Miika Haukkala

Lentävien saarien proseduraalinen generointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

3.4.2023

Tiivistelmä

Tekijä: Miika Haukkala

Otsikko: Lentävien saarien proseduraalinen generointi
Sivumäärä: 43 sivua
Aika: 3.4.2023

Tutkinto: Insinööri (AMK)
Tutkinto-ohjelma: Tieto- ja viestintätekniikka
Ammatillinen pääaine: Pelisovellukset
Ohjaaja: Lehtori Miikka Mäki-Uuro

Insinöörityössä pyrittiin proseduraalisella generoinnilla luomaan pelisovelluksiin käyttökelpoisia lentäviä saaria. Työ toteutettiin Unity-pelimoottorilla.

Työn aikana onnistuttiin luomaan lentäviä saaria kahdella eri toteutustavalla vaiheittain. Ensimmäisessä tavassa lentävä saari luotiin yhdistämällä kaksi samankaltaista korkeuskartasta luotua tahkoverkkoa. Toisessa toteutustavassa muodostettiin vokselien määrittämästä analyyttisesta volyymistä tahkoverkkoja marssikuutioalgoritmilla ruudukkoon.

Insinöörityön lopputuloksena syntyneiden toteutuksien perusteella voidaan arvioida eri lähestymistapojen vahvuuksia ja heikkouksia lentävien saarien proseduraalisessa generoinnissa. Tämän lisäksi voidaan insinöörityön pohjalta todeta laadukkaiden proseduraalisien algoritmien luomisen vaativan aikaa ja asiantuntemusta.

Avainsanat: proseduraalinen generointi, kohina, marssikuutiot, korkeuskartat, lentävät saaret

Abstract

Author:	Miika Haukkala
Title:	Procedurally generating floating islands
Number of Pages:	43 pages
Date:	3 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and communication technologies
Professional Major:	Game applications
Supervisor:	Miikka Mäki-Uuro, Lecturer

The goal of the final year project covered by this thesis was to use procedural generation to create floating islands that could be used in game applications. The project was carried out using the Unity game engine and related compatible libraries.

During the project, two different implementation methods were successfully used to create floating islands in a continuous manner. In the first method, a floating island was created by combining two similar point grids created from an elevation map. In the second implementation, a marching cube algorithm was used to generate chunked meshes from an analytical volume determined by voxels.

The resulting implementations from this final year project provide a base for evaluating strengths and weaknesses of different approaches in procedural generation of floating islands. In addition, it can be argued from the results of this thesis that the creation of high quality procedural algorithms requires time and domain expertise.

Keywords: Procedural generation, noise, marching cubes, heightmaps, floating islands

Sisällys

1 Johdanto 1

2 Proseduraalinen generointi 1

2.1 Proseduraalisuus 2
2.2 Esimerkkejä proseduraalisesta generoinnista 3
2.3 Proseduraalisia lentäviä saaria peleissä 5

3 Kohina, geometria ja lentävät saaret 7

3.1 Kohina 8
3.2 Geometria pelissä 12
3.3 Korkeuskartat ja saarien luominen 14
3.4 Koristelu 17
3.5 Olemassa olevien implementaatioiden tarkastelu 19

4 Marssikuutiot ja vokselit 20

4.1 Marssikuutiot 21
4.2 Yksisäikeinen implementaatio 22
4.3 Säikeistäminen ja näytönohjain 28
4.4 Tyylittely 36

5 Tulokset ja analyysi 38

6 Yhteenveto 40

Lähteet 42

1 Johdanto

Insinöörityön tarkoituksena oli tutustua peleissä yleisesti käytettyyn proseduraaliseen generointiin ja Unity-pelimoottorin tarjoamaan työnkulkuun proseduraalisten algoritmien tuottamisessa. Projektiksi valittiin tuottaa reaaliaikaisesti lentäviä saaria luova proseduraalinen algoritmi.

Työn tekemistä aiheesta motivoi aiheen syvyys ja haastavuus. Proseduraaliset algoritmit vaativat huomiota käytetyn algoritmin nopeuteen ja monipuolisuuteen, joten algoritmia ohjelmoitaessa on mahdollista tehdä paljon mielenkiintoisia valintoja. Monimutkaisen algoritmin tuottaminen on hyvä tapa kehittää ammatillista osaamista.

Luvussa 2 esitellään proseduraalisen generoinnin käsite ja sitä käyttäviä pelejä. Luvussa 3 esitellään tärkeitä käsitteitä työn taustoittamiseksi ja ensimmäinen, korkeuskarttapohjainen, implementaatio lentävien saarien toteuttamiseksi.

Luvussa 3 esiteltävä aiempi implementaatio koettiin kykenemättömäksi saavuttamaan persoonallisia tarpeita insinöörityötä tehdessä, tarkalleen ottaen kielekkeiden ja luolien mahdollisuuden puute. Luvussa 4 esitellään toinen implementaatiotapa pohjautuen vokseleihin ja marssikuutioalgoritmiin lentävien saarien toteuttamiseksi, ja luvussa 5 eritellään työssä saavutettua tulosta ja mahdollisia parannusvaihtoehtoja.

2 Proseduraalinen generointi

Proseduraalinen generointi on tekniikka, jota käytetään videopelien kehityksessä pelisisällön, kuten tasojen, luomiseen. Proseduraalisen generoinnin perusidea on luoda elementtejä dynaamisesti käyttämällä algoritmeja sen sijaan, että jokainen elementti luotaisiin manuaalisesti. Proseduraalisen generoinnin tavoitteena on luoda ainutlaatuista ja monipuolista sisältöä, joka voidaan tuottaa satunnaistetulla, toistettavalla tavalla. Tämä mahdollistaa kehittäjälle suuren

määrän uniikkia sisältöä, jota voidaan luoda yleensä suoritusajassa. Tällä pyritään pelaajalle arvaamattomampaan ja kiinnostavampaan pelikokemukseen.

Yleisiä esimerkkejä videopelien proseduraalisesta generoinnista ovat proseduraalisesti luodut tasot roguelike-peleissä, satunnaisesti luodut maastot avoimien maailmojen peleissä ja satunnaiset esineet erilaisissa roolipeleissä. [1.]

2.1 Proseduraalisuus

Kuten edellä mainittiin, perusidea proseduraalisen generoinnin käytössä on mahdollistaa pelien kehittäjille suurempi määrä pelisisältöä ilman, että heidän tarvitsee luoda sisältöä manuaalisesti.

Yksinkertaisesti ajateltuna proseduraalisuus vähentää kehittäjien työmäärää ja parantaa tehokkuutta, sillä algoritmit voivat luoda sisältöä nopeammin kuin pelisuunnittelija. Proseduraalisen generoinnin käyttö voi kuitenkin joskus johtaa epätasapainoisiin tai toistuviin pelikokemuksiin, jos algoritmit eivät ole tarpeeksi monipuolisia. Proseduraalisen algoritmin luominen ja hiominen vie usein paljon aikaa, jolloin saadut konkreettiset voitot manuaaliseen luomisprosessiin verrattuna voivat olla pieniä.

Pelinkehittäjän olisi hyvä olla tietoinen mahdollisista tuotantoriskeistä kehittäessään proseduraalisia algoritmeja ja valvoa pelikokemukseen liittyvää laatua kehityksen aikana.

Proseduraalisiin algoritmeihin liittyy yleensä vahva rakenteellisuus, jonka ympärillä kaikki satunnaiset osiot voidaan tuottaa turvallisesti algoritmien parametreina. Tämä luo hyvän kasvupohjan välttämättömälle toistettavalle kokemukselle.

Esimerkiksi satunnaista ampuma-asetta rakentaessa generaattori voisi valita aina järjestyksessä aseen yleisen tyypin, lippaan koon, piipun tyylin, aseen har-

vinaisuuden ja viimeisenä vahinkopisteiden määrän. Näin voitaisiin myöhempien vaiheiden satunnaiset valinnat tehdä sopivan suppeasta osajoukosta hyvän pelikokemuksen takaamiseksi.

Mahdollisimman laadukkaan generaattorin valmistamisessa on tärkeää löytää aikaisessa vaiheessa tarkka ja rajattu määritelmä sille, mitä generaattorilta halutaan. Esimerkiksi tekstiä luovassa generaattorissa on parempi keskittyä johonkin tiettyyn tekstityyliin, kuin alkaa valmistamaan yleispätevää algoritmia, joka kirjoittaa yleispätevää tekstiä. [2.] Lentävien saarten kohdalla yksityiskohtainen määrittely voi koskea esimerkiksi mahdollista luolien ja kielekkeiden olemassaoloa, kasvillisuuden tarvetta, piirrettyjen muotojen tarkkuutta, käyttövaatimusta peliin, mahdollisia fysiikkavaatimuksia, raakaa laskentanopeutta tai mahdollisuutta ladata alueita osissa.

Tarkkaa teknistä määrittelyä voidaan pitää toisarvoisena, mutta joskus huolimattomuus saattaa aiheuttaa kokonaisten järjestelmien uudelleenkirjoittamista. Näin kävi esimerkiksi kuuluisalle Dwarf Fortress -pelille, kun se kehittyi pisteeseen, jossa kehittäjän täytyi lisätä ominaisuuksia, jotka tarvitsivat x- ja y-akselin lisäksi kolmatta, syvyyttä edustavaa z-akselia proseduraalisiin maailmoihin. [3.]

Proseduraalisia algoritmeja ei aina käytetä pelin suoritusaikana toteutettavaan sisällön luomiseen, vaan usein myös pelinkehitystyökaluissa. Esimerkiksi suosioon nousseen Houdini-moottorin työnkulku perustuu proseduraalisuuteen, jossa erilaisia solmurakenteita voidaan käyttää visuaalisten elementtien tuottamiseen. [4.]

2.2 Esimerkkejä proseduraalisesta generoinnista

Mojang-pelistudion vuonna 2011 julkaisema peli Minecraft käyttää proseduraalista generointia maailmansa luomisessa. Pelissä esiintyy maaston korkeusvaihteluja ja erilaisia elementtejä, kuten jokia ja alueellisia biomeja (kuva 1) sekä maan alla erilaisia luolastoja. Maailma kannustaa pelaajaa tutkimaan sekä py-

symällä mielenkiintoisena että sijoittamalla erilaisia resursseja tietynlaisille alueille. Pelaaja voi muokata maailmaa ja rakentaa omia rakennelmia ja tehdä pelin edetessä luontomaisemasta ihmisasuttua.

Minecraft on hyvä esimerkki vokseleihin eli volumetrisiin pikseleihin perustuvasta generaatiosta, jossa 3D-koordinaatisto täytetään jokaiselle pisteavaruudessa määritetylle ruudukon pisteelle yksilöidyllä datalla, jonka peli piirtää sitten pelaajan kuvaruudulle näkyviin. Yksittäinen vokseli sisältää esimerkiksi tietoa pisteen materiaalista, valotustasosta ja biomista.

Kuva 1. Minecraftin maailma pysyy kiinnostavampana, kun maailmasta löytyvät asiat ovat sopivan monipuolisia [5].

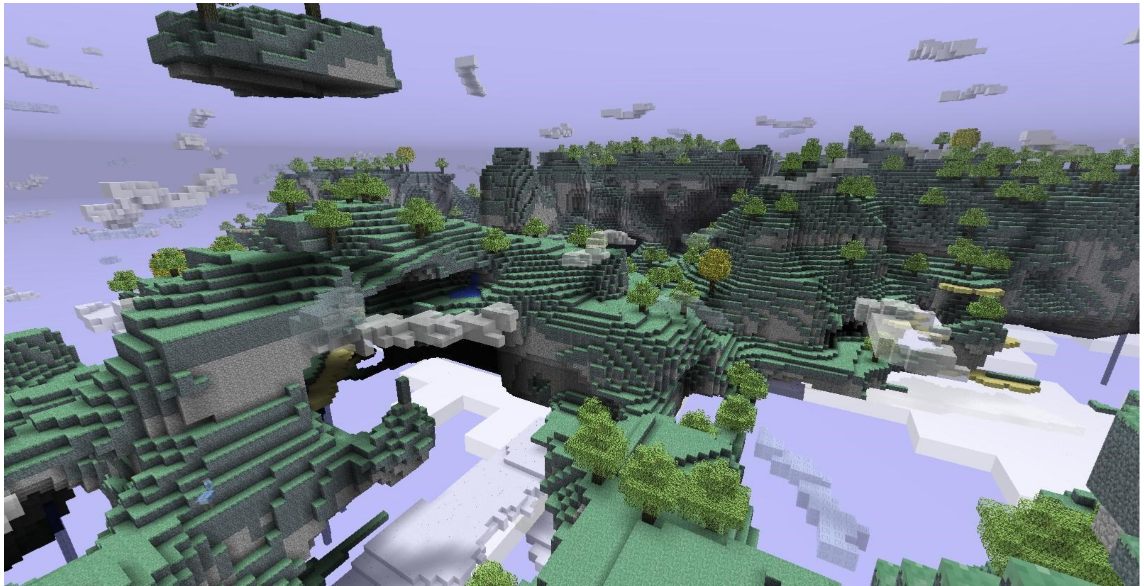
Deep rock galactic -peli on Ghost ship games -pelistudion kehittämä peli. Siinä pelaajat ohjaavat avaruuskääpiöiden tiimiä, joka lähetetään eri avaruusplaneetoille poimimaan arvokkaita mineraaleja ja muita luonnonvaroja planeetan sisällä olevista luolista. Joka kerta, kun pelaaja aloittaa uuden tehtävän, planeetan sisäiset luolastot ja resurssien sijainti luodaan proseduraalisesti (kuva 2).

Yksinkertaisiin luolastoihin saadaan erilaisia ympäristöjä vaihtelemalla luovasti kasvillisuutta, mineraaleja ja luolaston geometrian väripalettia. Suuressa roolissa ovat myös luolan geometrian muodot, kuten jääluolien piikkimäisyys suhteessa aavikkoluolien pyöreyteen.

Kuva 2. Deep Rock Galacticin luolat sisältävät erilaisia elementtejä, jotka pitävät maailmaa mielenkiintoisena [6].

2.3 Proseduraalisia lentäviä saaria peleissä

Lentävistä saarista on tehty muutamia erilaisia variaatioita vuosien varrella. Yksi hyvä esimerkki on Minecraft-modi ”The Ather”, jonka tekijä loi vastakohdan Minecraft-pelin jo valmiille ”The neather” -ulottuvuudelle, joka muistuttaa kristillistä helvettiä (kuva 3).



