Tampereen yliopisto

Juho Karhunen

# KAKSIDIMENSIONAALISTEN LUOLASTOJEN GENEROINTI PROSEDURAALISESTI C++ KIELELLÄ

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Pia Niemelä
5/2021

# TIIVISTELMÄ

Juho Karhunen: Kaksidimensionaalisten luolastojen generointi proseduraalisesti C++ kielellä
Procedural generation of two-dimensional caves with C++ language
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Kandidaatintutkielma
Toukokuu 2021

---

Proseduraalinen generointi on tapa luoda tietoa algoritmin avulla. Proseduraalisella generoinnilla on monia käyttötarkoituksia. Yleisimmin sitä käytetään sisällön, esimerkiksi karttojen, tekstuurien tai äänien tuottamiseen tietokonepeleissä. Proseduraalisella generoinnilla on myös mahdollista säästää muistitilaa antamalla käyttäjälle vain lähtöarvot ja algoritmi. Tällöin generoinnin tulosta ei tarvitse tallentaa muistiin.

Tässä työssä tutkitaan parametrien ja reunaehtojen vaikutusta proseduraalisessa generoinnissa ja pyritään muodostamaan kattava kokonaiskuva proseduraalisen generoinnin toimintaperiaatteista. Proseduraalisessa generoinnissa annetaan algoritmille jokin alkuarvo eli siemenluku ja saatu lopputulos on aina sama, kun siemenluku on sama. Proseduraalisen generoinnin menetelmiä on useita. Tämä työ keskittyy pääasiassa kaksidimensionaaliseen proseduraaliseen generointiin.

Työssä on kaksi osaa. Ensimmäisessä osassa tehtiin proseduraaliseen generointiin kirjallisuuskatsaus, jossa tutkittiin yleisimpiä proseduraalisen generoinnin käyttötarkoituksia ja menetelmiä. Lisäksi selvitettiin proseduraalisen generoinnin hyviä ja huonoja puolia sekä sitä, mikä tekee pelikartasta hyvän. Proseduraalisella generoinnilla on mahdollista säästää pelin valmistukseen kuluvaa aikaa ja lisätä pelin uudelleenpelattavuutta, sillä kartta generoituu erilaiseksi, kun siemenluku on eri. Toisaalta liian yksinkertaisella algoritmilla lopputulos voi tuntua pelaajasta itseään toistavalta. Lisäksi kaikkia mahdollisia generoinnin lopputuloksia on mahdotonta testata niiden lähes äärettömän määrän vuoksi.

Työn toisessa osassa tutkittiin proseduraalista generointia kirjoitetun ohjelman avulla. Ohjelmalla voidaan proseduraalisesti generoida kaksidimensionaalinen luolasto. Ohjelma tehtiin soluautomaattimallilla, ja sen sisällä voi muuttaa generoinnin parametrejä ja tarkastella niiden vaikutusta generoituun tulokseen. Tutkimuksesta esitellään prosessi, jossa luolasto generoituu alkuarvoista, sekä kuvia eri parametreillä tehdyistä generoinneista.

Tutkimus osoittaa, että pelimekaniikkojen suunnitteleminen ennen proseduraalisen generaattorin tekemistä on tärkeää, sillä parametrit vaikuttavat generoituun karttaan vahvasti. Pelikokemus riippuu esimerkiksi siitä, onko kartta mahdollista läpäistä. Lopuksi työssä esitellään kehitysideoita kirjoitettuun ohjelmaan.

# SISÄLLYSLUETTELO

# KUVALUETTELO

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| C++ | C++ -ohjelmointikieli |
| CImg | The CImg Library -kirjasto C++ kielelle |
| P | Mustien pikseleiden prosentuaalinen määrä alussa. |
| RGB | RGB-värimalli, jossa värit muodostetaan sekoittamalla punaista (red), vihreää (green) ja sinistä (blue). |
| X | Maksimimäärä mustia pikseleitä valkoisen pikselin naapurina, joka ylitettäessä valkoinen pikseli muuttuu mustaksi. |
| Y | Mustien pikseleiden ylipopulaation raja. |

# 1. JOHDANTO

Nykyään peliteollisuuden haasteena on pitää kustannukset kurissa. Peleistä tulee koko ajan suurempia sekä monimutkaisempia. Iso osa pelituotannon hinnasta koostuu karttojen, pelaajamallien sekä tekstuurien luomisesta. Tästä syystä peliteollisuus hyödyntää nykyään proseduraalista generointia automaattisen sisällön luomiseksi. (Frade et al. 2012; Antoniuk, Rokita 2016) Peliteollisuus on nopeasti kasvava ala, ja pelin valmistuksessa säästetty aika ja raha voi nostaa yrityksen sekä pelin kilpailijoiden edelle. Monet maineikkaat pelit, kuten Minecraft, Terraria ja No Man's Sky käyttävät proseduraalista generointia hyväkseen. No Man's sky tarjoaa proseduraalisesti generoidun universumin, jossa on yli $1.8 \times 10^{19}$ planeettaa omilla uniikeilla ominaisuuksillaan (Smed, Hakonen 2017). Proseduraalisella generoinnilla on kuitenkin ongelmansa, sillä on mahdotonta suunnitella ja testata pelikokemusta kokonaisuudessaan (Ripamonti et al. 2017).

Varhaisena esimerkkinä ohjatusta satunnaisuudesta voidaan pitää Dungeons and Dragons -lautapelin pelikentän luontia. Advanced Dungeons and Dragons: Players Handbook -ohjeissa opetetaan generoimaan kokonaisia vankiloita/luolastoja heittämällä noppaa, ja jokainen nopan numero vastaa tiettyä asentoa ja huonetyyppiä (Gygax 1978). Proseduraalisesti generoitua sisältöä voi siis luoda myös ihminen, jos hän seuraa tiettyjä selkeästi määriteltyjä menetelmiä (Smith 2015).

Luvussa kaksi esitellään proseduraalista generointia yleisesti. Luvussa kolme tarkastellaan proseduraalisen generoinnin hyviä sekä huonoja puolia. Luvussa neljä tutkitaan hyvän pelikartan ominaisuuksia, sekä sitä miten reunaehdoilla ja parametreillä voidaan vaikuttaa generoinnin laatuun. Luvussa viisi toteutetaan pienimuotoinen proseduraalinen generaattori karttojen luomista varten. Lopuksi luvussa kuusi käydään läpi johtopäätöksiä.

