Juri Palotie

Proseduraalinen tiegeneraattori

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

5.5.2017

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Juri Palotie Proseduraalinen tiegeneraattori 30 sivua + 1 liitettä 5.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro

Opinnäytetyön tavoitteena oli tutkia, mitä hyötyjä proseduraalinen sisällöngenerointi tarjoaa pelinkehityksessä ja muussa mediassa. Työssä esitellään yleisimpiä syitä käyttää proseduraalista sisällöngenerointia ja proseduraalisen sisällöngeneroinnin luokittelua.

Syyt on esitelty tutkimusten ja kolmen esimerkin, The Binding of Isaac, SpeedTree ja No Man's Sky, avulla. Sen lisäksi työssä käydään läpi seuraavat tekniikat: näennäissatunnaislukugeneraattorit (PRNG), keskipisteen siirto, kohina, Lindenmayer-systeemi ja avaruuskolonisaatio. Lisäksi projektissa pohjana käytetty L-systeemi-algoritmi on esitelty tarkemmin.

Tämän työn projekti on proseduraalisen sisällöngenerointityökalu, mikä on toteutettu Unity-pelimoottorilla. Työkalun päätarkoitus on luoda ja visualisoida realistisen kaupungin näköinen tiekartta algoritmilla, johon voidaan vaikuttaa pienillä parametrimuutoksilla.

Projektissa kehitetty työkalu auttoi proseduraalisen sisällöngenerointijärjestelmän tutkimisessa. Johtopäätöksenä proseduraalisen sisällöngenerointijärjestelmän tutkiminen ja määrittely antavat tärkeää tietoa sen toteuttamisesta ja käytöstä sekä uusien järjestelmien kehityksessä että valmiiden järjestelmien hyödyntämisessä.

Avainsanat	tiegeneraattori, proseduraalinen sisällöngenerointi, PSG, Unity, proseduraalinen generointi, proseduraalinen kaupungin generointi, kaupunkien generointi

Author(s) Title Number of Pages Date	Juri Palotie Procedural street generator 30 pages + 1 appendices 5 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

The aim of this thesis was to examine what benefits Procedural Content Generation offers in game development and other media. This paper discusses the reasons to use procedural content generation and classification of procedural content generation.

The reasons are presented in studies and with three examples: The Binding of Isaac, SpeedTree and No Man's Sky. In addition, the following techniques are explained: pseudorandom number generator (PRNG), midpoint displacement algorithm, noise, and Lindenmayer system and space colonization. In addition, a basis of the L-system algorithm the project uses is described in more detail.

The outcome of the present project is a procedural content generation tool implemented in the Unity game engine. The main purpose of the tool is to create and visualize a realistic city road map with small parameter changes.

The tool developed helped to study the procedural content generation system. The conclusion is that the study and definition of the procedural content generation system provide important information on its implementation and use for developing new systems, as well as utilizing in ready-made systems.

Keywords	PCG, procedural content generator, street generator, city generator, Unity, procedural city generator

Sisällys

Lyhenteet

PSG	Proseduraalinen Sisällön Generointi, ohjelmallinen sisällön generointi.
RNG	Random Number Generator, satunnaislukugeneraattori.
TRNG	True Random Number Generator, satunnaislukugeneraattori, joka generoi satunnaislukuja jonkin fyysisen ilmiön perusteella (esim. radioaktiivisen aineen hajoaminen).
PRNG	Pseudo-Random Number Generator eli näennäissatunnaislukugeneraattori, generoi deterministisen lukusarjan.

1 Johdanto

Tämä tutkielma esittelee proseduraalisen sisällöngeneroinnin (PSG) perusteorian, pohtii menetelmiä, joita käyttää ja lopuksi kuvailee yhden toteutuksen projektina. Tutkielman tavoitteena on vastata seuraavin kysymyksiin. Miksi proseduraalinen generointi on havaittu hyödylliseksi sisällön luonnissa? Miten voidaan määritellä PSG-järjestelmän ominaisuudet? Mitkä ovat usein käytetyt tekniikat PSG:ssä? Miten Unity-pelimoottoria käytettiin projektin PSG-työkalun toteuttamisessa?

Yleistä määrittelyä PSG:lle ei vielä ole, eikä mitään lopullista tutkimusta tai oppikirjaa aiheesta. Yhteisö on huomannut puutteen ja kirja Togelius, Shaker & Nelson (2015) on kirjoitusvaiheessa, jonka pitäisi täydentää puuttuvat osat. Sillä aikaa kun kirjaa kirjoitetaan laajasti, kehittäjien käytössä oleva wiki (Doull, 2017) sisältää suuren määrän erilaisia tekniikoita, joita on käytetty PSG-alalla.

Pohjimmiltaan PSG on prosessi, jolla luodaan pelin tai muun median sisältö tietokonealgoritmin avulla. Silti jokainen generointijärjestelmä on luotu sopimaan haluttuun lopputulokseen ja käyttää senhetkistä sisältöä, mikä vaatii aina oman lähestymistavan. Tavat, joilla analysoidaan tietty järjestelmä, jotka Togelius, Yannakakis, Stanley & Browne (2011) esittävät on esitetty tässä tutkielmassa käytännön esimerkeillä.

PSG:tä käytetään laajasti peleissä ja muussa mediassa sisällön luomiseen kuten tasoja, alueita ja luolia sekä graafisen sisällön kuten tekstuurien, puiden, jokien. PSG-järjestelmän toteutus ja algoritmin valinta riippuu vahvasti luodusta sisällöstä. Tämän takia tarvitsee määritellä PSG-järjestelmän generointityyli, minkä tämä tutkielma yrittää selittää esittämällä kategoriat, joilla pystytään havainnollistamaan PSG-järjestelmän ominaisuudet ja käyttö: suoritusaikainen (online) vai suunnitteluaikainen (offline), tarpeellinen vai vapaaehtoinen sisältö, siemenarvot vai parametrivektorit, stokastinen vai deterministinen generointi, rakentava vai generoi-ja-testaa.

On monia syitä, miksi käyttää PSG:tä. Seuraavat neljä syytä on määritelty tutkielmassa: muistin kulutus, turha työ manuaalisen sisällön luonnissa, kokonaan uusien pelien syntyminen ja potentiaali laajentaa ihmisen mielikuvitusta. Suurin osa PSG-järjestelmistä on luotu, koska yksi tai useampi näistä syistä on tapahtunut. Syyt eivät ole toisensa poissulkevia, mutta yksi syistä on yleensä pääsyy luoda PSG-järjestelmä. Tutkielmassa esitellään muutaman tosielämän esimerkki, jotka edustavat joitain näistä yleisistä syistä: videopeli The Binding of Isaac, puiden mallintamisohjelma SpeedTree ja videopeli No Man's Sky.

Tämän tutkielman osana on toteuttaa PSG-työkalu, mikä on toteutettu Unity-pelimoottorilla. Työkalun päätarkoitus on luoda ja visualisoida realistisen kaupungin näköinen tiekartta algoritmilla, johon voidaan vaikuttaa pienillä parametrimuutoksilla. Toisin sanoen työkalulla ei luoda alueita tai tasoja peleihin.

2 Proseduraalinen generointi

PSG on tietokoneen ohjaama järjestelmä, mikä käyttää algoritmeja luodakseen halutun sisällön. Tässä luvussa määritellään PSG, niin kuin se on aikaisemmissa tutkimuksissa määritelty ja samalla esitellään syyt, miksi käyttää PSG-järjestelmiä. Aihe jaetaan myös osiin, joiden avulla pystytään määrittelemään yksittäisiä PSG-järjestelmiä.

2.1 Määritelmä

PSG on määritelty monella eri tavalla riippuen henkilöstä, kuka sen on määritellyt. Joillekin PSG on lähtökohtaisesti stokastinen, kun taas muut väittävät, että PSG ei vaadi satunnaisuutta tai näennäissatunnaisuutta saadakseen halutun satunnaisuuden. Esimerkiksi hajautusfunktioita voidaan käyttää satunnaisten tulosten saamiseen ilman satunnaislukuja. Roguelike-pelin kehittäjä Andrew Doull (2008) on määritellyt PSG:n seuraavasti: ”ohjelmallinen pelin sisällön generointi käyttäen satunnais- tai näennäissatunnaisprosessia, jonka tulos on arvaamaton joukko mahdollisia pelitiloja” Togelius, Kastbjerg, Schedl ja Yannakakis (2011) määrittelevät PSG:n näin: ”algoritminen pelin sisällön luonti rajallisilla tai epäsuorilla käyttäjän syöteillä”. He jättävät tarkoituksella satunnaisuuden määrittelystä, sillä he ymmärtävät, että täysin deterministinen PSG-järjestelmä on olemassa. Sisältö PSG-kontekstissa voi olla laaja valikoima erilaista tietoa,

mitä pelit sisältävät. Pelin tyylistä riippuen se voi olla kartta, esineet, tasot, tekstuurit, pelihahmot, tarinat jne. Jokaista NPC- tai tekoäly-käyttäytymistä ei kuitenkaan pidetä sisältönä. Vaikka jotkut PSG-algoritmit voivat sisältää tekoälyalgoritmien ominaisuuksia, kaikki käyttäytyminen pidetään erillään PSG:sta, jotta saadaan selvä sisällön generointiprosessi. (Togelius et al. 2015.)

