Tommi Kilpeläinen

Proseduraalinen maailma 2D-peleihin

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

27.5.2022

Tiivistelmä

Tekijä:	Tommi Kilpeläinen
Otsikko:	Proseduraalinen maailma 2D-peleihin
Sivumäärä:	50 sivua
Aika:	27.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Antti Laiho

Insinöörityö toteutettiin omaan peliprojektiin suunniteltujen ominaisuuksien teknisten haasteiden ratkaisemiseksi ja niiden toteutuskelpoisuuden varmistamiseksi. Tavoitteena oli kehittää tapa luoda GameMaker-pelimoottorilla proseduraalisia pelimaailmoja, jotka sopisivat ylhäältäpäin kuvatun kaksiulotteisen hiekkalaatikkopelin kartoiksi. Pelimaailmojen haluttiin olevan rakenteeltaan monipuolisia ja ulkoasultaan näyttäviä sekä muokattavissa pelin suoritusaikana niin, että visuaalinen ilme säilyisi yhtenäisenä.

Työssä perehdyttiin GameMaker-pelimoottorin ominaisuuksiin ja sen käyttämään GameMaker Language -ohjelmointikieleen, jolla proseduraalista maailman luontia käyttävät sovellukset oli tarkoitus toteuttaa. Pelimoottorin kehityshistoriaa ja ominaisuuksia tarkasteltiin, jotta pystyttiin varmistumaan sen soveltuvuudesta haluttuun käyttötarkoitukseen. GameMakerin osalta kartoitettiin sen eroavaisuuksia usein pelikehityksessä käytetyn Unity-pelimoottoriin, ja samalla tutustuttiin työn toteutuksen kannalta GameMakerin tärkeimpiin tietorakenteisiin ja niiden ominaispiirteisiin.

Tärkeänä osana työtä käytiin läpi proseduraalisen sisällön tuotantoa peleissä sekä proseduraalisen kartan generointiin valitun Perlin noise -algoritmin toimintaperiaatteet, joiden pohjalta kehitettiin käyttötarkoitukseen soveltuva funktio. Tuotettuja karttoja analysoitaessa huomattiin ongelmia niiden rosoisuudessa ja ennalta-arvattavuudessa, joita varten kehitettiin menetelmät, joilla ongelmat saatiin ratkaistua.

Luotujen karttojen visualisointia varten perehdyttiin GameMakerin tarjoamiin suoristuskyvyltään tehokkaisiin maastolaattoihin ja toteutettiin kolme tekniikkaa, joilla voitiin asetella kartan eri materiaalien maastolaattojen reunapalat automaattisesti suoritusaikana. Tekniikat poikkesivat toisistaan tarvittavien reunapalojen ja tile-kerrosten määrän suhteen, joista jokainen tekniikka ratkaisi ilmenneitä visuaalisia haasteita edelliseen tekniikkaan nähden. Yksikään tekniikka ei kuitenkaan yksinään ollut ongelmaton tai soveltunut käytettäväksi jokaisessa tilanteessa ilman kompromisseja.

Lopputuloksena syntyi apuohjelma proseduraalisten karttojen esikatseluun sekä peliprojektin alku, jossa karttojen toimivuutta ja maastolaattojen asettelua testattiin käytännössä. Projektien perusteella voitiin todeta kehitetyt tekniikat toimiviksi ja suorituskyvyltään tarpeeksi tehokkaiksi, jotta peliprojektia voitaisiin jatkaa valmiiksi asti.

Avainsanat: 2D, GameMaker, pelikehitys, Perlin noise, proseduraalinen maailmanluonti, tile-järjestelmä

Abstract

Author:	Tommi Kilpeläinen
Title:	Procedural world for 2D games
Number of Pages:	50 pages
Date:	27 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Game Applications
Supervisors:	Antti Laiho, Senior Lecturer

The purpose of this thesis was to solve the technical challenges of the features designed for the personal game project and to ensure their feasibility. The goal was to develop a way to create procedural game worlds with the GameMaker game engine that would serve as top-down maps of the two-dimensional sandbox game. The game worlds were intended to be versatile in structure and have impressive appearance. The maps also had to be customizable during game execution in a way that the visual look remained consistent.

The thesis introduces the features of the GameMaker game engine and its programming language GameMaker Language. The goal was to implement their applications which are using the procedural world creation. The development history and features of the game engine were examined to ensure its suitability for the intended use. The features of GameMaker were compared to the Unity game engine, which is often used in game development, and the most important data structures of GameMaker and their characteristics were introduced.

This study focused on the production of procedural content in videogames and the operating principles of the Perlin noise algorithm chosen to generate the procedural map, on the basis of which a function suitable for world creation was developed. When analyzing the produced maps, problems with their roughness and predictability were identified, for which methods were developed to solve the problems.

To visualize the created maps, the high-performance terrain tiles provided by GameMaker were introduced, and three techniques were implemented to automatically place the edge pieces of terrain tiles of different materials at run time. The techniques differed in the number of edge pieces and tile layers required, each technique solving the visual challenges occurred to the previous technique. However, no single technique was unproblematic or suitable for use in every situation without compromises.

As a result, a utility for previewing procedural maps was created, as well as the beginning of a game project in which the functionality of the maps and the automatic placement of the tiles were tested in practice. Based on the projects, it could be stated that the developed technologies were functional and efficient enough, so that the game project can be further developed until the finished product.

Keywords: 2D, GameMaker, game development, Perlin noise, procedural world creation, tile system

Sisällys

Lyhenteet ja käsitteet

GML: *GameMaker Language*. GameMaker-pelimoottorin käyttämä pelikehitykseen suunniteltu skriptikieli.

Tile: Graafinen pelielementti taustojen ja tasojen tehokkaaseen piirtämiseen videopeleissä. Yleisin käyttötarkoitus on käyttää tile-elementtejä maastolaattoina, joilla pelimaailma visualisoidaan.

Tile-sarja: Tile set. Ruudukon tapaan osiin jaettu kuvasarja, jonka jokainen ruutu edustaa yhtä tile-elementtiä.

Tile-kartta: Tile map. Ruudukon kaltainen tietorakenne, joka pitää sisällään pelimaailmaan aseteltujen tile-elementtien tiedot, sekä tietorakenteelle osoitetun tile-sarjan, jos tilejä voi valita asettelua varten.

Tile-kerros: Tile layer. GameMaker-pelimoottorin tile-resursseille varattu syvyysnumeroitu kerros, jolle tile-kartan tile-elementit piirretään suhteessa muihin tile- tai muiden resurssien kerroksien syvyyteen.

1 Johdanto

Pelilaitteiden suorituskyvyn kasvaessa monet pelikehittäjät ovat alkaneet suosia pelejä, jotka tarjoavat proseduraalista sisältöä eli säännönmukaisella satunnaisuudella tuotettuja pelielementtejä. Proseduraalista sisältöä sisältävät pelit tyypillisesti tarjoavat enemmän pelattavaa verrattuna perinteisiin videopeleihin. Näissä peleissä on usein myös enemmän vapauksia vuorovaikuttaa pelimaailman kanssa, ja ne saattavat sisältää erilaisia pelielementtien vuorovaikutussimulaatioita, jotka rohkaisevat pelaajaa keksimään luovia ratkaisuja edetäkseen pelissä. Tämänkaltaiset niin kutsutut hiekkalaatikkopelit ovat nousseet erittäin suosituiksi pelaajien keskuudessa.

Proseduraalisen pelin kehityksessä on kuitenkin paljon haasteita, jotka liittyvät mielekkään sisällön tuottamiseen ja suorituskykyongelmien ratkaisuun. Varsinkin proseduraalisesti rakennettujen maailmojen luominen vaatii täysin erilaista lähestymistapaa kuin perinteinen kenttäsuunnittelu, jossa pelielementit asetellaan harkitusti omille paikoilleen.

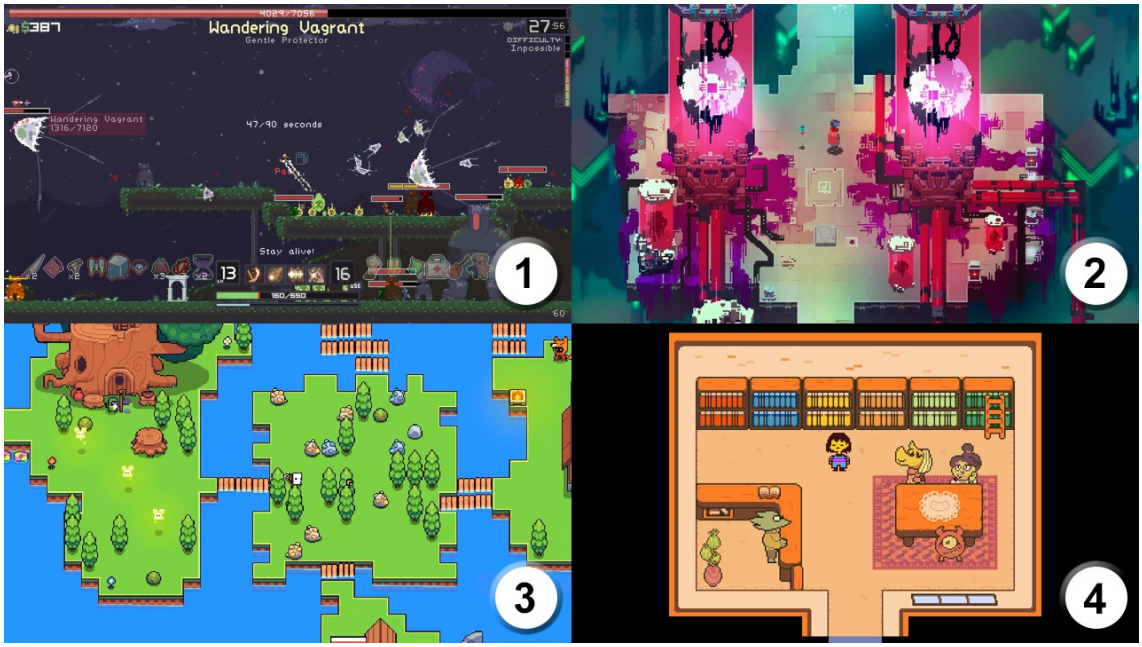
Insinöörityössä tarkastellaan ylhäältä kuvatun kaksiulotteisen kartan proseduraalista luomista ja sen visualisointia maastolaatoilla käyttäen GameMaker-pelimoottoria. Tavoitteena oli tuottaa monipuolisia ja visuaalisesti näyttäviä maailmoja, jotka olisivat myös rakenteeltaan sopivia pelimaailmoiksi.

Insinöörityössä perehdytään GameMaker-pelimoottorin ja GameMakerin käyttämän GameMaker Language -skriptikielen tärkeimpiin ominaispiirteisiin ja tietorakenteisiin. Lisäksi työssä esitellään myös esimerkkikoodia kartan avainelementtien luomiseen sekä tapoja toteuttaa automaattinen maastolaattojen asettelu pelin suoritusaikana. Ratkaisut pohjautuvat tunnettuihin algoritmeihin ja insinöörityötä tehdessä kokemuksen kautta opittuihin tapoihin lähestyä haasteita, jotka liittyvät proseduraaliseen sisällöntuotantoon.

2 GameMaker-pelimoottori

GameMaker (https://gamemaker.io) on kaksiulotteisten pelien kehittämistä varten tehty työkalu, joka suunniteltiin alun perin niille pelikehittäjille, joilla ei vielä ollut ohjelmointiosaamista, mutta on vuosien aikana kehittynyt palvelemaan myös edistyneitä ohjelmistokehittäjiä. GameMaker mahdollistaa myös kolmiulotteisten pelien tekemisen, mutta se on ominaisuuksiltaan hyvin rajoittunut muihin tarjolla oleviin pelimoottoreihin nähden. GameMaker mahdollistaa pelin kääntämisen kaikille moderneille konsoleille sekä yleisimmille pöytäkone- ja mobiilialustoille, minkä lisäksi kääntäminen onnistuu myös älytelevisioille sekä selaimissa toimivalle HTML5-kielle. [1.] GameMakeria kehitetään aktiivisesti, ja sen perusversio on ilmainen, mutta pelin julkaisemista varten tarvittavat lisenssit ovat kuukausimaksullisia.

GameMakerilla on tehty useita kaupallisia menestyksiä, kuten Spelunky Classic, Risk of Rain, Hyper Light Drifter, Forager ja Undertale (kuva 1) [2; 3 ;4; 5; 6]. Tyypillistä näillä peleille on se, että ne on pystytty toteuttamaan pienellä budjetilla ja kehitystiimillä tai jopa yksilöprojekteina.

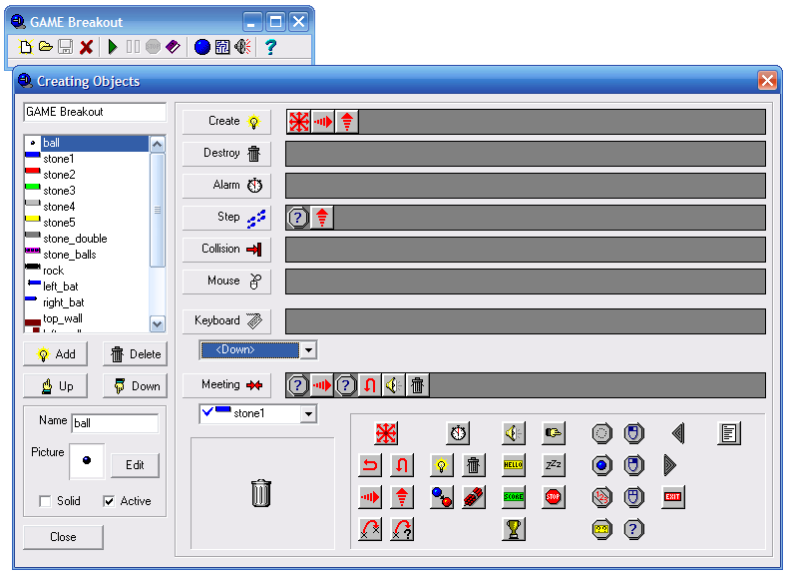


Kuva 1. GameMaker-pelimoottorilla toteutetut pelit 1. Risk of Rain [3], 2. Hyper Light Drifter [4], 3. Forager [5] ja 4. Undertale [6].

GameMaker onkin tunnettu aloittelijaystävällisyytensä lisäksi helppokäyttöisyydestään sekä nopeasta kehitysympäristöstään, ja sitä on pidetty yhtenä parhaista alustoista kaksiulotteisten pelien kehittämiseen [1; 7]. Sen heikkoutena pidetään yleisesti sen mainetta aloittelijoiden pelimoottorina, mikä ilmenee kokeneiden alustaa käyttävien ammattilaisten harvalukuisuutena työmarkkinoilla. Lisäksi GameMakerille ei ole saatavilla yhtä paljon valmiita peliresursseja kuin suosituimmille pelimoottoreille Unitylle ja Unreal Enginelle. [1.]

2.1 GameMakerin historia

GameMakerin ensimmäisen version kehitti Mark Overmars vuonna 1999. Ensimmäisillä versioilla pelikehitys oli toteutettu Drag and Drop™ -elementtejä käyttäen, mikä tarkoittaa, että pelikehitys tehdään visuaalisilla ohjelmointipalikoilla, joita sijoitetaan pelimoottorin eri toimintojen tapahtumaketjusyötteeseen antamalla niille parametrejä, jotka määrittävät tapahtuman luoteen (kuva 2). Ensimmäisistä versioista lähtien yhteen näistä elementeistä pystyi myös kirjoittamaan GameMakerin omaa ohjelmointikieltä GameMaker Languagea (GML). [8; 9.] Pelikehitys ohjelmointipalikoilla on säilynyt ominaisuutena viimeisimpään versioon asti, sillä se tarjoaa matalan tason oppimisympäristön niille pelikehittäjille, joilla ei vielä ole kokemusta ohjelmointikielistä [1].



Kuva 2. GameMakerin versio 1.1 [8].

Vuonna 2007 Overmars aloitti yhteistyön YoYo Games -ohjelmistotalon kanssa, joka otti vastuun ohjelman jatkokehityksestä [10]. Vuonna 2011 julkaistu GameMakerin versio 8.1 oli viimeinen ennen yhtiön vuonna 2012 julkaisemaa GameMaker Studiota, joka oli ominaisuuksiltaan suunniteltu helpottamaan moottorin oman ohjelmointikielen käyttöä ja toi mukanaan ensimmäisen C++-kääntäjän YoYoCompilerin, jonka avulla peleihin sai paljon lisää suoritustehoa käännösvaiheessa [9; 11]. Tämän version myötä myös pelien julkaiseminen mobiili- ja konsolialustoille oli mahdollista [9].

YoYo Games liitettiin osaksi uhkapeleihin keskittyvää Playtechiä vuonna 2015, jota seurasi vuonna 2017 julkaistu alusta alkaen uudelleen suunniteltu GameMaker Studio 2, jossa Drag and Drop™ ja GML erotettiin ohjelman sisällä erillisiksi työskentely-ympäristöiksi [10; 12; 13]. Käyttöliittymän lisäksi myös pelimoottorin tarjoamat kamera- ja tile-järjestelmä toteutettiin uudella tavalla tekemällä niistä monipuolisempia ja suorituskyvyltään tehokkaampia [13; 14]. Seuraava suuri päivitysversio 2.3 tuli vuonna 2020, ja se uudisti GML:ää lisäämällä siihen uusia, muista ohjelmointikielistä tuttuja ominaisuuksia ja muuttamalla tietorakenteiden toimintaperiaatteita [15].