Kuva 3. Minecraftiin käyttäjien tekemä ”The Aether” -modi sisältää tavallisen Minecraftin proseduraalisen maastogeneraattorin tyyppistä maastoa muunneltuna [7].

Toinen lentäviä saaria käyttävä peli on Terraria, jossa yksittäiset kohteet kartassa ovat nimeltään ”floating islands” (kuva 4).

Kuva 4. Terrariassa esiintyvä ”floating island”, josta löytyy aarrehuone [8].

Näiden kahden lisäksi videopeleistä löytyy paljon muitakin, myös klassisilla mallinnusmenetelmillä tuotettuja lentäviä saaria, kuten Hyrule temple -niminen kartta Super smash brothers -peleissä.

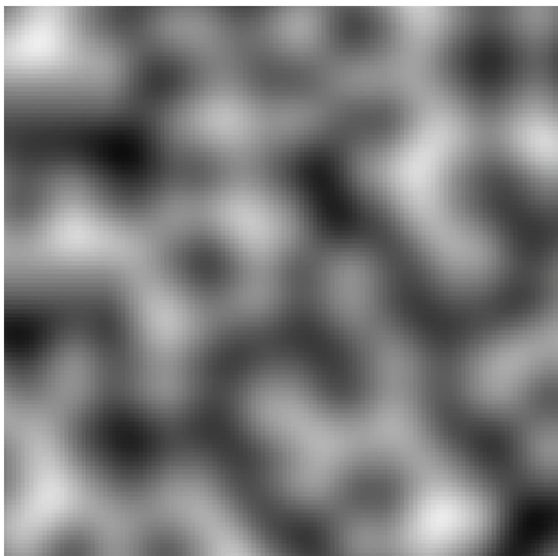
3 Kohina, geometria ja lentävät saaret

Tässä luvussa esitellään itse tekemäni projektin alkuosaa, jonka aikana tavoitteena oli luoda yksinkertaisia lentäviä saaria korkeuskarttoihin perustuen. Luvussa ensiksi kerrytetään taustatietoa myöhemmin luvussa esiintyvälle implementaatiolle. Viimeisessä alaluvussa käsitellään muiden lähteiden vastaavia implementaatioita.

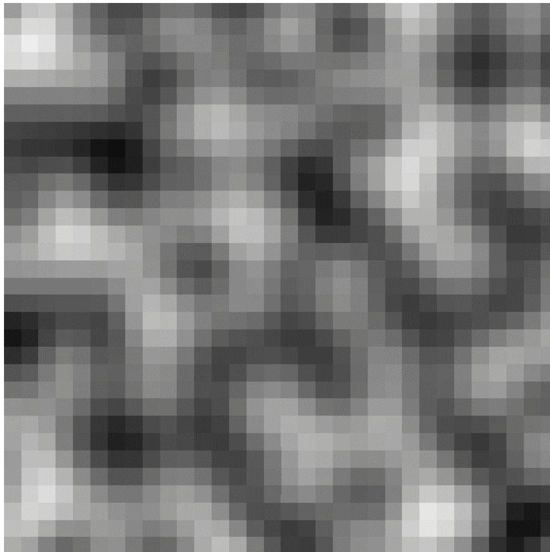
3.1 Kohina

Usein videopeleissä käytetty tekniikka maaston luomiseen ovat niin sanotut korkeuskartat. Tätä tekniikka tukevat muun muassa tunnetut maastogeneraattorit World Machine [9] ja Gaea [10]. Monet pelimoottorien maastoimplementaatiot suoraan tukevat korkeuskarttoja eri ohjelmien integraation helpottamiseksi. Nimensä mukaisesti korkeuskartoilla tehty maasto perustuu arvokarttoihin, joita käytetään määrittelemään korkeuksia geometriassa.

Joskus korkeusarvojen luonnissa käytetään apuna matemaattista kohinaa (engl. noise), joista tunnetuin on Perlin-kohina (engl. Perlin noise). Kohinasta tekee erityisen hyödyllisen deterministinen näennäissatunnaisuus, rajaton yksityiskohtaisuus ja raja-arvojen kulku käytännössä äärettömään. Kuvassa 5 on Perlin-kohinaa, josta lasketaan korkeammalla taajuudella näytteitä suhteessa kuvaan 6. Tämä demonstroi hyvin kohinan kykyä, kun tarvitaan lisää yksityiskohtia.



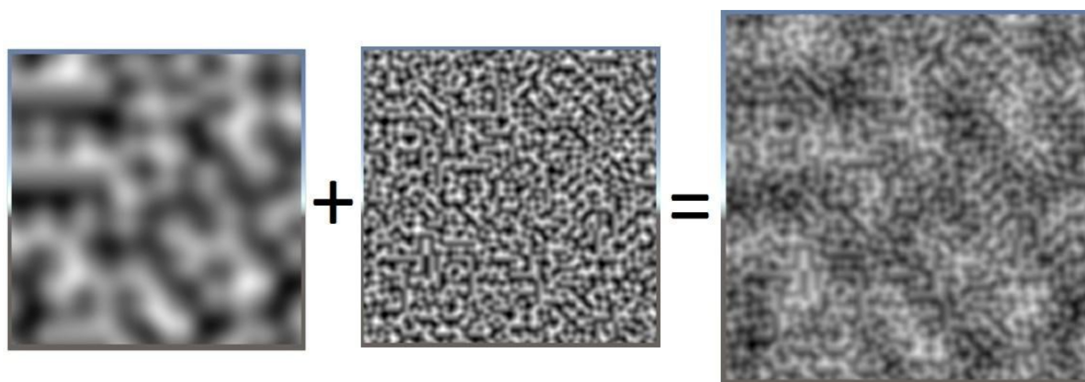
Kuva 5. Unity-pelimoottorin implementaatio Perlin-kohinasta, jonka arvoja näytetään tekstuurissa.



Kuva 6. Sama Unityn Perlin-kohinan implementaatio kuvassa 5, mutta näytteitä kohinasta otetaan pienemmällä taajuudella.

Kohinan sijasta saatetaan käyttää muistiin ladattavia kohinatekstuureja laskentatehon säästämiseksi. Näihin kohinatekstuureihin ohjataan nimensä mukaisesti kohinan arvoja määrittelemään tekstuurin väriarvot. Esimerkiksi mustavalkoisissa kuvissa kohinan arvo valkoisissa pikseleissä on 1 ja mustissa 0, harmaat arvot puolestaan edustavat välimaastoa.

Erilaisilla kohinan yhdistelmillä ja muunteluilla voidaan saada aikaan erilaisia muotoja, joita voidaan sitten hyödyntää esimerkiksi maaston luomiseen. Yksi tällainen mahdollisuus on esimerkiksi kohinan oktaavien lisääminen, jolloin kohinaa lisätään taajuudella itseensä (kuva 7). Toisin kuin musiikin oktaavissa, jossa taajuudet aina kaksinkertaistuvat toisiinsa nähden, kohinan taajuuksien väli voidaan määritellä vaihtelemaan mielivaltaisella arvolla [11]. Tämä tekniikka lisää kohinan yksityiskohtaisuutta.



Kuva 7. Kahden kohinan keskiarvo tuottaa sekä isoja että pieniä yksityiskohtia.

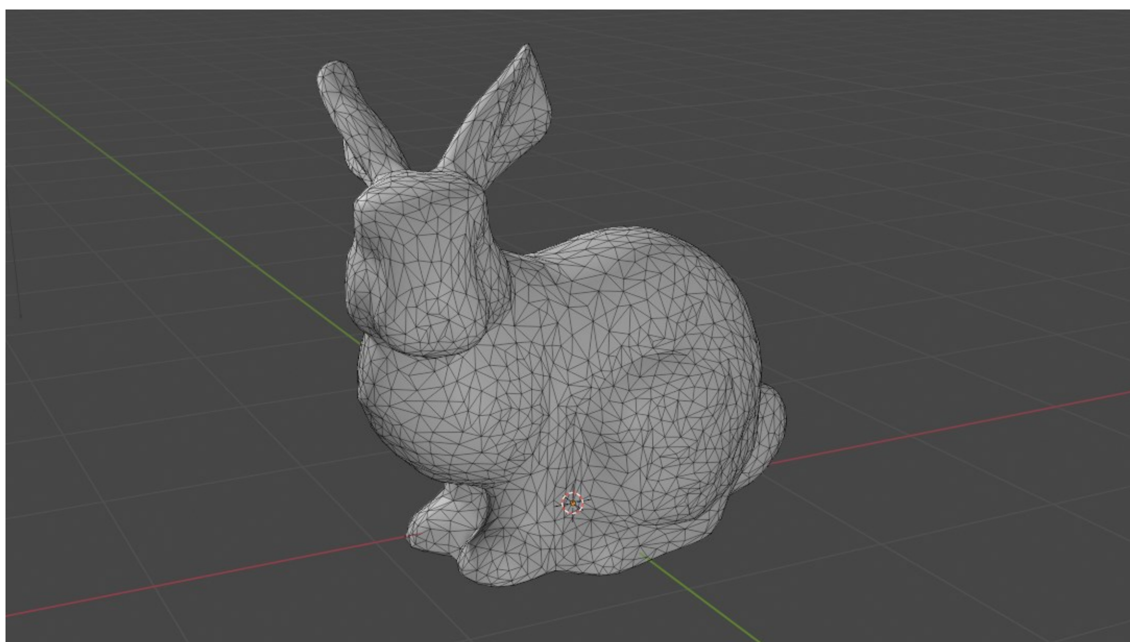
Insinöörityöprojektiin valitun pelimoottori Unityn sisäänrakennettu Perlin-kohinaimplementaatio ei ole kovinkaan laadukas. Unityn implementaatio kärsii kohinan liiallisesta kuvion toistuvuudesta, joka aiheuttaa esimerkiksi arvojen muodostumista viivoiksi kohinaan oktaaveja lisättäessä. Unityn implementaatio kärsii myös toistokäyttäytymisestä eri akseleiden lohkoissa (kuva 8). Molemmat aiheuttavat pahimmillaan suuria ongelmia algoritmeissa ja ovat tunnetusti huonoja pseudosatunnaisuuden ominaisuuksia.

Kuva 8. Unityn sisäänrakennettu Perlin-kohinan implementaatio toistuu akseleiden määrittelemissä lohkoissa.

Unityn sisäisen Perlin-kohinan voi korvata joko omalla tai jostain kirjastosta löytyvällä kohinalla. Vaihtoehtoisesti Unity on korvannut kohinatoteutuksen paremmilla vaihtoehdoilla, jotka löytyvät Unity Mathematics -paketista. Esimerkiksi kohinaan erikoistuva kirjasto FastNoiseLite [12] on kätevä, sillä siitä löytyy erilaisia kohinatyyppejä, funktioita on useilla kielillä ja funktiot ovat nopeampia, eivätkä kohinassa toistu samat ongelmat kuin Unityn kohinaimplementaatiossa.

3.2 Geometria pelissä

3D-peleissä nähdyt hahmot ja maisemat tehdään usein piirtämällä geometrisia malleja avaruudessa ruudulle. Näiden geometristen objektien manuaalista tuottamista tietokoneohjelmassa kutsutaan mallintamiseksi. Kuvassa 9 näkyy geometrinen malli, jolla on yritetty mallintaa pupun olemusta.

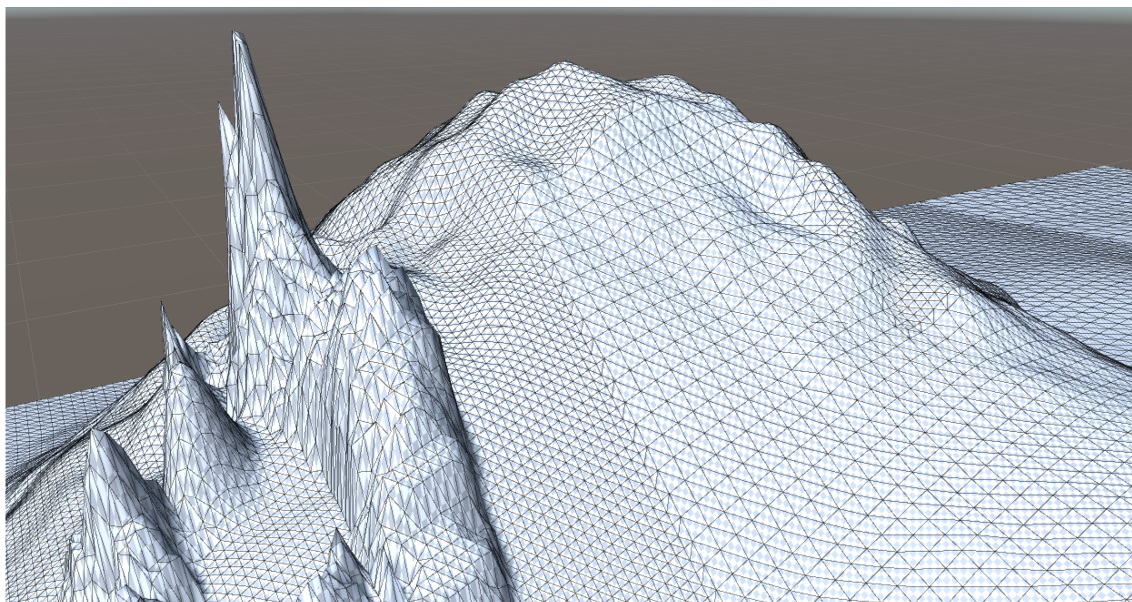


Kuva 9. ”Standford bunny” -malli avattuna Blender-ohjelmassa, josta on otettu näkyviin mallin kolmiot [13].

3D-mallien perustana toimivat usein nykypäivänä kolmiot. Kolmio on mallinnuksessa noussut suosioon, sillä se on yksinkertaisin geometrinen muoto, jolla voidaan kuvata mikä tahansa muu tahko ja kolmion kärkipisteet ovat aina samalla tasolla. Moderneista tietokoneista ja puhelimista löytyvän näytönohjain-komponentin yksi tärkein tehtävä on erikoistua kolmioihin liittyvään laskentaan.

