FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Procedural Generation of Infinite 3D Worlds for Games

**Luís Miguel Coelho e Magalhães**


Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Tiago Pinheiro Neto Jacob

Proponent: Pedro Amorim Brandão Silva

June 27, 2017

# Procedural Generation of Infinite 3D Worlds for Games

**Luís Miguel Coelho e Magalhães**

Mestrado Integrado em Engenharia Informática e Computação

June 27, 2017

# Abstract

Procedural Generation is commonly used to develop massive content, in a parametrised way, saving time, money and labour, whilst providing an easy and intuitive way to automatically develop content. It has been used over the last few years for the creation of, for example, 3D objects, and realistic or artificial natural environments.

Although some recent digital games, such as *Minecraft* or *No Man's Sky* present infinite worlds generation, these lack complex structures, such as roads, rivers or cities. Furthermore, scientific studies on the area of procedural [world] generation are not yet focused on infinite generation, particularly the study of infinite game worlds.

Given that infinite worlds cannot be generated as a whole - since memory and expected generation times are limited -, those must be generated in a fractioned manner. As such, the main issue when generating these worlds is to maintain continuity and consistency of continuous elements that cross the borders of generated content, such as roads or cities. With this in mind, once this problem is overcome, the generation of infinite worlds might be applied to open worlds for digital games, virtual simulators (for example, combat of flight simulators), virtual tourism, urban planning, publicity or even decorative art.

This work presents a deterministic approach to procedurally generate an infinite and complex game world, battling the continuity issue that arises with infinite worlds and bridging the gap between the vast knowledge applied to finite environments and infinite procedural generation. It surveys the existing methods to procedurally generate game worlds and composing elements, presenting the methodologies and proposed approaches to solve continuity problems. Finally, a framework is created to generate these complex worlds.

The last chapters of this paper present the validations tests applied to the framework in order to prove its correctness as well as the drawn conclusions.

# Resumo

A geração procedimental é geralmente utilizada no contexto de desenvolvimento de conteúdo em grande escala, de forma parametrizável, de forma a poupar tempo, dinheiro e esforço, fornecendo uma maneira fácil e intuitiva de criar automaticamente conteúdo. Tem sido uma técnica utilizada ao longo dos últimos anos, por exemplo, para a criação de objetos 3D e ambientes naturais e artificiais complexos e realistas.

Embora alguns jogos digitais recentes, como *Minecraft* ou *No Man's Sky* apresentem geração infinita de mundos, estes não apresentam estruturas complexas, tais como ruas, rios ou cidades. Adicionalmente, os estudos científicos na área da geração procedimental [de mundos] ainda não se foca na geração infinita, particularmente no estudo de mundos infinitos.

Dado que os mundos infinitos não podem ser gerados de uma só vez - pois a memória e os tempos de execução são limitados -, têm necessariamente de ser gerados de forma fracionada. Como tal, o principal problema que surge aquando da geração destes mundos é a falta de continuidade que pode existir em elementos, como ruas ou cidades, que ultrapassem as margens do conteúdo criado. No entanto, após resolvido este entrave, a geração de mundos infinitos poderia ser aplicada a jogos digitais *open world*, simuladores de guerra, ou voo, ou ainda a elementos estéticos e artísticos, como paisagens virtuais.

Este trabalho apresenta uma solução determinista para a geração de mundos de jogo infinitos e complexos, que resolva o problema de continuidade mencionado anteriormente, aplicando o vasto conhecimento na área de mundos finitos em conjunto com geração procedimental infinita. É apresentado um resumo sobre a geração procedimental de mundos virtuais, assim como os elementos que os constituem. Mais à frente, são descritas as abordagens propostas para resolver os problemas apresentados, assim como o programa densenvolvido para a geração destes mundo complexos.

Nos últimos capítulos do trabalho são apresentados os testes de validação que foram desenvolvidos a fim de provar a correção da solução, assim como as conclusões retiradas.

# Acknowledgements

To my supervisor, João Jacob, I would like to thank for the ever present support and experience, for helping me with tips regarding my work and its approaches. To my thesis proponent, Pedro Silva, I would like to thank for the support and brainstorming both while we worked together and afterwards. Without the help and contributions from both of them, this work would not have achieved half the quality of the final results.

I would also like to thank both the Faculty of Engineering of University of Porto and ObviousYellow LDA (where I conducted my internship) for having provided me with the means and workspaces for the development of this work.

To my family, I give my eternal appreciation for the help during the months that it took to complete this dissertation, which helped me keeping optimistic towards my work.

Last, but not least, a warm-hearted thank you to my girlfriend, for not giving up on me during this time of neglect and long-hour sections of programming nightmares and hair removal.

Luís Magalhães

# Contents

# List of Figures

LIST OF FIGURES

# List of Tables

# Chapter 1

# Introduction

This chapter introduces some notions about the work to be conducted, as well as providing insight on the problems that this work attempts to solve, as well as the related research questions and main objectives. In the end of the chapter is described the general structure of the paper.

## 1.1 Motivation/Context

Procedural generation is "the application of computers to generate [...] content" [HMVI13] through abstraction and parametric control. It is used to create massive content efficiently, and can be very effective in reducing costs [KM06] that come from the increasing need for more content, especially when applied to games.

This kind of generation relies on methods - or procedures (hence the name) - to generate content. The methods strictly follow **construction rules** and **constraints**, and can be manipulated through **input parameters**.

Procedural generation is a rather well-discussed area [STBB14, HMVI13, TYSB11], both in the scientific and academic communities, as well as in general society, particularly amongst game developers. However, procedural generation applied to infinite environments is an area open to investigation, with very little documentation regarding scientific studies on the matter, specifically when applied to complex and deterministic game worlds.

The massive content generation allows the game player to explore the world feeling as if the experience is always novelty (or so it is intended) and, given its infinity, every bit of the world is different, unique and never-ending [HMVI13]. Furthermore, infinite content generation allows for endless replayability and exploration of the generated worlds.

Apart from that, there can be more than game applications: it can be applied to virtual simulators [STDB10]: *Flight Simulator* and *ARMA* are a flight and a combat simulator, respectively, which use real world data for their world generation. However, they could be extended to use procedural generation to generate infinite and non-repeatable environments, providing, for example, unlimited content for as many training sessions as required, or even large enough areas for specific trainings.

Infinite procedural generation could also be applied to virtual tourism, publicity, landscape generation [HGA+10, PGGM09] or decorative art. To provide a concrete example of where decorative art could have been applied, the *Star Wars* saga might have benefited from infinite generation to create all the fictitious worlds in it present.

## 1.2 Problem

The main problem for infinite generation in game worlds lies in the finite nature of the storage elements: computer memory has its own limits; as such, in order to generate infinite continuous worlds, they must be created by parcels, providing only the required information to fill the world in a given range of the player[1].

However, as the infinite world is now composed by finite parcels, which are generated independently when required, **border and continuity issues** arise (see Figure 1.1). For example, how to create rivers that go through more than one parcel, if the parcel it extends to has not yet been generated or accounted for? Likewise, how to guarantee that the information in the border of one parcel exactly matches the information in the opposite border of the neighbouring parcel, so as to avoid gaps and discontinuities which could break the illusion of an infinite and seamless world?

This is where determinism can be helpful: besides providing a base of expectations for the designer creating the world, it can serve as a predictive aid for the generation system, allowing it to "know" where an element will be, even when the element has not yet been built.

Given the above information, the problem can be presented mainly as a two-fold: the world must be generated parcel by parcel and each parcel can have no discontinuity issues with its neighbouring parcels, forming a consistent environment all-round. Furthermore, the generation process must be deterministic and dependent on the world designer, by manipulation of input parameters.

Finally, it is very important to understand two notions: **a)** consistency persists across executions: the same input parameter combination generates the exact same world in every execution for the same locations; **b)** even if the position in the world from which the generation starts changes, same locations should always obtain the same world elements.

## 1.3 Research questions

Taking into account the previously laid information, the research will attempt the following four-fold question regarding infinite generation of worlds: how to infinitely generate a game world that is **complex** and **deterministic**, whilst maintaining **consistency** and **continuity**?

For the purpose of this work, a complex game world can be considered to contain four main structures: **surfaces** (terrain), **boundaries** (oceans, lakes, urban lots and forest areas), **networks** (roads and rivers) and **nodes** (trees and buildings). All elements must be created so that they are compatible and cohesive with each other (and coherent along the game world, as mentioned

[1] For the purpose of this work, player is an entity placed in the game world, being able to observe and transverse the world as a human being would.

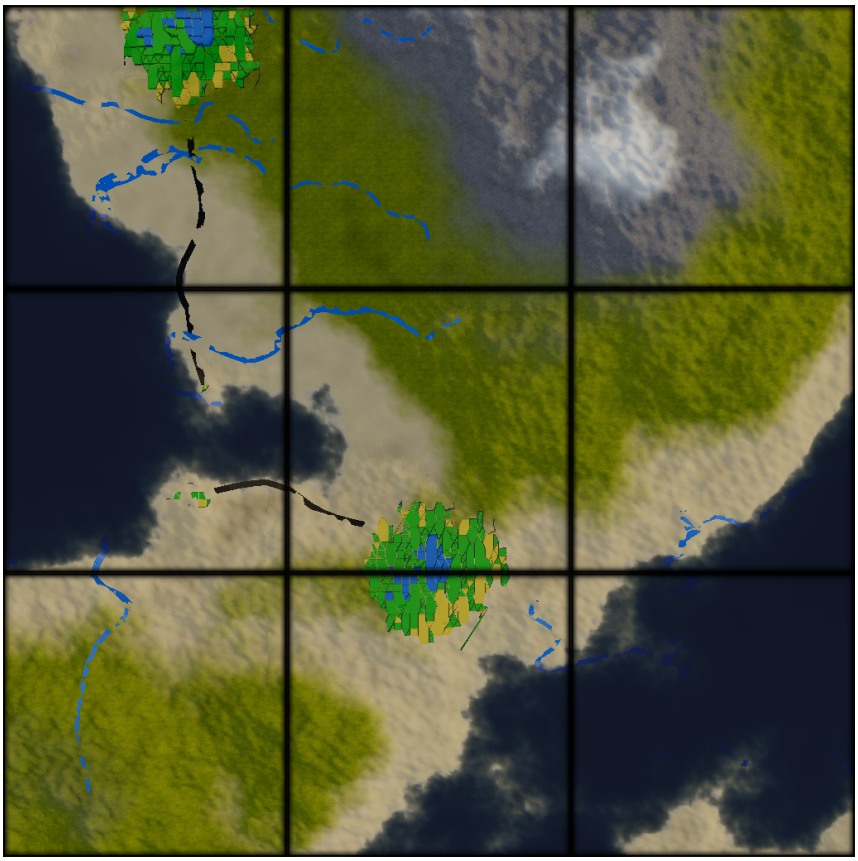

Figure 1.1: The black grid represents the parcels borders. Black lines represent highways, whilst blue lines represent rivers. The small areas coloured yellow, green and blue represent cities. The remaining environment is constituted by terrain and ocean.

before). As parcels of the world are generated, elements that cross parcels must be continuous. For example, a river cannot hit a parcel's edge and find no continuation in the next parcel - this represents lack of consistency and continuity.

Determinism will be enforced, as a way of creating predictable and controllable generation procedures, so that the results depend solely on the input parameters, as expected by the world designer.

## 1.4 Objectives

The purpose of this work is to provide a framework which implements infinite generation of game worlds as described previously. As such there are some objectives that are expected to be achieved by the end of this work:

- Creation of modules that procedurally generate:
  - Surfaces (terrain);
  - Boundaries (forests, oceans, lakes, building lots...);

  - Networks (roads and rivers);
  - Nodes (vegetation and buildings).

- Creation of a working prototype which allows:
  - Procedural generation of an infinite game world, by parcels, across time;
  - Tweaking of input parameters so as to change the resulting world (for example, changing the terrain height range, changing the river and city probabilities and sizes, etc.).
- Creation of an "exploration" mechanism which allows travelling, saving and loading the world's parameters and the player's position;
- Creation of a module that evaluates and validates the continuity and consistency of the generate game world.

## 1.5 Structure

Besides this chapter, the dissertation is structured as follows:

- **Chapter 2** presents an overview on the literature and methods presented up until now regarding procedural generation of world elements and game worlds;
- **Chapter 3** discusses the design of the framework and the generation algorithms, as well as the caveats found, presenting solutions;
- **Chapter 4** describes and discusses the implementation of the framework and presents the visual results of the same;
- **Chapter 5** discusses the methods used for the evaluation and validation of the generated world, presenting and discussing the final test results.
- **Chapter 6** draws the conclusions and final remarks for this work, while introducing some notions for future work.

# Chapter 2

# State of The Art

## 2.1 Introduction

The following sections present a revision on the state of the art for procedural generation of game worlds and game world elements[1]. It will be divided into 4 main categories: **surfaces**, **boundaries**, **networks** and **mixed**.

Although only these 4 categories will be expanded upon, a 5th category is implicit: **nodes**. Nodes represent individual points and can be used to represent simple structures, such as buildings, or as a geographical landmark for more complex structures.

## 2.2 Surfaces

In the context of this work, only terrain will be considered a surface, as it is the only real infinite surface, presenting the basis upon which every other finite element builds itself.

### 2.2.1 Terrain

Terrain generation is generally the first element to be used when attempting to simulate a game world. This is because so many elements (rivers, trees, streets, cities) depend upon the underlying terrain and its **topology**. However, there are biological factors that also influence the positioning and presence of certain types of trees, for example, whether or not a place is suitable for building a city (for example, an arid land is generally not appealing to human beings as a potential place to build a settlement). These biological factors will be represented as **biomes**.

[1] Along the sections, there will be numerous references to Perlin Noise. Being a particularly good method to obtain pseudo-random distribution of values [Per85, Per02], this type of noise function will be used, amongst others, to distribute elements and, for example, as source for elevation data. Any mention to Perlin noise, unless noted otherwise, will be referring to it in the $2^{nd}$ dimension, that is, as 2D noise.

#### 2.2.1.1 Biomes

Biomes[2]are used in the context of biology to describe "major terrestrial or aquatic life zones, characterized by vegetation type in terrestrial biomes or the physical environment in aquatic biomes" [CRU+09]. As said before, terrestrial biomes are classified based on vegetation type and can be determined based of annual precipitation, seasonal temperatures, elevation and disturbances - "disturbance, an event (such as a storm, fire or human activities) that changes a community, removing organisms from it and altering resource availability" [CRU+09].

Campbell describes several aquatic biome types but, for the sake of simplicity, only 3 will be considered in this work: **oceans**, **lakes** and **rivers**. As for terrestrial biomes, table 2.1 displays the biome types and their general characteristics, as described in Campbell's work.

Table 2.1: Terrestrial biomes, as described by Campbell [CRU+09] (chaparral has not been considered due to its generally low distribution rate and extremely specific conditions).

| **Biome** | **Subtype** | **Annual Precipitation (cm)** | **Temperature Range (°C)** |
|---|---|---|---|
| Tropical Forest | Rain | 200-400 | $25-29$ with little variation |
| | Dry | 150-200 (5-7 dry months) | |
| Desert | | $<30$, with high variations | $-30$-50 |
| Savanna | | 30-50 (8-9 dry months) | 24-29 |
| Temperate Grassland | | 30-100 (dry winter, wet summer) | $-10$-30 (cold winter, hot summer) |
| Taiga | | 30-70 (winter snow) | $-50$-20 (cold and long winter) |
| Temperate Broadleaf Forest | | 70-200 (winter snow) | 0-30 (cold winter, hot summer) |
| Tundra | | 20-60 (100 at high elevation) | $-30$-10 (cold and long winter) |

Figure 2.1 (first suggested by Robert Whittaker) shows also the observed relation between the levels of precipitation and temperature in a given area, and the underlying biome, and will be used as reference for parameter extraction.

[2]This section contains information that is in its majority pertaining to the branch of biology. As a result, such information might not particularly complete, but, since it will only be used for reference for extraction of possible parameters to classify areas of the game world, this acknowledged deficiency of precision will not be considered.

