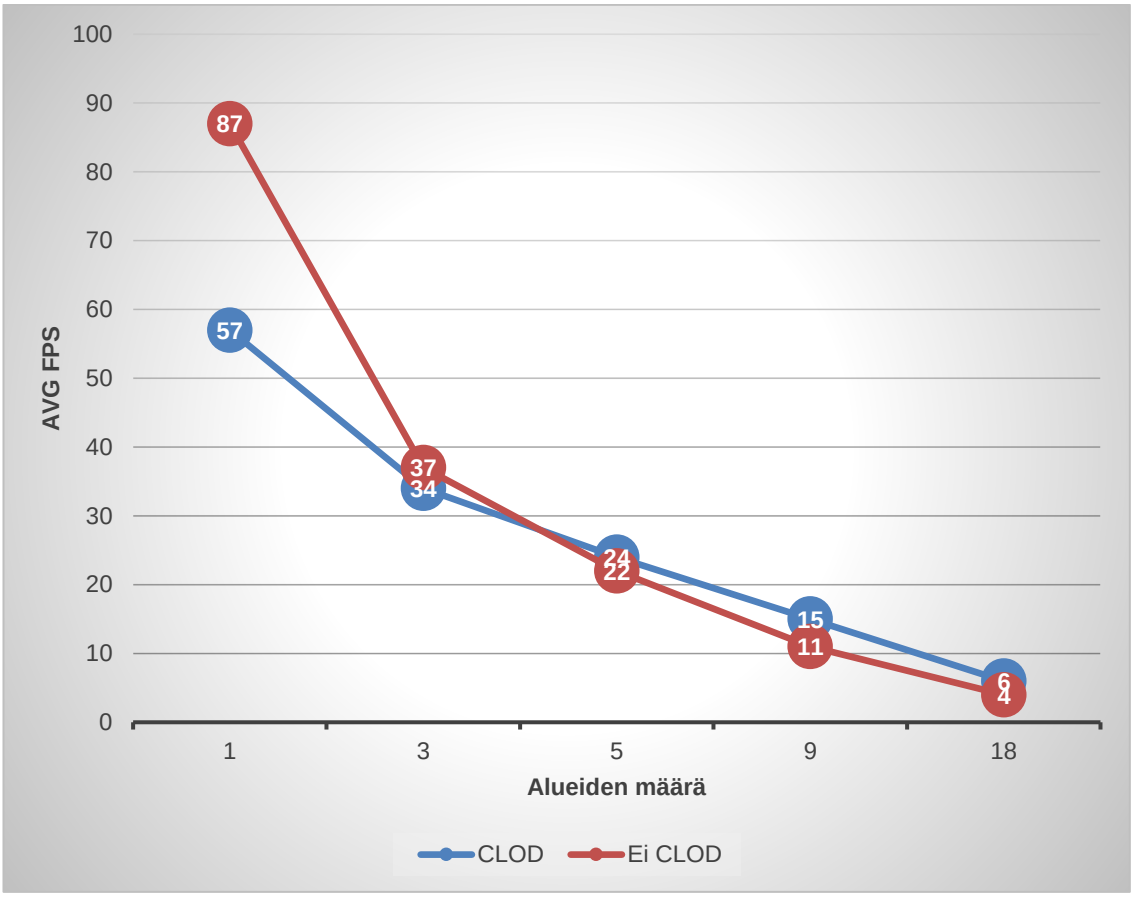
Mittauslaitteiston tekniset tiedot olivat seuraavat:

- käyttöjärjestelmä: Microsoft Windows 10 Home
- prosessori: Intel® Core™ i5-4570 CPU @ 3.2GHz, 4 Core(s)
- näytönohjain: NVIDIA GeForce GTX 970.

