Harri Linna

Jussi Parviainen

Vokseleihin perustuvat pinnanmuodostusalgoritmit ja maaston proseduraalinen generointi

Tietotekniikan pro gradu -tutkielma

15. marraskuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijät: Harri Linna ja Jussi Parviainen

Yhteystiedot: harri.s.linna@student.jyu.fi ja
jussi.e.k.parviainen@student.jyu.fi

Ohjaaja: Sanna Mönkölä

Työn nimi: Vokseleihin perustuvat pinnanmuodostusalgoritmit ja maaston proseduraalinen generointi

Title in English: Voxel-based surface reconstruction algorithms and procedural terrain generation

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 72+1

Tiivistelmä: Tutkimuksen tarkoituksena oli verrata marssikuutiot-algoritmin ja naiivin pintaverkkoalgoritmin suorituskykyä. Tutkielman kirjallisuuskatsaus sisältää maaston proseduraalisen generoinnin menetelmiä, joilla generoitavan korkeusdatan visualisointiin vokseleihin perustuvia pinnanmuodostusalgoritmeja sovellettiin. Tutkimus toteutettiin soveltamalla suunnittelutieteellistä viitekehystä ja konstruktiivista tutkimusotetta. Tutkimuksen aineisto koostui vertailtavien pinnanmuodostusalgoritmien suorituskyvyn mittauksista, jotka perustuivat simpleksikohinalla tuotettuun keinotekoiseen dataan. Tutkimus osoitti, että naiivi pintaverkkoalgoritmi suoriutui kaikilla osa-alueilla marssikuutioita paremmin, vaikka molemmat algoritmit suoriutuivat suhteellisen tasapuolisesti. Tutkimuksen perusteella voidaan päätellä, että naiivi pintaverkkoalgoritmi on suorituskyvyltään jonkin verran tehokkaampi, joten se kannattaisi valita ensisijaisesti käytännön sovelluksiin.

Avainsanat: algoritmitutkimus, fraktaalinen kohina, marssikuutiot-algoritmi, naiivi pintaverkkoalgoritmi, proseduraalinen generointi, simpleksikohina, suunnittelutiede, tietokonegrafiikka, Unity-pelimoottori

Abstract: The purpose of the study was to compare the performance of the marching cubes and the naive surface nets algorithms. The literature review of the treatise includes methods of procedural terrain generation by which voxel-based surface reconstruction algorithms for the visualization of elevation data to be generated were applied. The study was conducted by applying a framework for design science methodology and a constructive research method. The research data consisted of measurements in the performance of reconstruction algorithms being compared, based on artificial data generated by simplex noise. The study indicated that the naive surface nets algorithm performed better than the marching cubes algorithm in all aspects though both algorithms performed relatively equally. In conclusion, the naive surface nets algorithm is somewhat more efficient in performance, thus it appears to be worth choosing primarily for practical applications.

Keywords: algorithm research, computer graphics, design science, fractal noise, marching cubes, naive surface nets, procedural generation, simplex noise, Unity game engine

Termiluettelo

Algoritmikuvaus	algoritmin kuvaus sanallisesti tai pseudokoodina.
Digitaalinen korkeusmalli	maanpinnan muotojen numeerinen esitys, engl. *digital elevation model*.
Fraktionaalinen Brownin liike	Gaussin satunnaisfunktioiden perhe, engl. *fractional Brownian motion, fBm*.
Fraktaalinen kohina	kerrostettua kohinaa, engl. *fractal noise*.
Kaksoisääriviiva-algoritmi	volymetrisen datan mallintamiseen käytettävä pinnanmuodostusalgoritmi, engl. *dual controuring*.
Marssikuutiot-algoritmi	volymetrisen datan mallintamiseen käytettävä pinnanmuodostusalgoritmi, engl. *marching cubes*.
Marssinelitahokasalgoritmi	volymetrisen datan mallintamiseen käytettävä pinnanmuodostusalgoritmi, engl. *marching tetrahedra*.
Naiivi pintaverkkoalgoritmi	volymetrisen datan mallintamiseen käytettävä pinnanmuodostusalgoritmi, engl. *naive surface nets*.
Perlin-kohina	gradientteihin perustuva kohina-algoritmi, engl. *Perlin noise*
Pintaverkkoalgoritmi	volymetrisen datan mallintamiseen käytettävä pinnanmuodostusalgoritmi, engl. *surface nets*.
Proseduraalinen generointi	algoritmillista datan tuottamista, engl. *procedural generation*.
Simpleksikohina	simpleksiruudukon toimintaan perustuva gradienttikohina, engl. *simplex noise*.
Testausohjelma	tutkielman pinnanmuodostusalgoritmien mittauksissa käytetty tietokoneohjelma.
Unity-projekti	tutkielmatyöhön sisältyvä tutkimusprojekti, joka kehitettiin Unity-pelimoottorille.
Viitetoteutus	algoritmin toteutus jollain ohjelmointikielellä, engl. *reference implementation*.
Vokseli	dataa sisältävä piste kolmiulotteisessa avaruudessa. engl. *voxel*.
Vääristymä	syötealueen häirinnällä tuotettua vääristymää arvoaluessa, engl. *domain distortion*.

Kuvat

Listaukset

Taulukot

Sisällys

1 Johdanto

Pinnanmuodostusalgoritmit ovat osa soveltavaa tietojenkäsittelytiedettä. Niillä mallinnetaan kaksi- tai kolmiulotteisia kappaleita kolmioverkkoina, jotka sitten visualisoidaan tietokonegrafiikkana. Tutkimusaiheeksi valittujen vokseleihin perustuvien pinnanmuodostusalgoritmien tehtävänä on generoida 3D-malleja kappaleen tilavuutta ilmaisevan volymetrisen datan perusteella. Vokselit ovat säännöllisen välisiä datapisteitä kolmiulotteisessa avaruudessa, ja ne edustavat käsiteltävien algoritmien tapauksessa jonkin materiaalin tilavuudellisia arvoja. Toisen tutkielma-aiheen, maaston proseduraalisen generoinnin, yhteydessä vokselit mielletään ilmana tai maa-aineksena jotakin materiaalia kuvaavan raja-arvon suhteen, jonka perusteella 3D-mallin pinta muodostetaan. Tällaisia algoritmeja hyödynnetään monilla eri tieteenaloilla. Esimerkiksi lääketieteessä ja geologiassa niitä hyödynnetään datan analysoinnissa, jossa tietokonetomografialla kerätystä volymetrisestä datasta generoidaan havainnollistava 3D-malli. Data-analyysin lisäksi tutkielman käsittelemät algoritmit soveltuvat yleisemmin 3D-grafiikan generointiin, ja niitä hyödynnetään esimerkiksi videopeleissä.

Astroneer-pelissä käytetään tunnettua Lorenssenin ja Clinen (1987) kehittämää marssikuutiot-algoritmia planeettojen esittämiseen. Pelaaja pystyy lisäämään tai poistamaan maa-ainesta, joka tapahtuu ensin muokkaamalla vokseleissa olevia volyymien arvoja ja lopuksi päivittämällä maastoa esittävät 3D-mallit (Francis 2019). Kuva 1 esittää kyseisestä pelistä tilanteen, jossa pelaaja rakentaa siltaa kuilun ylitykseen. Vokseleihin perustuvia pinnanmuodostusalgoritmeja voidaan tällä tavoin hyödyntää erilaisten rakentamiseen liittyvien mekanismien taustalla sekä ympäristön tuhoamisessa. Volyymin mallintamiseen erikoistuvat pinnanmuodostusalgoritmit soveltuvat hyvin maaston esittämiseen, koska maanpinnan muodot mukautuvat volyymien voimakkuuksiin. Algoritmien käytön etuna on, että maanpinnan korkeuden lisäksi kyetään mallintamaan maaston volymetrisiä piirteitä, kuten luolia tai kielekkeitä.

Perustoimintaperiaate volyymiä mallintavissa pinnanmuodostusalgoritmeissa on muodostaa 3D-malleja asettamalla ehdon milloin vokseli on jotain materiaalia ja milloin se lakkaa olemasta kyseistä materiaalia. Vokseleissa olevia volyymien arvoja verrataan ennalta sovittuun raja-arvoon niin, että 3D-mallin pinta rakentuu joko raja-arvon ylittävästä vokselista ulos-

Kuva 1: Maaston esittäminen Astroneer-pelissä marssikuutiot-algoritmilla.

päin tai se mielletään tyhjänä ilmana. Raja-arvo määrittää siten kappaleen ulkopinnan, jonka perusteella arvioidaan sijaitseeko vokseli muodostettavan kappaleen sisä- vai ulkopuolella. Kappaleen ulkopinta määräytyy vierekkäisten vokselien välille, joiden arvot asettuvat eri-puolin raja-arvoa. Tyypillisesti volymetrinen data voidaan tallentaa esimerkiksi kolmiulotteiseen taulukkoon, ja se käsitellään vokseleina laskemalla taulukon alkioille sijainnit kolmiulotteisessa avaruudessa niiden indeksien perusteella. Tomografian sovelluksissa kuvapino on suoraan verrannollinen kolmiulotteiseen taulukkoon, ja vokselien volyymit ovat määriteltävissä pikselien väreistä.

Tutkielma sai alkunsa marssikuutiot-algoritmin implementoinnista pelinkehitykselliseen projektiin, jossa sitä hyödynnettiin käyttäjän muokattavissa olevan maaston esittämiseen. Tämä herätti kiinnostuksen volyymin mallinnuksessa käytettäviin pinnanmuodostusalgoritmeihin, tavoitteena tutkia ja kehittää vaihtoehtoinen pinnanmuodostusalgoritmi, joka kykenee kilpailemaan marssikuutiot-algoritmin kanssa suorituskyvyssä. Aluksi tutkittiin Jun ym. (2002) kehittämää kaksoisääriviiva-algoritmia sekä Gibsonin (1998) pintaverkkoalgoritmia, jotka molemmat osoittautuivat suorituskyvyiltään marssikuutiot-algoritmia vaativammiksi. Lopulta parhaimmaksi vaihtoehdoksi löydettiin akateemisesti niukasti käsitelty Lysenkon (2012) naiivi pintaverkkoalgoritmi.

Kaksoisääriviiva-algoritmin toimintaperiaatteeseen sisältyy kvadraattisen virhefunktion ratkaiseminen, johon tarvitaan tietoa muodostettavan pinnan normaaleista (Rashid, Sultana ja Audette 2016). Sitä vastoin marssikuutiot-algoritmilla 3D-mallin pistelaskenta tapahtuu suoraviivaisesti volymetrisistä arvoista lineaarisella interpolaatiolla, joten kaksoisääriviiva-algoritmi ei soveltunut vertailututkimuksen käyttötarkoitukseen erilaisen syötejoukkonsa takia. Pinnan normaalien laskenta olisi vaatinut myös huomattavasti enemmän suoritusaikaa, joten tutkimustulokset olisivat olleet siltä osin ennakoitavissa. Tämä olisi ollut huono lähtökohta vertailututkimukselle, mikä olisi asettanut ennalta-arvattavien tutkimustulosten hyödyllisyyden kyseenalaiseksi.

Gibsonin (1998) esittämä pintaverkkoalgoritmi olisi ollut myös laskennallisesti vaativampi kuin marssikuutiot-algoritmi, koska siinä tehdään lopullinen 3D-mallin pisteiden sijoittelu iteratiivisesti käyttämällä jälkeenpäin pinnan silotteluun erikoistunutta energiafunktiota. Pintaverkkoalgoritmi ei olisi kyennyt siten kilpailemaan suorituskyvyssä marssikuutioiden kanssa, joten vaihtoehtoisen pinnanmuodostusalgoritmin etsintää jatkettiin.

Vaihtoehtoina tutkitut algoritmit kuuluvat marssikuutioille duaalisten algoritmien joukkoon, jonka ominaisuuksia Schaefer ym. (2003) kuvailevat siten, että yhtä marssikuutiot-algoritmilla generoitua kolmion kärkipistettä vastaa yksi muodostettava nelikulmio 3D-mallissa. Marssikuutioissa kolmioiden pisteet lasketaan vokseleiden muodostamien kuutioiden särmille käyttämällä ennalta määrättyä kolmiointitaulua. Kaksoisääriviiva- ja pintaverkkoalgoritmin tapauksessa pinta muodostetaan sen sijaan nelikulmioina, ja niitä muodostavat pisteet lasketaan kuutioiden sisäpuolelle. Yksi kuutio voi sisältää korkeintaan yhden pisteen, ja nelikulmioita muodostetaan pinnanmuodostuksen ehdoista riippuen särmän jakavien kuutioiden kesken. Marssikuutioille duaalisten algoritmien erottava tekijä on pisteen laskenta, ja tutkittujen algoritmien perusteella oli selvää, että pisteiden laskentatapaan vaikuttamalla marssikuutioille oli mahdollista kehittää duaalinen vaihtoehto, joka kykenee kilpailemaan sen kanssa suorituskyvyssä.

Tämän havainnon seurauksena tutkittujen algoritmien pohjalta kehitettiin pintaverkkoalgoritmista suoraviivaisempi versio, jossa iteratiivisen silottelun asemasta hyödynnettiin Schaeferin ja Warrenin (2003) mukaisesti marssikuutioiden pisteen laskentaa. Kehitetty algoritmi erosi selvästi näistä aiemmin tutkituista algoritmeista, joten etsimme löytyisikö siitä aikai-

sempaa tutkimusta tai tarkempaa nimitystä. Etsinnässä selvisi, että Lysenko (2012) oli kehittänyt vastaavan algoritmin ja nimennyt sen naiiviksi pintaverkkoalgoritmiksi. Hän oli myös tutkinut Jun ym. (2002) kaksoisääriviiva-algoritmia sekä Gibsonin (1998) pintaverkkoalgoritmia ja kehittänyt naiivin pintaverkkoalgoritmin tämän prosessin myötä. Tämä naiivi pintaverkkoalgoritmi oli jäänyt huomaamatta keskittyessämme enimmäkseen akateemisiin julkaisuihin. Kyseinen algoritmi on arvokas tutkimuskohde, koska sitä on käsitelty akateemisesti vähän ja menetelmä osoittautui suorituskykymittausten perusteella tehokkaammaksi kuin marssikuutiot-algoritmi.

Tutkielmassa noudatetaan konstruktiivista ja kokeellista algoritmitutkimusta, jonka tutkimusmenetelmänä sovelletaan Hevnerin ym. (2004) kehittämää suunnittelutieteellistä viitekehystä. Tutkielmassa konstruoidaan suunnitteluartefakti, jolla tutkielman yhteydessä tarkoitetaan naiivia pintaverkkoalgoritmia. Kehitettyä algoritmia arvioidaan kokeellisesti numeeristen testien avulla marssikuutioihin. Vertailun kohteena oleva algoritmi edustaa suunnitteluratkaisun tavoitetilaa, johon vertaamalla arvioidaan suunnitteluratkaisun soveltuvuutta, eli hoidetaan suunnitteluartefaktin evaluointi. Tutkielman yhteydessä artefaktilla tarkoitetaan kehitettyä algoritmia ja evaluoinnilla tarkoitetaan algoritmin suorituskyvyn mittauksia. Toisin sanoen tutkielma edustaa konstruktiivista ja vertailevaa tutkimusta.

Tutkielman yhteydessä kehitetyssä Unity-projektissa pinnanmuodostusalgoritmeja käytettiin pääasiallisesti maastojen mallintamiseen, joten se tarjosi mahdollisuuden tehdä katsauksen proseduraalisista menetelmistä maastoa kuvaavan korkeusdatan generointiin. Tutkielman teoriaosuudessa käsitellään kattavasti maaston proseduraalista generointia, johon esitettyjä menetelmiä voi hyödyntää pinnanmuodostusalgoritmien kanssa tai soveltaa muihin tarkoituksiin, kuten erilaisten tekstuurien generointiin. Korkeusdatan lisäksi maastolle on mahdollista generoida proseduraalisesti volymetrisiä piirteitä, mikä tarjoaa pinnanmuodostusalgoritmien rinnalle hyvän jatkotutkimuskohteen.

Shakerin ym. (2016) mukaan proseduraalinen generointi tarkoittaa algoritmillista sisällön tuottamista. Tutkielmassa sillä tarkoitetaan maaston generoinnin yhteydessä korkeusdatan algoritmillista tuottamista. Proseduraalinen generointi on merkittävä työkalu peliteollisuudessa, jota hyödynnetään esimerkiksi tutkimuksen tekemiseen innoittaneissa Astroneer- ja Minecraft-pelissä. Niissä pelaajalle tuotetaan proseduraalisilla menetelmillä valtavan kokoi-

sia maailmoja tutkittavaksi, jotka vaatisivat käsin tehtynä huomattavasti aikaa ja resursseja.

Kohinafunktiot ovat suosittuja työvälineitä proseduraalisessa maaston generoinnissa, ja ne luokitellaan yleensä arvo-, gradientti- tai solukohinaksi. Tutkielmassa keskitytään Perlin- ja simpleksikohinaan, jotka luokitellaan gradienttikohinoiksi niiden karakteristisen gradienttivektoreihin pohjautuvan toimintaperiaatteen vuoksi. Yksinkertainen kohinafunktio on hyvä esimerkki proseduraalisesta algoritmista, jolle annetaan syötteenä koordinaatti, ja algoritmi palauttaa ulostulona yhden luvun. Maaston generoinnin yhteydessä paluuarvo voidaan mieltää koordinaatin kohdalla olevaksi maanpinnan korkeudeksi. Kohinalla sellaisenaan ei saada aikaan kiinnostavaa maastoa, joten tutkielmassa perehdyttiin tarkemmin fraktaalisen kohinan muodostamiseen ja siihen, kuinka maskikohinaa hyödyntämällä saadaan sekoitettua erilaisia maastotyyppejä mahdollistamaan vaihtelevan maaston generoinnin laajemmassa mittakaavassa. Merkittävä osa tutkielman maaston proseduraalisen generoinnin teoriaosuudesta liittyykin juuri fraktaalisiin funktioihin, joita voidaan käyttää yhdessä kohinan kanssa tuottamaan erilaisia maastotyyppejä muistuttavaa korkeusdataa.

Tutkielma rakentuu siten, että luvussa 2 tarkastellaan erilaisia menetelmiä maastoa edustavan korkeusdatan algoritmilliseen tuottamiseen. Luvussa 3 käsitellään vokseleihin perustuvia pinnanmuodostusalgoritmeja. Luvussa 4 kuvataan teorialukujen menetelmiä yhteensovittavan, Unity-pelimoottorille implementoidun, ohjelmatoteutuksen kehitysvaiheita mukaan lukien sen roolia osana tutkielman tekemisessä. Luvussa 5 esitellään tutkimus, jossa vertailtiin vokseleihin perustuvien pinnanmuodostusalgoritmien suorituskykyä mittaamalla 3D-mallin generointiin kuluvaa aikaa sekä generoidun 3D-mallin kolmioiden ja pisteiden lukumääriä. Lopuksi esitetään yhteenveto luvussa 6, jossa kerrataan tutkielman tärkeimmät johtopäätökset ja pohditaan mahdollisia jatkotutkimuskohteita.

2 Maaston korkeusdatan proseduraalinen generointi

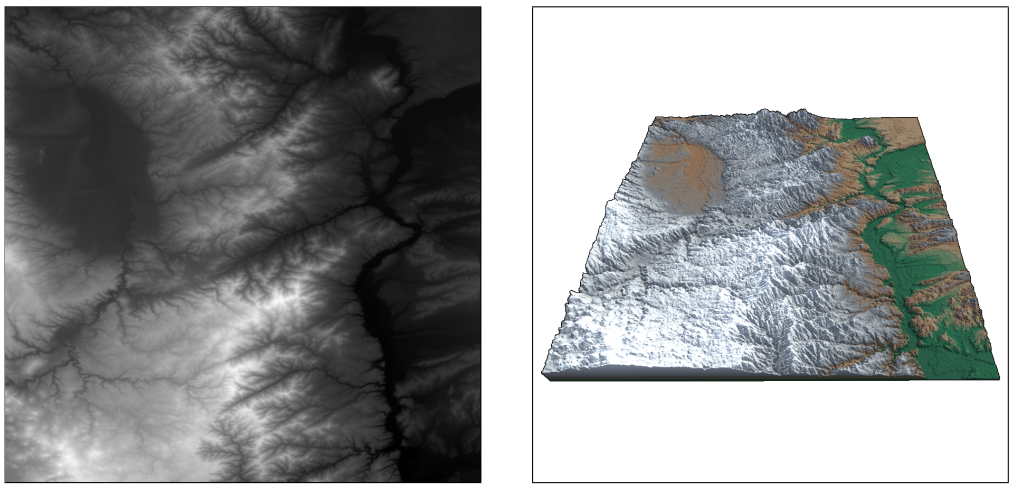
Ennen kuin proseduraalisesta korkeusdatan tuottamisesta puhutaan tarkemmin, on tärkeää kertoa mikä sen suhde on tutkielmassa käsiteltäviin pinnanmuodostusalgoritmeihin. Kaupallisissa peleissä vokseleihin perustuvia pinnanmuodostusalgoritmeja on käytetty maaston esittämiseen, joka oli teemana myös tutkielman kirjoittamisprosessin käynnistäneessä Unity-projektissa. Projektiin implementoitujen pinnanmuodostusalgoritmien yhtenä käyttötarkoituksena oli muodostaa 3D-malleja korkeusdatan pohjalta, mikä puolestaan tarjosi mahdollisuuden tehdä tutkielmaa varten kattavan katsauksen kohinaa hyödyntävistä proseduraalisista menetelmistä korkeusdatan tuottamiseen.

Tutkielmassa tutkitaan pääasiallisesti vokseleihin pohjautuvia pinnanmuodostusalgoritmeja. Luvun menetelmillä tuotettua korkeusdataa voidaan käyttää näiden pinnanmuodostusalgoritmien mallinnuksen kohteena tai johonkin muuhun käyttötarkoitukseen, kuten kuvien tuottamiseen. Tutkielmassa käsiteltävät pinnanmuodostusalgoritmit ovat erikoistuneet volumetrisen datan mallintamiseen, joten luvussa käsitellyt proseduraaliset menetelmät eivät hyödynnä niiden kykyä mallintaa korkeuden lisäksi maaston volumetrisia piirteitä, kuten luolia tai kielekkeitä. Tämä on yksi mahdollinen jatkotutkimuskohde, jos pinnanmuodostusalgoritmien sijaan tutkittaisiin tarkemmin maaston proseduraalista generointia volymetrisestä näkökulmasta.

Proseduraalinen generointi on algoritmillista datan tuottamista ja luvussa keskitytään maaston suhteen sen algoritmillisen korkeusdatan tuottamiseen, joka on numeerista. Yksinkertaisesti selitettynä proseduraalisesti korkeusdataa tuottava algoritmi ottaa syötteenä koordinaatin ja palauttaa reaaliluvun, jonka mielletään edustavan maaston korkeutta. Tyypillisesti algoritmi palauttaa lukuja tietyltä arvovälillä esimerkiksi $[-1,1]$ tai $[0,1]$, joten sen paluuarvot voidaan skaalata edustamaan realistisia maaston korkeuksia. Tällä tavoin generoitu korkeusdata tallennetaan tyypillisesti kaksiulotteiseen matriisiin, jonka pohjalta voidaan edelleen tuottaa muilla algoritmeilla esimerkiksi 3D-malleja tai kuvia.

Esimerkiksi kuvan tuottaminen tapahtuu muuntamalla maaston korkeuksia edustavan matriisin sisältämät arvot pikseleitä edustaviksi väreiksi ja värien määrittäminen tapahtuu lineaarisella interpolaatiolla alkion arvon ja tiettyä arvoväliä edustavan liukuvärin suhteen. Maastoa esittävä harmaasävykuva voidaan täten tuottaa kaksiulotteisessa matriisissa olevan korkeusdatan pohjalta yksinkertaisesti muuttamalla matriisin arvot pikseleiden väreiksi käyttäen mustasta valkoiseen vaihtuvaa liukuväriä lineaarisessa interpolaatiossa.

Kuva 2a antaa havainnollistavan esimerkin miltä Alppien alueelta otetun korkeusdatan pohjalta tuotettu harmaasävykuva näyttää. Harmaasävykuvassa tummat pikselit edustavat maastossa matalia kohtia ja vastaavasti valkoisemmat pikselit korkeampia kohtia. Kuva 2b esittää saman korkeusdatan pohjalta tuotetun 3D-mallin, joka on generoitu luvussa 3.2 käsiteltävällä naiivilla pintaverkkoalgoritmilla.



(a) Korkeuskartta. (b) Maastomalli.

Kuva 2: Korkeusdatan pohjalta muodostetut korkeuskartta ja maastomalli.

