OAMK
OULUN AMMATTIKORKEAKOULU

Eero Sivonen

PROSEDURAALINEN PELIKENTTIEN LUONTI

PROSEDURAALINEN PELIKENTTIEN LUONTI

Eero Sivonen
Opinnäytetyö
Kevät 2017
Tietojenkäsittelyn koulutusohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma, web-sovelluskehityksen sv

---

Tekijä: Eero Sivonen
Opinnäytetyön nimi: Proseduraalinen pelikenttien luonti
Työn ohjaaja: Matti Viitala
Työn valmistumislukukausi ja -vuosi: Kevät 2017 Sivumäärä: 27

Tässä opinnäytetyössä tutustuttiin proseduraalisen generoinnin (procedural generation) erilaisiin menetelmiin ja niiden käyttöön pelinkehityksessä. Tämän tutkimuksen pohjalta ohjelmoitiin toimiva proseduraalisen generoinnin esimerkki Unityssä, jonka tarkoituksena oli tehdä pelien kentiksi sopivia luolastoja. Opinnäytetyö lähti omasta kiinnostuksesta aiheeseen, eikä kyseessä ole siis toimeksianto.

Aineistoina käytettiin erilaisia digitaalisia lähteitä, joista tärkeimpänä proseduraaliseen generointiin jakautuvat aineistot.

Opinnäytetyöllä saavutettiin laajempi ymmärrys proseduraalisesta generoinnista pelinkehityksessä ja tuotettiin esimerkki proseduraalisesta pelikenttien generoinnista Unityssä. Unityssä toimivaa esimerkkiä voisi jatkokehittää monipuolisemmaksi ja parantaa sen muutamia ongelmallisia puolia. Opinnäytetyötä voi hyödyntää kaikki proseduraalisesta generoinnista tai sovelluskehitykseen liittyvistä menetelmistä kiinnostuneet.

Asiasanat: pelit, peliohjelmointi, pelisuunnittelu, algoritmit

ABSTRACT

Oulu University of Applied Sciences
Business Information Systems, web application development

Author: Eero Sivonen
Title of thesis: Procedural generation of game levels
Supervisor: Matti Viitala
Term and year when the thesis was submitted: Spring 2017 Number of pages: 27

The main purpose of this thesis was to research the topic of procedural generation and it's usage in game development and based on this research program a procedural generator in Unity that creates game levels based on the parameters and assets given by the user. The topic of this thesis came from my own interest in the subject and it was not a commission.

The research materials used for this thesis were mostly digital and most of them are on the topic of procedural generation.

A greater understanding of procedural generation and it's usage in game development was achieved by doing this thesis. The goal of making a functioning procedural generator in Unity was also successful. The procedural generator has some room for continued development as there are some issues to solve and more features could be added. The thesis could be used by anyone interested in procedural generation or it's usage in game development.

Keywords: games, game programming, game design, algorithms

SISÄLLYS

1 JOHDANTO ... 6
2 PROSEDURAALINEN GENEROINTI ... 7
2.1 Algoritmit ja menetelmät ... 7
2.1.1 Perlin noise ... 7
2.1.2 Cellular automata ... 8
2.1.3 Delaunay Triangulation ja Minimum spanning tree ... 10
2.1.4 TinyKeep -pelin menetelmä ... 11
2.1.5 Huonepohjainen generointi ... 12
2.2 Proseduraalisen generoinnin käyttö peleissä ... 13
2.2.1 Rogue ja Roguelike ... 13
2.2.2 Daggerfall ... 14
3 PROJEKTIN TOTEUTUS ... 16
3.1 Unity ... 16
3.2 Huoneiden teko ... 17
3.3 Luolaston generointi ... 19
3.4 Ongelmat ... 21
3.5 Yhteenveto ... 22
4 POHDINTA ... 24
LÄHTEET ... 25

1 JOHDANTO

Proseduraalinen generointi (Procedural Generation) on menetelmä datan tai sisällön luomiselle algoritmien avulla sen sijaan, että sitä luotaisiin käsin. Sitä käytetään pelinkehityksen lisäksi myös elokuvateollisuudessa. Tässä opinnäytetyössä tutustutaan proseduraalisen generoinnin erilaisiin algoritmeihin ja niiden käyttöön pelinkehityksessä. Esittelen myös oman proseduraalisen generoinnin tuotokseni, joka tehtiin Unity-pelimoottoria käyttäen.

Proseduraalista generointia on käytetty pelinkehityksessä jo tietokonepelien alkuajoista lähtien. Jo vuonna 1980 julkaistu Rogue-peli hyödynsi sitä ei pelkästään pelikenttien, mutta myös pelin hirviöiden ja aarteiden generointiin. Alkujaan proseduraalisen generoinnin käyttöperusteet saattoivat keskittyä enemmän senaikaisiin tallennustilaongelmiin, mutta ison pelattavan sisällön tarjoaminen oli myös osallisena. Muunmuassa Elite ja The Elder Scrolls II: Daggerfall –pelit hyödynsivät proseduraalista generointia tarjotakseen massiiviset pelimaailmat, joita ei järkevän aikarajan sisällä olisi kyetty tekemään käsin. Nykypäivän pelimaailmassa proseduraalinen generointi on itsenäisten pelinkehittäjien suosiossa ja se on keskeisessä osassa niinsanottuja Roguelike ja Roguelite –peligenrejä.

Halusin tehdä opinnäytetyötä varten proseduraalista generointia käyttävän ohjelman, joka generoisi peliin sopivia luolastoja. Inspiraationa opinnäytetyölle oli monet proseduraalista generointia hyödyntävät pelit ja etenkin Daggerfall-pelin monimutkaiset 3D-luolastot. Päädyin lopulta huonepohjaiseen ratkaisuun omaa ohjelmaani varten, koska se näytti parhaalta tavalta lähestyä tavoitettani.

2 PROSEDURAALINEN GENEROINTI

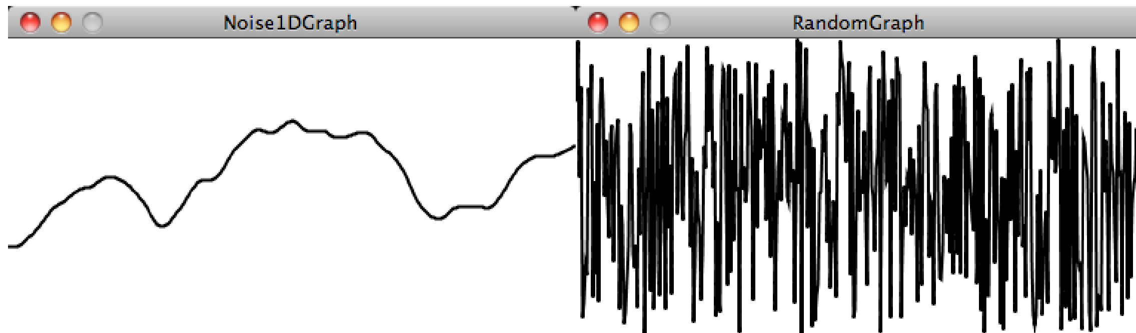
Tietojenkäsittelyssä proseduraalinen generointi on menetelmä datan luomiselle algoritmien avulla sen sijaan, että se luotaisiin käsin. Proseduraalista generointia kutsutaan useasti myös satunnaisgeneroinniksi. Pelinkehityksessä sitä käytetään muunmuassa pelikenttien, tekstuurien tai efektien luontiin (Wikipedia 2017a, viitattu 17.5.2017). Pelinkehityksessä saatetaan puhua myös proseduraalisesta sisällön generoinnista (Procedural Content Generation), jolloin tarkoitetaan nimenomaan pelin pelaamiseen vaikuttavan proseduraalisen generoinnin käyttöä (Wikidot 2017, viitattu 30.5.2017).

