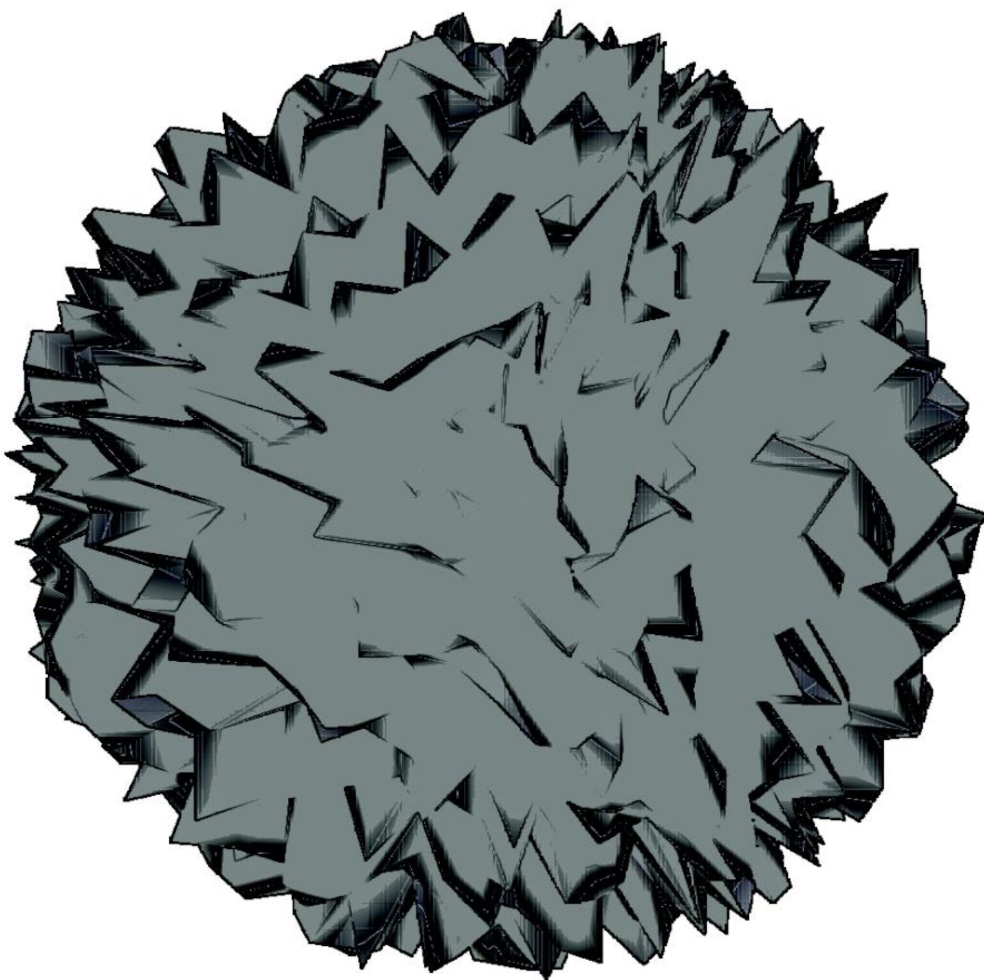
Matti Kemppainen

Laskentavarjostimien käyttö pelikehityksessä



Tradenomi

Tietojenkäsittely

Kevät 2025

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Kemppainen Matti

Työn nimi: Laskentavarjostimien käyttö pelikehityksessä

Tutkintonimike: Tradenomi (amk), tietojenkäsittely

Asiasanat: Laskentavarjostin, ohjelmointi

Laskentavarjostimet ovat olennainen osa modernia pelikehitystä, sillä ne mahdollistavat raskaan laskennan suorittamisen tehokkaasti näytönohjaimen avulla. Opinnäytetyön tavoitteena oli tutkia laskentavarjostimien toimintaa ja niiden hyödyntämistä peleissä erityisesti suorituskyvyn optimoinnin näkökulmasta. Laskentavarjostimet suoritetaan GPU:lla, kuten muutkin varjostintyypit, mutta niiden keskeinen ero on, että niitä ei käytetä suoraan renderöintiin, vaan niiden pääasiallinen tarkoitus on laskennan suorittaminen ja datan käsittely. Niiden käyttö on usein tehokkaampaa kuin vastaavien laskutoimitusten suorittaminen CPU:lla, erityisesti monimutkaisissa ja rinnastettavissa laskentatehtävissä.

Työssä toteutettiin käytännön kokeiluja Unity-pelimoottorissa, jossa tutkittiin laskentavarjostimien ohjelmointia. Erityisesti perehdyttiin Unityn tarjoamiin rajapintoihin, puskureihin ja renderöintitekstuureihin, joiden avulla C#-skripti ja laskentavarjostin voivat kommunikoida keskenään. Lisäksi käytiin läpi, kuinka laskentavarjostin käynnistetään ja miten sille välitetään tarvittavat tiedot. Tutkimuksen keskiössä oli myös se, kuinka laskentavarjostimet voivat parantaa pelin suorituskykyä siirtämällä raskaita laskentaprosesseja pois CPU:lta GPU:lle.

Keskeisimpinä tuloksina havaittiin, että laskentavarjostimet soveltuvat erinomaisesti rinnakkaislaskentaa vaativiin tehtäviin, kuten maaston luomiseen, reitinhakualgoritmeihin ja fysiikkasimulaatioihin. Opinnäytetyön toiminnallisessa osuudessa planeetan luomislogiikka siirrettiin C#-skriptistä laskentavarjostimelle, mikä mahdollisti suuren määrän verteksien käsittelyn rinnakkain. Lisäksi toteutettiin jälkikäsittelyefekti, joka hyödynsi laskentavarjostimia ja renderöintitekstuureja. Laskentavarjostinten käyttö osoittautui tehokkaaksi erityisesti silloin, kun käsiteltävä tietomäärä oli suuri ja rinnakkaislaskentaa voitiin hyödyntää kunnolla.

Johtopäätöksenä voidaan todeta, että laskentavarjostimet tarjoavat merkittäviä etuja pelikehityksessä, erityisesti suorituskyvyn optimoinnin kannalta. Ne mahdollistavat monimutkaisten laskentatehtävien suorittamisen tehokkaasti ilman, että CPU kuormittuu liikaa. Työssä toteutetut kokeilut vahvistavat, että laskentavarjostimien hyödyntäminen voi parantaa pelien suorituskykyä merkittävästi, kunhan niiden käyttö suunnitellaan huolellisesti. Lisäksi havaittiin, että pienemmissä laskentatehtävissä CPU voi joissain tapauksissa olla tehokkaampi vaihtoehto tiedonsiirrosta aiheutuvan viiveen takia. Näin ollen laskentavarjostimen käyttöä harkittaessa on tärkeää arvioida tapauskohtaisesti, missä tilanteissa sen käyttö on tarkoituksenmukaista.

KAMK • University of Applied Sciences

Abstract

Author(s): Kemppainen Matti

Title of the Publication: The Use of Compute Shaders in Game Development

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: Compute shader, programming

Compute shaders are an essential part of modern game development, enabling heavy computations to be performed efficiently using the graphics processing unit (GPU). The objective of this thesis was to examine the functionality of compute shaders and their application in games, particularly from a performance optimization perspective. Like other shader types, compute shaders are executed on the GPU, but their primary distinction is that they are not directly used for rendering. Instead, their main purpose is to perform computations and process data. Their use is often more efficient than executing equivalent calculations on the central processing unit (CPU), especially for complex and parallelizable computing tasks.