Termit *proseduraalinen* ja *generointi* viittaavat tietokoneen proseduureihin ja algoritmeihin, jotka generoivat haluttua sisältöä. Tietokone on olennainen osa PSG:tä, koska se ohjaa järjestelmää, mutta ihmisen syötteet voivat olla yhtä olennaisia. Käyttäjän syötteen merkitys määritellään generoidun sisällön ja käyttäjältä vaaditun vuorovaikutuksen määrän perusteella. Mixed-initiative PSG on PSG:n aliaihe, mikä selittää ja määrittelee ihmiseltä vaaditun syötteiden määrän, jolla on vaikutusta lopputulokseen. (Liapis, Smith, & Shaker 2015.)

2.2 Syitä käyttää proseduraalista generointia

Pelikehittäjillä on useita syitä käyttää PSG:tä. Togelius t al. (2011) määrittelee neljä erillistä perustelua: *muistin kulutus, turha työ manuaalisen sisällön luonnissa, kokonaan uusien pelien syntyminen ja potentiaali laajentaa ihmisen mielikuvitusta*. Nämä syyt on selitetty muutamalla tosielämän esimerkillä:

- muistin kulutus – sisältöä voidaan pitää ”pakattuna” kunnes sitä tarvitaan, esim. No Man’s Sky. Pakatulla tarkoitetaan, että sisällön generointiin tarvittavat tiedot ovat tallessa, kunnes niitä tarvitaan.
- turha työ manuaalisen sisällön luonnissa – proseduraalisen generoinnin työkalut tarjoavat kehittäjille nopean tavan luoda suuren määrän sisältöä vaivattomasti muuttamalla muutamaa parametria. Tämä on yhä arvostetumpaa, sillä pelien sisällön odotetaan olevan yhä suurempaa ja yksityiskohtaisempaa, esim. SpeedTree.
- kokonaan uusien pelien syntyminen – pelit, jotka ovat proseduraalisesti generoidut tarjoavat pelaajille loputtoman mahdollisuuden pelata peli uudestaan uudella tavalla, sillä algoritmi luo aina erilaisen sisällön. esim The Binding of Isaac.
- potentiaali laajentaa ihmisen mielikuvitusta – samankaltaisuus on odotettavissa, kun ihminen suunnittelee sisältöä ja suunnitteluaikaiset PSG algoritmit voivat tuottaa tuloksia, jotka innostaa suunnittelijaa ja tuottaa monipuolisemman lopputuloksen. esim. No Man’s Sky.

2.3 Proseduraalisen generoinnin luokittelua

Vaikka ei ole yleisiä tutkimuksia ja julkaisuja, jotka tarjoavat perustiedot PSG:n lähestymistapojen luokittelusta, seuraava Togelius et al. (2011) kuvaamat erot auttavat asentamaan selkeät esimerkit PSG-ääripääparien välille: *suoritusaikainen (online) vai suunnitteluaikainen (offline), tarpeellinen vai vapaaehtoinen sisältö, siemenarvot vai parametrivektorit, stokastinen vai deterministinen generointi, rakentava vai generoi-ja-testaa.* Määrittelemällä järjestelmä näillä pareilla, järjestelmän tarkoitus selkiintyy kehittäjille, käyttäjille ja mahdollisesti muille ihmisille, jotka tarkastelevat ja tutkivat järjestelmää.

2.3.1 Suoritusaikainen (online) vai suunnitteluaikainen (offline)

Ensimmäinen erottava tekijä on, jos sisällön generointi on prosessoitu *suoritusaikaisessa tilassa,* pelin ajonaikana vai *suunnitteluaikaisessa tilassa*, kun peliä kehitetään ennen sen julkaisua. PSG:n suoritusajassa olevalla prosessilla on joitain tiukkoja vaatimuksia: sen pitää olla erittäin nopea, ennustettava ajoaika ja tulokset pitää olla jonkin verran ennustettavissa. Esimerkiksi, suoritusajassa oleva tasogeneraattori luo sisältöä, kun pelin taso ladataan, kun taas suunnitteluaikainen generaattori generoi kehittäjille pohjan tasolle, jonka he sitten muuttavat ja parantelevat tarpeiden mukaan.

2.3.2 Pakollinen vai vapaaehtoinen sisältö

Pakollisen ja vapaaehtoisen sisällön generointi on erottava tekijä, joka vaikuttaa itse pelattavuuteen. Pelaajan on pelattava pakollinen sisältö päästäkseen peli läpi, kun taas vapaaehtoinen sisältö voidaan välttää. Edellistä esimerkkiä käyttäen voidaan sanoa, että generoidut tasot ovat pakollinen sisältö, jotka on pakko läpäistä, kun taas vapaaehtoista sisältöä voisivat olla esineet, joita pelaaja ei välttämättä saa. On tärkeää saada generaattori luomaan pakollinen sisältö oikein, jotta tulokset olisivat toimivat, kun taas vaatimukset vapaaehtoiselle sisällölle eivät ole niin tiukat.

2.3.3 Siemenarvot vai parametrivektorit

Toinen generointialgoritmia määrittävä ero on parametrisoinnin laajuus. Yhdessä ääripäässä algoritmi voi vaatia vain yhden siemenarvon ja generoi koko sisällön, toisessa ääripäässä algoritmi voi ottaa joukon reaaliarvoisia parametreja generoidakseen sisällön. Esimerkiksi PSG-algoritmi voi generoida koko tason yhdestä siemenarvosta tai ottaa rajaavia arvoja tason generointia varten esim. huoneiden määrän ja koon.

2.3.4 Stokastinen vai deterministinen generointi

Sisällön generoinnin satunnaisuus riippuu erittäin paljon siitä, mihin sitä ollaan käyttämässä. Ääripääesimerkeiksi voidaan ottaa Roguelikes-pelin tasojen generointialgoritmi, mikä ei ikinä tuota samanlaista tasoa samalla siemen arvolla, kun taas täysin determinististä järjestelmää voidaan käyttää datan pakkaajana niin kuin No Man’s Sky -pelissä.

2.3.5 Rakentava vai generoi-ja-testaa

Viimeinen ero on algoritmien välillä, jotka voidaan määritellä joko *rakentavaksi* tai *generoi-ja-testaa tyylisiksi.* Rakentavat algoritmit generoivat sisällön kerran ilman mitään seuraavia toimenpiteitä, kun taas generoi-ja-testaa-algoritmit varmistavat, että sisältö on toimiva. Generoi-ja-testaa-algoritmit voivat toteuttaa monenlaisia testiprosesseja saadakseen toimivat tuloksen ja vian tullessa. Jotkut tai kaikki generoitu sisältö tuhotaan ja generoidaan uudestaan. On kehitetty monenlaisia algoritmeja generoidun sisällön oikeellisuuden varmistamiseksi ja tuloksen testaamiseksi. Esimerkiksi tasogeneraattori voisi ajaa testin, mikä tarkistaa, onko mahdollista päästä jokaisen huoneen sisälle ajamalla polunetsintäalgoritmin tason läpi. Kuitenkin tämänkaltaiset tarkistukset riippuvat suuresti, minkälainen sisältö on generoitu ja mitä algoritmia on käytetty. Monimutkaisemmat esimerkit usein käyttävät geneettisiä algoritmeja PSG-järjestelmissä oppiakseen ja kehittääkseen generointiprosessia saadakseen paremmat tulokset.

2.4 Tosielämän esimerkkejä

Tässä luvussa esitellään joitain PSG-esimerkkejä. Ensimmäisenä on videopeli The Binding of Isaac, seikkailupeli vuodelta 2011, missä käytetään PSG-algoritmia uusien polkujen luomiseen joka tasolla. No Man's Sky on vuonna 2016 tullut avaruus-selviytymispeli. Peli voitti Guinness World Records palkinnon suurimmasta pelattavasta maailmasta videopelissä käyttämällä PSG-algoritmia sisällön luonnissa. Lopuksi on proseduraalinen grafiikka ja palkinnon voittanut ohjelma SpeedTree, mitä on käytetty proseduraalisesti generoimaan puita ja muuta kasvillisuutta peleissä ja muussa mediassa, kuten elokuvissa.

