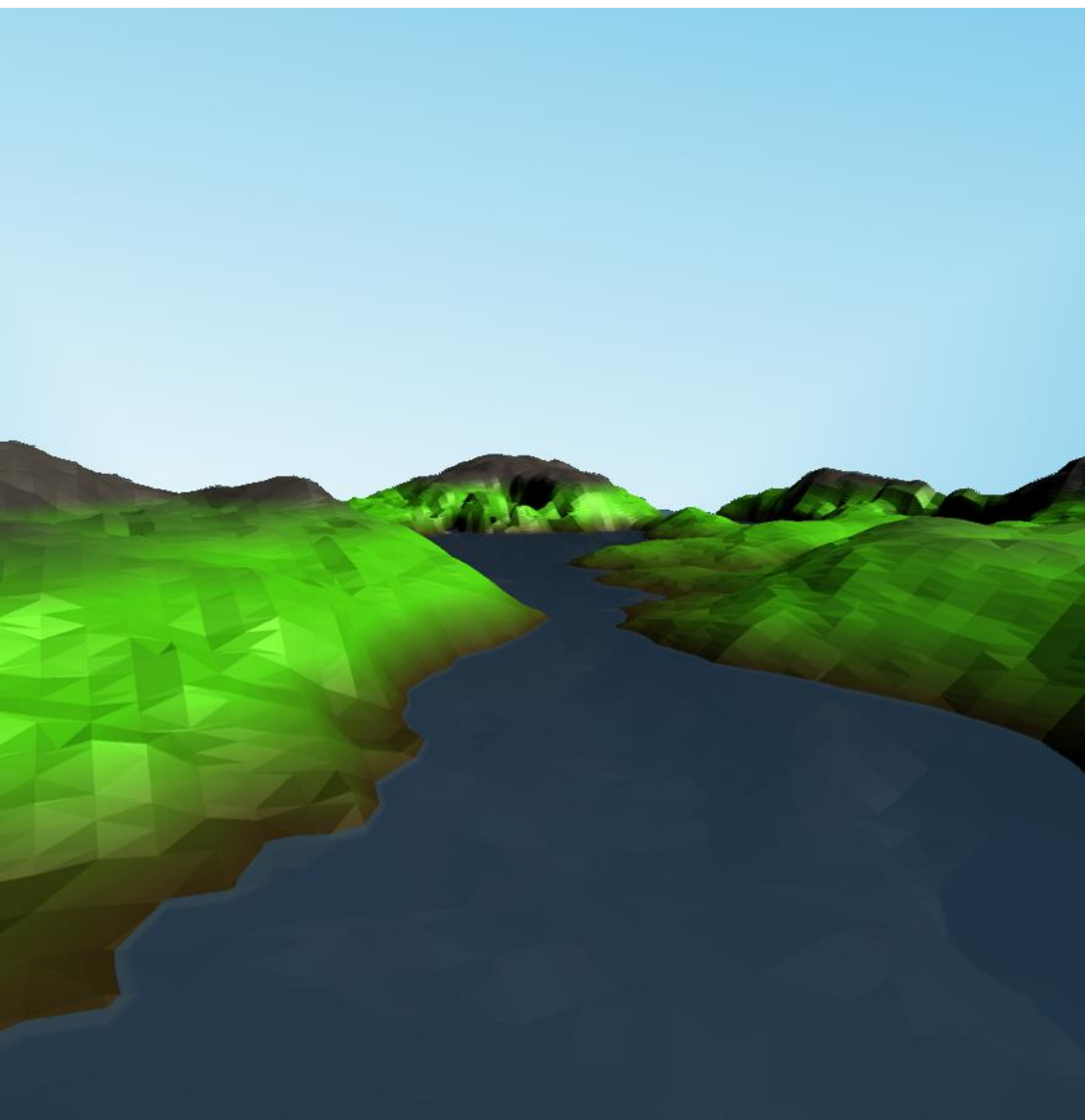


Anssi Remes

# Proseduraalinen kenttägenerointi kuutiomarssittamalla



Tradenomi  
Tietojenkäsittely  
Syksy 2020



KAMK • University  
of Applied Sciences

## Tiivistelmä

**Tekijä(t):** Remes Anssi

**Työn nimi:** Proseduraalinen kenttägenerointi kuutiomarssittamalla

**Tutkintonimike:** Tradenomi, tietojenkäsittely

**Asiasanat:** kuutiomarssitus, kenttägenerointi, proseduraalinen generointi, peliohjelmointi, pelikehitys, algoritmit

Kuutiomarssitusalgoritmi on keino, jolla voidaan muodostaa tarkkoja 3D-malleja joukoista kolmiulotteisen avaruuden pisteitä. Alkuperäisessä käyttötarkoituksessa pisteet voivat esittää esimerkiksi magneettikuvauksen mittaustuloksia. Algoritmi toimii hyvin nopeasti, sillä sen toiminta perustuu hakutaulukkoon, jonka mukaisesti jokaiselle pisteiden muodostamalle kuutiolle asetetaan sen tapausta vastaavat polygonit. Kun algoritmi on marssinut lävitse jokaisen datan kuution, muodostuu polygoneista valmis mitattavaa kappaletta esittävä malli.

Tietokonepeleissä sisältöä generoidaan usein automatisoidusti niin, että peleissä voi olla lähes rajaton määrä tavaroita, hahmoja tai kenttiä. Tällaista proseduraalisesti generoitua sisältöä ei tarvitse luoda käsin, eikä sitä tarvitse tallentaa, sillä se luodaan pelin aikana parametreista ja moduuleista. Kenttägeneroinnissa ohjelmallinen sisällön luominen mahdollistaa usein kenttien muokkaamisen niiden modulaarisen rakenteen vuoksi. Synnyttämällä kenttien geometria kuutiomarssittamalla voidaan luoda tällaisia kolmiulotteisia muokattavia pelikenttiä nopeasti. Algoritmin käyttäminen mahdollistaa myös monimutkaisen kenttägenerointilogiikan lisäämisen, sillä se tarvitsee toimiakseen ainoastaan taulukon vokseleita, joita voidaan muokata monella yksinkertaisella tavalla.

Tämän opinnäytetyön tavoite on esittää käytännön sovellutus kuutiomarssitusalgoritmilte tietokonepelien kenttägeneroinnissa sekä pohtia tekniikan rajoituksia ja potentiaalia. Tässä opinnäytetyössä tuotettiin pelidemo Unity-pelimoottorissa, jossa kentän lohkojen satunnaisella Perlin-kohinalla täytetyistä vokselitaulukoista luodaan kuutiomarssittamalla reaaliaikaisesti muokattavia pelikenttiä. Työ esittelee tekniikan toimintaa ja monta eri tapaa, joilla algoritmin käyttäminen voi palvella kenttägeneroinnin tarpeita, kuten luoda sisältävien vuorien luomisen yhdistämällä kaksi- ja kolmiulotteista Perlin-kohinaa. Lisäksi demossa esitellään pelattavuutta, jossa pelaaja voi muokata generoitua kenttää hyvinkin yksityiskohtaisesti. Kuutiomarssituksen toimiessa todella nopeasti kentän muokatut vokselitaulukot voidaan kuutiomarssittaa uudelleen pelinaikaisesti ja muokkaukset näkyvät kentässä lähes välittömästi.

Demon avulla havainnollistetaan, miten kenttägeneroinnin parametrien muuttaminen vaikuttaa algoritmin tekemään lopputulokseen sekä, miten generoinnissa kentän osia voidaan lisätä, poistaa tai yhdistellä. Kentän vokselitaulukkojen ollessa yksinkertaisesti muokattavissa sen vokselien arvoja voidaan muokata samalla tavoin kuin pikseleitä kuvankäsittelyssä. Tässä työssä käytetyt esimerkit eivät täytä minkään varsinaisen tietokonepelin kenttägeneroinnin tarpeita, vaan esittävät millä tavoin kehittäjän on mahdollista muokata kuutiomarssitettavaa kenttää. Mahdollisuuksien ja jatkokehitysideoiden lisäksi työssä onnistuttiin löytämään lukuisia vikoja algoritmin käyttämisessä peleissä. Kuutiomarssitusta käytettäessä pelin kenttägenerointiin on tärkeätä huomioida kuinka nopeasti kentän koon kasvattaminen nostaa tarvittavaa laskentatehoa ja muistin määrää. Algoritmin toimintaa voidaan optimoida useilla keinoilla, mutta laskenta-ajan säästäminen voi kasvattaa vaaditun muistin määrää ja päinvastoin. Pelintekijän on pystyttävä sovittamaan algoritmin toiminta omaan peliinsä tavalla, joka palvelee kehitettävän pelin visiota sen laitevaatimusten pysyessä hillittyinä.

## **Abstract**

**Author(s):** Remes Anssi

**Title of the Publication:** Procedural Level Generation Using Cube Marching

**Degree Title:** Bachelor of Business Administration

**Keywords:** cube marching, level generation, procedural content generation, game programming, game development, algorithms

Cube marching algorithm allows turning series of points in three-dimensional space into accurate 3D-models. In its original purpose, the points could be, for example, from the results of a magnet resonance imaging. The algorithm works rapidly as it uses a lookup table to place according polygons into each cube formed by the points in the data. After the algorithm has marched through every cube in the data, the polygons form a complete model of the scanned object.

Content in video games is often automatically created, so that there are nearly infinite amounts of items, characters or levels. This type of procedurally generated content is not made by hand and it does not need to be saved as it is generated in runtime from modules according to set parameters. Using procedural generation in levels often allows the editing of its modules. Generating these kinds of editable levels quickly is possible by using cube marching. Using the algorithm also enables implementing complex level generation logic because it requires only a table of voxels which are trivial to edit in multiple different ways.

This thesis contains the details of developing a game demo, in which cube marching is used to form levels from Perlin-noise and that can be edited in real-time. Demo has been developed in Unity game engine and it explores the possibilities and technical requirements of the algorithm. This thesis demonstrates the techniques available to the developers to use in their cube marching based level generation, like the possibility of creating mountains and caves by combining both two- and three-dimensional Perlin noise. Additionally, the demo presents gameplay in which the player can edit the level in high detail. Because the cube marching algorithm runs so quickly the edits done to the level's voxel tables can be displayed near instantly as a change in the terrain's shape.

The game demo is used to showcase how its level generation's parameters affect its outcome and how it is possible to add, remove or combine parts of the level. Because editing the level's voxel tables is trivial the voxels can be edited similarly to pixels in image processing. The examples in this thesis were not made for any video game in mind, but to demonstrate the ways of editing the cube marched level. The thesis succeeded to find several faults in the technique in addition to its possibilities when it is used in video games. When using cube marching in level generation it is important to note that increasing the size of the levels raises the amount of required processing power and memory rapidly. The algorithm can be optimized in a plethora of ways, but by saving processing time the amount of required memory increases and vice versa. The game developer must implement the algorithm in such a way that it serves the vision for their game while its hardware requirements stay reasonable.

## Alkusanat

Tämän opinnäytetyön aihe löytyi opiskelijatovereiden seurassa nostalgisia pelejä muistellessa ja pohtiessa, minkälaisia ne olisivat, jos ne olisi tehty nykypäivänä. Vaikka aiheeksi valitsemani tekniikka on kehitetty jo ennen syntymääni, sen käyttäminen tietokonepeleissä on vielä tänäänkin harvinaista. Minua motivoi ajatus sen käyttämisestä tavalla, joka herättäisi vanhojen klassikkopelien kenttägeneroinnit ja tuhoutuvat ympäristöt henkiin kolmiulotteisina.

Työ on tehty erikoisena aikana, jolloin maailmanlaajuinen pandemia teki elämästämme poikkeusoloissa ahdistavampaa ja yksitoikkoisempaa kuin ennen. Teknologian ansiosta ystävyysuhteet kuitenkin säilyivät ennallaan toisiaan tukevan ja kannustavan kanssakäymisen pysyessä lähes jokapäiväisenä.

Tahdon kiittää Kajaanin Ammattikorkeakoulun opiskelijayhteisöä ja Suomen pelinkehittäjiä kokemastani lämmihenkisyydestä ja pyyteettömästä avuliaisuudesta. Meitä yhdistää yhteinen intohimo ja rohkeus sen seuraamiseksi. Toivon teille kaikille onnea ja menestystä!

Kajaanissa 16.11.2020



Anssi Remes

## Sisällys

1	Johdanto .....	1
2	Kuutiomarssitus .....	2
3	Proseduraalisuus peleissä.....	6
3.1	Kenttägenerointi .....	6
3.2	Satunnaisuus ja siemenluvut.....	8
3.3	Perlin-kohina .....	9
4	Pelidemo.....	12
4.1	Suunnittelu .....	12
4.2	Toteutus .....	13
4.3	Testaus .....	20
4.3.1	Maaston generointi.....	26
4.3.2	Etäisyysfunktiot.....	29
4.3.3	Maaston muokkaus.....	31
5	Tulokset .....	33
5.1	Onnistumiset ja mahdollisuudet .....	33
5.2	Haasteet ja kehityskohteet.....	35
6	Yhteenvedo .....	39
	Lähteet .....	41

## Käsiteluettelo

Algoritmi	Tietojenkäsittelyssä laskentaan tai ongelmanratkaisuun käytetty sarja komentoja
Olio	Olio-ohjelmoinnissa ohjelman perusrakenteen osa, joka voi sisältää toiminnallisuutta ja tietoa, ja jonka määrittelee luokka
Polygoni	Tietokonegrafiikassa 3D-mallin muodostava pienin jakamaton monikulmio
Polygoniverkko	Tietokonegrafiikassa useista polygoneista rakentuva tietorakenne, jolla kuvataan 3D-malleja
Skalaarikenttä	Funktio, joka palauttaa jokaiselle matemaattisen avaruuden kohdalle skalaariarvon
Verteksi	3D-mallin polygonien kulmapiste
Vokseli	Kolmiulotteisen ruudukon pienin yksittäinen osa, johon liittyy yksi tai useampi lukuarvo

## 1 Johdanto

Lähes jokainen tietokonepeli sisältää jonkinasteista proseduraalisuutta, jossa pelin sisällön tuottamisesta vastaa pelinkehittäjien sijaan heidän ohjelmoimansa automatisoitu logiikka. Proseduraalinen sisältö voi olla mitä tahansa pelistä löytyviä asioita, joita voidaan generoida loputtomiin, mutta se on yleisesti yhdistetty pelin kenttägenerointiin. Proseduraalisten kenttien koostuessa osista, jotka generointi sovittaa toisiinsa, on pelaajan usein mahdollista purkaa tai rakentaa niitä. Tällainen pelattavuus avaa parhaimmillaan pelaajalle keinon toteuttaa mitä mielikuvituksellisimpia ideoitaan pelissä. Kuutiomarssitusalgoritmi on toimintaperiaatteiltaan oivallinen keino tällaisten kenttien luomiseksi, sillä se kohtaa ideaalisella tavalla kehittäjän tarpeen generoida loputtomiin pelattavaa sisältöä samalla mahdollistaen pelaajille sen muokkaamisen.

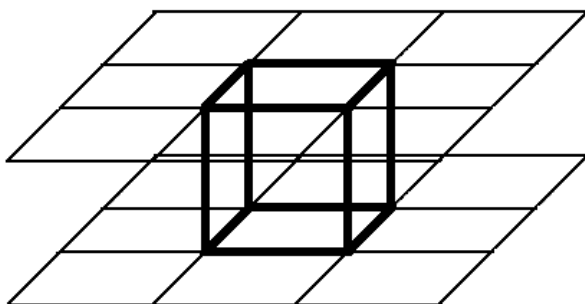
Algoritmin kyky muodostaa sarjasta kolmiulotteisen avaruuden pisteitä kokonaisia 3D-malleja mahdollistaa tietokonepelien kenttien tekemisen hyvin yksinkertaisista lähtökohdista. Alkuun tarvitaan vain määrätyn kokoinen ruudukko vokseleita. Pelinkehittäjän on helppoa tehdä muokkauksia vokseleihin ennen algoritmin suorittamista, ja muodostettavien kenttien monimuotoisuus on kiinni melkein yksistään kehittäjän mielikuvituksesta ja nokkeluudesta. Kuutiomarssitus toimii otollisissa olosuhteissa niin nopeasti, että sitä voidaan käyttää pelin toiminnan aikana pelikentän muokkaamiseksi, jolloin pelaaja voi muokata sitä mielin määrin.

