Jimi Jokela

Proseduraalinen pelikenttien generointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

30.11.2022

Tiivistelmä

Tekijä:	Jimi Jokela
Otsikko:	Proseduraalinen pelikenttien generointi
Sivumäärä:	42 sivua
Aika:	30.11.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoituksena oli suunnitella ja toteuttaa proseduraalisesti pelikenttiä generoiva algoritmi, eli kenttägeneraattori, peliprototyyppiin sekä tutkia pelikenttien generoinnissa käytettäviä menetelmiä ja algoritmeja. Lisäksi tavoitteena oli saada hyvä yleiskäsitys proseduraalisesta sisällön generoinnista ja siihen kuuluvien algoritmien ja menetelmien määrittelystä ja vertailusta ominaisuuksien, käyttökohteiden sekä hyötyjen ja haittojen osalta.

Insinöörityön kenttägeneraattori suunniteltiin roguelike-genren peliprototyyppiin, jota kehitettiin samaan aikaan. Kenttägeneraattorin tavoitteena oli tuottaa monipuolisia, pelattavia ja mielenkiintoisia pelikenttiä ruudukkopohjaiseen peliin. Pelikentän tuli koostua sokkelon muodostavista huoneista, joiden läpi kulki reitti yhdestä sisäänkäynnistä yhteen uloskäyntiin. Toteutettuun kenttägeneraattoriin valikoitiin soluautomaattialgoritmi tuomaan haluttuja piirteitä.

Insinöörityössä tutkittiin proseduraalisen sisällön generoinnin menetelmiä ja algoritmeja ja niistä erityisesti tiettyjä valikoituja ruutupohjaisten ja tyrmätyyppisten kenttien generointiin soveltuvia algoritmeja. Tutkituista algoritmeista soluautomaattia käytettiin myös toteutetussa kenttägeneraattorissa.

Insinöörityön tuloksena toteutettiin toimiva kenttägeneraattori, joka tuotti pelattavia ja monipuolisia pelikenttiä. Soluautomaatti lisäsi positiivista monipuolisuutta osana kenttägeneraattoria, mutta vähensi myös tulosten ennustettavuutta ja vaati mukautusta huonepohjaisessa kenttägeneraattorissa ongelmien välttämiseksi.

Avainsanat: proseduraalinen sisällön generointi, pelikenttien generointi, soluautomaatti, satunnaisuus

Abstract

Author:	Jimi Jokela
Title:	Procedural game level generation
Number of Pages:	42 pages
Date:	30 November 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Game Applications
Supervisor:	Miikka Mäki-Uuro, Senior Lecturer

The objective of this thesis was to design and implement a procedural level generation algorithm, or a level generator, for a game prototype, and to research algorithms and methods used in level generation. An additional objective was to gain an overview of procedural content generation and the specification and comparison of PCG algorithms and methods according to their properties, applications as well as their pros and cons.

The level generator of the thesis was designed for a roguelike game prototype, that was developed simultaneously. The goal of the level generator was to produce diverse, playable, and interesting levels for a grid-based game. A level should consist of rooms forming a maze with a route through from one entrance to one exit. Cellular automata were chosen for the implemented level generator to add the desired features.

Procedural content generation methods and algorithms were researched for the thesis, especially select algorithms suitable for generating grid-based dungeon levels. Cellular automata were chosen out of the researched algorithms to be used in the implemented level generator.

The thesis resulted in a working level generator that can generate playable and versatile levels. Cellular automata added positive variety as part of the level generator, but also reduced the predictability of the results, and required modifications in the room-based level generator to avoid problems.

Keywords: Procedural content generation, Level generation, Cellular automata, Randomness

Sisällys

Lyhenteet

ASP: *Answer Set Programming.* Hakupohjainen ratkaisu pelikenttien generointiin, jossa algoritmi hakee täydellisestä vastausten joukosta optimaalisimman vaihtoehdon.

CA: *Cellular Automata*. Soluautomaatti. Matemaattinen laskentamalli, jolla on monia käyttökohteita mukaan lukien pelikenttien generointi.

Dungeon: Tyrmä, erityisesti roguelike-genren pelien kontekstissa. Viittaa huoneista ja luolista koostuvaan suljettuun pelikenttään, jossa on vihollisia, aarteita sekä rajattu määrä sisään- ja uloskäyntejä.

PCG: *Procedural Content Generation*. Proseduraalinen sisällön generointi, yleisesti generointi. Sisältöä generoivaa algoritmia kutsutaan yleisesti generaattoriksi.

Roguelike: Peligenre, jonka nimi viittaa Rogue-pelin kaltaisiin peleihin. Käsite on laajentunut viittaamaan moniin roolipeleihin, jotka jakavat ainakin osittain tiettyjä piirteitä, kuten proseduraalisesti generoidut pelikentät tai tyrmät, satunnaisuus ja pysyvästi kuoleva pelihahmo (engl. permadeath). [Zapata 2017.]

1 Johdanto

Proseduraalinen pelikenttien generointi on suosittu tekniikka pelikenttien luomisessa tai osana sitä monissa nykyajan peleissä. Pelikenttien generointi on ollut suosittua jo 1980-luvulta lähtien roguelike- ja roolipeleissä, joissa pelaaja usein tutkii tyrmätyyppisiä pelikenttiä. Proseduraalisen generoinnin käyttökohteet ovat kuitenkin laajentuneet monenlaisiin peleihin, esim. No Man’s Sky -pelin generoitu galaksi ja sen monet generoidut planeetat, kasvit ja eläimet. [Moric 2016.]

Insinöörityön tarkoituksena on tutkia proseduraalisen pelikentän generoinnin suunnittelu- ja toteutusprosessia keskittyen ruutupohjaiseen pelikenttään ja toteuttaa toimiva kenttägeneraattori roguelike-peliin. Työssä selvitetään generointiin käytettäviä algoritmeja ja tekniikoita, niiden hyötyjä ja haittoja sekä tyypillisiä tuloksia.

Työssä tarkastellaan proseduraalisen sisällön generoinnin määritelmiä, menetelmiä ja algoritmeja etenkin pelikenttien ja tyrmien (engl. dungeon) generoinnin kannalta. Tyrmä viittaa rajattuun, huoneista tai luolista koostuvaan pelikenttään, ja termi on yleinen roguelike- ja roolipeligenreissä.

Työssä toteutetaan kenttägeneraattori roguelike-peliä varten. Generaattoriin valikoidaan esitellyistä algoritmeista ja tekniikoista sopivimmat pelin ennalta määritettyjen tavoitteiden toteutumiseksi sisällön rajoitukset huomioon ottaen.

Työ on jaettu aiheittain lukuihin, jotka etenevät teoreettisesta tutkimuksesta soveltavaan toteutukseen. Luku 2 käsittelee proseduraalista sisällön generointia yleisellä tasolla, sen tekniikoiden ja algoritmien luokittelua sekä yleisiä ominaisuuksia. Luvussa 3 esitetään työssä tutkittuja algoritmeja, joita käytetään erityisesti ruudukkopohjaisten ja tyrmätyyppisten pelikenttien generointiin. Luvussa 4 selostetaan insinöörityön kenttägeneraattorin toiminta ja toteutus. Lopuksi tarkastellaan kenttägeneraattorin tuloksia ja niiden kelpoisuutta.

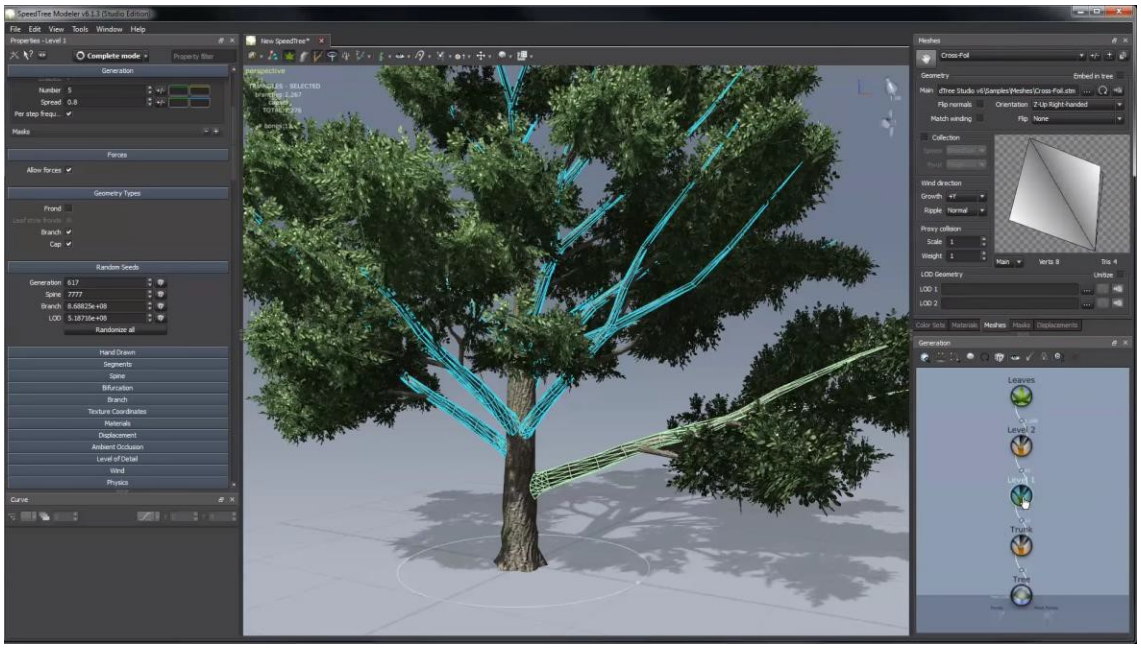
2 Proseduraalinen sisällön generointi

Proseduraalinen sisällön generointi (engl. procedural content generation, PCG) tarkoittaa sisällön luomista peliin algoritmisesti eli koodilla. Proseduuri viittaa suoritettavaan ohjeeseen, eli käytännössä ohjelmointikoodiin. Sisältö viittaa generoinnin tuloksena saatuun dataan, joka on käyttötarpeeseensa soveltuvassa muodossa. Generoimalla voidaan luoda monipuolisesti sisältöä. Erityisesti pelikontekstissa generoitu sisältö voi olla

- 3D-malleja
- tekstuureja
- ääniä
- dialogia
- tehtäviä
- esineitä
- hahmoja
- pelikenttiä. [Watkins 2016.]

Generoinnin avuksi on paljon valmiita algoritmeja ja tekniikoita, kuten soluautomaatit, Perlin-kohina, fraktaalit jne. Luvussa 3 käydään läpi tyrmätyyppisten pelikenttien generoinnissa käytettäviä algoritmeja. Kokonaisia kenttägenerointialgoritmeja pääsee tarkastelemaan helposti mm. klassisten roguelike-pelien pitkään jatkuneen avoimen lähdekoodin perinteen vuoksi, ja markkinoilta löytyy monia valmiita ratkaisuja osittaiseen ja kokovaltaiseen pelikentän generointiin. Valmiiden algoritmien avulla monimutkaisten ominaisuuksien lisääminen kenttägeneraattoriin helpottuu huomattavasti. [Algorithms for Procedural Content Generation 2019.]

Generointi on hyödyllinen työkalu, kun tavoitteena on luoda monipuolista, monimutkaista tai suuri määrä sisältöä. Tekniikkaa voi hyödyntää joko pelin kehitysvaiheessa tai ajonaikaisesti. Kehitysvaiheessa sisältöä voi generoida kokonaan tai osittain ja tuloksista voi poimia ja muokata mieleisensä osat lisättäväksi peliin (kuva 1). [Doull 2008.]



Kuva 1. Speedtree-ohjelmistolla voi luoda 3D-malleja puista generointityökaluilla.

Ajonaikainen generointi luo sisällön peliohjelman ajon aikana loppukäyttäjän laitteella. Tällöin tuloksia voidaan luoda lähes rajattomasti, mutta kehittäjät eivät voi tarkastella tuloksia yksilökohtaisesti ja tarvittaessa muokata niitä, vaan tulosten valikointi ja muokkaus on sisällytettävä osaksi generointialgoritmia. Ajonaikainen generointi säästää lisäksi tallennustilaa, koska generoitava sisältö ei ole valmiiksi mukana datana ohjelmassa (kuva 2). [Procedural Content Generation 2007.]

Kuva 2. .kkrieger-pelissä kaikki pelin sisältö eli 3D-mallit, tekstuurit, äänet jne. on luotu proseduraalisesti generoimalla. Näin pelin kokonaiskoko kiintolevyllä on vain 97 280 tavua. [.kkrieger 2004.]

2.1 PCG:n luokittelu

Proseduraalisen sisällön generoinnin (engl. procedural content generation, PCG) algoritmien ja menetelmien määrittelemiseen ja luokittelemiseen on esitetty taksonomisia luokkia, joissa generoinnin menetelmät asetetaan tiettyjen piirteiden mukaan ääripäiden välille. Luokat ovat hyödyllisiä PCG:n monimuotoisuuden ymmärtämisessä ja kartoittamisessa. Taksonomiset luokat ovat Shakerin ym. (2016) mukaan

- online vs. offline
- välttämätön vs. valinnainen (engl. necessary vs. optional)
- hallinnan määrä ja ulottuvuudet (engl. degree and dimensions of control) eli satunnainen siemenluku vs. parametrisointi

- geneerinen vs. mukautuva (engl. generic vs. adaptive)
- stokastinen vs. deterministinen (engl. stochastic vs. deterministic)
- rakentava vs. generoi ja testaa (engl. constructive vs. generate and test)
- automaattinen generointi vs. sekoitettu luominen (engl. automatic generation vs. mixed authorship).

Proseduraalista generointia voidaan hyödyntää pelin kehitysvaiheessa (offline), jolloin generoitua tulosta voidaan muokata ja jalostaa ja peliin sisällytettävä lopputulos on käynyt läpi ihmissuunnittelijan tarkastuksen. Proseduraalinen generaattori on täten työkalu muiden joukossa. Toisessa ääripäässä generaattori voi olla osa julkaistua peliä ja generointi tapahtuu käyttäjän laitteella pelin ajon aikana (online). Ajonaikainen generointi on yleinen tapa lisätä sisällön määrää ja vaihtelua generoimalla joka pelikerralla eri pelikenttä tai muuta sisältöä.

Välttämätön generointi tarkoittaa, että generaattorin tulokset ovat olennainen osa pelin sisältöä. Välttämättömälle generoinnille on asetettava tiukat laatuvaatimukset, jotta generointi on aina onnistunutta ja tulokset toimivat pelin kannalta, ettei generaattorin vuoksi synny pelissä etenemistä estäviä ongelmatilanteita, esim. generoitu sokkelopelikenttä, jonka läpi ei löydy validia reittiä. Välttämättömän generoinnin vastakohta on valinnainen generointi, jossa tulokset eivät ole pelin etenemisen kannalta olennaisia, esim. graafisia efektejä tai taustakuvia.