Luvussa 2.1 käsitellään pseudosatunnaista gradienttikohinaa, joka toimii lähtökohtana proseduraalisen korkeusdatan tuottamiselle. Luvussa 2.2 kartoitetaan erilaisia kohinan kerrostamisen menetelmiä, joilla saadaan tuotettua enemmän reaalimaailman maastotyyppejä muistuttavaa korkeusdataa. Luvussa 2.3 esitellään kohinan vääristämisen menetelmiä, joilla kohinatyyppien tunnusomaista piirteitä saadaan rikottua tehokkaasti siirtämällä koordinaatteja ennen niiden syöttämistä kohinafunktiolle. Luvussa 2.4 esitetään kuinka erilaisia maastotyyppejä ja maskikohinaa yhdistelemällä saadaan tuotettua monipuolisia maastomalleja.

2.1 Gradienttikohina

Kohina on tehokas työväline maaston proseduraalisessa generoinnissa, kun kaivataan yhtenäistä mutta satunnaiselta vaikuttavaa maastoa. Kohinafunktiolle annetaan generoitavan pisteen koordinaatit syötteenä, jota vastaa aina yksi ja sama paluuarvo. Kohinat erotetaan tyypillisesti kahteen luokkaan, jotka ovat arvokohina ja gradienttikohina. Menetelmissä on samankaltaisuutta, vaikka lähestymistavat eroavatkin toisistaan.

Shaker ym. (2016) kuvailevat, että arvokohinan toiminta pohjautuu tasavälisesti generoituihin satunnaisiin arvoihin, joiden välillä interpoloimalla saadaan laskettua kohinafunktiosta palautuva arvo. Gradienttikohinassa vuorostaan muodostetaan satunnaisia gradienttivektoreita, joita hyödyntämällä lasketaan lopullinen kohinafunktiosta palautuva arvo. Gradientit edustavat kohinafunktion derivaattoja, jolloin gradientti- ja etäisyysvektorin pistetulona saadaan kohinan tuottama arvo. Tutkielmassa käsitellään tarkemmin kahta gradientteihin perustuvaa Perlin- ja simpleksikohinaa, joka Archerin (2011) kokeissa osoittautui nopeammaksi näistä kahdesta algoritmista.

2.1.1 Perlin-kohina

Perlin (1985) kehitti gradientteihin perustuvan algoritmin, johon hän esitti myöhemmin merkittäviä parannuksia algoritmin kuvausta täydentävässä artikkelissa (2002b). Lisäksi Perlin (2002a) on julkaissut parannetusta algoritmistaan Java-ohjelmointikielisen viitetoteutuksen, johon tutkielmassa viitataan puhuttaessa Perlin-kohinan toteutusyksityiskohdista. Gustavson (2005) tarjoaa vaihtoehtoisen viitetoteutuksen, jossa keskitytään algoritmikuvauksen välivaiheiden esityksen selkeyteen enemmän kuin toteutuksen tehokkuuteen tarjoten erilaisen näkökulman algoritmin toteutusyksityiskohtiin.

Dustler ym. (2015) kuvailevat Perlin-kohinaa aaltopohjaisena menetelmänä, jossa n-ulotteinen pistejoukko ilmaistaan reaalilukuina käyttäen tasavälistä ja kokonaislukuihin jaettua n-ulotteista ruudukkoa. Esimerkiksi kaksiulotteisessa ruudukossa on neljä pistettä, kolmiulotteisessa kahdeksan ja yleisesti n-ulotteisessa ruudukossa pisteiden lukumäärä on yhteensä 2^n. Jokaiseen ruudukon pisteeseen on liitetty pseudosatunnainen gradienttivektori ja jokaiselle ruudukon pisteelle valitaan aina sama gradienttivektori. Erilaisia gradienttivektoreita on yhtä

monta kuin ruudukon muodostamilla nelikulmioilla on särmiä. Kaksiulotteisessa ruudukossa on neljä särmää, kolmiulotteisessa ruudukossa 12 ja n-ulotteisessa ruudukossa särmien lukumäärä on $n\,2^{n-1}$. Esimerkiksi neliulotteisessa ruudukossa särmiä olisi yhteensä $4 \cdot 2^{4-1} = 32$ kappaletta. Miksi sitten kolmiulotteisessa ruudukossa käytetään 16 gradienttivektoria eikä 12?

Perlin (2002b) selittää jakojäännöksen olevan laskennallisesti vaativa operaatio, joten gradienttien lukumäärä täytyy kasvattaa kahden potenssiin, jotta jakojäännös voidaan korvata bittitason AND-operaatiolla. Samalla negatiiviset luvut vaihtavat etumerkkiään, jolloin yhdellä binäärisellä operaatiolla vältytään yleiseltä indeksivirheeltä käytettäessä negatiivista kokonaislukua taulukkoviitteenä permutaatiotaulukkoon verrattuna tilanteeseen, jossa taulukkoviitteenä käytettäisiin ainoastaan jakojäännöstä. Lisäksi AND-operaatiossa $\wedge$ etumerkin vaihtuessa myös permutaatiotaulukon kiertosuunta vaihtuu kätevästi, esimerkiksi $-10 \wedge 255 = 246$. Perlinin (2002b) mukaan gradienttivektorien määrää kasvatettiin 16 lisäämällä neljä gradienttivektoria $(1,1,0)$, $(0,-1,1)$, $(-1,1,0)$ ja $(0,-1,-1)$ toistamiseen. Viitetoteutuksessa järjestys poikkeaa hieman Perlinin (2002a) algoritmikuvauksesta, jossa kaksi keskimmäistä gradienttia olivat toisinpäin. Algoritmikuvauksen pohjalta toteutettu menetelmä antaa silloin viitetoteutuksen kanssa saman lopputuloksen.

Permutaatiotaulukko koostuu 256 ennalta määrätystä kokonaisluvusta $\mathbb{Z}$, jotka sijoittuvat välille $[0,255]$. Permutaatiotaulukon kokorajoitteen seurauksena Perlin-kohina toistuu samanlaisena aina 256 kokonaisluvun jälkeen. Tarkkaan ottaen ohjelmatoteutuksessa permutaatiotaulukko toistetaan samanlaisena kahdesti, jolloin käytettävän hajautustaulukon p kooksi saadaan $2 \cdot 256 = 512$. Menettelyllä mahdollistetaan hajautusarvon laskeminen yhteenlaskun tuloksena koordinaateittain ilman, että summa ylittäisi missään vaiheessa hajautustaulukon rajoja. Hajautustaulukosta saadaan bittitason AND-operaatiolla hajautusarvo välille $[0,15]$, mikä toimii osoittimena gradienttivektoreihin. Hajautusarvon laskennassa tehtävien välivaiheiden ansiosta gradientit antavat vaikutelman kuvitteellisesta satunnaisuudesta.

Perlinin (2002b) mukaan gradientteja vastaavat arvot saadaan gradientti- ja etäisyysvektorien pistetulosta $g_{i,j,k} \cdot (x-i, y-j, z-k) = V_{i,j,k}$, missä $(x,y,z) \in \mathbb{R}^3$ merkitsee syötepistettä, $(i,j,k) \in \mathbb{Z}^3$ syötepisteen kokonaislukuosaa ja $v_{i,j,k} \in \mathbb{R}$ pistetuloa. Pistetulon takia kohinafunktioon sisältyy huomion arvoinen erikoistapaus, jossa kohinafunktion arvoksi määräytyy

aina nolla syötejoukon rajoittuessa kokonaislukuihin. Esimerkiksi $g_{i,j,k} \cdot (x-i, y-j, z-k) = 0$, kun $x = i$, $y = j$ ja $z = k$. Esimerkiksi kaksiulotteisten tekstuurien piirtämisessä suoraan pikselien koordinaattien sijaan kannattaa käyttää tiheämpää askellusta myös siksi, että tekstuurit alkavat toistaa itseään 256 kokonaisluvun jälkeen. Tekstuurien yhteydessä toistuvuus voi olla tietysti myös toivottu ominaisuus. Gustavson (2005) toteutti gradienttien arvojen laskennan kirjaimellisesti pistetulona, kun taas Perlin (2002a) toteutti gradientin arvomuunnoksen bittimanipulaatiolla.

Perlin-kohinassa (2002b) jokaisen syötepisteen arvo määräytyy siis painotettuna keskiarvona ympäröivien ruudukkopisteiden välille laskettavista gradientti- ja etäisyysvektorien pistetuloista. Ruudukkopisteet ovat kokonaislukuja $\mathbb{Z}$ ja niitä on 2^n kappaletta, missä n merkitsee ruudukon dimensiota. Pääakselien suuntaisten ruudukkopisteiden väliset painoarvot määräytyvät polynomin $s(x) = 6x^5 - 15x^4 + 10x^3$ arvosta suhteessa kutakin pisteparia yhdistävään pääakseliin. Quilez (2008) merkitsee näitä painoarvoja $u = s(x-i)$, $v = s(y-j)$ ja $w = s(z-k)$. Satunnaisuutta jäljittelevät kohinafunktiot ovat diskreetteinä funktioina epäjatkuvia, minkä vuoksi ne käyttävät interpolaatiota paluuarvojen yhtenäisyyden takaamiseen. Lineaarinen interpolaatio sellaisenaan aiheuttaa usein teräviä reunoja, jotka halutaan silotella piiloon korkeampiasteista interpolaatiota käyttämällä. Perlin-kohinan (2002b) arvon laskeminen vastaa siis kolmiulotteisen ruudukon tapauksessa trilineaarista interpolointia, johon Quilez (2008) on esittänyt ratkaisukaavaksi

$$n(x,y,z) = k_0 + k_1 u + k_2 v + k_3 w + k_4 uv + k_5 vw + k_6 wu + k_7 uvw, \tag{2.1}$$

missä kertoimet $k_0, \ldots, k_7$ saadaan yhtälöistä

$$\begin{aligned}
k_0 &= V_{0,0,0}, \\
k_1 &= V_{1,0,0} - V_{0,0,0}, \\
k_2 &= V_{0,1,0} - V_{0,0,0}, \\
k_3 &= V_{0,0,1} - V_{0,0,0}, \\
k_4 &= V_{0,0,0} - V_{1,0,0} - V_{0,1,0} + V_{1,1,0}, \\
k_5 &= V_{0,0,0} - V_{0,1,0} - V_{0,0,1} + V_{0,1,1}, \\
k_6 &= V_{0,0,0} - V_{1,0,0} - V_{0,0,1} + V_{1,0,1} \text{ ja} \\
k_7 &= -V_{0,0,0} + V_{1,0,0} + V_{0,1,0} - V_{1,1,0} + V_{0,0,1} - V_{1,0,1} - V_{0,1,1} + V_{1,1,1}.
\end{aligned}$$

Kaavassa (2.1) on huomion arvoista, että ratkaisua varten riittää tuntea pistetulot $V_{i,j,k}$ ja niiden painoarvot u, v ja w. Pistetulon sijasta voidaankin käyttää esimerkiksi arvokohinaa, jolloin riittäisi vaihtaa arvojen $V_{i,j,k}$ laskentaan käytettävä menetelmä toiseksi. Perlin (2002a) käyttikin viitetoteutuksessaan bittimanipulaatioon perustuvaa menetelmää, jolla saadaan laskettua samat arvot kuin gradienttivektoreita käyttämällä, mutta ilman pistetulon vaatimia kertolaskuja. Perlin (2002b) kuvailee kertolaskua laskennallisesti vaativaksi operaatioksi, joten bittimanipulaation valitsemisesta voidaan katsoa olevan ainoastaan etua. Tästä huolimatta monissa käytännön toteutuksissa suositaan pistetuloon perustuvaa menetelmää.

Quilez (2017) on määrittänyt kuinka gradienttikohinan osittaisderivaatat saadaan laskettua kaavasta (2.1). Menetelmä poikkeaa arvokohinan derivoinnista vain siinä, että ratkaisussa täytyy ottaa huomioon myös gradienttivektorit $g_{i,j,k}$, joiden avulla arvokohinan arvoja $V_{i,j,k}$ muistuttavat pistetulot laskettiin. Gradienttien lisävaatimuksen ansiosta ratkaisukaavaan tuodaan lisäinformaatiota summa-operaation muodossa, joten gradienttikohinan voidaan katsoa laajentavan arvokohinan ratkaisukaavaa tuoden siihen arvokasta lisäinformaatiota. Olkoon silottelufunktion derivaatta $s'(x) = 30x^2(x-1)^2$, jolloin Perlin-kohinan (2002b) osittaisderivaatat saadaan Quilezin (2017) esityksen mukaisesti ratkaisukaavasta

$$\begin{aligned}\frac{\partial n}{\partial x} &= d_{0.x} + d_{1.x}u + d_{2.x}v + d_{3.x}w + d_{4.x}uv + d_{5.x}vw + d_{6.x}wu + d_{7.x}uvw \\ &\quad + s'(x)\,(k_1 + k_4 v + k_6 w + k_7 vw),\end{aligned} \tag{2.2}$$

missä suuntavektorit $d_0, \ldots, d_7$ saadaan yhtälöistä

$$\begin{aligned}
d_0 &= g_{0,0,0}, \\
d_1 &= g_{1,0,0} - g_{0,0,0}, \\
d_2 &= g_{0,1,0} - g_{0,0,0}, \\
d_3 &= g_{0,0,1} - g_{0,0,0}, \\
d_4 &= g_{0,0,0} - g_{1,0,0} - g_{0,1,0} + g_{1,1,0}, \\
d_5 &= g_{0,0,0} - g_{0,1,0} - g_{0,0,1} + g_{0,1,1}, \\
d_6 &= g_{0,0,0} - g_{1,0,0} - g_{0,0,1} + g_{1,0,1} \text{ ja} \\
d_7 &= -g_{0,0,0} + g_{1,0,0} + g_{0,1,0} - g_{1,1,0} + g_{0,0,1} - g_{1,0,1} - g_{0,1,1} + g_{1,1,1}.
\end{aligned}$$

Maaston proseduraalisessa generoinnissa Perlin-kohinan (2002b) kuviointiin pystytään vaikuttamaan permutaatiotaulukkoa sekoittamalla. Sekoittamiseen voidaan hyödyntää esimerkiksi siemenlukua käyttävää pseudosatunnaisfunktiota, joka mahdollistaa hyväksi havaittujen kohinan kuviointien toisintamisen. Permutaatiotaulukon sekoittaminen on hyödyllistä silloin, kun videopeleissä halutaan luoda esimerkiksi useita erilaisia maailmoja, jotka muistuttavat pääpiirteiltään toisiaan. Siemenen puuttuminen vain tarkoittaa, että kaikki pelaajat saavat täsmälleen saman proseduraalisesti generoidun maaston. Käytännön toteutuksissa siemenluvun lisäämisen yhteydessä kohinan staattiset attribuutit kannattaa muokata oliokohtaisiksi, jotta niitä varioimalla voidaan luoda erilaisia maastotyyppejä käyttämällä eri siemenlukua tai muita parametrisoitavia muuttujia. Erilaisia maastotyyppejä ja niiden parametreja käsitellään tarkemmin luvussa 2.2.

2.1.2 Simpleksikohina

Perlinin (2001) kehittämää simpleksikohinaa verrataan usein Perlin-kohinaan, sillä molemmat menetelmät luokitellaan gradienttikohinaksi. Archer (2011) on vertaillut näiden kahden algoritmi ohjelmatoteutuksia, joten tutkielmassa keskitytään simpleksikohinan algoritmikuvaukseen, jonka ohjelmatoteutuksen yksityiskohtia perustellaan viitetoteutuksista poimituilla esimerkeillä. Tutkielmassa viitataan ensisijaisesti Perlinin (2001) Java-kieliseen viitetoteutukseen puhuttaessa yleisesti simpleksikohinan toteutusyksityiskohdista. Gustavsonin (2008) C-kieliseen ohjelmakoodiin viitataan erityisesti analyyttisten derivaattojen yhteydessä, sillä hän on esittänyt simpleksikohinalle helppolukuisemman toteutustavan Perlinin (2001) algoritmikuvausta täydentävässä tutkimusraportissaan.

Gustavson (2005) hyödyntää pseudosatunnaisten gradienttivektorien laskentaan samaa hajautustaulukkoa sekä gradientti- ja etäisyysvektorin pistetuloon perustuvaa taktiikkaa kuin Perlin-kohinassa. Menetelmä korostaa pistetulon merkitystä gradienttikohinassa, mutta ei ole sellaisenaan täysin Perlinin (2001) esittämän viitetoteutuksen tai algoritmikuvauksen mukainen. Perlin (2001) on keskittynyt algoritmikuvauksessaan entistä enemmän kuvaamaan viitetoteutuksessa käyttämänsä bittimanipulaation toimintaperiaatteen kuvaamiseen. Viitetoteutuksessa pseudosatunnaisesti sekoitetusta kokonaisluvusta h poimitaan kuusi vähiten merkitsevää bittiä, joista kolme ensimmäistä määrittää summattavat komponentit ja niiden

rotaation (x, y tai z) sekä kolme muuta niiden mahdolliset negaatiot ($-x$, $-y$ tai $-z$). Vertailuoperaatiot vastaavat siis gradientin pistetuloa komponentein 1, 0 tai -1. Toimintamalli ei ollut enää yhtäpitävä Perlinin (2002b) gradienttien kanssa, sillä bittimanipulaatio tuottaa aiempaan menetelmään nähden nelinkertaisen määrän $2^6 = 64$ gradienttivektoreita, jotka on esitetty listauksessa 6 käyttäen C#-ohjelmointikielen standardikirjastoa *System.Numerics*.

Gustavsonin (2008) ohjelmatoteutuksessa lasketaan simpleksikohinan arvon lisäksi sen analyyttiset derivaatat, jossa tarvitaan tietoa gradienttivektorien suuntakomponenteista. Niitä ei ollut suoraan saatavilla algoritmikuvauksesta, joten viitetoteutusta vastaavat gradienttivektorit taulukoitiin ohjelmallisesti hyödyntäen edellä kuvattua toimintamallia. Simpleksikohinan viitetoteutusta kutsuttiin 64 gradientille kunkin komponentin suuntaisesti sitä vastaavalla yksikkövektorilla, josta saatiin yksikkövektoria vastaava koordinaatti. Esimerkiksi listauksen 6 rivillä 4 gradientti $(1,-1,0) = 1 \cdot (1,0,0) + (-1) \cdot (0,1,0) + 0 \cdot (0,0,1)$. Pistetulot määräytyvät siten suoraan yksinkertaisista laskusäännöistä, jolloin äskeisen esimerkin tapauksessa voidaan päätellä $x - y = (1,-1,0) \cdot (x,y,z)$. Menetelmä todennettiin luotettavaksi kokeilemalla sitä ensin Perlin-kohinaan (2002b), jonka gradienttivektorit tiedettiin ennalta. Tämän jälkeen varmistettiin vielä yksikkötesteillä, että pistetulon ja bittimanipulaation ohjelmatoteutukset tuottavat samalla syötteellä yhtäpitävät paluuarvot, minkä perusteella menetelmät olivat keskenään vaihdettavissa ainakin sen hetkisellä testijoukolla.

Simpleksikohinan tehokkaampi suorituskyky liittyy ennen kaikkea algoritmin aikavaativuuteen, sillä Perlinin (2001) mukaan simpleksiruudukon ansiosta aikavaativuus onnistuttiin vähentämään aiemmasta eksponentiaalisesta $O(n\,2^n)$ kertalukua pienempään polynomiseen $O(n^2)$ aikavaativuuteen. Perlinin (2001) algoritmikuvauksen pohjalta n-ulotteista simpleksiruudukkoa edustava hyperkuutio jakautuu $n!$ määrään simpleksejä, joiden särmät määrittävät kulkemisjärjestyksen hyperkuution pienimmästä kärkipisteestä $(0,0...0)$ suurimpaan $(1,1...1)$ kulkien kaikkien simpleksin kärkipisteiden kautta. Simpleksin valinta tapahtuu järjestämällä hyperkuution pienimmän kärkipisteen (i,j,k) ja syötepisteen (x,y,z) koordinaattien väliset etäisyydet $(u,y,w) = (x-i, y-j, z-k)$. Järjestys määrittää simpleksiruudukon indeksit, jotka muodostavat kyseisen simpleksin, joten järjestyksessä summattavat siirtymävektorit voidaan mieltää janoina simpleksin kärkipisteiden välillä. Esimerkiksi jos lähtöpisteeksi saadaan $u \geq v \geq w$, niin simpleksiruudukon neljä indeksiä eli kärkipistettä kuljetaan

kantavektoreita vastaavassa järjestyksessä x, y ja z.

Koordinaattien muuntaminen kuutiollisen ja simpleksiruudukon välillä lasketaan vinouskertoimen F avulla, jonka valintaan koordinaatiston dimensiot vaikuttavat $F = \frac{\sqrt{n+1}-1}{n}$. Kohinan arvon laskeminen yksinkertaistuu merkittävästi käytettäessä simpleksiruudukkoa, sillä kärkipisteitä on vain $n+1$ kappaletta. Lisäksi kunkin kärkipisteen painoarvo eli vaikutus lasketaan toisistaan riippumattomasti, kun käytetään ytimenä symmetristä yhtälöä $t = 0.6 - (u^2 + v^2 + w^2)$, missä u, v, w merkitsevät etäisyyttä simpleksin kärkipisteen ja syötepisteen välillä (Perlin 2001). Yhden gradienttivektorin vaikutus kohinan arvoon saadaan gradienttivektorin ja etäisyysvektorin pistetulosta sekä kärkipisteen painoarvosta. Kuva 3a esittää simpleksikohinaa, joka lasketaan kaavalla

$$h(x,y,z) = \sum_{i=1}^{4} 8 \max(0, t^4) \left(u_i g_{i.x} + v_i g_{i.y} + w_i g_{i.z}\right). \tag{2.3}$$

Perlin (2001) suosittelee analyyttisen derivaatan laskemista kohinan yhteydessä etenkin normaaleihin perustuvan häiriön tai muiden derivaattaa hyödyntävien efektien tapauksessa. Quilez (2008) on käyttänyt derivaattoja esimerkiksi eroosion mallintamiseen, johon palataan vielä luvussa 2.2. Vaihtoehtoisesti derivaattaa voidaan approksimoida erotusosamääränä, ellei analyyttistä lauseketta tunneta, kuten Perlinin (2001) mukaan monesti tehtiinkin varhaisemman kohinan yhteydessä. Summalausekkeen ansiosta simpleksikohinan analyyttiset derivaatat ovat melko suoraviivaisesti ja kohtuullisella vaivalla johdettavissa derivoinnin laskusäännöistä. Simpleksikohinan derivaatta muuttujan x suhteen määritetään seuraavasti:

$$\frac{\partial h}{\partial x} = \sum_{i=1}^{4} -64 \max(0, t^3)\, u_i \left(u_i g_{i.x} + v_i g_{i.y} + w_i g_{i.z}\right) + 8 \max(0, t^4)\, g_{i.x}. \tag{2.4}$$

Ohjelmatoteutuksessa derivaattojen oikeellisuus voidaan testata varmistamalla, että simpleksiruudukon kärkipisteissä sijaitsee kohinafunktion nollakohta ja siinä osittaisderivaatat ovat samat kuin kyseisen pisteen gradienttivektori. Simpleksiruudukon sisäpuolella vastaavasti voidaan verrata ohjelmatoteutusten yhtäpitävyyttä kohinan arvojen ja derivaattojen osalta. Menetelmällä testattiin esimerkiksi Perlinin (2001) algoritmikuvauksen ja viitetoteutuksen sekä Gustavsonin (2005) ohjelmatoteutuksen yhtäpitävyyttä, kun käytettiin listauksen 6 gradientteja. Algoritmien kuvauksissa ja toteutuksissa esiintyi muutamia eroja yksityiskohdissa, jotka yhtenäistämällä saatiin menetelmät muokattua yhtäpitäviksi. Gustavson (2008) oli

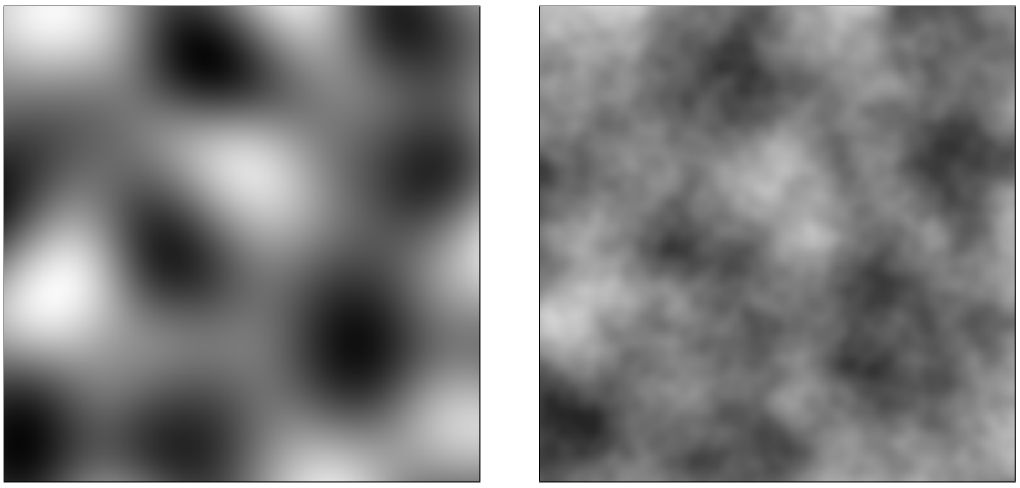
implementoinut analyyttiset derivaatat ohjelmatoteutukseensa, jota soveltaen muodostettiin kaava (2.4).

