KARELIA-AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma

Otso Nuortimo

KOLMIULOTTEISEN MAASTON PROSEDURAALISTA GENEROINTIA KÄYTTÄVÄ PELI JAVA-OHJELMOINTIKIELELLÄ

Opinnäytetyö
Lokakuu 2016

OPINNÄYTETYÖ
Lokakuu 2016
Tietotekniikan koulutusohjelma

Tikkarinne 9
80220 JOENSUU
013 260 600

Tekijä(t)
Otso Nuortimo

Nimeke
Kolmiulotteisen maaston proseduraalista generointia käyttävä peli Java-ohjelmointikielellä

Toimeksiantaja
-

Tiivistelmä

Opinnäytetyön tarkoituksena oli toteuttaa proseduraalista maaston luomista hyödyntävä peli Java-ohjelmointikielellä. Toteutukseen ei käytetty valmista pelinkehitysympäristöä tai jo olemassa olevaa pelimoottoria. Työllä ei ollut toimeksiantajaa, vaan se toteutettiin itsenäisesti. Aihe valittiin sekä sen kiinnostavuuden vuoksi että omasta halusta tutustua pelinkehitykseen.

Opinnäytetyön pääpaino oli pelin toteutuksella. Keskeisenä osana pelin toteutusta oli muokkautuvan ja laajenevan pelimaailman hallitsemiseen sopivan rakenteen kehittäminen. Toteutuksessa panostettiin myös pelin nopeaan suorituskykyyn. Tämän lisäksi opinnäytetyössä tutustuttiin maaston luomisessa hyödynnettäviin algoritmeihin, 3D-grafiikoiden luontiin käytettävään OpenGL-ohjelmointirajapintaan sekä niiden hyödyntämiseen pelinkehityksessä.

Pelin toteutus oli onnistunut ja halutut ominaisuudet saatiin toteutettua. Jatkokehitysmahdollisuuksia pelissä on runsaasti etenkin kehittyneempien grafiikoiden ja pelillisten elementtien kannalta.

Kieli

suomi

Sivuja 44

Asiasanat

pelinkehitys, Java, OpenGL, maaston generointi, Perlin Noise

THESIS
October 2016
Degree Programme in Information Technology

Tikkarinne 9
80220 JOENSUU
FINLAND
+358 13 260 600

Author (s)

Otso Nuortimo

Title

Procedural generation of 3D terrain in a game written in Java

Commissioned by

-

Abstract

The aim of this thesis was to implement a game which would utilize procedural terrain generation using the Java programming language. No game development environment or an existing game engine was used. The thesis was not commissioned and was carried out independently. The subject of the thesis was chosen both because of personal interest in the subject, and wish to get more knowledgeable in game development.

The main focus of the thesis was the implementation of the game. A central point of the implementation was to develop a suitable structure to control the dynamic and expanding game terrain. The game's good performance was also important during the implementation. In addition, algorithms utilized in terrain generation and the OpenGL application programming interface used to create 3D graphics were explored.

The implementation of the game was successful and all the planned features were implemented. There are many possibilities for further development of the game, especially in developing the graphics and adding more gameplay elements.

Language

Finnish

Pages 44

Keywords

game development, Java, OpenGL, terrain generation, Perlin noise

Sisältö

Keskeiset käsitteet ... 5
1 Johdanto ... 6
2 Ohjelman esittely ja toteutetut ominaisuudet ... 7
2.1 Proseduraalinen generointi ... 7
2.2 Biomit ja niiden puut, vesistöt, luolat ja pilvet ... 8
2.3 Muokattavuus ... 11
2.4 Maailman ja pelaajan tiedot ... 12
3 Ohjelman toteutus ... 13
3.1 Perlin ja Simplex -kohinat ... 13
3.1.1 Perlin-kohina ... 13
3.1.2 Simplex-kohina ... 15
3.2 Lohko ... 16
3.2.1 Ehdot luomiselle ... 16
3.2.2 Lohkoluokan vakiot, muuttujat ja konstruktori ... 16
3.2.3 Lohkojen varastointi ... 20
3.3 ChunkManager ... 21
3.4 Koordinaattien hallinta ... 22
3.5 Lohkojen hallinta ... 23
3.5.1 Datan pakkaaminen ja purkaminen ... 23
3.5.2 Maaston muutosten hallinta ... 25
3.6 Grafiikoiden luonti ... 26
3.6.1 LWJGL ... 26
3.6.2 OpenGL ... 27
3.6.3 Lohkon OpenGL-objektin luonti ... 31
3.7 Pääsilmukka ... 34
3.8 Pelaajan liikkuminen ... 35
3.9 Monisäikeisyys ... 37
4 Toteutuksen aikana tehdyt parannukset ... 39
4.1 Lohkoihin jako ... 39
4.2 Datan pakkaaminen ja purkaminen ... 40
4.3 Monisäikeistys ... 40
5 Jatkokehitysmahdollisuudet ... 41
5.1 Vesi ... 41
5.2 Graafinen ulkoasu ... 41
5.3 Ohjelman kontrolloimat hahmot ... 42
6 Pohdinta ... 43
Lähteet ... 44

Keskeiset käsitteet

Block	Block eli kuutio on pelimaailman perusrakennepalikka. Koko pelimaailma rakentuu näistä kuutioista. Kokoa yhdellä kuutiolla on yksi kuutiometri pelihahmon ollessa vajaan kahden metrin korkuinen. Kuutioita on useita erityyppisiä erinäköisten maastojen luomiseen.
Chunk	Chunk eli lohko on useammasta lähekkäin olevasta kuutiosta muodostuva kokonaisuus. Toteutetussa pelissä yhden lohkon koko on 32 * 32 * 32 eli 32768 kuutiota. Maaston jakaminen lohkoihin on tarpeellista kuutioiden suuren määrän vuoksi.
HashMap	HashMap eli hajautustaulu on tietorakenne, jossa jokaisella siihen tallennetulla tiedolla on uniikki arvo eli avain, jolla se voidaan hakea.
Shader	Shader eli varjostin on muusta ohjelmasta erillinen ohjelma, jolla voidaan vaikuttaa piirrettäviin grafiikoihin. Niiden avulla voidaan luoda kehittyneempiä erikoisefektejä, kuten realistia varjoja tai heijastuksia. OpenGL-varjostimet kirjoitetaan GLSL-ohjelmointikielellä.
Voxels	Voxels on toteutetun pelin nimi. Voxel eli vokseli on kolmiulotteinen vastine pikselille, kaksiulotteisen kuvan yksittäiselle pisteelle.

1 Johdanto

Tämän opinnäytetyön tavoitteena oli kehittää proseduraalista maaston generointia käyttävä peli, jonka grafiikat toteutettaisiin OpenGL-ohjelmointirajapinnalla. Käytettäväksi ohjelmointikieleksi valittiin Java sen tuttuuden sekä sille saatavilla olevien ohjelmointikirjastojen vuoksi.

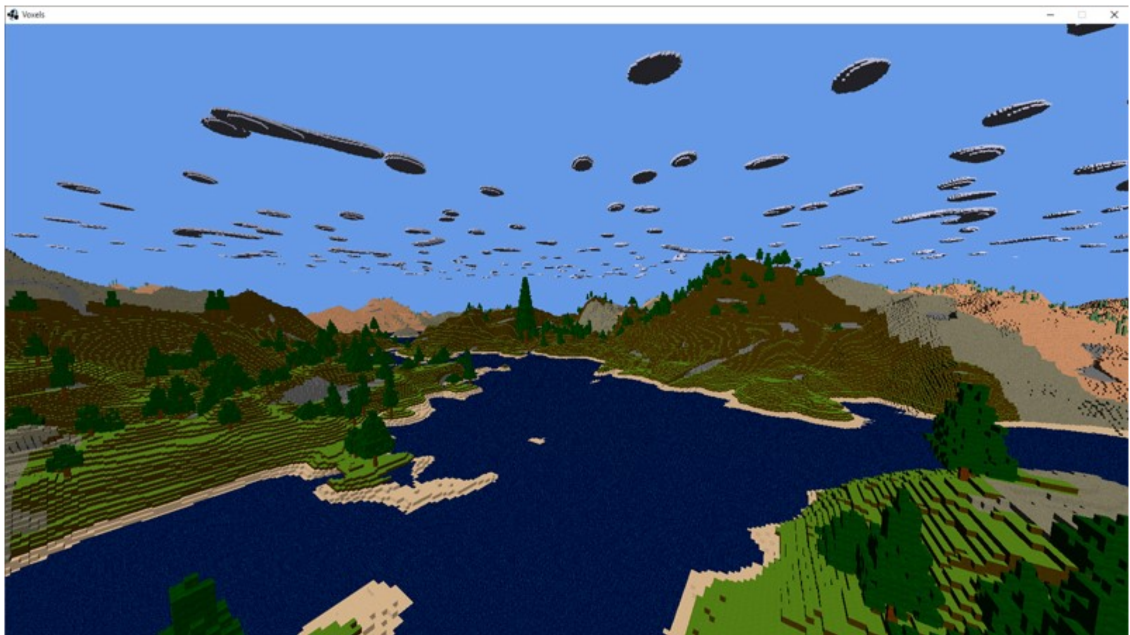
Olemassa olevaa pelinkehitysympäristöä tai pelimoottoria ei käytetty, vaan se ohjelmoitiin itse. Proseduraalisen maaston luomisessa hyödynnettiin sekä Perlin- että Simplex-kohina-algoritmeja. Inspiraationa toteutetulle pelille oli tunnettu hiekkalaatikkopeli Minecraft. Opinnäytetyön idea oli itse keksitty ja se toteutettiin itsenäisesti ilman toimeksiantajaa.

Opinnäytetyön pääpainona on pelin toteutuksen esittely. Tavoitteena on esitellä pelin toiminnan pääpiirteet sekä niiden toteuttamiseen valitut toteutustavat. Opinnäytetyöhön on poimittu havainnollistavia kuvia pelistä sekä lyhyitä lainauksia ohjelmakoodista esittelyn tueksi. Sekä OpenGL että Perlin- ja Simplex-kohina-algoritmeista esitellään pelin toteutuksen kannalta merkittävät osat.

Opinnäytetyössä esitellään ensimmäisenä peli ja siihen toteutetut ominaisuudet. Seuraavaksi pelin toteutus käydään läpi tutustuen tarkemmin sen rakenteeseen. Lopuksi kerrotaan pelin kehityksen aikana toteutetuista parannuksista, mahdollisista jatkokehitysmahdollisuuksista sekä pohditaan kyseisen opinnäytetyön prosessia.

2 Ohjelman esittely ja toteutetut ominaisuudet

Voxels on vokselipohjainen, proseduraalisesti generoitavan maailman sisältävä peli. Maailma on vapaasti muokattavissa, ja sitä voidaan tutkia ja luoda lähes rajattomasti. Vaihtelua maastoon tuovat erilaiset biomit, jotka sisältävät erilaisia maastoja ja kasvustoja. Maaston muokkaamisen ja tutkimisen lisäksi ohjelmassa ei ole toteutettu pelillisiä elementtejä. Inspiraationa ohjelman toteuttamiseen on ollut Minecraft-hiekkalaatikkopeli. Kuvassa 1 on nähtävissä näkymä pelistä.



Kuva 1. Pelinäkymä.

2.1 Proseduraalinen generointi

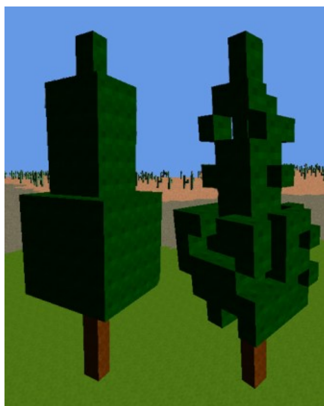
Ohjelman maasto luodaan kokonaan käyttäen proseduraalista maaston generointia. Tämä tarkoittaa sitä, että maastoa luodaan jatkuvasti pelaajan ympärille erilaisten algoritmien avulla. Maaston luomiseen ei käytetä minkäänlaista sattumanvaraisuutta, joten samoja parametreja käyttämällä sama maasto voidaan ladata koska tahansa uudella pelikerralla.

Algoritmien arvoja vaihtamalla voidaan maaston rakennetta muokata hyvin erilaiseksi vaikuttamalla muun muassa maaston jyrkkyyteen, veden korkeuteen, biomien kokoon ja kasvillisuuden tiheyteen. Proseduraalisen generoinnin algoritmeina käytetään sekä Perlin- että Simplex-kohina-algoritmeja.

2.2 Biomit ja niiden puut, vesistöt, luolat ja pilvet

Biomilla tarkoitetaan tämän ohjelman yhteydessä erinäköisiä maastoalueita, jotka eroavat sekä ulkonäöltään että kasvillisuudeltaan toisistaan. Ohjelmaan on toteutettu kolme erilaista biomia: nurmimetsä, hiekka-aavikko ja sorakenttä.

