UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Andreas Sepp

Infinite Procedural Infrastructured World Generation

Master's Thesis (30 ECTS)

Supervisors: Raimond-Hendrik Tunnel, MSc
Eero Vainikko, PhD

Tartu 2018

Infinite Procedural Infrastructured World Generation

Abstract:

This Master's thesis describes and provides an implementation of a novel algorithm for generating infinite deterministic worlds with both man-made and natural features commonly found in the civilized regions of the temperate climate zone. Considering that infinite worlds have to be generated in a piecewise manner without any of the neighbouring pieces necessarily existing, ensuring continuity and deterministic results for the generation of such features can be challenging. The algorithm uses an exponential generation technique, which enables the generation of varying sized features from traffic signs to rivers. The algorithm generates infinite road networks of different tiers, named cities and villages, power lines between them and common traffic signs like speed limits and navigation signs. Rural areas are generated based on three types of land usage – forestry, cultivation of crops and untouched nature reserves. The thesis also gives an overview of the previous work in the field of procedural world generation and proposes multiple new ideas for further expansion of infinite infrastructured terrain generation.

Keywords:

Procedural generation, computer graphics, exponential generation, infinite terrain generation, infrastructure generation, infinite road generation, city generation, village generation, agricultural generation, farm generation, forest generation, plant placement, electric network generation, simplex noise, curves, splines.

CERCS:

P150: Geometry, algebraic topology;

P170: Computer science, numerical analysis, systems, control;

P175: Informatics, systems theory.

Lõpmatu protseduurilise taristuga maailma genereerimine

Lühikokkuvõte:

Käesolev magistritöö kirjeldab uudset algoritmi lõpmatu deterministliku maailma genereerimiseks koos üldlevinud tehislike ja looduslike struktuuridega, mis leiduvad parasvöötme asustatud piirkondades. Kuna lõpmatuid maailmu tuleb genereerida jupikaupa ning ilma ühegi naabruses oleva tüki olemasoluta, on genereeritavate struktuuride järjepidev ja deterministlik genereerimine keeruline. Kirjeldatav algoritm kasutab eksponentsiaalse genereerimise metoodikat, mis võimaldab genereerida erineva suurusega struktuure alates liiklusmärkidest kuni pikkade jõgedeni. Algoritm genereerib erinevat tüüpi lõpmatuid teede võrgustikke, nimedega linnu ja külasid, elektriliine ja levinumaid liiklusmärke nagu kiiruspiirangud ja suunamärgid. Asulatest väljaspool olev maastik genereeritaske kolme üldlevinud maakasutuse kategooria vahel – metsandus, viljakasvatus ning looduskaitsealad. Lisaks kirjeldatavale algoritmile antakse ülevaade eelnevast teadustööst lõpmatu protseduurilise maailma genereerimise valdkonnas ning kirjeldatakse edasiarendusvõimalusi lõpmatute asustatud maailmade genereerimiseks.

Võtmesõnad:

Protseduuriline genereerimine, arvutigraafika, eksponentsiaalne genereerimine, lõpmatu maastiku genereerimine, taristu genereerimine, lõpmatu teede genereerimine, linnade genereerimine, külade genereerimine, põllumajanduse genereerimine, põldude genereerimine, taimede paigutus, elektrivõrgustiku genereerimine, simpleksmüra, kurvid, splainid.

CERCS:

P150: Geomeetria, algebraline topoloogia;

P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine;

P175: Informaatika, süsteemiteooria.

Contents

1 Introduction

The creation of high quality and large visual assets with little variance manually can be an arduous and costly process. For this reason, it is wise to automate the asset creation pipeline as much as possible and to create some of them procedurally. Procedural generation is the application of algorithms to procedurally generate content based on some input parameters [HMVDVI13]. Procedural generation in general is a pretty thoroughly investigated area in scientific research [HMVDVI13, SKK+14] and can be applied for a wide range of purposes, such as texture generation [Per85, LGD+18] and placement [LN03, LRD07], terrain [PGGM09, Ers12, Sep16], vegetation [OBRvdK17, Tun12], city [PM01, Per16] and building generation [MWH+06, EBP+12].

However, despite being popular in gaming[1], the scientific coverage of infinite world generation has been small, especially in regard to deterministic and complex worlds. Most existing infinite world generation solutions focus on terrain generation [BPd09] or city generation [SKK+14], but there is only one known solution for combining several complex features [eM17]. The goal of this thesis is to create a novel deterministic infinite world generation algorithm, that is capable of generating a varied world with different structures commonly found near civilized regions in the temperate climate zone. The algorithm generates an infinite road network with multiple tiers, including major highways and minor side roads with infrastructure like traffic signs and utility poles; cities, villages and rural areas. The rural areas are refined to look like respective real-life areas, where most of the land is in active use for cultivation or forestry while some is being preserved as a nature reserve. The algorithm could be used as a part of a game world for either a single- or a multiplayer experience; part of a learning environment for various artificial intelligence, such as self-driving vehicles, or a basic driving simulator. With some alterations, it could also be used as a world for a flight simulator.

[1] http://pcg.wikidot.com/category-pcg-games

Although a large number of papers on the topic of finite world generation already exist [STBB14], the existing techniques can not usually be easily converted for use in infinite world generation. The conversion problems arise from peculiar characteristics of infinite generation. First, an infinite world has to be divided into smaller discrete areas and each area has to be generated separately to match other pieces without any neighbouring pieces actually existing. Secondly, if the infinite world is desired to be deterministic, it is necessary for each generated area to always provide the same results when regenerated.

It is also possible for a world generation algorithm to not be globally deterministic by only considering existing world areas and generating new areas to match the existing area edges. This way circling back to the same area in world space will not necessarily provide the same results, as the generated areas depend on the direction from which it was approached. This is commonly seen in infinite runner games[2] where the player can choose between multiple paths and the track is only generated for the path chosen by the player, with everything behind disregarded.

To generate an infinite world, it is empirical to divide the world into handleable finite pieces, which can be generated independently and in parallel. The simplest and most common approach is to divide the world into cuboid shaped chunks, which will be placed spatially next to one another. However, as complex features can span a large number of chunks, it can be difficult to determine which features should be generated on a chunk without first determining the properties of other nearby chunks. For example, to determine the existence of a river on a chunk, it is usually necessary to know where all the nearby rivers start and end along with their shapes before the existence of any rivers on the current chunk can be determined.

As discussed by Johansen[Joh15], three common approaches to world generation exist . The *simulation approach* generates the environment by simulating the processes

[2]https://en.wikipedia.org/wiki/Temple_Run_2

that create it in near real-life, such as erosion, sunlight coverage and wind. These simulations usually run for a large number of iterations and are therefore usually unsuitable for real-time generation of infinite terrain, as the player will be moving to new areas and the generation has to keep up in real-time. The *planning approach* is used to generate worlds with specific constraints around which the world is generated. For example, it might be necessary for a guaranteed path to exist between two locations on the terrain and the world is generated around this path. As these constraints are usually reserved for use in games with specific goals, they are not commonly used in infinite world generation. Lastly, *functional approach* tries to approximate the environment's end result by using mathematical functions. For example using *gradient noise* to generate a heightmap of the terrain. Functional approach algorithms are usually quick to evaluate and are therefore perfect for an infinite world where new areas need to be generated in real-time.

Another important property of infinite generation is *context sensitivity* [Joh15], which describes how much neighbouring data has to be calculated in order to evaluate features at given coordinates. For example, in case of a featureless flat world, there is no context sensitivity since the position of terrain can be evaluated at any position by checking the coordinate values. On the contrary, checking if some given coordinates contain a river usually requires knowledge of all the nearby rivers that can reach this position.

In chapter 2, an overview of previous work in the field of procedural world generation is given. The featured papers mainly focus on one of four topics – terrain, road, city or vegetation generation.

In chapter 3, an overview of the main background concepts are described. The chapter starts with the description of the world representation, consisting of 2 differently sized types of nested chunks, called macrochunks and mesochunks. The chapter continues with the description of the algorithm's context sensitivity, generation of deterministic pseudorandomness at any world position, used methods to place vegetation on the terrain and a flowchart of the devised algorithm's generation pipeline. The second half of the

chapter focuses on more abstract concepts like computer generated noise, curves, splines and boolean operations on arbitrary polygons.

In chapter 4, the generation of the larger pieces of the world, called macrochunks is detailed. Macrochunks are used for generating local highways and the locations and the outlines of nearby cities.

Chapter 5 describes the generation of smaller chunks, called mesochunks. The mesochunks are generated inside the macrochunks and use the macrochunks data to generate smaller features. Mesochunks first generate side roads, a type of minor roads, and the overlapping internal layout of a city if one is present. Areas outside the cities are mostly populated with agricultural land and other smaller settlements. Additionally, traffic signs and utility poles are generated in suitable areas. Lastly, after generating their data, a description of how mesochunks are rendered is provided.

Chapter 6 details the implementation of the algorithm. That is followed by a visual and a performance assessment of the results. The visual assessment is done by comparing the algorithm's output to real-life imagery. Performance assessment chapter gives an overview of average frame rendering times, speed of world generation and memory usage of the algorithm.

In chapter 7, potential future expansions for the devised algorithm are discussed. The largest chunk, called megachunks, is introduced, which could be used to generate railroads and water bodies like rivers and lakes. Furthermore, a possible integration of AI agents, a more intricate electric network and improved junctions is discussed.

Appendix I contains the phrases used in the settlement name generation algorithm. Appendix II contains details of the final application created for the thesis. Appendix III contains the glossary of the new terms introduced in this thesis.

The reader is expected to know the terminology covered by the Master's program of Computer Science and the basic terminology used in computer graphics.

2 Previous Work

This chapter gives an overview of various existing solutions used to generate procedural worlds and their elements. The first subchapter gives an overview of methods used to generate procedural terrain and water bodies. The following subchapters give an overview of road and city generation and the last subchapter discusses vegetation generation and placement.

2.1 Terrain Generation

Terrain is perhaps the most common element to be procedurally generated other than textures. Modelling terrain by hand can give great results. However, working on vast areas gets cumbersome and a lot of terrain creation workflows involve at least some procedural generation [SDKT+09].

Terrain is most commonly presented as a 2D grid, called a heightmap, where each value represents the elevation at the given location. As Smelik *et al* [SDKT+09] point out, heightmaps are easy to implement and can be efficiently compressed and stored on GPUs. Heightmaps are usually generated using *gradient noise*, such as Perlin noise [Per85] or its improved successor simplex noise [Per02] which are further discussed in Chapter 3.6. Sometimes, noise-based terrain does not necessarily yield good enough results and further terrain processing is done with physics-based algorithms. These algorithms usually simulate the erosion by weather. As mentioned by Smelik *et al* [STBB14], erosion is typically implemented as a global operation and runs for thousands of iterations, making it unsuitable for infinite and real-time generation. However, they also point out that *GPGPU programming* has enabled ways to speed up erosion simulation remarkably for interactive use [VBHS11, KBKŠ09].

Heightmap based terrains are limited to variation on a single upwards axis and are not capable of generating overhangs. Several other approaches exist, which enable the

generation of more complex terrain with volumetric data structures with the use of *voxels* [Sep16] and their advancements (Figure 1) [PGGM09]. As fully procedurally generated landscapes with little user input can be rather monotonous, a notable amount of work has been done to the enable direct sculpting of the terrain. For example, The Arches framework (2009) enables the user to interactively sculpt the terrain by sweeping with a brush, which also places smaller terrain features like rocks automatically (Figure 2) [PGGM09]. Zhou *et al* described a method for generating terrain from 2D user line drawings and combining it with a pre-existing digital elevation model [ZSTR07]. Their system extracts features from the original digital elevation models, like mountain ridges and ravines, and uses the drawn sketch to apply these extracted features along the drawn lines. Gain *et al* and Hnaidi *et al* created systems to generate terrain from user-drawn 3-dimensional curves [GMS09, HGA+10], which lets the user also control the height of the terrain in the up direction.

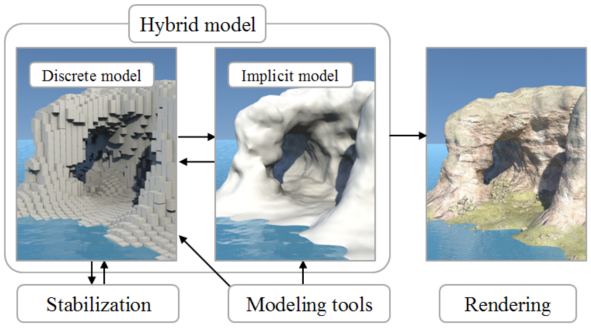

Figure 1. Arches terrain generation pipeline [PGGM09].

Figure 2. Terrain generated by the Arches framework [PGGM09].

In addition to general terrain generation algorithms, additional adaptations are required to also generate water bodies. Although water body generation has been somewhat under-addressed according to Smelik *et al* [STBB14], a few solutions do exist. The simplest way to generate water is to pick a specific height level as the global sea level

Figure 3. A procedural river that has been integrated with a terrain heightmap [GGG+13].

Figure 4. A river with vegetation generated with user controlled control points [HDBB10].

and render a plane of water at that height. This is especially useful for infinite world generation, as different water levels for different bodies of water would normally require knowledge of the whole body of water, which might not be available. Additionally, water bodies can be generated using flood algorithms for arbitrary water levels in different terrain basins. The generation of more complex water bodies like rivers requires a more intricate approach. Commonly either the terrain heightmap is generated with the rivers integrated into it (Figure 3) [KMN88, GGG+13] or the terrain is post-processed to fit a river to an existing heightmap, with the latter usually enabling the user to have interactive control over the shape of the river (Figure 4) [STdKB11, HDBB10].

Magalhães generates water bodies for his infinite world by using a fixed water level and using Perlin noise-based heightmap to determine if a point is underwater [eM17]. He also generates rivers with two different approaches. The first approach determines a number of random points in the world by sampling high frequency Perlin noise and picking vertices that are the local maximums of noise. Rivers are then constructed through these points in a fixed size area by traversing them, starting from a point in water. Each next point is chosen within a fixed radius with the smallest possible elevation still higher than the current point. (see Figure 5). The second approach also starts from the water and picks points on the heightmap with the minimal neighbouring height that is

higher than the current height Figure 6). The first approach generates longer rivers as it is possible for the rivers to pass local maximums on the terrain, while the second approach gets stuck at local maximums and ends earlier. Due to the first approach sometimes possibly going uphill, extra care has to be taken in the surrounding area to smooth out the terrain near the river. This is not a problem for the second approach as the river will always be guaranteed to only increase in elevation when starting from the initial water body. As both of these approaches also enforce a maximum river length, surveying an area with the given radius around any chunk will be guaranteed to find any intersecting rivers.

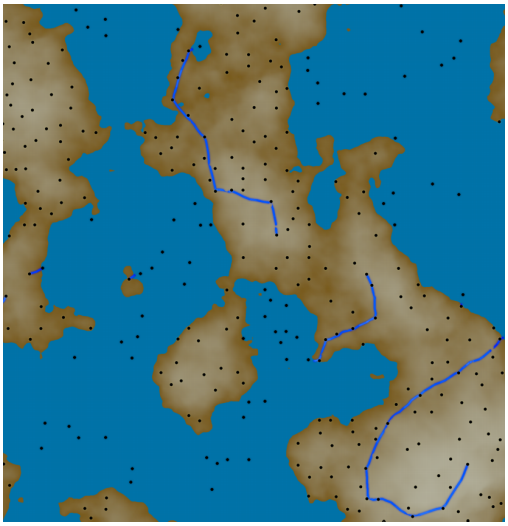

Figure 5. Rivers generated on infinite terrain through pseudorandom points (black) determined by Perlin noise [eM17].

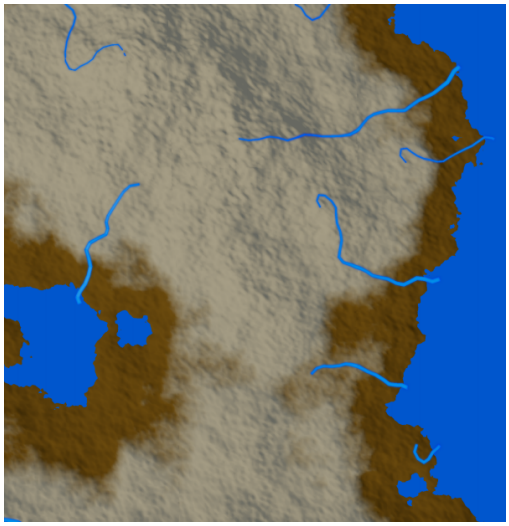

Figure 6. Rivers generated on infinite terrain by picking the direction with minimal elevation increase. [eM17].

2.2 Road Generation

To generate an urban environment the generation of roads is required. Smelik *et al* point out in their survey that procedural road generation has primarily been addressed for city generation, with interstate and country roads requiring further attention [STBB14].

For the urban area generation, usually a street network is first generated and used a basis for building generation [KM07, PM01, Per16]. For example, the Citygen (2007) algorithm first generates a high level graph for determining the connected road nodes and then generates a low level graph to make each high level graph edge adapt to the terrain (Figure 9) [KM07]. For the terrain adaptation, 3 different strategies are discussed: minimum elevation, least elevation difference and even elevation difference (Figure 10). After the primary road network generation, the plots of land confined between roads are each separately considered for further subdivision with the secondary roads. For secondary roads, three approaches are discussed: a grid based pattern, an industrial and an organic growth pattern (Figure 11). Another solution called CityEngine (2001) uses an extended L-system to generate 3 types of frequently appearing road patterns: branching, radial and grid-like (Figure 7); and additionally constraints the road placement near bodies of water by either following the shoreline or creating a bridge (Figure 8) [PM01]. For the country roads, bridge and road generation is also discussed by Galin *et al* [GPGB11]. The authors' algorithm places the roads by using an A*-based approach with a cost function dependent on the slope, bodies of water and surrounding vegetation (Figure 12). Another solution by McCrae *et al* exists which lets the user sketch rural roads, which are then placed on arbitrary terrain with support for bridges and tunnels [MS09].

For infinite road generation, e Magalhães generates city street networks by taking an axis-aligned grid bound by city borders and deterministically disturbing its vertices and adding some new edges by using Perlin noise (Figure 13) [eM17]. This creates a very

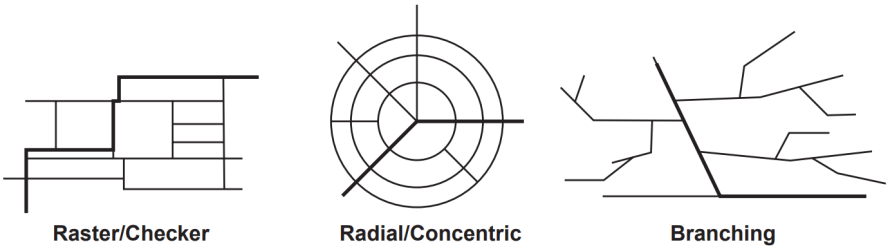

Figure 7. Frequent road patterns from CityEngine [PM01].

Figure 8. Street network with bridges generated with CityEngine [PM01].

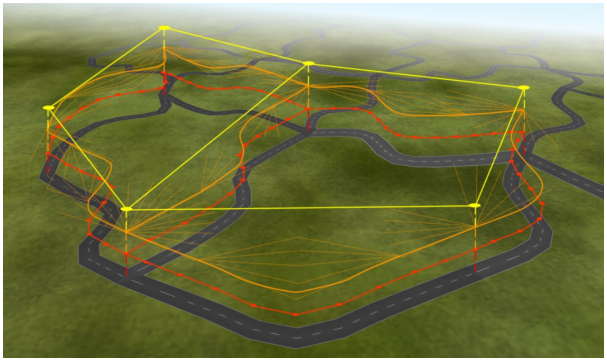

Figure 9. Road graph in CityGen. Yellow: high level graph. Red: low level graph that has been adapted to the terrain [KM07].

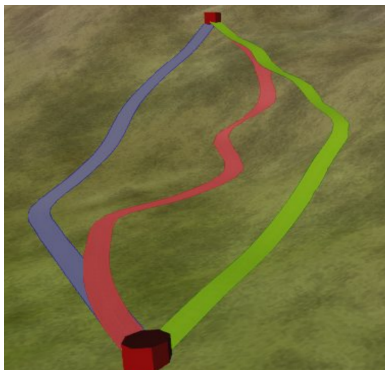

Figure 10. Adaptive road types in CityGen. Blue: minimum elevation. Red: least elevation difference. Green: even elevation difference [KM07].

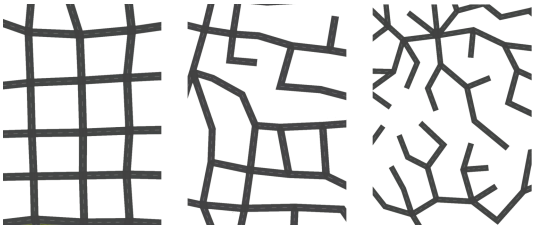

Figure 11. Grid-like, industrial and organic secondary road network patterns in CityGen [KM07].

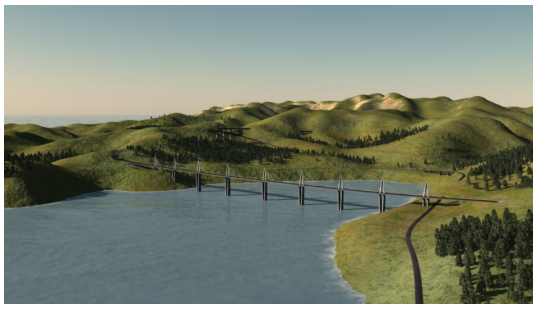

Figure 12. Procedurally generated adaptive roads with a bridge [GPGB11].

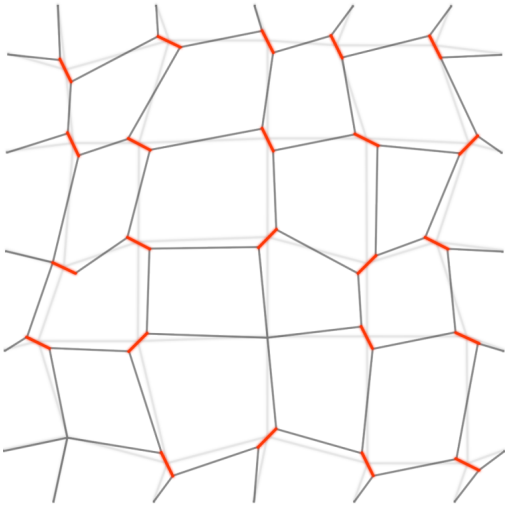

Figure 13. Grid and Perlin noise based street network for an infinite world by e Magalhães [eM17].

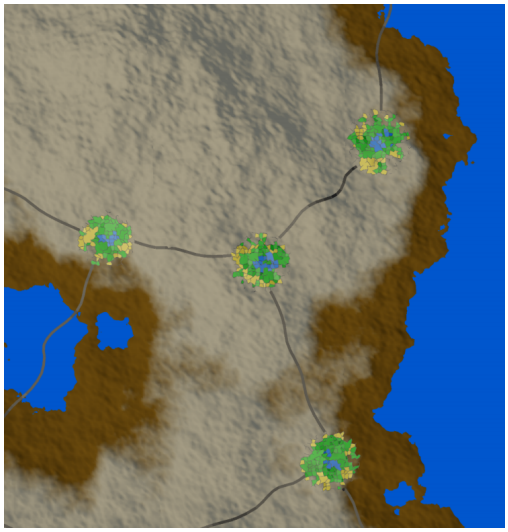

Figure 14. Cities and highways generated on infinite terrain by e Magalhães [eM17].

dense and oddly uniform street network. For intercity highways, cities are connected with straight highways fitted to the terrain (Figure 14).

Many other algorithms employ a similar approach to CityGen and CityEngine by first generating a primary road network and then generating additional secondary roads. For example, Chen *et al* use *tensor fields* to generate secondary roads by tracing the streamlines from seed points on primary roads (Figure 15) [CEW+08]. Lechner *et al* use an agent-based technique, which looks for unconnected areas and then connects them with optimal roads and generates smaller streets around them (Figure 16) [LWW03, LRW+06]. The agent-based approach gives good-looking results, but has a very long running time, making it unsuitable for real-time generation.

Figure 15. A street network generated from a tensor field [CEW+08].

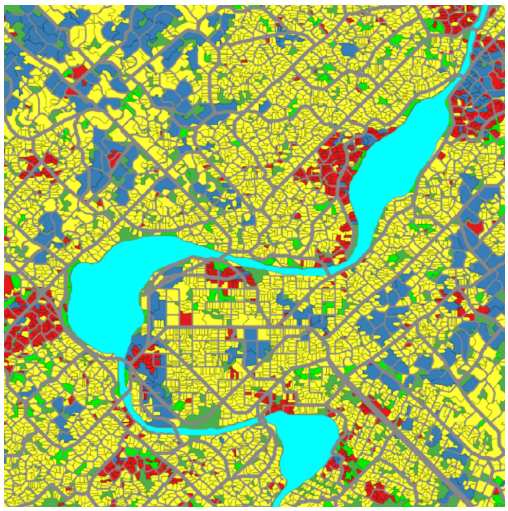

Figure 16. A street network generated with an agent-based technique [LRW+06].