Malleja tuottaessa on yleistä käyttää neliöitä, sillä neliöt mahdollistavat yksinkertaisia silmukoita, joilla on helppo mallintaa. Lisäksi jotkut mallien tuottamisessa käytetyt algoritmit tuottavat ennalta-arvattavampia tuloksia neliöillä. Mallinnusohjelmissa myös neliöt muodostuvat loppujen lopuksi kahdesta kolmiosta.

Usein yksinkertaisimmat maastonluontialgoritmit nojaavat ruudukkopohjaiseen geometriaan, jota muokataan erilaisten tarpeiden mukaan. Myös Unityn maastotyökalut käyttävät ruudukkopohjaista geometriaa tahkoverkossaan (kuva 10). En kuitenkaan itse pidä ruudukkopohjaista geometriaa lentävien saarien tapauksessa kovin optimaalisena, sillä saarilta vaadittujen reunojen tuottaminen monimutkaistuu.



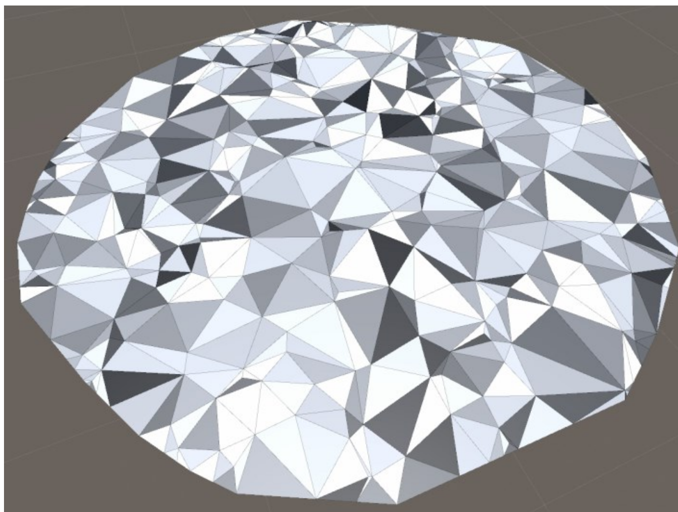
Kuva 10. Unity-pelimoottoriin sisältyy maasto-ominaisuus, jolla käyttäjä voi maalata maasto-objektiin korkeuksia ja tekstuureja.

Omassa lentäviä saaria tuottavassa algoritmissani generoin pisteitä satunnaisesti litteän ympyrän muotoon, jolloin sain ympyränmuotoista tasoa muistuttavan pistejoukon avaruudessa.

Tämän jälkeen pisteistä pystyy tuottamaan geometriaa käyttämällä Delaunay-kolmiointia (kuva 11). Delaunay-kolmiointi tekee kolmioita pisteistä niin, että se maksimoi pienimpiä kolmion kulmia eikä tee päällekkäisiä kolmion osia. Tällä vältetään pitkulaiset kolmiot ja oudot visuaaliset artefaktit. [14.]

Delaunay-kolmiointiin löytyy erilaisia valmiita ratkaisuja, joita voi olla helpompi käyttää, kuin rakentaa oma implementaatio. Itse käytin apuna kolmioinnissa Triangle.NET-kirjastoa [15]. Triangle.NETin tuottamat datarakenteet pitää kääntää

Unitylle sopiviksi listoiksi geometrian rakentamista varten, mutta prosessi ei ole kovin monimutkainen.

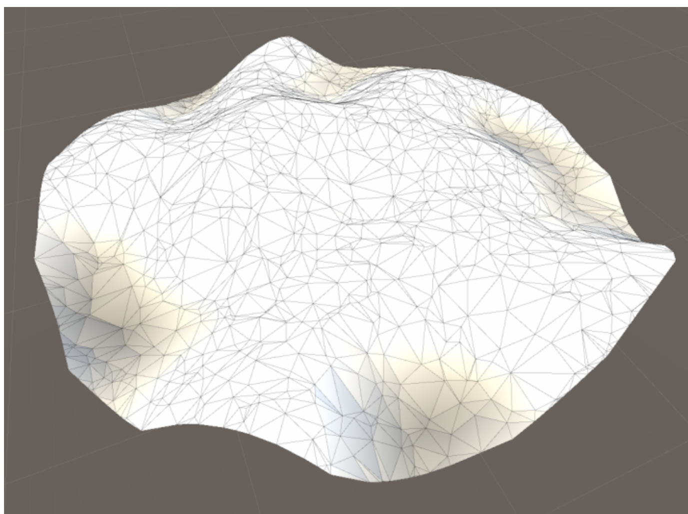


Kuva 11. Unity-pelimoottorissa on luotu geometriaa käyttäen Triangle.NET-kirjastoa kolmiointiin.

3.3 Korkeuskartat ja saarien luominen

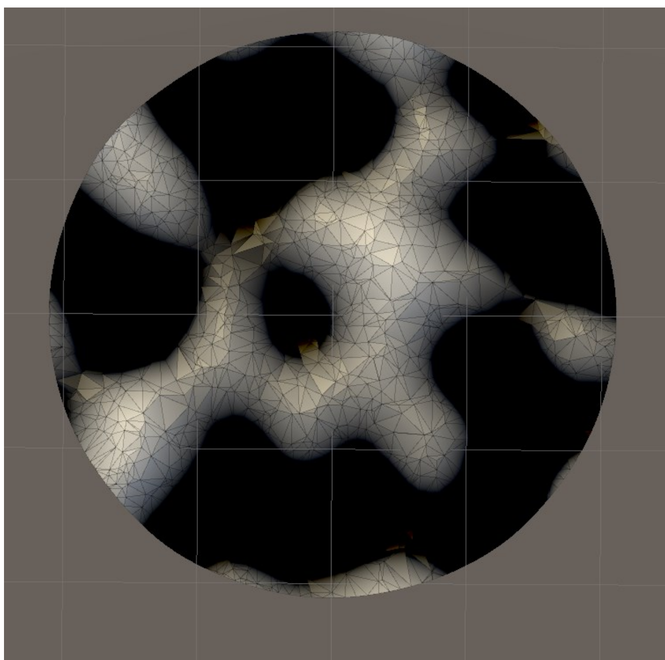
Mainitulla kohina-algoritmilla muodostetaan korkeuskartta määrittämään kunkin pisteen korkeutta (kuva 12).

Geometrian reunoille lisätään pisteitä manuaalisesti, jotta reunaverteksit on helpompi nivoa yhteen myöhemmin. Lisättyjä reunaverteksejä näkyy kuvan 12 reunojen tasaisuudessa suhteessa kuvaan 11.



Kuva 12. Kolmioituun geometriaan on lisätty kohinasta luodulla korkeuskartalla korkeusvaihtelua.

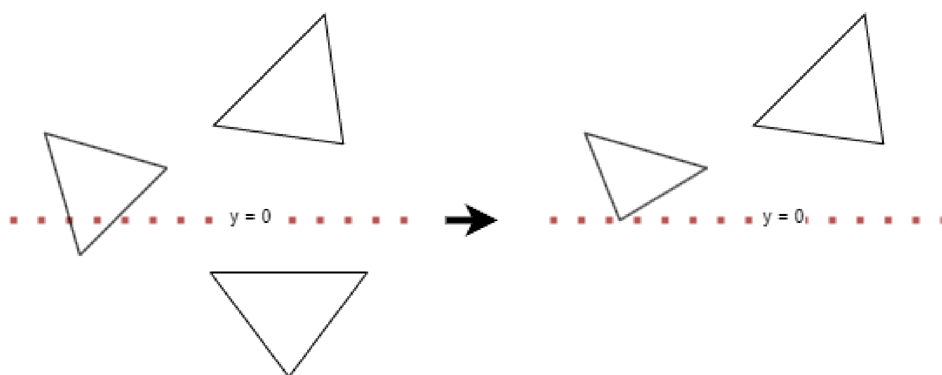
Saaren ulkomuotoa muistuttava kappale saadaan leikkaamalla kuvan 13 mustien palojen mukaisesti kolmioita pois geometriasta.



Kuva 13. Proseduraalisesti luotua geometriaa, jossa on merkitty mustalla negatiiviset korkeusarvot ja valkoisella positiiviset korkeusarvot. Mustat eli negatiiviset arvot leikataan algoritmin seuraavassa vaiheessa pois.

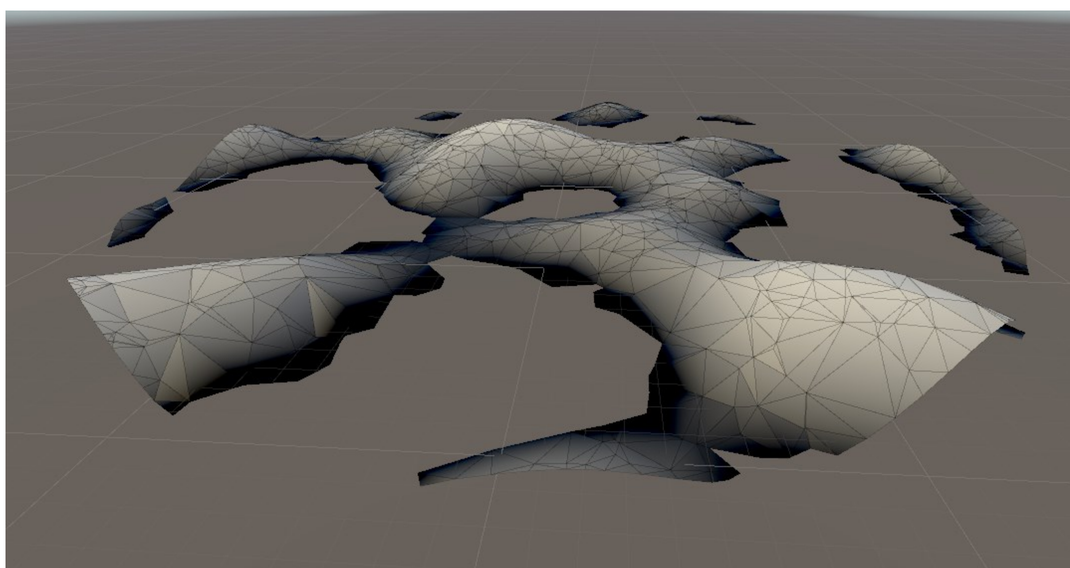
Kolmiot leikataan seuraavalla logiikalla (kuva 14):

- Jos kolmion kaikki verteksit ovat nollan alapuolella, kolmio poistetaan.
- Jos kolmiosta jakautuu verteksejä nollan ylä- ja alapuolella, nollan alapuolella olevat verteksit nostetaan nollatasoon.



Kuva 14. Kolmioiden leikkaamisprosessia kuvaava kaavio, jossa poistetaan kaikki verteksit nollan alapuolella joko siirtämällä niitä tai poistamalla kolmio. Punainen viiva edustaa korkeuden nollatasoa.

Kolmioiden leikkaamisella pyritään kuvan 15 mukaiseen geometriaan.

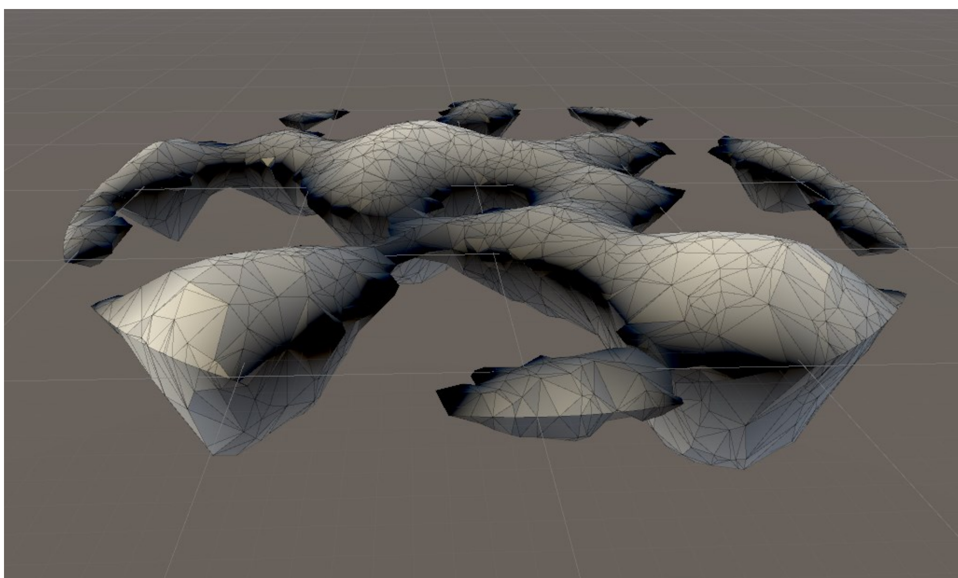


Kuva 15. Geometriasta on leikattu korkeusarvon mukaisesti osa generoidusta maastosta pois.

Kuvan 15 mukaisen geometrian luomisen jälkeen voidaan alkaa luomaan saaren pohjana toimivaa vastakappaletta. Se luodaan tekemällä yläosan pisteistä ja korkeusarvoista kopio geometrian alapuolelle.

Vastakappale liimataan kiinni geometriaan liikuttamalla saaren reunassa olevat verteksit nollatasoon, mutta myös reunan lähellä olevien verteksien korkeutta kannattaa loiventaa, jotta reuna ei ole niin jyrkkä.

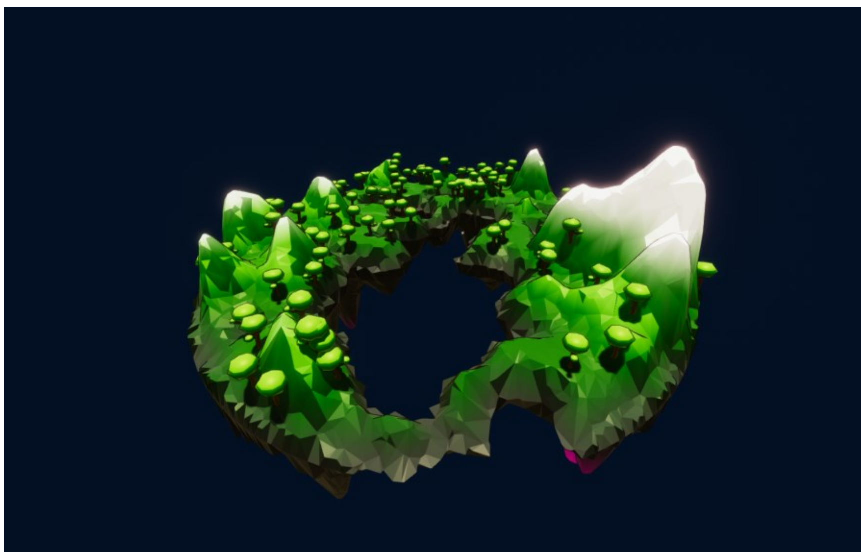
Lopuksi korkeusarvoja suurennetaan sopivalla kertoimella. Kuva 16 esittää tämän koko prosessin lopputulosta.