2.4.1 The Binding of Isaac (2011)

The Binding of Isaac (kuva 1) on yksi *Roguelikes* peleistä, joka käyttää proseduraalista generointia generoidakseen uudet tasot ja esineet joka kerta, kun peli aloitetaan alusta.

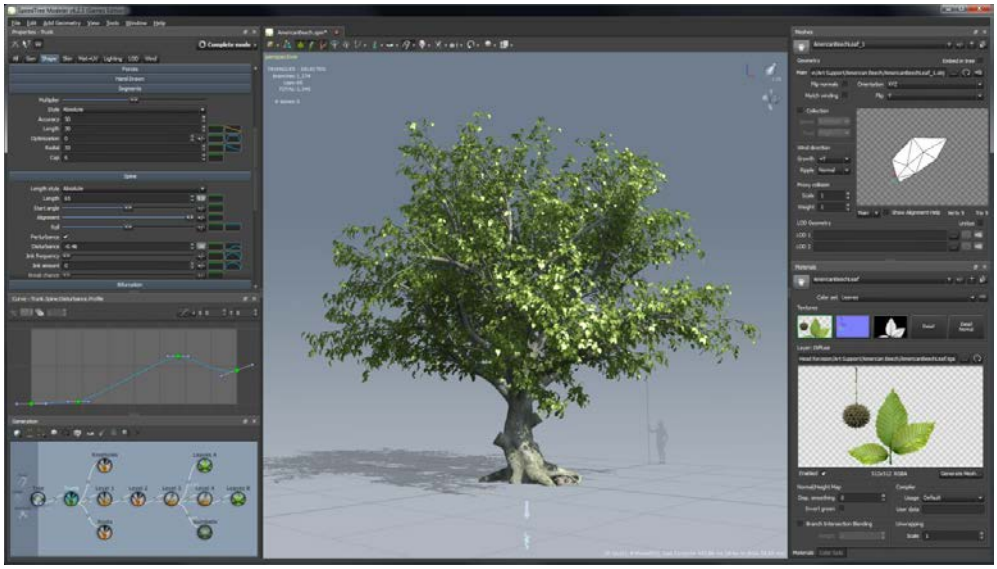
Kuva 1. The Binding of Isaac, kuvassa näkyy kartta yhdestä pelin tasoista

Tämänkaltaiset pelit ovat saaneet ideansa Rogue (1980) pelistä, mikä oli ensimmäisiä pelejä, jotka käytti PSG-järjestelmä tasojen luonnissa ja on malliesimerkki kokonaan uudenlaisesta pelin syntymisestä, kun PSG-järjestelmä on mukana luonnissa. Rogue-pelin vaikutus oli aikanaan niin merkittävä, että se loi oman pelityypin: Roguelikes. Roguelikes-pelit perustuvat suuresti näennäissatunnaisprosesseilla generoituun sisältöön. Tällaiset pelit tyypillisesti tapahtuvat fantasiamaailmassa ja sisältävät erilaisia olentoja ja esineitä, kun taas pelattavuus koostuu olentojen tappamisesta, esineiden keräämisestä

ja etenemisestä satunnaisesti luoduissa tasoissa. Roguelikes-pelit myös sisältävät pysyvän kuoleman, mikä tarkoittaa, kun pelaaja kuolee, hänen on aloitettava peli alusta nollasta. Vaikka pysyvä kuolema oli yleisempi 80- ja 90-luvun peleissä, roguelikes-pelit ovat pitäneet ominaisuutta tähän päivään asti suurimaksi osaksi sen takia, koska PSG tarjoaa niin vaihtelevaa kokemusta pelisessioiden välissä, että kuolema on merkityksetön.

2.4.2 Speedtree

Proseduraalista generointia on usein myös käytetty pelien graafisen sisällön luonnissa ja muussa mediassa Tämä huojentaa kustannuksia, kun ei tarvitse manuaalisesti luoda jokaista kiveä, puuta ja lehteä. Yleisenä esimerkkinä ovat proseduraaliset tekstuurit, joita käytetään realistisen näköisten luonnonmateriaalien generoinnissa (puut, kivi, metalli jne.) Poseduraalinen prosessi takaa, että tekstuurit omistavat samat ominaisuudet, mutta satunnaisuus tekee kaikista tekstuureista vähän erilaiset (muuttaa ne näyttämään luonnonmukaisilta), estää mahdollisesti näkyvän mallin, mikä näkyisi käsin piirretyllä tekstuurilla. SpeedTree (kuva 2) on yksi laajasti käytetty ohjelma, mikä pystyy proseduraalisesti generoimaan monenlaista kasvillisuutta. SpeedTree pystyy generoimaan tekstuurien lisäksi 3D-malleja ja varjostimia puista.



Kuva 2. SpeedTree, puun generointi

2.4.3 No Man’s Sky (2016)

No Man’s Sky (kuva 3) on peli, joka käyttää PSG-järjestelmää generoidakseen lähes koko pelin sisällön.

Kuva 3. No Man’s Sky, kuvassa näkyy joitain olentoja pelistä

Peli pääsi Guinness World Records-kirjaan suurimmasta pelattavasta maailmasta videopelissä sisältämällä 18 quintillion planeettaa. Pelin sisältö on niin suuri, että muisti ei riittäisi lataamaan koko sisältöä heti. Sen takia käytetään PSG-algoritmia planeettojen sisällön tietojen pakkaamisessa ja tarvittavalla hetkellä puretaan ja generoidaan näkyville. Pelissä planeetat ja olennot generoidaan myös PSG-algoritmilla, mikä säästää aikaa ja vaivaa, jos sisältö olisi käsin tehty ja luo uutta sisältö, mihin ihmisen mielikuvitus ei välttämättä riittäisi.

3 Proseduraalisen generoinnin tekniikat

Proseduraalisen generoinnin tekniikat ja algoritmit riippuvat suuresti generoidusta sisällöstä. Yhteisö on koonnut yhteen kokoelman yleisimmin käytetyistä tekniikoista. Jotkut yleisimmin käytetyt tekniikat on selitetty tässä luvussa: *näennäissatunnaislukugeneraattorit (PRNG), keskipisteen siirto*, *kohina, Lindenmayer-systeemi, satunnaispisteet,* ja jotkut vähän vähemmän käytetyt tekniikat, joista yksi on *avaruuskolonisaatio.*

3.1 Näennäissatunnaislukugeneraattori (PRNG)

Satunnaislukugenerointi (RNG) on kenties tärkein yksittäinen tekniikka koko proseduraalisessa generoinnissa. Vaikka sitä ei välttämättä aina tarvita PSG:ssa, siitä saatu arvaamattomuus on loistava työkalu useimmille proseduraalisille prosesseille. Verrattuna todellisiin satunnaislukugeneraattoreihin (TRNG), jotka usein tarvitsevat jonkin fyysisen ilmiön generoidakseen todelliset satunnaisluvut, PRNG:t eivät tarvitse mitään ylimääräistä generoidakseen satunnaisuutta, mikä tekee niistä nopeita ja tehokkaita. PRNG:t ovat myös deterministisiä, joten ne generoivat sarjan numeroita, mikä on mahdollista toistaa, kunhan lähtötiedot ovat tiedossa. Tämänkaltaiset ominaisuudet tekevät PRNG:stä hyvin sopivan proseduraalista generointia varten, kun nopeus on ensisijalla tietokoneprosesseissa ja determinismi auttaa ennustamaan lopputulokset. (Haahr 2017.)

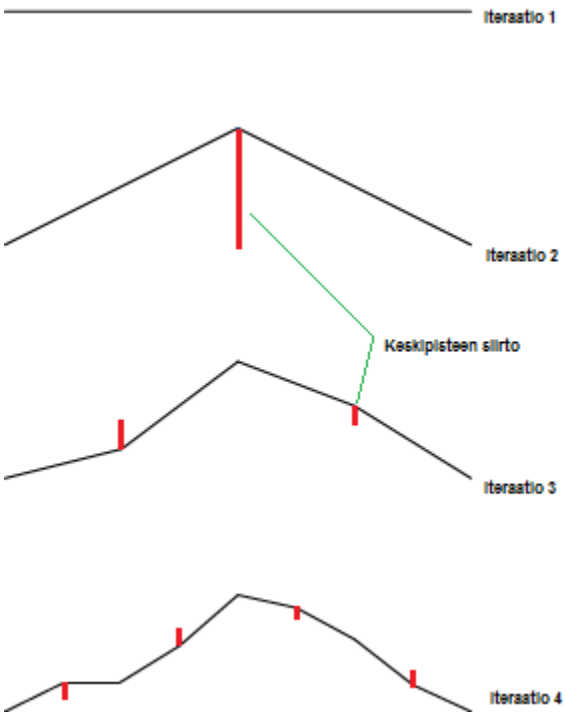
3.2 Korkeuserojen tuottaminen

Luvussa esitetään kaksi yleisesti käytettyä maaston generointialgoritmia: *keskipisteen siirto ja Perlin kohina*. Ensimmäinen algoritmi on fraktaali eli siinä lähetään liikkeelle isoista ja määrittävistä piirteistä, jonka jälkeen toistetaan vaiheet haluttuun tarkkuuteen asti, jotta saadaan yksityiskohtaisemmat piirteet. Toinen algoritmi on gradienttikohinaan perustuva. Ensin luodaan satunnaisarvoista ristikko, jonka jälkeen ristikon väleissä olevien pisteiden avulla lasketaan korkeusarvot.