# 2. PROSEDURAALINEN GENEROINTI YLEISESTI

Proseduraalinen generointi viittaa sisällön tuottamiseen algoritmien avulla, joko käyttäjän avustamana tai ilman avustusta (Smed, Hakonen 2017). Proseduraalinen generointi toisin sanoen on tiettyjen menetelmien eli proseduurien seuraamista ja suorittamista (Watkins 2016). Proseduraalinen generointi mahdollistaa automaattisen sisällön tuottamisen. On esimerkiksi mahdollista generoida metsää, jossa jokainen kasvilaji on esitetty tiettyinä parametreinä ja jokainen puu on hieman erilainen, vain muuttamalla generoinnin alustavaa siemenlukua. Proseduraalisella generoinnilla on mahdollista luoda esimerkiksi karttoja, tekstuureja, animaatioita ja ääniä (Green 2016; Watkins 2016). Proseduraalisia sisällön luonnin metodeja on kuitenkin vaikea valmistaa, ja parametrien valinta niihin on vaikeaa. (Frade et al. 2012)

## 2.1 Satunnaisuus ja näennäissatunnaisuus

Satunnaisuus on jokin täysin ennustamaton tapahtuma, jolla ei ole mitään sääntöjä. Esimerkiksi heitetyn nopan silmäluku on satunnainen jokaisella heitolla. Yleensä puhuttaessa proseduraalisesta generoinnista sillä tarkoitetaan proseduraalista generointia, joka käyttää hyväkseen satunnaisuutta (Green 2016).

Tietokoneet eivät voi luoda täyttä satunnaisuutta, sillä ne ovat deterministisiä. Kun syötetyt arvot ja algoritmit ovat samoja, tulee vastauksesta aina sama, eli käytännössä tietokoneet ovat vain hienompia taskulaskimia (Green 2016). Näennäissatunnaisuus on vain tietokoneen luomia arvoja, jotka eivät todellisuudessa ole satunnaisia, vaan ne seuraavat jotain tiettyä kaavaa tai algoritmia. Näennäissatunnaisia numeroita käytetään ohjelmoinnissa, koska niitä on helpompi generoida. (Watkins 2016)

## 2.2 Siemenluku

Proseduraalisessa generoinnissa käytettävä näennäissatunnaisuus voidaan saavuttaa esimerkiksi käyttämällä tietokoneen kellonaikaa lähtöarvona eli siemenlukuna. Siemenluku antaa algoritmille lähtöpaikan. Vaikka algoritmi olisi todella monimutkainen, generoiminen tietyllä siemenluvulla johtaa joka kerta samaan lopputulokseen. (Green 2016)

Pelit, kuten Minecraft, käyttävät siemenlukuja generoidessaan maailmoja. Mikäli kaksi henkilöä haluaa aloittaa pelin täysin samannäköisestä maailmasta, he voivat generoida maailmansa samalla siemenluvulla. Algoritmi luo tällöin aloitusmaailmoista täysin samanlaiset (Green 2016).

## 2.3 Karttojen luominen

Maastot ovat pelien osia, joiden generointi voidaan tehdä automaattisesti. Tämä lisää pelin uudelleenpelattavuutta (Frade et al. 2012). Joka kerta kun siemenlukua muutetaan, tulee maastosta erilainen. Proseduraalisten generaattorien on tärkeää varmistaa, että niiden luoma sisältö on vähintäänkin pelattavaa, ja usein tämä vähimmäisvaatimus on riittävä.

Pelattavuus riippuu usein modulaarisuudesta. Jokaisen palasen, josta kartta muodostuu, tulisi sopia keskenään yhteen. (Smith 2015) Tästä hyvä esimerkki on Spelunky-pelin karttoja generoiva järjestelmä (Kazemi; Smed, Hakonen 2017). Järjestelmä valitsee etukäteen hyväksytyistä palasista toisiinsa sopivat ja asettelee ne yhteen. Seuraavalla läpimenokerralla lisätään viholliset ja aarteet sääntöjen sallimiin paikkoihin. Tämän mahdollistaa hyvin suunnitellut rakennuspalat. (Smith 2015)

Luotaessa kolmedimensionaalisia tasoja proseduraalisella generoinnilla käytetään yleensä apuna melukarttoja, kuten Perlin-melua. Melukartan ominaisuuksille, kuten intensiteetille, varianssille ja konsentraatiolle, on monia sovelluksia. Maastoa luotaessa Perlin-melun ominaisuuksia voidaan käyttää esimerkiksi korkeuden määrittämiseen. (Green 2016; Panagiotou et al. 2020) Kuvassa 1 tämä näyttäytyy siten, että korkeamman intensiteetin kohdat ovat korkeammalla kolmedimensionaalisessa kuvassa.


***Kuva 1.*** *Kolmedimensionaalisen maailman generointi Perlin-melun avulla (Panagiotou et al. 2020)*

## 2.4 Tekstuurien luominen

Tekstuurien luominen proseduraalisella generoinnilla tarkoittaa, että tekstuureja voi käytännössä olla loputon määrä ilman, että ne vievät muistissa tilaa (Green 2016). Voitaisiin esimerkiksi luoda seinätekstuuriin likaa niin, että jokainen kohta seinästä on erilainen. Jokaista generoitua seinätekstuuria ei kuitenkaan tarvitse tallentaa erikseen muistiin. Tämä vähentää suunnittelijoiden taakkaa luoda uniikkia sisältöä peliin (Watkins 2016).

## 2.5 Muistitilan säästäminen

Nykyaikana jopa heikoimmissa tietokoneissa on muistia satoja gigatavuja eli pelien tallentaminen tietokoneen muistiin ei ole ongelma. Kun muistin rajallisuus vielä rajoitti peliyhtiöitä, proseduraalinen generointi oli yksi tapa saada paljon sisältöä mahtumaan pieneen tilaan (Green 2016). Vuonna 1983 julkaistun Nintendo Entertainment System-konsolin (NES) pelit olivat yleisimmin vain 128–384 kilotavun kokoisia (Nintendo Company history).

Suurien tasojen rakentaminen ja tallentaminen vei paljon tilaa ja niiden mahduttaminen muistiin oli mahdotonta. Pelit sisälsivät vain tarvittavat resurssit ja ohjelma kasasi ne pelaajan käynnistäessä pelin tai tason. (Green 2016; Watkins 2016) Esimerkiksi 1980-luvulla julkaistussa Rogue-pelissä tätä tapaa säästää tilaa käytettiin hyväksi (Watkins 2016).

# 3. PROSEDURAALISEN GENEROINNIN HYVÄT JA HUONOT PUOLET

Proseduraalinen sisällön luonti applikaatioihin kuten tietokonepeleihin on suosittu aihealue, joka tarjoaa paljon haasteita. Pelien valmistaminen sisältää paljon muutakin kuin ohjelmoinnin. Aluksi on tehtävä pelisuunnittelua, sen jälkeen pelistä riippuen on valmistettava hahmot, kartat, tasot, aseet, tekstuurit, äänet, tehtävät, esineet, oliot ja paljon muuta. Kaiken tämän luominen vaatii paljon suuremman panostuksen ja budjetin kuin itse pelin ohjelmoiminen. (Togelius et al. 2011)

## 3.1 Proseduraalisen generoinnin edut

Proseduraalisen generoinnin avulla pystytään luomaan suurempia pelimaailmoja, sillä jokaista asiaa tai esinettä ei tarvitse asettaa manuaalisesti. Jopa massiivisimmat käsin luodut maailmat, jotka löytyvät The Witcher 3: Wild Hunt ja Grand Theft Auto V -peleistä, häviävät sille, mitä proseduraalisella generoinnilla voidaan tehdä (Green 2016).