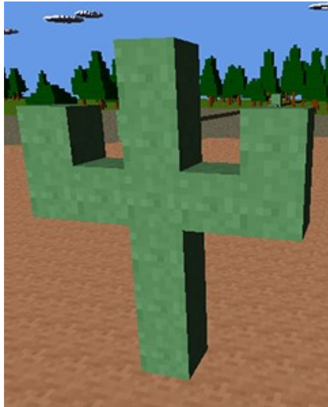
Nurmimetsässä kasvaa kuusia ja hiekka-aavikolla kaktuksia. Kuusten korkeus ja leveys vaihtelevat monipuolisemman maaston aikaansaamiseksi. Korkeuden ja leveyden vaihteluun käytetään ennalta määrättyä satunnaislukugeneraattoria todellisen satunnaisuuden poistamiseksi. Harvinaisissa tapauksissa kuusi kasvaa monin kerroin muita kuusia suuremmaksi, jolloin sen erottaa maastosta jo kaukaa.



Kuva 2. Puu ennen ja jälkeen.

Kuvassa 2 on havainnollistettu puiden luomiseen käytettävää menetelmää. Kuusi koostuu puun rungosta sekä kolmesta päällekkäin olevasta, erikokoisesta heksaedrista. Tämä on nähtävissä kuvan vasemmalla puolella. Tämän jälkeen kuusesta poistetaan noin 35 % kuusikuutioista ennalta määrättyä satunnaislukufunktiota käyttäen. Lopputuloksena oleva puu on kuvan oikealla puolella.

Kaktusten luomisessa käytetään samalla tavalla sattumanvaraisuutta. Kaktuksella on maksimissaan neljä L-kirjaimen muotoista haaraa, jotka jokainen osoittavat eri suuntiin. Jokaisella haaralla on vain 25% todennäköisyys olla kaktuksessa, minkä seurauksena maaston kaktukset vaihtelevat sekä muodoltaan että haarojen määrältään. Kuvassa 3 on nähtävissä kaksihaarainen kaktus.

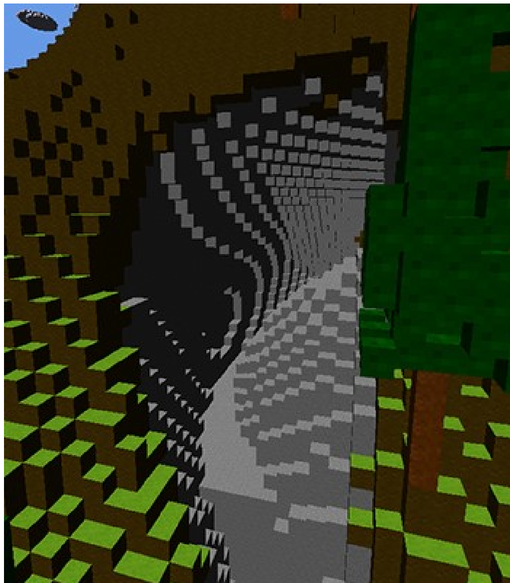


Kuva 3. Kaksihaarainen kaktus.

Erilaisten biomien lisäksi pelimaailmassa on vesialueita, jotka täyttävät maaston matalammat alueet. Vesistöjen korkeus on säädettävä parametri, jota muokkaamalla voidaan säätää koko maailman veden korkeutta. Veden korkeuden kasvattaminen johtaa näkyvien maa-alueiden pienenemiseen. Vesialueiden reunat ovat hiekkakuutioita, jotka muodostavat yksinkertaisen rannikon. Biomista riippumatta rannikon hiekalla ei kasva kaktuksia tai puita. Esimerkkinä vesialueesta ja sen rannasta on kuva 4.

Kuva 4. Vesialue ja rantaa.

Maastossa on maanalaisia luolastoja, jotka välillä nousevat maanpinnalle asti. Muodostuneiden luolastojen koissa ja muodoissa on suurta vaihtelua. Syvimmät luolastot mahdollistavat kävelyn maailman pohjakerrokseen asti. Kuten maaston korkeuserojen luomisessa, myös luolastojen muodostamisessa käytetään apuna Perlin-kohinaa. Kuvassa 5 on nähtävissä esimerkki luolan sisäänkäynnistä.



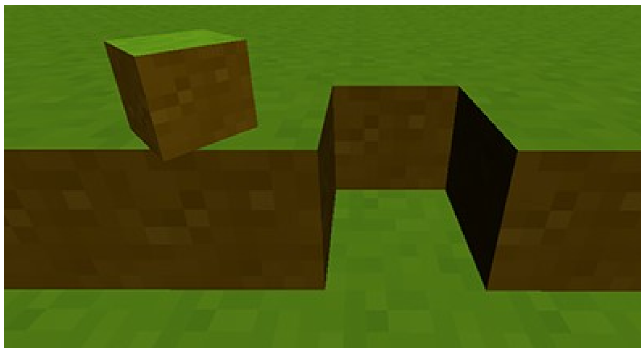
Kuva 5. Luolan sisäänkäynti.

Edellä esiteltyjen maaston ominaisuuksien lisäksi pelimaailmassa on myös taivaalla pilviä. Näistä on esimerkkinä kuva 6. Pilvet muodostuvat lähelle maailman maksimikorkeutta, eivätkä korkeimmatkaan maastoon muodostuvat mäet yllä niiden korkeudelle. Muokkaamalla pelimaailmaa pelaaja voi kuitenkin nousta pilvien korkeudelle ja kävellä niiden päällä kuten normaalissakin maastossa.

Kuva 6. Taivaalla olevia pilviä.

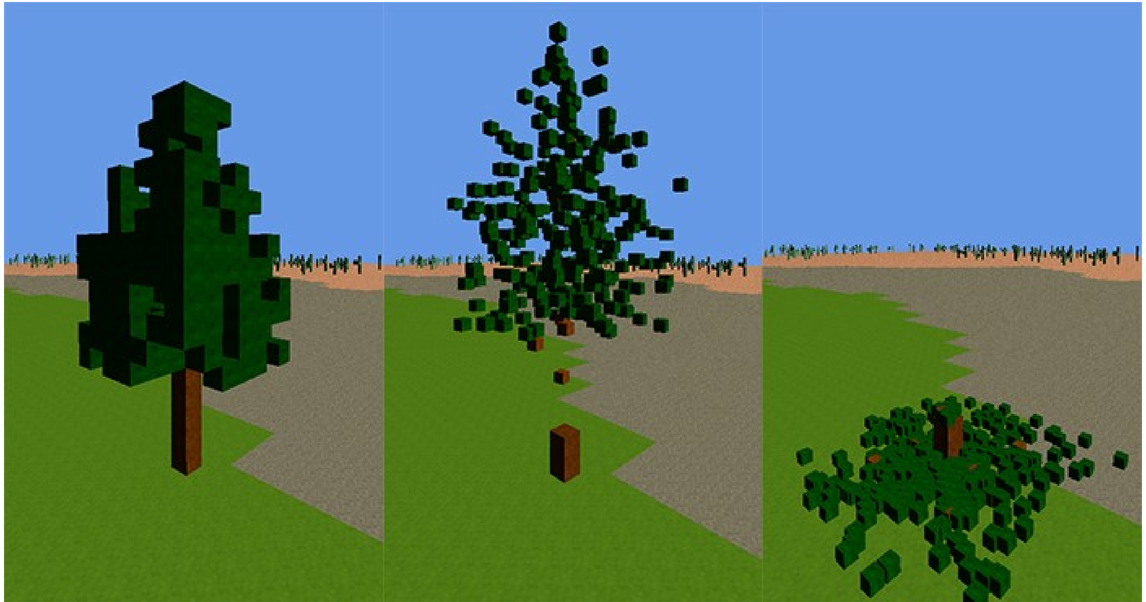
2.3 Muokattavuus

Pelimaailman kaikki kuutiot maailman alinta kerrosta lukuun ottamatta ovat täysin pelaajan muokattavissa. Pelaaja voi lisätä tai poistaa minkä tahansa tyyppisiä kuutioita mistä tahansa kohdasta pelimaailmaa. Hiiren rullalla pelaaja voi valita haluamansa kuution tyypin, ja lisätä sellaisen maailmaan osoittamalla ruudun keskellä olevalla tähtäimellä haluttuun kohtaan ja painamalla vasenta hiirennäppäintä. Vastaavasti tähtäämällä ja oikeaa hiirennappia painamalla voidaan poistaa yksittäisiä kuutioita, jolloin ne irtoavat maastosta lennähtäen hieman ylöspäin ja jäävät näkyville maaston pinnalle kuten kuvassa 7.



Kuva 7. Poistettu kuutio jää näkyviin maaston pinnalle.

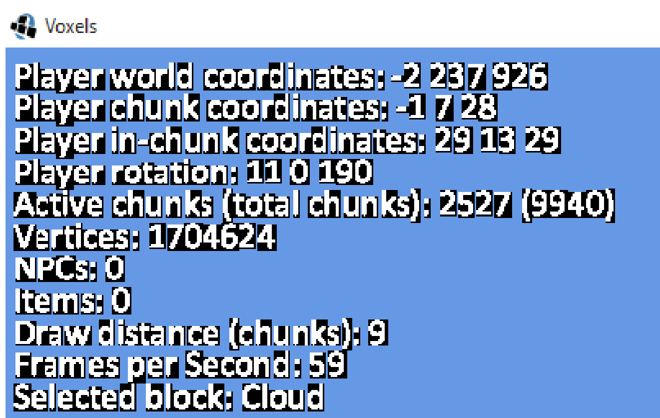
Yksittäisten kuutioiden poistamisen ja lisäämisen lisäksi ohjelmassa on mahdollista tehdä kerralla suurempia muutoksia maaston. Samaan tapaan tähtäämällä haluttuun kohtaan maastoa ja painamalla X-näppäintä maastosta poistetaan kerralla 12 * 12 * 12 kokoinen alue, eli yhteensä maksimissaan hieman alle 2000 kuutiota. Painamalla C-näppäintä pelaaja voi vastaavasti luoda uutta maastoa valitulla kuutiotyypillä samankokoisen alueen verran. Esimerkkinä edellä mainitusta poisto-ominaisuudesta kuvassa 8 on havainnollistettu kokonaisen puun poistaminen kerralla. Kuvassa on myös nähtävissä, kuinka poistetut palikat putoavat maahan.



Kuva 8. Useiden kuutioiden poistaminen kerralla.

2.4 Maailman ja pelaajan tiedot

Peliin toteutettiin pelin aikana nähtävillä oleva informaatioruutu, joka näytetään peli-ikkunan vasemmassa yläkulmassa. Informaatioruutu on nähtävillä kuvassa 9. Siitä on nähtävissä pelaajan tarkka sijainti maailmassa useassa eri koordinatistojärjestelmässä sekä pelaajan katseen suunta asteina ilmoitettuna.



Kuva 9. Pelaamisen aikana nähtävillä olevia tietoja.

Pelaajan tietojen lisäksi siitä nähdään aktiivisen eli näkyvillä olevan ja koko luodun maailman koko, piirretyn maailman kärkipisteiden määrä, piirtoetäisyys, ruudunpäivitysnopeus sekä maastonmuokkausta varten valittu kuutiotyyppi.

NPCs listaisi pelimaailmassa olevien ohjelman kontrolloimien hahmojen määrän, mutta niitä ei tähän opinnäytetyöhön otettu mukaan. Items listaa maailmassa näkyvillä olevien maastonmuokkauksessa irrotettujen kuutioiden määrän.

Informaatioruutu toteutettiin helpottamaan ohjelman teustausta. Sen avulla esimerkiksi eri koordinaatistojärjestelmien toimivuuden testaaminen oli huomattavasti helpompaa kuin lukemalla arvoja kehitysympäristön konsolista.

3 Ohjelman toteutus

Tässä luvussa esitellään, kuinka ohjelma on toteutettu ohjelmoinnin näkökulmasta. Tarkoituksena on selventää lukijalle ohjelman toiminnan perusrakennetta sekä tuoda esille tärkeitä sekä mielenkiintoisia yksittäisten ominaisuuksien toteutustapoja. Tämän lisäksi kappaleessa esitellään maaston luomisessa apuna käytetyt Perlin- ja Simplex-kohinat sekä grafiikoiden luomiseen käytetty OpenGL-ohjelmointirajapinta. Kappaleessa käytetään lyhyitä koodilainauksia toteutetusta Java-koodista toteutuksen esittelyn ohella.