The study involved practical experiments in the Unity game engine, where the programming of compute shaders and their integration into game development workflows were explored. Special attention was given to Unity's provided interfaces, including buffers and render textures, which enable the communication between C# scripts and compute shaders. Additionally, the study covered how compute shaders are initialized and how necessary data is passed to them. A key focus of the research was understanding how compute shaders can improve game performance by offloading heavy computational processes from the CPU to the GPU.

The main findings indicate that compute shaders are highly suitable for tasks that require parallel computing, such as terrain generation, pathfinding algorithms, and physics simulations. In the functional part of the thesis, the logic for generating a planet was transferred from a C# script to a compute shader, allowing a large number of vertices to be processed in parallel. Additionally, a post-processing effect utilizing compute shaders and render textures was implemented. The use of compute shaders proved to be particularly effective when handling large amounts of data and when parallelism could be fully leveraged.

In conclusion, compute shaders offer significant advantages in game development, particularly in terms of performance optimization. They enable complex computational tasks to be executed efficiently without overloading the CPU. The experiments conducted in this study confirm that utilizing compute shaders can significantly improve the performance in games, provided their use is carefully planned. Furthermore, it was observed that for smaller computational tasks, the CPU can sometimes be a more efficient option due to data transfer latency. Therefore, when considering the use of compute shaders, it is crucial to assess on a case-by-case basis whether their implementation is appropriate for a given scenario.

Sisällys

1 Johdanto 1

2 Laskentavarjostimen toiminta 2

3 Käyttötarkoitukset peleissä 3

4 Laskentavarjostimen kirjoittaminen 4

4.1 Puskurit 4

4.2 Renderöintitekstuuri 4

4.3 Laskentavarjostimet Unity-pelimoottorissa 5

4.3.1 ComputeShader-luokka 5

4.3.2 ComputeBuffer-luokka 6

4.3.3 Laskentavarjostimen ja c#-skriptin kommunikointi 8

5 Käytännön toteutus Unity-pelimoottorissa 9

5.1 Planeetan luominen laskentavarjostimella 9

5.2 Jälkikäsittelyefekti käyttäen laskentavarjostinta 14

6 Päätäntö 19

Lähteet 21

Liitteet

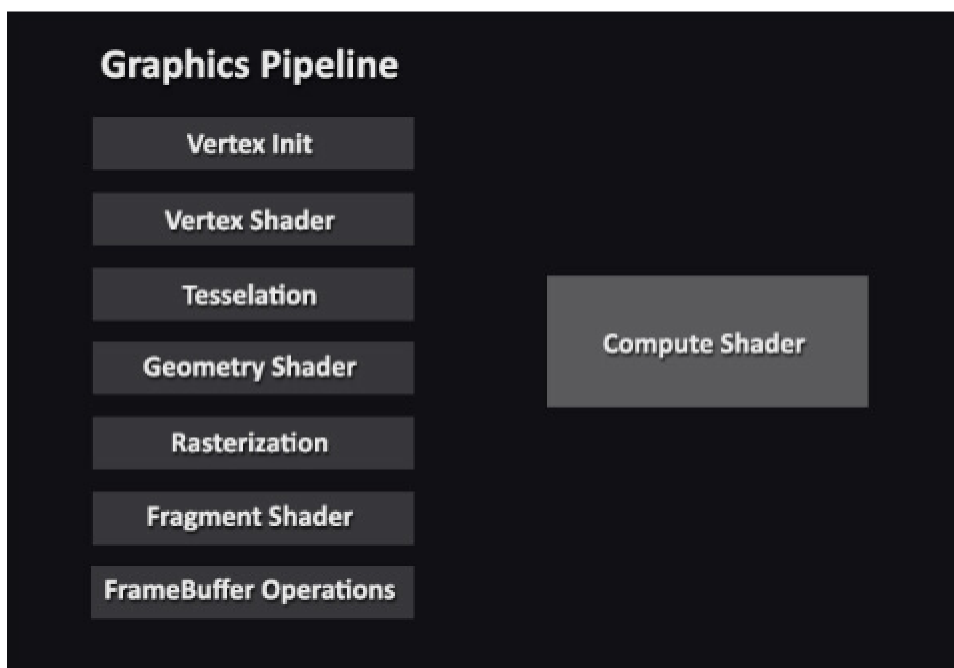
Symboliluettelo

Assetti	Resurssi tai ominaisuus pelimoottorissa. Pelin osa.
Editori	Tietokoneohjelma tai työkalu, jolla voidaan tehdä ja muokata esimerkiksi pelejä.
CPU	Central Processing Unit. Tietokoneen suoritin. Se suorittaa ohjelmien palvelimelle antamat käskyt.
GPU	Graphics Processing Unit. Näytönohjain eli tietokoneen komponentti, jonka tehtävänä on piirtää graafiset elementit näytölle.
HLSL	High Level Shader Language. Microsoftin kehittämä shading-kieli.
Kerneli	Käyttöjärjestelmän ydin. Tässä tapauksessa se tarkoittaa laskentavarjostimen pääfunktiota.
Säie	Näytönohjaimen pieniä yksikköjä, joilla laskentaa suoritetaan joko rinnakkain, tai sarjassa.
URP	Universal Render Pipeline. Unity-pelimoottorin yleinen renderöintiputki.
Varjostin	Perinteisesti grafiikkasuorittimelle suunnattu ohjelma, joka suorittaa näkymän renderöintiin liittyviä pikseli- tai verteksikohtaisia erikoistehosteita jollain varjostimeen määritetyllä algoritmilla omalla renderöintiputkellaan. Engl. shader.

1 Johdanto

Laskentavarjostimet ovat ohjelmia, jotka suoritetaan näytönohjaimen (GPU) puolella hyödyntäen sen suurta laskentatehoa. Toisin kuin tavanomainen koodi, joka ajetaan suorittimella (CPU), laskentavarjostimet käyttävät GPU:n rinnakkaisia laskentaytimiä, mikä tekee niistä tehokkaita erityisesti raskaissa laskentatehtävissä. [1.]

Ensisijainen käyttötarkoitus laskentavarjostimille on erilaisten laskentatehtävien suorittaminen. Laskentavarjostinta ei yleisesti käytetä renderöintiin, eikä kolmiomallien tai pikselien piirtämiseen, toisin kuin muita varjostintyyppejä, eli verteksi-, fragment-, ja geometriavarjostimia käytetään. Tämän vuoksi laskentavarjostin ei ole osana renderöintiputkea, kuten kuvasta 1 näkyy, eikä sillä myöskään ole vastaavanlaisia sisääntuloarvoja kuin muilla varjostintyypeillä. [2.]



Kuva 1. Renderöintiputken eri vaiheet ja kuinka laskentavarjostin on sen ulkopuolella [3].

Siirtämällä raskasta tietojenkäsittelyä suorittimelta näytönohjaimelle laskentavarjostimet mahdollistavat tehokkaamman ohjelman suorituskyvyn. Suuren ydinmäärän ansiosta GPU kykenee rinnakkaislaskentaan, minkä vuoksi laskentavarjostimet ovat hyödyllisiä suurten laskutoimitusten suorittamisessa.

2 Laskentavarjostimen toiminta

Laskentavarjostin voi suorittaa rinnakkaislaskentaa hyödyntäen GPU:n säikeitä [1]. Säikeiden määrä liikkuu tuhansissa. CPU:lla on paljon vähemmän resursseja vastaavanlaiseen laskentaan. Tämän takia laskentavarjostimia voidaan käyttää esimerkiksi optimoimaan videopelien suorituskykyä. Jos videopelissä on esimerkiksi todella raskas maaston luonti, on se todennäköisesti parempi suorittaa laskentavarjostimella.