2.3 City Generation

Procedural city generation is a widely covered topic with various approaches. Commonly, the cities are generated in a finite space [Per16, KM07] or infinitely covering the whole space [SKK+14, GPSL03]. Usually a street layout is generated and then the plots between the streets are filled with procedurally generated buildings. Common methods for street layout generation were covered in the previous chapter. Some building generation algorithms also take into consideration the location of the plot within the city (Figure 17) [LHdV17, Per16] and generate appropriate buildings for each city district. For example, the central areas of the city tend to contain tall and commercial buildings while the city outskirts contain mainly lower residential or industrial buildings.

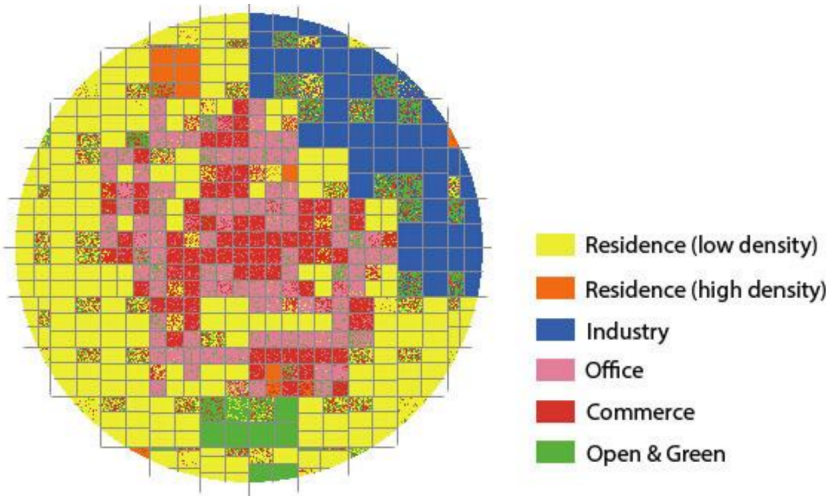

Figure 17. Procedural city land use distribution as demonstrated by Lyu *et al* [LHdV17].

Procedural building generation is a generally well covered area of procedural generation and commonly uses some form of a formal rewriting system, such as an L-system, a split grammar or a shape grammar [STBB14]. For example, Greuter *et al* generate buildings by combining primitive polygons for a floor plan and extruding them to different heights (Figure 18) [GPSL03]. Müller *et al* generate buildings via extended shape grammars [MWH+06], which are specifically designed for building facade gener-

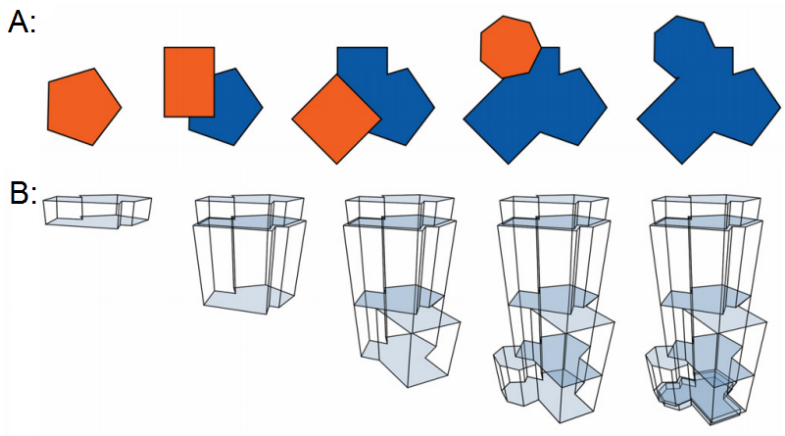

Figure 18. A: Generating building floor plan by combining primitive polygons. B: Extruding the building shape from the combined polygons [GPSL03].

Figure 19. Procedurally generated buildings in the style of Pompeii [MWH+06].

Figure 20. Procedurally generated buildings in a modern style [MWH+06].

Figure 21. A procedurally generated village on a coast [EBP+12].

Figure 22. A procedurally generated flower with temporal growth [PHM93].

ation (Figures 19 and 20). Coelho *et al* generate buildings using L-systems [CdSF05]. Additionally, there has also been notable work done in the field of building interior generation, which is discussed in greater detail by Smelik *et al* [STBB14]. Silveira *et al* generate buildings using a predefined floor plan and building style semantics [SCM+15].

Contrary to cities, there has been less work in the field of procedurally generating smaller settlements. Emilien *et al* generate villages by first generating a suitable outline for the village based on the surrounding terrain and then generating suitable meshes and locations for the houses (Figure 21) [EBP+12].

2.4 Vegetation Generation

Although the algorithm developed in this thesis does not generate vegetation meshes procedurally and uses pre-existing models instead, many solutions to generating vegetation procedurally exist. One approach to generate plants is to use L-systems [Lin68] which can be used to create a large number of unique but similar looking plants [Tun12] and to simulate continuous plant development (Figure 22) [PHM93]. For higher interactivity, an algorithm by Longay *et al* enables the user to alter the shape of the generated structures by manually placing splines [IMG] [LRBP12].

Figure 23. Procedurally generated meadow with multiple types of plants [OBRvdK17].

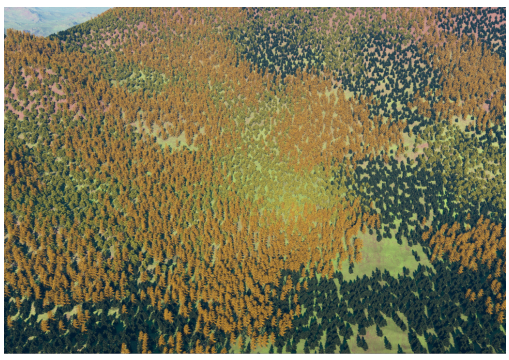

Figure 24. Procedurally generated forest with multiple species of trees [LGL+18].

For procedural vegetation placement, multiple solutions exist. Deussen *et al* generate the placement of multiple varied sized plants using a simulation approach which runs for many iterations in a fixed area [DHL+98]. The algorithm also simulates plant survival by eliminating some neighbouring plants as they grow and compete for resources. Due to the slower simulation approach of their algorithm and the fact that plants being placed in the system described in this thesis have much less variance in size, a different approach was taken. Another solution by Hammes *et al* uses the elevation, slope and noise to select one of predefined ecosystems [Ham01]. Each ecosystem determines the number of plants for each species, which are then scattered randomly on a grid. Their system uses a grid where each tile can be independently and recursively divided into smaller tiles to any depth. With each final tile containing up to one plant, which has been slightly offset within the bounds of the tile. This could in theory cause a noticeable grid-like layout when its not wanted in a very densely populated area and was therefore not used for the current algorithm. Onrust *et al* discuss methods to place plants based on ecological statistical data and landscape maps and rendering them (Figure 23) [OBRvdK17], but the goal of this thesis was to try to use pre-existing data as little as possible and therefore the developed algorithm did not use this approach. Li *et al* discuss large-scale forest generation to achieve realistic forests (Figure 24) [LGL+18] by scattering seeds and letting natural clusters of different species of trees to emerge. In the current algorithm, the tree generation is handled by a slightly similar approach where instead every forest area is subdivided beforehand and each subdivided area is separately populated with different types of forests with some regard for neighbouring forests.

3 Generation Preliminaries

This chapter gives an overview of the main concepts that are used throughout the following generation chapters. In the first half algorithm-specific methods and in the second half more general concepts are discussed. The first half covers the world representation as chunks, chunk window-based generation, generating a deterministic sequence of pseudorandom numbers for each chunk, generating vegetation placement and an overview of the pipeline of the created generation algorithm. The second half covers more general techniques concepts such as noise, splines and boolean operations on 2-dimensional polygons.

3.1 World Representation

As different real life features from rivers to street signs differ greatly in scale, it is a good idea to adapt to this. To accommodate the varied size features, an exponential generation technique is used. Exponential generation uses exponentially sized nested chunks similar to a quadtree[3] – each large chunk contains a number of a tier smaller chunks, which in turn contain a number of the next tier chunks and so on. As the different tier chunks vary a lot in size, they are suitable for generating features of their respective size. For example, a large chunk with width in hundreds of kilometers could be used for generating gigantic features such as mountain ranges and large rivers while smaller chunks with width in hundreds of meters can be used for generating smaller features like city streets and traffic signs. The generation of any chunk assumes that all the larger tier parent chunks containing this chunk have already been generated. It is then possible for smaller chunks to query their parent chunks for the larger features.

In the context of the algorithm described in this thesis, the world is considered to be an infinite plane, which is divided into chunks (see Figure 25). Each chunk is responsible

[3] https://en.wikipedia.org/wiki/Quadtree

for generating everything from bottom to the top of the world within its bounds of the world plane. There are 2 types of these chunks in the current implementation:

- mesochunks of size 512×512 meters;
- macrochunks of size 8192×8192 meters (16×16 mesochunks).

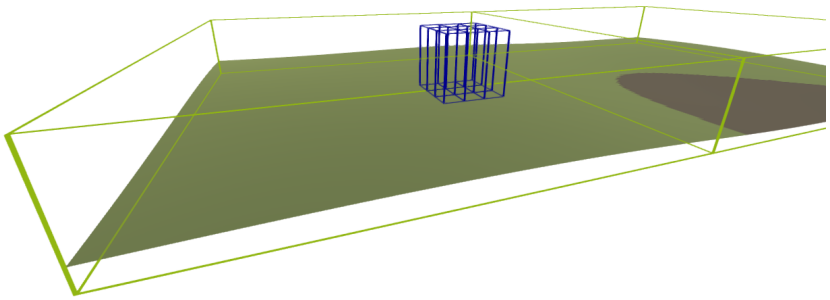

Figure 25. 2 macrochunks (green) with 9 mesochunks (blue) inside one of them.

Macrochunks are used to generate large scale features like highways and city outlines and act as data containers for generating mesochunks. Mesochunks can then query their parent macrochunk for the locations of cities and highways and generate smaller features like side roads, villages and inner city structure. It is necessary for a parent macrochunk to exist before a mesochunk can be generated. A third potential chunk called a megachunk is also discussed in Chapter 7, which could be used to generate water bodies and rivers. Each chunk has distinct global 2-dimensional integer coordinates, defining their position in the world space.

3.2 Chunk Window

For context sensitivity, the described algorithm uses a window-based approach, which ensures that a cluster of neighbouring chunks, called a *chunk window*, around any specific chunk will contain all the data required to generate the specific chunk (Figure 26). It is important to note that the other chunks in the window do not have to actually exist to generate the central chunk. All the required context data for the other chunks can be quickly calculated based on their coordinates as discussed in the upcoming chapters.

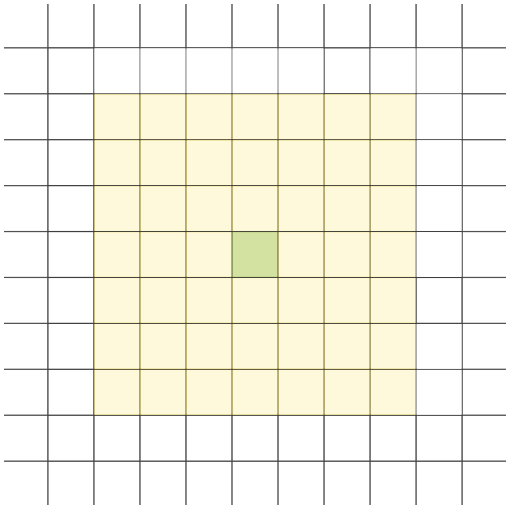

Figure 26. To generate a specific chunk (green), some context data about its neighbouring chunks (yellow) in a 7×7 window is required.

3.3 Chunk-Based Pseudorandomness

To ensure that the algorithm's output is deterministic but still random-looking, a method of providing deterministic pseudorandom numbers to each chunk is required. A simple method for this is to use the given chunk's unique global coordinates. The coordinates can be used as an input for a 2-dimensional integer hash function $f_{hash} : \mathbb{Z}^2 \to \mathbb{Z}$ and the hash used to instantiate a seeded pseudorandom number generator[4] (PRNG). The hash function used in this algorithm is

$$f_{hash} : (x, y) = (x \cdot 73856093)\text{XOR}(y \cdot 83492791).^{5}$$

The described algorithm requires every chunk to provide a lot of different pseudorandom values for the generation of various features. To remain consistent, it is important that when asked for the n-th element of a chunk's PRNG output, the same value is always returned. However, PRNGs only provide one number at a time based on the previously generated number and generally do not provide a way to roll back to fetch previously previous values. Therefore, to get the n-th element, it is necessary to either cache each chunk's PRNG output in an array or to recreate a PRNG and to iteratively generate the n-th element.

As further described in Chapter 4.2, the generation of one macrochunk requires generating pseudorandom values for itself and its 48 neighbours, which would require 49 PRNG instances. As such, it is wiser to instead cache the first 32 pseudorandom values, which are most commonly needed, for each chunk and use these instead to save some overhead. If in some cases more than 32 values are required, an instantiated PRNG can be used to generate more values.

[4] https://en.wikipedia.org/wiki/Pseudorandom_number_generator

[5] http://www.beosil.com/download/CollisionDetectionHashing_VMV03.pdf

3.4 Vegetation Placement

As the described algorithm has to place a large number of plants in areas like forests and farms, it is necessary to have a performant solution to quickly choose their positions. Some solutions for this were discussed in chapter 2.4, which were not directly suitable for use here. Instead, another system with two types of vegetation placement was developed. The *grid placement* method replicates the way plants are planted in farms and planted forests by aligning plants with the bounds of the area they are generated in. *Poisson-Disk placement* method replicates the natural placement of plants, where no clear pattern emerges but each plant is at least a fixed minimal distance away from its neighbours.

3.4.1 Grid Placement

When plants are planted by humans, they are usually planted with a distinct pattern. This is especially clear on farmlands, where plants have been planted in rows with roughly uniform steps, but it can also be seen in planted forests. An approach to simulate this distribution for a region is to take the region's longest edge and sample points at uniform intervals parallel to it as seen in Figure 27. Easiest way to do this is to rotate the polygon so that its longest edge is aligned with the x-axis and then calculate the new polygon's axis-aligned bounding box. Next, axis-aligned points can be sampled within the bounding box and checked if they are contained in the rotated polygon. After a point has been determined to be inside the rotated polygon, it can be translated back to the original polygon's space.

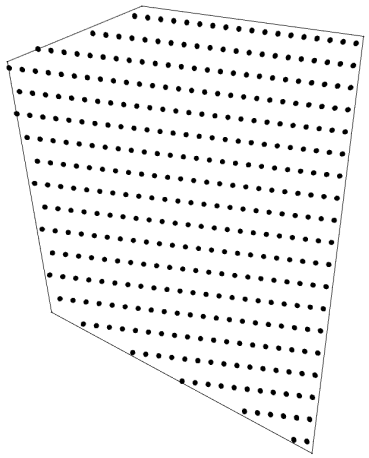

Figure 27. Grid sampled points in a polygon, aligned with its longest edge.

3.4.2 Poisson-Disk Placement

When plants have grown without direct interference by humans, they are usually placed *quasirandomly*. This means that there is no direct pattern to their positions, however their placement is rather uniform with some minimal distance between neighbouring plants, due to their competitive nature for sunlight and nutrition. For this reason, generating plain pseudorandom points for their positions would not work, as some objects would overlap with each other, be very packed in some areas and be very sparse in other areas.

One suitable algorithm for generating such a distribution is called Poisson-disk sampling. It produces points that are uniformly distributed but no closer to one another than a specified minimum distance d (Figure 28). Such a distribution of n objects can be generated with $\Theta(n)$ complexity [Bri07]. This distribution can be used to place plants in nature reserves and other regions that require a natural looking distribution.

In addition, Poisson-disk can be modified to also place a secondary type of plants, albeit less frequently, with unlikely collisions between either type. By storing the last sampled Poisson-disk point and comparing the distance from it to the next sampled Poisson-disk point, it is possible to place a secondary plant in the midpoint of these 2 points as long as the edge is shorter than $1.75d$ (Figure 29).

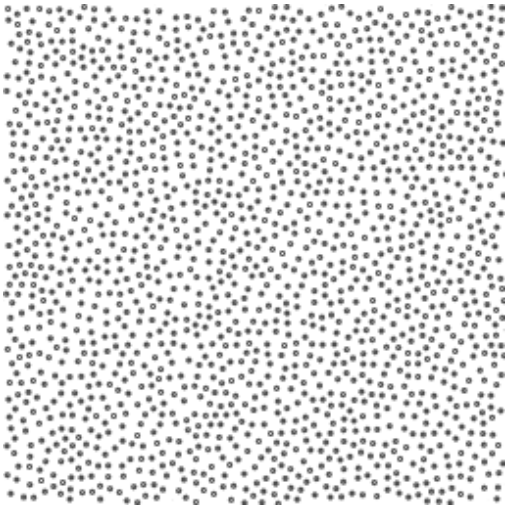

Figure 28. Poisson-disk sampled points.

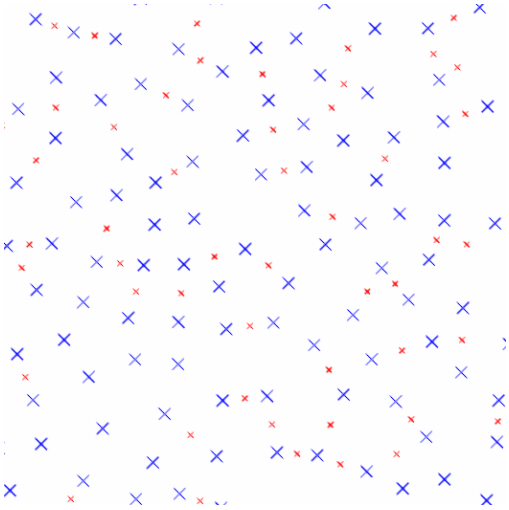

Figure 29. Poisson-disk sampled points (blue) with secondary points (red).

3.5 Generation Pipeline

This chapter gives a visual overview of all the steps involved in the developed system, which are explained in further detail in upcoming chapters. The generation pipeline is visualized in Figure 30. The user can request any macro- or mesochunk to be generated, though only requesting mesochunks is enough, as any required macrochunks for them will be generated automatically.

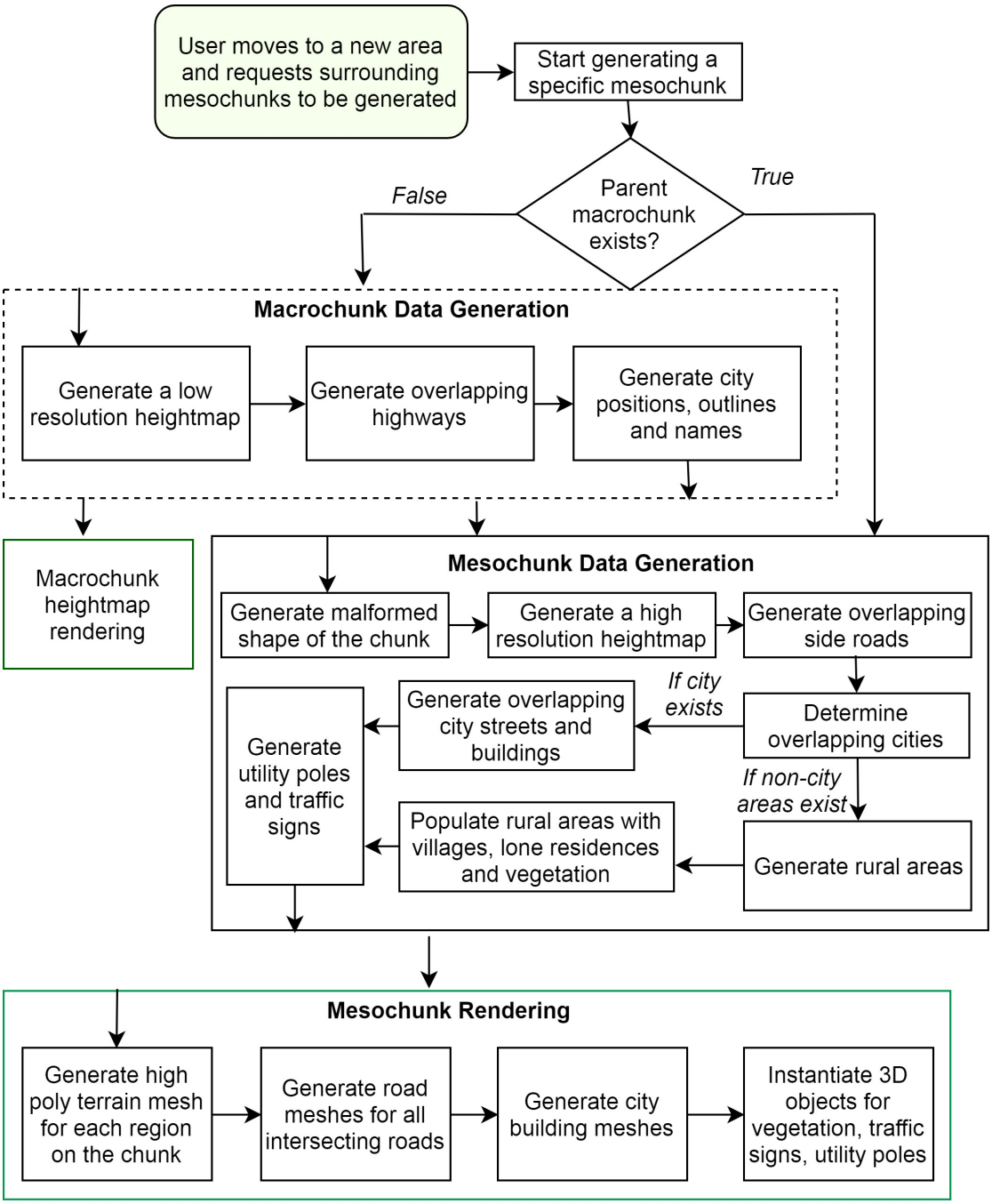

Figure 30. Generation pipeline of the described algorithm.

3.6 Noise

Computer generated visual noise is one of the fundamental tools for generating rich visual detail for synthetic images as introduced by Perlin [Per85]. Noise functions are also widely used in procedural terrain generation. The value of a noise function can be evaluated at any point without the need for any other neighbouring data to exist, but neighbouring values will still be similar. This makes noise very useful for generating simple and continuous infinite terrain, as a 2-dimensional noise function can be used as an infinite heightmap that can be queried in a fast and simple manner (Figure 31).

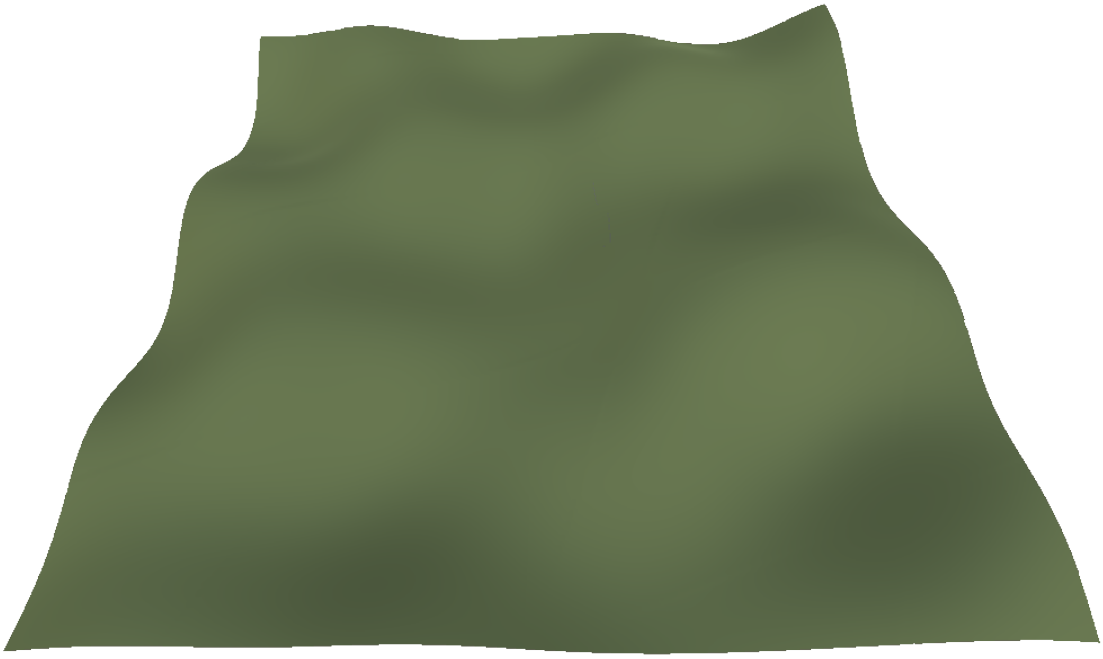

Figure 31. A heightmap generated using simplex noise that can be easily expanded in any direction.