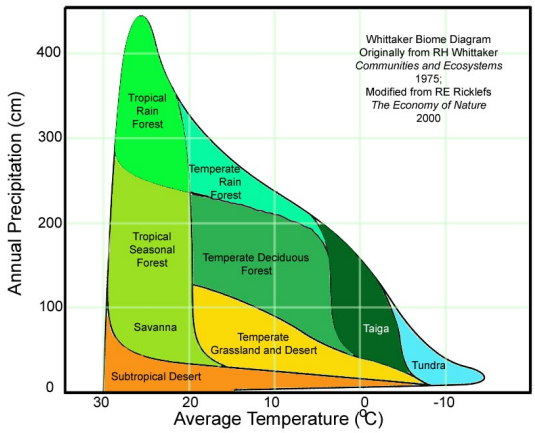


Figure 2.1: Graph proposed by R. Whittaker that represents the observed relation between precipitation, temperature and the resulting biome.

#### 2.2.1.2 Topology

Topology, in simple terms, represents the geographical perturbations in a terrain, such as mountains and valleys. As mentioned in his work, Smelik [STBB14] introduces a series of structures to represent a terrain topology, the most common being a 2D grid (or *heightmap*) in which each value in the grid matches the elevation of the vertex on that position. There are several algorithms to create *heightmaps* [SDG+09], such as the mid-point displacement method, fractal noise (of which PN is an example - see Figure 2.2) and erosion simulations (see Figure 2.3).

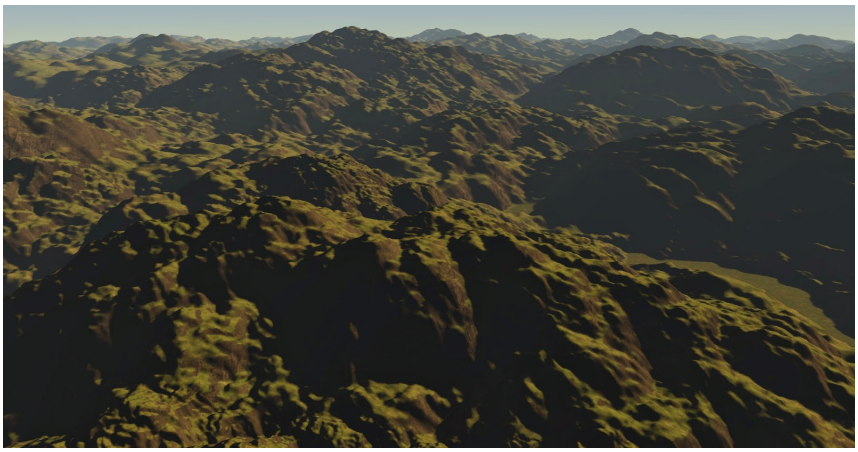

Figure 2.2: Terrain generated by Perlin noise

However, as mentioned in both papers, this representation (*heightmap*) presents several limitations, being incapable of dealing with complex structures, such as caverns or overhangs, being the suggested alternatives [STBB14]: layered data structures, *voxel* data or 3D meshes. In 2009, Peytavie [PGGM09] presented in his paper the platform of terrain generation *Arches*, which allowed

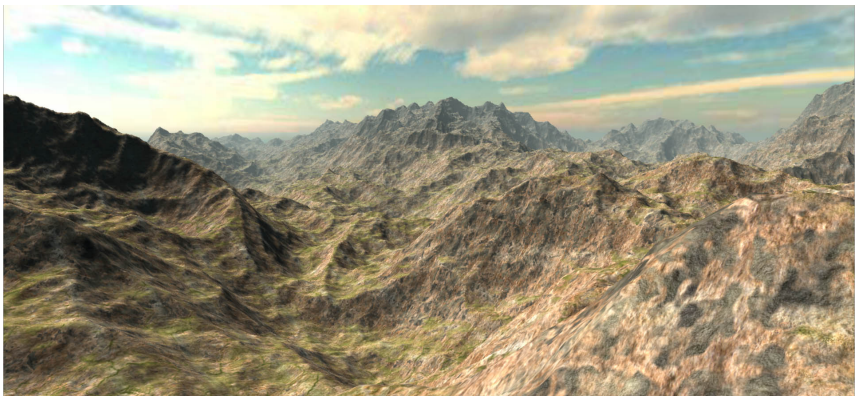

Figure 2.3: Terrain generated by fractal noise and erosion methods

the creation of the aforementioned complex structures. In Figure 2.4 we can observe the results of *Arches'* generation of complex structures such as caverns, canyons and overhangs.

Figure 2.4: Complex structures generated by *Arches*

In his work, Hnaidi [HGA+10] presents an alternative to the generation of the terrain: a combination between user-defined splines, a limited number of deformation features and a diffusion equation to generate river beds, mountains, ridges or cliffs, amongst others (see figure 2.5).

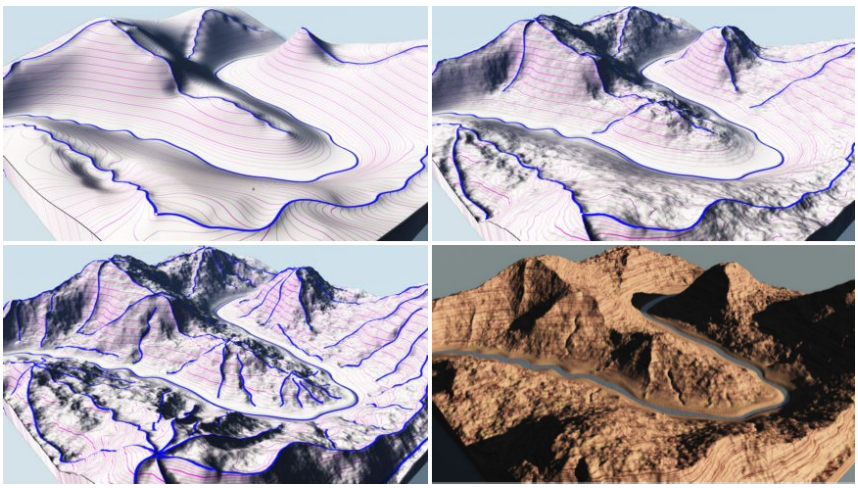

Figure 2.5: Iterations of the process, as presented by Hnaidi [HGA+10].

Another way of generating terrain elevation is to think about it as a result of other features.

That being so, terrain adapts itself to the existing features, and not the way around. This is the approach presented by Génevaux [GGG+13]. In this work, terrain is generated after the river network, and it is carved in respect to the existing river lines.

Lastly, terrain can also be generated from real-world elevation data [Par14]. This data is obtained, generally, from satellite imagery as a *heightmap*, and used *as is* to be directly mapped to the generated terrain as elevation, producing a realistic result, but still limited by the handicaps that come from using the *heightmap* technique .

## 2.3 Boundaries

Boundaries refer to any element that can be represented as a bounded area, or a closed polygonal area. With this in mind, anything from oceans and lakes, to forest and building lots can be considered boundaries.

### 2.3.1 Oceans and Lakes

As mentioned by Smelik [SDG+09] in his survey of procedural methods for terrain modelling mentions a notion - corroborated by this work's research: there has been little attention regarding the procedural creation of features such as oceans and lakes, and there are little methods on how to generate those. This may be due to the simplistic nature of these masses, but a further research on the matter is necessary.

**Oceans** are generally created using a fixed water elevation, beneath which every parcel is water, or by using flooding algorithms from a low elevation point [SDG+09]. **Lakes**, as oceans, can also be generated by using a flooding algorithm starting at given elevation point, since they can be seen as small bounded oceans.

### 2.3.2 Forests

Forests might be thought in terms of closed polygonal areas [STDB10], rather than individuals. So, once those areas are well defined, one might then decide where to instantiate particular trees or tree groups, considering the parameters under evaluation. Here, one can consider the parameters mentioned in section 2.2.1, such as elevation (and associated slope), temperature and resulting biomes as the defining characteristics to which kind of vegetation is present, or whether it should or not exist in that area. In [STDB11], this polygonal areas are defined as mentioned above, and are subject to almost every other feature. This is show in Figure 2.6.

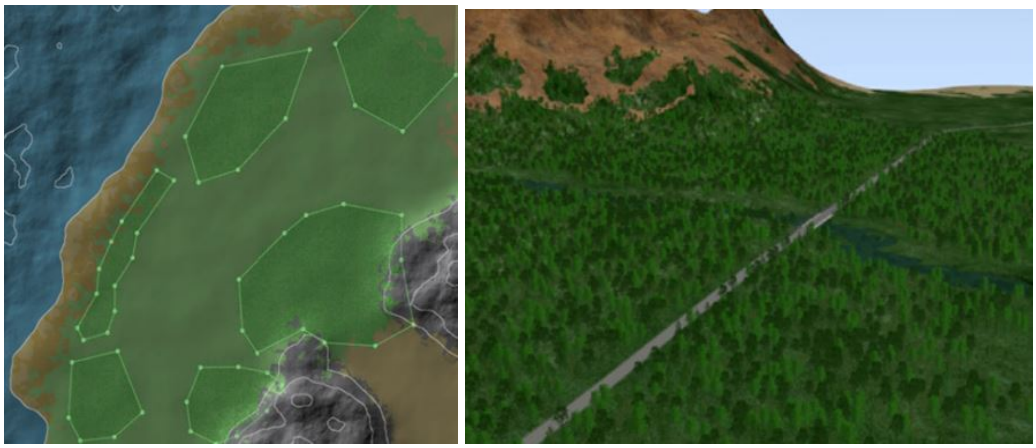

Figure 2.6: On the left, a polygonal distribution of forests [STDB10]. On the right, the resulting environment. Notice the forest's adaptation to its surroundings.

.

There are two ways to generate forests: they are either generated after the terrain and all elements, and are subsequently subject to alteration/destruction by later features, or they are generated as the last elements and are forced to respect already existing features, altering or deferring their own placement [STDB10].

## 2.4 Networks

Networks are all elements that can be represented as graphs, or ways, and will be divided into 2 subcategories: roads (urban and rural) and rivers, as these present the most abundant and observable networks in the real world. It could also be considered the creation of road variants, such as railways, but since they can be achieved from the former, those will be left to future work.

### 2.4.1 Rivers

River procedural generation usually requires either the **river source**, the **river mouth**, or both, and is generally created based upon path-finding or physic-based algorithms - which can be as simple as finding the nearest low or high elevation point, depending on whether we start at a river source or a river mouth.

#### 2.4.1.1 River source

A river source is fairly intuitive to place: it has to be an elevated point, from where the river can flow, affected by gravity. However, to avoid placing dozens or hundreds of rivers, these points are generally filtered from a set of elevated points that we might be presented with in, for example, a mountain range.

Having selected the river source, one can then choose which method to use in order to guide the river flow down the terrain. There have been some approaches in this area. For example,

Audibert [BA05] presents an approach which takes into account a physics-based model. From randomly selected elevated points, particles are "dropped", going down the terrain as if affected by gravity. Along with an associated friction coefficient, particles follow the terrain in a quite realistic way, producing satisfying results (see Figure 2.7).

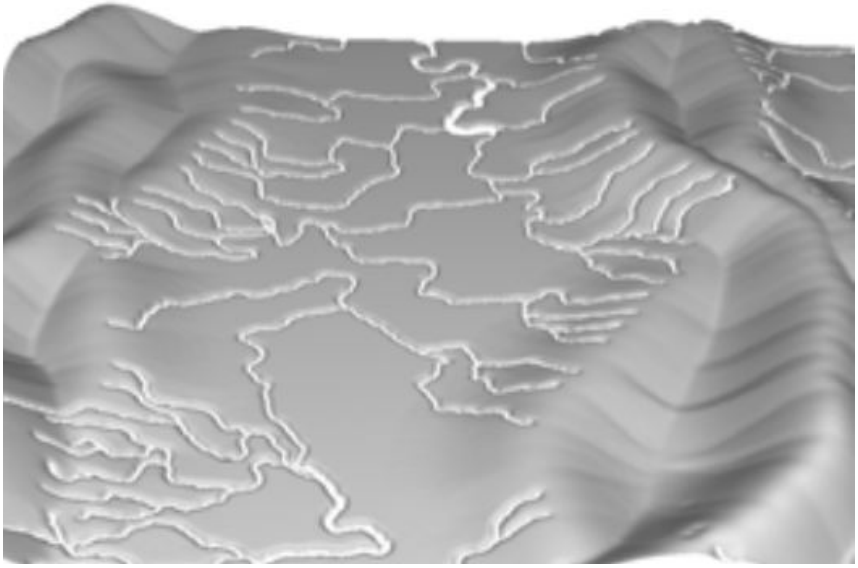

Figure 2.7: Physics-based approach to generate a river network

Other approaches include using either a greedy or an A* search algorithms with the purpose of decreasing elevation. This approach trades the physical realism result for performance of computation, although the results are quite good, especially when using A* in conjunction with PN or Midpoint displacement methods as a way of introducing some pseudo-randomness to the results.

#### 2.4.1.2 River mouth

If a river-mouth approach is chosen, the first step is to define a starting point, from a low elevation set of points. [3] After it has been chosen, its flow can be calculated by using a reverse approach of the previous techniques, trying to find the next points in such a way that the river flow upwards, towards it source.

A relevant difference to the river-source method is that, rather than joining rivers as they flow down (as it happens in the real world), rivers split themselves as they flow, generating sub-rivers that continue to flow until they find a sufficiently elevated spot. This is the approach used by Genevaux [GGG+13] and, as can be observed in figure 2.8, it produces a complex river network.

[3] Again, as in the former technique, the difficulty remains regarding the choosing of this point in a controlled way

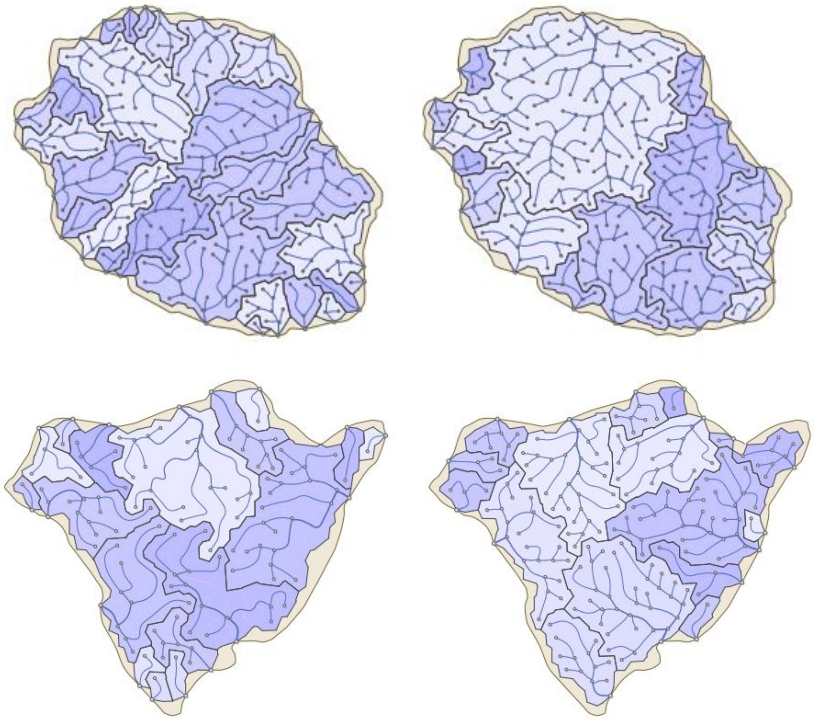

Figure 2.8: River networks generated using a river-mouth approach

#### 2.4.1.3 River delta

There is very little mentioning of the creation of river deltas in most papers that relate to procedural generation of rivers. However, Teoh [Teo08] does shortly describe a simple dispersion method based on proximity to the river mouth (or to great water masses) that generates a realistic river delta.

### 2.4.2 Roads

#### 2.4.2.1 Urban Network

Urban networks comprise all the inner-city networks. There are many different methods to build urban networks [SYBG02, LHD15, KM06, KM07, Gro09, LWWF03], such as **Grid Layout**, **L-Systems**, **Template Based Generation** and **Agent Based Simulation**. Some of the methods use notions such as **districts**, **land use** and **hierarchic networks**. All the highlighted terms will be explained further on.

Additionally, Groenewegen [Gro09] describes how patterns in cities can be a result of local and historic developments. He provides two general examples: western European and north American cities. In his work, he mentions how European cities are organized around landmarks, and generally economically grade down from the historic centre. He also mentions that this historic centres generally result from mercantile guilds, feudal rule (and concentric castle architecture), royal

palaces or historic markers, around which everything was built. In contrast, he also describes how American cities have better defined districts than European ones, still with an economic gradient, from the suburbs to the city centre, where most of the social, business and commercial activities are held.