Sen sijaan, että ohjelma suorittaisi laskentaa sarjassa eli yksi laskenta kerrallaan, laskentavarjostin suorittaa laskentaa rinnakkain. Tämä tarkoittaa sitä, että laskentavarjostin suorittaa laskutoimitukset samanaikaisesti, mikä on tehokkaampaa ajallisesti ja resurssillisesti. Laskentavarjostimelle lähetetään suuri määrä laskettavaa tietoa, jonka laskentavarjostin käsittelee tiettyjen ohjeiden mukaan. Ohjeet laskentaan on määrätty itse laskentavarjostimella.

Kaikkea laskentaa ei kuitenkaan kannata automaattisesti suorittaa laskentavarjostimella. GPU on tehokkaampi, jos siitä otetaan kunnolla tehoja irti. Myös viive saattaa olla ongelma. Tiedon siirtäminen vie aikaa ja se voi ruuhkauttaa ohjelman suorittamista. Tätä voidaan ehkäistä suunnittelemalla laskentavarjostin käyttämään juuri se määrä tietoa kuin laskenta vaatii, mutta se ei silti täysin ratkaise ongelmaa. [4.] Kannattaa siis miettiä tapauskohtaisesti, onko kyseinen tehtävä laskentavarjostimen arvoinen vai voisiko sen suorittaa ilman.

3 Käyttötarkoitukset peleissä

Videopeleissä monimutkaiset laskennat, kuten esimerkiksi fysiikkasimulaatiot, AI-reitinhakualgoritmit, partikkelisimulaatiot tai proseduraalisen maaston luonti on hyvä suorittaa laskentavarjostimen avulla. Kun laskentatehtävät suoritetaan GPU:n puolella, jää CPU:lle resursseja itse pelin logiikan suorittamiseen. Laskentavarjostin siis mahdollistaa sen, että peli pyörii paremmin.

Säteenseuranta (engl. ray tracing) on tekniikka, jolla simuloidaan realistista valaistusta ja sen käyttäytymistä. Laskenta ottaa huomioon jokaisen valonsäteen kulkeman reitin ja heijastukset eri pinnoilta, minkä tuloksena saadaan aikaan realistinen valaistus ja heijastumiset. [5.]

Instruments of Destruction [6] on videopeli, jossa on käytetty runsaasti laskentavarjostimia fysiikoiden ja muiden tehtävien laskentaan. Pelissä tarkoituksena on tuhota ympäristöä erilaisilla mekaanisilla laitteilla, kuten kuvassa 2 näkyy. Sortuvista rakennuksista kertyy valtava määrä murskaa ja pieniä partikkeleita, joita simuloidaan yksitellen. Jokaiselle partikkelille lasketaan myös törmäykset, mikä saa simulaation näyttämään todella vaikuttavalta. Tämä peli tuskin olisi mahdollista toteuttaa ilman laskentavarjostimia.

Kuva 2. Instruments of Destruction -pelin ympäristön tuhoutuminen [6].

4 Laskentavarjostimen kirjoittaminen

Laskentavarjostin ei osaa toimia itsekseen, vaan se tarvitsee käskyjä pääskriptin puolelta eli CPU:n puolelta. Lisäksi laskentaan tarvitaan tietoa, joka myös tulee pääskriptistä. Seuraavissa kappaleissa käyn läpi, missä muodoissa tietoa voidaan siirtää ja miten laskentavarjostimia käytetään Unity-pelimoottorissa.

4.1 Puskurit

Puskureita (engl. buffer) käytetään muissakin tehtävissä kuin vain laskentavarjostimien kanssa. Puskuri sisältää tietoa, jota kuljetetaan kahden osapuolen välillä. Yleensä nämä kaksi osapuolta toimivat eri nopeuksilla, ja juuri tämän takia puskuria käytetään, jotta tiedon siirtyminen ei ruuhkautuisi [7]. Puskuriin tallennetaan tietoa pääskriptin puolella, josta se sitten lähetetään laskentavarjostimelle. Laskentavarjostin käsittelee puskurissa olevaa tietoa ja lähettää lopuksi käsitellyn tiedon takaisin pääskriptiin.

Kun puhutaan laskentavarjostimista, yksi puskuri sisältää vain yhdentyyppisen tiedon joukon eli Arrayn. Tieto voi olla myös tietueena, jolloin saadaan kuljetettua paljon enemmän tietoa kerralla. Yhteen tietueeseen voidaan tallentaa monentyyppistä tietoa, ja tietueesta voidaan tehdä Array. Tällöin riittää, että laskentavarjostimelle lähetetään vain yksi puskuri usean sijaan. [1.]

4.2 Renderöintitekstuuri

Renderöintitekstuurit ovat nimensä mukaan tekstuureja, joihin voidaan renderöidä kuvaa. Yleinen käyttötarkoitus, johon renderöintitekstuuria käytetään, on laittaa se kameran kohdetekstuuriksi, mikä tarkoittaa sitä, että kameran kuva renderöidään näytön sijasta tekstuurille. Renderöintitekstuuria voi myös käyttää materiaalissa, jonka voi esimerkiksi kiinnittää objektin pintaan pelimaailmassa. [8.]

Renderöintitekstuuria voidaan käyttää laskentavarjostimen kanssa. Se toimii periaatteessa samalla tavalla kuin puskurit, mutta tieto on tekstuurin muodossa. Joitain esimerkkejä, joissa renderöintitekstuuria voitaisi käyttää ovat proseduraalinen tekstuurin luonti maailman luomista varten tai kuvan jälkikäsittelyefektit (engl. Post-processing effects). [8.]

4.3 Laskentavarjostimet Unity-pelimoottorissa

Unity tarjoaa hyvät dokumentaatiot laskentavarjostimien käyttöön. Dokumentaatiosivuilla kerrotaan yleisesti laskentavarjostimien tekniikasta ja toiminnasta. Sivuilta löytyy myös suoraa ohjeistusta, kuinka laskentavarjostimen kanssa pääsee alkuun. [9.]

Unityssä kaikki varjostimet kirjoitetaan HLSL-ohjelmointikielellä. Dokumentaatiossa sanotaan, että jokaisella laskentavarjostinassetilla tulee olla vähintään yksi kerneli eli pääfunktio. Kerneli määritellään varjostinkoodissa #pragma-direktiivillä:

```
#pragma kernel KernelName
```

Laskentavarjostimet suoritetaan näytönohjaimen säikeillä. Säikeet voidaan määrittää koodissa numthreads-attribuutilla ennen pääfunktiota. Säikeiden joukko voi olla, yksi-, kaksi-, tai kolmiulotteinen, ja ulottuvuuksien mitat määritetään sulkeissa. Mitä suurempi säikeiden joukko, sitä enemmän laskentatehoa tehtävälle varataan. [9.] Kirjoitusmuoto on seuraavanlainen:

```
[numthreads(1, 1, 1)]
```

4.3.1 ComputeShader-luokka

Unityssä on valmis luokka nimeltään ComputeShader, joka tarjoaa rajapinnan c#:n ja laskentavarjostimien välille. Sen avulla voidaan ladata laskentavarjostimia, määrittää niille syötteitä sekä ajaa niitä GPU:lla. Tätä luokkaa siis käytetään c#-skriptin puolella.