2.1 Algoritmit ja menetelmät

Proseduraalisen generoinnin algoritmit ovat yleensä johonkin tiettyyn käyttötarkoitukseen suunniteltuja. Algoritmin kehittäjällä oli jokin ongelma, joka piti ratkaista ja algoritmi kehitettiin sitä varten. Niitä ei välttämättä ole tarkoitettu laajalle levinneeseen käyttöön, mutta poikkeuksiakin löytyy. Esimerkiksi Perlin Noise kehitettiin alkujaan Tron-elokuvan tietokonegrafiikan teksturointia varten, mutta se on nykyään varsin suosittu proseduraalisen generoinnin algoritmi ja sitä on sovellettu moneen eri käyttötarkoitukseen.

2.1.1 Perlin noise

Ken Perlin kehitti Perlin Noisen turhautuessaan tietokonegrafiikan konemaisuuteen (Wikipedia 2017b, viitattu 19.5.2017). Perlin Noise luo varsin luonnollisen näköisiä muotoja, eikä siitä kykene helposti näkemään minkäänlaista toistuvaa kuviota. Algoritmin suosio perustuu sen monipuolisuuteen, Perlin Noisea voidaan käyttää yksi-, kaksi- ja kolmiulotteisessa muodossa erilaisin tuloksin. Sen lisäksi jokaiseen ulottuvuuteen voi lisätä yhden edustamaan aikaa, jota voidaan käyttää sen animoinnissa (KUVIO 1) (Biagioli 2014, viitattu 19.5.2017).



KUVIO 1. Yksiulotteinen Perlin Noise verrattuna satunnaisgeneroituun. X-akseli edustaa aikaa

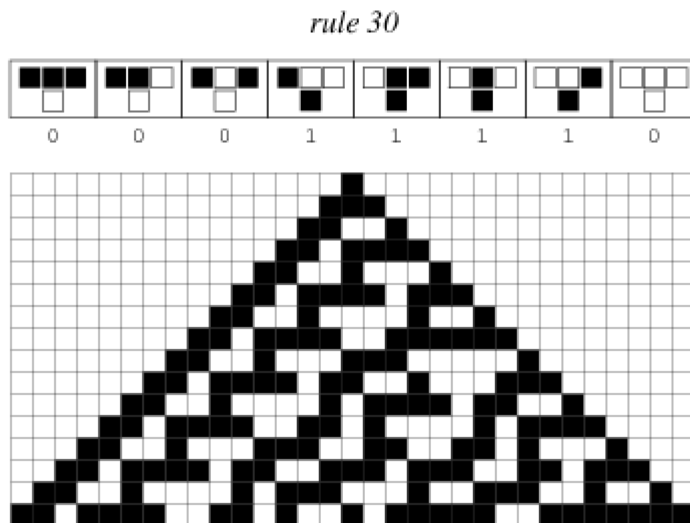
Perlin Noisea voidaan käyttää peleissä esimerkiksi tekstuurien proseduraaliseen generointiin. Se soveltuu erittäin hyvin esimerkiksi tulen, veden tai pilvien teksturointiin, mutta tarpeeksi soveltamalla käy se myös 3D-mallien teksturointiin (KUVIO 2). Kolmiulotteinen Perlin Noise soveltuu myös kolmiulotteisien mallien generointiin. Perlin Noise on tehokas menetelmä, muttei ainoa ja erilaiset menetelmät mahdollisesti sopivat erilaisiin projekteihin paremmin.

KUVIO 2. Unityn kaksiulotteinen Perlin Noise esimerkki

2.1.2 Cellular automata

Cellular automata on kokoelma värillisiä soluja määritetyn kokoisessa ruudukossa, jossa solut vaihtavat väriä *n* määrän kertoja niille annettujen sääntöjen perusteella (Weisstein 2017, viitattu 19.5.2017). Säännöt perustuvat yleensä solua ympäröiviin soluihin, mutta aina ei tarvitse käyttää

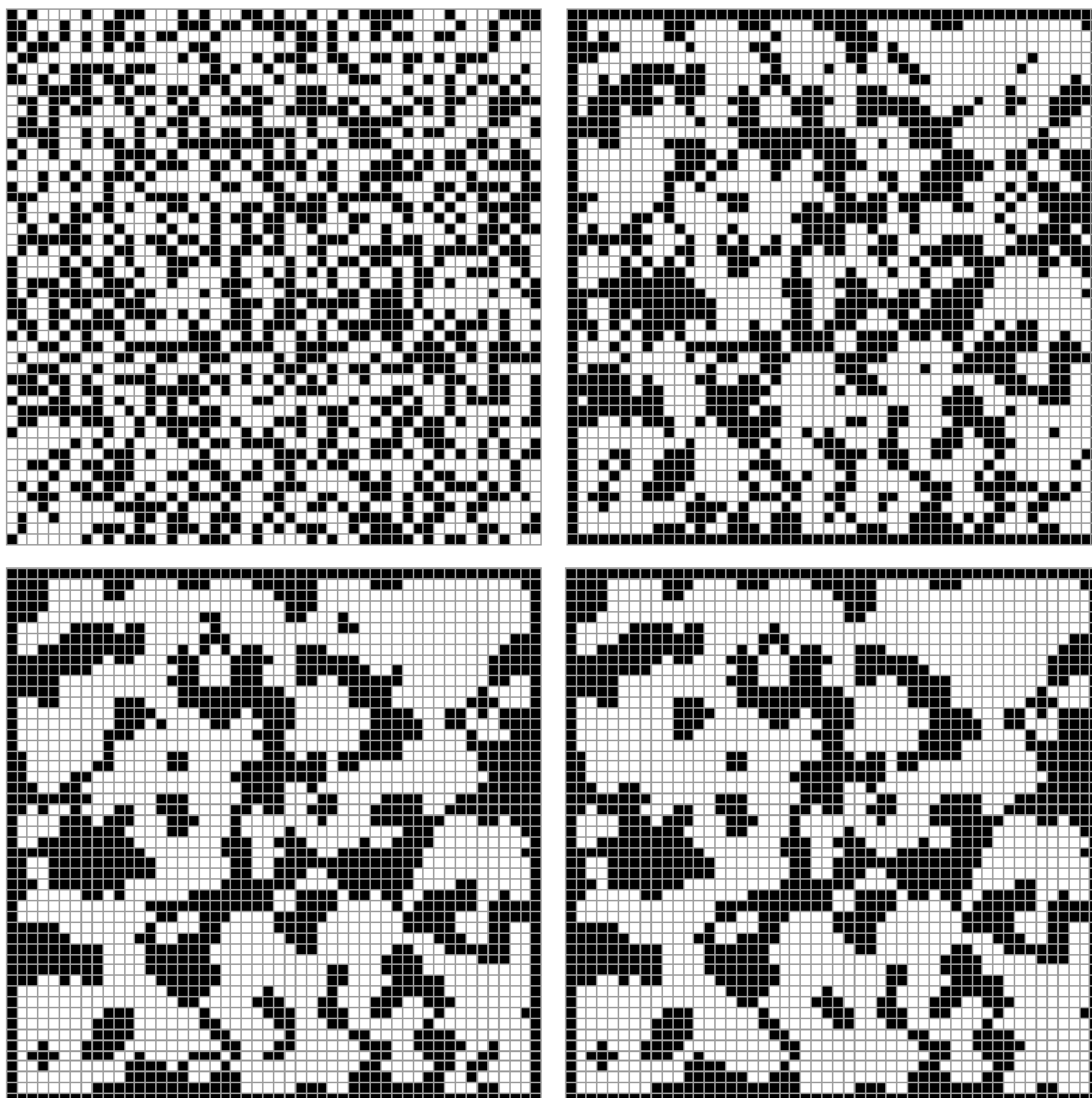
kaikkea kahdeksaa naapurisolua. Esimerkiksi Rule 30 nimellä kulkeva sääntö, pohjaa päätöksen solun kolmeen yläpuolella olevaan soluun (KUVIO 3).