2.2 Fraktaalinen kohina

Kohinaan saadaan tuotettua fraktaalisia piirteitä summaamalla eri taajuuksista kohinaa. Menetelmä tunnetaan nimellä fraktaalinen kohina, mutta Dustlerin ym. (2015) sekä Galinin ym. (2019) mukaan siitä käytetään toisinaan nimitystä fraktionaalinen Brownin liike (fBm). Perlin (1985) käytti vastaavaa menetelmää ja kuvaili sen muistuttavan visuaalisesti Brownin liikkeettä. Tutkielman yhteydessä fraktaalisella kohinalla viitataan juuri fBm-funktion avulla kerrostettuun kohinaan, josta esitellään maaston korkeusdatan generointiin hyvin soveltuvia toteutustapoja. FBm-funktion osalta tutkielman viitetoteutuksena toimii Perlinin (1985) turbulenssifunktio, joka julkaistiin Perlin-kohinan (1985) liitemateriaalina. Muita ohjelmatoteutuksia ovat kuvanneet esimerkiksi Vivo (2015) ja Quilez (2019), mitkä soveltuvat runsaan parametrisoinnin ansiosta hyvin maaston korkeusdatan generointiin. Matemaattisesti fBm-funktio voidaan esittää seuraavasti:

$$k(x,y) = \sum_{i=0}^{n-1} ap^i h(fl^i x, fl^i y). \tag{2.5}$$

Kaavassa (2.5) summattavien kohinakerroksien lukumäärästä käytetään nimitystä oktaavit n. Taajuus määräytyy oktaaveittain i laskettavien lakunaarisuuden l ja frekvenssin f tulosta, joka vuorostaan vaikuttaa kohinafunktiolle h syötettävän pisteen (x,y) sijaintiin. Kohinan paluuarvoa kerrotaan voimakkuudella, joka määräytyy amplitudin a ja pysyvyyden p tulosta vastaavasti kuin taajuuden kohdalla. Frekvenssi ja amplitudi käyttäytyvät siten taajuuden ja voimakkuuden alkuarvoina ensimmäisellä iteraatiolla. Potenssimerkintä i kertoo kuinka monta kertaa alkuarvoja on kerrottu lakunaarisuudella ja pysyvyydellä eli montako oktaavia sisältyy taajuuteen. Kuvassa 3b fraktaalisia piirteitä on synnytetty käyttäen parametreja $l = 2$ ja $p = 2^{-1}$. Kyseisillä parametreilla taajuus kaksinkertaistuu ja voimakkuus puolittuu jokaisella oktaavin kierroksella. Fraktaalisessa kohinassa voimakkuus on verrannollinen taajuuteen, mikä Perlinin (1985) mukaan antaa vaikutelman Brownin liikkeestä.



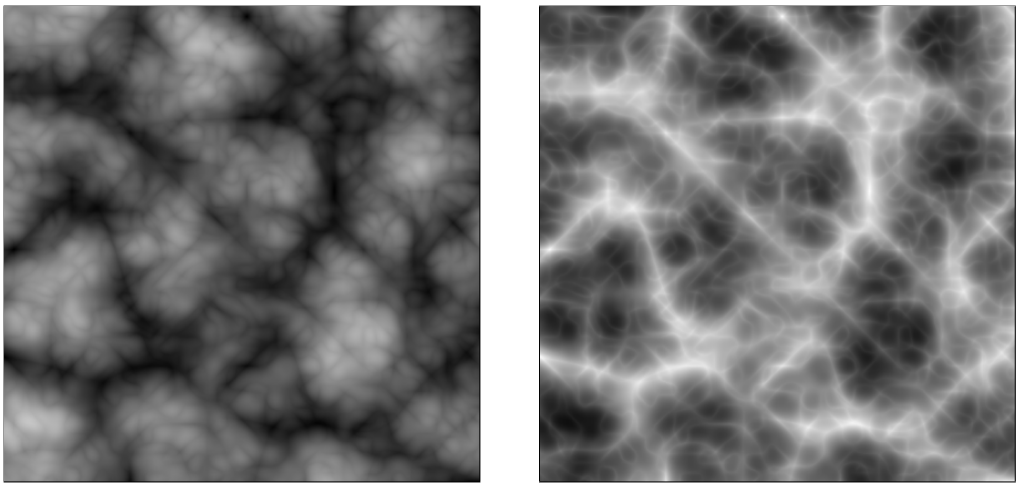
(a) Tavallista simpleksikohinaa. (b) Kerrostettua simpleksikohinaa.

Kuva 3: Fraktaalisia piirteitä lisätään yhdistämällä eritaajuista kohinaa.

Summaamalla kohinaa eri tavoilla voidaan tuottaa erilaisia maastotyyppejä. Esimerkiksi ottamalla itseisarvon kohinafunktion palauttamasta arvosta $g(x) = |x|$ saadaan epäjatkuvaa aaltoilevaa kohinaa, jota Perlin (1985) kutsui turbulenssiksi. Itseisarvon komplementtia $g(x) = (1 - |x|)^2$ kutsutaan kuvaavasti harjanteiseksi kohinaksi, jonka vaikutusta voidaan korostaa entisestään potenssiin korotuksella. Kohinan tuottamaa arvoa merkittäisiin silloin yhdistettynä funktiona $g(h(x,y)) = (g \circ h)(x,y)$. Kuva 4a esittää aaltoilevaa kohinaa ja kuva 4b harjanteista kohinaa. Dustler ym. (2015) esittävät kaavan, jolla fBm-funktion sisältämän kohinan tuottamaa reaalilukua muokataan ennen arvon summaamista

$$k(x,y) = \sum_{i=0}^{n-1} ap^i \, (g \circ h)(fl^i x, fl^i y). \tag{2.6}$$

Fraktaalisen kohinafunktion voi toteuttaa monella eri tavalla. Dustler ym. (2015) ovat esimerkiksi kuvailleet oktaavien määrittämistä varten vaihtoehtoista toteutustapaa, jossa kohinan kerrostamisen lähtökohdaksi valitaan oktaavien määrän sijaan kohinan suuruutta kuvaava skaala. Perlin (1985) asetti turbulenssifunktion skaalalle taajuudesta riippuvan rajaarvon, jonka avulla fraktaaliselle kohinalle saatiin rajattua taajuusalue. Raja-arvon asettaminen erottaa menetelmän kaavan (2.5) summalausekkeen oktaaveista, sillä iteraatioiden lopetusehto määräytyy suoraan skaalan määräämästä taajuudesta. Skaalan perusteella määritettiin myös kohinan voimakkuus verrannollisena taajuuteen, mistä Perlin (1985) käytti nimitystä $1/f$-kohina. Merkintätavan taustalla on ajatus, että kohinan voimakkuus määräytyy



(a) Aaltoilevaa simpleksikohinaa. (b) Harjanteista simpleksikohinaa.

Kuva 4: Maastotyyppien erityispiirteitä korostetaan vaihtamalla summafunktiota.

taajuuden käänteislukuna, esimerkiksi taajuuden kaksinkertaistuessa $f = 2$ voimakkuus puolittuu $a = 2^{-1}$, joten merkitään $a = f^{-1} = 1/f$. Bourke (1998) määritteli yleisemmin, että fBm-funktio voidaan esittää potenssilain mukaisesti $a = f^{-B} = 1/f^B$, missä kohinan voimakkuuden katsotaan vaihtelevan käänteisen taajuuden potenssina

$$k(x,y) = \sum_{i=0}^{n-1} \frac{h(fl^i x, fl^i y)}{f^B l^{Bi}}, \text{ kun } B \geq 0. \tag{2.7}$$

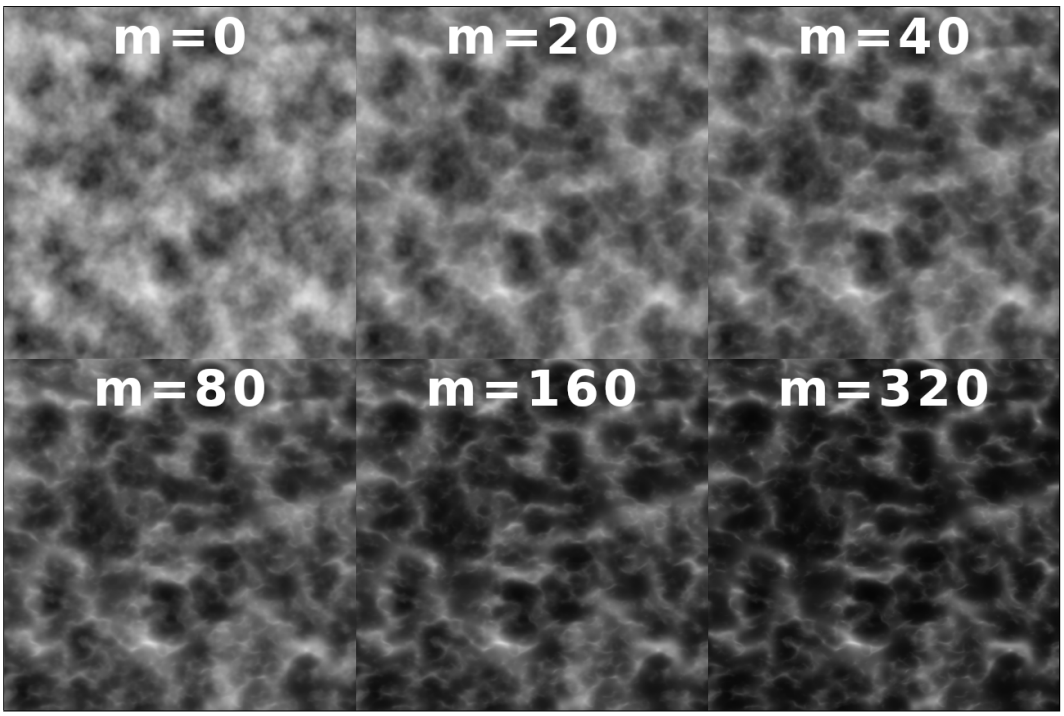
Eroosion vaikutusta on aiemmin simuloitu erillisillä eroosio-algoritmeilla, joista yleisimmät ovat Archerin (2011) vertailemat lämpö- ja vesieroosio. Molemmissa eroosion mallinnuksen tavoissa maastoa kulutetaan eli maata irtoaa ja kulkeutuu alemmas eri tekijöiden, kuten lämpötilan tai sadeveden vaikutuksesta. Algoritmit olennaisesti käsittelevät kohinalla tuotettuja arvoja iteratiivisesti. Quilez (2008) esittää kohinan derivaattoihin perustuvan menetelmän eroosion mallintamiseen. Eroosion vaikutus näkyy siinä, että jyrkät alueet jyrkkenevät entisestään ja vastaavasti tasangot tasoittuvat. Eroosio synnytetään kohinafunktioista saatavien derivaattojen summalla, joka huomioidaan fBm-funktioissa

$$k(x,y) = \sum_{i=0}^{n-1} \frac{ap^i h(fl^i x, fl^i y)}{1 + m \|d_i(x,y)\|^2}, \tag{2.8}$$

missä kohinan arvot suhteutetaan sen derivaattavektorin pituuden neliöön

$$d_i(x,y) = \sum_{j=0}^{i} \left(\frac{\partial h}{\partial x}(fl^j x, fl^j y), \frac{\partial h}{\partial y}(fl^j x, fl^j y) \right). \tag{2.9}$$

Kaavan (2.8) menetelmä ei ole välttämättä yhtä realistinen kuin iteratiiviset menetelmät, mutta kuvassa 5 on nähtävissä samantyyppistä vaikutusta. Toisaalta eroosion voimakkuus saadaan parametrisoitua muuttujalla m, mikä laajentaa fBm-funktion toimintaa. Tällä tavalla saadaan valjastettua kohinan analyyttiset derivaatat tehokkaasti hyötykäyttöön ja erilliselle eroosio-algoritmin käytölle ei välttämättä ole tarvetta.



Kuva 5: Eroosion simulointi analyyttisten derivaattojen avulla.

2.3 Arvoalueen vääristyminen

Ebertin ym. (2003) mukaan arvoalueen vääristymisellä tarkoitetaan syötteen häirintää, jossa arvoja evaluoivan funktion syötettä häiritään toisella funktiolla. Syötteen häirintää käytetään tyypillisesti erilaisten graafisten efektien tuottamiseen (Chen, Dachille ja Kaufman 1999), ja maaston korkeusdatan generoinnissa sitä voidaan käyttää vastaavasti hajottamaan kohinan isotrooppisuutta eli toistuvuutta (Nguyen 2007).

Arvoja evaluoivan funktion $f(p)$ ja syötettä häiritsevän funktion $g(p)$ yhdistelmästä rakentuu arvoaluetta vääristävä funktio $f(g(p))$. Quilez (2002) käyttää syötteen häirinnässä kaavaa $g(p) = p + h(p)$, jossa funktiolla $h(p)$ tuotettu häiriö summataan alun perin annettuun

syötteeseen. Tällä tavoin häiriötä on tarkempaa kontrolloida erillisellä funktiolla. Fraktaalisen kohinan käytön tapauksessa arvoja evaluoivana funktiona f on fBm-funktio ja funktiolla $g(p)$ häiritään syötteenä annettua pistettä p.

Syötettä häiritsevä funktio g voi teoriassa olla mitä tahansa kunhan se palauttaa evaluoivalle funktiolle soveltuvan syötteen. Kohinan käyttö on havaittu hyväksi tavaksi tehdä syötteen häirintää, esimerkiksi Perlin (1985) tuottaa marmoria muistuttavan tekstuurin häiritsemällä sinifunktiolla tuotettua tekstuuria fraktaalisella kohinalla. Vastaavasti Quilez (2002) demonstroi fraktaalisella kohinalla tuotettua häiriön käyttöä fBm-funktioilla generoituihin tekstuureihin.

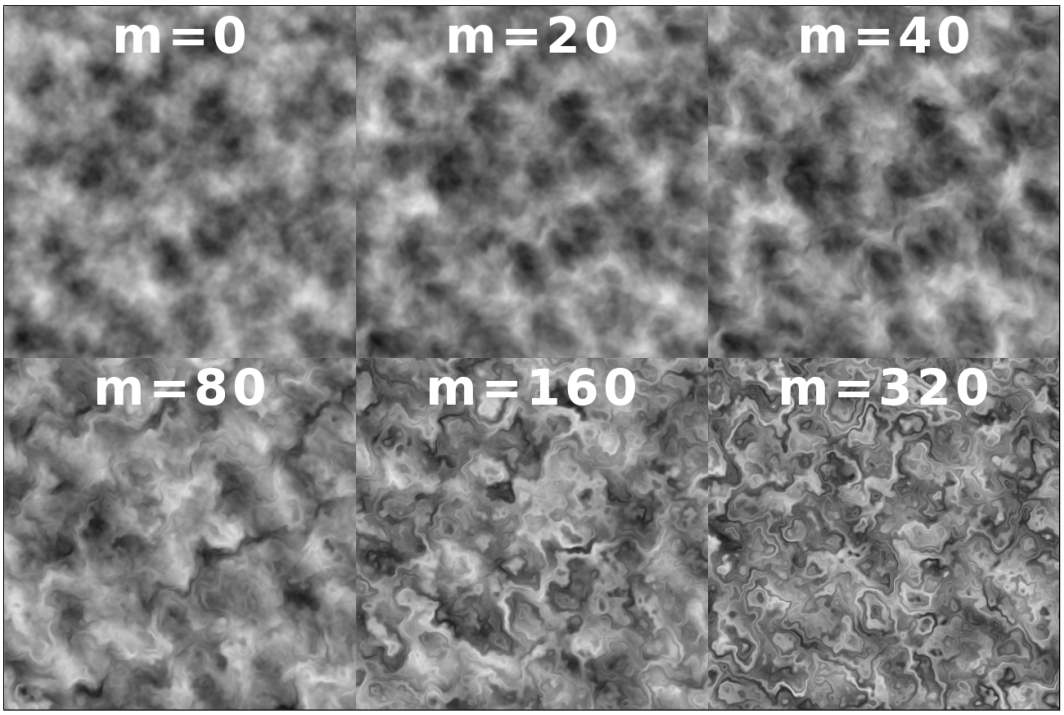
Fraktaalista kohinaa käyttäessä syötteen häirintään tulee huomioida, että se palauttaa arvona yksittäisen reaaliluvun. Jos arvojen evaluointiin käytettävä fBm-funktio ottaa syötteenä pisteen p, joka sisältää n määrän reaalilukuja, niin syötteen häirintää voidaan tehdä erikseen fBm-funktioilla jokaisen pisteen p ulottuvuuksia edustavan reaaliluvun suhteen. Esimerkiksi fraktaaliselle kohinafunktiolle voi esittää häiriötä tuottavan funktion g seuraavasti:

$$g(x,y) = (x + m\,h_0(x,y), y + m\,h_1(x,y)) \tag{2.10}$$

Kaava (2.10) soveltuu häiriön tuottamiseen evaluoinnissa käytettävälle funktiolle $f(p)$, joka ottaa parametrina kaksi reaalilukua sisältävän pisteen. Kirjaimella h merkityt funktiot edustavat häiriön tuottamiseen käytettäviä fBm-funktioita ja m on erikseen määrätty kerroin, jolla voidaan skaalata häiriön voimakkuutta.

Häiriön generointiin voi käyttää luovuutta ja sen tuottamiseen käytettävä funktio h voi olla kohinafunktion sijaan esimerkiksi trigonometrinen funktio. Vastaavasti h kirjaimella merkittyjen funktioiden ei tarvitse olla keskenään samat ja niiden välillä voi olla eroja. Esimerkiksi Quilez (2002) erottaa häiriön tuottamisessa käytettävät fBm-funktiot toisistaan lisäämällä niihin siirtymiä: $h(x + 5.2, y + 1.3)$. Quilez (2002) lisäksi havainnollistaa tapauksia, jossa häiriötä lasketaan rekursiivisesti $g(p) = p + h(p + h(p))$ useampia kertoja ennen lopullisen arvon evaluointia.

Maaston korkeusdatan generoinnissa kaava (2.10) riittää hyvin pohjaksi häiriön tuottamiselle, koska se kykenee tehokkaasti rikkomaan kohinan tyypillistä rakennetta. Kuva 6 antaa esimerkin kaavalla (2.10) tuotetusta vääristymästä, jossa kuvan generointiin sekä häiriön tuottaminen on tehty käyttäen samaa fBm-funktiota. Kuvassa vääristymän tuottamiseen käytetyn häiriön voimakkuutta on kasvatettu tapausten välillä, josta ilmenee häiriön tuottama vaikutus evaluoinnissa käytettyyn fBm-funktioon.



Kuva 6: Vääristymän tuottaminen syötteen häirinnän avulla.

2.4 Laajojen maastojen generointi

Luvussa käsitellään yhteenvetona miten erilaisia kohinan kerrostamisen menetelmiä voi soveltaa laajojen maastojen korkeusdatojen generointiin. Menetelmillä sellaisenaan saadaan generoitua erilaisia isotrooppisia maastotyyppejä, mutta laajan maaston tapauksessa tavoitteena on muodostaa korkeusdataa, jossa nämä erilaisilla kohinoilla tuotetut maastonpiirteet vaihtelevat alueittain. Monipuolinen maasto rakentuu useiden eri maastotyyppien yhteisvaikutuksesta ja maskeja hyödyntämällä niiden esiintymistä voi kontrolloida laajemmassa mittakaavassa, jolloin saadaan tuotettua monipuolisesti vaihtelevaa maastoa.

Maskien käyttö tarjoaa joustavuutta, sillä maski voi olla esimerkiksi fBm-funktio tai ennalta määrätty harmaasävykuva. Unity-projektissa käytimme fBm-funktioita maskeina, mutta ennalta määrätyillä maskeillakin on paikkansa, jos tiettyjen proseduraalisesti tuotettujen maastonpiirteiden esiintymistä halutaan kontrolloida tarkasti artistin toimesta. Esimerkiksi Smelik ym. (2010) käyttivät tällaista lähestymistapaa sotilaskäyttöön suunnatussa viitekehyksessä, jossa 3D-malli generoidaan maastosta yksinkertaisen luonnoksen pohjalta.

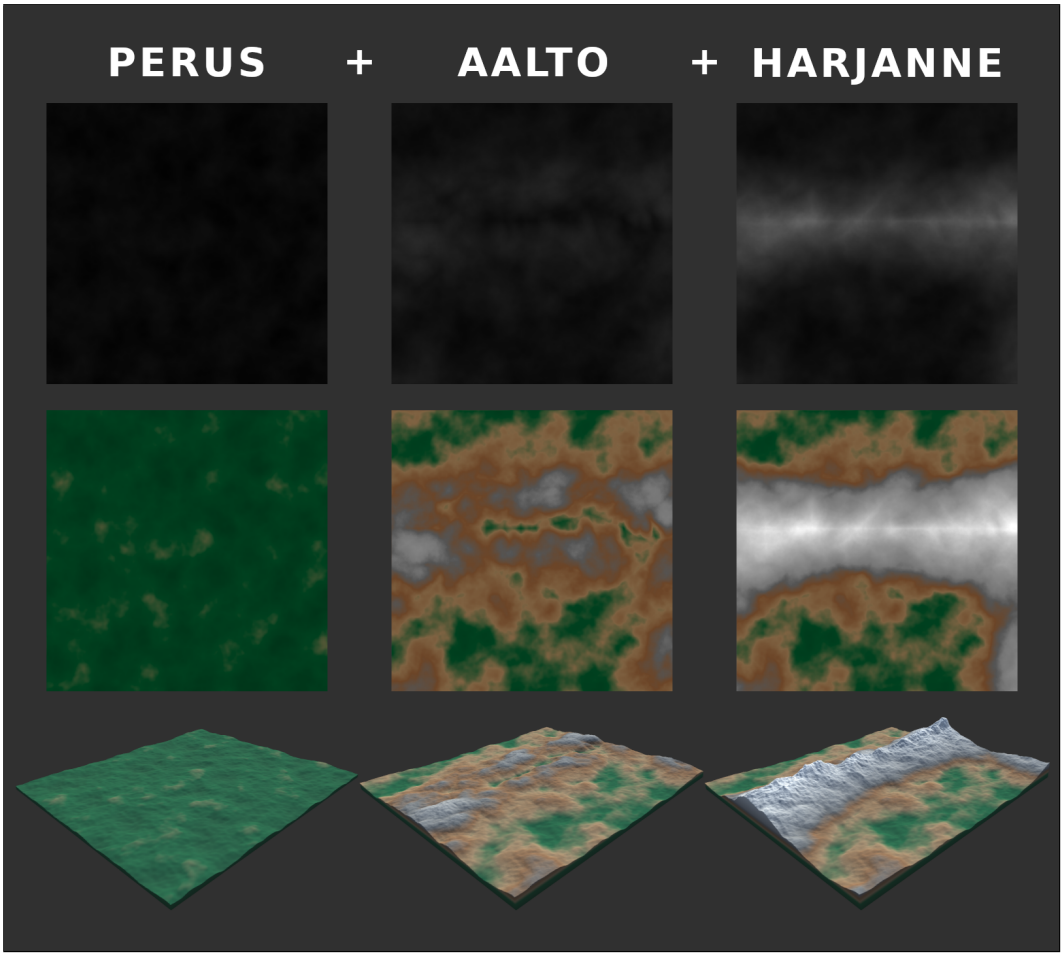
Yksittäinen korkeutta generoiva kerros rakentuu arvoja palauttavasta fraktaalisesta kohinafunktiosta $f(x,y)$ sekä sille erikseen osoitetuista maskeista $M = \{m_1(x,y), \ldots, m_n(x,y)\}$. Maskien tehtävä on määrätä maastonpiirteen voimakkuus parametrina annetun koordinaatin kohdalla ja maaston korkeutta edustava luku lasketaan kaavalla

$$l(x,y) = f(x,y) \prod_{i=1}^{n} m_i(x,y). \tag{2.11}$$

Maskien määrää ei ole rajoitettu, koska valtavien maastojen tapauksessa voidaan esimerkiksi käyttää pohjamaskia, joka määrää mantereiden esiintymisen ja sen lisäksi muita maskeja mantereella esiintyvien maaston piirteiden kuten vuoristojen muodostamiseen. Lopullinen maaston korkeusdatan generointiin käytettävä funktio muodostuu korkeutta generoivien kerroksien joukosta $L = \{l_1(x,y), \ldots, l_n\}$ ja maaston korkeutta edustava luku lasketaan niistä palautuvien arvojen summana:

$$e(x,y) = \sum_{i=1}^{n} l_i(x,y). \tag{2.12}$$

Maskien käyttöä sovellettiin tutkielman rinnalla kehitetyssä Unity-projektissa ja se osoittautui toimivaksi ratkaisuksi yhdistää monia korkeusdataa tuottavia menetelmiä keskenään. Kuva 7 esittää ohjelmalla generoitua korkeusdataa, jossa maskeja käyttäen on määritetty vuoriston esiintyminen sekä sekoitettu keskenään kolmea erilaista kohinafunktiota. Kuvassa demonstroidaan vaiheittain korkeusdatan piirteiden muutokset, kun kaavalla (2.5) tuotetun kohinan rinnalle on ensiksi lisätty aaltokohinaa ja sen jälkeen harjannekohinaa.