Keskeisiä funktioita, joita ComputeShader-luokasta löytyy:

1. FindKernel

   Tämän avulla voidaan hakea tietyn kernelin indeksi. Kernelin nimi on määrätty laskentavarjostimessa #pragma kernel -direktiivillä. [10.]

   public int **FindKernel**(string **name**);

2. SetBuffer

   Laskentavarjostin tarvitsee usein puskurin tiedon lukemiseen ja kirjoittamiseen, ja tällä funktiolla voidaan yhdistää puskuri johonkin tiettyyn kerneliin. Funktion parametreinä ovat kyseisen kernelin indeksi, puskurin nimi sekä itse käsiteltävä puskuri. [10.]

   public void **SetBuffer**(int **kernelIndex**, string **name**, ComputeBuffer **buffer**);

3. SetTexture

   Käytetään, jos tietoa siirretään tekstuurin muodossa. Toimii samalla tavalla kuin SetBuffer-funktio, mutta puskurin sijasta käytetään tekstuuria. [10.]

   public void **SetTexture**(int **kernelIndex**, string **name**, Texture **texture**);

4. Dispatch

   Tärkein funktio, sillä se suorittaa laskentavarjostimen. Funktiossa määrätään, minkä indeksin kerneli suoritetaan ja kuinka suuri säikeiden määrä on kussakin ulottuvuudessa. [10.]

   public void **Dispatch**(int **kernerIndex**, int **threadsX**, int **threadsY**, int **threadsZ**);

4.3.2 ComputeBuffer-luokka

ComputeBuffer-luokkaa käytetään yleisimmin laskentavarjostimien kanssa. Tätä luokkaa käytetään puskurina, johon voidaan tallentaa c#-koodilla tietoa, jota laskentavarjostin sitten käyttää laskennassa. Tämäkin luokka on c#-skriptin puolella, kuten ComputeShader-luokka.

ComputeBuffer-luokan keskeiset funktiot ja ominaisuudet:

1. ComputeBuffer-olio

   ComputeBuffer-luokan konstruktorissa luodaan uusi olio, jolle määrätään puskurissa olevien elementtien määrä sekä niiden koko. Koon on oltava neljällä kerrollinen ja maksimissaan 2048. [11].

   public **ComputeBuffer**(int **count**, int **stride**);

2. SetData

   Asettaa laskentavarjostimessa olevaan tietueeseen tietoa puskurilla. Puskurin tiedot lähetetään Array-muodossa. [12.]

   public void **SetData**(Array **data**);

   Jos halutaan asettaa vain osa Arrayn tiedosta, käytetään managedBufferStartIndex-, computeBufferStartIndex- ja count-parametrejä [12].

   public void **SetData**(Array **data**, int **managedBufferStartIndex**, int **computeBufferStartIndex**, int **count**);

3. GetData

   Tiedon hakeminen takaisin laskentavarjostimelta. Tieto tulee takaisin Array-muodossa. [12].

   public void **GetData**(Array **data**);

4. Release

   Kun puskuria ei enää tarvita, on tärkeää kutsua Release-funktiota. Tämä vapauttaa puskurin käytöstä ja GPU-muistista. [12.]

   public void **Release**();

4.3.3 Laskentavarjostimen ja c#-skriptin kommunikointi

Aiemmin mainittuja Unityn tarjoamia luokkia käytetään c#-skriptin ja varjostinskriptin väliseen kommunikointiin. Yleensä laskentavarjostimessa alustetaan muuttujat, joita laskennassa käytetään, ja c#-skriptillä muutetaan näiden muuttujien arvoja. Laskentavarjostimessa voidaan esimerkiksi alustaa float-tyypin muuttuja. Tämän arvoa voidaan muuttaa c#-skriptistä käsin käyttämällä ComputeShader-luokan tarjoamaa SetFloat-funktiota. [13.]

Mikäli tietoa on enemmän, voidaan käyttää puskuria. Laskentavarjostimessa alustetaan tietue, joka sisältää joitain muuttujia. Lisäksi alustetaan puskuri käyttämällä tietuetta, joka juuri alustettiin. Ohjelma 1. on esimerkki siitä, miten tietue ja puskuri alustetaan. [13.]

```
struct Cube
{
    float position;
    float4 color;
};

RWStructuredBuffer<Cube> cubes;
```

Ohjelma 1. Tietueen määrittäminen ja puskurin alustaminen laskentavarjostinskriptissä.

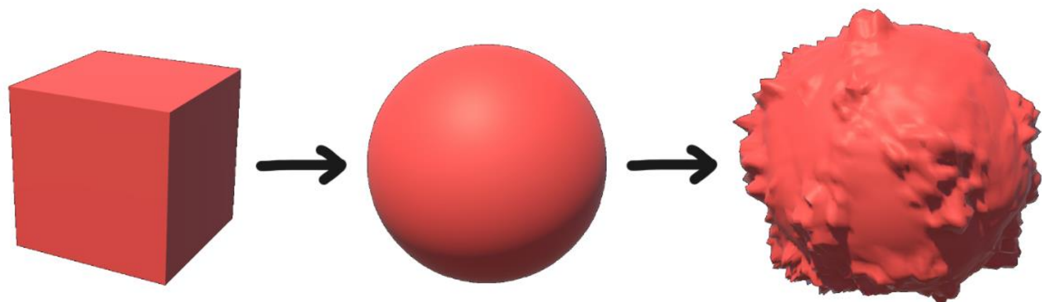
C#-skriptissä tehdään vastaavanlainen tietue, jolla on saman tyypin muuttujat. Tästä tietueesta tehdään Array, johon syötetään tietoa. Lopulta tämä Array voidaan lähettää laskentavarjostimen puolelle käyttämällä ComputeBuffer-luokan SetData-funktiota. [13.]

Kun laskentaan tarvittavat tiedot on saatu siirrettyä laskentavarjostimelle, kutsutaan Dispatch-funktiota, joka aloittaa laskennan suorittamisen. Laskettu tieto saadaan takaisin c#-skriptiin käyttämällä ComputeBuffer-luokan GetData-funktiota. Kun laskenta on suoritettu eikä puskuria enää tarvita, on tärkeää muistaa kutsua Release-funktiota. [13.]

5 Käytännön toteutus Unity-pelimoottorissa

Käytännön toteutus aloitetaan seuraamalla Sebastian Laguen tekemää opasta proseduraalisen planeetan luomisesta [14]. Oppaassa ei käytetä laskentavarjostimia, vaan ohjelma suoritetaan kokonaan c#-skriptin puolella eli CPU:ta käyttäen. Kun valmiina on tarpeeksi hyvä pohja planeetan luomiselle, on seuraavana tehtävänä siirtää planeetan luomislogiikka laskentavarjostimelle. Kun planeetta saadaan luotua laskentavarjostimella, voidaan verrata suorituskykyä laskentavarjostimen ja CPU:n välillä.

Kuvassa 3 on selvennys planeetan luomisprosessista. Ensin luodaan tavallinen kuutio, jolla on kuusi sivua, joista jokaisella on joku resoluutio (esimerkiksi 128x128). Jokaisella sivulla on silloin resoluution verran verteksejä. Seuraavaksi kuution jokaisen verteksin sijainti normalisoidaan keskikohdasta katsoen ja kerrotaan säteen mitalla, jolloin kuutiosta saadaan luotua pallo. Lopuksi jokaisen verteksin sijaintia muutetaan vielä melusuodattimien avulla. Mitä suurempi tarkkuus planeetalle annetaan, sitä yksityiskohtaisempi planeetta saadaan luotua, mutta samalla suorituskyky hidastuu huomattavasti. Siksi kyseinen toimenpide on hyvä suorittaa laskentavarjostimen avulla.



Kuva 3. Planeetan luomisprosessi.