Kuva 16. Kahdesta kappaleesta luotu lentävän saaren geometria.

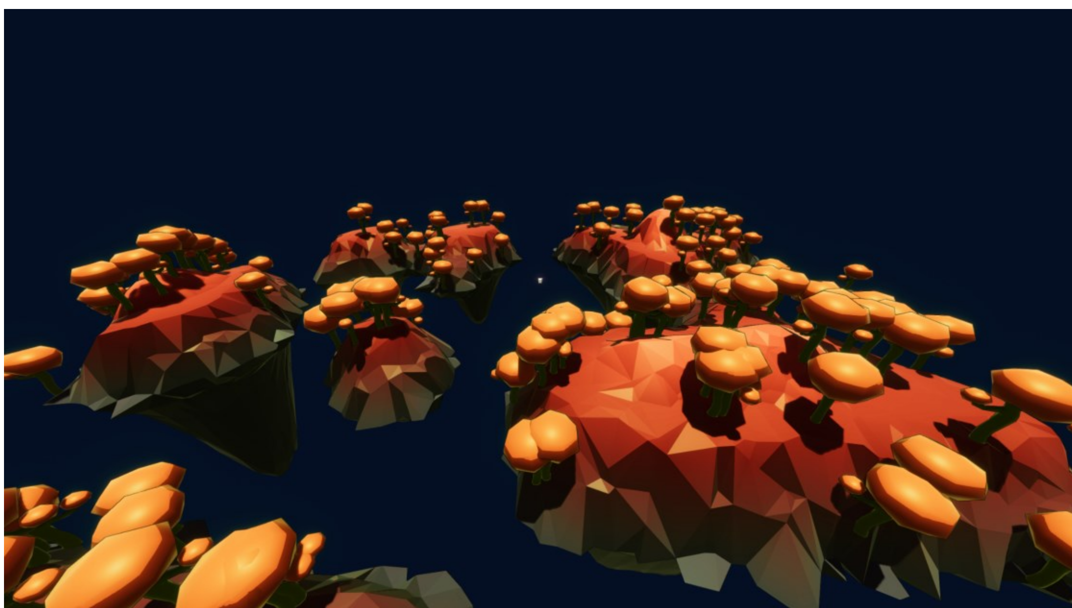
3.4 Koristelu

Kun suurin osa geometriasta on luotu, voidaan ulkonäköä hioa erilaisin keinoin. Yksi näistä keinoista on kohinan oktaavien lisääminen, jolloin saadaan aikaiseksi pientä vaihtelua isojen muotojen lisäksi. Erilaisten visuaalisten elementtien luomiseen voidaan käyttää generoinnista saatua dataa, kuten luomalla suhde värien ja korkeusarvojen tai vaihtoehtoisesti kolmioiden normaalien välille (kuva 17).



Kuva 17. Saareke, johon on lisätty värit, kerrostettua kohinaa ja päälle kasvillisuutta.

Tässä vaiheessa on myös hyvä etsiä mahdollisuuksia erilaisille teemoille ja ympäristöille. Kuvassa 18 on esimerkkinä syksyteemainen ympäristö.



Kuva 18. Teemoja ja värejä voidaan kokeilla jälkikäsittelyefekteillä, joilla voi vaihtaa väritaajuuksia toisiksi. Kuvassa vihreät värit on vaihdettu oranssiksi jälkikäsittelyefektillä.

3.5 Olemassa olevien implementaatioiden tarkastelu

Insinöörityön projektin aloitusajankohtana lokakuussa 2020 kaikkia tässä alaluvussa käsiteltäviä lähteitä ei ollut olemassa tai vaihtoehtoisesti en ollut niitä löytänyt. Olen täten muualla käytettäviä menetelmiä päässyt vertailunomaisesti käsittelemään nyt myöhempänä ajankohtana.

Useat eri proseduraalisten lentävien saarien generaattorit käyttävät hyödykseen spliniä tai muuta vastaavaa struktuuria määrittelemään saaren muotoa.

Yhdessä implementaatiotavassa jollain rajauksella käytännössä jaetaan joukko tasolla sijaitsevia pisteitä saaren geometrian käytettäväksi tai poisjätettäväksi. Pisteet voivat olla joko ruudukossa tai satunnaisesti aseteltuna, jos käytössä on kolmiointialgoritmi. Ruudukkoimplementaation tapauksessa geometriaan saatetaan lisätä epäsäännöllisyyttä luomaan luonnollisuuden tuntua ja kohinan avulla määriteltyä värinää verteksEihin kuvan 19 mukaisesti. Käytin itse omassa implementaatiossani samankaltaista kohinaan perustuvaa värinää epäsäännöllisyyden rikkomiseksi, jota voi nähdä esimerkiksi kuvassa 17 saaren reunoilla. [16; 17.]

Kuva 19. Värinän aiheuttaman epäsäännöllisyyden havainnollistamista. Oikeanpuoleisessa kuvassa ruudukolla muodostettua maastoa ja vasemmalla lisätty samaan maastoimplementaatioon värinää. [16.]

Toisessa implementaatiotavassa alkuperäiseen spliniin asetetaan pisteitä. Tämä splinistä tehty pistejoukko kopioidaan ja kopiot liikutetaan keskustaa kohti. Tämä toistetaan niin monta kertaa kuin tarvitsee, riippuen siitä, kuinka paljon algoritmilta halutaan yksityiskohtia geometriaan. Lopuksi pisteet muutetaan neliöiksi. [18.]

Kummastakaan implementaatiotavasta en ole löytänyt esimerkkiä, jossa saaren keskelle voisi tulla leikkaus, vaikka sellainen voisi mielestäni olla hyvin mahdollinen. Hyvä puoli molemmissa implementaatioissa on muodon helppo muokattavuus, jollaista oma implementaationi jäi kaipaamaan.

4 Marssikuutiot ja vokselit

Korkeuskarttoihin perustuva maasto on yksinkertaisuudessaan tehokas ratkaisu, mutta sisältää heikkouksia. Yksi tällainen heikkous on esimerkiksi luolien ja kielekkeiden generoimisen vaikeus. Vokselit tarjoavat yhden vaihtoehtoisen yleisessä käytössä olevan menetelmän tämän vaikeuden ylittämiseksi, ja juuri

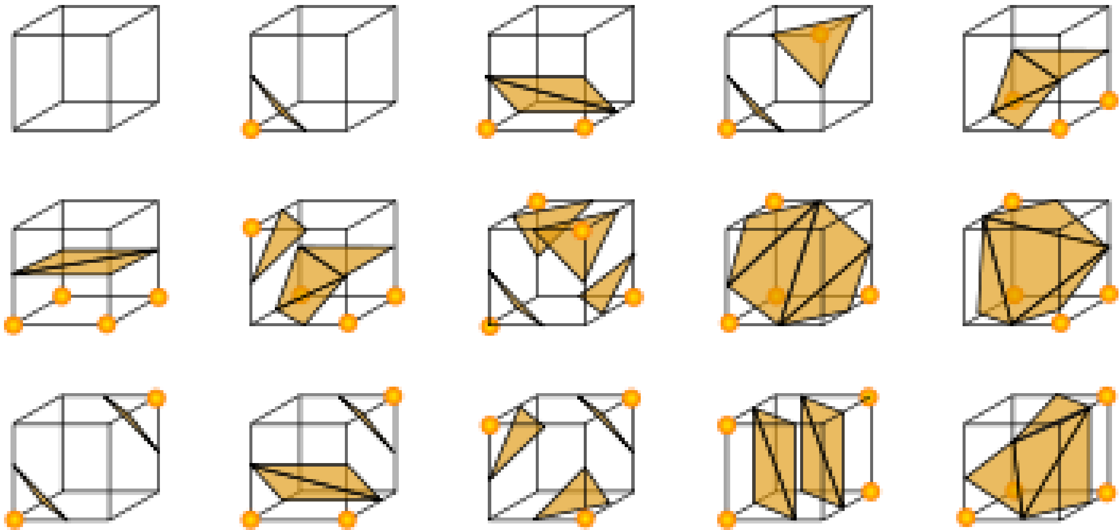
tämän takia aloin tutkimaan vokselien käyttöä, kun olin saanut korkeuskarttatoteutuksen jo pitkälle.

Vokseleita käytetään kuvaamaan erilaisia analyyttisia volyymejä, yksinkertaisesti sanottuna muotoja, 3D-koordinaatiston avulla. Koordinaatisto täytetään pisteillä valitulla taajuudella, jotka kukin pitävät sisällään dataa kuvatusta volyymista, kuten onko piste tyhjä vai täynnä. Tällaista yksittäistä dataa sisältävää pistettä kutsutaan vokseliksi.

Vokselien kuvaaman volyymin piirtämiseen on erilaisia keinoja. Minecraft-peli käyttää tunnetusti erilaisia kuutioita datan piirtämiseen. Toinen yleinen tapa on marssikuutioalgoritmi, jolla voidaan piirtää naiivi approksimaatio vokseli-koordinaatiston sisältämästä volyymista ilman, että yksittäinen vokseli erottuu. Vokselien piirtämistekniikka vaikuttaa tuloksen ulkonäköön ja mahdollisesti pelin muihin mekaniikkoihin. Päätin, että marssikuutiot sopivat hakemaani lopputulokseen paremmin.

4.1 Marssikuutiot

Marssikuutiot perustuvat 3D-koordinaatiston ”läpimarssimiseen”, jossa kunkin vokselia edustavan pisteen väliin valitaan 15:sta eri mahdollisesta konfiguraatiosta sopiva riippuen siitä, mikä vokseli on täynnä ja mikä ei (kuva 20). Vokselien itsessään voi ajatella sijaitsevan yksittäisen marssittavan kuution kulmissa. [19; 20.]

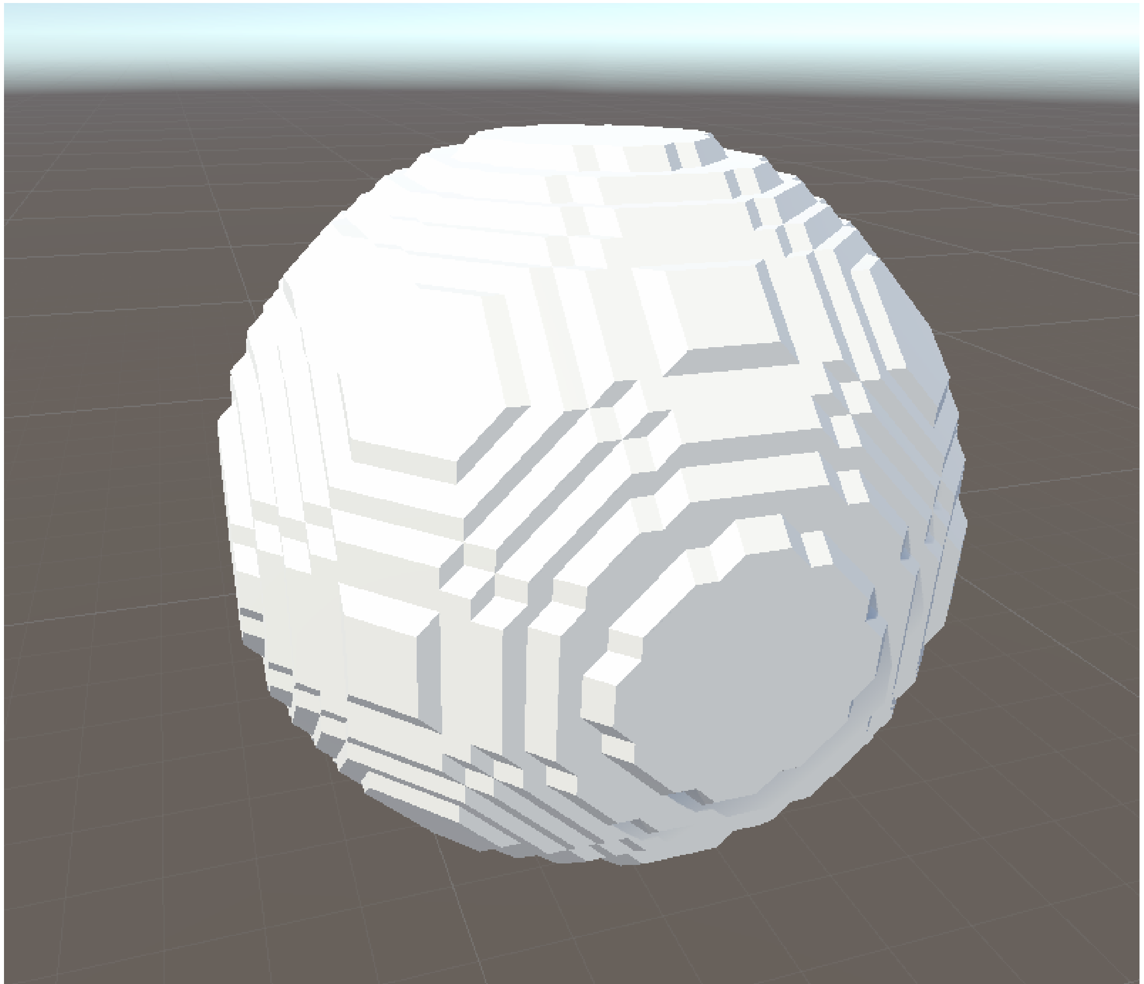


Kuva 20. Marssikuutioalgoritmissa valitaan 15 eri konfiguraatiosta sopiva vaihtoehto kulmissa sijaitsevien vokselien täyttöarvon mukaan. Kuvassa eri konfiguraatiot ovat kolmioita sisältäviä kuutioita. Jos vokseli on täynnä, se on merkitty oranssilla pisteellä. [19.]