3.2.1 Keskipisteen siirto

Keskipisteen siirto (midpoint displacement) on fraktaalialgoritmi, missä aluksi lähdetään liikkeelle suorasta viivasta kahden pisteen välillä. Viivan puoliväliin tulee uusi piste, jonka korkeusarvoksi tulee kahden ulomman pisteen keskiarvo, johon lisätään satunnainen arvo (virhearvo). Virhearvo voi olla esimerkiksi satunnaislukugeneraattorin tuottamat arvot halutulta väliltä. Tätä jatketaan rekursiivisesti jokaiselle uudelle viivalle, kunnes haluttu tulos on saavutettu. Tämän lisäksi virhearvon pitäisi määräytyä viivan pituuden avulla. Muussa tapauksessa kaikki keskipisteet voivat hypätä epäsäännöllisesti. Vaihteluväliä pienentämällä alkupiirteet ovat isot ja lättänät, kun taas myöhemmät piirteet ovat

pieniä ja yksityiskohtaisia. Lopullinen kokonaiskuva koostuu isoista määrittävistä piirteistä ja pienistä yksityiskohtaisista piirteistä. (Archer 2011.) Tällaista algoritmia usein käytetään 2D- tai 3D-pelien maaston generoinnissa.



Kuva 4. Keskipisteen siirtoalgoritmi vaiheittain

3.2.2 Kohina

Yksi yleisimmin käytetyistä proseduraalisen generoinnin tekniikoista on gradienttikohina. Ensimmäinen gradienttikohinatoteutus on nimeltään Perlin kohina (kuva 5), minkä kehitti Ken Perlin vuonna 1983. Kohina luodaan generoimalla näennäissatunnaisarvoristikko, mikä sitten interpoloidaan ja saadaan arvot ristikoiden välistä. (Ebert, Musgrave, Peachey, Perlin, & Worley 1994.)

Kuva 5. Perlin Noise

Kohina on yleensä käytetty tekstuurien generoinnissa, koska se generoi tekstuureja ilman näkyviä ristikkoartefakteja. (Perlin 2001). Perlin kohinaa kumminkin käytetään myös muihin PSG-toteutuksiin. Se on usein käytetty maaston ja kartan generoimiseen esimerkiksi erittäin tunnettu proseduraalisesti generoitu peli Minecraft käyttää Perlin kohinaa yhdistämällä eri kohina-arvoja yhteen ja luo sillä loputtoman kuutioisen maaston. (Persson 2011.)

3.3 Lindenmayer-systeemi (L-systeemi)

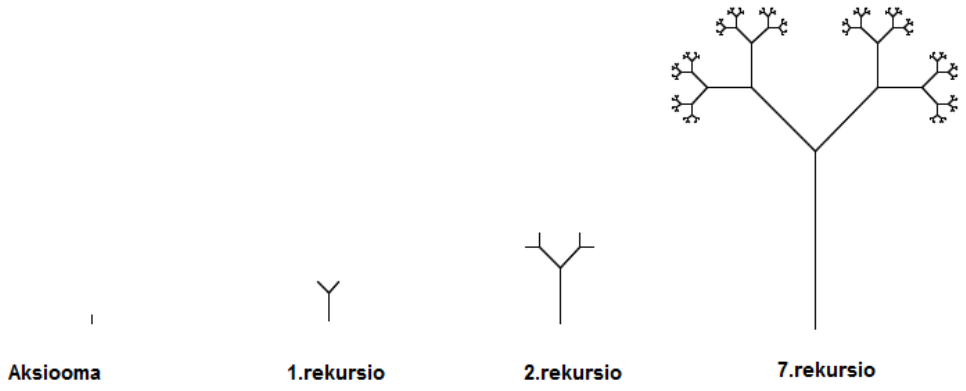
L-systeemi koostuu neljästä osasta: symbolisista aakkosista, joista pystytään muodostamaan merkkijonoja; sääntökokoelmasta, joka laajentaa jokaisen symbolin isompaan symbolien merkkijonoon. Alustava ”aksiooma” merkkijono, josta kokoaminen alkaa, ja mekanismi, joka muuttaa generoidut merkkijonot geometrisiksi kuvioksi.

L-systeemin kehitti unkarilainen kasvitieteilijä Aristid Lindenmayer vuonna 1968. Hän käytti L-systeemiä kuvaamaan kasvisolujen käyttäytymistä ja mallintamaan kasvien kasvuprosessia. Sen takia L-systeemi on usein käytetty generoimaan kasveja proseduraalisesti (kuva 6). Klassinen L-systeemin esimerkki on Pythagoran puu (kuva 6), mikä näyttää, miten L-systeemi toimii ja, miten se muistuttaa kasvien kasvuprosessia.

Esimerkki: Pythagoran puu

- muuttujat: 0, 1
- vakiot: [,]
- aksiooma: 0
- säännöt: (1→11), (0→ 1[0]0)

- 0: piirtää viivan, mikä päättyy lehteen
- 1: piirtää viivan
- [: tallentaa sijainnin listaan ja kääntää 45 astetta vasemmalle
-]: aloittaa tallennetusta sijainnista ja kääntyy 45 astetta oikealle
- aksiooma: 0
- 1.rekursio: 1[0]0
- 2.rekursio: 11[1[0]0]1[0]0
- 3.rekursio: 1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0



Kuva 6. L-systeemin eri rekursiot

Vaikka L-systeemi on tarkoitettu käyttämään kasvien generointia, se on myös muokattu sopimaan esimerkiksi kaupunkien tieverkoston generointiin. (Parish & Müller 2001.) Verrattuna kasvin kasvun simulointiin tieverkoston generointi vaatii erittäin kehittyneen tarkistuksen L-systeemiä käyttäessä. Parishin ja Müllerin toteutus käyttää L-systeemiä vain pohjana ja muokkaa sitä käyttämällä valikoiman rajoitteita ja parametreja, joilla tarkistetaan ja asetellaan tarkistetut tiet näkyville.

3.4 Satunnaispisteet

Kohina on useimmiten käytetty lähtöpisteenä PSG:ssa generoimalla sarjan satunnaispisteitä 2D- tai 3D-avaruudessa. Näennäissatunnaislukugeneraattorin käyttäminen

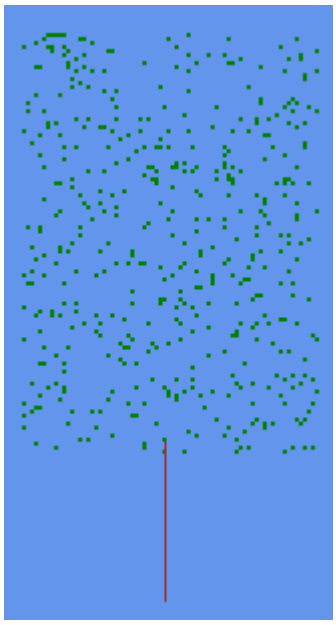
apuna on yhtä yleistä. Kun sarja pisteitä on generoitu, näitä pisteitä voidaan käyttää eri malleissa, kuten *avaruuskolonisaatiota* niin kuin on selitetty seuraavaksi.

3.4.1 Avaruuskolonisaatio

Avaruuskolonisaatio PSG-kontekstissa on prosessi, millä luodaan sisältö täyttämällä rajoitettu tila. Yksi esimerkki tälle on puun generointi: L-systeemiä käyttämällä pystytään generoimaan puita aloittamalla juuresta ja jatkamalla sääntöihin perustuvilla oksilla. Avaruuskolonisaatio aloittaa generoinnin aluksi luomalla lehdet, jotka toimivat oksia puoleensa vetävinä pisteinä. (Gallant 2014.)