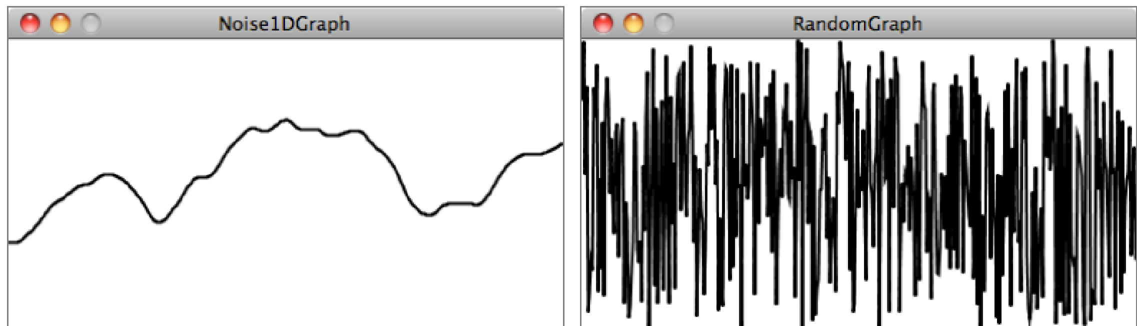
3.1 Perlin ja Simplex -kohinat

3.1.1 Perlin-kohina

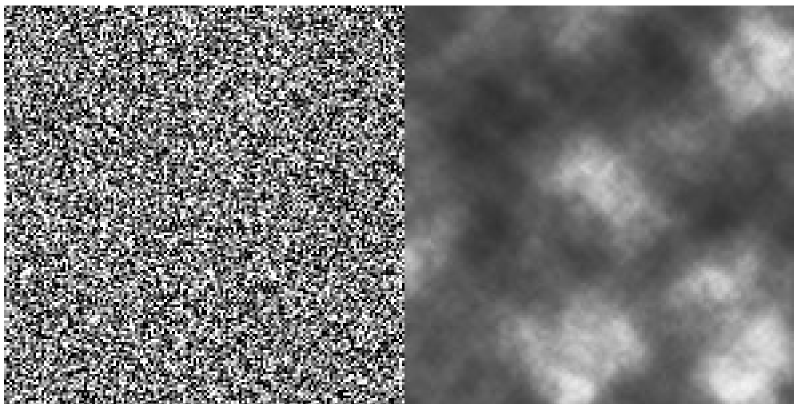
Perlin-kohina on Ken Perlinin vuonna 1983 kehittämä algoritmi tietokonegrafiikoiden luomiseen. Sen avulla voidaan luoda myös esimerkiksi pilviä tai maastonmuotoja. (Hoffman 2012).

Perlin-kohina-algoritmilla voidaan luoda koherenttia kohinaa yhdessä tai useammassa ulottuvuudessa (Hoffman, 2012). Koherentilla eli yhtenäisellä kohinalla tarkoitetaan sitä, että minkä tahansa kahden koordinaattipisteen välillä arvot kasvavat tai nousevat sulavasti ilman äkillisiä arvovaihteluita (Zucker 2011). Kuvissa

10 ja 11 on nähtävissä Perlin-kohinan ja satunnaisen kohinan välinen ero. Kuvista voidaan huomata, kuinka Perlin-kohinan avulla luodun käyrän tai pinnan arvot muuttuvat sulavasti.



Kuva 10. Yksiulotteisen Perlin-kohinan ja satunnaisen kohinan vertailu (Hoffman 2012). Perlin-kohina kuvan vasemmalla puolella.



Kuva 11. Kaksiulotteista satunnaista sekä koherenttia kohinaa. Perlin-kohina kuvan oikealla puolella.

Algoritmia käytetään antamalla käytetyn toteutuksen kohinametodille parametreina koordinaatit, joista Perlin-kohinan arvo luetaan (Biagoli 2014). Kuvassa 11 harmaan eri sävyt esittävät eri lukuarvoja, joita algoritmi palauttaa eri koordinaateissa. Alkuperäisen Perlin-kohina-algoritmin toteutus palauttaa arvoja väliltä -1 ja 1 (Zucker 2012).

Skaalaamalla algoritmille annettuja arvoja voidaan vaikuttaa kohinan tuottamaan lopputulokseen (Processing Foundation 2016). Esimerkiksi käytettäessä kaksiulotteista kohinaa maaston luomiseen, jos x-koordinaattina annetun luvun arvoja muutetaan eri tahtiin kuin z-koordinaatin, muodostuneet maaston pinnanmuodot

ovat venyneitä hitaammin muutetun koordinaatin akselin suuntaisesti. Muuttamalla molempien koordinaattien muutosnopeutta voidaan vaikuttaa maaston jyrkkyyteen. Mitä pienempi muutos annetuissa koordinaateissa on kahden vierekkäisen pisteen välillä, sen tasaisempi tulee luodusta maastosta.

Pelissä Perlin-kohinan käyttämiseen käytetään kirjastoa nimeltä FastNoise. Toisin kuin alkuperäisessä Perlin-kohinan toteutuksessa, kirjaston kyseinen metodi palauttaa arvoja 0−255. Tällä ei kuitenkaan ole merkitystä, koska palautetut arvot skaalataan pelin asetuksissa määriteltyyn maaston maksimikorkeuteen sopiviksi. Perlin-kohinan avulla luodaan pelimaailman korkeusvaihtelut ja sen sisältävät biomit.

3.1.2 Simplex-kohina

Perlin-kohinan lisäksi ohjelmassa käytetään Simplex-kohinaa, joka on uudempi ja paranneltu toteutus Perlin-kohinasta (Gustafson 2012). Simplex-kohina-algoritmi on alkuperäistä Perlin-kohina-algoritmia laskennallisesti nopeampi, eikä siinä ole Perlin-kohinassa esiintyviä artefakteja (Gustafson 2012). Ohjelman kannalta tärkeimpänä syynä Simplex-kohinan käytölle on kuitenkin sen toteuttavan SimplexNoise-kirjaston tarjoama mahdollisuus luoda kohinaa useammassa kuin kahdessa ulottuvuudessa. Perlin-kohina skaalautuu useampaan ulottuvuuteen Simplex-kohinaa hitaammin (Gustafson 2012), eikä käytetty FastNoise-kirjasto mahdollistanut kuin ainoastaan kaksiulotteisen kohinan luonnin.

Kolmiulotteinen kohina käyttäytyy samalla tavalla kuin kaksiulotteinen, parametreina annettavia koordinaatteja on ainoastaan kolme kahden sijaan. Ohjelmassa kolmiulotteista kohinaa käytetään sekä maastossa olevien luolastojen että taivaalla olevien pilvien luomiseen.

3.2 Lohko

Pelimaailmassa lähekkäin olevat kuutiot muodostuvat suurempia kokonaisuuksia, joita kutsutaan lohkoiksi. Tämä tekee ohjelman rakenteesta monimutkaisemman, mutta se on myös ratkaisevasti nopeammin ja tehokkaammin toimiva ratkaisu muokattavaa maailmaa käsiteltäessä. Maailman jakaminen lohkoihin vähentää huomattavasti käsiteltävien olioiden määrää verrattuna siihen, että maailman jokainen kuutio olisi oma olionsa.

3.2.1 Ehdot luomiselle

Uuden lohkon luomisen ehtona on se, että sen etäisyys pelaajan sijainnista on alle määritellyn lohkojen luontietäisyyden. Tällä varmistetaan, että pelaajan ympärillä on aina uutta maastoa pelaajan liikkuessa maailmassa. Tarkistaminen on toteutettu omassa säikeessään toimivalla luokalla, jossa lyhyin aikavälein suoritetaan tarkistusmetodi.

Tässä metodissa tarkistetaan pelaajan aktiivisesta lohkosta lähtien spiraalimaisessa järjestyksessä lohkot lohkojen maksimiluontietäisyyteen asti. Syynä spiraalimaiselle tarkistysjärjestykselle on se, että se tarkastaa lohkot järjestyksessä pelaajan lähimmästä lohkosta kauimpana olevaan. Tällä priorisoidaan läheltä puuttuva maasto tärkeimmäksi. Tarkistettavan lohkon koordinaateista luodulla avaimella tarkistetaan, onko kyseistä lohkoa vielä luotuna. Mikäli ei ole, tarkistus lopetetaan ja luodaan uusi lohko kyseisiä koordinaatteja lohkon konstruktorin parametreina käyttäen.

3.2.2 Lohkoluokan vakiot, muuttujat ja konstruktori

Chunk- eli lohkoluokka sisältää useita staattisia eli luokkakohtaisia vakioita, joiden arvoja vaihtamalla voidaan vaikuttaa muun muassa luotavan maaston korkeuteen, veden määrään tai luolastojen, puiden ja pilvien tiheyteen maailmassa. Luokkakohtaisten vakioiden lisäksi jokaisella lohkolla on omia muuttujia, joista

tärkeimmät ovat sen x-, y- ja z-koordinaatit sekä kolmiulotteinen tavutaulukko, joka sisältää sen jokaisen sisällä olevan kuution tyypin. Kuutio voi olla tyypiltään esimerkiksi hiekkaa, vettä, ilmaa, tai kiveä.

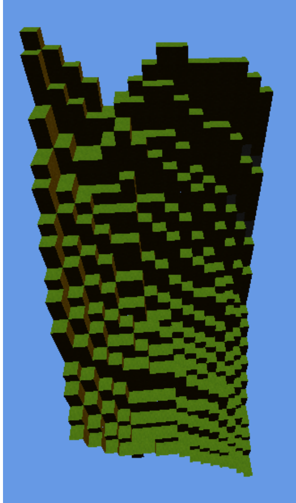
Lohkon konstruktorille annetaan parametreina sen koordinaatit maailmassa. Koordinaattien avulla täytetään kaksi väliaikaista lohkon leveyden kokoista kaksiulotteista taulukkoa, joista toinen sisältää tiedot maaston maksimikorkeudesta kussakin koordinaatissa, toinen siinä olevasta biomista. Tämä on nähtävissä koodiesimerkissä. Lohkon maksimikorkeudet ja biomit luodaan Perlin-kohinan avulla.

```
1. short[][] maxHeights = new short[CHUNK_SIZE][CHUNK_SIZE];
2. byte[][] biomes = new byte[CHUNK_SIZE][CHUNK_SIZE];
3. for (int x = 0; x < CHUNK_SIZE; x++) {
4.     for (int z = 0; z < CHUNK_SIZE; z++) {
5.         maxHeights[x][z] = Voxels.getNoise(x + xCoordinate, z + zCoordinate);
6.         biomes[x][z] = Voxels.getBiomeNoise(x + xCoordinate, z + zCoordinate);
7.     }
8. }
```

Erään lohkon maxHeights-taulukon arvot ovat nähtävissä kuvassa 12. Näiden arvojen avulla luodun lohkon korkeusvaihtelut ovat kuvassa 13. Taulukon koordinaateissa (1,1) oleva suurin arvo 40 vastaa kuvan 13 lohkon korkeimmalla olevaa kuutiota kuvan vasemmassa ylälaidassa.

(X,Z)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	40	39	38	37	35	36	37	38	39	39	38	38	38	38	37	36
	2	39	37	36	35	35	36	36	37	37	36	35	34	34	33	31	30
	3	36	34	33	33	34	35	34	34	34	33	31	30	29	27	25	23
	4	36	34	33	33	34	35	34	33	32	31	30	28	26	24	21	20
	5	34	32	31	31	31	32	30	30	29	28	26	24	22	20	17	15
	6	34	32	31	30	30	30	29	28	27	25	24	23	21	19	16	14
	7	33	31	30	29	28	28	26	25	23	22	21	21	20	18	14	12
	8	31	29	28	27	25	25	23	22	21	20	19	18	17	15	12	10
	9	30	28	26	25	23	23	21	21	19	18	17	16	15	13	11	8
	10	28	26	24	23	22	21	21	20	18	17	16	15	13	12	9	6
	11	27	25	23	21	21	20	19	19	17	16	15	13	12	10	7	4
	12	25	23	21	20	19	19	18	18	17	15	14	13	12	9	6	4
	13	21	20	18	17	15	15	15	15	13	12	12	11	9	7	4	3
	14	20	18	16	14	13	13	13	13	12	11	10	9	7	5	3	2
	15	18	16	14	13	12	12	12	12	10	9	8	7	6	4	2	0
	16	17	15	13	12	12	12	11	11	10	8	7	6	5	4	2	0

Kuva 12. Esimerkki maxHeights-taulukon mahdollisista arvoista.