KUVIO 3. Cellular Automata Rule 30, jossa aloituspisteenä on yksi musta piste ylimmällä rivillä

Yleensä värejä on kaksi, jota kutsutaan nimellä Life-like automata. Siinä valkoinen vastaa tyhjää tai ”kuollutta” solua ja musta täyttä tai ”elävää” solua. Soluille annetut säännöt päättävät syntyykö uusia soluja tai selviävätkö olemassa olevat elävät solut, niitä ympäröivän kahdeksan naapurisolun tilan perusteella. Tätä merkitään usein merkinnällä Bx/Sy, jossa x ja y ovat yksilukuisten numeroiden listoja. Esimerkiksi B3/S23 tarkoittaisi, että uusi solu syntyy, jos sillä on kolme elävää naapuria, ja selviää, jos sillä on kaksi tai kolme elävää naapuria. Muuten solu kuolee tai merkitään valkoiseksi (Kun 2012, viitattu 19.5.2017).

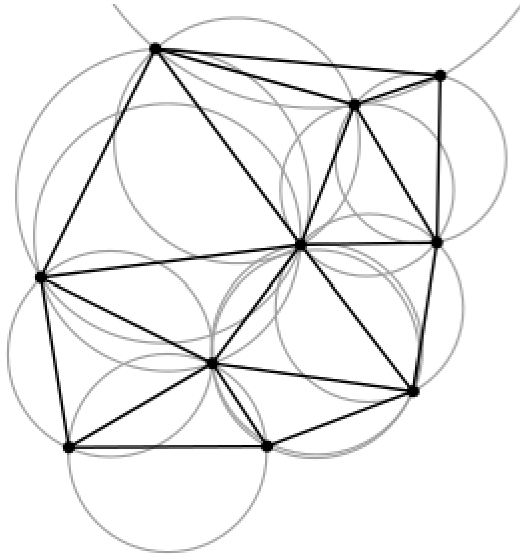
Tarpeeksi isolla ruudukolla sopivan aloituskokoelman ja säännön perusteella voidaan generoida varsin luolaston näköisiä kokonaisuuksia (KUVIO 4). Tätä lopputulosta voi vielä käsitellä cellular automatan ulkopuolella esimerkiksi varmistamalla, että kaikki valkoiset alueet ovat yhteydessä toisiinsa. Tämä varmistaisi pelikäytössä, että pelaaja kykenisi liikkumaan kaikkiin valkoisiin soluihin.



KUVIO 4. Cellular Automata toiminnassa satunnaisesti generoidussa aloituskokoelmassa säännöllä B678/S345678

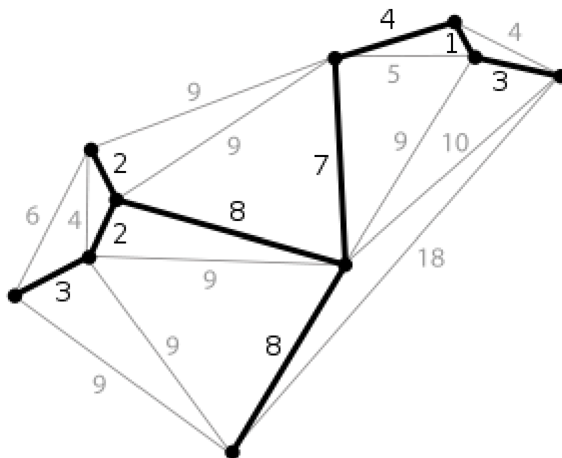
2.1.3 Delaunay Triangulation ja Minimum spanning tree

Delaunay Triangulation on triangulaatio algoritmi, joka yhdistää kokoelman pisteitä kolmioiksi siten, että yhdenkään kolmion kärkien kautta kulkevan ympyrän sisällä ei ole kokoelman pistettä. Lopputuloksena saadaan kytketty suuntaamaton verkko (KUVIO 5). Tämä tarkoittaa sitä, että mistä tahansa verkon pisteestä päästään mihin tahansa muuhun pisteeseen kulkemalla tarpeeksi monen pisteen ja viivan kautta (Wikipedia 2017c, viitattu 20.5.2017).



KUVIO 5. Delaunay Triangulation ja kolmioiden ympäri piirretyt ympyrät

Minimum Spanning Tree on algoritmi jota voidaan käyttää kytkettyyn verkkoon, josta saadaan alaverkko, joka yhdistää verkon pisteet ilman yhtään kierrosta ja jossa pisteiden välisten viivojen painot ovat mahdollisimman pienet (KUVIO 6). Täten jokaisesta pisteestä pääsee jokaiseen toiseen pisteeseen vain yhtä reittiä ja kuljettu kokonaispaino on pienin mahdollinen annettujen painojen perusteella (Wikipedia 2017d, viitattu 20.5.2017).



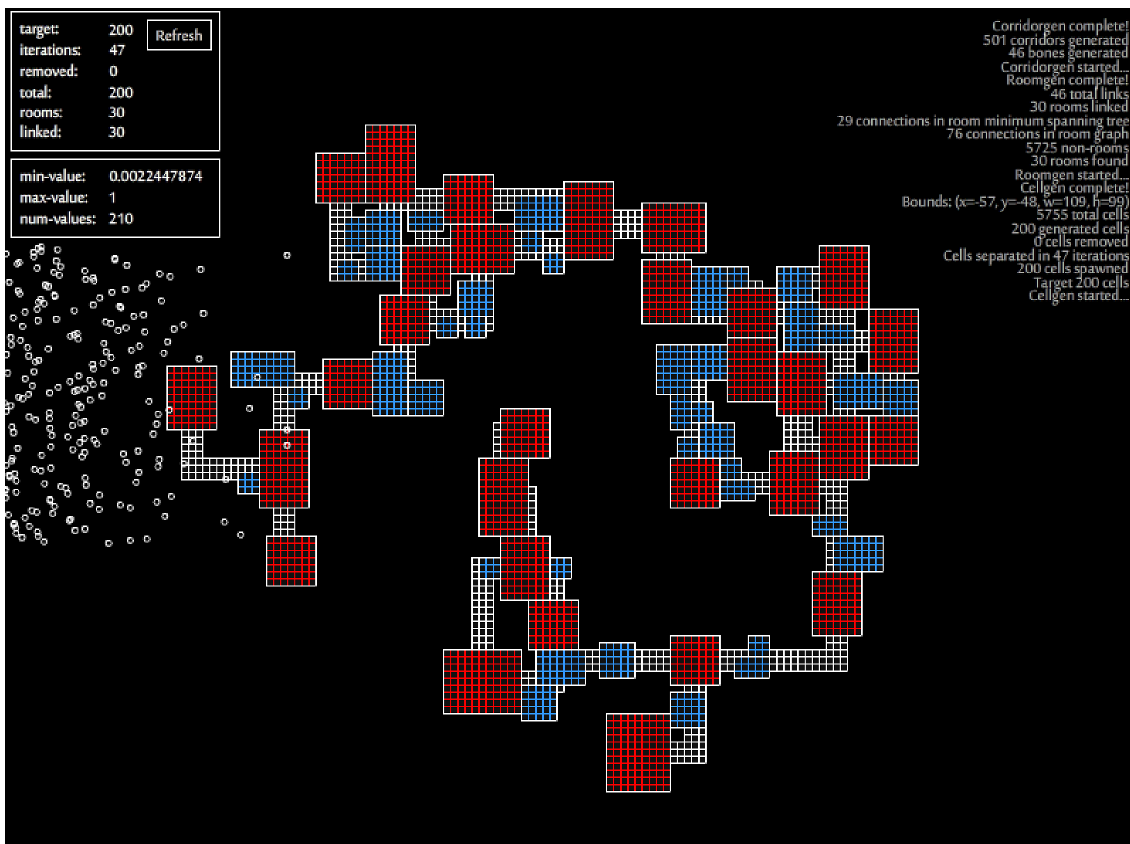
KUVIO 6. Minimum Spanning Tree

2.1.4 TinyKeep -pelin menetelmä

Phi Dinh kehitti TinyKeep peliään varten algoritmin joka hyödyntää Delaunay Triangulaatiota ja Minimum Spanning Treetä muun satunnaisgeneroinnin ohella. Hän on pitänyt menetelmästään muutaman esitelmän ja hänen nettisivuillaan on esimerkkigeneraattori, joka visualisoi miten

TinyKeep pelin proseduraalinen generointi toimii (KUVIO 7). Hänen menetelmästä on myös kirjoitettu Gamasutra artikkeli, jossa annetaan esimerkkejä, miten menetelmää lähestyttäisiin ohjelmoinnin kannalta (Adonaac 2015, viitattu 27.5.2017).

