KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Business Information Technology

Henri Viitanen

PROCEDURAL CITY GENERATION TOOL WITH UNITY GAME ENGINE

Thesis
February 2016

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
February 2016
Degree Programme in Business Information Technology

Karjalankatu 3
80220 JOENSUU
FINLAND
Tel. +358 13 260 6800

Author
Henri Viitanen

Title
Procedural City Generation Tool With Unity Game Engine

Commissioned by
Mental Moustache Ltd.

Abstract

The aim of this thesis was to study procedural content generation (PCG) through its usage in game development. The thesis discusses the benefits of utilizing PCG in games, focusing more closely in city generation. An open-source city generation tool, EdgeGraph, was developed to support this thesis and its implementation was described in detail.

Three city generation tools implemented with Unity game engine, EdgeGraph, Horizon: City Generator, and CityScaper, were analysed and compared. The ways to dissect a PCG-system introduced in this thesis were utilised in the analysis of each compared tool. The aim of the comparison was to examine usage and implementation differences between the tools. In addition, the reasons to use PCG in games were introduced through previous research and three examples: Elite, Rogue, and SpeedTree. The following often-used techniques in PCG were also introduced in this thesis in order to illustrate implementation of a PCG system: pseudo-random number generators, gradient noise, Lindenmayer systems, and random points. Additionally, the technique of space colonization and its usage in the EdgeGraph was explained more thoroughly.

The EdgeGraph provided a viable platform in researching the implementation of PCG systems. It was concluded through comparison of the EdgeGraph with the two other tools that PCG systems with similar intentions could differ greatly in implementation and usage. In conclusion, analysing and dissecting a PCG system provides valuable information in its implementation and use cases, both during development of a new system, and when utilising a complete system.

Language

English

Pages 56

Keywords

game development, procedural generation, procedural content generation, PCG, Unity, procedural city generation, city generation, EdgeGraph, EdgeBuilder

Karelia UNIVERSITY OF APPLIED SCIENCES	**OPINNÄYTETYÖ** **Helmikuu 2016** **Tietojenkäsittelyn koulutusohjelma** Karjalankatu 3 80220 JOENSUU p. 013 260 6800

Tekijä
Henri Viitanen

Nimeke
Proseduraalinen kaupunkien generointityökalu Unity-pelimoottorilla

Toimeksiantaja
Mental Moustache Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli tutkia, miten proseduraalista sisällön generointia (myöhemmin PSG) hyödynnetään pelinkehityksessä. Työssä selvitetään, mitä etuja proseduraalisesta generoinnista on pelinkehityksessä. Työssä keskitytään tarkemmin kaupunkien generointiin ja sen tueksi kehitettiin avoimen lähdekoodin kaupunkien generointityökalu, EdgeGraph, jonka toteutus esitellään työssä yksityiskohtaisesti.

Työssä analysoidaan ja vertaillaan kolmea Unity-pelimoottorilla toteutettua kaupunkigenerointityökalua, työhön kehitettyä EdgeGraphia, Horizon: City Generatoria, ja CityScaperia, hyödyntäen työssä esiteltyjä tapoja tarkastella PSG-järjestelmiä. Vertailun tavoitteena on selvittää työkalujen käyttötarkoituksia ja toteutustapoja. Lisäksi työssä esitellään perimmäiset syyt PSG-järjestelmien hyödyntämiselle peleissä aikaisempien tutkimusten sekä kolmen esimerkin, Eliten, Roguen ja SpeedTreen, avulla. Työssä esitellään myös seuraavat usein käytetyt PSG-tekniikat havainnollistamaan PSG-järjestelmien toteutusta: näennäissatunnaislukugeneraattorit, gradientti kohina, Lindenmayer-järjestelmät ja satunnaiset pisteet. Lisäksi EdgeGraph-työkalussa hyödynnetty tilantäyttötekniikka selvitetään tarkemmin.

Työhön kehitetty EdgeGraph tarjosi hyödyllisen alustan PSG-järjestelmien tutkimiseen. EdgeGraph-työkalun vertailu kahteen muuhun työkaluun osoitti, että pinnallisesti samankaltaisten PSG-järjestelmien toteutustavat voivat erota toisistaan merkittävästi. Johtopäätöksenä PSG-järjestelmän analysointi ja määrittely tarjoavat tärkeää tietoa sen toteutus- ja käyttötavoista sekä uuden järjestelmän kehityksessä että valmiin järjestelmän hyödyntämisessä.

Kieli	Sivuja 56
englanti	

Asiasanat

pelinkehitys, proseduraalinen generointi, proseduraalinen sisällön generointi, PSG, Unity, proseduraalinen kaupunkien generointi, kaupunkien generointi, EdgeGraph, EdgeBuilder

Contents

1 Introduction

This thesis will introduce basic theory of procedural content generation (PCG), discuss the choice of methods to use, and lastly describe one implementation as the case study. The goal of this thesis is to answer to the following questions. Why procedural generation is found useful in content creation? How to define the desired properties of a PCG system? What are some often-used techniques used in PCG? How was the Unity game engine used in implementing the PCG tool of the case study?

The definition of procedural content generation will be introduced in this thesis. A conclusive definition of the procedural content generation does not exist yet, and there are no definitive researches or textbooks about the subject. The lack of definitive study is recognized and a book by Togelius, Shaker & Nelson (2015) is underway to fill the gap. While the book is not yet complete, a wiki (Doull, 2015) containing a vast number of different techniques utilized in the PCG field is widely used by the developers.

Essentially, the procedural content generation is the process of generating game (or other media) content with the help of computer algorithms. Still, each generation system is created to suit the desired result and use target at the time, and each type of content requires its own type of approach. Therefore, a single implementation of a PCG system is easier to define, and the ways to dissect a specific system presented by Togelius, Yannakakis, Stanley & Browne (2011) are introduced in this thesis through a practical example.

There are a number of reasons for using procedural content generation. The following four reasons are defined in this thesis: *memory consumption*, *prohibitive expense of manually creating game content*, *emergence of completely new types of games*, and *potential to augment human imagination*. The existence of most

of the PCG systems can be articulated to happen because of at least one of them. The reasons are not mutually exclusive, but usually one of the reasons is the main reason for the development of a PCG system. There will be a few real life examples introduced that represent some of these common reasons: video games Elite and Rogue, and a tree modelling software SpeedTree.

Procedural content generation is used widely in games and other media to generate the game content like levels, maps, and dungeons as well as graphical content like textures, rocks, and trees. The implementation of the PCG system and the choice of algorithms depend heavily on the content being generated. This leads to the need to define the behaviour of the PCG system, so this thesis will try to satisfy this need by presenting a group of pairs of extremes, between which a PCG system can be placed to help define its characteristics and usage: *online versus offline, necessary versus optional content, random seeds versus parameter vectors, stochastic versus deterministic generation*, and *constructive versus generate-and-test.*

The case study of this thesis is a procedural content generation tool implemented in Unity game engine editor. The primary purpose of the tool is to provide a way for level designers to generate cities in a way that they manifest as gameplay spaces. That is to say, the tool is not meant to generate realistic looking cityscapes, as large cities would be too complex and repetitive in first[1] or third[2] person games.

[1] In first person perspective, the game is rendered from the viewpoint of the player character.

[2] In third person perspective, the player character is visible on the screen.

2 Procedural content generation

Procedural content generator is a computer driven system that uses algorithms to produce desired content. This chapter will define the subject as it has been defined in earlier research as well as present the reasons to utilize a PCG system. The subject will also be dissected to reveal the ways to define individual PCG systems.

2.1 Definitions

Procedural content generation has been defined differently by various people. For someone PCG is inherently stochastic[3], while someone might argue that PCG does not require random or pseudo-random process to have the desired unpredictability. For example, hash functions[4] can be used to generate unpredictable results without random numbers. According to a Roguelike developer Andrew Doull (2008) PCG is "the programmatic generation of game content using a random or pseudo-random process that results in an unpredictable range of possible game play spaces." Togelius, Kastbjerg, Schedl, and Yannakakis (2011) define PCG as "the algorithmical creation of game content with limited or indirect user input". They deliberately leave the randomness out of the definition because they recognise the existence of entirely deterministic[5] PCG systems.

---

[3] *Stochastic* is a term used to describe a system, which is unpredictable due to a random variable.

[4] *A hash function* is any function that can be used to map digital data of arbitrary size to digital data of fixed size, with slight differences in input data producing very big differences in output data.

[5] *Deterministic* system is not affected by randomness in creating the future states. A deterministic system will produce the same outcome given the same beginning state.

The *content* in context of PCG can be a wide variety of data that games contain. Depending on the type of game, it can be maps, items, game rules, textures, characters, stories, weapons etc. Any non-player character or AI behaviour, however, is not considered *content*. While some PCG algorithms might include characteristics from AI algorithms, all behaviour is kept separate from PCG in order to clarify the content generation process. (Togelius et al., 2015)

The terms *procedural* and *generation* refer to the computer procedures and algorithms that generate some output. The computer is the essential part of PCG as it drives the system, but human input can be equally important. The kind of content that is generated and the amount of oversight wanted from the user determines the significance of the user input. The sub topic of PCG discussing and defining the amount of human input that influences the final output is called mixed-initiative procedural content generation. (Liapis, Smith, & Shaker, 2015)

2.2 Reasons to use procedural content generation

There are several reasons for game developers to implement procedural content generation. Togelius et al. (2011) define the four distinct arguments: *memory consumption*, *prohibitive expense of manually creating game content*, *emergence of completely new types of games*, and *potential to augment human imagination*, which are described with some real life examples as follows:

- memory consumption – content can be kept ”unexpanded” within the seed values[6] of the PCG system, example: Elite[a];
- prohibitive expense of manually creating game content – procedural generation tools provide game designers a way to produce vast amounts of content with a couple of parameters, which is ever more

[6] Seed value (random seed, seed state, or seed) is a value or vector used to initialize pseudo-random number generators and other deterministic algorithms.

valued as games are expected to have more and more highly detailed content, example: SpeedTree[b];

- emergence of completely new types of games – games built around the procedural content generation can provide infinite replay value as the algorithms give player infinite amount of meaningfully different content, example: Rogue[c];
- potential to augment human imagination – a certain amount of sameness can be expected when a human designer creates a lot of content, and usage of offline algorithms can provide results that inspire the human designer and result in a more diverse end product.