4.2 Yksisäikeinen implementaatio

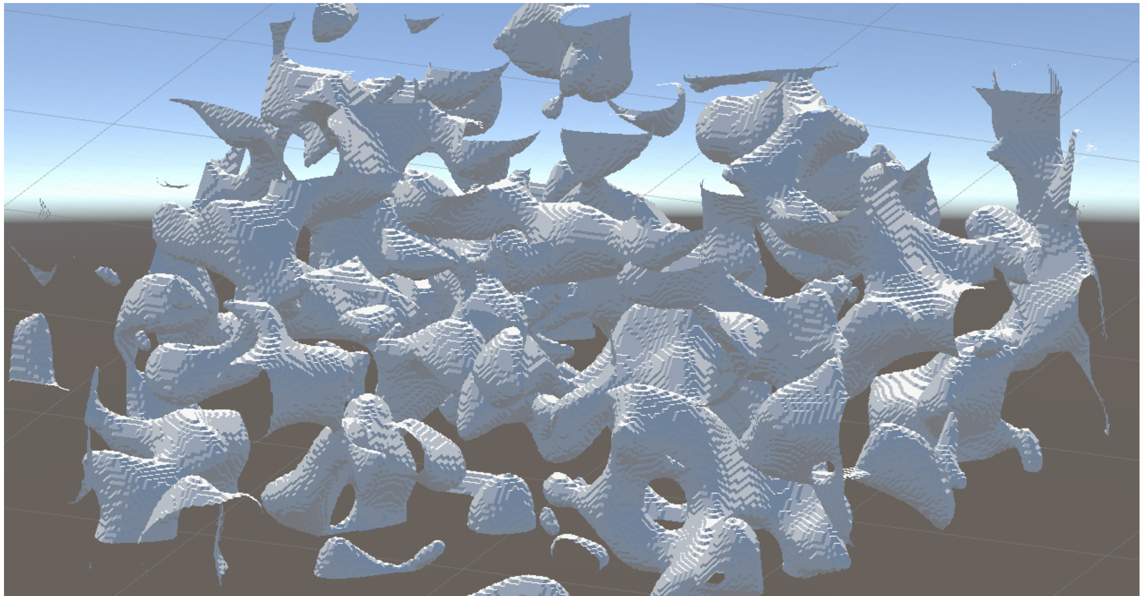
Siinä missä marssikuutioalgoritmi on kohtuullisen vakioitunut prosessi valmiiksi määriteltyjen konfiguraatioiden takia, ensimmäiseksi tehtävä vokselikartan täyttäminen on implementaation kannalta uniikki.

Marssikuutioalgoritmin oikeanlaisen toiminnallisuuden todentamiseksi voidaan täyttää kaikki vokselit joltain koordinaatin etäisyydeltä, jolloin saadaan aikaiseksi datassa pallo. Vokselikartan täyttämisen jälkeen voidaan data syöttää marssikuutioalgoritmille, jolloin saadaan vähintään etäisesti palloa muistuttava geometria (kuva 21).



Kuva 21. Marssikuutioilla piirretty pallo.

Kuten mainittu, vokselien yksi paras puoli on mielenkiintoisten rakenteiden, kuten kielekkeiden ja luolien, lisääminen. Yksinkertainen keino näiden luomiseen on kohinan implementaation laajentaminen 3D-tilaan. 3D-kohinalla (kuva 22) voidaan pohjustaa muitakin erilaisia generaattoreita, joilla generoidaan esimerkiksi koralliriuttoja tai planeettoja.

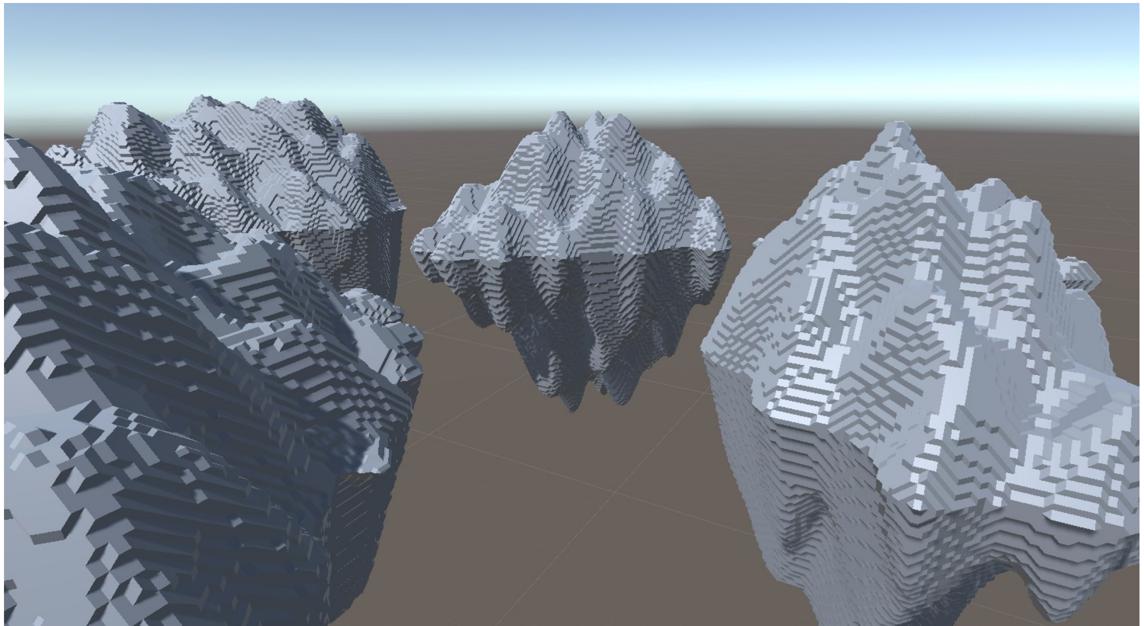


Kuva 22. Perlin-kohinaa vokselikoordinaatistossa visualisoituna marssikuutioilla.

Toinen usein esiintyvä tapa toteuttaa luolia on Perlin-mato, mutta sitä ei käsitellä tässä insinöörityössä.

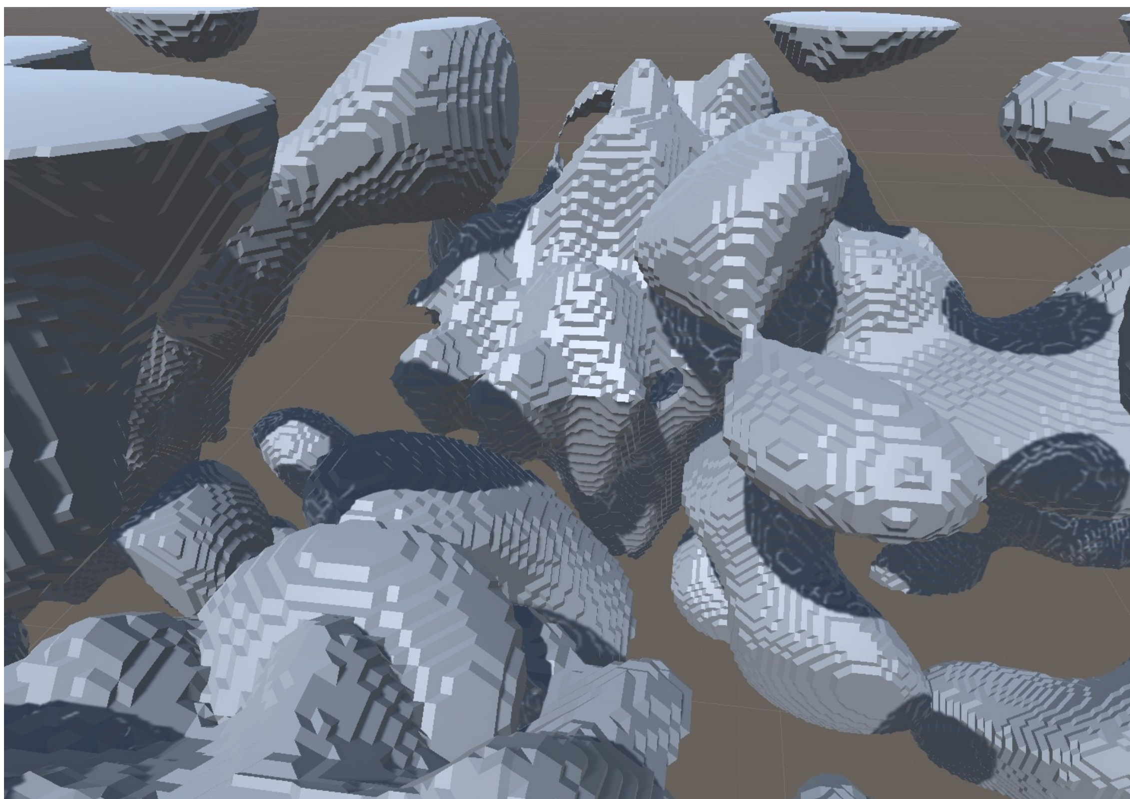
Aiemmin tehty korkeuskarttaimplementaatio on hyvä pohja rakentaa saaria myös monimutkaisemmassa 3D-koordinaatistossa. Korkeuskartalla voidaan määritellä yleinen korkeustaso xz-koordinaattia kohden. Vokseleille annetaan täyttöarvoja y-akselilla yleisen korkeustason mukaan, jolloin saarelle saadaan aikaiseksi sekä yläosa että pohja.

Esimerkiksi jos maksimikorkeus saarien yläosalle olisi 20 ja tietyssä xz-koordinaatissa saataisiin korkeusarvoksi 0,75 välillä 0–1, voitaisiin laskea, että maantaso osuisi y-koordinaatille 15, jolloin vokselit 0–15 olisivat täynnä ja vokselit 16–20 olisivat tyhjiä. Tämänkaltaisen algoritmin tulos on esitetty kuvassa 23.



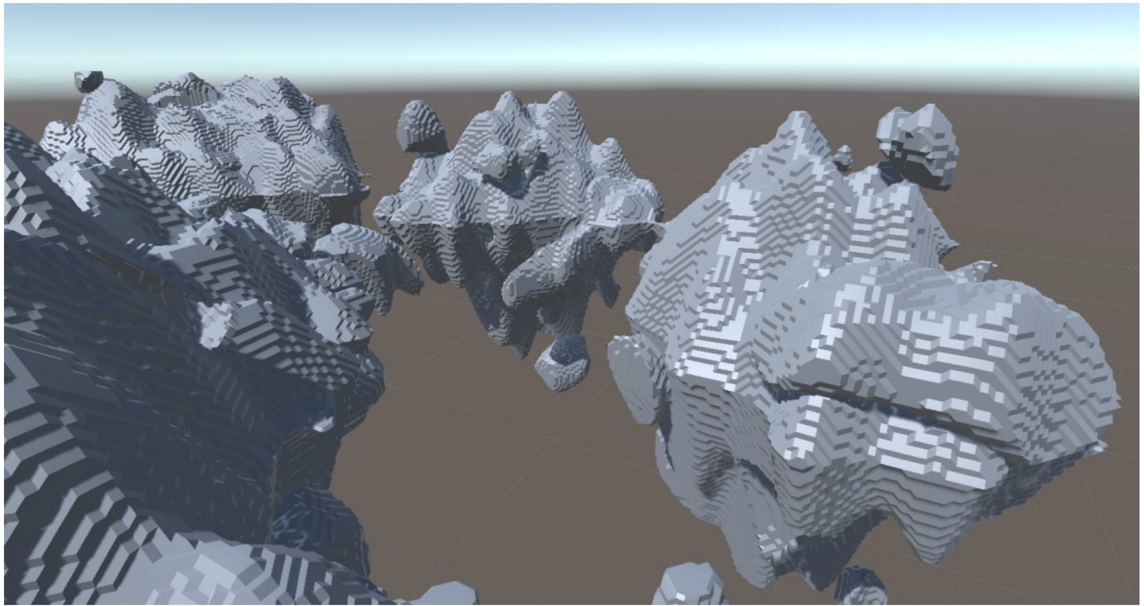
Kuva 23. Korkeuskarttapohjaisen saari-implementaation pohjalta toteutettu vokselikordinaatiston täyttö ja visualisointi.

Jos korkeuskartasta tehtyyn vokselikarttaan lisätään suoraan 3D-kohinaa, saadaan kuvan 24 mukainen sekasotkuinen geometria.



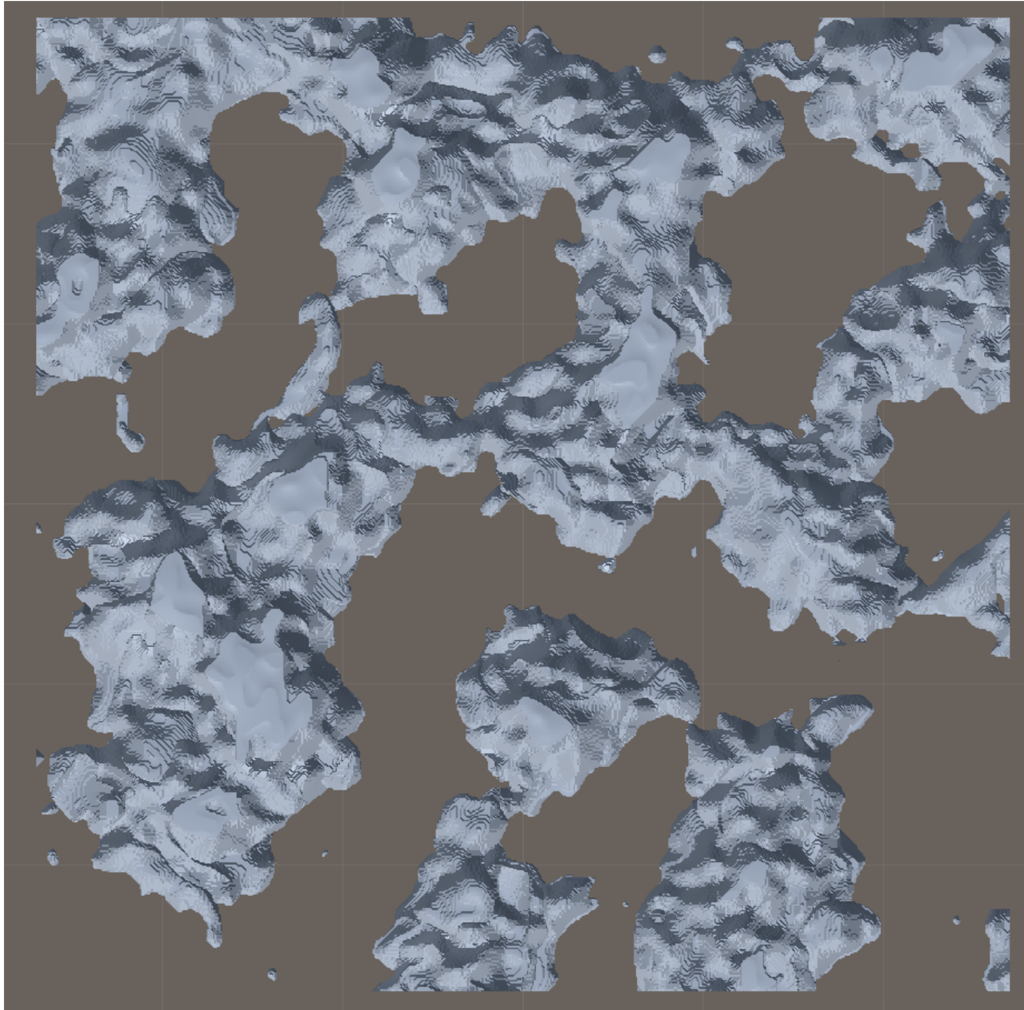
Kuva 24. 3D-kohinan sotkemaa geometriaa.