Seuraavaksi on Gallantin (2014) selitys, mikä helpottaa ymmärtämään, miten puun generointi toimii avaruuskolonisaatiota käyttäen:

1. Määritellään alue puulle.
2. Asutetaan määritetty alue puoleensa vetävillä pisteillä (lehdet).
3. Luodaan puun runko lisäämällä oksia määritetyn alueen alapuolelle. Kasvatetaan oksia pystysuunnassa; kunnes saavutetaan maksimietäisyys (MaxDistance) lehtien ja oksien välillä. Tämä luo puun rungon (kuva 7). MaxDistance on parametri, mikä määrittelee, kuinka kaukana lehdet voivat olla oksista ennen kuin ne alkavat vetää puoleensa. Lehdet, jotka ovat MaxDistance-etäisyydellä oksista, eivät vedä niitä puoleensa. Tässä kohtaan runkoa muodostavat oksat ovat MaxDistance-etäisyydellä ensimmäisistä lehdistä ja alkavat muodostamaan oksia lehtien suuntiin.



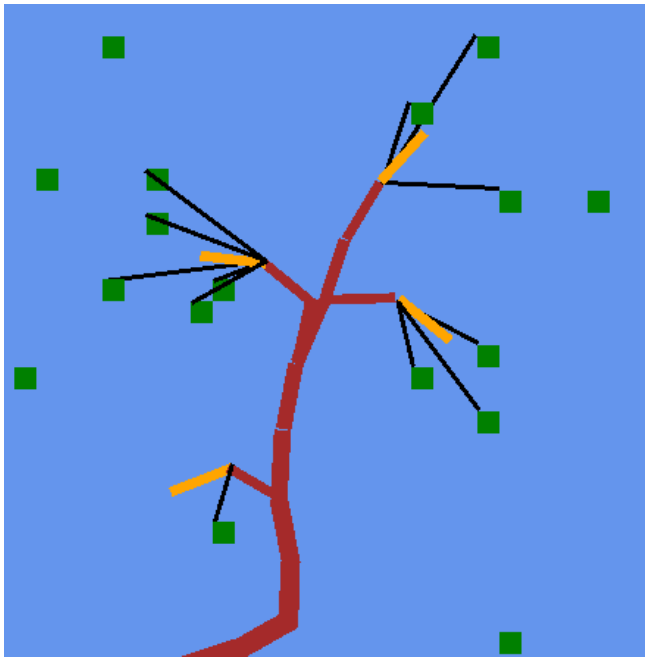
Kuva 7. Puun runko ennen oksien jakaantumista

4. Prosessoidaan lehdet vertaamalla ne jokaiseen oksaan. Lasketaan etäisyys ja suunta lehdestä oksaan. Jos etäisyys on pienempi kuin minimietäisyys (MinDistance), voidaan poistaa lehti, koska se on saavutettu. Jos etäisyys on suurempi kuin MaxDistance, jätetään se huomioimatta, koska se on liian kaukana. Muuten tarkastetaan, onko kyseinen oksa muihin oksiin verrattuna lähimpänä lehteä. Jokainen lehti voi vetää puoleensa vain yhden oksan.

5. Kun lähin oksa on määritelty, voidaan kasvattaa tämän oksan kasvumäärä (GrowCount) ja lisätään lehden suunta oksan kasvusuuntaan (GrowDirection). Jos useampi lehti vetää oksaa puoleensa, silloin GrowDirection on kaikkien niiden lehtien suuntien keskiarvo.

6. Nyt käydään kaikki oksat läpi ja prosessoidaan kaikki oksat, joilla GrowCount > 0. Jaetaan GrowDirection GrowCountilla, jotta saadaan suunnan kesiarvo ja sitten luodaan uusi oksa tällä kasvusuunnalla yhdistämällä se oksaan, mikä on määritelty sen vanhemmaksi (kuva 8). Sitten nollataan vanhemman oksan GrowCount ja GrowDirection.

Toistetaan kohdasta 4, kunnes ei ole enää lehtiä tai oksat eivät enää kasva.



Kuva 8. Avaruuskolonisaatio, oranssit viivat ovat keskiarvosuuntaan kasvavat oksat, mustat viivat kuvaavat lehtien vaiukutuksen.

Avaruuskolonisaatio kuvaa puun kasvuprosessia, mikä sitten tuottaa tuloksena realistisen näköisen puun. L-systeemi tekee saman asian omalla lähestymistavallaan. Kun puhutaan puiden generoinnista, voidaan sanoa, että avaruuskolonisaation on *teleologinen*. Se muodostaa tarkan fysikaalisen mallin ympäristöstä ja prosessista, joka luo generoidun asian, ja sitten ajaa simulaation. Tuotoksen pitäisi näyttää samalta kuin luonnossa, ja L-systeemi on *ontogeninen* eli tarkkaillaan prosessin lopputulosta. Sitten yritetään suoraan toistaa nämä tulokset ad hoc -algoritmeilla. Ontogeniset lähestymistavat käytetään usein reaaliaikaisissa sovelluksissa kuten peleissä. (West 2008.)

4 Projekti: Proseduraalinen tiegeneraattori

Ideana oli saada toteutettua Unity-pelimoottorilla generaattori, joilla pystyy luomaan realistisen näköisen kaupungin tiekartan proseduraalisesti. Projektin idea tuli IVSoftwaren Subversion proseduraalisesta tiegeneraattorista 2007 vuoden versiosta (kuva 9). IVSoftware oli luonut oman pelimoottorin tiegeneraattoria varten ja päätin, että oman pelimoottorin luonti veisi liikaa aikaa ja olisi paljon työtä yhdelle henkilölle. Myöhemmin löysin Danny Goodaylen (Gooddayle, 2015) tekemän proseduraalisen

kaupunkigeneraattorin Unity-pelimoottoria käyttäen (kuva 10) ja päätin, että voin käyttää omassakin projektissa Unitya. Projektin kehitysvaiheessa tuli ideoita, että generaattoriin voisi totetuttaa muutakin sisältöä esimerkiksi talojen generointi, mutta tämä tutkielma keskittyy teiden luontiin, visualisointiin ja proseduraalisten tekniikoiden käyttöön projektissa.

Kuva 9. IVSoftwaren Subversion proseduraalinen tiegeneraattori 2007 vuoden versio.

Kuva 10. Danny Goodaylen kaupunkigeneraattori 2015.

4.1 Luokittelu

Luvun 2.3 mukaan PSG-järjestelmä voidaan luokitella viidellä eri tavalla generaattorin toiminnan perusteella. Projektissa toteutettu tiegeneraattori luokitellaan ensimmäisestä ääripääparista *suoritusaikainen*, koska generointi tapahtuu ajon aikana eikä generoitua tulosta muokata generoinnin jälkeen. Toisesta ääripääparista ei valita kumpaakaan, koska projektia ei erikseen sovelleta mihinkään isompaan kokonaisuuteen, sen takia ei voida sanoa, onko sisältö *pakollinen* vai *vapaaehtoinen*. Kolmannesta ääripääparista valitaan *parametrivektorit*, koska parametreja on useampi eikä niitä ”pakata” myöhempää käyttöä varten. Neljännestä ääripääparista valitaan *stokastinen*, koska generaattori generoi aina eri tuloksen uudella ajolla. Viidennestä ääripääparista valitaan *rakentava*, koska generoinnin jälkeen ei tehdä muita toimenpiteitä.

4.2 Toimintaperiaate

Generointi tapahtuu ensin määrittelemällä alue, mihin tiet generoidaan. Tämän jälkeen L-systeemin tavoin annetaan säännöt, miten tiet generoidaan. Säännöillä määritellään teiden kulmat, milloin mennään eteenpäin, oikealle, vasemmalle ja milloin luodaan uusia teitä. Jotta teistä saadaan luontevan näköiset, käytetään satunnaisarvoja eteenpäin, oikealle, vasemmalle mentäessä ja uusia teitä luodessa. Uudet tiet generoidaan haarautumalla ne jo valmiina olevista teistä satunnaisesti. Uusien teiden generointi rajoitetaan käyttämällä Unityn tarjoamia collidereita. Collideri luodaan tien vanhaan sijaintiin aina, kun tie liikkuu. Kun tie törmää collideriin, pysäytetään tie siihen kohtaan ja lopetetaan uusien teiden haarautuminen kyseisestä tiestä. Vaikka generointi käyttää satunnaisarvoja muuttujina, voidaan silti sanoa, että prosessi on deterministinen, koska samoilla muuttujan arvoilla tulee aina sama lopputulos.

4.3 Käytetyt tekniikat

4.3.1 L-systeemi

Kuten luvussa 3 selitettiin, L-systeemi käyttää merkkijonoja ja sääntöjä generoimaan sisältöä. Niin kuin Parishin ja Müllerin toteutus on muokattu ja käyttää L-systeemiä vaan pohjana, samoin minunkin toteutus on muokattu omaan tarkoitukseen sopivaksi. Oma

L-systeemin toteutus ei käytä merkkijonoja, vaan liikkuu halutun ajan verran haluttuun suuntaan. Toistojen määrä riippuu, kuinka kauan käännytään oikealle tai vasemmalle ja kuinka paljon käännytään oikealle ja vasemmalle ovat vakiot. Havainnollistetaan tätä seuraavalla esimerkillä vertaamalla normaalia L-systeemiä ja minun muokattua:

Esimerkki: Pythagoran puu luvussa 3.3

Vaikka oma menetelmä ei käytä merkkijonoja vaan liikkuu ajan mukaan, yritän esittää, miltä se näyttäisi merkkijonona.