Vuonna 2021 GameMakerin osti verkkoselaimestaan tunnettu Opera, joka on jatkanut pelimoottorin kehittämistä ja tuonut on siihen uusia ominaisuuksia [2; 12]. Opera ei toistaiseksi ole tehnyt pelimoottoriin yhtä suuria muutoksia, kuin GameMaker Studio 1, 2 ja 2.3 toivat tullessaan. Vuonna 2021 Opera kuitenkin muutti GameMakerin perusversion ilmaiseksi ja vuonna 2022 yksinkertaisti pelimoottorin nimen GameMakeriksi jättäen pois studio-sanan sen vanhan nimen lopusta [16; 17].

2.2 GameMaker Language -ohjelmointikieli

GameMaker Language, joka yleisesti lyhennetään GML, on GameMaker-pelimoottorissa käytetty ohjelmointikieli. GML on kehitetty mahdollisimman helposti lähestyttäväksi, ja sen kaikki ominaisuudet on suunniteltu pelikehitystä silmällä pitäen. [18.] Kieltä voisi verrata esimerkiksi Unity-pelimoottorin tarjoamaan Unity

Scripting -rajapintaan, jonka avulla pelimoottorin elementteihin pääsee käsiksi C#-kielellä. Erona on, että GML ei ole rajapinta vaan itsenäinen ohjelmointikieli, mikä useimmiten näkyy yksinkertaisempina komentosarjoina. GML:n ominaispiirteisiin lukeutuu sen vapaus tietotyyppien määrittelemisessä, eli toisin kuin monissa muissa vahvasti tyypitetyissä kielissä muuttujat ja peliobjektit eivät vaadi tyyppiesittelyä [1; 19]. Lisäksi tietorakenteet mahdollistavat erityyppisten tietojen tallentamisen samaan rakenteeseen: esimerkiksi samaan taulukkoon pystyy tallentamaan numeroita, tekstiä ja viittauksia peliobjekteihin, mikä on poikkeuksellista moniin muihin kieliin verrattuna, joissa taulukon täytyy aina sisältää vain yhdentyyppistä dataa [20].

GML on myös anteeksiantava kieli, sillä se sallii useita eri tapoja kirjoittaa koodia [1; 18]. Useissa tapauksissa, joissa muut ohjelmointikielet edellyttävät sulkeita, puolipisteitä tai sisennyksiä toimiakseen suorittaa GML-koodin ongelmitta ilman niitäkin, mutta ei myöskään rankaise niiden käytöstä. GML sisältää myös automaattisen roskienkeruutoiminnon eli pyrkii vapauttamaan muistista tiedot, joihin sovellus ei enää viittaa. Poikkeuksena roskienkeruussa ovat DS-tietorakenteet, eli data structure -tietorakenteet, joiden poistamisesta ohjelmoija on vastuussa [20].

GML:n salliman vapauden kääntöpuolena on suoritustehon heikkeneminen. GameMaker tarjoaa kuitenkin mahdollisuuden kääntää peli tehokkaalle C++-kielelle pelin kääntövaiheessa pelimoottorin omalla YoYoCompiler-työkalulla, joka pelin ominaisuuksista riippuen yleensä moninkertaistaa suoritusnopeuden [11; 17]. Ohjelmointikielen heikkouksiin lukeutuu lisäksi moniajon puuttuminen, joka tarkoittaa kaiken prosessorilla tapahtuvan laskennan suorittamista vain yhdellä sen ytimistä [21].

2.3 GameMaker-tapahtumat

GML ei käytä olio-ohjelmoinnista tyypillistä luokka-ajattelutapaa samalla tavalla kuin muut yleisesti käytetyt ohjelmointikielet, vaan GameMakerin käyttöliittymässä luodaan peliobjektit, jotka itsessään ovat verrattavissa oman luokkansa

määritelmiksi [1]. Peliobjektien sisälle voidaan käyttöliittymässä luoda eventejä eli tapahtumia, joiden sisälle kooditoteutus kirjoitetaan. Erilaisia tapahtumia on valittavana 14, ja niiden alta löytyy lukuisia alitapahtumia. Tapahtumat vastaavat esimerkiksi Unityn rajapinnassa luokan sisälle kirjoitettavia sisäänrakennettuja metodeja. Tapahtumista yleisimmin käytetyt ovat Create, Step ja Draw, joita ilman ei yhtäkään kunnollista peliä pystyisi toteuttamaan. [22.]

Create-tapahtuma vastaa Unityn metodia Start. Tapahtuman sisällä kirjoitetaan koodi, joka halutaan suorittaa, kun peliobjekti luodaan. Koodi suoritetaan vain kerran, ja tämän tapahtuman yleisin käyttötarkoitus on alustaa objektin sisäiset muuttujat tai suorittaa koodia, joka halutaan tapahtuvaksi objektin syntyhetkellä. [23.]

Step-tapahtuma puolestaan vastaa Unityn FixedUpdate-metodia. Tapahtuman sisälle kirjoitettu koodi suoritetaan pelin jokaisella askeleella, eli pelin toimiessa tyypillisellä 60 päivityksen kuvataajuudella koodi suoritetaan 60 kertaa sekunnissa. Tätä tapahtumaa käytetään yleisesti pelitilanteen muutosten tarkkailuun ja niihin reagoimiseen. [23.] Useat muut valittavissa olevat tapahtumat pystytään korvaamaan kirjoittamalla Step-tapahtuman sisälle oma toteutus tapahtumasta: esimerkiksi pelaajan antamille näppäimistösyötteille on jokaiselle olemassa oma tapahtumansa, mutta GML:n sisäänrakennetuilla funktioilla nämä syötteet voidaan tunnistaa myös Step-tapahtuman sisällä.

Draw-tapahtuman sisälle määritellään, mitä objektin halutaan piirtävän pelimaailmaan. Tälle tapahtumalle ei vastaavuutta Unityn rajapinnasta löydy, sillä Unity-pelimoottori piirtää kuvan ruudulle automaattisesti. Draw-tapahtuma voidaan halutessa myös jättää tyhjäksi, jolloin GameMaker piirtää peliobjektille asetetun kuvan automaattisesti peliobjektin koordinaatteihin. [23.]

2.4 GameMakerin tietorakenteet

GameMaker-pelimoottori tarjoaa useita dynaamisia tietorakenteita, jotka pystyvät kasvamaan ja supistumaan suoritusaikana. Älykkäiden tietorakenteiden

etuliitteenä on GML:ssa kirjaimet DS lyhenteenä englannin sanoista data structure, joka tarkoittaa suomeksi tietorakennetta. Kaikki tietorakenteet ovat lisäksi heikosti tyypitettyjä, jolloin kaikentyyppisen datan tallentaminen yhteen tietorakenteeseen on mahdollista. Tietorakenteita voidaan myös ketjuttaa moniulotteisiksi hybriditietorakenteiksi käyttämällä tietorakenteen tunnistesymbolia ennen tietorakenteen ensimmäistä argumenttia (esimerkkikoodi 1). [15; 24.] Yleisimmät tietorakenteet ovat taulukot, DS-listat, DS-ruudukot ja DS-hajautuskartta.

```
// Access a grid that has been added to a list that is part of a map:
var _a = data[? "lists"][| 0][# 0, 0];

// Access an array nested in a list from a script and modify it:
data[| 0][@ 10] = 100;

// Access a map nested in a grid nested in a list nested in an array:
data[0][| 10][# 3, 4][? "key"] = "hello world";
```

Esimerkkikoodi 1. Ketjutettuja hybriditietorakenteita [15].

Yleisimmät tietorakenteet ovat seuraavat:

- Taulukko voi pitää sisällään useita arvoja, jotka tallentuvat siihen listan kaltaisena jonona [25]. Taulukko poikkeaa muista tietorakenteista siinä, ettei sen käyttö aina vaadi taulukon @-tunnistesymbolia ennen argumentteja. Tunnistesymbolin käyttö on kuitenkin suositeltavaa, kun taulukko annetaan funktiolle parametrina ja taulukon sisältöä on tarkoitus muokata tai lukea funktion sisällä, koska silloin vain taulukon osoitin siirtyy funktion käytettäväksi. Ilman tunnistesymbolia taulukko kopioidaan funktioon, mikä kuormittaa prosessoria turhaan, jos funktion tarkoituksena ei ole tuottaa uutta taulukkoa. [24.]
- DS-lista on taulukon joustavampi versio, jonka metodit sallivat arvojen lisäämisen listan keskelle, listan satunnaisen sekoittamisen ja järjestämisen nousevaan tai laskevaan järjestykseen [26]. DS-listan tunnistesymboli on |, jonka käyttö on pakollista, kun listalle halutaan antaa indeksinumero parametrina [24].
- DS-ruudukko on pohjimmiltaan kaksiulotteinen taulukko, jonka tunnistesymboli on # [24]. DS-ruudukolla on useita metodeja, joilla voi esimerkiksi käsitellä erimuotoisten alueiden dataa, etsiä arvoja tai sekoittaa ruudukon arvot satunnaisesti [27]. DS-ruudukon ominaisuudet soveltuvat erinomaisesti ruutupohjaisten pelien datan hallintaan, jolloin esimerkiksi kartan tiedot voidaan tallentaa DS-ruudukon soluihin.

- DS-hajautuskartta on tietorakenne, joka käyttää ?-tunnistesymbolia [24]. DS-hajautuskarttaan tallennetaan avain-arvopareja, joissa avain voi olla tallennettavaa asiaa kuvaava tekstisyöte, joka on ohjelmoijan kannalta helpommin hahmotettavissa kuin pelkät indeksinumerot. Avaimen avulla arvon hakeminen ja muuttaminen on erittäin tehokasta, mutta hajautuskartan ominaisuuksiin kuuluu, ettei data yleensä ole siinä järjestyksessä, missä se on sinne tallennettu, joten arvojen etsiminen ilman avainta on hidasta ja sitä tulisi siksi välttää. [28.]

GML:n tietorakenteet vastaavat ominaispiirteiltään muiden korkeamman tason ohjelmointikielien älykkäitä tietorakenteita, mutta ne eroavat joustavuudellaan tietotyyppien määrittämisessä ja ovat tunnistesymboliensa ansiosta helpommin tunnistettavissa lähdekoodissa. Pelimoottorissa on myös muita DS-tietorakenteita, jotka ovat DS-kasa, DS-jono ja DS-prioriteettijono, mutta niillä ei ole tunnistesymboleita, koska niiden alkioihin ei voi suoraan päästä käsiksi indeksinumeroin [20].

3 Proseduraalinen pelimaailman luonti

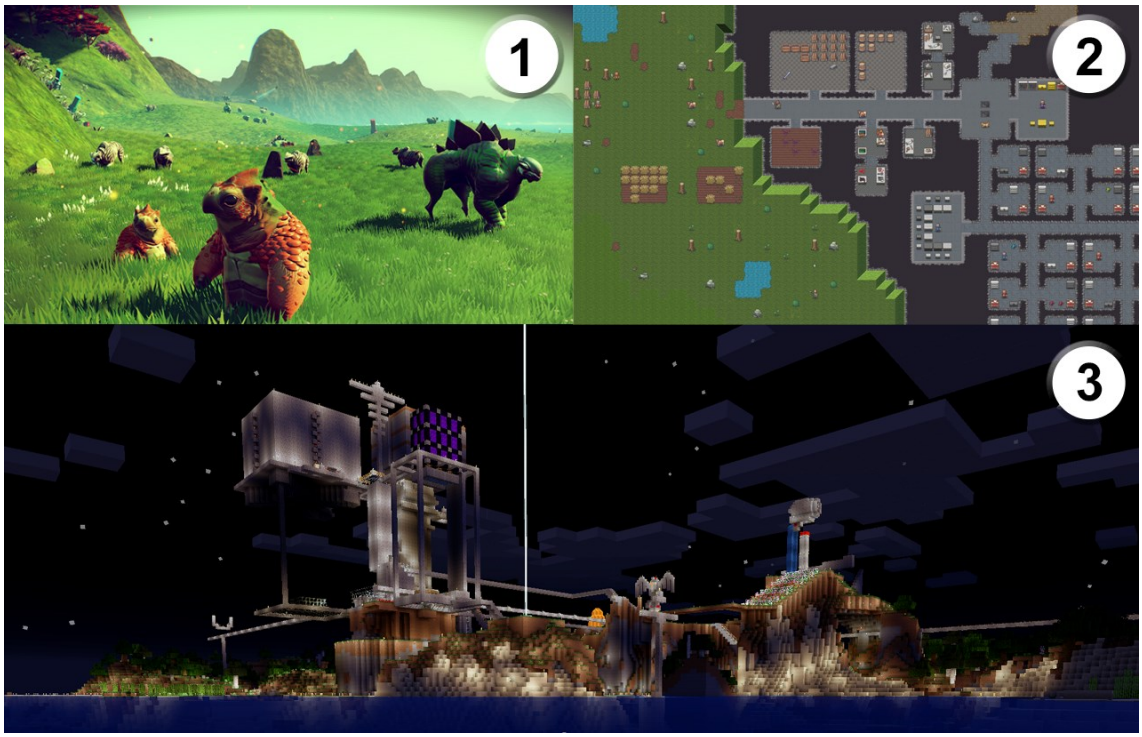
Peleissä proseduraalisella generoinnilla tarkoitetaan pelielementtejä, jotka on rakennettu satunnaisuuteen pohjautuvalla algoritmilla, kuitenkin niin, että satunnaisuus noudattaa ennalta asetettuja matemaattisia ehtoja ja sääntöjä. Proseduraalinen sisällön tuotanto mahdollistaa sen, ettei kaikkea pelimaailmassa tarvitse ennalta suunnitella, mallintaa ja toteuttaa käsityönä, vaan algoritmi kokoaa sille annetuista ominaisuuksista uusia mielekkäitä kokonaisuuksia. Proseduraalisesti peleihin voidaan tuottaa kaikenlaista sisältöä, kuten karttoja, pelikenttiä, esineitä, vihollisia ja jopa tarinoita. Lopputulos riippuu siitä, kuinka hyvin algoritmi on toteutettu ja miten satunnaisuus on saatu tasapainotettua haluttuun pelitilanteeseen. [29.] Liiallinen satunnaisuus voi aiheuttaa kaoottisuutta ja tuottaa liian helpon, vaikean, epäloogisen tai jopa mahdottoman pelitilanteen [30]. Liian vähäinen satunnaisuus puolestaan generoi liikaa toisiaan muistuttavia asioita, mikä saa pelikokemuksen tuntumaan itseään toistavalta.

Oikein tehtynä proseduraalisella sisällöntuotannolla voidaan tuottaa näennäisesti rajaton määrä mielekästä sisältöä peleihin niin, että jokainen pelikerta on

uniikki aikaisempiin nähden ja näin kasvatetaan pelin uudelleenpelattavuusarvoa. Proseduraalinen sisältö ei kuitenkaan aina sovi kaikkiin peleihin: esimerkiksi kun pelin halutaan kertovan jonkin tietyn tarinan, joka vaatii pelaajan etenemistä ennalta suunnitellulla tavalla, ei proseduraalisuutta voida käyttää koko pelimaailman luomiseen. Käsityönä tehdyt pelielementit toimivat myös tarkoituksessaan suuremmalla todennäköisyydellä kuin satunnaisuudella luotu sisältö, joka pelaajan onnesta ja pelikerrasta riippuen tuottaa helpompia tai haastavampia versioita kyseisen pelivaiheen elementeistä. [30.]

Proseduraalinen generointi on hyvä työkalu, kun halutaan tuottaa sellaisia pelejä, joiden tasoja ja tilanteita pelaaja ei voi opetella ulkoa, vaan joutuu aina reagoimaan uusiin tilanteisiin erilaisilla lähestymistavoilla. Pelikehityksen näkökulmasta proseduraaliset pelielementit voivat myös vähentää työmäärää, kun kaikkea sisältöä ei tarvitse suunnitella ja tehdä etukäteen, mutta satunnaisuuden toimivuutta voidaan joutua testaamaan enemmän, jotta saadaan varmuus sisällön toimivuudesta ja vaikeustason sopivuudesta. [29; 30.]

Erityisen hyvin proseduraalinen maailmanluonti soveltuu niin kutsuttuihin hiekkalaatikkopeleihin, koska niiden pelimekaniikat keskittyvät usein seikkailemiseen, selviytymiseen, rakentamiseen ja uusien asioiden tutkimiseen (kuva 3). Hiekkalaatikkopeleiksi kutsutaan videopelejä, joissa pelaajalle annetaan vapaus pelata peliä haluamallaan tavalla ja jätetään paljon tilaa luovuudelle rajoittamatta liikaa pelaajan toimintaa. Tyypillisesti hiekkalaatikkopelit ovat avoimen maailman pelejä, joissa pelaaja voi tutkia ympäristöä vapaasti. Lisäksi ne sisältävät usein muokattavaa maastoa sekä mahdollisuuden tehdä omia rakennelmia. Kaikki hiekkalaatikkopelit eivät kuitenkaan sisällä näitä ominaisuuksia, vaan keskiössä on pelin antama vapaus vaikuttaa ympäröivään maailmaan ja olla vapaasti vuorovaikutuksessa sen tarjoaman sisällön kanssa. Hiekkalaatikkopeleissä ei välttämättä ole pelintekijöiden asettamia päämääriä, vaan pelaaja asettaa itse omat tavoitteensa. Tyypillisesti hiekkalaatikkopelit ovat seikkailu-, kaupunginrakennus- tai simulaatiopelejä. [31; 32.]



Kuva 3. Proseduraalista sisältöä hyödyntävät hiekkalaatikkopelit 1. No Man's Sky [29], 2. Dwarf Fortress [33] ja 3. Minecraft.