Pelien tekeminen on kallista ja suurimpien peliprojektien teko maksaa kymmenistä tuhansista jopa miljooniin dollareihin. Mallintajien tarvitsee tehdä vain muutamia mallinnuksia ja proseduraalista generointia käyttämällä voidaan luoda loput pelin tarvitsemat resurssit. (Green 2016)

Pelimaailmat, jotka luodaan käsin, takaavat sen, että jokaisen pelaajan kokemus on samanlainen. Proseduraalisesti generoiduilla peleillä on kuitenkin se ominaisuus, että jokaisella pelikerralla kokemus on erilainen ja pelaaja kohtaa jotain odottamatonta (Green 2016; Garzia 2020). Tämän takia proseduraalisesti generoidut pelit tarjoavat uudelleenpelattavuutta, jota käsintehdyt maailmat eivät pysty tarjoamaan.

## 3.2 Proseduraalisen generoinnin haitat

Algoritmit, joita proseduraalinen generointi käyttää, voivat olla haastavia sekä vaatia paljon laskentatehoa. Jos jokainen puu tietyssä pelimaailmassa generoidaan proseduraalisesti, on tämä prosessorille ja näytönohjaimelle raskasta ja voi aiheuttaa peliin nykimistä. (Green 2016) On siis mahdollista, että pelaajan tietokone tai pelikonsoli ei ehdi tekemään näitä laskentoja ilman, että pelikokemus kärsii.

Generoitu sisältö on yleensä hienoa, mutta lopputuotteen muodon hallitseminen saattaa olla haastavaa (Antoniuk, Rokita 2016). Kun luodaan iso maailma proseduraalisen generoinnin avulla, voi se tuntua itseään toistavalta. Kun suuren maailman luomiseen

käytetään vain muutamia yksinkertaisia algoritmeja, on toistuvuus helposti huomattavissa ja tämä heikentää pelaajan pelikokemusta (Green 2016).

Kun peli luodaan proseduraalisella generoinnilla, menetetään ihmisen luovuus. Suunnittelijan tekemät pienet muutokset ja yksityiskohdat jäävät puuttumaan. On siis mahdollista, että toiselle pelaajalle generoituu toimiva maailma, joka edistää pelattavuutta hyvin, ja toiselle maailma, joka haittaa sitä. (Green 2016)

Generoituja maailmoja on lähes ääretön määrä ja tästä syystä kaikkien erilaisten mahdollisten lopputulosten testaaminen on mahdotonta. On siis täysin mahdollista, että proseduraalisesti generoitua tasoa on mahdoton läpäistä. Esimerkiksi jokin tarvittava esine generoituu saavuttamattomaan paikkaan tai maali on liian korkealla pelaajan ulottumattomissa. Tällaisista asioista muodostuu pelaajalle huono kokemus.

# 4. HYVÄN KAKSIDIMENSIONAALISEN PELIKARTAN OMINAISUUDET

Hyvä tasosuunnittelu pyrkii tuottamaan laadukasta pelattavuutta, mukaansatempaavaa kokemusta, sekä tarinaan pohjautuvissa peleissä viemään tarinaa eteenpäin (Ripamonti et al. 2017). Pelikartalla tulisi olla asioita, joita pelaaja voi löytää, ja vaaroja, joita pelaajan tulisi välttää. Pelikartalla tulisi olla myös jokin logiikka niin, että pelaaja voi löytää seuraavan alueen ilman ongelmia (Moore 2011). Voidaan tulla siihen johtopäätökseen, että jokaisella tasolla tulisi olla maali tai jokin tavoite. Pelaaja ei myöskään saa jäädä jumiin, eikä tason suoritus saa olla mahdoton (Garzia 2020).

Proseduraalisessa generoinnissa varmistetaan, että pelikartasta tulee halutunlainen valitsemalla reunaehdot ja parametrit. Seuraavassa luvussa kerrotaan tarkemmin näistä reunaehdoista ja parametreista sekä niiden soveltamisesta.

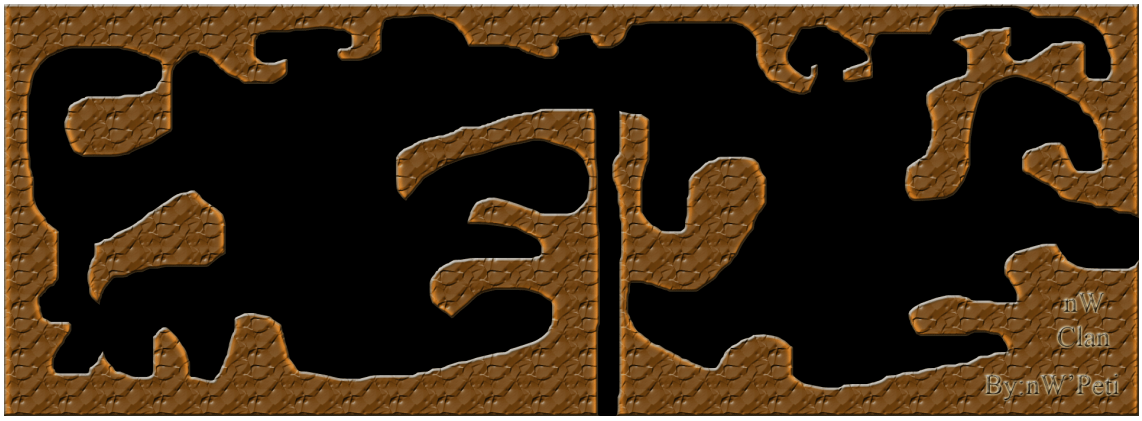
## 4.1 Reunaehdot ja parametrit

Reunaehdot proseduraalisessa generoinnissa ovat sääntöjä, joiden mukaan asiat generoituvat. Reunaehto voi määritellä esimerkiksi sen, että kaksi puuta eivät saa generoitua liian lähelle toisiaan, tai puun on aina generoiduttava maan päälle. Nämä reunaehdot määrittävät tuloksista järkeviä ja haluttuja.

Parametrit proseduraalisessa generoinnissa ovat lähtöarvoja, jotka algoritmeille asetetaan generointia varten. Kun proseduraalista generaattoria tehdään, on parhaiden parametrien löytäminen tärkeää, jotta saadaan generoitua hyviä karttoja. Jokaisen parametrin valinta on toisaalta uniikki sen sovellukselle, eli sille mitä kyseinen peli vaatii ja mitä pelin mekaniikat mahdollistavat. Esimerkiksi metsää luotaessa voidaan parametriksi valita puiden lukumäärä. Jos tämä parametri on liian pieni, tulee metsästä harva.

# 5. TESTIOHJELMAN TOTEUTUS

Työn tavoitteena oli toteuttaa pienimuotoinen kaksidimensionaalisia luolastoja luova proseduraalinen generaattori. Ohjelmalla voidaan kokeilla eri parametrien ja reunaehtojen muuttamista ja sitä, miten ne vaikuttavat karttaan. Tarkoituksena oli valita parametrit, joita muuttamalla generoinneissa voitiin nähdä eroavaisuuksia. Kuvassa 2 nähdään luolasto, joka on tehty Worms-tietokonepelille.