2.3 Real life examples

Here are introduced some classic examples of PCG. First the Elite, a space trading game from 1984 which implemented PCG algorithm as a data compression method, then the Rogue, one of the first games with procedural dungeon generation, and finally procedural graphics and an award-winning software SpeedTree, which is used to procedurally generate trees and other vegetation not only in games but other media, such as movies, too.

2.3.1 Elite (1984)

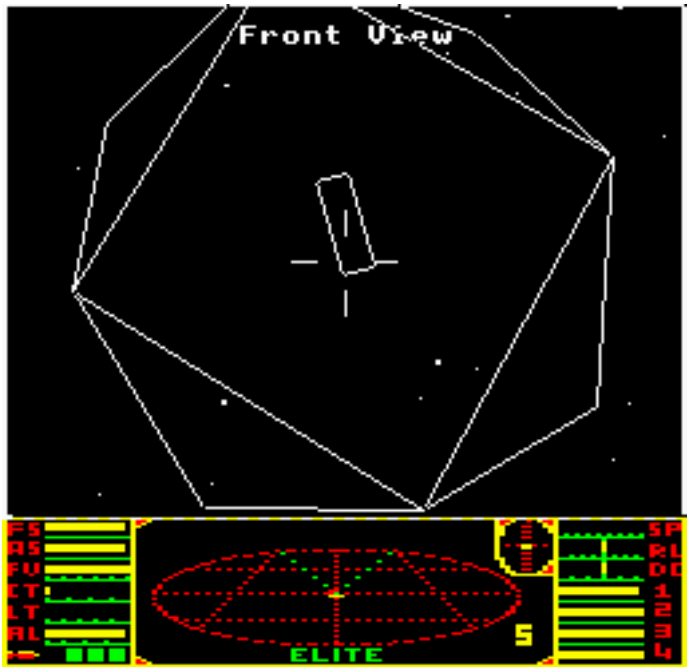

Figure 1 The BBC Micro version of *Elite*, showing the player approaching a Coriolis space station (Picture: ThomasHarte)

Elite (Figure 1) is an early example of completely deterministic PCG that utilises PCG solely to reduce memory consumption. Taking into account the hardware of the BBC Micro (for which the Elite was first developed), it would have been impossible to get all the star system information into the 32 KB memory of the BBC Micro Model B (Wikipedia, 2016). The solution was to develop a deterministic PCG algorithm, and store the entire game universe in a series of seed values (presumably one for each star system). Therefore, in the final game, the developers knew everything about the resulting star systems even though they were procedurally generated because the seed values were static and the generator deterministic. In this context, the purpose of PCG is not to generate unpredictable content, but to compress the game data to be generated during runtime. (Togelius et al., 2015)

2.3.2 Rogue (1980)

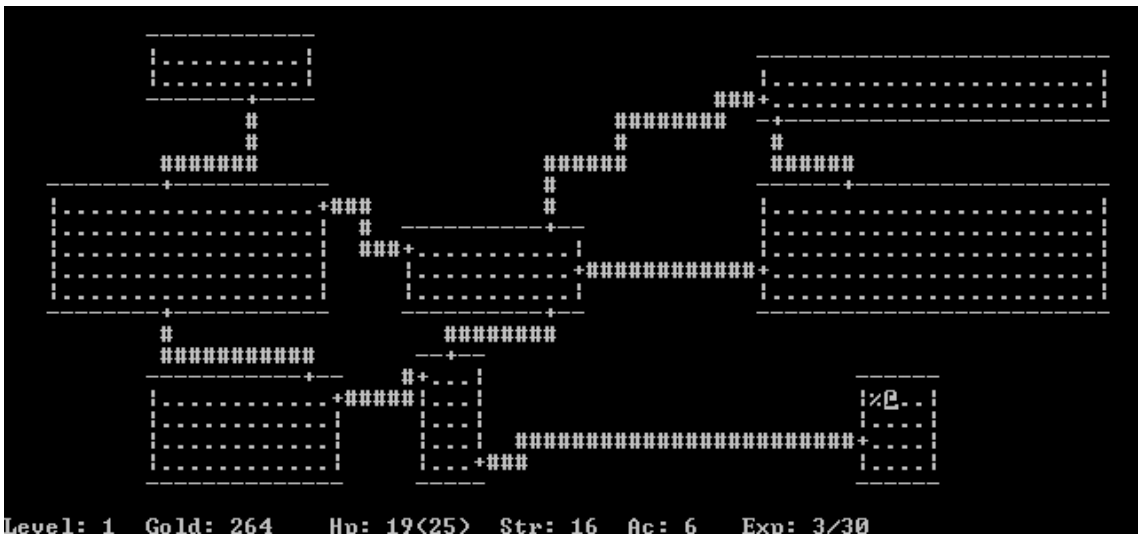

Figure 2 Rogue in an ASCII terminal

Rogue (Figure 2) is one of the earliest games that uses procedural dungeon generation, and it is a prime example of emergence of completely new types of games when a PCG system is implemented. It is one of the most significant games in regards to the traditional procedurally generated game content, so significant in fact, that it spawned its own game genre: Roguelikes. Roguelikes rely highly on the pseudo-random process to generate most of the game content. These games typically take place in some kind of fantasy environment containing random creatures and items while the gameplay consist of killing the creatures, collecting items, and progressing the random generated dungeons. Roguelikes also implement the permanent death (permadeath) into the game, which means when the player dies, the game has to be started over without any progress being saved. While the permadeath was more common in games in the 80's and 90's, roguelikes have kept this feature to this day, mostly because the PCG provides such a different experiences between game sessions that the death is negligible.

2.3.3 Procedural graphics and SpeedTree

Procedural generation is also often used for the graphical content of games and other media, partly to relieve the prohibitive expense of manually creating every unique rock, tree, and flower. The classic example is procedural textures, which are used to generate realistic representation of natural materials (wood, stone, metal, etc.) The procedural process ensures that the textures have always the same characteristics, but random factor makes every texture slightly different (making them look more organic), breaking the possible visible pattern that would be present in hand-drawn tiled textures. SpeedTree is one widely used software that can be used to procedurally generate wide range of vegetation. The SpeedTree generates not only the textures, but also the 3D models and shaders[7] of the trees.

2.4 Procedural content generation dissected

While there are no definitive researches or publications which would offer a basic taxonomy of approaches for PCG, the following distinctions drawn by Togelius et al. (2011) help placing a particular example of PCG between each of these pairs of extremes. *Online versus offline, necessary versus optional content, random seeds versus parameter vectors, stochastic versus deterministic generation*, and *constructive versus generate-and-test.* By defining the system with these pairs the purpose of the system becomes clearer to the developer, the user, and possible other people who examine and study the system.

[7] Shader is a computer program that calculates the rendering of 3D models.

2.4.1 Online versus offline

The first definitive distinction is whether the content generation is processed *online*, during the *runtime*[8] of the game, or *offline* during the development before shipping the game. The process of an online PCG has some strict requirements: it has to be very fast, have predictable runtime and its results have to be of predictable quality. For example, an online dungeon generator creates the content when game level is loaded, while in an offline generator, the algorithm suggests the layout of the dungeon and it is edited and perfected by a human designer.

2.4.2 Necessary versus optional content

The next distinction is the necessity of the generated content regarding the gameplay. The player is required to get through the necessary content in order to complete the game, while the optional content can be avoided. An example for the former could be rooms of the dungeon generator that must be traversed, while the latter could be items that the player might not encounter. It is important to get the generator creating necessary content to output correct and working results, while requirements for optional content are not as strict.

2.4.3 Random seeds versus parameter vectors

Another distinction defining the generation algorithm is the extent of its parameterisation. On one extreme the algorithm might only take one seed value and generate the whole content with it, while on the other extreme the algorithm might take a set of real-valued parameters to specify the properties of the generated content. For example, a PCG algorithm could generate the entire dungeon from

---

[8] The term *runtime* is used in programming to refer to a process that is done when the program is run. The other alternatives are *compile time*, which means the process is done when the program is compiled, and *author time*, which means during the development.

the initial seed value, or take restricting area and a number of rooms and corridors as parameters in addition to the seed.

2.4.4 Stochastic versus deterministic generation

The amount of randomness in content generation depends highly on the purpose of the system. As extreme examples, dungeon-generation algorithm of a Roguelike never produces the same content given the same seed, while completely deterministic system can be used as a form of data compression as seen in Elite.

2.4.5 Constructive versus generate-and-test

The final distinction is between algorithms that can be defined as either *constructive* or *generate-and-test*. Constructive algorithms are those that generate the content once without any following procedures, while generate-and-test algorithms at least make sure the content is correct. The generate-and-test algorithms can implement various evaluation processes to guarantee a working result, and in a case of failure, some or all of the generated content is discarded and regenerated. There are various algorithms developed to ensure the correctness of the content during the generation and to evaluate the outcome. For example, a dungeon generator could perform a test to see if the dungeon can be traversed through by running a pathfinding algorithm through the level. However, these evaluations depend highly on the type of generation done and algorithms used in it. As a more advanced example, genetic algorithms are often used in various PCG systems to learn and evolve the generation process to get the strongest results.

3 Procedural generation techniques

Procedural generation techniques and algorithms depend highly on the generated content. The technical community has established a collection of them that are generally used. Some of the most common techniques are explained here: *pseudo-random number generators (PRNG)*, *gradient noise* and *random points*, as well as a bit more special but widely utilized technique *Lindenmayer systems*.

3.1 Pseudo-random number generators

Random number generation (RNG) is perhaps the single most important technique in all of procedural generation. While it is not always required in PCG, the unpredictability it provides is a great tool for most of the procedural processes. As opposed to true random number generators (TRNG) that often require some sort of physical phenomena for the computer to be able to generate truly random numbers[9], PRNGs are very efficient as they can generate many numbers in a short time. PRNGs are also deterministic, so they generate a sequence of numbers that can be reproduced as long as the starting point is known. These characteristics make PRNGs very suitable for procedural generation as speed is often high priority in computer processes, and the determinism provides a useful tool to be able to foresee the output. (Haahr, 2016)

[9] Random.org uses atmospheric noise recorded with a radio (Haahr & Haahr, 2016) while HotBits numbers are “generated by timing successive pairs of radioactive decays detected by a Geiger-Müller tube interfaced to a computer” (Walker, 2006).