Tämän opinnäytetyön tavoite on esitellä, miten kuutiomarssitusalgoritmia voidaan käyttää tietokonepelien kenttägeneroinnissa ja mitä merkittäviä etuja sillä voidaan saavuttaa. Työssä ohjelmoitiin pelidemo Unity-pelimoottorissa, jonka avulla tutkittiin kenttägeneroinnin mahdollisuuksia ja teknisiä rajoituksia. Demossa muodostetaan pelikenttiä kuutiomarssittamalla kentän lohkojen vokselitaulukoita, joiden pisteiden arvot ovat peräisin satunnaisesta Perlin-kohinasta. Työssä käytetty tekniikka mahdollistaa monenlaisia erilaisia tapoja muokata generoitavaa kenttää, joka voidaan tehdä myös reaaliajassa. Mahdollisuuksien ja haasteiden lisäksi opinnäytetyössä pyrittiin tuomaan esiin niiden jatkokehitysideat sekä ratkaisemiskeinot kaikkein monipuolisimpien ja mielenkiintoisten kenttien generoimiseksi.

## 2 Kuutiomarsitus

Cube marching -algoritmin eli kuutiomarsittamisen kehittivät William Lorensen ja Harvey Cline vuonna 1987. Algoritmilla muodostetaan tietojenkäsittelyssä pistedatasta 3D-malleja, joilla voidaan esittää hyvinkin tarkkaan mitattavan kohteen pinnan muodot. Lorensen ja Cline kehittivät algoritmin aluksi lääketieteen tarkoituksiin, kuten esimerkiksi magneettikuvauksen tuloksien muuttamiseksi 3D-malleiksi edellisiä tunnettuja algoritmeja yksityiskohtaisemmin. [1.]

Kuutiomarsinnassa 3D-malli muodostetaan tutkimalla algoritmilla mittaustuloksista luotavan mallin pinnan muoto ja muodostamalla sen polygonit. Lääketieteessä mittaustulokset voivat olla poikkileikkauksia mitattavasta kohteesta, joissa kaksiulotteiselle tasolle on sijoitettu jokaisen mitauskohdan tulosarvo. Tasot pinoamalla voidaan muodostaa kolmiulotteinen skalaarikenttä, jossa voidaan muodostaa kuutio kahden leikkauksen väliin. Kuutio koostuu kahdeksasta kulmasta, johon kuuluvat neljä mittauspistettä kummastakin poikkileikkauksesta, kuten kuvassa 1 esitetään. [1.]

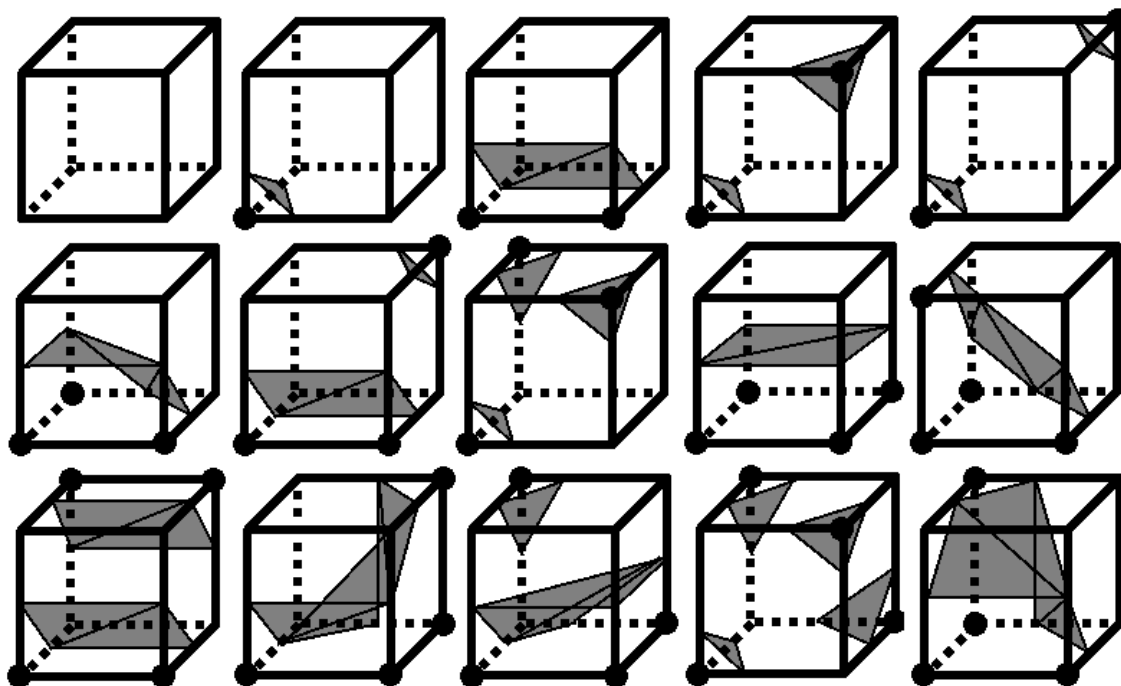


Kuva 1. Kuutio muodostettuna kahden poikkileikkaustason pisteistä [1]

Algoritmi tutkii jokaisen datasta näin muodostuvan kuution ja asettaa niiden kulmiin lukuarvon yksi tai nolla, riippuen, siitä ylittävätkö niissä mittauservot muodostettavan mallin tilavuutta kuvaavaksi valitun numeron arvon. Numero yksi tarkoittaa, että piste sijaitsee rakennettavan mallin ulkopuolella ja nolla tarkoittaa sen sijaitsevan mallin pinnan sisäpuolella. Koska kulmapisteellä voi olla vain kaksi eri arvoa, kuution kahdeksalle kulmalle syntyy  $2^8 = 256$  erilaista mahdollista tapaa, joilla mallin pinta voi läpäistä kuution tilavuuden. Jokaisen tavan muodostamat kolmiot on määritetty valmiiksi yhteen hakutaulukkoon. 256 tapausta voidaan pelkistää kuvauksen vuoksi viiteentoista erilaiseen tapaukseen, joita kääntämällä tai lukuarvoja vastakkaiseksi muuntamalla voidaan muodostaa jokainen kuutiolle mahdollinen tapaus. Tapauksista yksinkertaisimmat ovat ne, joissa jokainen kuution kulma saa saman lukuarvon, jolloin kuutio on kokonaan mallin sisä- tai

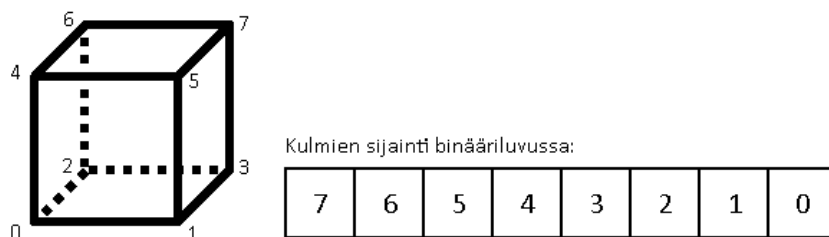


ulkopuolella. Kuvassa 2 on kuvattu kaikki mahdolliset tavat, joilla mallin pinta voi leikata kuution tilavuuden. [1.]

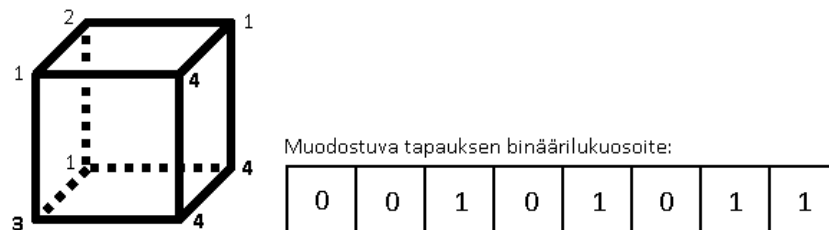


Kuva 2. Viisitoista erilaista kolmioiden asetelmaa, joista voidaan muodostaa kaikki 256 kuution mahdollista tapausta [1]

Koska jokaisella kulmalla on vain kaksi arvoa, yksi tai nolla, niistä voidaan koostaa binäärimuotoinen hakutaulukon osoite siten, että jokaisen kulman arvon paikka pysyy osoitteessa vakiona. Tällaisen kahdeksanbittisen kokonaisluvun minimiarvo ja maksimiarvo kymmenlukujärjestelmässä 0 ja 255. Osoitetta käytetään sitä vastaavan tapauksen noutamiseen hakutaulukosta. Kuvassa 3 on havainnollistettu binäärimuotoisen osoitteen muodostamista kuution kulmien lukuarvoista. [1.]

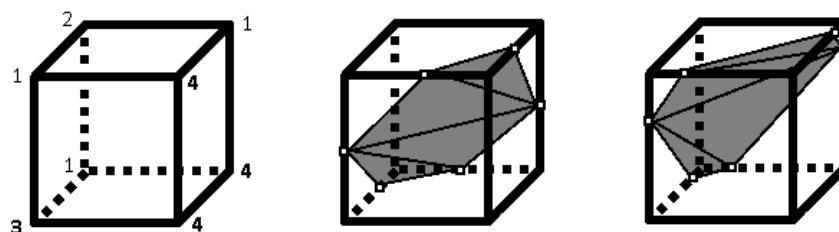


Valittaessa suuremmat numerot kuin 2:



Kuva 3. Esimerkki kulmien arvoja vastaavan tapauksen binääriosoitteen muodostamisesta [1]

Kun yhden kuution tapaus on algoritmisesti määritetty, seuraavaksi sen kolmion tai kolmioiden kulmat eli verteksit sijoitetaan tarkemmin kuution särmille lineaarisesti interpoloiden niiden sijainti särmällä kulmien alkuperäisten lukuarvojen mukaisesti [1]. Verteksin etäisyyksiä särmän pääty pisteistä muutetaan siten, että niiden etäisyyksien suhde on sama kuin kuution kulmien lukuarvojen suhde [2]. Verteksit ikään kuin siirretään kauemmaksi suuremmasta arvosta kuution särmää myöten, jotta lopullinen malli kuvaisi tarkemmin muodostettavan mallin pinnan sijaintia. Interpoloinnin vaikutusta verteksin sijaintiin on havainnollistettu kuvassa 4.



Kuva 4. Kuutio, johon sijoitetaan sen tapausta vastaavat kolmiot, ja joiden verteksin sijainti ratkaistaan lineaarisesti interpoloimalla

Tässä vaiheessa data muodostettavan mallin vertekseistä ja niistä muodostettavista kolmioista voidaan tallentaa 3D-mallin tietorakenteeseen. Muodostetuista kolmioista kyetään ratkaisemaan niiden pinnan normaalivektorit laskemalla niiden verteksin määräämän tason suunta. Edellä mainittujen työvaiheiden jälkeen siirrytään seuraavaan kuutioon skalaarikentässä. Tällä tavoin

marssitaan läpi jokainen kuutio kentässä muodostaen siitä kolmioitu kolmiulotteinen polygonimalli.

### 3 Proseduraalisuus peleissä

Proseduraalinen sisällön generointi tarkoittaa tietokonepeleissä sisällön tuottamista automatisoidusti algoritmien avulla sen manuaalisesti käsin valmistamisen sijaan [3]. Proseduraalinen sisällön generointi ei tapahdu kuitenkaan täysin itsestään, vaan algoritmi noudattaa tyypillisesti sille kehittäjänsä tai käyttäjänsä määrittämiä sääntöjä ja parametreja [3]. Proseduraalisia osia sisältävän tietokonepelin tarkoitus on tuottaa useilla pelikerroilla vaihteleva ja ennalta arvaamaton kokemus tai vain luoda pelaajalle näennäisesti loputon määrä pelisisältöä. Proseduraalisuutta voidaan hyödyntää lisäksi teknisiin tarkoituksiin, kuten muistin säästämiseen. Kenttiä, grafiikkaa, tehtäviä, esineitä tai mitä tahansa pelin sisältöä generoiva ohjelma voi säästää näennäisesti loputtoman määrän tietokoneen muistia kehitysajan lisäksi, sillä sisältöä ei tarvitse laisinkaan tallentaa tai valmistaa ennen pelin julkaisua. Pelin kehittäjän on silti varmistuttava, että ohjelmakoodi tuottaa aina miellyttävän lopputuloksen, mikä voi olla laadunvalvonnan painajainen. [4, s. 8.]

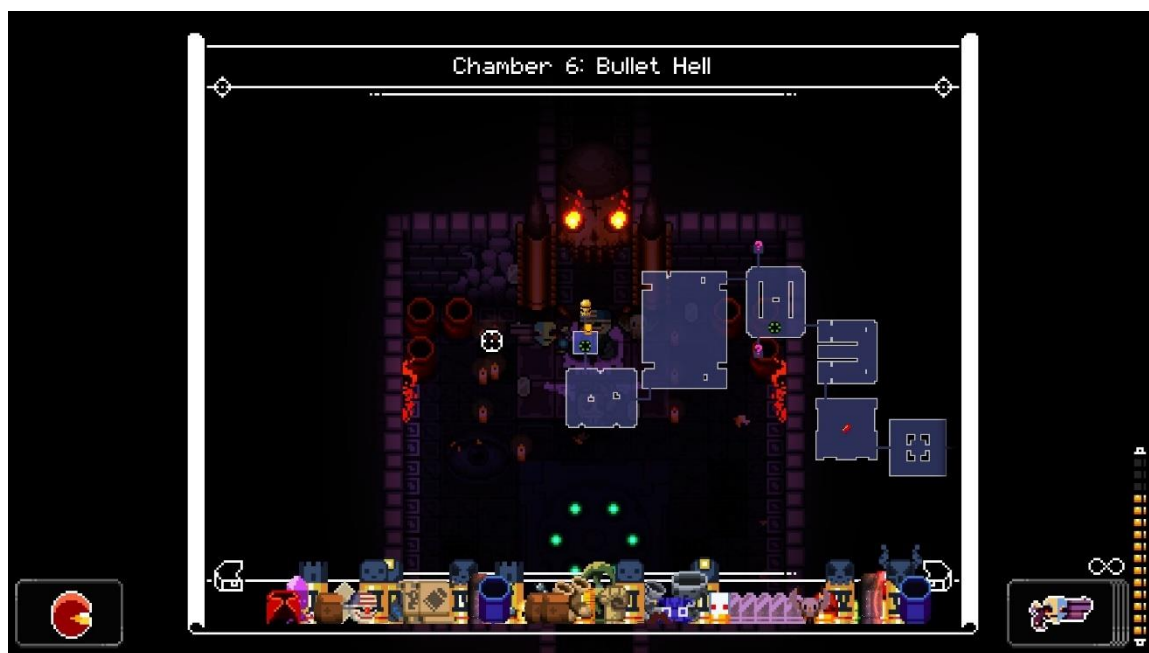
#### 3.1 Kenttägenerointi

Proseduraalisen kenttägeneroinnin suunnittelu kannattaa aloittaa varhain tietokonepelin kehityksessä. Pelinkehittäjällä tulisi olla visio siitä, mitä pelissä tehdään tai mitkä ovat sen ominaisuudet ja säännöt. Täten kehittäjä tietäisi minkälaisia ympäristöjä generoinnin olisi tarkoitus mallintaa ja minkälaisia pelinsisäisiä objekteja ne sisältäisivät. [4, s. 57.]

Kenttägeneroinnin tarkoitus ei ole ripotella kentän palasia täysin satunnaisesti pelikentän muodostamiseksi, vaan oikea generointi noudattaa kehittäjänsä sille asettamia sääntöjä mahdollisesti sisältäen niitä noudattavaa satunnaisuutta. Säännöt voivat määrittää esimerkiksi yksinkertaisimmillaan, että kentässä kuuluu olla lähtöpiste ja maali. Pelin kehittäjä voi määrittää myös mielivaltaisesti paljon monimutkaisempaa logiikkaa generoinnille, kuten esimerkiksi millä tavalla pelaajan aikaisempi pelityyli vaikuttaa kentän sisältöön. [3.]

Tyypillinen kenttägenerointi on aina modulaarinen, eli generoitavat kentät koostuvat yhdistelmästä sen pienimpiä jakamattomia osia – siis moduuleista [4, s. 29]. Kentät voivat koostua todella pienistä yksiköistä tai suuremmista käsin tehdyistä paloista, jotka ainoastaan sovitetaan yhteen niiden järjestystä vaihtamalla. Dodge Rollin kehittämässä Enter the Gungeon -pelissä pelaaja seikkailee suurissa tyrmissä, jonka huoneet ovat käsin tehtyjä. Pelin kenttägenerointi sovittaa satun-

naisen osan kaikista mahdollisista huoneista toisiinsa ja asettaa niihin hirviöitä, aarteita sekä koriste-esineitä. Tyrmät ovat jokaisella pelikerralla erilaisia, sillä käytettyjen huoneiden kombinaatio muuttuu ja niiden sisältö vaihtuu. Pelin kenttägeneroinnin moduuleita ovat kokonaiset huoneet, niihin asetettavat oliot ja esineet. Pelin huoneiden järjestystä kuvaavan kartan voi nähdä kuvassa 5.