***Kuva 2.*** *Worms-pelikartta (Peti 2010)*

Kuvassa oleva kartta on todennäköisesti tehty käsin. Testiohjelman tavoitteena on näyttää, että voidaan luoda samankaltainen luolasto, joka on eri siemenluvuilla generoituna erilainen, mutta silti pätevä pelikäyttöön.

## 5.1 Kirjastot

Työkaluksi luolastojen esittämiseen valittiin The CImg Library, sillä se mahdollisti luolastojen esittämisen ja arvojen muuttamisen helposti (Tschumperlé). CImg oli myös hyvä työkalu, sillä sen käyttämisen edellytyksenä oli ainoastaan paketin lataaminen ja sen kopioiminen ohjelman otsikkotiedostoksi. CImg käsittelee kuvat RGB-värimallin mukaan, joten tämä mahdollisti myös muiden kuin valkoisen ja mustan värin käyttämisen.

Näennäissatunnaisten numeroiden generointiin valittiin C Standard General Utilities kirjaston funktio srand (C Standard General Utilities Library). Srand on funktio, joka luo täysin samat näennäissatunnaiset arvot, kun sille annettu siemenluku on sama. Tämä on hyödyllistä siksi, että voidaan verrata samojen lähtöarvojen generoinnin tuloksia eri parametreillä. Srand-funktio alustetaan yleensä muuttujalla, kuten tietokoneen kellonajalla, jonka mukaan sen näennäissatunnaiset numerot generoituvat. Siemenlukuna

käytettiin käyttäjän syöttämää numeroa tai C Time Libraryn time-funktiota, jolla pystytään hakemaan tietokoneen aikaleima (C Time Library).

## 5.2 Valittu algoritmi

Proseduraaliseen generointiin voidaan käyttää useita erilaisia algoritmeja ja malleja. Eräs luotettava ja tehokas tapa generoida proseduraalisesti karttoja on soluautomaatti. Soluautomaatilla on kyky organisoida itsensä. (Johnson et al. 2010) Soluautomaatti on tapa mallintaa ja simuloida monimutkaisia järjestelmiä tieteissä, tekniikassa ja taiteessa. Se muodostuu ruudukosta soluja, joilla jokaisella on rajallinen määrä eri tiloja. (Zhang, Sarjoughian 2017) Ohjelmassa eriväriset pikselit kuvaavat solujen tiloja. Ohjelman generoimissa luolastoissa tilat ovat 0 eli musta pikseli sekä 1 eli valkoinen pikseli. Mustat solut kuvaavat luolastossa kiveä ja valkoiset solut avointa luolaa. Luolastoon lisätään myös myöhemmässä vaiheessa punaisia soluja eli vihollisia ja keltaisia soluja eli kultaa.

Jokaisella solulla on joukko muita soluja, jotka määrittelevät sen naapuruston. Mooren naapurustoksi kutsutaan rakennetta, jossa rakenne muodostuu solun kahdeksasta ympäröivästä solusta. Solujen tiloja muutetaan vaiheittain, ja uusi tila riippuu solun omasta tilasta sekä naapurisolujen tiloista. (Zhang, Sarjoughian 2017). Ohjelmassa solujen arvoihin vaikuttavat reunaehdot ja parametrit käsitellään luvussa 5.3.

Soluautomaatin heikko puoli on kuitenkin se, että se soveltuu vain kaksidimensionaalisten karttojen luomiseen sekä se, että generoinnin tuottamaa tulosta ei voida suoraan hallita. Esimerkiksi kahden generoidun huoneen yhdistymistä ei voida taata pelkästään tällä algoritmilla (van der Linden et al. 2014). Tämä algoritmi valittiin kuitenkin, koska sillä on kyky luoda työn vaatimia luolastoja. Sen avulla on myös helppoa säätää parametrejä ja nähdä niiden vaikutus generoituun luolaan.

Soluautomaattimallilla on mahdotonta luoda kartta, jolla on tietyt vaatimukset, kuten montako huonetta malli generoitiin luo. Tästä syystä parhaiten pelimekaniikkoihin soveltuva malli on etsittävä kokeilemalla eri parametrejä. (van der Linden et al. 2014)

## 5.3 Koodin oleelliset ominaisuudet

Tässä luvussa esitellään generoidun lopputuloksen kannalta oleellisimpia osia kirjoitetusta ohjelmakoodista. Ohjelma 1 -koodissa nähdään, että ohjelman käynnistyessä valitun kokoinen ruudukko muodostuu rivillä 2. Tämän jälkeen näennäissatunnaiset luvut alustetaan käyttämällä joko tiettyä valittua arvoa tai tietokoneen aikaa. Tämän jälkeen

ruudukko alustetaan valituilla prosenteilla kiveä ja avointa luolaa. Luvun 5.4 kuvassa 4 (A) nähdään alustettu ruudukko.

```
    // Initialize original empty image
2   CImg <unsigned char> cave(50, 50, 1, 3, 0);

4   // srand(seed); // Used in case of wanting a certain seed
    srand((unsigned) time(NULL));
6
    // Initialize the first cave with pseudo-random black and white
8   // pixels with the system time used as the seed.
    for(int x = 0; x < cave.height(); x++){
10      for(int y = 0; y < cave.width(); y++){

12          float number = (float) rand()/RAND_MAX;

14          if(number >= percentageOfOpenCave)
                cave(x,y) = 255;
16
            else
18              cave(x,y) = 0;
        }
20  }
```

***Ohjelma 1.*** *Ruudukon alustaminen näennäissatunnaisilla arvoilla.*

Luolastoa aletaan tämän jälkeen iteroimaan niin, että jokainen pikseli käydään läpi yksitellen ja lasketaan sille uusi arvo. Tämä nähdään ohjelma 2 koodissa. Uusi arvo perustuu tiettyihin reunaehtoihin. (van der Linden et al. 2014)

- Jos pikseli on avointa luolaa ja naapurina on yli X kiveä, niin pikseli muuttuu itse kiveksi.
- Jos pikseli on avointa luolaa ja naapurina on alle X kiveä, niin pikseli pysyy ennallaan.
- Jos pikseli on kiveä ja naapurina on yli Y kiveä, niin pikseli muuttuu avoimeksi luolaksi, ylipopulaation takia.
- Jos pikseli on kiveä ja naapurina on alle Y kiveä, niin pikseli pysyy ennallaan.