Most of the works on this area mention also some of the patterns that can be encountered in cities[4], such as **radial**, **raster** or **population-based** patterns [KM06, Gro09, LHD15, SYBG02]. The city of Paris, as shown in Figure 2.9, is a perfect example of a radial network of a city that grew around an historic marker. However, the city of New York (see Figure 2.10) presents a very different layout. Its a major example for raster pattern, with long parallel and orthogonal lines, displaying a grid-like pattern.

Figure 2.9: The "Arc de Triunf" in Paris. A perfect example of how a city grew in a radial pattern around an historic marker.

**Districts** They are seen as a city's underlying organization system [Gro09] - each district deriving from the activities practised there or as a division of economic power - for example, separating high, medium and low-class residential areas.

[4] Albeit these works generally focus either on western European of north American cities.

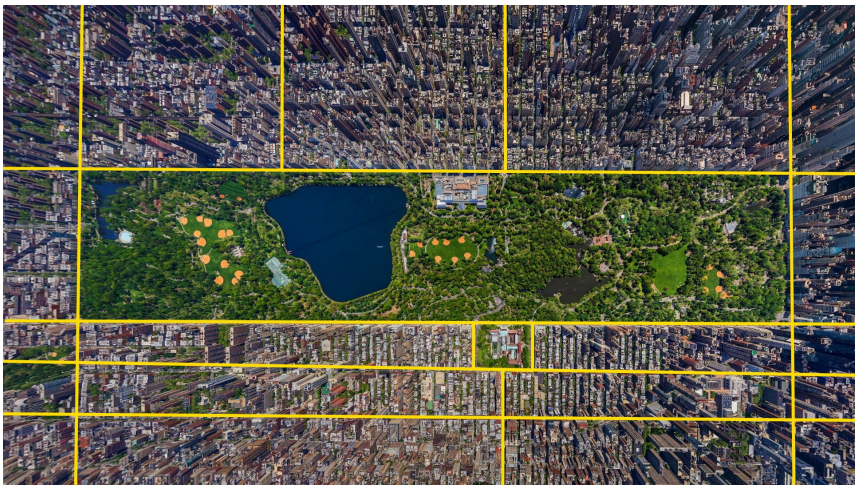

Figure 2.10: New York displays a perfect raster pattern, characteristic of north American larger cities.

**Land use** Land use [LWWF03, LHD15, SYBG02] derives from the purpose to which a portion of land is allocated to: administrative, commercial, residential and industrial are some examples present in the referenced works.

**Hierarchic networks** Hierarchic networks refer to the hierarchical organization that can be observed in cities, and can be roughly summarized in highways and secondary roads [LWWF03, KM07][5].

**Grid Layout** The grid layout is as simple as it seems and applies the raster pattern: each road is either parallel or perpendicular to the existing, forming a checkers pattern.

**Template-based generation** Template-based generation, as presented by Sun [SYBG02], relies on the integration of previously-defined road templates, such as population based (which resembles an evenly-distributed voronoi graph), radial, raster (earlier presented as regular or grid pattern) and a mix between the previous. This method generates sparse networks (see Figure 2.11), an can be seen as a representation of more important roads, such as inner-city highways. However, it fails at generating convincing denser networks, although other methods, such as L-systems or distorted grid patterns might do the work left.

**L-Systems** Lindermayer-systems (L-systems) consist on a set of construction rules and constraints. As the object is being built, construction rules are chosen randomly (using probabilities, for example) from the available ones, provided they do not break any constraint, resulting in organic systems which are many times used to represent natural occurrences. As mentioned by Kelly [KM06]:

[5] Additionally, streets that serve specific or very small areas might event branch secondary into tertiary roads.

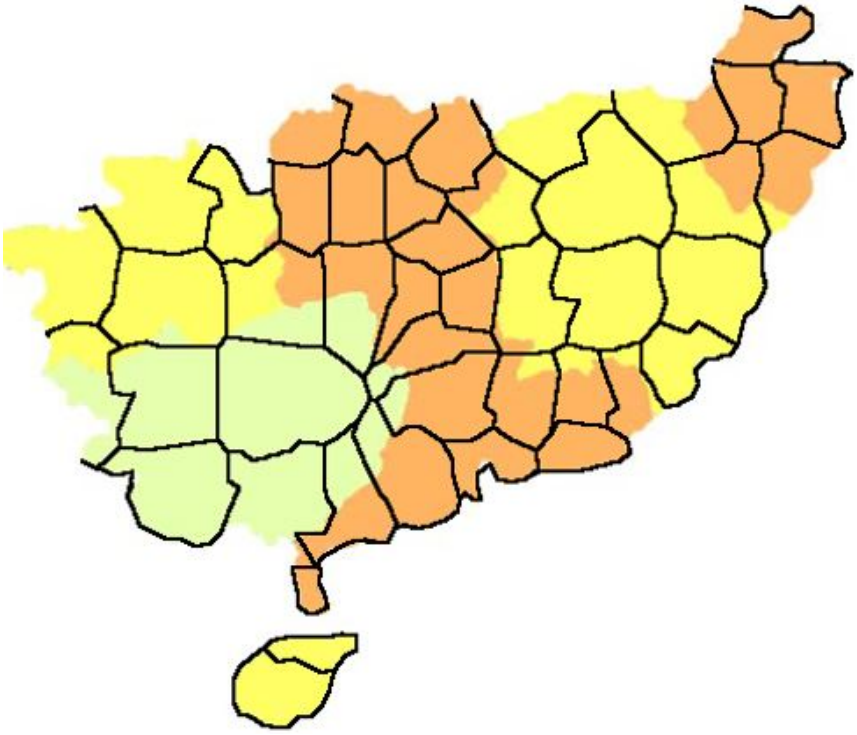

Figure 2.11: Template-based generation, as presented in Sun's work.

> "Lindenmayer-systems have traditionally been used to model natural phenomena but are also suitable for the generation of cities due to their concise nature, computational efficiency and data amplification properties."

Thus, this approach can be used to create road network, and represents a scalable and efficient method. In 2001, Parish and Müller presented a tool named *CityEngine* [PM01], which used a L-System approach to the road network generation, obtaining believable results (see Figure 2.12). This may be due to the fact that the branching of trees and leaves sometimes resembles the branching of main highways into smaller roads. The advantage that L-systems provide regarding the last method is that they are very malleable, since construction rules and parameters can be changed as desired, generating very distinct results.

**Agent-based simulation** Agent-based simulation is an alternative to the previous methods, which takes into account city development, mainly the division between commercial, residential and industrial areas, and their distribution rules. The idea behind this approach is that agents roam around creating road networks and road connections, whilst other agents populate those areas to serve either commercial, residential or industrial purposes, thus simulating the actual growth behaviour of a real city.

This is the approach presented by Lechner [LWWF03] and renders pretty good results in terms of realism, as seen in Figure 2.13. However, as mentioned by Kelly [KM06], despite presenting

Figure 2.12: A resulting roads network using L-systems, as seen in Parish's work.

realistic results and effectively blending regular and organic patterns, this approach is computationally intensive and, for the purpose of procedural generation, inefficient (the layout presented in figure 2.13 takes about 15 minutes to be generated and its size can be compared to a small village).

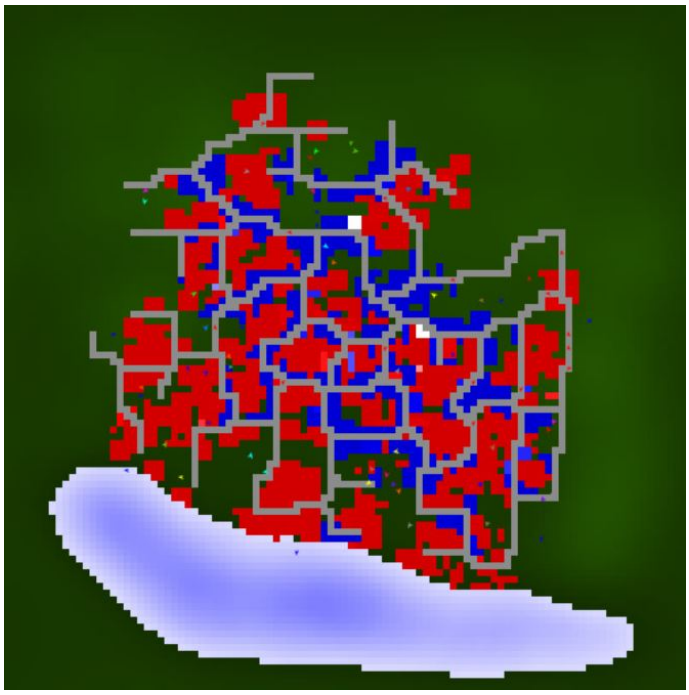

Figure 2.13: Red indicates commercial areas, blue residential ones.

#### 2.4.2.2 Rural Network

A rural network differs from a urban network mainly in the purpose it serves. Rather than connecting different streets and focusing on providing a network to that covers the entirety of the city, a rural network exists to connects cities.

As observed by Galin [GPMG10], in his approach using anisotropic shortest path algorithm, these roads can take different behaviours: the can either adapt to the surroundings - for example, bending to follow a ridge, mountain or river line - or attempt to cross the terrain with the shortest possible length - creating tunnels and bridges (these might be more expensive, though). For these results, one might consider slope, tunnel and bridges cost per distance unit and forest building cost. The results of his algorithm are shown in figure 2.14.

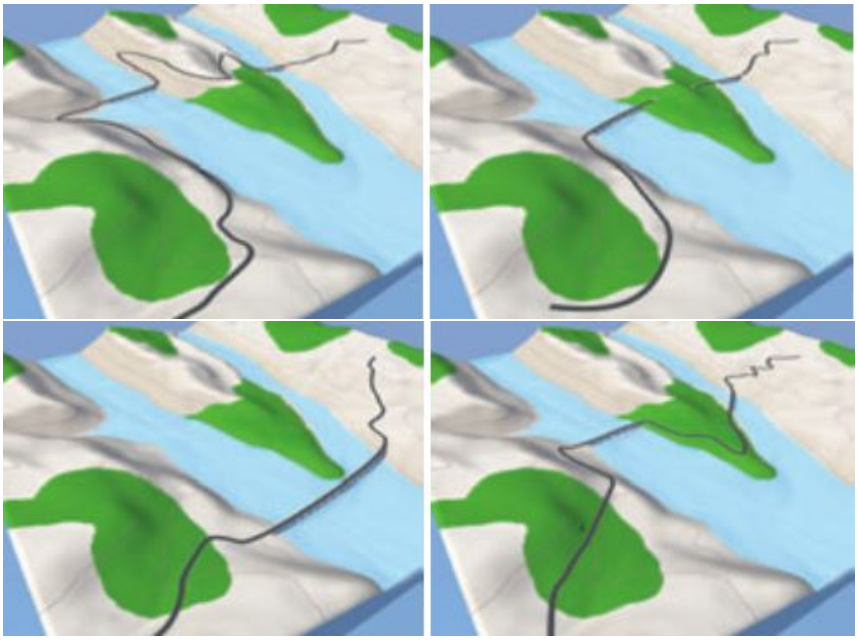

Figure 2.14: On the left are the results obtained by changing the slope cost function (tunnels and bridges). On the right are the results obtained by changing the cost to build with forest.

## 2.5 Mixed

Mixed-type element are those that comprise some or all of the previous categories. Cities are an example of a mixed element: far as **cities** are concerned, they are constituted by networks (streets), boundaries (building lots, parks, etc.) and individual nodes (such as buildings).

### 2.5.1 Cities

There are several approaches on how to build a city and its buildings, as described by the survey by Kelly and McCabe [KM06]. It mentions agent-based simulation, template based generation, L-systems and grid layouts. These methods were earlier described in urban networks 2.4.2.1. This is due to the fact that in many papers road networks are the first step to creating a city.

In 2003, Lechner [LWWF03] presented an approach based on agent simulations, where he mentions 2 main types of agents: road and land use developers. These agents work closely with each other, providing local info. So, for example, if a road agent encounters an unserviced patch of land, it tries to connect that patch with a road segment to the existing road network. Land use developers travel within the road network coverage and try to find unallocated land patches, that might be used for commercial or residential purposes, taking into account population density and land value (see Figure 2.15).

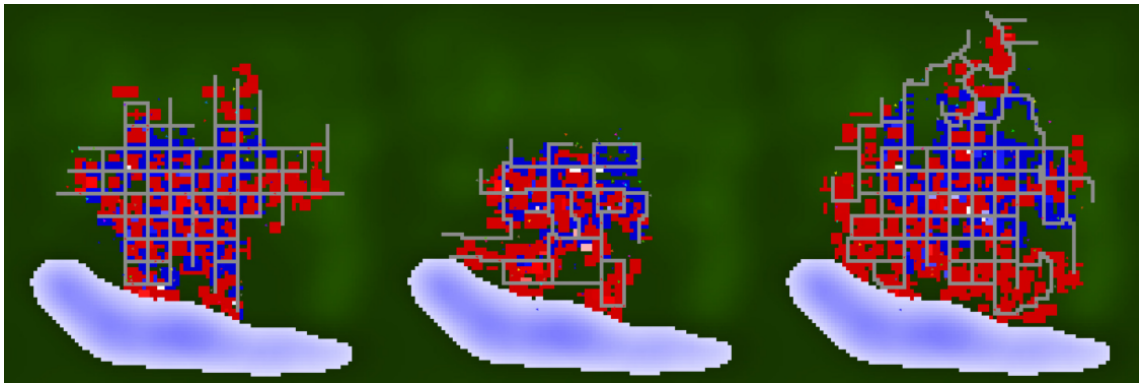

Figure 2.15: Red indicates commercial areas, blue, residential ones.

Another approach is proposed by Greuter [GPSL03], in which he describes a city as a grid layout and buildings as sets of geometric primitives, and presents a framework that generates a city using this assumptions. The results can be seen in Figure 2.16, and entitle a very noticeable grid pattern, which might resemble a raster pattern mentioned before in urban networks (see Section 2.4.2.1).

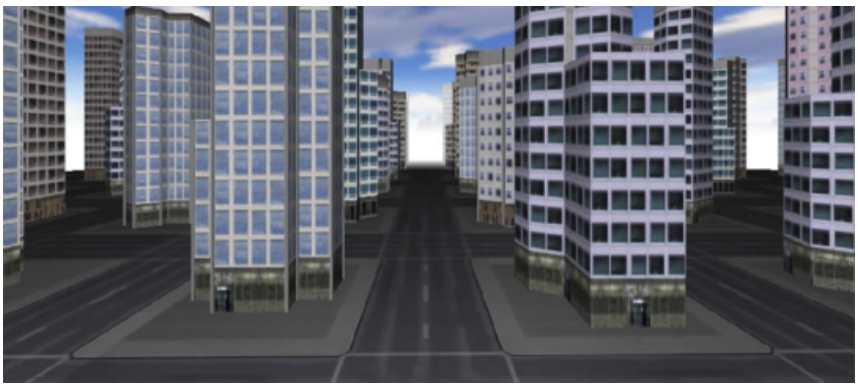

Figure 2.16: Greuter's grid layout city.

As mentioned before, city networks can be created by L-systems. Likewise, L-systems with different construction rule-sets can be used to procedurally create buildings. This is the approach presented by Parish and Muller [PM01], where they introduce the term *self-sensitive L-systems*, which are used to create the road networks. These can be influenced by population density or superimposed patterns. After the road network is built, the closed areas (encircled by roads) are

then divided into lots (see Figure 2.17), where buildings shall be created. The geometry of the buildings also relies on L-systems, with 3 separate rule-sets: skyscrapers, commercial building and residential buildings. Figure 2.18 demonstrates the result of a city with 26000 buildings and corresponding road network. This kind of generation seems to produce believable results. Of course, building generation based on L-systems might not be a scalable solution, since very different architectures require different rule-sets. In 2006, Muller [MWH+06] presented a solution to efficiently create buildings relying on shape grammars. However, same problem applies, since a shape grammar is, as an L-system, consistent of a set of rules which much be changed to produce distinct results.

Figure 2.17: Division of road network in lots.