Kohinaa voi laimentaa suhteessa etäisyyteen maantasosta, jolloin saadaan pidettyä 3D-kohinan vaikutus kurissa (kuva 25). Kohinan laimennus on hyvä esimerkki tyylittelyn parametrisoinnista: esimerkiksi kuinka voimakas 3D-kohina on ilman etäisyyteen pohjaavaa laimennusta ja kuinka paljon etäisyys vaikuttaa laimentamiseen.



Kuva 25. 3D-kohinaa on laimennettu suhteessa etäisyyteen yleisestä maantasosta.

Suurin ongelma, jonka yksisäikeinen vokseli-marssikuutioimplementaatio kohtaa on algoritmin nopeus. Yksinkertaisenkin maaston generointi vie suhteellisen pitkän aikaa reaaliaikaista generointia tavoitellessa. Esimerkiksi 18 x 18 ruudukon kokoinen alue, jossa kukin ruutu sisältää 30 x 150 x 30 -kokoisen vokselikartan (kuva 26), generointiin menee 37 sekuntia vuonna 2022 hankitulla modernilla tietokoneella. Tämä vastaa noin 43 740 000 vokselia, eli vokseleita lasketaan ja piirretään noin 1,1 miljoonaa sekunnissa.



Kuva 26. 18 x 18 ruudun kokoinen kartta, jossa kukin ruutu on 30 x 150 x 30 -kokoinen.

Algoritmista voi löytyä paljon optimoitavaa, mutta suhteelliset voitot eivät välttämättä jää kovin suuriksi. Parempia tuloksia saavutetaan, kun keskitytään suuren mittakaavan muutoksiin, kuten säikeistämiseen.

4.3 Säikeistäminen ja näytönohjain

Jos proseduraalinen implementaatio ei tarvitse tietoa lähellä olevista vokseleista, vokseli-marssikuutioalgoritmi on malliesimerkki säikeistämisen hyödyistä. Säikeistämistä varten algoritmin voi jakaa useisiin toisistaan riippumattomiin

osiin; esimerkiksi yhdelle prosessorin säikeelle voi antaa tehtäväksi laskea yhden tai useamman generaattoriruudukon lohkon geometrian tai pienimillään jopa yksittäisen vokselin arvon.

Proseduraalisen implementaation säikeistäminen prosessorilla on suhteellisen yleistä, mutta vahvasti rinnakkaistettavia proseduraalisia algoritmeja voidaan laskea tehokkaammin compute shader -ohjelmilla.

Compute shaderit ovat tietokonegrafiikassa eräänlaisia varjostinohjelmia (engl. shader-program), joilla on tehokasta suorittaa erittäin rinnakkaistettavia ohjelmia näytönohjaimella. Toisin kuin muut shader-ohjelmat, jotka on suunniteltu suorittamaan renderöintiin liittyviä tehtäviä, compute shaderit keskittyvät yleiseen laskentaan ja täten niiden käyttötarkoitukset ovat varsin laajoja.

Compute shaderit toimivat ottamalla syöttötietoja ja tekemällä operaatioita näiden syöttötietojen mukaan 3D-indeksoidussa laskenta-avaruudessa. Tämä prosessi on erittäin tehokas, koska nykyaikaiset grafiikkaprosessorit sisältävät suuren määrän prosessointiytimiä, jotka kukin täyttävät yksittäisen paikan 3D-indeksoidussa laskenta-avaruudessa. Tuotettu data tallennetaan tyypillisesti puskuriin, ja tulokset voidaan tallentaa johonkin muuhun struktuuriin tai käyttää suoraan osana renderöintiliukuhihnaa.

Joitakin yleisiä compute shaderien käyttökohteita ovat

- fysiikan simulaatiot, kuten nestesimulaatiot
- kuvankäsittely
- koneoppiminen
- datankäsittely, kuten suurien listojen käsittely.

Proseduraaliset geometriaan liittyvät algoritmit ovat myös varsin sopivia näytönohjaimella laskettavaksi, koska näytönohjainten tärkeä käyttökohde, analyyttinen geometria renderöinnissä, jakaa paljon samoja ongelmia proseduraalisten algoritmien kanssa. [21; 22.]

Unity tukee HLSL-pohjaisia compute shader -ohjelmia, ja ne ovat kohtalaisen helppoja implementoida. Esimerkkikoodit 1 ja 2 esittelevät yksinkertaisen compute shader -kokonaisuuden Unityssä.

```
using UnityEngine;

public class ComputeShaderExample : MonoBehaviour
{
    public ComputeShader computeShader;

    public void Awake()
    {
        const int dataSize = 100;
        const int y = 1;
        const int z = 1;

        float[] data = new float[dataSize];

          // Create read/write buffer.
        ComputeBuffer buffer =
          new ComputeBuffer(dataSize, sizeof(float));

        // Set the buffer variable in shader.
        computeShader.SetBuffer(0, "MyRWBuffer", buffer);

        // Start the GPU calculations.
        // The compute shader uses the x value to fill our data array.
        // X needs to be the size of our array.
        computeShader.Dispatch(0, dataSize, y, z);

        // Get data from the GPU.
        buffer.GetData(data);

        // Here, the data array is filled with values from 0 to 99.

        // Manual disposal to avoid a memory leak.
        buffer.Dispose();
    }
}
```

Esimerkkikoodi 1. Unity C# -koodia, joka alustaa yksinkertaisen compute shader -ohjelman.

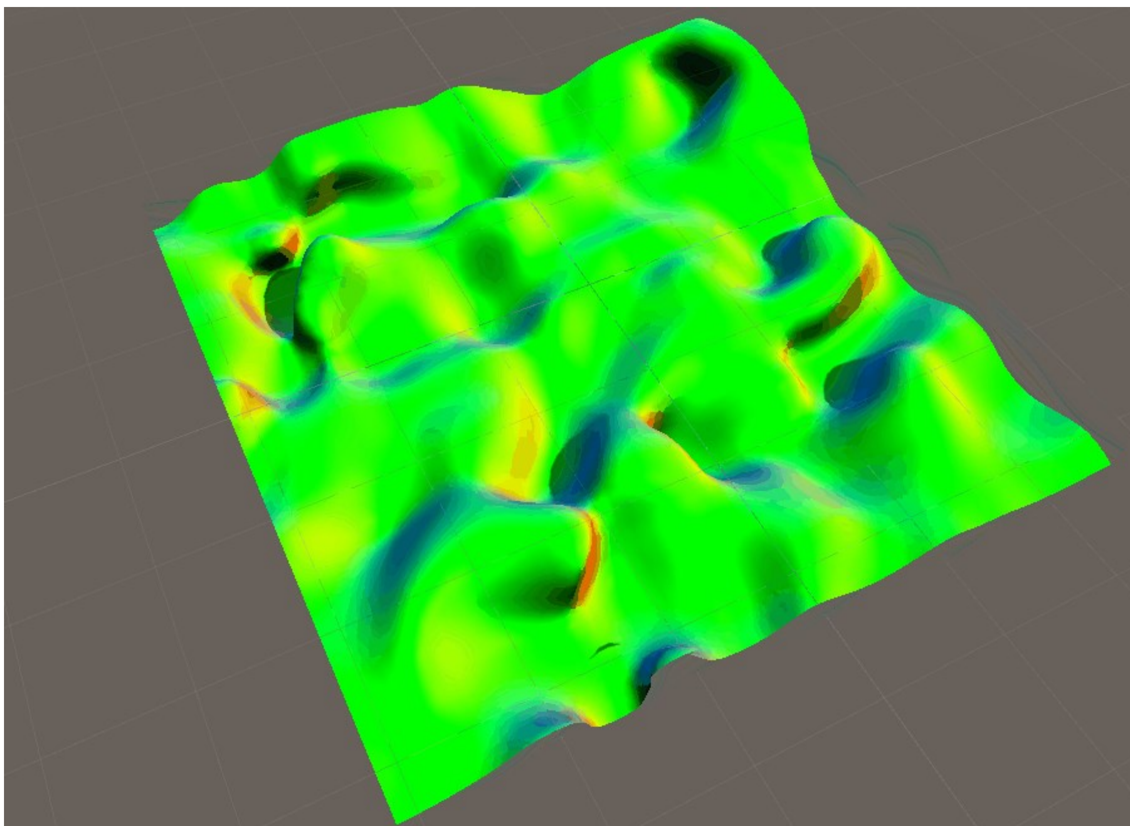
```
#pragma kernel Compute

RWStructuredBuffer<float> MyRWBuffer;

[numthreads(1,1,1)]
void Compute (uint3 id : SV_DispatchThreadID)
{
    // Example: A thread with the id of x = 42 will fill the array
    // spot of 42 with the id number, which is 42.
    MyRWBuffer[id.x] = id.x;
}
```

Esimerkkikoodi 2. Compute shader -koodia, joka täyttää liukulukuja varastoivan puskuriin alkioita id:n mukaan. Compute shadereiden id koostuu aina x-, y- ja z-arvoista.

Aikaisempi lentävien saarien marssikuutioalgoritmi on rinnastettavissa yksittäisen vokselin ja marssikuution tasolle. Tästä syystä kääntämällä koodi suoritettavaksi compute shaderilla on mahdollista saada suuria parannuksia koodin suoritusajassa grafiikkaprosessorilla, jolla on säikeitä enemmän kuin tarpeeksi.

Marssikuutioiden implementaatiossa päädyin käyttämään GitHubista löytyvää ComputeMarchingCubes-repositoriota [23]. Marssikuutio on kohtuullisen yksinkertainen ja lyhyt algoritmi, kun sen oppii ymmärtämään, joten ulkoisen koodin sisäistäminen ei ole kovin työlästä. Implementaatiossa hyvä puoli on se, että se on optimoitu valmiiksi näytönohjaimen arkkitehtuurille, joka on mielestäni compute shadereiden käytön vaikein osa. Kuvassa 27 on esimerkkinä repositorion avulla visualisoitu vokselikartta.

Yksi huomattava puoli ComputeMarchingCubes-kirjastossa on sen käyttämä Unity mesh ”advanced” API [24], johon voi suoraan syöttää näytönohjaimella olevan puskuridatan piirrettäväksi. Jos luotu geometria ei tarvitse fysiikkaa, on tämän mesh API:n käyttäminen huomattavasti nopeampaa kuin vaihtoehtoisten menetelmien.



Kuva 27. Marssikuutioilla visualisoitu vokselikartta käyttäen ComputeMarchingCubes-kirjastoa.

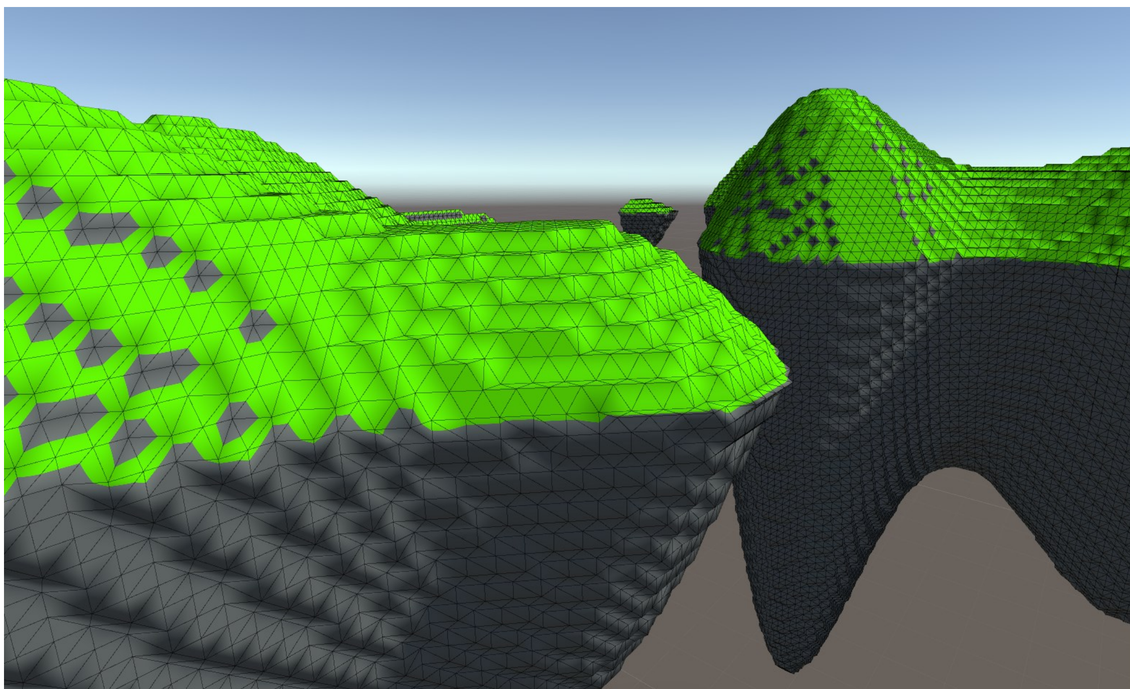
Varsinainen saarien kääntäminen compute shaderilla toimivaksi ei ole kovin monimutkainen prosessi, sillä suurin osa koodikannasta muistuttaa valmiiksi HLSL-koodia. Kohinakirjastot voivat helpottaa algoritmien siirtämistä kieleltä toiselle, sillä esimerkiksi FastNoiseLite [12] tukee sekä C#- että HLSL-implementaatioita syntaksieroilla.