3.2 Noise

One of the most widely used techniques in procedural generation is the utilization of gradient noise. The first gradient noise implementation, called Perlin noise (Figure 3), was developed by Ken Perlin in 1983. Gradient noise is created by generating a lattice of pseudo-random values, which are then interpolated to obtain values between the lattices. (Ebert, Musgrave, Peachey, Perlin, & Worley, 1994; Piiroinen, 2014, pp. 28-31)

Figure 3 Perlin noise (Figure: Maksim)

The gradient noise is originally used in texture generation, as it generates textures without visible grid artifacts (Perlin, 2001). Perlin noise is however used in all kinds of other PCG implementations too. It is widely used, for example, in terrain and map generation by layering different noise outputs together to create landmasses and biomes within those landmasses. One extremely popular procedurally generated game is Minecraft[d], which uses 3D Perlin noise to generate the infinite blocky terrain. (Persson, 2011)

3.3 Lindenmayer system (L-system)

An L-system is a rewriting system that operates on strings of *symbols*. The system is defined by assigning an *alphabet* of symbols, an initial string of symbols, and a set of rewriting rules. The initial string of symbols is also referred to as the *axiom*, whereas the rewriting rules are also called *productions*. The productions specify how a symbol is replaced by a single or a string of symbols at each rewriting step. As symbols stem

> from a finite alphabet, a string of symbols is commonly refer[r]ed to as a *word*. (Eilertsen, 2013)

L-system was first developed by Aristid Lindenmayer in 1968 to model plant growth. It is used widely on generating plants procedurally as it creates fractal structures, which represent accurately the way plants grow in the nature (Figure 4).

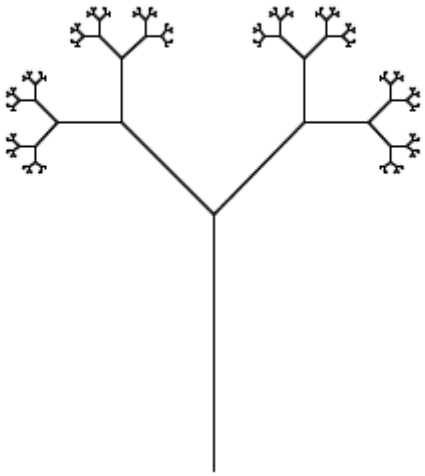

Figure 4 An L-system implementation called Pythagoras tree after seven recursions (Figure: Svick)

While the L-system itself only generates the *words* of symbols, how these *words* are interpreted as different procedural generation systems depend on the context. One popularly adapted technique is to utilize *words* generated by an L-system in city generation when generating the streets of the city (Parish & Müller, 2001). Compared to plant growth simulation, the street generation requires a highly sophisticated evaluation when using L-systems. The Parish and Müller's implementation uses the L-system as a generic template, while implementing an assortment of constraints and parameters that are used to evaluate each production after which the road segments that were accepted during the evaluation are placed into the content.

3.4 Random points

While some kind of noise is often used as the starting point of procedural content generation, generating a set of random *points* in 2D or 3D space with the help of a pseudo-random number generator is equally as common. After a set of points (sometimes within certain boundaries) is generated, these points can then be used in different models, such as *voronoi diagram* and *space colonization* as described here.

3.4.1 Voronoi diagram

Voronoi diagram is a way of partitioning a plane with the use of set of points (called seeds) in that plane. The regions in the diagram are based on the distance of each point in the plane to the closest seed. The diagram is drawn by drawing lines through points that are equal distance away from their closest seed. (Figure 5)

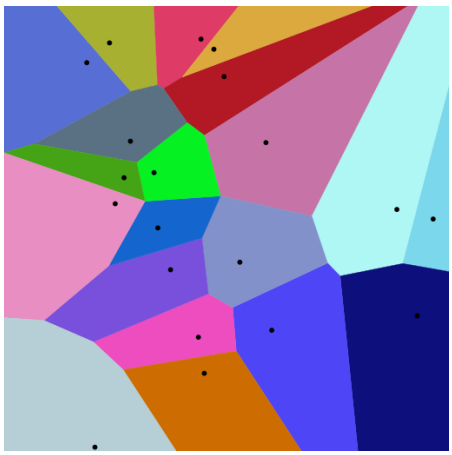

Figure 5 Voronoi diagram (Figure: Balu Ertl)

The result of Voronoi diagram calculation is normally used in map and texture generation, and it results in more organic division than rectangular or circular division would. In map generation, it can be used to represent countries, vegetation, sea, land, or almost anything that presents itself as regions in a map.

Voronoi diagram is also used as the basis for *Worley noise*, noise algorithm developed by Steven Worley in 1996. The Worley noise is meant to complement Perlin noise and “to produce textured surfaces resembling flagstone-like tiled areas, organic crusty skin, crumpled paper, ice, rock, mountain ranges, and craters.” (Worley, 1996)

3.4.2 Space colonization

Space colonization in the context of PCG is a process of filling a space defined by boundaries. One example of this is tree generation: While L-system can be used to generate trees by starting from the root and creating branches by assigning “branch growing” instructions to the L-system’s symbols, space colonization starts by assigning the positions of leaves that serve as attraction points for the branches (Gallant, 2014).

Next is a description of tree generation by Gallant (2014) to help understand how space colonization is implemented in tree generation:

1. Define an area for the crown of the tree.
2. Populate the defined area with attraction points.
3. Create the trunk of the tree, by adding Branches below the defined area. Keep growing branches upwards until the MaxDistance between a Leaf and a Branch is reached, which will result in the initial trunk (Figure 6). MaxDistance is a parameter that defines how far a Leaf can be to attract a Branch. A Branch is not affected by Leaves that are further away than MaxDistance. At this point, branches creating the trunk will be within MaxDistance from some of the lower points, and the branching can be started.
4. Process the Leaves, by comparing it to all the Branches. Calculate the direction and distance from the Leaf to the Branch. If the distance is smaller than MinDistance, we remove the Leaf for it has been reached. If the distance is greater than MaxDistance, we ignore it,

since it is too far. Otherwise, we check if the Branch is the closest Branch to this Leaf. Each Leaf can only affect one Branch at a time.

5. Once the closest Branch is determined, we increment the GrowCount of that Branch, and add the direction of the Leaf to the GrowDirection of the Branch. If multiple Leaves attract a branch, then the GrowDirection will be an average of all of the Leaf directions.

6. Now loop through the Branches, and process any Branch with a GrowCount > 0. Divide the GrowDirection by the GrowCount, to get the average direction, and then create a new branch with this GrowDirection, linking it to the Branch being processed as its parent (Figure 7). Then reset the GrowCount and GrowDirection of the parent Branch.

7. Repeat from step 4 until there are no Leaves left, or no more Branches are growing.

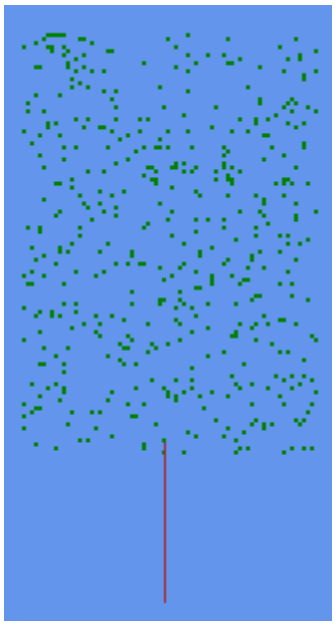

Figure 6 The tree trunk before space colonization branching starts. (Figure: Jon Gallant)

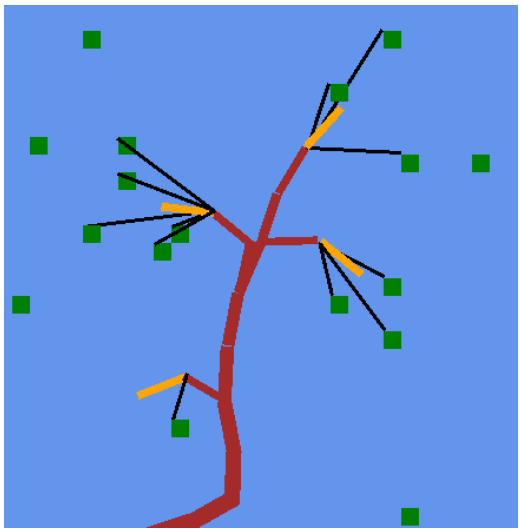

Figure 7 Space colonization growing a tree, orange lines are the branches grown in current iteration (Figure: Jon Gallant)

Space colonization simulates the growth process of the tree, which then results into realistic looking tree, while L-system uses its own approach to reach the same result. When used to generate trees, space colonization can therefore be said to be *teleological* while the usage of L-system is *ontogenetic.*

> – – The teleological approach creates an accurate physical model of the environment and the process that creates the thing generated, and then simply runs the simulation, and the results should emerge as they do in nature.
>
> The ontogenetic approach observes the end results of this process and then attempts to directly reproduce those results by ad hoc algorithms. Ontogenetic approaches are more commonly used in real-time applications such as games. – – (West, 2008)

4 Case: City generation tool for Mental Moustache Ltd.

The idea for this tool started from the thought of possible uses for procedural generation in a game development process with Unity game engine[e]. Game and level designers have often a preference of being in total control of their work, so highly parametric and deterministic procedural generation system was imminent. The content, streets and buildings of a city, came from a game idea of a third person game in a city. During the development of the tool, an idea of other content arose also, so the tool could possibly be used to place content such as trees, for example. Nevertheless, the focus in this thesis will be the implementation of procedural generation methods in the tool.

The development started by implementing a version of procedures introduced in Citygen by George and Hugh (2007). Citygen utilizes the minimal cycle basis (Eberly, 2005), which in this system processes a *graph* of *nodes* and *edges* into *primitives*. This graph defines the areas in which the procedural generation takes place. By defining the area precisely, the user can tell explicitly where the procedural content will take place. After the user has defined the nodes and edges for the graph, the minimal cycles (defined below) can be found and primitives are defined. These primitives will then be the units in which the procedural roads are generated.

4.1 EdgeGraph system

The graph is the combining structure that contains the initial node and edge lists as well as runs the processing methods. While a more compact way to represent the graph would be a matrix with the elements representing the edges, having an object-oriented structure for the nodes and edges enables them to have different individual properties, such as edge width. The data defined by the user is kept in its original form inside the graph, and copies of the user defined nodes and edges

are made when processing them in order to preserve the initial state as it makes regeneration easier.

The edges does not hold any position related data in them. They have a unique ID, references to each node's ID they are connected to (node1 and node2), and the width of the edge. The width is used in the processing step of offsetting primitives inward.

The nodes are the structure in which the real data is stored. Every node has a unique ID, position, and a list of adjacent node IDs.