Esimerkki: Muokattu L-systeemi

- muuttujat: 0, 1
- vakiot: +, -
- aksiooma: 0
- säännöt: (0→1), (1→ 1+), (+→10)), (1→ 1-), (-→10)
- 0: piirtää viivan, mikä päättyy lehteen
- 1: piirtää viivan
- +: kääntää 45 astetta oikealle
- -: kääntää 45 astetta vasemmalle
- aksiooma: 0
- 1.rekursio: 1
- 2.rekursio: 1+
- 3.rekursio: 1+10
- 4.rekursio: 1+1+1

Kun aika on kulunut, aloitetaan samasta sijainnista ja tehdään sama alusta kääntyen vasemmalle. Tämä prosessi tehdään jokaiselle tielle erikseen. Lopuksi voidaan yhdistää molemmat merkkijonot ja saadaan lopputulos (1+1+11-1-1).

4.3.2 Satunnaisarvot

Satunnaisarvoja käytetään projektissa teiden kulmien muutoksesta uusien teiden generointiin. Satunnaisarvojen generointiin käytetään Unityn Random-luokkaa, mikä käyttää näennäissatunnaislukugeneraattoria generoidakseen arvoja. Kuten luvussa 2 selitettiin, PRNG ei ole yhtä satunnainen kuin TRNG, mutta tähän tarkoitukseen se soveltuu hyvin, koska prioriteettina on nopeus eikä niinkään satunnaisuus. Satunnaisarvolla saadaan teistä satunnaisia ja luontevan näköisiä.

4.3.3 Tekninen kuvaus

Seuraava selitys helpottaa ymmärtämään, miten algoritmi toimii ja mitä komentoja käytetään prosessissa.

Kuva 11. Yksinkertaisen tien generointiprosessi projektissa.

1. Ensin määritellään alue, minkä sisään generoidaan kaupunki. Tällä *if*-lauseella, missä koordinaatit x ja z asetetaan rajaaviksi tekijöiksi (kuva 12).

```
if (transform.position.z < Areaz && transform.position.z > -Areaz && transform.position.x < Areax
    && transform.position.x > -Areax) {
```

Kuva 12. Projektin alueen rajaus koodissa.

2. Kuvassa 14 kohdassa 1. piirretään tietä 1.6 sekunnin ajan. Sen jälkeen katsotaan, onko x=1(kuva 13), koska x ei ollut kyseisellä hetkellä 1, niin jatketaan tietä edelleen eteenpäin. Koodissa käytetään Unityn Random-luokkaa saadakseen satunnaisesti arvon 1.

```
x = Random.Range (1, CreateStraightRoad);
```

Kuva 13. Koodissa x-arvon määritys.

3. Kuvan 11 kohdassa 2 x sai arvoksi 1 ja silloin käännetään tietä 90 astetta vasemmalle. Tämä tapahtuu käyttämällä Unityn tarjoamaa *instantiate*-funktiota (kuva 17), mikä tekee kopion halutusta objektista.

```
clone = Instantiate (gameObject, transform.position, Quaternion.Euler (new Vector3 (0,angle, 0)));
```

Kuva 14. Instantiate-funktio koodissa.

4. Kohdassa 3 tien generointia jatketaan seuraavaksi kääntämällä 90 astetta oikealle ja uudestaan kohdassa 4. Oikealle ja vasemmalle kääntymistä hallitaan myös Unityn Random-luokkaa käyttäen.

```
angle = transform.eulerAngles.y + list [Random.Range (0, 2)];
```

Kuva 15. Kulman määrittely koodissa.

Niin kuin tämän luvun alussa kerrottiin, projektin algoritmi käyttää L-systeemiä vain pohjana. Ensimmäinen ero on, että mennään ajan mukaan eikä luoda merkkijonoa. Jos käytettäisiin merkkijonoja, jouduttaisiin kirjoittamaan aina merkki, kun liikutaan eteenpäin, oikealle tai vasemmalle. Tästä tulisi erittäin pitkä merkkijono, ja se hidastaisi prosessia merkittävästi. Toinen merkittävä ero on, että generointi tapahtuu ajonaikana, eikä niin

kuin L-systeemissä, missä ensin generoidaan merkkijono ja sitten piirretään niin kuin selitetty luvuissa 3.3 ja 3.4.1.

4.4 Tiegeneraattori

Tässä luvussa kerrotaan ja näytetään, miten Unity-editorin liitteessä 1 mainitut asiat on hyödynnetty teiden generoinnissa. Inspector-ikkuna käytetään datan ja parametrien hallitsemiseen. Scene-näkymää käytetään aloitus sijaintien määrittämiseen ja testaukseen.

4.4.1 Inspector-ikkuna

Tiegeneraattorin inspector-ikkunassa käyttäjä pystyy muuttamaan generaattorin parametreja, joilla luodaan teitä (kuva 18). Samalla käyttäjä näkee datamuutokset, jotka tapahtuvat generoinnin aikana esimerkiksi sijainti, teiden lukumäärä ja niin edelleen.

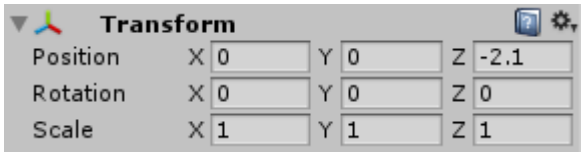
4.4.2 Scene-näkymä

Generaattori käyttää kuutioita, viivoja ja triggereitä luodakseen sisältöä. Kuutioilla pystytään määrittelemään aloitussijainti generoinnille. Unity tarjoaa oman ohjaimen sijainnin määrittelylle, missä käytetään xyz-koordinaatistoa, joissa x on leveys, y on korkeus ja z on etäisyys eteenpäin. Generaattori käyttää vain x- ja z-akseleita, joilla määritellään sijainti.

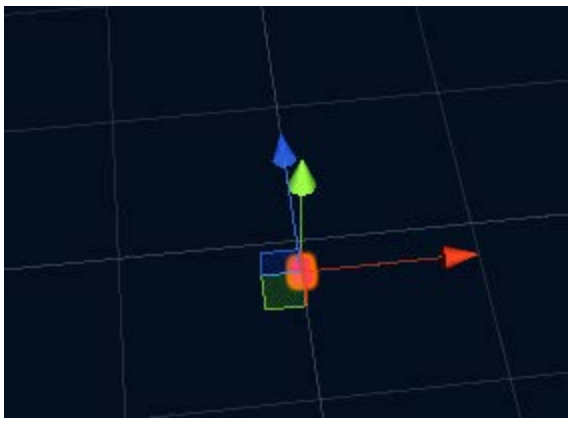
4.4.3 Generointi

Tässä luvussa näytetään ja kerrotaan, miten generointi tapahtuu Unityssä. Ensin valitaan aloitussijainti. Tämä voidaan tehdä kahdella eri tavalla, joko käytetään inspecor-ikkunaa ja muutetaan arvot kirjoittamalla ne itse (kuva 16) tai scene-näkymässä siirretään

objektia käyttämällä Unityn tarjoamaa ohjainta (kuva 17).

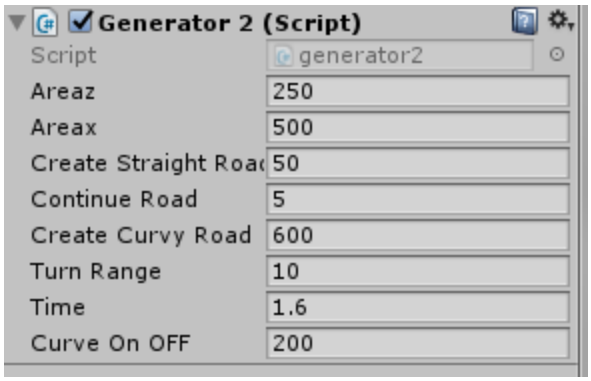


Kuva 16. Inspector-ikkunan Transform-komponentti.



Kuva 17. Scene-näkymä ja valittu objekti.

Parametrien asettelu tapahtuu inspector-ikkunassa (kuva 18).



Kuva 18. Inspector-ikkunan Generator 2-komponentti.

Parametrit on selitetty kuvan 18 parametrien järjestyksen mukaan:

Areaz: Määritellään rajatun alueen z-arvo.