The first commonly used type of *gradient noise* was Perlin noise [Per85], which was later improved by himself to become what is now commonly known as simplex noise [Per02]. Simplex noise is computationally cheaper and has no noticeable directional

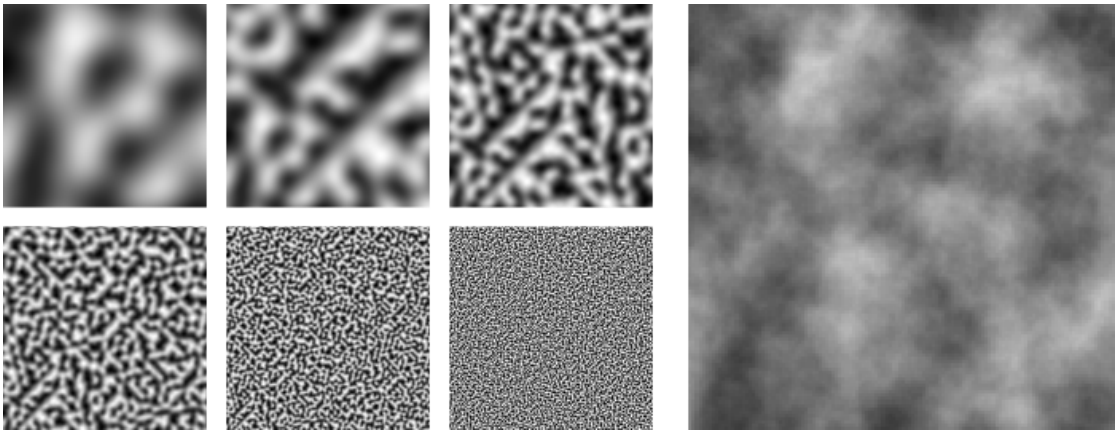

Figure 32. Left: 6 separate layers of noise. Right: The same 6 layers combined together forming the final noise image. The first layer has frequency $\frac{1}{64}$. Lacunarity is 2 and persistence is 0.5. Dark colors show minimum values and light colors show maximum values.

artifacts [Gus05] and is extensively used throughout the algorithm. Simplex noise is constructed by combining multiple layers of different frequency noise values together into a single noise value. Each layer has a frequency, which determines the minimal distance after which the noise gradient can change direction. Usually only the frequency of the first layer is explicitly denoted and the frequencies of the following layers are determined using a lacunarity multiplier. Lacunarity multiplier determines the frequency of a layer by taking the frequency of the previous layer and multiplying it by the lacunarity. Lastly, a persistence multiplier is used to determine the amplitude of effect of each layer in the final result. The amplitude of a layer is calculated by taking the last layer's amplitude, starting with 1, and multiplying it with the persistence multiplier. The result of this process with each separate layer can be seen in Figure 32.

3.7 Curves and Splines

The generation of roads in Chapter 4.2 requires a way to generate smoothly bending roads through a set of fixed points on the road to imitate the shapes of real-life roads. This chapter gives a short overview of terminology involved with curves and splines, based on the Fundamentals of Computer Graphics book by Marschner and Shirley [MS15] and introduces a new type of curve that is used to generate suitable road shapes in the devised algorithm.

A *curve* is a continuous image of some interval in an n-dimensional space. Intuitively, any drawn line on a piece of paper can be regarded as a 2-dimensional free-form curve. In current context, only 2-dimensional curves are of interest, as the road curvature is viewed from top down.

A curve can be specified mathematically by using a parametric representation. A *parametric curve* is a curve that provides a mapping from a parameter t to the set of points on a curve using a parametric function. For example, time can be considered the parameter of a hand drawn curve on paper. In 2-dimensional space, the parametric function $f(t)$ of a curve maps to a point (x, y) on the curve. For instance, a unit circle can be represented as a parametric curve $f(t) = (\cos t, \sin t)$ for $t \in [0, 2\pi)$.

To represent an arbitrary curve as a parametric curve, a divide-and-conquer approach is taken. The curve is broken into a number of smaller segments, each of which can be described using a simple polynomial function, called a *parametric piece*. Each parametric piece has its own parametric curve representation $f_i(t)$ for $t \in [0, 1]$. These parametric pieces can be grouped into a single piecewise polynomial function, called a *spline*, which then approximates the arbitrary curve.

The part of a spline where two sequential parametric pieces f_i and f_{i+1} meet can have various properties. If $f_i(1) \neq f_{i+1}(0)$, then the curve is not positionally continuous as the end and start positions of respective pieces are not at the same position. In addition to

positional continuity, the derivatives of the pieces can be compared. If $f_i'(1) \neq f_{i+1}'(0)$, then the first derivatives at the meeting point for both pieces are different, which makes the spline have an abrupt change at the given point. To describe how smooth a spline is at the meeting points of parametric pieces a C^n continuity property is used. If a spline is said to be C^n *continuous*, the first n derivatives of each sequential parametric pieces are continuous throughout. As seen in Figure 33, with increasing n, the spline gets smoother.

Empirical testing showed that a suitable continuity level for a road is C^1, which ensures that meeting road segments merge realistically and is quicker to compute when compared to higher levels of continuity.

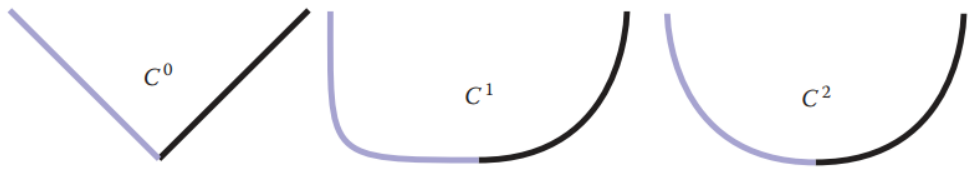

Figure 33. 3 splines with 2 parametric pieces (blue and black) with various C^n continuities at parametric piece meeting points[MS15].

Some special types of curves have *control points*, which can be moved to alter the shape of the curve. For example, a Bézier curve uses a number of control points which when moved change the curve (Figure 34). A Bézier curve with 4 control points p_0, p_1, p_2, p_3 is called a cubic Bézier curve and is only guaranteed to pass through its endpoints p_0 and p_3.

When generating an arbitrary road segment in the devised algorithm, commonly only 4 fixed sequential points v_0, v_1, v_2, v_3 on the road are known and a curve has to be generated between the middle points v_1 and v_2. The curve must be generated so that when the two other curves between v_0v_1 and v_2v_3 have also been generated using their respective 4 sequential road points, the resulting three curves between v_0v_1, v_1v_2 and v_2v_3 form a C^1 continuous spline.

[6]http://www.e-cartouche.ch/content_reg/cartouche/graphics/en/html/Curves_learningObject2.html

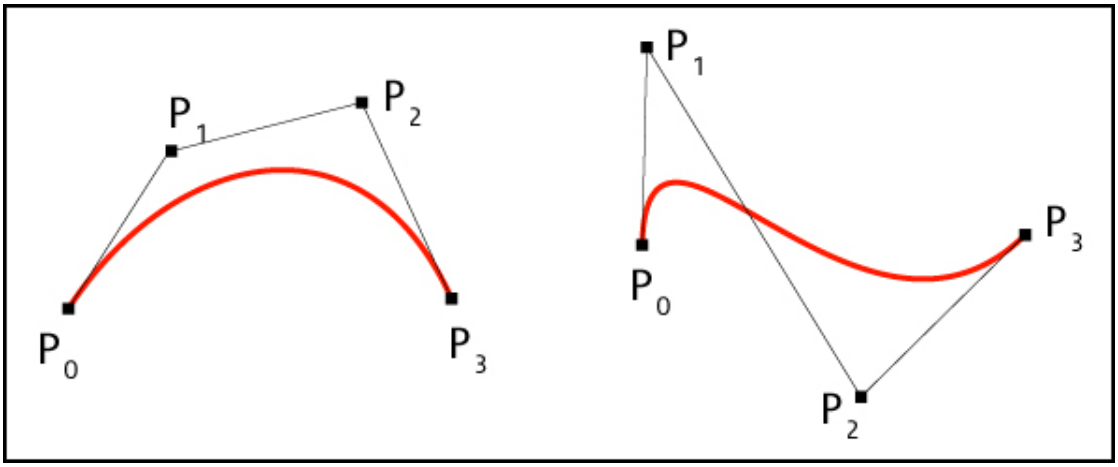

Figure 34. Two cubic Bézier curves with different control points[6].

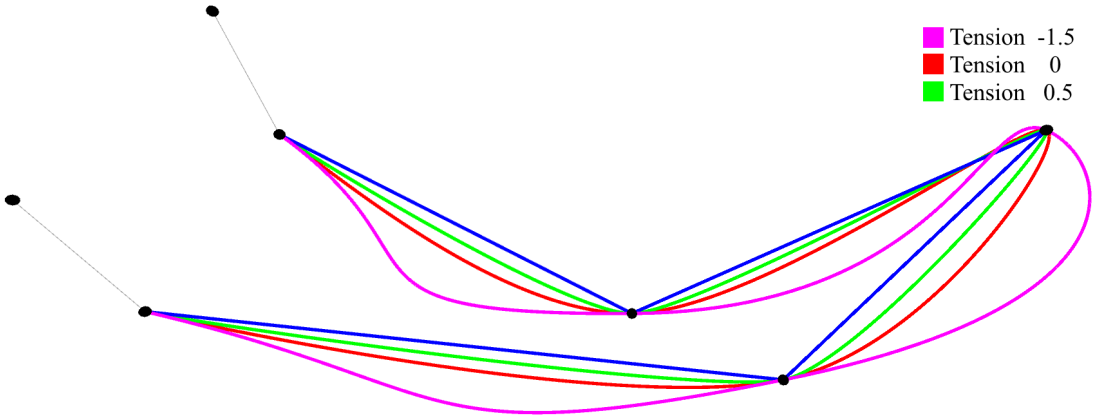

Figure 35. Four cardinal splines with 7 control points and different tensions. Tension 1 creates a linear spline, which is not C^1 continuous, while the other splines are.

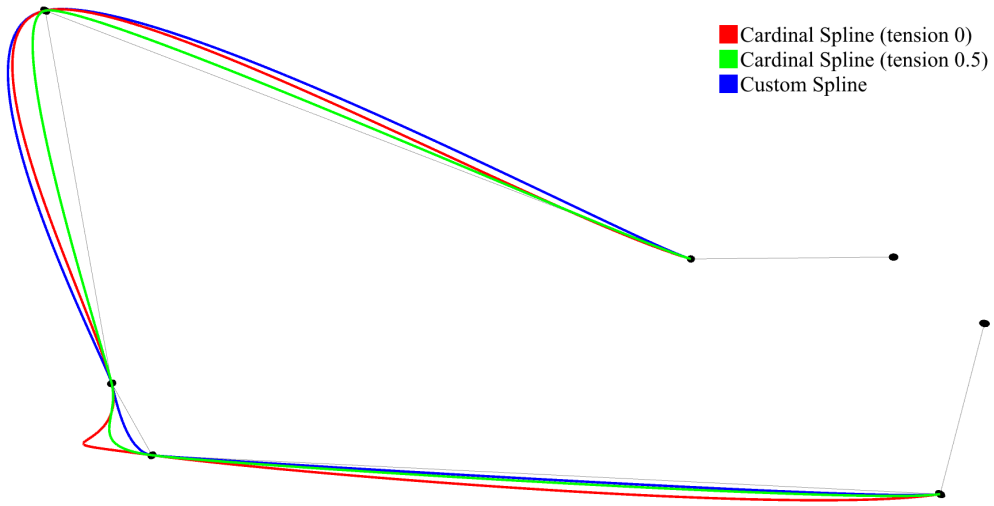

Figure 36. Red, green: 2 cardinal splines that bulge in the bottom left corner due to large variation in control point distances. Blue: a custom spline that avoids these bulges.

Next, 2 ways to generate a curve between v_1 and v_2 are discussed. To generate one such curve using these points, a cardinal spline [MS15] construction can be used. A *cardinal spline* uses n control points to generate $n-2$ parametric pieces, with the first and last control point pair ignored and is guaranteed to pass through all the control points except the first and the last. In current case, this would mean that with 4 control points a single parametric piece would be calculated connecting the middle points v_1 and v_2. A cardinal curve also has a parameter called tension, which controls how much the curve can deviate from a simple linear curve between control points (Figure 35). cardinal splines with $tension \neq 1$ are C^1 continuous. Although cardinal curves seem to be suitable for the roads, due to the way they are constructed, undesirable results can happen in some cases (Figure 36) where the spline bulges too much. For this reason, a different type of curve construction was developed. The *custom road curve* is generally similar to a cardinal curve but generates some additional control points around pre-existing road points v_1 and v_2. More specifically, 4 points p_0, p_1, p_2, p_3 are derived from the existing points, which are then used to construct a cubic Bézier curve. Since a Bézier curve

is guaranteed to only pass through its endpoints, v_1 and v_2 are assigned to p_0 and p_3 respectively. The additional points p_1 and p_2 are calculated by offsetting the points v_1 and v_2 based on the following formula. First, the neighbouring points of v_1 and v_2 are taken and unit vectors $\hat{u}_{v1}$ and $\hat{u}_{v2}$ parallel to the neighbours found:

$$\hat{u}_{v1} = \frac{v_2 - v_0}{|v_2 - v_0|}, \qquad \hat{u}_{v2} = \frac{v_1 - v_3}{|v_1 - v_3|}.$$

Secondly, the offset distance d is found by taking the shorter edge from edges v_0v_2 and v_1v_4 and multiplying its length by 0.3:

$$d = min(|v_2 - v_0|, |v_1 - v_3|) \cdot 0.3.$$

The points p_2 and p_3 are derived by offsetting points v_2 and v_3 in the previously found direction by the given amount:

$$p_2 = v_1 + \hat{u}_{v1} \cdot d, \qquad p_3 = v_2 + \hat{u}_{v2} \cdot d.$$

Lastly, the points are used to construct a cubic Bézier curve (Figure 37) by evaluating $t \in [0, 1]$ at uniform steps using the curve function:

$$\begin{aligned} g(t) =& p_0 \cdot (1-t)^3 \\ &+ p_1 \cdot 3 \cdot (1-t)^2 \cdot t \\ &+ p_2 \cdot 3 \cdot (1-t) \cdot t^2 \\ &+ p_3 \cdot t^3. \end{aligned}$$

The spline generated by joining these curves as parametric pieces will also be C^1 continuous like the cardinal spline. When 2 sequential curves u and v are constructed for sequential 5 road points $v_0, ..., v_4$, 4 points will be calculated for each: p_0, p_1, p_2, p_3 and q_0, q_1, q_2, q_3 respectively (see Figure 38). These points have the following properties:

- the last point p_3 of the curve u is the same as the first point q_0 of the curve v;
- points $p_3(= q_0); p_2$ and q_1 are collinear;
- the distance between p_3 and p_2 is the same as the distance between p_3 and q_2.

These properties are the same constraints imposed by Stärk for a C^1 continuous Bézier curve construction [[Stä76] via [PBP13]], which is proven to be C^1 smooth.

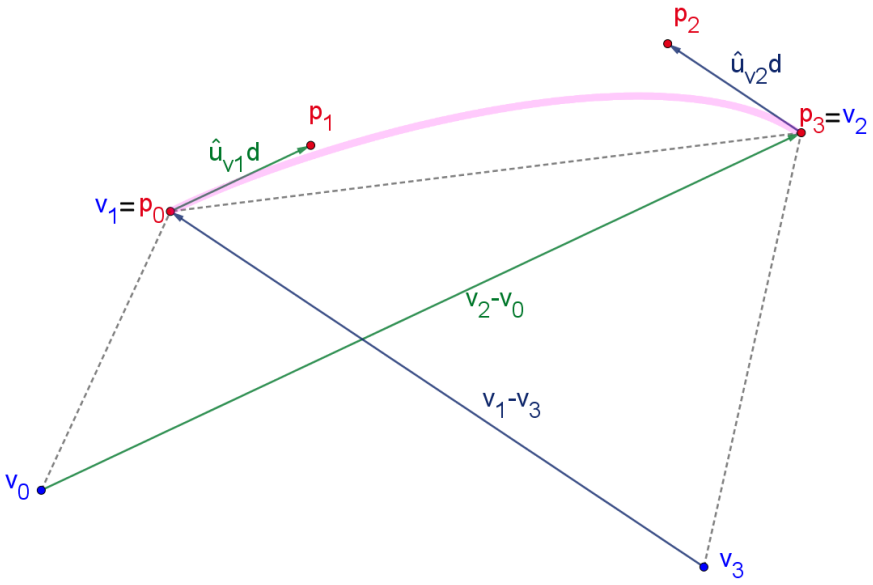

Figure 37. A Bézier curve (pink) generated by using the points p_0, p_1, p_2, p_3 derived from 4 fixed road points v_0, v_1, v_2, v_3.

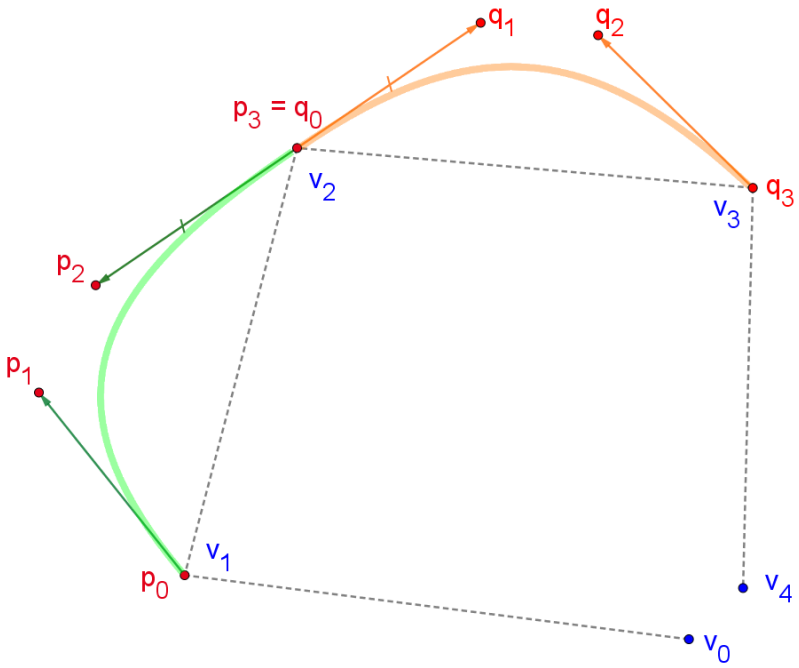

Figure 38. A C^1 continuous Bézier spline (green and orange) generated through 5 road points $v_0, ..., v_4$.

3.8 Boolean Operations on Polygons

Several parts of the described algorithm require the manipulation of various polygons. For example, an intersection operation is required to determine the intersection of a road polygon (see Chapter 4.2) and an union operation is required to calculate the full outline of a city (see Chapter 4.3). A suitable algorithm is the Vatti's clipping algorithm [Vat92], which implements 4 boolean operations on any number of arbitrary subject polygons with any number of arbitrary clip polygons. The 4 operations are intersection, difference, union and xor (Figure 39). As an alternative, Greiner-Hormann clipping algorithm [GH98] could be used, which can perform better than the Vatti clipping algorithm. The shortcoming of Greiner-Hormann clipping algorithm is that it can not handle common edges or intersections exactly at vertices. Their paper suggests perturbing the vertices to remove these degeneracies. However, as these special conditions occur often in this algorithm and would require specific handling, Vatti's clipping algorithm is used instead as it is fast enough for this application. The implementation uses the Clipper [7] library, which extends the original Vatti's algorithm and also implements polygon offsetting based on the discussion by Chen and McMains [CM05].

This concludes all the preliminary algorithms used in the devised algorithm. Now the thesis proceeds with describing the actual world generation algorithm.

[7] http://www.angusj.com/delphi/clipper.php

[8] http://www.angusj.com/delphi/clipper.php

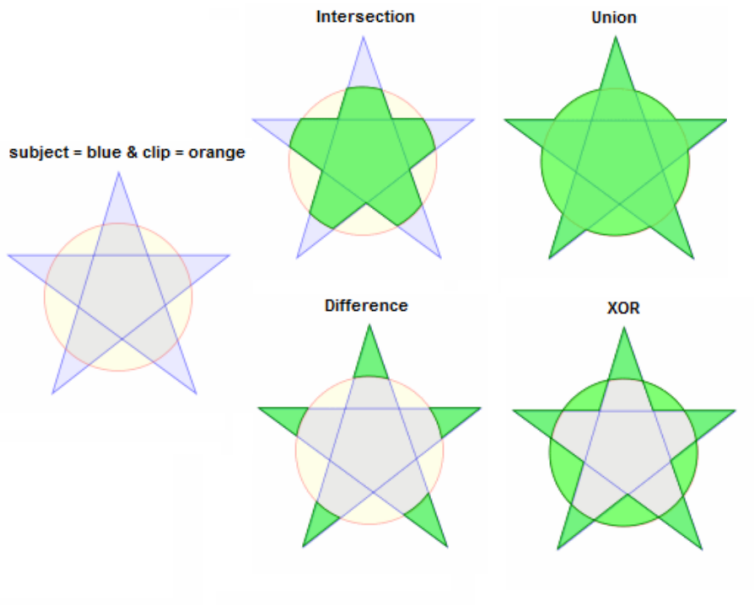

Figure 39. The 4 boolean operations on polygons using Vatti's clipping algorithm[8].

4 Macrochunks

Macrochunks are the largest chunks with an area of about $8 \times 8\,\text{km}^2$. They are used to generate large connected features that cover large areas and distances in real life, for example highways and cities. Macrochunks only generate basic information about cities, such as their outlines and positions in the world. Macrochunks are mainly used as data containers by mesochunks. Mesochunks query data about their parent macrochunk features for the generation of mesochunk features, such as city internal structure, villages, rural roads and agricultural land. Every macrochunk also generates a low resolution heightmap, which is used to generate and render low resolution terrain far away from the camera to increase the render distance. More specifically, macrochunks generate the following:

1. low resolution terrain (chapter 4.1);
2. highways that overlap with the chunk (chapter 4.2);
3. the centers and outlines of any cities that overlap with the chunk (chapter 4.3).

4.1 Terrain Generation

Each macrochunk generates a *heightmap* using simplex noise and builds a mesh for it. Since these meshes are only visible from far away, they can have a relatively low level of detail. The noise used for the terrain has 4 octaves, a base frequency of $\frac{1}{16384}$, lacunarity 2 and persistence 0.5. The generated noise value in range $[0, 1)$ is then multiplied with 1024, defining the maximum variance in elevation in meters. The noise is sampled at 64 meter intervals and a texture is chosen for each vertex. First, a check is made if the vertex is inside a city (discussed further in chapter 4.3), which would mean a concrete-like texture is chosen. If the vertex is not inside a city, the type of land there is determined (discussed further in chapter 5.2), based on which either a grass or a brown soil texture is chosen. Description of how the type of land is determined with the use of another simplex noise function is explained further in the upcoming chapter 5.2. This results in a landscape as seen in Figure 40. The terrain is also offset by 10 meters downwards so when later mesochunks are generated on top of it, the macrochunk terrain will not intersect with mesochunks' geometry. As the transition point from macrochunks mesh to mesochunks mesh is far away from the camera, the 10 meter difference in their elevation is unnoticeable.

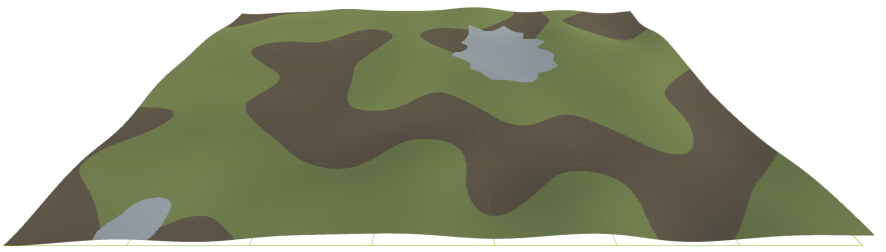

Figure 40. Simplex noise heightmap based textured terrain generated for macrochunks.