The primitives have their own copies of nodes and edges created during the processing procedure in the graph system. The copying keeps the relationships between nodes and edges intact while assigning new unique IDs for all entities. The system utilizes the methods introduced by Eberly (2005), implemented for this tool in C# using the nodes-and-edges structure. In short, the minimal cycle extraction starts with the left-most node and goes through the adjacent edges counter-clockwise, ending either in the starting node and creating a primitive (Figure 8), or ending in a node with no next edge resulting in a filament. The filaments are discarded in this implementation as of now, and only the primitives are kept. The processing of filaments and therefore the possibility of creating dead-end roads could be added in further development.

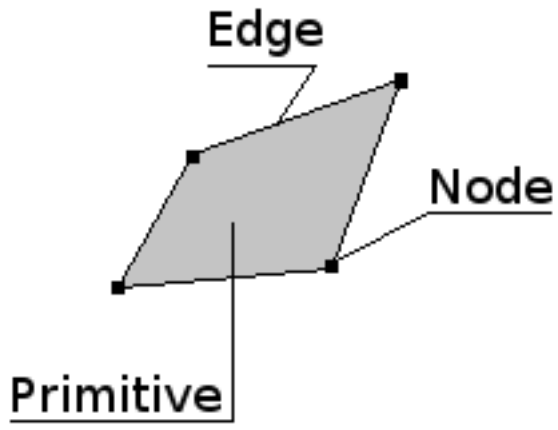

Figure 8 Primitives are formed by traversing around the graph counter-clockwise, starting from the left-most node.

The minimal cycle procedure produces a list of primitives that are then processed. The processing procedure is as follows:

1. Make copies of the given nodes and edges to ensure breaking the link to the nodes and edges created by the graph
2. Offset the primitive polygon inwards by shifting each node by average of the width of adjacent edges
3. Cut angles sharper than 45 degrees by creating a 1 unit long edge between the edges that create the acute angle
4. Combine nodes that are too close (half a unit) to each other after the angle cutting
5. Calculate bounding box for the primitive
6. Sort nodes to be in counter-clockwise order
7. Run evaluation method which checks if there are angles sharper than 5 degrees (changing the angle doesn't provide much difference in the result, so it is hard coded) and mark the global evaluation result accordingly

Steps 3 and 4 are only run if *makeNice* parameter is set to true when calling the processing method as the acute edge cutting might not be wanted, for example in a case when the node count of resulting primitive has to be the same after the processing.

Primitive offset

Each node in the primitive is offset inward depending on the each edge adjacent to the node. The width given to each edge is the gap that is made between the offset edges, so in the node offsetting algorithm the widths are halved. The algorithm starts by finding the adjacent edges of the current node, and calculates their inward normal. Lines are then defined going through the point determined by the inward normal and halved width. The lines go along the new offset edges with twice the length of the original edge, to ensure intersection points. The intersection of these two offset edges is the new position of the current node (Figure 9). The original graph with several possible primitives is split into each individual graph with their own primitive and nodes, so that this process can be done to every primitive without them interfering with each other.

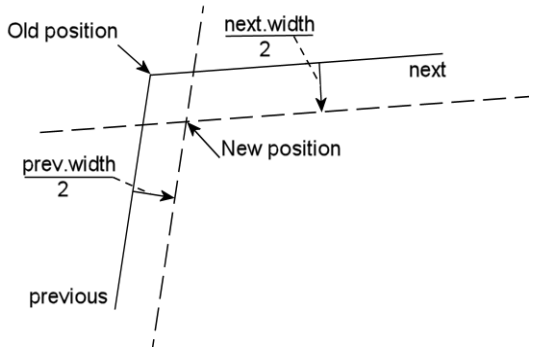

Figure 9 Node offsetting logic

4.2 Sub edge generation

The procedural part of the tool is the ability to generate edges within the primitives. In this system, these are called *sub edges*. The intent in the beginning was to follow the footsteps of other procedural city generators, mainly the George and

Hugh's (2007) CityGraph and the Parish and Müller's (2001) CityEngine. These systems utilize L-Systems for the road generation, and they have implemented highly sophisticated generation and evaluation systems for the L-systems to work in a predictable way.

4.2.1 Usage of space colonization

To utilize L-system in a city generator high amount of evaluation is required for the system to create acceptable results. Implementing the evaluation procedures takes time, as it has to be tested and tweaked to reach good results. For this reason, the space colonization was chosen for this tool instead. Space colonization provides a simpler way to generate the streets inside given boundaries with less evaluation than the L-system. The EdgeGraph tool provides following parameters for space colonization:

- number of (attraction) points to generate,
- "margin" value to determine how close to the primitive edges the points are allowed to be,
- the starting point of the generation,
- minimum and maximum distances of the target points from road growth,
- segment length traversed on each iteration, and
- minimum angle, which determines sharpest angle that the road builder is allowed to make (if sharper, the builder will make a 90 degree turn).

EdgeGraph's algorithm uses the bounding box of the primitive to determine the random values on x and z-axes. While the bounding box is always quadrilateral, the random points have to be tested if they are inside the primitive polygon. The test is done by using the "even-odd rule" (Wikipedia, 2016). The point is saved if it is inside the polygon as well as more than a "margin" distance from the closest edge. After generating desired amount of points, the growing algorithm is started.

4.2.2 EdgeBuilder

The growing algorithm, in this tool called EdgeBuilder, differs in some points from the Gallant's (2014) tree generation algorithm, as the purpose of EdgeBuilder is not to generate tree branches. The goal was to utilize space colonization to create simple branching roads without excessive amount of intersections. Following are the procedures the EdgeBuilder uses in the sub edge generation.

Average distance vs closest attraction point

Because the three-branch generation results in high amount of small branches, the algorithm had to be altered to result in simpler "branches". The EdgeBuilder does not calculate the averages, but takes only the closest attraction point towards which it advances until at MinDistance from the point.

Ensuring all targets are visited

In case all the non-visited targets are over MaxDistance away, the EdgeBuilder traverses back the generated edges until a non-visited target is below MaxDistance. If all non-visited targets are over MaxDistance, it picks the closest visited point to the non-visited target and continues generation there.

Preventing nodes inside straight edges

During determining the new edges, the advancing algorithm checks with vector dot product if it is continuing on the same direction as in the previous step. By moving the previous node to the current position, the result does not have nodes inside straight edges.

End point connection

At this point, the resulting structure is a sprawling line from target to target, branching at a couple of places. To have the edge generation end up with closed

spaces that are later processed for primitives, an additional end point connection procedure is run after the EdgeBuilder has stopped. The connection algorithm has two different processes for each end node (nodes with only one adjacent edge): *connect to a node close by* or *run a line cast and connect to edge that was hit*. The choice between these two is made with the use of "*subnode end connection range*" parameter, which determines the distance in which a node has to be from the ending node, in addition to a rough direction (a 30 degree segment), in order to be connected to the ending node with a new edge (Figure 10). The line cast is done if no node was found (or if the range is set to zero) and a new node is created on the intersection point of the line cast and the edge that was hit. Finally, a new edge is created between the ending node and this newly created node.

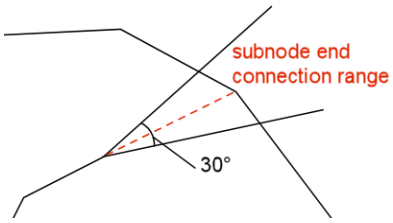

Figure 10 *Subnode end connection range* and the rough direction illustrated

Node combining

In addition to the processes above, there is also a node combining procedure that was added in order to make simpler sub edges. If the "*subedge combine range*" parameter is above zero, the nodes that are closer to each other than this range are combined into one node, which is the average of the predecessor nodes. The combining algorithm is run separately for the EdgeBuilder result and the final result after the end connection algorithm.

4.3 Unity Editor

In this chapter, the different ways of implementing custom editor behaviour in Unity will be introduced. The three areas of Unity editor that the user can make custom behaviour for are *Inspector*, *Editor Window*, and *Scene View* (Figure 11). The *Inspector* is a specific editor window that shows information about each component of selected objects. The custom editor behaviour in inspector happens through implementing a script that describes inspector behaviour of a component. Custom *Editor Windows* are windows inside the Unity editor similar to the Unity's own windows for different parts of the editor, such as animator or game view. Editor Windows are implemented for more general-purpose editors, while through Inspector each component can have their own editor tools visible for the user. The last one is *Scene View*, in which the user can manipulate the objects in the game scene. Inside the Scene View, developers can implement their own handles and GUIs (graphical user interfaces) for the object manipulation.

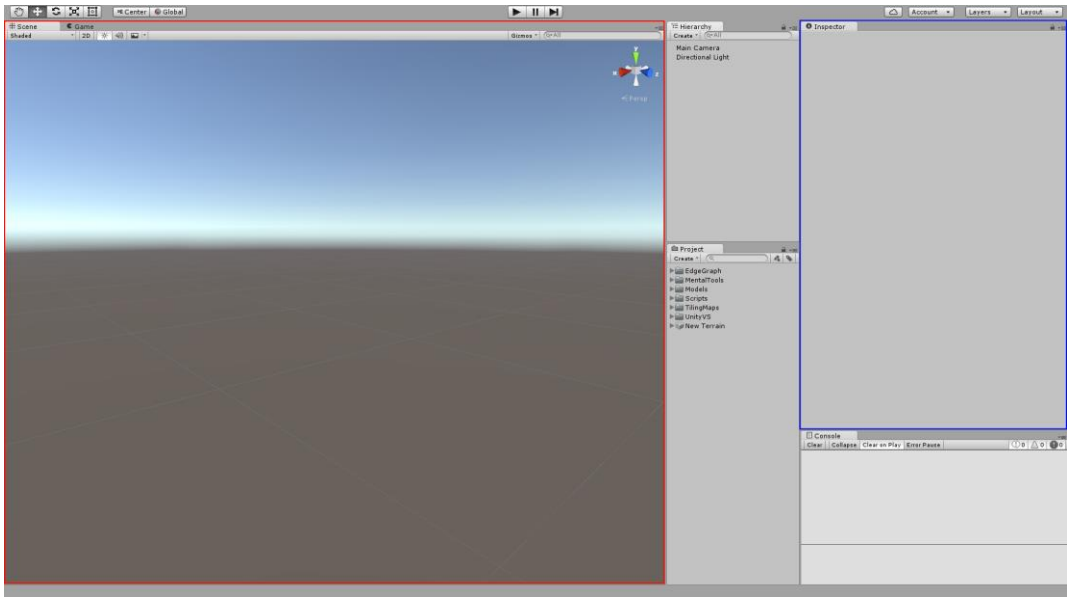

Figure 11 Unity editor showing Scene View (red) and Inspector (blue).