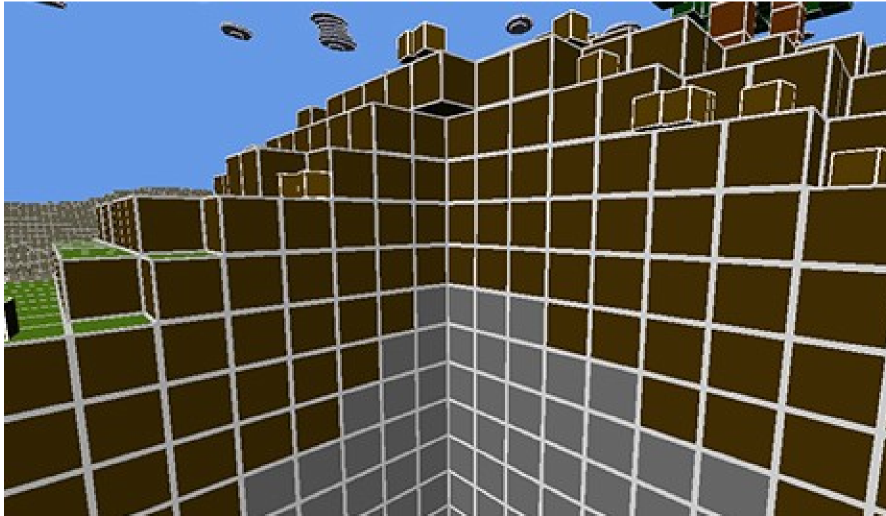


Kuva 13. Kuvan 12 arvoilla luotu lohko.

Seuraavaksi maxHeights- ja biomes-taulukkoja käyttäen täytetään lohkon kolmiulotteinen blocks-taulukko, joka sisältää lohkon jokaisen kuution tyypin. Blocks-taulukko käydään läpi sisäkkäisillä for-silmukoilla koodisesimerkin mukaisesti.

```
1.  for (int x = 0; x < blocks.length; x++) {
2.      for (int y = 0; y < blocks[x].length; y++) {
3.          for (int z = 0; z < blocks[x][y].length; z++)
4.              // Block type is set here.
5.          }
6.      }
7.  }
```

MaxHeights-taulukon arvot kertovat kuinka korkealla maasto on missäkin koordinaateissa, biomes-taulukon arvot kertovat minkä tyyppistä pinnalla oleva maasto on niissä koordinaateissa. Maaston pinnalla on viiden kuution korkeudelta biomin omaa kuutiotyyppiä, alemmat kerrokset muodostuvat kivikuutioista. Tämä on nähtävissä kuvan 14 poikkileikkauksessa. Maksimikorkeuden yläpuolella olevat kuutiot jätetään tyhjiksi ilmakuutioiksi. Mikäli lohko on maaston pohjimmaisena, sen alin kuutiokerros muodostuu tuhoutumattomista kuutioista. Näin estetään pelaajan putoaminen ulos maailmasta.



Kuva 14. Poikkileikkaus maaston pinnasta. Käytössä on kuutioiden reunoja korostava tekstuuri.

Tässä vaiheessa perusmaasto on valmis, mutta siitä puuttuvat vielä vedet, pilvet, luolat ja puut. Jäljellä olevista ilmakuutioista ne, jotka ovat asetuksissa määritellyn veden korkeuden alapuolella, muutetaan vesikuutioiksi. Rannat vesien reunoille muodostetaan muuttamalla veden maksimikorkeuden kohdalla oleva pintamaasto hiekkakuutioiksi.

Pilvien muodostumiselle on asetettu minimikorkeus, joka on maaston yläpuolella. Näissä korkealla taivaalla olevissa lohkoissa pilvet muodostetaan käyttäen kolmiulotteista Simplex-kohinaa. Simplex-kohina-algoritmi palauttaa arvoja nollan ja yhden väliltä x-, y- ja z-koordinaattien perusteella. Ilmakuutioista pilvikuutioiksi muutetaan ne kuutiot, joiden koordinaateissa algoritmin antama arvo on yli 0.95:n eli lähellä maksimia. Y- eli korkeuskoordinaattia on skaalattu algoritmiin kolminkertaiseksi muihin koordinaatteihin verrattuna. Tämä tarkoittaa sitä, että muutokset algoritmin antamissa arvoissa tapahtuvat y-koordinaateissa kolminkertaisella nopeudella. Tästä johtuen pilvet ovat pystysuunnassa litteitä eivätkä pallomaisia.

Luolat luodaan pilvien tapaan käyttäen Simplex-kohinaa. Mielenkiintoisempien luolamuodostumien aikaansaamiseksi luolat tehdään käyttäen kahta, eri koordinaateissa olevaa Simplex-kohinaa. Mikäli molempien arvot ovat 0.45:n ja 0.55:n välillä, kyseinen kuutio muutetaan ilmakuutioksi.

Maastoon muodostuvat puut eivät jakaannu tasaisesti koko maaston pinnalle, vaan muodostavat ryhmittymiä, ikään kuin pieniä metsiä. Näiden metsien välillä on alueita, joissa puita ei ole lainkaan. Jokaisen maaston pinnalla olevan kuution kohdalla tarkistetaan Perlin-kohinaa hyödyntävältä puidenluontimetodilta, kasvaako kyseisissä koordinaateissa puu. Metodi on esiteltynä koodiesimerkissä.

```
1.  public static boolean getTreeNoise(float x, float y, float z) {
2.      if (FastNoise.noise(x / 100f, z / 100f, 3) > 100f) {
3.          int noise = (int) (FastNoise.noise(x + 1000, z + 1000, 3));
4.          if (noise == 10 || noise == 50) {
5.              if (getCaveNoise(x, y, z) == false) {
6.                  return true; // Is tree
7.              }
8.              return false;
9.          }
10.         return false;
11.     }
12.     return false;
13. }
```

Kyseisessä metodissa käytetään kahta erillistä Perlin-kohinaa puiden ryhmittymien luomiseksi. Ensimmäisen kohinan parametreina annettuja koordinaatteja jaetaan sadalla, jolloin siinä tapahtuvat muutokset ovat hyvin pieniä pienillä koordinaattimuutoksilla. Näin saadaan luotua suurempia alueita, joissa puita voi mahdollisesti kasvaa. Näiden alueiden sisällä puiden määrää täytyy kuitenkin rajoittaa. Koodiesimerkin rivillä 3 käytetyn kohinan parametreja ei ole jaettu, joten muutokset siinä tapahtuvat nopeasti. Tällä tavalla minimoidaan se, että kaksi vierekkäistä koordinaattia sisältäisivät molemmat puun, koska on epätodennäköistä, että nopeasti muuttuvassa kohinassa kaksi vierekkäistä koordinaattia molemmat palauttaisivat saman puun luontiin sallitun arvon. Viimeisenä täytyy myös tarkistaa, että kyseisissä koordinaateissa ei ole luolan suuaukkoa.

3.2.3 Lohkojen varastointi

Valmiit lohkon tallennetaan ConcurrentHashMap-luokan hajautustauluun. ConcurrentHashMap-luokan hajautustaulun ero tavalliseen hajautustauluun on sen tuki useammalle säikeelle, eli sinne voidaan laittaa ja lukea dataa yhtäaikaisesti

useammasta säikeestä (Oracle 2016a). Hajautustaulun etuna listoihin tai taulukoihin verrattuna on muun muassa se, että halutun arvon hakemisen nopeus ei riipu tietorakenteen koosta, vaan se pysyy keskimäärin vakiona (Big-O Cheat Sheet 2016).

Hajautustaulussa jokaisella sinne tallennetulla arvolla on uniikki avain, joka annetaan lisättäessä arvo hajautustauluun. Tallennettuja arvoja voidaan hakea käyttämällä oikeita avaimia. Lohkojen hajautustaulussa avaimina käytetään lohkon x-, y-, ja z-koordinaateista laskettua uniikkia avainta ja arvoina lohkoja. Tämä tarkoittaa sitä, että hajautustaulusta voidaan hakea mikä tahansa lohko antamalla sen koordinaatit maailmassa. Koodiesimerkissä on havainnollistettu lohkojen laittamista ja niiden lukemista chunkMap-nimisestä hajautustaulusta.

```
2.  chunkMap.put(new Triple(chunk.x, chunk.y, chunk.z).hashCode(), chunk);
3.  Chunk chunk = chunkMap.get(new Triple(chunkX, chunkY, chunkZ).hashCode());
```

3.3 ChunkManager

Ohjelma sisältää ChunkManager-nimisen luokan, jonka tarkoituksena on nimen mukaisesti hallita ohjelman lohkoja. ChunkManager-luokalla on pääsy lähes kaikkiin ohjelman luokkiin ja olioihin, ja se sisältää runsaasti erilaisia metodeja, joiden avulla muut luokat voivat suorittaa erilaisia toimintoja. Tällaisia metodeja ovat esimerkiksi tietyissä koordinaateissa olevan lohkon tai kuution hakeminen, pelimaastoon tehtyjen muutosten hallitseminen sekä lohkojen ja niiden OpenGL-objektien luonti ja päivitys.

Tämän lisäksi luokka hallitsee muutamaa muuta samankaltaista pienempää luokkaa, jotka ovat vastuussa tietyistä ohjelman osa-alueista. Tällainen luokka on esimerkiksi ItemManager, joka vastaa maailmasta irrotetuista kuutioista, niiden sijainneista ja OpenGL-objekteista.

3.4 Koordinaattien hallinta

Ohjelma sisältää kolme erillistä koordinaatistojärjestelmää: maailmakoordinaatiston, lohkojen koordinaatiston sekä lohkojen sisäisen koordinaatiston niiden kuutioille. Nämä ovat nähtävissä kuvassa 15. Maailmakoordinaatiston arvo muuttuu yhdellä jokaista liikuttua kuutiota kohden, lohkokoordinaatiston arvo muuttuu yhdellä jokaista liikuttua lohkoa kohden. Kuten maailmakoordinaatiston, myös lohkojen sisäinen block- eli kuutiokoordinaatiston arvo muuttuu myös yhdellä jokaista kuutiota kohden, mutta se kertoo ainoastaan sijainnin lohkon sisällä, ei koko pelimaailmassa.

x-, y- tai z-koordinaatin käyttäytyminen eri koordinaatistoissa

Maailmakoordinaatisto	-9	-8	-7	-6	-5	-4	-3	-2	-1	**0**	1	2	3	4	5	6	7	8	9	10	11
Chunk-koordinaatisto	-2	-1	-1	-1	-1	-1	-1	-1	-1	**0**	0	0	0	0	0	0	0	1	1	1	1
Chunkkien block-koordinaatisto	7	0	1	2	3	4	5	6	7	**0**	1	2	3	4	5	6	7	0	1	2	3

Kuva 15. Kolme koordinaatistoa. Esimerkissä lohkon koko eli leveys on kahdeksan, pelissä se on 32.

Tämän perusteella maailmakoordinaatisto on informatiivisin ja tarkin kolmesta koordinaatistosta. Kuvasta 15 voidaan myös huomata, että maailmakoordinaatisto on ainoa koordinaatisto, joka voidaan muuttaa muiksi koordinaatistoiksi. Tämä tarkoittaa sitä, että sopivalla muunnoskaavalla maailmakoordinaatistolla voidaan viitata sekä lohkoihin että niiden sisäisiin kuutioihin. Muunnoskaavat ovat esiteltynä koodiesimerkissä.

```
1.  // Convert world coordinate to block-coordinate inside a chunk
2.  public final static int getBLockX(float x) {
3.      int value = floatToInt(x);
4.      if (value < 0) {
5.          value = CHUNK_SIZE + value % CHUNK_SIZE;
6.      }
7.      return value % Chunk.CHUNK_SIZE;
8.  }
9.  // Convert world coordinate to chunk-coordinate
10. public final static int getChunkX(float x) {
11.     int value = floatToInt(x);
12.     if (value < 0) {
13.         value -= CHUNK_SIZE - 1;
14.     }
15.     return value / CHUNK_SIZE;
16. }
17. // Convert coordinate from a float to an integer
18. private static int floatToInt(float f){
19.     return (f >= 0) ? (int) f : (int) (f - 1);
20. }
```

Koodiesimerkissä positiivisten koordinaattien muutoskaavat ovat yksinkertaisia. Lohkojen koordinaatti saadaan jakamalla maailmakoordinaatti lohkon koolla, lohkojen sisäisen kuution koordinaatti saadaan maailmakoordinaatin ja lohkon koon jakojäännöksestä. Negatiivisia koordinaatteja muutettaessa tehdään ylimääräinen välivaihe, jotta muutettava arvo antaa oikean arvon. Välivaihe on nähtävissä koodiesimerkin riveillä 5 ja 13.

3.5 Lohkojen hallinta

3.5.1 Datan pakkaaminen ja purkaminen

Datan pakkaamisella tarkoitetaan sen muuttamista toiseen formaattiin, jossa sen käyttämä muistinmäärä on alkuperäistä pienempi. Tästä pakatusta formaatista data voidaan palauttaa alkuperäiseen versioonsa. Voxels-pelissä datan pakkaamista käytetään pelin kuluttaman muistin minimoimiseen. Datan pakkaamiseen käytetään Fast-serialization ja Compress-LZF-kirjastoja.

Fast-serialization on sarjoittamiskirjasto, jonka tarkoituksena on korvata Javan sisäänrakennettu sarjoittaminen. Sarjoittamisella tarkoitetaan olioiden muuttamista tallennettavaan muotoon tavutaulukoksi (Oracle 2016b). Fast-serialization on Javan sisäänrakennettua sarjoittamista keskimäärin moninkertaisesti nopeampi sekä sen tuottama tavutaulukko on pienempi (Moeller 2014).