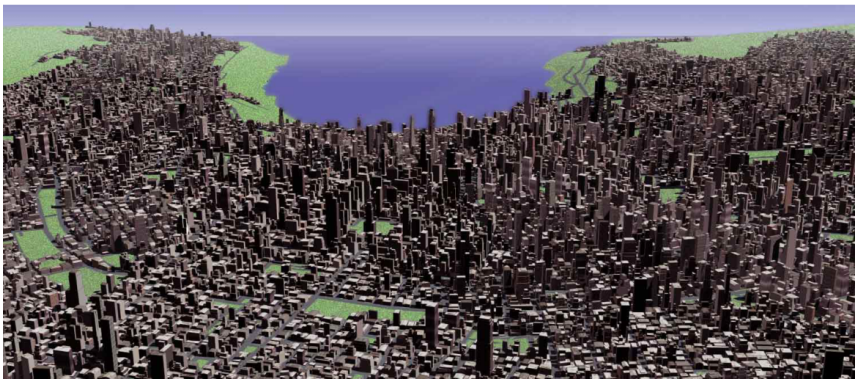

Figure 2.18: L-system generated city

Recently, Lyu [LHD15] proposed an approach based on hierarchy. First, the population density is analysed in order to determine where highways and arterial[6]roads shall be placed. Then, The enclosed lots are divided into districts, which are then allocated according to land use, and a secondary network is created to serve this smaller divisions. These lots will result in neighbourhoods, which are again subdivided according to the land use, and a new tertiary network is created to serve the local accesses, which are divided into slots and given land uses of their own, resulting in a complex city model.

[6]As in the human body, arterial here refers to important and large roads.

## 2.6 Infinite Procedural Generation of Worlds

The research for this work revealed no relevant documents on this particular topic. As such, it is reasonable to assume that any approach taken regarding the generation of infinite worlds can be considered a novelty, from a scientific point of view.

## 2.7 Summary

As it was seen in the previous subsections, there is a large number of methods for the procedural creation of terrains, bounded areas, networks and cities.

Despite the observable realism of some of these methods, there are some which might present themselves as **overly complex** (such as layered data structures or voxel data for terrain representation) or **time-consuming**, considering the objectives of this work (for example, the generation of the roads as seen in Figure 2.14, following Galin's method, took approximately 15 minutes. The generation of the small city network by agent-based simulation in Figure 2.13 took 30 minutes).

Although Hnaidi's [HGA+10] approach produces interesting results (see Figure 2.5), its dependency on user interaction in order to create the splines might not be particularly interesting for this work, as it pursues complete procedural independence in the generation phase. However, maybe an accommodation could be made in order to produce the splines automatically and then interleave with Hnaidi's process.

Another interesting approach to terrain and river generation was Génevaux's. The concept of blending and carving could certainly be used in this work, as a way of adapting terrain to presented features and vice-versa. However, the idea of creating a river network before the terrain might not be of particular interest, since it might limit the actual roughness of the terrain and appearance of mountain ranges.

Regarding procedural generation, it has been observed [HMVI13, STBB14] that most of the time the lack of control of the resulting features can be fought by providing more natural and intuitive methods, so that users can better control the generated content. However, a great downside does exist: the methods become more dependant on user interaction and, thus, less autonomous. Seen as this work intends to provide a large independence in the matter of the generation itself, one of the challenges to overcome will be the ability to provide as many variables as possible, whilst maintaining generation an essentially black-boxed process.

Considering the last subsection, it is apparent that infinite generation of worlds is an underdeveloped study area in the scientific community. However, one might take into account results produced in non-academic circles, such as digital games mentioned before (for example, **Minecraft** and **No Man's Sky**) - which visible apply infinite world generation - as a base from which deductions and inferences can be made, in order to assist the production of scientifically correct results.

# Chapter 3

# Solution Design

This chapter presents any relevant decisions, conclusions, approaches and algorithms that define the solution design regarding the world generation framework, as well as the tools that were used in the development of the framework. Algorithms presented in this chapter have been written in pseudo-code, so as to maintain the logic underneath with a more readable syntax. The last section summarizes the content of the chapter.

As mentioned before in section 1.2, the concept of infinity does not refer to actual infinite worlds, but to the generation of the worlds themselves, where more terrain and features can be requested indefinitely to create a perception of infinite environments. Another important characteristic of this system is the ability to allow for this subsequent generation to blend seamlessly with the already created world elements and terrain.

## 3.1 Chunks

As mentioned in section 1.2, infinite worlds cannot be fully generated at once, and must be divided in parcels, which are created and destroyed as necessary.

For the purpose of this work, these parcels are called **chunks**. A chunk is a cubic area that covers a dimension of $k \times k \times h$ in 3D space, where $k$ is the chunk's horizontal dimensions and $h$ the difference between the chunk's lowest and highest points. Since the generation will not consider vertical layers (and this value varies from chunk to chunk), the $h$ value will be discarded.

Each chunk becomes then represented as a bounded square in 2D space, with a $(x, y)$ **position** - representing the bottom-left corner of the square - and a width and depth values which, for a square, are the same, and will be named **size**. For a position of $(x_1, y_1)$, a chunk spans from $(x_1, y_1)$ to $(x_1 + size, y_1 + size)$. This is the chosen approach because a square area is simple to understand, visualise and store, making it manageable and flexible.

Chunks also possess a fourth characteristic: a $(x_{offset}, y_{offset})$ **offset**. Unlike the position value, the offset does not refer to the absolute position of the chunk in the world, but rather to its relative position to the initial chunk (i.e., the one positioned at the world origin).

Elements like networks and bounds areas may spawn across chunks. However, after each chunk's generation, they should by clipped to be confined to the chunk's bounds. This is not the same as saying that their nature will be modified. So, for example, let's consider a network containing edges $e1$, $e2$, and two chunks $c1$ and $c2$, with $e1$ inside $c1$ and $e2$ inside $c2$, then, the final result for the generation of $c1$ should only render $e1$, and $c2$ should only render $e2$[1].

## 3.2 Approaches

Considering the problems presented in section 1.2, the generation of every chunk must be somewhat predictive, considering the centre chunk's neighbouring area and the fact that whatever elements are present within that area which connect or transverse the target chunk must be coherent with the elements that would be generated if the target chunk changed. Taking that into consideration, two different approaches were taken regarding how the chunks' elements were generated.

It is important to highlight that these approaches are not direct competitors. That is, they were not developed concurrently, and were not expected to achieve the same (or comparable) results. They were, instead, created one after the other, with the second approach (approach B) benefiting from a more deeper understanding of the practical challenges, limitations and caveats that became clear while developing approach A.

The first approach - hereby mentioned as approach **A** - consists in a fairly intuitive method: for the generation of a given chunk for **offset** $(x_{offset}, y_{offset})$, the framework also generates all neighbouring chunks in a given integer *count* radius (this value relates to offset, not to absolute world position). So, for example, for a chunk with an offset of $(1,1)$, with $count = 2$, chunks from offset $(-1,-1)$ to $(3,3)$ will be generated, resulting in $(2count+1)^2$ chunks. This *count* value can be adjusted for different generation configurations. For features that span across great distances, with chunks dimensioned $\alpha$, $count \times \alpha$ must be greater or equal to the maximum possible spanning length of that feature. This is to ensure that the solution has full knowledge of all features that may be present in the chunk in question.

Perlin noise is a self-contained method. As such, terrain height values can be obtained without actually generating any extra chunk. The second approach - mentioned as **B** - takes advantage of this by avoiding generating unnecessary terrain surfaces, other than the one of the chunk in question. Approach B has different methods for generating some of the elements, as described in detail in each element's subsection.

## 3.3 Terrain

The terrain is represented as a vertex grid in 3D space, with a given position, number of columns and rows, as well as a specific cell size, which represents the spacing between vertices. Each

[1] This is a simplification. A more complete example would also consider shared edges, which should be split in subsections so that each subsection would only be contained in a single chunk.

vertex is constituted by a given $(x,y)$ world position and an associated height value (or elevation), which can be understood as the $z$ component of the vertex.

The value for each vertex's height is different for approaches A and B, as well as the implementation across the generation. However, lets consider that each approach has a generic method $GetHeightAt(x,y)$ to obtain the elevation of a vertex at $(x,y)$ coordinates. Then, terrain surfaces are created as follows:

```
1 var terrain = new Surface(columns, rows)
2 for x = 0 to columns // inclusive
3   for y = 0 to rows // inclusive
4       terrain[x, y] = GetHeightAt(x, y) // <- varies according to the approach
```

### 3.3.1 Approach A

For approach A, $GetHeightAt(x,y)$ is obtained as a single normalized Perlin value[2], considering a given set of input parameters for the noise generator [3], following equation 3.1. Since Perlin noise returns a normalized value (between 0 and 1), *scale* is used to determine the range of final values. So, for example, for a *scale* value of 10, terrain height values will vary between 0 and 10.

$$GetHeightAt(x,y) = scale \times Perlin(x,y) \tag{3.1}$$

One of the main disadvantages of using only one Perlin value is the non-existence of highly contrasting features (isolated mountain peaks or valleys). Additionally, more Perlin values could add additional layers of roughness to the terrain.

### 3.3.2 Approach B

For this approach, $GetHeightAt(x,y)$ is obtained by the equation 3.2.

$$GetHeightAt(x,y) = scale_{\alpha} \times Perlin_{\alpha}(x,y) \times scale_{\beta} \times Perlin_{\beta}(x,y)^2 \tag{3.2}$$

$Perlin_{\alpha}$ and $Perlin_{\beta}$ refer to Perlin noise with $\alpha$ and $\beta$ input parameter values, respectively (with distinct values for each parameter field). The *scale* value, as mentioned for approach A, is used to determine the range of final values for each Perlin method. This value, multiplied by $scale_{\alpha} \times Perlin_{\alpha} \times Perlin_{\beta}$, guarantees the existence of valleys and mountain peaks which would otherwise not be present, by decreasing or increasing severely the resulting value.

By using more than one Perlin value, this approach introduces some "noise" to the resulting terrain, which results in a more rough topology.

[2] Although a parameter called tiling is used to scale up or down the terrain, it is used internally by the Perlin method. As such, these values $(x,y)$ are always absolute world coordinates.

[3] The internal parameters used by Perlin noise function are: **frequency**, **octaves**, **scale**, **tiling**, **lacunarity** and **persistence**.

### 3.3.3 Tiling

One of the limitations of Perlin noise is the fact that the function repeats itself every multiple of a certain value $\alpha$ - this repetition behaviour can be designated as "wrap around". Although the wrap around value can be tweaked, for performance and memory reasons[4]it must be set to a relatively low value. That means, for this particular design, that chunks' dimensions may approximate $\alpha$.

Since Perlin wraps around $\alpha$, then $Perlin(x) = Perlin(x \bmod \alpha)$. This is problematic when chunks have similar or larger dimensions than this value, since every chunk's terrain is either a repetition of the neighbours or, even worse, the noise pattern repeats inside the chunk itself, and in so becoming obviously noticeable for each chunk.

In order to avoid the sensation of pattern repetition, the Perlin noise method exposes a parameter named **tiling**. Tiling works as a multiplier, changing the requested coordinates to a percentage. That is, if tiling is set to $\delta$ and a request for $Perlin(x,y)$ is made, this request is converted to $Perlin(\delta \times x, \delta \times y)$. This way, for a chunk of size $m$, wrapping occurs at the value of $\frac{m}{\delta}$. As a practical example, for $tiling = 0.01$, for chunks sized 256, wrapping will occur every 100 chunks. This is particularly important for the main objective of this work, which is to create the perception of infinity. As such, for practical purposes, a player traversing a world where features do not easily repeat would effectively be led to believe to be either traversing an infinite world or a very large planet (eventually coming around to the starting point).

By simply altering the tiling value, however, a new problem is introduced: by scaling down the requested values, one is not creating new Perlin values, but rather stretching up the Perlin noise lookup map. As a consequence, the generated chunks appear much smoother, since they only represent a small portion of the Perlin noise map. This is generally unwanted behaviour, since smooth terrain does not accurately represent rough imperfections that naturally occur in terrain, such as rocks or small hills. A way to combat this effect is by merging more than one Perlin maps, provided that the extra Perlin maps are given relatively small tiling values, so as to provide finer granularity, which results in more roughness, or "noisy" aspect.

Approach B implements this solution by multiplying 2 Perlin values, each one generated with different *tiling* values. In order to decrease the chance of recognizable terrain patterns, the two tiling values were chosen so that they share a high common divisor, thus decreasing the probability for simultaneously wrap, producing different combinations of Perlin values for a longer distance.

### 3.3.4 Consistency Regards

Perlin noise provides a wrapping set of values. Because of this, neighbouring values will always be consistent between themselves, not rendering visible "seams" when they wrap. This behaviour means that Perlin noise is tileable. Thus, the same Perlin noise map can be concatenated both seamlessly and endlessly, as show in figure 3.1.

[4]This value determines the hash size for the Perlin noise method. The program's memory usage and access time depend on this size. So, a larger hash size means more memory consumption and slower access time.

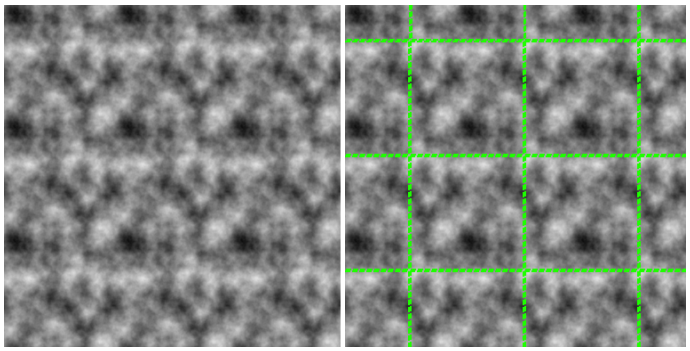

Figure 3.1: Left shows the seamless concatenation of the same Perlin noise map. Right shows the noise pattern.

This ability to tile means that, regarding this work, terrain can be generated endlessly without the concern of reaching world "edges" or visible limits.

## 3.4 Rivers

Rivers, being networks, are represented as edge-connected vertices, resembling a **graph structure**. Although other structures could be used, such a simple one directional, indivisible line or a tree, a graph structure is a more flexible and allows for both river divergence (separation of a river stream in one or more) and convergence (merging of tributary/affluent streams into a main stream).

A vertex in the river network is composed of a $(x, y, z)$ world position, and the set of edges that connect to it. Each edge contains two vertices that it links. For each river, there also some specially designated control vertices, described further.

The generation of individual rivers can be done uphill or downhill. These techniques are explained further and whether they are or not applicable for the designed framework.

### 3.4.1 Control Vertices

A river is built as a graph network, and contains special control vertices: **sources** and **mouths**. **Sources** are the vertices in which the river begins. Typically, these are located at high elevation points, such as mountain tops, from where the river is able to flow downhill, but can also be located, for example, at the overflow points of a lake. **Mouths**, on the other hand, are the vertices where the river ends its course. This can happen when the river encounters either a lake or ocean. In the event of a river delta being formed, a river may contain several river mouths. However, for the purpose of this work, a river can only contain **one** mouth (no deltas are formed).

### 3.4.2 Downhill Generation

If the algorithm starts its computation at the source, then it will trace a similar route to a real river, flowing with gravity, from high to low elevation values. However, this approach branches in two,

depending on whether the algorithm knows or not where it should flow to - in short, whether or not it is aware of the target mouth. In the first case, the algorithm flows freely in any valid direction with lower elevation, until it meets a local minimum, from where it can no longer flow, or the limit spanning length has been reached. There are, however, issues regarding this approach:

1. If the river halts at the first low elevation point it encounters, it may stop the generation too soon, producing a river that spans for too short a distance. This issue can be tackled using several strategies:
    - Carving the terrain, thus removing the local minimum and allowing the algorithm to proceed;
    - Allow the river to flow under the terrain;
    - By applying a recursive strategy, allowing the algorithm to continuously attempt to find the longest spanning river, in every viable direction, until a time or length target are met or all the spanning possibilities exhausted.
2. If the generation stops once the maximum spanning length as been reached, it could be that the river ends in a slope, instead of a lake or ocean, rendering an unrealistic result to the generation. The only way to prevent this from happening is to redirect the river to a near local minimum, when the river approximates the maximum spanning length, but it may be infeasible if no minimum can be found.
3. River branching almost always happens uphill[5]. Instead, rivers flowing downhill become themselves branches when they merge into other rivers. This implies that, upon river generation, the system knows about every other river in the surroundings, in order to merge them as a branch, when applicable. Although not an impossible problem to solve with the algorithm itself, it might severely impact performance, if a large number of rivers is generated.