4.2 Highway Generation

The macrochunk generation algorithm begins by generating *highways*. In current context, highways are a type of paved major roads that have two 3 meters wide lanes in both directions and are used for high throughput traffic over large distances. Highways also have an 8 meters wide margin on both sides to enable better visibility and clearing for roadside infrastructure.

4.2.1 Graph Generation

The process starts with generating an Euclidean graph[9], where each vertex, called a *highway node*, represents a *highway junction* and each edge corresponds to a *highway segment* between the two end vertices. The later generated highway segments do not follow the precise Euclidean path of the graph edges, although they are guaranteed to pass through the end vertices of the edge. Instead, for the final highways, each graph edge is processed and a C^1 continuous spline is generated, representing the final curvature of the highway segment with a more natural look.

To generate an ideal Euclidean graph for highways on a specific macrochunk, the following conditions should be met:

1. If the macrochunk contains a highway going out of the chunk, the neighbouring chunk must also contain the same highway and vice versa;
2. The graph must avoid singular disconnected edges and be reasonably connected;
3. The graph must not be too dense to avoid unnatural-looking short cycles.

One method to generate a graph suitable for real-time use that adheres to these 3 conditions in nearly all the cases works by generating a small graph for each macrochunk that overlaps and matches the neighbouring macrochunks' graphs. To do this, first a number of deterministically calculated highway nodes are found in the vicinity of the

[9] http://mathworld.wolfram.com/EuclideanGraph.html

generating macrochunk. The vicinity that will be considered for the generation of a specific macrochunk is bound by a 7×7 macrochunk window around the generating macrochunk.

For simplicity's sake, each macrochunk can contain at most 1 highway node. The probability p of a macrochunk containing a highway node is fixed. Empirical testing showed that a good probability value is 0.22, which ensures that the highway nodes are not too far apart and also not too crowded. To determine, if a macrochunk contains a highway node, the chunk's first PRNG sequence value r_1 is mapped to range $[0, 1]$ and compared against the static threshold p. If the value is below the threshold, the location of the highway node on the macrochunk is determined with the next 2 PRNG sequence values r_2 and r_3. They are used to determine the local x- and y-axis coordinates on the macrochunk respectively. By using this approach, the positions of all the highway nodes in any 7×7 macrochunk window can be cheaply evaluated by using the respective chunks' PRNG sequences. This will not require any of the macrochunks in the window to be previously generated and every macrochunk will always evaluate to the same highway node position if any.

After all the highway nodes in the current macrochunk window have been determined, it is necessary to determine the highway edges. This is done by first generating a relatively dense Euclidean graph using the highway nodes and then culling some of its edges. The initial graph is constructed by considering all the highway nodes as graph vertices and then adding an edge for every pair of highway nodes, which are at most 2 macrochunks apart from each other on their longer axis. This produces a graph as seen in Figure 41.

Then, the graph edges are culled. As the first step, the graph is traversed and all intersecting edge tuples are removed. This ensures that highway junctions can only appear at the positions of highway nodes. Secondly, all pairs of 2 neighbouring edges are iterated and the angle between them calculated. If the angle between the neighbouring edges is smaller than a fixed threshold, the longer of the 2 edges is removed. Empirical

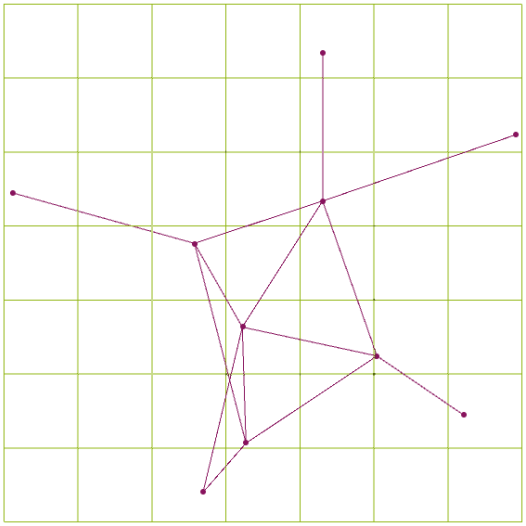

Figure 41. Local highway graph before culling.

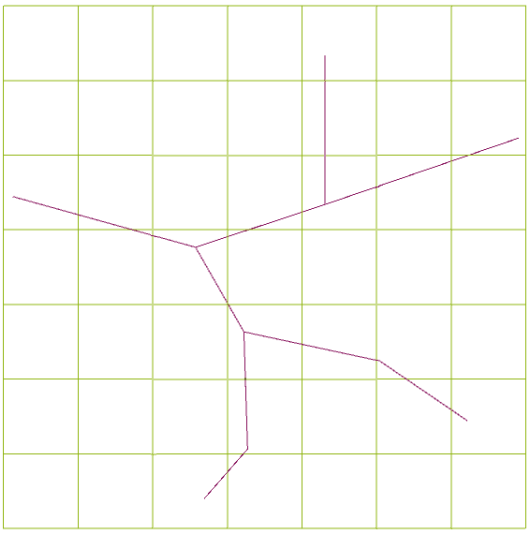

Figure 42. Local highway graph after culling.

testing showed that a good threshold is 58 degrees, which reduces the number of relatively parallel highways, yielding a more realistic road network.

At the end of this process, all the highway nodes and edges in the macrochunk window have been found, yielding a good-looking highway network that is not too sparse or dense, as seen in Figure 42.

It is possible for this approach to breach the first condition of the ideal highway graph conditions. This can happen if an edge is not culled at the boundary of the macrochunk window due to some of its neighbouring highway nodes being outside of the current window. It is then possible for this extra edge to propagate erroneous edges in the graph towards the middle of the window, causing an extra or a missing edge on the central chunk. Ultimately this would mean that a macrochunk could have a specific intersecting edge but its neighbour would not generate it or vice versa. However, empirical testing over very vast areas showed that the probability of this happening is minuscule and not really noticeable, making this algorithm suitable for the fast generation of highway networks.

4.2.2 Spline Generation

Since the edges of the graph are straight lines, they are not suitable for direct use as highways. Instead, a curve for each edge is generated using the technique that was discussed in chapter 3.7. The generated graph covers a large macrochunk window but only the highway curves that the central macrochunk intersects with are needed. Checking the macrochunk bounds and edge intersections to determine for which graph edges a curve has to be generated is not enough, because an edge might not intersect with the macrochunk but the curve generated for it might.. As such, all edges with at least one node inside a 3×3 macrochunk window are considered for a chunk-curve intersection.

The curve generation algorithm described in chapter 3.7 used 4 fixed points on the highway to generate a curve through the middle two of them. These fixed points can be chosen from the highway nodes with some additional considerations. Since each highway node can have more than 2 adjacent highway nodes, a choice of adjacent highway node pairs has to be made through which the highway will be C^1 continuous. All of the curves can not form a C^1 continuous spline with each other, because a shared derivative at the junction would look odd. For every node, all adjacent node pairs are iterated and the pair with the maximal difference in their directions is marked as a continuous pair. If the degree of the highway node is 3 or higher, this process is repeated with the remaining adjacent node pairs to determine any other node pairs that should also form C^1 continuous highways through the given highway node. If a highway node has an odd degree, one adjacent node will simply end up without an opposite adjacent node. The same happens if a highway node only has one edge.

After the C^1 continuous adjacent highway node pairs are determined for each highway node, all the edges within the 3×3 macrochunk window are iterated. For each edge v_2v_3, a check is made to determine if the edge is part of a C^1 continuous adjacent highway node pair at nodes v_2 and v_3. If the edge is part of a C^1 continuous highway at node v_2, the other adjacent node of that C^1 continuous pair is selected as v_1. Similarly, v_4 is

found by checking if the edge is part of a C^1 continuous highway at node v_3. If the edge happens to be the one edge at v_2 or v_3 that did not get an opposite adjacent node due to an odd degree for a C^1 continuous highway, v_1 is set to v_2 or v_4 set to v_3 respectively. This also applies to nodes with a single edge. After doing this, 4 points v_1, v_2, v_3, v_4 have been determined for each edge v_2v_3 and the curve construction from chapter 3.7 can be applied. The result of the spline generation on a small scale can be seen in Figure 43 and a larger highway network can be seen in Figure 44. The curves are stored for each highway edge as a list of points that have been generated at uniform steps of $t \in [0, 1]$ using the Bézier curve function $g(t)$. The points are then connected linearly.

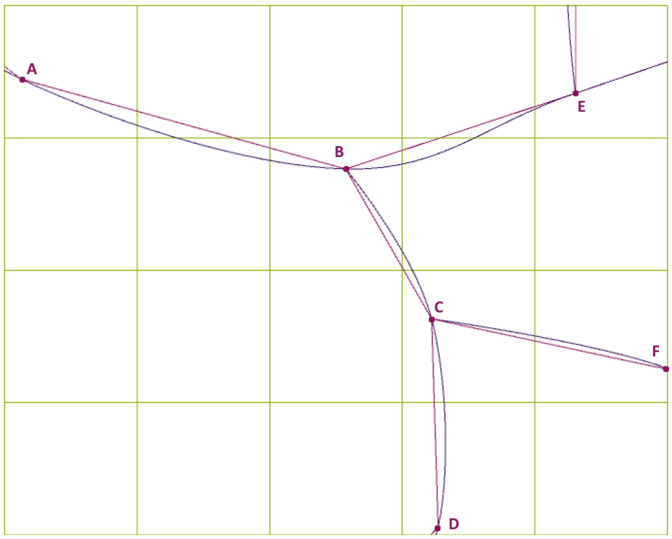

Figure 43. Highway splines generated from the highway graph. In case of highway node C, the curves generated for edges $\overline{BC}$ and $\overline{CD}$ form a C^1 continuous spline, because the highway nodes B and D were marked as a C^1 continuous adjacent node pair for highway node C. In contrast, highway node B did not find an opposite adjacent highway node for node C and the curve for edge $\overline{BC}$ does not have C^1 continuity through node B.

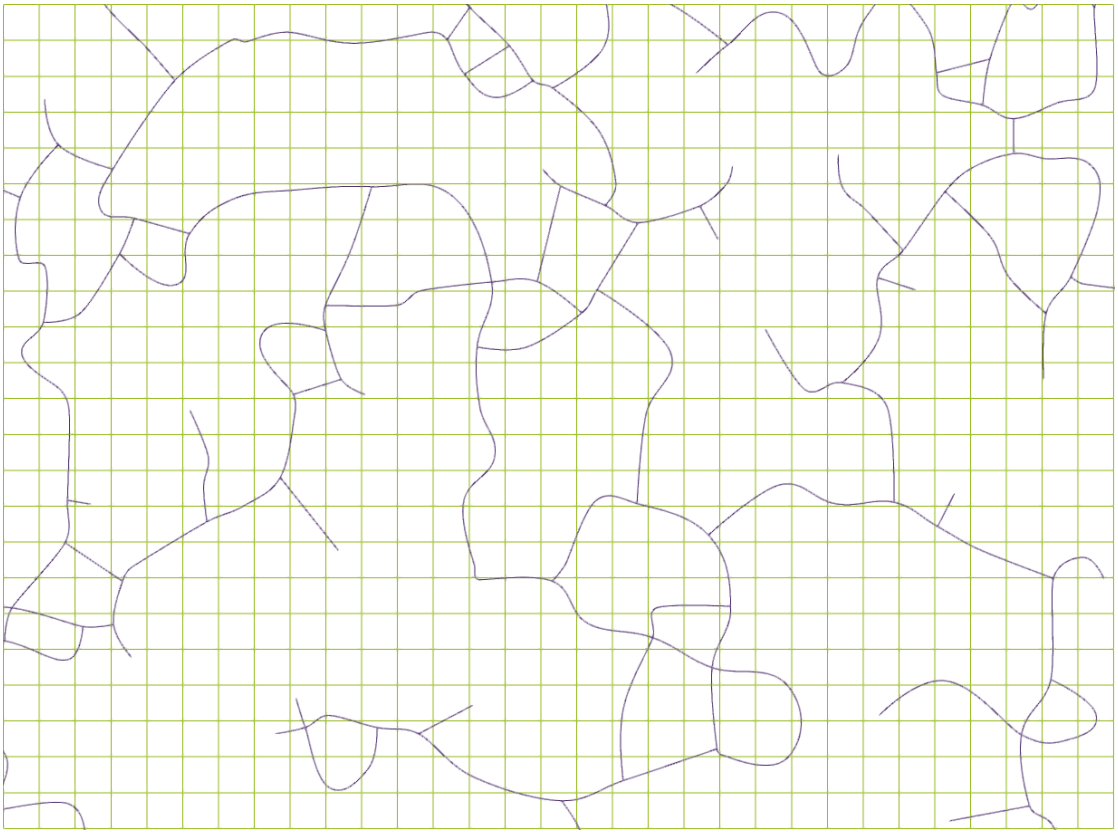

Figure 44. A highway network for 31×23 macrochunks, covering $254 \cdot 188\text{km}^2$ of land.

4.2.3 Highway Dilation

After the curves of each highway edge that intersects with the generating macrochunk have been created, the curves are dilated to form the highways' outlines and their margins' outlines in the form of polygons as seen in Figure 45. These outlines are later used to determine which parts of a chunk are covered by a road and to leave enough space (the margin) for generating the roadside features like traffic signs and utility poles.

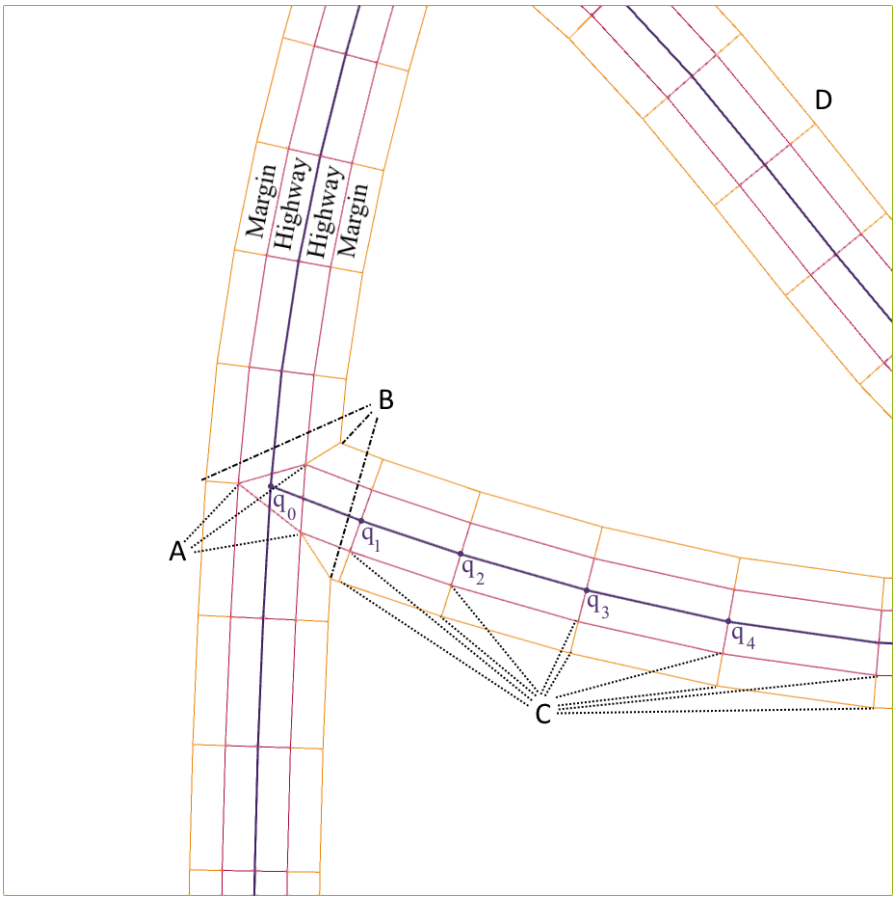

Figure 45. Highway outlines (magenta) and highway margin outlines (orange) calculated for highway curves (blue) for each curve segment. Note that this highway has been $50\times$ upscaled to be better visible on the massive $8192 \cdot 8192$ m^2 macrochunk.

The generation of the dilated outlines begins with a check to determine if the macrochunk contains a highway node. If the macrochunk does contain a highway node, the curves of its edges are sorted in a clockwise order based on the direction of each curve's first segment. That makes it possible to get the neighbouring curve to the left of right of a specific curve. Next, the points surrounding each curve are generated. Each curve was stored as a list of points $q_0, ..., q_{n-1}$ and these points are iterated. The first points to the left and right of the curve (see Figure 45 A) are generated by using the first segment of the curves to the left and right of the current curve. This ensure that neighbouring road segment outlines do not overlap and match each other at points A for highways and B for margins in the figure 45. The locations of the rest of the points (see C in the figure 45) on both sides of the curve from q_1 onwards are calculated by determining a direction perpendicular to the sum of the last and next line segment vectors. For example, for placing points corresponding to curve point q_i, a vector perpendicular to the sum of the segments $q_{i-1}q_i$ and q_iq_{i+1} is found. By the end of processing each curve, most of the needed highway outlines are found. However generating the polygons for highways did not cover the middle area of the junction (the polygon defined by points A in the figure). The polygon for the middle of the junction is constructed by taking the first position to the right of q_0 on each curve forming a list of the needed points.

For the highway curves that intersect with the macrochunk but are not one of the macrochunk's highway node curves (see D in the figure), the approach with a perpendicular vector described in the above paragraph is used. The found polygons are saved with each edge of the highway graph of the macrochunk for later use.

4.3 City Outline and Name Generation

After the positions and shapes of highways have been generated, the algorithm determines if the generating macrochunk will contain any cities. The highway nodes are considered for possible locations of city centers. If a highway node has exactly one or more than three edges, it is considered suitable for a city. The reasoning being that if a highway has a single edge, a city placed there ensures that the highway does not end in the middle of nowhere. Additionally, it is reasonable for a city to be located at the convergence of multiple major roads, such as in the case with more than 3 highways converging.

The maximum city size is limited to 2×2 macrochunks or roughly 16×16 km^2. This ensures that when a 3×3 macrochunk window centered around a generating macrochunk is evaluated, all cities that overlap with the central macrochunk will be found. Therefore, the highway nodes of all the 9 macrochunks in the window are considered for city centers.

After the positions of the city centers in the window are found, the algorithm generates the outline of each of these cities. The outlines are then compared with the central chunk's bounds to find the cities that overlap with the macrochunk. To generate a city's outline, some additional points inside the city, called city subcenters, are generated and an dilated outline encapsulating them is created. The positions of the city subcenters are determined by using the edges of the highway node that represents the main city center. For each edge, the corresponding macrochunk's PRNG sequence values $r_4, ...$ are used to generate a linearly interpolated point on the line between the main city center position and the highway edge's other end vertex. The distance of this interpolated point

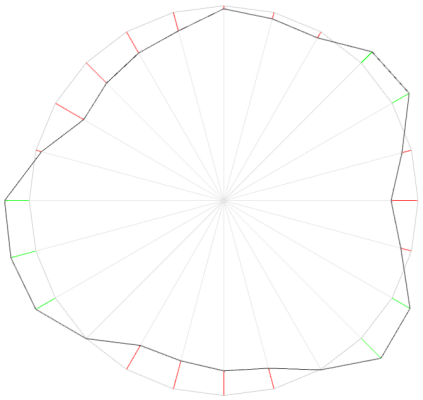

Figure 46. Simplex noise based vertex offsetting for malforming regular polygons.

from the city center is bound in the range $[0.15, 0.2]$, meaning that a city subcenter is placed at most 20% of the edge length away from the main city center. Then the radius of the city subcenter is calculated by taking the distance between the city subcenter and an another point on the edge, that is 40% of the edge length away from the main city center. This ensures that the subcenter radius is between 0.2 and 0.25 of the edge length. After the distances from the city center and radii of every subcenter is found, the smallest subcenter radius is picked and also used as the radius for the main city center.

After the center positions and their radii have been calculated, an regular icositetragon[10] approximation of a circle is generated for each of these positions with their respective radius. Next, the vertices of every icositetragon are offset. A low frequency simplex noise, yielding a pseudorandom offset in the range $[-1, 1)$ for every vetrex. The noise value is then multiplied by a factor of 0.1 of the respective subcenter's highway edge length. The resulting value is then used to offset each vertex on a line through the center position and the vertex's original position as seen in Figure 46. This gives each polygon a more varied shape. The offset is chosen so that if two cities happen to share a highway edge, they can not intersect with one another, because both cities are no more than 0.45 of the shared edge length away from their respective main city centers. The use of low frequency noise ensures that neighbouring vertices have a similar noise value and therefore form a less spiky polygon than regular pseudorandom offsets.

After all the polygons have been calculated, they are combined into one large polygon using a polygon union operation. This polygon defines the final outline of the city as seen in Figure 47. After all the 9 macrochunks in the window of interest have been evaluated for cities, the final city outline polygons are checked for intersection with the generating macrochunk's bounds and stored in the macrochunk for later use.

As the very last step, a pseudorandom name is generated for the city using lists of common prefixes and suffixes. The algorithm generates localized names that seem like

[10]https://en.wikipedia.org/wiki/Icositetragon

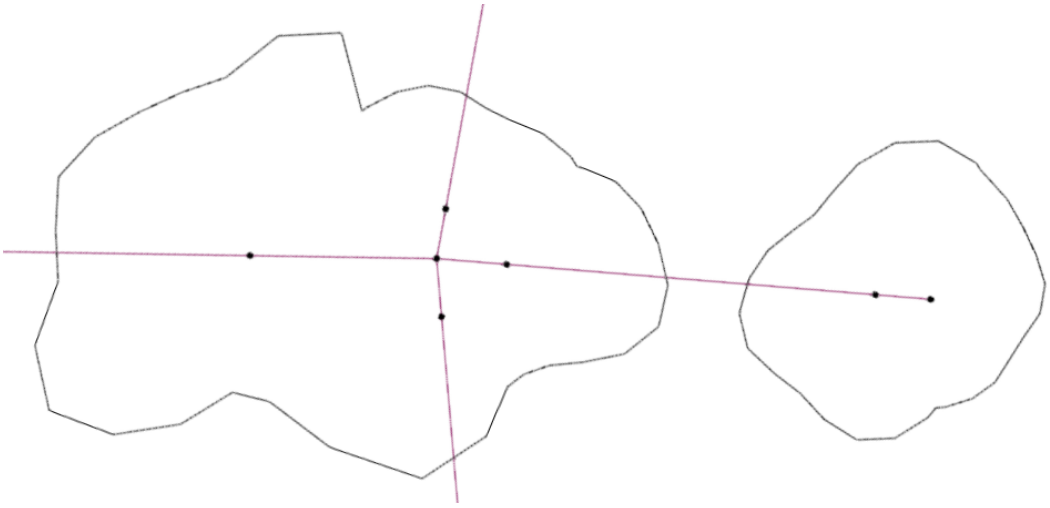

Figure 47. Highway edges (purple), city centers (black dots) and outlines (black lines) of 2 neighbouring cities.

plausible real places in Estonia, but could be changed to generate names for any country with ease. There are 2 types of names generated. The first type uses combined names, where an actual word is chosen as a prefix and suffixed with another actual word. This generates names like *Meresalu*, *Paunsoo* and *Kurelaane*. The other type of names uses partial phrases which are also combined to create the final name. Possible generated names include names like *Kurnja*, *Kabaste* and *Räpsa*. The full list of possible name prefixes and suffixes is available in appendix II. To determine the name of a city, the hash of the city center's position is taken and the the remainder of the hash after division by 100 taken. For the hash function, the hash function from Chapter 3.3 is reused. As city centers are placed pseudorandomly, this will yield a relatively pseudorandom number in the range $[0, 99]$. If the value is less than 80, a long name is generated and otherwise a short name is generated. To determine the suffix, the chunk position is multiplied by 73 and hashed once more. The remainder of the hash after division by the length of either the combined name or short name prefix list is then used as an index for the suffix selection. The same is then done for the prefix, however the coordinates are instead multiplied by 89 to reduce the possibility of overlapping hashes.

5 Mesochunks

Mesochunks are medium sized chunks with a size of $512 \times 512\,\mathrm{m}^2$. They are used to generate medium-sized and computationally more expensive features that are not necessarily visible to the camera from large distances. These features include higher resolution terrain, side roads, internal city structure, villages, vegetation, traffic signs and utility poles. The generation of a mesochunk assumes that its parent macrochunk has already been generated and can be used to query data about city outlines and highways. More specifically, mesochunks generate the following:

1. side roads that overlap with the mesochunk (subchapter 5.1);
2. intersecting rural areas, villages and their features (subchapter 5.2);
3. city streets, building shapes and classifications (subchapter 5.3);
4. utility pole networks (subchapter 5.4);
5. traffic signs for cities and villages (subchapter 5.5);
6. meshes for the terrain, all types of roads and terrain objects (subchapter 5.6).