Aluksi algoritmi generoi määritetyn määrän satunnaiskokoisia suorakulmioita. Sen jälkeen suorakulmiot hajautetaan siten, että ne eivät ole päällekkäin. Tietyn koon ylittävät suorakulmiot valitaan ja merkitään huoneiksi ja niiden pohjalta rakennetaan Delaunay Triangulaation avulla verkko. Kyseiseen verkkoon ajetaan Minimum Spanning Tree, jonka lisäksi pieni määrä ylimääräisiä reittejä säilytetään pisteiden välillä, jotta verkkoon saadaan muutama kierros. Tämä tekee generoiduista kartoista hieman melenkiintoisempia, koska jokainen haara ei johdakaan umpikujaan. Lopuksi huoneet yhdistetään verkossa olevien reittien pohjalta L muotoisilla käytävillä (Dinh 2013, viitattu 22.5.2017).



KUVIO 7. TinyKeep pelin proseduraalisesti generoitu esimerkkiluolasto

2.1.5 Huonepohjainen generointi

Tutustuessani erilaisiin proseduraalisen generoinnin ratkaisuihin löysin useamman eri henkilön kirjoittamia esimerkkejä varsin samankaltaisista ideoista, joilla ei ollut yhtenäistä nimeä, eikä

mitään yhteistä algoritmia. Yksinkertaistettuna yleinen idea kaikissa näissä ratkaisuissa oli, että aloitetaan asettamalla yksi huone pelitilaan ja yhdistetään siihen proseduraalisella logiikalla toinen huone ja toistetaan tätä kunnes saavutetaan halutun kokoinen pelialue. Huonepohjainen generointi (Room based generation) ei ole vakiintunut termi, mutta sitä on käytetty muutamassa keskustelussa. Se kuitenkin kuvaa kyseistä ideaa varsin hyvin.

Huonepohjaisen generoinnin etuna on se, että pelin kehittäjä voi itse suunnitella huoneet käsin, jolla voidaan saavuttaa käsin tehty (handcrafted) tuntuma pelikenttiin, mutta silti säilyttää jokainen pelikerta hyppynä tuntemattomaan. On myös mahdollista käyttää proseduraalista generointia jokaisen huoneen sisäisesti, jotta voidaan esimerkiksi pitää koko pelikentästä löytyvä aarteiden määrä kentän vaikeuteen sopivana. Samalla saman huoneen näkeminen eri pelikerroilla ei tarkoita joka kerta täsmälleen samaa huonetta.

2.2 Proseduraalisen generoinnin käyttö peleissä

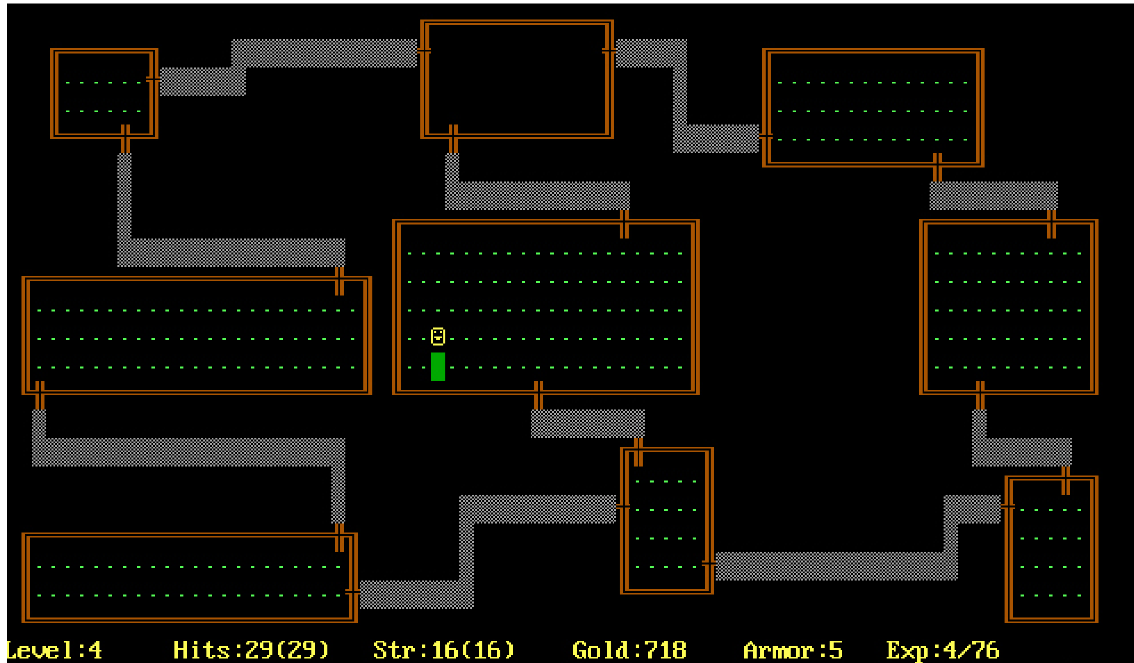
Pelejä harrastavalle proseduraalinen generointi käsitteenä yhdistetään mitä todennäköisemmin pelikenttien proseduraaliseen generointiin. Tämä ei tule yllätyksenä, koska monet peliklassikot ja uudet indiepeli suosikit käyttivät proseduraalista generointia nimenomaan pelikenttiin. Esimerkiksi Diablo ja Spelunky -pelit ovat tunnettuja proseduraalisista pelikentistään.

Proseduraalista generointia käytetään kuitenkin erittäin monipuolisesti peleissä. Vuonna 1996 julkaistu Diablo käytti proseduraalista generointia pelikenttien, mutta myös hirviöiden tiputtamaan aarteiden määrään ja itse aarteiden maagisiin ominaisuuksiin. 2016 valmistunut Moon Hunters oli isolta osin kokeilu proseduraaliseen tarinankerrontaan. Pelissä pelaajan tekemät valinnat tallennetaan pelikertojen välillä ja ne saattavat vaikuttaa tuleviin pelikertoihin. Pelit Left 4 Dead ja Warhammer: End Times – Vermintide hyödyntävät pelin vihollisten hallinnassa tekoälyä, joka dynaamisesti ohjaa pelaajien kimppuun erilaisia vihollisaaltoja ja erikoisvihollisia (Wikipedia 2017a, viitattu 17.5.2017).

2.2.1 Rogue ja Roguelike

Vuonna 1980 julkaistu Rogue ei ollut ensimmäinen peli, joka hyödynsi proseduraalista generointia, mutta se oli niin suuri hitti aikansa skaalassa, että se inspiroi muita ohjelmoijia

kehittämään omia versioitaan pelistä ja lopulta synnytti uuden peligenren Roguelike (KUVIO 8). Rogue käytti proseduraalista generointia sen pelikentissä ja niiden hirviöissä ja aarteissa (Wikipedia 2017e, viitattu 22.5.2017). Proseduraalisesta generoinnista muodostui Roguen esimerkkiä seuraten Roguelike genren perusominaisuus.