4.3.1 Inspector

In Unity's editor, Inspector is the window that shows all the information about selected object(s). Unity utilizes a component system for all objects in the game scenes, in which a game object contains a Transform component as well as all the different components for physics simulation, gameplay behaviour, audio, and graphics. Every class that is inherited from MonoBehaviour[10] can be added to a game object, and have its own view in the Inspector among all the other components of the game object. The view in the Inspector contains all serialized fields of the behaviour by default, but the view can be altered with by creating a custom editor for the behaviour.

4.3.2 Scene view

The scene view is the window inside the Unity editor that is used to inspect the game scene. The scene camera is separate from the camera that the game uses, and it can be moved during the editing. On the scene view, the custom editors of behaviours can implement their own controls and GUIs that the user can use to manipulate the data in the behaviour.

The different 3D controls are called Handles. There is a number of functions in Unity for shapes (2D and 3D) and lines that can act as handles, and they can be used to provide the user a way to interact with the data. Functions for the traditional 3D manipulation handles such as position, rotation and scaling handles are also provided, and using them can be beneficial as the user will be familiar with them already. (Unity Technologies, 2016a)

[10] MonoBehaviour is the name of the class that implements the component pattern in Unity.

Scene view can also contain GUIs that are “floating” on the scene view window. These UI windows provide a way for the user to concentrate on the scene view only, and still modify values of the objects manipulated.

4.3.3 Editor Window

Every individual window inside the Unity editor is an editor window, and they can be created by the user. Editor Windows are usually more general editors, used for editing asset files or sub-systems of the game.

4.4 Editor for EdgeGraph

In this chapter, the usage of the above-mentioned areas in the EdgeGraph editor will be described. The Inspector is used for most of the data and parameter handling of the system with the generation process calls, while the handles in the Scene View provide an easy method for the user to manipulate the nodes and edges in the game world.

4.4.1 Inspector

In the EdgeGraph’s Inspector, the user can manipulate the node and edge data, control the generation process and manipulate the parameters. The node and edge data modification fields can be hidden, as they are not used in the typical workflow of the tool, but are still a valuable information for the user, as the scene view does not show any coordinate values for the nodes or edges. The sub edge generation parameters and controls are hidden if there are no primitives in the graph (Figure 12).

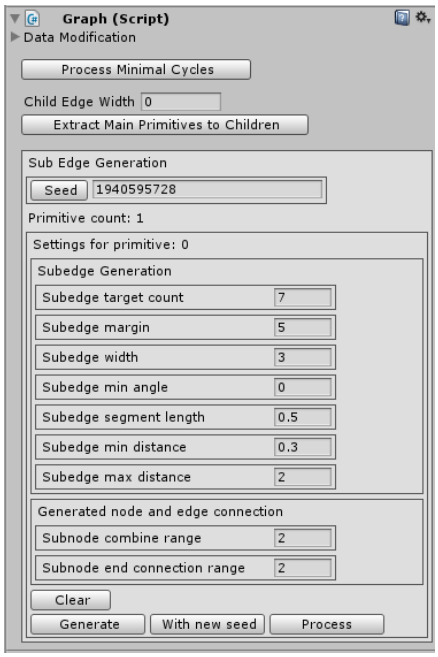

Figure 12 Graph inspector user interface

The sub edge generation parameters in the inspector are as follows:

- *Target count* is the amount of points generated inside the primitive.
- *Margin* is the minimum distance that the generated points are allowed to be from the primitive edges.
- *Width* is the edge width that is set to every sub edge.
- *Min Angle* is the minimum angle the edge builder can turn on one iteration. Smaller angles result in a 90-degree turn.
- *Segment Length* is the distance the edge builder advances in each iteration.
- *Min Distance* is the minimum distance from current position on edge builder to generated points at which the point is considered visited.
- *Max Distance* is the maximum distance at which the edge builder considers the generated points towards which to advance. Closest point is always chosen among ones at less than max distance.

- *Sub node combine range* is the range within which sub nodes are combined. The generation results are better when the range is more than zero as the resulting nodes are not very close to each other.
- *Sub node end connection range* is the range within which ending nodes (nodes with one adjacent node) are connected to other nodes in order to ensure full primitives in the result.

4.4.2 Scene View and sub edge generation

The EdgeGraph editor utilizes cubes and lines for presenting the nodes, edges, and primitives for the user. The node handles in the tool are small cubes. The use of a cube instead of the 3D position handle is because the data in the graph is in two dimensions, so the arrow for Y-axis would not be used. In addition, the scene view would be too cluttered if there were three arrows for each node position.

The position modification of nodes is made by moving the nodes in the XZ – plane[11], so when the user drags a node's cube handle, only the node position's X and Z values change. The existing nodes can be removed by holding shift key and selecting a node. New nodes can be added in two ways: adding a node to cursor position and adding a node by splitting an edge. First is done by holding control key, and the second is done by holding both shift and control keys. When the new node is being added to an existing edge, an indicator (blue dot) is drawn on the closest point on the closest edge to the cursor, where the new node will be created (Figure 13). The node adding on the edge was added during the development because the need to be able to split current areas with edges came up.

[11] Unity uses left-handed coordinate system with Z-axis pointing forward and Y-axis pointing upward.

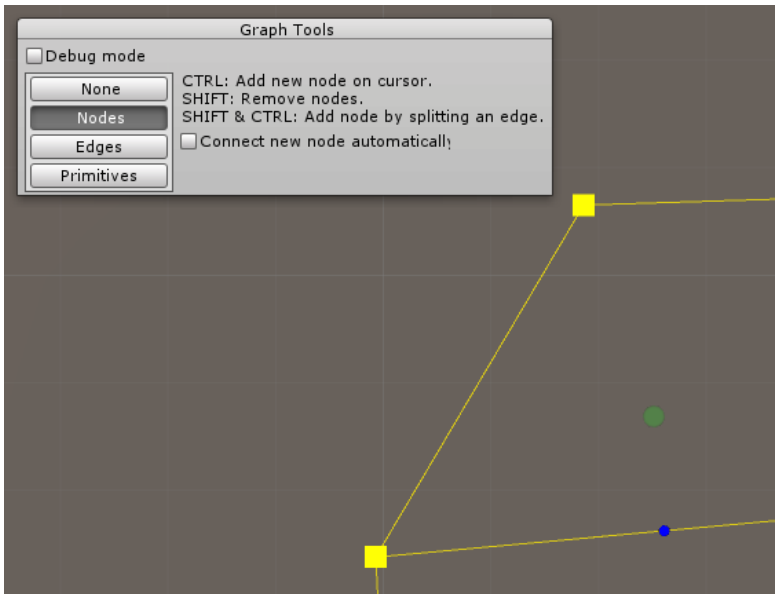

Figure 13 Node adding by splitting an edge

The handle functions used to indicate the edges are lines, and when in edge editing mode the cubes indicating nodes cannot be moved. The line handles cannot be interacted in any way so they are only meant to inform the user. The node cubes are used when new edges are created. The user starts by clicking a node they want the edge to start from and simultaneously pressing control key, and drag towards other nodes. The tool will draw a differently coloured line to the closest node from the cursor, indicating where the new edge will be created if the user lets go of the mouse button (Figure 14). If the closest node is the node where the user started, no line will be drawn and this way the user can cancel the new edge adding. In order to remove existing edges, when shift key is pressed cubes are drawn in the middle of the edges and by clicking these the edges are removed.

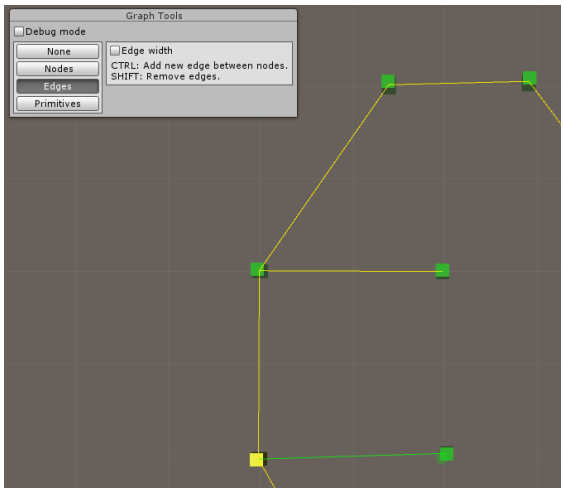

Figure 14 Edge editing tools

Edge widths can be edited in the edge editing mode by enabling a toggle. When in width editing mode, the user can use a brush-like tool to help change widths of several edges easily, or change width of every edge to the set value. (Figure 15)

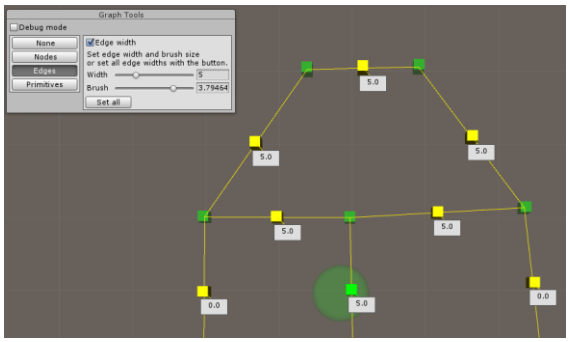

Figure 15 Edge tools with width editing

When the user has processed the minimal cycles by pressing the button in the inspector (Figure 12), the primitive mode is enabled on the scene view. In the primitive mode, the user can select one or more primitives that were found on the graph by holding control key and clicking inside the primitives (Figure 16) and change the generation parameters for the selected primitives. If none is selected, the settings are set to every primitive. If the user holds shift key in the primitive mode, the root node selection tool is enabled (Figure 17). The tool will indicate the closest node of the selected primitive from the cursor, and by clicking the user

sets the node. The root node is the node from which the sub edge generation starts.