6.11 Tulokset

Kun suorituskyvyn mittaukset oli saatu valmiiksi, täytyi niistä tehdä jonkinlainen yhteenveto. Visualisoinnin helpottamiseksi tärkeimmistä tuloksista tehtiin kuvaajia. Ensimmäisenä tarkasteltiin keskimääräistä FPS-lukemaa.

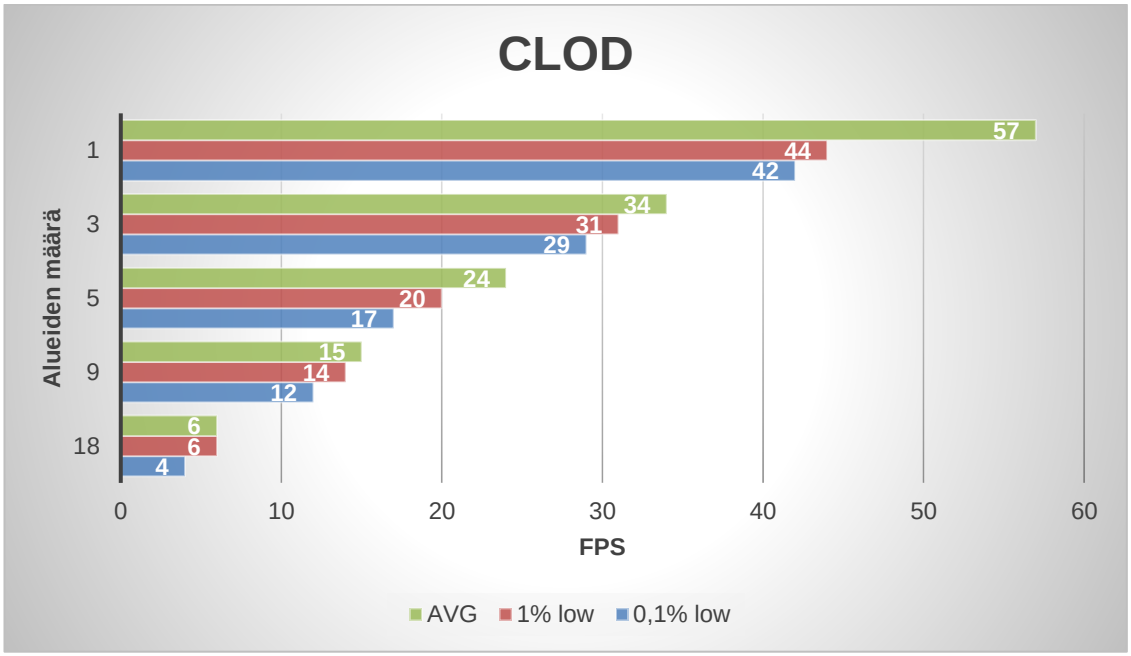
Kuten kuvasta 25 voidaan todeta, kun mittauksissa oli käytössä vain yksi alue, niin ilman CLOD:a keskimääräinen FPS oli 87, mikä on huomattavasti korkeampi kuin CLOD:n vastaava lukema, joka oli vain 57. Maailmoja lisätessä keskimääräiset FPS-lukemat kuitenkin lähestyivät toisiaan. Kun maailmoja oli mittauksessa viisi CLOD:n keskimääräinen FPS oli 24. Tämä oli jopa suurempi kuin ilman CLOD:a otettu vastaava lukema, joka oli tässä mittauksessa vain 22. Siitä eteenpäin alueita lisättäessä CLOD:n keskimääräinen FPS pysyi korkeampana kuin ilman CLOD:a.