5.1 Side Road Generation

Side roads are supplementary roads to highways that can be found throughout civilized areas. In current context, they are paved, have one lane in both directions and are 8 meters wide. Similarly to highways, they have a 6 meter margin on each side for a clearing for placing traffic signs, utility poles and sidewalks.

The general idea behind side road generation is very similar to the one of highway generation as discussed in chapter 4.2. The main difference is that *side road nodes* are generated differently from highway nodes. Side road nodes are generated on a smaller scale and have some additional constraints on their placement.

Similarly to highways, a 7×7 mesochunk window is fixed around a generating mesochunk. It is possible that some of these mesochunk positions in the window are

contained on a different macrochunk that does not exist yet. As the process of determining the position of side road nodes requires knowledge about intersecting highways and cities, it is therefore necessary to also generate any missing macrochunks for all the mesochunks inside the current window before proceeding.

First, the first values of the PRNG output sequence of each mesochunk in the window is used to determine which mesochunks contain a side road node and the positions of the side road nodes within the corresponding mesochunks. While highway generation used the probability 0.22 for a macrochunk to contain a highway node, mesochunks use a slightly higher probability of 0.25. This makes the side road network slightly more dense, as there are more supporting roads than major roads in the real world. However, if the mesochunk happens to intersect with a city, the probability threshold is ignored and the mesochunk is guaranteed to contain a side road node. This means that near and inside cities the side road network is even denser due to the large amount of people living in the area.

After it has been determined if a mesochunk contains a side road node, the node's position is calculated. First, a check is made to detect if the mesochunk overlaps with any highways. If the mesochunk does overlap with a highway, the overlapping segment of the highway is found and its midpoint marked as the position of the side road node. This ensures that side road nodes stick to nearby highways if one is present. If, however, a mesochunk does not overlap a highway, another check is made to determine if the mesochunk overlaps with a city. If the mesochunk does not overlap with a city, the previous procedure of using the chunk's PRNG output to determine the side road node position is used. In case the mesochunk does overlap with a city, two possibilities for the side road node placement are considered. When looking at the layouts of real cities, commonly 2 different patterns emerge. Some cities have naturally evolved and expanded over time, with less options for uniform urban planning because of historic preservation and various other reasons as seen in Figure 48. However, some cities have instead been

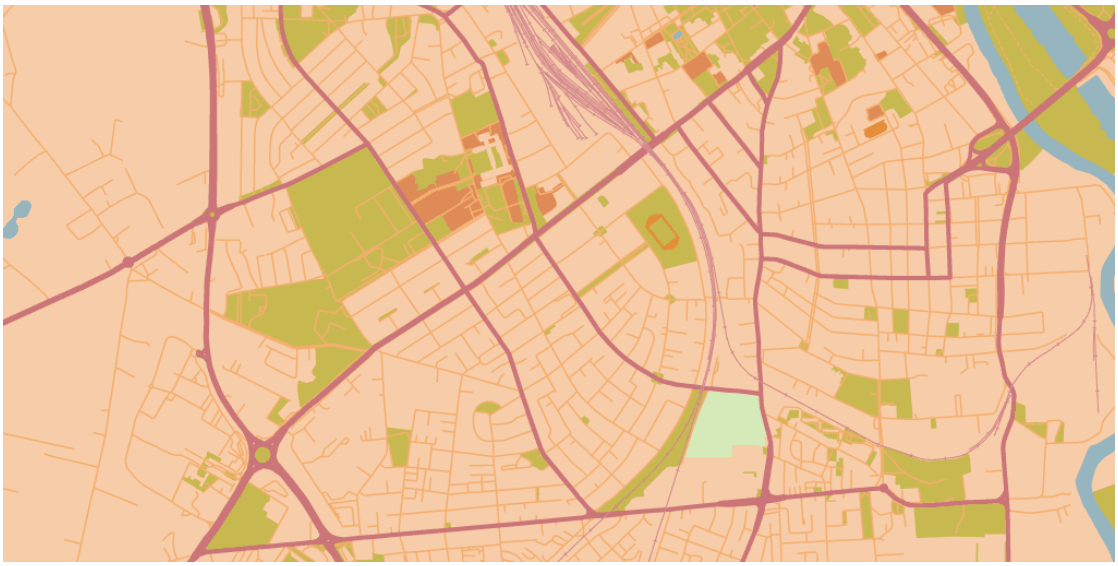

Figure 48. Partial street layout of Tartu, Estonia. Image taken from *Google Maps*.

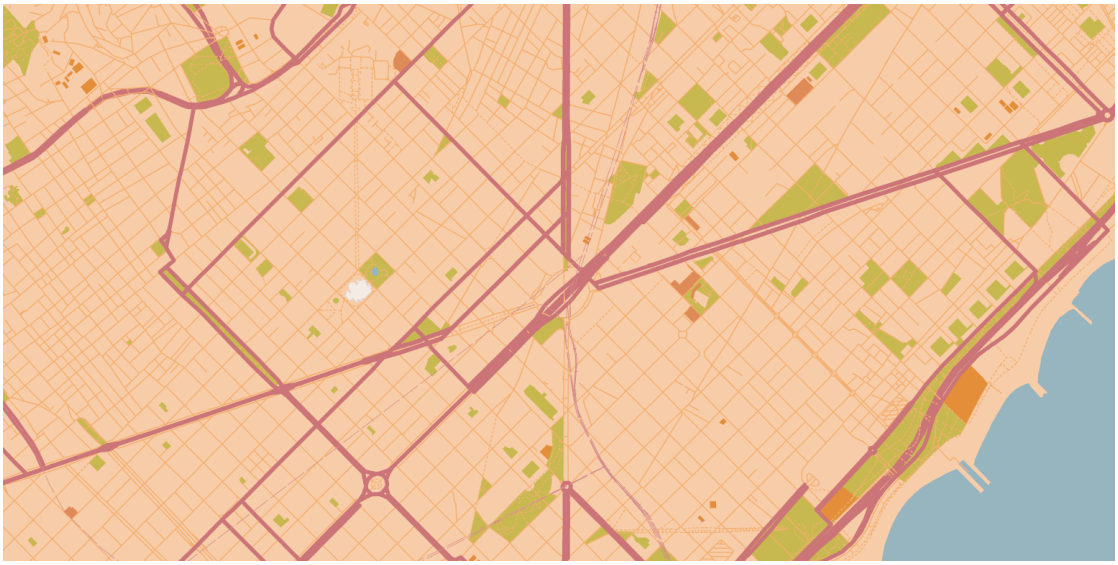

Figure 49. Partial street layout of Barcelona, Spain. Image taken from *Google Maps*.

planned more in advance with some neighbourhoods or even whole cities following a strict grid pattern for more optimal land use as seen in Figure 49. For the algorithm to generate both of these types of street layouts, a simple approach can be taken. The center of each mesochunk containing a city can be taken and used to calculate a low frequency simplex noise value in the range $[0, 1)$. If the value is less than 0.5, a more random layout is chosen and if not, a more uniform layout is chosen. If the random layout is chosen, the previous procedure of using the chunk's PRNG output is used. If the uniform layout is chosen, the side road node is simply placed in the middle of the mesochunk. Since low frequency noise is used, neighbouring mesochunks are likely to have a noise value in a similar range. This enables some areas of cities to have a more random layout while other areas have a more planned, grid-like look.

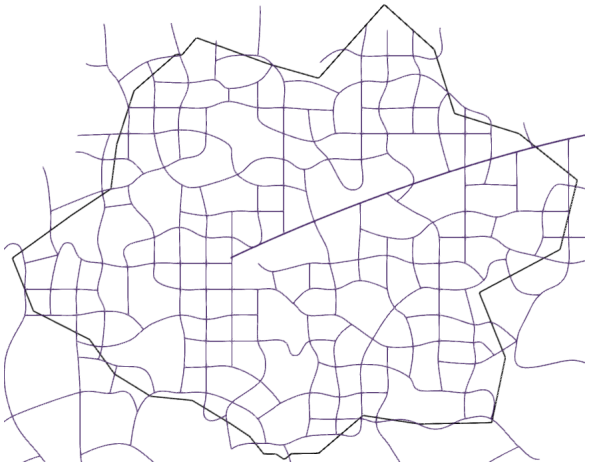

Figure 50. Side roads generated inside a city (black) and one highway (the thicker line). Some regions of the city use the more uniform side node placement while others use a more random placement.

Figure 51. Rural side roads and one highway (the thicker line).

After the positions of the side road nodes have been determined in the window, an identical process to highway node processing is applied. A number of edges is added to

an Euclidean graph between every side road node pair that is no more than 2 mesochunks apart on their longer axis. Next, all intersecting edges and neighbouring edges with an inner angle below a threshold are culled. A side road network generated for a city can be seen in Figure 50 and a side road network for a rural area in Figure 51.

5.2 Countryside Generation

This chapter gives an overview of the methods used to populate the areas outside of cities with various natural and man-made features. The algorithm works by subdividing the land into large polygons, called regions. Each region is assigned a general type and some are then further subdivided into smaller, more specialized regions.

The algorithm generates one large region per mesochunk. To generate this region, a point is fixed on the generating mesochunk that will serve as the center of the region. If the mesochunk contains a side road node, the node's position is selected as the center. Otherwise, the mesochunk's PRNG output sequence values r_{10} and r_{11} are used to determine the point on the mesochunk at random. Then additional 8 points are found for the 8 immediate neighbours of the central mesochunk in the same fashion. The 9 points are used to calculate a Voronoi cell [Aur91] around the central mesochunk's region center. The Voronoi cell is then considered as the outline of the region. This process can be repeated for any mesochunk and due to the deterministic choice of points on each mesochunk, neighbouring Voronoi cells will share the same edges.

Before proceeding, the algorithm checks if the region overlaps with a city. If the region is completely covered by a city, the city takes priority and the region is dropped from subsequent generation. If the region is only partially covered, the segment covered by the city is subtracted from it and the algorithm proceeds with a smaller portion of the initial Voronoi cell.

Next, each region is assigned a general type. The 4 possible types are *village*, *nature preserve*, *forestry* and *cultivation*. First, it is determined if the region will be a village. If

the mesochunk contained a side road node, the number of its edges is considered. If the side road node has one edge, the region is marked as a village to make sure side roads do not end in the middle of nowhere. Additionally, if the side road node has more than 2 edges, the chunk's PRNG output sequence normalized value $r_{12} \in [0, 1]$ is compared against a static threshold of 0.35. If the value is below the threshold, the region is also marked as a village. If the region was not marked as a village, a low frequency simplex noise value at the position of the region center is sampled and used to determine the type as seen in Table 1.

Table 1. The distribution of the types of the large regions based on noise value.

Noise Range	**Region Type**	**Region Description**
$[0, 0.43)$	Cultivation	Further subdivided land consisting mostly of farms
$[0.43, 0.69)$	Forestry	Further subdivided land consisting mostly of forests
$[0.69, 1)$	Nature reserve	Land consisting of undisturbed nature

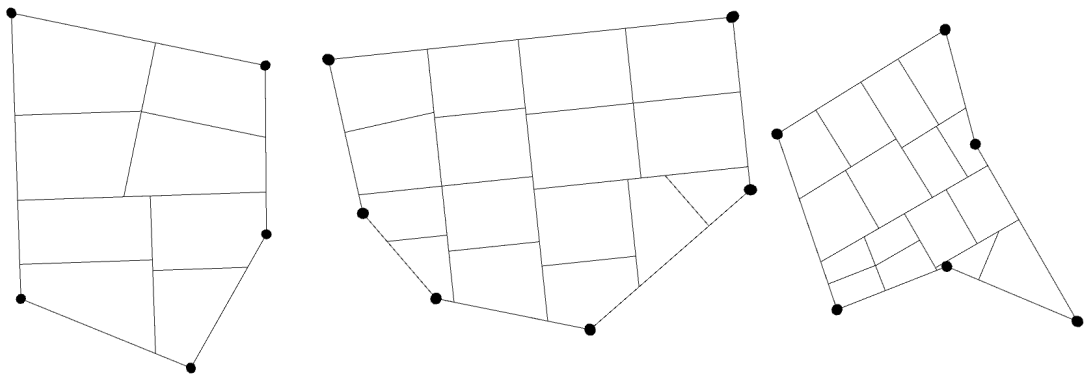

Figure 52. Subdivided region polygons.

After the region outline and the type has been determined for a generating mesochunk, the algorithm proceeds with generating the internal structure of each region. First, all the regions are subdivided into smaller subregions. The regions are subdivided via

a recursive algorithm that divides a polygon into multiple sections as can be seen in Figure 52. Starting with the original outline of the region, the following procedure is applied:

1. Pick the longest edge e of the polygon;
2. Calculate the centroid c of the polygon;
3. Calculate the orthogonal projection c' of point c on line e;
4. Cut the polygon with a line formed through points c and c';
5. Repeat from step 1 for each new polygon if the new polygon's area is above a threshold and maximum depth is not reached.

Next, the generation of cultivation, forestry and nature reserve regions are further discussed.

5.2.1 Cultivation Generation

Subregions of large regions marked for cultivation are assigned a type with the probabilities shown in Table 2. To determine the type, sequential values of the mesochunk's PRNG values are used. After the subregions have been assigned one of these types, each subregion is populated with the corresponding terrain objects. The objects are placed by using the corresponding vegetation placement algorithm, as discussed in chapter 3.4. The generated farmlands can be seen in Figure 53.

Table 2. Cultivation subregion types and their probabilities.

Probability	**Subregion Type**	**Placement Algorithm**	**Subregion Description**
0.35	Large Crop Farmland	Grid	Farmland with several meters tall crops.
0.35	Small Crop Farmland	Grid	Farmland with small crops.
0.1	Empty Farmland	Poisson-disk	Empty farmland with mostly nothing growing. Contains a few random plants sprouted from last season's seeds.
0.2	Grove	Poisson-disk	Land with a small group of trees growing on it. If near a road, has a 0.2 chance to contain a house with a garden.

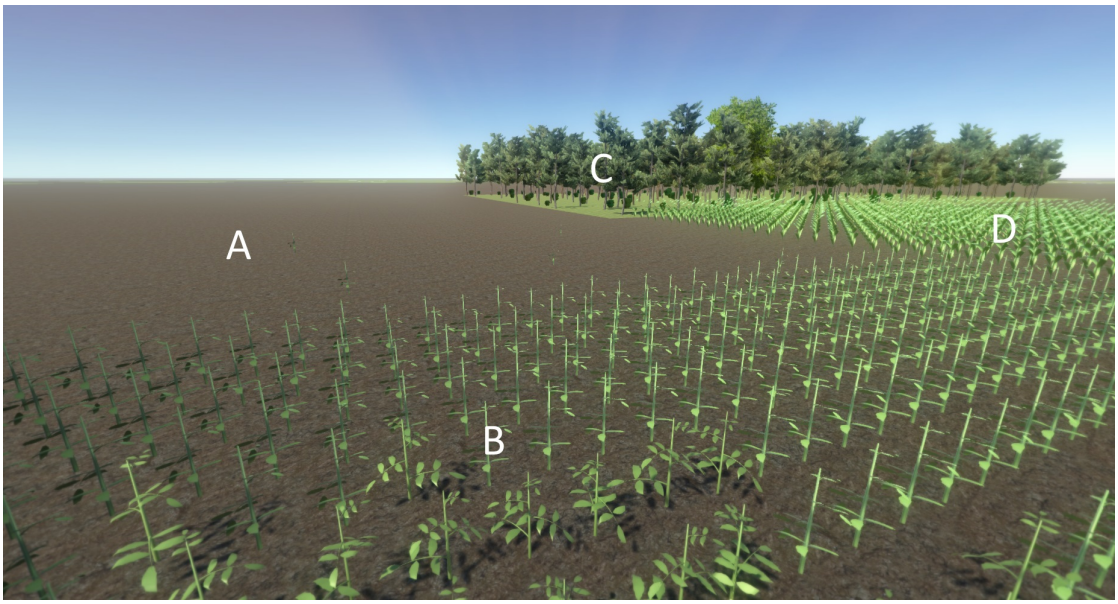

Figure 53. A selection of generated farmlands and a grove. A: an empty farm. B: farmland with small crops. C: A grove. D: farmland with larger crops.

5.2.2 Forestry Generation

The forestry regions are mainly subdivided into different types of forest subregions. The subregions are assigned a type with the probabilities shown in Table 3. Subregions next to a road also have a 0.2 chance to have a rural house with a garden placed in it (Figure 54). After the subregions have been assigned a type, they are populated with objects like cultivation subregions, as seen in Figure 55.

Table 3. Forestry subregion types and their probabilities.

Probability	**Subregion Type**	**Placement Algorithm**	**Subregion Description**
0.15	Coniferous Forest	Grid	A spruce forest with shrubs that has been planted by humans.
0.2	Coniferous Forest	Poisson-disk	A spruce forest with shrubs that has grown naturally.
0.15	Deciduous Forest	Grid	A maple forest with shrubs that has been planted by humans.
0.2	Deciduous Forest	Poisson-disk	A maple forest with shrubs that has grown naturally.
0.2	Mixed Forest	Poisson-disk	A natural forest with all types of trees.
0.1	Sparse Field	Poisson-disk	A field with some trees and shrubs.

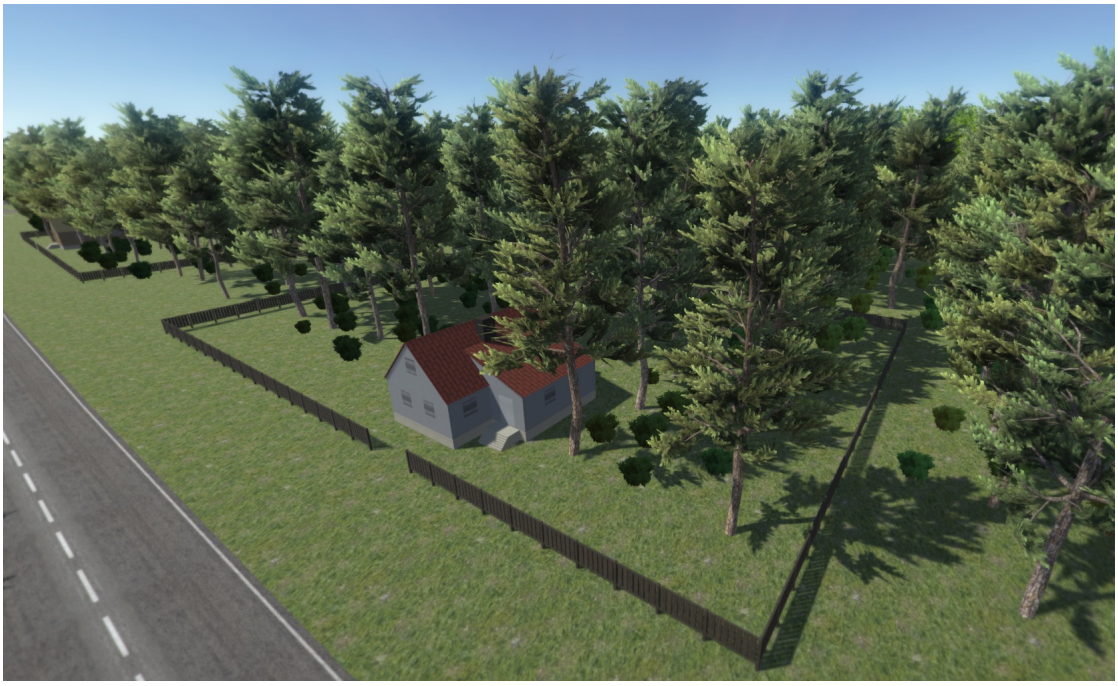

Figure 54. Two houses with gardens generated in a forest next to a road.

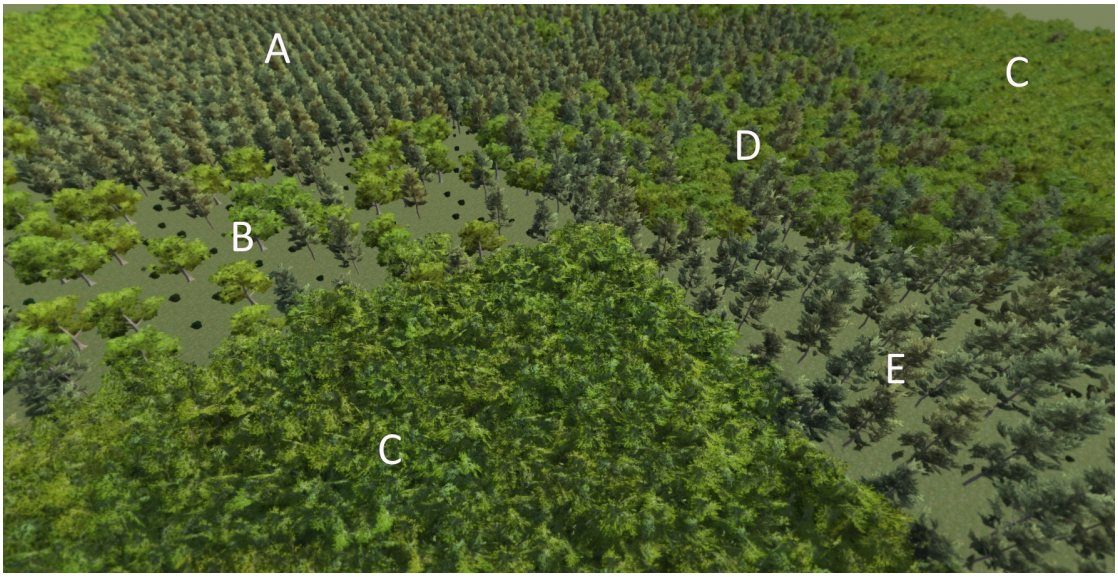

Figure 55. A selection of generated forests. A: a planted coniferous forest. B: a sparse field. C: a natural deciduous forest. D: a mixed forest. E: a natural coniferous forest.

5.2.3 Nature Reserve Generation

The nature reserves are populated using the Poisson-Disk sampling as the vegetation there has grown naturally. The type of the plant is chosen by sampling 2 octaves of low frequency simplex noise. Poisson-disk sampled points with large noise values are populated with mainly trees creating a large mixed forest, medium values with mainly shrubs with occasional trees and low values nearly exclusively with shrubs as seen in Figure 56.

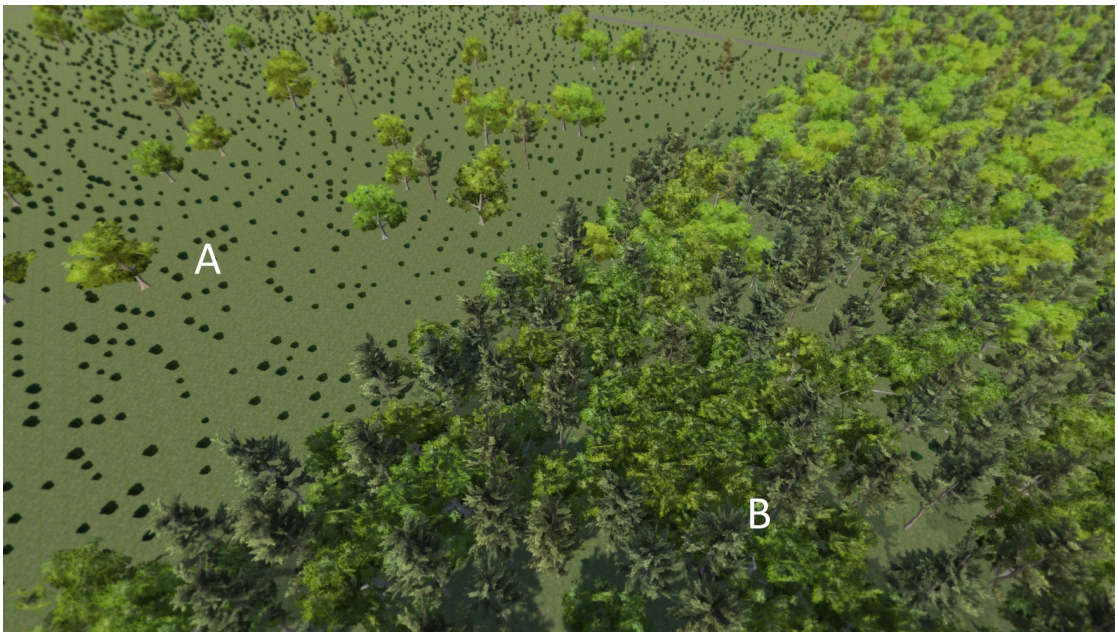

Figure 56. A generated nature reserve. A: a medium density field with shrubs and trees. B: A large mixed forest.