Proseduraalisesta sisällöstään tunnettuja hiekkalaatikkopelejä ovat

- No Man's Sky, jossa pelaaja voi seikkailla universumissa, jonka planeetat ja elämänmuodot on tuotettu proseduraalisesti [34]
- Minecraft, jossa pelaaja voi seikkailla, tuhota ja rakentaa maailmassa, joka on proseduraalisesti rakennettu eri materiaaleja edustavista kuutioista [35]
- Dwarf Fortress, jossa pelaajan tehtävä on selvitä proseduraalisesti rakennetussa fantasiamaailmassa. Maailmanluonnin lisäksi proseduraalisuus ulottuu kaikkiin pelihahmoihin ja niiden käyttäytymismalleihin, minkä pohjalta peli pystyy jopa generoimaan uskottavan ja uniikin historian jokaiselle pelimaailmalle [36].

Proseduraalisia elementtejä sisältävän pelin suunnittelussa tulee ottaa huomioon, että suoritusaikana luotu sisältö voi viedä paljon resursseja laskentatehosta. Varsinkin isompia kokonaisuuksia, kuten pelitasoja tai karttoja, generoidessa algoritmin suoritus tulisi ajoittaa niin, ettei se aiheuta pelin toiminnan hidastumista kriittisillä hetkillä. Luotu sisältö voi vaatia myös huomattavan määrän väliaikaismuistia suoritusaikana [29]. Proseduraalista generointia käyttävän

pelin pelitiedosto vie kuitenkin yleensä vähemmän tallennustilaa. Tallennustila ei nykyaikaisilla laitteilla ole useinkaan ongelma, mutta esimerkiksi mobiilialustoilla on hyvä pyrkiä pitämään pelitiedostot mahdollisimman pieninä, jotta niiden lataus sovelluskaupasta olisi nopeaa.

3.1 Perlin noise -kohina-algoritmi

Kohina-algoritmi on matemaattinen kaava, jota käytetään pelikehityksessä apuna monien realismia tavoittelevien elementtien ja efektien luomiseen, kuten maaston, tulen, veden tai savun mallintamiseen [37]. Kohina-algoritmeilla voidaan tuottaa yksi- tai moniulotteista satunnaisuuteen perustuvaa kohinaa, jossa on enemmän yhtenäisiä rakenteita verrattuna täysin satunnaisia arvoja sisältävään valkoiseen kohinaan. Algoritmeilla tuotetut numeraaliset arvot muutetaan käyttötarkoituksen mukaan tekstuuriksi, korkeuskartaksi tai peliobjektien koordinaateiksi tai niiden pohjalta voidaan valita esimerkiksi maaston pintojen tekstuurit.

Perlin noise on yksi suosituimmista kohinaa tuottavista algoritmeista, ja se on nimetty kehittäjänsä Ken Perlinin mukaan. Perlinin tavoite oli luoda luonnollisemman näköistä tietokoneella tuotettua grafiikkaa elokuvateollisuuden käyttöön työskennellessään Walt Disney -viihdeteollisuusyhtiölle, ja hän oli suurelta osalta vastuussa vuoden 1982 TRON-scifi-elokuvan visuaalisista efekteistä. Vuonna 1996 hänelle myönnettiin teknisten saavutusten Oscar-palkinto Perlin noise -tekniikan kehittämisestä. [37.]

Perlin noisella toteutetun kohinan yhtenäiset rakenteet tuottavat luonnollisen lopputuloksen, joka saavutetaan algoritmissa luomalla riippuvuussuhteita interpoloimalla satunnaisia arvoja kahdesta eri pisteestä. Interpolointi takaa sulavan siirtymän pisteiden välille ja tuottaa terävien kulmien sijaan pehmeitä luonnollisia käyriä. Elävämmän ja monipuolisemman lopputuloksen saavuttamiseksi kohinaa iteroidaan useita kertoja. Iteraatioita, eli sitä, kuinka monta kertaa silmukka suoritetaan, kutsutaan oktaaveiksi. Jokaisella oktaavilla muokataan kohinan taajuutta, joka vaikuttaa yhtenäisten alueiden laajuuteen, ja amplitudia eli

värähtelylaajuutta, joka vaikuttaa ääreisarvoihin. Jokaisen iteraation tulos lisätään funktion paluuarvoon, jolloin erilaiset kohinan versiot sulautuvat yhdeksi kokonaisuudeksi. [38; 39.]

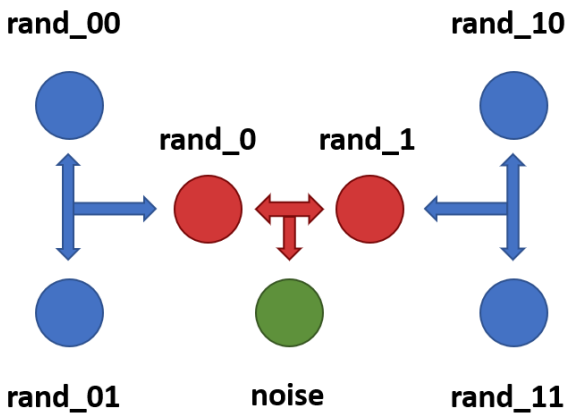
Insinöörityössä kartan pohjan toteuttamista varten ohjelmoitiin Perlin noise -funktio, joka tuottaa parametreina annettuja x- ja y-koordinaatteja varten kohinaa edustavan lukuarvon halutulla asteikolla (esimerkkikoodi 2). Asteikon suuruus annetaan funktiolle range-parametrilla: esimerkiksi asteikon koon ollessa 100 funktio palauttaa arvoja 0:n ja 100:n väliltä. Funktiolle annetaan myös seed eli siemennumero, jota käytetään yhdessä x- ja y-koordinaattien kanssa pelimoottorin satunnaislukugeneraattorin siemennumeron asettamiseen. Käyttämällä pisteiden satunnaisluvun muodostukseen x- ja y-koordinaattia siemennumeron kanssa pystytään varmistamaan, että satunnaisluku on aina sama kyseisten koordinaattien pisteessä, kun Perlin noise -funktion sisällä kutsutaan irandom_range-funktiota. Kyseinen funktio arpoo kokonaisluvun parametreina asetetusta numeroavaruudesta.

```
function scr_perlin_noise(_xx, _yy, _range, _seed, _chunk_size)
    {
    var noise = 0;
    var range = _range div 2;
    var chunk_size = _chunk_size;
    var index_x, index_y;
    var prog_x, prog_y;
    var rand_0, rand_1;
    var rand_00, rand_01, rand_10, rand_11;

    while (chunk_size > 0)
        {
        index_x = _xx div chunk_size;
        index_y = _yy div chunk_size;

        prog_x = (_xx % chunk_size) / chunk_size;
        prog_y = (_yy % chunk_size) / chunk_size;

        random_set_seed(index_x + index_y * _seed);
        rand_00 = irandom_range(0, range);
        random_set_seed(index_x + (index_y + 1) * _seed);
        rand_01 = irandom_range(0, range);
        random_set_seed(index_x + 1 + index_y * _seed);
```

```
        rand_10 = irandom_range(0, range);
        random_set_seed(index_x + 1 + (index_y + 1) * _seed);
        rand_11 = irandom_range(0, range);

        rand_0 = lerp(rand_00, rand_01, prog_y);
        rand_1 = lerp(rand_10, rand_11, prog_y);

        noise += lerp(rand_0, rand_1, prog_x);

        chunk_size = chunk_size div 2;
        range = max(1, range div 2);
        }
    return noise;
    }
```

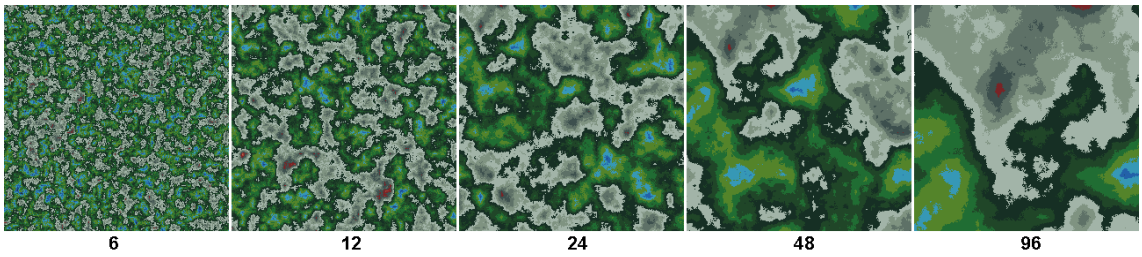
Esimerkkikoodi 2. Perlin noise -funktio, jolla voidaan tuottaa luonnollisen näköistä kohinaa.

Jokaista palautettavaa arvoa kohden Perlin noise -funktio interpoloi neljän pisteen arvot kaksiulotteisessa koordinaatistossa. Pisteet valitaan niin, että ne sijaitsevat käsiteltävän pisteen lisäksi sen oikealla, alaoikealla ja alapuolella. Tämä onnistuu kasvattamalla koodinaattien arvoja yhdellä random_set_seed-funktion parametreissa riippuen siitä, missä piste sijaitsee kohdepisteeseen nähden koordinaatistossa (kuva 4). [40.] Interpolointifunktio lerp, kokonaislukuja arpova irandom_range ja pelimoottorin siemenarvon asettava random_set_seed ovat GameMakerin omia funktioita, joten niiden toteutusta ei tarvitse itse kirjoittaa.



Kuva 4. Muuttujien välinen kaksivaiheinen interpolointi noise-arvon tuottamiseksi.

Parametria chunk_size käytetään määrittämään algoritmin oktaavit ja taajuus niin, että mitä suurempaa arvoa käytetään, sitä suurempi on valittujen pisteiden väli ja sitä yhtenäisempiä alueita saadaan tuotettua. Kuvasta 5 näkee myös, että suurempia arvoja käytettäessä kartta on muodoiltaan sama, mutta yksityiskohdiltaan tarkempi ja suurennettu versio pienemmillä arvoilla tuotetuista kartoista. Näin voidaan säätää alueiden koko sopivaksi haluttuun peliympäristöön niin, että ne ovat suhteessa tarpeeksi tilavia pelihamoja varten. Suurempi chunk_size-arvo lisää iteraatioiden määrää ja on siksi raskaampi suorittaa.

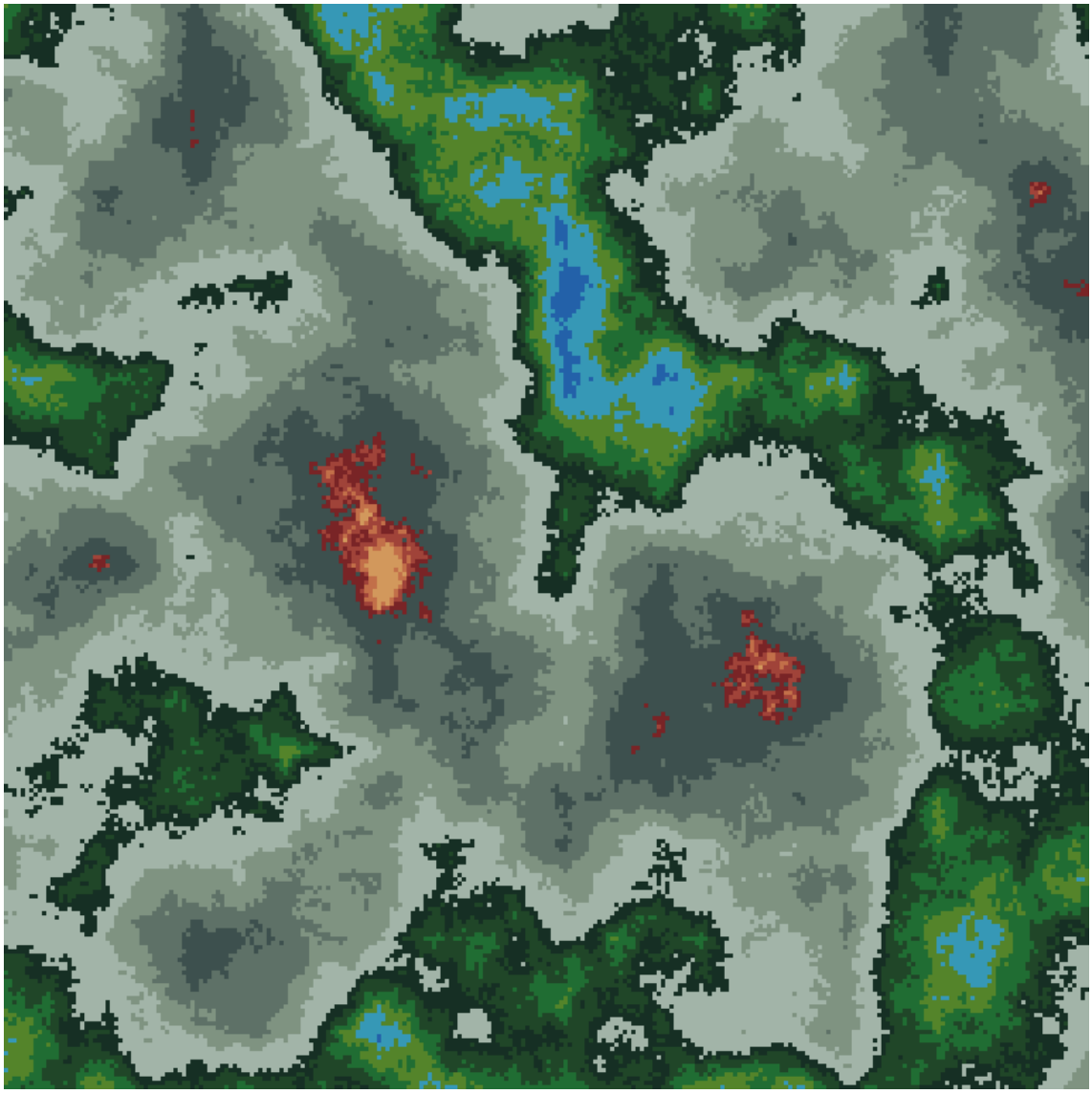


Kuva 5. Eri chunk_size-arvoin generoidut viisi karttaa.

Toteutetussa karttojen esikatseluun tarkoitetussa apuohjelmassa generoitiin sivuiltaan 256 solun kokoisia maastokarttoja erikokoisilla chunk_size-arvoilla. Funktion suoritusaika kasvoi kaksinkertaiseksi, kun chunk_size-arvoa kasvatettiin 6:sta 96:een. Ero kasvoi kolminkertaiseksi käytettäessä YoYoCompileria, mutta sen ansiosta funktio suoritti tehtävän 2–3 kertaa nopeammin.

Pelimaailmaa muodostaessa Perlin noise -funktio suoritetaan jokaiselle karttapisteelle. Ennen arvojen tallentamista DS-ruudukkoon ne skaalataan funktion paluuarvosta käyttötarkoituksen mukaiselle asteikolle. Apuohjelmassa Perlin noise tuotti arvoja 0:n ja 100:n väliltä, ja ne muutettiin vastaamaan materiaaleja edustavia numeroita esimerkiksi niin, että luvut, jotka olivat alle 25 edustivat vettä, luvut väliltä 25 ja 50 nurmikkoa, luvut väliltä 50 ja 75 kiveä ja siitä suuremmat arvot asetettiin laavaksi. Asteikkoa tuli säätää haluttujen materiaalien määrän ja tasapainon mukaan. Suurentamalla yhden materiaalin väliä suhteessa toisiin pystyttiin tuottamaan karttoja, joissa kyseisen materiaalin osuus kasvoi.

Näiden uusien arvojen pohjalta pystyttiin piirtämään tekstuuri, joka näytti realistiselta maastokartalta (kuva 6).



Kuva 6. Perlin noise -funktiolla generoitu neljän materiaalin maastokartta.

Visualisoinnin jälkeen oli havaittavissa, että luotu kartta saattoi olla liian rosoinen käytettäväksi sellaisenaan pelimaailmana, sillä eri materiaalien reunoille jäävät yksittäiset pisteet ja pienet ilmataskut kiviaineksessa näyttivät epäorgaanisilta ja saattaisivat aiheuttaa ongelmia pelihamojen navigaatiolle. Mikäli pelimaailma olisi tarkoitus kansoittaa peliobjekteilla esimerkiksi siten, että kartasta haetaan satunnainen nurmikkoa edustava piste, johon pelaaja halutaan pelin

alussa sijoittaa, voidaan päätyä tilanteeseen, jossa pelihahmo seisoo yhden solun kokoisella maastolaatalla keskellä järveä. Tämä voi olla pelaajan kannalta mahdoton tilanne, jos pelihahmo on kykenemätön etenemään vettä pitkin. Pienien maastossa olevien yksityiskohtien silottaminen ei kokonaan poista ongelmaa, mutta monissa tapauksissa estää epätoivottujen tilanteiden muodostumisen.

3.2 Kartan jälkikäsittely

Rosoisuuksien poistamiseksi otettiin vaikutteita soluautomaatin toimintaperiaatteista. Tietojenkäsittelyssä soluautomaateiksi kutsutaan soluista koostuvaa rakennetta, jossa solut vaihtavat tilaansa noudattaen ennalta määrättyjä sääntöjä. Tilanvaihdokset pohjautuvat yleensä käsittelyssä olevan ja sen viereisten solujen tilaan, mutta joissain malleissa voidaan huomioida myös solun aiemmat tilat tai sille annettu perimätieto.

Soluautomaateista tunnetuin on John Conwayn vuonna 1970 kehittämä Game of Life eli ”elämän peli”, jossa kaksiulotteisessa ruudukossa solut voivat olla joko mustia elossa olevia tai valkoisia kuolleita soluja. Soluilla on kaksi sääntöä, joista ensimmäinen herättää solun eloon, jos sen naapureista tasan kolme on myös elossa. Toisen säännön mukaan solu pysyy elossa, jos sillä on kaksi tai kolme naapuria ja muissa tapauksissa solu kuolee. Päivittämällä sääntöjen mukaisesti ruudukkoa, jossa on lähtöasetelmana jonkinlaisia solurykelmiä, syntyy erilaisia solukokonaisuuksia, joilla voi olla elämään liitettäviä ominaisuuksia, kuten liikkuminen tai monistuminen, josta pelin nimikin juontuu. [41.]