Kuva 7: Maskien avulla yhdistettyä kohinaa.

3 Vokseleihin perustuva pinnanmuodostus

Pinnanmuodostusalgoritmien tehtävänä on tuottaa dataa 3D-mallin esittämistä varten. Tyypillisesti 3D-mallin pinta määritetään kolmioina, joita grafiikkaprosessorit ovat erikoistuneet käsittelemään (Owens ym. 2008). Pinnanmuodostusalgoritmin täytyy siis generoida vähintäänkin kolmioita merkitsevät kolmiulotteisen avaruuden pisteet sekä ulkopintojen suuntia osoittavat normaalivektorit, jotta 3D-malli kyetään esittämään tietokonegrafiikkana. Yleensä 3D-mallille määritetään myös teksturointiin liittyvää dataa.

Vokselit muodostavat joukon säännöllisen välimatkan päässä sijaitsevista datapisteistä kolmiulotteisessa avaruudessa. Tutkielman tapauksessa vokselit edustavat tilavuudellista dataa eli volyymiä, joka on tiettyyn raja-arvoon verrattuna joko pienempää, suurempaa tai yhtäsuurta. Vokseleihin pohjautuvat pinnanmuodostusalgoritmit mallintavat volyymiä niin, että 3D-mallin pinta muodostuu ulospäin vokseleista, joissa on raja-arvoa suurempi luku. Tarkemmin sanottuna 3D-mallin pinta rakennetaan sellaisten vokselien välille, jossa vierekkäisten vokselien arvot ovat eripuolin raja-arvoa.

Luvussa käsitellään marssikuutiot-algoritmia ja naiivia pintaverkkoalgoritmia, jotka ovat vokseleihin pohjautuvia pinnanmuodostusalgoritmeja. Molemmat algoritmit ottavat syötteenä kolmiulotteisen taulukon, jonka alkiot ovat vokseleiden volyymiä edustavia lukuja. Kolmiulotteisen taulukon alkiot tulee muuntaa algoritmien tapauksessa vokseleiden sijainneiksi siihen kolmiulotteiseen avaruuteen, missä 3D-mallin generointi tapahtuu. Vokselin sijainti lasketaan taulukon alkion indeksin perusteella kaavalla

$$P = (x,y,z) + s\,(i,j,k), \text{ kun } s > 0. \tag{3.1}$$

Kaavassa (3.1) vokselin sijainti P lasketaan summaamalla taulukon ensimmäistä arvoa edustavaan pisteeseen (x,y,z) alkion indeksi (i,j,k), joka kerrotaan määritetyllä vokselin koolla s. Taulukko muodostaa siten vokseleiden joukon, joka koostuu säännöllisen välisistä datapisteistä kolmiulotteisessa avaruudessa. Käänteisesti saadaan selvitettyä vokselin sijainnin perusteella missä taulukon indeksissä vokselin volyymiä kuvaava arvo sijaitsee

$$(i,j,k) = \frac{P-(x,y,z)}{s}, \text{ kun } s > 0. \tag{3.2}$$

Luvussa 3.1 käsitellään tunnettua marssikuutiot-algoritmia, joka esitetään listauksessa 1 algoritmin perusrakenteen kuvaavana pseudokoodina. Luvussa 3.2 käsitellään vähemmän tunnettua naiivia pintaverkkoalgoritmia, jonka toimintaperiaate kuvataan pseudokoodina listauksessa 2.

3.1 Marssikuutiot-algoritmi

Lorensen ym. (1987) kehittivät marssikuutiot-algoritmin lääketieteellisen datan kuvantamista varten. Se on edelleen laajasti käytetty algoritmi, jolle on vuosien saatossa löytynyt useita erilaisia käyttötarkoituksia monilta tietokonegrafiikkaa soveltavilta aloilta (Lorensen 2020). Algoritmi on toiminut merkittävänä vaikutteena monille volyymin mallintamiseen käytettäville pinnanmuodostusalgoritmeille, kuten Doin ja Koiden (1991) marssinelitahokas-, Gibsonin (1998) pintaverkko-, Jun ym. (2002) kaksoiääriviiva- ja Lysenkon (2012) naiivi pintaverkkoalgoritmi sekä monille muille vähemmän tunnetuille algoritmeille. Marssikuutiot-algoritmi oli patentoitu vuoteen 2005 asti, mikä on toiminut yhtenä kannusteena vaihtoehtoisten algoritmien kehittämiseen ja tutkimiseen. Volyymin mallintamiseen käytettävien algoritmien tarjonnan lisääntymisestä huolimatta marssikuutiot-algoritmi on säilyttänyt suosionsa. Seuraavaksi luvussa käsitellään tarkemmin algoritmin toiminta.

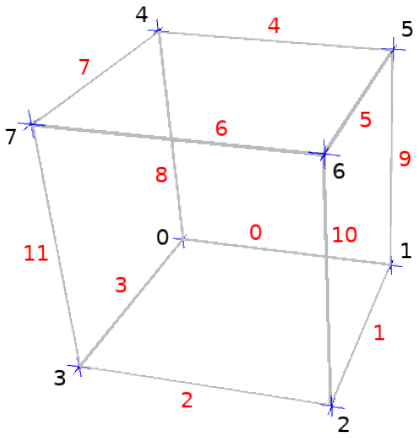
Listaus 1 esittää sanallisesti kuvatun pseudokoodin marssikuutiot-algoritmin toiminnasta. Algoritmi ottaa syötteenä vastaan kolmiulotteisen taulukon, jonka alkiot edustavat volymetristä dataa. Ensimmäisessä vaiheessa alustetaan tietorakenteet generoitavan 3D-mallin datalle. Tyypillisesti 3D-mallin esittämistä varten tarvitaan tietorakenteet pisteille, kolmioille ja normaaleille, mutta generoidessa on myös mahdollista tuottaa teksturointiin liittyvää dataa.

```
1. Alustetaan 3D-mallille määriteltävä data listoina:
   pisteet, kolmiot ja normaalit.

2. Silmukoidaan funktiolle syötetty taulukko, jossa
   on kolme dimensiota.

    2.1. Määritetään käsiteltävän kuution konfiguraatio.
```

```
2.2. Silmukoidaan kolmioittain (i+=3) konfiguraation
     osoittama kohta kolmiointitaulusta.

  2.2.1. Jos taulun osoittama särmä i on −1,
         lopetetaan taulun silmukointi.

  2.2.2. Lasketaan taulun osoittamien kuution
         särmille sijoittuvat kolmion pisteet.

  2.2.3. Lisätään kolmion data 3D−mallille.

3. 3D−mallin data on generoitu.
```

Listaus 1: Marssikuutiot-algoritmin toiminta esitettynä pseudokoodina.

Tarvittavien tietorakenteiden alustamisen jälkeen listauksessa 1 siirrytään algoritmin toiseen vaiheeseen, jossa generoidaan 3D-mallin data. Algoritmi käsittelee sille syötetyn taulukon kuutioittain askeltamalla ja silmukoinnin tapauksessa tämä tarkoittaa, että taulukkoa käydään läpi normaalisti alkio kerrallaan ja kuutio peilataan käsiteltävän alkion suhteen ennalta määrätyillä siirtymillä. Kuva 8 havainnollistaa tätä kuvitteellista kuutiota ja numerolla kolme merkitty nurkka edustaa aina käsiteltävää alkiota, jonka suhteen kuution muita kärkipisteitä edustavat taulukon alkiot ovat määriteltävissä.



Kuva 8: Kuution särmien ja kärkipisteiden indeksointi.

Taulukon silmukoinnissa tulee huomioida, että täysin kiinteän 3D-mallin luomiseksi tulee käsitellä myös taulukon ulkopuoliset kuutiot, jotka muodostuvat taulukon kattaman alueen reunalla olevien vokseleiden ja ulkopuolisten vokseleiden kanssa. Aikaisemmin mainittu ehto pinnanmuodostukselle täyttyy ulkopuolelle muodostuvissa kuutioissa, jos taulukon ulkopuoliset arvot mielletään automaattisesti ilmana ja taulukon reunamilla on materiaalia edustavia alkioita. Jos pinta jätetään muodostamatta tällaisista osittain ulkopuolisista kuutioista, niin generoitavan 3D-mallin pinnan yhtenäisyys rikkoontuu. Algoritmilla tehtävä silmukointi riippuu paljon ohjelman taustalla tehtävästä datan käsittelystä. Esimerkiksi tarvetta askeltaa taulukon ulkopuolelle ei ole, jos syötteen ulkopuolelle jäävät kuutiot ovat osana toisia generoitavia 3D-malleja.

Listauksen 1 kohdassa 2.1 esitetty ensimmäinen operaatio silmukan sisällä on konfiguraation määrittäminen käsiteltävälle kuutiolle, joka toimii osoittimena 3D-mallin datan generoinnissa hyödynnettävään kolmiointitauluun. Konfiguraatio on kahdeksan-bittinen luku, joka lasketaan summana läpikäymällä käsiteltävän kuution osoittamat nurkkien arvot taulukosta ja vertaamalla niitä määrättyyn raja-arvoon. Matemaattisesti konfiguraation määrittämisen voi esittää seuraavasti:

$$f(x) = \sum_{i=0}^{7} \begin{cases} 0, \text{ kun } v_i \leq r \\ 2^i, \text{ kun } v_i > r \end{cases} \quad , \text{kun } x \in X. \tag{3.3}$$

Kaavassa (3.3) joukko $X = \{v_0, \ldots, v_7\} \in \mathbb{R}$ edustaa kuution nurkkien osoittamaa volymetristä dataa. Jos käsiteltävän nurkan arvo v_i on suurempaa kuin ennalta määrätty raja-arvo r, laskettavaa konfiguraatiota summataan kaavalla 2^i. Joukoon X määritetyissä arvoissa noudatetaan kuvassa 8 osoitettua nurkkien järjestystä, jonka perusteella 3D-mallin pisteiden laskennassa käytettävä kolmiointitaulu on määritetty konfiguraatioille. Konfiguraatio on mahdollista määrittää päinvastaisesti niin, että tapaus 2^i suoritetaan silloin, kun volyymin arvo v_i on pienempää kuin raja-arvo r. Esimerkiksi Bourke (1994) suorittaa konfiguraation määrityksen niin päin. Valintaan vaikuttaa ensisijaisesti ajattelutapa, että muodostuuko pinta ulospäin raja-arvon ylittävistä vokseleista vai ei.

Erilaisia konfiguraatioita on yhteensä 2^8 tapausta, joka määräytyy kaavasta (3.3), jossa laskettava indeksi on kokonaisluku välillä [0,255]. Jokainen konfiguraation tapaus ei kuitenkaan

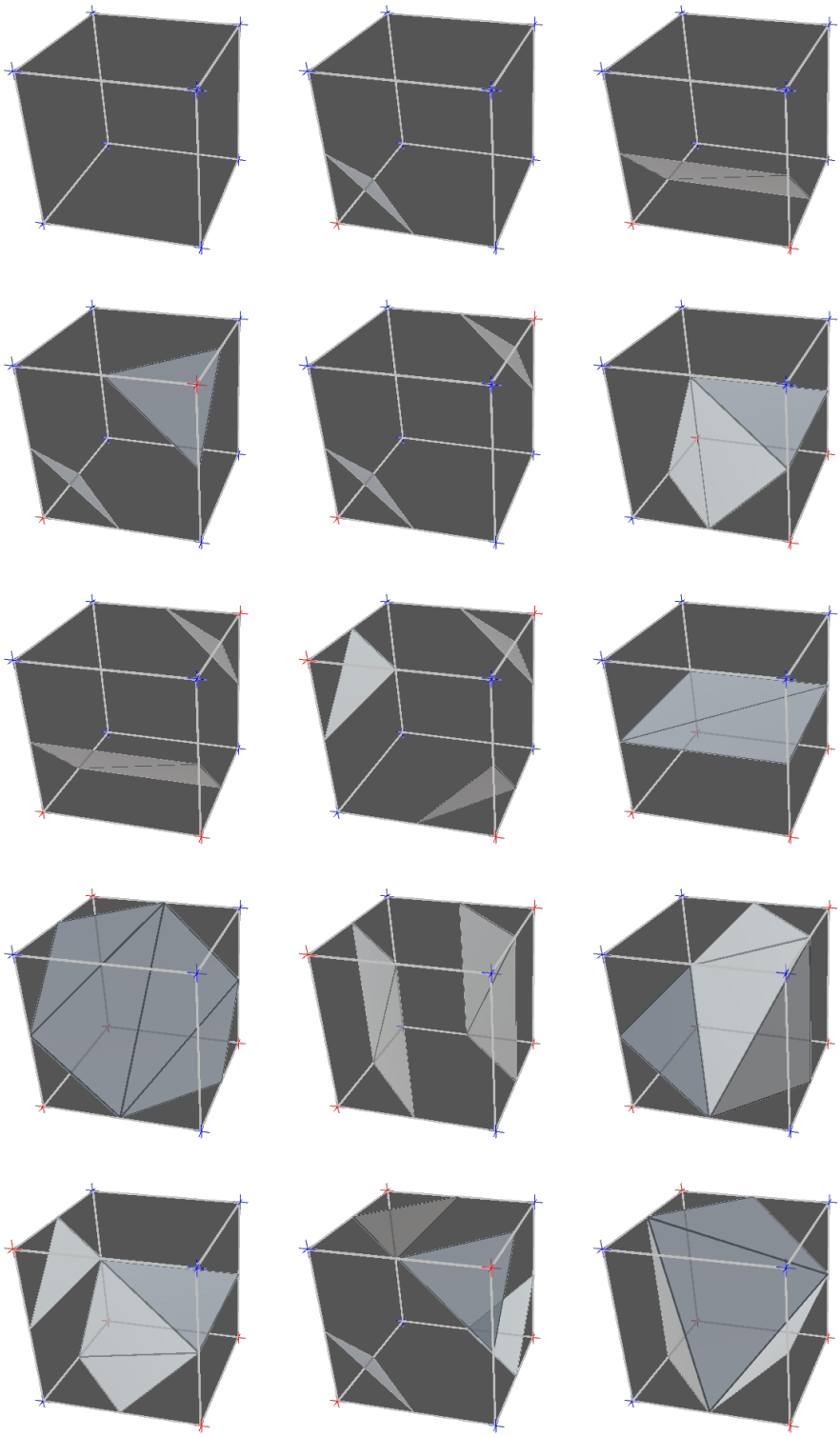
ole täysin uniikki, vaan tietyt tapaukset toistuvat useasti erilaisella rotaatiolla. Lorensen ja Cline (1987) kiteyttivät erilaiset konfiguraatiot viiteentoista erilaiseen perustapaukseen. Kuva 9 esittää nämä perustapaukset, jossa on käytetty Bourken (1994) käyttämää kolmiointitaulua.

Konfiguraation määrittämisen jälkeen siirrytään generoimaan 3D-mallin dataa käyttäen ennalta määrättyä kolmiointitaulua, joka osoittaa konfiguraation perusteella vokseleiden muodostaman kuution särmät, joihin 3D-mallille tulevat pisteet lasketaan. Marssikuutiot-algoritmilla 3D-mallin pinta muodostetaan kolmioina, joten taulu silmukoidaan $i+=3$ kokoisilla askelluksilla laskien kerralla kolme pistettä kolmion muodostusta varten. Tutkielman ohessa kehitetyssä Unity-projektissa käytettiin Bourken (1994) artikkelissa annettua taulua. Silmukoidessa luku -1 tarkoitti, että konfiguraatio ei sisällä eteenpäin käsiteltäessä laskettavia 3D-mallin pisteitä ja tämä tarkistus tehdään listauksen 1 kohdassa 2.2.1. Muussa tapauksessa taulun alkiot toimivat osoittimina kuvassa 8 esitettyihin kuution särmiin, joissa ne ovat numeroitu punaisella värillä. Kuution särmälle laskettavan 3D-mallin pisteen laskemiseksi voidaan käyttää lineaarista interpolaatiota raja-arvon suhteen, joka voidaan esittää kaavalla

$$l(p_1, p_2, v_1, v_2) = p_1 + \frac{p_2 - p_1}{v_2 - v_1} \cdot (r - v_1), \text{ kun } v_1 \neq v_2. \tag{3.4}$$

Kaavassa (3.4) särmän muodostavat kuution pisteet p_1 ja p_2 saadaan laskettua käyttäen kaavaa (3.1), jossa taulukon indeksi muunnettiin vokselin sijainniksi kolmiulotteisessa avaruudessa. Volyymiä edustavat särmän vokseleiden arvot v_1 ja v_2 ovat vuorostaan saatavilla algoritmille syötetystä taulukosta ja r edustaa ennalta määrättyä raja-arvoa. Interpolaation käytöllä pisteiden laskennassa saadaan aikaiseksi, että generoitavan 3D-mallin pinta mukautuu volyymin voimakkuuksiin. Mitä suurempi erotus volyymillä on raja-arvoon sitä voimakkaammin se työntää laskettavaa 3D-mallin pisettä vastakkaista särmän pistettä kohti.

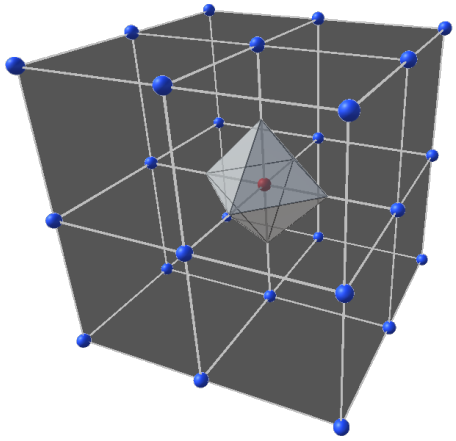
Kolmion pisteiden laskemisen jälkeen data 3D-mallille voidaan lisätä listauksen 1 kohdassa 1 alustettuihin tietorakenteisiin. Kolmioinnissa tulee ottaa huomioon kolmiointitaulussa käytetty kolmion pisteiden järjestys, koska eri grafiikkaa tarjoavien rajapintojen suhteen se voi olla myötä- tai vastapäiväinen. Bourken (1994) tarjoamassa kolmiointitaulussa laskettavat kolmion pisteet on annettu vastapäiväisessä järjestyksessä.



Kuva 9: Marssikuutiot-algoritmin viisitoista erilaista konfiguraatiota.

Normaalin laskeminen kolmion pinnalle tehdään ristitulolla kahden sen muodostaman särmän välillä ja ne saadaan määritettyä seuraavasti käyttäen aikaisemmin laskettuja kolmion pisteitä $U = p_2 - p_1$ ja $V = p_3 - p_1$ (The Khronos Group 2013). Särmien laskennassa pisteiden p_1, p_2 ja p_3 oletetaan noudattavan kolmioinnin mukaista järjestystä. Jos pisteet annetaan käytettävään kolmioinnin suuntaan nähden väärässä järjestyksessä, niin normaali osoittaa pinnasta päinvastaiseen suuntaan.

Listauksessa 1 syötetyn taulukon käsittelyn jälkeen tarvittava 3D-mallin data on generoitu ja kuvassa 10 on esitetty 3^3 kokoisesta taulukosta tuotettu 3D-malli, jossa keskimmäisen alkion arvo on suurempaa kuin raja-arvo. Laskettujen 3D-mallin pisteiden sijoittuessa vokseleiden muodostamien kuutioiden särmille, muodoksi syntyy oktaedri. Vastaavasti kuvaa 9 tarkastelemalla voimme havaita, että toisena esitetty konfiguraatio peilautuu kuvassa 10 monesta eri suunnasta, joka tukee aikaisemmin mainittua teoriaa konfiguraatioiden kiteytymisestä tiettyihin perustapauksiin. Oktaedri on myös yksinkertaisin konfiguraatioiden pohjalta muodostuva yhtenäinen kappale.



Kuva 10: Yksinkertaisin marssikuutiot-algoritmilla tuotettu kappale.

3.2 Naiivi pintaverkkoalgoritmi

Naiivi pintaverkkoalgoritmi kuuluu marssikuutioille duaalisten algoritmien joukkoon, johon kuuluvat myös Gibsonin (1998) esittämä pintaverkkoalgoritmi sekä myöhemmin Jun ym. (2002) kehittämä kaksoisääriviiva-algoritmi. Marssikuutioille duaaliset algoritmit tuottavat yksittäistä marssikuutioiden generoimaa kolmion kärkipistettä kohden nelikulmion (Schaefer ja Warren 2003). Nelikulmiot muodostetaan särmän jakavien kuutioiden kesken ja niiden pisteet lasketaan kuutioiden sisäpuolelle. Schaeferin ym. (2003) kuvauksesta tulee huomioida, että marssikuutiot-algoritmin oletetaan jakavan lasketut pisteet kolmioiden kesken, josta generoitavien nelikulmioiden lukumäärä määrittyy.

Merkittävä erotteleva tekijä marssikuutioille duaalisten algoritmien välillä on tapa laskea piste kuution sisäpuolelle ja ennen kuin naiivia pintaverkkoalgoritmia käsitellään tarkemmin, tutustutaan lyhyesti tapoihin kuinka 3D-mallin pisteiden laskentaa tehdään muissa mainituissa duaalisissa algoritmeissa.

Gibsonin (1998) pintaverkkoalgoritmi on ensimmäinen kehitetty marssikuutioihin nähden duaalinen algoritmi, jossa 3D-mallin pisteet määritetään alustavasti vokseleiden muodostamien kuutioiden keskelle. Algoritmissa pinnan silottelu tehdään iteratiivisesti jälkeenpäin siihen erikoistuneella algoritmilla, joka säätelee määritettyjen pisteiden sijainteja tehden pinnasta sulavamman. Myöhemmin kehitetyssä kaksoisääriviiva-algoritmissa pisteiden laskenta tehdään pintaverkkoalgoritmiin verrattuna suoraviivaisemmin käyttäen toisen asteen virhefunktiota (engl. *Quadratic Error Function, QEF*), joka hyödyntää laskennassa pinnan normaaleja särmien leikkauspisteissä (Ju ym. 2002).

Jos marssikuutioille etsitään suorituskyvyllisesti vaihtoehtoista algoritmia ja pelkkä vokseliavaruuden volyymi on tiedossa, niin sekä pintaverkko että kaksoisääriviiva vaativat marssikuutioita enemmän laskentaa. Pintaverkkoalgoritmin hidastava tekijä on jälkeenpäin tehtävä pinnan silottelu ja kaksoisääriviiva-algoritmia varten pitää tehdä pinnan normaalien määrittäminen, joka voidaan tehdä erikseen käyttäen marssikuutiot-algoritmia. Naiivissa pintaverkkoalgoritmissa piste kuution sisään lasketaan ottamalla keskiarvo niistä pisteistä, jotka olisi laskettu marssikuutioiden tapauksessa kuution särmille (Lysenko 2012). Pisteiden laskenta kuvattuun tilanteeseen nähden tapahtuu huomattavasti suoraviivaisemmin verrattuna

pintaverkko- sekä kaksoisääriviiva-algoritmiin, joten se soveltuu näistä kolmesta algoritmista parhaiten haastamaan marssikuutiot-algoritmin suorituskyvyssä.

Naiivin pintaverkon ollessa algoritmina suhteellisen yksinkertainen, se toteutettiin tutkielman ohella kehitetyn Unity-projektin tapauksessa Schaeferin ja Warrenin (2003) kuvauksen pohjalta, jossa he mainitsivat vastaavan pisteiden laskennan puhuessaan Gibsonin (1998) pintaverkkoalgoritmista. Lysenko (2012) on kuitenkin ensimmäinen lähde, joka on osoittanut kyseisen algoritmin olemassaolon itsenäisenä ja nimennyt sen. Erilaisesta pisteen laskennasta johtuen naiivi pintaverkkoalgoritmi voidaan luokitella pintaverkosta sekä kaksoisääriviivasta erilliseksi algoritmiksi.