5.2.4 Village Generation

A village is a small settlement with a small number of houses, usually located in rural areas. As described in Chapter 5.2, some of the generated large regions were assigned to contain villages if they contained a side road node and had met additional conditions regarding the number of the node's edges. Each village is also assigned a name using the same naming algorithm as discussed in chapter 4.3. To generate the internal structure of the village, first a number of supplementary unpaved roads within the region are generated. Secondly, the outlines of the plots of land between roads are calculated. Each plot is considered for multiple residences. Then, suitable plots are populated with buildings and vegetation. Unsuitable plots are populated with the features of neighbouring mesochunks to make them blend in with the village surroundings.

To generate the unpaved roads, first the side road node's number of edges is checked. If the number of edges is odd, the edge representing the first not C^1 continuous road is found. Then the first unpaved road is placed from the side road node to village outline based on the direction of the first segment of the found road (Figure 57). If the number of edges is even, the previous step is skipped. This divides the village into an even number of simple polygons with almost convex shapes.

Each of these simple polygons bound by both the roads and the village outline is processed separatedly. Each polygon is subdivided recursively using a similar algorithm to the one discussed for countryside generation with one additional step (5):

1. Pick the longest edge e of the polygon;
2. Calculate the centroid c of the polygon;
3. Calculate the projection c' of point c on line e;
4. Cut the polygon with a line formed through points c and c';
5. *If c or c' lie on a road, mark the new edge formed on the line also as a road;*
6. Repeat from step 1 for each new polygon if the new polygon's area is above a threshold and maximum depth is not reached.

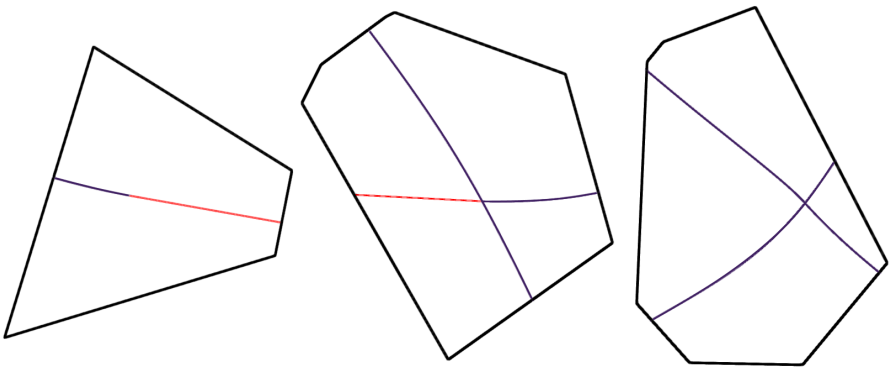

Figure 57. Left and center: First placed unpaved road (red) in the case of uneven number of previously existing side roads (blue) inside a village (black). Right: If there is an even number of side roads, no such unpaved road is placed.

These additional unpaved roads are merged with existing side roads and highways and form the village road network as seen in Figure 58. The polygons between the roads and the village outlines are then calculated and their edges, which are next to a side road or a highway, are offset inwards. This is done so there is a clearing for the later generation and placement of utility poles and traffic signs. The resulting polygons are then subdivided using the region subdivision algorithm that was discussed in chapter 5.2 and each subdivided polygon is tested for a residence. If the subdivided polygon is next to at least one road, the polygon is considered suitable for a residence. An attempt is made to place a house inside the polygon next to the middle of the polygon's longest edge next to a road. If the house placed there would not be bound the polygon, the next longest edge next to a road is considered. If no suitable position for a house is found, the polygon is no longer considered suitable for a residence and is instead filled with the features of the closest countryside region. Currently, one house out of 9 available premodelled houses is chosen at pseudorandom, however as discussed in Chapter 2, a custom country house generation algorithm could be implemented if one so desires. If the house plot happens to be the closest plot to the village center and no church has been placed yet, a church model is placed instead (Figure 60).

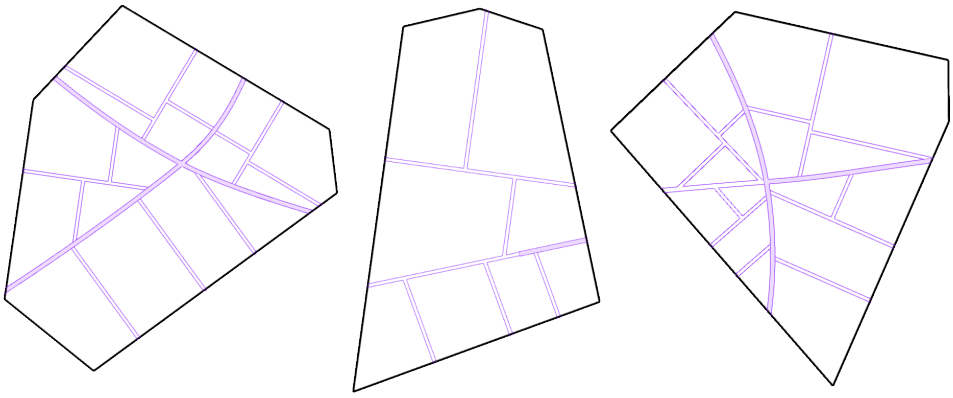

Figure 58. Final road network for a village, consisting of side roads (highlighted) and unpaved roads (outlined).

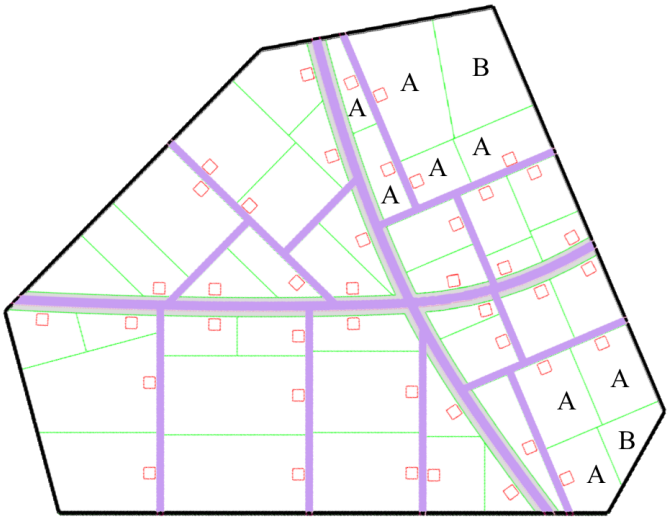

Figure 59. Subdivided polygons considered for residences (green). For most (A), a place for a house (red) is found. If no house is able to be placed (B), the plot is instead filled with the features of the neighbouring mesochunk. Note that the polygons next to side roads (purple)have been offset away using side road margins (grey) to leave space for later placement of traffic signs and utility poles.

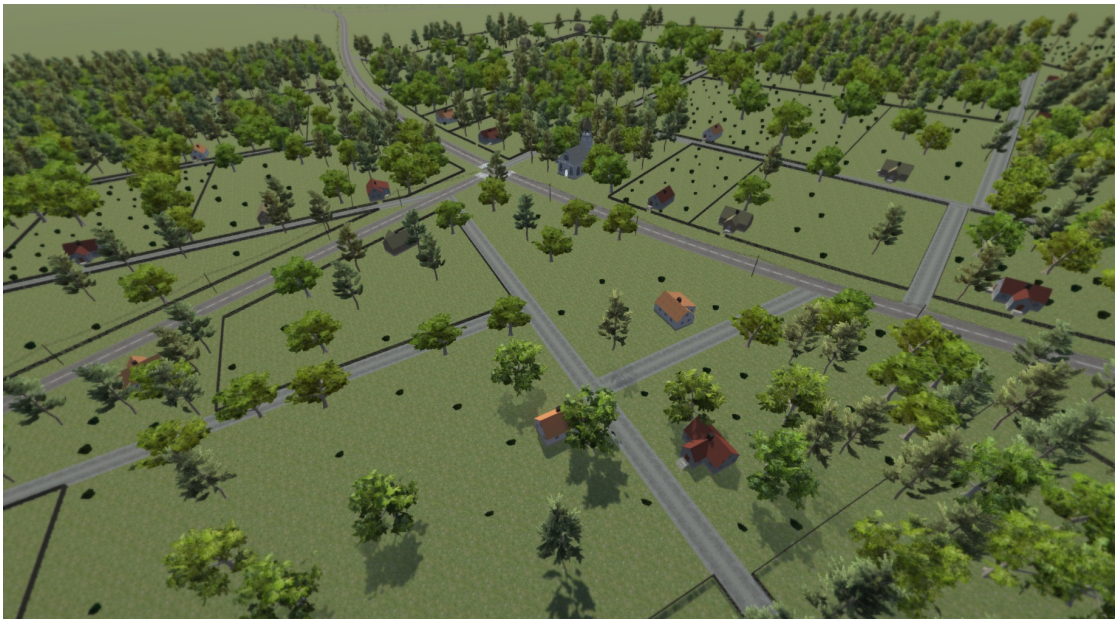

Figure 60. A typical rendered village.

5.3 City Generation

After the side roads that overlap with the generating mesochunk have been generated, the mesochunk generation process moves on to generating the internal structure of any cities that the mesochunk overlaps with. First, an overview of the 3 different approaches for generating street networks are discussed. Secondly, the division of land into plots and the assignment of a a general plot type like suburban, industrial and commercial to each are covered. Thirdly, the description of the used building generation algorithm is described.

5.3.1 Street and Plot Generation

The first approach would generate the whole city at once. As described in chapter 2, many solutions for generating detailed finite procedural cities exist. Since every city generated by the current system has a known finite outline, many of the existing algorithms could be adapted for use in the devised algorithm. However, this is not always desired as generating a large scale city can take too long for real-time use. In addition, this will become a larger problem if the maximal size of the city is changed. Although the algorithm described in this thesis limits the city size to $16 \times 16\,\text{km}^2$, it can be fairly trivially modified to support cities with larger areas, which would increase the time to generate a whole city at once even more. This problem can be alleviated by modifying the chosen finite city generation algorithm when possible to only generate what is necessary for a specific part of the city.

The second approach would generate a city partially, one district at a time. A district would be defined as a plot of land enclosed with side roads, highways and the city outline. Since every mesochunk that overlaps with a city is guaranteed to have a side road node, these districts are relatively small, usually covering 2×2 mesochunks as seen in Figure 61. Each district could then be independently filled with additional streets and buildings as seen in Citygen [KM07] for example. However, some districts cover an

area larger than the currently generating mesochunk has accurate side road data about. For example, one such district is the highlighted district in Figure 61, where the bottom left mesochunk (A) would not have enough information to calculate the whole district outline without generating additional side road data in the upper right area (B). This makes determining the outlines of some districts difficult by requiring one of the two approaches. First, additional side road information could be generated in the direction that the district of interest is extending towards. Second, every too large district could be cut into smaller pieces so each mesochunk's generated side road network is sufficient to determine all the city districts within the mesochunk.

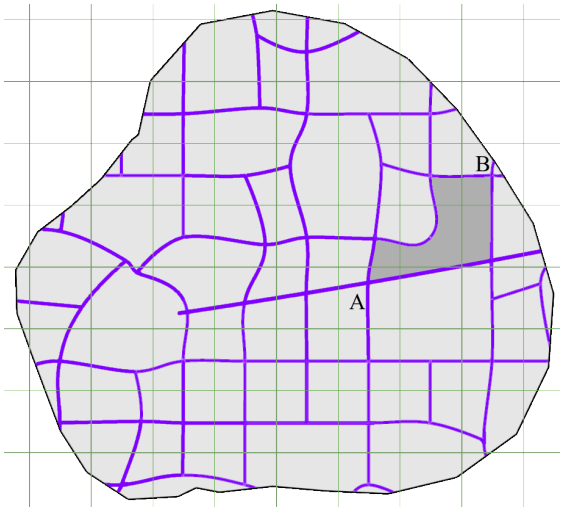

Figure 61. Districts (grey) generated by the second approach between highways and side roads (purple) with one district highlighted. Size of mesochunks grid (green) for scale.

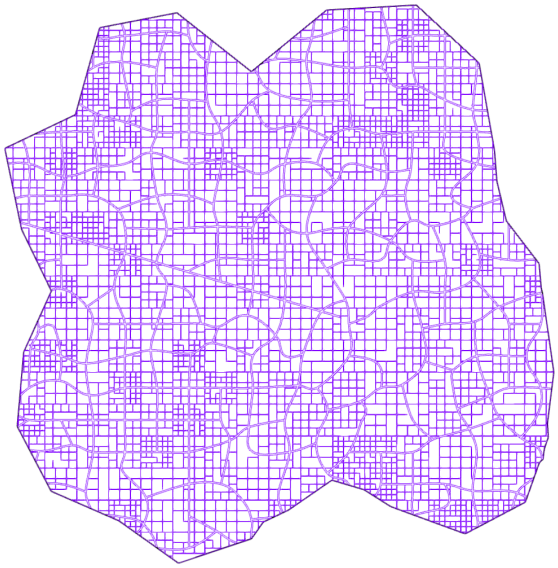

Figure 62. City street network (purple) generated by the third approach.

Since city generation has been thoroughly covered in previous research as discussed in Chapter 2, a simpler third approach was taken to generate the internal structure of the city and focus on the generation of other features. The devised approach overlays

an axis-aligned grid over the side roads and highways within the city one mesochunk at a time and places city streets on some of these grid lines. To add some variance, the grid size varies between mesochunks, with some having streets every 86, 128 or 172 meters. Each grid edge also has a probability of 0.1 to not be placed at all, making some plots of land longer and wider. To reduce odd street placement near side road junctions and highway junctions, all placed streets that are too parallel to a nearby road are also removed. This results in a street network as seen in Figure 62, which can then be filled with buildings for each plot between streets.

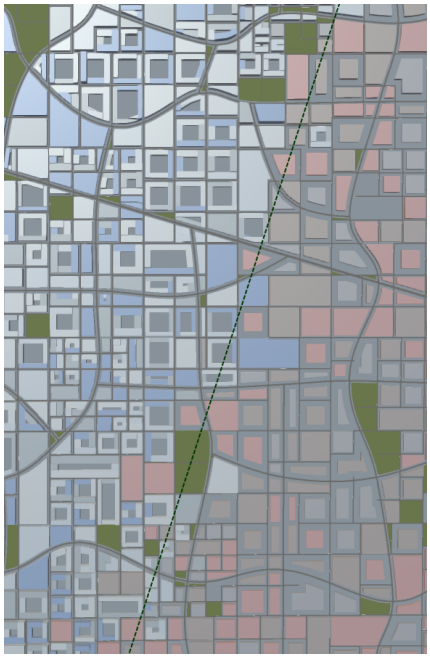

Figure 63. 2 different districts (red/brown and blue/white) meet at the line dividing 2 city subcenters.

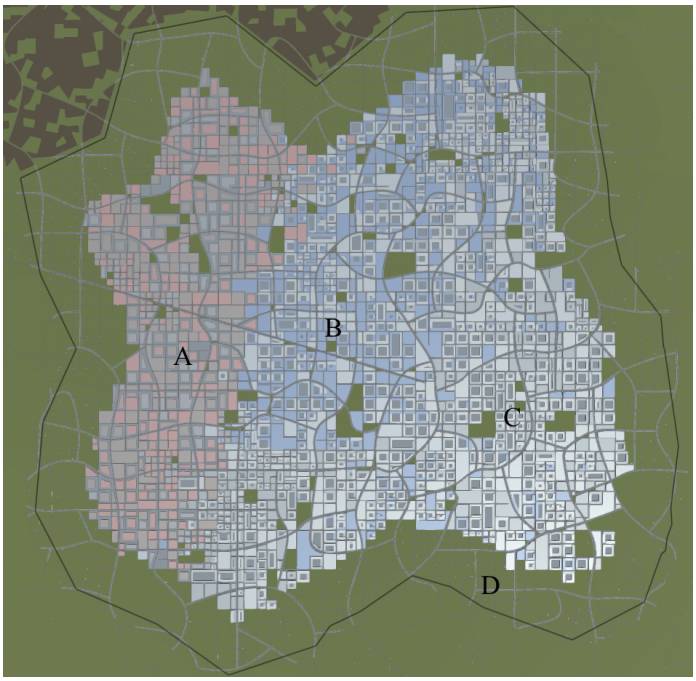

Figure 64. The final city layout. A: an industrial region, B: a commercial region, C: a tall residential region with apartment buildings, D: a suburban area with smaller houses (not rendered)

After the street layout has been generated, the polygon for each plot of land between the streets is found and assigned a type. If the plot is less than 500 m^2, the plot is marked as a small park with some vegetation. Otherwise, the distance to the city border is

determined. If the plot is closer than 800 meters to the city border, the plot is marked as a suburban area. If the plot is further inside the city, the type of the plot is determined with the use of the city center and subcenters. The main city center is marked as a commercial region. The furthest city subcenter is marked as an industrial region and the rest of the subcenters are evenly divided between tall residential areas, industrial areas and suburban areas. To determine the type of a plot using the city center and the subcenters, the closest city center to the plot is found and the type of the city center taken. To prevent a more noticeable division line between two centers (see Figure 63), a small chance, depending on the distance from the second closest city center, to instead pick the type of the second closest city center is applied. The final distribution of the types of plots for a small city can be seen in Figure 64.

5.3.2 Building Generation

After each plot has been assigned a type, an attempt to generate buildings for it is made. As the first step, each plot is offset inwards by a pseudorandom amount ranging from 2 to 7 meters. If the offset plot is convex and the plot is marked for tall residential or commercial buildings, the offset plot is further subdivided (Figure 65). The offset plot polygon is iterated with its longest edges processed first. A pseudorandom building width w is picked and a new line parallel to the longest edge w units away through the plot polygon is generated, which slices the polygon into two polygons. The smaller polygon is then marked as a building and the process repeated for the remaining polygon with the condition that the next edge can only be chosen from the original plot polygon.The remaining area in the middle of the polygon will not be visible to the user travelling by land and is left empty.

As the algorithm above does not work for high residential and commercial plots with a concave outline, a single large building outline is generated for them instead. For industrial plots, the initial offset outline is used to generate a tall concrete fence

with probability 0.9. If a fence was generated, another further inwards offset polygon is generated and used as the building outline. If no fence was generated, the initial offset outline is used as the building outline instead.

The buildings are then generated by first determining the lowest height of the building. To do this, terrain height is sampled at every corner of the house and the midpoint of every edge and the lowest height chosen. The height of the building is then determined at pseudorandom, with a bias for taller buildings near the center of the city and a texture chosen depending on the type of the building. Next, each edge of the building outline is iterated and a rectangular wall for the facade generated and lastly the building outline polygon is triangulated and a mesh for the roof generated (Figure 66).

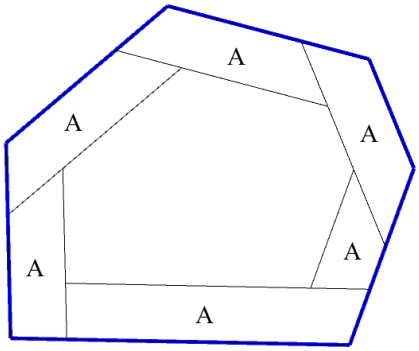

Figure 65. A city building plot (blue) after subdivision into multiple building outlines (A).

For suburban areas, each plot is subdivided using the same algorithm used in the village generation chapter and a selection from pre-existing house models is made. Each plot also has a 0.005 chance to place a church instead of a house. Additionally fences are generated around the houses and the gardens are populated with different vegetation.

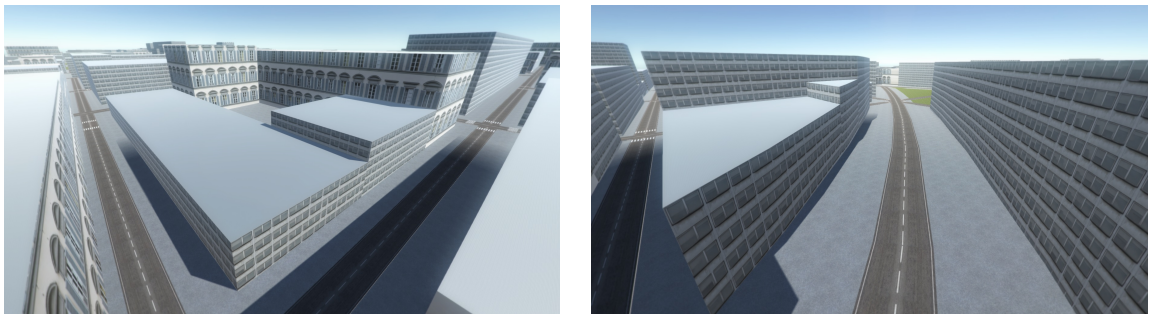

Figure 66. Generated city buildings in downtown.

5.4 Utility Pole Generation

To add more finer detail to the world, utility poles are generated. Utility poles are commonly placed alongside roads in the real life for the ease of access of maintenance. Although it is not uncommon to see utility poles pass through a forest without any major roads. In the devised algorithm, utility poles are placed next to side roads on their margins, which are free from any nearby trees.

To place utility poles next to a side road, the length of the road curve has to be calculated. The number of poles to be placed next to a road is determined by dividing and then rounding the length of the road curve with the average distance of 40 meters between poles. The road curve is then iterated with the given step and an attempt to place an utility pole made at the roadside of each point on the curve. To determine on which side of the road the utility pole should go, the direction vector v of the curve at the given point is checked. If $v_x \leq v_z$, the segment is placed to the right of the direction vector; otherwise to the left. If the initial utility pole position happens to intersect with an intersecting feature like a street or an unpaved road, up to 4 additional attempts to place the pole in increments of 2 meters further along the road are made. If no suitable place for an utility pole is still found, the pole is not placed. This happens sometimes in the cities when streets cause a pole to not be placed. This leaves a gap in the power lines but unless the user is actively looking for these gaps, they are infrequent enough to not be noticed.

To connect utility poles with power lines, each utility pole generates a list of neighbouring utility poles it should connect to. This is trivial in the middle of the curve as each utility pole has 2 neighbours, with only side road junctions requiring special care. At side road junctions, a single utility pole is placed close to the junction and rotated towards the center of the junction. The first pole of every road originating from the junction is then added to the neighbour list of the pole placed close to the junction. To simulate the

Figure 67. Generated utility poles with their connections at a side road junction.

realistic shape of hanging electric cables, a catenary[11] curve is used. The final generated utility poles and power lines are illustrated in Figure 67.

5.5 Traffic Sign Generation

The devised algorithm also generates settlement enter and exit signs, navigation signs, speed limit signs and crossroads (Figure 68). The settlement enter (see figure 68 A) and leave signs (see figure 68 B) are generated for both cities and villages. As villages were only generated on mesochunks that contained a side road node, the road curves corresponding to each of the side road node edges are iterated and the intersection point with village borders found. The signs are then be placed to the right of the intersection point next to the road. The same is done with cities, where signs are first added to highways and then to every side road which has one end inside the city and the other

[11] https://en.wikipedia.org/wiki/Catenary

outside the city.

Furthermore, direction signs (see figure 68 C and D) are generated at every side road node by traversing the local side road graph of the node. If a highway connecting to a city or a village is found up to 2 side road nodes away, a named sign with the distance is generated. Note that the city or the village does not have to exist yet since the city and village naming is based on their highway and side road node positioning respectively. This means that the settlement name can be generated well in advance of the settlement instead with the use of the corresponding road node. For villages, this means that a direction sign can appear up to roughly 2 kilometers ahead of it which is the maximal distance without increasing the size of a mesochunk's local side road graph window. In practice this is not a major problem as the user is unlikely to look for a specific village. The maximal distance for city signs is about 20 kilometers.

A crosswalk (see figure 68 E) is generated at every side road node that has at least 2 neighbours in a village and a city. The outline of the crosswalk is determined by taking the first segment of each road curve and calculating a rectangle perpendicular to it. Then texture coordinates of each corner of the rectangle are found with the use of the road curve direction vector.

Some speed limit signs are placed throughout the world. The 70 km/h speed limits (see figure 68 F) are placed about 100 meters ahead of the settlement enter signs. They are generated together with settlement enter signs and placed 100 meters further down the spline in the direction heading out of the settlement. The 90 km/h speed limits (see figure 68 G) are placed similarly a few hundred meters after settlement exit signs as a reassurance of the current speed limit. Lastly, 110 km/h signs (see figure 68 H) are placed on highways after junctions with side roads, to alert drivers entering the highway of a higher speed limit.