Hallinnan määrä ja ulottuvuudet generoinnissa toteutuvat parametrisoinnissa. Generaattori on harvoin täysin parametriton, sillä tuloksen hallinta olisi erittäin rajoitettua. Yleisimpiä parametreja on satunnaislukugeneraattorin siemenluku, mutta myös se voidaan valita satunnaisesti esim. kellonajasta ajohetkellä. Generaattorin parametreille voidaan myös määrittää ääriarvot, joiden välillä voidaan luottaa generaattorin tuottavan tarpeeksi vaihtelua ja samalla poissulkea ei-toivotut piirteet (ks. luku 2.4 Pseudosatunnaisuus, siemenluvut ja parametrisointi). [Shaker ym. 2016.]

Proseduraalinen generointi on useimmiten geneeristä, eli lopputulos ei muutu käyttäjän toiminnan perusteella. Generointi voi kuitenkin olla myös adaptiivista, jolloin käyttäjän syötteet ja vuorovaikutus pelin kanssa voivat suoraan vaikuttaa

generoitavaan sisältöön. Peli voi esim. generoida seuraavan pelikentän helpommaksi, jos pelaajalla on liikaa vaikeuksia tai pelikentän läpikäymiseen kului enemmän aikaa kuin on tarkoitus.

Deterministinen generointi toteutuu, kun tietty lopputulos voidaan tuottaa uudelleen käyttämällä samoja parametrejä mukaan lukien siemenlukua. Stokastinen generointi taas voi tuottaa eri tuloksia, vaikka generoinnin lähtökohta olisi sama, jos generointiin sisältyy joka ajokerralla vaihtuvaa satunnaisuutta.

Rakentavat generaattorit on suunniteltu toteuttamaan tietty ennalta suunniteltu joukko vaiheita, minkä jälkeen generaattori tuottaa tuloksen, kun taas generoi ja testaa -tyypin generaattorit sisällyttävät lopputuloksen kelpoisuuden tai muiden ominaisuuksien testaamisen tarvittaessa toistettavaksi vaiheeksi generointiaskeleiden välille tai jälkeen. Generaattorit voivat täten jatkojalostaa omia tuotoksiaan tai löytää niiden joukosta parhaan tuloksen (ks. luku 3.4 Evoluutio ja hakupohjaiset ratkaisut) tai yksinkertaisesti validoida tuloksen toimivuuden.

Automaattinen generointi toteutuu, kun generaattori toimii itsenäisesti lopputuloksen tuottamiseen asti ja suunnittelijan hallinta muodostuu parametrien tai itse algoritmin muuttamisesta. Sekoitettu luominen generoitaessa toteutuu, kun suunnittelija tai käyttäjä on suorassa vuorovaikutuksessa generaattorin kanssa, esim. generaattori täydentää käyttäjän suunnittelemaa pelikenttää tai käyttäjä muokkaa generaattorin tuloksia kesken generoinnin. [Shaker ym. 2016.]

2.2 Teleologinen ja ontogeneettinen lähestymistapa

Teleologinen (eli simuloiva) lähestymistapa perustuu generoitavan aiheen luonnollisen muodostumisen simuloimiseen. Luonnollinen prosessi pyritään toistamaan mahdollisimman tarkasti vaihe vaiheelta. Lopputulos on siten sitä lähempänä aiheen todellisuutta, mitä tarkemmin aiheen syntyprosessia simuloidaan. Esim. maaston generointi aloitetaan simuloimalla mannerlaattojen tektoniikkaa, eri maalajien eroosiota tuulen, sateiden ja veden virtausten vaikutuksesta ja

muita luonnollisia maastoa muokkaavia ilmiöitä. [West 2008; Teleological vs. Ontogenetic 2020.]

Teleologisten algoritmien etuna ovat luonnollisemmat ja myös yllätyksellisemmät tulokset, mutta tulosten hallinta tai ohjaaminen tiettyyn suuntaan on vaikeampaa, sillä simulaatioiden parametrien muuttaminen vaikuttaa vain välillisesti tuloksiin ja yllättävyys voi muodostua ongelmaksi. [Teleological 2009.]

Ontogeneettinen (eli lopputulosta matkiva) lähestymistapa kiinnittää huomiota vain generoitavan aiheen lopulliseen muotoon ja pyrkii matkimaan lopputulosta välittämättä sen syntyprosessista. Tällöin yksinkertaisemmat ratkaisut voivat toteuttaa halutun lopputuloksen helpommin kuin teleologiset menetelmät ja tulokset ovat ennalta-arvattavampia. Myös niiden hallinta parametreillä on selkeämpää, kun parametrit liittyvät suoraan lopputulokseen. Toisaalta liika ennalta-arvattavuus saattaa vähentää positiivista yllättävyyttä ja monipuolisuutta. [West 2008.]

Eri algoritmit voivat soveltua paremmin tiettyyn lähestymistapaan tai olla osana kumman tahansa lähestymistavan generaattoria. Lähestymistavan valinta riippuu tavoiteltavan lopputuloksen luonteesta, esim. tavoitellusta realistisuudesta luonnolliseen maastoon verrattuna.

2.3 PCG:n hyödyt ja ongelmat käsintehtyyn verrattuna

Proseduraalisen generoinnin suurimmat hyödyt ovat

- suuri sisällön määrä
- monipuolisuus ja vaihtelevuus
- positiivinen ennalta-arvaamattomuus ja yllätyksellisyys
- suunnittelijoiden ja artistien työn määrän väheneminen.

Generoinnin suurimmat ongelmat ovat

- käsin tehdyn sisällön tarkoituksellisempisuus

- hallinnan puute
- negatiivinen ennalta-arvaamattomuus
- ohjelmoijien työn määrän kasvu.

Generoimalla voidaan luoda erittäin suuria määriä sisältöä. Generoitu pelikenttä voi olla joka pelikerralla uniikki. Pelikentät voivat olla hyvin erilaisia ja monipuolisia. Sisällön monipuolisuuden ja tulosten hallinnan välille muodostuu kuitenkin ristiriita, josta kehittäjän on löydettävä tasapaino. Tulosten hallinnan eli tietynlaisten tulosten saamiseksi algoritmin parametreja on kavennettava, mikä vähentää vaihtelevuutta. Parametrejä kavennettaessa tuloksista tulee homogeenisempia, jolloin menetetään monipuolisuutta. Toisaalta parametrien laajentaminen lisää monipuolisuutta, mutta samalla vähentää hallintaa ja tulosten tarkoituksellisuutta eli sitä, kuinka lähellä sisältö on suunnittelijan tarkoittamaa lopputulosta. [Devs weigh in on the best ways to use (but not abuse) procedural generation 2018.]

Ennalta-arvaamattomuus voi olla samaan aikaan hyöty ja ongelma. Generaattorin luodessa massiivisia määriä tuloksia algoritmin suunnittelija ei voi tarkastella tai tietää tarkalleen kaikkia lopputuloksia. Generaattori voi siis tuottaa tuloksia, joita edes suunnittelija ei ole ennalta-arvannut ja jotka täten voivat olla mielenkiintoinen yllätys. Ennalta-arvaamattomat tulokset voivat kuitenkin myös olla pelin kannalta ongelmallisia. Generoitu pelikenttä voi pahimmillaan olla pelaamiskelvoton, jos esim. jokin reitti on täysin tukittu satunnaisesti luotujen seinien vuoksi. Toimimattomien tulosten suodattaminen pois onkin keskeinen haaste algoritmia luotaessa. Yksi ratkaisu on suorittaa ylimääräinen validointivaihe, jossa arvioidaan generoidun tuloksen kelpoisuutta, ja jos havaitaan ongelmakohtia, ne voidaan täsmällisesti korjata tai todennäköisemmin generointi aloitetaan uudestaan alusta eri siemenluvulla. [Procedural Content Generation 2007.]

Generaattorin puutteita voidaan paikata lisäämällä algoritmiin erillisiä vaiheita, jotka muokkaavat tai suodattavat tuloksia. Tällöin generaattorin parametrin voidaan pitää mahdollisimman laajoina ja täten tulokset ovat mahdollisimman monipuolisia ja vaihtelevia, mutta epäkelvolliset ja ongelmalliset tulokset eivät ilmaannu sellaisenaan.

Proseduraalisella sisällön generoinnilla voidaan vähentää artisteille ja suunnittelijoille tulevaa työtä korvaamalla se generaattorin tekemällä työllä. Toisaalta generaattorin kehittämisestä tulee lisää työtä ohjelmoijille. Sisällön määrän kasvaessa lineaarisesti artistien työn määrä kasvaisi myös lineaarisesti, mutta generoitaessa ohjelmoijien työn määrä kasvaa vain tiettyyn pisteeseen asti, minkä jälkeen sisällön määrä voi kasvaa ilman merkittävää lisätyöpanosta. Pienille määrille sisältöä kokonaistyömäärä voi silti olla käsin tehtynä merkittävästi pienempi. Ottaen huomioon edellä mainitut muut käsin tekemisen edut, se sopii generointia paremmin, kun sisältöä tarvitaan suhteellisen vähän. Generointi taas on sitä edullisempaa, mitä enemmän sisältöä vaaditaan. [Procedural Content Generation 2007.]

Pelin ydinominaisuuksien ja tavoitteiden tulisi ohjata hyvän kenttägeneraattorin suunnittelua. Generaattorin luomien kenttien on tuettava pelikokemusta ja olla mielenkiintoisia. Generoituja kenttiä on tarpeen testata pelaamalla, mikä mahdollistaa iteroimisen generaattoria kehittäessä. Iteroimalla ja testaamalla voidaan korjata monia generoinnin ongelmakohtia ja varmistaa tulosten toimivuus. [Yu 2016: 34–35, 39.]

2.4 Pseudosatunnaisuus, siemenluvut ja parametrisointi

Satunnaislukugeneraattorit ovat työkalu satunnaiselementin lisäämiseksi proseduraaliseen generointiin. Satunnaislukugeneraattori luo lukujonon, joka koostuu pseudosatunnaisluvuista eli satunnaiselta vaikuttavista luvuista. Pseudosana viittaa todellisen satunnaisuuden saavuttamisen vaikeuteen tietokoneilla. Todellisesti satunnaisena pidetään esim. nopan heittoa, jonka tuloksen määrittävät erittäin monimutkaiset fysiikan lait, jolloin tulevia tuloksia ei ole käytännössä mahdollista laskea etukäteen nykyteknologialla. Satunnaislukugeneraattori sen sijaan luo suhteellisen yksinkertaisella kaavalla pitkän lukujonon, joka määrittelee kaikki tulevat tulokset generaattorista. [Watkins 2016.]

Satunnaislukugeneraattorin tulokset vaihtelevat riippuen käytetystä siemenluvusta. Siemenluku on luku, joka syötetään alkuarvoksi generaattoriin. Samalla

siemenluvulla saadaan aina sama lukujono tulokseksi generaattorista. [Weisstein 2022a.]

Siemenluku voitaisiin ennalta määrittää, jolloin tulos on aina sama, kun ohjelma ajetaan. Sen sijaan tavoiteltaessa vaihtelua siemenlukuna on tapana käyttää järjestelmän kellonaikaa, joka on jatkuvasti vaihtuva luku ja siten generaattorin tuloksetkin vaihtuvat. Tietokone laskee järjestelmäaikaa tikkeinä (engl. ticks), jotka ovat noin millisekunnin luokkaa. Täten järjestelmäajasta otettu siemenluku on käytännössä jokaisella ohjelman ajokerralla uusi. [Weisstein 2022a; System Time 2018.]

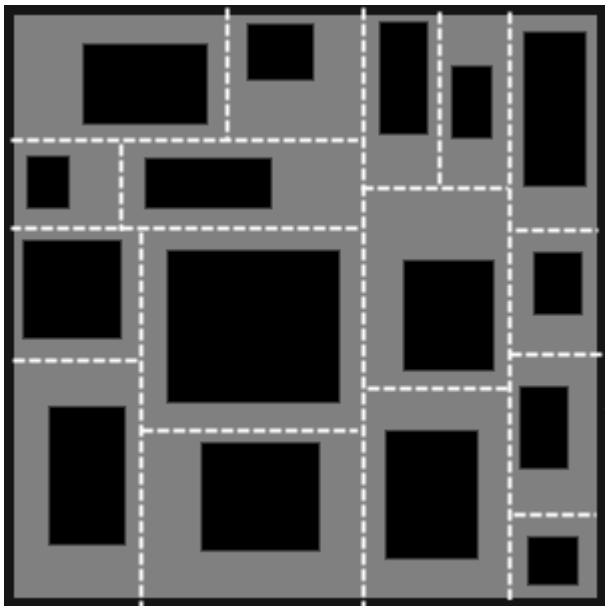
Käyttäjän tai pelaajan voidaan myös antaa määritellä käytettävä siemenluku, jolloin pelaajat voivat jakaa pelaamiansa pelikenttiä muille jakamalla käytetyn siemenluvun, jolla peli luo koneesta riippumatta saman pelikentän. Pelaaja voi myös haluta pelata samaa pelikenttää uudestaan löytäessään erityisen mielenkiintoisen tai haastavan pelikentän luovan siemenluvun.

3 Pelikenttien generoinnissa käytettäviä algoritmeja

Insinöörityön osana tutkittiin valikoituja tyrmätyyppisten pelikenttien generoinnissa käytettäviä algoritmeja ja niiden tuloksia.

3.1 Tilan jakaminen

Tilan jakaminen (engl. space partitioning) on yksinkertainen huoneista koostuviin kaksiulotteisiin ruudukkopohjaisiin pelikenttiin sopiva generointialgoritmi. Laajemmin tilan jakaminen voi myös viitata mm. kuvien ja 3D-mallien käsittelyssä käytettävään algoritmiin. Algoritmi perustuu nimensä mukaisesti annetun tilan jakamiseen osiin, joita jokaista edelleen rekursiivisesti jaetaan osiin, kunnes tietyt loppuehdot täyttyvät, esim. tietty määrä jakoja. Yleinen ja yksinkertaisin muoto, binäärinen tilan jakaminen (engl. binary space partitioning), jakaa tilaa aina kahteen osaan satunnaisesta kohdasta joko pysty- tai vaakatasossa. Seuraava iteraatio jakaa kunkin osan vastaavasti toisella akselilla. Kun osia edelleen jaetaan, voidaan määrittää todennäköisyys, että osan jakaminen ei enää toteudu. Jakamisen loputtua jokaiseen osaan voidaan asettaa huone, joko täyttäen koko osan tai satunnaisen suorakulmion osan sisältä (kuva 3). Lopulta huoneet voidaan yhdistää satunnaisista kohdista käytävillä ja näin luoda toimiva sokkelomainen pelikenttä. [Shaker ym. 2016: 33–35.]



Kuva 3. Esimerkki binäärisellä tilan jakamisella luoduista huoneista [Basic BSP Dungeon Generation 2020].

Yleinen, yksinkertainen algoritmin toteutus tuottaa rakenteellisen, järjestäytyneen tyrmätyylisen pelikentän. Huoneet eivät osu päällekkäin, ja erikokoisten huoneiden määrät ovat helposti säädettävissä. Sellaisenaan pelikentissä vaihtelevuus ja monipuolisuus jää vähäiseksi, mutta menetelmän pohjalta jatkokehitys voi lähteä moneen suuntaan, esim. huoneiden generoinnissa ja yhdistämisessä. [Niemann 2015.]