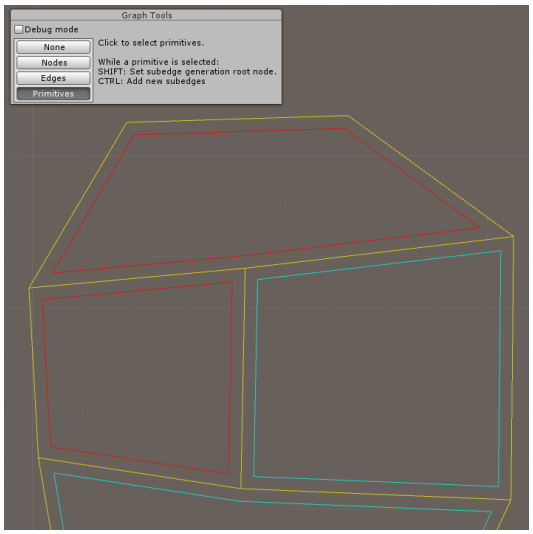

Figure 16 Primitive selection

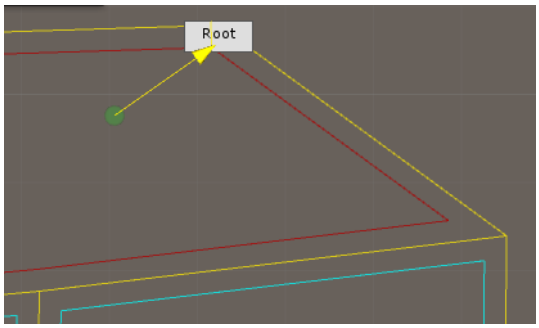

Figure 17 Root node selection

After setting the parameters, the sub edges can be generated either by using the current seed visible in the UI or by generating or inputting a new random seed before the generation (Figure 12). When the sub edges are generated, the primitive editing mode will show the generated sub edges by drawing a line handles for the sub edges, and cube handles for the nodes that the sub edges go through. The user can then edit the generation results, so the user is in charge of the generation before and after the generation process. (Figure 18)

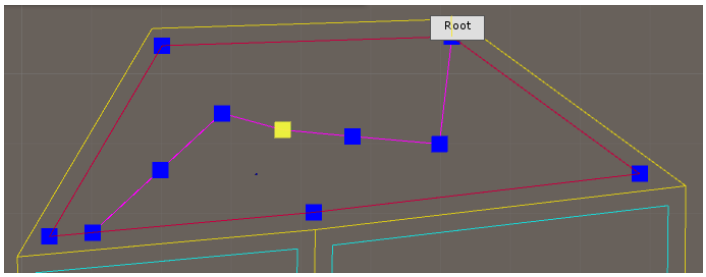

Figure 18 Sub edge editing

5 Discussion

When developing the EdgeGraph tool incrementally, in the beginning some parts of the system were better planned than others. During the development cycle, the function and utility of parts of the implementation became clearer, and some parts ended up working well, while others could have been made better. In this chapter, both possible improvements, and problems solved during the initial development of the EdgeGraph tool, are discussed. This is to provide a starting ground for further development as well as the reasons behind current solutions.

5.1 Defining the EdgeGraph tool

In this thesis the definition and dissection of a PCG system was introduced, and in this chapter, the EdgeGraph tool will be inspected within these terms. The first terms were the reasons why one might implement a PCG system, and the second terms were the dissecting pairs. The example helps open up the behaviour of the tool as well as provides an example on how to define and dissect any PCG system.

Four reasons to utilize PCG were introduced in this thesis: *memory consumption*, *prohibitive expense of manually creating game content*, *emergence of completely new types of games*, and *potential to augment human imagination*. The EdgeGraph tool falls mostly into the category where the tool is meant to **decrease the amount of manually created game content**. Secondary reason was the **potential to augment human imagination**. These reasons manifest in the fact that the purpose behind the tool is to help designers and artist to create the gameplay spaces for games by generate city with buildings and vegetation. While the system reduces the amount of manual work by the designers, the workload is still large. The tool **augments the human imagination** as the amount of content increases.

The dissection pairs of extremes introduced in this thesis are: *online versus offline*, *necessary versus optional content*, *random seeds versus parameter vectors*, *stochastic versus deterministic generation*, and *constructive versus generate-and-test*. The EdgeGraph tool works purposefully **offline**, as its main function is to help in content creation during game development, not generate the cities during runtime. While the content is necessary in the way that the city streets are where most of the gameplay happens, the designer input negates some of the high requirements for traditional necessary content. However, because the designer's workflow benefits from well working generation as it reduces the amount of fixing by hand and regeneration, the content the tool generates can be defined as **necessary**.

The parameters of the systems in EdgeGraph tool are discussed below in chapter 5.4.2. The tools is meant to be **highly parameterised** in order to comply with the designer-centric generation process. The work of the designer starts by defining the nodes and edges that form the primitives. The primitives are the first parameter vectors given to the system, and additional values, including a seed value for the PRNG, that dictate the behaviour of the sub edge generation can be modified in the editor. The EdgeBuilder is **entirely deterministic**, as the same seed value with the same root node generate the same results every time. By changing the root node only, the attraction points for the space colonization are the same but the generation starts from different node.

The EdgeBuilder is still a **constructive** process, as it has very little testing and evaluating. For it to be able to perform well during extended use, evaluation to guarantee working results without human input after the generation is desirable. This puts the current implementation to the constructive end of the pair, while the intent for the finished tool would be closer to the generate-and-test end. It can be deduced from this dissection that this is one of the biggest developing focuses for the future of the tool.

5.2 Node position and manipulation

The EdgeGraph tool was created from the beginning to be a very user-input heavy PCG system, and the interface to manipulate the nodes and edges as the parameters for the generation are very important for the usability of the tool. In this chapter, the implementation of the tool and its future development goals are discussed from the perspective of the node manipulation.

The final data structure of the graph was the result of some iteration. The relationship between the graph, edges, and nodes was chosen because in its current form the position of each node is only saved in one place. In the first iterations, the edges held a reference to each node, but it became difficult to ensure the relationship between edges and nodes because of Unity's internal serialization. The difficulty rises from the fact that Unity's serialization treats custom classes as structs (Unity Technologies, 2016b) so in order to keep the real references to node objects in edge objects "re-referencing" algorithms had to be run to refresh the objects references according to the node IDs after each time Unity serializes its data. In the current iteration, edges have references for node IDs and a list of nodes is given as a parameter when retrieving the node object.

The node offsetting algorithm in the primitive class could be improved to provide results that are more reliable. The implementation in its current form in the EdgeGraph tool is a simple one, and the results are sometimes faulty. Especially in very narrow primitives, the offsetting can result in the edges crossing each other or even going outside the original primitive. There is some evaluation to check the crossing of edges in place, but it simply rejects the primitive altogether. To reach a better evaluation and therefore better functioning generation, the offsetting could be more robust. One solution could be to utilize a polygon offsetting library, such as Clipper[f].

The nodes in the EdgeGraph are positioned in a XZ–plane to ensure that the graph is in two dimensions, mainly because the Eberly's (2005) minimal cycle

algorithms work in two dimensions. First, the plane could be made to work in its own planar space, freeing it from any world axes. This would require some additional computation in the minimal cycle finding, but it would make the tool much more versatile. Second, the graph could work in three dimensions. Three-dimensional graph would have some problematic cases for minimal cycle finding when edges branch in a way where only the third dimension differs. When releasing the graph from the plane, the generation could follow terrain topography and create the roads in uphill and downhill.

5.3 Increasing usability

The user interface, user experience, and further usability of the tool could be greatly increased, as it was smaller priority during the initial development. This chapter will discuss the ways of increasing the usability through UI and the data system.

User interface

The user interface for the tool could be implemented in a variety of ways that the Unity editor offers. In the current state of the tool, the UI is implemented through the inspector GUI, and the handles and a GUI window inside the scene view. To improve the usability and workflow of the tool further, some of the editor components inside the inspector could be moved to the scene view GUI. The placement of the sliders and numeric fields is trivial, as the same methods that draw them can be called in either GUI.

Data saving

The goal for the EdgeGraph tool was to create gameplay spaces in a game placed in a city environment. This resulted in the utilization of the Inspector, and the EdgeGraph being a component of a game object. Implemented this way, each component generating the primitives is tied to the object, and can be saved as a

prefab[12] if wanted. The generated data is only serialized in these objects and prefabs, which means it cannot be used elsewhere outside the Unity editor.

If the data was intended to be saved in a file and used in a completely separate software, for example, it would have to be created through a different interface. While the data could be saved in a file from the current interface, it would not be intuitive, as the Inspector is meant to modify the properties of objects in the scene. There are at least two different ways for the interface to be implemented in Unity for the purpose of handling data serialized in a separate asset file: editor window inside the Unity editor, or interface created to be used during runtime in a built executable.

The editor window inside Unity editor is better suited to edit different asset files than the inspector is. The implementation through editor window would still have to use the scene view for the editing to be intuitive and easy, but the data could be held in a different asset file than a prefab of an object. Unity has a structure created for this, ScriptableObject[13], which can be derived from to create asset files to be used with Unity. Through this interface, the generated EdgeGraph data could be saved into an asset file, and later used for different scenes inside the game. While this implementation would provide a more generic place for all the EdgeGraph data, the content placed in the scenes through this generation system is still saved in the scene itself (buildings, streets, trees, etc.), which makes the primitive data less relevant for the end content. In addition, the data inside Unity asset files is not very accessible, as it is only accessed through editor views implemented inside Unity.

---

[12] A prefab is an asset created to preserve a GameObject in Unity.

[13] ScriptableObject is a class in Unity that you can derive from if you want to create objects that do not need to be attached to game objects. They are most useful for assets that are only meant to store data.

A better alternative for an interface to save the EdgeGraph data to files would be to save the data into a binary file, or even in clear text to a text file. This could be done from a ScriptableObject through the Unity editor by converting the data in the asset to a different file type, but why stop there? The whole generation tool could be implemented using the in-game UI system of the Unity. This way the tool could be built and run as a separate procedural content generation software. While this method does not fit for the purpose of the EdgeGraph tool, it is a great way of implementing a more generic generation tool for different types of content. The downside of this approach is the work needed, as many of the features in the Unity's editor tools would have to be replicated in the in-game UI system.

5.4 EdgeBuilder

The current procedural component of the EdgeGraph tool is the EdgeBuilder. It is an implementation of space colonization technique, and it generates sub edges inside the primitives that are constructed from nodes and edges. The utilization of all these nodes and edges (created by the user or generated through the builder) is what makes the tool useful for the game content creation. In this chapter, the implementation of the EdgeBuilder will be discussed, and a couple of prototypes that utilize the data created will be introduced.