Listaus 2 esittää pseudokoodina naiivin pintaverkkoalgoritmin toiminnan, jossa algoritmille syötetään marssikuutioiden tavoin kolmiulotteinen vokseleiden volyymejä edustava taulukko. Algoritmin ensimmäisessä vaiheessa alustetaan normaalisti tietorakenteet, joihin 3D-mallin data generoidaan. Erilaisesta pinnanmuodostuksessa johtuen kohdassa 2 alustetaan taulukot, joihin tallennetaan dataa generoidessa tieto vokseleiden muodostamien kuutioiden sisään lasketuista 3D-mallin pisteistä. Pinnanmuodostuksessa samoja kuutioiden sisään laskettuja pisteitä tarvitaan useita kertoja, joten algoritmin suorituskyvylle on tärkeää, että pisteet kuutioiden sisään lasketaan vain kerran. Luvussa 3.1 esitetyn marssikuutiot-algoritmin tavoin, listauksessa 2 askelletaan syötetyn taulukon ulkopuolelle. Vokseleiden muodostamia kuutioita on siten taulukon koon suhteen yhden verran enemmän dimensioittain ja tämä huomioidaan kohdassa 2 tehtävässä taulukoiden alustuksessa.

```
1. Alustetaan 3D-mallille määriteltävä data listoina:
   pisteet, kolmiot ja normaalit.

2. Alustetaan syötetystä taulukosta dimensioittain
   yhden verran suuremmat taulukot, joista toiseen
   tallennetaan kuutioiden sisään lasketut pisteet
   ja toiseen tieto onko kyseisen kuution sisään
   laskettu piste vai ei. Tehtävänä on vähentää
   laskennan tarvetta 3D-mallin dataa generoidessa.
```

```
3. Silmukoidaan funktiolle syötetty taulukko, jossa
   on kolme dimensiota.

    3.1. Jos käsiteltävä arvo <= raja-arvo,
         jatketaan seuraavaan alkioon.

    3.2 Käydään läpi alkion naapurit kuudessa suunnassa:
        edestä, takaa, vasemmalta, oikealta, ylhäältä ja
        alhaalta.

        3.2.1 Jos naapurin arvo > raja-arvo,
              pintaa ei muodosteta. Jatketaan
              seuraavaan käsiteltävään suuntaan.

        3.2.2 Haetaan tai lasketaan alkioiden muodostaman
              särmän jakavien neljän kuution sisään
              lasketut 3D-mallin pisteet. Jos pistettä
              ei ole laskettu kuution sisään aikaisemmin,
              lasketaan se ja merkitään käsitellyksi.
              Muussa tapauksessa kuution piste on valmiiksi
              saatavissa kohdassa 2. määritetystä
              taulukosta.

        3.2.3 Nämä neljä pistettä muodostavat nelikulmion,
              lisätään nelikulmion data 3D-mallille.

4. 3D-mallin data on generoitu.
```

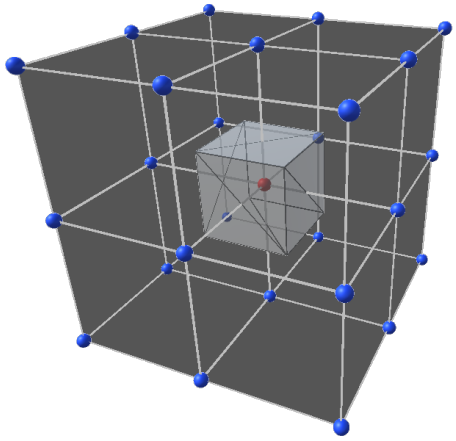
Listaus 2: Naiivin pintaverkkoalgoritmin toiminta esitettynä pseudokoodina.

Listauksen 2 kohdassa 3 syötetty taulukko silmukoidaan ja suoritetaan 3D-mallin datan generointi. Pintaa yritetään muodostaa pelkästään raja-arvon ylittävistä alkioista ja kohdassa 3.1 tehdään kyseinen tarkistus. Nelikulmio voidaan muodostaa vokselia ympäröivien kuutioiden suhteen kuudesta eri suunnasta: edestä, takaa, vasemmalta, oikealta, ylhäältä ja alhaalta (Gibson 1998). Ehtona nelikulmion muodostamiseen on, että nelikulmion pinnan osoittamassa suunnassa olevan naapurin arvo on pienempää kuin raja-arvo. Kohdassa 3.2 on alus-

tettu silmukka naapurin arvojen tarkistusta varten ja silmukan sisällä kohdassa 3.2.1 verrataan naapurin arvoa suhteessa raja-arvoon. Pinnan muodostuksen ehdon täyttyessä nelikulmio muodostetaan niiden kuutioiden sisältämien pisteiden kesken, jotka jakavat käsiteltävän vokselin ja sen naapurin muodostaman särmän. Kolmiulotteisten mallien pohjautuessa kolmioihin perustuvaan topologiaan, nelikulmio muodostetaan kahdella kolmiolla käyttäen näitä neljää kuutioiden sisään laskettua pistettä.

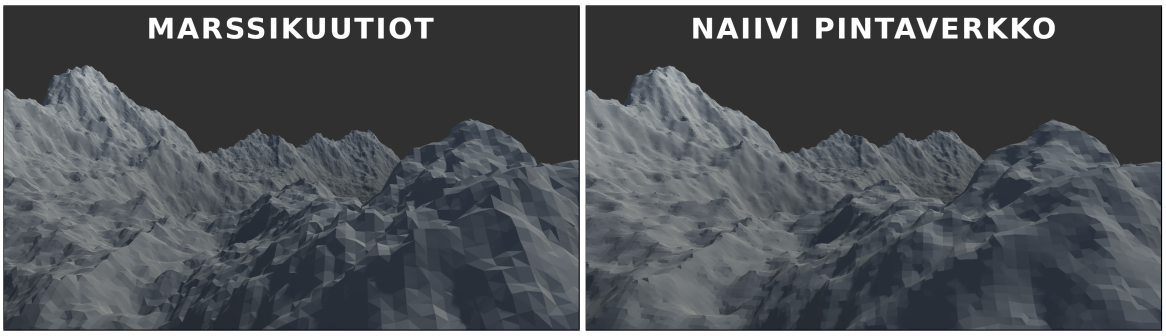
Kuution sisään tehtävä pisteiden laskeminen tapahtuu läpikäymällä sen särmät ja laskemalla summaa särmille lasketuista pisteistä, joissa särmän muodostavien vokseleiden volyymit ovat eripuolin raja-arvoa. Tällaiselle särmälle lasketaan marssikuutioiden tapauksessa aina piste ja sen laskemiseen voidaan käyttää samaa aikaisemmin esitettyä lineaarisen interpolaation kaavaa (3.4). Lopullinen kuution sisään sijoittuva piste saadaan ottamalla keskiarvo särmille lasketuista pisteistä. Lasketut pisteet tallennetaan aina listauksen 2 kohdassa 2 alustettuun taulukkoon, jotta samoja pisteitä ei lasketa useita kertoja uudestaan.

Kuvassa 11 on havaittavissa kuinka naiivi pintaverkkoalgoritmi on generoidut nelikulmiot yksittäisen pintaa muodostavan vokselin ympärille. Schaeferin ym. (2003) esittämät duaalisen menetelmän erot ovat nähtävissä verratessa sitä kuvan 10 vastaavaan tilanteeseen. Marssikuutiot-algoritmin kuudella kärkipisteellä esitetystä oktaedrista on tullut kuudella nelikulmiolla muodostettu kuutio naiiviin pintaverkkoalgoritmin tapauksessa.



Kuva 11: Yksinkertaisin naiivilla pintaverkkoalgoritmilla tuotettu kappale.

Kuvissa 12a ja 12b on esitetty vertailu marssikuutioiden sekä naiviin pintaverkon välillä muodostetuista 3D-malleista, jotka ovat generoitu samasta maastoa edustavasta volyymistä. Maaston tapauksessa algoritmien generoimien 3D-mallien visuaaliset erot ovat melko vähäiset, mutta erilaisesta pinnanmuodostuksesta johtuen pinnassa on havaittavissa pieniä eroja algoritmien välillä. Marssikuutioiden tapauksessa yksittäiset kolmiot erottuvat tietyissä tilanteissa 3D-mallista selvästi ja vastaavasti naiivilla pintaverkolla tuotetusta mallista on havaittavissa nelikulmioita. Nelikulmioilla muodostettu pinta noudattaa kokonaisvaltaisesti säännöllisempää rakennetta.



(a) Marssikuutiot-algoritmi. (b) Naiivi pintaverkkoalgoritmi.

Kuva 12: Marssikuutiot- ja naiivin pintaverkkoalgoritmin maastomallien vertailu.

4 Pinnanmuodostusalgoritmien soveltaminen

Tutkielma sai alkunsa pelikehityksellisen Unity-projektin pohjalta, jonka ideana oli tutustua marssikuutiot-algoritmiin. Marssikuutioiden toteuttamisen jälkeen heräsi kiinnostus tutkia vokseleihin perustuvia pinnanmuodostusalgoritmeja tarkemmin. Tutkimuksen aikana projektin rooli oli tärkeä, koska sillä toteutettiin ja testattiin luvuissa 2 ja 3 käsiteltyjä teorioita käytännössä. Lisäksi projektin pohjalta pystyttiin toteuttamaan tutkielmatyöhön sisältyvä tutkimusprojekti, joka mahdollisti luvun 5 algoritmien suorituskyvyn vertailun.

Tutkielman algoritmitutkimus toteutettiin pelinkehityksellisistä taustoista johtuen Unity-pelimoottorilla, jonka ohjelmointiympäristössä käytettiin C#-ohjelmointikieltä. Pinnanmuodostusalgoritmien osalta marssikuutiot-algoritmi toteutettiin Bourken (1994) C-kielisen esimerkkikoodin pohjalta. Naiivi pintaverkkoalgoritmi toteutettiin puolestaan Schaeferin ym. (2003) artikkelissa esiintyneen virkkeen perusteella. Proseduraalisen generoinnin osalta analyyttisillä derivaatoilla laajennettu simpleksikohina implementoitiin Gustavsonin (2008) C-kielisen esimerkkikoodin pohjalta. Kyseisen ohjelmatoteutuksen tekijä on sama henkilö, joka kirjoitti tutkimusraportin simpleksikohinasta (Gustavson 2005). Fraktaalisen kohinan ja arvoalueen vääristämisen toteutukset sovellettiin pääosin Quilezin (2002, 2019) GLSL-kielisten esimerkkikoodien pohjalta. Quilez (2008) esittää myös GLSL-kielisten esimerkkien valottamana kuinka Perlin-kohinasta saadaan laskettua analyyttiset derivaatat, kunhan tiedetään Perlinin (2002b) käyttämät gradienttivektorit.

Luvussa 4.1 käsitellään suunnittelutieteellistä viitekehystä, jossa tutkielman aikana kehitetyt pinnanmuodostusalgoritmit toimivat artefakteina. Luvussa 4.2 selitetään kuinka päädyimme löytämään sekä kehittämään naiivin pintaverkkoalgoritmin tutkimusprojektiin. Luvussa 4.3 käsitellään maastoa pinnanmuodostusalgoritmien sovelluskohteena. Luvussa 4.4 käsitellään tarkemmin kuinka luvun 2 proseduraalisia menetelmiä hyödynnettiin Unity-projektissa maaston korkeusdatan generointiin.

4.1 Suunnittelutieteellinen viitekehys

Hevner ym. (2004) kertovat artikkelissaan kuinka suoritetaan, arvioidaan ja esitellään oikeaoppinen suunnittelutieteen tutkimus. He kuvaavat suunnittelutieteen alan käsitteellisen viitekehyksen ja esittävät joukon ohjesääntöjä tutkimuksen suorittamiseen ja arviointiin. Tutkimuksessa tulee ottaa kantaa jokaiseen seitsemästä ohjesäännöstä, jotta suunnittelutieteen tutkimus täyttäisi menetelmälle asetetut vaatimukset. Tämän perusteella tutkimuksen arvioijat tekevät omat tulkintansa siitä, kuinka hyvin tutkimus käsittelee ohjesääntöjen kehittämisen taustalla olleita aikomuksia ja tarkoitusperiä (Hevner ym. 2004).

Suunnittelutieteen tutkimuksessa konstruoidaan ja evaluoidaan artefakteja, jotka on suunniteltu johonkin reaalimaailman käyttötarkoitukseen (Hevner ym. 2004). Tutkielmaan valittu konstruktiivinen tutkimusote onkin suunnittelutieteen menetelmä. Artefaktit ovat ihmisten tekemiä esineitä, rakennelmia tai vastaavia konstruktioita (Hevner ym. 2004), mutta tutkielman tapauksessa artefaktilla tarkoitetaan pinnanmuodostusalgoritmeja. Evaluoinnilla tarkoitetaan pinnanmuodostusalgoritmien suorituskyvyn arviointia. Suunnittelutieteellisellä tutkimuksella konstruoidaan joko innovatiivinen ratkaisu johonkin ratkaisemattomaan ongelmaan tai parannetaan olemassa olevia ratkaisuja (Hevner ym. 2004).

4.2 Vaihtoehtoisen pinnanmuodostusalgoritmin valintaperusteet

Lorensenin ym. (1987) marssikuutioiden toteuttamisen jälkeen kiinnostuksemme heräsi muihin vastaaviin pinnanmuodostusalgoritmeihin ja tutkimuksen toteuttamista varten tavoitteena oli löytää marssikuutioille mahdollinen korvaava algoritmi. Näistä seuraavaksi tunnetuin vaikutti olevan Jun ym. (2002) kehittämä kaksoisääriviiva-algoritmi. Marssikuutioista poiketen kaksoisääriviiva oletti tunnetuksi muodostettavan 3D-mallin tarkat pinnan normaalit, joten se ei soveltunut tutkielman käyttötarpeisiin. Marssikuutiot pystyivät generoimaan 3D-mallin suoraan vokseleissa olevan volyymin pohjalta, joten korvaavalle algoritmille haluttiin vastaavaa lähestymistapaa 3D-mallien generointiin.

Kaksoisääriviiva-algoritmin jälkeen selvitettiin Gibsonin (1998) pintaverkkoalgoritmia vaihtoehtoiseksi vertailun kohteeksi, mutta tavoitteena oli tehdä pisteen laskenta ilman jälkeenpäin tehtävää silottelua. Miettiessämme erilaisia ratkaisuja pisteen laskentaan huomasimme

Schaeferin ym. (2003) tarjoavan ratkaisun hyödyntää suoraan marssikuutioiden pisteen laskentaa artikkelissaan puhuessaan pintaverkkoalgoritmista ja se implementointiin ohjelmaan lupaavin tuloksin. Tässä vaiheessa tiesimme, että kyseessä ei ole suoraan Gibsonin (1998) esittämä pintaverkkoalgoritmi, joten tutkimme löytyykö kehitetystä algoritmista aikaisempaa tutkimusta tai tarkempaa nimitystä. Tätä kautta löysimme lopulta Lysenkon (2012), joka oli nimennyt algoritmin naiiviksi pintaverkoksi. Lysenko (2012) oli päätynyt kehittämään algoritmin melko samanlaista reittiä pitkin tutustumalla aluksi olemassa oleviin vaihtoehtoihin ja kehittämällä niiden pohjalta omia tarkoituksia palvelevan vaihtoehdon.

Naiivi pintaverkkoalgoritmi valittiin tutkielmassa marssikuutioiden vertailukohdaksi, koska se vastasi asettamiamme suorituskyvyllisiä vaatimuksia. Lisäksi siitä ei ollut paljon akateemista tutkimusta, joten se nähtiin ajankohtaisena sekä tutkimisen arvoisena pinnanmuodostusalgoritmina.

4.3 Maasto pinnanmuodostusalgoritmien sovellusalueena

Tutkielman yhteydessä kehitetyn Unity-projektin yhtenä tavoitteena oli tuottaa monipuolisen näköistä maastoa proseduraalisilla menetelmillä, mistä rakentui luvun 2 sisältö. Maaston korkeusdata tuotettiin käyttämällä kaksiulotteista kohinaa, josta palautuva arvo miellettiin maaston korkeudeksi syötteenä annetussa sijainnissa. Karteesisessa XYZ-koordinaatistossa, jossa Y on vertikaalinen akseli, koordinaatiston akselit X ja Z edustavat funktiolle syötettäviä koordinaatteja ja Y-akseli edustaa funktiosta palautuvia arvoja, jotka mielletään maaston korkeutena. Korkeusdata voidaan muuntaa volyymiksi seuraavasti:

$$f(x,y,z) = g(x,z) - y. \tag{4.1}$$

Kaavassa (4.1) funktiolle f syötettävä koordinaatti (x,y,z) on vokselin sijainti kolmiulotteisessa avaruudessa, jossa 3D-malli generoidaan. Samoin kohinafunktiosta g palautuvan arvon tulee olla skaalattuna maaston korkeutta edustavaksi luvuksi generoitavan 3D-mallin avaruudessa. Määrätyn raja-arvon ollessa 0, vokseleiden edustamat volyymit ovat erotuksen myötä positiivisia maanpinnan alapuolella ja negatiivisia yläpuolella, joten kaavan (4.1) tuloksena generoitava 3D-malli esittää tarkasti volyymiksi muunnettua maaston korkeusdataa.

Tutkielman yhteydessä kehitetyssä Unity-projektissa oli mahdollista generoida 3D-malleja harmaasävykuvista eli proseduraalisen generoinnin sijaan pystyttiin käyttämään esimerkiksi reaalimaailman korkeusdataa. Tämä onnistui yksinkertaisesti korvaamalla kaavan (4.1) kohinafunktio g toisella funktiolla, joka määritti maaston korkeuden kuvapisteen värin perusteella.

Minecraft-pelissä pelaajalle pystytään esittämään proseduraalisen generoinnin avulla lähes loputon maailma. Pelin kehittänyt Persson (2011) hyödyntää vokseleihin perustuvaa dataranetta, jolla maailma jaetaan säännöllisen kokoisiin lohkoihin. Lohkot ovat suoraan verrattavissa kolmiulotteisiin taulukoihin, ja vokselit ovat tämän taulukon alkioita. Unity-projektissa otettiin käyttöön samanlainen lohkoihin pohjautuva datarakenne, jossa lohkot ovat vastuussa yksittäisestä käyttäjälle esitettävän 3D-mallin generoinnista syöttämällä oman datansa pinnanmuodostusalgoritmille. Pelaajaa ympäröivien lohkojen data saadaan generoitua luvun 2 proseduraalisilla menetelmillä ja muuntamalla korkeusdata volyymiksi kaavalla (4.1). Tällä tavalla saadaan esitettyä Minecraft-pelin kaltainen maailma.

Vokseleihin perustuvia pinnanmuodostusalgoritmeja on käytetty kaupallisissa peleissä menestyksellisesti maaston esittämiseen, joten se oli luontainen kohdealue tutustua aluksi marssikuutioiden ja myöhemmin naiivin pintaverkkoalgoritmin toimintaan. Tutkielmalla ollessa vahvat pelinkehitykselliset taustat, pinnanmuodostusalgoritmeja sovellettiin myös tilanteissa, joissa käyttäjä voi itse vaikuttaa suoraan generoituihin 3D-malleihin. Tällainen tilanne esiteltiin esimerkiksi johdantoluvun kuvassa 1, jossa pelaaja rakensi siltaa kuilun ylittämistä varten. Toiminto tapahtuu yksinkertaisesti muokkaamalla lohkojen edustamien vokseleiden arvoja käyttäjän osoittamalta alueelta sekä päivittämällä niiden 3D-mallit.

Unity-projektin kehittämisen myötä on saatu selville, että vokseleihin perustuvia pinnanmuodostusalgoritmeja voi hyödyntää monenlaisissa sovelluksissa. Algoritmien käytössä on tärkeää ymmärtää ja hahmottaa vokseleihin perustuvia datarakenteita. Tutkittuja pinnanmuodostusalgoritmeja voidaan hyödyntää esimerkiksi graafisissa 3D-mallinnustyökaluissa, joissa käyttäjä pystyy itse suunnittelemaan ja tekemään 3D-malleja suhteellisen hallitusti. Samoin vokseleihin pohjautuvat pinnanmuodostusalgoritmit soveltuvat työkaluiksi erilaisten datojen analysointiin, sillä raja-arvoa säätelemällä voidaan paikantaa tiettyjä osia mallinnettavasta volyymistä.

4.4 Proseduraalisten menetelmien hyödyntäminen

Seuraavaksi käsitellään luvun 2 proseduraalisten menetelmien ohjelmatoteutusta. Unity-projektissa käytettiin korkeusdatan generointiin listauksessa 3 esitettyä luokkaa *FractalNoise*, joka yhdisti luvussa 2.2 esitetyt fraktaaliset menetelmät sekä luvussa 2.3 esitetyn syötteen häirinnän. Fraktaalisen kohinan parametreja säätelemällä pystyttiin tuottamaan monipuolisesti erilaisia maastotyyppejä, joten luokka täytti tehtävänsä luvun 2.4 laajempien maastojen generoinnista vastaavan luokan *MaskedFractalNoise* avustajana listauksessa 4.

```
public class FractalNoise
{
    public enum SumType { Basic, Billow, Ridge }
    SumType sum_type = SumType.Basic;

    int octaves = 4;
    float frequency = 0.01f;
    float lacunarity = 2f;
    float persistence = 0.5f;

    float warp_mult = 0f;
    Vector2 warp_offset_to_X = new Vector2(0f, 0f);
    Vector2 warp_offset_to_Y = new Vector2(0f, 0f);

    float erosion_mult = 0f;

    Simplex simplex;

    public FractalNoise(int seed)
    {
        simplex = new Simplex(seed);
    }

    public float Evaluate(Vector2 p)
    {
        return Fbm(Warp(p));
    }
```

```
private float Fbm(Vector2 p)
{
    float max_n_val = 0f;
    float n_sum = 0f;
    float freq = frequency;
    float amp = 1f;

    Vector2 dsum = new Vector2(0f, 0f);

    for (int i = 0; i < octaves; i++)
    {
        max_n_val += 1f * amp;

        Vector3 n_val = simplex.Evaluate(p.x * freq,  p.y * freq);
        float n_abs = n_val.x;
        if (n_abs < 0f) n_abs = -n_abs;

        dsum.x += n_val.y;
        dsum.y += n_val.z;

        float erosion = 1f + erosion_mult * Vector2.Dot(dsum, dsum);

        if (sum_type == SumType.Basic)
            n_sum += (n_val.x + 1f) * 0.5f * amp / erosion;

        else if (sum_type == SumType.Billow)
            n_sum += n_abs * amp / erosion;

        else if (sum_type == SumType.Ridge)
            n_sum += (1f - n_abs) * (1f - n_abs) * amp / erosion;

        freq *= lacunarity;
        amp *= persistence;
    }
    return n_sum / max_n_val;
}
```

```
    private Vector2 Warp(Vector2 p)
    {
        if (warp_mult <= 0f) return p;
        Vector2 interference = new Vector2(0f, 0f);
        interference.x = Fbm(p + warp_offset_to_X);
        interference.y = Fbm(p + warp_offset_to_Y);
        return p + warp_mult * interference;
    }
}
```

Listaus 3: Korkeusdatan tuottaminen fraktaalisen kohinan ja vääristymän avulla.

Evaluate-funktio toimii rajapintana, jonka kautta korkeusdatan generointi tapahtuu. *Evaluate*-funktio käyttää arvojen laskemiseen *Fbm*-funktiota. *Warp*-funktio vastaa syötteen häirinnästä, jonka voimakkuutta edustaa luokan parametri *warp_mult*. Häirinnässä käytetään Quilezin (2002) esityksen tapaan fraktaalista kohinaa syötteenä annetun pisteen *x* ja *y* koordinaateille. Parametreilla *warp_offset_to_X* ja *warp_offset_to_Y* vaikutetaan häiriötä generoivien funktiokutsujen evaluointipisteisiin.

Fbm-funktiota on laajennettu palauttamaan kohinafunktion arvon lisäksi sen osittaisderivaatat, jotka summataan muuttujaan *dsum*. Kohinan ja eroosion yhteisvaikutus saadaan, kun kohinan voimakkuus suhteutetaan tämän gradienttivektorin pituuden neliöön. Käytännössä lasketaan gradientin pistetulo itsensä kanssa ja tulos summataan jakajaan, jolla ei ole oletusarvoisesti vaikutusta lopputulokseen. Jos laskettu eroosio on suurempaa kuin yksi, se pienentää lopullista summattavaa korkeusarvoa. Eroosiomainen efekti syntyy tämän vaikutuksesta.

```
public class MaskedFractalNoise
{
    public List<Layer> layers = new List<Layer>();
    public List<Mask> masks = new List<Mask>();

    public class Layer
    {
        public FractalNoise noise;
        public List<int> mask_pointers;
```

```
        public Layer(FractalNoise noise)
        {
            this.noise = noise;
            mask_pointers = new List<int>();
        }
    }

    public class Mask
    {
        public FractalNoise noise;
        public float last_value;

        public Mask(FractalNoise noise)
        {
            this.noise = noise;
        }
    }

    public float Evaluate(Vector2 p)
    {
        for (int i = 0; i < masks.Count; i++)
            masks[i].last_value = masks[i].noise.Evaluate(p);

        float sum = 0f;
        for (int i = 0; i < layers.Count; i++)
        {
            float val = layers[i].noise.Evaluate(p);
            for (int j = 0; j < layers[i].mask_pointers.Count; j++)
                val *= masks[layers[i].mask_pointers[j]].last_value;

            sum += val;
        }
        return sum / layers.Count;
    }
}
```

Listaus 4: Laajojen maastojen korkeusdatan tuottaminen maskien avulla.