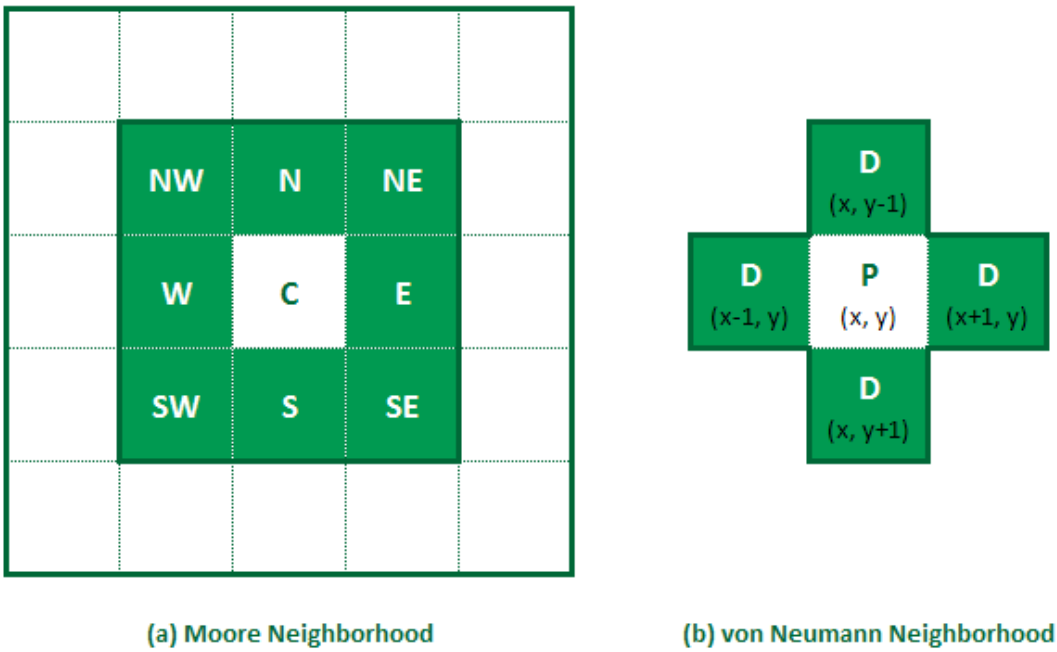
3.2 Agenttipohjainen kasvu

Agenttipohjainen kasvualgoritmi generoi pelikentän tekoälyllä ohjattavan agentin avulla. Agentti kulkee ruudukossa tekoälyalgoritmin ohjaamana ja luo käytäviä kulkiessaan ja huoneita ympärilleen tietyllä todennäköisyydellä. Menetelmä tuottaa luonnollisia, järjestelemättömiä muodostelmia. Satunnaisuuden määrällä voi suoraan vaikuttaa lopputuloksen kaoottisuuteen. Kehittyneempi versio voi tutkia ympärillään olevia ruutuja ja täten välttää huoneiden päällekkäisyyksiä, jolloin tuloksena on selkeämpi ja rakenteellisempi pelikenttä. [Shaker ym. 2016: 38–39.]

3.3 Soluautomaatti

Soluautomaatti (engl. Cellular Automata) on laskentamalli ruudukossa tai muussa järjestyksessä oleville soluille, jotka kehittyvät naapureidensa perusteella tiettyjen sääntöjen mukaan [Kozliner 2019]. Tietyn soluautomaatin ominaisuudet ovat solujen saamat arvot, tarkastelu-ulottuvuudet ja säännöt, joiden perustella solu saa arvon. Eri soluautomaatteja voidaan määritellä rajattomasti lisäämällä ulottuvuuksia tai mahdollisia arvoja, mutta yleisesti tarkastelussa ovat yksinkertaiset järjestelmät, etenkin kaksiulotteiseen ruudukkoon sijoittuvat, kaksiarvoiset eli binääriset soluautomaatit. Yksinkertaisimmatkin soluautomaatit tuottavat hyödyllisiä ja mielenkiintoisia piirteitä, kuten fraktaalista toistuvuutta, satunnaisuutta tai monimutkaisia, muotoa vaihtavia kuvioita. Soluautomaatti soveltuu laajasti eri tieteenaloilla monipuolisiin ongelmiin, kuten nesteiden, kaasujen, kristallirakenteiden kasvun ja biologisten rakenteiden simulointiin. [Wolfram 1983.]

Yleiset kaksiulotteisessa ruudukossa toimivien soluautomaattien tarkastelu-ulottuvuudet tai naapurustot ovat niin sanotut von Neumannin naapurusto (engl. von Neumann neighbourhood) ja Mooren naapurusto (engl. Moore neighbourhood). Kun soluautomaatti tarkastelee käsiteltävänä olevan ruudun viereisiä, eli suoraan ylä- ja alapuolella sekä vasemmalla ja oikealla olevia, neljää ruutua, puhutaan von Neumannin naapurustosta. Jos käsiteltävän ruudun koordinaatit ovat x ja y, von Neumannin naapurustoon kuuluisivat koordinaatit {(x-1, y), (x, y-1), (x+1, y), (x, y+1)} (kuva 4). Kun soluautomaatti tarkastelee ruudun ympäröiviä kahdeksaa ruutua, eli aiempien lisäksi myös kulmittain diagonaalisesti yhdistyviä ruutuja, puhutaan Mooren naapurustosta. Vastaavasti koordinaatit Mooren naapurustolle olisivat {(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y+1), (x+1, y-1), (x+1, y), (x+1, y+1)} (kuva 4). Molempia voidaan laajentaa myös tarkastelemaan naapurustoa kahden tai useamman ruudun etäisyydellä. [Shaker ym. 2016: 42.]



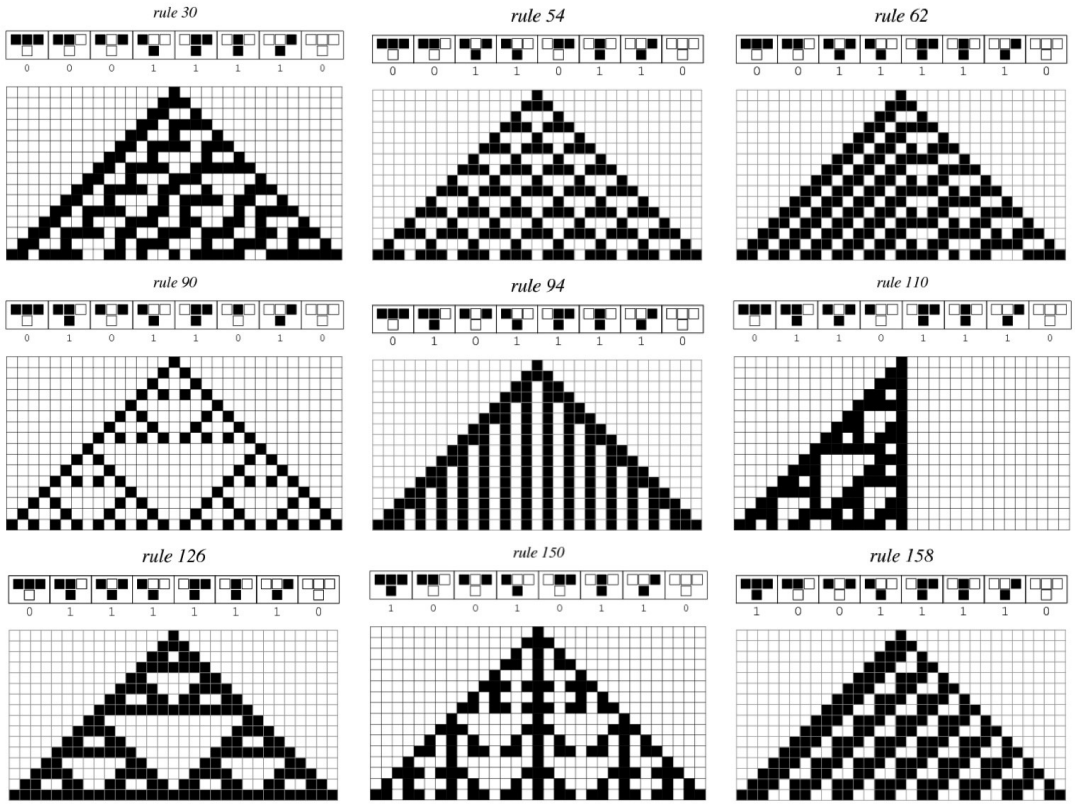
Kuva 4. Mooren naapurusto ja von Neumannin naapurusto. Naapurustot kuvaavat ympäröivien solujen tarkastelujoukkoja soluautomaatissa. [Pedersen 2014.]

Soluautomaatti on ruutupohjaisen pelikentän generoinnissa yleinen, monipuolinen ja laajasti hyödyllinen algoritmi. Soluautomaatin avulla voi luoda koko pelikentän esimerkiksi tavoiteltaessa luolamaisia pelikenttiä, tai algoritmia voi käyttää yksityiskohtien siivoamiseen, esimerkiksi yksittäisten ongelmallisten ruutumuodostelmien poistamiseen. Soluautomaatilla voi toteuttaa myös agenttipohjaisen kasvun generointialgoritmin (ks. luku 3.2). [Kozliner 2019.] Luvussa 4.2 käydään läpi insinöörityössä toteutettu soluautomaattialgoritmi.

Alkeissoluautomaatit

Alkeissoluautomaatit (engl. Elementary Cellular Automata) ovat yksinkertaisimpia soluautomaatteja, joilla on silti monia mielenkiintoisia piirteitä (kuva 5). 1980-luvulla matemaatikko Stephen Wolfram määritteli ja luokitteli alkeissoluautomaatit ja näki soluautomaattien potentiaalin monimutkaisten luonnollisten ilmiöiden tehokkaaseen simulointiin. Alkeissoluautomaatit perustuvat ruudukossa yksiulotteisesti eteneviin soluja määrittäviin sääntöihin, joilla jokainen solu saa binäärisen arvon yläpuolella olevien kolmen solun perusteella rivi tai sukupolvi

kerrallaan alaspäin kuljettaessa. Tällöin yhden alkeissoluautomaatin määrittelemiseen tarvitaan sääntöjä vain kahdeksan (yläpuolisten kolmen binäärisen arvon mahdollisia permutaatioita on 2 * 2 * 2 = 8). Tästä seuraa, että alkeissoluautomaatteja, eli niiden erilaisia sääntökokoelmia, on yhteensä 256 (kahdeksan säännön permutaatioita on 2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256). Jokainen alkeissoluautomaatti voidaan ilmaista kahdeksanbittisenä arvona eli lukuna välillä 0–256. [Weisstein 2022b.]



Kuva 5. Esimerkkejä eri alkeissoluautomaattien tuloksista, joista näkyvät ensimmäiset 16 sukupolvea eli kuvassa riviä. Sääntö 30, kuvassa vasen yläkulma, tuottaa pseudosatunnaisen binäärilukujonon tarkasteltaessa keskimmäistä pystyriviä. [00011110 2019.]

Jopa vierekkäiset eli vain yhden säännön osalta eroavat alkeissoluautomaatit saattavat olla perustavanlaatuisesti erilaisia. Joissain alkeissoluautomaateissa, esim. säännössä 30, esiintyy pseudosatunnaisuutta, ja niitä käytetään pseudosatunnaislukualgoritmeina. Toisissa esiintyy monimutkaisia muodostelmia tai

toistuvia kuvioita, kun taas joidenkin tulos kehittyy paljon sukupolvien kasvaessa, esim. monimutkaiset kuviot, jotka lopulta päättyvät yksinkertaisempiin toistuviin riveihin. [Weisstein 2022b.]

Conway’s Game of Life

Yksi tunnettu esimerkki yksinkertaisesta, mutta mielenkiintoisesta, soluautomaatista on matemaatikko John Conwayn 1970-luvulla kehittämä ”Elämän peli” tai yksinkertaisesti ”Elämä” (engl. ”Game of Life” tai ”Life”). Elämä on kaksiulotteinen Mooren naapuruston soluautomaatti. Pelaaja määrittää vain järjestelmän alkutilanteen eli ruudukon arvot. Alkuasetelmasta lähtien soluautomaatti muuttaa solujen arvoja sukupolvi kerrallaan. [Callahan 2005.]

Tyhjä solu elää eli saa arvon 1, jos ympärillä on tasan kolme elävää solua, elävät solut pysyvät elävinä, jos ympärillä on kaksi tai kolme elävää solua, ja muissa tapauksissa solu kuolee tai pysyy kuolleena eli saa arvon 0. Näistä yksinkertaisista säännöistä löytyi järjestelmä, joka voi alkuasetelmasta riippuen luoda monimuotoisia ja monimutkaisia muodostelmia, joiden muutokset ja liike sukupolvien kuluessa muistuttavat eläviä olentoja, kuten mikrobeja. Elämä sai nimensä, koska järjestelmä toteuttaa erään elämän määritelmän vaatimukset: sen muodostelmat pystyvät lisääntymään itsenäisesti ja peli pystyy simuloimaan Turing-konetta. [Callahan 2005.]

3.4 Evoluutio ja hakupohjaiset ratkaisut

Hakupohjaiset algoritmit käyttävät evaluointifunktioita ratkaisujen arvottamiseen. Suorat evaluointifunktiot laskevat suoraan arvoja lopputuloksesta, esim. asetettujen laattojen tai rakennusten määriä. Suoran evaluoinnin tuloksia voi olla vaikea yhdistää pelisuunnittelun ongelmiin. [Shaker ym. 2016.]

Simulaatiopohjaiset evaluointifunktiot käyttävät tekoälyä lopputuloksen arvioimiseen. Tekoäly suunnitellaan pelaamaan peliä pelaajan tavoin, esim. hakemaan

reitti sokkelon alusta loppuun, ja tekoälyn käyttäytymistä arvioidaan, esim. löytyykö reittiä ja kuinka pitkä se on tai kuinka moneen umpikujaan tekoäly päätyi matkalla. Interaktiiviset evaluointifunktiot toimivat pelaajan tai pelitestaajan mielipiteitä tai käyttäytymistä mittaamalla, esim. mitä generoiduista aseista pelaaja käyttää mieluiten. [Shaker ym. 2016: 24.]