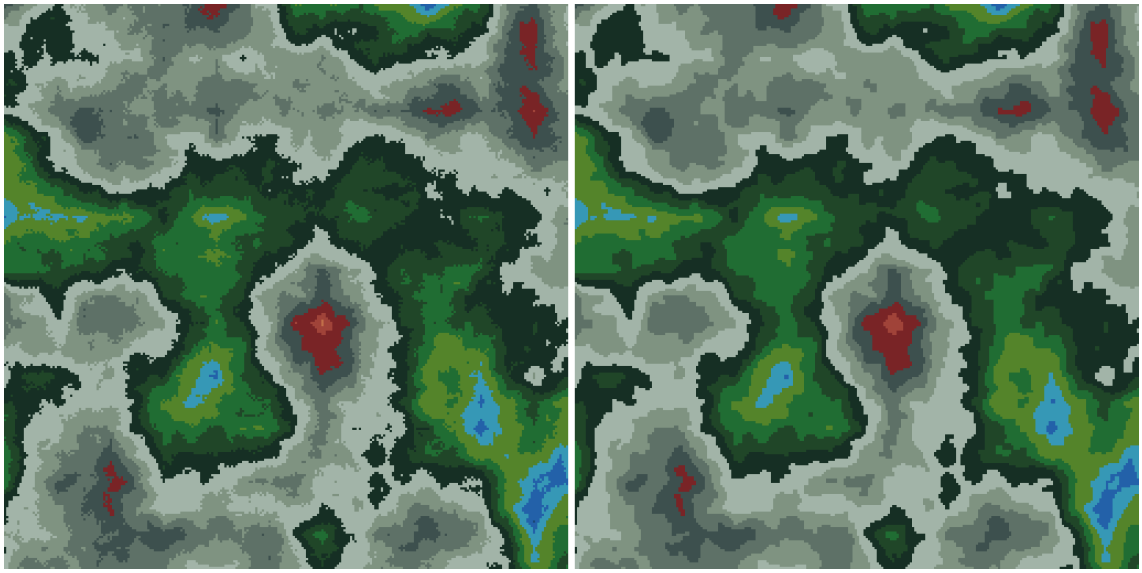
Toteutettu rosoisuuksien suodatusfunktio käy parametrina saadun DS-ruudukon kaikki solut läpi ja kerää nelialkioiseen taulukkoon tiedot naapurisolujen arvoista (esimerkkikoodi 3). Arvot käydään läpi silmukassa, joka laskee, kuinka moni naapurisoluista sisältää samaa materiaalia edustavan arvon kuin käsiteltävänä oleva keskimmäinen solu, ja jos sama arvo esiintyy naapurisoluissa kahdesti, pidetään solu ennallaan. Muussa tapauksessa vaihdetaan keskimmäisen solun arvoksi se materiaali, jota tulee listalla ensimmäisenä vastaan kahdesti.

```
function scr_filter_grid(grid)
    {
    var array_materials_cross;

    for (var yy = 1; yy < ds_grid_height(grid)-1; ++yy)
        {
        for (var xx = 1; xx < ds_grid_width(grid)-1; ++xx)
            {
            var new_material_1 = -1;
            var new_material_2 = -1;
            var collision_counter = 0;
            var current_material = grid[# xx, yy];

            array_materials_cross[0] = floor(grid[# xx + 1, yy + 0]);
            array_materials_cross[1] = floor(grid[# xx - 1, yy + 0]);
            array_materials_cross[2] = floor(grid[# xx + 0, yy - 1]);
            array_materials_cross[3] = floor(grid[# xx + 0, yy + 1]);

            for (var i = 0; i < 4; ++i)
                {
                if (array_materials_cross[i] == current_material)
                    {
                    ++collision_counter;
                    }
                else
                    {
                     if (new_material_1 == -1)
                        {
                        new_material_1 = array_materials_cross[i];
                        }
                    else if (new_material_1 == array_materials_cross[i]
                         && collision_counter > 1)
                        {
                        new_material_2 = new_material_1;
                        break;
                        }
                    else if (new_material_2 == array_materials_cross[i])
                        {
                        new_material_1 = new_material_2;
                        }
                    else
                        {
                        new_material_2 = array_materials_cross[i];
                        }
                    }
                }

            if (collision_counter > 1)
```

```
                {
                continue;
                }

            grid[# xx, yy] = new_material_1;
            }
        }
    }
```

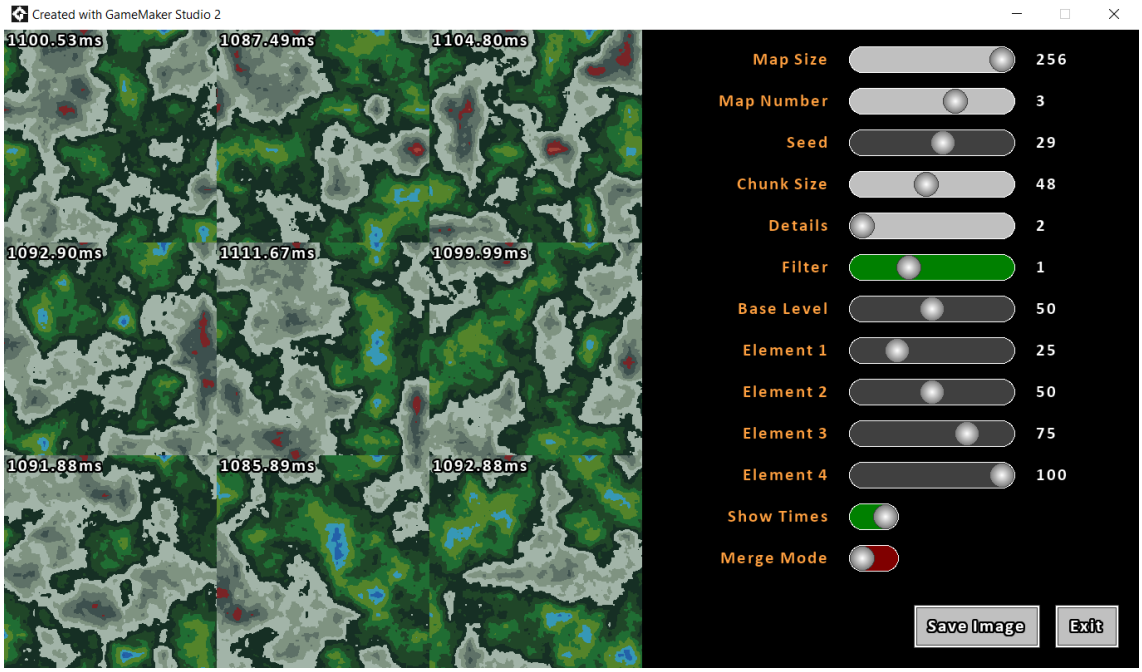
Esimerkkikoodi 3. Kohinan rosoisuuksia vähentävä suodatusfunktio.

Täydellisen tuloksen saamiseksi ruudukko joudutaan käymään läpi useaan kertaan, mutta jo yhdellä iteraatiolla lähes kaikki rosoisuudet saadaan karsittua pois (kuva 7). Jokaisen iteraation lisätessä suoritusaikaa ja vaikutuksen pienentyessä dramaattisesti rosoisuuksien vähetessä oli selvää, ettei karttaa kannattanut suodattaa funktion avulla kuin kerrasta kahteen.



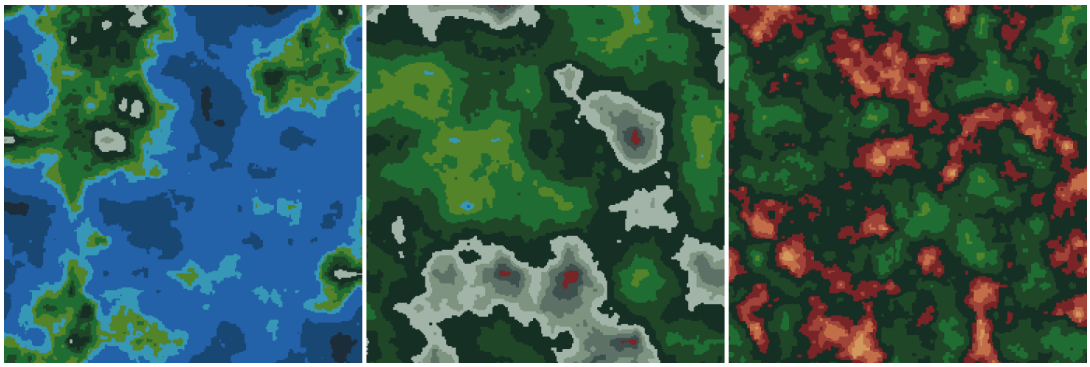
Kuva 7. Maastokartta ennen, kuin rosoisuudet oli poistettu suodatusfunktiolla, ja sen jälkeen.

Apuohjelmassa pystytään esikatselemaan erilaisia karttoja ja parametrimuutosten vaikutusta niihin (kuva 8). Luodut kartat vaikuttivat mielenkiintoisilta pelata ja sopivilta hiekkalaatikkotyyppisen pelin pohjaksi, koska niihin muodostui tarpeeksi vaihtelevia rakenteita, jotta pelikokemus olisi jokaisessa kartassa erilainen.



Kuva 8. Apuohjelma, jolla maastokarttoja voi luoda eri parametrein ja esikatsella useita eri lopputuloksia yhtäaikaisesti.

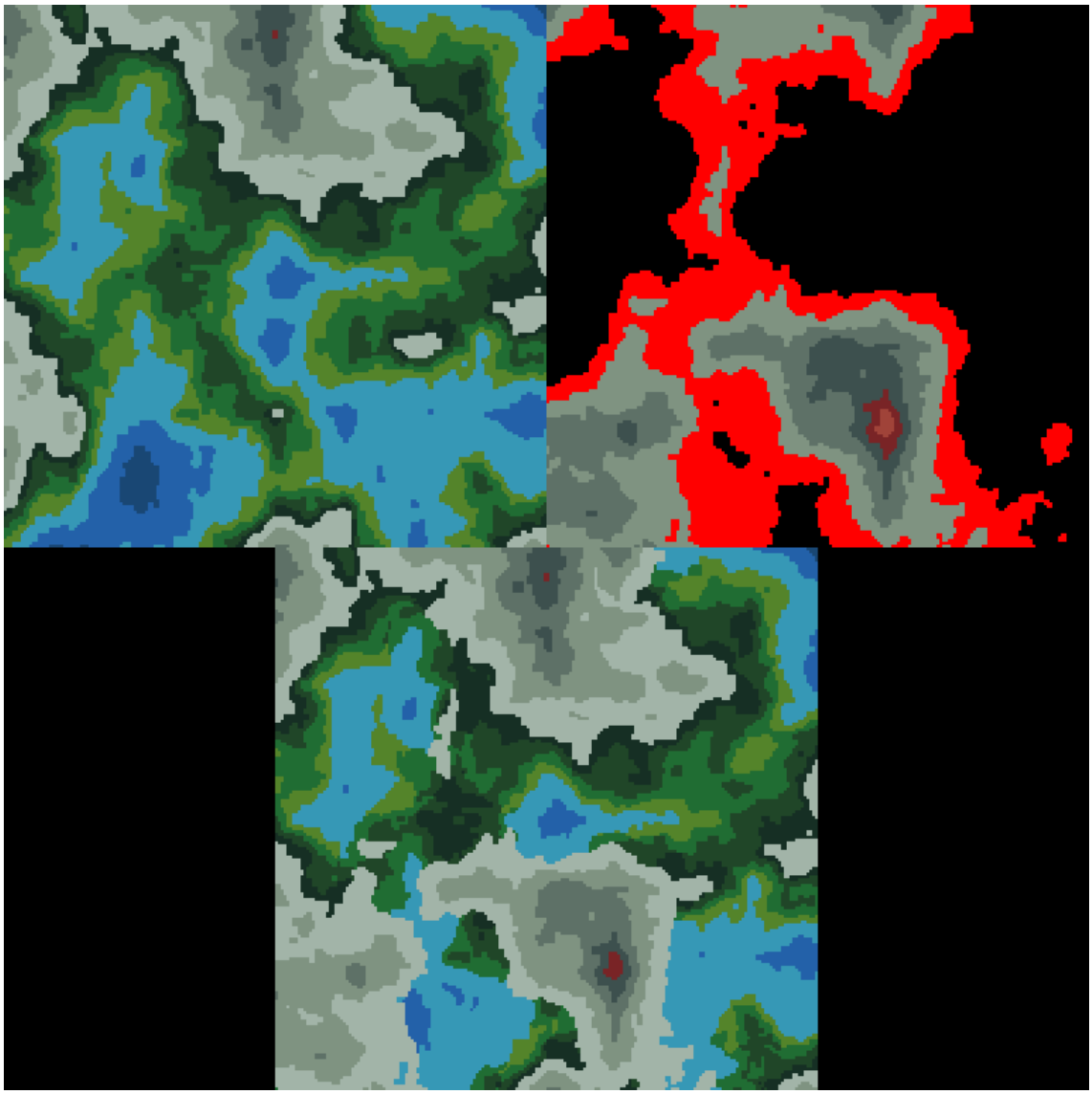
Apuohjelman parametreja muuttamalla pystyi helposti hallitsemaan halutun lopputuloksen yleisilmettä ja materiaalitasapainoja (kuva 9). Vaikka apuohjelmassa oli käytössä vain neljä eri materiaalia, oikeaan peliprojektiin implementoituna Perlin noise -funktio olisi helppo saada tuottamaan erilaisia biomeja eli kasvillisuusvyöhykkeitä, kuten hiekka-aavikkoa, lumista tundraa tai elotonta laavamaailmaa. Muutoksen aikaansaamiseksi tarvitsi valita vain, mitä materiaalia Perlin noisen tuottama paluuarvo edustaa ja kuinka suurella painotuksella kyseistä materiaalia maailmaan tuotetaan.



Kuva 9. Kolme erilaista karttaesimerkkiä, joissa materiaalitasapainoa oli muuteltu.

Vaikka Perlin noisella tuotetut kartat olivat kokonaisuuksina monipuolisia, oli kohinan sulavasta rakenteesta johtuen havaittavissa, että materiaalivyöhykkeiden sijainnit suhteessa toisiinsa noudattivat kaavaa, jossa maatyypit pystyivät saamaan naapurikseen vain materiaaliasteikon viereisiä maatyyppejä. Kartoissa, jossa oli enemmän kuin kaksi materiaalia, eivät kaikki maatyypit voineet olla toistensa naapureita. Käytännössä tämä tarkoitti sitä, ettei vettä koskaan muodostunut kiven viereen tai laavaa ei voinut muodostua muualla kuin kiven sisässä. Pelimekaniikan kannalta tämä voi olla haitallista, sillä näin kartasta voi tulla helposti ennustettava. Pelaaja pystyisi esimerkiksi päättelemään, että kävelemällä poispäin kivestä hän todennäköisesti löytäisi vettä tai seuraamalla vesirajaa hän ei koskaan törmäisi kiviseinään.

Karttojen homogeenisen luonteen rikkomiseksi kehitettiin ratkaisu, jossa kaksi eri parametrein luotua karttaa yhdistettiin niin, että toisesta kartasta otettiin vain yhden materiaalin arvot ja niillä korvattiin osa ensimmäisen kartan arvoista. Apuohjelmaan tehtiin mekaniikka, joka lisäsi toisen kartan kiveä edustavat arvot alkuperäiseen karttaan niin, että se korvasi sillä kaikki muut kiveä pienemmät arvot (kuva 10).



Kuva 10. Apuohjelman havainnollistama karttojen yhdistäminen kolmen eri kuvan avulla.

Kooditoteutus karttojen yhdistämiselle oli kaksinkertainen silmukka, joka vertaili molempien karttojen solujen arvoja ja päätti sen perusteella, korvataanko alkuperäinen materiaali (esimerkkikoodi 4). Muokkaamalla ehtolauseketta menetelmä soveltuisi myös käyttötarkoitukseen, jossa maailmaan haluttaisiin lisätä useita erilaisia kasvillisuusvyöhykkeitä tai esimerkiksi vuorten saartama laakso, jonka biomi poikkeaisi muusta maailmasta.

```
function scr_add_material(grid_1, grid_2, material)
    {
    for(var yy=0; yy<ds_grid_height(grid_1); ++yy)
        {
        for(var xx=0; xx<ds_grid_height(grid_1); ++xx)
            {
            if (grid_1[# xx, yy] < material
            && grid_2[# xx, yy] == material)
                {
                grid_1[# xx, yy] = material;
                }
            }
        }
    }
```

Esimerkkikoodi 4. Funktio, joka kopioi yhden materiaalin arvot toisena parametrina annetusta DS-ruudukosta lisäten ne ensimmäisenä parametrina annettuun DS-ruudukkoon.

Sopivien parametrien asettamisen ja kartan rosoisuuksien suodattamisen sekä rakenteen rikkomisen jälkeen lopputulos näytti siltä, että ne soveltuisivat hyvin pelimaailmojen pohjiksi ja tarjoaisivat pelaajalle tarpeesi avointa tila liikkua. Karttoihin jäi myös usein kiviseinän erottamia taskuja, jonne pelihahmolla ei olisi suoraa pääsyä ilman, että hän joutuisi kaivamaan tunnelia kallion läpi.