5.1 Planeetan luominen laskentavarjostimella

Suurin suorituskyvyn hidastaja on verteksien sijaintien laskenta. Tämä tapahtuu c#-skriptissä for-loopissa, jossa verteksiin kohdistuva laskenta käydään läpi yksi verteksi kerrallaan. Tässä on siis

oiva tilaisuus hyödyntää laskentavarjostimen rinnakkaislaskentaa. Ohjelmassa 2. nähdään, kuinka verteksien sijaintien laskenta tapahtuu yksitellen. Tämän lisäksi samassa for-loopissa lasketaan, mitkä verteksit muodostavat minkäkin kolmion.

```
for (int y = 0; y < resolution; y++)
{
    for (int x = 0; x < resolution; x++)
    {
        int i = x + y * resolution;
        Vector2 percent = new Vector2(x, y) / (resolution - 1);
        Vector3 pointOnUnitCube = localUp + (percent.x - 0.5f) * 2 * axisA + (percent.y - 0.5f) * 2 * axisB;
        Vector3 pointOnUnitSphere = pointOnUnitCube.normalized;
        vertices[i] = shapeGenerator.CalculatePointOnPlanet(pointOnUnitSphere);

        if (x != resolution - 1 && y != resolution - 1)
        {
            triangles[triIndex] = i;
            triangles[triIndex + 1] = i + resolution + 1;
            triangles[triIndex + 2] = i + resolution;

            triangles[triIndex + 3] = i;
            triangles[triIndex + 4] = i + 1;
            triangles[triIndex + 5] = i + resolution + 1;

            triIndex += 6;
        }
    }
}
```

Ohjelma 2. Verteksien sijaintien ja kolmioiden laskenta.

For-loopissa suoritettavan laskentalogiikan siirtäminen laskentavarjostimelle alkaa siten, että ensin alustetaan laskentavarjostin ja puskurit kyseiseen c#-skriptiin. Puskurit nimetään vertexBufferiksi ja triangleBufferiksi. Puskureiden kooksi asetetaan verteksien ja kolmioiden määrät, jotka saadaan laskettua resoluution avulla. Ohjelmassa 3. alustetaan puskurit.

```
int vertexCount = resolution * resolution;
int triangleCount = (resolution - 1) * (resolution - 1) * 6;

vertexBuffer = new ComputeBuffer(vertexCount, sizeof(float) * 3);
triangleBuffer = new ComputeBuffer(triangleCount, sizeof(int));
```

Ohjelma 3. Puskureiden alustaminen.

Laskentavarjostimessa on alustettu neljä muuttujaa, joiden arvot asetetaan c#-skriptin puolelta ohjelman 4. mukaisesti. Lisäksi aiemmin luodut puskurit asetetaan samannimisiin laskentavarjostimesta löytyviin puskureihin. Kun puskureiden arvoja asetetaan, tarvitaan laskentavarjostimen kernelin indeksi, joka saadaan selville käyttämällä FindKernel-funktiota.

```
computeShader.SetInt("resolution", resolution);
computeShader.SetVector("localUp", localUp);
computeShader.SetVector("axisA", axisA);
computeShader.SetVector("axisB", axisB);

int kernerHandle = computeShader.FindKernel("PlanetMeshCompute");

computeShader.SetBuffer(kernerHandle, "vertexBuffer", vertexBuffer);
computeShader.SetBuffer(kernerHandle, "triangleBuffer", triangleBuffer);
```

Ohjelma 4. Muuttujien sekä puskureiden arvojen asettaminen laskentavarjostimelle.

Tässä vaiheessa kutsutaan laskentavarjostimen Dispatch-funktiota, joka suorittaa laskentavarjostimen. Dispatch-funktiossa asetetaan säiejoukkojen määrät x-, y- ja z-akseleilla. Laskentavarjostimessa on määrätty säikeiden määräksi x- ja y-akseleilla kahdeksan, joka tarkoittaa sitä yhden säiejoukon koko on 8x8 eli 64 säiettä.

Ennen kuin Dispatch-funktiota kutsutaan, on laskettava, kuinka monta säiejoukkoa tarvitaan, että kaikki tieto päätyy laskentavarjostimelle. Jos säiejoukkoja lähetetään laskentavarjostimelle liian pieni määrä, jää osa tiedosta silloin laskematta. Ohjelmassa 5. lasketaan säiejoukkojen määrä ja kutsutaan Dispatch-funktiota.

```
int threadGroups = Mathf.CeilToInt(resolution / 8.0f);
computeShader.Dispatch(kernerHandle, threadGroups, threadGroups, 1);
```

Ohjelma 5. Säiejoukkojen määrän laskenta.

Otetaan esimerkkiresoluutioksi 30. Tällöin kuution jokaisella sivulla on 30x30 verteksiä. Pidetään mielessä, että laskentavarjostimessa säikeiden määräksi oli asetettu kahdeksan kappaletta yhdelle akselille. Kun resoluutio jaetaan säikeiden määrällä eli kahdeksalla, tulee vastaukseksi 3,75. CeilToInt-funktio pyöristää vastauksen lähimpään suurempaan kokonaislukuun eli neljään. Tämä siis tarkoittaa, että jos resoluutio on 30, säiejoukkoja tarvitaan neljä kappaletta yhdelle akselille, jotta kaikki tieto saadaan laskettua.

Laskentavarjostimen puolella tapahtuva laskenta toimii täysin samaan tapaan kuin se aiemmin toimi c#-skriptissä. Isoin eroavaisuus on vain se, että verteksien laskenta voidaan suorittaa nyt samanaikaisesti. Ohjelmassa 6. näkyy koko laskentavarjostinskripti.

```
#pragma kernel PlanetMeshCompute

RWStructuredBuffer<float3> vertexBuffer;
RWStructuredBuffer<int> triangleBuffer;

int resolution;
float3 localUp;
float3 axisA;
float3 axisB;
float radius;

float SimpleNoise(float3 position)
{
    // very simple noise
    return frac(sin(dot(position.xy, float2(12.9898, 78.233))) * 43758.5453);
}

[numthreads(8, 8, 1)]
void PlanetMeshCompute(uint3 id : SV_DispatchThreadID)
{
    int x = id.x;
    int y = id.y;
    if (x >= resolution || y >= resolution) return;

    int i = x + y * resolution;
    float2 percent = float2(x, y) / (resolution - 1);

    //convert from a cube to sphere
    float3 pointOnUnitCube = localUp + (percent.x - 0.5f) * 2.0 * axisA + (percent.y - 0.5f) * 2.0 * axisB;
    float3 pointOnUnitSphere = normalize(pointOnUnitCube);

    // Calculate noise for elevation modification
    float noiseValue = SimpleNoise(pointOnUnitSphere * 10.0); // Scale as needed
    float elevation = radius + noiseValue * 2.0; // Modify elevation with noise

    float3 finalPosition = pointOnUnitSphere * elevation;

    vertexBuffer[i] = finalPosition;

    //generate triangles
    if (x != resolution - 1 && y != resolution - 1)
    {
        int triIndex = (x + y * (resolution - 1)) * 6;

        triangleBuffer[triIndex] = i;
        triangleBuffer[triIndex + 1] = i + resolution + 1;
        triangleBuffer[triIndex + 2] = i + resolution;

        triangleBuffer[triIndex + 3] = i;
        triangleBuffer[triIndex + 4] = i + 1;
        triangleBuffer[triIndex + 5] = i + resolution + 1;
    }
}
```