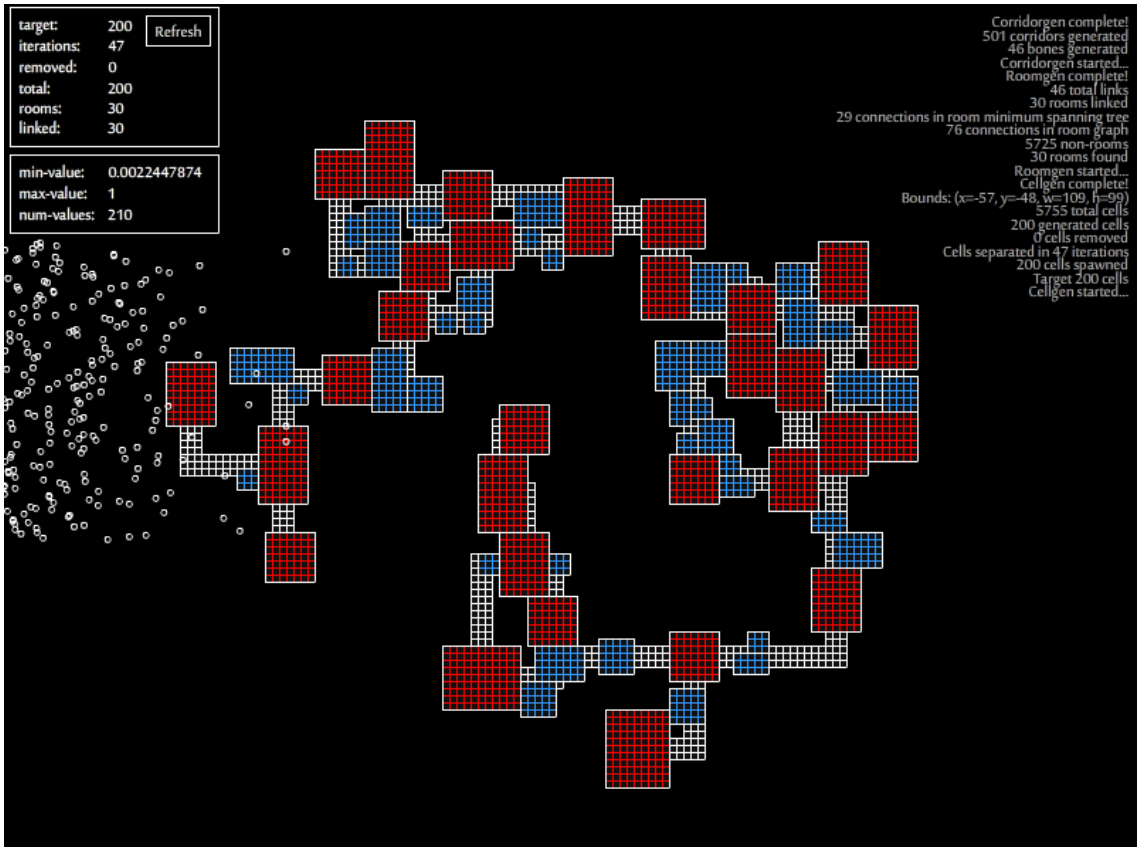
Hakupohjaisia ja evoluutioalgoritmeja voi käyttää mm. sokkeloiden luomiseen, mutta pitkän suoritusajan vuoksi ne sopivat offline-generointiin. Monimutkaisemissa pelikentissä nämä toteutukset voivat mennä erittäin monimutkaisiksi ja suoritukseltaan raskaiksi muihin menetelmiin verrattuna. [Siitonen 2017.]

ASP (answer set programming) soveltuu sokkelo ja tyrmä -tyyppisten pelikenttien generointiin etsimällä sopivan ratkaisun täydellisestä ratkaisujoukosta. ASP on lähestymistapa loogiseen ohjelmointiin, ja siinä määritellään rajoitukset ja loogiset suhteet, joiden perusteella hakualgoritmi löytää sopivimman ratkaisun. ASP-algoritmi siis käy läpi kaikki mahdolliset ratkaisun permutaatiot ja arvioi niiden kelpoisuutta. Huomioitavaa on, että kaksiulotteiseen ruudukkoon sijoittuvan pelikentän mahdolliset ratkaisut kasvavat logaritmisesti ruudukon kokoa kasvattaessa jokaisen lisätyn ruudun moninkertaistaessa vastausjoukon kokonaismäärän. Haun voi myös pysäyttää tietyn ajan kuluttua tai tietyn seulontamäärän täyttyessä, jolloin valitaan siihen mennessä sopivin ratkaisu. [Shaker ym. 2016: 143.]

3.5 Fysiikkapohjainen ratkaisu

Huoneista koostuva pelikenttä voidaan myös luoda fysiikkamoottoria käyttäen. Fysiikkamoottoria käytetään huoneiden asettelussa. Huoneet, tai huoneita tarkoittavat ympyrät, luodaan päällekkäin lisäämällä vain pieni satunnaisvaihtelu niiden sijaintiin. Huoneille asetetaan törmäytinkomponentit (engl. collider), ja fysiikkamoottori työntää törmäyttimet erilleen. Fysiikkasimulaatiota ajetaan, kunnes mitkään törmäyttimet eivät ole enää päällekkäin. Simulaation pysähdyttyä huoneet lukitaan ruudukkoon. Lopulta niistä voidaan valita satunnainen määrä haluttavin kriteerein, ja loput poistetaan (kuva 6). Huoneiden yhdistäminen ja

sokkelon muodostaminen voidaan toteuttaa Delauneyn kolmio -menetelmällä (engl. Delauney triangulation), joka muodostaa kaavion yhdistäen huoneet lähimpiin naapureihinsa, ja minimum spanning tree -algoritmilla, joka löytää lyhimmän reitin kaaviosta yhdistäen kaikki huoneet. Fysiikkapohjainen ratkaisu voi tuottaa suuria ja monimutkaisia sokkelomaisia pelikenttiä. [Procedural Dungeon Generation Algorithm 2015; Bradfield 2018.]



Kuva 6. Fysiikkapohjaisella kenttägeneraattorilla luotu tyrmäpelikenttä Tiny Keep -peliin [Procedural dungeon generation algorithm explained 2017].

4 Kenttägeneraattorin toteutus peliä varten

Insinöörityön osana toteutettiin kenttägeneraattori Scoundrel Squad -nimistä peliprototyyppiä varten (kuva 7). Peliprototyyppi suunniteltiin toteutuksen yhteydessä, joten kenttägeneraattorin suunnittelu oli hyvin avoin eri mahdollisuuksille. Pelimekaniikkojen selkeytyessä kenttägeneraattorille asetetut tavoitteet selkeytyivät myös. Pelin suunniteltiin olevan ruudukkoon pohjautuva vuoropohjainen taktinen cyberpunk roguelike, jossa pelaaja kontrolloi neljää pelihahmoa. Pelaajan tavoitteena on edetä pelikenttien läpi taistellen vihollishahmoja vastaan ja keräten uusia varusteita ja kykyjä kukistetuilta vihollisilta ja pelikentältä löytyvistä aarrelaatikoista ja kaupoista. Peli sijoittuisi tulevaisuuteen, joidenkin vuosikymmenien tai vuosisatojen päähän, ja pelihahmot saisivat käyttöönsä futuristisia aseita ja kyberneettisiä lisävarusteita, jotka antaisivat monenlaisia hyödyllisiä kykyjä.

Kuva 7. Scoundrel Squad -peli pelaajan näkökulmasta.

Roguelike-genren mukaisesti pelikenttien tuli olla satunnaisgeneroituja niin, että pelaaja kohtaisi uuden pelikentän joka pelikerralla. Pelikentän tuli muodostaa

yksinkertainen sokkelo, jossa olisi vähintään yksi reitti alkupisteestä loppupisteeseen, ja lisäksi monia sivuhaaroja reitissä, joista voisi löytyä lisää vihollisia, aarteita, parannuspisteitä tai kauppoja. Päätavoitteena oli mielenkiintoinen kokonaisuus, jota pelaajan olisi hauska tutkia ja kenttä olisi sopivan haastava päästä läpi.

Taksonomisesti toteutettu kenttägeneraattori voidaan määritellä seuraavasti: online, välttämätön, parametrisointi, geneerinen, stokastinen, rakentava, automaattinen. Generointi tapahtuu ajonaikaisesti pelissä (online). Kenttägeneraattori on olennainen osa pelin pelattavuuden kannalta (välttämätön). Kenttägeneraattori on parametrisoitu jokaisen vaiheen vaikutusten määrittämiseksi (parametrisointi). Generointi on geneeristä, eli pelaajan toiminta ei vaikuta kenttägeneraattorin tuloksiin (geneerinen). Generointi on pääasiassa parametrien mukaan determinististä, mutta jokaisessa vaiheessa on mukana satunnaiselementti. Satunnaisfunktiot ottavat siemenlukunsa tietokoneen kellonajasta eikä siemenlukua tallenneta, ja täten tuloksia ei ole suoraan mahdollista toistaa (stokastinen). Generaattori luo pelikentän vaiheittain pyrkien suoraan kelvolliseen lopputulokseen (rakentava), mutta sisällyttää myös joitain kelpoisuustestejä vaiheiden sisälle, kuten yksittäisten huoneiden generoinnissa. Generointi tapahtuu automaattisesti, eikä kehittäjä ole vuorovaikutuksessa generaattorin kanssa generoinnin aikana (automaattinen). Luokittelu on tosin enemmän tarkoitettu yksittäisten algoritmien määrittelemiseen eikä monia algoritmeja yhdistelevän generaattorin määrittelemiseen, mutta se antaa yleiskuvaa generaattorin piirteistä. [Shaker ym. 2016; Siitonen 2017.]

Projekti toteutettiin käyttäen Unity3D-pelimoottoria ja C#-ohjelmointikieltä. Unity3D:n C#-kirjastoista hyödynnettiin mm. Random-luokkaa pseudosatunnaisuuteen. Parametrisoinnin apuna käytettiin Unity3D:n ScriptableObject-luokkaa, jonka perivistä luokista voidaan luoda serialisoituja objekteja.

4.1 Valikoidut tekniikat

Kenttägeneraattorin tekniikoiden valikoinnissa priorisoitiin toteutettavuutta ja lopputuloksen pelattavuutta. Suunnittelun ideointivaiheessa käytiin läpi monenlaisia pelimekaanisia piirteitä, jotka jätettiin myöhempään kehitykseen, mm. lukitut ovet, tietokonehakkerointi ja muut vaiheittain avautuvat reitit.

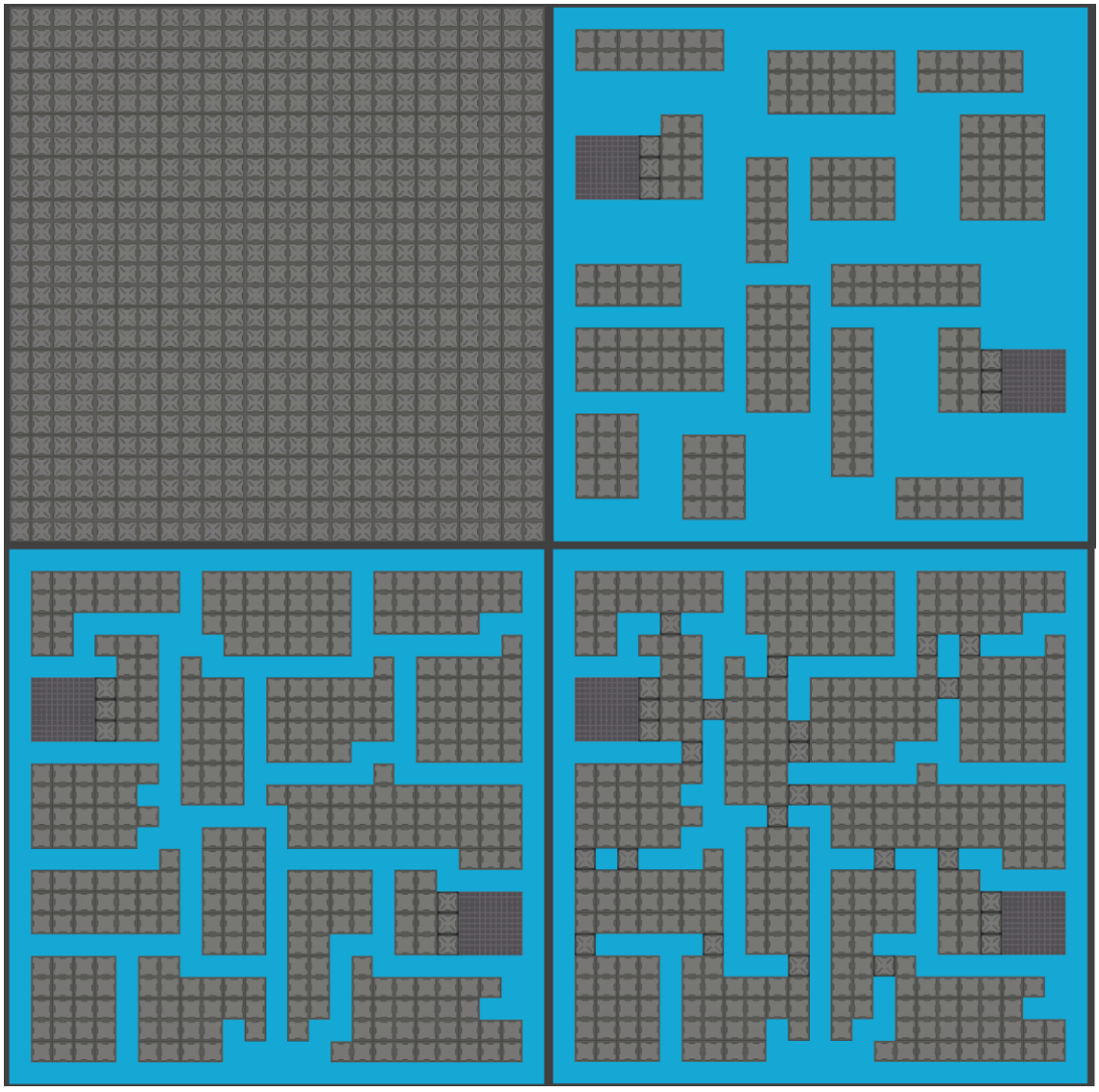
Pelikentän suunnitelma rajattiin yksinkertaiseen, huoneista koostuvaan, monireittiseen sokkeloon. Huoneet täyttäisivät pelikentän, ja ne yhdistettäisiin ovilla, jotka pelaajan hahmot voivat avata vuorovaikuttamalla niiden kanssa viereisestä ruudusta. Huoneet olisivat suorakulmion muotoisia, satunnaisesti generoitujen mittojen pituisia ja levyisiä, ja niiden välille jäisi ainakin yhden ruudun levyiset seinät. Ovet veisivät yhden ruudun, kuten myös erikoispaikat pelikentällä: aarrelaatikot, kaupat ja parannusautomaatit. Kaupat ja muut vuorovaikutuspisteet olisivat myyntiautomaattien kaltaisia laitteita, jolloin ne sopisivat järkevästi yhden ruudun sisään.

Huoneiden asettelussa päädyttiin toteuttamaan satunnaisesti suorakulmioita asettelemalla, vaikka tilanjakamisalgoritmi oli myös varteenotettava vaihtoehto. Satunnaisasettelussa joudutaan tekemään päällekkäisyystarkastuksia ja eri kokoisten huoneiden jakautumista on vaikeampi säätää.

Soluautomaatti valikoitiin toteutukseen, kun pelkät suorakulmioiden muotoiset huoneet jäivät helposti tylsäksi kokonaisuudeksi ja huoneiden välinen tyhjä tila oli epäluonnollisen oloinen, kun pelikentän on tarkoitus olla kerros pilvenpiirtäjästä. Huoneiden väliin jäävä tyhjä tila olisi pitänyt ottaa erikseen huomioon huoneiden yhdistämisessä. Soluautomaatin avulla huoneet voivat täyttää tyhjän tilan ja saada mielenkiintoisempia, tosin vähemmän rakenteellisia, muotoja. Yleisesti soluautomaatilla saadaan aikaan luolamaisia muodostelmia, mutta koska suorakulmioiden muotoiset huoneet asetellaan ennen soluautomaattivaihetta, ne muodostavat rajat, joihin soluautomaatilla laajennettavat huoneet mukautuvat. Lopputuloksena saadaan monipuolisesti muotoutuneita huoneita, jotka ovat vierekkäin toistensa kanssa ilman liikaa tyhjää tilaa.