Kuva 5. Kuvankaappaus pelin Enter the Gungeonin kartasta [5]

Paljon pienimmistä moduuleista koostuvat Mojangin huippusuositun Minecraft-pelin kentät syntyvät erilaisista yhtenevän kokoisista kuutioista, jotka noudattavat kenttägeneroinnin sääntöjä. Eri materiaaleja kuvaavista palikoista rakentuu maailmoja, joissa on muun muassa autiomaita, metsiä, vuoria, merta ja luolia. Kuvassa 6 on nähtävillä pelin proseduraalisesti generoitu maailma.



Kuva 6. Kuvankaappaus Minecraft-pelistä [6]

Kenttägenerointi on pienten moduulien vuoksi monimukainen, mutta samalla niiden ansiosta pelaajan helposti muokattavissa. Pelaaja voi poimia yksittäisiä kuutioita ja rakentaa vapaasti melkein mitä tahansa. Minecraftin kenttägenerointi voi kestää huomattavasti kauemmin kuin Enter the Gungeonin, mutta luodun sisällön vaihtelun määrä uusilla pelikerroilla on siinä äärettömän suurta. Enter the Gungeonin pelattavuuteen ei kuitenkaan kuulu rakentelu, joten sen ei tarvitse muodostua Minecraftin tapaan pienistä keskenään yhtenäisistä moduuleista. Moduulien koko ja määrä vaikuttavat peleissä kenttägeneroinnin yksityiskohtaisuuteen ja monimutkaisuuteen. Pienien moduulien käyttäminen vaatii kehittäjältä monimutkaisemman generoinnin luomista, mutta se voi toteutuksensa mukaan mahdollistaa pelaajalleen paljon muokkautuvamman ympäristön.

### 3.2 Satunnaisuus ja siemenluvut

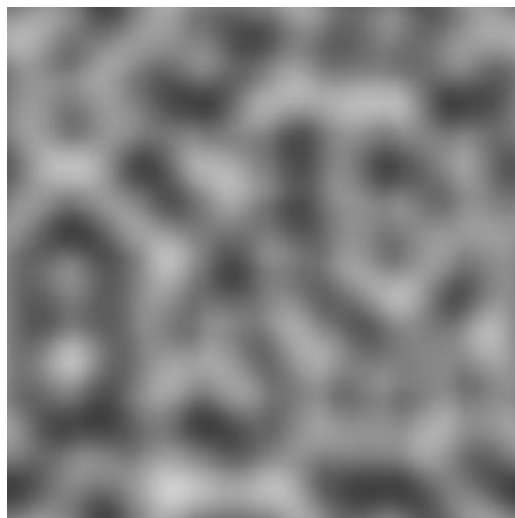
Satunnaisluvut ovat olennaisia proseduraalisissa ja käyttäjälleen arvaamattomasti toimivissa ohjelmissa. Kenttägenerointialgoritmin on tavallaan mahdollista valita arpomalla kentän moduulit tai mitkä tahansa tarvitsemansa lukuarvot. Täysin arvaamattomia lukuja tuottava satunnaislukugeneraattori voi olla sopimaton keino proseduraalisen sisällön tuottamiseen, sillä sitä käyttävä ohjelma generoi erilaisen lopputuloksen jokaisella käyttökerralla. On niin pelin kehittäjän kuin

pelaajan etu, mikäli pelissä samanlaiset lopputulokset ovat mahdollisia. Satunnaislukugeneraattorit ovat tyypillisesti pseudosatunnaisia, eli ne palauttavat aina samanlaisen lukusarjan, mikäli ne käyttävät samaa siemenlukua. [4, s. 271.]

Generaattori käyttää siemenlukua jokaisen palauttamansa satunnaisarvon pohjana, joten samaa siemenlukua käyttäen se palauttaa samanlaisen numerosarjan aina, kun sitä kutsutaan ohjelmassa samassa kohdassa ja samassa järjestyksessä. Siemenluku mahdollistaa esimerkiksi pelaajien keskinäisen proseduraalisesti generoidun sisällön jakamisen antamalla toisilleen ainoastaan käytetyn siemenluvun. Sisällön suuriin tiedostoihin tallentamisen sijaan voidaan muistiin kirjoittaa vain muutaman tavun mittainen numero. Lisäksi moninpeleissä tiedonsiirron määrää voidaan keventää lähettämällä jokaiselle pelaajalle generointivaiheessa identtinen siemenluku. [4, s. 271.]

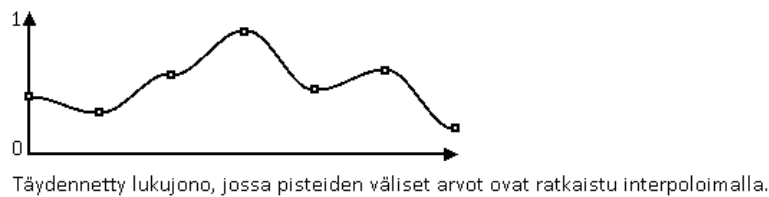
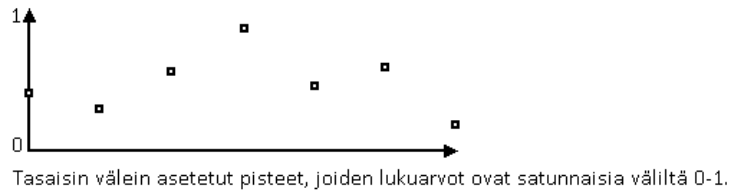
### 3.3 Perlin-kohina

Perlin-kohina on saanut nimensä kehittäjänsä Ken Perlinin mukaan. Sen kehittäessään hän pyrki tekemään luonnollisen näköisiä tekstuureja tietokonegrafiikkaa varten. Täydellisen kohinan sijaan se näyttää luonnolliselta ja satunnaiselta huippukohtineen, mutta on tiheydeltään kokonaan yhtenäinen. Ohjelmoinnissa kohinaa voidaan muodostaa tarvittavan monessa ulottuvuudessa interpoloimalla satunnaisia arvoja sisältävien pisteiden väliset arvot. Tällainen funktio toimii skaalarikenttänä, joka palauttaa pisteen arvon halutussa kohdassa. Kaksiulotteinen Perlin-kohina voidaan esittää kuvana, jossa jokainen pikseli saa väriarvonsa siitä. Kuvassa 7 musta väri merkitsee kohinan lukuarvon nollakohtaa ja valkoinen maksimiarvoa. [4, s. 281.] [7.] [8.]



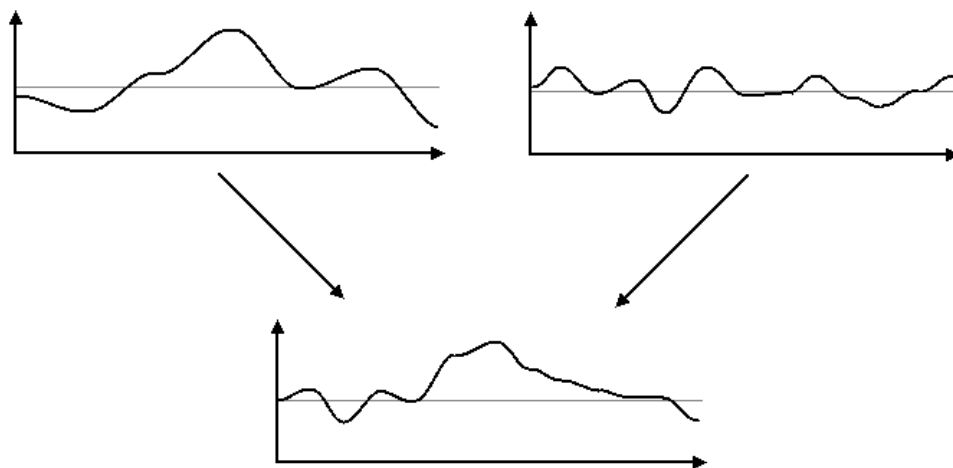
Kuva 7. Kuva kaksiulotteisesta Perlin-kohinasta

Alkeellisessa yksiulotteisessa Perlin-kohinassa lukujonolle asetetaan satunnaislukutaulukon avulla valittuja lukuarvoja, joiden välille muodostetaan gradientti interpoloimalla arvot pisteiden välillä [8]. Kuvassa 8 kuvataan kohinan muodostamista yksiulotteisella lukujonolla.



Kuva 8. Yksiulotteisen Perlin-kohinan muodostaminen [8]

Perlin-kohinasta voidaan muodostaa monimutkaisempia ja yksityiskohtaisempia kuvioita yhdistämällä sitä useita erimuotoisia kerroksia [4, s. 281]. Kohinasta voidaan tehdä yksityiskohtaisempaa lisäämällä siihen oktaaveja, eli eritaajuisia ja amplitudista Perlin-kohinaa [9]. Yhdistämisessä säilyvät kohinan suuret muodot, mutta niistä on nähtävissä lisätyn kohinan yksityiskohdat [9]. Kuvassa 9 on havainnollistettu miltä yhdistetty kohina näyttää.



Kuva 9. Kahden yksiulotteisen Perlin-kohinan yhdistäminen [9]

Proseduraalisessa generoinnissa kaksiulotteisesta Perlin-kohinasta voidaan muodostaa esimerkiksi korkeuskartta, joka esittää kolmiulotteisen maaston korkeuseroja ylhäältäpäin kuvattuna.



Koska kohinan satunnaisarvot voidaan tuottaa muuttumattomasta satunnaislukutaulusta, on kohina pseudosatunnaista, ja taulukon muodostamiseksi voidaan käyttää siemenlukua. Mikäli kohinan satunnaisuutta halutaan muuttaa, se onnistuu satunnaislukutaulun muokkaamisella.

## 4 Pelidemo

Tämän työn tarkoitus on tuottaa demonstraatio kuutiomarssitusalgoritmin käytännön hyödytyksestä tietokonepeleissä. Pelidemo käyttää kuutiomarssitusta kenttägeometrian muodostamiseen skalaarikentästä, joka luodaan Perlin-kohinasta, luonnollisia muotoja sisältävän ympäristön luomiseksi. Määrätyn kokoisen kentän muodostuessa käytännössä kolmiulotteiseen avaruuteen sijoitetuista lukuarvoista voidaan niitä muokata monimutkaisemman kenttägeneroinnin tarkoituksiin yksinkertaisesti. Demonstraation on tarkoitus näyttää lisäksi, minkälaisia mahdollisuuksia käytetty kenttägenerointitapa mahdollistaa. Kenttägenerointiin käytetään pseudosatunnaisuutta, joten sillä voi generoida samanlaisia pelikenttiä käyttäen samaa siemenlukua ja ohjelman parametreja.

Proseduraalisen sisällön generoimisen lisäksi kuutiomarssituksen käyttäminen demossa mahdollistaa pelikentän reaaliaikaisen muokkaamisen. Demon pelaajan on mahdollista vaikuttaa toimintoillaan kuutiomarssitettavaan vokselitaulukkoon kasvattamalla tai vähentämällä sen pisteiden arvoja. Generoimalla algoritmillä kentän muuttunut geometria pelaaja voi täten kaivaa tai rakentaa kentän muotoja reaaliaikaisesti.

### 4.1 Suunnittelu

Demon kenttägeneroinnin keskeinen funktio generoi pelikentän, jonka koon käyttäjä pystyy määrittelemään. Pelikentällä on xyz-koordinaatistossa leveys, korkeus ja syvyys, jotka jaetaan mielimääräisesti pienempiin osiin. Kentän pienempiin osiin jakamisella voidaan vähentää vaadittua laskentatehokkuutta kenttää reaaliaikaisesti muokattaessa ja samalla kenttägeometriaa kuutiomarssitettaessa. Pienemmät osat tai lohkot sisältävät valituista generoinnin parametreista riippumatta vain murto-osan kaikista kentän vokseleista, joiden kaikkien iteroiminen ei ole aina tarpeellista.

Aluksi lohkoille generoidaan vokselitaulukko Perlin-kohinasta. Lohkon resoluutio eli pisteiden määrä ja muodostuvien kuutioiden suuruus on käyttäjän määritettävissä. Pisteet saavat arvonsa niiden sijainnin mukaan kohinasta, joka käyttää koko kenttägeneroinnin ajan muuttumatonta satunnaislukutaulukkoa. Satunnaislukutaulukon arvot on arvottu käyttäen yhtä määritettyä siemenlukua. Perlin-kohina kattaa koko generoitavan kentän ja jokainen lohko saa siitä sijaintinsa

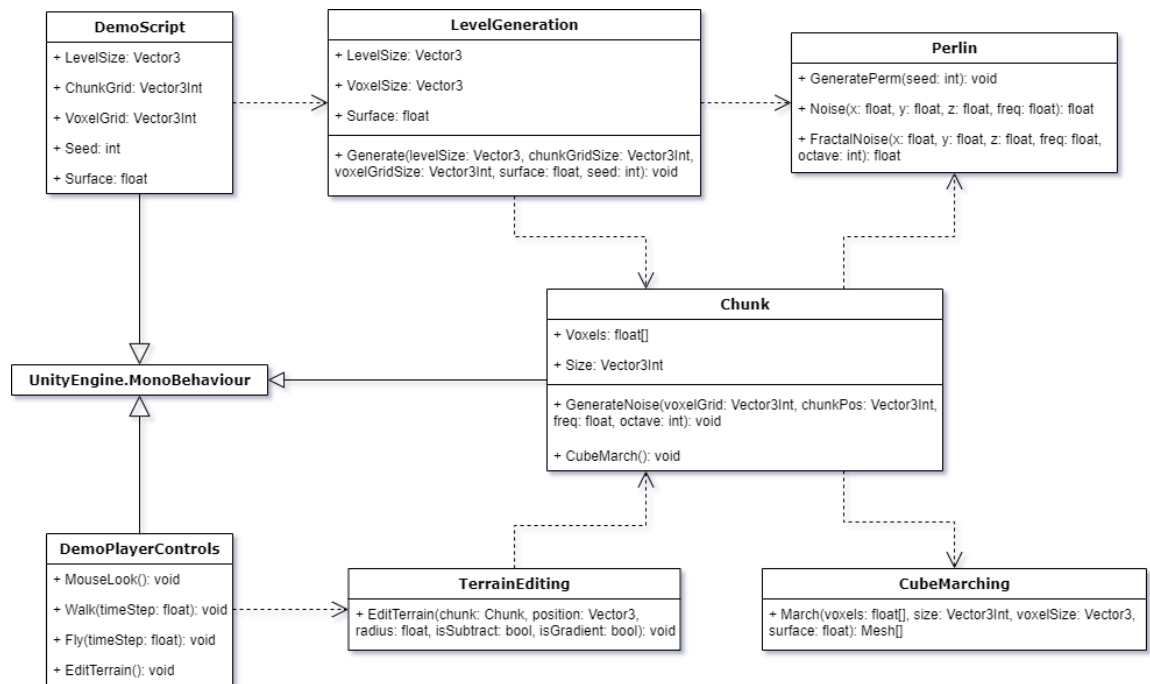
mukaan uniikin vokselitaulukon, joka kuitenkin yhdistyy vierekkäisten lohkojen taulukkoihin saumattomasti. Lohkojen vokselitaulukkojen luomisen ja niiden kuutiomarssittamisen väliin, milloin taulukot ovat yksinkertaisesti muokattavissa, voidaan ohjelmoida monimutkaisempi kenttägenerointilogiikka. Tässä vaiheessa kenttään voidaan ohjelmoidusti luoda arvoja kasvattamalla tai vähentämällä esimerkiksi käytäviä, huoneita tai seinämiä.

Kun lohkolle on ohjelmassa tässä kohtaa luotu mieluisa vokselitaulukko, voidaan se kuutiomarssittaa. Kuutiomarssitus käy lävitse jokaisen vokselitaulukon kahdeksan vierekkäisen pisteen muodostaman kuution ja tallettaa kulmien lukuarvojen mukaiset kolmiot muodostaen 3D-mallin. Jokaisen lohkon malleille voidaan asettaa materiaali, ja ne piirtyvät moottorissa pelin aikana muodostaen pelaajan havaitseman pelikentän.

Pelaaja voi generoinnin jälkeen liikkua kentässä kävellen tai vapaasti lentäen. Kenttä on kuvattu pelaajan hahmon ensimmäisestä perspektiivistä. Pelaajan on mahdollista muokata kenttää näennäisesti kaivamalla siihen kuoppia tai lisäämällä materiaalia olemassa oleville pinnoille. Muokkaaminen tapahtuu pelaajan painaessa toimintoa vastaavaa näppäintä samalla osoittaessa jotain kentän kohtaa. Tuolloin ohjelman koodissa tarkistetaan, minkä kentän lohkojen kanssa pelaajan toiminta on limittäin. Lohkojen, joihin toiminta ulottuu, vokselitaulukkoa muokataan pelaajan toiminnon kohdassa. Muokattu taulukko kuutiomarssitetaan uudelleen ja vanha malli korvataan uudella algoritmilla muodostetulla mallilla.