4 Pelimaailman visualisointi maastolaatoilla

Proseduraalisesti luodun pelimaailman visualisointi näkyviksi maastoelementeiksi toteutetaan hakemalla aikaisemmin prosessoitu data tietorakenteesta ja muuttamalla indeksinumerot koodinaateiksi ja arvot materiaaleiksi. Helpoin tapa olisi luoda maasto peliobjekteilla, jotka voivat olla esimerkiksi haluttua materiaalia edustavia neliöitä. Peliobjektit ovat käytännöllisiä siksi, että niillä on GameMakerissa automaattisesti luodut osumalaatikot ja niiden kuvadata on helposti muokattavissa. Peliobjektien tapahtumiin voi myös kirjoittaa koodia, minkä ansiosta kaiken vuorovaikutuksen toteutus muiden peliobjektien kanssa on helppo toteuttaa. Peliobjektit ovat kuitenkin raskaita pelin suorituskyvyn kannalta, sillä GameMakerissa jokaisella peliobjektilla on sisäänrakennettuja metodeja eli tapahtumia, joita pelimoottori käy läpi jokaisella ruudunpäivityksellä. Vaikka pelimoottori suoriutuu yleensä ongelmitta satojen peliobjektien olemassaolosta, käy

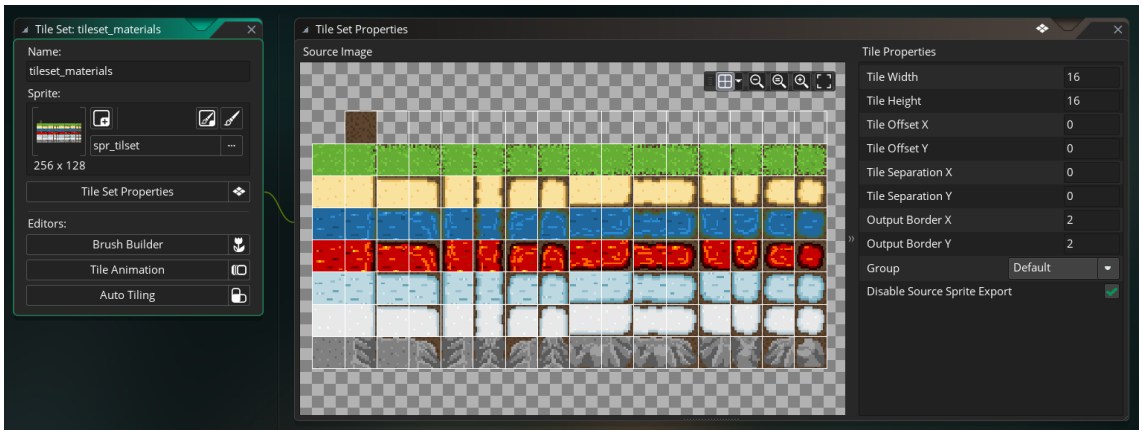
niistä rakennetun pelimaailman ylläpito nopeasti erittäin raskaaksi. Jos pelimaailman on tarkoitus koostua maastolaatoista ja ne kaikki olisi toteutettu peliobjektein, olisi 100 ruutua leveässä ja korkeassa maailmassa jo 10 000 objektia, minkä lisäksi tarvitaan muun sisällön tuomat objektit, kuten pelihahmot, viholliset, rakennukset, tavarat ja monet muut asiat, jotka täytyy toteuttaa peliobjektein.

4.1 Tile-järjestelmä

Suurten maastolaatoista koostuvien pelimaailmojen toteutukseen ei pelkkiä peliobjekteja voi suorituskyvyn takia käyttää. GameMakerissa on sisäänrakennettu ominaisuus luoda suorituskyvyltään tehokkaita maastolaattoja, joita kutsutaan englannin kielen tile-sanan mukaisesti tile-elementeiksi. Tile-elementit ovat graafisia pelielementtejä, ja ne soveltuvat erinomaisesti taustojen ja tasojen piirtämiseen, sillä ne ovat erittäin nopeita piirtää näytölle ja vievät vain vähän prosessorin laskentatehoa. Tile-elementeille ei kuitenkaan voi liittää kuvadatan lisäksi muita ominaisuuksia, joten ne eivät suoraan voi olla vuorovaikutuksessa muiden peliobjektien kanssa. [42.]

GameMakerin käyttöliittymässä maastografiikka tuodaan pelimoottoriin spriteksi kutsuttuna kuvaresurssina samalla tavalla kuin muutkin grafiikkaelementit, mutta sen lisäksi niistä täytyy luoda tile-sarjaresurssi, joka jakaa kuvan laatoiksi (kuva 11). Valmis tile-sarja liitetään tile-elementtien säilytykseen tarkoitetulle tile-kartalle, joka on pelimoottoriin sisäänrakennettu ruudukonkaltainen tietorakenne. [42.] Tälle tietorakenteelle löytyy GameMakerista useita valmiita funktioita, joilla sitä voidaan muokata. Tile-kartalle asetetaan myös kerros, joka määrittää, missä järjestyksessä se piirretään ruudulle suhteessa muihin pelin kerroksiin. Myös peliobjektit sijaitsevat omissa kerroksissaan, joten maastoa sisältävät tile-kerrokset tulee sijoittaa niiden alle. Suurempi kerrosnumero tarkoittaa, että sen kerroksen elementit piirretään pienemmällä arvolla varustettujen kerrosten alle. Kerrosnumeroa voidaan siis ajatella myös kyseisen kerroksen syvyytenä. Tile-kerroksia voi lisäksi olla useita eri syvyyksissä, jolloin useita eri tile-elementtejä voidaan kasata päällekkäin, vaikka yhteen tile-kartan soluun pystyykin

tallentamaan vain yhden tile-elementin [42]. GameMaker tukee myös osittain tai kokonaan läpinäkyviä tile-elementtejä, jolloin alemman kerroksen tile-elementit voivat näkyä ylemmän tile-kerroksen alta.



Kuva 11. Tile-sarjaresurssi GameMakerin käyttöliittymässä.

Tile-sarjaresurssi jakaa kuvan automaattisesti samankokoisiin osiin. Koko on vapaasti päätettävissä, mutta osittain historiallisista syistä tile-elementin kooksi valitaan yleensä jokin kahden potensseista, esimerkiksi 16, 32 tai 64, jotka ovat muistinhallinnan kannalta optimaalisia lukuja [43]. Nykyään muistin määrä muodostuu harvoin pullonkaulaksi, mutta valtavia maailmoja sisältävissä peleissä, kuten Minecraftissa, on huomattavissa, että kahden potenssin sääntöä on noudatettu varmasti juuri optimointisyistä. Lisäksi on tärkeää huomioida, että GameMaker varaa aina tile-sarjan ensimmäisen ruudun tyhjälle tile-elementille [42]. Tile-sarjan järjestyksellä ei tavallisesti ole merkitystä, ja se saa sisältää myös tyhjiä ruutuja. Kuitenkin, jos tile-elementtejä halutaan asettaa suoritusaikana algoritmillisesti, kannattaa tile-sarjan tile-elementtien järjestys suunnitella tarkasti, jolloin tile-elementtien asettelulogiikka on helpompi toteuttaa.

Tile-kerroksen ja -kartan luominen onnistuu GameMakerin käyttöliittymän huone-editorissa, mutta käytännön syistä nämä kannattaa useimmiten luoda koodilla, jolloin luotu tile-kartta saadaan otettua talteen muuttujaan. Insinöörityötä varten tehdyssä hiekkalaatikkopeliprojektissa luotiin managerina toimiva peliobjekti, joka vastaa maailman luonnista ja sen ylläpidosta. Managerin

create-tapahtumassa luotiin sivuiltaan 128 ruudun kokoinen tile-kartta, ja se tallennettiin tilemap_terrain-nimiseen muuttujaan, minkä jälkeen alustettiin DS-ruudukko, johon Perlin noisella tuotetut arvot tallennettiin ja lopuksi siistittiin rosoisuuksia poistavalla suodatusfunktiolla (esimerkkikoodi 5).

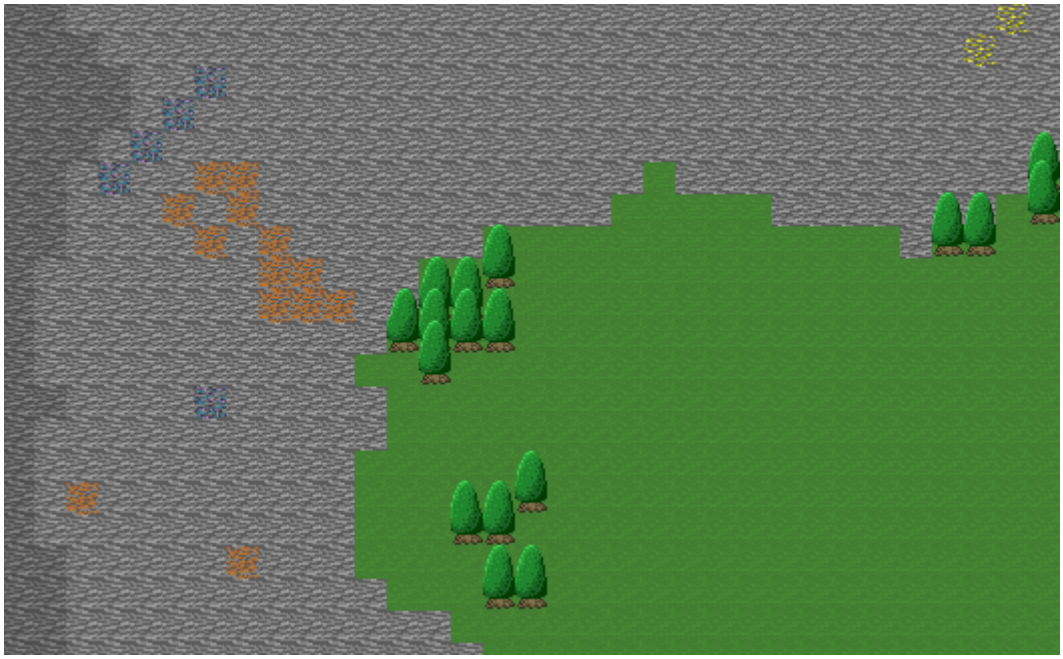
```
room_size = 128;
var layer = 10;

lay_id = layer_create(layer);
tilemap_terrain = layer_tilemap_create(lay_id, 0, 0, tile-set_materials,
room_size, room_size);

globalvar grid;
grid = ds_grid_create(room_size, room_size);

randomize();
var seed = irandom(12345) + 12345;

for(var yy = 0; yy < room_size; ++yy)
    {
    for(var xx = 0; xx < room_size; ++xx)
        {
        grid[# xx, yy] = round(scr_perlin_noise(xx, yy, 3, seed, 48));
        }
    }

scr_filter_grid(grid);
```

Esimerkkikoodi 5. Karttapohjan luominen ja siistiminen karttaa hallinnoivassa peliobjektissa.

Lisäksi on tärkeä tiedostaa, että GameMaker käyttää satunnaislukugeneraattorissaan aina samaa siemenlukua. Jos karttojen halutaan olevan joka kerta erilaisia, täytyy ennen kartan luomista kutsua pelimoottoriin sisäänrakennettua randomize-funktiota, joka vaihtaa siemenluvun. Kartan muuttumattomuus voi kuitenkin kehitysvaiheessa olla kätevää vianetsinnän kannalta, kun kartassa olevat mahdolliset virheet löytyvät aina samasta kohdasta jokaisella suorituskerralla. [44.]

4.2 Automaattinen tile-elementtien asettelu

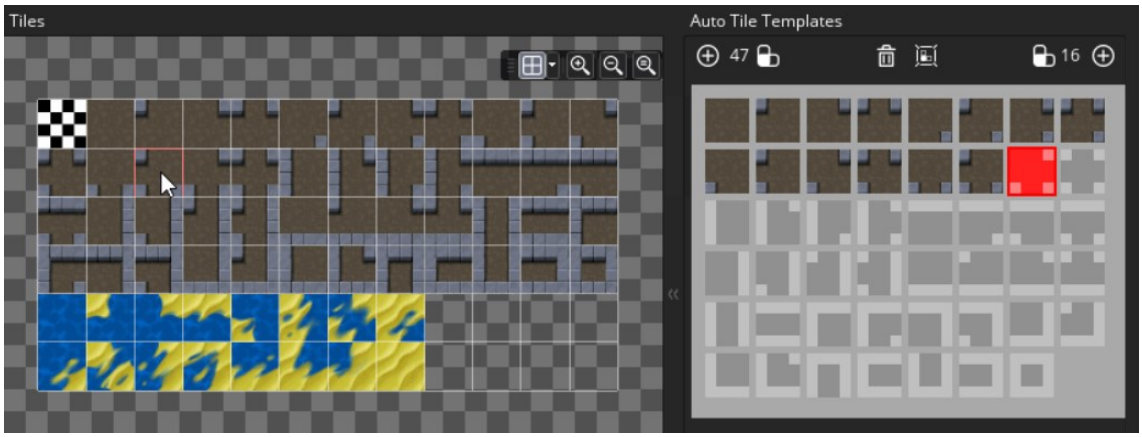
Maastolaatoista koostuvan pelimaailman materiaalit halutaan useimmissa tapauksissa sulauttaa keskenään tai kasata päällekkäin tavalla, jossa eri materiaalien kohtaamispisteessä maastolaatta sisältää tekstuuria molemmista materiaaleista tai niin, että materiaalille luodaan reunus, joka näennäisesti rikkoo maastolaatan neliömäisen rakenteen. Myös maastolaatoilla toteutetut korkeusvaihtelut, kuten seinät, seinämät ja rinteet, halutaan usein toteuttaa tavalla, jossa osa seinämästä jää näkyviin, jotta lopputulos näyttäisi luonnolliselta. Ilman reunapaloja rakennetussa tile-kartassa materiaalit näyttävät usein epäorgaanisilta ja tekstuurin toistuminen on helpommin huomattavissa, minkä lisäksi myös korkeuserot voivat olla vaikeasti hahmotettavissa (kuva 12).



Kuva 12. Pelimaailma, jossa maastolaatoissa ei ole erillisiä reunapaloja.

Maastolaattojen sulavampaa lomittumista varten täytyy tehdä erilaisia reunapaloja sisältävä tile-sarja, josta voidaan tile-elementtejä asetellessa valita sellaisia reunapaloja, jotka vastaavat tekstuuriltaan ympäröivien solujen materiaalia. Valintaprosessi voidaan monissa peleissä hoitaa manuaalisesti GameMakerin kenttäeditorissa, mutta suurissa maailmoissa on nopeampaa ja suoritusaikana

luoduissa maailmoissa pakollista toteuttaa tile-elementtien asettelu automatisoidulla toiminnolla. GameMakerin kenttäeditorissa on pelimoottoriin sisäänrakennettu Auto Tile -ominaisuus, joka voidaan konfiguroida valitsemaan oikeat reunapalat tile-elementin tekstuuriksi, mutta työkalun käyttö rajoittuu vain etukäteen käsityönä tehtyjen pelimaailmojen sisustamiseen (kuva 13) [42]. Jotta automaattinen tile-elementtien asettelu pystyttäisiin toteuttamaan suoritusaikana, täytyy pelikehittäjän toteuttaa oma algoritmi, joka vastaa tile-elementtien asettelulogiikasta.



Kuva 13. GameMaker pelimoottorin automaattisten tile-elementtien asettelun konfiguraatioikkuna [45].

Tile-elementtien asettelulogiikan toteutus riippuu siitä, kuinka monta kuvaa materiaalia kohden halutaan käyttää. Kuvien lukumäärä määräytyy halutun lopputuloksen ulkoasun ja materiaalin luonteen mukaan. Useissa tapauksissa maastolaatat, jotka kuvaavat tasaista materiaalia, kuten nurmikkoa tai hiekkaa, voidaan käyttää 16 kuvan sarjaa, mutta paremman lopputuloksen saavuttamiseksi tarvitaan 47 kuvaa, mikä sisältää kaikkien mahdollisten kulmien variaatiot. Yleensä maantasosta poikkeavat seinämät vaativat suuremman määrän kuvia, jotta saavutetaan illuusio korkeuseroista.