4.2 Toteutus vaiheittain

Seuraavaksi käydään läpi yksittäin kenttägeneroinnin vaiheet ja niiden vaikutukset kenttään (kuva 8). Vaihekohtaiset havainnekuvat on erikseen luotu pysäyttämällä generointi kunkin vaiheen jälkeen. Normaalissa ajossa koodi käy läpi kaikki vaiheet käsitellen pelikenttää vain taulukkona ja 3D-mallit asetellaan vasta, kun kaikki vaiheet on saatu päätökseen.



Kuva 8. Pelikentän generoimisen neljän ensimmäisen vaiheen tulokset vierekkäin. Siniset ruudut kuvaavat seiniä ja harmaat ruudut lattioita tai ovia. Ylhäällä vasemmalla on ruudukon luonti, ylhäällä oikealla huoneiden luonti, alhaalla vasemmalla on huoneiden laajentaminen soluautomaatilla, ja alhaalla oikealla on huoneiden yhdistäminen.

Generoinnin vaiheet toteutettiin mahdollisimman modulaarisesti, mikä mahdollisti tietyn vaiheen vaikutusten eristämisen ja eri vaiheiden iteratiivisen kehittämisen. Vaiheet ovat ruudukon luonti, huoneiden luonti, mukautettu soluautomaatti, huoneiden yhdistäminen, esineiden ja vihollisten lisäys sekä mallien valinta, yhdistely ja asettelu.

4.2.1 Ruudukko ja tietotyypit

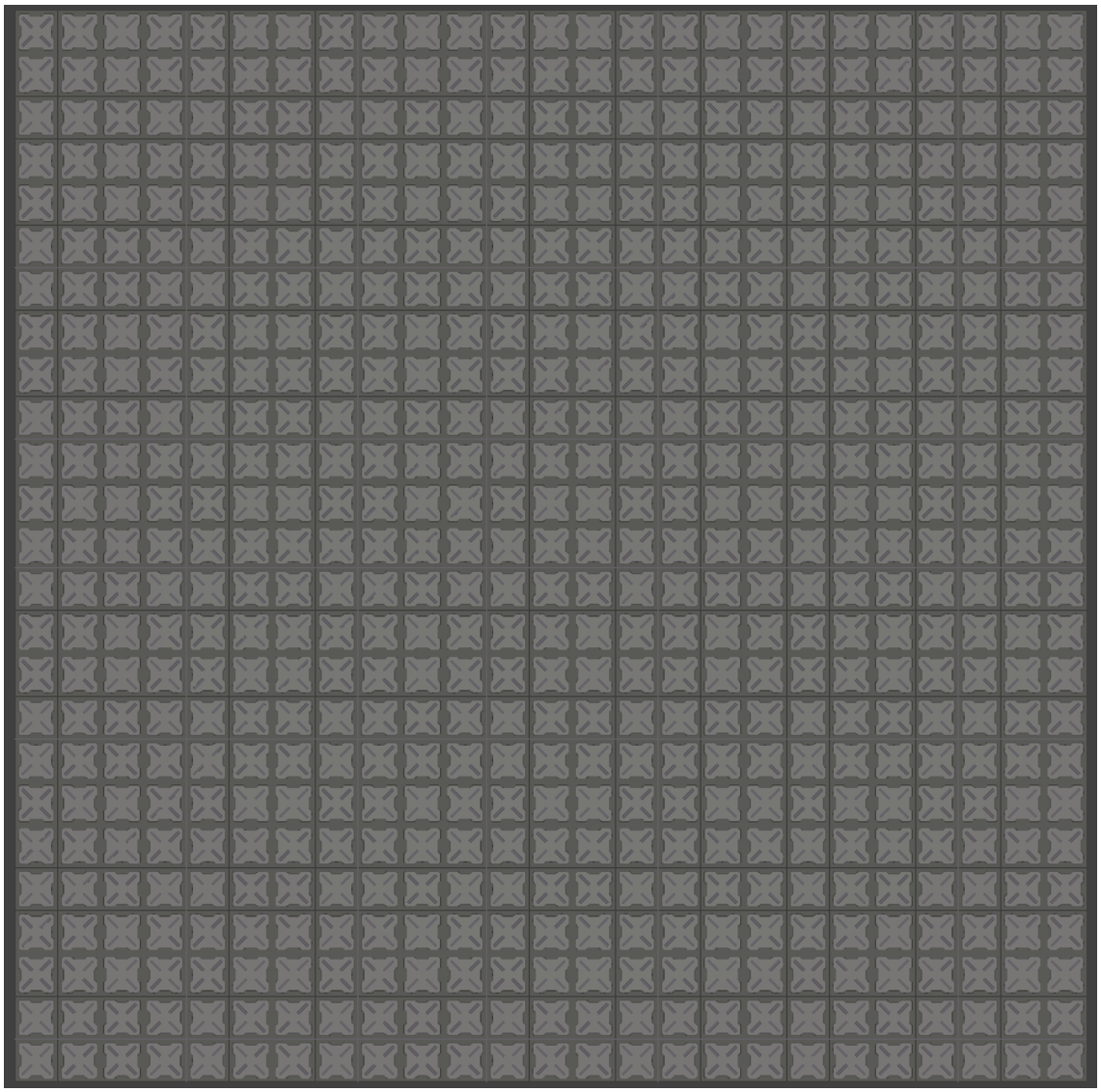
Pelikenttä koostuu 25 x 25 -ruudukosta, jossa jokainen ruutu on neliön muotoinen. Yhdessä ruudussa voi olla joko tyhjä lattia, seinä, ovi, suojana toimiva esine, kauppa tai aarre. Tyhjä lattia voi myös olla erityisesti merkitty alku- tai loppuhissiin kuuluvaksi. Generoitaessa ruudukkoa käsitellään kaksiulotteisena taulukkona, joka pitää sisällään enum-tyyppisenä muuttujana tiedon ruudun sisällöstä (esimerkkikoodi 1). Yksinkertaisen datatyypin vuoksi ruudukon käsittely on hyvin tehokasta ja generointi on suhteellisen nopeaa.

```
public TileData[, ] tileData;

/// <summary>
/// Value for determining static nature of a tile.
/// </summary>
public enum TileData
{
    Floor,
    Wall,
    Door,
    Cover,
    StartElevator,
    EndElevator,
    Shop,
    Loot,
    ShopOrLoot
}
```

Esimerkkikoodi 1. Ruudukon ruutujen sisällöt enum-tietotyyppinä.

Generoinnin alussa ruudukko luodaan ja täytetään seinillä (kuva 9).



Kuva 9. Generoinnin alussa luotu pelikentän ruudukko.

Ympäröivät ruudut -luokka on kokoelma tietoja tiettyä ruutua ympäröivistä kahdeksasta ruudusta (Mooren naapurusto) (esimerkkikoodi 2). Tarkasteltavan ruudun ollessa kentän laidalla ympäröiviä ruutuja voi olla vähemmän kuin kahdeksan, mikä sisältyy luokan tietoihin. Tietoja käytetään erityisesti soluautomaattivaiheessa. Luokka sisältää tiedot, kuinka monta seinä- ja avointa ruutua tarkasteltavan ruudun ympärillä on ja ovatko avoimet ruudut yhtenäisessä jonossa vai onko niiden välissä seinäruutuja. Jälkimmäinen tieto on tärkeä vihje siitä, onko ruutu kahden eri huoneen välissä. Tietoja ympäröivistä ruuduista käytetään

myös suojaesineiden, kauppa- ja aarrepaikkojen asettelun aikana varmistamaan, ettei muutettava ruutu sulje mahdollista käytävää tai muuta kulkureittiä.

```
/// <summary>
/// Data on surrounding tiles. Useful during level generation for
/// cellular automata, and tile validation. Gather data with Get
/// SurroundingTilesData method.
/// </summary>
public class SurroundingTiles
{
    public int tileCount = 0;
    public int open = 0;
    public int openIslands = 0;
    public int walls = 0;
    public int other = 0;
    public int blocked { get { return walls + other; } }

    public bool door = false;
    public bool elevator = false;
    public bool shopOrLoot = false;
    public MapTile shopOrLootTile = null;

    public int neighbouringOpen = 0;
    public int neighbouringWalls = 0;
    public int neighbouringOther = 0;
    public int neighbouringBlocked
      { get { return neighbouringWalls + neighbouringOther; } }
}
```

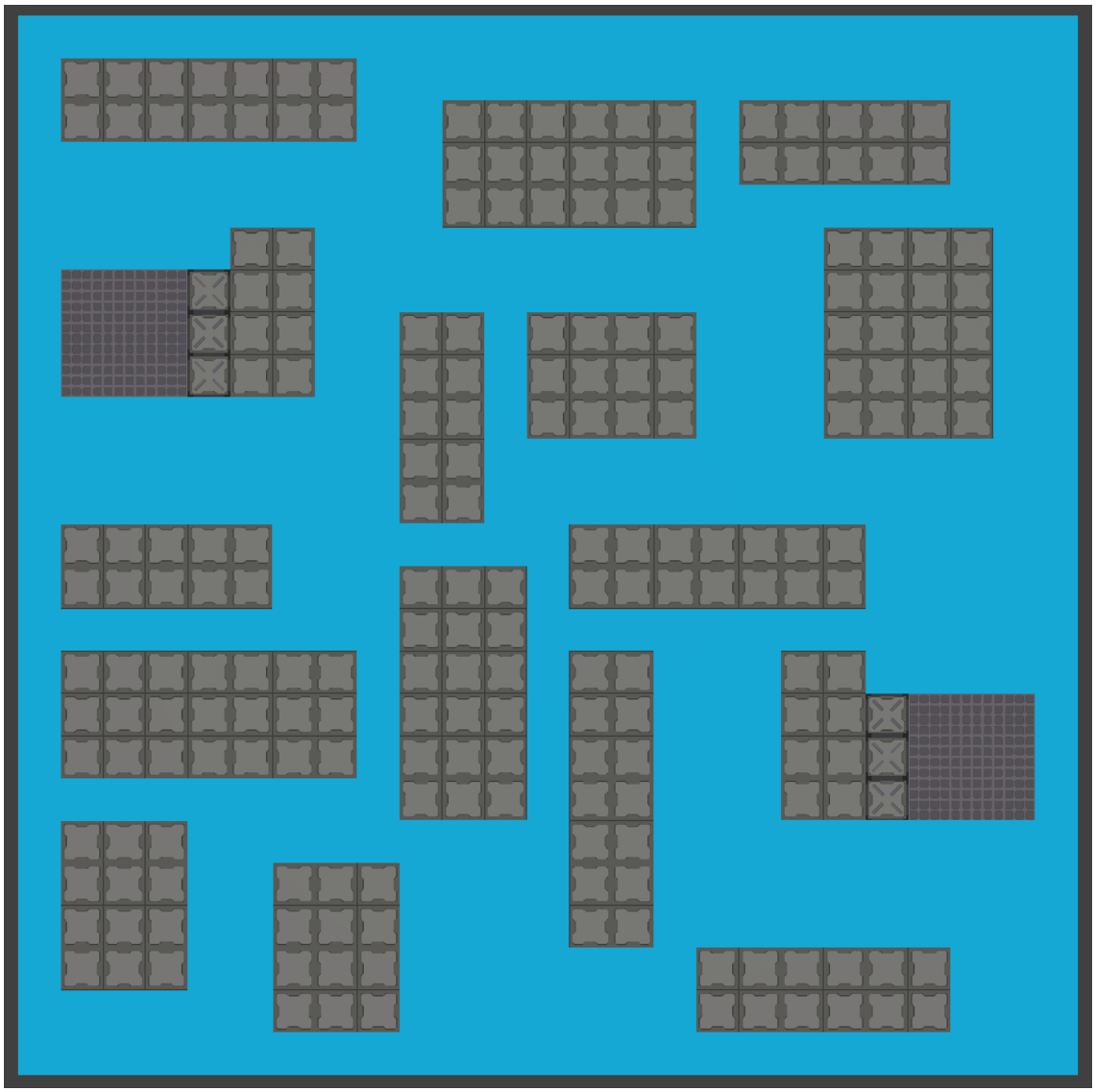
Esimerkkikoodi 2. SurroundingTiles-luokka sisältää koottua tietoa yksittäisen ruudun ympäröivistä ruuduista.

Huoneita luodessa kootaan lista huoneista. Lista muodostuu RoomData-tyypin olioista, jotka sisältävät listan huoneeseen kuuluvista ruuduista. Listaa päivitetään huoneita laajentaessa soluautomaatilla ja käytetään hyödyksi huoneita yhdistäessä.

4.2.2 Huoneiden luonti

Huoneiden asettelu aloitetaan asettamalla kentän vasemman laidan viereen satunnaiseen kohtaan aloitushuone eli hissi, jolla pelaajan hahmot saapuvat tasolle, ja kentän oikean laidan viereen satunnaiseen kohtaan loppuhuone eli hissi, johon pelaajan on tavoitteena päästä. Hissihuoneiden eteen luodaan ovet ja niiden jälkeen parin ruudun levyinen huone, jolla hoidetaan alku- ja loppuhuoneiden yhdistäminen muuhun pelikenttään.

Alku- ja loppuhuoneiden asettelun jälkeen kentälle asetellaan satunnaisen kokoisia huoneita satunnaisiin paikkoihin (kuva 10). Huoneiden mitat generoidaan RoomGeneratorFormula-luokassa (huonegeneraattori). Kenttägeneraattoriin voi määritellä listan huonegeneraattoreista, eli eri parametreillä generoitavista huoneista ja niiden määristä. Kenttägeneraattori käy läpi listan ja pyytää vuorollaan kutakin huonegeneraattoria generoimaan huoneen mitat. Huonegeneraattori generoi huoneen mitat parametriensa mukaisesti. Huoneiden leveys ja pituus valitaan satunnaisesti minimi- ja maksimiparametrien väliltä. Lisäksi tarkistetaan, että huoneen pinta-ala on sille asetettujen minimi- ja maksimiparametrien välillä.



Kuva 10. Pelikentälle asetetut suorakulmion muotoiset huoneet ja laidoilla sijaitsevat sisään- ja uloskäyntihuoneet. Huoneisiin kuuluvat ruudut näkyvät harmaina, ja sinisellä on merkitty ylijäävä tila, joka koostuu seinäruuduista.

Valitun mittaiselle huoneelle etsitään sopiva satunnainen paikka kentältä valitsemalla huoneen vasemman alakulman paikka. Huoneen paikan x- ja y-koordinaatit valitaan satunnaisesti, mutta tarkistaen, että huone ei mene päällekkäin muiden huoneiden kanssa eikä ruudukon ulkopuolelle.