## 4.2 Toteutus

Pelidemo on toteutettu Unity-pelimoottorin versiolla 2019.4.10f1 ja sen lähdekoodi on C#-kieltä. Moottorin on tuottanut Unity Technologies, ja se on kehittäjien ilmaiseksi ladattavissa, mutta sen kaupallinen käyttäminen on rajattua. Moottori suo pelidemon kehittämiselle laajan viitekehyyksen, jonka ansiosta työssä voidaan keskittyä kuutiomarssitusalgoritmin toteuttamiseen. Keskeisiä Unity-moottorin työssä käytettyjä luokkia ovat *MonoBehaviour*, josta perivät moottorissa käytetyt skriptiluokat, matemaattisia vektoreita esittävät *Vector*-luokat, matemaattisia funktioita sisältävä *Mathf*-luokka ja polygoniverkolle moottorissa tarkoitettu *Mesh*-luokka. Työssä toteutetut luokat joko perivät tai käyttävät edellä mainittuja Unityn luokkia. Tässä kappaleessa on esitelty pelidemon teknistä toimintaa ja toteutusta esittelemällä sen luokkia. Kuva 10 esittää demon luokkakaaviota. [10.]



Kuva 10. Demon luokkakaavio

*DemoScript* on luokka, joka toimii Unity-moottorin graafisessa editorissa demon rajapintana, josta käyttäjä voi säätää kenttägeneroinnin parametreja. Käyttäjä pystyy säätämään generoitavan kentän koon lisäksi, kuinka moneen lohkoon se on jaoteltu ja mikä on lohkojen resoluutio. Resoluutio tarkoittaa, kuinka moneen kuutioon yksi lohko on kullakin akselilla jaettu. Ohjelman alussa luokka käynnistää kenttägeneroinnin parametrien mukaisesti perimänsä Unityn *MonoBehaviour*-luokan *Start*-funktion toteutuksessaan.

*DemoPlayerControls* on pelissä pelaajaa vastaavalle oliolle ohjelmoitu luokka, joka sisältää kaiken vuorovaikuttamisen logiikan. Pelaaja voi liikkua joko vapaasti lentämällä tai kävellen kentässä. Pelaaja ohjaa liikkumista WASD-näppäimin tai nuolinäppäimin. Lentäessään kontrollit imitoivat Unityn editorin kameran kontrolleja. Ctrl-näppäin liikuttaa pelaajaa alaspäin, välilyönti ylöspäin ja vaihtonäppäin moninkertaistaa pelaajan liikenopeuden. Kameran näkymä kääntyy hiiren liikkeillä. Pelaajan klikatessa vasenta hiiren painiketta ammutaan näennäinen säde pelaajan näkymän keskelle. Säteen osuessa kenttään kutsutaan *TerrainEditing*-luokan maastonmuokkausfunktiota, joka kaivaa osan kentästä säteen osumakohdasta. Hiiren oikea painike kasvattaa osumaan kohtaan maastoa.

*LevelGeneration* on staattinen luokka, jonka tarkoitus on luoda pelikenttä. Pelikentän luomisen parametreiksi tarvitaan sen haluttu koko, lohkojen määrä, lohkojen resoluutio, kuutiomarssituk-

nessa pintaa kuvaava arvo sekä satunnaisgeneroinnin siemenluku. Pinta-arvoksi voi valita liukulu-  
vun väliltä  $[-1, 1]$  ja se määrittää, ovatko lohkojen vokselitaulukon pisteet muodostuvan kentän  
pinnan sisä- tai ulkopuolella. Siemenlukua käytetään Perlin-kohinan tuottamiseen käytetyn luo-  
kan permutaatiotaulukon alustamiseen. Siemenlukua on tarkoitus käyttää kaiken satunnaislukuja  
tarvitsevan generoinnin satunnaislukugeneraattorissa. Luokan generointifunktiossa luodaan jo-  
kainen kentän lohko, jolloin kutsutaan niiden *GenerateNoise*- ja *CubeMarch*-funktioita. Näiden  
kahden funktioiden kutsumisen välissä lohkon vokseleita voidaan muokata erilaisen kenttä-  
generoinnin toteuttamiseksi. Generointifunktiossa myös luodaan Unity-moottorin *GameObject*-  
oliot kentälle ja sen lohkoille. Oliot tarvitaan, jotta lohkojen 3D-mallit voidaan piirtää mootto-  
rissa. [10.]

*Perlin* on luokka, joka on jatkettu Keijiro Takahashin avoimen lähdekoodin toteutuksesta [11].  
Luokan avulla voidaan tuottaa Perlin-kohinaa moniulotteisessa koordinaatistossa ja lisäksi yhdis-  
tää sen oktaaveja. Alkuperäistä luokkaa on jatkettu siten, että sillä tuotettavan kohinan taajuutta  
ja satunnaislukutaulukkoa voidaan muuttaa. Kohinan taajuuden muuttaminen tarkoittaa, että sen  
näennäisen koordinaatiston kokoa kasvatetaan tai kutistetaan. Luokan *GeneratePerm*-funktio luo  
257 kokonaislukua sisältävän taulukon satunnaislukugeneraattorilla, joka käyttää kenttä-  
generointiin syötettyä siemenlukua.

*Chunk*-luokka on kentän lohkoja kuvaava luokkaa, joka perii Unity-pelimoottorin käytännön syistä  
*MonoBehaviour*-luokan. Lohkon *GenerateNoise*-funktiossa luodaan resoluution mukainen tau-  
lukko. Jokaiseen resoluution akselin kokoon lisätään kokonaisluku yksi, sillä resoluutio vastaa  
muodostettavien kuutioiden määrää lohossa ja ilman lisäystä ääripäissä muodostettavat kuutiot  
jäävät paitsi vähintään neljää kulmaa. Koska jokaisen vokselin sijainti lohossa voidaan päätellä  
sen indeksistä lohkon koon ollessa vakio, tarvitsee taulukkoon tallettaa vain vokseleiden lukuarvo.  
Koodiesimerkissä 1 on esitetty, kuinka vokseleiden indeksi lasketaan sisäkkäisissä for-silmukoissa.  
Siinä iterointimuuttujat  $i$ ,  $j$  ja  $k$  kuvaavat vokseleiden järjestystä  $x$ -,  $y$ - ja  $z$ -akseleilla ja *size*-muut-  
tuja on kolmiulotteinen vektoriluokka, joka kuvaa lohkon vokseleiden määrää akseleilla. Funktion  
parametreiksi on myös annettu Perlin-kohinan taajuus ja yhdistettävien oktaavien lukumäärä.

```

for (int k = 0; k < size.z; k++)
{
    for (int j = 0; j < size.y; j++)
    {
        for (int i = 0; i < size.x; i++)
        {
            int index = i + j * size.x + k * size.x * size.y;

            voxels[index] = Perlin.Fbm(
                (float)i / (size.x - 1) + chunkPos.x,
                (float)j / (size.y - 1) + chunkPos.y,
                (float)k / (size.z - 1) + chunkPos.z,
                freq,
                octave);
        }
    }
}

```

Koodiesimerkki 1. Lohkon vokseleiden iteroiminen ja lukuarvojen asettaminen kohinasta.

Perlin-kohinan käyttämä satunnaislukutaulukko pysyy samana jokaisen lohkon generoinnissa yhdistäen ne toisiinsa saumattomasti. *Chunk*-luokan *CubeMarch*-funktiossa kutsutaan *CubeMarching*-luokan funktiota, jossa varsinainen kuutiomarssitus tapahtuu palauttaen taulukon Unity-moottorin polygoniverkko-olion. Marssittamisen jälkeen funktiossa luodaan lohkon *GameObject*-oliolle tarvittavien luokkien oliot sen polygoniverkkojen piirtämiseksi moottorissa ja sekä lisäksi kollisioiden laskemiseksi. Funktion lähdekoodi on kuvattu koodiesimerkissä 2.

```

public void CubeMarch()
{
    Mesh mesh = CubeMarching.March(voxels, size, voxelSize, surface);

    Transform nextMesh = new GameObject("Mesh").transform;
    nextMesh.SetParent(transform, false);

    MeshFilter filter = nextMesh.gameObject.AddComponent<MeshFilter>();
    filter.mesh = mesh;

    MeshRenderer renderer = nextMesh.gameObject.AddComponent<MeshRenderer>();
    renderer.sharedMaterial = new Material(Shader.Find("Standard"));

    nextMesh.gameObject.AddComponent<MeshCollider>();
}

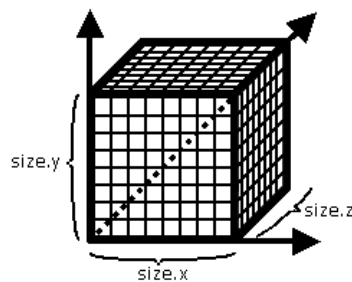
```

Koodiesimerkki 2. *Chunk*-luokan kuutiomarssitusfunktio

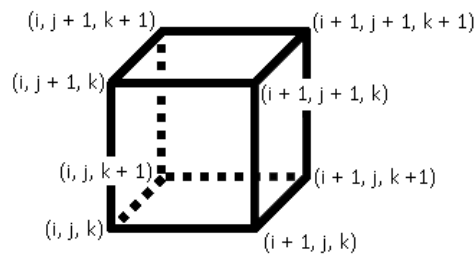
*CubeMarching*-luokka sisältää kuutiomarssitusalgoritmin toteutuksen. Kolmessa for-silmukassa iteroidaan lävitse jokainen funktiolle syötetystä vokselitaulukosta muodostettava kuutio. Koska taulukko ei sisällä tietoja vokseleiden sijainneista, funktioon on tämän vuoksi syötettävä myös taulukon ulottuvuudet. Kun kuution kulmien arvojen mukaan muodostetaan niitä vastaava kol-

miotaulukon osoite, voidaan jokaisen vokselin indeksi laskea samalla tavalla kuin koodiesimerkissä 1. Kahdeksanbittinen osoite muodostuu vertaamalla jokaisen kulman vokselin arvoa kenttägenerointiin parametriksi syötettyyn pinta-arvoon. Mikäli vokselin arvo on sitä suurempi, se on mallin ulkopuolella, ja sitä vastaava numero binääriluvussa saa arvon yksi. Kuvassa 11 on havainnollistettu, kuinka kuution kulmien indeksit voidaan ratkaista. Sitä seuraavassa koodiesimerkissä 3 muodostetaan binäärimuotoinen osoite, joka muutetaan kokonaisluvuksi.

Lohkon kattavat vokselit:



Vokseleista koostuva kuutio:



Yhden vokselin indeksi taulukossa:

$$\text{index} = i + j * \text{size.x} + k * \text{size.x} * \text{size.y}$$

Kuva 11. Kuution vokselien indeksien ratkaiseminen [1]

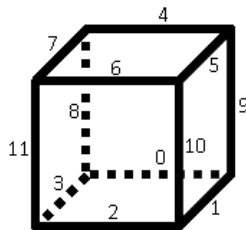
```
for (int k = 0; k < size.z - 1; k++)
{
    for (int j = 0; j < size.y - 1; j++)
    {
        for (int i = 0; i < size.x - 1; i++)
        {
            int index = i + j * size.x + k * size.x * size.y;
            int[] row_num = new int[1];
            BitArray row_Byte = new BitArray(8);

            row_Byte[0] = voxels[index] >= surface;
            row_Byte[1] = voxels[index + 1] >= surface;
            row_Byte[2] = voxels[index + size.x] >= surface;
            row_Byte[3] = voxels[index + 1 + size.x] >= surface;
            row_Byte[4] = voxels[index + size.x * size.y] >= surface;
            row_Byte[5] = voxels[index + 1 + size.x * size.y] >= surface;
            row_Byte[6] = voxels[index + size.x + size.x * size.y] >= surface;
            row_Byte[7] = voxels[index + 1 + size.x + size.x * size.y] >= surface;

            row_Byte.CopyTo(row_num, 0);
            ...
        }
    }
}
```

Koodiesimerkki 3. Kolmiotaulukon binäärisen osoitteen muodostaminen

Työssä on käytetty redundantin työn välttämiseksi Paul Brouken esimerkin mukaista kolmiotaulukkoa [2]. Brouken käyttämässä taulukossa kuution eri tapaukset on jaoteltu 256:lle eri riville, joissa sarakkeisiin on sijoitettu kokonaislukuja alkaen numerosta -1 ja päättyen numeroon 12. Kokonaisluvut vastaavat särmiä, jotka on indeksoitu Brouken käyttämällä tavalla, ja jota on havainnollistettu kuvassa 12. Luku -1 ei vastaa mitään särmää, vaan siihen kohtaan saavuttaessa iterointi lopetetaan. Tuolloin kuution tapauksessa ei ole enempää sen särmille sijoitettavia verteksejä.



```
private static readonly int[,] TriangleTable = new int[,]
{
    ...
    {2, 3, 11, 0, 1, 8, 1, 7, 8, 1, 5, 7, -1, -1, -1, -1},
    {11, 2, 1, 11, 1, 7, 7, 1, 5, -1, -1, -1, -1, -1, -1},
    {9, 5, 8, 8, 5, 7, 10, 1, 3, 10, 3, 11, -1, -1, -1, -1},
    ...
}
```

Kuva 12. Brouken tapa indeksoida kuution särmät ja pätkä kolmiotaulukon lähdekoodista [2]

Koska kolmiotaulukosta saatavat särmät on tunnettu, niiden päätepisteistä voidaan laskea 3D-mallia varten sijoitettavan verteksin sijainti. Päätepisteistä tiedetään kuvan 11 tapaan tapauskohtaisesti vokselien sijainti ja lukuarvo. Sijoitettavan verteksin todellinen sijainti mallissa ratkaistaan interpoloimalla, siten että sen etäisyyksien särmän vokseleihin suhde on yhtä suuri kuin vokselien lukuarvojen suhde. Koodiesimerkissä 4 on kuvattu kolmiotaulukon sarakkeiden iteroimisen lisäksi verteksin sijoittamisen lähdekoodia.



```

for (int col_num = 0; col_num < TriangleTable.GetLength(1); col_num++)
{
    int edgeIndex = TriangleTable[row_num[0], col_num];

    if (edgeIndex < 0)
        break;

    Vector3 a;
    Vector3 b;
    float valueA;
    float valueB;

    switch (edgeIndex)
    {
        case 0:
        {
            a = new Vector3(
                i * voxelSize.x,
                j * voxelSize.y,
                (k + 1) * voxelSize.z);
            b = new Vector3(
                (i + 1) * voxelSize.x,
                j * voxelSize.y,
                (k + 1) * voxelSize.z);
            valueA = voxels[index + size.x * size.y];
            valueB = voxels[index + 1 + size.x * size.y];
            break;
        }
        case 1:
        ...
    }

    Vector3 vertice = a + (surface - valueA) * (b - a) / (valueB - valueA);

    verticeList.Add(vertice);
    triangleList.Add(verticeList.Count - 1);
}

Mesh mesh = new Mesh();

mesh.vertices = verticeList.ToArray();
mesh.triangles = triangleList.ToArray();

return mesh;

```

Koodiesimerkki 4. Kolmiotaulukon sarakkeiden iterointi ja verteksin sijainnin interpolointi sekä sijoittaminen verteksi- ja kolmiolistoihin

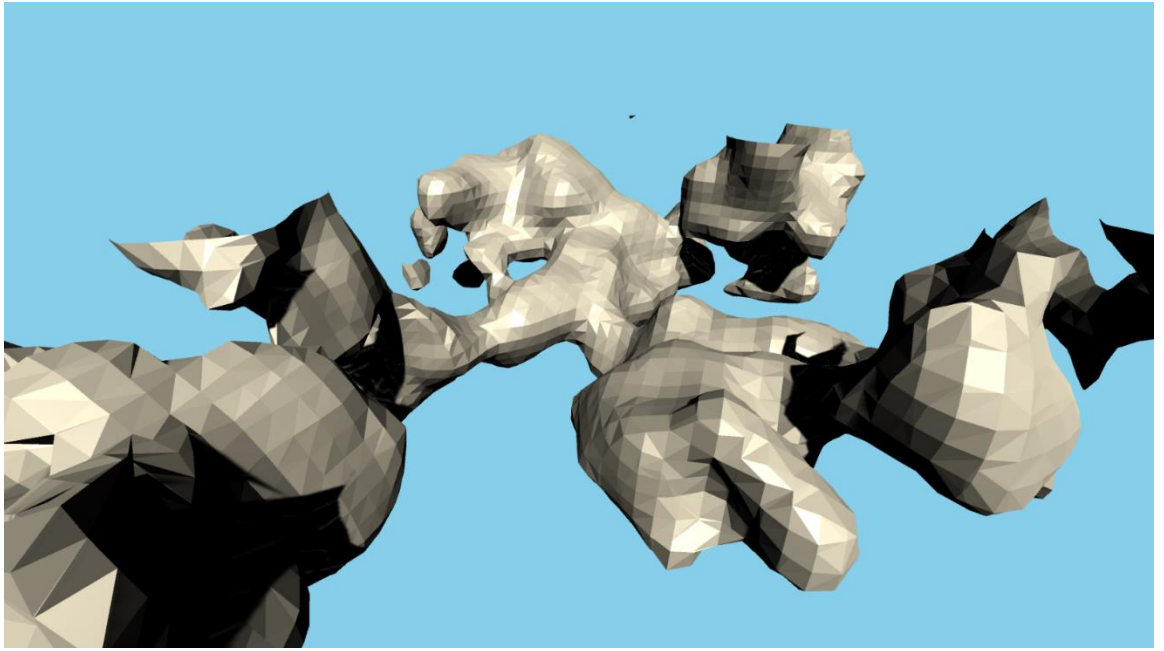
Tämän jälkeen verteksi ja sen indeksi voidaan tallentaa polygoniverkon verteksi- ja kolmiotaulukoon Unityn *Mesh*-luokkaan. 3D-mallin kolmiotaulukko sisältää tiedon verteksin indekseistä verteksitaulukossa, joista mallin kolmiot muodostuvat kolmen kimpuissa. Mallin kolmioiden normaalit on laskettu käyttämällä *Mesh*-luokasta löytyvää *RecalculateNormals*-funktiota, joka aset-

taa jokaisen kolmion verteksien normaaliksi suoraan niiden jakaman kolmiulotteisen tason pinnannormaalini. *CubeMarch*-funktio palauttaa lopuksi näin muodostetun polygoniverkkoluokan. [10.]