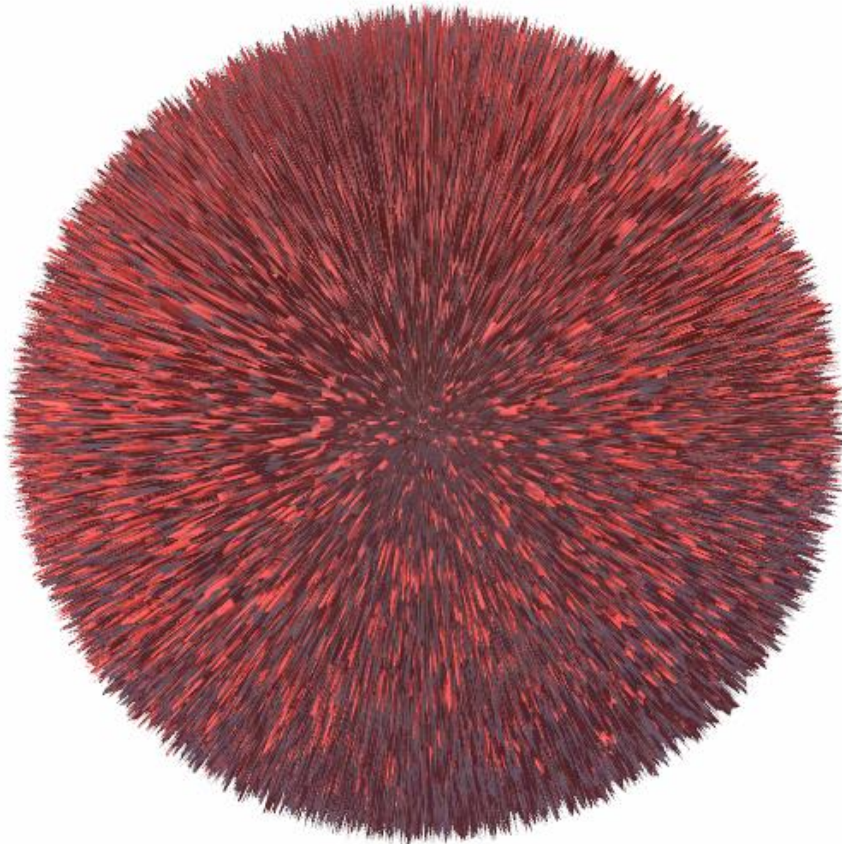
Ohjelma 6. Laskentavarjostimessa suoritettava laskenta vertekseille ja kolmioille.

Lopuksi laskettu tieto otetaan talteen GetData-funktioiden avulla. Kerätty tieto asetetaan oikeille paikoilleen ja puskurit vapautetaan muistista Release-funktiolla, kuten ohjelmasta 7. voidaan nähdä.

```
Vector3[] vertices = new Vector3[vertexCount];
int[] triangles = new int[triangleCount];

vertexBuffer.GetData(vertices);
triangleBuffer.GetData(triangles);

mesh.Clear();
mesh.vertices = vertices;
mesh.triangles = triangles;
mesh.RecalculateNormals();

vertexBuffer.Release();
triangleBuffer.Release();
```

Ohjelma 7. Lasketun tiedon kerääminen ja puskureiden vapauttaminen muistista.

Lopputulos ei näytä samalta kuin ohjevideossa luotu planeetta, koska melusuodattimien sisällyttäminen tähän laskentavarjostimeen ei ollut järkevää niiden monimutkaisuuden takia. Kuvassa 4 on laskentavarjostimella luotu planeetta, jossa on todella yksinkertainen melusuodatin. Planeetassa on lähes 800 000 verteksiä.



Kuva 4. Lopullinen ”planeetta”, joka on luotu laskentavarjostimella.

5.2 Jälkikäsittelyefekti käyttäen laskentavarjostinta

Planeetasta saadaan mielenkiintoisemman näköinen, kun tehdään sille jälkikäsittelyefekti. Teknisesti tämä tapahtuu hyvin samalla tavalla kuin aiemmin kirjoitettu planeetanluomisvarjostin, mutta tällä kertaa varjostimen asetetaan tiedot renderöintitekstuurin muodossa. Kameran näkymä tallennetaan renderöintitekstuuriin c#-skriptissä joka kerta, kun kamera renderöi ruudun kuvan uudelleen. Renderöintitekstuuria muokataan laskentavarjostimen puolella erilaisilla kaavoilla. Sen jälkeen tekstuuri tuodaan takaisin c#-skriptiin, jossa se asetetaan lopulliseksi renderöintikohteeksi.

Unity-pelimoottorissa on valmiina olemassa OnRenderImage-funktio, joka toimii, kun se on yhdistettynä kameraobjektiin. Se ottaa parametreinä kaksi tekstuuria – lähteen ja kohteen. Lähde on kameran näkymä ja kohteeseen asetetaan lopullinen tekstuuri. Ohjelmassa 8. näkyy funktion alkupäätä, jossa myös luodaan uusi renderöintitekstuuri, jos sellaista ei ole vielä olemassa. Uuden tekstuurin kooksi asetetaan kameran näkymän leveys ja korkeus.

```
private void OnRenderImage(RenderTexture source, RenderTexture destination)
{
    // Ensure the RenderTexture is initialized
    if (postProcessTexture == null
        || postProcessTexture.width != source.width
        || postProcessTexture.height != source.height)
    {
        if (postProcessTexture != null)
        {
            postProcessTexture.Release();
        }
        postProcessTexture = new RenderTexture(source.width, source.height, 0);
        postProcessTexture.enableRandomWrite = true;
        postProcessTexture.Create();
    }
```

Ohjelma 8. OnRenderImage-funktion alkupuoli, jossa renderöintitekstuuri luodaan.

On otettava huomioon, että Unityn yleinen renderöintiputki eli URP, ei tue tätä funktiota. Tämän takia projektissa on siis siirryttävä käyttämään Unityn sisäänrakennettua renderöintiputkea eli Built-in Render Pipelinea. URP:ssä on oma tyyli tehdä vastaava toiminnallisuus, mutta tässä projektissa oli järkevämpää mennä yksinkertaisemmalla ratkaisulla.

Laskentavarjostimelle lähetetään kaksi tekstuuria, joista toisessa on kameran näkymä eli lähde, ja toisessa ei mitään. Tyhjää tekstuuria muokataan laskentavarjostimella. Tekstuurien lisäksi varjostimelle lähetetään kaksi muuta muuttujaa, joita hyödynnetään jälkikäsittelyefektin kirjoittamisessa. Ohjelmassa 9. on kaikki laskentavarjostimelle lähetetty tieto.

```
// Set compute shader parameters
computeShader.SetTexture(kernelHandle, "InputTexture", source);
computeShader.SetTexture(kernelHandle, "Result", postProcessTexture);

computeShader.SetVector("EdgeColor", edgeColor);
computeShader.SetFloat("EdgeStrength", edgeStrength);
```

Ohjelma 9. Tietojen lähettäminen laskentavarjostimelle.

Kun laskentavarjostimen Dispatch-funktiota kutsutaan, kannattaa säiejoukkojen määrät laskea erikseen x- ja y-akseleille, koska tekstuuri ei välttämättä ole neliön muotoinen. Ohjelmassa 10. lasketaan säiejoukkojen koot eri akseleilla ja kutsutaan Dispatch-funktiota.

```
// Dispatch compute shader
int threadGroupsX = Mathf.CeilToInt(source.width / 8f);
int threadGroupsY = Mathf.CeilToInt(source.height / 8f);
computeShader.Dispatch(kernelHandle, threadGroupsX, threadGroupsY, 1);
```

Ohjelma 10. Laskentavarjostimen Dispatch-funktion kutsu ja säiejoukkojen määrien laskenta.