Ensin muodostetaan lista kaikista seinäksi merkityistä ruuduista. Paikkakoordinaatit valitaan satunnaisesti listasta. Paikkakoordinaateista lähtien tarkistetaan huoneen mittojen mukaisesti potentiaalisesti huoneen alla olevat ruudut, joista

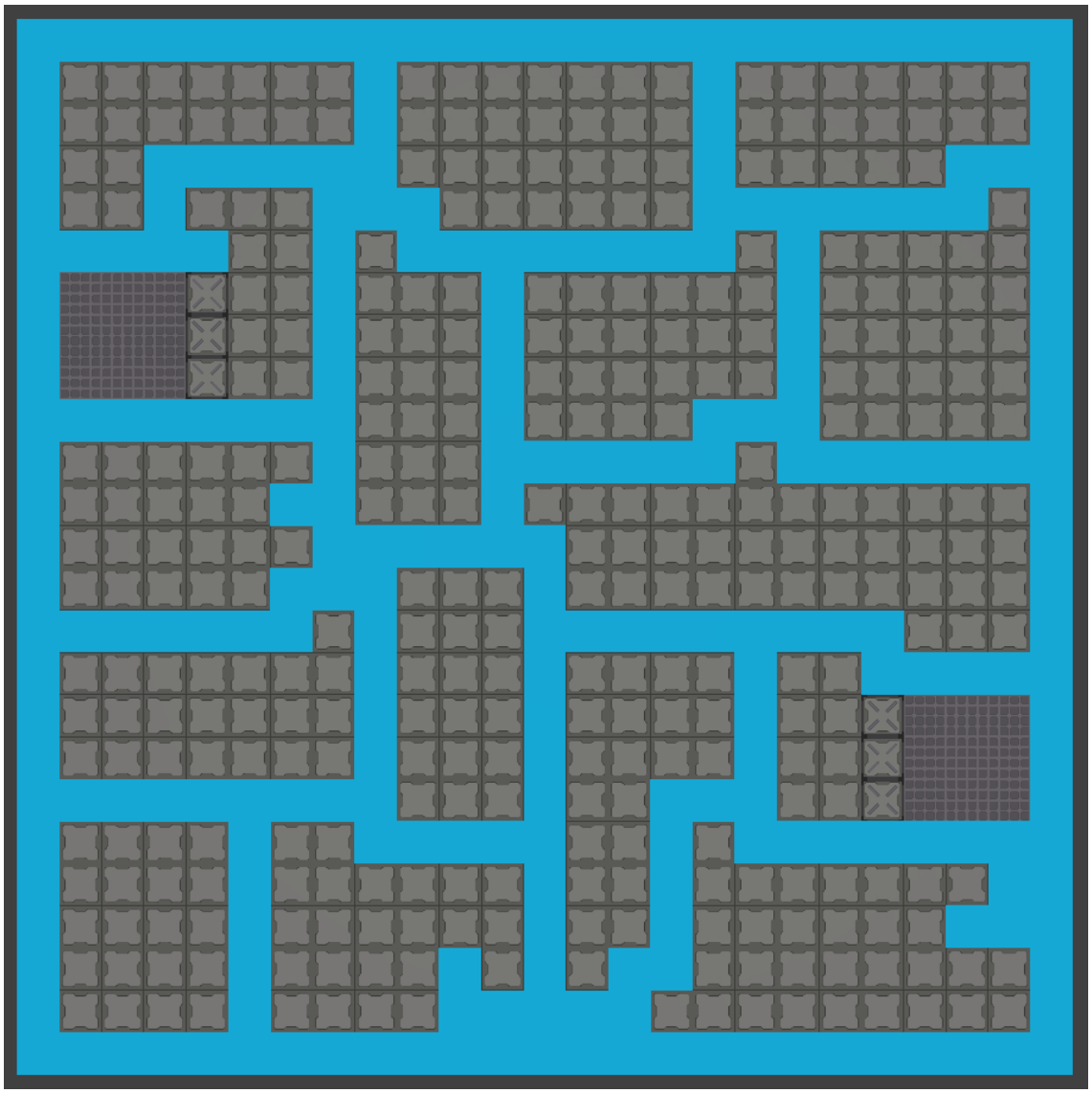
jokaisen on oltava seinäksi merkitty ruutu, joka ei vielä kuulu mihinkään huoneeseen eli sen vieressä ei ole lattiaksi merkittyä ruutua. Jos mikään tarkistuksista epäonnistuu, aloitetaan paikan etsiminen uudestaan generoimalla uudet paikkakoordinaatit. Jos mitoitetulle huoneelle ei löydy validia paikkaa eli ruudukosta ei löydy tilaa valitun mittaiselle huoneelle, algoritmi palaa huoneen mittojen generoimiseen uudelleen. Kun validi paikka löytyy, merkitään huoneen sisäiset ruudut lattiaruuduiksi ja lisätään onnistuneesti luotu huone huonelistaan. Jos paikka hylätään, poistetaan tarkistetut ruudut mahdollisten paikkakoordinaattien listalta, jotta samoja epävalideja koordinaatteja ei kokeilla uudestaan.

Huoneiden asettelua jatketaan toistamalla prosessi parametrien määrittämällä määrällä toistoja tai kunnes kentällä ei ole enää mahdollisia ruutuja, joihin huone olisi mahdollista asettaa.

Parametreissa on mahdollista määrittää lisäksi myös generoitavaksi joukko huoneita, joiden paikkaa haettaessa ei välitetä päällekkäisyyksistä tavallisten huoneiden kanssa. Vain alku- tai loppuhuoneen kanssa päällekkäisyys tai ruudukon ulkopuolelle meneminen on esteenä näiden huoneiden asettelulle. Tämä lisäasetus voi siis asettaa ylimääräisiä suorakulmion muotoisia huoneita muiden huoneiden kanssa päällekkäin ja luoda monimutkaisempia muotoja huoneille.

4.2.3 Tilan täyttäminen käyttäen mukautettua soluautomaattia

Huoneiden asettelun jälkeen jäljelle jäävä ylimääräinen tila täytetään mukautetulla soluautomaattialgoritmilla muuttamalla seinäruutuja lattiaruuduiksi. Soluautomaatin avulla huonepohjaisesta kentästä tehdään orgaanisempi ja sokkelomaisempi (kuva 11). Kenttägeneraattorissa käytetään muokattua soluautomaattialgoritmia, jossa on muutettu ruutujen käsittelyjärjestystä ja otettu huomioon erikoistapauksia, kuten aloitushuoneet ja niiden ovet, joita ei haluta muokata.



Kuva 11. Soluautomaatilla muokattu pelikenttä.

Algoritmi aloittaa käymällä ruudukkoa läpi ruutu kerrallaan. Käsiteltävän ruudun ympäröivät kahdeksan ruutua tarkastellaan ja niistä lasketaan ruutujen kokonaismäärä, erikoisruudut, seinä- ja lattiaruutujen määrät ja niiden vierekkäisyys. Tiedot kootaan SurroundingTiles-luokkaan (ks. luku 4.2.1 Ruudukko ja tietotyypit).

Ruudun käsittelyä ei jatketa, jos ympäröivistä ruuduista löytyy oviruutu tai alku- tai loppuhuoneeseen kuuluva ruutu, koska kyseisiä huoneita ei haluta muuttaa. Käsittelyä ei myöskään jatketa, jos ympäröivien ruutujen kokonaismäärä on alle

kahdeksan, koska tällöin ruutu on pelikentän ulkoreunalla, joka muodostuu aina seinistä. Jos ruudun ympärillä on vähintään neljä muuta seinää ja vähintään kaksi avointa ruutua, jotka ovat vierekkäin, seinä muutetaan lattiaksi. Muutettu ruutu lisätään lähimmän huoneen lattiaruutulistaan. Algoritmi jatkuu seuraavaan käsiteltävään ruutuun (esimerkkikoodi 3).

```
/// <summary>
/// Recursively expands a room by changing surrounding walls
/// to floors based on a custom cellular automata principle
/// </summary>
protected virtual LevelData CarveWall(LevelData levelData,
    LevelData.RoomData sourceRoom, int x, int y, int levelWidth,
    int levelLength, int maxSurroundingIslands,
    int minSurroundingOpen, int minSurroundingWalls)
{
    if (levelData.tileData[x, y] != LevelData.TileData.Wall)
    {
        return levelData;
    }
    // Get data on tiles surrounding current tile
    LevelData.SurroundingTiles tiles =
        levelData.GetSurroundingTilesData(x, y);

    // If next to door, do not change tile to floor
    if (tiles.door) return levelData;
    // Check for elevator in any surrounding tile, if found do not
    // change tile to floor
    if (tiles.elevator) return levelData;
    // If next to level edge, do not change tile to floor
    if (tiles.tileCount < 8) return levelData;

    // Change tile to floor if conditions filled
    if (tiles.openIslands <= maxSurroundingIslands &&
        tiles.open >= minSurroundingOpen &&
        tiles.walls >= minSurroundingWalls)
    {
        levelData.tileData[x, y] = LevelData.TileData.Floor;

        if (sourceRoom == null)
        {
            sourceRoom = levelData.AddRoom(x - 1, y - 1, 3, 3,
                true);
        }
        else
        {
            sourceRoom.AddTile(x, y);
        }

        // Continue to surrounding tiles recursively
        List<MapTile> surroundingTilesCoords =
            levelData.GetSurroundingTileCoords(x, y);
        foreach (MapTile tile in surroundingTilesCoords)
        {
            levelData = CarveWall(levelData, sourceRoom, tile.x,
            tile.y, levelWidth, levelLength,
            maxSurroundingIslands, minSurroundingOpen,
            minSurroundingWalls);
        }
    }

    return levelData;
}
```

Esimerkkikoodi 3. Mukautettu soluautomaattialgoritmi.

Ruutujen käsittelyjärjestys

Projektin soluautomaatin lopputulos voi muuttua huomattavasti ruutujen käsittelyjärjestyksen mukaan. Tavallisesti soluautomaatti käy läpi koko sukupolven ruudut ja muuttaa ne sitten kerralla, jolloin sukupolven sisällä ruutujen käsittelyjärjestyksellä ei ole väliä. Projektin soluautomaatti kuitenkin muuttaa solujen arvoja heti, mikä vaikuttaa vierekkäisten solujen käsittelyn tuloksiin. Yleinen järjestys on käydä läpi kaikki ruudukon ruudut yksi kerrallaan riveittäin vasemmalta oikealle kulkien ja tarvittaessa toistaa koko ruudukon läpikäynti ennalta määritelty määrä kertoja tai kunnes evaluointifunktion tulos on haluttu tai jokin muu ehto täyttyy. Yleinen soluautomaatti tällä käsittelyjärjestyksellä tuottaa usein luolamaisia muodostelmia, joiden reunat vähitellen laajenevat tilaan ja tilan reunojen muodot ja kulmat pehmentyvät kaarimaisiksi ja suorat 90 asteen kulmat vähentyvät isommassa skaalassa. Efekti voimistuu käsittelyn toistokertojen lisääntyessä. [Kun 2012.]

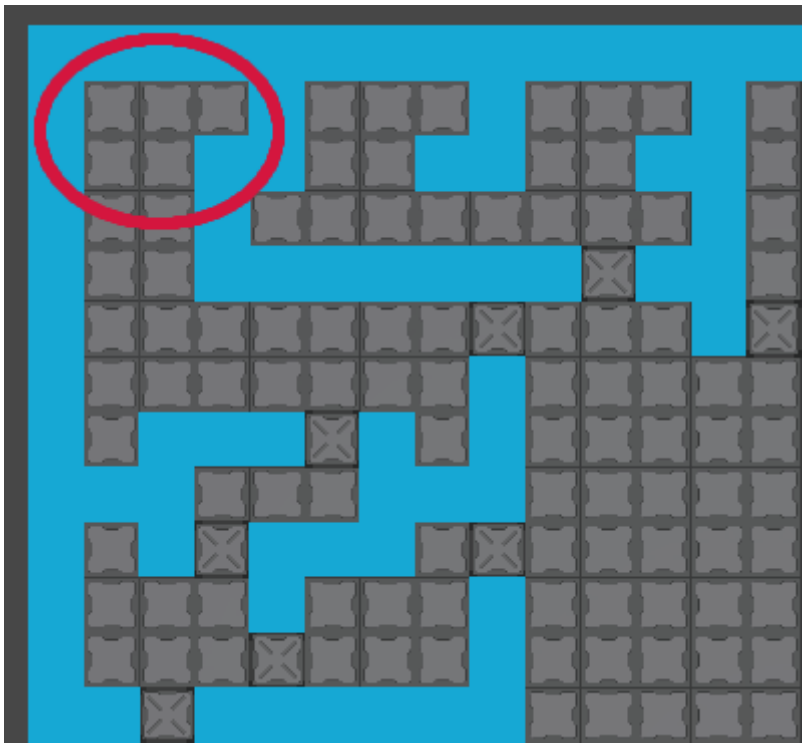
Rekursiivinen käsittelyjärjestys aloittaa tietystä ruudusta, esim. satunnaisesti valitusta ruudusta, ja jatkaa ympäröiviin ruutuihin aina, kun käsittely onnistuu muuttamaan ruudun, eli ympäröivät ruudut lisätään käsiteltävien ruutujen jonoon. Ympäröivät ruudut voivat tässä tarkoittaa vain vierekkäisiä (von Neumannin naapurusto), ympäröiviä (Mooren naapurusto) tai joitakin muita joukkoja, esim. laajennettua Mooren naapurustoa. Soluautomaatti käy ruutuja läpi, kunnes vastaan tulee vain ruutuja, joita ei onnistuta muuttamaan. Riippuen käsiteltävien ruutujen valinnasta algoritmin alussa rekursiivinen soluautomaatti ei välttämättä käsittele kaikkia ruudukon ruutuja.

Projektin algoritmissa ruutujen läpi käyminen tapahtuu ensisijaisesti järjestyksessä ruudukon mukaan, mutta jos ruutu täyttää seinäruudusta lattiaruuduksi muuttamisen ehdot, käydään läpi myös kyseisen ruudun ympärillä olevat ruudut (Mooren naapurusto), ennen kuin edellinen käsittelyjärjestys jatkuu. Prosessi jatkuu rekursiivisesti aina, kun ruutu onnistuneesti muutetaan, ja alkuperäiseen ruudukon järjestykseen palataan vain, kun rekursiivinen käsittely päättyy. Käytännössä algoritmi tällöin käy läpi huoneen kerrallaan ja jatkaa sen huoneen

seinien laajentamista, kunnes muiden huoneiden rajat tulevat vastaan. Rekursiivisuuden vuoksi huoneet laajenevat ympäröivään tilaan.

Jokainen ruudukon ruutu käydään läpi ainakin kerran, mutta kun onnistuneen ruudun muutoksen yhteydessä käydään läpi sitä ympäröivät ruudut, voi prosessi jatkua edelleen tästä rekursiivisesti niin kauan, kuin ympäröivistä ruuduista jokin myös onnistuneesti muutetaan. Tällöin ruutuja saatetaan lopulta käydä läpi useamman kerran koko prosessin aikana. Ruutu voi muuttua vasta myöhemmällä käsittelykerralla, koska sitä ympäröivät ruudut ovat voineet muuttua viime tarkastelukerran jälkeen.