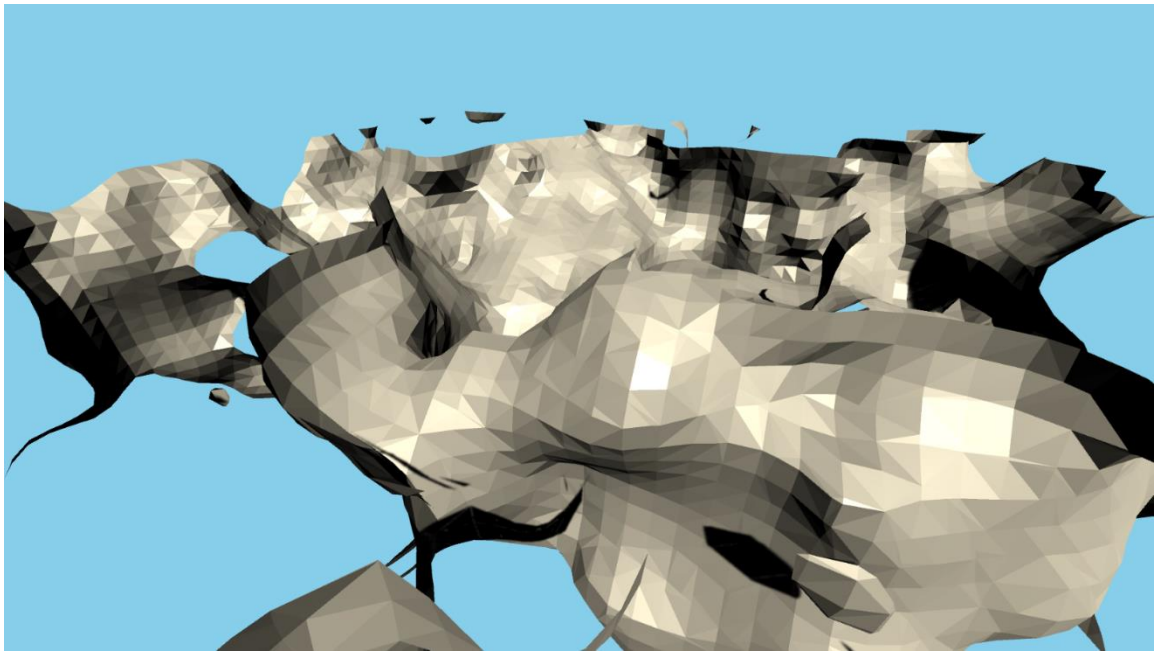
*TerrainEditing*-luokkaa käyttää *DemoPlayerControls*-luokka muokatakseen kuutiomarssittamalla muodostettua kenttää. Pelaajaluokka kutsuu luokan *EditTerrain*-funktioita, jossa muokataan lohkoja, joihin pelaajan tekemät muokkaukset ylettyvät. Muokkauksen parametreiksi on annettu sen sijainti, koko, tieto onko se lisäävä vai vähentävä ja käytetäänkö muokkaukseen gradienttia. Kolmessa sisäkkäisessä for-silmukassa vertaillaan vokseli kerrallaan, yltääkö muokkaus niihin, ja muokkauksen ylettyessä niihin lisäävässä muokkauksessa vähennetään vokselin lukuarvoa. Vähentävässä muokkauksessa lukuarvoa päinvastoin kasvatetaan. Vokselin arvo on asetettu vaihtelemaan välillä -1 ja 1, jossa asetetun pinnan lukuarvoa suurempi luku on kentän mallin pinnan ulkopuolella. Gradienttia käyttämällä voidaan tehdä näennäisen sileitä muokkauksia, joissa pallon muotoisen muokkauksen pinnalla olevien vokseleiden arvon muutos on sitä pienempi, mitä kauempana ne sijaitsevat muokkauskohdasta. Näin estetään uutta mallia muodostettaessa pinnan sisä- ja ulkopuoleisten arvojen suhteen kasvaminen liian suureksi. Lukuarvojen suhteen kasvaessa liian suureksi malli menettää Perlin-kohinasta saadun luontaisen ulkonäkönsä ja pinnasta tulee etäisyysiltään yhtenäinen. Vokselitaulukon muokkaamisen jälkeen lohko kuutiomarssitetaan uudelleen ja kentän vanha polygoniverkko korvataan uudella mallilla.

#### 4.3 Testaus

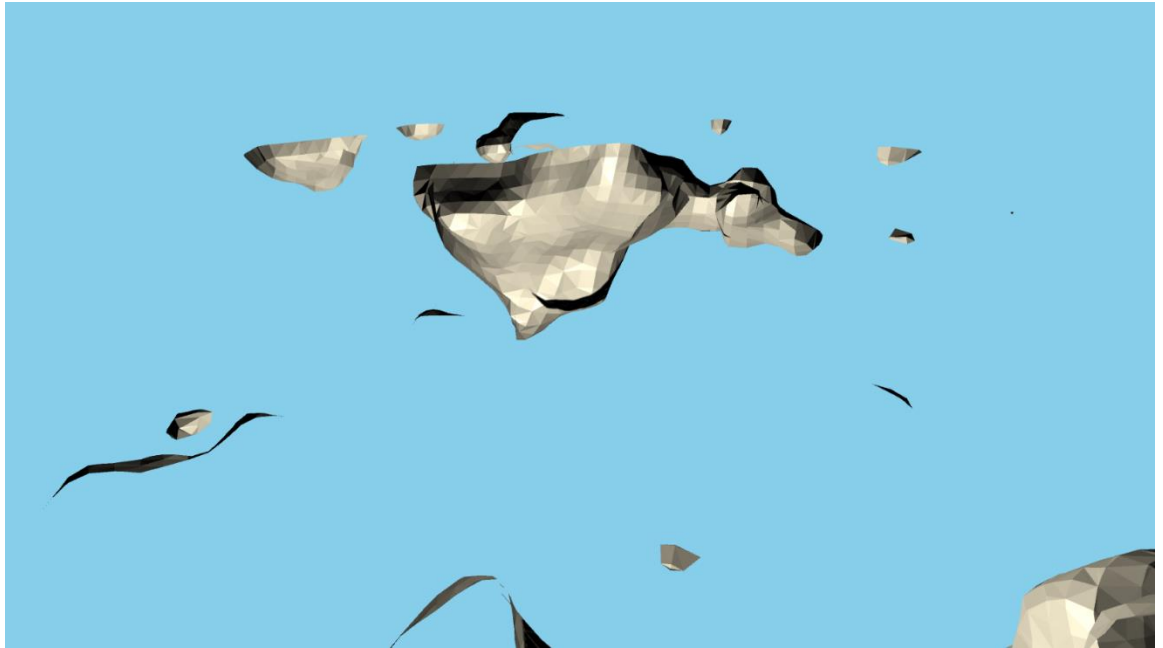
Työssä tuotetussa pelidemossa pystytään generoimaan kentän muodostavia 3D-malleja lohkojen vokselitaulukoista, joissa lukuarvot generoidaan Perlin-kohinasta. Kenttägenerointi käyttää siemenlukua, joten sillä voidaan generoida täysin samanlainen pelikenttä lukuisia kertoja. Generoitavan kentän kokoa, lohkojen määrää sekä lohkojen resoluutiota voidaan säätää vapaasti eri kokoisten ja tarkkuudeltaan vaihtelevien kenttien luomiseksi. Kenttägeneroinnin lopputulokseen vaikuttaa eniten valittavan siemenluvun lisäksi kuutiomarssitukseen käytetty pinta-arvo. Pienissä kentissä, jotka koostuvat vain muutamasta hillityn kokoisesta lohkoista, generointi tapahtuu murto-osasekunneissa. Kuvat 8, 9 ja 10 havainnollistavat muuten identtisin parametrein generoituja kenttiä, joissa on käytetty eri pinta-arvoja.



Kuva 13. Pinta-arvolla -0.2 generoitu kenttä sisältää pitkulaisia muotoja

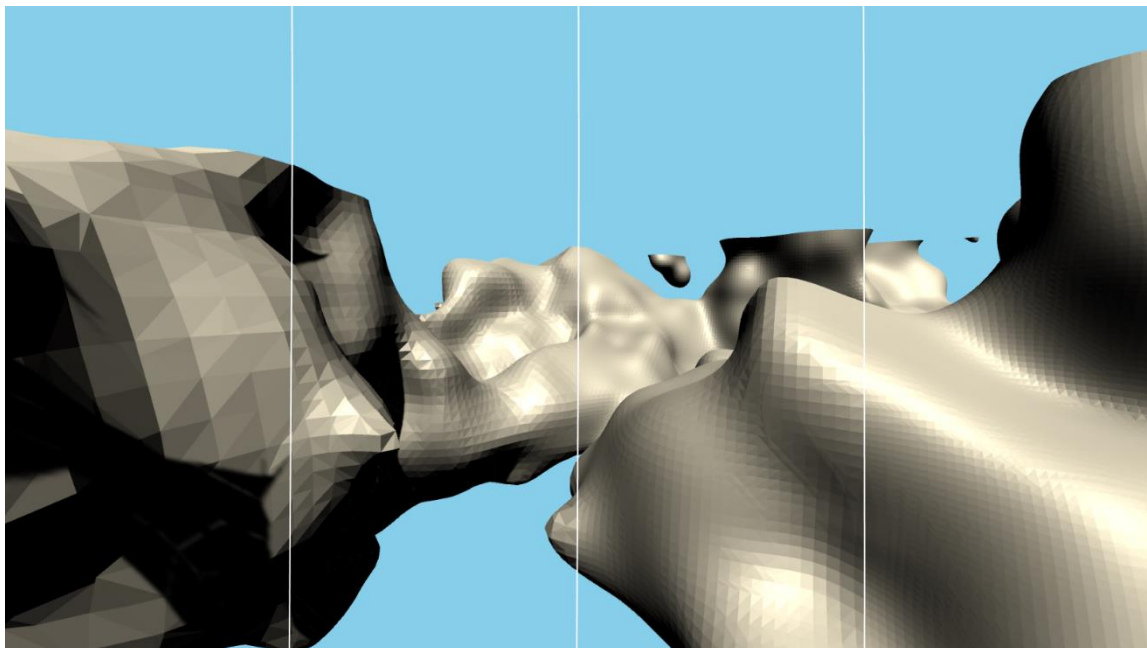


Kuva 14. Pinta-arvolla 0 generoidussa kentässä on lähes yhtä paljon tilaa muodostuvien mallien sisä- kuin ulkopuolellakin



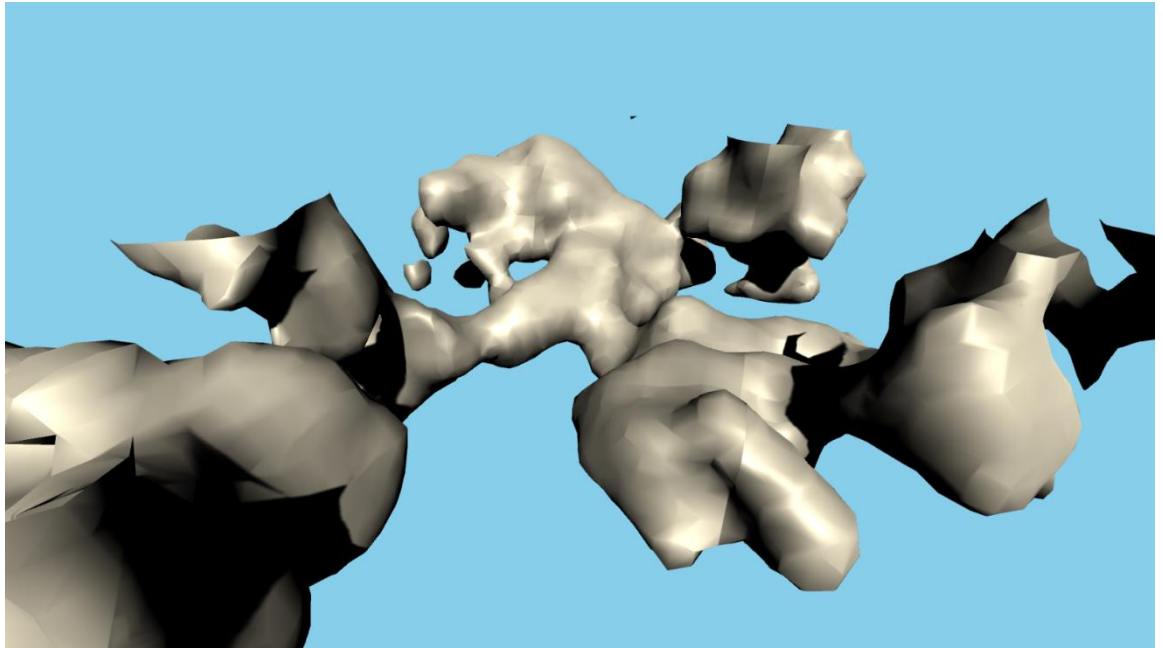
Kuva 15. Pinta-arvolla 0.2 generoituva kenttä on suurimmaksi osaksi mallin sisällä muodostaen vain luolia, joiden seinien takaa näkee sisään

Kentän malleista voidaan nähdä lisäksi selkeästi generoinnissa käytetyn lohkojen resoluution vaikutus 3D-mallien pinnan ulkoasuun. Mitä tiheämmin Perlin-kohinasta arvonsa saavia vokseleita lohkoissa on, sitä yksityiskohtaisemmaksi malli muuttuu, samalla kasvattaen kenttägeneroinnin ajallista kestoa. Mikäli lohkojen resoluutiota kasvatetaan kolmeen suuntaan yhtä paljon, kasvaa vokseleiden määrä taulukoissa muutoksen kuutiona. Todella yksityiskohtaisen ja hienojakoisen kentän generointi kestää huomattavasti kauemmin ja varaa enemmän muistia. Kuvasta 16 näkee tarkemmin resoluution vaikutuksen kentän polygonimallien pintaan.