KUVIO 8. Alkuperäinen Rogue-peli IBM Color PC:llä

Roguelike-genren suosiosta huolimatta sen pelit olivat monelle liian vaikeita tai turhauttavia monien genren oleellisien ominaisuuksien takia. Indie-pelien nousun jälkeen hiljalleen muodostui uusi peligenre, jota alettiin kutsua Roguelite-nimellä. Roguelite-pelit sisälsivät yleensä muutamia Roguelike-peleille ominaisia ominaisuuksia, mutta olivat lähestyttävempiä ja graafisesti näyttävämpiä. Proseduraalinen generointi oli tärkein mukana pidetty osuus ja tämä synnytti puolestaan paljon mielenkiintoa niin pelaajien kuin pelinkehittäjien piireissä proseduraalista generointia kohtaan.

2.2.2 Daggerfall

The Elder Scrolls II: Daggerfall on vuonna 1996 julkaistu avoimen maailman fantasiaroolipeli PC:lle. Peli keskittyy massiiviseen alueeseen, joka kuvastaa pelin maailman, Tamrielin, High Rock ja Hammerfall provinsseja. Pelimaailman alueen koko on arviolta 160000m^2 ja se on suurelta osin proseduraalisesti generoitu (Wikipedia 2017f, viitattu 22.5.2017). Yli 20 vuotta

myöhemmin yksikään peli ei ole saavuttanut lähellekään yhtä suurta mittakaavaa ja siltä osin Daggerfall on erittäin merkittävä teos vielä nykypäivänkin pelimaailmassa.

Monet huonepohjaisen generoinnin ratkaisut mainitsivat inspiraationa tai esimerkkinä Daggerfall pelin proseduraaliset luolat ja kyseinen menetelmä kuvaakin pelissä olevaa generointia hyvin. Daggerfallin luolastot vaihtelivat pienistä koloista massiivisiin rakennelmiin, joihin pelaaja voisi eksyä tuntikausiksi. Luolastojen monimutkaisuus olikin pelin yksi vihatuimpia ja rakastetuimpia ominaisuuksia. Kuviossa 9 on esimerkki pienestä Daggerfall-pelin luolasta.

KUVIO 9. Daggerfall-pelin luola

3 PROJEKTIN TOTEUTUS

Ennen opinnäytetyön aloittamista olin tutustunut Cellular Automataan Sebastian Laguen (2017, viitattu 24.5.2017) tekemässä videosarjassa, jossa luodaan toimiva versio algoritmista Unityssä. Tämän pohjalta tiesin, että halusin enemmän kontrollia siihen minkälaisia pelikenttiä ohjelma kykenee tuottamaan. Halusin myös mahdollisuuden, että kentissä voisi olla korkeuseroja, johon kaksiulotteinen Cellular Automata ei sovellu ja kolmiulotteinen Cellular Automata vaikutti hieman liian monimutkaiselta ratkaisulta.

Huonepohjainen generointi vaikutti parhaalta lähestymistavalta. Sitä varten olisi helppo tehdä muutama esimerkkihuone, jotta toteutuksen graafiseen puoleen ei tarvitsisi käyttää paljoa aikaa. Marcin Seredynskin (2014, viitattu 24.5.2017) artikkeli 3D-luolastojen proseduraalisesta generoinnista esitti idean huonepohjaisesta ratkaisusta, minkä kautta lähdin lähestymään toteutusta. Varsinaisen toteutuksen tein Unity pelinkehitysympäristössä.

3.1 Unity

Unity on vuodesta 2005 kehitteillä ollut pelimoottori ja pelinkehitysympäristö. Unity Technologiesin kehittämä Unity oli alunperin saatavilla vain Applen macOS:lle, mutta nykyään sitä voidaan käyttää myös Microsoft Windows ja Linux -käyttöjärjestelmillä ja sillä voidaan tehdä pelejä 27:lle eri kohdealustalle. Kohdealustoista tärkeimpänä tietenkin Microsoft Windows, mobiililaitteiden Android ja iOS -käyttöjärjestelmät ja pelikonsolit Nintendo Switch, Playstation 4 ja Xbox One. Unityssä käytettävät ohjelmointikielet ovat C#(Csharp) ja Javascript (Unity Technologies 2017a, viitattu 25.5.2017).

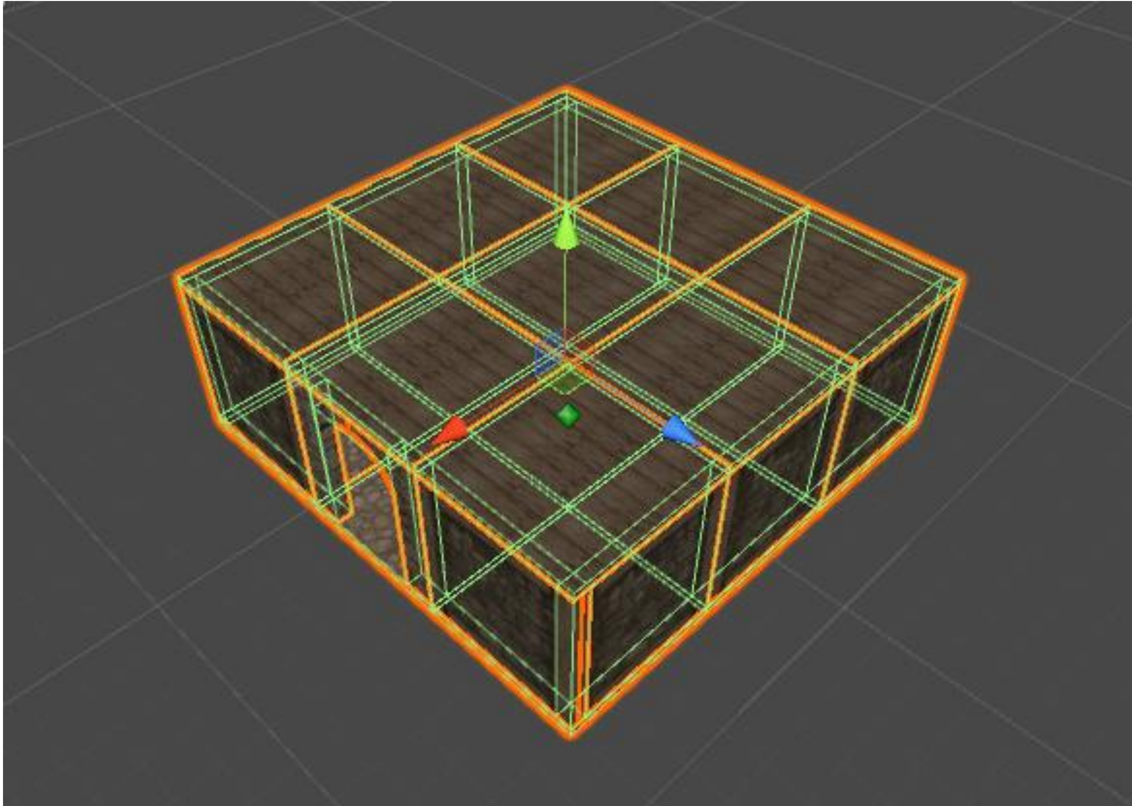
Unityssä pelinkehitys keskittyy näyttämölle (Scene) sijoitettuihin objekteihin (GameObject). Näyttämöt säilyttävät niihin tallennetut objektit ja ne tallentuvat omiksi tiedostoiksi, joita voi ladata peliä ajaessa (Unity Technologies 2017b, viitattu 29.5.2017). Objektit edustavat kaikkea mitä Unityssä tehdyssä pelissä on, mutta objekteille on annettava komponentteja (Component), jotta niillä voi tehdä erilaisia asioita (Unity Technologies 2017c, viitattu 29.5.2017). Komponentit ovat erilaisia paloja, jotka mahdollistavat objekteille erilaisia toimintoja (Unity Technologies 2017d, viitattu 29.5.2017). Näiden lisäksi tärkeänä osana ovat ohjelmoijan itse kirjoittamat scriptat

(Script), jotka vastaavat kaikesta mihin Unity ei tarjoa vakioratkaisua itse (Unity Technologies 2017e, viitattu 29.5.2017). Scriptoja voi myös asettaa objekteihin komponenteiksi. Objekteja voi myös tallentaa prefab-tiedostoiksi, joita voi dynaamisesti tuoda näyttämölle scriptojen kautta (Unity Technologies 2017f, viitattu 29.5.2017).