MaskedFractalNoise-luokassa korkeusdataa generoivat kerrokset ja maskikohinat ovat määritelty erillisiin listoihin, joille on annettu säilytettäväksi niitä vastaavien luokkien *Layer* ja *Mask* olioita. Molemmat luokat käyttävät fraktaalista kohinaa arvojen generoimiseen, mutta ne eroavat toisistaan siinä, että *Layer*-tyyppiset oliot sisältävät listan osoittimista sen käyttämiin *Mask*-tyypin olioihin. Kerrokset edustavat maastotyyppejä ja maskit niiden painoarvoja, joiden avulla säädellään eri maastotyyppien esiintymistä.

Evaluate-funktion suorittaminen tapahtuu läpikäymällä ensiksi kaikki luokalle määritetyt maskikohinat sekä tallentamalla niihin fraktaalisella kohinalla tuotetut kertoimet. Tämän jälkeen lopullinen funktiosta palautuva arvo lasketaan korkeutta generoivien kerroksien arvojen summana, jossa summattavaa kerroksen arvoa kerrotaan sille osoitetuilla maskien kertoimilla. Tässä yhteydessä on tärkeää huomioida, että kohinan oletetaan palauttavan arvoja väliltä $[-1, 1]$, mutta Unity-projektissa fraktaalinen kohina muunnettiin palauttamaan luvun arvoväliltä $[0, 1]$. Tämä antoi täsmällisen kontrollin arvojen jatkokäsittelyä varten, sillä arvoväli on aina vakio kohinan parametreista riippumatta. Samaa käytäntöä noudatetaan laajempien maastojen generoinnissa, missä muuntaminen arvovälille $[0, 1]$ tapahtuu jakamalla laskettu summa sen korkeutta generoivien kerroksien lukumäärällä.

5 Algoritmien suorituskyvyn mittaus

Tutkimuksessa verrattiin marssikuutiot- ja naiivin pintaverkkoalgoritmin suorituskykyä keskenään. Tavoitteena oli selvittää, kumpi algoritmeista on nopeampi mittaamalla 3D-mallien generointiin kulunutta aikaa millisekunneissa. Samalla kartoitettiin algoritmeilla muodostettujen 3D-mallien eroja kolmioiden ja pisteiden lukumäärissä. Tutkimus vahvistaa Lysenkon (2012) saamia tuloksia ja tarjoaa lisätietoa tutkittavien algoritmien välisistä eroista.

Ennen tutkimuksen toteuttamista olimme perehtyneet algoritmien teoreettiseen taustaan sekä tehneet niistä ohjelmatoteutukset, joten osasimme ennakoida mittaustuloksia. Arvioimme, että samankaltaisen pisteen laskennan ja geometrisen duaalisuuden seurauksena naiivi pintaverkko on suoritusajaltaan nopeampi ja tämän takia tuottaa myös vähemmän pisteitä generoitavaan 3D-malliin. Kolmioiden lukumäärät tulisivat olemaan algoritmien välillä melko samanlaiset, koska 3D-mallit generoidaan samasta testiaineistosta, jolloin algoritmien tuottamat 3D-mallit luonnostaan muistuttavat toisiaan.

Luvussa 5.1 käsitellään suunnittelutieteellisen tutkimuksen ohjesääntöjen toteutumista. Luvussa 5.2 esitetään tutkimuksen kulku vaiheittain ja käytetty tutkimusvälineistö. Luvussa 5.3 esitetään tutkimustulokset ja niistä saatavia johtopäätöksiä. Luvussa 5.4 esitetään tutkimuksen tuloksien yhteenveto.

5.1 Suunnittelutieteellisen tutkimuksen ohjesääntöjen toteutuminen

Ennen tutkimuksen esittelyä on tärkeää arvioida suunnittelutieteellisen viitekehyksen ohjesääntöjen toteutumista tutkimuksen luotettavuuden kannalta. Seuraavaksi selitetään kuinka viitekehyksen sisältämiä ohjesääntöjä on noudatettu. Oheisessa taulukossa 1 on yhteenveto viitekehyksen ohjesäännöistä sekä niiden luokitusjärjestelmän mukaiset numerot. Seuraavaksi käsiteltävien ohjesääntöjen esitysjärjestys noudattaa samaa järjestystä kuin Hevnerin ym. (2004) antamissa ohjesääntöjen soveltamista koskevissa esimerkeissä.

Ohjesäännön kuvaus	**nro**
Tutkimuksen relevanssi	2
Tutkimuksen täsmällisyys	5
Suunnittelu etsintäprosessina	6
Suunnittelu artefaktina	1
Suunnittelun evaluointi	3
Tutkimuksen kontribuutiot	4
Tutkimusviestintä	7

Taulukko 1: Suunnittelutieteellisen tutkimuksen ohjesäännöt.

Suunnittelutieteellisen tutkimuksen relevanssi määräytyy kohdealueesta, jossa tutkimuksen tuloksia olisi tarkoitus soveltaa. Tavoitteena on usein kehittää artefakti, joka ratkaisee jonkin kohdealueen suunnitteluongelman. Artefaktin tulee siten tuottaa jonkinlaista hyötyarvoa kohdealueeseen, jotta tutkimus olisi relevantti. (Hevner ym. 2004) Kehitettävällä artefaktilla tarkoitetaan naiivia pintaverkkoalgoritmia, jonka nopeampi suoritusaika tarjoaa hyötyarvoa vertailukohteen marssikuutiot-algoritmille. Tutkielman tapauksessa kehitetty artefakti parantaa olemassa olevan suunnitteluratkaisun suorituskykyä.

Hevnerin ym. (2004) mukaan tutkimuksen teoreettinen viitekehys ja sovellettava tutkimusmenetelmä vaikuttavat yleisesti tutkimuksen täsmällisyyteen, jota edellytetään sekä artefaktin kehittämisessä että arvioinnissa. Täsmällisyyteen vaikuttaa olennaisesti se, kuinka tarkasti toteutuksen yksityiskohdat kuvataan, mutta liiallisen täsmällisyyden tavoittelu saattaa kuitenkin heikentää tutkimuksen relevanssia. Tämän takia tutkimuksessa on tärkeää löytää sopiva tasapaino täsmällisyyden ja relevanssin välillä.

Marssikuutiot-algoritmi kehitettiin täsmällisen algoritmikuvauksen pohjalta ja naiivi pintaverkkoalgoritmi on iteratiivisten kehitysvaiheiden tulos. Artefaktin arvioinnissa käytettävät arviointikriteerit tukeutuvat usein kehitettävän artefaktin mitattavissa oleviin ominaisuuksiin. Artefaktien arvioinnissa tehtiin suorituskyvyn mittaukset siten, että mittauskertojen välillä vaihdeltiin ainoastaan mittauksen kohteena ollutta pinnanmuodostusalgoritmia. Toisin sanoen mittauksissa havaitut erot selittyvät algoritmien ominaisuuksilla.

Suunnittelutiede on iteratiivinen etsintäprosessi, jossa etsitään suunnitteluongelmaan tehokasta ratkaisua. Usein voidaan tyytyä riittävän hyvään ratkaisuun vertaamalla sitä vaihtoehtoisiin suunnitteluratkaisuihin, jotka ratkaisevat samankaltaisen suunnitteluongelman (Hevner ym. 2004). Tutkielman luvussa 4.2 kuvattiin naiivin pintaverkkoalgoritmin etsintäprosessia, jonka tehokkuutta verrattiin vastaavalla syöte- ja tulosjoukolla toimivaan marssikuutiotalgoritmiin.

Suunnittelutieteellisessä tutkimuksessa tulee kehittää toteutettavissa oleva artefakti, jonka kehittämiseen hyödynnetään suunnittelutieteellisiä menetelmiä (Hevner ym. 2004). Toteuttamalla täsmällisesti määritetyn artefaktin osoitetaan, että suunniteltu artefakti voidaan toteuttaa tutkimuksessa kuvatulla tavalla. Hevner ym. (2004) perustelevat ensimmäisen prototyypin käyttöönottoa välttämättömäksi kehitysvaiheeksi, sillä toteutettavuuden todentamisen jälkeen prototyyppiä voidaan parantaa merkittävästi jatkotutkimuksessa. Tutkielman luvussa 3.2 kuvataan kehitettävän artefaktin konstruointi sisältäen täsmällisen algoritmikuvauksen.

Hevnerin ym. (2004) mukaan suunnitteluartefaktin laadunvarmistus tulee (ja voidaan) osoittaa täsmällisesti käyttäen siihen soveltuvaa evaluointimenetelmää. Evaluointi edellyttää tarkoituksenmukaisten muuttujien määrittelyä sekä aineiston keräämistä analysointia varten. Evaluointimenetelmä tulee siis valita asianmukaisesti vastaamaan suunniteltua artefaktia sekä siitä mitattavissa olevia muuttujia (Hevner ym. 2004). Artefaktia voidaan evaluoida relevanttien laatuominaisuuksien, kuten suorituskyvyn perusteella. Suunnitteluratkaisun voidaan katsoa olevan riittävän tehokas, jos evaluoitava suunnitteluartefakti täyttää sille asetetut vaatimukset ja rajoitteet. Tutkielmassa suunnitteluartefaktia evaluoitiin kokeellisesti simuloimalla siten, että pinnanmuodostusalgoritmeja suoritettiin simpleksikohinalla tuotetulla keinotekoisella datalla Hevnerin ym. (2004) periaatteita noudattaen.

Suunnittelutieteellisen tutkimuksen tulee myötävaikuttaa tieteenalan kehittymiseen ja useimmiten kontribuutio on itse artefakti, jonka avulla esimerkiksi sovelletaan olemassa olevaa informaatiota uudella tavalla. Tutkimus voi myös laajentaa artefaktin teoriapohjaa tai kehittää tutkimusmenetelmiä vaikkapa esittelemällä mitattavia tai mittauksista johdettavia muuttujia. Suunnittelutieteellisen tutkimuksen tulee kuitenkin ennen kaikkea tarjota selkeä kontribuutio ratkaisemalla jokin liiketoiminnan kannalta merkittävä suunnitteluongelma (Hevner ym. 2004). Tutkielman avulla voidaan korvata esimerkiksi peliteollisuudessa yleistynyt

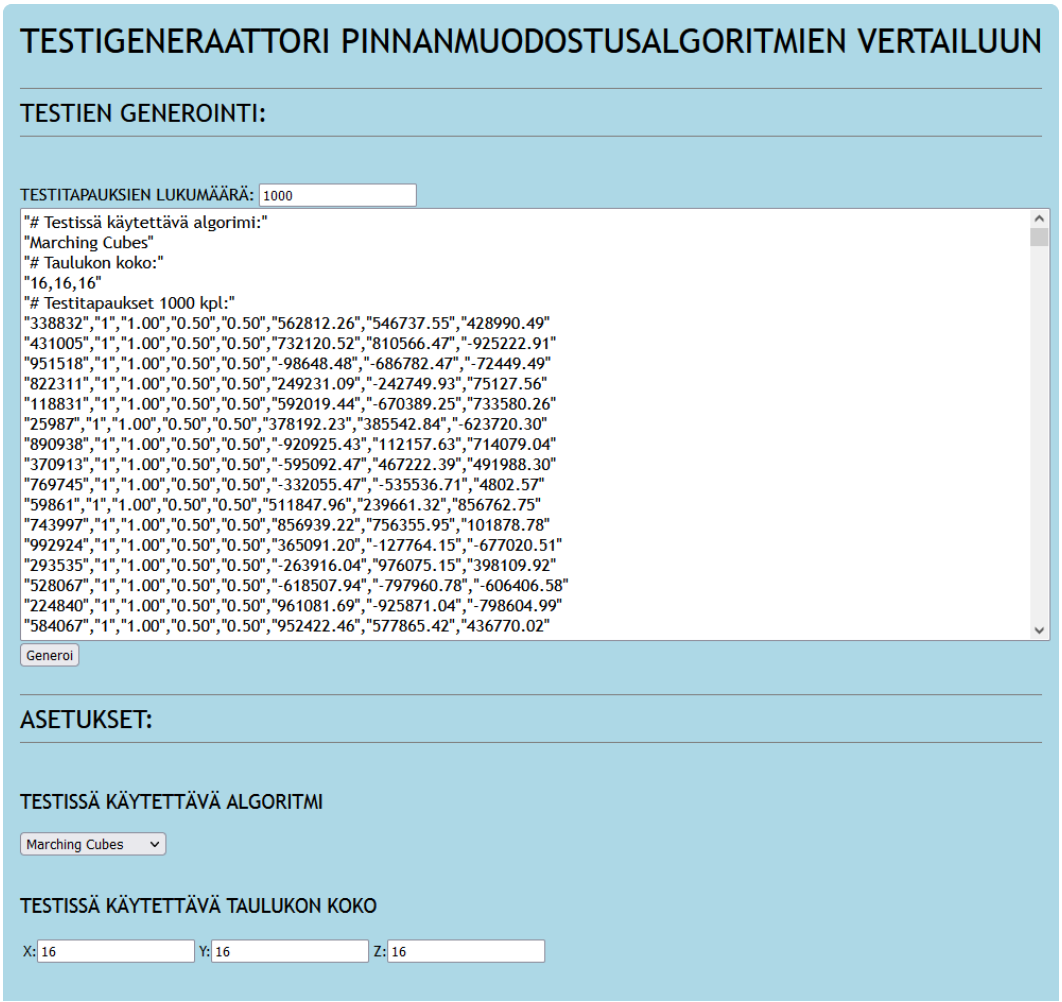
marssikuutiot-algoritmi tehokkaammalla pinnanmuodostusalgoritmilla. Tutkimuksen teoriatausta ja kehitetty artefakti itse laajentavat osaltaan pinnanmuodostusalgoritmien teoriapohjaa jatkotutkimusta varten. Tutkimuksen läpiviennin kuvauksella tuodaan suomenkielistä kontribuutiota suunnittelutieteellisen tutkimuksen ja konstruktiivisen tutkimusotteen hyödyntämiseen tulevissa tietokonegrafiikan pro gradu -tutkielmissa.

Suunnittelutieteellisen tutkimusraportissa artefakti tulee kuvata riittävän täsmällisesti, jotta se voidaan ottaa käyttöön tuotantoympäristössä (Hevner ym. 2004). Lisäksi artefaktin konstruoinnin ja evaluoinnin vaiheet tulee kuvata ymmärrettävästi, jotta tutkimus olisi toistettavissa esimerkiksi jatkotutkimuksen tapauksessa. Artefaktin ominaisuuksien täsmällisen kuvauksen lisäksi tulee esitellä, kuinka artefakti voidaan valjastaa hyötykäyttöön liiketoimintaympäristössä, jotta artefaktin mahdollisen käyttöönoton kannattavuutta voidaan arvioida. Hevnerin ym. (2004) mukaan artefaktin yksityiskohtainen esittämiseen soveltuvat tiiviit ja hyvin jäsennetyt liitteet, mutta tutkielmassa valittiin liitetiedostojen sijaan tasapaino esitystavan relevanssin ja täsmällisyyden väliltä. Tällä tavoin tutkielmassa otetaan huomioon molemmat Hevnerin ym. (2004) esittämät kohderyhmät.

5.2 Pinnanmuodostusalgoritmien vertailututkimuksen toteuttaminen

Tutkimusta varten kehitettiin erillinen tietokoneohjelma, jonka tehtävänä oli suorittaa vertailtaville pinnanmuodostusalgoritmeille määrätyt testitapaukset ja tallentaa mittaustulokset taulukkolaskentaohjelmille soveltuvana CSV-tiedostona. Luvussa kuvataan tarkemmin tutkimuksessa käytetyn tietokoneen laitekokoonpano sekä, kuinka kyseistä Unity-pelimoottorilla kehitettyä tietokoneohjelmaa hyödynnettiin tutkimuksen läpiviennissä.

Tutkimusaineisto koostui testausohjelman suoritusohjeita kuvaavista tekstitiedostoista, jotka määräsivät kyseisellä mittauskerralla käytettävän algoritmin, taulukon koon ja suoritettavat testitapaukset. Testitapauksissa määritettiin luvussa 2.2 kuvatun fBm-funktion parametrit, joilla generoitiin volymetrinen data taulukkoon pinnanmuodostusta varten. Oheinen kuva 13 havainnollistaa testausohjelman käyttämän tekstitiedoston rakennetta, jonka testitapauksissa fBm-funktiolle määrättiin kohinan siemenluku, oktaavit, taajuus, lakunaarisuus, pysyvyys ja siirtymä.



Kuva 13: Kuvakaappaus testigeneraattorista.

Testiaineistoa haluttiin saada kattavasti, joten suuren aineistomäärän keräämisessä oli luontevaa hyödyntää proseduraalisen generoinnin menetelmiä. Käsinkirjoittaminen olisi ollut äärimmäisen työlästä ja virhealtista, joten testitapausten laatiminen päätettiin toteuttaa tietokoneavusteisesti. Sitä varten kehitettiin kuvassa 13 esitetty testigeneraattori, jonka käyttöliittymän syötekenttien avulla voitiin määrittää fBm-funktion parametrien ylä- ja alarajat. Testigeneraattori huolehti yksittäisten testitapausten parametrien satunnaistamisesta kyseiselle arvovälille.

Testien suorittamista varten koostettiin yhteensä tuhat satunnaistettua testitapausta sisältävä CSV-tiedosto. Kohinan arvovaihtelu pidettiin tarkoituksenmukaisesti korkeataajuisena, joten kohinan käyttäytymiseen vaikuttavista parametreista oktaaveja, taajuutta, lakunaarisuutta ja pysyvyyttä ei satunnaistettu siemenluvun ja siirtymän tavoin. Testitapauksiin lisättiin monipuolisuutta muuttamalla vakioina pidettäviä parametreja kahdensadan testitapauksen välein. Oheisessa taulukossa 2 on yhteenveto testeissä käytetyistä fBm-funktion parametreista.

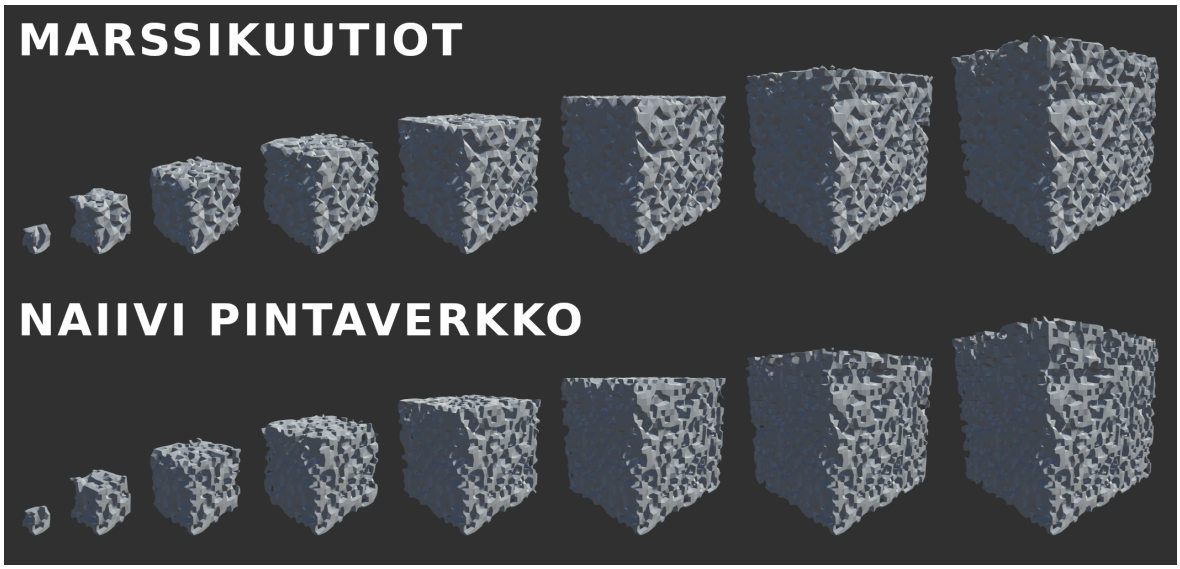
siemenluku	**oktaavit**	**taajuus**	**lakunaarisuus**	**pysyvyys**	**siirtymät**
$[0, 10^6]$	1	1,0	1,0	1,0	$[-10^6, 10^6]$
$[0, 10^6]$	4	1,0	2,0	0,5	$[-10^6, 10^6]$
$[0, 10^6]$	8	0,5	2,0	0,5	$[-10^6, 10^6]$
$[0, 10^6]$	16	0,1	1,5	1,4	$[-10^6, 10^6]$
$[0, 10^6]$	4	2,0	0,75	1,0	$[-10^6, 10^6]$

Taulukko 2: Parametrien vaihteluväliä muutettiin kahdensadan testitapauksen välein.

Pinnanmuodostusalgoritmien suoritusnopeudet määräytyvät käsiteltävän taulukon koosta ja generoitavien kolmioiden lukumäärästä, joten algoritmien suorituskyvyn vertailussa oli tarkoituksenmukaista muodostaa 3D-malleille mahdollisimman paljon pintaa. Generoitava pinnan määrä on riippuvainen volyymin vaihtelusta raja-arvon suhteen, joten runsaan pinnan määrän generoimiseksi testitapauksissa käytettiin korkeataajuista kohinaa. Tällä varmistettiin, että tulokset antavat realistisen kuvan algoritmien taulukkokohtaisesta suorituskyvystä niiden koon suhteen. Lisätutkimuksena selvitettiin algoritmien suoritusnopeuden skaalautuvuutta esitettävän pinnan määrälle tilanteessa, jossa pinnanmuodostusalgoritmit eivät generoineet volyymin pohjalta ainuttakaan kolmioita.

Testiaineiston sisältävää CSV-tiedostoa käytettiin kahdeksalle eri kolmiulotteisen taulukon koolle niin, että kokoa kasvatettiin porrastetusti neljällä: 4^3, 8^3, ...ja 32^3. Mittausten aikana generoitiin yhteensä kuusitoistatuhatta 3D-mallia. Kuvan 14 esimerkki havainnollistaa testausohjelman generoimien 3D-mallien tiheää volyymivaihtelua. Siinä ylärivin mallit generoitiin marssikuutiot-algoritmilla ja alarivin mallit vuorostaan naiivilla pintaverkkoalgoritmilla. Lisätutkimuksena tehdyissä mittauksissa testitapauksia generoitiin erikseen viidellä eri taulukon koolla: 16^3, 32^3, ...ja 80^3, missä ei tarkoituksella muodostettu esitettävää pin-

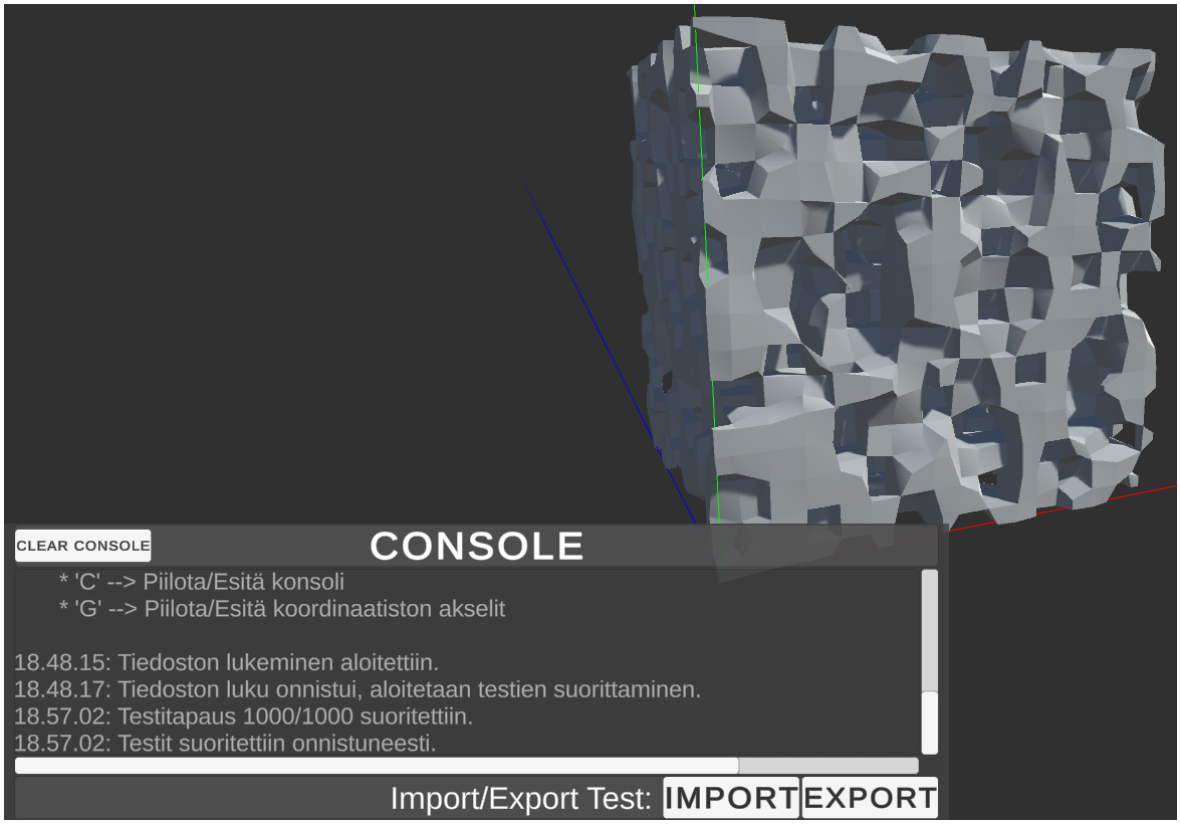
taa kymmenelle tuhannelle 3D-mallille. Tästä syystä lisätutkimuksen tuloksista taulukoitiin pelkästään algoritmien suoritusajat.