Compress-LZF-kirjasto mahdollistaa tavutaulukkojen pakkaamisen ja purkamisen Java-ohjelmassa LZF-pakkausalgoritmia käyttäen. Kirjaston käyttö on esiteltynä koodiesimerkissä. Tämä on myös mahdollista Javan sisäänrakennetuilla Deflater- ja Inflater-luokilla, jotka käyttävät ZLIB-pakkauskirjastoa (Oracle 2016c). Verrattuna ZLIB-kirjaston käyttämään algoritmiin LZF-algoritmilla pakkaaminen voi olla 5-6 kertaa nopeampaa, ja purkaminen kaksin kerroin nopeampaa (Saloranta 2014). Kirjaston tarjoaman pakkausalgoritmin nopeus oli syy sen valinnalle Javan sisäänrakennettujen luokkien käytön sijaan. Myös toteutetun pelin kehityksen aikana todettiin LZF-algoritmin nopeampi toiminta.

```
1. byte[] compressed = LZFEncoder.encode(uncompressedData);
2. byte[] uncompressed = LZFDecoder.decode(compressedData);
```

Aikaisemmassa lohkojen varastoinnin yhteydessä olleessa koodiesimerkissä esiteltiin lohkojen tallentaminen hajautustaulukkoon myöhempää käyttöä varten. Pelissä ei kuitenkaan ole tarpeen pitää kaikkia maailmaan luotuja lohkoja pakkaamattomina yhden hajautustaulukon sisällä, sillä pelaajan näköetäisyyden ulkopuolella olevia epäaktiivisia lohkoja ei piirretä eikä niihin voida vaikuttaa.

Tästä syystä peliin on toteutettu pakkaamattomien aktiivisten lohkojen hajautustaulukon rinnalle toinen hajautustaulukko, joka sisältää jokaisen luodun lohkon pakatun version. Aktiivisten lohkojen hajautustaulukossa pidetään ainoastaan aktiivisia lohkoja, ja sen sisältöä päivitetään jatkuvasti pelaajan sijainnin mukaan. Pelaajan liikkuessa pelimaailmassa epäaktiiviset lohkot poistetaan aktiivisten lohkojen hajautustaulukosta, ja näkyviin tulleet lohkot puretaan pakattujen lohkojen hajautustaulusta aktiivisten lohkojen hajautustauluun. Koodiesimerkissä on esiteltynä lohkojen pakkaaminen ja purkaminen edellä mainittuja kirjastoja käyttäen.

```
1. map.put(new Triple(chunk.x, chunk.y, chunk.z).hashCode(), toByte(chunk));
2. Chunk chunk = toChunk(map.get(new Triple(chunkX, chunkY, chunkZ).hashCode()));
3.
4. public byte[] toByte(Chunk chunk) {
5.     return LZFEncoder.encode(serialize(chunk));
6. }
7. public Chunk toChunk(byte[] bytes) {
8.     return deserialize(LZFDecoder.decode(bytes));
9. }
```

Kahden erillisen lohkojen hajautustaulun etuna on sekä vähäinen muistinkulutus että nopea pääsy aktiivisiin lohkoihin maastonmuokkausta varten. Pakkaamattomien lohkojen muokkaaminen ja lukeminen on huomattavasti nopeampaa kuin pakattujen lohkojen, sillä pakatun lohkon purkaminen luettavaan muotoon vaatii aikansa, eikä sitä ole mielekästä tehdä jatkuvasti jokaista ruudunpäivitystä varten.

3.5.2 Maaston muutosten hallinta

Jokaisella tehtävällä maaston muutoksella on koordinaatit, jotka kertovat muokattavan kuution sijainnin, sekä tyyppi, joka kertoo minkä tyyppiseksi kyseinen kuutio muutetaan. Ennen kuin muutosta voidaan tehdä, täytyy tarkistaa, että koordinaateissa olemassa oleva kuutio on sallittu muokkauksen kohde. Koodiesimerkin metodilla voidaan parametrina annetuilla maailmakoordinaateilla saada minkä tahansa maailmassa olevan kuution tyyppi. Metodissa käytetään aikaisemmin esiteltyjä koordinaatistonmuunnosmetodeita.

```
1.  public byte getBlock(Vector3f v) {
2.     Chunk chunk = getActiveChunk(getChunkX(v.x), getChunkY(v.y), getChunkZ(v.z));
3.     if (chunk != null) {
4.        return chunk.blocks[getBlockX(v.x)][getBlockY(v.y)][getBlockZ(v.z)];
5.     }
6.     // Chunk does not exist or is not in active chunks.
7.     else
8.        return -1;
9.  }
```

Koodiesimerkissä käytetty Vector3f on LWJGL-kirjaston tarjoama apuluokka, joka sisältää kolme julkista liukulukumuuttujaa x, y ja z. Tämän luokan avulla yhdessä oliossa voidaan siirtää kolme liukulukua esimerkiksi koordinaatteja tai sijaintia varten.

Alla olevan koodiesimerkin metodilla voidaan muokata annetuissa koordinaateissa olevan kuution tyyppiä. Koordinaattienmuunnosmetodeilla haetaan annetut koordinaatit sisältävä lohko, sekä koordinaateissa oleva kuutio. Kuution tyyppi vaihdetaan annettuun arvoon, jonka jälkeen lohkon OpenGL-objekti luodaan uudelleen, jotta tehty maastonmuokkaus näkyisi pelimaailmassa. OpenGL-objektin muokkaamisen jälkeen on tarpeellista tarkistaa myös viereiset lohkot, sillä mikäli maastonmuokkaus oli kuution reunalla, myös viereisen lohkon OpenGL-objekti täytyy päivittää. OpenGL-objektien luonnista on kerrottu luvussa Grafiikoiden luonti.

```
1. public void setBlock(Vector3f pos, byte type) {
2.     int chunkX = getChunkX(pos.x);
3.     int chunkY = getChunkY(pos.y);
4.     int chunkZ = getChunkZ(pos.z);
5.     Chunk chunk = getChunk(chunkX, chunkY, chunkZ);
6.     if (chunk != null) {
7.         int blockX = getBlockX(pos.x);
8.         int blockY = getBlockY(pos.y);
9.         int blockZ = getBlockZ(pos.z);
10.        chunk.blocks[blockX][blockY][blockZ] = type;
11.        ChunkVBOMaker.update(chunk);
12.        checkAdjacentChunks(pos);
13.    }
14. }
```

Kun maastoon tehdään kerralla isompia muokkauksia, ei ole järkevää käyttää edeltävässä koodiesimerkissä esiteltyä metodia. Toteutettu ominaisuus, jolla voidaan yhdellä napinpainalluksella muokata 12 * 12 * 12 kokoista aluetta tarkoittaisi OpenGL-objektien päivittämistä jopa tuhansia kertoja. Tästä syystä isojen muokkausten tekemiseen toteutettiin koodiesimerkistä hieman muokattu metodi, jossa ensin tehdään kaikki halutut muutokset lohkojen blocks-taulukoihin, ja vasta lopuksi päivitetään lohkot, joihin on tehty muutoksia. Tällä menetelmällä suurienkin muutosten tekeminen voidaan suorittaa vain muutaman OpenGL-objektin päivityksellä, jolloin ohjelman toiminta ei hidastu eikä pelaajan pelikokemus kärsi maastonmuutosten tapahtuessa välittömästi.

3.6 Grafiikoiden luonti

Jotta lohkon sisältämä data voidaan piirtää näytölle, siitä täytyy muodostaa piirrettävä OpenGL-objekti. Tässä luvussa on esitelty projektissa käytetty OpenGL-kirjasto, OpenGL-objektin luomisen perusteet, sekä kuinka yksittäisestä lohkosta muodostetaan OpenGL-objekti.

3.6.1 LWJGL

Light Weight Java Game Library eli LWJGL on kirjasto joka mahdollistaa OpenGL:n natiivien komentojen käyttämisen Java-ohjelmointikielellä. Se ei ole

pelimoottori, eikä se tarjoa olemassa olevien komentojen lisäksi korkeamman tason ominaisuuksia (LWJGL 2016).

LWJGL-kirjaston versio 3.0.0 julkaistiin 3.6.2016 (GitHub 2016a). Aikaisempaan 2.x-versioon verrattuna LWJGL:n mukana tulee GLFW-kirjasto. GLFW on C-kirjasto, joka tarjoaa useita ominaisuuksia muun muassa ikkunoiden hallintaan ja syötteen lukemiseen (GLFW 2016). LWJGL 2.x -versioissa ikkunoiden luontiin ja syötteen lukemiseen ei käytetty GLFW-kirjastoa, vaan omaa ratkaisua (GitHub 2016b). Toteutetussa pelissä käytetään LWJGL 2 -kirjastoa, koska opinnäytetyön alkaessa LWJGL 3 -kirjastoa ei ollut vielä julkaistu.

3.6.2 OpenGL

OpenGL on matalan tason ohjelmointirajapinta, eikä se sisällä valmiita metodeja monimutkaisten muotojen luomiseen. Kaikki OpenGL-grafiikat koostuvat primitiiveistä, eli yksinkertaisista muodoista joita yhdistämällä voidaan muodostaa monimutkaisempia kokonaisuuksia. Yleensä tämä primitiivi on yksinkertaisin mahdollinen tasainen pinta eli kolmio. Toteutetussa ohjelmassa käytettiin quad-primitiiviä eli nelikulmiota kuution sivun muodon vuoksi. Vaihtoehtoisessa toteutetussa ratkaisussa quad-primitiivi korvattiin kahdella, vastakkaisella kolmiolla.

Jokaisella quad-primitiivillä on neljä vertex-pistettä eli kärkipistettä, joiden koordinaatit määrittävät sen muodon. Jokaisella kärkipisteellä on normaali, joka kertoo sen suunnan maailmassa. Suunta vaikuttaa primitiivin kirkkauteen riippuen siitä, missä kulmassa se on maailman valonlähteeseen nähden. Normaalien lisäksi jokaisella kärkipisteellä on tekstuurikoordinaatti, joka määrittää sen primitiivissä käytettävän tekstuurin.

Kuva 16. Ohjelmassa käytössä oleva tekstuurikuvatiedosto eli tekstuuriatlas. Tekstuurin koordinaatit ovat nähtävillä kuvan reunoilla.

Kuvassa 16 on nähtävissä ohjelmassa käytetty kuvatiedosto, joka sisältää kaikki ohjelmassa käytetyt tekstuurit. Tällaisesta useita tekstuureita sisältävästä kuvasta käytetään nimeä tekstuuriatlas. Kuvassa näkyvät koordinaattipisteet ovat tekstuurikoordinaatit kuvan reunapisteissä. Quad-primitiivin teksturoimiseen vasemmassa reunassa olevalla vesi-tekstuurilla tekstuurikoordinaatit olisivat seuraavanlaiset: (0, 0), (0, 1), (1/12, 1) ja (1/12, 0). Tekstuurikoordinaatit kiertävät myötä päivään aloittaen tekstuurin vasemmasta alakulmasta. Koska kuvassa on yhteensä 12 erilaista tekstuuria, vesitekstuurin oikean reunan x-koordinaatti on 1/12.

Yhden OpenGL-objektin luomiseen sen sisältämien primitiivien kärkipisteiden koordinaatit, normaalit ja tekstuurikoordinaatit tallennetaan kolmeen erilliseen liukulukupuskuriin, joiden koko määräytyy piirrettävän OpenGL-objektin kärkipisteiden määrän mukaan kuten koodiesimerkissä. Koodiesimerkissä on esitelty myös liukulukupuskurien luominen.

```
1. /*
2.     Size is determined by the number of coordinates.
3.     vertexSize = 3
4.     normalSize = 3
5.     textureSize = 2
6. */
7. FloatBuffer vertexData = createFloatBuffer(amountOfVertices * vertexSize);
8. FloatBuffer normalData = createFloatBuffer(amountOfVertices * vertexSize);
9. FloatBuffer texData = createFloatBuffer(amountOfVertices * textureSize);
```

Liukulukupuskuri on liukulukuja sisältävä taulukko, jonka sisältämien arvojen perusteella OpenGL luo näytölle piirrettävän OpenGL-objektin. Alla olevassa koodiesimerkissä näytetään yksittäisen quad-primitiivin neljän kärkipisteen vaatimien arvojen syöttäminen liukulukupuskureihin.

```
1. normalData.put(new float[]{0,0,1, 0,0,1, 0,0,1, 0,0,1});
2. vertexData.put(new float[]{0,1,1, 0,0,1, 1,0,1, 1,1,1});
3. texData.put(new float[]{0,0, 1,0, 1,1, 0,1});
```

Koodiesimerkissä käytetyt arvot syötetään selkeyden vuoksi vakioina. Toteutetussa ohjelmassa vakioiden tilalla käytetään muuttujia, joiden arvot riippuvat piirrettävän quad-primitiivin sijainnista ja siitä, mikä kuution sivu on kyseessä.