5.4.1 Space colonization vs L-system

The work on the EdgeGraph tool started by implementing a generic context sensitive L-System with Unity editor toolset so the system with different axioms and rules could be tested. A context sensitive L-system takes into account the possibly given *left* and *right context* words when determining the successor for a symbol in the next production. The system takes all rules of which predecessor and contexts match and then uses given weight values to pick one at random. Eventually it was decided not to use the L-system, so it is not used in the tool at all. (Eilertsen, 2013)

The reason the EdgeGraph tool did not end up using the L-system was the complicated evaluation tied with using the system to generate roads. The Parish and Müller's (2001) sophisticated "self-sensitive" usage of L-system in their CityEngine is understandable as the system is intended to create all the streets and roads of a whole city with the only user input before the generation. Even though the CityEngine uses coastlines and parks as limiting factors on where the roads can be generated, the generation system has to fill very large areas with realistic road network without any user interaction during the process. The George and Hugh's (2007) CityGen has similar secondary road generation (while the primary roads are defined by user) and has similarly complicated generation method as it generates a large set of data (Figure 19) with L-system and then implements different snapping and testing algorithms. A simpler approach was chosen to satisfy the requirements of this tool by utilizing space colonization.

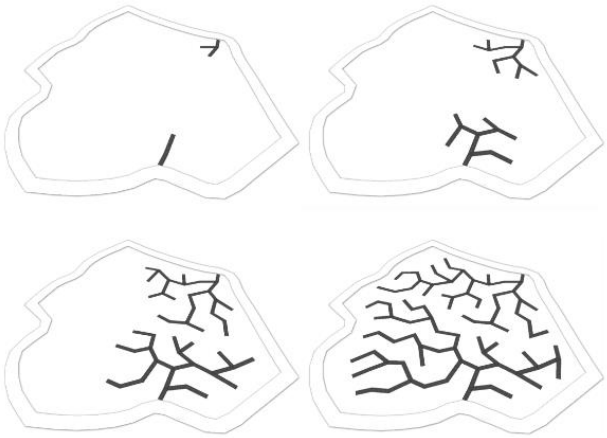

Figure 19 CityGen road Growth 10, 100, 300, & 1000 steps. **(George & Hugh, 2007)**

The purpose of this tool is to generate cities as gameplay spaces for first and third person games, so the realistic generation of a massive raster of streets and roads into the city was not necessary, as the gameplay space would be too complex and repetitive. This resulted in the use of space colonization to generate the secondary roads, as it could be used to generate simple forking roads inside the primitives with less evaluation than the L-system would have required.

5.4.2 Parameters

The EdgeGraph tool provides following parameters for the EdgeBuilder:

- number of (attraction) points to generate,
- “margin” value to determine how close to the primitive edges the points are allowed to be,
- the starting point of the generation,
- minimum and maximum distances of the target points from road growth,
- segment length traversed on each iteration, and
- minimum angle, which determines sharpest angle that the road builder is allowed to make (if sharper, the builder will make a 90 degree turn).

This set of parameters emerged naturally from the different needs to be able to control the generation output. Some parameters affect the result more than others do, and some have very narrow ranges in which they work. The input for the parameters is still a number field, which could be replaced with a slider, for instance.

The segment length might be the most problematic parameter as it is very important for the tool to be able to work in different scales, but by setting too long a segment the builder can overshoot the primitive boundaries easily. The builder operates inside the given primitive, so each segment should be a fraction of a side from the dimensions of the primitive while the tool can operate with primitives of one unit or ten units wide.

The minimum angle is a parameter that came about when iterating the solution for shard edges inside the primitives. The result of this parameter is not very visible because with small segment length, the individual turns are not very distinctive, and the node combining process removes some cases where points very close to each other result in these sharp edges. To improve the intuitiveness of

the parameters, this one could be hard coded like the corner cutting, and the user could be given a choice to use it or not.

5.4.3 Dividing the primitives

Division of generated areas between the roads is an integral part in city generators. Both CityGen and CityEngine use division algorithms to divide areas into lots, in which the buildings are generated. EdgeGraph could benefit from subdivision implementation in order to have smaller primitives inside the currently generated ones.

Primitive division would be overall the next step in the development of EdgeGraph tool. The subdivision results could be used to create small streets or alleyways, or only as the perimeters for generated buildings. In the former case, the results could possibly be too repetitive, but the latter case provides a way to eliminate this repetition and use the subdivision data only for the visual results. In the vein of user-dictated generation of the EdgeGraph, the user could decide when the subdivisions would results into alleyways.

5.4.4 Building the city

During the end of the tool's development process for this thesis, a couple of prototypes to utilize this data were created in order to test the final city generation. While the node, edge, and primitive structure of EdgeGraph tool is a versatile data system useful in defining areas, the concentration in development was city generation. For that reason, the usability of the tool for city generation was prototyped. Two different kinds of city structures were used as a reference for these prototype generators: suburbs with separate houses, and old Central European cities with a constant façades going around the whole perimeter of the block (Figure 20).

Figure 20 3D models of buildings in Bruges, Belgium, taken with Google Earth

The suburb-style building placement can be as simple as placing certain amount of buildings and checking on every placement that it does not overlap with already placed buildings. The overlap check can be made two-dimensional by defining a footprint for each building, and checking if it overlaps with other footprints or goes over the primitive's edges. This was an easy and effective way to populate the primitives to what looked like suburbs filled with buildings, especially when the footprints were aligned with the closest edge.

The façade implementation follows closely the building generation method of CityEngine and CityGen. Both systems create lots by dividing the initial blocks and then generate buildings to these lots with their own building generation systems. A building mesh generator can be a highly complex system in its own right, and one was not considered to be in the scope of this tool. A simpler façade building process was created, that placed three different sized façades (small, medium, and large) along the edges of the primitive, and generated a mesh for roof. A fourth, scalable façade was also present, and it would fill the smaller-than-small gap that was left in the end of the edge. Texturing this scalable wall to look realistic was one problem with this approach.

In the end, a more complex building generation system might be necessary to create immersive gameplay spaces with this tool, especially for first-person games where the player camera is closer to the building textures. The prototyping resulted, however, in a better picture of what the tool's future requirements are. As mentioned in the chapter 5.4.3, dividing the primitives to create lots would be the first future development target in this regard.

5.5 Comparison to other implementations on Unity

Several content generation software have plug-ins for Unity (Side Effects Software, 2016a; Esri, 2016a), but they are not very comparable to EdgeGraph tool, as it is meant to be used inside the Unity game engine. The external tools also cost more, the Houdini engine costing hundreds of dollars annually, while the pricing of Esri's engine is not available in their website (Side Effects Software, 2016b; Esri, 2016b). Therefore, the external tools should be utilized only if the PCG system is required to be very extensive and the need for an external generation engine is required. The choice between commercial software and Unity engine extensions should be made based on the cost and breadth of the use intention. For bigger companies the cost of external tools might be negligible and therefore provide the fastest way in getting functional and diverse procedural content in the game. On the other hand, smaller game companies implementing the tools by themselves or spending a couple of dozens of dollars in a PCG-system for Unity might be a more attractive solution.

In this section, two city generation systems that run entirely inside Unity are compared to EdgeGraph. The first is Horizon: City Generator, which was chosen because it aims to designer-centric generation that was one of the main goals of the

EdgeGraph tool. The second is commercially available plug-in[14] for Unity, City-Scaper, which was chosen because of the way it implements Unity's Editor similarly to the EdgeGraph tool.

5.5.1 Horizon: City Generator

Horizon: City Generator (Horizon) is a proof-of-concept city generator made with Unity game engine (Thompson, 2015). The system works during run-time of the software as was discussed in chapter 5.3. The Horizon generator has a different approach to implementing city generator with Unity, as it uses Unity merely as the platform on which to implement a standalone[15] executable. The cities generated with Horizon can be exported as .obj files to further edit in a 3D modelling software.

Definition

Using the dissection defined by Togelius et al (2011) for the Horizon: City Generator, the Horizon's generation is an offline process that generates optional content (*online vs offline*). While the graphics of the city might have a central focus on the game, the visual fidelity does not affect the ability for the player to complete the game (*necessary vs optional*). Like any such city generator, the Horizon is intended for offline use of the designers with many options to affect the results before and after the generation. Therefore, a number of parameters are provided in the editor, making the Horizon highly parameterised system (*random seeds vs parameter vectors*). The whole road and building generation system is also very self-evaluating as it has the L-System implementation (*constructive vs generate-*

[14] Plug-in is a piece of software that adds specific features to an existing software.

[15] Standalone software is a computer software that runs as a separate process, instead of running inside other software, an editor for example.

and-test) (Thompson, 2015, pp. 12-24). As for the randomness, the Horizon implements circle noise and L-systems that are generally deterministic (*stochastic vs deterministic*).

5.5.2 CityScaper

CityScaper is a Unity editor plug-in developed by TikiHubGames (TikiHubGames, 2016). CityScaper is a commercially available system for Unity editor (Unity Asset Store, 2016), that has its intended use closely similar to the EdgeGraph tool. Both systems are implemented using the editor tools for Unity, intended to integrate into the Unity workflow instead of being an external generator.

Definition

Similar to other city generators, CityScaper is an *offline* system with a wide range of *parameters* intended for used inside the Unity editor during development. As the original intention of the tool was to generate background graphics (TikiHubGames, 2016), the tool generates very *optional* content. The amount of evaluation (*generate-and-test*) is difficult to determine for the CityScaper, but it can be concluded that it implements a fair amount of evaluation to make the neat street grids. The system utilises noise algorithms similarly to the Horizon, which points to it being a *deterministic* system.

5.5.3 Comparison

All three systems compared here, **Horizon**, **CityScaper**, and **EdgeGraph**, are tools intended to help designers to generate cityscapes for games. The implementation of each tool is very designer-centric and provides a number of parameters to fine-tune the generation process. The reason why these systems were created in terms of the reasons to use PCG (chapter 2.2) is mostly to help the workload of the designers.

The reason why these tools exist is similar, but the implementations differ in both the way they are used, and the techniques used in the generation process. While **CityScaper** and **EdgeGraph** are implemented to be Unity Editor plug-ins, the **Horizon** is a standalone generator merely developed with Unity Engine (Table 1). This distinction defines the workflow of these tools, as a plug-in integrates to the overall Unity workflow, while the standalone tool is its own program. The **Horizon** generates the cities and exports 3D models that can then be used in any game engine or 3D modeling software. Both ways are valid, as the plug-in integrates to the overall workflow with Unity, but the standalone software works just as well with any software that supports the exported models.

The techniques used in the three tools are different (Table 1). The **Horizon** follows most closely the Parish and Müller's (2001) implementation, and its city street generation relies on circle noise for heat maps of the city density, followed by an L-system (Thompson, 2015). The **CityScaper** utilizes noise too, but it provides the whole of LibNoise[g] to the user to determine the density map. The **EdgeGraph** uses space colonization instead of the more widely used L-System as was described in chapter 5.4.1. The way the used techniques differ shows how the PCG systems can be implemented using different techniques, while the generated content is similar.