Kuva 14: Esimerkit testausohjelman generoimista 3D-malleista.

Testauskerran suorittaminen aloitettiin käynnistämällä testausohjelma uudelleen ja syöttämällä sille testitapaukset sisältävän CSV-tiedoston. Tiedoston alussa kuvattiin testauskerralla käytettävä algoritmi ja kolmiulotteisen taulukon koko, jonka jälkeen ohjelma muunsi testitapaukset olioiksi rivi kerrallaan. Tiedoston lukemisen valmistuttua aloitettiin testitapausten suorittaminen. Olioksi muunnetut testitapaukset käsiteltiin yksi kerrallaan, jonka yhteydessä tyhjennettiin edellisellä kierroksella generoidun 3D-mallin data. Seuraavaksi täytettiin 3D-mallin generoinnissa käytettävä taulukko kolmiulotteisella simpleksikohinalla. Testiasetelman alustavien toimenpiteiden jälkeen oltiin valmiita suorittamaan 3D-mallin generointi ja siihen liittyvät mittaukset. 3D-mallin generoinnin päätteeksi mittaustulokset tallennettiin samaan testitapauksen määrittävään olioon. Taulukon edustamien vokseleiden ulkopuolelle muodostuvat kuutiot otettiin huomioon, jotta kaikkien generoitavien 3D-mallien pinta oli täysin kiinteä. Tällaisessa tilanteessa taulukon ulkopuolisia arvoja käsiteltiin ilmana.

Algoritmeille syötettävän volyymin generointi kohinalla suoritettiin erillisessä säikeessä, jotta pääohjelman puolella ei tehtäisi 3D-mallien generoinnin lisäksi muita laskennallisesti vaativia operaatioita. Tällä tavalla testausohjelmassa estettiin volyymin generointia aiheuttamasta pääohjelmaa hidastavia piikkejä juuri ennen 3D-mallin generointia.



Kuva 15: Kuvakaappaus testausohjelmasta.

Testitapausten suorittamisen jälkeen mitatut tulokset olivat saatavissa CSV-formaatissa sisältäen tiedon 3D-mallien generoitiin kuluneesta ajasta millisekunteina sekä generoidun 3D-mallin kolmioiden ja pisteiden määrät. Yhden testitiedoston käsittely testausohjelmalla kesti keskimäärin kymmenen minuuttia, sillä 3D-mallien generoinnin väliin sisältyi puolen sekunnin viive. Testausohjelman tulostetta näytettiin ruudulla puolen sekunnin ajan, jotta 3D-mallien generointia voitaisiin seurata valvotusti reaaliajassa. Jokaisen testauskerran suorittamisen jälkeen tulokset kirjoitettiin CSV-tiedostoon, jotka vietiin jatkokäsittelyä varten taulukkolaskentaohjelmaan. Mittaustuloksista laskettiin algoritmin keskimääräinen suoriutuminen, mitä käytettiin luvun 5.3 tuloksia esittävissä kuvissa.

Tutkimuksen päätarkoitus oli verrata algoritmien suoriutumista keskenään, joten testit suoritettiin yhdellä taulukossa 3 kuvatulla laitekokoonpanolla. Usealla eri laitekokoonpanolla tehdyt testit antavat tarkempaa tietoa eri komponenttien laskennallisesta suorituskyvystä eikä sillä olisi saavutettu tutkimuksen kannalta kiinnostavaa lisätietoa algoritmien vertailuun.

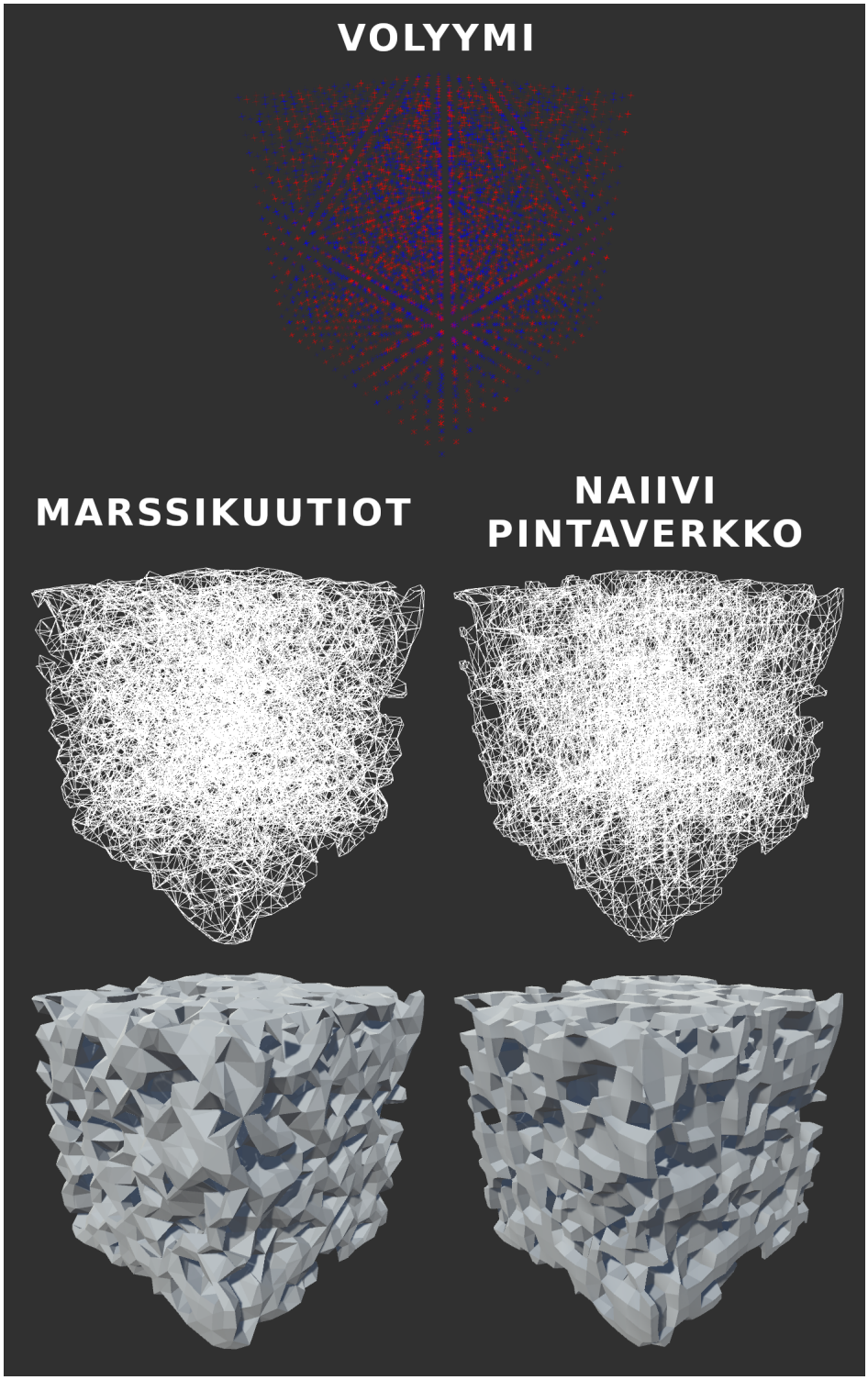
Komponentti	Tarkemmat tiedot
Prosessori	Intel(R) Core(TM) i5-4460s CPU @ 2.90 GHz
Näytönohjain	NVIDIA GeForce GTX 1070 8 GB
Keskusmuisti	16 GB RAM

Taulukko 3: Tutkimuslaitteiston laitekokoonpanon tarkemmat tiedot.

5.3 Kokeellisten testien numeeriset tulokset ja niiden esittäminen

Luvussa verrataan marssikuutiot-algoritmin ja naiivin pintaverkkoalgoritmin suorituskykyä numeerisesti mitattavissa olevilla suureilla. Tutkimustulokset on eritelty alalukuihin mittaustulosten mukaisessa järjestyksessä niin, että ensin käsitellään suoritusaika, sitten kolmioiden ja lopuksi pisteiden lukumäärä. Mittaustulokset esitetään pylväsdiagrammeina, joissa vaakasuora akseli kuvaa kolmiulotteisen taulukon kokoa syvyys-, rivi- ja sarakesuunnissa sekä pystysuora akseli kuvaa yhtä kolmesta mittaustuloksesta. Tuloksia esittävissä kuvissa lyhenteistä NSN:llä tarkoitetaan naiivia pintaverkkoalgoritmia ja MC:llä marssikuutiot-algoritmia.

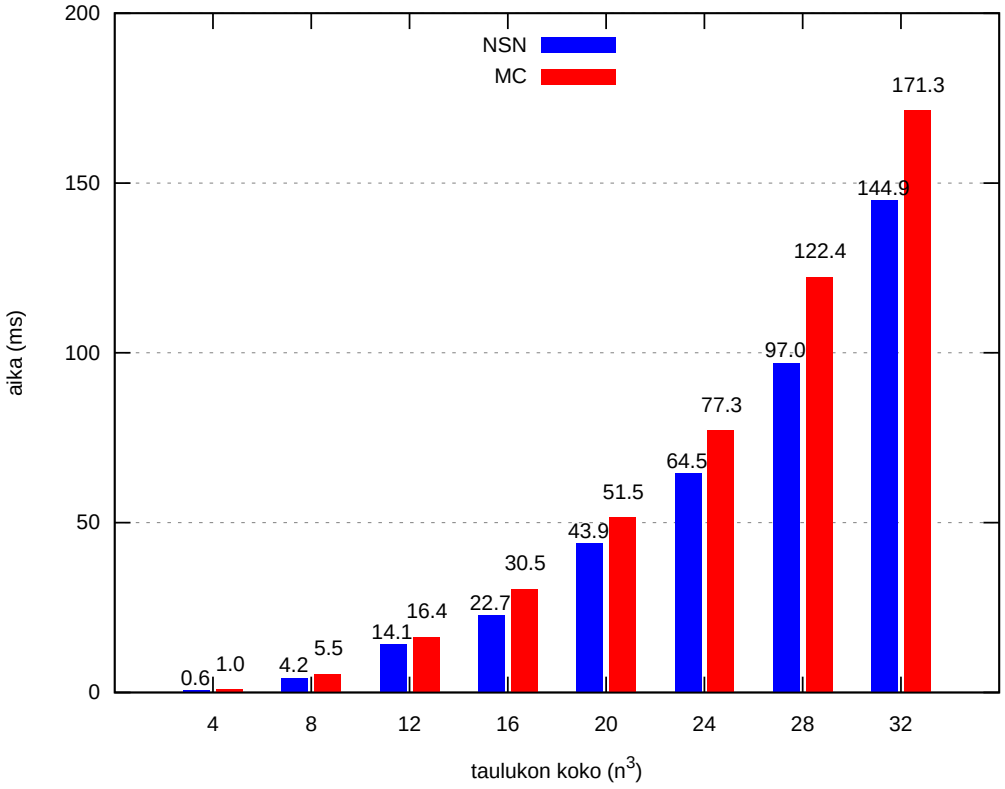
Kuvassa 16 on havainnollistava esimerkki kohinalla generoidusta volyymistä ja sen pohjalta tuotetuista 3D-malleista vertailussa käytettävillä algoritmeilla. Ensimmäisenä kuvassa visualisoidaan korkeataajuisella kohinalla generoitu volyymi, jossa pintaa muodostavia arvoja on merkitty punaisella ja ilmana miellettäviä arvoja sinisellä värillä. Keskimmäisenä kuvassa on esitetty valkoisella värillä rautalankamallit volyymin pohjalta generoiduista 3D-malleista, joista erottuu tarkemmin kolmioiden runsaslukuisuus. Lopuksi on esitetty testausohjelman tuottamat 3D-mallit, joissa pinta on teksturoitu harmaalla värillä.



Kuva 16: Esimerkki volyymin pohjalta generoitavasta 3D-mallista.

5.3.1 Suoritusaika

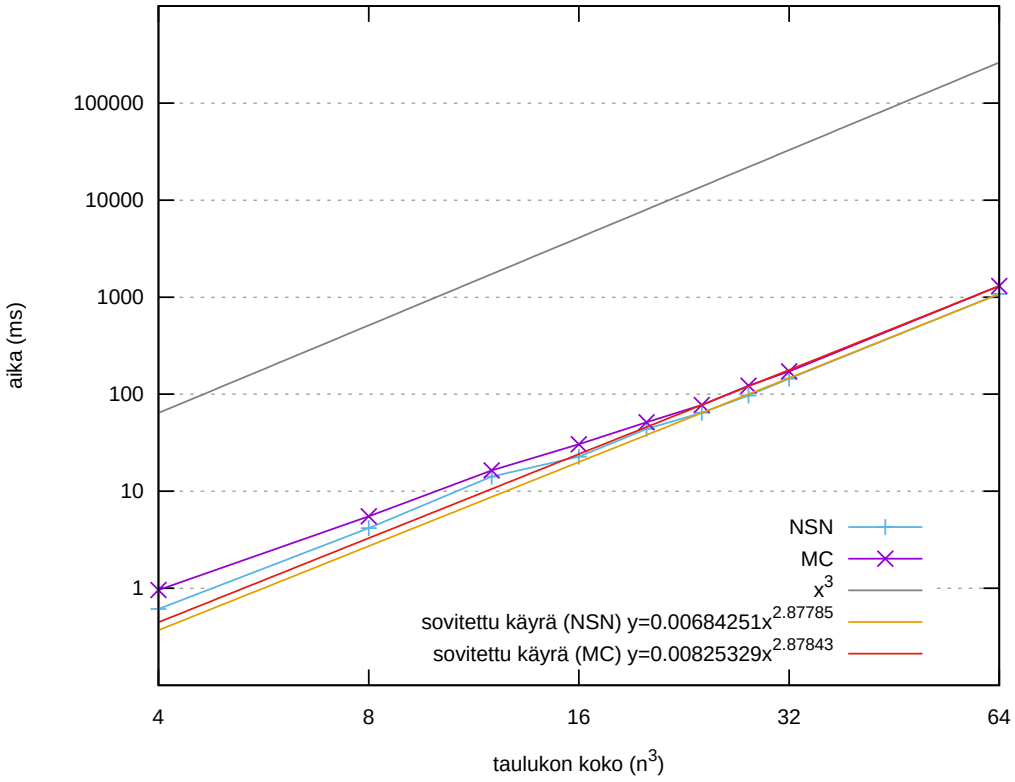
Algoritmien suoritusaikaa mitattiin suunnitellusti taulukon 2 kuvaamalla aineistolla. Algoritmien suoritusaika on sidonnainen käytettävän taulukon kokoon ja generoitavien kolmioiden lukumäärään, joten mittausten aikana 3D-malleille generoitiin esitettävää pintaa mahdollisimman paljon taulukon koon suhteen. Suoritetuissa mittauksissa taulukoiden dimensioiden kokoa n kasvatettiin lineaarisesti, josta alkioiden lukumäärä on laskettavissa kaavalla n^3 käsiteltävien taulukoiden ollessa kolmiulotteisia. Kuvaan 17 on koostettu mittaustulokset tilanteesta, jossa algoritmia on kuormitettu paljon.



Kuva 17: Suoritusaika millisekunneissa.

Mittaustuloksia tarkastelemalla voimme havaita, että algoritmit ovat suoritusajoiltaan tasavertaiset, mutta naiivi pintaverkkoalgoritmi käsittelee syötettävät tiedot lievästi nopeammin kuin marssikuutiot-algoritmi. Erot suoritusajoissa ovat suuremmat taulukkojen koon kasvaessa. Siihen vaikuttaa erityisesti vokseleiden muodostamiin kuutioihin kohdistuvien operaatioiden suurempi määrä, johon palataan vielä myöhemmin. Molempien algoritmien suo-

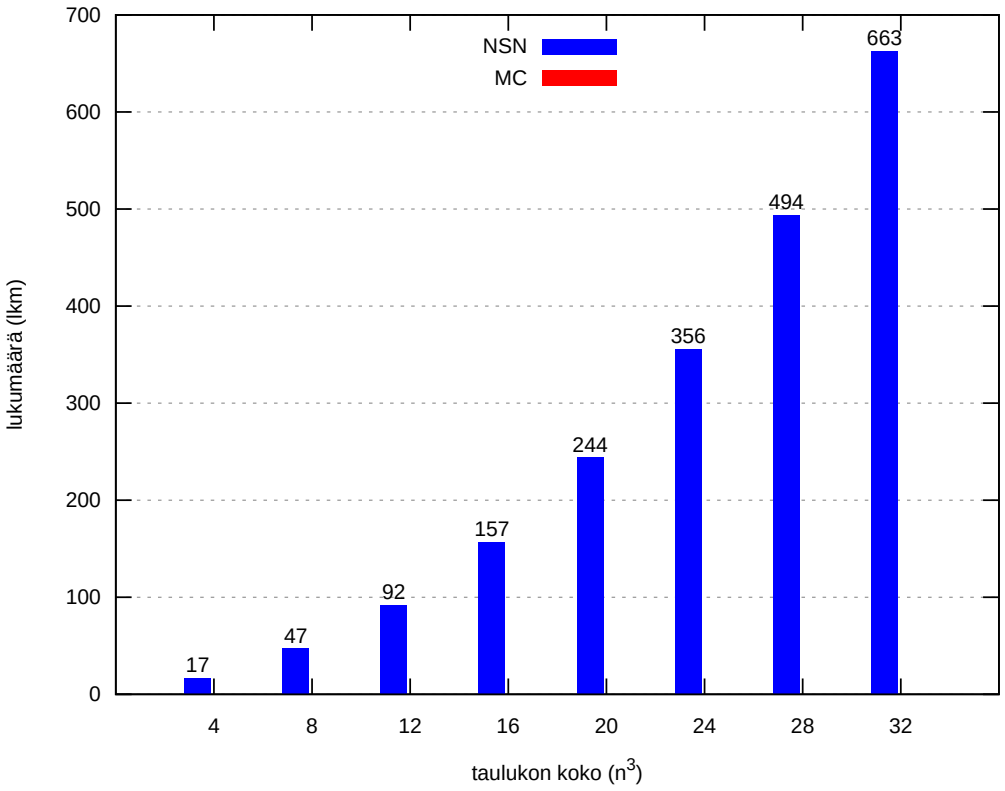
ritusajan riippuvuus taulukon kokoon ja generoitavien kolmioiden lukumäärään erottuu selvästi mittaustuloksista niiden noudattaessa taulukon alkioiden määrän kuutiollista kasvua. Tämä lineaarinen yhteys ilmenee vielä selkeämmin kuvasta 18, joka esittää samat tulokset kuin kuvassa 17, mutta logaritmisella asteikolla. Sovitetuista käyristä ja kuvaajasta x^3 pystytään havaitsemaan molempien algoritmien aikavaativuudeksi $O(n^3)$. Se määräytyy syötteenä annetun taulukon alkioiden lukumäärän perusteella.



Kuva 18: Suoritusaika millisekunneissa logaritmisella asteikolla.

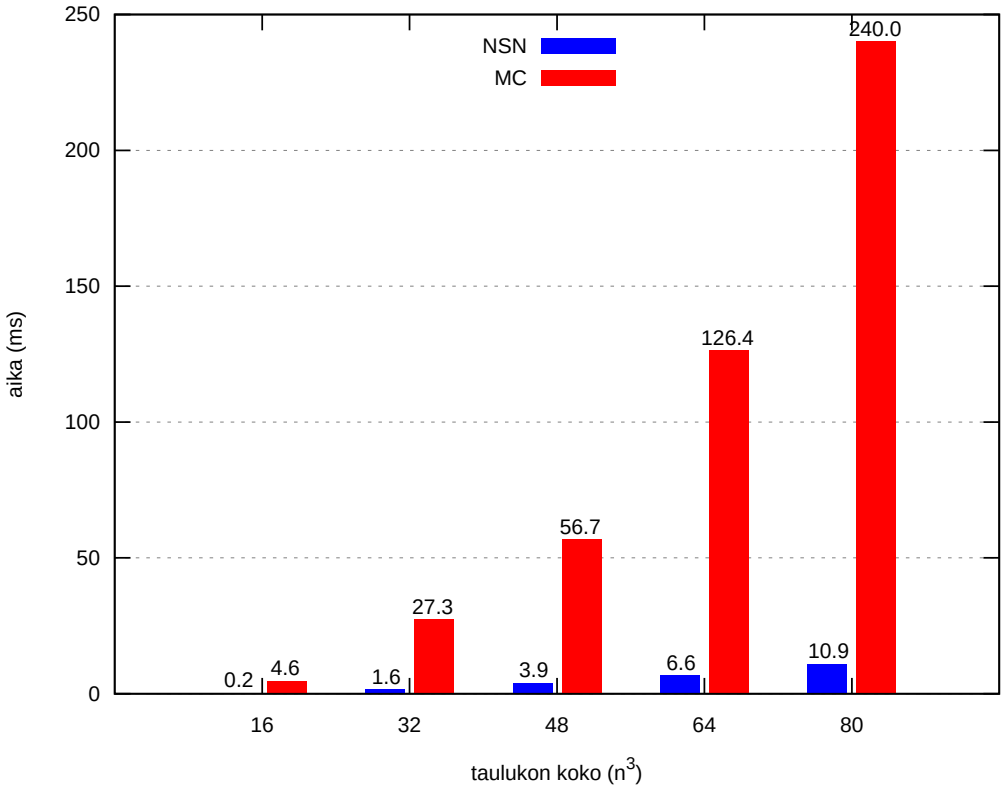
Vertailtavissa algoritmeissa vokseleiden muodostamien kuutioiden käsittely on osittain yhdistävä piirre, jolla voi selittää niiden suorituskyvyllisiä eroja. Marssikuutiot-algoritmin tapauksessa kuutioiden käsittelyllä tarkoitetaan konfiguraation määritystä ja sen perusteella tehtävää kolmiointitaulun silmukointia. Kaikki vokseleiden muodostamat kuutiot käsitellään ja niiden lukumäärä on laskettavissa kaavalla n^{3+1}. Erilaisesta pinnanmuodostuksesta johtuen naiivi pintaverkkoalgoritmi käsittelee vain sellaiset kuutiot, joiden sisäpuolelle lasketaan nelikulmion piste läpikäymällä kaikki kuution särmät sekä ottamalla keskiarvo niille

lasketuista pisteistä. Laskennallisena operaationa kuutioiden käsittely on algoritmien välillä melko samanlainen, mikä tarkoittaa marssikuutioiden tapauksessa 8–13 silmukan kierrosta ja naiivin pintaverkon tapauksessa aina 12 silmukan kierrosta. Naiivi pintaverkko saa edun marssikuutioihin nähden siitä, että se kykenee jättämään osan kuutioista käsittelemättä riippuen pinnanmuodostuksen ehdoista. Kuva 19 esittää keskimääräiset lukumäärät käsittelemättä jätetyistä kuutioista.



Kuva 19: Käsittelemättä jätettyjen kuutioiden lukumäärä.

Naiivi pintaverkkoalgoritmi suorittaa kasvavassa määrin vähemmän kuutioiden käsittelyä kuin marssikuutiot-algoritmi, sillä käsittelemättä jätettyjen kuutioiden määrä kasvaa taulukon koon mukana. Tämä selittää hyvin algoritmien tasavertaisuutta pienikokoisilla taulukoilla sekä erojen kasvua suuremmilla taulukoilla. Koska algoritmien suoritusaika on riippuvainen taulukon koon lisäksi generoitavien kolmioiden lukumäärästä, suoritettiin lisämittaukset tilanteessa, jossa 3D-malleille ei muodostunut datan pohjalta esitettävää pintaa lainkaan. Kuvassa 20 on esitetty lisämittauksista saadut tulokset.