Näitä parametreja on mahdollista säätää, ja tuloksien vertailua eri parametreillä nähdään luvussa 5.6.

```
CImg<unsigned char> makeCaves(CImg<unsigned char> &cave, int stoneTo-
Cave, int caveToStone)
{
    // Calculate neighbour values for each pixel and change their
    // value based on the rules.
    CImg<unsigned char> newCave = cave;
    for(int x = 0; x<cave.width(); x++){
        for(int y = 0; y < cave.height(); y++){
            int numberOfNeighbours = countNeighbours(cave, x, y);

            if(cave(x,y,0)==255){
                if(numberOfNeighbours<caveToStone)
                    newCave(x,y) = 0;
                else
                    newCave(x,y) = 255;
            }

            else{
                if(numberOfNeighbours>stoneToCave)
                    newCave(x,y) = 255;
                else
                    newCave(x,y) = 0;
            }
        }
    }
    return newCave;
}

```

***Ohjelma 2.*** *Iteraatioprosessi, jossa luolasto muodostuu*

Iteroinneissa saadut uudet pikselien arvot sijoitetaan toiseen samankokoiseen ruudukkoon, sillä ei haluta, että jo muuttuneet pikseleiden arvot vaikuttavat sillä hetkellä pyörivään iteraatioon. Pikselit käydään läpi ylävasemmalta alaoikeaan reunaan ja kuvasta 3 nähdään, miten uudet lasketut arvot vaikuttaisivat sillä hetkellä laskettavan arvon naapureiden määrään, jos sijoitus tehtäisiin samaan ruudukkoon.


***Kuva 3.*** *Uusien arvojen vaikutus laskettavaan arvoon*

Kun käyttäjän valitsema määrä iteraatiokierroksia on tehty luolastolle, sille tehdään tarkastus, jossa poistetaan kaikki pikselit, jotka ovat ympäröity kokonaan toisella solun arvolla. Tarkastus tehdään, jotta luolastoon ei jäisi yksittäisen pikselin kokoisia kiviä leijumaan eikä luolastossa olisi turhia reikiä.

Tämän jälkeen lisätään luolastoon vihollisia sekä aarteita. Nämä aarteet sekä viholliset lisätään myös proseduraalisesti käymällä jokainen pikseli läpi ja tarkastelemalla kyseisen pikselin ja sen naapureiden arvoja. Jos sopiva kohta löydetään, siihen lisätään vihollinen tai aarre.

Vihollisten lisäämisen reunaehdot ovat:

- Pikselin alapuolella on kiveä, sekä yläpuolella avointa luolaa.
- Viereisellä pikselillä ei ole jo toista vihollista.
- Naapurina on vähintään kaksi kiveä.
- Valitun prosentin todennäköisyydellä lisätään tähän kohtaan vihollinen.

Aarteiden lisäämisen reunaehdot ovat:

- Aarre on kokonaan ympäröity kivellä.
- Valitun prosentin todennäköisyydellä lisätään tähän kohtaan aarre.

Reunaehdot ja parametrit valittiin niin, että löydettiin hyvä jako aarteita ja vihollisia. Tämän vaiheen tarkoitus ei ollut arvioida parhaita parametrejä, vaan esittää, mitä proseduraalisesti voidaan tehdä. Näiden lisäämisoperaatioiden valmistumisen jälkeen luolasto olisi valmis käytettäväksi pelissä. On kuitenkin otettava huomioon, että työssä ei ole tehty pelimekaniikkoja.

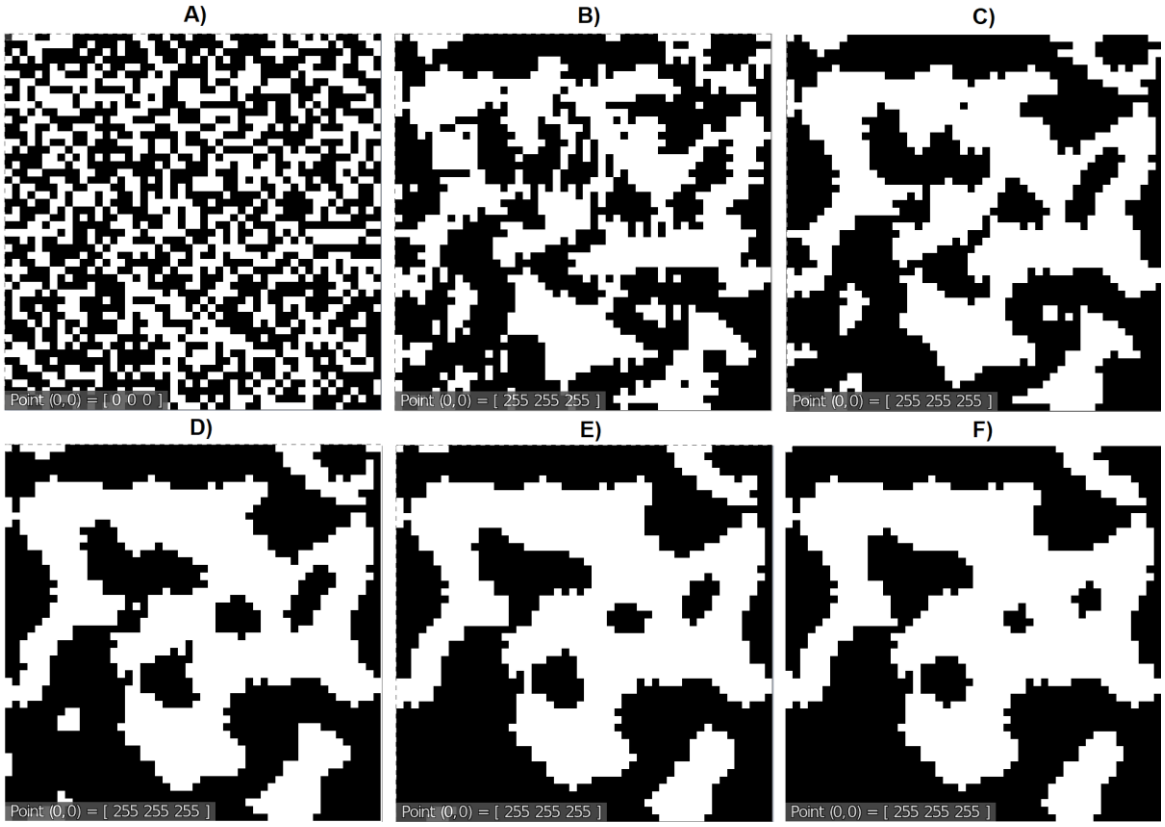
## 5.4 Generoinnin tulokset

Kuvasta 4 nähdään, että luolastoja muodostuu ohjelmassa hyvin. Parametrien arvot, joilla kuvien generointi tehtiin:

- kivien määrä alussa 55 %
- raja milloin avoin luola muuttuu kiveksi 4
- kivien ylipopulaation raja 5
- iteraatioiden määrä 6.

Tässä vaiheessa parametreina käytettiin arvoja, joilla saatiin ohjelman kirjoitusvaiheessa haarukoitua ensimmäinen kuvan 2 luolastoa muistuttava lopputulos. Testiohjelmalle ei ole asetettu pelimekaniikkojen kannalta mitään erityisvaatimuksia, eli esimerkiksi kuinka korkealle pelihahmo pystyy hyppäämään. Kuvaan 2 verrattaessa nähdään, että kuvassa 4 generoitu luolasto on halutun näköinen.

Lisäksi havaitaan tässä vaiheessa jo luvussa 3.2 mainittu ongelma: kaikkia mahdollisia kombinaatioita ei voida kokeilla ja kuvan 4 (D) iteraation keskelle kiven sisälle on muodostunut yksittäinen avoimen luolan pikseli. Tässä nähdään proseduraalisen generoinnin heikon puolen, eli täydellistä pelaajakokemusta ei voida taata, koska jokaista mahdollista kombinaatiota ei pystytä käymään läpi. Tähän ongelmaan tehtiin kuitenkin tarkastus ja iterointien jälkeen muutettiin kaikki ne pikselit, jotka olivat ympäröityjä toisen värisillä pikseleillä.