In the second case, in which the river knows the target mouth, issue number 2 does not persist and the same techniques can be applied to overcome the remaining issues, not disregarding the fact that branch merging must consider that, when the mouth is known, the river might not actually flow there after merging with another.

By reducing the number of difficulties to tackle, the second approach becomes more appealing than the previous one.

### 3.4.3 Uphill Generation

If the generation starts in the river mouth, then the river flows uphill, searching for higher elevation points, contrary to the effects of gravity. This approach has a solid advantage regarding the previous techniques: even if the river sources are not known, the generation can halt at any time, since the stopping point will necessarily be uphill and at a higher elevation than the mouth - as such, a

[5]Downhill branching may occur in terrain with marginal slopes or at river deltas. Both cases will not be discussed in this work.

valid river source. Furthermore, river branching can occur by allowing the algorithm to explore, at given point, more than one valid (uphill) direction. Of course, as was previously mentioned, each river branch must still be aware of all the rivers surrounding it, in order to avoid intersections - unlike when going downhill, river intersections when going uphill may create infeasible downhill branches.

For the mentioned reasons and the advantages of this method, **uphill** generation is used by **both** approaches in this work.

### 3.4.4 River Spanning Length

Another important aspect in river generation, especially considering infinite deterministic worlds (and, in this particular case, the world division in chunks) is the **river spanning length**. Since generation depends upon the knowledge of any control point (be it source or mouth), any river which intersects or is contained by a chunk must be generated upon that chunks generation. For that to happen, the lookup method that finds these control points has to be able to find all the correct points. In order for such to happen, a maximum spanning length must be defined. The importance of this aspect is further discussed in subsection 3.4.9.

### 3.4.5 Knowledge

River generation depends upon the knowledge of the control vertices, and 3 main cases can be designated: **full knowledge**, **source-only**, **mouth-only**. In **Full Knowledge** both source and mouth points are known, and the algorithm can either run from source to mouth (downhill) or mouth to source (uphill). In **Source-only**, only the river's source is known, and the algorithm is forced to run downhill. Contrarily, **Mouth-only**, is when only the river's mouth is known and the algorithm is forced to run uphill.

As mentioned in subsection 3.4.3, both approaches will generate rivers going uphill. As such, **full knowledge** or **mouth-only** could be applied. However, the implementation of a maximum spanning length, as mentioned in subsection 3.4.5 offers the possibility to apply a mouth-only approach, since the river source will be determined by the algorithm's stopping point, which will in turn be dependant on the maximum spanning length (as will be mentioned in both approaches, the river generation can stop earlier if a local maximum is encountered. However, this does not change the fact that the river cannot span beyond the defined maximum spanning length).

### 3.4.6 Approach A

In this approach, the control vertices are found by sweeping the generated terrain's vertices, in order to find local **maximums** and **minimums**. **Maximums** represent mountain tops and suitable locations for **river sources**. Since this approach does not use them, these values are discarded. **Minimums**, on the other hand, represent lake or ocean bottoms, depending on whether they are above or below sea level, and will be used as **river mouths**. Although lake and ocean bottoms may not necessarily represent the exact places where a river ends its course, they represent the points

to which the river naturally flows - even if it meets the lake or the ocean border before reaching these points.

From the minimums, a set is obtained by filtering these points according to a given *threshold*, following the behaviour of equation 3.3. The filtered vertices represent the valid river mouths, and the starting points for the generation algorithm.

$$StartRiverAt(x,y) = Perlin(x,y) \leq threshold \tag{3.3}$$

As mentioned before (see section 3.4.3), rivers in this work are generated using an uphill-based generation. As such, the generation's starting vertex is necessarily a river mouth. The next vertices are chosen by the algorithm from a set of predefined vertices. These are obtained from a high frequency Perlin noise distribution, by filtering the local maximums and minimums. By using a high frequency Perlin noise generation, the resulting points are dense and reasonably well distributed (see figure 3.2), and can be used as anchor vertices for the river to connect to in its path.

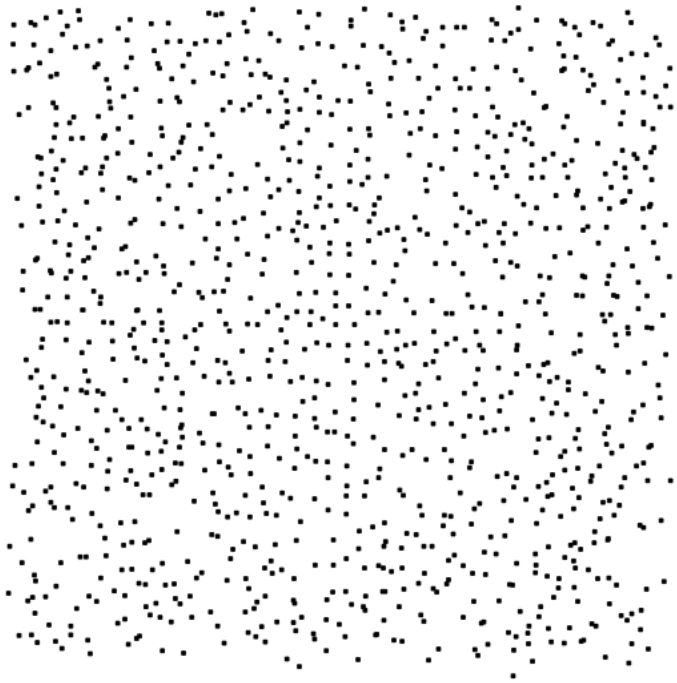

Figure 3.2: A Perlin-based point distribution obtained by generating a high-frequency Perlin noise texture and filtering the local maximums and minimums.

After the set of vertices is created, the algorithm then proceeds as shown in listing 3.1.

```
1 var current = riverMouth;
2 var last = current;
3 var length = 0;
4 while(current != null)
5 {
```

```
6   length += DistanceBetween(last, current);
7   if(length >= maxSpanningLength)
8     break;
9
10  last = current;
11
12  // All neighbours within a radius r of the current point with a larger elevation
        value are selected
13  var set = ChooseHigherPointsWithinRadius(current, r);
14
15  if(set.Empty())
16    break;
17
18  if(set.Count == 1)
19  {
20    current = set.First;
21    continue;
22  }
23
24  var closest = set.GetClosestTo(current);
25  if(closest.Count == 1)
26  {
27    current = closest.First;
28    continue;
29  }
30
31  // If there is 1+ closest points equidistant to the current point, then the one
        with higher Perlin noise value at its position is chosen.
32  current = closest.FindWithHigherPerlinValue();
33 }
```

Listing 3.1: River generation algorithm for approach A.

Figure 3.3 demonstrates an example of how the algorithm behaves. From **X** to **Y**, the algorithm evaluates all points within a valid range of the current point and choose the most appropriate (points evaluated but not chosen are shown in a green dotted line), choosing that point as the current point. The algorithm runs until all points under evaluation have either invalid elevation or the river has reached its maximum spanning length (these invalid cases are shown by the red dotted lines).

Given the algorithm as described above, this method possesses two characteristics:

- The algorithm does not need to know of river source(s);
- This approach might generate small rivers, since the anchor vertices are dependant on a random distribution of vertices and the current vertex might not find a valid vertex in the neighbour vertices. A recursive approach that would explore every valid path could diminish the impact of this problem.

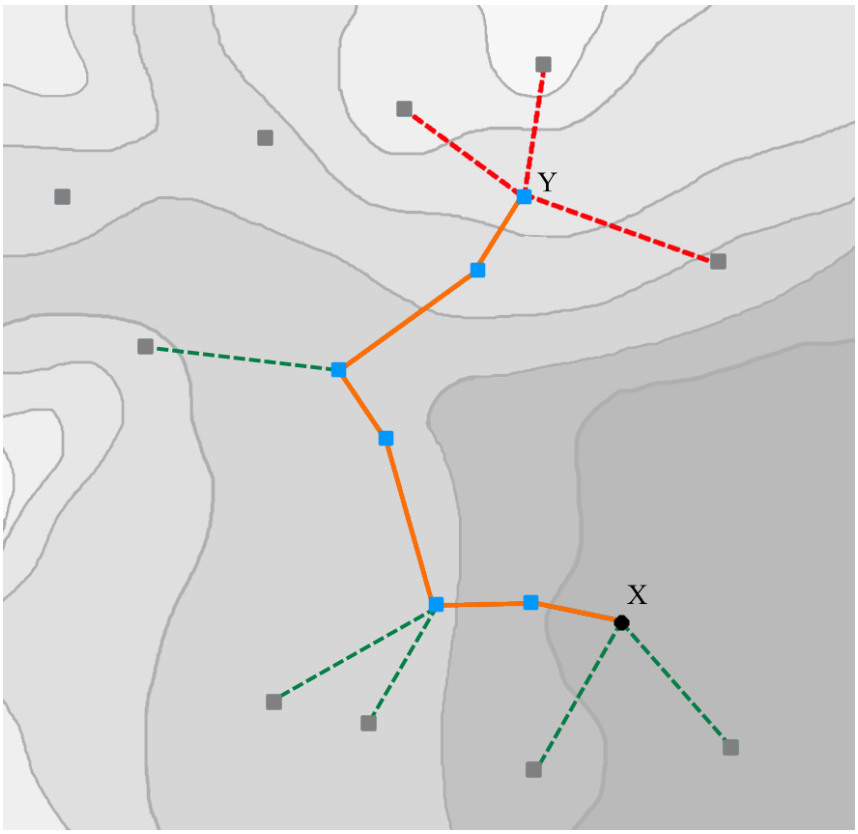


Figure 3.3: The algorithm starts at X. Points evaluated are linked with a green dotted line. Chosen points are shown in blue. The algorithm stops at Y because it has reached the maximum spanning length (links shown in red became invalid).

### 3.4.7 Approach B

Instead of gathering the river mouths from the set of minimums, the algorithm creates a regular grid of vertices, spaced evenly by $k$ units, and chooses the river mouths following the criteria: a vertex can be a river mouth if it is an ocean vertex or a local minimum of terrain. Then, very similarly to the previous approach (see equation 3.3), it filters the points based on a *threshold* value.

After the river mouths have been chosen, the algorithm goes from the starting vertex, choosing the next vertex as the first valid neighbour. To ensure that the neighbours are visited in a pseudo-random fashion, an array of directions vectors is used and shuffled each iteration of the algorithm, as shown below.

```
1  // Lets consider an array Directions with a set of directional arrays, representing
       the 8 geographic directions.
2  var current = riverMouth;
3  var last = current;
4  var length = 0;
5  while(current != null)
6  {
7    length += DistanceBetween(last, current);
8    if(length >= maxSpanningLength)
9      break;
10
11   last = current;
12
13   var seed = (int)(Perlin(current) * 1000);
14   var shuffledDirections = Directions.Shuffle(seed);
15
16   foreach direction in shuffledDirections
17   {
18     var neighbour = current + direction;
19     if(neighbour.z > current.z)
20     {
21       current = neighbour;
22       break;
23     }
24   }
25 }
```

Figure 3.4 shows an example of the algorithm. Starting at **X**, the algorithm shuffles the directions array. *South*, *West* and *Northwest* directions are evaluated in this order. Since Northwest is the first valid direction, it is chosen. The algorithm repeats the described process until **Y**, where no more valid directions are found. The light blue dotted lines represent the river's main course.

As with approach A, this method possesses two inherent characteristics:

- The algorithm does need to know about river source(s);

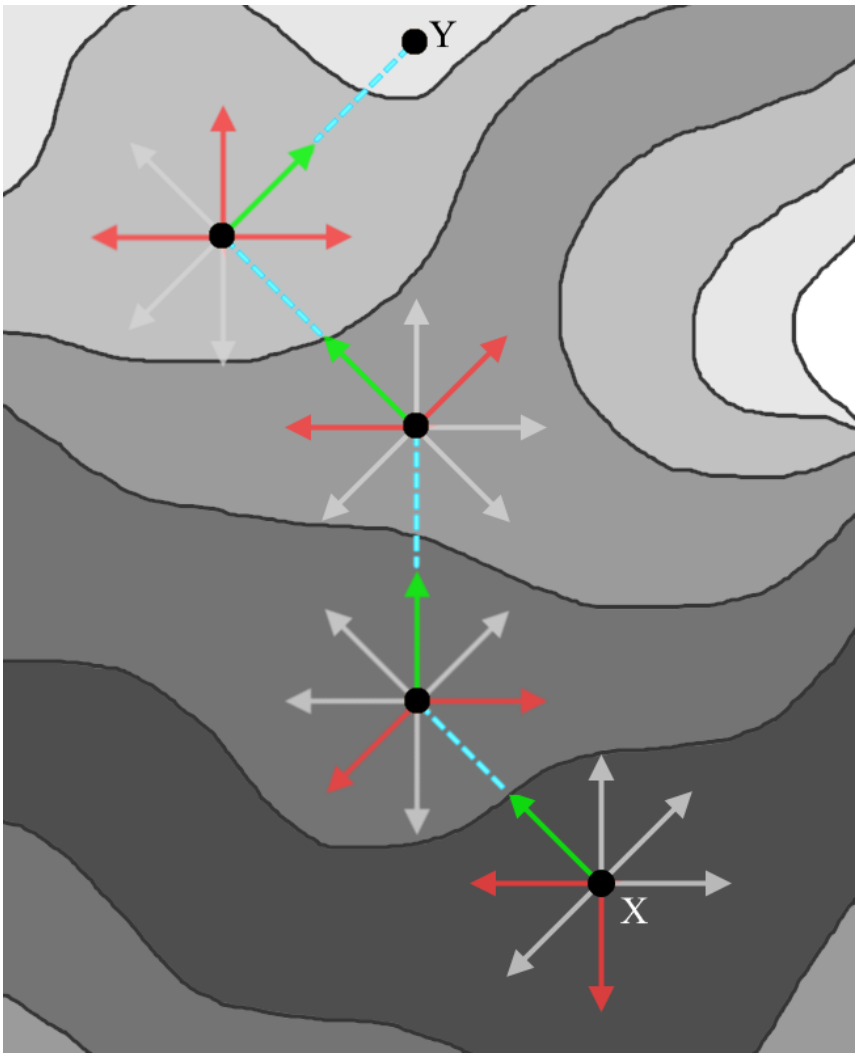


Figure 3.4: The algorithm goes from X to Y, evaluating in a random order all directions until the first valid one appears.

- Given its "greedy" nature, this algorithm may generate small rivers, since the current vertex might not find a valid vertex in any direction by poor early decisions. A recursive approach could improved the results of this method.

### 3.4.8 River Bending

In order to create a smoother curve, river paths are applied a simple interpolation to each edge (see figure 3.5), based on a value $\alpha$, which can vary between 0 and 1. The interpolations uses the line segment between B and the midpoint of *AC* as reference, and $\alpha$ signifies how close to the midpoint of *AC* the the interpolated point should be. A value of 0 obtains the point B, while a value of 1 obtains the point B4.

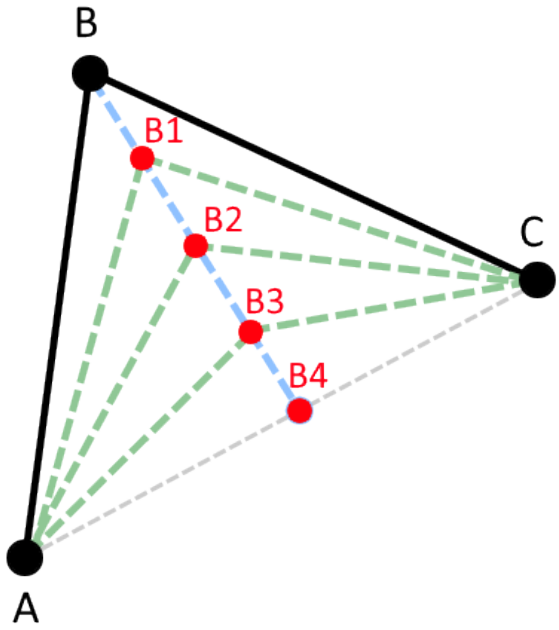


Figure 3.5: Interpolating B, using the midpoint of AC as reference point for the line segment in which the interpolation occurs. B1, B2, B3 and B4 are interpolations with $\alpha$ values of 0.25, 0.5, 0.75 and 1, respectively.