Kun liukulukupuskureihin on syötetty kaikki halutut arvot, kutsutaan niiden flip-metodia. Tämä tarkoittaa sitä, että liukulukupuskuri vaihdetaan kirjoitustilasta lukutilaan, eli niihin ei olla enää lisäämässä uusia arvoja ja ne ovat valmiita lukemista varten. Tämä on nähtävissä koodiesimerkissä.

```
1.  vertexData.flip();
2.  normalData.flip();
3.  texData.flip();
```

Viimeisenä vaiheena liukulukupuskureiden datasta luodaan Vertex Buffer Object eli VBO, joka voidaan myöhemmin piirtää näytölle. Tämän vaiheen suorittamiseen toteutettu metodi on esiteltynä koodiesimerkissä.

```
1.  public void bindBuffers(Data data) {
2.  
3.      int vboVertexHandle = glGenBuffers();
4.      glBindBuffer(GL_ARRAY_BUFFER, vboVertexHandle);
5.      glBufferData(GL_ARRAY_BUFFER, data.vertexBuffer, GL_STATIC_DRAW);
6.      glBindBuffer(GL_ARRAY_BUFFER, 0);
7.  
8.      int vboNormalHandle = glGenBuffers();
9.      glBindBuffer(GL_ARRAY_BUFFER, vboNormalHandle);
10.     glBufferData(GL_ARRAY_BUFFER, data.normalBuffer, GL_STATIC_DRAW);
11.     glBindBuffer(GL_ARRAY_BUFFER, 0);
12. 
13.     int vboTexHandle = glGenBuffers();
14.     glBindBuffer(GL_ARRAY_BUFFER, vboTexHandle);
15.     glBufferData(GL_ARRAY_BUFFER, data.texBuffer, GL_STATIC_DRAW);
16.     glBindBuffer(GL_ARRAY_BUFFER, 0);
17. }
```

Koodiesimerkin metodi saa parametrina Data-tyypin olion, joka on yksinkertainen apuluokka liukulukupuskureiden kuljettamiseen yhdessä oliossa. Koodiesimerkin rivillä 3 kärkipisteet sisältävää liukulukupuskuria varten luodaan uniikki tunniste, jolla siihen voidaan viitata myöhemmin VBO:ta piirrettäessä. GlBindBuffer-komennolle annetaan parametrina luotu tunniste, jonka jälkeen glBufferData-komennolla sidotaan kärkipisteet sisältävän liukulukupuskurin tiedot kyseiseen tunnisteeseen. Lopuksi tunniste vapautetaan antamalla glBindBuffer-komennon

parametriksi nolla. Sekä normaalit että tekstuurikoordinaatit sisältäville liukuluku-puskureille luodaan samalla tavalla tunnisteet, joihin niiden arvot sidotaan.

Pelissä nämä tunnisteet tallennetaan Handle-nimiseen apuluokkaan, joka tallennetaan hajautustauluun myöhempää käyttöä varten. Tämä on nähtävissä koodiesimerkissä. Tunnisteet sisältävä hajautustaulu toimii samalla periaatteella kuin aikaisemmassa koodiesimerkissä esitelty lohkojen hajautustaulu. Lohkon OpenGL-objektin tunnisteisiin päästään käsiksi lohkon koordinaateilla.

```
1.  private ConcurrentHashMap<Integer, Handle> handles;
2.
3.  int vertexCount = data.vertices;
4.  Handle handle = new Handle(
5.  vboVertexHandle, vboNormalHandle, vboNormalHandle, vertexCount);
6.
7.  handles.put(new Triple(data.chunkX, data.chunkY, data.chunkZ), handle);
```

Tämän jälkeen liukulukupuskureita ei enää tarvita, vaan VBO:ta piirrettäessä käytetään sen uniikkeja tunnisteita. VBO:n piirtämistapa on esiteltynä koodiesimerkissä. Tunnisteiden lisäksi piirtämiseen tarvitaan piirrettävän VBO:n kärkipisteiden määrä.

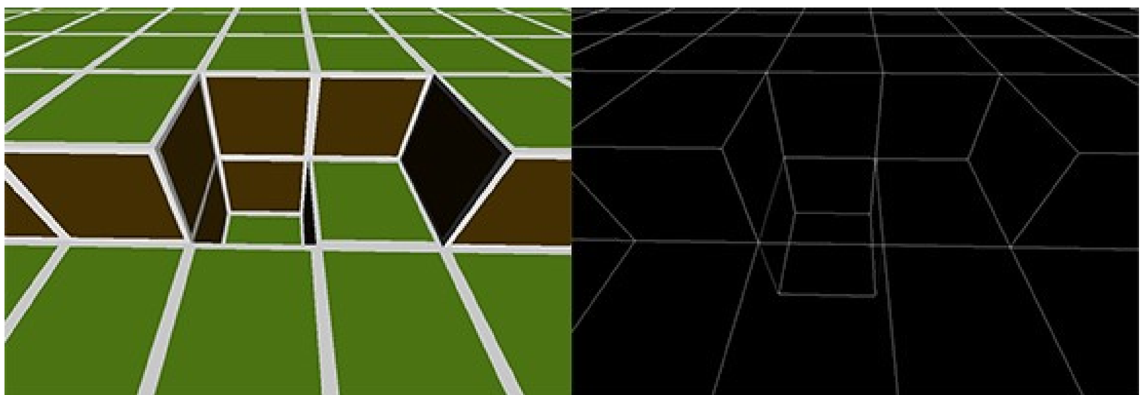
```
1.  int vboVertexHandle = handles.vertexHandle;
2.  int vboNormalHandle = handles.normalHandle;
3.  int vboTexHandle = handles.texHandle;
4.  int vertices = handles.verticeAmount;
5.
6.  glBindBuffer(GL_ARRAY_BUFFER, vboVertexHandle);
7.  glVertexPointer(3, GL_FLOAT, 0, 0L);
8.
9.  glBindBuffer(GL_ARRAY_BUFFER, vboNormalHandle);
10. glNormalPointer(GL_FLOAT, 0, 0L);
11.
12. glBindBuffer(GL_ARRAY_BUFFER, vboTexHandle);
13. glTexCoordPointer(2, GL_FLOAT, 0, 0L);
14.
15. glEnableClientState(GL_VERTEX_ARRAY);
16. glEnableClientState(GL_NORMAL_ARRAY);
17. glEnableClientState(GL_TEXTURE_COORD_ARRAY);
18. glDrawArrays(GL_QUADS, 0, verticeAmount);
19. glDisableClientState(GL_TEXTURE_COORD_ARRAY);
20. glDisableClientState(GL_NORMAL_ARRAY);
21. glDisableClientState(GL_VERTEX_ARRAY);
22.
23. glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Koodiesimerkissä GlBindBuffer-komennoille annetaan parametreina piirrettävän VBO:n tunnisteet. GlVertexPointer, glNormalPointer ja glTextCoordPointer-komennoilla määritetään jokaisen puskurin oikea tyyppi. Tämän jälkeen glEnableClientState-komennoilla määritetään, mitä ollaan piirtämässä. VBO piirretään näytölle glDrawArrays-komennolla, jolle annetaan parametreina käytetyn primitiivin tyyppi sekä kärkipisteiden määrä. Lopuksi palataan alkuperäiseen tilanteeseen poistumalla piirtotiloista ja vapauttamalla puskurit.

3.6.3 Lohkon OpenGL-objektin luonti

Luotaessa OpenGL-objektia yksittäisestä lohkosta on syytä miettiä, tarvitseeko lohkon jokaisen kuution jokaista sivua piirtää. Vain pieni osa lohkon kuutiosta on pelaajan nähtävissä, sillä monet kuutiot ovat maan alla muiden kuutioiden ympäröiminä, eikä niitä ole mahdollista nähdä. Myöskään pelaajan nähtävillä olevista kuutioista tarvitsee vain harvoin piirtää kuution kaikki kuusi sivua.

Tämä on havainnollistettu kuvassa 17, jossa sama maastonosa on nähtävissä sekä tavallisessa pelinäkymässä että käyttäen wireframe-tilaa, jossa piirretään ainoastaan primitiivien ääriviivat. Wireframe-tilassa voidaan huomata, että kuutioille piirretään ainoastaan ne sivut, jotka ovat potentiaalisesti pelaajan nähtävissä. Tällaisia sivuja ovat ne, jotka ovat ilmakuution vieressä.



Kuva 17. Pelinäkymä ja wireframe-tila.

Ohjelmassa lohkon OpenGL-objektin luomiseksi lohkon kolmiulotteisen blocks-taulukon kaikki alkiot käydään läpi ja niitä verrataan kaikkiin kuuteen niiden vieressä olevaan alkioon, eli pelimaailmassa kuution vieressä oleviin kuutioihin. Vertailun tarkoituksena on selvittää, mitkä kuutiot ja mitkä niidet sivut ovat tarpeellisia piirrettäviä. Vertailun tulokset tallennetaan kuuteen kolmiulotteiseen totuusarvomuuttujataulukkoon, yksi taulukko jokaista kuution sivua kohti. Tämä on havainnollistettu koodisesimerkissä.

```
1.  boolean[][][] right = new [CHUNK_SIZE][CHUNK_SIZE][CHUNK_SIZE];
2.  boolean[][][] left = new [CHUNK_SIZE][CHUNK_SIZE][CHUNK_SIZE];
3.
4.  for (int x = 1; x < chunk.blocks.length - 1; x++) {
5.      for (int y = 1; y < chunk.blocks[x].length - 1; y++) {
6.          for (int z = 1; z < chunk.blocks[x][y].length - 1; z++) {
7.              if (chunk.blocks[x][y][z] != Type.AIR) {
8.                  if (chunk.blocks[x + 1][y][z] == Type.AIR)
9.                      right[x][y][z] = true;
10.                 if (chunk.blocks[x - 1][y][z] == Type.AIR)
11.                     left[x][y][z] = true;
12.                 // Continue comparison for all six sides.
13.                 if (chunk.blocks[x][y + 1][z] == Type.AIR)...
```

Koodiesimerkin ensimmäisessä if-lausekkeessa tarkistetaan, onko kuutio tyypiltään ilma-kuutio. Mikäli on, voidaan sen viereisten kuutioiden arvojen tarkistaminen jättää väliin, sillä ilmakuutioita ei piirretä. Tässä on hyvä huomata, että sivuja joita ei piirretä ei tarvitse merkitä totuusarvomuuttujataulukkoihin asettamalla niiden arvoksi false, sillä Java-ohjelmointikielessä se on totuusarvomuuttujan aloitusarvo.

Koodiesimerkissä ei-ilmakuutioiden oikean- ja vasemmanpuoleisen kuution tyyppi tarkistetaan. Tämä tehdään pienentämällä tai suurentamalla x-muuttujan arvoa yhdellä. Viereisen kuution ollessa ilmakuutio kuution sivu merkitään piirrettäväksi asettamalla totuusarvomuuttujataulukon kuution koordinaattien kohdassa olevan alkion arvoksi true. Vastaavalla tavalla käydään läpi lohkon kaikki kuutiot, jotka eivät ole lohkon reunoilla.

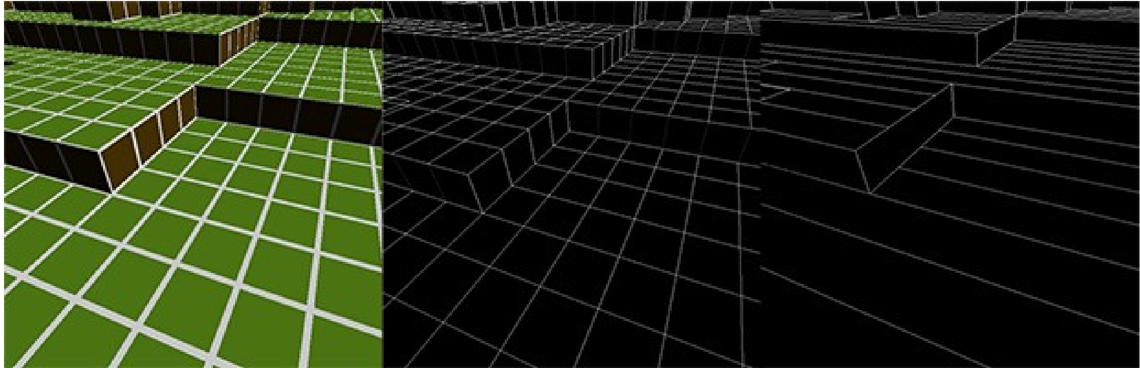
Lohkon reunoilla, eli x-, y- tai z-koordinaattien arvon ollessa 0 tai lohkon leveys vertailua ei voida suorittaa samalla tavalla kuin muissa koordinaateissa, koska osa kuutioiden viereisistä kuutioista kuuluu viereiseen lohkoon. Tästä johtuen

vertailun tekemiseksi lohkolla täytyy olla tiedot sen vierekkäisistä kuudesta lohkosta. Ennen kuutioiden vertailun aloittamista nämä kuusi lohkoa haetaan kuten seuraavassa koodiesimerkissä.