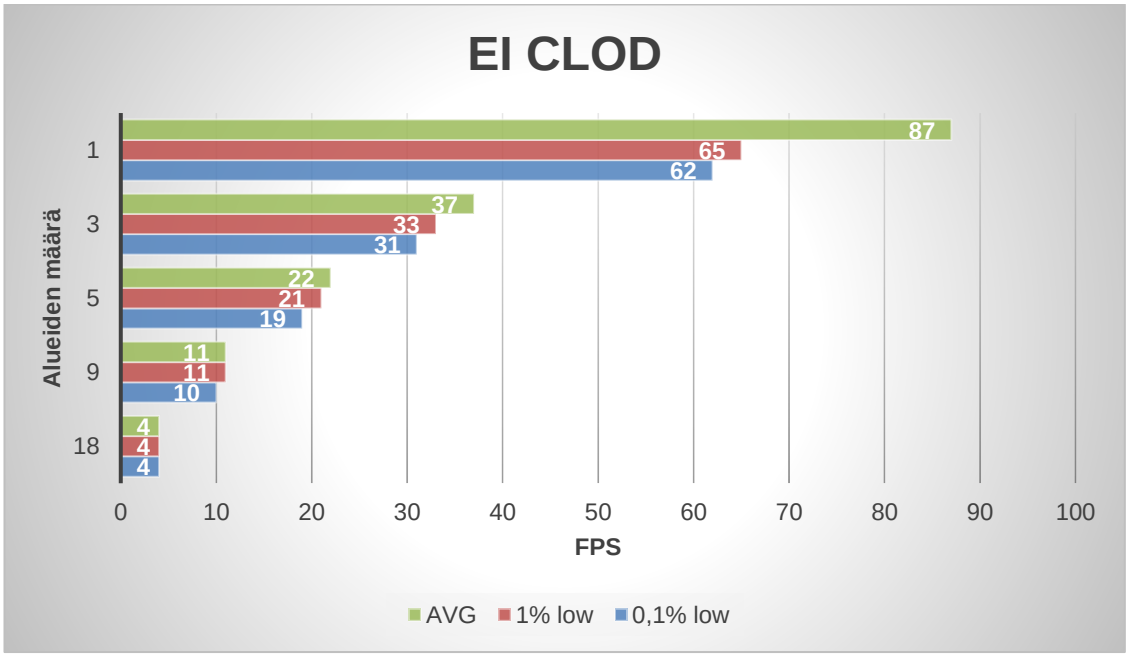
Kuva 25. Keskimääräinen FPS alueiden määrän mukaan.

Seuraavaksi tarkasteltiin keskimääräistä FPS-lukemaa verrattuna alimpaan 1 %:iin ja 0,1 %:iin. Kuvissa 26 ja 27 FPS-lukemat on esitelty yleisellä tietokoneen prosessorien ja näytönohjainten suorituskykymittausten esitystavalla. Näitä kuvaajia katsoessa erot keskimääräisen ja alimpien 1 %:n ja 0,1 %:n FPS-lukemien välillä vaikuttavat tasaisilta sekä CLOD:ssa että ilman sitä. Silmäänpistävää on kuitenkin se, että erot ovat sitä suurempia, mitä korkeampi keskimääräinen FPS oli testin aikana. Tämän perusteellä voi olla vaikea

tehdä johtopäätöstä suuntaan tai toiseen CLOD:n ja ilman sitä olevien tulosten välillä. Tästä syystä voidaan tarkempaa analyysiä varten katsoa kuvaa 28.