3.2 Huoneiden teko

Aloitin projektin toteutuksen tekemällä muutaman esimerkkihuoneen, jotta minulla olisi heti valmiina jotain, mitä voisin käyttää ohjelman testaamiseen. Huoneiden tekemiseen käytin vanhassa peliprojektissa käytettyjä palasia. Paloja oli yhteensä 6 erilaista, yksi lattia, yksi katto, kaksi seinää ja kaksi oviaukkoa (KUVIO 10). Palat olivat sopivasti neljän Unity-mittayksikön leveitä ja korkeita, joka helpotti huomattavasti palojen yhdistelemistä huoneiksi. Annoin jokaiselle palaselle sille sopivan Collider-komponentin ja tein palasista Prefab-objekteja. Collider-komponentit mahdollistavat objektien väliset törmäykset (Unity Technologies 2017g, viitattu 29.5.2017). Prefabit mahdollistavat palojen helpon ja nopean uudelleenkäytön. Kuviossa 11 näkyy huoneen erilliset palaset oransseilla reunoilla ja niiden Collider-komponentit vihreillä reunoilla. Tällainen käytäntö olisi varsin epätehokasta oikean pelin sisällä, mutta se toimii tämän projektin käyttötarkoitukseen ongelmitta.

KUVIO 10. Huoneiden rakennuspalat



KUVIO 11. Huone-objekti, jossa yksi oviaukko

Huoneiden yhdistämistä varten lisäsin jokaisen huoneen jokaiseen oviaukkoon tyhjän objektin, joka edustaa oviaukon sijaintia Unityn koordinaatistossa. Ovi-objekteja varten kirjoitin myös lyhyen scriptin, joka auttaisi ovien yhdistämisessä toisiinsa (KUVIO 12). Tämä Door-scripta säilyttää tiedon siitä onko ovi yhdistetty johonkin toiseen oveen ja sen lisäksi tietää myös minkä huoneen oveen se on yhdistetty. Päädyin myös tekemään toisen scriptan ovien hallinnan avuksi, joka asetettaisiin jokaisen huoneen pää (parent) objektiin. Tämä huone scripta sisältää funktion, jolla haetaan kaikki huoneen ovet yhteen listaan, josta on helppo käsitellä ovia yksi kerrallaan ohjelmassa (KUVIO 13). Lopuksi kirjoitin ScriptableObject-luokkaa käyttävän scriptan, joka säilyttää yhden huonekokoelman kaikki huoneet prefabeina (KUVIO 14). ScriptableObject on Unityn luokka, joka mahdollistaa datan säilönnän ja luonnin varsinaisen pelirakenteen ulkopuolella (Unity Technologies 2017h, Viitattu 29.5.2017). ScriptableObjectit erottuvat normaaleista objekteihin kiinnitettävistä scriptoista siten, että niistä tehdään tiedostoja jotka säilyvät projektissa ja niitä ei voi laittaa objekteihin komponenteiksi.

```
public class Door : MonoBehaviour {

    public bool isConnected = false;
    public DungeonTile parentTile;
    public DungeonTile connectedTile;
    public int myDoorIndex;
    public int connectedDoorIndex;
}
```

KUVIO 12. Ovi-objekteille annettu scripta

```
public class DungeonTile : MonoBehaviour {

    public Door[] doors;

    public void FindDoors ()
    {
        if (doors.Length == 0) {
            doors = gameObject.GetComponentsInChildren<Door> ();
            for (int i = 0; i < doors.Length; i++) {
                doors [i].myDoorIndex = i;
                doors [i].parentTile = this;
            }
        }
    }
}
```

KUVIO 13. Huone-objekteille annettu scripta

```
public class DungeonTileSet : ScriptableObject
{

    public string setName;
    public List<GameObject> setTiles = new List<GameObject> ();
    public List<GameObject> spawnTiles = new List<GameObject> ();
    public List<GameObject> deadEndTiles = new List<GameObject> ();
}
```

KUVIO 14. ScriptableObject-rakenne huonekokoelman säilytykselle

3.3 Luolaston generointi

Luolaston generointia varten kirjoitin oman scriptan, joka tulisi vastaamaan niinsanotusta raskaasta työstä. Scripta keskittyy GenerateDungeon-funktion ympärille, joka hoitaa yleisellä tasolla luolaston generointia ja kutsuu tarvittavia alafunktioita (KUVIO 15). Ensin se valitsee käytettävän huonekokoelman SelectTileSet-funktiossa. Tämä funktio on varsin yksinkertainen, mutta oikeaa peliä varten sitä kautta olisi helppo laajentaa ja käskyttää scriptaa generoimaan halutun kaltaisia luolia. Sen jälkeen valitaan aloitushuone. Jokaisessa huonekokoelmassa on

määritetty vähintään yksi aloitushuone ja jos niitä on useampi arvotaan yksi niistä ja kopioidaan se näyttämölle. Näiden kahden askeleen jälkeen päästään while–rakenteeseen, jota ajetaan kunnes huoneissa olevat avoimet ovet loppuvat kesken tai saavutetaan scriptan alussa määritetty haluttu huonemäärä. GenerateNextRoom-funktio on iso kokonaisuus, mutta kuviossa 16 on kommenteilla yksinkertaistettu versio sen toiminnasta. Funktio on vastuussa uuden huoneen valinnasta, siihen yhdistettävästä oviaukosta ja huoneen siirtämisestä ja pyörittämisestä oikealle kohdalleen. Sen lisäksi funktiossa tarkistetaan IsRoomColliding-funktion avulla meneekö uusi huone yhdenkään edellisen huoneen päälle. Jos menee, niin se tuhotaan ja ohjelma palaa GenerateDungeon-funktioon, jotta se voi yrittää uudestaan. Huoneiden päällekkäisyyden testaukseen käytetään Unityn objektien BoundingBox-ominaisuutta, joka vastaa kuutiota, joka sisältää koko objektin (Unity Technologies 2017i, viitattu 29.5.2017). Jos uusi huone ei mene päällekkäin yhdenkään edellisen huoneen kanssa se käsitellään osaksi luolastoa ja palataan GenerateDungeon-funktioon.

```
void GenerateDungeon () {
    SelectTileSet ();

    SelectStartingRoom ();

    while (unprocessedDoors.Count > 0 && allTiles.Count <= dungeonSize) {
        if (unprocessedDoors [0].isConnected) {
            unprocessedDoors.RemoveAt (0);
            if (unprocessedDoors.Count < 1) {
                break;
            }
        }
        GenerateNextRoom (unprocessedDoors [0]);
    }
}
```

KUVIO 15. Luolastogeneraattorin GenerateDungeon-funktio

```
void GenerateNextRoom (Door connectingDoor)
{
    //pick a room from the selectedTileSets tile list

    //pick a door to be connected to the previous room

    //rotate for the doors vectors to match
    AlignRoom (newRoom, newDoorIndex, connectingDoor.transform);

    //check for collision with other rooms
    //if collision found discard room and redo
    if (IsRoomColliding (newRoom)) {
        //Not enough space for room. Discarding.
    }
    //else continue

    //mark door as connected

    //add the tile to the list of allTiles
    //add all doors (except the one we just connected) to the list of unprocessedDoors
    allTiles.Add (newRoom);
    foreach (Door door in newRoom.doors) {
        if (door.myDoorIndex != newDoorIndex) {
            unprocessedDoors.Add (door);
        }
    }
}
```