***Kuva 4.*** *Luolaston muodostuminen iteraatioprosessissa. Iteraatioiden määrä aakkosjärjestyksessä: 0, 2, 4, 6, 8, 10 iteraatiota.*

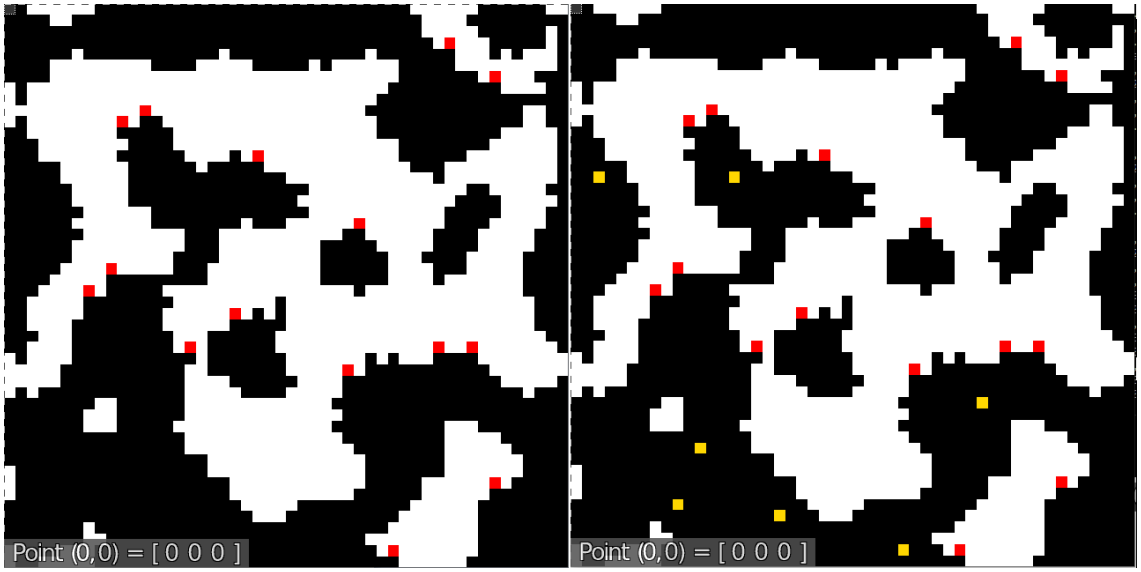
Kuvasta 4 huomataan, että luolasto määrittyy hieman jokaisella iteraatiokerralla. Tässä tapauksessa kuudella iteraatiolla saadaan kuitenkin tulos, joka on lähellä haluttua, pienimmällä mahdollisella määrällä iteraatioita.

Proseduraalinen generointi, joka tekee laskennat pelin pyörimisen aikana, asettaa generaattorin teholle tiukat vaatimukset. 60 hertsin kuvataajuudella pyörivässä pelissä on

generaattorilla vain 16 millisekuntia tehdä tekoälyn, fysiikkamoottorin ja renderöinnin laskennat. (Grendel Games 2018) On siis valittava sopiva määrä iteraatioita menettämättä generaattorin tehoa liikaa. Yksinkertaisellekin testiohjelmalle kuuden iteraation ajoaika oli kolme millisekuntia. Tästä voidaan päätellä, että tehokkuuden optimointi ja nopeuden säilyttäminen monimutkaisemmille generaattoreille voi olla haasteellista.

## 5.5 Vihollisten ja aarteiden lisääminen

Vihollisten ja aarteiden lisääminen tehtiin siksi, että voitiin monipuolisemmin esitellä, mihin proseduraalinen generointi pystyy. Käytännössä kaikki mitä pelimaailmasta löytyy, voitaisiin generoida proseduraalisesti. Kuten kuvasta 5 huomataan, viholliset tulevat valittujen reunaehtojen mukaan järkeviin paikkoihin. Viholliset siis löytyvät avoimilta alueilta, eivätkä synny kohtiin, jossa ne tukkivat luolan käytävän kokonaan.



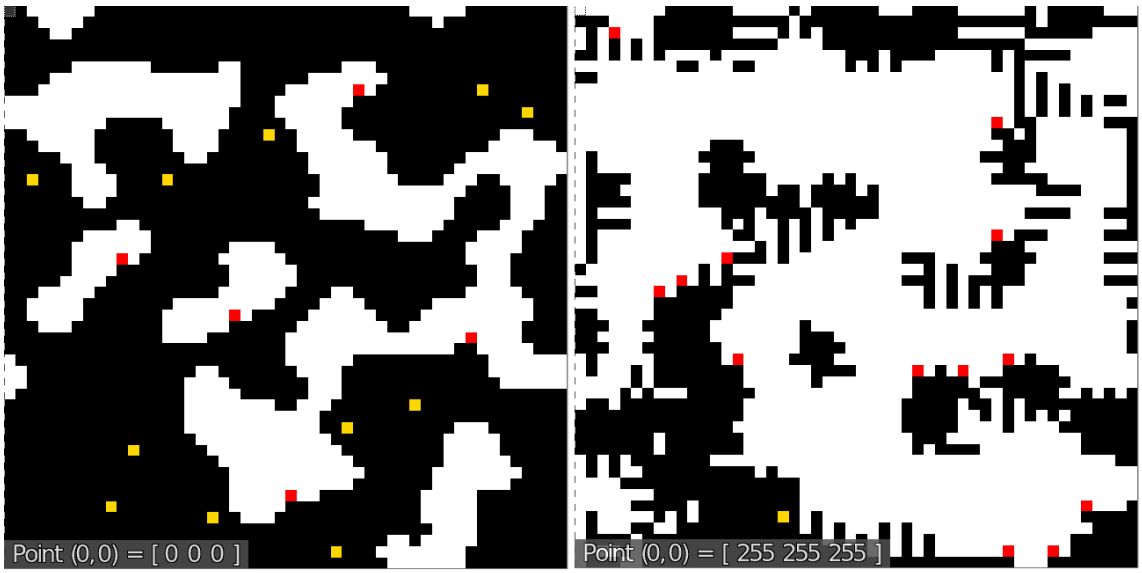

***Kuva 5.*** *Vihollisten ja aarteiden lisääminen luolastoon*

Tässäkin huomattiin pieni ongelma, sillä joissakin tilanteissa vihollinen generoitui leijumaan ilmaan ja reunaehtoihin piti tehdä muutoksia. Aarteita luova algoritmi onnistui hyvin, havaittiin, että noin prosentin todennäköisyydellä aarteen luominen paikkaan, jossa se on ympäröity kivellä, oli järkevää ja näin saatiin kartalle sopiva määrä aarteita. Näillä aarteiden luomisen reunaehdoilla vaaditaan kuitenkin, että pelihahmo pystyy kaivamaan kiveä. Nämäkin satunnaisuudet ovat sidottuja ohjelmassa siemenluvulla muodostettuun näennäissatunnaisuuteen, eli aarteet ja viholliset löytyvät samasta paikasta saman siemenluvun generoinnin luolastosta jokaisella kerralla.