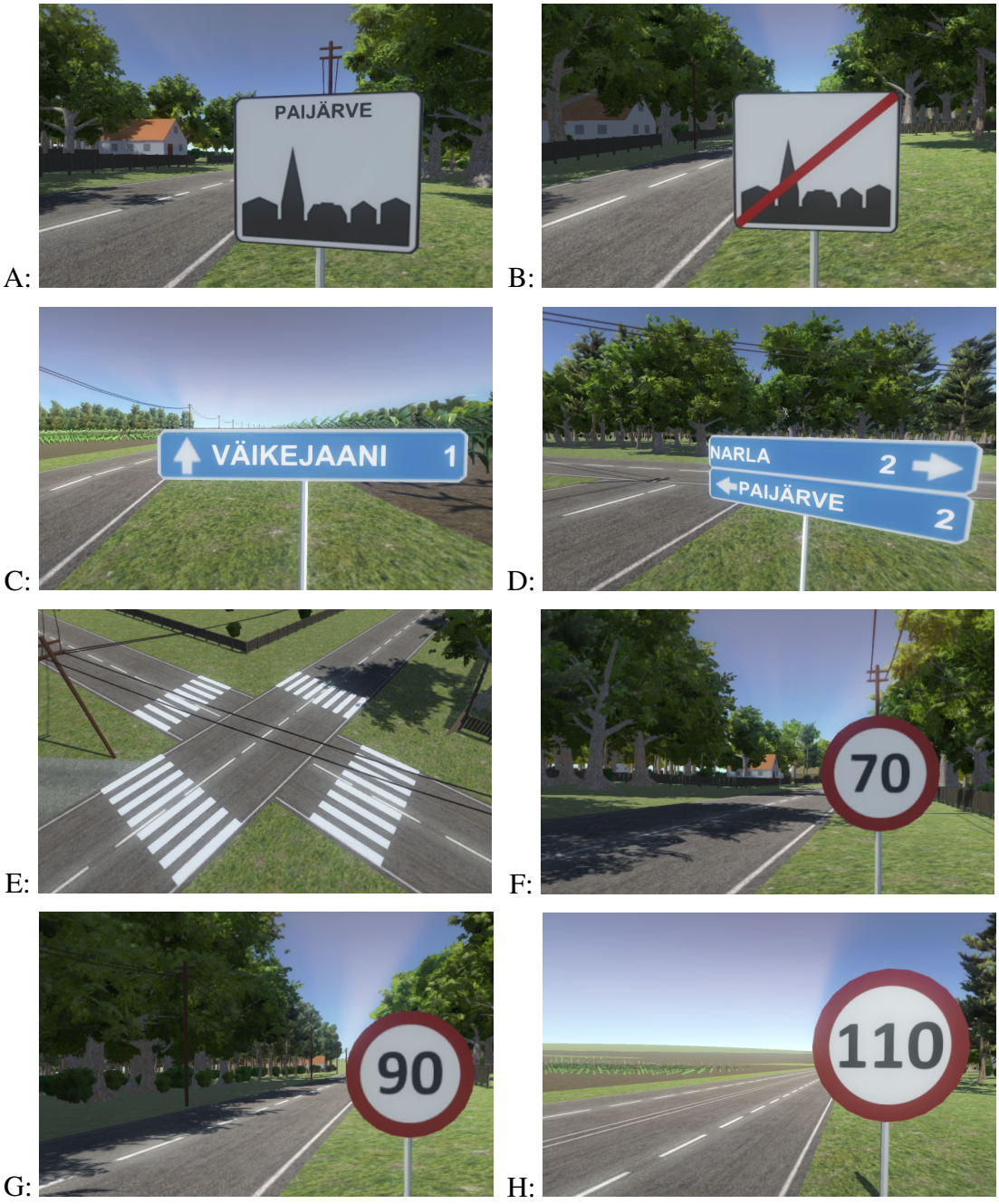

Figure 68. Various traffic signs placed throughout the world. A: a settlement enter sign. B: a settlemente xit sign. C and D: direction signs indicating the distance to a settlement. E: crosswalks at side road junctions. F and G: speed limit signs placed on side roads near entering and exiting settlemetns. H: speed limit signs placed on highways near side roads merging with the highway.

5.6 Rendering

This chapter gives an overview of how the generated features are rendered. First, the methods used to generate the terrain meshes are discussed, followed by the road mesh generation. The last subchapter gives an overview of how other terrain objects like the trees, fences and traffic signs are placed and removed from the world.

5.6.1 Terrain Mesh Generation

To generate a terrain mesh for a mesochunk, the same noise function as discussed in the macrochunk terrain generation chapter 4.1 is used. Instead of sampling it every 64 meters, it is sampled every 8 meters, creating a higher resolution heightmap. As every mesochunk is divided into multiple regions with different types of ground textures, each region is processed separately. Each region is triangulated into a terrain mesh so that it would generate vertices for each heightmap grid point contained inside the region along with any additional vertices required to create the outline triangles as seen in Figure 69. This is done by finding all the intersecting heightmap tiles. Two triangles are generated for the tiles that are completely within the region. For tiles that intersect with the region border, the intersection points between the tile and the region border are found and used to generate a triangle fan[12] fitting the tile.

[12]https://en.wikipedia.org/wiki/Triangle_fan

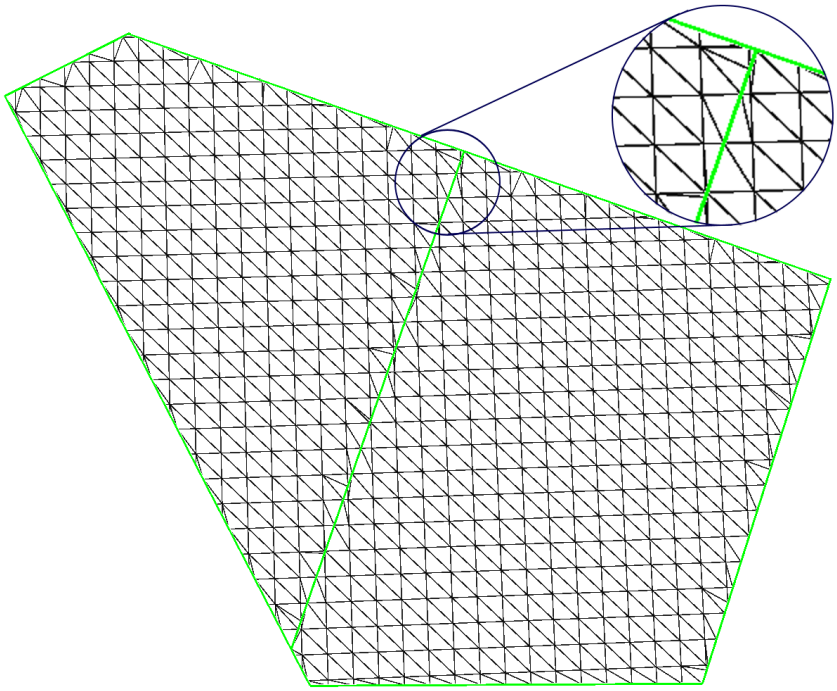

Figure 69. Two mesochunk regions (outlined with green) after being divided into triangles (black).

5.6.2 Road Mesh Generation

To generate the road mesh, an approach similar to the terrain generation is used with the addition of having to generate rotated texture coordinates for each road segment. For each straight road segment, the vertices shared with the heightmap grid and any additional vertices on the outline of the road segment are calculated. For texture coordinates, a line in the direction of the road and a line perpendicular to the road are calculated (see Figure 70) and the texture coordinates of every vertex interpolated using the texture coordinates at the endpoints of these lines.

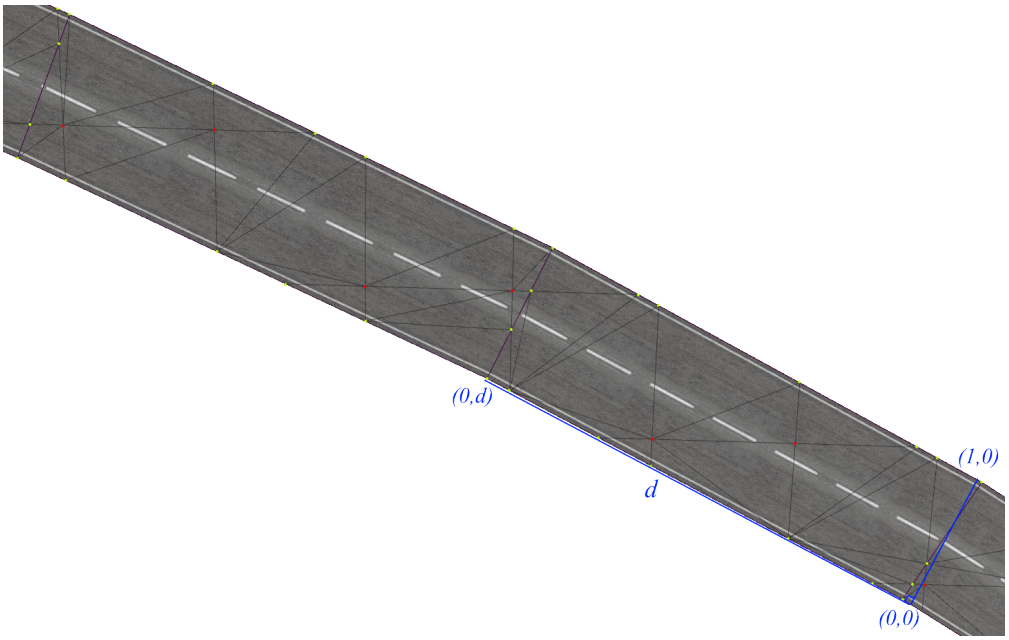

Figure 70. Texture coordinates (blue) and the triangulation of a road segment. Red vertices are taken from the heightmap, green vertices are calculated based on the outline of the road.

5.6.3 Terrain Object Instantiation and Rendering

Although terrains and roads are generated for several kilometers ahead, performance problems emerge when trying to populate it all with vegetation and other terrain objects. To control the number of these objects, a maximum terrain object instantiation radius is defined, within which terrain objects are created. The performance of various instantiation radii are discussed in chapter 6.3. To improve performance, all plants have several levels of detail, which are dynamically switched as the camera moves (Figure 71). The lowest level of detail uses a technique called *billboarding*, which renders a single 2D textured quad instead of a 3D model.

Figure 71. Left to right: decreasing level of detail of a maple tree.

6 Implementation and Assessment

This chapter gives a brief overview of the software used for the devised algorithm's implementation and describes the user interface. Afterwards, the visual results of the algorithm are compared with real-life imagery and an overview of conducted performance testing is given. Lastly, the key differences with existing solutions are discussed.

6.1 Implementation

The algorithm described in the previous chapters was implemented using the Unity[13] game engine. Unity was chosen because of the ease of development as it provides a lot of useful high level functionality and the author had previous experience with it. The algorithm uses an implementation of FastNoise SIMD[14] library for the fast generation of simplex noise by utilizing the SIMD capabilities[15] of the CPU when possible. For polygon clipping, the *clipper* library[16] is used, which enables high performance usage of boolean operations on arbitrary polygons. The main algorithm logic is separated into separate classes with minimal code specific to the Unity engine. Unity calls are handled by special wrapper classes to enable easier porting to a different engine if one so desires. The source code is available as an attachment and from a Bitbucket git repository[17] and a build is available as an attachment or from Google Drive[18].

The implementation is multithreaded, with all the macro- and mesochunks data being generated on separate threads. Both types of chunks have their own thread pool. The meshes are created and objects instantiated on the main thread due to constraints by the Unity engine. For this reason, they are created in limited size batches to prevent severe

[13] https://unity3d.com/

[14] https://github.com/Auburns/FastNoiseSIMD

[15] https://en.wikipedia.org/wiki/SIMD

[16] http://www.angusj.com/delphi/clipper.php

[17] https://bitbucket.org/AndreasGP/mastersthesis/

[18] https://drive.google.com/open?id=1QSqh7hQItECjR_iY_IJRj1XpkMSdltfs

frame rate drops.

Figure 72. The user interface of the application while driving in a car.

The user can choose to fly around freely or drive around in a default car provided by Unity. The user interface enables the user to also control different render distances to reduce or increase performance. Additionally, a list of common points of interests are provided (Figure 72).

6.2 Visual Assessment

This chapter compares the results of the algorithm to real world imagery from Google Maps[19] and suggests potential future improvements based on them.

The rural road networks generated by the devised algorithm are mostly connected, but dead-end roads are quite common (Figure 73). To mitigate this, villages were placed at the ends of such roads to give each road a function. In real life, rural roads tend to be longer and do not contain as short length cycles as the generated roads (Figure 74). In addition, rural roads branch frequently into more minor dead-end roads which lead to remote residences or agricultural roads. Real roads also have more turbulence in their shapes.

Figure 73. Rural side roads generated by the devised algorithm.

Figure 74. Road network near Rakvere, Estonia. Taken from *Google Maps*.

The generated side road junctions (Figure 75) are very basic and could be improved by introducing additional turn lanes and branches along with proper road markings (Figure 76).

The generated highway junctions merge with each other like side roads do. These junctions could be vastly improved by generating multi-tiered junctions to help the traffic flow (Figure 77).

[19] https://maps.google.com/

Figure 75. Rural side roads generated by the devised algorithm.

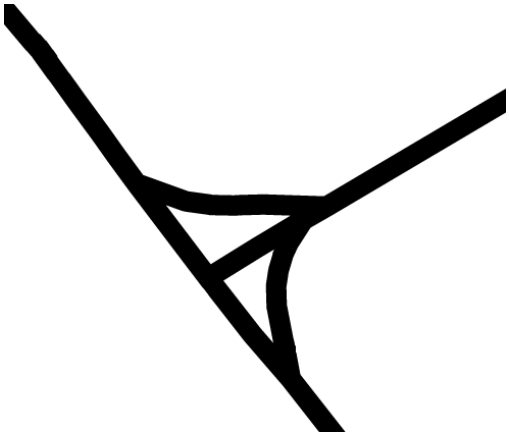

Figure 76. Road junction near Jõgeva, Estonia. Taken from *Google Maps*.

Figure 77. A clover junction near Leipzig, Germany. Taken from *Google Maps*.

The generated city layouts are of acceptable quality when travelling on land. From top view, the axis-aligned streets of the city emerge (Figure 78). This could be improved by adapting a more refined city generation algorithm from chapter 2. Additionally, more concentric circular roads could be introduced to be generated around city centers, as seen in Tirana, Albania for example (Figure 79).

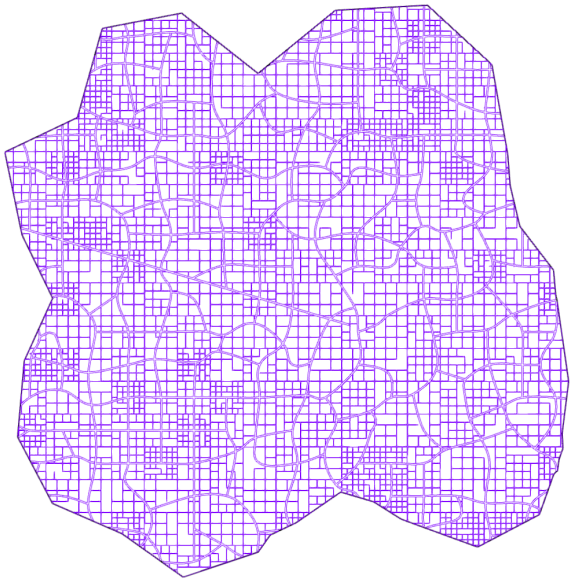

Figure 78. City street layout generated by the devised algorithm.

Figure 79. A street network in Tirana, Albania. Taken from *Google Maps*.

The generated villages (Figure 80) approximate real villages pretty well (Figure 81).

The generated forests (Figure 82) are divided into relatively small patches with different growth patterns due to being grown and owned by different entities. Similar division can be seen in real life (Figure 83).

The generated farmlands (Figure 84) are also similar to real world farmlands (Figure 85), with occasional forest groves and farms commonly aligned with roads.

The generated nature reserves (Figure 86) are quite simplistic and could use more plant variety when compared to real nature reserves (Figure 87). However when moving

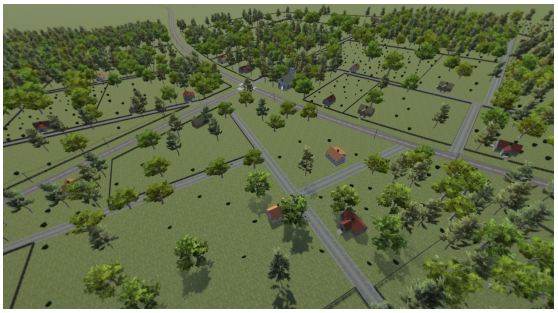

Figure 80. Village generated by the devised algorithm.

Figure 81. A village named Drelów, Poland. Taken from *Google Maps*.

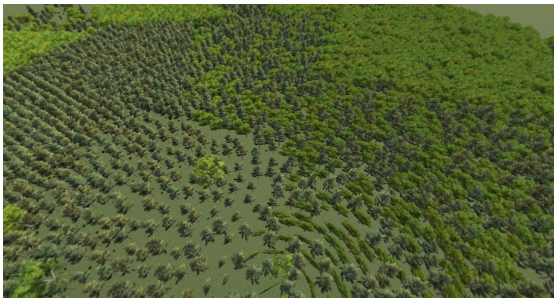

Figure 82. Forests generated by the devised algorithm.

Figure 83. A forest near Jõgeva, Estonia. Taken from *Google Maps*.

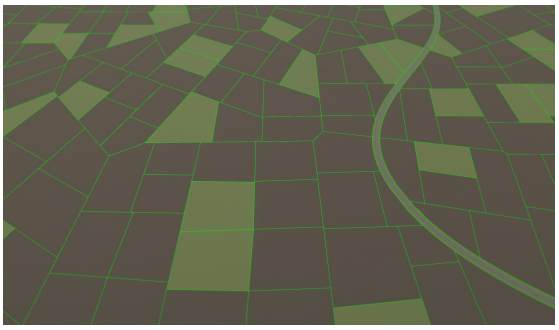

Figure 84. Farmland plots generated by the devised algorithm.

Figure 85. Agricultural land usage near Tartu, Estonia. Taken from *Google Maps*.

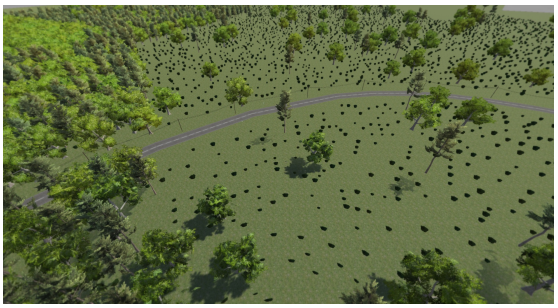

Figure 86. Nature reserves generated by the devised algorithm.

Figure 87. Nature Reserve in Saaremaa, Estonia. Taken from *Google Maps*.

around a nature reserve, it is clear that is more natural than the planted forests and farmlands.

6.3 Performance Assessment

The devised algorithm was tested on two computers. The first computer had an *Intel Core i5 4460* processor, *16 GB* of RAM, a *Nvidia GeForce GTX970 4GB* graphics card and was tested on the highest quality setting with a 1920×1080 px resolution. The second computer had an *Intel Core i5 3437U* processor, *8 GB* of ram, an *Intel HD Graphics 4000* integrated graphics card and tested on the lowest quality setting with a 1366×768 px resolution. The two most important parameters that the user can change are the mesochunk generation radius and the terrain object instantiation radius. The mesochunk generation radius controls how many mesochunks around the player will be generated in every direction. The terrain object instantiation radius controls up to how far from the player terrain objects like trees will be instantiated. There were 2 tests done with both computers with different mesochunk and terrain object radii. The tests included measuring average frame rates when looking around in different regions

while remaining stationary (Figure 88) and when moving through different areas at $100\frac{\text{m}}{\text{s}}$ ($360\frac{\text{km}}{\text{h}}$) (Figure 89). Furthermore, additional tests measured the time taken to generate specific features 4.

As can be seen from the data in table **??**, the frame rate drops the most in forests due to a large number of terrain objects. This could be improved by buying or creating more optimized tree models. The average frame rates on the first machine were acceptable. To maintain a relatively consistent $60+$ frames per second, mesochunk radius of 2 and terrain object instantiation radius of 200 meters could be used. The second machine suffered pretty heavily in frame rates, averaging about 40 frames per second due to an integrated graphics card. On modern hardware, the application is usable and could possibly be further optimized to run steadily even on the older, second machine. The movement speed ($360\frac{\text{km}}{\text{h}}$) used in the test was quite high and users travelling by land are less likely to experience lag induced by quick movement, as their speed will likely be lower. As the application had to work with only freely available models of varying quality, the frame rates on lower end computers and even on higher end computers in dense forests were lower than they could be. Additional frame rate problems were caused by non-optimal usage of object pooling, which could be further improved in future work.

As can be seen from the data in table 4, macrochunk generation is quick even if it contains a city and most of the time is spent generating mesochunks, which take half a second to generate their data and another half a second to generate meshes and instantiate objects. On computer 1, a total average generation time of 728 ms for a mesochunk would indicate that 82 mesochunks could be generated and rendered per minute, however as the mesochunks are generated in parallel on different threads, 50 chunks were able to be generated in 10 seconds on machine 1 and 14 seconds on machine 2. Extrapolating these numbers yields that machine 1 could roughly generate 300 mesochunks ($9 \cdot 9$ km^2) of fully detailed land per minute and machine 2 could roughly generate 214 mesochunks ($7.5 \cdot 7.5$ km^2) per minute.

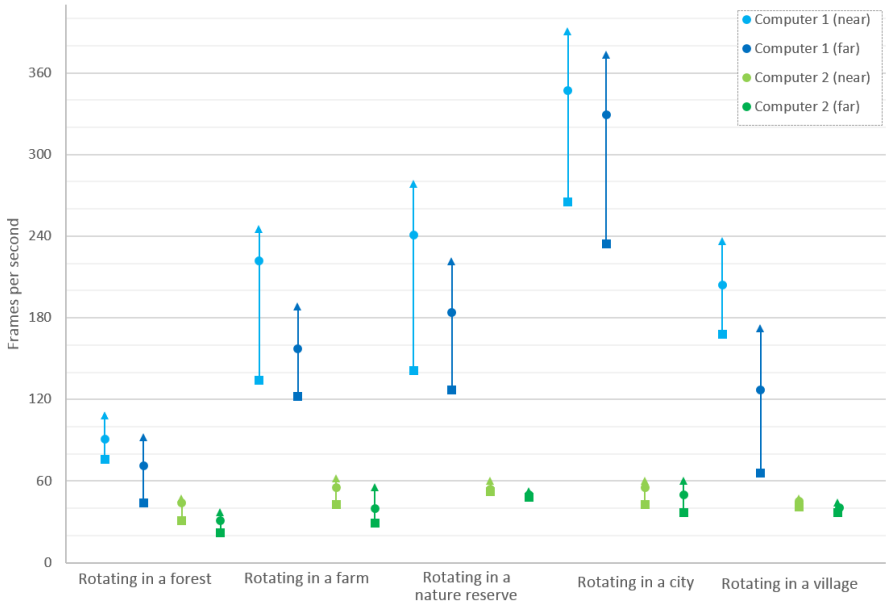

Figure 88. Average (circle), minimum (square) and maximum (triangle) frame per second while looking around in various environments. The near settings used mesochunk generation radius 1 and terrain object instantiation radius 150 meters.

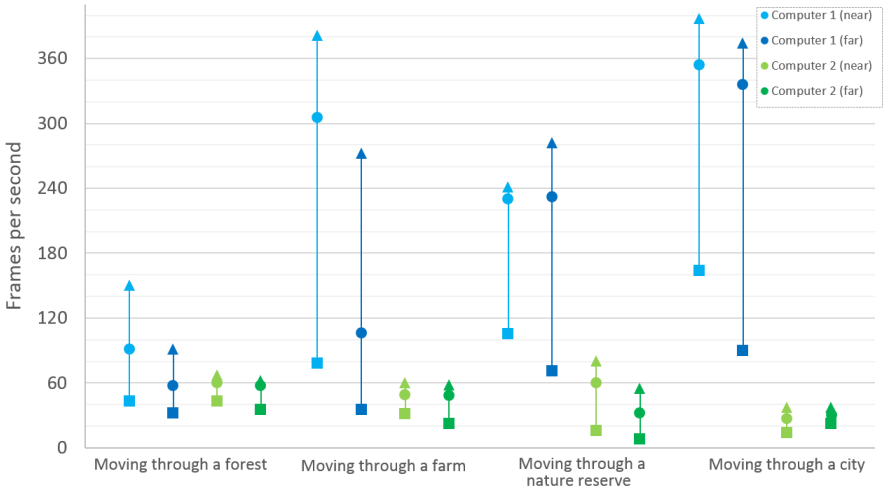

Figure 89. Average (circle), minimum (square) and maximum (triangle) frame per second while moving through and generating various environments. The near settings used mesochunk generation radius 1 and terrain object instantiation radius 150 meters.

Table 4. Average generation times of different features.