Vokseliarvot pitää kirjoittaa puskuriin hajautetusti koordinaattien mukaan marssikuutiokirjaston kanssa yhtenevästi (esimerkkikoodi 3).

```
int Index = id.x + Dimensions.x * (id.y + Dimensions.y * id.z);
```

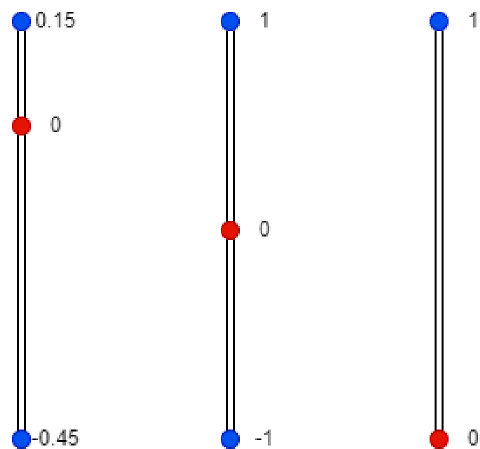
Esimerkkikoodi 3. Kaava, jolla lasketaan vokselin indeksi puskuriin.

Kun vokselit on kirjoitettu puskuriin entisen proseduraalisen algoritmin tapaan, syntyy käytännössä identtinen lopputulos. Erotuksena on mm. valotus, johon vaikuttavat eroavasti lasketut normaalit (kuva 28).

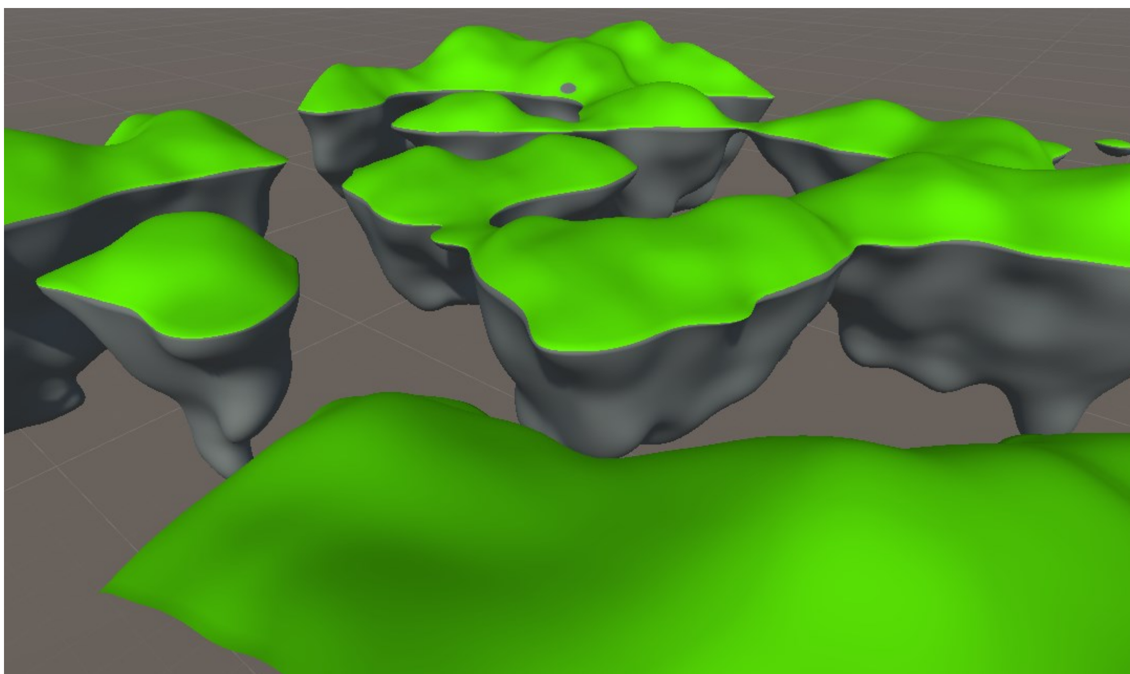


Kuva 28. Compute shaderilla laskettu saareke ilman 3D-kohinaa. Saarekkeen päällä normaaleihin perustuva väritysmalli.

Marssikuutiokirjasto käyttää täyttöarvoihin perustuvaa verteksi-interpolaatiota. Käytännössä siis vokseleiden ei tarvitse olla joko täynnä tai tyhjiä, vaan jokaisen vokselin täyttyminen määritellään liukuluvulla, esimerkiksi välillä (-1) − 1. Geometria puolestaan voidaan määritellä piirtämään jotain tiettyä arvoa, esimerkiksi arvoa 0. Kun algoritmi piirtää kolmion vokselien väliin, voidaan kolmion verteksin sijainti lineaarisesti interpoloida tarkemmin riippuen siitä, mihin sijaintiin piirtoarvo osuisi (kuvat 29 ja 30).



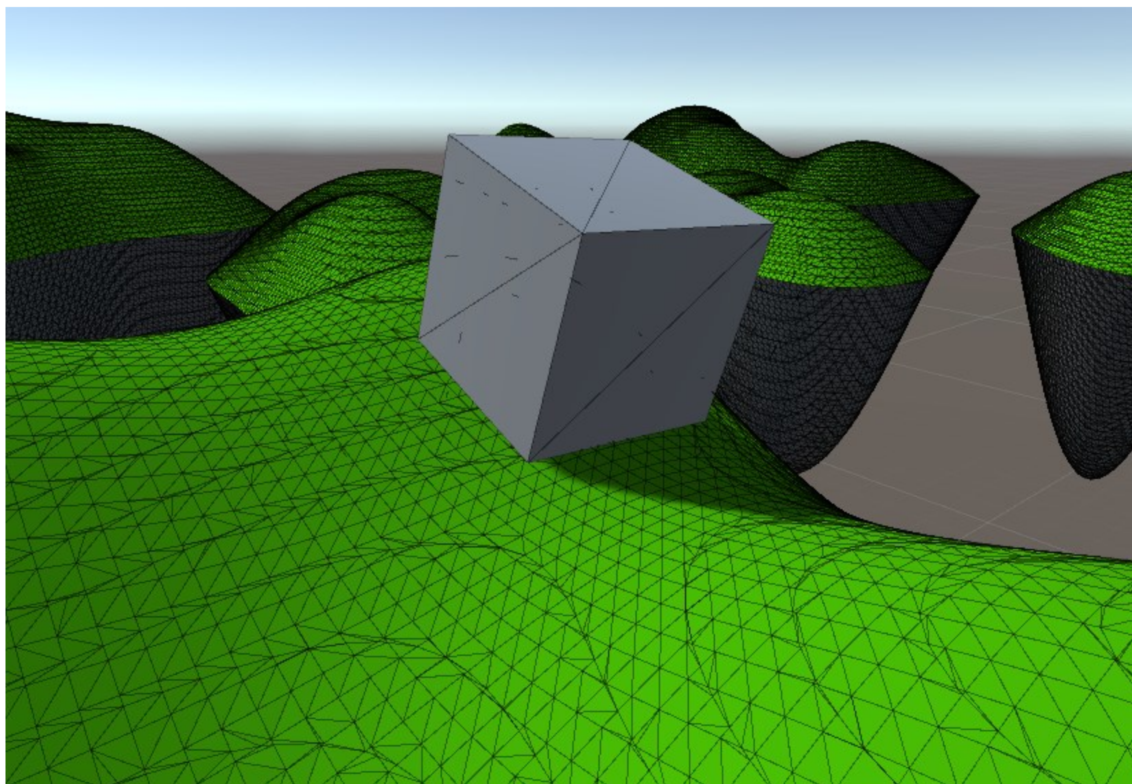
Kuva 29. Verteksin sijainnin interpolaatiota esittävä kaavio. Siniset pisteet esittävät vokselien sisältämää täyttöarvoa. Punainen piste esittää interpoloitua sijaintia haetulla piirtoarvolla 0.



Kuva 30. Interpoloimalla verteksisijainteja lineaarisesti saadaan sileämpää geometriaa, ja naiivi geometria vastaa paremmin vokselien kuvaamaa kohinavolyymia.

Tässä vaiheessa saarien luomiseen käytetty algoritmi on niin nopea, että proseduraalisen generaattorin parametrejä pystytään muokkaamaan ja havaitsemaan muutokset luoduissa saarissa reaaliaikaisesti.

Jos proseduraalisesti tuotettua geometriaa käytetään fysiikan laskemiseen (kuva 31), data pitää siirtää takaisin prosessorille. Unityn tapauksessa datasta tarvitaan verteksien sijainnit ja indeksilista, jolla merkitään kolmioiden hierarkia sijaintilistasta. Datan siirtäminen takaisin prosessorille on yleisesti ottaen määrittävä pullonkaula kokonaisuudelle. Siksi proseduraalisen algoritmin implementoija saattaa kokea tarpeellisena eriyttää tyylittelyyn tarkoitettu, ainoastaan näytönohjaimella toimiva editori ja oikeasti pelissä käytettävä prosessorille tuotu geometria toisistaan.



Kuva 31. Saarigeometrian päällä nojaa kuution muotoinen fysiikkaobjekti Unityn fysiikkamoottorin avulla.

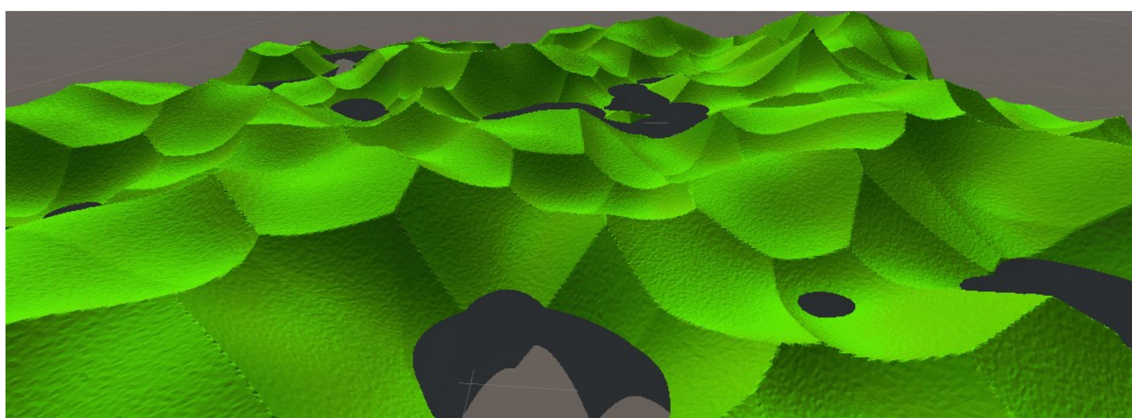
4.4 Tyylittely

Tyylittely on tärkeä osa geometriaa luovaa proseduraalista algoritmia, sillä se takaa koodilla luotujen visuaalisten elementtien samankaltaisuuden muihin pelissä esiintyviin visuaalisiin elementteihin nähden. Visuaalisen tyylisuunnan muokkaamiseen on paljon keinoja, joista osa on laskennallisesti raskaampia kuin toiset. Joskus on hyvä turvautua käyttäjän huijaamiseen laskentatehon säästämiseksi.

Esimerkiksi maastogeneroinnissa voidaan simuloida eroosiota seuraamalla yksittäisiä simulaatiopisaroita ja niiden vaikutusta vokseleihin [25]. Realistinen eroosio on hyvä keino saada realistisia yksityiskohtia, mutta ongelmaksi muodostuu simulaatioiden korkea laskenta-aika. Compute shadereillä voidaan nopeuttaa simulaatioprosesseja, mutta se on silti kohtuullisen kallista.

Eroosiosimulaation sijasta voidaan vähentää yksityiskohtia tahkoverkosta ja tehdä maastosta ”low poly” -tyylisuunnan mukaista. Sen jälkeen voidaan käyttää sopivia kohina-algoritmeja matkimaan eroosiota.

Lisäämällä Voronoin diagrammista johdettua Voronoi-kohinaa ja lisäämällä pientä satunnaisuutta maastogenerointiin saadaan vuoriston eroosiota muistuttava rakenne (kuva 32).