## 5.6 Parametrien vaikutus generointiin

Parametrien vaikutusta generoituun tulokseen tutkittiin käyttämällä samaa siemenlukua ja parametrejä kuin aiempien generointien kuvissa. Parametreja muutettiin yksi kerrallaan, jotta niiden vaikutus lopputulokseen voitiin havaita. Generoitunutta tulosta voi verrata kuvaan 5.

Ensimmäiseksi muutettiin kivien ylipopulaation rajaa. Kuvasta 6 huomataan, että kun kivien ylipopulaation raja on kolme, syntyy luolastoon kapeampia käytäviä. Kun raja on viisi, alkaa luolastoon muodostumaan ei-toivottua raidallista rakennetta.

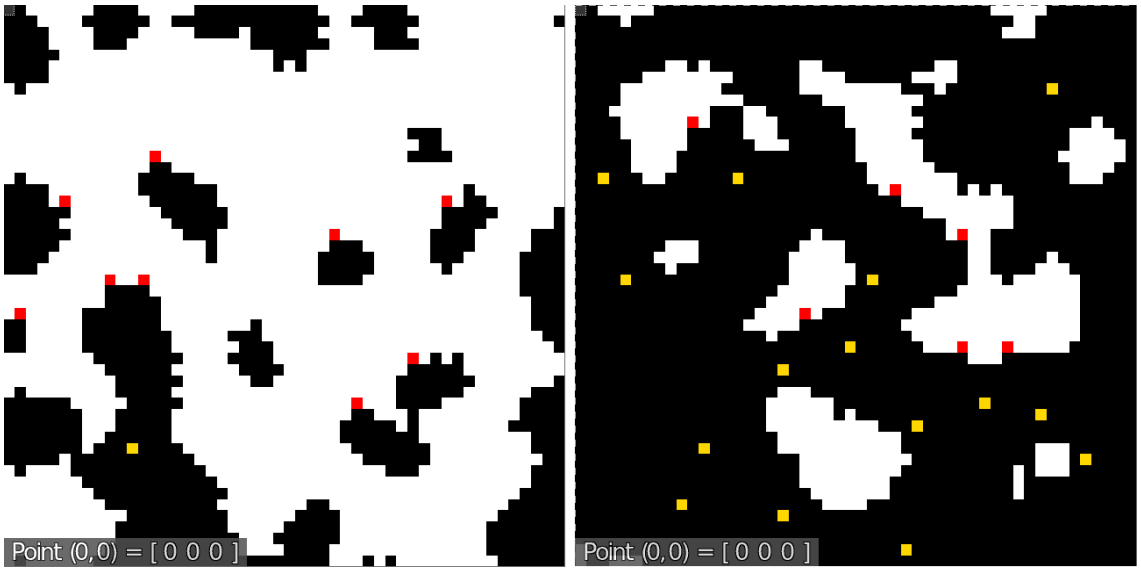



***Kuva 6.*** *Kiven ylipopulaation raja-arvo, vasemmalla Y = 3 ja oikealla Y = 5*

Tarkasteltaessa rajaa, jolloin avoimen luolan naapureina on liikaa kiviä ja se itse muuttuu kiveksi, huomataan kuvasta 7, että arvolla neljä luolastoon muodostuu samanlaista ei-toivottua raidallista kuviota, kuin edellisessä vaiheessa. Kun arvo nostetaan kuuteen, tulee luolasta avointa ja tavoitteenmukaista.


***Kuva 7.*** *Naapurikivien määrä, vasemmalla X = 4 ja oikealla X = 6*

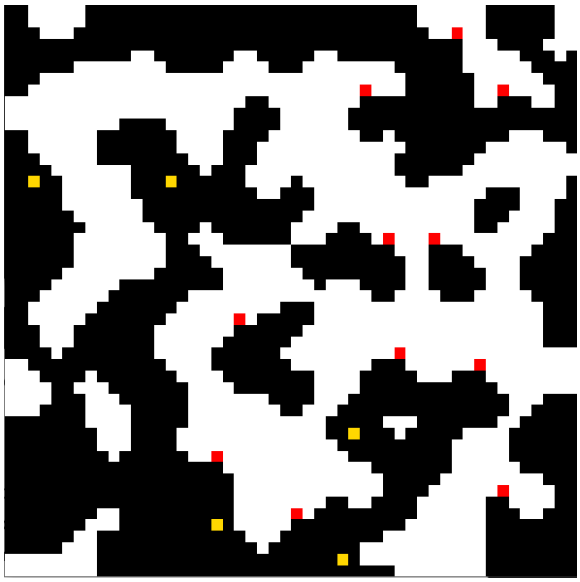
Kun lähdetään kokeilemaan alussa muodostettavien mustien pikselien määrää, huomataan kuvassa 8, että luola muuttuu pienellä parametrin muutoksella paljon. Alussa olevien mustien pikseleiden määrä vaikuttaa esimerkiksi siihen, kuinka paljon yhtenäisiä käytäviä luolastoon muodostuu.




***Kuva 8.*** *Mustien pikseleiden määrä alussa, vasemmalla P = 50 % ja oikealla P = 60 %*

Generoinneista voidaan huomata tiettyjä samankaltaisuuksia, esimerkiksi selkeimmät muodot ovat nähtävissä suurimmassa osassa kuvista, vaikka generointien tulokset muuttuvat paljon. Tuloksia vertaillessa ei voida täsmällisesti sanoa, mikä generoinneista on paras, sillä jokainen peli asettaa erilaisia vaatimuksia kartalle riippuen siitä,

miten sen pelimekaniikat toimivat. Oikeaa vastausta parametreille ei siis ole, vaan ne joudutaan valitsemaan pelikohtaisesti. Aiemmin verratuista kuvista valittiin paremmat lopputulokset, eli ne, joihin ei muodostunut ei-toivottua raidallista rakennetta. Mustien pikseleiden alkumäärä asetettiin 55 %:iin. Käyttämällä näitä parametrejä löydettiin hyvä tasapaino avointa luolaa ja kiveä. Näillä arvoilla luotu lopullinen generointi on esitetty kuvassa 9.



***Kuva 9.*** *Parametrit Y = 3, X = 6 ja P = 55 %*

Kuvaan 2 verrattuna generoitu kuva 9 on tavoitteen mukainen. Kuvaan on tullut luolastoa muistuttavaa ennalta-arvaamatonta rakennetta ja käytäviä. Vihollisia ja aarteita on generoitunut reunaehtojen ja parametrien mukaisesti eli riittävästi ja sopiviin kohtiin.

## 5.7 Kehitysideat

Ohjelmaa voisi parantaa laskemalla luolastolle jokin arvo, mikä kertoisi, onko luolasto hyvä vai huono. Tämän tekeminen on kuitenkin tämän työn osalta liian monimutkaista, sillä olisi ensin määriteltävä todella tarkasti matemaattiset tavoitearvot luolan ominaisuuksille, kuten esimerkiksi avoimen tilan koolle. Tässä vaiheessa parametrejä muuttamalla ja tarkastelemalla löydettiin työlle kelpaava tulos.