	Computer 1	**Computer 2**
Macrochunk generation without a city (overall)	7 ms	15 ms
Macrochunk generation with a city (overall)	18 ms	27 ms
Mesochunk data generation (overall)	413 ms	553 ms
Mesochunk city data generation	82 ms	117 ms
Mesochunk village data generation	270 ms	361 ms
Mesochunk rural region data generation	320 ms	423 ms
Mesochunk mesh generation and object instantiating (overall)	315 ms	448 ms
Mesochunk terrain mesh generation	142 ms	218 ms
Mesochunk road mesh generation	8 ms	20 ms
Mesochunk terrain object instantiating	137 ms	187 ms

6.4 Comparison with Previous Work

This chapter outlines the main new features of the devised algorithm that are not previously described for infinite world generation. The three known works on infinite world generation are the Charack framework by Bevilacqua *et al* [BPd09], which discusses infinite terrain generation; infinite city generation framework by Steinberger *et al* [SKK$^+$14] and the most closely related framework by Magalhães [eM17].

The Charack framework is used to generate infinite worlds with no artificial features. It focuses on land and sea generation with a large focus on generating coastlines. The algorithm devised in this thesis uses a noise-based heightmap, which generates more basic terrain and could potentially be integrated with the Charack framework.

The infinite city generation framework by Steinberger *et al* generates infinite cities on the CPU using shape grammars. The achieved speeds of generation are very impressive, but as the city layout was not a large focus, they used an axis-aligned grid with a few diagonal lines for the streets, which has significantly less variance than the devised road generation algorithm in this thesis.

Magalhães's master's thesis focuses on a similar topic to this thesis. He generates 2-dimensional procedural infinite worlds with some infrastructure, namely cities and highways, along with basic terrain and water bodies. For the land and oceans, a Perlin noise based heightmap is used with points below a global water level threshold marked as oceans. In addition to oceans, he also generates rivers in a fixed size area, which was described in chapter 2.1. The devised algorithm for this thesis does not generate water as reimplementing the fixed global water level based approach had little novel value and the generation of other new features was a priority. For the cities and highways, Magalhães describes two approaches, which were discussed in chapter 2.2. A shortcoming of his algorithm is that the city street networks look very artificial and the city outlines have a relatively convex Voronoi cell-like shape (Figure 90). Additionally, no internal city

structure is generated and his highways are placed as almost straight lines between cities, which go through water (Figure 91). No additional features are generated outside the cities, other than the rivers and highways.

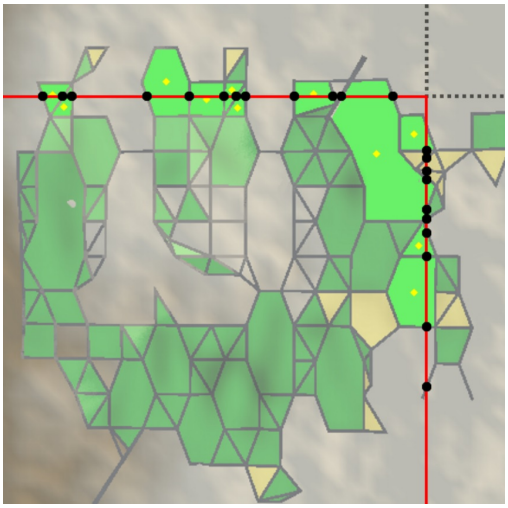

Figure 90. A city layout generated by Magalhães [eM17].

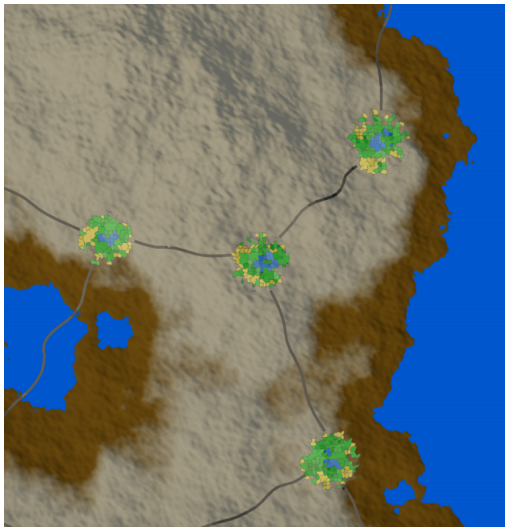

Figure 91. Cities and highways generated by Magalhães [eM17].

7 Future Work

This chapter discusses possible expansions of the algorithm to further expand the variety of the generated features. First, a potential new large chunk, called a megachunk, is introduced and the possibilities of using them for water body and railway generation are discussed. Next, a brief description of an improved utility pole generation algorithm and ideas for improving the junctions are discussed.

7.1 Megachunks

Chapter 3.1 introduced 2 types of chunks: macrochunks of size 8192×8192 m^2 and mesochunks of size 512×512 m^2. However, the larger, macrochunks, are not necessarily big enough to generate more intricate large scale features like rivers and varied height lakes. Therefore, even larger chunks, called megachunks, could be implemented to support these features. A megachunk would be 128×128 km^2 of size and therefore contain 16×16 macrochunks.

Similarly to how a macrochunk generated a low resolution heightmap that was later refined by a higher resolution heightmap for mesochunks as discussed in chapter 4.1, an even lower resolution heightmap could be generated for a megachunk using the same noise. The generated heightmap would then roughly approximate the terrain of any macrochunks generated on top of it.

7.1.1 Water Bodies

The heightmap of a megachunk could potentially be used to generate rivers. Each megachunk could be considered separately for a river in the area bounded by the megachunk. Since a megachunk covers a relatively large area of 128×128 km^2, the generated rivers would be of a reasonable length and it would be unlikely for the user to notice that rivers do not cross any megachunk borders as they are so infrequent. By

limiting the area for the river generation to a single megachunk and having access to the heightmap of the megachunk, a previously existing algorithm from chapter 2.1 could be implemented which generates rivers by post-processing the heightmap to find suitable paths. The generated rivers could then be stored in the megachunk's data and when a macrochunk or a mesochunk is being generated, queried from it.

The addition of rivers would require some changes to the macrochunk and mesochunk generation process. First, the roads must also generate bridges when crossing the rivers and avoid generating roads in places, where a river could overlap with a road in the same direction. The bridges could be generated modularly with the use of a predefined 3D model of a bridge segment. That could then be placed sequentially over the distance of the river, with parts of the bridge clipped under the terrain. To avoid too long road and river overlaps, a check could be made for the roads near rivers to determine the rough distance of the overlap and to remove the road from generation.

With the generation of rivers, the city placement algorithm could be changed to favour areas close to rivers as cities have historically emerged near rivers due to water commerce and having a source of food. The generation of the inner structure of cities could stay largely unchanged and just clip the areas near rivers from street and building generation, although it would be possible to further expand this by introducing special waterfront generation for areas near water. Furthermore, villages could also be placed near rivers with a higher frequency, with the river areas culled from the building generation step. Forthe rural areas, a new special region for the riverside could be introduced, which could contain plants accustomed to near water bodies.

In addition to the rivers, varied height lakes could be generated. Lakes could be generated similarly to the rivers with the use of a megachunk's heightmap and being bound by the megachunk's borders. A grid of points could be sampled on the heightmap and some, that are the local minima of the surrounding area, considered as the lake seed points. Then, each seed point could be generated on the heightmap with a flood algorithm

until water levels reach a specific height or a maximum surface area is reached.

As lakes are usually passable by going around and bridges are rarely built on lakes, the road algorithm should be modified to discard any roads that intersect with a lake. As another improvement, it would be possible to modify the road algorithm so instead of completely discarding a road, several attempts would be made to place the road so that it would follow the shoreline [PM01, Per16].

7.1.2 Railroads

In addition to water bodies, megachunks could be used to generate railroads. Railroad generation differs from road generation by having to generate longer continuous tracks and paying more attention to the terrain and junctions. As trains are more sensitive to terrain gradient than cars, the train tracks should be placed on the terrain with minimal elevation differences. As train tracks are a lot less frequent in real life, generating a few tracks on megachunks would be enough. The tracks could be generated by attempting to connect different cities on the megachunk by finding a minimal elevation path using the megachunk's heightmap. Special care has to be taken at railway junctions as tracks should merge without any sharp turns. For example, at a junction with 3 tracks, two of them could be merged with the third track with C^1 continuity by picking a derivative at the junction in the direction directly between the two tracks merging with the third one.

7.2 AI Navigation Agents

As all the road networks are available as graphs, it would be possible to implement AI agents driving around on the roads to add a new layer of immersion to the world. However these agents can likely not be generated to be temporally deterministic, as determining their locations would require simulating their interactions with other agents. In addition to vehicles, it would also be possible to add pedestrian agents in settlements.

7.3 Improved Utility Pole Generation

As a future improvement, it would be possible to tweak the side road generation algorithm to also generate power lines through areas without roads. To do this, a network very similar to side road network could be generated using the same side road nodes, but during the graph culling phase a smaller angular threshold could be used. This would create a network that would have nearly all the same edges as the side road network plus some extra edges that were not culled. That network could then be used to place utility poles directly instead of the current side road network. This would also require the countryside and city generation algorithms to be modified to accommodate the more advanced placement of utility poles.

7.4 Improved Junctions

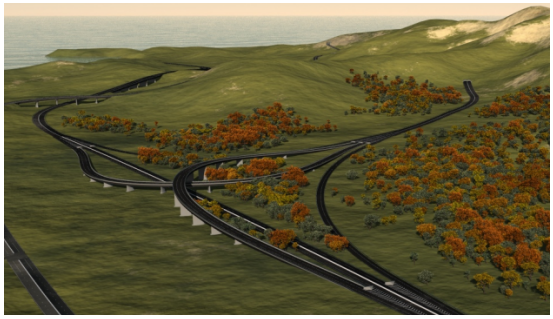

Figure 92. A junction generated by Galin *et al* [GPGB11].

As discussed in chapter 6.2, the currently generated junctions are not as realistic as they could be. To improve them, an area of land around them could be reserved for further junction generation, with additional merge lanes and road markers generated. In addition to standard junctions, roundabouts could be generated by generating a circular road which merges with all the incoming roads. For proper multi-tiered junctions at highway junctions, a more complex system would have to be developed, enabling the generation of multi-tiered highways with exit and enter ramps like the one created by Galin *et al* (Figure 92) [GPGB11].

8 Conclusion

In this thesis an algorithm that procedurally generates deterministic infinite worlds with infrastructure was created. The aim of the thesis was to create an algorithm capable of generating common infrastructure elements like roads, cities and villages along with appropriate nature features for areas close to human settlements. The created algorithm uses a newly devised exponential generation technique, which divides the world into differently sized nested pieces, called chunks. Each differently sized chunk was used to generate world features of similar size. Two different chunks, called macro- and mesochunks, were implemented and third potential chunks, called megachunks, discussed. The larger of the implemented chunks, macrochunks generated highways and city locations, shapes and names and provided approximate terrain far away from the camera for a large render distance. The smaller mesochunks generated side roads, named villages, city layouts, buildings and traffic signs, including speed limits and direction signs. Mesochunks also divided the rural areas between different agricultural areas. The three main types of generated rural land were cultivation, forestry and nature reserves, each of which were further populated with various vegetation layouts and occasional residences. The generation of an infinite world with this level of complexity has not been tackled before to the extent of author's knowledge. The achieved results were of an acceptable level with many possibilities for further research. Applications of the devised algorithm include usage as a base terrain for a computer game, which could make use of the niche of being infinite or having a very small initial file size. Originally, the idea of using the algorithm as a world for a flight simulator was entertained, but the current algorithm is not performant enough to be capable of having an acceptable render distance for flight, although performance while driving in a car is acceptable.

The generated features were compared with real world imagery and found to share the basic common traits with them, albeit with less variety than the real world. The main

constraint for little variety was due to the lack of free high quality assets. The algorithm's performance was measured and it was determined to be fast enough to generate 9×9 km^2 of fully detailed terrain per minute on a modern computer. The implementation's frame rate suffered slightly from varying quality free assets and unoptimized object creation in dense forests. By creating high quality optimized assets and implementing further optimizations, the algorithm could likely be modified to run steadily even on older computers with integrated graphics cards. The algorithm was implemented in the Unity game engine and lets the user to fly around freely or to drive around in a car to explore the infinite world.

Reflecting on the work, devising this algorithm required a lot of experimentation and clever combining of previously existing techniques to generate seamless infinite features. A huge help was previous knowledge gained from the author's bachelor's thesis, which discussed infinite terrain generation using voxels. Overall, infinite procedural world generation is a very interesting topic and the author would recommend anyone to help further expand this exciting field of research. The development of the algorithm will continue with initial efforts going towards optimizing performance for lower end machines and hopefully making the algorithm capable of running on mobile devices.

References

[Aur91] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[BPd09] Fernando Bevilacqua, Cesar Tadeu Pozzer, and Marcos Cordeiro d'Ornellas. Charack: Tool for real-time generation of pseudo-infinite virtual worlds for 3d games. In *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*, pages 111–120. IEEE, 2009.

[Bri07] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. In *SIGGRAPH sketches*, page 22, 2007.

[CdSF05] António Fernando Coelho, António Augusto de Sousa, and Fernando Nunes Ferreira. Modelling urban scenes for lbms. In *Proceedings of the tenth international conference on 3D Web technology*, pages 37–46. ACM, 2005.

[CEW+08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. In *ACM transactions on graphics (TOG)*, volume 27, page 103. ACM, 2008.

[CM05] Xiaorui Chen and Sara McMains. Polygon offsetting by computing winding numbers. In *ASME 2005 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 565–575. American Society of Mechanical Engineers, 2005.

[DHL+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286. ACM, 1998.

[EBP+12] Arnaud Emilien, Adrien Bernhardt, Adrien Peytavie, Marie-Paule Cani, and Eric Galin. Procedural generation of villages on arbitrary terrains. *The Visual Computer*, 28(6-8):809–818, 2012.

[eM17] Luis Miguel Coelho e Magalhaes. Procedural generation of infinite 3d worlds for games. Master's thesis, University of Porto, 2017.

[Ers12] Erich Erstu. Maakaartide juhugeneraator. Bachelor's thesis, Universty of Tartu, 2012.

[GGG+13] Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG)*, 32(4):143, 2013.

[GH98] Günther Greiner and Kai Hormann. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics (TOG)*, 17(2):71–83, 1998.

[GMS09] James Gain, Patrick Marais, and Wolfgang Straßer. Terrain sketching. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 31–38. ACM, 2009.

[GPGB11] Eric Galin, Adrien Peytavie, Eric Guérin, and Bedřich Beneš. Authoring hierarchical road networks. In *Computer Graphics Forum*, volume 30, pages 2021–2030. Wiley Online Library, 2011.

[GPSL03] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '03, pages 87–ff, New York, NY, USA, 2003. ACM.

[Gus05] Stefan Gustavson. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*, 2005.

[Ham01] Johan Hammes. Modeling of ecosystems as a data source for real-time terrain rendering. In *Digital Earth Moving*, pages 98–111. Springer, 2001.

[HDBB10] Remco Huijser, Jeroen Dobbe, Willem F Bronsvoort, and Rafael Bidarra. Procedural natural systems for game level design. In *Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium on*, pages 189–198. IEEE, 2010.

[HGA+10] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. In *Computer Graphics Forum*, volume 29, pages 2179–2186. Wiley Online Library, 2010.

[HMVDVI13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1, 2013.

[Joh15] Skovbo Rune Johansen. Procedural world generation: The simulation, functional and planning approaches. Gamasutra, October 2015. Visited 21.05.2018.

[KBKŠ09] Peter Krištof, Bedrich Beneš, Jaroslav Křivánek, and Ondrej Šťava. Hydraulic erosion using smoothed particle hydrodynamics. In *Computer Graphics Forum*, volume 28, pages 219–228. Wiley Online Library, 2009.

[KM07] George Kelly and Hugh McCabe. Citygen: An interactive system for procedural city generation. In *International Conference on Game Design and Technology-Fifth*, pages 8–16, 2007.

[KMN88] Alex D Kelley, Michael C Malin, and Gregory M Nielson. *Terrain simulation using a model of stream erosion*, volume 22. ACM, 1988.

[LGD+18] Jun Liu, Yanhai Gan, Junyu Dong, Lin Qi, Xin Sun, Muwei Jian, Lina Wang, and Hui Yu. Perception-driven procedural texture generation from examples. *Neurocomputing*, 291:21–34, 2018.

[LGL+18] Jiaqi Li, Xiaoyan Gu, Xinchi Li, Junzhong Tan, and Jiangfeng She. Procedural generation of large-scale forests using a graph-based neutral landscape model. *ISPRS International Journal of Geo-Information*, 7(3):127, 2018.

[LHdV17] Xiaoming Lyu, Qi Han, and Bauke de Vries. Procedural modeling of urban layout: population, land use, and road network. *Transportation research procedia*, 25:3333–3342, 2017.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[LN03] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 203–212. ACM, 2003.

[LRBP12] Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. Treesketch: interactive procedural modeling of trees on a tablet. In *Proceedings of the international symposium on sketch-based interfaces and modeling*, pages 107–120. Eurographics Association, 2012.

[LRD07] RG Laycock, GDG Ryder, and AM Day. Automatic generation, texturing and population of a reflective real-time urban environment. *Computers & Graphics*, 31(4):625–635, 2007.

[LRW+06] Thomas Lechner, Pin Ren, Ben Watson, Craig Brozefski, and Uri Wilenski. Procedural modeling of urban land use. In *ACM SIGGRAPH 2006 Research posters*, page 135. ACM, 2006.

[LWW03] Thomas Lechner, Ben Watson, and Uri Wilensky. Procedural city modeling. In *In 1st Midwestern Graphics Conference*. Citeseer, 2003.

[MS09] James McCrae and Karan Singh. *Sketch-based path design*. Canadian Information Processing Society, 2009.

[MS15] Steve Marschner and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2015.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Acm Transactions On Graphics (Tog)*, volume 25, pages 614–623. ACM, 2006.

[OBRvdK17] Benny Onrust, Rafael Bidarra, Robert Rooseboom, and Johan van de Koppel. Ecologically sound procedural generation of natural environments. *International Journal of Computer Games Technology*, 2017, 2017.

[PBP13] Hartmut Prautzsch, Wolfgang Boehm, and Marco Paluszny. *Bézier and B-spline techniques*. Springer Science & Business Media, 2013.

[Per85] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[Per02] Ken Perlin. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.

[Per16] Kristjan Perli. Protseduuriline linnade genereerimine. Bachelor's thesis, Universty of Tartu, 2016.

[PGGM09] Adrien Peytavie, Eric Galin, Jérôme Grosjean, and Stéphane Mérillou. Arches: a framework for modeling complex terrains. In *Computer Graphics Forum*, volume 28, pages 457–467. Wiley Online Library, 2009.

[PHM93] Przemyslaw Prusinkiewicz, Mark S Hammel, and Eric Mjolsness. Animation of plant development. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 351–360. ACM, 1993.

[PM01] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[SCM+15] Ivan Silveira, Daniel Camozzato, Fernando Marson, Leandro Dihl, and Soraia Raupp Musse. Real-time procedural generation of personalized facade and interior appearances based on semantics. In *Computer Games and Digital Entertainment (SBGames), 2015 14th Brazilian Symposium on*, pages 89–98. IEEE, 2015.

[SDKT+09] Ruben M Smelik, Klaas Jan De Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, pages 25–34, 2009.

[Sep16] Andreas Sepp. Protseduuriline lõpmatu maastiku genereerimine. Bachelor's thesis, Universty of Tartu, 2016.

[SKK+14] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Peter Wonka, and Dieter Schmalstieg. On-the-fly generation and rendering of infinite cities on the gpu. In *Computer graphics forum*, volume 33, pages 105–114. Wiley Online Library, 2014.

[Stä76] Ekkehart Stärk. *Mehrfach differenzierbare Bézierkurven und Bézierflächen*. Technische Universität Carolo-Wilhelmina zu Braunschweig, 1976.

[STBB14] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. In *Computer Graphics Forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[STdKB11] Ruben Michaël Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2):352–363, 2011.

[Tun12] Raimond-Hendrik Tunnel. Protseduuriline puude genereerimine. Bachelor's thesis, Universty of Tartu, 2012.

[Vat92] Bala R Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[VBHS11] Juraj Vanek, Bedrich Benes, Adam Herout, and Ondrej Stava. Large-scale physics-based terrain editing using adaptive tiles on the gpu. *IEEE Computer Graphics and Applications*, 31(6):35–44, 2011.

[ZSTR07] Howard Zhou, Jie Sun, Greg Turk, and James M Rehg. Terrain synthesis from digital elevation models. *IEEE transactions on visualization and computer graphics*, 13(4):834–848, 2007.

Appendices

I. Settlement Name Generator

As initially discussed in chapter 4.3, settlements are assigned a pseudorandom name by selecting one of the prefixes and one of the suffixes from either the short names or combined names selection in table 5.

Table 5. Short and combined settlement name prefixes and suffixes.

Short Names		Combined Names	
Suffix	**Prefix**	**Suffix**	**Prefix**
Val	linn	Valge	salu
Tar	ge	Must	saare
Tal	va	Põllu	järve
Nar	ste	Saan	nõmme
Kur	na	Pai	jaani
Kaba	sa	Palu	vere
Pär	la	Kolu	metsa
Rap	si	Ada	jõe
Räp	nja	Ima	vee
Sal		Kure	maa
Põr		Tam	küla
		Lusti	kivi
		Väike	la
		Suure	mäe
		Vana	linna
		Kohtla	laane
		Kilingi	palu
		Paun	lepa
		Silla	kase
		Vastse	oja
		Metsa	oru
		Jaama	mõisa
		Järva	jala
		Soo	välja
		Männi	soo
		Harju	nurme
		Savi	nurga
		Pika	ranna
		Aru	taguse
		Kassi	
		Koera	
		Pudu	
		Taga	
		Uue	
		Mere	

II. Source Code and Build

The source code is available as an attachment and from a Bitbucket git repository[20] and a build is available as an attachment or from Google Drive[21].

[20] https://bitbucket.org/AndreasGP/mastersthesis/

[21] https://drive.google.com/open?id=1QSqh7hQItECjR_iY_IJRj1XpkMSdltfs

III. Glossary

Billboarding - a technique used to represent 3D objects far away as a simple 2D textured quad instead to save on performance.

Chunk window - A collection of neighbouring chunks, forming a square shape, that are used for the generation of the central chunks.

Context sensitivity - Property of infinite world generation that defines the maximum area around a point in the world that can affect the generation of features at the given point.

City center - the highway node used as the center of a city. When used plurally, also city subcenters are included.

City subcenter - a place in a city that serves as a secondary city center away from the main city center.

Curve - a continuous image of some interval in a 2-dimensional space in the context of this thesis.

Curve segment - Generated curves in the algorithm are stored as a list of points and rendered as linear line segments between these points. These line segments are called curve segments.

Exponential generation - a technique employed in this thesis to generate a nested chunk structure, where each different sized chunk can be used for generating different sized features.

Gradient noise - a computer generated visual noise, where each grid point has been assigned a gradient to control the neighbouring values.

GPGPU programming - General purpose graphics processing unit programming, which uses the GPU to do highly parallel generic calculations.

Heightmap - a regular 2D grid representing terrain, where each vertex has the height value at given point. Commonly stored as a greyscale image.

Highway - the largest type of road used in the thesis. Generated between macrochunks with up to one highway node per macrochunk.

Highway node - a point on macrochunk that is used as an endpoint for highway edges and curves.

Highway curve - a Bezier curve generated for an highway edge, that passes through the highway edge endpoints but has a more bent shape than the straight edge itself.

Highway edge - an edge connecting two macrochunks' highway nodes.

Highway junction - a highway node that has multiple highway edges.

L-system - short for Lindenmayer system, is a parallel rewriting system and a type of formal grammar, useful for generating repeating self-similar patterns.

Macrochunk - the larger generated chunk, that generates highways and city outlines, positions and names.

Mesochunk - the smaller generated chunk, that generates side roads, city internals and rural regions.

Noise - see gradient noise.

Parametric curve - a curve that can be represented as a polynomial function.

Region - a region of a chunk that has an explicit outline and a type assigned to it. For example, types can be *mixed forest*, *nature reserve* and *village*.

Side road - the second largest type of road used in the thesis. Generated between mesochunks with up to one side road node per mesochunk.

Side road node - see highway node.

Side road curve - see highway curve.

Side road edge - see highway edge.

Side road junction - see highway junction.

Spline - a large curve created by joining multiple parametric curves together.

Tensor field - a grid with a tensor assigned to each vertex. Tensors are geometric objects like vectors.

Voxel - a value of a regular 3D grid. Commonly visualized as a cube.

Quasirandomness - a seemingly random low-discrepancy distribution that is more uniform than completely uncorrelated distribution.

Window - see chunk window.

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Andreas Sepp**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis

   Infinite Procedural Infrastructured World Generation

   supervised by Raimond-Hendrik Tunnel and Eero Vainikko
2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 21.05.2018