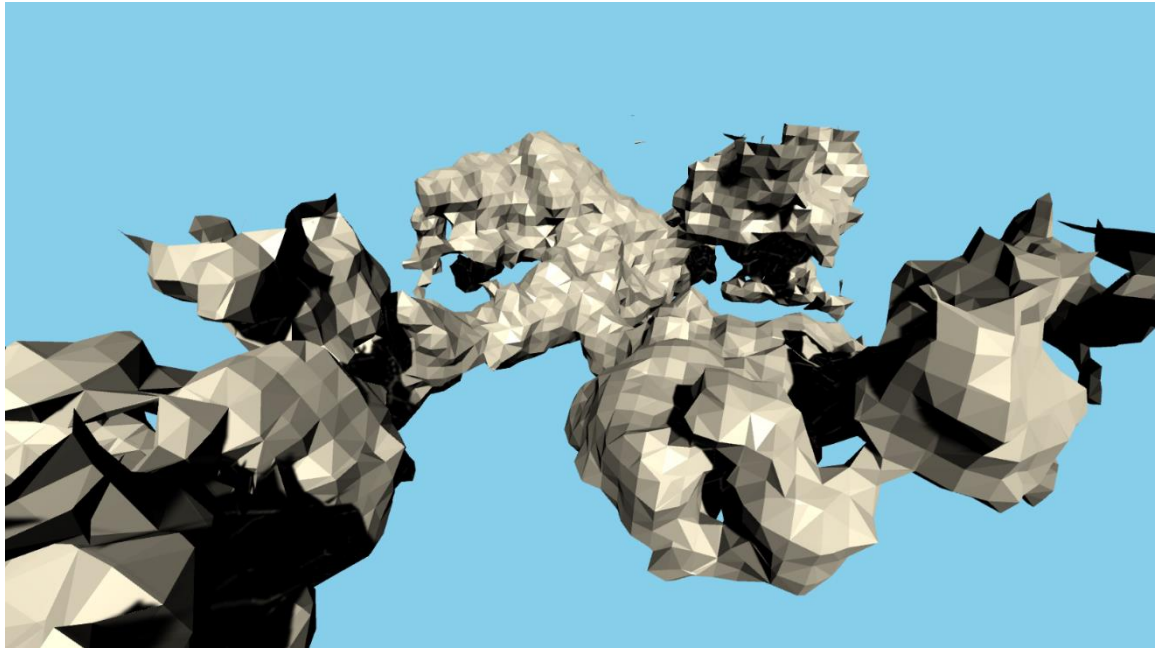
Kuva 16. Vertailu lohkojen resoluution vaikutuksesta malliin, jossa vasemmalta oikealle vokselien määrää lohkoissa on kasvatettu askeleittain kahdeksankertaiseksi

Mikäli kuutiomarssituksella halutaan luoda kenttään tasaisen näköisiä pinnan muotoja kasvattamatta kenttägeneroinnin kestoa sietämättömän suureksi, voidaan mallin normaalien määrittämistä muuttaa. Sijoittamalla mallin vertekseihin yhden kolmion pinnannormaalien sijaan verteksin viereisten kolmioiden pinnannormaalien keskiarvo voidaan luoda Unity-moottorissa piirtyvä tasoitettu malli. Mallin pinnasta on vaikeampi erottaa kuutiomarssituksessa syntyvä ominainen kuvio. Kuitenkin, koska lohkojen äärikohdissa olevat verteksit eivät polygoniverkoissaan ole vierekkäisen lohkojen kolmioiden vieressä, ilmestyvät lohkojen saumakohdat näkyviin generoiduissa malleissa. Tasoittamalla kenttä voi näyttää subjektiivisesti mielekkäämmältä, mutta käyttämättä tasoitusta lopputulos näyttää yhtenäisemmältä ja mallin pinnassa näkyvä ruudukko voi avustaa pelaajaa syvyyšnäköä. Tasoituksen vaikutuksen voi nähdä kuvasta 17.



Kuva 17. Pinta-arvolla -0.2 generoitu kenttä, jonka mallit on tasoitettu ja jonka lohkojen sauma-kohta näkyy kuvassa oikealla

Kentän koon lisäksi demossa käytettyä Perlin-kohinaa muokkaamalla voidaan vaikuttaa kentän ulkoasuun. Perlin-kohinan taajuutta muuttamalla muuttuu kohinan näennäinen koko. Tämä vaikuttaa kenttään siten, että yksityiskohdat muuttavat kokoaan. Kun kenttä kuutiomarssitetaan tavallaan pienemmästä versiosta itsestään muuttamatta lohkojen resoluutiota, lopputuloksesta häviää pieniä yksityiskohtia. Lisäämällä oktaavien määrää kohinassa kohinaan tulee paljon lisää pienempiä yksityiskohtia samalla muuttamatta suuria muotoja. Kuvaa 18 tarkastelemalla voi nähdä, kuinka oktaavien lisääminen kohinaan johtaa muuttumattomassa lohkon resoluutiossa rosoisempiin pintoihin.



Kuva 18. Pinta-arvolla -0.2 generoitu kenttä, jossa Perlin-kohinaan on lisätty kolme oktaavia lisää

Kuutiomarssituksen verteksien interpoloiminen kasvattaa kenttägeneroinnin kestoja, mutta se ei lisää malliin varattavan muistin määrää. Verteksien tallentaminen vie saman verran tietokoneen muistia riippumatta sen sijainnista mallissa, kunhan niiden määrä pysyy samana. Interpoloinnin voi jättää tekemättä prosessorilta vaaditun laskentatehokkuuden vähentämiseksi tai muuten vain tyylisyistä. Interpoloimattomassa kentässä on paljon selkeämmin nähtävissä lohkon vokseleiden sijainnit ja se on pintojen etäisyyksiltään täysin yhtenäinen. Kuvassa 19 on demonstroitu, miltä kenttä, jonka verteksejä ei ole interpoloitu kohinan arvojen mukaisesti, näyttää.



Kuva 19. Pinta-arvolla -0.2 generoitu kenttä, jonka verteksien sijaintia kuution särmillä ei ole interpoloitu

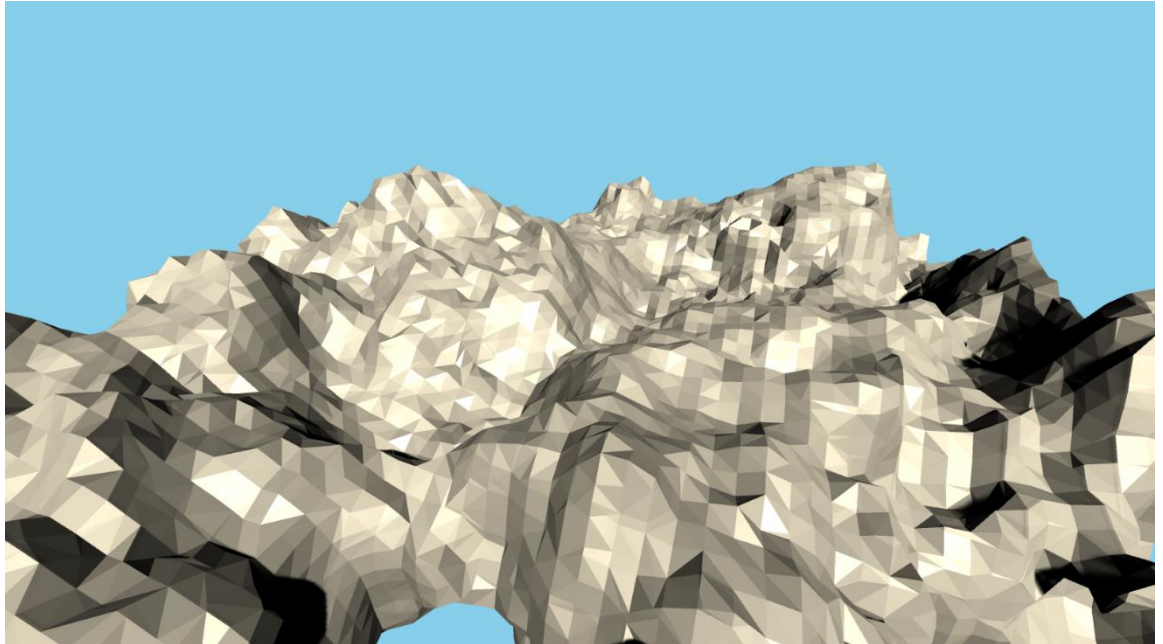
#### 4.3.1 Maaston generointi

Lohkon vokselitaulukon alustamisen ja sen kuutiomarssittamisen välissä kehittäjä voi mielivaltaisesti ohjelmoida taulukkoon tehtäviä muokkauksia. Yksi mahdollisista muokkauksista on synnyttää lohkojen päälle kiinteä maanpinta. Perlin-kohina toimii oivallisena keinona generoida kentän kattava korkeuskartta. Kuutiomarssitus algoritmilla voidaan saavuttaa huomattava hyöty yhdistämällä aikaisempi 3D-kohina kaksikulotteiseen korkeuskarttaan. Tällä tavoin voidaan luoda korkeuskarttaa noudattava maanpinta, jonka alle peittyvät kohinasta peräisin olevat maanlaiset muodot. Muokatessa generoidun vokselitaulukon lukuarvoja on tärkeää säilyttää alkuperäisen kohinan ominaisuus, jossa vierekkäiset vokselit sisältävät toisiaan lähellä olevia lukuarvoja. Mikäli kohina näennäisesti katkeaa, kuutiomarssituksessa tapahtuva interpolointi lakkaa luomasta luonnollisen näköisiä sulavia muotoja. Vierekkäisten vokseleiden arvojen suhde verteksejä sijoitettaessa voi kasvaa erityisen suureksi, jolloin kohinan katkeamiskohtaan generoituu etäisyysiltään yhtenäinen leikkaus. Leikkauskohta muistuttaa kenttägeneroinnin tulosta ilman interpolointia, joka on esitetty aikaisemmin kuvassa 19.

Maanpinnan generointia voi demossa testata vertaamalla kaksikulotteisen Perlin-kohinan lukuarvoja vokseleiden korkeuteen lohossa. Jokaiselle kentän kohdalle syntyy maanpinnan korkeuden



lukuarvo. Vokseleiden korkeutta voidaan verrata maanpinnan korkeuteen. Mitä korkeammalla vokselit ovat maanpinnasta, sitä enemmän niiden lukuarvoa kasvatetaan. Tällä tavoin säilytetään Perlin-kohinan luonnolliset muodot, sillä kentän pinnan viereisten vokselien lukuarvot jäävät lähelle toisiaan. Kuvassa 20 on esitetty demossa kokeiltua maanpinnan generointia.



Kuva 20. Kolmi- ja kaksiulotteisia Perlin-kohinoita yhdistävä kenttägenerointi

Maanpinnan generoinnilla voidaan saada aikaan suuria maisemia, joiden värittämiseksi demossa on kokeiltu ratkaista 3D-mallien vertekseille uv-koordinaatteja. Uv-koordinaatit esittävät kaksiulotteista sijaintia tekstuurikartalla ja vaihtelevat välillä  $[0, 1]$ . Ne osoittavat polygonimalleja tietokoneohjelmissa piirrettäessä, mitkä kohdat tekstuureista piirtyvät mallin kolmioissa. Demossa on käytetty kuvan 21 mukaista tekstuuria, joka sijoitetaan kentän materiaaliin ja josta verteksit voivat saada eri värejä korkeutensa mukaan kentässä.



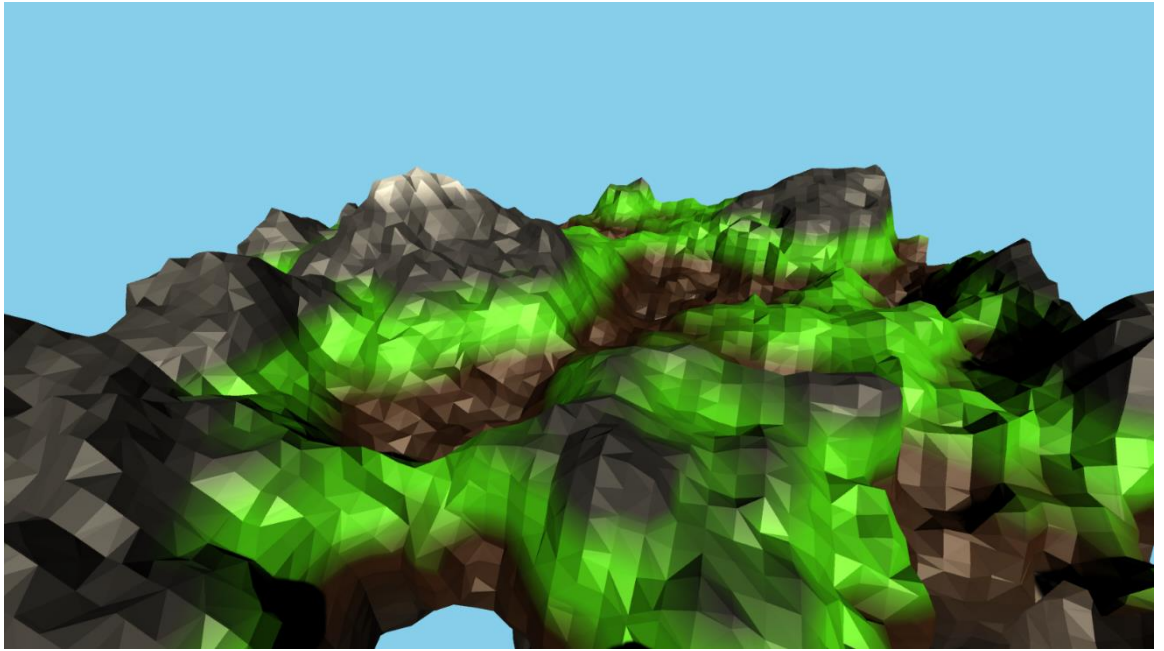
Kuva 21. Maaston väriin käytetty tekstuuri

Kuutiomarssitettaessa vokselitaulukkoa verteksien korkeus on tunnettu. Sitä vertaamalla generoitavan kentän korkeuteen voidaan ratkaista verteksin uv-koordinaatti, josta verteksi saa värinsä korkeutensa perusteella. Koodiesimerkissä 5 on esitetty uv-koordinaatin lisääminen polygoniverkon tietorakenteeseen tallettavaan listaan kuutiomarssituksessa. Koordinaatit liitetään Unityn *Mesh*-luokkaan kuten verteksit ja kolmiotkin. Uv-koordinaattien ja tekstuurin käyttämisen vaikutuksen voi nähdä kuvasta 22.

```
...
uvList.Add(new Vector2(0.5f, Mathf.Lerp(0, 1, vertice.y / levelSize.y)));
...
Mesh mesh = new Mesh();

mesh.vertices = verticeList.ToArray();
mesh.triangles = triangleList.ToArray();
mesh.uv = uvList.ToArray();
...
```

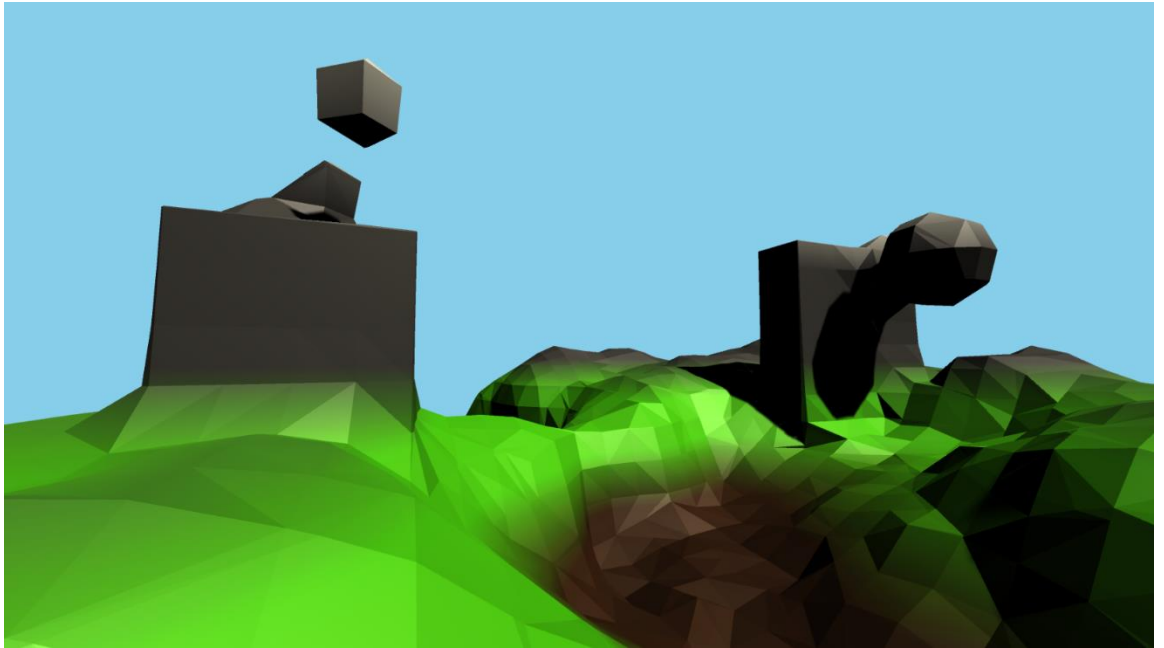
Koodiesimerkki 5. Verteksin uv-koordinaatin interpoloiminen sen korkeuden mukaan kentässä



Kuva 22. Kenttägeneroitu maasto, jonka vertekseille on annettu uv-koordinaatit korkeutensa mukaan

#### 4.3.2 Etäisyysfunktiot

Myös näennäisesti yksinkertainen proseduraalinen kenttägenerointi on mahdollista ennen vokselitaulukon kuutiomarssittamista. Kehittäjä voi lisätä kenttägenerointiin mielivaltaisesti monimukaisempaa generointilogiikkaa. Demossa on kokeiltu lisätä kenttään primitiivimuotoja etäisyysfunktioilla, joissa tarkistetaan, sijaitsevatko lohkojen vokselit muotojen sisäpuolella. Tarkistuksen jälkeen koodissa voidaan lisätä tai poistaa kentästä näitä yksinkertaisia muotoja. Kuvassa 23 kenttägeneroinnissa lohkoihin on lisätty kuusitahokkaita sekä lohkojen resoluutioon nähden pieniä palloja.