KUVIO 16. Yksinkertaistettu GenerateNextRoom-funktio

3.4 Ongelmat

Luolastogeneraattorin toteutuksessa ilmeni muutamia ongelmia. Isoimpana ongelmana oli uusien huoneiden pyöritys ja paikalleen sijoitus. Lähestyin ongelmaa ehkä huonosta näkökulmasta, mutta ongelmaa etsiessä rajoitin huoneiden pyörityksen 90 asteen väleihin. Jouduin kirjoittamaan osion varmistamaan AlignRoom-funktion toimintaa ja en millään meinannut saada toimivaa tulosta. Jossain vaiheessa tajusin, että alkuperäisen ongelman aiheutti luultavasti niinsanottu Floating Point Error. Floating Pointit eli floatit ovat tietokoneissa käytetty esitystapa reaaliluvuille, jotka uhraavat osan tarkkuudestaan, jotta niitä voidaan käyttää erittäin isojen tai pienien lukujen kuvaamiseen (Wikipedia 2017g, viitattu 29.5.2017). Tämä epätarkkuus luo nopeasti desimaalieroja. Objektin y-akselin kulma saattoi olla teoriassa 90 astetta, mutta käytännössä oikea kulma oli 90,000001. Yritin selvittää ovien pyöritykset tarkoilla vertauksilla ja nämä pienet desimaalivirheet tuottivat ongelmia, joita en älynnyt kuin vasta useamman korjausyrityksen jälkeen. On täysin mahdollista, että ensimmäinen versio huoneiden pyörityksestä olisi toiminut, jos olisin huomioinut floattien epätarkkuuden siinä.

Toinen merkittävä ongelma ilmeni BoundingBoxien käytössä huoneiden päällekkäisyyksiä tarkistaessa. Erilaisia huonemuotoja kokeillessani huomasin, että ovi-objektin on pakko olla huone-objektin BoundingBox-alueen reunalla, tai siihen ei voi yhdistää toista ovea koska huoneiden BoundingBoxit menevät päällekkäin. Ratkaisin tämän ongelman käyttämällä vain oviaukkoja jotka olivat huoneen BoundingBoxin reunalla.

Luolastogeneraattorissa ilmeni myös ongelma isoja huonemääriä generoidessa. Ohjelma jäätyi, koska se koitti tehdä koko luolastoa yhdellä kertaa ja koko luolaston generointi saattoi kestää jopa kymmeniä sekuntteja. Ratkaisin tämän muuttamalla GenerateDungeon-funktion Coroutine-rakenteeksi (KUVIO 19). Coroutinet ovat funktioita, joille voi antaa odotuskäskyjä (Unity Technologies 2017j, viitattu 29.5.2017). Tekemällä tämän muutoksen kykenin lisäämään odotuskäskyn jokaisen huoneen lisäämisen jälkeen ja ratkaisin jäätymisen ongelman. Sen lisäksi lisäsin vaihtoehdon asettaa odotusajan sekunteina, jotta voisin visualisoida generoinnin tapahtuman paremmin. Jo puolen sekuntin viiveellä ohjelmaa on helppo seurata, kun se generoi uusia huoneita.

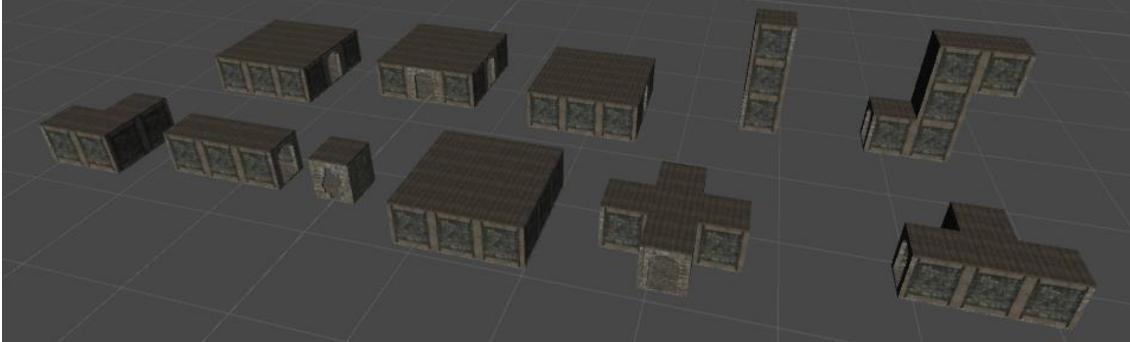
```
IEnumerator GenerateDungeon ()
{
    SelectTileSet ();

    SelectStartingRoom ();

    while (unprocessedDoors.Count > 0 && allTiles.Count <= dungeonSize) {
        if (unprocessedDoors [0].isConnected) {
            unprocessedDoors.RemoveAt (0);
            if (unprocessedDoors.Count < 1) {
                break;
            }
        }
        GenerateNextRoom (unprocessedDoors [0]);
        if (useWaitTime) {
            yield return new WaitForSeconds (waitTime);
        } else {
            yield return WaitFrame ();
        }
    }
}
```

KUVIO 17. GenerateDungeon-funktio Coroutineksi muutettuna

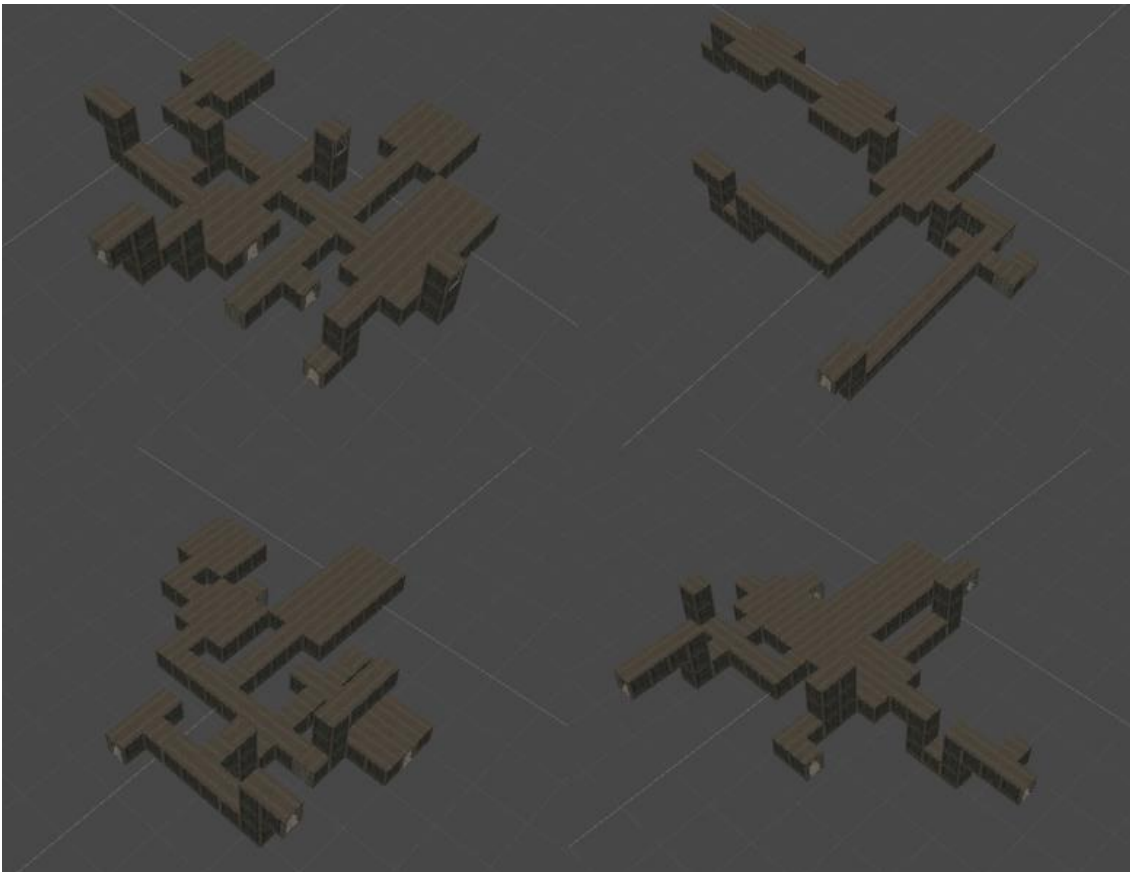
3.5 Yhteenveto

Edellä esitetyn kokonaisuuden avulla luolastogeneraattori kykenee generoimaan erikokoisia luolastoja. Generoinnin säännöt ovat varsin yksinkertaiset ja kaikkia huoneita kohdellaan samoin,