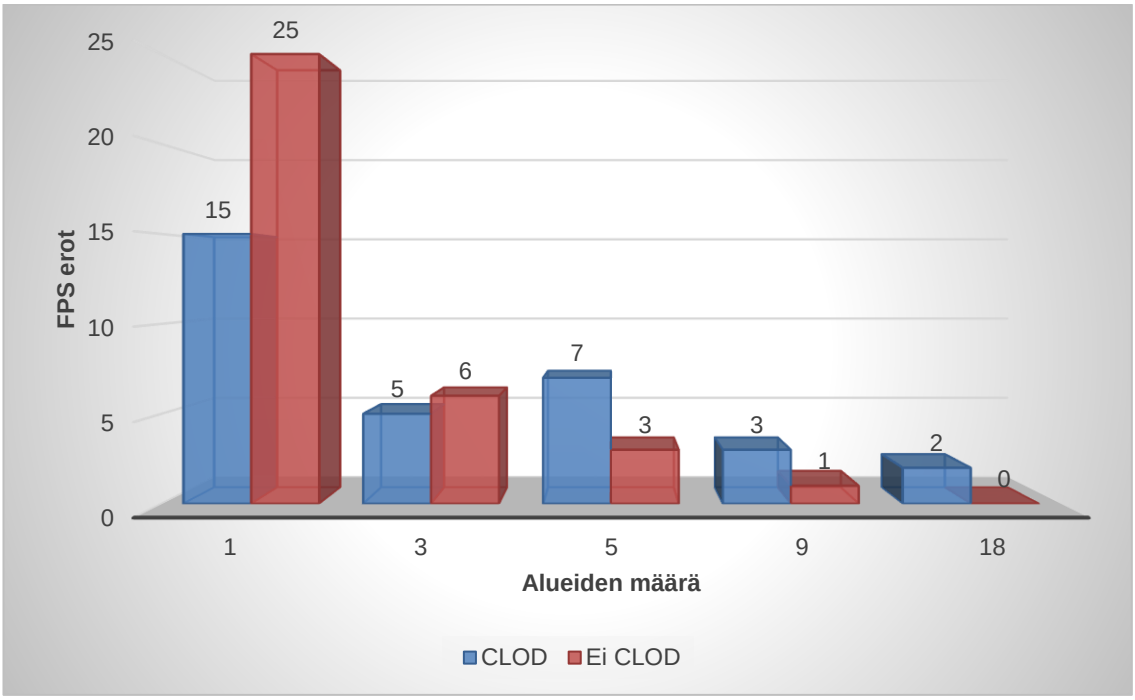


Kuva 26. Keskimääräinen, alin 1 %:n ja alin 0,1 %:n FPS-lukema alueiden määrän mukaan CLOD:ssa.



Kuva 27. Keskimääräinen, alin 1 %:n ja alin 0,1 %:n FPS-lukema alueiden määrän mukaan ilman CLOD:a.

Kuvasta 28 huomataan vielä selkeämmin se, että erot alimpien 0,1 %:n ja 1 %:n sekä keskimääräisen FPS-lukeman välillä ovat sitä suuremmat, mitä korkeampi keskimääräinen FPS oli testin aikana. Tämä näkyy suoraan myös siinä, että kun ilman CLOD:a otetun tuloksen keskimääräinen FPS menee alle CLOD:n, niin myös sen ero pienimpien 0,1 %:n, 1 %:n ja keskimääräisen FPS-lukeman välillä on pienempi kuin CLOD:ssa.



Kuva 28. Keskimääräisen ja alimman 0,1 %:n FPS-lukeman ero alueiden määrän mukaan.

Voidaanko näiden tulosten perusteella sitten tehdä johtopäätös siitä, oliko CLOD:n kanssa vai ilman sitä tehty maailma parempi näissä testeissä? Yksiselitteisesti ei voida todeta kummankaan olevan parempi vaihtoehto. Ilman CLOD:a tehdyt testit suoriutuivat paremmin maailmassa, jossa oli vain vähän renderöitäviä objekteja, kun taas CLOD suoriutui paremmin, jos objekteja oli runsaasti. Keskimääräisen, alimman 1 %:n ja alimman 0,1 %:n FPS-lukujen eroista ei voida tehdä päätelmiä CLOD:n ja ilman CLOD:a toteutettujen testien välillä. FPS-lukujen ero laski samassa suhteessa sekä CLOD:ssa, että ilman sitä, kun keskimääräinen FPS laski.