### 3.4.9 Consistency Regards

The greatest concern for river consistency is the ability to cross chunks: even a chunk that does not have the river's starting point within its bounds might need to know about the river's existence.

Let's consider the example in figure 3.6. Whether is A, B or C that is being generated, the algorithm must consider the river (marked in a blue line). For the remaining chunks, however, it is

irrelevant whether they know of the river's existence or not, since their generation does not depend on it[6].

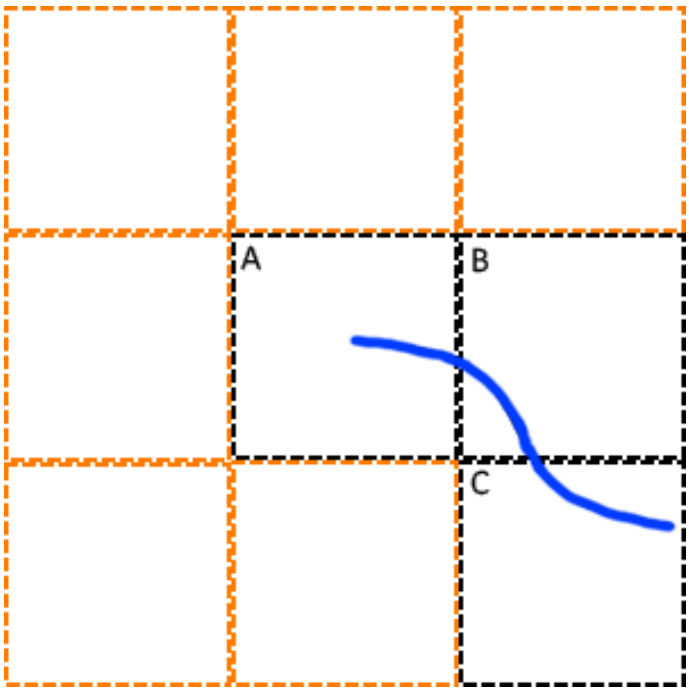


Figure 3.6: The solid curved line represents a pseudo-river, whilst the dotted lines represent chunk's borders. The chunks marked with an orange border do not need to know about the river's existence.

In order to obtain a perfectly deterministic generation, for every chunk, the generation algorithm must know about all rivers that cross or are contained in that chunk. However, in order to know the rivers that satisfy these conditions, all possible rivers would have to be generated, which is obviously impossible. As such, a more realistic goal is for each chunk to determine and generate all rivers that can **potentially** be contained or intersect it, by limiting the characteristics of the rivers in the generation algorithm.

As such, lets consider that 2 characteristics are known: the **river's mouth**[7] and the river's **maximum spanning length**. Since the river cannot span for more than the given spanning length, then it must be contained within a circumference with radius equal to the maximum spanning length and centred in the starting point - in this case, the river mouth. As shown in figure 3.7, the edge cases are those in which the river might end precisely in one of the chunk's border points. Covering these cases guarantees that any chunk is aware of all the rivers that might cross or intersect it. The solution is also presented in the figure, as a blue square, representing the area within which a chunk must know all river starting points (although the area could be represented as a rounded rectangle, a simple rectangle is easier to represent and still guarantees full coverage,

[6]Note that this might not be true if, for example, a more realistic world was being considered, in which the river's existence might, for example, constrain the course of a road or highway. For this project, that is not the case.

[7]Depending whether the algorithm generates the river downhill or uphill, this could be either the river's source or mouth but, as mentioned before, this work will generate rivers uphill.

while covering only a small percentage of unnecessary area). For chunks dimensioned $k$ and a river maximum spanning length of $w$, the square's side must measure at least $k+2w$.

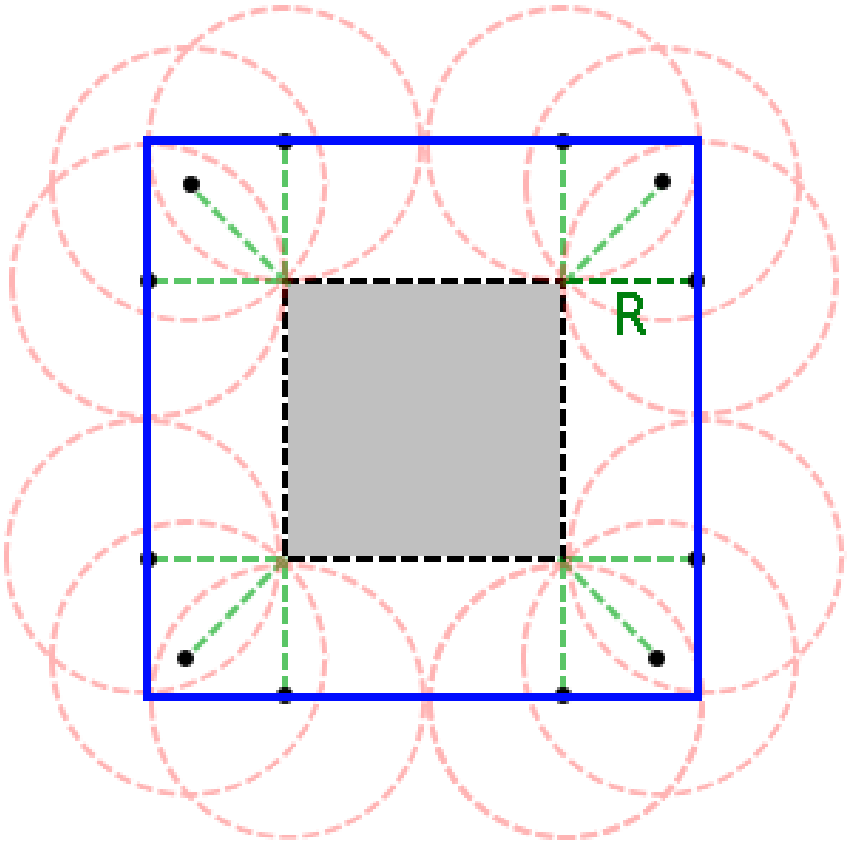


Figure 3.7: The blue border represents the minimum area that must be considered in order to obtain the river's starting point. The green lines represent the river's maximum spanning length.

## 3.5 Cities

The process for city generation is devised in three steps: creation of the **road network**, creation of **urban lots** and lot subdivision by **land usage**.

### 3.5.1 Road Network

A road network is composed by edges and vertices, corresponding to roads and road connections, resembling an undirected graph structure. The creation of a road network as a basic city layout has two main advantages: a clear definition of the city's boundaries - which provides connection points with highways - and can be a facilitator to the creation of the urban lots, given that these must be connected by roads. Two distinct methods were approached in this work to generate the road network: **Perlin-based network** and **radial concentric network**. This section will also explain how the road network generation algorithm works.

#### 3.5.1.1 Perlin-based Network

This approach begins by creation a road network to fill the chunk's area, using an algorithm created specifically for this purpose (see section 3.5.1.3). Afterwards, each point in the network is evaluated for removal, based on a threshold $\alpha$ and whether or not the point is an ocean point. The threshold evaluation compares the threshold value $\alpha$ to the Perlin noise value at the point's 3D world position. If the value is smaller than $\alpha$, then the point does not belong to a city network and is removed. Otherwise, the point belongs to the network and is kept. The same principle applies to the removal based on the point being an ocean point. Given the elevation of the point, if this value is beneath sea level, the point is considered to be an ocean point and removed. The resulting fragments of the original network will compose the city networks in that chunk.

The algorithm used for the generation of the network guarantees that city networks clipped at the chunk's edges are guaranteed to be linked to the neighbouring chunks' networks (see section 3.5.1.3). One of the limitations of this method is its abstract approach, which disallows to know beforehand what actual area of the world will be covered by cities. Actually, only after the city network fragments are generated can these be evaluated to obtain a list of the actual cities.

#### 3.5.1.2 Radial Concentric Network

This method consists in generating a radial network around a single concentric point, designated city centre. Each network is generated using the same algorithm as in the previous method, clipped to belong within a given radius $\theta$ of the corresponding city centre. As in the previous method, all ocean points are removed from the city grid. To avoid overly symmetric circle networks, points are removed if the Perlin value at their position is smaller than a given threshold $\alpha$. This threshold may diminish as the distance to the city centre increases, so increasing the probability for point removal, simulating an effect similar to the reduction of density in the outskirts of cities, as opposed to more populated populated central areas.

Another particular aspect of this method is the ability to avoid unnecessary calculations: no city has to be generated if the radial area around its centre does not intersect neither is contained by the chunk's boundaries. This, unlike the previous method, prevents the generation of a larger grid which might end up being almost fully clipped afterwards.

#### 3.5.1.3 Generation Algorithm

Under an initial approach, Voronoi graphs were considered for the creation of road networks, through the use of Fortune's implementation method [For87]. Although Voronoi graphs are, by definition, sets of closed polygonal lots, Fortune's method is non-trivial, possesses a time complexity of $O(n \log n)$ [For87] and, because of the implementation based on a "sweep line", it is not suitable for optimization through parallelisation. Because of this, it was hypothesised that it could become a performance bottleneck. Accordingly, a faster alternative was then devised. This algorithm works in 3 phases, as shown in figure 3.8. With a reasonable value for the offset amount,

the final result of this method resembles a Voronoi graph, whilst being $O(n)$ in time complexity and suitable for concurrent processing in each phase.

1. **Grid Creation**: a evenly spaced vertex grid is created;
2. **Vertex Offsetting**: all vertices are applied random offsets, based of a predefined set of directional vectors. The chosen vector depends on the Perlin value on the vertex's position. This guarantees not only a sense of randomness, but also that this method will offset the same vertex always in the same direction (this is particularly relevant to guarantee determinism);
3. **Vertex Division**: the third and last phase of the generation method consists in dividing a percentage of the vertices in two, by offsetting the vertex twice in opposite directions, effectively creating a new edge. The offset directions are chosen from a predefined set of directional vectors, according the vertex's position Perlin value.

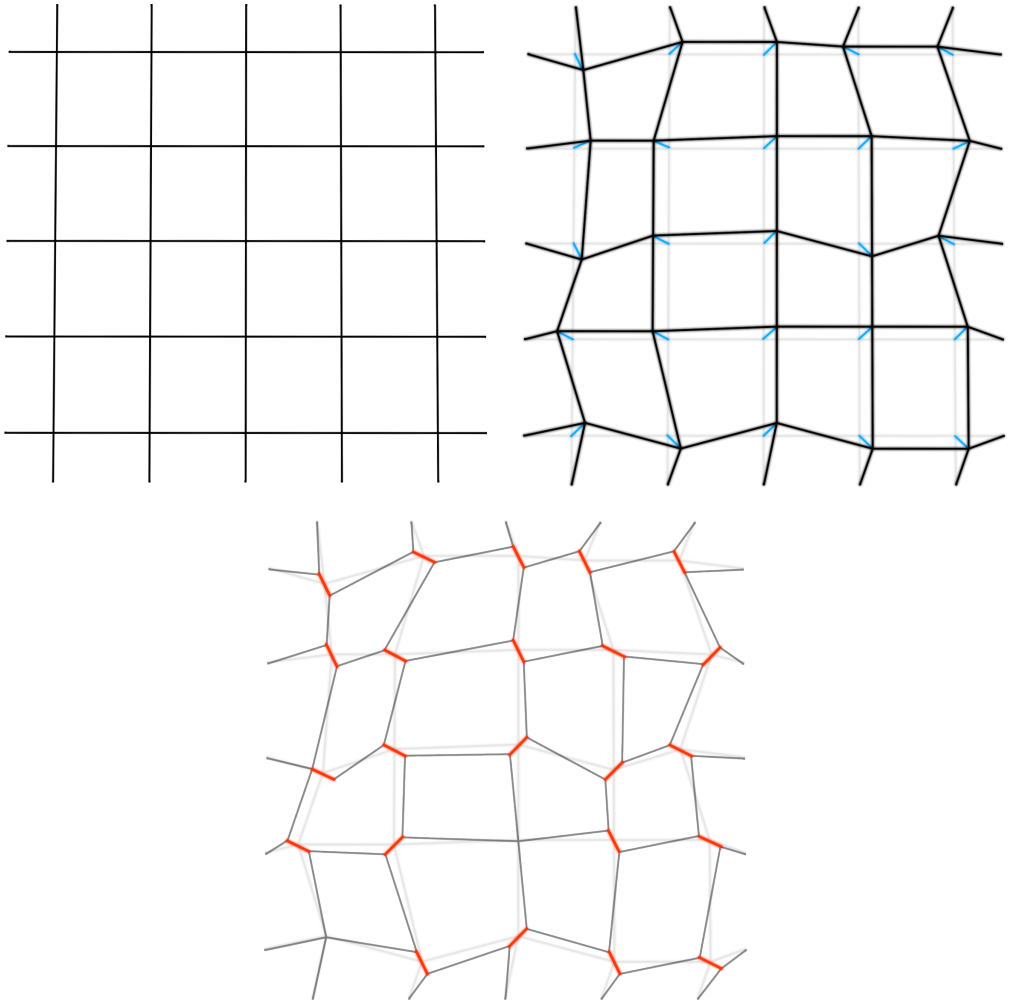

Figure 3.8: Top left shown the regular grid. Top right shows the offsetting of the grid's vertices. Bottom image shows the division of the vertices, creating new edges.

### 3.5.2 Urban Lots

Urban lots are directly created from the road network's closed areas. In figure 3.9 an example of this subdivision is shown. Notice that the urban lots - represented in grey - are only created inside areas enclosed by the road network. This subdivision in lots resembles the subdivision created in real cities, as mentioned in [Gro09, KM06, KM07, LHD15, LWWF03], under the aliases "lots", "blocks", "cells" and - to some extent - "districts". These lots represent the areas in which buildings are placed.

Since the process of building generation is a well discussed subject, with many solutions already presented, any of which could be used for this work, it was not approached. Nevertheless, lots are given a "purpose", or usage, as is mentioned in the section 3.5.3.

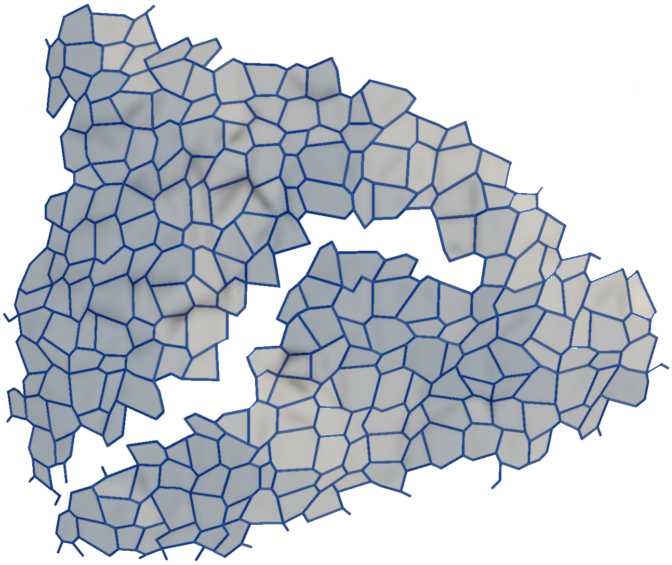

Figure 3.9: The blue lines represent the road network. The urban lots are represented by the grey polygons.

### 3.5.3 Land Usage

Each urban lot is attributed a land usage, which can be **commercial**, **residential** or **industrial**. These values were chosen due to the fact that they represent the main roles - or logical partitions - which can be found in actual cities (see [Gro09, KM06, KM07, LHD15, LWWF03]). **Commercial** lots represent the areas which would be occupied by commercial of office buildings. As mentioned in Lechner's work [LWWF03], these tend to be concentrated near major networks and city centres, as they are areas with a higher flow of people. **Residential** areas, on the other hand, concentrate themselves in self contained blocks, away from major networks, connected by secondary networks, and represent the areas reserved for housing. **Industrial** areas are constituted by factories and large industrial complexes, and are usually located in less densely populated areas,

in a city's outskirts. These locations are more fit to contain large buildings and provide easier access to railways and highways from other cities, for example. Some examples of land usage distributions are shown in figure 3.10.