Tällainen käsittelyjärjestys voi johtaa toistuviin epätoivottuihin muodostelmiin. Esimerkiksi huoneiden reunaan usein muodostuu toistuva viiden ruudun kuvio, koska rekursiivinen käsittelyjärjestys aina tarkastelee ruutuja myötäpäivään muutetun ruudun ympärillä (kuva 12). Joskus erikoiset muodostelmat voivat olla kokonaisuuteen sopivia, jolloin on hyvä tehdä muistiinpanoja muutoksista ja parametreista, joilla tulokset saatiin aikaan ja joilla tulokset ovat uudelleentoteutettavia jatkossa. Ennalta-arvaamattomat tulokset voivat olla antoisia ja mielenkiintoisia, jos niiden toteutumista ja esiintymistiheyttä pystyy hallitsemaan, eli ne eivät esim. toistu liikaa. Toisaalta soluautomaatin parametrit ja käsittelyjärjestyksen muuttaminen vaikuttavat usein monimutkaisesti ja vaikeasti ennustettavasti lopputulokseen, jolloin tulosten hallinta voi olla vaikeaa.



Kuva 12. Osa generoidusta pelikentästä, jossa näkyy monesti toistuva viiden ruudun kuvio (punaisella ympyröity esimerkki), joka on seurausta soluautomaatin parametreista ja ruutujen käsittelyjärjestyksestä.

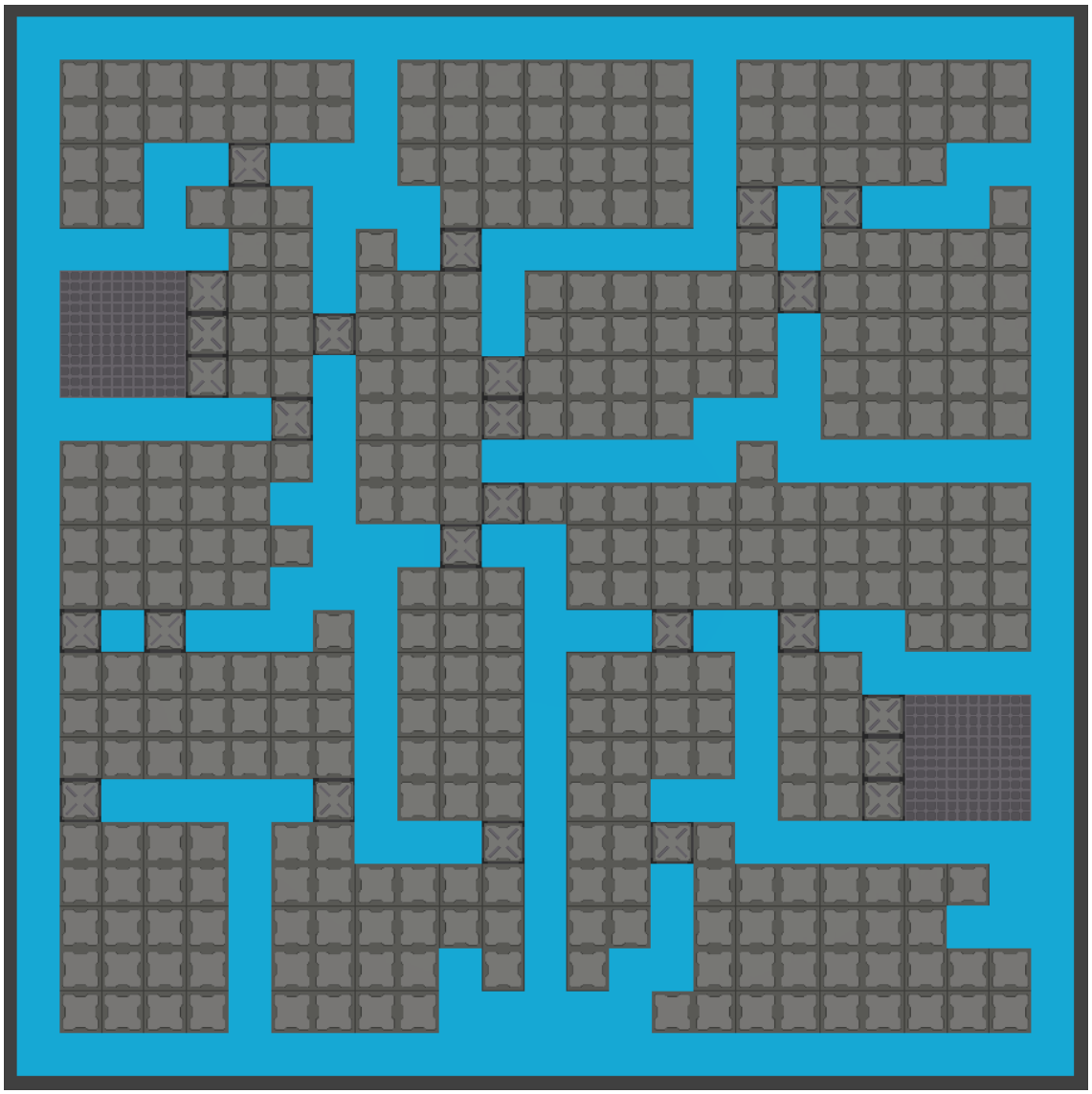
4.2.4 Huoneiden yhdistäminen

Huoneet yhdistetään toisiinsa etsimällä oviksi kelpaavat ruudut. Ovien etsiminen aloitetaan alkuhuoneesta. Tarkastellaan kaikkia seiniä, jotka yhdistävät huoneen viereisiin huoneisiin, ja valitaan ja muutetaan oveksi niistä yksi satunnaisesti. Algoritmia jatketaan tarkastelemalla nyt alkuhuonetta ja uutta yhdistettyä huonetta yhdessä ja listataan kaikki seinät, jotka yhdistävät aluetta muihin yhdistämättömiin huoneisiin. Seinistä jälleen valitaan yksi oveksi ja prosessi toistetaan. Lisäksi välillä avataan ylimääräinen seinä oveksi, mikä lisää kentän sokkelomaisuutta ja uusia reittejä (kuva 13).

Seinät, jotka yhdistävät huoneita, etsitään tarkastelemalla huoneen seinäruutujen viereisiä ruutuja ja mihin huoneisiin ne kuuluvat. Kaikki kahden eri huoneen välissä olevat seinäruudut kartoitetaan, ja jokaiselle huoneelle muodostetaan

lista sitä ympäröivistä oviksi kelpaavista seinistä. Seinäruutu on kahden eri huoneen välissä, jos sen vierekkäisissä (von Neumannin naapurusto) ruuduissa on vähintään kaksi lattiaruutua, jotka kuuluvat eri huoneisiin. Tällöin merkitään muistiin seinä ja huoneet, joiden välillä se on.

Kun huoneita käydään läpi ja yksi seinistä valitaan oveksi satunnaisesti, huonekohtaisista listoista poistetaan kaikki muut samoja huoneita yhdistävät seinät. Samalla kuitenkin seinillä on pieni todennäköisyys valikoitua ylimääräiseksi oveksi.



Kuva 13. Yhdistetyt huoneet. Ruudut, joissa on X-kirjainta muistuttava kuvio, ovat oviruutuja, jotka yhdistävät huoneita.

Sokkelon läpi kulkevaa reittiä ei validoida, koska käytännössä tilannetta, jossa reittiä ei muodostuisi, ei toteutuksessa käytettävillä parametreilla ilmene. Sokkelon läpi pitäisi teoriassa aina muodostua reitti, koska soluautomaatti laajentaa huoneet niin, että niiden välille jää vain yhden ruudun paksuinen seinä. Seinältä löytyy aina ruutuja, jotka voi muuttaa oviksi, ja näin jokainen huone yhdistyy sokkeloon. Parametrien muuttuessa ongelmatilanteita saattaisi kuitenkin syntyä (ks. luku 5 Tulokset ja jatkokehitys).

4.2.5 Esineiden ja vihollisten lisäys

Kenttä viimeistellään asettelemalla lattiaruutuihin satunnaisesti vihollisia ja esineitä, kuten suojana toimivia laatikoita, kauppoja, aarteita ja parannusasemia. Tärkeää on, että esineet eivät tuki oviaukkoja tai kulkureittejä ylipäätään. Erikoisesineille, eli muille kuin suojana toimiville esineille, sopivat paikat merkitään ensisijaisesti lattiaruuduista, joilla on vain yksi ympäröivä lattiaruutu. Jos paikkoja ei löydy tarpeeksi, etsitään ruudut, joilla on kaksi ympäröivää lattiaruutua ja niin edelleen. Lisäehtona on, että ympäröivät lattiaruudut ovat yhdessä saarekkeessa eli vierekkäin, jolloin poissuljetaan ruudut, joissa esine tukkisi käytäviä tai kulkureittejä. Tällöin esineet sijoittuvat ensisijaisesti seinien koloihin, nurkkiin ja muihin luonnolliselta tuntuviin paikkoihin.

Suojana toimivat laatikot asetellaan vapaammin satunnaisiin paikkoihin myös huoneiden keskelle. Myös niiden osalta kuitenkin tarkistetaan, että ympäröivät lattiaruudut muodostavat yhden saarekkeen, millä varmistetaan, että laatikot eivät voi tukkia kulkureittejä. Asettelu on silti hyvin satunnaista ja laatikoiden paikat jäävät siksi monesti epäluonnollisiksi. Yksinkertainen ratkaisu kuitenkin toimii pelattavuuden kannalta.

Eri generoitavat esineet (ruutuihin menevät suojana toimivat laatikot, kaupat, arkut jne.) on jaettu omiin luokkiin abstraktin yläluokan (InteractablePoint) alle. Näin uudentyyppisten esineiden lisääminen on ollut helppoa ja tietyn tyypin esineiden generoinnin parametrit ovat helposti säädettävissä. Uusia esineitä voidaan määritellä datatasolla pelimoottorissa.

Viholliset lisätään pelikentälle viimeisenä. Ruudukosta etsitään kaikki tyhjät lattiaruudut, jotka eivät kuulu alku- tai loppuhuoneisiin, ja niiden joukosta valitaan satunnaisesti paikat, joihin luodaan vihollishahmot. Vaihe on toteutettu hyvin yksinkertaisesti, ja vihollisten jakautuminen on täysin sattumanvaraista. Vain vihollisten määrä ja tyypit on määritelty parametreissa.

4.2.6 Mallien valinta, yhdistely ja asettelu

Generoinnin lopussa 3D-mallit asetellaan kentälle ruudukon datan mukaisesti ja tiedot muutetaan polunetsimisalgoritmille sopivaan muotoon (kuva 14). Ruudukko käydään läpi ja tarkastellaan jokaiselle ruudulle annettua enum-tyyppistä TileData-arvoa. Ruudukkoa on käsitelty vain enum-arvoilla tähän vaiheeseen mennessä. Pelikenttä koostuu 3D-maailmaan ruudukoksi asetelluista 3D-malleista, joihin on liitetty erillinen ”Ruutu”-skripti, joka sisältää dataa ja metodeja polunetsintään ja näkökenttien laskemiseen.

Kuva 14. Pelin sisäinen näkymä valmiista generoidusta pelikentästä.

Jokaiseen ruutuun vaihdetaan ruututyypille kuuluva lattia ja luodaan ruututyyppiä vastaava 3D-malli, esim. seinä tai ovi. Ruutuihin, joihin on merkitty esine,

kauppa tai aarre, 3D-mallit valitaan satunnaisesti tyyppiä vastaavasta listasta, jossa on valikoima vaihtoehtoisia malleja. Ruutuihin asetetaan tyypin mukaisesti myös arvot, jotka määrittävät, onko ruutu läpikuljettava, läpinäkyvä ja suojana toimiva. Näiden arvojen perusteella polunetsintä ja näkökenttien laskenta pystyvät toimivasti käsittelemään ruudukkoa.

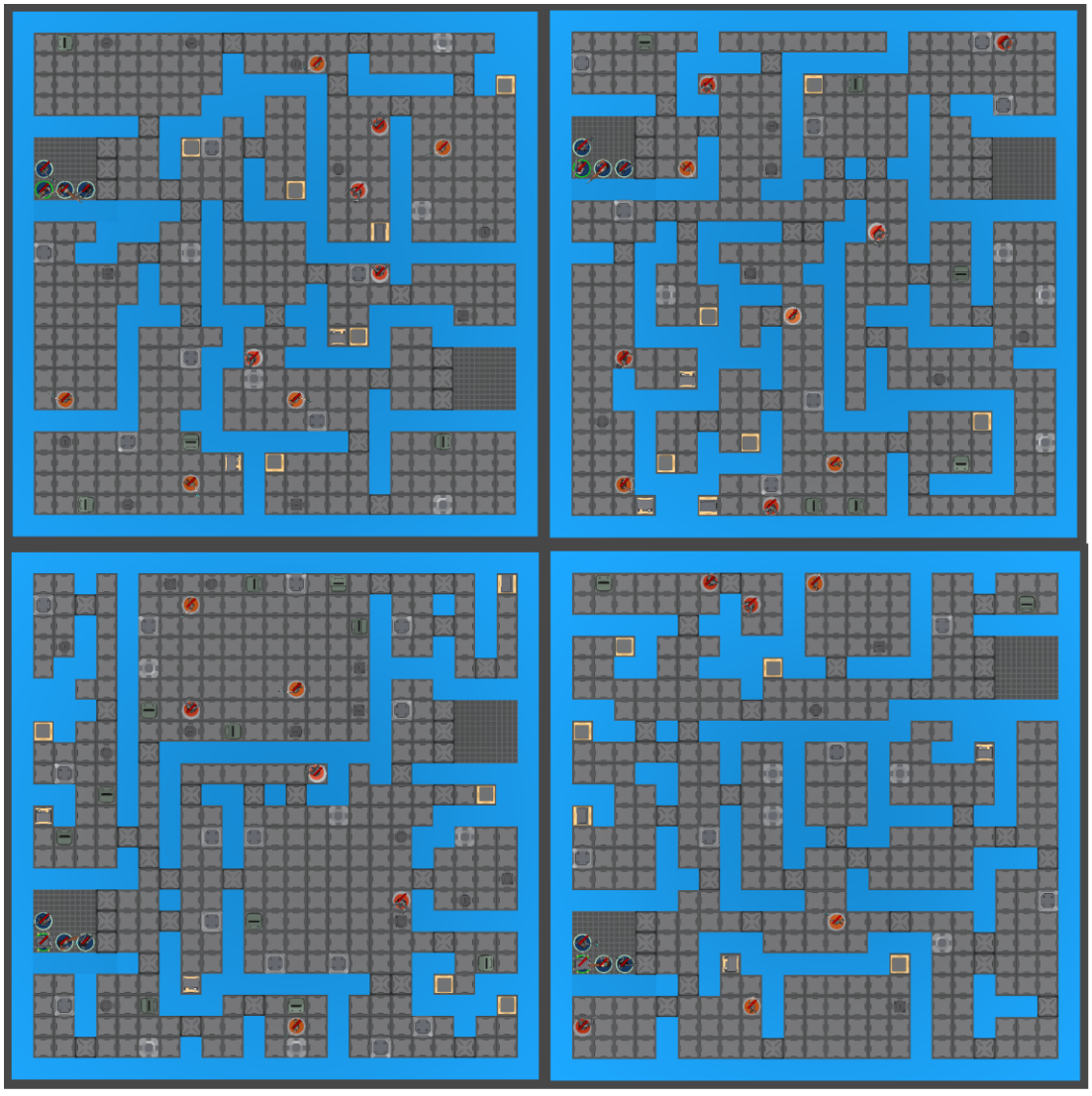
4.3 Monipuolisuus ja progressio parametrisoimalla

Pelikentän generointi toteutettiin kerroksittain niin, että generaattoreita voi määritellä datatasolla useamman erilaisen, jotka voi sitten asettaa käytettäväksi tietyille tasoille, esim. generaattori 1 luo tasot 1–3, generaattori 2 tasot 4–6 ja generaattori 3 tason 7. Pelin progressio voidaan siis määrittää kenttägeneraattorien avulla. Generaattorin alla on parametrit kaikkien vaiheiden säätämiseksi, mukaan lukien pelimekaaniset elementit, kuten aarteet, kaupat, ja viholliset ja niiden määrät.