Automaattinen tile-elementtien asettelu 16 kuvalla

Insinöörityötä varten tehdyssä peliprojektissa toteutettiin funktio 16 tile-elementin sarjan automaattiselle asettelulle, ja sille annettiin parametreinä kohteena

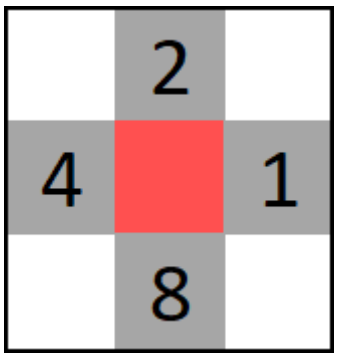
oleva tile-kartta, materiaalitiedon sisältävä DS-ruudukko, kartan koko ruutuina, kohdesolun koordinaatit sekä etäisyys, joka määräsi, kuinka monen ruudun etäisyydeltä soluja käsitellään (esimerkkikoodi 6). Etäisyys asetettiin oletuksena ulottumaan myös viereisiin soluihin, koska kohdesolun tile-elementin vaihtuessa täytyi myös sen ympäröivien tile-elementtien vaihtua, jotta materiaalien reunukset saatiin näyttämään oikealta. Jos funktiolla haluttiin asetella koko kartan tile-elementit kerralla, asetettiin kohderuudun koordinaateiksi kartan nollakoordinaatit ja etäisyydeksi kartan koko.

```
function scr_autotile_16(_tilemap, _grid, _size, _map_size, _x, _y,
    _range = 2)
    {
    var start_x = clamp(_x - _range + 1, 0, _map_size);
    var end_x   = clamp(_x + _range + 0, 0, _map_size);

    var start_y = clamp(_y - _range + 1, 0, _map_size);
    var end_y   = clamp(_y + _range + 0, 0, _map_size);

    var count;
    var cx_minus, cx_plus, cy_minus, cy_plus;
    var target, target_right, target_up, target_left, target_down;

    for (var yy = start_y; yy < end_y; ++yy)
        {
        for (var xx = start_x; xx < end_x; ++xx)
            {
            cx_plus  = clamp(xx + 1, 0, _map_size - 1);
            cy_minus = clamp(yy - 1, 0, _map_size - 1);
            cy_plus  = clamp(yy + 1, 0, _map_size - 1);
            cx_minus = clamp(xx - 1, 0, _map_size - 1);

            target       = _grid[# xx, yy];
            target_right = _grid[# cx_plus, yy];
            target_up    = _grid[# xx, cy_minus];
            target_left  = _grid[# cx_minus, yy];
            target_down  = _grid[# xx, cy_plus];

            count = 0;

            if (target_right != target) {count |= 1;}
            if (target_up    != target) {count |= 2;}
            if (target_left  != target) {count |= 4;}
            if (target_down  != target) {count |= 8;}
```

```
            tilemap_set(_tilemap, target * 16 + count, xx, yy);
            }
        }
    }
```

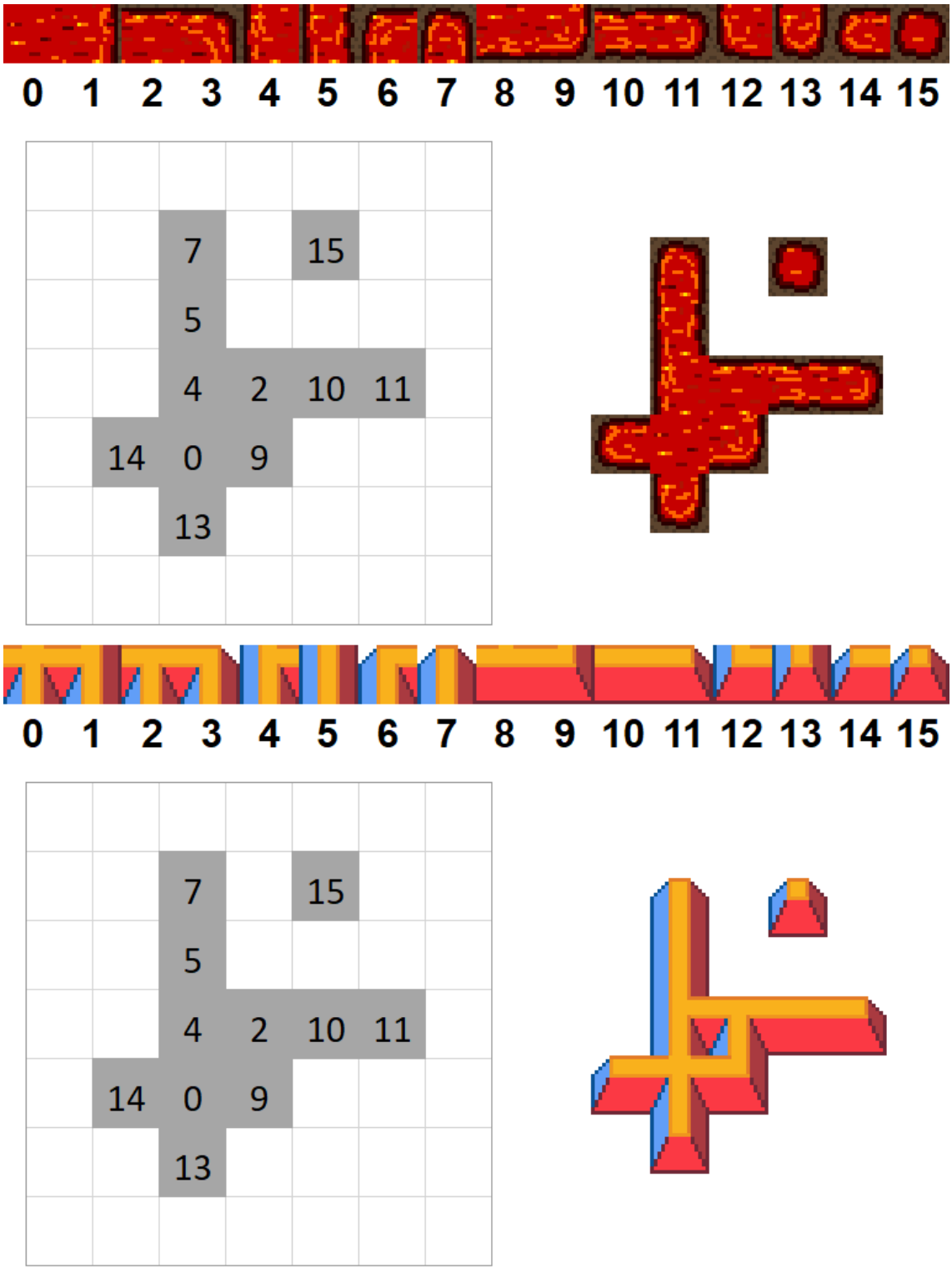
Esimerkkikoodi 6. Automaattinen tile-elementtien asettelufunktio 16 kuvan tile-sarjalle.

Tile-elementtien asettelu -funktio otti huomioon sen, ettei ruudukon ulkopuolelta yritetty hakea tai muuttaa dataa rajaamalla lähtöarvot ja viereisten solujen koordinaatit GameMakerin sisään rakennetulla clamp-funktiolla. Valitun alueen ruudut käytiin läpi silmukassa, joka vertasi viereisten solujen arvoja kohdesolun arvoon ja eriävän naapurisolun kohdalla kasvatti laskuria ennalta määrätyllä luvulla riippuen naapurisolun sijainnista suhteessa kohdesoluun. Ennalta määrätyt luvut olivat kasvavia kahden potensseja, alkaen numerosta yksi ja päättyen numeroon kahdeksan, mikä varmisti sen, että lopputuloksena saatiin aina jokin numero 0:n ja 15:n väliltä, ja sen, että laskurin numero oli aina uniikki verrattuna muihin mahdollisiin naapurisoluyhdistelmiin (kuva 14). [46; 47.]



Kuva 14. Asettelufunktion laskurin kasvatuslogiikka suunnittain 16 kuvan automaattisessa tile-elementtien asettelussa.

Käytetyn tile-sarjan kuvat oli aseteltu niin, että yksi rivi sisälsi aina yhden materiaalin kuvat. Näin funktiossa pystyttiin käyttämään DS-ruudukosta saatua materiaaliarvoa tile-sarjan rivinumerona, kun se vielä kerrottiin 16:lla, eli yhden materiaalin kuvien määrällä. Kerrottu rivinumero lisättynä laskurin arvoon kertoi tarvittavan tile-elementin sijainnin tile-sarjassa, minkä perusteella funktio asetti oikean maastolaatan tile-karttaan tilemap_set-funktiolla (kuva 15).



Kuva 15. Kahden erilaisen tile-sarjan kuvajärjestys ja asettelufunktion tuottamat arvot esimerkkimuodostelmassa.

16 kuvan tile-sarjan ongelmana voi pitää sisäkulmiin jääviä poikkeavuuksia, jonne materiaalireunusta ei piirretä. Vaihtoehtoisesti tile-sarja voidaan piirtää niin, että reunus jatkuu kaikkien kuvien sisäosissa. Tällöin kuitenkin myös materiaalin keskelle syntyy reunuksia, jolloin niitä ei useimmissa tapauksissa voi käyttää. Ongelmaa ei kuitenkaan välttämättä muodostu, jos kyseessä on esimerkiksi seinämä, joka estää sisäosien näkymisen lopullisessa pelissä (kuva 16).

Kuva 16. Korkeiden maastomuotojen sisäosien piilottaminen valaistuksen avulla.

Visuaalisia ongelmia 16 kuvan tile-sarjassa voidaan myös peittää ylemmän tile-kerroksen kuvilla. Toisessa tile-kerroksessa käytettävä tile-sarja voisi esimerkiksi sisältää vain materiaalin keskiosia tai valaistusta simuloivia mustia tile-elementtejä. Parempi lähestymistapa on kuitenkin valmistaa kyseiselle materiaalilla 47 kuvan tile-sarja, jolloin kompromisseja tai useita tile-kerroksia ei tarvita.

Automaattinen tile-elementtien asettelu 47 kuvalla

Jotta reunus saataisiin jatkumaan myös sisäkulmissa, täytyy tile-sarjassa olla 47 kuvaa, mikä kattaa kaikki sisäkulmavariaatiot [47]. Toteutettu funktio 47

kuvan automaattiselle tile-elementtien asettelulle vastasi periaatteeltaan 16 kuvan versiota, mutta neljän suunnan sijaan se vertaili myös soluja, jotka olivat viistossa kohdesoluun nähden (esimerkkikoodi 7; kuva 17).

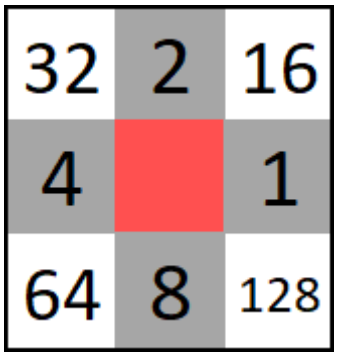
```
function scr_autotile_47(_tilemap, _grid, _tile_size, _room_size, _x, _y,
    _range = 2)
    {
    var start_x = clamp(_x - _range + 1, 0, _room_size);
    var end_x   = clamp(_x + _range + 0, 0, _room_size);

    var start_y = clamp(_y - _range + 1, 0, _room_size);
    var end_y   = clamp(_y + _range + 0, 0, _room_size);

    var count;
    var cx_minus, cx_plus, cy_minus, cy_plus;
    var target, target_right, target_up, target_left, target_down;

    for (var yy = start_y; yy < end_y; ++yy)
        {
        for (var xx = start_x; xx < end_x; ++xx)
            {
            cx_plus  = clamp(xx + 1, 0, _room_size - 1);
            cy_minus = clamp(yy - 1, 0, _room_size - 1);
            cy_plus  = clamp(yy + 1, 0, _room_size - 1);
            cx_minus = clamp(xx - 1, 0, _room_size - 1);

            target       = _grid[# xx, yy];
            target_right = _grid[# cx_plus, yy];
            target_up    = _grid[# xx, cy_minus];
            target_left  = _grid[# cx_minus, yy];
            target_down  = _grid[# xx, cy_plus];

            count = 0;

            if (target_right < target) {count |= 1;}
            if (target_up    < target) {count |= 2;}
            if (target_left  < target) {count |= 4;}
            if (target_down  < target) {count |= 8;}

            tilemap_set(_tilemap, target * 16 + count, xx, yy);
            }
        }
    }
function scr_autotile_47(_tilemap, _grid, _room_size, _x, _y, _range = 2)
    {
    var start_x = clamp(_x - _range + 1, 0, _room_size);
```

```
    var end_x   = clamp(_x + _range + 0, 0, _room_size);
    var start_y = clamp(_y - _range + 1, 0, _room_size);
    var end_y   = clamp(_y + _range + 0, 0, _room_size);

    var target, count;
    var cx_minus, cx_plus, cy_minus, cy_plus;

    for (var yy = start_y; yy < end_y; ++yy)
        {
        for (var xx = start_x; xx < end_x; ++xx)
            {
            cx_plus  = clamp(xx + 1, 0, _room_size - 1);
            cy_minus = clamp(yy - 1, 0, _room_size - 1);
            cy_plus  = clamp(yy + 1, 0, _room_size - 1);
            cx_minus = clamp(xx - 1, 0, _room_size - 1);

            target = _grid[# xx, yy];
            count = 0;

            if (_grid[# cx_plus, yy]  < target) {count |= 1;}
            if (_grid[# xx, cy_minus] < target) {count |= 2;}
            if (_grid[# cx_minus, yy] < target) {count |= 4;}
            if (_grid[# xx, cy_plus]  < target) {count |= 8;}

            switch (count)
                {
            case 0:
                if (_grid[# cx_plus,  cy_minus] > target) {count |= 16;}
                if (_grid[# cx_minus, cy_minus] > target) {count |= 32;}
                if (_grid[# cx_minus, cy_plus]  > target) {count |= 64;}
                if (_grid[# cx_plus,  cy_plus]  > target) {count |= 128;}
                break;
            case 1:
                if (_grid[# cx_minus, cy_minus] > target) {count |= 32;}
                if (_grid[# cx_minus, cy_plus]  > target) {count |= 64;}
                break;
            case 2:
                if (_grid[# cx_minus, cy_plus]  > target) {count |= 64;}
                if (_grid[# cx_plus,  cy_plus]  > target) {count |= 128;}
                break;
            case 3:
                if (_grid[# cx_minus, cy_plus]  > target) {count |= 64;}
                break;
            case 4:
                if (_grid[# cx_plus,  cy_minus] > target) {count |= 16;}
                if (_grid[# cx_plus,  cy_plus]  > target) {count |= 128;}
                break;
            case 6:
```

```
            if (_grid[# cx_plus,  cy_plus]  > target) {count |= 128;}
            break;
        case 8:
            if (_grid[# cx_plus,  cy_minus] > target) {count |= 16;}
            if (_grid[# cx_minus, cy_minus] > target) {count |= 32;}
            break;
        case 9:
            if (_grid[# cx_minus, cy_minus] > target) {count |= 32;}
            break;
        case 12:
            if (_grid[# cx_plus, cy_minus]  > target) {count |= 16;}
            break;
            }

        switch(count)
            {
            case 0:     count = 0;  break;
            case 16:    count = 17; break;
            case 20:    count = 18; break;
            case 24:    count = 19; break;
            case 28:    count = 20; break;
            case 32:    count = 21; break;
            case 33:    count = 22; break;
            case 40:    count = 23; break;
            case 41:    count = 24; break;
            case 48:    count = 25; break;
            case 56:    count = 26; break;
            case 64:    count = 27; break;
            case 65:    count = 28; break;
            case 66:    count = 29; break;
            case 67:    count = 30; break;
            case 80:    count = 31; break;
            case 96:    count = 32; break;
            case 97:    count = 33; break;
            case 112:   count = 34; break;
            case 128:   count = 35; break;
            case 130:   count = 36; break;
            case 132:   count = 37; break;
            case 134:   count = 38; break;
            case 144:   count = 39; break;
            case 148:   count = 40; break;
            case 160:   count = 41; break;
            case 176:   count = 42; break;
            case 192:   count = 43; break;
            case 194:   count = 44; break;
            case 208:   count = 45; break;
            case 224:   count = 46; break;
            case 240:   count = 47; break;
```

```
                }

            tilemap_set(_tilemap, target * 16 + count, xx, yy);
            }
        }
    }
```

Esimerkkikoodi 7. Automaattinen tile-elementtien asettelufunktio 47 kuvan tile-sarjalle.

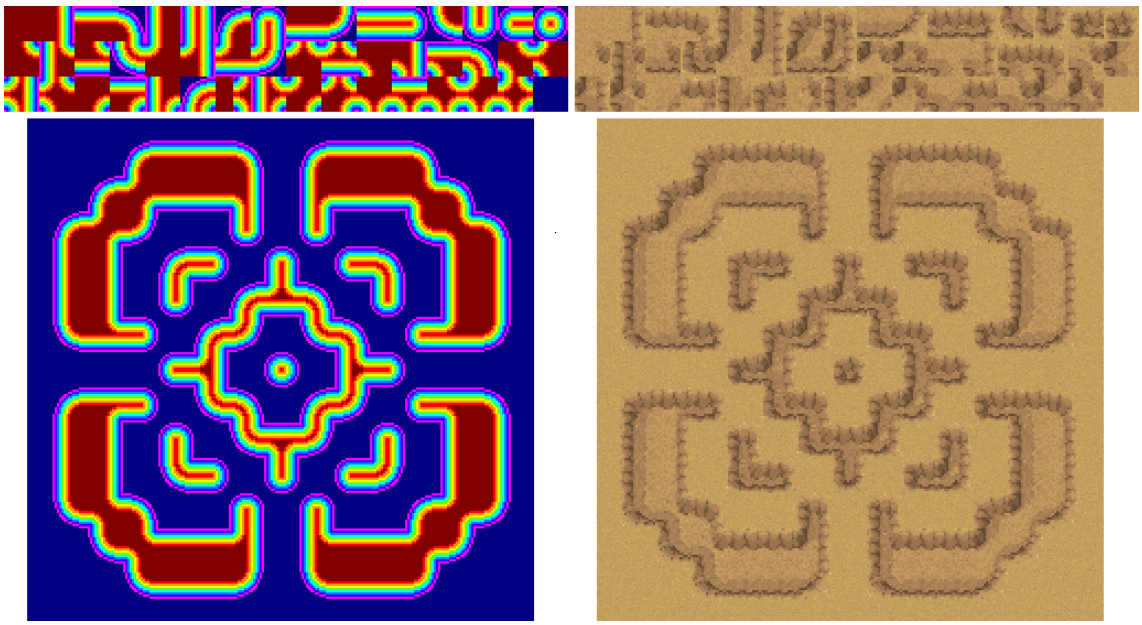


Kuva 17. Asettelufunktion laskurin kasvatuslogiikka suunnittain 47 kuvan automaattisessa tile-elementtien asettelussa.

Toinen tärkeä eroavaisuus oli se, että funktion laskuri palautti arvoja 0:n ja 255:n väliltä, vaikka tarvittavia kuvia oli vain 47 [47]. Olennaista oli, ettei horisontaalisten ja vertikaalisten solujen viereisiä soluja tarvinnut tarkastella, kun kyseisessä solussa oli eri materiaalia kuin kohdesolussa. Ylimääräiset tarkastukset saatiin rajattua pois käyttämällä switch-case-valintarakennetta. Jäljelle jäävät tapaukset laskurin tuottamista luvuista eivät kuitenkaan olleet peräkkäisiä numeroita tai edes loogisessa järjestyksessä, jotta niiden perusteella olisi voitu valita oikea maastolaatta tile-sarjasta algoritmillisesti [47]. Ratkaisuksi tehtiin toinen valintarakenne, johon kirjattiin kaikki mahdolliset tapaukset alkuperäisistä laskurin arvoista, jotka sitten muutettiin vastaamaan tile-sarjana kuvien järjestystä.