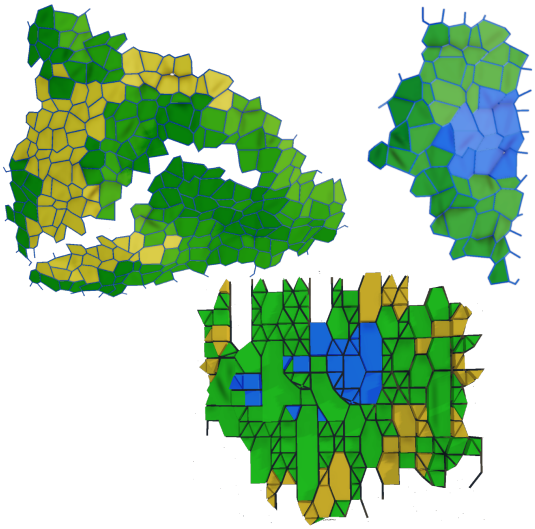

Figure 3.10: City land usage examples

#### 3.5.3.1 City Layout

As mentioned by Groenewegen, regarding urban structure: "The identification and explanation of patterns in the urban structure form a major topic in the field of urban geography. [...] Simple generic models divide a city into rings or wedges, while more advanced models take into account different local and historic developments."([Gro09]). For a matter of simplicity, this work assumes a city layout loosely resembling some older western European feudal cities: besides the geometric centre, the city will possess an historic centre (in feudal settlements, this is were the great manors and the feudal castle would be located). The algorithm considers areas closer to the historic and geometric centres more densely populated, and as such more likely to be commercial areas. As the distance to both the centres diminishes, so does the probability for commercial areas to appear, increasing the probability for residential areas to be attributed. When the distance is large enough, and the algorithm finds itself close to the city's outskirts, the probability for industrial areas increases dramatically.

#### 3.5.3.2 Historic Centre

The historic centre is calculated by offsetting the city's geometric centre by a certain amount $\gamma$ in a random direction $\theta$. $\gamma$ is a parameter fed to the generation algorithm. As for $\theta$, it is obtained by choosing a random director from a set of eight possible director vectors, representing the eight geographic directions. The director is chosen as shown by listing 3.2.

```
// Contains the 8 geometric directors
array Directors = {...};
// Geographic directions count (N, NE, E, SE, S, SW, W, NW)
int numDirections = 8;

// geoCentre represents the city's geometric centre
function GetRandomDirector(geoCentre)
{
  var noise = Perlin(geoCentre);

  // since noise is a value in range [0...1],
  // index will be in the range [0...7].
  var index = (int)Round(noise * (numDirections - 1));

  // return the pseudo-random director
  return Directors[index];
}
```

Listing 3.2: Obtention of the pseudo random director

#### 3.5.3.3 Adjustable Parameters

For the calculation of the land usage the following lot-dependant parameters are used: *historicWeight*, *centreWeight* and *perturbance*. *Perturbance* is the Perlin noise value (ranged [0...1]) for the centroid position of the lot. *HistoricWeight* is a normalized value inversely proportional to the lot's distance to the historic centre. *CentreWeight* is also a normalized value, inversely proportional to the lot's distance to the geometric centre. Having these values, the calculation is based on table 3.1. The values presented were obtained through experimentation until visually pleasant results were obtained - whilst maintaining the formerly explained reasoning valid. The value of perturbation is used as a probability to decide between the commercial, residential and industrial options, based on the tabled value.

Table 3.1: City Land Usages Calculation Table

| **HistoricWeight** | **CentreWeight** | **Probability** | | |
|---|---|---|---|---|
| | | Commercial | Residential | Industrial |
| $> 0.75$ | $> 0.50$ | 0.45 | 0.55 | - |
| | $\leq 0.50$ | 0.50 | 0.50 | - |
| $> 0.50 \bigcap \leq 0.75$ | $> 0.50$ | 0.45 | 0.55 | - |
| | $\leq 0.50$ | 0.40 | 0.60 | - |
| $> 0.25 \bigcap \leq 0.50$ | $> 0.75$ | 0.75 | 0.25 | - |
| | $> 0.25 \bigcap \leq 0.75$ | 0.20 | 0.75 | - |
| | $\leq 0.25$ | - | 0.75 | 0.25 |
| $\leq 0.25$ | $> 0.75$ | 0.70 | 0.30 | - |
| | $> 0.50 \bigcap \leq 0.75$ | - | 0.66 | 0.33 |
| | $\leq 0.50$ | - | 0.50 | 0.50 |

Since all parameter values can be converted to probabilities, the tree in figure 3.11 shows the probabilities for each decision.

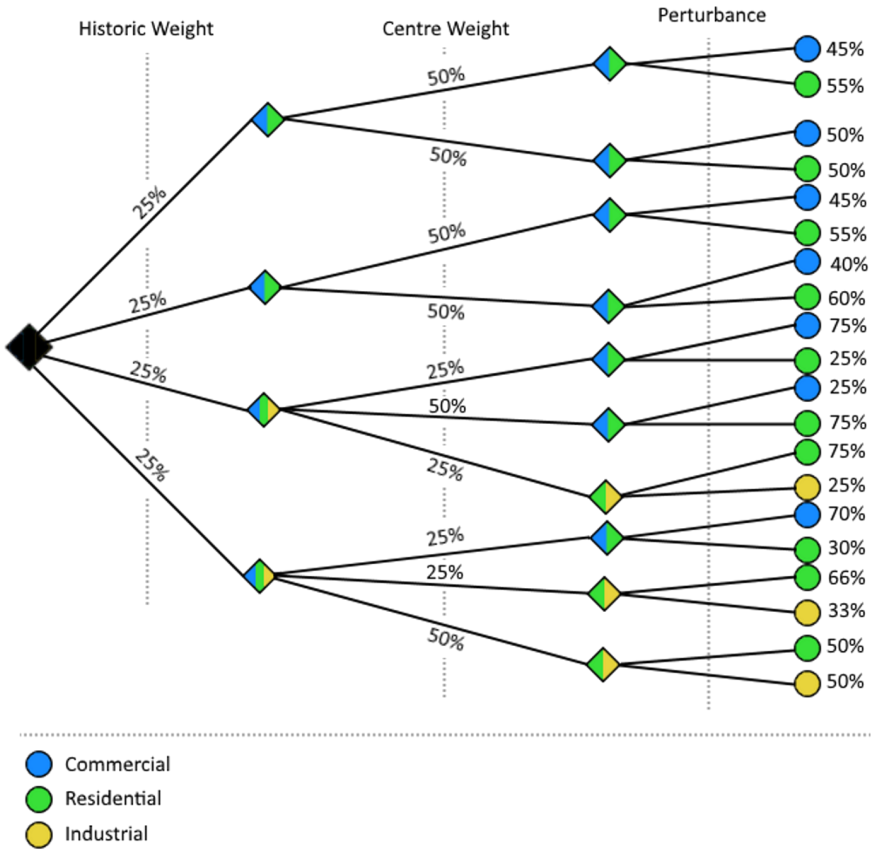


Figure 3.11: Probabilities distribution for land usages

### 3.5.4 Approach A

This approach uses the **Perlin-based network** method for road network generation. An important detail about this approach is the impossibility to know where city geometric centres are before all the cities have been generated - this limitation is a direct consequence of the city generation method, as mentioned in subsection 3.5.1.

One of the characteristics of this method is the ability to generate diversely shaped cities, due to the nature of Perlin noise maps, which generate non-regular distributions. This enables the generation of small city networks, which might work as satellite locations for bigger cities. However, this characteristic limits the calculations for land usage, as the normalization for parameters *historicWeight* and *centreWeight* depends on the city radius. Cities with irregular shapes may have poorly calculated radii, with many lots in the outskirts considered to be very close to the city centre, and most likely to be attributed **residential** or **commercial** land usages. Although not a crippling limitation (not all cities must be entirely surrounded by industrial areas) it ends up effectively deceiving the calculation method.

### 3.5.5 Approach B

This approach uses the **radial-concentric network** method for road network generation. Since this approach knows the city centre and city radius, the calculations for parameters *historicWeight* and *centreWeight* are much simpler and direct than with the former method. This ease of calculation makes land usage attribution much more controllable. However, cities can be come much less dynamic, as their shape is based on a regular circumference. This can be diminished by decreasing the density with distance (as mentioned in the algorithm description in subsection 3.5.1), but not entirely removed.

### 3.5.6 Consistency Regards

A major concern regarding consistency is the the clipping of cities that cross city edges. This issue is two-fold: both road networks and lots may be clipped. So, the main question is: how to guarantee that roads and lots that are generated by a given chunk are guaranteed to be seamlessly linked to the next chunk and the remaining city generation?

Taking the example of figure 3.12: black circles represent the spots in which roads are clipped, while yellow diamonds represent the centroids for lots that are present in more than one chunk. The solution that was devised consisted in the following: for **road networks**, since they are simple graphs, a division of the road's edge in the clipping point is enough.

For **city lots**, however, the problem becomes more complex. City lots are represented as mesh polygons. In order to allow the clipping, the solution would be to apply boolean operations to these polygons. Given the time limitations for the implementation of this work, a shortcut was taken: instead of clipping meshes, the algorithm considers the centroids of each lot (in figure 3.12, these centres are represented by the yellow diamonds) and erases the mesh if the centroid does not belong within the chunk under generation.

This solution guarantees that the meshes are kept in one of the chunks only, since the meshes' centroids are unique. However, a problem arises: since some meshes are erased, how to guarantee that the erased meshes are always on chunks other than the one being generated? The simple response is that there is no such guarantee. Nevertheless, this method was used for three main reasons, the should guarantee its reasonableness:

1. After observation of the generated lots, it was concluded that most of the times these form simple convex polygons. Hence, the centroids actually divide the lot in half. By maintaining the polygon in the chunk which contains the centroid, it guarantees that the biggest possible area of the lot is maintained inside the generated chunk.

2. Since only city blocks in the blocks edges are clipped - and considering that the chunk's dimensions exceed by far the dimension's of a single lot - it is reasonable to assume that this clipping is not easily noticed.

3. Considering that chunks' generation in games is intended to be continuous - that is, if a player is traversing a certain chunk and is near a chunk's border -, the immediate neighbour

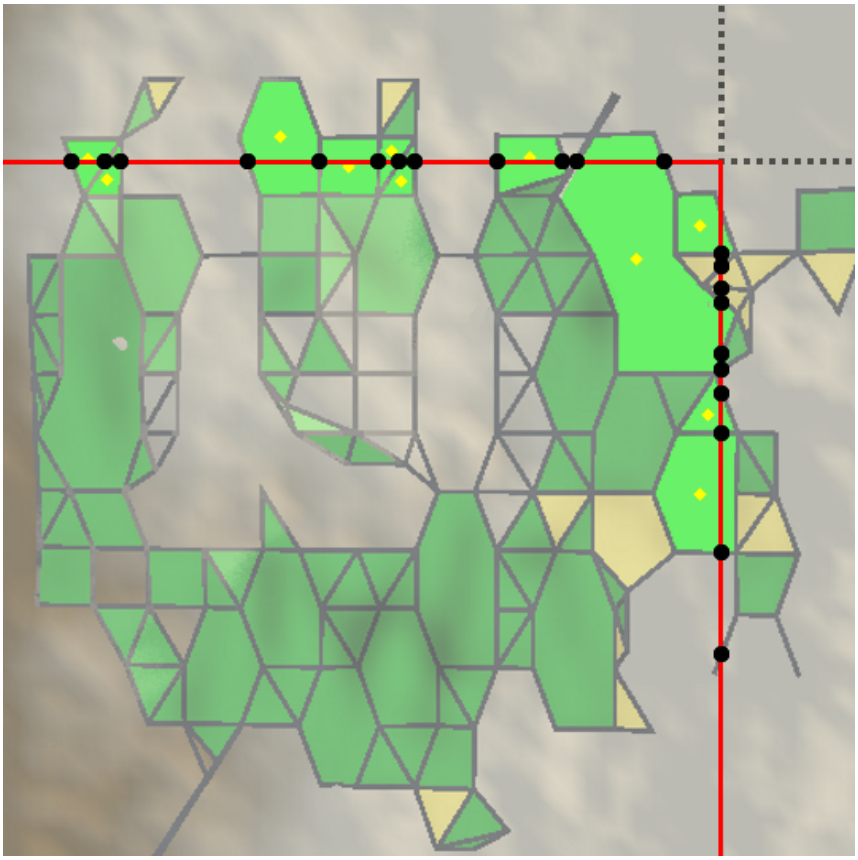

Figure 3.12: The red line represents the chunk's border. The black dots represent the road network clipping points and the yellow diamonds represent the lots' centroids.

of the central chunk is supposed to be generated also. In so being, all lots erased in the central chunk will be generated by the neighbouring chunk. So, pragmatically speaking, this issue should not be observable.

## 3.6 Highways

Highways are major networks that connect cities, thus highway generation is highly dependant on the method used to generate the smaller city networks, as well as their general locations in the world. In this work, each highway connects to the the limits of the cities' networks[8].

For the purpose of this work, highways are represented as simple edges between cities (for embellishment purposes, approach B allows highways to deviate and bend in small amounts). A more realistic highway system would take into account the lay of the land to obtain the optimal highway network. Two methods for highway generation are approached: **city-based network** and **master-based network**.

### 3.6.1 Approach A

This approach uses a **city-based network**: highways are defined **after** the generation of the city networks. As a result, it is not known *a priori* where to place highways. The procedure for highway generation becomes as follows: for each city network, a group of neighbouring city networks is selected. The selected group contains the cities to which the first will connect, and each of these cities is chosen as being the closest to the target city without any in between, within a given tolerance. That is, let the target city be $a$ and the current city under comparison $b$, with a tolerance value of $k$ - there cannot be a given city $c$ such that:

$$distance(a,b) > k \times distance(a,c) \tag{3.4}$$

After the connections are selected, for each pair of city network the two closest points are discovered - these points will necessarily be located at each network's limits - and connected by a simple edge, roughly representing the guiding segment of the highway.

Since this method does not prevent *a priori* intersections, these are removed after all highways are generated. In order to guarantee that the removal is deterministic, for each pair of intersection highways $h1$ and $h2$ and their respective midpoints $m1$ and $m2$, the rule in equation 3.5 is applied.

$$HighwayToRemove(h1,h2,m1,m2) = \begin{cases} h1, & \text{if } Perlin(m1) < Perlin(m2) \\ h2, & \text{if } Perlin(m1) \geq Perlin(m2) \end{cases} \tag{3.5}$$

This method will be used along with *Perlin-based network* generation method, since Perlin-based network is the only described method for city generation which does not need to know where cities are located **a priori** (see subsection 3.5).

[8]This is not always true in real cities but, for a matter of simplicity, is assumed to be true.

### 3.6.2 Approach B

This approach considers a **master-based network**. Unlike the previous method, this one generates the highways first, as a wide master network (using the same algorithm as described for road networks (see section 3.5.1). This master network extends well beyond the chunk's dimensions, providing a very wide lookup - the area covered by this lookup is parametrised. Afterwards, the method requires city networks to be generated also, in order to discover the points in the smaller networks to which the highway network really connects - this is necessary because every highway is initially connected to the city's centre (each city's centre is represented by the vertices in the graph of the master network). However, it usually happens that such point does not belong to the city network's limits and, therefore, is not a valid connection point. This method will be used along with the *radial concentric network* method for city generation (see subsection 3.5), given that the latter requires the definition of a city centre, which can be provided. Furthermore, if cities were generated first, this method would not guarantee valid connections. The results of generating this master network can be observed in figure 3.13

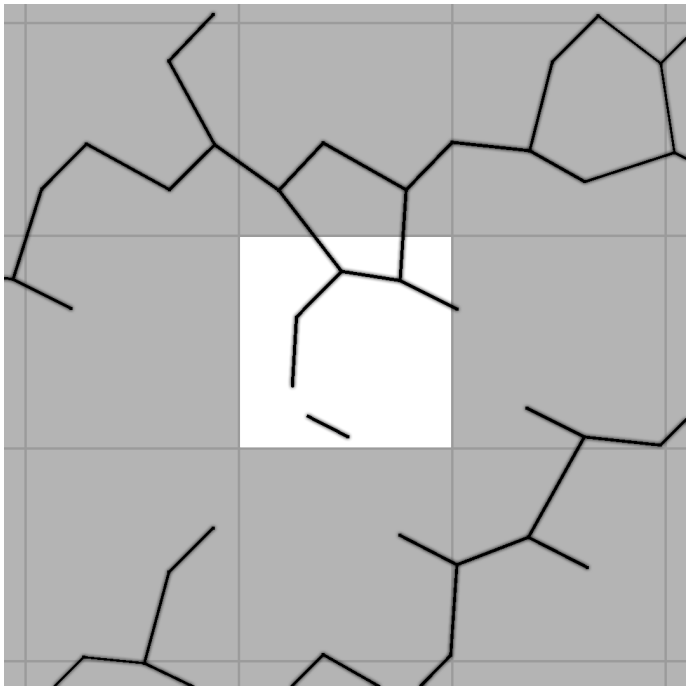

Figure 3.13: The white section represents the chunk's area, whilst the bold black lines represent the master highway network.