Areax: Määritellään rajatun alueen x-arvo.

Create Straight Road: Mikä mahdollisuus luoda uusi suora tie.

Continue Road: Mahdollisuus jatkaa tietä kurvikkaana tienä, kun on saavutettu *CurveONOFF* raja.

Turn Range: Kuinka paljon tie kulma muuttuu oikealle ja vasemmalle kääntyessä.

Time: Kuinka kauan käännytään oikealle ja vasemmalle.

Curve ON OFF: Milloin voidaan alkaa kääntää tietä.

Lopputulos näyttää tältä(kuva 19):

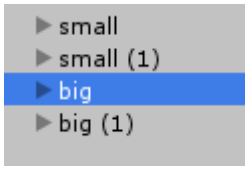
Kuva 19. Generoitu kaupunki edellä esitetyillä parametreilla.

4.5 Vertailua

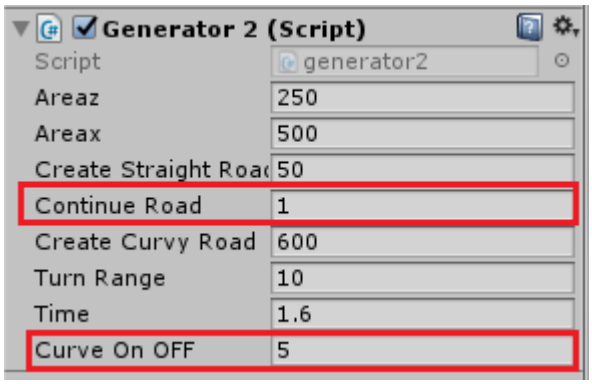
Seuraavaksi verrataan kuvassa 19 olevaa tulosta toiseen generoituun tulokseen (kuva 20) ja verrataan, miten parametrit vaikuttivat tulokseen.

Kuva 20. Toinen generoitu tulos uusilla parametreilla: isot tiet(valkoinen) ja pienet tiet(punainen).

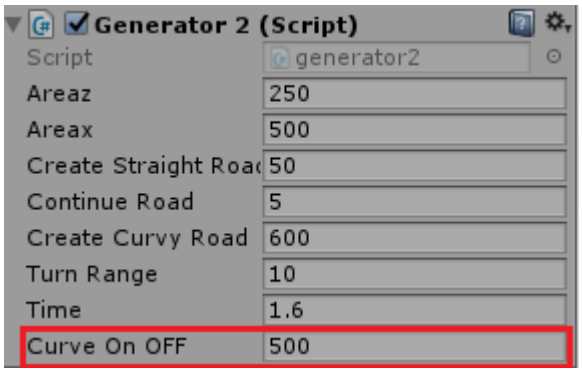
Huomataan, että kaupunki on nyt suurempi ja näkyy erikokoisia teitä. Tämä tulos saadaan lisäämällä uusia aloituspisteitä, mitkä näkyvät helposti *hierarchy*-näkymässä(kuva 21) ja muuttamalla parametrien arvoja (kuva 22 ja kuva 23).



Kuva 21. Hierarchy-näkymä, missä on kaksi aloituspistettä pienille teille (small) ja isoille teille (big).



Kuva 22. Punaisella ovat merkitty muutettu parametrit, jotta saadaan generoitua isot tiet.



Kuva 23. Punaisella on merkitty muutettu parametri, jotta saadaan generoitua pienet tiet isommalle alueelle.

Lopuksi verrataan edellisiä tuloksia oikean kaupungin tiekarttaan (kuva 24). Huomataan, että tiet ovat jaettu tasaisin välein ja osa pienistä teistä alkaa mutkittelemaan kaupungin keskuksessa. Näin tarkkoihin yksityiskohtiin projektin tiegeneraattori ei pysty. Vaikka tulokset eivät saavuta ihan oikean kaupungin tasoa, samankaltaisuuksia kumminkin näkyy.

Kuva 24. Los Angelesin tiekartta.

5 Yhteenveto

Tämän tutkielman ensisijainen tarkoitus oli esittää, määritellä ja luokitella PSG kokonaisuudessa ja samalla esittää joitain käytettyjä tekniikoita. Toisena tutkielman tarkoituksena oli tutkia kaupungin generointia ja PSG:tä Unity-pelimoottorissa. Johdantoluvun oli tarkoitus tutustuttaa lukija PSG-järjestelmään yleisesti ennen kuin esitellään PSG-järjestelmä esimerkeillä ja näytetään toteutettu toteutus kaupungin generoinnista.

Miksi proseduraalinen generointi on havaittu hyödylliseksi sisällön luonnissa?

Perustelut PSG-järjestelmän hyödyllisyydelle on esitelty ja kerrottu luvussa 2.2: *muistin kulutus, turha työ manuaalisen sisällön luonnissa, kokonaan uusien pelien syntyminen ja potentiaali laajentaa ihmisen mielikuvitusta*. Nämä syyt antavat pohjan päätökselle toteuttaa PSG-järjestelmä sisällön luonnissa ja yleensä yksi näistä syistä on pääsyy käyttää PSG:tä.

Nämä syyt myös selitettiin tarkemmin esimerkeillä luvussa 2.4. Ensimmäinen esimerkki oli The Binding of Isaac -videopeli, jossa PSG-järjestelmää käytettiin uusien tasojen luonnissa. Toinen esimerkki oli proseduraalisesta kasvillisuuden generaattorista SpeedTreestä, missä ohjelmaa käytettiin helpottamaan suunnittelijoiden työtä lisäämällä generoidut puut ja muu kasvillisuus sisältöön. Kolmas esimerkki on videopelistä No Man's Sky, missä PSG-järjestelmää hyödynnettiin auttamalla suunnittelijoita luomaan uusia eläimiä ja vähentämään muistin kulutusta pelissä.

Miten voidaan määritellä PSG-järjestelmän ominaisuudet?

PSG-järjestelmä voidaan määritellä vertailemalla kutakin **viiden ääripääparin** kanssa, jotka esiteltiin luvussa 2.3. **Ensimmäinen** pari vertailee, missä tilassa prosessi suoritetaan: *suoritusaikana* vai *suunnitteluaikana*? **Toinen** pari määrittää, onko sisältö *pakollinen vai vapaaehtoinen*? **Kolmas** määrittää, kuinka tarvittavien parametrien määrän vertaamalla *siemenarvoja* ja *parametrivektoreita*. **Neljäs** määrittää, kuinka satunnaista generointi on, vertaamalla *stokastista* ja *determinististä* generointia. Viimeisenä **viidentenä** parina vertaillaan, minkälainen tarkistus tehdään generoidulle tulokselle *rakentava* vai *generoi-ja-testaa*.

Mitkä ovat usein käytetyt tekniikat PSG:ssä?

Usein käytetyt tekniikat on esitelty ja selitetty luvussa 3. **Näennäissatunnaislukugeneraattorit** toimivat satunnaisuuden pohjana PSG-järjestelmissä ja niitä käytetään

usein muiden tekniikoiden kanssa samanaikaisesti. Sen takia ne esitellään ensimmäisenä. Muita tekniikoita olivat **keskipisteen siirto** ja **kohina**, joilla usein luodaan maastoa, mutta myös tekstureita, **L-systeemi**, **satunnaispisteet** ja lopuksi **avaruuskolonisaatio**, mikä käyttää satunnaispisteitä generoinnissa.

Miten Unity-pelimoottoria käytettiin projektin PSG-työkalun toteuttamisessa?

Projekti, joka toteutettiin tätä tutkielmaa varten, käyttää Unity-pelimoottoria. Unity tarjoaa kolme työkalua, joilla pystyttiin kehittämään ja visualisoimaan projektin tulos. Nämä työkalut esiteltiin liitteessä 1: **Inspector-ikkuna** , **Scene-näkymä** ja **Hierachy-ikkuna** . Näiden työkalujen käyttö projektissa selitettiin luvussa 4.4.

Lähteet

Archer, T. 2011. Procedurally Generating Terrain. http://micsymposium.org/mics_2011_proceedings/mics2011_submission_30.pdf. 3.4.2017.

Doull, A. (2008). *The death of the level designer.* http://roguelikedeveloper.blogspot.fi/2008/01/death-of-level-designer-procedural.html. 3.4.2017.

Doull, A. (2015). *Procedural Content Generation Wiki*. http://pcg.wikidot.com/. 3.4.2017.

Ebert, D., Musgrave, K., Peachey, D., Perlin, K., & Worley, S. (1994). *Texturing and Modeling: A Procedural Approach.* Academic Press. 3.4.2017.

Eilertsen, B. G. (2013). *Automatic road network generation with.* https://brage.bibsys.no/xmlui/bitstream/handle/11250/253548/663032_FULLTEXT01.pdf?sequence=3&isAllowed=y. 3.4.2017.