```
1.  Chunk rightChunk = null;
2.  Chunk leftChunk = null;
3.  /*
4.  If the requested chunk is not available in an uncompressed state,
5.  load it from the compressed state instead (slower).
6.  */
7.  rightChunk = chunkManager.getActiveChunk(chunk.x + 1, chunk.y, chunk.z);
8.  if (rightChunk == null)
9.    rightChunk = chunkManager.getCompressedChunk(chunk.x + 1, chunk.y, chunk.z);
10.
11. leftChunk = chunkManager.getActiveChunk(chunk.x - 1, chunk.y, chunk.z);
12. if (leftChunk == null)
13.   leftChunk = chunkManager.getCompressedChunk(chunk.x - 1, chunk.y, chunk.z);
```

Kaikkien sivujen piirtämisestä siirtyminen ainoastaan ilmakuutioiden vieressä oleviin sivuihin on välttämätöntä suorituskyvyn kannalta. Testimielessä identtinen pelimaailma ladattiin ensin piirtäen kuutioiden kaikki pinnat, sen jälkeen piirtämällä ainoastaan ilmakuutioiden vieressä olevat. Jälkimmäisessä piirrettävien kärkipisteiden määrä oli noin 75 000, vain 0.2% ensimmäisen noin 32 000 000 kärkipisteen määrästä.

Edellä mainitun parannuksen lisäksi piirrettävien primitiivien määrää vähennettiin käyttämällä Greedy Meshing -algoritmia. Tällä algoritmilla vierekkäiset, samaa tekstuuria käyttävät kuutioiden pinnat piirretään yhdessä, eikä jokaiselle tarvita omaa quad-primitiiviä. Käytettävää tekstuuria toistetaan useita kertoja peräkkäin, jolloin maaston ei visuaalisesti tule minkäänlaista muutosta. Kuvassa 18 on nähtävissä ero primitiivien määrässä ja koossa ennen ja jälkeen algoritmin toteutuksen.



Kuva 18. Greedy Meshing -algoritmi. Vasemmalla pelinäkymä, keskellä wireframe-tilan näkymä ennen algoritmin toteutusta, oikealla algoritmin toteutuksen jälkeen.

Testauksessa todettiin algoritmin vähentävän piirrettävien primitiivien määrän vajaaseen puoleen, riippuen maaston tasaisuudesta. Hyvin tasaisessa maastossa algoritmi toimii huomattavasti tehokkaammin.

3.7 Pääsilmukka

Pelin pääsilmukalla tarkoitetaan ohjelmakoodin osaa, joka toistuu silmukassa koko pelaamisen ajan. Pelissä käytetty pääsilmukka on nähtävillä koodiesimerkissä.

```
1.  while (!Display.isCloseRequested() && !Keyboard.isKeyDown(Keyboard.KEY_ESCAPE)){
2.      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
3.      processInput();
4.      chunkManager.processBufferData();
5.      render();
6.      renderDebugText();
7.      Display.update();
8.      Display.sync(60);
9.  }
```

Silmukasta poistumisen ehtoina on peli-ikkunan sulkeminen, tai ESC-näppäimen painaminen. Yksi kierros silmukassa vastaa yhtä ruudunpäivitystä pelissä. Tämä tarkoittaa sitä, että ruudunpäivitysnopeuden ollessa 60 ruutua sekunnissa, uusi kierros suoritetaan noin 17 millisekunnin välein.

Silmukan alussa suoritettava glClear-toiminto tyhjentää peli-ikkunan kaikista edellisellä kierroksella piirretyistä OpenGL-objekteista uutta piirtämistä varten.

ProcessInput-metodissa luetaan pelaajan syöte näppäimistöltä ja hiirestä. Syötteellä pelaaja voi suorittaa erilaisia toimintoja, kuten liikkua maailmassa, vaihtaa katseen suuntaa tai muokata maastoa.

Pääsilmukassa suoritetaan myös processBufferData-metodi, jonka sisällä kaikista valmiiksi luoduista lohkojen liukulupuskureista luodaan jokaisesta VBO. Tähän käytetään aikaisemmin esiteltyä metodia.

Näiden lisäksi silmukka sisältää render-metodin, jonka sisällä piirretään koko näkyvä pelimaailma. Pelaajan ympärille piirretään haluttuun piirtoetäisyyteen asti lohkoja, mikäli ne ovat jo olemassa. Tämä on nähtävissä koodiesimerkissä.

```
1. int playerChunkX = getCurrentChunkXId();
2. int playerChunkZ = getCurrentChunkZId();
3. for (int x = -chunkRenderDistance; x <= chunkRenderDistance; x++) {
4.     for (int z = -chunkRenderDistance; z <= chunkRenderDistance; z++) {
5.         for (int y = 0; y < Chunk.VERTICAL_CHUNKS; y++) {
6.           Handle handle = getHandle(playerChunkX + x, y, playerChunkZ + z);
7.             if (handle != null) {
8.                 // Render the chunk using the handle.
```

Koodiesimerkissä pelaajan ympärillä olevat lohkot käydään yksi kerrallaan läpi kolmen for-silmukan avulla. Jokaisen lohkon koordinaateissa tarkistetaan, onko siinä kohdassa oleva lohko piirrettävissä. Mikäli on, se piirretään aiemmin esiteltyä piirtometodia käyttäen.

3.8 Pelaajan liikkuminen

Peli-ikkunaan piirtyvä pelinäkymä riippuu pelaajan sijainnista ja katseen suunnasta. Pelaajan katseen eli kameran kontrollointiin käytettiin esimerkkiluokkaa, jossa oli valmiina kameran liikuttaminen sekä sen suuntaaminen hiiren avulla.

Tätä luokkaa muokattiin peliin sopivaksi lisäämällä siihen yksinkertainen fysiikkamallinnus sekä törmääminen pelin maastoon. Tämä tarkoittaa, että ollessaan ilmassa pelaaja putoaa kiihtyvällä vauhdilla alaspäin, kunnes törmää maahan. Maastosta irrotetut kuutiot toimivat samoilla fysiikoilla. Pelaajalle toteutettiin myös

hyppäämistoiminto, jolla voidaan ylittää maastossa olevia matalia esteitä tai kivetä loivaa rinnettä ylöspäin.

Jotta pelaaja voi kävellä maastonpintaa pitkin, täytyy tietää, minkä tyyppinen kuutio pelaajan alla on. Tämä saadaan selville pelaajan koordinaattien avulla käyttäen aikaisemmin esiteltyä metodia, jolla voidaan hakea annetuissa koordinaateissa olevan kuution tyyppi. Pelaajan putoaminen on esiteltynä koodiesimerkissä.

```
1.  // x, y and z are player coordinates.
2.  Vector3f blockPos = new Vector3f(x, y - fallingSpeed - pHeight, z);
3.  byte block = chunkManager.getBlock(blockPos);
4.  if (block == Type.AIR) {
5.      fallingSpeed += fallingSpeedIncrease;
6.      y -= fallingSpeed;
7.  } else {
8.      y = (int) y;
9.      fallingSpeed = 0;
```

Koodiesimerkki suoritetaan osana jokaista pääsilmukan kierrosta pelaajan syötteen yhteydessä. Mikäli pelaajan alla oleva kuutio on tyypiltään ilmaa, pelaajan y-koordinaatin arvoa pienennetään sen hetkisellä putoamisnopeudella, ja putoamisnopeuden arvoa kasvatetaan kiihtyvän putoamisliikkeen aikaansaamiseksi. Siinä tapauksessa, että alla oleva kuutio on maastoa, pelaaja y-koordinaatti asetetaan maaston pinnan tasolle leikkaamalla y-koordinaatista desimaalin jälkeiset luvut pois.

Peliin toteutettiin myös askelten äänet riippuen maastosta, jolla pelaaja kävelee. Äänitiedoston lataamiseen ja toistamiseen käytettiin TinySound-nimistä kirjastoa. Kirjaston tarkoituksena on helpottaa erilaisten äänitiedoston lataamista ja niiden toistamista. Koodiesimerkissä on esitelty äänitiedoston lataaminen, sen toistaminen ja toistamisen lopettaminen kyseisellä kirjastolla.

```
1.  Sound onGrass = TinySound.loadSound(
2.                  Voxels.class.getClassLoader().getResource(
3.                  "resources/sounds/walk_grass.wav"));
4.  onGrass.play();
5.  OnGrass.stop();
```

Pelaajan alla olevan kuution tyyppi määrää, mitä askelten ääntä käytetään. Ohjelmaan toteutettiin erilaiset kävelyäänet sekä kivialustalle että ruohikolle. Askelten äänten toistamisen ehtona on, että pelaaja ei ole ilmassa, ja että jokin pelaajan liikuttamiseen käytetyistä näppäimistä on pohjassa.

3.9 Monisäikeisyys

Ilman säikeiden käyttämistä Java-ohjelmat toimivat vain yhdessä säikeessä. Tämä tarkoittaa sitä, että mikäli tietokoneen prosessori sisältää useamman kuin yhden ytimen, ohjelma ei hyödynnä koko prosessorin laskentatehoa. Säikeiden avulla Java-ohjelmassa voidaan suorittaa yhtäaikaisesti useampia prosesseja, jotka jakautuvat useammalle tietokoneen prosessorin ytimelle. Sopivalla toteutustavalla säikeiden käyttö voi nopeuttaa ohjelman toimintojen suoritusta moninkertaisesti.

Säikeistys on hyödyllistä myös pelin käyttömukavuuden kannalta. Mikäli koko ohjelma toimii yhdessä säikeessä, yli 17 millisekuntia vievien prosessien suorittaminen johtaa ruudunpäivitysnopeuden hidastumiseen tai hetkelliseen pysähtymiseen. Säikeiden käytön avulla ohjelman pääsilmukka voi toimia keskeytyksettä omassa säikeessään, kun samanaikaisesti muissa säikeissä voidaan suorittaa muita, mahdollisesti pidemmän aikaa vaativia prosesseja.

Monisäikeisyyttä ei kuitenkaan voi käyttää minkä tahansa prosessin nopeuttamiseksi. Yksittäisessä säikeessä suorittavan osan tulee olla muista osista riippumaton, itsenäinen osa, joka voidaan suorittaa tarvitsematta muissa säikeissä olevia osia.

Voxels-pelissä monisäikeisyyttä hyödynnetään lohkojen sekä niiden OpenGL-datojen laskentaan. Koska yksittäisen lohkon ja sen OpenGL-datan laskenta voidaan suorittaa itsenäisenä osana eikä se riipu muista lohkoista, se soveltuu hyvin prosessiksi, jota voidaan suorittaa useammassa säikeessä samanaikaisesti.

OpenGL:n komentoja voidaan kuitenkin suorittaa ainoastaan aktiivisessa pääsäikeessä. Tämä on syy siihen, miksi aikaisemmin esitellyssä koodiesimerkissä processBufferData-metodi suoritetaan pääsäikeessä pääsilmukan sisällä eikä säikeissä, joissa lohkot ja niiden OpenGL-data lasketaan valmiiksi.

```
1.  int maxThreads = Runtime.getRuntime().availableProcessors();
2.  ChunkMaker[] threads = new ChunkMaker[maxThreads];
3.
4.  //If there's a free thread available, create a new chunk
5.  if(hasFreeThreads()){
6.      int threadId = getFreeThread();
7.      threads[threadId] = new ChunkMaker(newChunkX, newChunkY, newChunkZ);
8.      threads[threadId].setPriority(Thread.MIN_PRIORITY);
9.      threads[threadId].start();
10. }
11.
12. private boolean hasFreeThreads() {
13.     for (int i = 0; i < threads.length; i++)
14.         if (threads[i] == null)
15.             return true;
16.     return false;
17. }
18.
19. private int getFreeThread() {
20.     for (int i = 0; i < threads.length; i++)
21.         if (threads[i] == null)
22.             return i;
23.     // Error, no free thread available
24.     return -1;
25. }
```

Koodiesimerkissä on esitelty periaate, kuinka säikeitä käytetään pelissä uusien lohkojen luomiseen. Pelin käynnistyessä luodaan ChunkMaker-taulukko, jonka koko riippuu tietokoneen prosessorin säikeiden määrästä. Pelaamisen aikana uusia lohkoja luotaessa taulukosta tarkistetaan, onko siinä vapaita paikkoja uudelle säikeelle. Mikäli on, siihen luodaan uusi ChunkMaker-olio halutuilla lohkon koordinaateilla. Koodiesimerkin rivin 9 start-metodi käynnistää ChunkMaker-oliossa olevan säikeille ominaisen run-metodin, jossa lohkon luonti tapahtuu.

Toteutuksessa jokaista säiettä käytetään vain yhden lohkon luomiseen, jonka jälkeen säikeelle ei ole enää käyttöä. Jotta käytetyn säikeen tilalle voidaan luoda uusi säie uuden lohkon luomista varten, käytetyt säikeet tulee poistaa taulukosta. Tämä on esiteltynä koodiesimerkissä. Jokainen run-metodinsa suorittanut säie merkitään tyhjäksi.