olivat ne sitten oikeasti huoneita, käytäviä, portaikkoja tai risteyksiä. Kuviossa 18 näkyvät lopulliset projektin testaukseen käytetyt huoneet. Huoneiden muotoihin monipuolisuutta lisäämällä voisi helposti lisätä myös generoitujen luolastojen monipuolisuutta. Kuviossa 19 näkyvät neljä erilaista luolastoa jotka on generoitu 20 huoneella.



KUVIO 18. Luolastogeneraattorin testausta varten tehdyt huoneobjektit



KUVIO 19. Neljä esimerkkiluolastoa 20 huoneella

4 POHDINTA

Proseduraalinen generointi oli varsin mielenkiintoinen aihe opinnäytetyöksi. Siitä olisi voinut kirjoittaa huomattavasti isomman määrän tekstiä, mutta oli järkevää rajata aihe tarkemmaksi. Työn tavoitteet tutustua proseduraalisen generoinnin algoritmeihin ja kehittää oma ratkaisu Unityssä onnistuivat mainiosti, vaikka omaa ratkaisua olisi vielä voinut kehittää monipuolisemmaksi. Opinnäytetyön isoimmaksi ongelmaksi muodostui motivaatio kirjoitustyölle, joka kuitenkin lopulta tapahtui nopeasti, kunhan sain sen alulle. Unityn käyttö oman luolastogeneraattorin kehityksessä oli hyvä valinta, koska minulla oli ennestään tuntemusta sen käytöstä ja se antoi kaikki tarvittavat työkalut 3D-pelikenttien esittämiseen, jotta kykenin keskittymään täysin ohjelmakoodin toteutukseen.

Luolastogeneraattori jäi kokonaisuutena aika yksinkertaiseksi ja minulla jäi mieleen muutama asia, mitä siinä olisi voinut korjata. Huoneiden pyöritys osoittautui isoksi ongelmaksi ja sen ratkaiseminen voisi mahdollistaa ovien kulmat muutenkin kuin 90 asteen välein. Toisaalta huoneiden suunnittelun kannalta voisi olla hyvä idea asettaa jotain rajoitteita, missä kulmissa ovet voisivat olla. Huoneiden päällekkäisyyksien tarkistaminen BoundingBoxia käyttämällä osoittautui aika isoksi rajoitukseksi huoneiden muodoille. Voisi olla hyvä idea kehittää jonkinlainen huoneen kokonaismuotoon perustuva ratkaisu, jotta huoneiden muodoista saataisiin mielenkiintoisempia. Huoneiden yhdistämisperusteet voisi myös uudistaa esimerkiksi erottelemalla huoneet eri kategorioihin.

Tulen varmasti käyttämään proseduraalista generointia tulevaisuuden peliprojekteissani. On mahdollista, että kehitän luolastogeneraattoria pidemmälle, jotain peliprojektia varten. Esittämäni algoritmit ja menetelmät jättivät jo muutamia uusia ajatuksia muhimaan päähäni. Lisäksi on vielä monia proseduraalisen generoinnin menetelmiä, joihin en vielä kerennyt tässä opinnäytetyössä tutustumaan.

LÄHTEET

Adonaac, A. 3.9.2015. Procedural Dungeon Generation Algorithm. Gamasutra. Viitattu 27.5.2017, http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php.

Biagioli, A. 9.8.2014. Understanding Perlin Noise. Viitattu 19.5.2017, http://flafla2.github.io/2014/08/09/perlinnoise.html.

Dinh, P. 3.5.2013. My Procedural Dungeon Generation Algorithm Explained. Viitattu 22.5.2017, https://www.reddit.com/r/proceduralgeneration/comments/1dlyow/my_procedural_dungeon_generation_algorithm/.

Kun, J. 29.7.2012. The Cellular Automaton Method for Cave Generation. Viitattu 19.5.2017, https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/.

Lague, S. 2017. Procedural Cave Generation tutorial. Unity Technologies. Viitattu 24.5.2017, https://unity3d.com/learn/tutorials/projects/procedural-cave-generation-tutorial.

Unity Technologies. 2017a. Unity. Viitattu 25.5.2017, https://unity3d.com/unity.

Unity Technologies. 2017b. Scenes. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/CreatingScenes.html.

Unity Technologies. 2017c. GameObjects. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/GameObjects.html.

Unity Technologies. 2017d. Introduction to components. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/Components.html.

Unity Technologies. 2017e. Scripting. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/ScriptingSection.html.

Unity Technologies. 2017f. Prefabs. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/Prefabs.html.

Unity Technologies. 2017g. Colliders. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/CollidersOverview.html.

Unity Technologies. 2017h. Scriptable Object. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/class-ScriptableObject.html.

Unity Technologies. 2017i. Bounds. Viitattu 29.5.2017, https://docs.unity3d.com/ScriptReference/Bounds.html.

Unity Technologies. 2017j. Coroutines. Viitattu 29.5.2017, https://docs.unity3d.com/Manual/Coroutines.html.

Seredynski, M. 8.2.2014. Bake Your Own 3D Dungeons With Procedural Recipes. Envato Tuts+. Viitattu 24.5.2017, https://gamedevelopment.tutsplus.com/tutorials/bake-your-own-3d-dungeons-with-procedural-recipes--gamedev-14360.

Weisstein, E. 2017. Cellular Automaton. Viitattu 19.5.2017, http://mathworld.wolfram.com/CellularAutomaton.html.

Wikidot. 2017. What is Procedural Content Generation. Procedural Content Generation Wiki. Viitattu 30.5.2017, http://pcg.wikidot.com/.

Wikipedia. 2017a. Procedural Generation. Viitattu 17.5.2017, https://en.wikipedia.org/wiki/Procedural_generation.

Wikipedia. 2017b. Perlin Noise. Viitattu 19.5.2017, https://en.wikipedia.org/wiki/Perlin_noise.

Wikipedia. 2017c. Delaunay Triangulation. Viitattu 20.5.2017, https://en.wikipedia.org/wiki/Delaunay_triangulation.

Wikipedia. 2017d. Minimum Spanning Tree. Viitattu 20.5.2017, https://en.wikipedia.org/wiki/Minimum_spanning_tree.

Wikipedia. 2017e. Rogue (video game). Viitattu 22.5.2017, https://en.wikipedia.org/wiki/Rogue_(video_game).

Wikipedia. 2017f. The Elder Scrolls II: Daggerfall. Viitattu 22.5.2017, https://en.wikipedia.org/wiki/The_Elder_Scrolls_II:_Daggerfall.

Wikipedia. 2017g. Floating-Point arithmetic. Viitattu 29.5.2017, https://en.wikipedia.org/wiki/Floating-point_arithmetic.