Gallant, J. (2014). *Procedurally Generated Trees with Space Colonization Algorithm in XNA C#*. http://www.jgallant.com/procedurally-generating-trees-with-space-colonization-algorithm-in-xna/. 3.4.2017.

George, K., & Hugh, M. (2007). *Citygen: An Interactive System for Procedural City Generation.* http://www.citygen.net/files/citygen_gdtw07.pdf. 3.4.2017.

Haahr, M. (2016). *Introduction to Randomness and Random Numbers*. https://www.random.org/randomness/. 3.4.2017.

Haahr, M., & Haahr, S. (2016). *The History of RANDOM.ORG*. https://www.random.org/history/. 3.4.2017.

Liapis, A., Smith, G., & Shaker, N. (2015). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Mixed-initiative.* http://pcgbook.com/. 3.4.2017.

Parish, Y. I., & Müller, P. (2001). *Procedural Modeling of Cities.* http://www.cs.berkeley.edu/~sequin/PAPERS/Parish_Mueller_Cities.pdf. 3.4.2017.

Perlin, K. (2001). *Standard for perlin noise.* http://www.google.com/patents/US6867776. 3.4.2017.

Persson, M. (2011). *Terrain generation, Part 1*. http://notch.tumblr.com/post/3746989361/terrain-generation-part-1. 3.4.2017.

Thompson, M. (2015). *EVALUATING THE HYBRIDISATION OF PROCEDURAL CONTENT GENERATION WITH A DESIGN-CENTRIC EDITOR.* http://www.markthompsonportfolio.com/uploads/2/6/1/6/26160187/markthompson_dissertation.pdf. 3.4.2017.

Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. N. (2011). *What is procedural content generation?: Mario on the borderline. In: Proceed-ings of the 2nd Workshop on Procedural Content Generation in Games.* http://www.ccs.neu.edu/course/cs5150f14/readings/togelius_what.pdf. 3.4.2017.

Togelius, J., Shaker, N., & Nelson, M. J. (2015). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* http://pcgbook.com/. 3.4.2017.

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). *Search-based Procedural Content Generation: A Taxonomy and Survey.* http://julian.togelius.com/Togelius2011Searchbased.pdf. 3.4.2017.

Unity Technologies. (2016a). *Unity Documentation, Handles*. http://docs.unity3d.com/ScriptReference/Handles.html. 3.4.2017.

Unity Technologies. (2016b). *Unity Manual: Script Serialization*. http://docs.unity3d.com/Manual/script-Serialization.html. 3.4.2017.

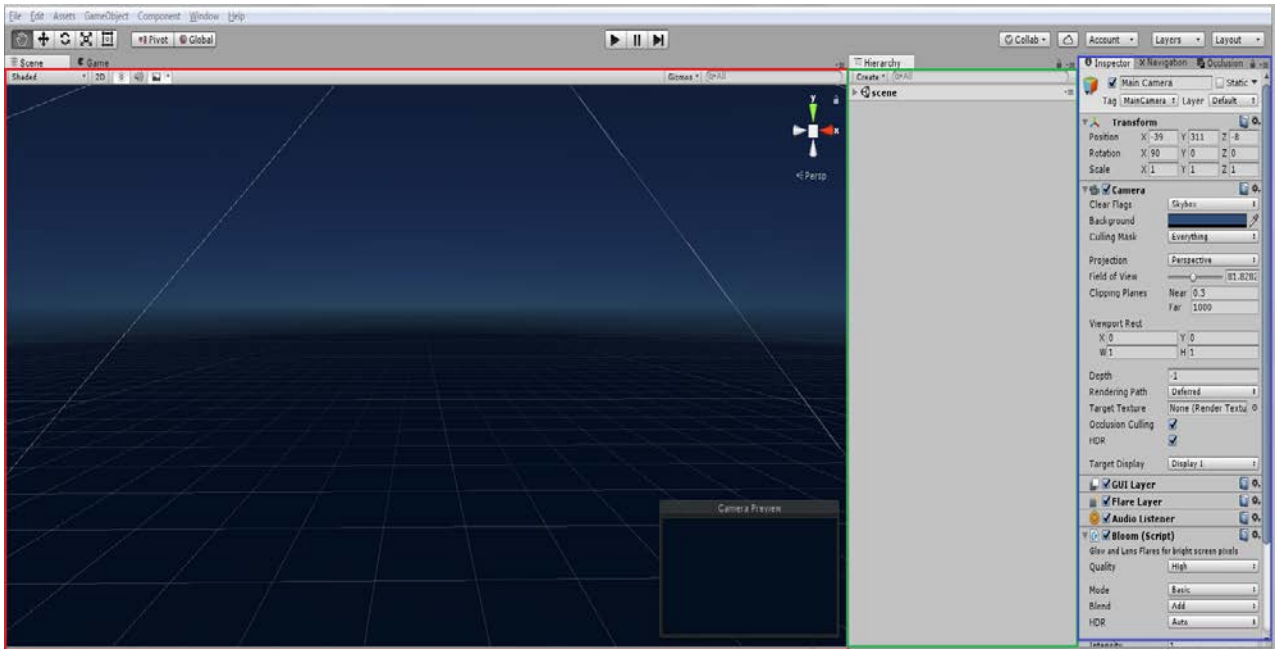
Walker, J. (2006). *HotBits: Genuine random numbers, generated by radioactive decay*. fourmilab.ch: http://www.fourmilab.ch/hotbits/. 3.4.2017.

West, M. (2008). *Random Scattering: Creating Realistic Landscapes*. http://www.gamasutra.com/view/feature/130071/random_scattering_creating_.php?page=2. 3.4.2017.

Worley, S. (1996). *A Cellular Texture Basis Function.* http://www.rhythmiccanvas.com/research/papers/worley.pdf. 3.4.2017.

Unity

Tässä luvussa esitellään Unity editoria. Unity:ssä on kolme näkymää, joissa käyttäjä pystyy muokkaa omaa sovellusta haluamallansa tavalla *Inspector Window, Hierarchy Window* ja *Scene View* (kuva 25). *Inspector*-ikkunassa näkyy kaikki informaatio jokaisesta valitusta komponentista. Tätä ikkunaa pystytään muokkaamaan skripteilä, jotka viittaavat halutun komponentin inspector-ikkunaan. Hierarchy-ikkunassa nähdään jokaisen objektin hierarkkinen asema, tätä muokkaamalla voidaan muuttaa, miten mikäkin objekti näkyy Game ja Scene-näkymässä. Viimeisenä on Scene-näkymä, missä käyttäjä pystyy manipuloimaan objektien sijaintia, muotoa, jne. Scene-näkymään pystytään myös lisäämään omat GUIt (graaffinen käyttöliittymä) objekteja varten.



Kuva 25. Unity-editori näyttää Scene(punainen), Hierarchy(vihreä) ja Inspector(sininen).

Inspector-ikkuna

Unityssä inspector-ikkuna on ikkuna, missä näkyy kaikki informaatio valituista objekteista. Unity luo kaikille objekteille, jotka ovat scene-näkymässä transform-komponentin sekä muita komponentteja fysiikan simulointi, pelattavuus, ääni ja grafiikka. Jokainen luokka, mikä perii MonoBehaviourin, voidaan lisää peliobjektiin ja saa oman inspector-

ikkunan muiden komponenttien lisäksi. Inpectorin ikkuna sisältää kaikki sarjalliset kentät oletuksena, joita on mahdollista muokata itselle sopiviksi.

Scene-näkymä

Scene on Unityn näkymä, missä pystytään hallitsemaan pelinäkymää. Scene-näkymän ja pelinäkymän kamerat ovat erilliset ja scene kameraa voidaan liikutella editoinnin aikana. Näkymää voidaan muokata lisäämällä omia ohjaimia ja GUItä, joilla käyttäjä voi manipuloida dataa.

Ohjaimilla pystytään manipuloimaan 2D- ja 3D-objektien kokoja, muotoja ja sijaintia. 2D- ja 3D-objekteilla on omat ohjaimet, jotka Unity tarjoaa.

Scene-näkymässä voidaan liittää graafisia käyttöliittymiä. Tämä antaa kehittäjille mahdollisuuden keskittyä mahdollisimman paljon scene-näkymässä ja silti manipuloida objektien dataa. Tallainen asettelu helpottaa ja nopeuttaa testausta ja kehitystä.

Hierachy-ikkuna

Näkymässä nähdään kaikki objektit, jotka ovat scene-näkymässä. Näkymässä voidaan piilotta ja järjestää objekteja halutulla tavalla. Hierarchy-ikkuna tarjoaa myös nopean pääsyn objektien inspector-ikkunaan.