```
1. for (int i = 0; i < threads.length; i++) {
2.     if (threads[i] != null)
3.         // Finished creating a chunk?
4.         if (!threads[i].isAlive())
5.             threads[i] = null;
6. }
```

Esitellystä toteutuksesta on huomioitavaa, että siinä ei käytetä Java-ohjelmointikielen kehittyneempiä säikeidenhallintamenetelmiä, kuten esimerkiksi Thread Pool -luokkaa. Näiden käytöstä ei esitellyn säietoteutuksen toteuttamisen aikana ollut vielä tietoa. Koska oma säikeidenhallintatoteutus on kuitenkin toimiva, ei myöhemmässä vaiheessa löydetyn Thread Pool -luokan käyttöönotolle ollut tarvetta.

4 Toteutuksen aikana tehdyt parannukset

4.1 Lohkoihin jako

Ohjelman varhaisessa vaiheessa maastoa ei ollut jaettu lohkoihin, vaan koko maasto oli yhdessä isommassa OpenGL-objektissa. Tämän toteutuksen rajoituksena oli kuitenkin maaston staattisuus, sillä mikäli maastoa olisi haluttu laajentaa tai muokata, koko maailman sisältävä OpenGL-objekti olisi täytynyt päivittää, mikä ei ollut sen koon vuoksi mahdollista tehdä reaaliaikaisesti.

Tämä teki myös ohjelman käynnistymisestä hitaampaa, sillä ennen kuin maailmaa voitiin piirtää, koko ennalta määritellyn kokoinen alue täytyi laskea kokonaan valmiiksi ja luoda siitä OpenGL-objekti. Tähän verrattuna lohkoja käyttävä peli käynnistyy erittäin nopeasti, sillä siinä tarvitsee laskea valmiiksi ainoastaan pelaajan sisältävä lohko, jonka jälkeen pelaaja voi alkaa liikkua muun maailman muodostuessa pelaajan ympärille.

Lohkoista koostuva maailma tekee ohjelman rakenteesta huomattavasti monimutkaisemman, mutta sen etuna on maailman nopea muokattavuus ja dynaamisuus.

4.2 Datan pakkaaminen ja purkaminen

Lohkojen pakkaaminen ja tallentaminen erilliseen hajautustauluun niiden ollessa kaukana pelaajasta lisättiin peliin melko myöhäisessä vaiheessa. Pelattavuuden kannalta se oli kuitenkin ratkaiseva optimointi, sillä ennen sen toteuttamista ohjelman muistinkulutus kasvoi nopeasti liian suureksi pelaajan tutkiessa maailmaa pidemmän aikaa ja kartoitettaessa uusia alueita.

Yhden lohkon käyttämä tietokoneen muistin määrä koostuu lähes kokonaan sen sisältämästä kolmiulotteisesta tavutaulukosta joka sisältää tiedot siitä, minkälainen kuutio missäkin kohtaa on. Suunta antavasti voidaan laskea yhden pakkaamattoman lohkon käyttämän muistin määräksi noin 32 * 32 * 32 tavua, eli noin 32 kilotavua. Pelaajan aktiivinen alue näköetäisyyden ollessa yhdeksän lohkoa koostuu yhteensä 19 * 19 * 8 lohkosta, joten niiden käyttämä muisti on jo noin 95 megatavua. Tästä arvosta voidaan arvioida, millä nopeudella pelin muistinkulutus kasvaisi pelaajaan liikkuessa maailmassa, jos datan pakkaus ei olisi käytössä.

Yksittäisen lohkon data on kuitenkin erittäin hyvin pakkautuvaa, sillä se sisältää hyvin paljon suuria osuuksia samoja arvoja, kuten ilma- tai maastokuutioita. Lohkojen datan pakkauksen ollessa käytössä ohjelman kuluttama tietokoneen muistin määrä pysyykin melko vakiona. Lopullisena optimointina voitaisiin toteuttaa kaukaisten, epäaktiivisten lohkojen tallentaminen pakatusta tilasta käyttäjän kovalevylle. Tätä ei kuitenkaan ole tässä vaiheessa ohjelmaa toteutettu.

4.3 Monisäikeistys

Säikeiden lisäys lohkojen luomiseen moninkertaisti nopeuden, jolla pelimaailman maastoa luodaan. Pelin varhaisessa vaiheessa lohkojen luontiin ei käytetty säikeitä lainkaan, vaan niitä luotiin pääsäikeessä pääsilmukassa. Sen lisäksi, että luomiseen käytettiin vain yhtä prosessoriydintä, ongelmana oli ajoittainen ruudunpäivitysnopeuden hidastuminen, jos lohkojen luominen kesti yhtä ruudunpäivitystä pidempään.

Ensimmäisenä ratkaisuna tähän toteutettiin yksittäinen säie, jossa luotiin kaikki lohkot. Tämä ratkaisi ruudunpäivitysnopeuteen liittyvät ongelmat säikeen ollessa käynnissä samanaikaisesti pääsäikeen kanssa. Ratkaisu ei kuitenkaan edelleenkään hyödyntänyt kuin yhtä prosessoriydintä. Yhden lohkosäikeen toteutusta kehitettiin nykyiseen toteutukseen dynaamisesti skaalautuvaksi hyödyntämään kaikkia saatavilla olevia prosessorin ytimiä.

5 Jatkokehitysmahdollisuudet

5.1 Vesi

Tämänhetkisessä toteutuksessa vedellä ei ole minkäänlaisia vedenomaisia piirteitä sinistä tekstuuria lukuun ottamatta eikä fysiikkamallinnusta. Mielenkiintoinen jatkokehitysmahdollisuus olisi tehdä vedestä virtaavaa, jolloin se valuisi mäkiä pitkin alaspäin. Tämä on ollut kehityksen alla ja potentiaalisia vaihtoehtoja on muutamaa jo kokeiltu, mutta tähän opinnäytetyöhön niitä ei saatu valmiiksi asti.

Suurimpana ongelma virtaavan veden toteutuksessani on potentiaalisesti liian suureksi kasvava maaston päivitysten määrä, jos veden virtaus leviää suuremmalle alueelle. Tämä johtaa siihen, että sekä virtaavien vesikuutioiden että lohkojen päivitysten suuri määrä hidastaa tai jopa kaataa ohjelman. Yhtenä ratkaisuna päivitysten suureen määrään on pohdittu veden käsittelemistä erillisenä osana muusta maastosta.

5.2 Graafinen ulkoasu

Graafiselta ulkoasultaan peli on tällä hetkellä hyvin pelkistetty ja yksinkertainen. Grafiikat eivät ole olleetkaan projektissa tärkeänä kehityksen kohteena, vaan toteutuksessa on pikemminkin pyritty minimaaliseen mutta toimivaan ratkaisuun.

Graafisen ulkoasun merkittävä parantaminen vaatisi shader-ohjelmien eli varjostimien käyttöönottamista. Varjostimilla olisi mahdollista lisätä maailmaan muun muassa auringonvalosta aiheutuvat realistiset varjot, valonlähteillä valaistavat pimeät luolat, sekä kauniimman näköinen, aaltoileva vedenpinta vesialueisiin ja virtaavaan veteen.

Mikäli projektia olisi alusta asti toteutettu minimaalisiakin varjostimia käyttämällä, olisi näiden ominaisuuksien lisääminen huomattavasti helpompaa. Nykyinen toteutus vaatisi jonkin verran muutoksia OpenGL-osuuteen, jotta varjostimet saataisiin otettua käyttöön. Tutkimusta ja muutoksia ohjelman rakenteeseen varjostimien käyttöönottoa varten on jo jonkin verran tehty.

5.3 Ohjelman kontrolloimat hahmot

Ohjelman kontrolloimien hahmojen lisääminen ohjelmaan olisi mielenkiintoista, sillä se mahdollistaisi useiden pelillisten elementtien kehittämisen ohjelmaan. Tällaisia pelillisiä elementtejä voisivat olla esimerkiksi interaktio hahmojen kanssa, taistelulliset elementit, sekä pelaajan taitojen kehittäminen näiden seurauksena.

Ohjelman kontrolloimista hahmoista on jo tehty yksinkertainen prototyyppi, jota ei tähän opinnäytetyöhön sisällytetty. Kehitysvaiheessa olevat hahmot esitetään graafisesti eri värisinä kuutioina, jotka liikkuvat maailmassa hyppimällä. Hahmojen ainoana tavoitteena on hyppiä pelaaja kohti, mutta minkäänlaista reitinetsintää niille ei ole toteutettu, vaan ne pyrkivät liikkumaan pelaajaa kohti välittämättä edessä olevista esteistä.

6 Pohdinta

Opinnäytetyöni oli aiheeltaan mielenkiintoinen ja se toteuttaminen oli hyvin opettavaista. Ennen Voxels-pelin toteutusta olin aikaisemmin tehnyt muutamia pienempiä peliprojekteja Javalla, mutta laajuudeltaan sekä graafiselta osuudeltaan tämä oli kuitenkin omaa luokkaansa.

Ennen pelin toteutusta olin jo jonkin aikaa ollut kiinnostunut monimutkaisempien grafiikoiden luomisesta ja lisäämisestä peleihin. Aloittaessani pelin toteutuksen olin tutustunut OpenGL-ohjelmoinnin perusteisiin ja tehnyt muutaman harjoitustehtävän, mutta taitoni olivat kuitenkin hyvin alkeelliset. Projektin edetessä taidot sekä OpenGL-ohjelmoinnin että Javan osalta kasvoivatkin huomattavasti. Tästä johtuen toteutuksen aikana tulikin uusia ideoita, kuinka useita aikaisemmin jo toteutettua ominaisuuksia voisi toteuttaa paremmin. Pelin kehityksen kannalta näistä tärkeimmät toteutettiin, mutta monia ideoita jätettiin myös mahdollista myöhempää kehitystä varten.

Valmiisiin kehitysympäristöihin kuten Unityyn verrattuna pelin toteuttaminen alusta asti itse on huomattavasti hitaampaa. Toisaalta itse tehdyssä toteutuksessa ohjelman toteutuksesta voi tehdä juuri sellaisen kuin haluaa ja pelin toimintaa voi muokata syvemmällä tasolla kuin valmiissa kehitysympäristössä. Mikäli jossakin vaiheessa aloitan uuden 3D-grafiikoita hyödyntävän peliprojektin, toteutan sen kuitenkin luultavasti valmiissa kehitysympäristössä.

Lähteet

Biagoli, A. 2014. Understanding Perlin Noise. https://flafla2.github.io/2014/08/09/perlinnoise.html. 30.9.2016.

Big-O Cheat Sheet 2016. Know Thy Complexities!. http://bigocheatsheet.com/. 3.10.2016.

GitHub 2016a. LWJGL 3. https://github.com/LWJGL/lwjgl3/releases. 4.10.2016.

GitHub 2016b. LWJGL 3. https://github.com/LWJGL/lwjgl3-wiki/wiki/2.6.3-Input-handling-with-GLFW. 4.10.2016.

GLFW 2016. What is GLFW?. http://www.glfw.org/faq.html#what-is-glfw. 4.10.2016.

Gustafson, S. 2005. Simplex noise demystified. Linköpingin yliopisto. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. 3.10.2016.

LWJGL 2016. What is LWJGL 3?. https://www.lwjgl.org/#learn-more. 4.10.2016.

Moeller, R. 2014. Benchmark. https://github.com/RuedigerMoeller/fast-serialization/wiki/Benchmark. 4.10.2016.

Oracle 2016a. Class ConcurrentHashMap <K,V>. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html. 3.10.2016.

Oracle 2016b. Serializable Objects. https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html. 4.10.2016.

Oracle 2016c. Class Deflater. https://docs.oracle.com/javase/7/docs/api/java/util/zip/Deflater.html. 4.10.2016.

Processing Foundation 2016. Noise(). https://processing.org/reference/noise_.html. 2.10.2016

Saloranta, T. 2014. LZF Compressor. https://github.com/ning/compress#lzf-compressor. 4.10.2016.

Shiffman, D. 2012. The Nature of Code. http://natureofcode.com/book/introduction/. 2.10.2016.

Zucker, M. 2011. The Perlin noise math FAQ. https://web.archive.org/web/20150607183420/http:/webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html. 29.9.2016.