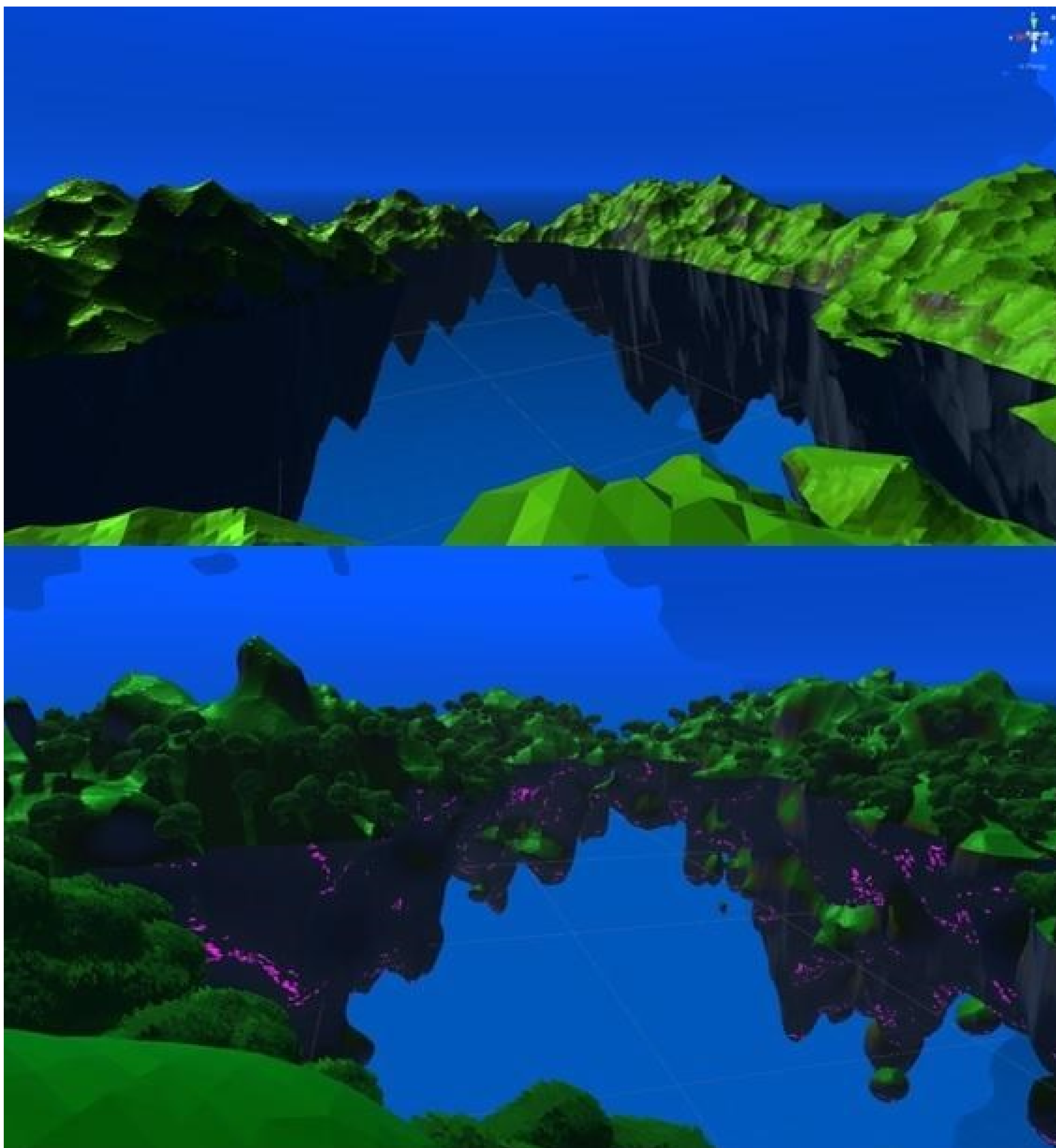


Kuva 32. Voronoi-kohinalla matkittua eroosiota. Kohinan vahvuudella voidaan saada aikaan erilaisia efektejä.

Hienosäätämällä ympäristön väriä, erilaisia parametreja ja maailman elementtejä kuten kasvillisuutta saadaan aikaiseksi varsin monipuolisia ympäristöjä (kuva 33).

Kasvillisuuden ja muiden satunnaisten visuaalisten elementtien lisäämiseen löytyy erilaisia keinoja. Jotkut algoritmit lisäävät kasvillisuuden suoraan vokselikarttaan ja jotkut lisäävät sen erillisinä objekteina maailmaan jollakin jakelumenetelmällä. Objektien asettelemiseen saatetaan käyttää algoritmeja, jotka välttävät objektien kiinnikkeisyyttä, kuten poisson disk sampling [26].

Saarialgoritmissani tein varsin yksinkertaisen ja epäelegantin ratkaisun puiden asettamiseen. Valitsen satunnaisen kolmion ja siitä satunnaisen pisteen, jonka jälkeen voin tarkistaa pisteen korkeustason ja kolmion normaalin. Näin saan implementoitua yksinkertaisen puunrajan ja varmistan, etteivät puut kasva suurilta osin maaston sisäpuolella.

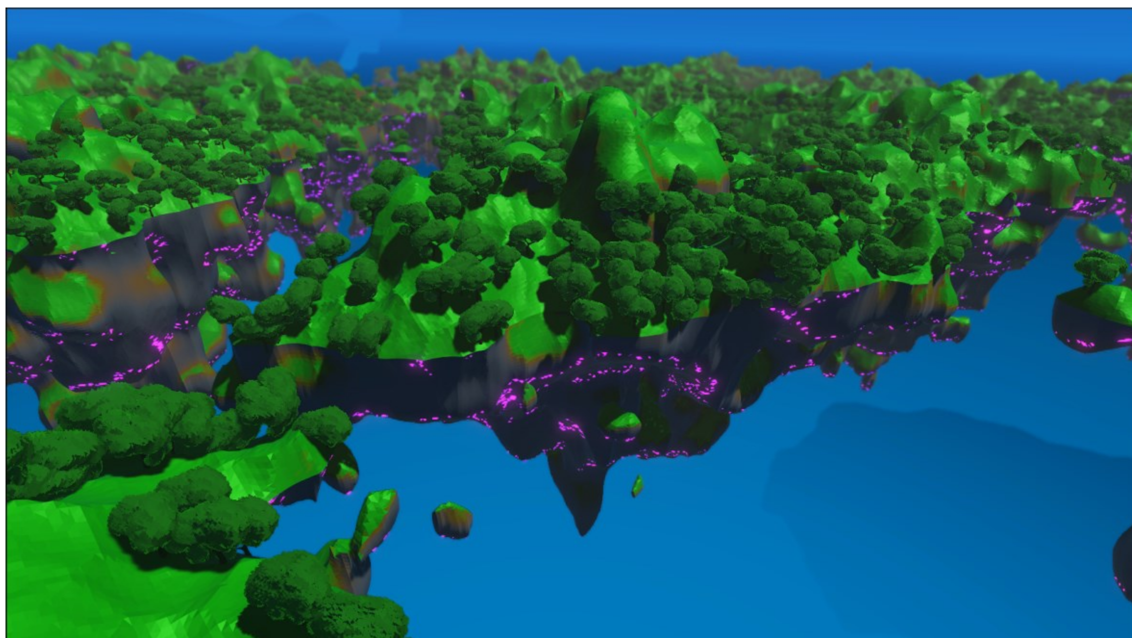


Kuva 33. Vertailu väreistä, kasvillisuuden lisäämisestä ja 3D-kohinan vahvistamisesta.

5 Tulokset ja analyysi

Olen itse tyytyväinen lentävien saarien visuaaliseen ilmeeseen, vaikka joitain asioita voi jäädä hiomaan. Esimerkiksi marssikuutioalgoritmin mahdollistamat kielekkeet ja luolat jäävät varsin maltillisiksi nykyisellä generaattorilla. Toinen hyvä visuaalinen parantelupaikka on saaren ylä- ja alaosan rajan häivyttäminen toisiinsa. Tällä hetkellä tiukka raja ja osien symmetrisyys sinistä taivasta vasten

aiheuttaa osuvasta kulmasta oudon vesimäisen heijastusefektin. Kuva 34 esittelee insinöörityön lopputulosta.



Kuva 34. Työn lopputuloksena syntyneitä proseduraalisia saaria.

Proseduraalinen maailma tarvitsisi myös lisää asioita, joita tutkia. Niitä voisivat olla esimerkiksi erilaiset rakennukset ja muuttuva kasvillisuus. Lisäämällä jonkinlainen materiaalijärjestelmä ja parempi tapa levittää kasvillisuutta generaattoriin olisi mahdollista rakentaa erilaisia biomeja. Niitä voisivat esimerkiksi olla aavikko, viidakko ja ruohikkoalueet.

Laskentatehoon toteutettiin arvio saarigeometrian laskemisesta törmäysfysiikan kanssa:

Algoritmi laskee 81 788 928 vokselin täyttämisen ja marssikuutioalgoritmin läpikäynnin 5,3 sekunnissa eli noin 15,4 miljoonaa vokselia sekunnissa. Compute shaderilla laskettu algoritmi on noin 14 kertaa nopeampi kuin yhdellä prosessorin säikeellä ajettu versio, jossa vokseleita laskettiin 1,1 miljoonaa sekunnissa. Compute shaderilla toteutettu algoritmi on kalliimpi laskea usean erilaisen kohinan ja interpolaation lisäämisen jälkeen suhteessa alkuperäiseen prosessori-

pohjaiseen implementaatioon. Kuitenkin fysiikkaa varten datan siirtäminen näytönohjaimelta prosessorille jättää näiden muutoksien vaikutuksen minimaaliseksi.

Riippuen implementaation käyttötarkoituksesta saattaa olla järkevää yrittää siirtää nykyinen algoritmi mahdollisimman asynkroniseksi. Täten tehoa voitaisiin hyödyntää reaaliajassa uusien alueiden lataamiseen paloina, sitä mukaa, kuin pelaaja niitä lähestyy tai tarvitsee.

Usein myös maastonluontialgoritmeissa voidaan ladata yksityiskohtia geometriaan kameraetäisyyden mukaan. Englanniksi tällaisen yksityiskohtaisuuden vaihtelemisen nimi on ”Level of Detail”, lyhennettynä LOD.

6 Yhteenveto

Proseduraaliset algoritmit ovat monimutkaisia tapauskohtaisesti luotavia algoritmeja, joiden tekemisessä yhdistyvät taide ja tietotekniikka. Insinöörityössä tavoiteltiin lentävien saarien proseduraalista generointia, joka työssä saavutettiin onnistuneesti ja mahdollisesti tulevaisuudessa laajennettavaksi.

Proseduraalisia saaria generoidessa joudutaan tekemään geometriaa, joka pystyy tuottamaan sekä ylä- että alaosia tahkoverkossa. Työssä käytiin läpi kaksi tapaa. Ensimmäinen oli kolmiointitekniikka, jossa liimattiin yhteen kaksi erillistä korkeuskarttojen mukaan luotua tahkoverkkoa. Tässä tekniikassa käytettiin korkeuskarttojen luomiseen kohina-algoritmia ja kolmiointiin Delaunay-kolmiointia. Toinen oli marssikuutiotekniikka, jolla algoritmisesti konfiguraatiopalasilla piirrettiin vokselien kuvaama volyymi. Analyyttisen volyymin tuottamiseen käytettiin liukuvia vokseleita, jotka täytettiin erilaisilla kohina-algoritmeilla. Viimeinen implementaatio toteutettiin compute shader -pohjaisesti, mikä vauhditti algoritmia merkittävästi.

Työssä jäi käymättä läpi implementaatioon vaikuttavia tekijöitä, kuten erilaisten laitteiden mahdolliset rajoitukset. Esimerkiksi mobiililaitteen laskentakyky ja

mahdollisuudet laskemiseen eroavat merkittävästi pelitietokoneesta. Myös implementaation käyttötarkoitus oikeassa tuotteessa vaikuttaa huomattavasti tehtäviin päätöksiin ja mahdolliseen algoritmin mittakaavaan.

Insinöörityössä kuvatut tekniikat ovat laajemmin käytettävissä myös muissa proseduraalisissa algoritmeissa, mutta se vaatii otollisia tilanteita ja vähän luovuutta.

Lähteet

1 Shan, Yuanqi. 2021. A procedural character generation system. Master's thesis. Aalto University. Aaltodoc-tietokanta.

2 Compton, Kate. Practical Procedural Generation for Everyone. Verkkoaineisto. YouTube/GDC. <https://youtu.be/WumyfLEa6bU> Luettu 1.3.2023.

3 How Two People Created Gaming's Most Complex Simulation System. Verkkoaineisto. YouTube. <https://youtu.be/1ieGQ_YddX0> Luettu 1.3.2023.

4 Houdini engine. Verkkoaineisto. SideFX. <https://www.sidefx.com/products/houdini-engine/> Luettu 14.2.2023.

5 Minecraft. 2011. Mojang Studios.

6 Deep Rock Galactic. 2020. Ghost Ship Games.

7 The Aether. Verkkoaineisto. Aether Wiki. <https://aether.fandom.com/wiki/The_Aether/AetherI> Luettu 16.2.2023.

8 Floating Island. Verkkoaineisto. Terraria wiki. <https://terraria.fandom.com/wiki/Floating_Island> Luettu 16.2.2023.

9 World Machine. Verkkoaineisto. World Machine. V<https://www.world-machine.com/> Luettu 16.2.2023.

10 Gaea. Verkkoaineisto. QuadSpinner. <https://quadspinner.com/> Luettu 2.3.2023.

11 Iho, Jussi. 2019. Proseduraaliset kohinageneraattorit. Kandidaatintyö. Tampereen yliopisto. Trepo-tietokanta.

12 FastNoiseLite repositorio. Verkkoaineisto. GitHub. <https://github.com/Auburn/FastNoiseLite> Katsottu 3.3.2023.

13 Standford bunny. 1994. Stanford University. <http://graphics.stanford.edu/data/3Dscanrep/>

14 Delaunay Triangulation. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Delaunay_triangulation> Katsottu 3.3.2023.

15 Triangle.Net repositorio. Verkkoaineisto. GitHub. https://github.com/gary-kac/triangle.net Luettu 3.3.2023.

16 Procedurally Generating Floating Islands | Devlog 1. Verkkoaineisto. YouTube. <https://youtu.be/XwzTmW4klQk> Luettu 5.3.2023.

17 Sandberg, Roland. 2013. Generation of floating islands using height maps. BTH-Blekinge Institute of Technology. DiVa-tietokanta.

18 Wandering island repositorio. Verkkoaineisto. GitHub. <https://github.com/clillianhong/wanderingisland> Luettu 5.3.2023.

19 Marching cubes. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Marching_cubes> Luettu 7.3.2023.

20 Bourke, Paul. 1994. Polygonising a scalar field. Verkkoaineisto. <http://paulbourke.net/geometry/polygonise> Luettu 7.3.2023

21 Compute shader overview. 2021. Verkkoaineisto. Microsoft. <https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader> Luettu 31.3.2023.

22 Compute shaders. Verkkoaineisto. Unity-dokumentaatio. <https://docs.unity3d.com/Manual/class-ComputeShader.html> Luettu 31.3.2023.

23 ComputeMarchingCubes repositorio. Verkkoaineisto. GitHub. https://github.com/keijiro/ComputeMarchingCubes Luettu 7.3.2023.

24 Mesh. Verkkoaineisto. Unity-dokumentaatio. <https://docs.unity3d.com/ScriptReference/Mesh.html> Luettu 26.2.2023.

25 Lague, Sebastian. Coding Adventure: Hydraulic Erosion. Verkkoaineisto. YouTube. <https://youtu.be/eaXk97ujbPQ> Luettu 7.3.2023.

26 Lague, Sebastian. [Unity.] Procedural Object Placement (E01: poisson disc sampling). Verkkoaineisto. YouTube. <https://youtu.be/7WcmyxyFO7o> Luettu 7.3.2023.