Tarkoituksena on tehdä jälkikäsittelyefekti, joka tunnistaa reunoja ja piirtää niiden kohdalle valitun värisiä pikseleitä. Tällaisella efektillä saadaan luotua sarjakuvamainen ulkonäkö. Ensimmäisenä selvitetään tekstuurin UV-koordinaateilla, mitä pikseliä aiotaan tarkastella. Pikselin väri otetaan talteen ja sen kahdeksaa ympäröivää pikseliä verrataan siihen. Mikäli pikseleissä on suuri ero joko väreissä tai kirkkaudessa, on pikseli silloin reunan kohdalla. Reunan vahvuus voidaan laskea ja heikommat reunat saadaan karsittua pois yhtälöstä. Tarpeeksi vahvojen reunapikseleiden väriarvoa ja vahvuutta muokataan c#-skriptistä tuotujen muuttujien avulla. Lopuksi käsitelty pikseli tallennetaan tyhjään tekstuuriin. Ohjelmassa 11. näkyy reunan tunnistuslogiikka kokonaisuudessaan.

```
[numthreads(8, 8, 1)]
void PostProcessing (uint3 id : SV_DispatchThreadID)
{
    int width, height;
    Result.GetDimensions(width, height);
    float2 uv = id.xy / float2(width, height);

    // Sample the input texture
    float4 originalColor = InputTexture.SampleLevel(sampler_PointClamp, uv, 0);

    float3x3 sobelX = float3x3( 1, 0, -1,
                                2, 0, -2,
                                1, 0, -1);

    float3x3 sobelY = float3x3( 1, 2, 1,
                                0, 0, 0,
                                -1, -2, -1);
    // Calculate gradients
    float Gx = 0.0;
    float Gy = 0.0;
    for (int x = -1; x <= 1; x++)
    {
        for (int y = -1; y <= 1; y++)
        {
            float4 sampleColor = InputTexture.SampleLevel(sampler_PointClamp, uv + float2(x, y) / float2(width, height), 0);
            Gx += sampleColor.r * sobelX[x + 1][y + 1];
            Gy += sampleColor.r * sobelY[x + 1][y + 1];
        }
    }

    // Calculate magnitude and filter out weak edges
    float edgeMagnitude = sqrt(Gx * Gx + Gy * Gy);
    float threshold = 0.01;
    if (edgeMagnitude < threshold) edgeMagnitude = 0.0;

    // Blend the edge color with the original color based on edge magnitude
    float4 finalColor = originalColor + EdgeColor * edgeMagnitude * EdgeStrength; // Subtract the edge effect

    // Clamp the final color to [0, 1]
    finalColor.rgb = saturate(finalColor.rgb);

    // Write to output
    Result[id.xy] = finalColor;
}
```

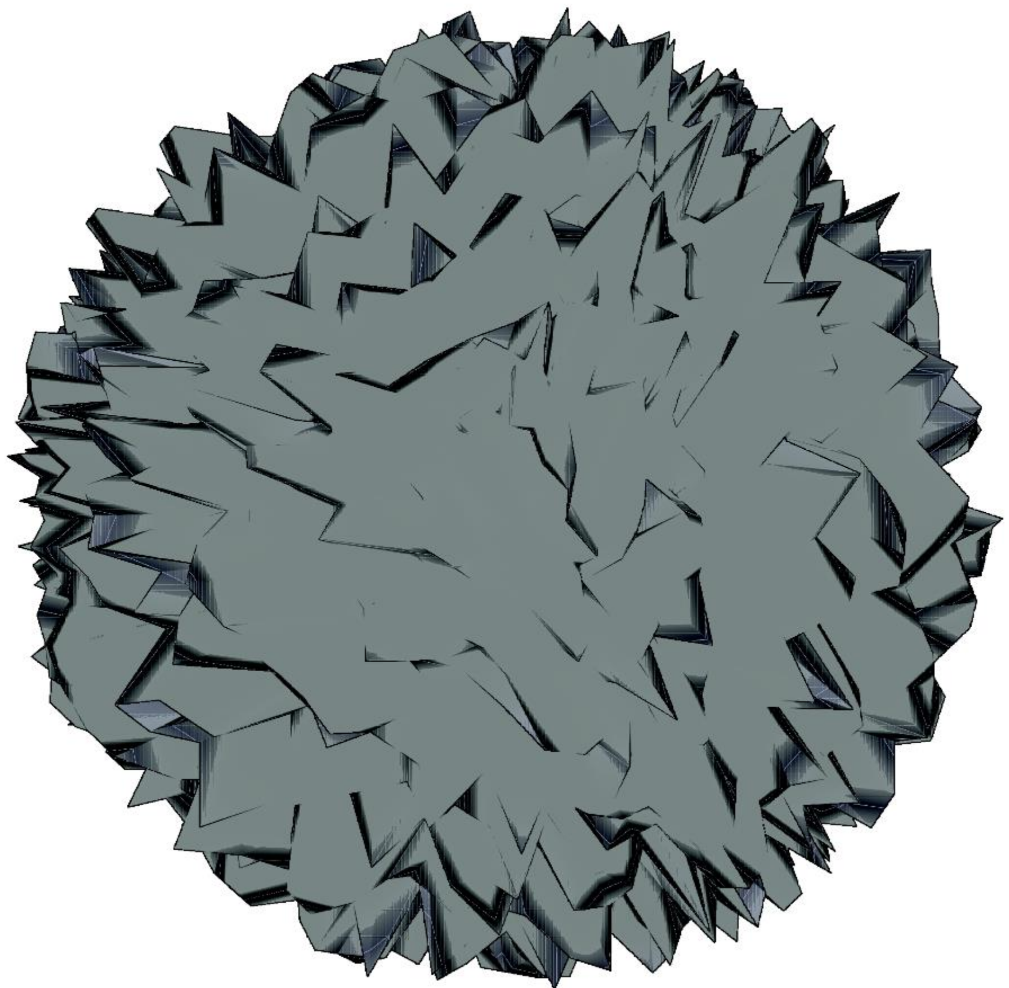
Ohjelma 11. Reunan tunnistaminen alusta loppuun laskentavarjostimella.

Koska laskentavarjostin kirjoittaa tietoa suoraan tyhjälle renderöintitekstuurille, ei tietoa tarvitse kerätä takaisin c#-skriptissä. Riittää vain, että asetetaan muokattu tekstuuri OnRenderImage-funktion kohdetekstuurin tilalle. Ohjelmassa 12. tämä tapahtuu Blit-funktiolla, joka kopioi muokatusta tekstuurista tiedot lopulliseen kohdetekstuuriin.

```
Graphics.Blit(postProcessTexture, destination);
```

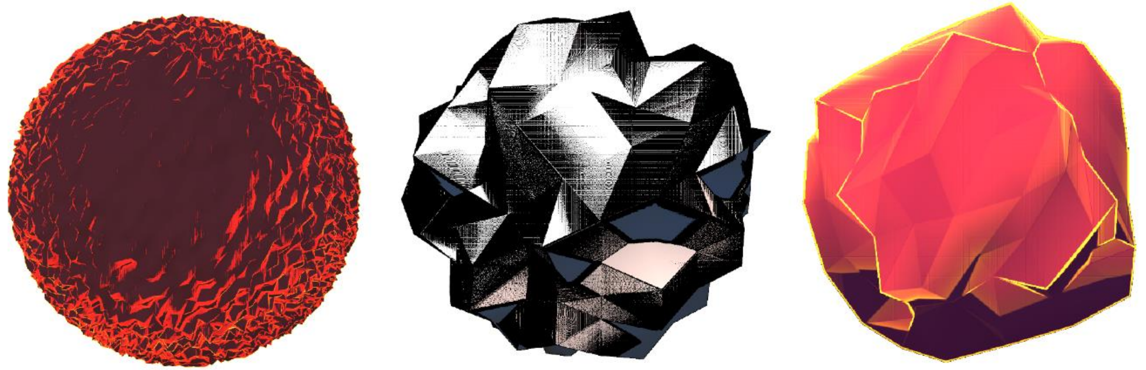
Ohjelma 12. Muokatun tekstuurin tieto kopioidaan kohdetekstuuriin.