47 tile-kuvan sarjan piirtäminen ja järjestäminen jokaista materiaalia varten on työläs prosessi, mutta tuottaa yleensä aina parhaimman näköisen lopputuloksen (kuva 18). Ajan säästämiseksi kannattaa kuitenkin harkita, että vain ne materiaalit, joiden yksityiskohdilla on merkittävä vaikutus visuaalisuuteen, kuten

korkeuseroja sisältävät maastolaatat, toteutettaisiin tällä tekniikalla ja loput yksinkertaisemmalla 16 kuvan tile-sarjoilla.



Kuva 18. Kaksi 47 kuvan tile-sarjaa ja lopputulokset testimuodostelmissa.

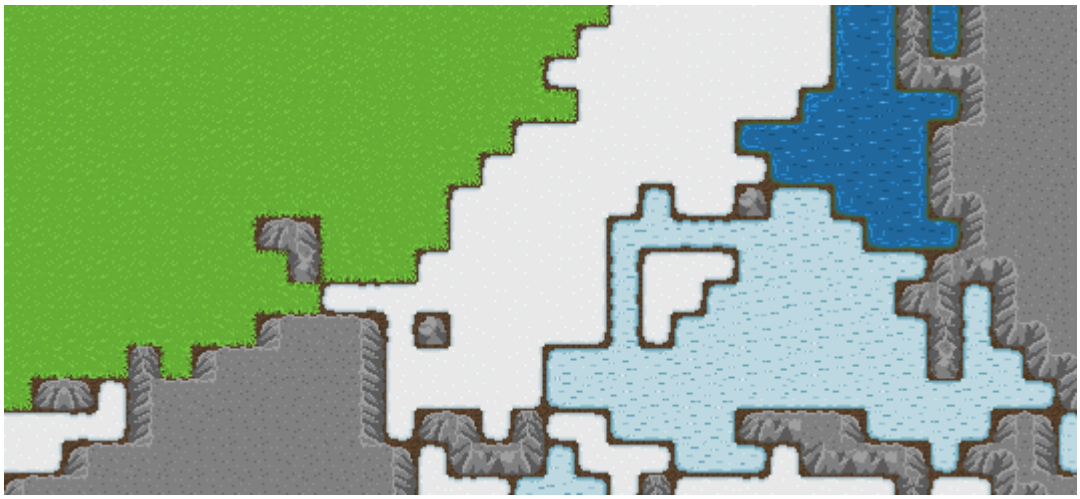
Useita eri maastotyyppejä sisältävät maailmat ovat kuitenkin ongelmallisia siksi, että maastolaatat voivat saada naapurikseen enemmän kuin yhden erilaisen materiaalin. Automaattisesta tile-elementtien asettelulogiikasta riippuen naapuriruuduissa voi olla enintään neljä tai kahdeksan eri materiaalia. Esimerkiksi 47 kuvan sarja riittää kattamaan vain kahden eri materiaalin kohtaamiseen tarvittavat kuvat. Materiaaleja lisätessä kuvien määrä kasvaa eksponentiaalisesti ja jo kolmen eri materiaalin kohtaamiseen tarvittaisiin niin paljon kuvia, ettei niiden piirtäminen käsityönä olisi järkevää. [46.]

Automaattinen tile-elementtien asettelu päällekkäisillä tile-kerroksilla

Pelissä, jossa maastomateriaaleja voi olla useita, voidaan ongelmaa lähestyä kolmella eri tavalla. Ensimmäinen vaihtoehto on rajata vierekkäisten materiaalien syntyminen niin, että tietyn materiaalin vieressä voi olla vain toista tiettyä materiaalia, esimerkiksi niin, että veden ympärille voi syntyä vain ruohoa, ruohon ympärille vain kiveä ja niin edelleen, jolloin tarvittavien kuvien määrä pysyy

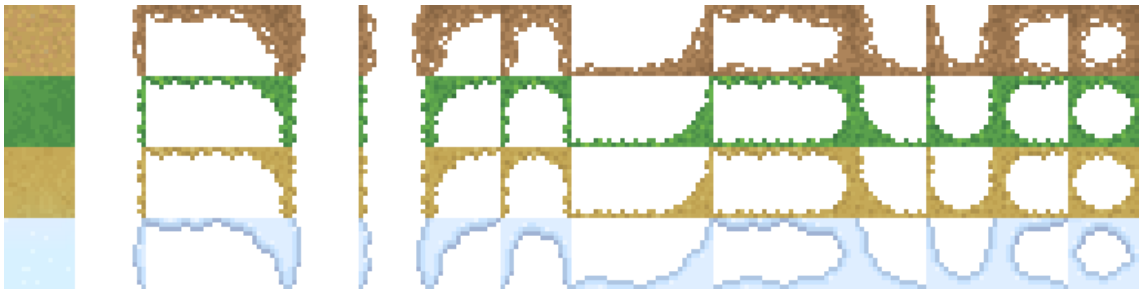
maltillisena [48]. Hiekkalaatikkopelissä tämä ei kuitenkaan usein ole mahdollista, mikäli maailman halutaan olevan vapaasti muokattavissa, jolloin pelaaja voisi toimillaan muuttaa materiaalien järjestystä niin, että viereisillä tile-elementeillä ei olisi enää yhteen liittämiseen tarvittavia kuvia tile-sarjassa.

Toinen vaihtoehto on luoda jokaiselle materiaalille yhtenäinen reunus reunapalojen tekstuuriin, jolloin mikä tahansa materiaali voidaan sijoittaa minkä tahansa toisen materiaalin viereen. Yhtenäisen reunusmateriaalin tekniikka tuottaa kuitenkin visuaalisia ongelmia, koska se muodostaa rajoja eri materiaalien väleille. Esimerkiksi reunuksen ollessa multaa monet materiaalit sulautuvat siedettävästi keskenään, mutta multareunuksen muodostuminen jään ja lumen reunoille saa lopputuloksen näyttämään epäorgaaniselta (kuva 19).



Kuva 19. Pelimaailman osa sisustettuna tile-elementeillä, joissa yhdistävä reunamateriaali.

Lähestymistavoista joustavin olisi erottaa reuna- ja kulmapalat omille tile-kerroksilleen ja siten piirtää ne perusmateriaalilaattojen päälle. Jokaisen tile-kerroksen edustaessa horisontaalisen tai vertikaalisen naapurimaastolaatan reunusta tarvitaan niitä yhteensä viisi, kun mukaan lasketaan keskipaloille tarkoitettu tile-kerros. Lisäksi tarvittavien kuvien määrä pysyy vähäisenä, sillä jokaista materiaalia varten tarvitaan vain 16 kuvaa (kuva 20). [48.]



Kuva 20. Tile-sarja, jossa neljä materiaalia reunapaloineen.

Peliprojektissa toteutettiin myös automaattisesta tile-elementtien asettelusta versio, joka hyödynsi useita tile-kerroksia. Jotta uusi asettelufunktio pystyttiin toteuttamaan, luotiin maailman ylläpidosta vastaavan peliobjektin create-tapahtumassa tarvittavat tile-kerrokset ja ne tallennettiin arr_tilemaps-nimiseen taulukkoon (esimerkkikoodi 8). Pohjimmaiselle eli materiaalien keskipaloja sisältävälle tile-kartalle asetettiin tile-kerroksen syvyydeksi nolla ja reunapaloille tarkoitetut kerrokset asetettiin negatiivisiksi, jotta ne piirrettäisiin päällimmäisiksi.

```
var size = 128;
list_terrains = ds_list_create();
arr_tilemaps = array_create(5);

lay_id = layer_create(0);
arr_tilemaps[4] = layer_tilemap_create(lay_id, 0, 0, tileset, size,size);
lay_id = layer_create(-1);
arr_tilemaps[0] = layer_tilemap_create(lay_id, 0, 0, tileset, size,size);
lay_id = layer_create(-2);
arr_tilemaps[1] = layer_tilemap_create(lay_id, 0, 0, tileset, size,size);
lay_id = layer_create(-3);
arr_tilemaps[2] = layer_tilemap_create(lay_id, 0, 0, tileset, size,size);
lay_id = layer_create(-4);
arr_tilemaps[3] = layer_tilemap_create(lay_id, 0, 0, tileset, size,size);
```

Esimerkkikoodi 8. Viiden tile-kerroksen luominen ja tile-karttojen alustus.

Useita tile-kerroksia hyödyntävä tile-elementtien asettelufunktio muistutti periaatteiltaan 16 kuvan asettelufunktiota, mutta ennen kuin laskurimuuttujaa ryhdyttiin kasvattamaan, kerättiin DS-lista-tietorakenteeseen tieto siitä, kuinka monta erilaista naapurimateriaalia solulla oli (esimerkkikoodi 9). Sen perusteella pystyttiin rajaamaan tarkastusten määrää. Funktio asetti reunuksen

viereisen solun poiketessa tarkasteltavasta solusta materiaalirajan molemmille puolille, jotta reunus olisi mahdollisimman orgaanisen näköinen, kun ruudukkomaista rakennetta saatiin rikottua rajan molemmin puolin.

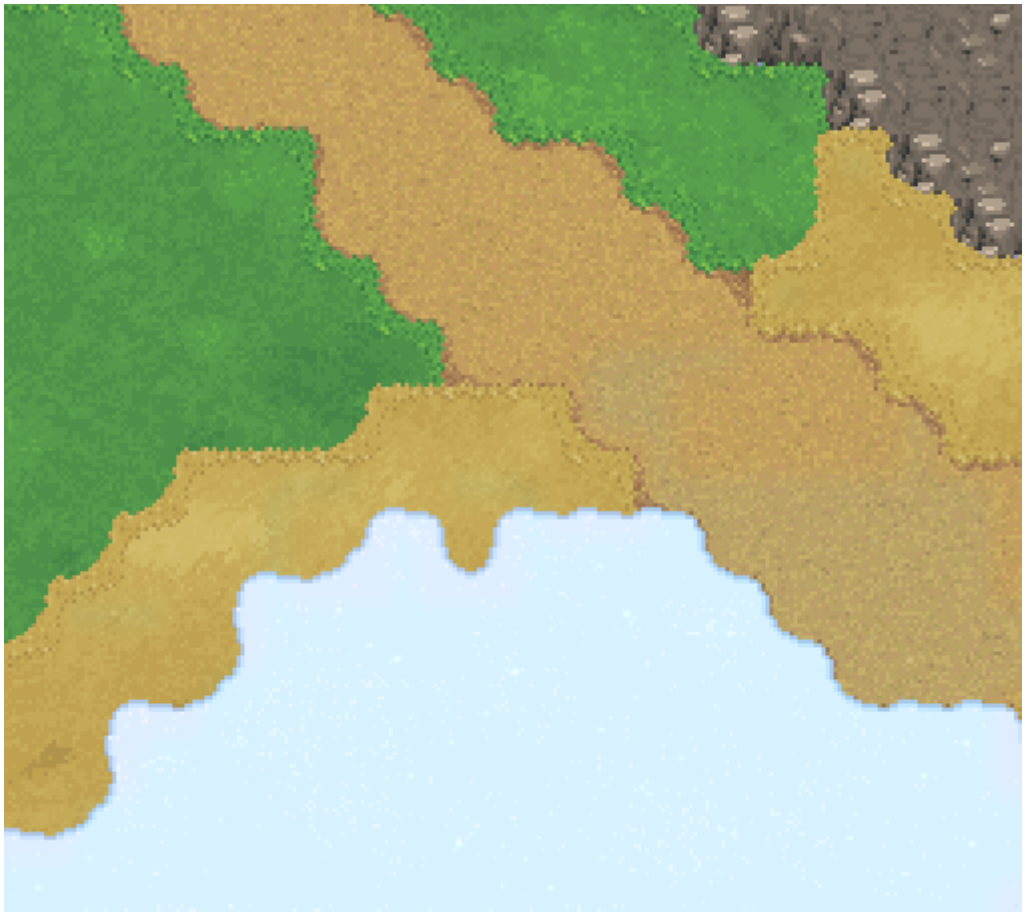
```
function scr_autotile_edges(_tilemap, _grid, _rm_size, _x, _y, _range=2)
    {
    var start_x = clamp(_x - _range + 1, 0, _rm_size);
    var end_x   = clamp(_x + _range + 0, 0, _rm_size);
    var start_y = clamp(_y - _range + 1, 0, _rm_size);
    var end_y   = clamp(_y + _range + 0, 0, _rm_size);

    var count;
    var cx_minus, cx_plus, cy_minus, cy_plus;
    var target, target_right, target_up, target_left, target_down;

    var list_terrains = ds_list_create();

    for (var yy = start_y; yy < end_y; ++yy)
        {
        for (var xx = start_x; xx < end_x; ++xx)
            {
            cx_plus  = clamp(xx + 1, 0, _rm_size - 1);
            cy_minus = clamp(yy - 1, 0, _rm_size - 1);
            cy_plus  = clamp(yy + 1, 0, _rm_size - 1);
            cx_minus = clamp(xx - 1, 0, _rm_size - 1);

            target       = _grid[# xx, yy];
            target_right = _grid[# cx_plus, yy];
            target_up    = _grid[# xx, cy_minus];
            target_left  = _grid[# cx_minus, yy];
            target_down  = _grid[# xx, cy_plus];

            tilemap_set(_tilemap, target * 16, xx, yy);

            tilemap_set(arr_tilemaps[0], 0, xx, yy);
            tilemap_set(arr_tilemaps[1], 0, xx, yy);
            tilemap_set(arr_tilemaps[2], 0, xx, yy);
            tilemap_set(arr_tilemaps[3], 0, xx, yy);

            if (target_right != target)
                {ds_list_add(list_terrains, target_right);}
            if (target_up    != target)
                {
                if (ds_list_find_index(list_terrains, target_up) == -1)
                    {ds_list_add(list_terrains, target_up);}
                }
```

```
            if (target_left  != target)
                {
                if (ds_list_find_index(list_terrains, target_left) == -1)
                    {ds_list_add(list_terrains, target_left);}
                }
            if (target_down  != target)
                {
                if (ds_list_find_index(list_terrains, target_down) == -1)
                    {ds_list_add(list_terrains, target_down);}
                }

            var size_list = ds_list_size(list_terrains);
            if (size_list > 0)
                {
                ds_list_add(list_terrains, target);

                ds_list_sort(list_terrains, false);

                for (var i = 0; i <= size_list; ++i)
                    {
                    count = 0;

                    if (list_terrains[| i] == target)
                        {
                        if (target_right != target) {count |= 1;}
                        if (target_up    != target) {count |= 2;}
                        if (target_left  != target) {count |= 4;}
                        if (target_down  != target) {count |= 8;}
                        }
                    else if (list_terrains[| i] > target)
                        {
                        if (target_right == list_terrains[| i])
                            {count |= 1;}
                        if (target_up    == list_terrains[| i])
                            {count |= 2;}
                        if (target_left  == list_terrains[| i])
                            {count |= 4;}
                        if (target_down  == list_terrains[| i])
                            {count |= 8;}
                        }

                    if (count > 0)
                        {
                        ++count;
                        tilemap_set(arr_tilemaps[i],
                            list_terrains[| i] * 16 + count,xx, yy);
                        }
                    }
```

```
                ds_list_clear(list_terrains);
                }
            }
        }
        ds_list_destroy(list_terrains);
    }
```

Esimerkkikoodi 9. Automaattinen tile-elementtien asettelufunktio, joka asettelee maastolaatat tile-kartoille viiteen eri kerrokseen.

Reunus kuitenkin jätettiin piirtämättä, kun viereinen materiaaliarvo oli suurempi kuin käsiteltävässä solussa, jolloin materiaaleille muodostui vaikutelmaa syvyydestä toisiinsa nähden. Käytännössä tämä tarkoitti peliprojektissa sitä, että lumen reunukset piirrettiin aina nurmikon päälle ja nurmikon reunukset hiekan päälle, mutta ei päinvastoin, koska lumi oli arvohierarkiassa suurempi arvo kuin nurmikon arvo ja nurmikon arvo suurempi kuin hiekan (kuva 21).



Kuva 21. Pelimaailman osa, jossa maastolaattojen reunukset on aseteltu viereisen materiaalin tile-elementtien päälle ylempiin tile-kerroksiin.

Lisäksi hiekan erottuvuuden parantamiseksi laadittiin poikkeussääntö, jossa hiekalle piirrettiin reunus, vaikka se olikin materiaaliasteikon pohjimmaisena. Ylimääräinen ruskea reunus, joka näkyi muiden reunusten alta, korosti osaltaan pieniä korkeuseroja materiaalien välillä ja teki materiaaleista helpommin tunnistettavia.