While the three tools are all created to generate cities, the use of these models and structures differs greatly (Table 1). The **Horizon** is a software meant to generate diverse, grid-based cities with unique buildings. The use of the content is similar to the Parish and Müller's CityEngine, as it is a real, unique, looking city when observed from afar. The **CityScaper's** original purpose was to generate realistic-looking cities as a background for a side-scrolling game[16]

[16] In a side-scrolling game, the player character moves only in two dimensions (side-to-side) on the screen viewed by the camera on the side, while the background might be in three dimensions.

(TikiHubGames, 2016). This means the cities generated by CityScaper are observed only from one side, which reduces the required complexity of the generated content. The **EdgeGraph's** initial purpose was to generate gameplay spaces for first and third person games, so its target is streets that are more interesting rather than realistic cityscapes.

	Horizon: City Generator	**CityScaper**	**EdgeGraph**
Techniques Used	Circle noise and L-system	LibNoise library	Space colonization
Standalone/Plug-in	Standalone	Plug-in	Plug-in
Intended Use	City and building generation that can be exported	Cityscapes as background	Gameplay spaces for 1st and 3rd person games

Table 1 City generator comparison

6 Conclusion

The purpose of this thesis was first to introduce, define, and dissect the procedural content generation as a whole while providing some example techniques, and second to research into city generation and the utility of Unity editor used in procedural content generation system implementation. The introductive first part was meant to familiarise the reader to PCG systems in general before giving concrete examples of implementations and one specific implementation of a city generation system.

6.1 Results

Here is the conclusion of the answers for the research questions of this thesis. The earlier chapters answer to these questions more in depth, and they are referenced here. However, this chapter offers a brief answer to each of the questions.

Why procedural generation is found useful in content creation?

The arguments as to why PCG is useful in content creation for games and other media is defined were laid out and explained in chapter 2.2: *memory consumption*, *prohibitive expense of manually creating game content*, *emergence of completely new types of games*, and *potential to augment human imagination.* These reasons provide the basis on the decision to implement a PCG system in content creation, and one of these reasons is usually the main reason as to why use PCG at all.

These reasons were also opened more in depth with examples in chapter 2.3. First example was the video game Elite, which has a PCG system to decrease memory consumption in the distribution media. The second example was the

video game Rogue, from which emerged a new type of game: Roguelikes. The third example was procedural vegetation generator SpeedTree, which is meant to decrease the workload of designers in placing unique trees and other vegetation to a 3D scene.

How to define the desired properties of a PCG system?

Definition of a PCG system can be done by placing the system between each of the **five pairs** of extremes that were introduced in chapter 2.4. The **first** pair is the distinction of when the generation process is done: is the generation run *online* or *offline*? The **second** determines whether the content is *necessary* or *optional* regarding to the completion of the game. The **third** determines the amount of parameterisation by differentiating between *random seeds* and *parameter vectors*. The **fourth** defines the amount of randomness by differentiating between *stochastic* and *deterministic* generation. Lastly, the **fifth** pair outlines the amount of evaluation of the generation output with *constructive* and *generate-and-test* generation.

An example of defining a PCG system was done first with the definition of the tool in the case study, the EdgeGraph, in chapter 5.1, and later two more examples of defining PCG systems were later made when comparing different implementations of city generation in Unity in chapter 5.5.

What are some often-used techniques used in PCG?

Some techniques that are often utilized in PCG systems were introduced and explained in chapter 3. The **pseudo-random number generators** are the very basis of the randomness in PCG systems and used in most of the other techniques, which is why they were introduced first. Other techniques were **gradient noise**, which is also a basic building block of PCG systems but relies on PRNG, vegetation generating **L-systems**, **random points**, and lastly two example techniques that utilize the random points: **Voronoi diagram** and **space colonization**.

How was the Unity game engine used in implementing the PCG tool of the case study?

For this thesis, a city generation tool was developed in Unity game engine editor. The editor provides three main ways to implement user interfaces for the user as was introduced in chapter 0: **Inspector**, **Scene view**, and **Editor Window**. Inspector and Scene view were utilized for the case study, EdgeGraph, and their usage in the tool was described in chapter 4.4.

The EdgeGraph tool consists of two parts: the primitive creation tool **EdgeGraph**, and the sub edge building **EdgeBuilder** introduced in depth in chapter 5.4. The former implements the designer-centric process by providing the user tools to define areas in which the generation process takes place, while the latter implements some PCG techniques to create unpredictable areas within the defined boundaries. The EdgeBuilder is deterministic system with a number of parameters used to tweak its behaviour (chapter 5.4.2).

The data structure of the nodes and edges generated by EdgeGraph is functional for building and vegetation placement as is, but especially the node manipulation could be improved (chapter 5.2). The user experience of the tool requires closer inspection and improvement to be useful in extended use as was described in chapter 5.3.

6.2 Future of the EdgeGraph tool

The EdgeGraph tools and the EdgeBuilder that were created for this thesis are not finished. They provided a viable platform in researching the implementation of procedural content generation systems, as the different means to approach city generation were prototyped and fitting ways for this specific use case were chosen (chapter 5.4.1). The tool is meant to be developed further if found useful, and the source code of the tool is openly available at https://github.com/famerij/EdgeGraph in order to encourage further inspection and possible usage.

References

Doull, A. (2008). *The death of the level designer.* Retrieved from http://roguelikedeveloper.blogspot.fi/2008/01/death-of-level-designer-procedural.html 18.1.2016

Doull, A. (2015). *Procedural Content Generation Wiki*. Retrieved from http://pcg.wikidot.com/ 18.1.2016

Eberly, D. (2005). *The Minimal Cycle Basis for a Planar Graph.* Retrieved from http://www.geometrictools.com/Documentation/MinimalCycleBasis.pdf

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

Eilertsen, B. G. (2013). *Automatic road network generation with.* Retrieved from http://www.diva-portal.org/smash/get/diva2:663032/FULLTEXT01.pdf

Esri. (2016a). *CityEngine SDK GitHub Repository*. Retrieved from https://github.com/Esri/esri-cityengine-sdk 18.1.2016

Esri. (2016b). *ArcGIS for Desktop, pricing*. Retrieved from http://www.esri.com/software/arcgis/arcgis-for-desktop/pricing 18.1.2016

Gallant, J. (2014). *Procedurally Generated Trees with Space Colonization Algorithm in XNA C#*. Retrieved from http://www.jgallant.com/procedurally-generating-trees-with-space-colonization-algorithm-in-xna/ 18.1.2016

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

Haahr, M. (2016). *Introduction to Randomness and Random Numbers*. Retrieved from random.org: https://www.random.org/randomness/ 18.1.2016

Haahr, M., & Haahr, S. (2016). *The History of RANDOM.ORG*. Retrieved from random.org: https://www.random.org/history/ 18.1.2016

Liapis, A., Smith, G., & Shaker, N. (2015). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Mixed-initiative.* Retrieved from http://pcgbook.com/ 18.1.2016

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

Perlin, K. (2001). *Standard for perlin noise.* Retrieved from http://www.google.com/patents/US6867776 18.1.2016

Persson, M. (2011). *Terrain generation, Part 1*. Retrieved from http://notch.tumblr.com/post/3746989361/terrain-generation-part-1 18.1.2016

Piiroinen, T. (2014). *Three-Dimensional Terrain Generation (in Finnish).* Retrieved from http://theseus.fi/bitstream/handle/10024/75343/Piiroinen_Tero.pdf

Side Effects Software. (2016a). *Unity Plug-in for Houdini Engine*. Retrieved from http://www.sidefx.com/index.php?option=com_content&task=view&id=2739&Itemid=381 18.1.2016

Side Effects Software. (2016b). *Houdini Engine*. Retrieved from http://www.sidefx.com/index.php?option=com_content&task=blogcategory&id=227&Itemid=381 18.1.2016

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

TikiHubGames. (2016). *CityScaper - Procedural City Generator*. Retrieved from Unity Forums: http://forum.unity3d.com/threads/released-cityscaper-procedural-city-generator.247856/ 18.1.2016

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.* Retrieved from http://www.ccs.neu.edu/course/cs5150f14/readings/togelius_what.pdf

Togelius, J., Shaker, N., & Nelson, M. J. (2015). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Retrieved from http://pcgbook.com/ 18.1.2016

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

Unity Asset Store. (2016). *CityScaper Store Page*. Retrieved from https://www.assetstore.unity3d.com/en/#!/content/17982 18.1.2016

Unity Technologies. (2016a). *Unity Documentation, Handles*. Retrieved from http://docs.unity3d.com/ScriptReference/Handles.html 18.1.2016

Unity Technologies. (2016b). *Unity Manual: Script Serialization*. Retrieved from http://docs.unity3d.com/Manual/script-Serialization.html 18.1.2016

Walker, J. (2006). *HotBits: Genuine random numbers, generated by radioactive decay*. Retrieved from fourmilab.ch: http://www.fourmilab.ch/hotbits/ 18.1.2016

West, M. (2008). *Random Scattering: Creating Realistic Landscapes*. Retrieved from gamasutra.com: http://www.gamasutra.com/view/feature/130071/random_scattering_creating_.php?page=2 18.1.2016

Wikipedia. (2016). *BBC Micro*. Retrieved from Hardware features: Models A and B: https://en.wikipedia.org/wiki/BBC_Micro#Hardware_features:_Models_A_and_B 18.1.2016

Wikipedia. (2016). *Point in polygon: Ray casting algorithm*. Retrieved from https://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm 18.1.2016

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

Glossary

[a] Elite (1984) is a space trading game, written and developed by David Braben and Ian Bell and originally published by Acornsoft for the BBC Micro and Acorn Electron computers.

[b] SpeedTree is a group of vegetation programming and modeling software products developed and sold by Interactive Data Visualization, Inc.

[c] Rogue (1980) by Michael Toy, Glenn Wichman, Ken Arnold and Jon Lane is a dungeon crawling game that uses ASCII art.

[d] Minecraft (2011) is a sandbox independent video game originally created by Swedish programmer Markus "Notch" Persson, later developed and published by the Swedish company Mojang.

[e] Unity is a game engine developed by Unity Technologies.

[f] Clipper is an open-source freeware library for clipping and offsetting lines and polygons. It has the source code in C#, and is included in the C++ Boost library.

[g] LibNoise is an open-source noise generator library.