Toinen tapa parantaa ohjelmaa olisi, että luolaston osat, jotka eivät kiinnity mihinkään, etsisivät reitin kiinnittyä toiseen luolaston osaan. Ohjelmaa voisi parantaa myös niin, että generoitaisiin uusi luolasto aina kun pelaaja liikkuisi reunan yli. Tässä pitäisi kuitenkin ottaa huomioon, millainen edellinen luola on ollut, jotta luola jatkuisi loogisesti.

# 6. JOHTOPÄÄTÖKSET

Proseduraalinen generointi on tapa luoda tietoa algoritmin avulla. Proseduraalisella generoinnilla on peleissä paljon eri sovelluksia ja näihin kuuluu esimerkiksi kartat, tekstuurit, animaatiot ja äänet. Monet suuret peliyhtiöt ovat alkaneet käyttää proseduraalista generointia tapana säästää aikaa ja rahaa pelituotannon aikana.

Proseduraalinen generointi lisää peliin uudelleenpelattavuusarvoa, sillä eri siemenluvulla alustettu generointi on jokaisella kerralla erilainen. Proseduraalisessa generoinnissa kaikkia generoinnin lopputuloksia on mahdotonta testata. Tästä syystä pelimekaniikkojen ja pelin tavoitteiden suunnittelemista ennen proseduraalisen generaattorin koodaamista olisi hyvä tehdä, sillä generaattorin reunaehtojen ja parametrien valitseminen vaikuttaa voimakkaasti jokaiseen generoituun tasoon. Myös erilaisia algoritmeja ja tapoja on useita ja siitäkin syystä voidaan todeta, että suunnittelu on proseduraalisen generoinnin kannalta yksi tärkeimmistä osa-alueista.

Parametrien vaikutuksen tutkimiseksi toteutettiin ohjelma, jonka avulla voitiin tarkastella muutoksia generoinneissa parametrejä muutettaessa. Ohjelman tekoon käytettiin soluautomaattimallia, joka soveltui ohjelman toiminnan kannalta testiohjelmaan hyvin.

Parametreille ei ole oikeita vastauksia, sillä jokainen peli vaatii kartoiltaan eri asioita. Matemaattisen oikeellisuuden määrittäminen generoidulle kartalle on myös haastavaa. Parhaaksi tavaksi löytää parametrien arvot todettiin generointien vertailu eri arvoilla. Kokeilemalla löydettiin arvot, joilla luola näytti tavoitellulta. Parametrit luolalle olivat: kiven ylipopulaation raja Y = 3, maksimimäärä kiviä avoimen luolan naapurina, joka ylitettäessä avoin luola muuttuu kiveksi X = 6 ja mustien pikseleiden määrä alussa P = 55 %. Lopputulos paranee hieman lähes jokaisella iteraatiokerralla. On kuitenkin hyvä etsiä sopiva raja hyvän generoinnin lopputuloksen ja tehokkuuden väliltä. Sopivaksi määräksi iteraatioita löydettiin arvo 6.

# LÄHTEET

C Standard General Utilities Library. Saatavilla (Luettu 16.4.2021): http://www.cplusplus.com/reference/cstdlib/

C Time Library. Saatavilla (Luettu 16.4.2021): http://www.cplusplus.com/reference/ctime/

Nintendo Company history. Saatavilla (Luettu 16.4.2021): https://www.nintendo.co.jp/corporate/en/history/index.html

ANTONIUK, I. & ROKITA, P., 2016. Generation of Complex Underground Systems for Application in Computer Games with Schematic Maps and L-Systems, L.J. CHMIELEWSKI, A. DATTA, R. KOZERA and K. WOJCIECHOWSKI, eds. In: *Computer Vision and Graphics* 2016, Springer International Publishing, pp. 3-16.

FRADE, M., DE VEGA, F.F. & COTTA, C., 2012. Automatic evolution of programs for procedural generation of terrains for video games: Accessibility and edge length constraints. *Soft computing (Berlin, Germany), Vol.* 16(11), pp. 1893-1914.

GARZIA, A.A., 2020. *Roguelike development with JavaScript : build and publish roguelike genre games with JavaScript and Phaser.* 1 edn. Apress.

GREEN, D., 2016. *Procedural Content Generation for C++ Game Development.* 1 edn. Packt Publishing.

GRENDEL GAMES, Procedural Generation - Based on Everything Procedural Conference. Saatavilla (Luettu 16.5.2021): https://grendelgames.com/procedural-generation/

GYGAX, G., 1978. *Advanced D&D Players Handbook.* TSR Games.

JOHNSON, L., YANNAKAKIS, G.N. & TOGELIUS, J., 2010. Cellular Automata for Real-Time Generation of Infinite Cave Levels, *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Association for Computing Machinery.

KAZEMI, D. Spelunky Generator Lessons. Saatavilla (Luettu 15.4.2021): http://tinysubversions.com/spelunkyGen/

MOORE, M.E., 2011. *Basics of game design.* Boca Raton: CRC Press.

PANAGIOTOU, E., CHOCHLAKIS, G., GRAMMATIKOPOULOS, L. & CHAROU, E., 2020. Generating Elevation Surface from a Single RGB Remotely Sensed Image Using Deep Learning. *Remote sensing (Basel, Switzerland), Vol.* 12(12), pp. 2002.

PETI. Worms Map Database - Maps - Cavern Shopper02. Saatavilla (Luettu 14.5.2021): https://www.wmdb.org/13701

RIPAMONTI, L.A., MANNALÀ, M., GADIA, D. & MAGGIORINI, D., 2017. Procedural content generation for platformers: designing and testing FUN PLEdGE. *Multimedia Tools and Applications, Vol.* 76(4), pp. 5001-5050.

SMED, J. & HAKONEN, H., 2017. *Algorithms and networking for computer games.* Hoboken, NJ: Wiley.

SMITH, G., 2015. An Analog History of Procedural Content Generation, *FDG*, 22-25.6.2015.

TOGELIUS, J., WHITEHEAD, J. & BIDARRA, R., 2011. Procedural Content Generation in Games (Guest Editorial). *Computational Intelligence and AI in Games, IEEE Transactions on, Vol.* 3, pp. 169.

TSCHUMPERLÉ, D., *The Cimg Library - C++ Template Image Processing Toolkit.*

VAN DER LINDEN, R., LOPES, R. & BIDARRA, R., 2014. Procedural Generation of Dungeons. *IEEE transactions on computational intelligence and AI in games., Vol.* 6(1), pp. 78-89.

WATKINS, R., 2016. *Procedural content generation for Unity game development: harness the power of procedural content generation to design unique games with Unity.* Packt Publishing.

ZHANG, C. & SARJOUGHIAN, H.S., 2017. Cellular Automata DEVS: A Modeling, Simulation, and Visualization Environment, *Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques* 2017, Association for Computing Machinery, pp. 11–19.