7 Yhteenveto

Insinöörityössä perehdyttiin käsitteeseen CLOD: mitä kaikkea se sisältää, mitä sen ympäriltä löytyy ja mihin sitä voidaan hyödyntää. CLOD tarkoittaa jatkuvaa yksityiskohtien tasoa, eli objektin yksityiskohtaisuus muuttuu jatkuvasti, riippuen sen etäisyydestä katsojaan. Työssä tutustuttiin myös polygonien generointiin ja niiden yksinkertaistamiseen. Näitä tietoja apuna käyttäen rakennettiin Unity-pelimoottorilla testisovellus, jossa vertailtiin suorituskykyä CLOD:lla ja ilman sitä toteutettujen maailmojen välillä.

Vaikka osaan aiheista ehdittiin tutustua vain pintapuolin, insinöörityön tavoitteet täyttyivät hyvin. Testisovelluksesta saadun datan perusteella voitiin vertailla CLOD:lla ja ilman sitä toteutettuja maailmoja. Vertailusta selvisi, että mitä enemmän objekteja testimaailmassa oli, sitä paremmin CLOD:lla toteutettu maailma suoriutui verrattuna ilman sitä toteutettuun maailmaan. Kun objekteja oli maailmassa vain vähän, ilman CLOD:a toteutettu maailma suoriutui testeistä paremmin.

Valmiissa testisovelluksessa käytettiin vain yksinkertaisia muotoja, kuten pallo ja kuutio. Jos työn kehittämistä jatkettaisiin tästä eteenpäin, seuraava vaihe olisi toteuttaa CLOD monimutkaisemmilla objekteilla. Parhaassa tapauksessa Unity-projektiin pystyttäisiin lisäämään mikä tahansa polygonimalli ja CLOD toimisi siinä suoraan yleisellä algoritmilla.

Lähteet

1 Denham, Thomas. What is LOD (Level of Detail) in 3D Modeling? Verkkoaineisto. Concept Art Empire. <http://conceptartempire.com/3d-lod-level-of-detail/>. Luettu 4.9.2020.

2 Level of Detail. Verkkoaineisto. Glasnost. <http://glasnost.it-carlow.ie/~powerk/3DGraphics2/Theory/Levelofdetail.htm>. Luettu 10.9.2020.

3 Huebner, Robert; Reddy, Martin; Watson, Benjamin A.; Varshney, Amitabh; Luebke, David & Cohen, Jonathan D. 2002. Level of Detail for 3D Graphics. E-kirja. Morgan Kaufmann Publishers.

4 Zamri, Muhamad Najib Bin & Sunar, Mohd Shahrizal. 2008. Advances in Computer Graphics and Visual Environment Vol. 2. E-kirja. UTM Press.

5 Ramos, Francisco; Chover, Miguel; Ripolles, Oscar & Granell, Carlos. 2006. Continuous Level of Detail on Graphics Hardware. Teoksessa Palágyi, Kálmán; Nyúl, László & Kuba, Attila. Discrete Geometry for Computer Imagery. Szeged: Springer.

6 The NEXT Generation lexicon: gaming terminology from A to Z. 1996. NEXT Generation 1.3.1996, s. 32.

7 How Did They Do That – Spyro's Detailed Draw Distance. 2015. Verkkoaineisto. Dinosaur Bytes. Youtube. <https://www.youtube.com/watch?v=G5tBQfi-BASM&t>. Luettu 2.9.2020.

8 Gröger, Gerhard; Kolbe, Thomas H.; Nagel, Claus & Häfele, Karl-Heinz. 2012. OCG City Geography Markup Language (CityGML) Encoding Standard. Open Geospatial Consortium, s. 11-12.

9 Bilkecki, Filip; Ledoux, Hugo & Stoter, Jantier. 2016. An Improved LOD specification for 3D building models. Delft University of Technology. Pure.tudelft.