Lopputulos jälkikäsittelyefektille on kuvan 5 mukainen. Kuvassa on säädetty materiaalia ja valaistusta, jotta reunat tulisivat halutulla tavalla esille. Efektin laatuun vaikuttaa tekstuurin resoluutio, mikä käy järkeen, sillä efekti perustuu yksittäisten pikseleiden muokkaamisen. Lopputuloksen kuva on 4k-laatuisesta tekstuurista.



Kuva 5. Lopputulos reunantunnistusefektin jälkeen.

Lopuksi kuvassa 6 on vielä muutama eri planeetta, jotka on luotu aiemmin tehdyillä laskentavarjostimilla. Planeetoissa on käytetty eri värejä ja planeetoille on asetettu eri resoluutiot, minkä avulla niille saadaan hieman eri muotoja.



Kuva 6. Erilaisia planeettoja, jotka on kaikki luotu samalla koodilla, mutta eri arvoilla.

6 Päätäntö

Kun aloitin suunnittelemaan opinnäytetyötä, tiesin että haluan tehdä sen varjostimista. Aloin siis tutkimaan eri varjostintyyppejä ja olin jopa valmis tekemään opinnäytetyö, jossa tutustun jokaiseen varjostintyyppiin. Tämä kuitenkin tuntui hieman liian kunnianhimoiselta ja epäselvältä, mutta onneksi huomioni kiinnittyi yhteen varjostintyyppiin, joka erosi muista.

Laskentavarjostimet eivät olleet minulle ennestään tuttuja, joten tätä opinnäytetyötä tehdessäni olen päässyt oppimaan paljon uutta. Oikeastaan mitkään varjostimet eivät olleet tuttuja varjostinkoodin kirjoittamisen kannalta. Aiempaa kokemusta löytyi vain visuaalisista varjostineditoreista, kuten Unityn Shader Graph -työkalusta ja Unrealin materiaalieditorista. HLSL-kielen oppimisesta on yleisesti hyötyä minulle, koska graafisen ohjelmoijan tai teknisen artistin rooliin pääseminen on minulla tavoitteena tulevaisuudessa.

Ohjevideon ottaminen pohjaksi käytännön toteutukselle oli mielestäni hyvä idea, sillä sen avulla sain selkeän tavoitteen projektille. Olin jo pitkään halunnut tehdä jonkunlaisen planeettageneraattorin, ja tässä siihen ilmeni jonkinlainen mahdollisuus. Projektia on mahdollista vielä jatkaa pidemmälle lisäämällä vastaavanlaiset melusuodattimet kuin ohjevideossa oli tehty, mutta planeetan luomislogiikkaa voidaan myös viedä aivan eri suuntaan. Opittuja asioita voidaan myös hyödyntää muissakin kuin vain planeettojen luomisessa. Esimerkiksi tasaisten maastojen luominen on nyt huomattavasti selkeämmän tuntuista, kun on ensin niin sanotusti hypännyt syvään päätyyn, tekemällä planeetan.

Jälkikäsittelyefektin luominen oli viime hetken idea. Alkuperäisenä suunnitelmana minulla oli optimoida ja vertailla tehokkuutta c#-skriptin ja laskentavarjostimen välillä, mutta mielenkiinto siihen puuttui. Jälkikäsittelyefektin tekeminen oli mielenkiintoisempaa ja siinä pääsi tekemään samantyylisiä asioita, joita muilla varjostintyypeillä tehdään. En ole varma, onko laskentavarjostin tehokkain tapa tehdä jälkikäsittelyefektejä, jos verrataan muihin varjostintyyppeihin, mutta ainakin tuli kokeiltua. Myös se, että tapa, jolla tein jälkikäsittelyefektin, ei toimi Unityn URP:ssä, on mielestäni huono asia. Olisi ollut hyödyllisempää opetella URP:n tapa tehdä sama asia.

Yleisesti opinnäytetyössä pysyin hyvin aikataulussa ja sain joka viikko edistettyä joko käytännön toteutusta tai itse kirjoittamista. Ainoa hetki, kun opinnäytetyö tuntui jumittavan paikallaan, oli

kun yritin vertailla ja optimoida planeetanluomislogiikkaa, mutta onneksi siihen löytyi vaihtoehtoinen ratkaisu. Olen tyytyväinen lopputulokseen ja oppimaani. Tästä on hyvä jatkaa varjostimien käyttöä tulevissa peliprojekteissa.

Lähteet

1. Microsoft. Compute Shader Overview. [Internet]. 4.9.2021 [viitattu 24.11.2024]. Saatavilla: https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader

2. OpenGL. Compute Shader. [Internet]. 22.4.2019 [viitattu 24.11.2024]. Saatavilla: https://www.khronos.org/opengl/wiki/Compute_Shader

3. Compute Shaders in Unity: GPU Computing, First Compute Shader. Danylo Hoshko [Internet]. 15.03.2023 [viitattu 27.3.2025]. Saatavilla: https://www.artstation.com/blogs/degged/vVzZ/compute-shaders-in-unity-gpu-computing-first-compute-shader

4. Halladay K. Getting Started With Compute Shaders In Unity. [Internet] 27.6.2014 [viitattu 24.11.2024]. Saatavilla: https://kylehalladay.com/blog/tutorial/2014/06/27/Compute-Shaders-Are-Nifty.html

5. NVIDIA. RTX Technology [Internet]. 2024 [viitattu 24.11.2024]. Saatavilla: https://developer.nvidia.com/rtx/ray-tracing

6. Radiangames. Instruments of Destruction. [videopeli]. 2024. Radiangames

7. Baeldung. What's a Buffer? [Internet]. 18.3.2024 [viitattu 24.11.2024]. Saatavilla: https://www.baeldung.com/cs/buffer

8. Unity Documentation. RenderTexture. [Internet]. 2024 [viitattu 24.11.2024]. Saatavilla: https://docs.unity3d.com/ScriptReference/RenderTexture.html

9. Unity Documentation. Create a compute shader. [Internet]. 2024 [viitattu 24.11.2024]. Saatavilla: https://docs.unity3d.com/Manual/class-ComputeShader-create.html

10. Unity Documentation. ComputeShader. [Internet]. 2024 [viitattu 24.11.2024]. Saatavilla: https://docs.unity3d.com/ScriptReference/ComputeShader.html

11. Unity Documentation. ComputeBuffer.stride. [Internet]. 2024 [viitattu 24.11.2024]. Saatavilla: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/ComputeBuffer-stride.html

12. Unity Documentation. ComputeBuffer. [Internet]. 2024 [viitattu 24.11.2024]. Saatavilla: https://docs.unity3d.com/ScriptReference/ComputeBuffer.html

13. Game Dev Guide. Getting Started with Compute Shaders in Unity [Video]. YouTube. 31.12.2020 [viitattu 24.11.2024]. Saatavilla: https://youtu.be/BrZ4pWwkpto

14. Sebastian Lague. Procedural Planet Generation [Video]. Youtube. 17.8.2018 [viitattu 21.2.2025]. Saatavilla: https://youtu.be/QN39W020LqU?list=PLFt_AvWsXl0cONs3T0By4puYy6GM22ko8