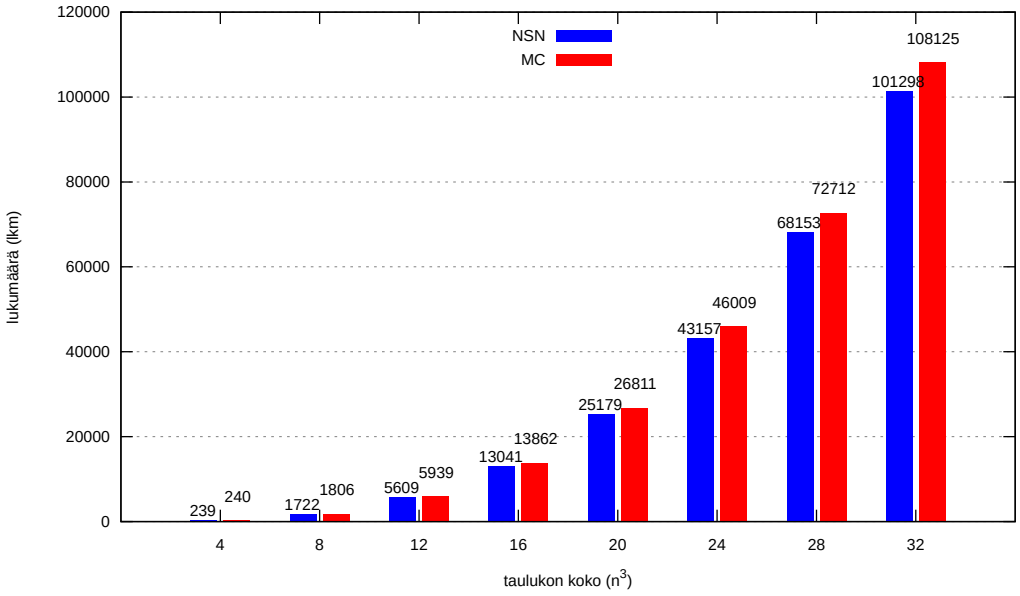
Kuva 20: Suoritusaika millisekunneissa ilman 3D-malleille muodostuvaa pintaa.

Molempien algoritmien suoritusajat nopeutuivat merkittävästi kuvan 17 esittämiin tuloksiin verrattuna, joten generoitavien kolmioiden lukumäärä vaikuttaa suuresti kummankin algoritmin suoritusaikaan. Toisaalta suoritusajat eroavat algoritmien välilläkin valtavasti, ja naiivi pintaverkkoalgoritmi skaalautuu huomattavasti paremmin generoitavien kolmioiden lukumäärän vähentyessä. Tämä johtuu erilaisesta taulukon silmukoinnista, jossa marssikuutiot-algoritmia hidastava tekijä on jokaiselle kuutiolle tehtävä konfiguraation määritys. Naiivi pintaverkkoalgoritmi on nopeampi, koska se ei käsittele yhtään vokseleiden muodostamaa kuutiota nelikulmioiden pisteiden laskemiseksi, ellei 3D-mallille muodosteta pintaa.

Lysenko (2012) on saanut tekemässään kokeessa vastaavanlaisia tuloksia, missä naiivi pintaverkkoalgoritmi suoriutui marssikuutiot-algoritmia paremmin kaikilla mittauskerroilla. Tämän johdosta tutkimuksen tulokset vahvistavat Lysenkon (2012) tekemiä havaintoja. Lysenkon (2012) suorittamissa mittauksissa taulukon koko pidettiin vakiona 65^3 ja volyymi generoitiin sinifunktiolla, jonka taajuutta nostamalla kasvatettiin generoitavan pinnan määrää.

5.3.2 Kolmiot

Kuva 21 esittää saadut mittaukset generoitujen kolmioiden lukumääristä. Kolmioiden lukumäärät ovat algoritmien välillä melko tasapuoliset, koska molemmat algoritmit ovat generoineet 3D-malleja samasta volyymistä. Lieviä eroja kolmioiden lukumäärissä syntyy erilaisesta pinnanmuodostuksesta johtuen ja naiivissa pintaverkossa on keskimäärin vähemmän kolmioita, koska se muodostaa pinnan pelkkinä nelikulmioina, jotka muodostetaan neljällä pisteellä ja kahdella kolmiolla. Marssikuutioiden tapauksessa konfiguraatio voi generoida 3D-mallille 0–5 kolmiota ja testitapauksissa käytetyn tiheään vaihtelevan volyymin pohjalta on todennäköisempää, että marssikuutioille on määräytynyt pääasiasiallisesti 2–4 kolmiota sisältäviä konfiguraatioita, joka selittää hyvin mittauksista saatuja eroja algoritmien välillä.

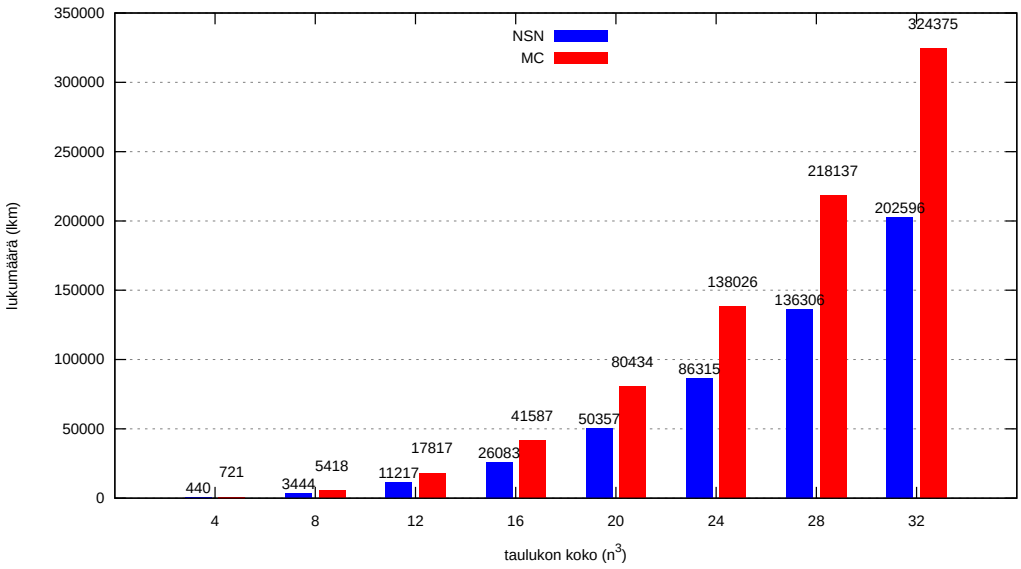


Kuva 21: Kolmioiden määrä.

Mittauksissa saadut kolmioiden lukumäärät noudattavat kuutiollista kasvua mittausten välillä eli niillä on suoritusaikojen tavoin lineaarinen suhde taulukon alkioiden lukumäärän. Tämä suhde kertoo siitä, että volyymiin generointiin käytetyt parametrit onnistuivat tehtävässään luoda pinnanmuodostusalgoritmeille runsaasti generoitavaa pintaa riippumatta taulukon koosta.

5.3.3 Pisteet

Kuvan 22 perusteella naiivi pintaverkkoalgoritmi tuottaa keskimäärin kolmanneksen vähemmän pisteitä marssikuutioihin verrattuna. Pinnan esittämiseen tarvittavien pisteiden lukumäärien merkittävät erot johtuvat siitä, että naiivissa pintaverkossa pinta muodostetaan nelikulmioina. Nelikulmio muodostetaan kahdella kolmiolla ja naiivin pintaverkon tapauksessa nämä kolmiot voidaan määrittää neljällä pisteellä. Marssikuutioiden tapauksessa yksi kolmio tarkoittaa normaalisti kolmea 3D-mallin pistettä, joten naiivin pintaverkon lopullinen pisteiden lukumäärä on keskimäärin kolmanneksen pienempi. Naiivin pintaverkon pisteiden lukumäärä voidaan laskea kaavalla $\frac{k}{2} \cdot 4 = 2 \cdot k$ ja marssikuutioiden pisteiden lukumäärä kaavalla $k \cdot 3$, joissa k tarkoittaa kolmioiden lukumäärää.



Kuva 22: Pisteiden määrä.

Kuvasta 21 havaitsimme, että kolmioiden määrät vaihtelevat algoritmeilla generoitujen mallien välillä, joten naiivin pintaverkon pisteiden määrä ei ole mitatuissa tuloksissa täsmällisesti kolmanneksen pienempi. Naiivissa pintaverkkoalgoritmissa keskimääräinen kolmioiden lukumäärä oli lievästi vähäisempää, joten se myös tarkoittaa, että pisteiden keskimääräinen lukumäärä on yli kolmanneksen pienempää marssikuutioihin nähden. Vähäisemmän pisteiden määrän etu on, että 3D-malli vaatii vähemmän muistia ja kolmanneksen pienempi muistin käyttö pisteiden suhteen on huomattava kolmioiden lukumäärän ollessa suuri.

5.4 Johtopäätökset pinnanmuodostusalgoritmien vertailusta

Tutkimuksen tavoitteena oli selvittää, kumpi algoritmeista on nopeampi ja kartoittaa eroja 3D-mallien kolmioiden sekä pisteiden lukumäärissä. Vertailun tulokset olivat ennakoitavissa paitsi arvioimalla ohjelmakoodin toteutuksia myös perehtymällä Lysenkon (2012) tuloksiin. Tutkimuksessa tehdyissä mittauksissa algoritmit olivat suhteellisen tasapuolisia, mutta naiivi pintaverkkoalgoritmi suoriutui paremmin kaikilla osa-alueilla.

Suoritusaikaa mittaavista tuloksista havaittiin naiiviin pintaverkkoalgoritmin skaalautuvan huomattavasti paremmin kuin marssikuutiot-algoritmi, kun generoitava kolmioiden lukumäärä on alhainen taulukon kokoon nähden. Ero suoritusajoissa algoritmien välillä syntyi erilaisesta silmukoinnista. Havainnon pohjalta naiivi pintaverkko-algoritmi vaikuttaisi soveltuvan erityisen hyvin esimerkiksi kolmiulotteisen maastojen generointiin, koska tyypillisesti esitettävää pintaa ei ole niin paljoa kuin suoritetuissa testeissä.

Kolmioiden lukumäärät algoritmien välillä olivat odotetusti tasaiset, mutta naiivin pintaverkkoalgoritmin tarvitsi laskea noin kolmanneksen vähemmän pisteitä saman kolmiomäärän esittämiseen kuin marssikuutiot-algoritmi. Ero syntyy siitä, että naiivi pintaverkkoalgoritmi muodostaa pinnan nelikulmioina. Runsaasti kolmioita sisältävien 3D-mallien tapauksessa vähäisempi pisteiden lukumäärä voi olla merkittävä valintakriteeri muistinkulutuksen kannalta. Kolmioiden ja pisteiden lukumääriä mittaavat testit ovat yleistettävissä muihin marssikuutiot-algoritmille duaalisiin algoritmeihin niiden samanlaisesta pinnanmuodostuksesta johtuen.

Lysenkon (2012) ansiosta naiivi pintaverkkoalgoritmi on saanut jonkin verran näkyvyyttä, mutta valitettavasti sen tunnettavuus on edelleen yllättävän alhainen lähes vuosikymmenen jälkeenkin. Yksi tutkielman tavoitteista onkin tehdä algoritmia tunnetummaksi lisäten siitä saatavaa tietoa. Jatkotutkimuksena voisi tutkia enemmän marssikuutioille duaalisia algoritmeja sekä kehittää uusia menetelmiä pisteiden laskentaan.

6 Yhteenveto

Tutkielman tavoitteena oli tutkia ja kehittää volumetrisen datan generointiin ja mallintamiseen soveltuvia tekniikoita. Tutkimusaineisto koostui suuresta määrästä keinotekoisesti tuotettua dataa, ja tutkimusmenetelmänä sovellettiin tietojenkäsittelytieteissä tyypillistä suunnittelutieteellistä viitekehystä. Tutkimuksessa verrattiin vokseleihin perustuvista pinnanmuodostusalgoritmeista marssikuutioita ja naiivia pintaverkkoalgoritmia, joka tutkimustulosten perusteella osoittautui näistä kahdesta algoritmista suorituskyvyltään tehokkaammaksi kaikissa keskimääräisissä mittauksissa.

Molempien menetelmien aikavaativuudet olivat samat, mutta naiivi pintaverkkoalgoritmi osoittautui merkittävästi nopeammaksi tilanteessa, jossa 3D-malliin ei sisältynyt juurikaan muodostettavaa pintaa. Naiivi pintaverkkoalgoritmi soveltuukin erityisen hyvin kaikenlaisen volymetrisen datan mallintamiseen ja säilyttää johtoasemansa suorituskyvyssä riippumatta generoitavan 3D-mallin monimutkaisuudesta. Maaston mallintamisessa naiivilla pintaverkolla on siis suorituskyvyllinen etu ja nelikulmioilla muodostettava pinta noudattaa kolmioita säännöllisempää rakennetta, mitkä voivat olla merkittäviä algoritmin valintaan vaikuttavia tekijöitä esimerkiksi pelinkehityksessä. Algoritmikuvauksen ja viitetoteutuksen puuttumisen vuoksi algoritmi ei ole juurikaan yleistynyt.

Tutkielma tarjosi hyvät mahdollisuudet myös tehdä kattavan katsauksen erilaisista maaston proseduraalisen generoinnin menetelmistä, joilla tuotettiin korkeusdataa pinnanmuodostusalgoritmien sovelluskohteena. Lähtökohdaksi valittiin gradienttikohinat, joiden avulla generoitu korkeusdata yhdistettiin useiden välivaiheiden kautta volumetriseen pinnanmuodostukseen. Maaston generoinnissa eroosion mallintamiseen on käytetty tyypillisesti iteratiivisia menetelmiä, kuten lämpö- ja vesieroosiota, mutta kohinan analyyttiset derivaatat tarjoavat tehokkaan vaihtoehdon eroosion mallintamiseen. Derivaatat tuovat muutenkin arvokasta lisäinformaatiota kohinafunktion käyttäytymisestä, sillä korkeusdatan tapauksessa tiedetään esimerkiksi generoitavan maaston kaltevuus tietyssä sijainnissa.

Tutkielman pinnanmuodostusalgoritmit erikoistuvat volumetrisen datan mallintamiseen, joten tutkielman käsittelemät proseduraaliset menetelmät eivät hyödynnä niiden kykyä mallintaa korkeuden lisäksi volumetrisiä piirteitä, kuten luolastoja tai kielekkeitä. Tämä on yksi mahdollinen jatkotutkimuksen kohde, jos pinnanmuodostusalgoritmien sijaan tutkittaisiin tarkemmin maaston proseduraalista generointia volymetrisestä näkökulmasta. Toiseksi jatkotutkimuksen aiheeksi soveltuu uusien volumetristä dataa mallintavien pinnanmuodostusalgoritmien kehittäminen. Aihealueeksi voisi valita myös päinvastaisen lähestymistavan, ja tutkia erilaisia tapoja määrittää vokseleiden arvot ohjelmalle syötetyn 3D-mallin perusteella.

Unity-projektissa kehitettiin pelinkehitykseen soveltuvaa työkalua, jonka avulla pystyttäisiin esittämään pelaajan muokattavissa olevaa proseduraalista maastoa. Korkeusdataa tuottavat proseduraaliset menetelmät yhdistettiin volumetrista dataa käsittelevien pinnanmuodostusalgoritmien kanssa, minkä tuloksena projektille asetetut pelinkehitykselliset tavoitteet saavutettiin. Reaaliaikaisesti muokattavissa oleva maasto asettaa korkeita vaatimuksia pinnanmuodostusalgoritmien suoritusteholle, mikä kannusti tekemään vertailevaa tutkimusta vaihtoehtoisista pinnanmuodostusalgoritmeista. Tutkimuksen tuloksena kehitettiin naiivi pintaverkkoalgoritmi, joka tarjoaa suorituskyvyltään tehokkaan vaihtoehdon volumetrisen datan 3D-mallinnukseen.

Lähteet

Archer, Travis. 2011. “Procedurally generating terrain”. Teoksessa *44th annual midwest instruction and computing symposium, Duluth,* 378–393.

Bourke, Paul. 1994. *Polygonising a scalar field.* Saatavilla WWW-muodossa, http://paulbourke.net/geometry/polygonise/, viitattu 12.2.2021.

———. 1998. *Generating noise with different power spectra laws.* Saatavilla WWW-muodossa, http://paulbourke.net/fractals/noise/, viitattu 15.2.2021.

Chen, Baoquan, F. Dachille ja A. Kaufman. 1999. “Forward image mapping”. Teoksessa *Proceedings Visualization ’99 (Cat. No.99CB37067),* 89–514. https://doi.org/10.1109/VISUAL.1999.809872.

Doi, Akio, ja Akio Koide. 1991. “An efficient method of triangulating equi-valued surfaces by using tetrahedral cells”. *IEICE TRANSACTIONS on Information and Systems* 74 (1): 214–224.

Dustler, Magnus, Predrag Bakic, Hannie Petersson, Pontus Timberg, Anders Tingberg ja Sophia Zackrisson. 2015. “Application of the fractal Perlin noise algorithm for the generation of simulated breast tissue”. *Progress in Biomedical Optics and Imaging - Proceedings of SPIE* 9412 (maaliskuu). https://doi.org/10.1117/12.2081856.

Ebert, David S, F Kenton Musgrave, Darwyn Peachey, Ken Perlin ja Steven Worley. 2003. *Texturing & modeling: a procedural approach.* Morgan Kaufmann.

Francis, Bryant. 2019. *What Astroneer’s devs learned while leaving Early Access.* Saatavilla WWW-muodossa, https://www.gamasutra.com/view/news/338417/What_Astroneers_devs_learned_while_leaving_Early_Access.php, viitattu 17.2.2021.

Galin, Eric, Eric Guérin, Adrien Peytavie, Guillaume Cordonnier, Marie-Paule Cani, Bedrich Benes ja James Gain. 2019. “A Review of Digital Terrain Modeling”. *Computer Graphics Forum* 38 (2): 553–577. https://doi.org/https://doi.org/10.1111/cgf.13657. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13657.

Gibson, Sarah F. F. 1998. “Constrained elastic surface nets: Generating smooth surfaces from binary segmented data”. Teoksessa *Medical Image Computing and Computer-Assisted Intervention — MICCAI’98,* toimittanut William M. Wells, Alan Colchester ja Scott Delp, 888–898. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-49563-5.

Gustavson, Stefan. 2005. “Simplex noise demystified”. *Linköping University, Linköping, Sweden, Research Report.*

———. 2008. *flownoisedemo/srdnoise23.c.* Saatavilla WWW-muodossa, https://weber.itn.liu.se/~stegu/aqsis/flownoisedemo/srdnoise23.c, viitattu 21.5.2021.

Hevner, Alan R, Salvatore T March, Jinsoo Park ja Sudha Ram. 2004. “Design science in information systems research”. *MIS quarterly,* 75–105.

Ju, Tao, Frank Losasso, Scott Schaefer ja Joe Warren. 2002. “Dual Contouring of Hermite Data”. Teoksessa *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques,* 339–346. SIGGRAPH ’02. San Antonio, Texas: Association for Computing Machinery. ISBN: 1581135211. https://doi.org/10.1145/566570.566586.

Lorensen, William E. 2020. “History of the Marching Cubes Algorithm”. *IEEE Computer Graphics and Applications* 40 (2): 8–15. https://doi.org/10.1109/MCG.2020.2971284.

Lorensen, William E., ja Harvey E. Cline. 1987. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. *SIGGRAPH Comput. Graph.* (New York, NY, USA) 21, numero 4 (elokuu): 163–169. ISSN: 0097-8930. https://doi.org/10.1145/37402.37422.

Lysenko, Mikola. 2012. *Smooth Voxel Terrain (Part 2).* Saatavilla WWW-muodossa, https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/, viitattu 1.4.2021.

Nguyen, Hubert. 2007. *Gpu Gems 3.* First. Addison-Wesley Professional. ISBN: 9780321545428.

Owens, John D., Mike Houston, David Luebke, Simon Green, John E. Stone ja James C. Phillips. 2008. “GPU Computing”. *Proceedings of the IEEE* 96 (5): 879–899. https://doi.org/10.1109/JPROC.2008.917757.

Perlin, Ken. 1985. “An image synthesizer”. *ACM Siggraph Computer Graphics* 19 (3): 287–296.

Perlin, Ken. 2001. “Noise hardware”. *Real-Time Shading SIGGRAPH Course Notes.*

———. 2002a. *Improved Noise reference implementation.* Saatavilla WWW-muodossa, https://mrl.cs.nyu.edu/~perlin/noise/, viitattu 21.7.2021.

———. 2002b. “Improving Noise”. Teoksessa *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques,* 681–682. SIGGRAPH ’02. San Antonio, Texas: Association for Computing Machinery. ISBN: 1581135211. https://doi.org/10.1145/566570.566636.

Persson, Markus. 2011. *Terrain generation, Part 1.* Saatavilla WWW-muodossa, https://n0tch-blog-blog.tumblr.com/post/4231184692/terrain-generation-part-1, viitattu 4.2.2021.

Quilez, Inigo. 2002. *Domain warping.* Saatavilla WWW-muodossa, https://www.iquilezles.org/www/articles/warp/warp.htm, viitattu 17.3.2021.

———. 2008. *Value noise derivatives.* Saatavilla WWW-muodossa, https://www.iquilezles.org/www/articles/morenoise/morenoise.htm, viitattu 22.4.2021.

———. 2017. *Gradient noise derivatives.* Saatavilla WWW-muodossa, https://www.iquilezles.org/www/articles/gradientnoise/gradientnoise.htm, viitattu 28.7.2021.

———. 2019. *fBM.* Saatavilla WWW-muodossa, https://www.iquilezles.org/www/articles/fbm/fbm.htm, viitattu 13.3.2021.

Rashid, Tanweer, Sharmin Sultana ja Michel A Audette. 2016. “2-manifold surface meshing using dual contouring with tetrahedral decomposition”. *Advances in Engineering Software* 102:83–96.

Schaefer, Scott, ja Joe Warren. 2003. “Dual Contouring: The Secret Sauce"” (maaliskuu).

Shaker, Noor, Julian Togelius ja Mark J Nelson. 2016. *Procedural content generation in games.* Springer.

Smelik, Ruben M., Tim Tutenel, Klaas Jan de Kraker ja Rafael Bidarra. 2010. “Declarative Terrain Modeling for Military Training Games”. *Int. J. Comput. Games Technol.* (London, GBR) 2010 (tammikuu). ISSN: 1687-7047. https://doi.org/10.1155/2010/360458.

The Khronos Group. 2013. *Calculating a Surface Normal.* Saatavilla WWW-muodossa, https://www.khronos.org/opengl/wiki/Calculating_a_Surface_Normal, viitattu 21.7.2021.

Vivo, Patricio Gonzalez. 2015. *The Book of Shaders: Fractal Brownian Motion.* Saatavilla WWW-muodossa, https://thebookofshaders.com/13/, viitattu 13.3.2021.

Liitteet

A Gradienttikohinan viitetoteutuksen mukaiset gradienttivektorit

```
public static Vector3[] grad3 =
{
    new(1,1,0), new(-1,1,0), new(1,-1,0), new(-1,-1,0),
    new(1,0,1), new(-1,0,1), new(1,0,-1), new(-1,0,-1),
    new(0,1,1), new(0,-1,1), new(0,1,-1), new(0,-1,-1),
    new(1,1,0), new(0,-1,1), new(-1,1,0), new(0,-1,-1),
};
```

Listaus 5: Perlin-kohinan viitetoteutuksen mukaiset gradienttivektorit.

```
public static Vector3[] grad3 =
{
    new(-1,1,-1),  new(-1,-1,0), new(0,-1,-1), new(-1,0,-1),
    new(-1,1,-1),  new(-1,0,1),  new(1,-1,0),  new(0,1,-1),
    new(-1,-1,1),  new(1,-1,0),  new(0,1,-1),  new(-1,0,1),
    new(-1,-1,1),  new(1,0,-1),  new(-1,1,0),  new(0,-1,1),
    new(1,-1,-1),  new(-1,1,0),  new(0,-1,1),  new(1,0,-1),
    new(1,-1,-1),  new(-1,0,-1), new(-1,-1,0), new(0,-1,-1),
    new(1,1,1),    new(1,1,0),   new(0,1,1),   new(1,0,1),
    new(1,1,1),    new(1,0,1),   new(1,1,0),   new(0,1,1),
    new(1,-1,1),   new(1,1,0),   new(0,1,1),   new(1,0,1),
    new(1,-1,1),   new(1,0,-1),  new(-1,1,0),  new(0,-1,1),
    new(1,1,-1),   new(-1,1,0),  new(0,-1,1),  new(1,0,-1),
    new(1,1,-1),   new(-1,0,1),  new(1,-1,0),  new(0,1,-1),
    new(-1,1,1),   new(1,-1,0),  new(0,1,-1),  new(-1,0,1),
    new(-1,1,1),   new(1,0,1),   new(1,1,0),   new(0,1,1),
    new(-1,-1,-1), new(-1,-1,0), new(0,-1,-1), new(-1,0,-1),
    new(-1,-1,-1), new(-1,0,-1), new(-1,-1,0), new(0,-1,-1),
};
```

Listaus 6: Simpleksikohinan viitetoteutuksen mukaiset gradienttivektorit.