Kuva 23. Kenttägenerointi, joka tuottaa maanpinnan lisäksi kenttään primitiivimuotoja

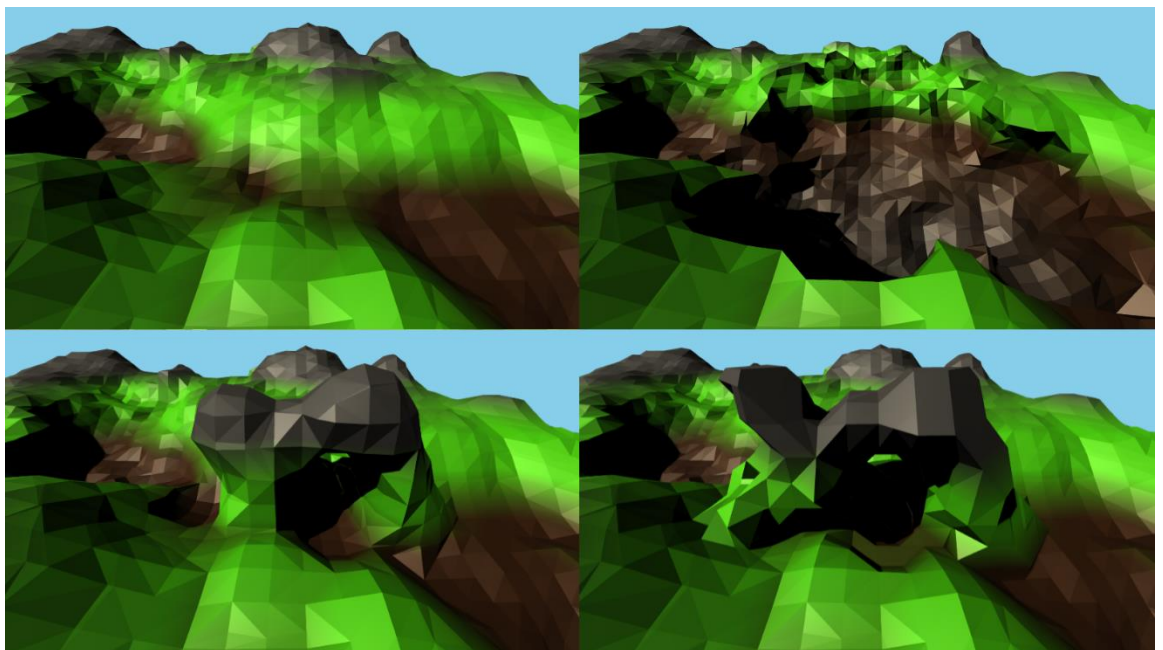
Etäisyysfunktioilla voidaan primitiivimuotojen lisäämisen tai poistamisen lisäksi rajata mielivaltaisia osia kentistä niiden yhdistelemiseksi tai irrottamiseksi muusta kentästä kokonaan. Säilyttämällä vain kentän kokoisen pallon sisällä olevat lukuarvot voidaan generoida esimerkiksi kuvan 24 mukaisia mielenkiintoisia planeettoja pelaajan tutkittavaksi.



Kuva 24. Kolmiulotteisesta Perlin-kohinasta erotettu pyöreä planeetta

#### 4.3.3 Maaston muokkaus

Demoon ohjelmoidun maaston muokkaamisen ansiosta pelaaja voi tehdä muutoksia generoituun kenttään. Demossa on mahdollista säätää asetusta, joka päättää, toimiiko maaston muokkaus siirtämällä vokseleita ääriarvoja kohti pelaajan osoittamassa kohdassa, vai asettaako se ne suoraan ääriarvoihin -1 tai 1. Vokseleiden arvoja vaihtamalla katkeaa taulukossa Perlin-kohinalle ominainen jatkuvuus, jolloin katkeamiskohdan verteksien etäisyydet muuttuvat yhtenäisiksi, mikä ei ole välttämättä kehittäjän tarkoitus. Kuvassa 25 on esitetty kohinan ominaisuuden säilyttävän maaston muokkauksen sekä suoraan ääriarvoihin asettavan muokkauksen toimintaa.



Kuva 25. Muokkaamaton maasto (vas. ylä), pelaajan kaivama maasto (oik. ylä), pelaajan lisäämä maasto vokseleiden arvoja siirtämällä (vas. ala) sekä suoraan ääriarvoihin asettamalla (oik. ala)

Demossa pelaaja kykenee vapaasti kaivautumaan maan pinnan alapuolelle, kuten on tehty kuvassa 26, tai luomaan maastoon lohkojen resoluutiota noudattavia muotoja. Pelaajan muokkauksien on kuitenkin pysyttävä kentän ja sen lohkojen rajaamalla alueella, jotta ne tallentuisivat vokselitaulukkoihin.



Kuva 26. Pelaajan kaivama maanalainen tunneli, joka on valaistu spottivalolla

Mikäli lohkojen vokselitaulukot ovat suuria, maaston muokkaaminen voi kestää odottamattoman kauan. Varsinkin jos muokkaus tapahtuu usean lohkon yhtymiskohdassa. Muokatessa koodissa on iteroitava uudelleen jokaisen muokattavan lohkon vokselitaulukot. Koska demossa algoritmit lasketaan tietokoneen prosessorilla kutsumalla funktioita Unityn *MonoBehaviour*-luokan *Update*-funktioista, ruudunpäivitys pysähtyy funktion toiminnan ajaksi. Pelaajalle tämä käytännössä ilmenee epätasaisena tökkivänä ruudunpäivityksenä, mikä on harvoin mielekäästä. Tökkimistä voidaan hillitä pitämällä lohkojen ja muokkauksien koot pieninä, optimoinnilla tai kasvattamalla prosessorin laskentatehoa.

## 5 Tulokset

Työssä tuotettu pelidemo esittää kuutiomarssituksen käyttämisen vahvuuksia kenttägeneroinnissa, mutta siitä on nähtävissä myös tekniikan heikkoudet. Mahdollisuus generoida näennäisesti loputon määrä pelikenttiä vain muutamasta käyttäjän syöttämästä numerosta on innoittavaa, mutta ilman pelisuunnittelua ja hienompia kenttägeneroinnille asetettuja sääntöjä se tuskin onnistuu luomaan kenttiä, jotka tarjoavat välittömästi hauskaa tai mielenkiintoista pelattavuutta. Seuraavissa kappaleissa on pohdittu vahvuuksien jatkokehitysmahdollisuuksia ja tapoja puuttua käytetyn tekniikan heikkouksiin.

### 5.1 Onnistumiset ja mahdollisuudet

Työssä luotu demo on hyvin skaalautuva ja se mahdollistaa kehittäjän ihanteellisten asetusten etsimisen kenttägenerointia varten. Generoitavan kentän koko, resoluutio sekä kuutiomarssituksessa käytetty pinta-arvo ovat määritettävissä kehittäjän mielivaltaisesti. Kenttää määrittävän Perlin-kohinan taajuutta muuttamalla voidaan vaikuttaa generoituvien yksityiskohtien kokoon ja sitä yhdistämällä niiden määrään. Demon kenttägenerointi noudattaa pseudosatunnaisuutta siemenlukua hyödyntäen. Kehittäjä voi nähdä parametrien muuttamisen vaikutuksen samaan kenttään pitämällä useiden generointikertojen ajan siemenluvun vakiona. Tallentamalla siemenluku pelaajatkin voivat generoida mielekkäitä kenttiä toistuvasti. Kuutiomarssitusalgoritmin generoimissa yhdenmukaisia 3D-malleja lohkoista vokseleiden muokkaaminen mahdollistaa loputtomiin tapoja muuttaa kenttägeneroinnilla luotavia kenttiä.

Kuutiomarssitus toimii hyvin tehokkaasti demossa generoitavien lohkojen pysyessä hillityn kokoisena, ja sitä voidaan käyttää sen vuoksi myös reaaliaikaisesti kentän muokkaamiseen. Kentän muokkaamiseksi tarvitaan vain tieto, missä muokkaus tapahtuu ja kuinka suuri se on. Esimerkiksi moninpelissä siitä on merkittävää hyötyä, sillä tämä tieto voitaisiin lähettää nopeasti verkon ylitse nettipelissä kokonaisen muokatun lohkon sijaan, ja pelaajat voisivat nähdä toistensa tekemät muokkaukset nopeasti. Jokaisen peliin liittyneen pelaajan kenttä kuutiomarssitettaisiin heidän omilla laitteillaan, säästäten verkkokaistan määrää muille pelin lähettämille tiedoille.

Lohkojen vokselitaulukot on talletettu muistiin todella yksinkertaisella tavalla, jossa taulukossa on vain vokseleiden lukuarvot. Vokseleiden sijainti voidaan päätellä kentän lohkojen koosta, niin

kauan kuin taulukot iteroidaan aina samalla tavalla, jossa jokaista kohtaa vastaa sama vokselitaulukon indeksi. Taulut sisältävät kuitenkin paljon dataa, ja niitä ei voi olla ladattuna tietokoneen muistiin rajattomasti. Niitä tarvitaan pelissä suorituksen aikana vain, jotta maaston muokkaaminen olisi mahdollista, joten ne poistamalla voidaan vapauttaa tietokoneen työmuistia kentissä, joita ei voida muokata pelin aikana.

Vokselitaulukkoon liitettäviä ominaisuuksia pystyisi laajentamaan lisäämällä vokseleihin enemmän dataa. Taulukon kohtiin voitaisiin tallentaa pinta-arvon lisäksi ominaisuuksia, kuten mistä materiaalista vokseli on tehty, jolloin sille asetettaisiin kuutiomarssituksessa materiaalia vastaava tekstuuri tai väri. Vokseleihin tiedon lisääminen kuitenkin moninkertaistaisi vokselitaulukoiden vaatiman muistin määrän lohkojen koon mukaisesti, mutta muistia voitaisiin myös säästää vähentämällä pinta-arvon numeron tarkkuutta. Demossa vokselin arvo on talletettu 32-bittiseen liukulukuun, johon se voidaan tallettaa jopa miljoonasosien tarkkuudella. Näin suuri tarkkuus voi olla tarpeetonta pinta-arvon vaihdellessa lukuarvojen -1 ja 1 välillä.

Kentän data olisi mahdollista kirjoittaa tiedostoon, josta niitä voitaisiin ladata tietokoneen työmuistiin tarvittaessa. Muistia voitaisiin tällä tavoin vapauttaa pelaajan ollessa riittävän kaukana ladatusta lohkoista. Ratkaisu mahdollistaisi todella suurien ja muokattavien kenttien generoimisen, joista työmuistiin ladattaisiin aina vain lohkoja, jotka olisivat pelaajan välittömässä läheisyydessä. Ratkaisussa kerralla ladattuna olevan kentän lohkojen määrää voitaisiin säätää pelaajan tietokoneen muistin määrän mukaan. Todella tehokkailla tietokoneilla voitaisiin ladata kerralla todella laakeita maisemia. Kentät voisivat olla myös näennäisen loputtomia ja niiden kokoa rajoittaisi ainoastaan pelaajan kovalevyn koko, jolle kirjoitettaisiin jatkuvasti kasvavan kentän data.

Vokselitaulukot ovat rajattoman muokattavissa ennen kuutiomarssittamista, mikä mahdollistaa kehittäjälle lukemattomasti erilaisia tapoja toteuttaa kenttägenerointi. Pienien työssä esitettyjen muokkausten lisäksi kehittäjä voisi lisätä vierekkäisiä vokseleista tietoisia muokkauksia ja efektejä, joissa esimerkiksi taitetaan, vääristetään tai peilataan lohkojen taulukkoja. Vokseleita voidaan muokata hyvin samaan tapaan kuin pikseleitä kuvankäsittelyssä niiden alkuperäisiä arvoja hävittämättä. Pelin kehittäjän kekseliäisyys voi olla ainut rajoittava tekijä mahdollisissa muokkauksissa, sillä niistä suurin osa tehtäisiin pelissä vasta kenttää luodessa, jolloin pelaajan ei tarvitse olla vuorovaikutuksessa sen kanssa. Kuitenkin laajat pelin aikana tapahtuvat muokkaukset voisivat tuottaa mielenkiintoista pelattavuutta, mutta niiden tulisi tapahtua riittävän tehokkaasti, jotta ne eivät haittaisi pelikokemusta.



Perlin-kohinan avulla taulukkoja alustettaessa saadaan aikaiseksi luonnollisia ja mielenkiintoisen näköisiä aihioita kenttägeneroinnille. Kohinan yhdennäköisyys realistista maastoa generoitaessa vuoriin ja rotkoihin on kuitenkin tavallaan sattumanvaraista, ja se ei synnytä oikeasta maailmasta löydettäviä asioita. Joet sekä korkeat kalliot voivat syntyä kenttään vain täysin sattumalta. Kehittäjän on vokselitaulukoiden yksinkertaisen muokkaamisen ansiosta kuitenkin täysin mahdollista ohjelmoida logiikka, joka pystyy luomaan luonnonkauniita maisemia, jotka simuloisivat tuulen ja veden aiheuttaman eroosion sekä maan aineksien liikkumisen vaikutuksia kenttägeometriassa.

Koska kuutiomarssituksen aikana verteksien sijainnit ovat tunnettuna, monimutkaisempien tekstuurien asettaminen malleihin on mahdollista. Tämä tarkoittaisi verteksien uv-koordinaattien läpikohtaisempaa ratkaisemista. Kuutiomarssituksessa generoituvat kolmiot voivat olla monissa erilaisissa asetelmissa, koska kuutioilla on lukuisia niitä vastaavia eri tapauksia algoritmin kolmio- taulukossa. Koska tapaukset ovat kuitenkin hyvin rajattuja ja ennalta määrättyjä, niistä on mahdollista ratkaista verteksille sellaiset uv-koordinaatit, että mallille voidaan asettaa toistuvia tekstuureja pelkän värin sijaan. Koordinaattien ratkaiseminen voi kasvattaa kenttien visuaalista antia huomattavasti, mutta myös algoritmin laskenta-aikaa. Yksinkertaisimmassa ratkaisussa kuution tapausta vastaavat kolmiot voitaisiin litistää uv-kartalle, mutta se aiheuttaisi jonkin verran tekstuurin venymistä tai kuvioitten katkeilemista. Käyttämällä planar mapping tekniikkaa uv-koordinaatteja ei tarvitse ratkaista ollenkaan, minkä takia se soveltuisi parhaiten kuutiomarssituksella muodostettavien mallien teksturointiin. Planar mapping tekniikassa 3D-mallille ikään kuin heijastetaan tekstuurit jokaisesta suunnasta, mihin tarvitaan ainoastaan polygonin pinnannormaalit.

Kuutiomarssitusta voidaan pelkästään kentän generoimisen lisäksi käyttää muidenkin pelin proseduraalisten objektien luomiseen. Algoritmia voitaisiin käyttää esimerkiksi pilvien, kiven lohka- reiden tai tyylitellyn kasvillisuuden luomiseen. Tällaisia objekteja generoitaessa niitä ei tarvitsisi tallentaa lohkojen vokselitaulukkoihin ja ne voisivat käyttää eri resoluutiota kuin maasto. Kenttä- generoinnin olisi mahdollista lisätä kenttiin myös kehittäjän käsin tekemiä moduuleja kuutiomars- situsalgoritmin muodostaessa vain lähtökohtaiset maastonmuodot.

## 5.2 Haasteet ja kehityskohteet

Kuutiomarssituksen toimiessa tehokkaasti pienissä määrissä iteroitavia vokseleita sen rajoitukset nousevat esiin demossa generoitavien kenttien kokoa kasvattamalla. Vokselitaulukon alustami- seen ja muokkaamiseen kuluu huomattavasti enemmän aikaa suuremmissa kentissä, mikä ei ole

suuri haitta sen tapahtuessa peliä käynnistettäessä latausruudun aikana. Suuren määrän dataa käsitteleminen on kuitenkin oikukasta, mikäli pelin kehittäjä tahtoo generoitavien kenttien olevan pelaajan muokattavissa. Muita ongelmia käytetyssä tekniikassa ovat sen rajatapauksissa lohkon 3D-malleihin ja niiden yhtymiskohtiin syntyvät aukot sekä malliin asetettavat ylimääräiset verteksit. Ongelmat on mahdollista silotella, mutta niistä useimmissa pelin kehittäjä joutuu harkitsemaan, halutaanko prosessiin käytettävän lisää laskenta-aikaa tai muistia.