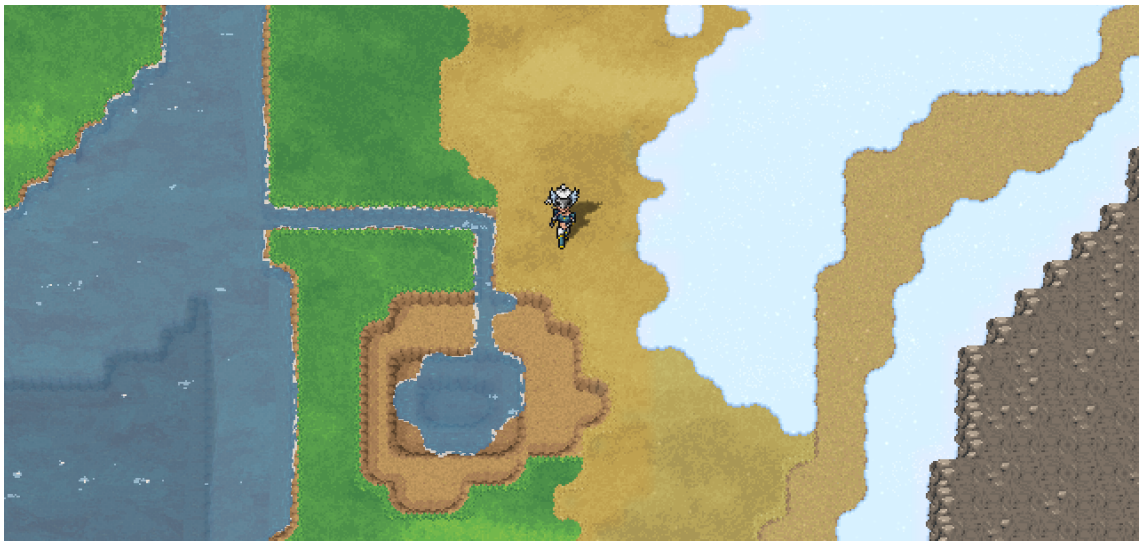
Useita tile-kerroksia hyödyntävän tekniikan heikkoutena voidaan kuitenkin pitää sitä, ettei sillä pystytä toteuttamaan kovin pieniä ja yksityiskohtaisia muodostelmia, sillä tekniikka käytettäessä kohdesoluun asetetaan aina materiaalin keskipalaa edustava tile-elementti ja reunapalat sijoitetaan sen ympärillä oleviin soluihin tile-kartalla. Tekniikka tuottaa kuitenkin visuaalisesti miellyttävän näköisen lopputuloksen, kun materiaalien rajat saadaan toteutettua luonnollisen näköisesti naapurisolun materiaalista riippumatta ja soveltuu siksi hyvin maaperän materiaalien visualisointiin.

5 Tulokset

Insinöörityön lopputuloksena valmistui peliprojektin alku, johon kehitettiin maailmanluonnin lisäksi erilaisia mekaniikkoja, aina nesteiden virtauksista kaasu- ja lämpötilasimulaatioihin. Pelihahmo pystyi kaivamaan tunneleita kiveen ja uria maahan, ja niitä pitkin vesi pääsi virtaamaan. Pelimaailmassa materiaalit muuttuivat lämpötilan mukaan siten, että ilman kylmetessä ruoho alkoi kellastua ja lopuksi peittyä lumella. Pelihahmo pystyi myös sytyttämään muita peliobjekteja tuleen, mikä tuotti lämpöä ja ilmansaasteita. Saasteet levisivät kartalla tuulensuuntaisesti sumumaisina pilvinä, joiden on seuraavassa kehitysvaiheessa tarkoitus vaikuttaa muihin peliobjekteihin (kuva 22).

Kuva 22. Saastepilviä edustavia tile-elementtejä muodostuu palavista puista.

Peliprojektissa päädyttiin käyttämään kaikkia kehitettyjä tile-elementtien asettelutekniikoita eri materiaaleille, jotta saavutettiin mahdollisimman monipuolinen lopputulos. Kuoppia edustavat maastolaatat, jotka vaativat paljon yksityiskohtia, toteutettiin 47 kuvan sarjoilla. Kallio ja vesi pystyttiin puolestaan toteuttamaan 16 kuvan tekniikalla ja loput maastoelementit hyödynsivät useampia tile-kerroksia, jotta niiden reunat saatiin näyttämään luonnollisilta riippumatta viereisten maastolaattojen materiaalista (kuva 23).



Kuva 23. Nestesimulaatio levittää vettä edustavia tile-elementtejä maahan kaivettua uraa pitkin syvempään kuoppaan.

Veden elävöittämiseksi käytettiin myös varjostinta, joka lisäsi tile-elementteihin lainehtimista muistuttavaa liikettä ja kuohuja, jotta materiaali näyttäisi uskottavalta nestemäiseltä aineelta. Saastesumu käytti myös varjostinta, joka puolestaan hyödynsi Perlin noise -funktiolla tuotettua tekstuuria luodakseen tile-elementeille savumaisen ulkoasun.

Kehitystyön myötä oli selvää, että GameMaker-pelimoottori sopii proseduraalista maailmanluontia käyttävän pelin luomiseen, jossa maailma olisi vapaasti muokattavissa. Jatkokehityksen aikana luodut erilaiset virtaussimulaatioiden toteuttamiset onnistuivat myös, vaikka etukäteen pelkona oli, että ne kuormittavat liikaa prosessorin suorituskykyä. Onnistunut lopputulos vahvisti käsitystä siitä, että peliprojektia kannattaa kehittää eteenpäin, sillä proseduraalisuutta ja simulaatioita sisältävät pelit ovat erittäin suosittuja ja niillä on suuret markkinat. Esimerkkeinä mainittakoon miljoonia kopioita myyneet pelit Minecraft, Terraria ja Factorio [49; 50; 51].

6 Yhteenveto

Insinöörityön tarkoitus oli kehittää menetelmä, jolla kaksiulotteinen ylhäältäpäin kuvattu pelimaailma pystyttäisiin totuttamaan proseduraalisesti GameMaker-pelimoottorilla ja visualisoimaan se niin, että lopputulos olisi mahdollisimman näyttävä. Tavoitteena oli myös tehdä pelimaailmasta muokattava suoritusaikana siten, että pelaaja voisi toimillaan muuttaa sen ympäristön rakenteita.

Työssä perehdyttiin GameMaker-pelimoottoriin ja sen tarjoamiin suorituskyvyltään tehokkaiden maastolaattojen hyödyntämiseen paloista koostuvan pelimaailman rakentamisessa. Lisäksi tutkittiin proseduraalista pelisisällön tuottamista ja hiekkalaatikkopelien ominaispiirteitä. Lopputuotoksena kehitettiin proseduraalisten karttojen esikateluun tarkoitettu apuohjelma ja peliprojektin alku, joka käytti Perlin noise -algoritmia tuottaakseen tietorakenteita, joiden arvojen pohjalta maastolaattojen asettelu voitiin suorittaa.

Työtä tehdessä ilmeni useita haasteita, jotka liittyivät niin karttapohjan luontiin kuin maastolaattojen asetteluun, joita varten täytyi kehittää useita erilaisia ratkaisuja. Karttapohjan luomiseen proseduraalisesti löytyi paljon lähdemateriaalia, mutta sen jatkojalostukseen toimivaksi pelimaailmaksi täytyi soveltaa ja kehittää muita tekniikoita, joita harvassa lähteessä yhdistettiin proseduraaliseen maailman luontiin. Maastolaattojen asettelulogiikan kehityksessä ilmeni lukuisia visuaalisia ongelmia, joiden ratkaiseminen vaati perehtymistä moniin eri lähestymistapoihin. Maastolaattojen asettelua varten valitut tekniikat vaativat myös paljon jatkokehitystä, jotta lopputulos saatiin näyttämään hyvältä.

Lopputulos täytti kuitenkin kaikki odotukset mielenkiintoisen ja näyttävän pelimaailman luomisesta. Lisäksi tavoitteena ollut hiekkalaatikkopeleille tyypillinen maailman muokattavuus pystyttiin toteuttamaan insinöörityön aikana kehitetyssä peliprojektissa. Pelimaailman jatkokehityksen seuraava vaihe on kansoittaa maailmaa peliobjekteilla ja ennalta rakennetuilla kartanosilla, kuten metsillä, kylillä, linnoilla ja luolilla sekä niihin liittyvillä rakennuksilla, pelihahmoilla ja kasvillisuudella.

Lähteet

1 Dealessandri, Marie. 2020. What is the best game engine: is GameMaker right for you? Verkkoaineisto. <https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-gamemaker-the-right-game-engine-for-you#section-1>. 16.1.2020. Luettu 18.4.2022.

2 Opera Acquires YoYo Games, Launches Opera Gaming. 2021. Verkkoaineisto. Cision PR Newswire. <https://www.prnewswire.com/news-releases/opera-acquires-yoyo-games-launches-opera-gaming-301211728.html>. 20.1.2021. Luettu 18.4.2022.

3 Risk of Rain – Stats on Steam. Verkkoaineisto. Games-stats. <https://games-stats.com/steam/game/risk-of-rain>. Luettu 18.4.2022.

4 Hyper Light Drifter – Stats on Steam. Verkkoaineisto. Games-stats. <https://games-stats.com/steam/game/hyper-light-drifter>. Luettu 18.4.2022.

5 Forager – Stats on Steam. Verkkoaineisto. Games-stats. <https://games-stats.com/steam/game/forager>. Luettu 18.4.2022.

6 Undertale – Stats on Steam. Verkkoaineisto. Games-stats. <https://games-stats.com/steam/game/undertale>. Luettu 18.4.2022.

7 Schardon, Lindsay. 2022. Best Game Engines for 2022 – Which Should You Use? Verkkoaineisto. <https://gamedevacademy.org/best-game-engines>. 22.2.2022. Luettu 11.3.2022.

8 Ten Years of Game Maker 1999-2009. GameMakerBlog. 5.11.2009. Verkkoaineisto. <http://gamemakerblog.com/2009/11/15/ten-years-of-game-maker-1999-2009>. Luettu 11.3.2022.

9 GameMaker Versions. 2014. Verkkoaineisto. GameMaker Wiki. <http://game-maker.wikidot.com/game-maker-versions>. 22.12.2014. Luettu 11.3.2022.

10 Minotti, Mike. 2015. Playtech buys GameMaker creator YoYo Games for $16.4 million. Verkkoaineisto. <https://venturebeat.com/2015/02/17/playtech-buys-gamemaker-creator-yoyo-games-for-16-4-million>. 17.2.2015. Luettu 11.3.2022.

11 GameMaker: Studio. Verkkoaineisto. GameMaker Wiki. <https://80.lv/vendors/gamemaker-studio>. Luettu 11.3.2022.

12 Kerr, Chris. 2021. GameMaker: StudioUpdated: Opera buys GameMaker Studio maker YoYo Games for $10 million. Verkkoaineisto. <https://www.gamedeveloper.com/business/-b-updated-b-opera-buys-gamemaker-studio-maker-yoyo-games-for-10-million>. 19.1.2021. Luettu 11.3.2022.

13 Matharoo, Gurpreet. 2017. GameMaker Studio 2 Review: Features. Verkkoaineisto. <https://gdpalace.wordpress.com/2017/03/12/gms2-review>. 12.3.2017. Luettu 11.3.2022.

14 Review: Gamemaker 2. 2017. Verkkoaineisto. Junglist. <https://www.lifehacker.com.au/2017/04/review-gamemaker-2>. 21.4.2017. Luettu 11.3.2022.

15 Alexander, Mark. 2020. Gamemaker Studio 2.3: New GML Features. Verkkoaineisto. <https://gamemaker.io/en/blog/gamemaker-studio-2-dot-3-new-gml-features>. 18.8.2020. Luettu 11.3.2022.

16 YoYoGames Release GameMaker Studio 2 Free Version. 2021. Verkkoaineisto. GameFromScratch.com. <https://gamefromscratch.com/yoyogames-release-gamemaker-studio-2-free-version>. 30.6.2021. Luettu 11.3.2022.

17 Matharoo, Gurpreet. 2022. Version 2022.3: Video Plauback, Steamdeck & More. Verkkoaineisto. <https://gamemaker.io/en/blog/release-2022-3>. 31.3.2022. Luettu 31.3.2022.

18 Wirtz, Bryan. 2022. GML: Create Cross-Platform Video Games in GameMaker Studio. Verkkoaineisto. <https://www.gamedesigning.org/engines/gml>. 8.1.2022. Luettu 31.3.2022.

19 Variables And Variable Scope. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Overview/Variables_And_Variable_Scope.htm>. Luettu 5.3.2022.

20 Data Structures. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Data_Structures/Data_Structures.htm>. Luettu 6.3.2022.

21 Stoltzfus, Justin. 2022. Multithreading. Verkkoaineisto. <https://www.techopedia.com/definition/24297/multithreading-computer-architecture>. 27.4.2022. Luettu 1.5.2022.

22 Object Events. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/The_Asset_Editors/Object_Properties/Object_Events.htm>. Luettu 5.3.2022.

23 GameMaker Language – Beginner's Guide. 2020. Verkkoaineisto. The GameDev Palace. <https://gdpalace.wordpress.com/learn-gml>. 16.1.2020. Luettu 18.4.2022.

24 Accessors. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Overview/Accessors.htm>. Luettu 6.3.2022.

25 Arrays. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Overview/Arrays.htm>. Luettu 6.3.2022.

26 DS Lists. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Data_Structures/DS_Lists/DS_Lists.htm>. Luettu 6.3.2022.

27 DS Grids. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Data_Structures/DS_Grids/DS_Grids.htm>. Luettu 6.3.2022.

28 DS Maps. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Data_Structures/DS_Maps/DS_Maps.htm>. Luettu 6.3.2022.

29 Hämäläinen, Tuukka. 2016. Proseduraalinen generointi on No Man's Skyn salaisuus – mutta mitä se on. Verkkoaineisto. <https://muropaketti.com/pelit/peliartikkelit/proseduraalinen-generointi-no-mans-skyn-salaisuus-mita>. 21.6.3026. Luettu 1.3.2022.

30 Procedural Generation in Game Development. Verkkoaineisto. Davidepsce.com. <https://www.davidepesce.com/2020/02/24/procedural-generation-in-game-development>. Luettu 10.3.2022.

31 Sandbox. 2017. Verkkoaineisto. Techopedia. <https://www.techopedia.com/definition/3952/sandbox-gaming>. 30.6.2017. Luettu 8.3.2022.

32 Bycer, Josch. 2015. The problems with aimless game design. Verkkoaineisto. <https://game-wisdom.com/critical/problems-aimless-game-design>. 17.2.2015. Luettu 11.4.2022.

33 Dwarf Fortress. 2016. Verkkoaineisto. Steam. <https://store.steampowered.com/app/975370/Dwarf_Fortress>. 17.5.2016. Luettu 10.4.2022.

34 The algorithms of No Man’s Sky. Verkkoaineisto. Rambus. <https://www.rambus.com/blogs/the-algorithms-of-no-mans-sky-2>. 17.5.2016. Luettu 12.3.2022.

35 Sorvari, Mika. 2011. Minecraft. Verkkoaineisto. <https://www.gamereactor.fi/arviot/82990/Minecraft>. 12.5.2011. Luettu 12.3.2022.

36 Dwarf Fortress – The Reference for Complexity. Verkkoaineisto. Gush Gaming. <https://gushgaming.wordpress.com/2018/10/24/dwarf-fortress>. Luettu 12.3.2022.

37 Perlin noise. Verkkoaineisto. Khanac Academy. <https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise>. Luettu 14.3.2022.

38 Touti, Raouf. Perlin Noise: A Procedural Generation Algorithm. Verkkoaineisto. <https://rtouti.github.io/graphics/perlin-noise-algorithm>. Luettu 13.3.2022.

39 Programming Perlin-like Noise (C++). 2017. Verkkoaineisto. YouTube. <https://www.youtube.com/watch?v=6-0UaeJBumA>. 30.10.2017. Katsottu 15.3.2021.

40 Coding 2D Perlin Noise in Game Maker. 2015. Verkkoaineisto. YouTube. <https://www.youtube.com/watch?v=vkHYpuo29YI>. 15.10.2015. Katsottu 10.10.2021.

41 Laaksonen, Pasi. Game of Life. Verkkoaineisto. <http://www.tiikoni.net/gameoflife>. Luettu 17.3.2022.

42 Alexander, Mark. 2021. The Tile Set Editor. Verkkoaineisto. <https://gamemaker.io/en/tutorials/tile-set-editor>. 1.1.2021. Luettu 5.3.2022.

43 Make Better Textures, The 'Power Of Two' Rule & Proper Image Dimensions. Verkkoaineisto. KatsBits. <https://www.katsbits.com/tutorials/textures/make-better-textures-correct-size-and-power-of-two.php>. Luettu 20.3.2022.

44 Number Functions. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/GameMaker_Language/GML_Reference/Maths_And_Numbers/Number_Functions/Number_Functions.htm>. Luettu 1.4.2022.

45 Auto Tiles. Verkkoaineisto. GameMaker Manual. <https://manual.yoyogames.com/The_Asset_Editors/Tile_Set_Editors/Auto_Tiles.htm>. Luettu 20.3.2022.

46 Driver, Sam. 2010. A Bitwise Method For Applying Tilemaps. Verkkoaineisto. <https://web.archive.org/web/20150906102436/http:/www.saltgames.com/2010/a-bitwise-method-for-applying-tilemaps>. 26.1.2010. Luettu 20.3.2022.

47 Bone, Sonny. 2016. How to Use Tile Bitmasking to Auto-Tile Your Level Layouts. Verkkoaineisto. <https://gamedevelopment.tutsplus.com/tutorials/how-to-use-tile-bitmasking-to-auto-tile-your-level-layouts--cms-25673>. 3.2.2016. Luettu 20.3.2022.

48 Michael, David. 2000. Tile/Map-Based Game Techniques: Handling Terrain Transitions. Verkkoaineisto. <https://www.gamedev.net/tutorials/_/technical/game-programming/tilemap-based-game-techniques-handling-terrai-r934>. 23.2.2000. Luettu 20.3.2022.

49 Warren, Tom. 2020. Minecraft till incredibly popular as sales top 200 million and 126 million play monthly. Verkkoaineisto. <https://www.vg247.com/terraria-35-million-sales-lifetime>. 18.5.2020. Luettu 10.4.2022.

50 Peppiatt, Dom. 2021. Terraria has sold over 35 million copies in Its lifetime. Verkkoaineisto. <https://www.vg247.com/terraria-35-million-sales-lifetime>. 21.3.2021. Luettu 10.4.2022.

51 Factorio has sold over 3 million copies. 2022. Verkkoaineisto. Aroged. <https://www.aroged.com/2022/02/26/factorio-has-sold-over-3-million-copies>. 26.2.2022. Luettu 10.4.2022.