Datamääritellyt generaattorit voivat erota toisistaan useiden parametrien avulla ja huonegeneraattoreita vaihtamalla. Huonegeneraattorit ovat myös datatasolla määriteltävissä. Yksinkertainen huonegeneraattori sisältää parametrit suorakulmaisen huoneen leveys- ja pituusmittojen minimi- ja maksimiarvoille ja huoneen pinta-alan minimi- ja maksimiarvoille. Huonegeneraattorit perustuvat abstraktiin yläluokkaan, jolloin jatkokehitystä silmällä pitäen on mahdollista lisätä kenttägeneraattorin valikoimaan huonegeneraattoreita eri muodoille kuin suorakulmioille tai muuten määriteltyjä huoneita, esim. ihmissuunnittelijan kokonaan määrittelemiä, ”käsintehtyjä” erikoishuoneita.

5 Tulokset ja jatkokehitys

Insinöörityön tuloksena luotiin onnistuneesti pelin kannalta toimiva ja pelattavia kenttiä luova kenttägeneraattori. Suunnitelluista ominaisuuksista toteutuivat oleellisimmat, eli generaattori luo sokkelomaisia kenttiä, jotka ovat pelattavissa alusta loppuun. Valikoidut tekniikat, parametrisointi ja iterointi veivät suhteellisen paljon aikaa, ja vastaavasti pelattavia tuloksia voisi toteuttaa yksinkertaisemminkin. Toisaalta generoidut kentät sisältävät mielenkiintoisia piirteitä ja monipuolisuutta (kuva 15).



Kuva 15. Eri parametreilla generoituja pelikenttiä. Lisää vaihtelua tuloksiin saadaan helposti generoimalla joukko erikokoisia huoneita parametreja muuttamalla.

Monimutkaisemmat ominaisuudet, kuten kentän jakaminen lukittuihin osiin tai erikseen generoituihin sivuhaaroihin, eivät toteutuneet projektin mitoissa.

Huoneiden luonti toteutettiin periaatteessa yksinkertaisella satunnaisasettelulla, mikä kuitenkin toi omat haasteensa parametrisointiin ja tulosten hallintaan. Yksi kenttägeneraattorin suunnitelluista ominaisuuksista oli huoneiden määrän ja kokojakauman parametrisointi, mikä toteutuksessa vaati mukautetun järjestelmän ja luokkarakenteen, joka silti tuotti vaikeasti hallittavia tuloksia. Yksi mahdollisesti toimiva vaihtoehto huoneiden asettelun ratkaisuksi olisi voinut olla tilan jakaminen -algoritmi. Sen avulla huoneiden kokojakauman hallinta olisi ollut mahdollisesti selkeämmin parametrisoitu osa algoritmia.

Soluautomaatti algoritmina loi mielenkiintoisia tuloksia ja lisäsi tulosten monipuolisuutta. Algoritmin mukautus huonepohjaisen kentän täyttämiseen toi omat haasteensa, mutta ruutujen rekursiivinen käsittelyjärjestys loi haluttuja tuloksia. Mukautettu algoritmi silti tuotti ongelmallisia toistuvia kuvioita, joiden välttämiseksi algoritmi vaatisi lisää hienosäätämistä. Kuviot eivät kuitenkaan ole pelattavuuden kannalta kriittisiä ongelmia.

Soluautomaatti vähentää tulosten ennustettavuutta, kun se esim. laajentaa pieniksi tarkoitettuja huoneita suuremmiksi. Tällöin huoneiden kokojakauman määritteleminen ei aina tuota haluttuja tuloksia soluautomaatin lisäämän satunnaisuuden vuoksi. Mahdollinen ratkaisu voisi olla tiettyjen huoneiden merkitseminen soluautomaatin käsittelyn ulkopuolelle, jolloin esim. pitkulaisiksi käytäviksi määritellyt huoneet eivät muodostu joksikin muuksi soluautomaattivaiheessa, vaan soluautomaatti laajentaa ympärillä olevia muita huoneita sen sijaan.

Kentän koon kasvattaminen lisää jonkin verran generointiin käytettävää aikaa, mikä ei itsessään ole suuri ongelma, mutta ruudukon koon kasvaessa pelihahmojen polunetsintäalgoritmissa kestää huomattavasti pitempään. Pelihahmojen liikkumiseen muodostuu huomattava, jopa yli sekunnin, viive nostettaessa ruudukon koko 40 x 40:een tai suuremmaksi. Tämä ei varsinaisesti ole kenttägeneraattorin ongelma, mutta optimoinnin tarve polunetsintäalgoritmissa rajoit-

taa kenttägeneraattorin parametreja. Ruudukon koon kasvaessa kenttägeneraattori pystyisi luomaan monipuolisempia rakenteita, joten ulkoisia rajoitteita asettavien ongelmien, kuten polunetsinnän optimoinnin, ratkaisu olisi suotavaa.

Pelattavuuden varmistamiseksi pelikentän sokkelo olisi hyvä validoida eli varmistaa, että sokkelon läpi kulkee ainakin yksi validi reitti. Toteutuksen parametreilla generaattori käytännössä aina tuottaa läpikuljettavan sokkelon. Parametreja mahdollisesti muutettaessa sama ei välttämättä päde. Jos esim. soluautomaatin toimintaa rajoitetaan tai parametreja muutetaan niin, että se ei rekursiivisesti jatka huoneiden laajentamista niin kauan kuin tilaa riittää, vaan huoneiden väliin jäisi enemmän kuin yhden ruudun paksuisia seiniä, generaattori saattaisi tuottaa läpikulkemattomia pelikenttiä. Huoneiden yhdistämisvaihetta tulisi muokata joustavammaksi niin, että se voisi käsitellä myös tapaukset, joissa huoneiden välissä on paksumpi kuin yhden ruudun seinä.

Validointi voitaisiin toteuttaa pelissä olevan reitinhakualgoritmin avulla. Pelikentän generoinnin lopussa alkuhuoneesta etsittäisiin reittiä loppuhuoneeseen, ja jos mahdollista reittiä ei löytyisi, generoitu pelikenttä hylättäisiin ja generointi aloitettaisiin alusta. Jos pelikentän generoinnin on testauksessa todettu tuottavan tarpeeksi myös valideja ratkaisuja, epävalidin kentän hylkäys olisi toimiva ratkaisu. Ratkaisu saattaisi kuitenkin pidentää generointiin kuluvaa aikaa merkittävästi etenkin, jos valideja kenttiä muodostuu vain harvoin.

Reitinhaulla toimivan validoinnin yhteydessä voitaisiin myös tarkastella sokkelon läpikulun haastavuutta ja reitin pituutta ja varmistaa sopivasti haastavan sokkelon generointi. Hakemalla reitti alkuhuoneesta loppuhuoneeseen laskettaisiin lyhimmän reitin pituus, joka on yleisesti hyvä merkki sokkelon haastavuudesta. Generaattori voitaisiin määritellä hylkäämään kentät, joissa lyhimmän reitin pituus jää liian lyhyeksi.

6 Yhteenveto

Insinöörityön tavoitteena oli löytää käytännön ratkaisu proseduraaliseen pelikenttien generointiin tutkimalla proseduraalisen sisällön generoinnin menetelmiä ja toteuttamalla kenttägeneraattori valikoiduilla tekniikoilla. Insinöörityössä hahmotettiin yleiskuva proseduraalisesta sisällön generoinnista, sen alaisten menetelmien taksonomiasta ja vertailtavista ominaisuuksista sekä yleisesti sen hyödyistä ja haasteista. Lisäksi tutkittiin toteutettavaan kenttägeneraattoriin mahdollisesti soveltuvia erityisesti tyrmämäisten pelikenttien generointiin käytettäviä algoritmeja, joista soluautomaatti valittiin osaksi toteutusta.

Insinöörityössä todettiin myös, että proseduraalinen generointi ja siihen liittyvät menetelmät, kuten soluautomaatti, ovat käyttökelpoisia ja hyödynnettävissä laajasti myös pelikehityksen ulkopuolella eri tieteenaloilla.

Insinöörityön kenttägeneraattori onnistui tuottamaan tavoitteiden mukaisia sokkelomaisia, monipuolisia pelikenttiä. Kenttägeneraattori toteutettiin helposti laajennettavaksi, ja jo suoraan parametreja muuttamalla kenttägeneraattori pystyy tuottamaan vaihtelevia pelikenttiä. Soluautomaatti toimii hyödyllisenä osana kenttägeneraattoria ja tuo monipuolisuutta pelikenttien huoneiden muotoihin. Samalla soluautomaatti kuitenkin vähentää tulosten ennustettavuutta ja saattaa tuottaa toistuvia kuvioita, joita on myös vaikea ennustaa ja hallita.

Lähteet

Algorithms for Procedural Content Generation. 2019. Verkkoaineisto. Wikidot. <http://pcg.wikidot.com/category-pcg-algorithms> Luettu 13.5.2022.

Basic BSP Dungeon Generation. 2020. Verkkoaineisto. Roguebasin. <http://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation> Luettu 17.5.2022.

Bradfield, Chris. 2018. Procedural Generation in Godot - Part 6: Dungeons. Verkkoaineisto. KidsCanCode. <http://kidscancode.org/blog/2018/12/godot3_procgen6/> 4.12.2018. Luettu 13.5.2022.

Callahan, Paul. 2005. What is the game of life? Verkkoaineisto. Math.com. <http://www.math.com/students/wonders/life/life.html> Luettu 13.5.2022.

Devs weigh in on the best ways to use (but not abuse) procedural generation. 2018. Verkkoaineisto. Game Developer. < https://www.gamedeveloper.com/design/devs-weigh-in-on-the-best-ways-to-use-but-not-abuse-procedural-generation> 12.3.2018. Luettu 13.5.2022.

Doull, Andrew. 2008. The Death of the Level Designer: Procedural Content Generation in Games. Verkkoaineisto. <http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html> 14.1.2008. Luettu 13.5.2022.

.kkrieger. 2004. Verkkoaineisto. .theprodukkt. <https://web.archive.org/web/20120204065621/http://www.theprodukkt.com/kkrieger> Luettu 13.5.2022.

Kozliner, Evan. 2019. Algorithmic Beauty: An Introduction to Cellular Automata. Verkkoaineisto. Towards Data Science. <https://towardsdatascience.com/algrithmic-beauty-an-introduction-to-cellular-automata-f53179b3cf8f> Luettu 24.10.2019.

Kun, Jeremy. 2012. The Cellular Automaton Method For Cave Generation. Verkkoaineisto. <https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/> 29.7.2012. Luettu 17.5.2022.

Moric, Roc. 2016. Inside the Artificial Universe That Creates Itself. Verkkoaineisto. The Atlantic. <https://www.theatlantic.com/technology/archive/2016/02/artificial-universe-no-mans-sky/463308/> Luettu 13.5.2022.

Niemann, Marco. 2015. Constructive Generation Methods for Dungeons. Seminar-Thesis. Westfälische Wilhelms-Universität Münster. Saatavilla verkossa:

<https://www.wi.uni-muenster.de/sites/wi/files/users/mpreu_02/games-material/ba-vm-ss2015/marco_niemann_-_constructive_generation_methods_for_dungeons_-_thesis.pdf> Luettu 13.5.2022.

Pedersen, Kim. 2014. Procedural Level Generation in Games using a Cellular Automaton: Part 1. Verkkoaineisto. <https://www.raywenderlich.com/2425-procedural-level-generation-in-games-using-a-cellular-automaton-part-1> 30.4.2014. Luettu 17.5.2022.

Procedural Content Generation. 2007. Verkkoaineisto. Introversion Software. <https://www.gamecareerguide.com/features/336/procedural_content_.php> 2.6.2007. Luettu 13.5.2022.

Procedural Dungeon Generation Algorithm. 2015. Verkkoaineisto. Game Developer. <https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm> 3.9.2015. Luettu 13.5.2022.

Procedural dungeon generation algorithm explained. 2017. Verkkoaineisto. Reddit. <https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/> Luettu 17.5.2022.

Shaker, Noor; Togelius, Julian & Nelson, Mark J. 2016. Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer.

Siitonen, Samuli. 2017. Hakuperustainen proseduraalinen pelien sisällön generointi geneettisellä algoritmilla. Kandidaatintutkielma. Lappeenrannan teknillinen yliopisto. LUTPub.

System Time. 2018. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/windows/win32/sysinfo/system-time> Luettu 13.5.2022.

Teleological. 2009. Verkkoaineisto. Wikidot. <http://pcg.wikidot.com/pcg-algorithm:teleological> Luettu 13.5.2022.

Teleological vs. Ontogenetic. 2020. Verkkoaineisto. Wikidot. <http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic> Luettu 13.5.2022.

Watkins, Ryan. 2016. Procedural Content Generation for Unity Game Development. Packt Publishing.

Weisstein, Eric W. 2022a. Random Number. Verkkoaineisto. MathWorld. <http://mathworld.wolfram.com/RandomNumber.html> Luettu 13.5.2022.

Weisstein, Eric W. 2022b. Elementary Cellular Automaton. Verkkoaineisto. Mathworld. <https://mathworld.wolfram.com/ElementaryCellularAutomaton.html> Luettu 13.5.2022.

West, Mick. 2008. Random Scattering: Creating Realistic Landscapes. Verkkoaineisto. Game Developer. <https://www.gamedeveloper.com/disciplines/random-scattering-creating-realistic-landscapes> 6.8.2008. Luettu 13.5.2022.

Wolfram, Stephen. 1983. Statistical Mechanics of Cellular Automata. Reviews of Modern Physics, 55 (July 1983), s. 601-644.

Yu, Derek. 2016. Spelunky. Boss Fight Books.

Zapata, Santiago. 2017. On the historical origin of the roguelike term. Verkkoaineisto. Slashie's Journal. <https://blog.slashie.net/on-the-historical-origin-of-the-roguelike-term/> Luettu 4.5.2022.

00011110. 2019. Verkkoaineisto. Binaura. <https://www.binaura.net/projects/00011110> Luettu 17.5.2022.