Kuutiomarssituksen käsitellessä jokaisen kuution tapaus yksitellen se ei ole tietoinen sen vierekkäisistä kuutioista ja niitä vastaavista tapauksista. Kaksi vierekkäistä kuutiota jakavat keskenään aina neljä särmää, joille särmän päätepisteiden arvojen mukaan asetettaessa verteksi se tehdään kaksi kertaa. Näin tehdään kumpaakin kuutiota käsiteltäessä erikseen. Tällä tavoin lopullisessa 3D-mallissa on huomattavasti ylimääräisiä verteksejä, jotka lisäävät pelissä ruudun piirtämiseen vaadittua laskenta-aikaa ja mallin vaatimaa muistin määrää. 3D-mallissa jokainen kolmio ei tarvitse omia uniikkeja verteksejä, vaan yhtä verteksiä voidaan käyttää osana mielivaltaista määrää kolmioita. Moninkertainen verteksimäärä voitaisiin poistaa ohjelmoimalla puskuri, josta voitaisiin vertailla kuutiomarssittamisen aikana vierekkäisten jo marssitettujen kuutioiden tapauksia. Näin tiedettäisiin, sijaitseeko särmillä jo verteksi ennen uuden sijoittamista. Uusia verteksejä ei tarvitsisi lisätä mallin verteksitaulukkoon, vaan sen kolmiotaulukkoon lisättäisiin vain vanhemman verteksin indeksi. Tuolloin uuden verteksin sijaintia ei tarvitsisi edes ratkaista interpoloimalla. Puskurin käyttäminen kasvattaisi kuutiomarssitusalgoritmin tarvitsemaa muistin määrää riippuen marssitettavan lohkon koosta, sillä siihen pitäisi tallentaa kaikki vokselitaulukon kahden akselin rajaamien kuutioiden särmät ja niille asetetut verteksit. Puskurin käyttämisen lisäksi lopullisesta mallista voitaisiin vähentää verteksejä etsimällä mallista ne kohdat, joissa useamman vierekkäisen kolmien pinnannormaalit olisivat identtiset. Tällaisissa suurissa täysin tasaisissa kohdissa on tarpeettomia kolmioita, joita voitaisiin yhdistellä tapauskohtaisesti osaksi suurempia polygoneja. Tämä kasvattaisi taas algoritmin vaatimaa laskenta-aikaa, jolla saavutettaisiin näennäisen pieni optimointi mallien verteksien määrässä.

Perlin-kohinaa käyttämällä on mahdollista, että valitun pinta-arvon takia kenttään generoituu sen muista osista irrallisia paloja. Palat voivat olla epäloogisia leijuessaan tai muuten vain epäesteettisiä varsinkin realistista ympäristöä generoitaessa. Osia syntyy sitä enemmän, mitä tiheämpää ja monimutkaisempaa kohinaa kenttägeneroinnissa on käytetty. Niiden poistamiseksi kehittäjän täytyisi valita hyvin hillittyjä parametreja kenttägenerointiinsa, mutta satunnaisuuden vuoksi tämä ei takaa varmasti toivottua lopputulosta. Varmin tapa estää rakeita ilmaantumasta kenttiin

olisi ohjelmoida mallia tutkiva ohjelma, joka poistaa pienet muusta mallista erillään olevat pinnanmuodot.

Työn demon suurin ongelmakohta on maastonmuokkausfunktioiden kutsuminen Unity-pelimootorissa *Update*-funktioista. Funktio kutsutaan yhden kerran jokaisen ruudunpäivityksen aikana, jolloin seuraavaa ruudunpäivitystä ei aloiteta ennen kuin funktio on suoritettu loppuun. Vaikka kuutiomarssitus toimisikin nopeasti ruudunpäivityksen pysähtyessä ainoastaan sekuntien murtoosan ajan, pelaaja voi huomata muutoksen tasaisessa päivityksessä ikään kuin nykyksenä ja hetken kestäväenä kontrollien lukkiutumisena. Tämä huonontaa pelikokemusta merkittävästi sen tapahtuessa jatkuvasti jokaisessa muokkauksessa. Jotta maaston muokkaaminen olisi varsinaisessa pelitoteutuksessa mielekästä kuutiomarssittaminen kannattaisi toteuttaa siten, että algoritmi lasketaan täysin erillisellä prosessorin säikeellä, jolloin ruudunpäivitys ei pysähtyisi odottamaan sen laskemista. Algoritmi voitaisiin kirjoittaa myös laskettavaksi useilla prosessorin säikeillä tai jopa tietokoneen näytönohjaimella varjostinohjelman avulla. Kun ruudunpäivitys lakkaa odottamasta algoritmin toimimista syntyy uusi ongelma, koska ruutu voi päivittyä useita kertoja pelaajan syötteen ja kentän geometrian muuttumisen välillä. Algoritmin laskemisen keston mukaan pelaaja näkee sulavan ruudunpäivitysnopeuden ansiosta, että varsinainen muokkaus tapahtuukin näennäisesti viivästyneesti. Tehottomilla prosessoreilla algoritmin laskeminen kestää entistä kauemmin ja viive kasvaa. Pelin kehittäjä voi kuitenkin peittää viivettä, siten että pelaaja näkee välittömän audiovisuaalisen tehosteen muokkauskohdassa. Tehoste toimii välittömänä palautteena pelaajalle toiminnosta, ja se voi verhota muokkautuvan maastonkohdan, siten että sen hälventyessä sen takaa paljastuu kentän muokattu kohta. Mikäli muokkauksia tapahtuu paljon samanaikaisesti, esimerkiksi jos tekniikkaa käytetään verkkomoninpelissä, viiveen vuoksi muokkaukset näkyvät pelaajalle asynkronisesti. Kehittäjä voi ongelman ratkaisemiseksi ja laskentatehon säästämiseksi jaksoittaa kuutiomarssitusalgoritmin käyttämistä siten, että toisiaan ajallisesti lähekkäin tapahtuvat muutokset yhdistetään. Useat muokkaukset tehdään ensin yhteen lohkon vokselitaulukkaan, jonka jälkeen se kuutiomarssitetaan ainoastaan kerran. Tällöin kaikki muokkaukset näkyvät pelaajalle samanaikaisesti, mutta eri viivein.

Demossa maaston muokkauksessa vokselitaulukon iteroiminen pidentää eniten prosessin laskemisen kestoja. Alkukantaisessa toteutuksessaan ohjelma iteroi lävitse jokaisen taulukon vokselin ja tekee sen vielä uudestaan kuutiomarssituksessa. Vokseleiden muokkaamisen ja kuutiomarssittamisen voisi yhdistää yhteen for-silmukkaan erillisessä funktion toteutuksessa tai kentän vokseleille voisi ohjelmoida vikkellämmän tietorakenteen. Pienet lohkot pitävät iteroitavien vokseleiden

määrän vähäisenä. Kuitenkin käyttämällä esimerkiksi kasipuu-tietorakennetta voitaisiin vokseleita muokatessa taulukosta palauttaa vain osa kaikista lohkon vokseleista. Kasipuu-rakenteelle on kuitenkin ominaista, että siitä tiedon hakeminen kasvattaa hiukan vaadittua laskenta-aikaa ja se kasvattaa varattavan muistin määrää vähän. Jokaisen vokselin pinta-arvon lisäksi täytyisi rakenteeseen tallentaa sen rakenne ja järjestys. Laskentatehoa säästyisi kuitenkin pienempää määrää vokseleita iteroitaessa varsinkin suuremmissa kentissä. Oikean tietorakenteen sovittaminen tavalla, joka on todella parempi, voi vaatia kehittäjältä paljon testaamista. Kehittäjä voisi ohjelmoida ratkaisun, jossa vokselitaulukon koko muuttuisi dynaamisesti. Näennäisesti tyhjiä kentän osia ei tallennettaisi siihen. Tuolloin taulukon iteroiminen ratkaisemalla vokseliden indeksit lohkon koon perusteella olisi kuitenkin mahdotonta.

Vokselitaulukon arvojen pysyessä pelaajalle näkymättöminä ne voivat olla yhdentekeviä. Pelaaja voi tavallaan vain aistia, missä kentän kohdissa joko on tai ei ole kiinteää osaa katsomalla piirtyvää 3D-mallia. Kehittäjälle näennäisesti tyhjä tila kentän mallien ulko- ja sisäpuolella ei kuitenkaan ole täysin merkityksetöntä, sillä kehittäjä saattaa haluta säilyttää Perlin-kohinan luonnollisen näköisen olemuksen tai sitten hävittää sen siten, että vokselitaulukoissa kaikki tyhjä tila asetetaan maksimilukuarvoon yksi, jolloin siinä ei voida tulkita olevan yhtään mitään. Tämä voi vaatia kehittäjältä kuitenkin monimutkaista luovaa ratkaisua, etteivät generoitujen mallien pinnat muutu luonnottoman rosoisen tai etäisyyksiltään yhtenevän näköisiksi. Pelin kehittäjän on siis harkittava, miten kentän tyhjää tilaa käsitellään. Tyhjä tila voi olla kehittäjälle myös todella hyödyllistä, jos vokselitaulukon arvoista kuutiomarssitetaan käänteinen malli, syntyy tyhjää tilaa kuvaava polygoniverkko. Verkkoa ei välttämättä tarvitse piirtää moottorissa, mutta sen avulla voidaan tutkia kentän tilavuutta, ja se voi toimia tietona kentässä toimiville tekoälyä hyödyntäville olioille. Mallista selviää muun muassa, minkä muotoisessa tilassa on mahdollista liikkua ja miten huoneet tai luolat yhdistyvät toisiinsa.

Tekniikan ongelmien optimoiminen ja 3D-mallin ominaisuuksien, kuten uv-koordinaattien ja pinnannormaalien ratkaiseminen kasvattaa vaadittua laskenta-aikaa nopeasti. Vaaditun tehon kasvaminen ei välttämättä haittaa kentägenerointia, sillä se voi tapahtua pelaajalta näkymättömissä pelin latausruudun aikana. Kasvava odottamisen määrä voi kuitenkin huonontaa pelaajan yleistä kokemusta pelistä. Kentän yksityiskohtaisuutta kasvatettaessa ja pelin performanssia parannettaessa kehittäjän on etsittävä ja harkittava erikoista ratkaisua. Ratkaisussa kenttägeneroinnin koon, yksityiskohtaisuuden, laskenta-ajan ja varaaman muistin määrän on pysyttävä tasapainossa. Ohjelman vaateiden määrä ei voi kasvaa epärealistiseksi suhteessa pelaajien käytettävissä olevan laitteiston ominaisuuksiin.

## 6 Yhteenveto

Kuutiomarssitusalgorithmi on hakutaulukkonsa ansiosta hyvin nopeasti toimiva algoritmi, jota voidaan käyttää monimutkaisten mallien luomisessa. Tietokonepeleissä tämä mahdollistaa erilaisten kenttien lähes välittömän generoimisen yksinkertaisesta datasta. Kuutiomarssitus tarvitsee toimiakseen iteroitavakseen joukon yhtenäisiä kolmiulotteisen avaruuden vokseleita, joista se muodostaa valmiita 3D-malleja tutkimalla ja sijoittamalla pistejoukon kohtiin sen taulukkonsa mukaisia polygoneja. Nopean toimintansa ansiosta sitä on kannattavaa käyttää myös pelin toiminnan aikana pelaajan reaaliaikaisen maaston muokkauksen mahdollistamiseksi, mikä voi lisätä erikoista pelattavuutta.

Kenttägeneroinnissa algoritmin tarvitsema vokselitaulukko voidaan luoda käyttäen luonnollisen näköistä Perlin-kohinaa. Kohina noudattaa pseudosatunnaisuutta siemenlukua käyttäen, ja sen avulla saadaan nopeasti aikaan mielenkiintoisia muotoja. Kohinan arvoja tallentamalla vokselitaulukkoon ne ovat triviaalisti muokattavissa. Niitä voidaan käyttää pohjana monimutkaisempien kenttien generoinnissa lisäämällä, vähentämällä tai siirtämällä niitä. Perlin-kohina ei ole pakollinen osa kuutiomarssitusta käytävässä kenttägeneroinnissa, mutta sen kyky synnyttää suuria määriä toistettavaa geometriaa on jotain, mikä on erinomainen ominaisuus kenttägeneroinnille. Algoritmin tarvitsemaa tietorakennetta voidaan muokata loputtomilla tavoilla, ja sillä voidaan mahdollisten huppurealististen maisemien lisäksi kuvata myös muita pelikentästä löytyviä asioita kuin maastoa. Vokselitaulukkojen syntyessä pienestä siemenluvusta ja muokkautuessa suoraviivaistetuilla operaatiolla kuutiomarssitettava peliympäristö soveltuu hyvin verkkopelien tarkoituksiin. Kaikille pelaajille voidaan luoda identtinen pelikenttä ja tiedonsiirtomäärät pysyvät vähäisinä.

Mikäli kuutiomarssittamista käytetään tekniikkana tietokonepelin kenttägeneroimiseksi, kehittäjän on tärkeätä ottaa huomioon, että kentän koko ja tarkkuus kasvattavat tarvittavaa laskenta-aikaa ja muistin määrää eksponentiaalisesti. Suurien kenttien muokkaamiseksi tietokoneen muistiin on ladattuna valtavasti tietoa vokseleiden lukuarvoista ja niiden nopea iteroiminen vaatii luovia ratkaisuja. Työn pelidemon kentän 3D-malleihin liitetään paljon tarpeettomia verteksejä, jotka eivät haittaa tehokkailla tietokoneilla performanssia merkittävästi, mutta niiden poistaminen matalatasoisempia koneita varten ei vähennä algoritmin prosessoreille asettamaa taakkaa. Algoritmin parhaan mahdollisen implementaation ratkaiseminen voi vaatia paljon optimoitaessa sen suorittamistapa ja tietorakenteet. Demossa esitelty epäoptimikin toteutus kuitenkin avaa lukemattomia mahdollisuuksia tuottaa loputonta proseduraalisesti generoitua sisältöä.

Kehittäjän on kyettävä löytämään oikea tasapaino generoinnin monimutkaisuuden ja algoritmin suorittamistehokkuuden välillä. Tavoiteltavaa lopputulosta määrittää visio pelistä, jonka pelinkehittäjä pyrkii toteuttamaan. Kuutiomarssituksella voidaan luoda tyylieltyyn näköinen pelikenttä, joka mahdollistaa paljon erilaisia asioita niin kehittäjälle kuin pelaajalle, mutta se yksin ei riitä tekemään pelistä erinomaista viihdettä. Pelisuunnittelijan on kyettävä ymmärtämään kuutiomarssituksen mahdollisuudet perinpohjaisesti, jotta teknologia kohtaisi pelin vision parhaalla tavalla luoden loputtomiin ainutkertaista pelattavuutta.

## Lähteet

- 1      Lorensen WE, Cline HE. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. ACM SIGGRAPH Computer Graphics 1987;21(4):163-169.
- 2      Brouke P. Polygonising a scalar field. 1994. Viitattu: 13.10.2020. Saatavilla: <http://paulbourke.net/geometry/polygonise>.
- 3      Togelius J, Kastbjerg E, Schedl D, Yannakakis GN. What is procedural content generation? PCGames '11: Proceedings of the 2nd International Workshop on Procedural Content Generation 2011:1-6.
- 4      Short TX, Adams T. Procedural generation in game design. Boca Raton: CRC Press, Taylor & Francis Group; 2017.
- 5      Dodge Roll. Enter the Gungeon. Devolver Digital. 2016. Viitattu: 26.10.2020
- 6      Microsoft Studios, Mojang. Minecraft. Microsoft Studios. 2017. Viitattu: 26.10.2020
- 7      Perlin K. An Image Synthesizer. ACM SIGGRAPH Computer Graphics 1985; 19(3):287-296.
- 8      The Coding Train. 1.2 Introduction – Perlin Noise and p5.js Tutorial. 2016. Viitattu: 9.10.2020. Saatavilla: <https://www.youtube.com/watch?v=Qf4dIN99e2w>.
- 9      Reunanen M. Maisemia Perlin-kohinalla. Skrolli. 2015;(1):46-47.
- 10     Unity Technologies. Unity Scripting Reference. 2020. Viitattu: 22.10.2020. Saatavilla: <https://docs.unity3d.com/ScriptReference>.
- 11     Takahashi K. Perlin Noise Function for Unity. 2015. Viitattu: 20.10.2020. Saatavilla: <https://github.com/keijiro/PerlinNoise>.