10 Brodkin, Jon. 2013. How Unity3D Became a Game Development Beast. Verkkoaineisto. Dice. <https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/>. Luettu 2.10.2020

11 Unity. 2020. Verkkoaineisto. Unity Technologies. <https://unity.com/>. Luettu 1.9.2020.

12 Buckley, Ian. 2019. 7 Unity Game Development Languages to Learn: Which Is Best? Verkkoaineisto. Makeuseof. <https://www.makeuseof.com/tag/unity-game-development-languages/>. Luettu 10.9.2020.

13 Level of Detail (LOD) for meshes. 2020. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/LevelOfDetail.html>. Luettu 15.9.2020.

14 LOD Group. 2018. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-LODGroup.html>. Luettu 15.9.2020.

15 Arlebrink, Ludvig & Linde, Fredrik. 2018. A Study on Discrete Level of Detail in the Unity Game Engine. Blekinge Institute of Technology.

16 Petty, Josh. What is a Polygon Mesh? Verkkoaineisto. Concept Art Empire. <https://conceptartempire.com/polygon-mesh/>. Luettu 10.9.2020.

17 Introduction to Polygon Meshes. Verkkoaineisto. Scratchapixel. <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh>. Luettu 14.9.2020.

18 Owen, Steven J. 2016. An Introduction to Automatic Mesh Generation Algorithms – Part 1. Sandia National Lab. OSTI.gov.

19 Jayanti, Sreenivas. Mod-07 Lec-46 Delaunay triangulation method for unstructured grid generation. Verkkoaineisto. Youtube. <https://www.youtube.com/watch?v=tWf1z9i-ORg>. Luettu 20.9.2020.

20 Shewchuk, Jonathan. 1999. Lecture Notes on Delaunay Mesh Generation. University of California. CiteSeetX.

21 Shewchuk, Jonathan. 1996. Ruppert's Delaunay Refinement Algorithm. Verkkoaineisto. Carnegie Mellon School of Computer Science. <https://www.cs.cmu.edu/~quake/tripaper/triangle3.html>. Luettu 24.9.2020.

22 Cacciola, Fernando; Rouxel-Labbé, Mael & Şenbaşlar, Baskin. Triangulated Surface Mesh Simplification. Verkkoaineisto. CGAL. <https://doc.cgal.org/latest/Surface_mesh_simplification/index.html>. Luettu 22.9.2020.

23 Mesh Simplification. Verkkoaineisto. Computer Graphics at Stanford University. <http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf>. Luettu 29.9.2020.

24 Shene, Ching-Kuang. 2010. Mesh Simplification. Michigan Technological University.

25 Van Kreveld, Mark. Triangle mesh processing. Verkkoaineisto. Slideserve. <https://www.slideserve.com/elroy/triangle-mesh-processing>. Luettu 1.10.2020.

26 InfoQ eMag: A Preview of C# 7. 2016. Verkkoaineisto. InfoQ. <https://www.infoq.com/minibooks/emag-c-sharp-preview/>. Luettu 3.10.2020.

27 TIOBE Index for October 2020. 2020. Verkkoaineisto. TIOBE. <https://www.tiobe.com/tiobe-index/>. Luettu 20.9.2020.

28 Cube Chopper. Verkkoaineisto. CGLearn. <https://cglearn.eu/pub/computer-graphics/task/cube-chopper-1>. Luettu 10.9.2020.

29 Fraps News. 2013. Verkkoaineisto. Fraps. <https://fraps.com/news.php>. Luettu 5.9.2020.

30 Wasson, Scott. 2011. Inside the second: A new look at game benchmarking. Verkkoaineisto. The Tech Report. <https://techreport.com/review/21516/inside-the-second-a-new-look-at-game-benchmarking/8/>. Luettu 5.10.2020.

31 Burke, Steve. 2016. Testing Methodology Explanations: 1% & 0.1% Lows, Delta T over Ambient. Verkkoaineisto. Gamers Nexus. <https://www.gamers-nexus.net/site-news/2513-testing-methodology-explained-1percent-lows-and-delta-t>. Luettu 5.10.2020.