### 3.6.3 Consistency Regards

Consistency in highway generation is two-fold: clipping and city connections. Clipping occurs when the highway crosses two chunks, and is easily solved by subdividing the crossing edge at the clipping point. City connections represent, however, a more complex challenge - and may

even constrain clipping. These connections problems are different for both approaches and will be discussed separately.

#### 3.6.3.1 Approach A

The problem that arises with a city-based network is that it is necessary to limit the connections' maximum length. Otherwise one might end up with a situation similar to the one depicted in figure 3.14.

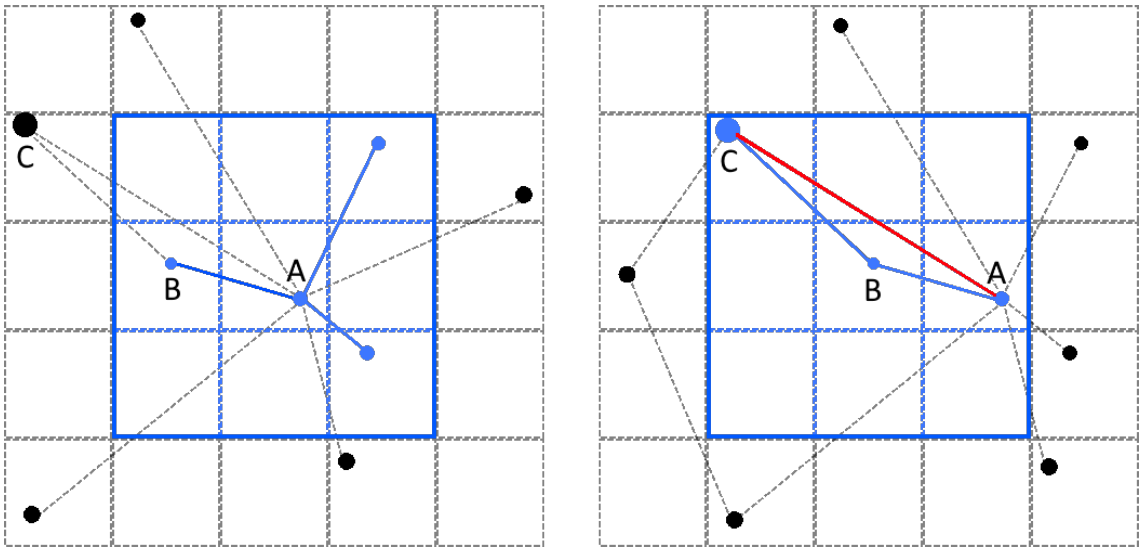


Figure 3.14: Notice how, when the generation shifts, a connection between cities A and C is suddenly considered. However, since it was not being considered before, the highway that appears in the second case would not be generated in the first one.

By not limiting the connection's length, the connection between cities A and C is considered when the generation is centred in the chunk that contains B. However, since the generated chunks for lookup is limited to a radius of 1 and the distance between A and C far outreaches the size of 1.5 chunks (central chunk's radius + outer lookup chunk dimension), one might end up considering the highway in one generation and discarding it - or never being aware to begin with - in the other generation.

In order to maintain consistency, a constraint must be introduced: considering that chunks are dimensioned $\alpha$ and the radius for lookup is $n$, then the maximum connection distance in approach A cannot be superior to $\alpha(n+0.5)$.

#### 3.6.3.2 Approach B

The consistency issues that arise with this approach are not related to highway connection between cities as described for the former approach, since these connections are known *a priori*, and given the wide lookup network, these are not inconsistent across chunks. However, since the connections are altered by cities' layouts, a new issue appears, as represented in figure 3.15

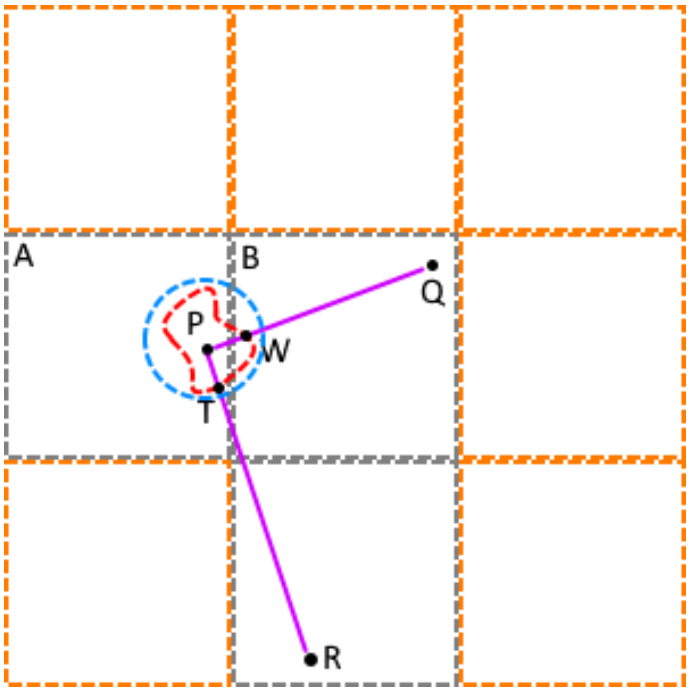


Figure 3.15: The purple lines represent highways, whilst the points P, Q and R represent city centres. The points T and W represent the points were the highways actually start, when considering the outline for city P, represented by the red dotted line. The blue dotted line represents the city's maximum radius.

Let's consider the points P, Q and R as being city's centres. It is intuitive to think that chunk A is forced to know about the highway that connects to Q. However, considering the city's outline as the red dotted line, and point W the point were the highway effectively starts, the situation changes, as chunk A does not necessarily need to know about the highway's existence, since it is now fully contained in chunk B. The solution employed to solve this issue is the redefinition of the highways connection points at the maximum radial border of each city (note that this radius is configurable and known to the algorithm), as shown by the blue border in figure 3.15. After the generation of the city, the highway can then be correctly connected to the nearest city road.

## 3.7 Forests

Forests are bounded areas, with atomic elements distributed within, such as trees and low vegetation. In order to generate these bounded areas in a deterministic fashion, one can rely on a similar approach to the Perlin-based threshold approach as discussed in section 3.5. However, instead of generating a network and clipping it to belong in the resulting area, this area is itself the forest area.

In order to determine the position of elements such as trees, the approach of **Perlin-distributed points** - as described in section 3.4 - can be used. By obtaining an even distribution of points in a

space, these can represent the location of the elements. This approach can benefit from the use of overlaying point distributions, in order to increase density in pseudo-random areas.

## 3.8 Oceans

Oceans are large bounded areas, which span for very wide areas, splitting terrain into continents and islands. As mentioned before (see section 2.3.1), there are two mainly discussed methods for ocean generation: **flooding** and **fixed water elevation**.

### 3.8.1 Flooding

This method works by choosing a few local minimums of the world and flooding them until a certain elevation threshold is reached. Although this method covers the main limitation of the previous one - by allowing areas under the elevation threshold to be unaffected - the obstacles it presents, concerning the generation of infinite worlds, are far more limiting, since this method requires a very wide lookup on the world's topology in order to choose the starting points. The other great handicap is the inability to immediately know whether a given point in space is or not an ocean point, unless a flooding algorithm is ran every time or has already been run and its information stored somewhere, resulting in a poor approach, performance-wise.

### 3.8.2 Fixed Water Elevation

Given the drawbacks of the flooding method, the method chosen for this work is the **fixed water elevation** method.

This method is quite simple and straightforward: the world is provided a given sea-level - or elevation - and any terrain point below that elevation is considered to be submerged, and vice-versa. This information is very relevant to the generation of other elements, by providing the means to immediately know - without any previous lookup - whether a given 3D point in space is or not an ocean point. Although this approach' results actually approximates the world's ocean layout, it prevents the formation of some land masses whose elevation is lower than sea level - this phenomenon can be observed, for example, in the Netherlands, where most of the country's landmass sits below sea-level.

Given this method, the determination of whether or not a 3D position represents an ocean point follows equation 3.6 (or equation 3.7, when considering terrain lookup), where $\psi$ is the water elevation threshold.

$$IsOcean(position) = position.z \leq \psi \tag{3.6}$$

$$IsOcean(x,y) = Terrain.GetHeightAt(x,y) \leq \psi \tag{3.7}$$

As to the representation of an ocean by each chunk, it is considered a terrain surface with a single fixed elevation value. Given this property, it is also acceptable to simplify this surface into a 4 vertex layout: bottom-left, bottom-right, top-left and top-right, spaced by the chunk's world dimensions.

### 3.8.3 Distribution

Perlin noise does not guarantee normal distribution of points. That is, even if a given ocean elevation is set at average the height scale of the generated terrain, this does not guarantee that 50% of the generated world is below sea level. Thus, ocean distribution is guided by visual feedback, and the value is set through iterative tempering, until a visually pleasant result is obtained.

## 3.9 Lakes

Much like oceans, but in smaller dimensions, lakes are bounded water areas. Very similarly, these can also be generated by flooding algorithms (the water elevation technique is not applicable, since the threshold value would have to be higher than the ocean value, and such value would effectively create a new pseudo-ocean which would flood the ocean itself).

Lake generation through flooding requires the choosing of local minimums, as with oceans. From these starting points, the algorithm floods the terrain, incrementally increasing the flooding elevation threshold, until a **limit elevation** or a **limit flooding area** is reached.

### 3.9.1 Consistency Regards

The main constraint for this method is the possibility that a small lake is suddenly transformed into a large lake, due the limited lookup of the surroundings. For example, lets consider a minimum point $A$, chosen as the starting point of a lake. Lets also consider that point $A$ is standing in a small bounded area, near a mountain peak. Even for a small flooding threshold, the algorithm might soon overflow the small bounded area, flooding a large portion of the world, which would be infeasible, both in realism and due to computational limitations.

To prevent this, a wide lookup method must be used in order to provide the algorithm with the necessary information required to generate a believable small lake (there are large lakes in the world, such as those in North America and Northern Europe. However, the generation of these required a very wide lookup which is not discussed in this work). Providing this predictive lookup might become computationally expensive for a relatively large number of lakes.

## 3.10 Biomes

As seen before, biomes are either aquatic or terrestrial. In this particular world, biomes can be obtained either by analysis of temperature and precipitation maps or by analysis of world characteristics (such as elevation).

Terrestrial biomes are defined based on temperature and precipitation values, as according to the chart 2.1 proposed by Whittaker. These temperature and precipitation values are obtained by the following behaviour: two Perlin texture are generated, representing temperature and precipitation base values. Each temperature value is then applied a latitude multiplier, which depends on the position's $z$ coordinate and follows equations 3.8 and 3.9.

$$latitudeMultiplier(y) = \begin{cases} \frac{y \mod L}{0.5L}, & \text{if } y \mod L \leq \frac{L}{2} \\ 1 - \frac{y \mod L}{0.5L}, & \text{if } y \mod L > \frac{L}{2} \end{cases} \tag{3.8}$$

$$f(temperature, y) = temperature \times latitudeMultiplier(y) \tag{3.9}$$

In these equations, $L$ stands for the maximum value of $z$. Since this work intends to infinitely generate worlds, the latitude functions is intended to tile, as shown in figure3.16. This is intended to emulate the behaviour of temperature fluctuation from the world's poles to the equator - in this case, the equator would be at $0.5L, 1.5L, 2.5L, ....$ Of source, given that the latitude value tiles, there are infinite "poles" and infinite "equators".

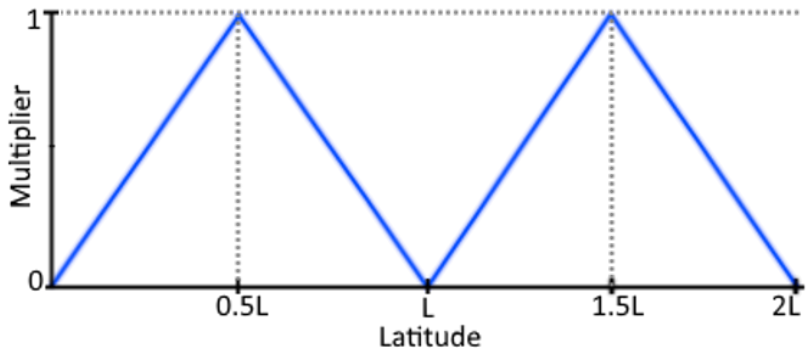


Figure 3.16: Function that shows how the multiplier correlates with the value $L$ for latitude tiling.

Aquatic biomes are not considered, and no aquatic elements are generated. As such, the only behaviour associated to elements such as oceans and lakes is to prohibit the designation of terrestrial biomes.

## 3.11 Summary

This chapter describes all relevant design decisions and algorithms regarding the creation of the world generation framework. It also presents both approaches that were taken, as well as explaining in detail each algorithm for the creation the chunks' elements. Table 3.2 provides a summarised comparison between the two approaches, by element.

Each element section contains the description for both approaches and a subsection regarding consistency issues that arise along with the presented solutions.

Table 3.2: Approach comparison, by element

| **Elements** | **Approach A** | **Approach B** |
|---|---|---|
| Terrain | Generates in a *count* radius, resulting in $(2count + 1)^2$ terrain surfaces. | Generates for the central chunk only, resulting in 1 terrain surface. |
| Ocean | Generates in a *count* radius, resulting in $(2count + 1)^2$ ocean surfaces. | Generates for the central chunk only, resulting in 1 ocean surface. |
| Highway | Generates by connecting existing cities, up to a limit distance. | Generates a large network around the chunk's centre position, within a given radius. Network edges represent highways, whilst vertices represent city centres. |
| City | Generates within the area of created terrain, by filtering a Perlin noise distribution. | Generates concentric cities around the highway network vertices. |
| River | Generates starting at the river mouth and connecting points in an uphill direction. | Generates starting at river mouth and moving uphill in a given direction chosen from a directions array. |
| Lake | Generates a bounded area by flooding terrain minimums above ocean. | |
| Forest | Generates a bounded area by filtering a Perlin noise distribution. | |
| Biome | Generates a matrix of biome values by intersecting 2 Perlin noise maps (corresponding to temperature and precipitation) and a given latitude value. | |

# Chapter 4

# Implementation

The following chapter explains the approaches and steps that were taken towards developing the **world generation framework**. Due to time constrains and complexity problems that arose along the framework development, not all of the elements that were mentioned in the design section were able to be created in the framework, and will not have their counterpart here.

For each generated elements, its subsection will describe its role in the generation framework and the rendered results in the virtual 3D world.

The chapter introduces the **tool** Sceelix, as well as some important concepts regarding its architecture and data management. Afterwards, each **element**'s implementation will be presented according to each approach. Finally, there will be an **overview and summary** of the framework and its results.

## 4.1 Sceelix

The framework was developed in Sceelix, a procedural generation engine which provides a visual graph-based editor and a C# API. Graphs, in Sceelix, are sets of nodes. Nodes, as mentioned in the documentation page (see [Sce]), represent "operations that create, analyse, transform, import, export or otherwise do something with data". That said, the framework functioned as an independent plugin featuring, besides the built-in nodes, a large set of custom nodes, data entities and custom data visualisers (visualisers translate the data into formats that can be rendered).

Nodes possess input and output ports, and can be linked through these ports, forming a graph structure, as can be seen in figure 4.1. Another important aspect of nodes is the ability to be super-nodes: that is, nodes that contain graphs. These nodes are useful to provide modularity and to comprise complex node modules into an abstract node. A super-node is presented in figure 4.1 (notice the octagonal shape).

When executing a graph, data flows from output ports to input ports until the execution is finished - that is, there are no more output ports with unhandled data. At this time, the results that can be rendered are shown in a 3D visualizing window.

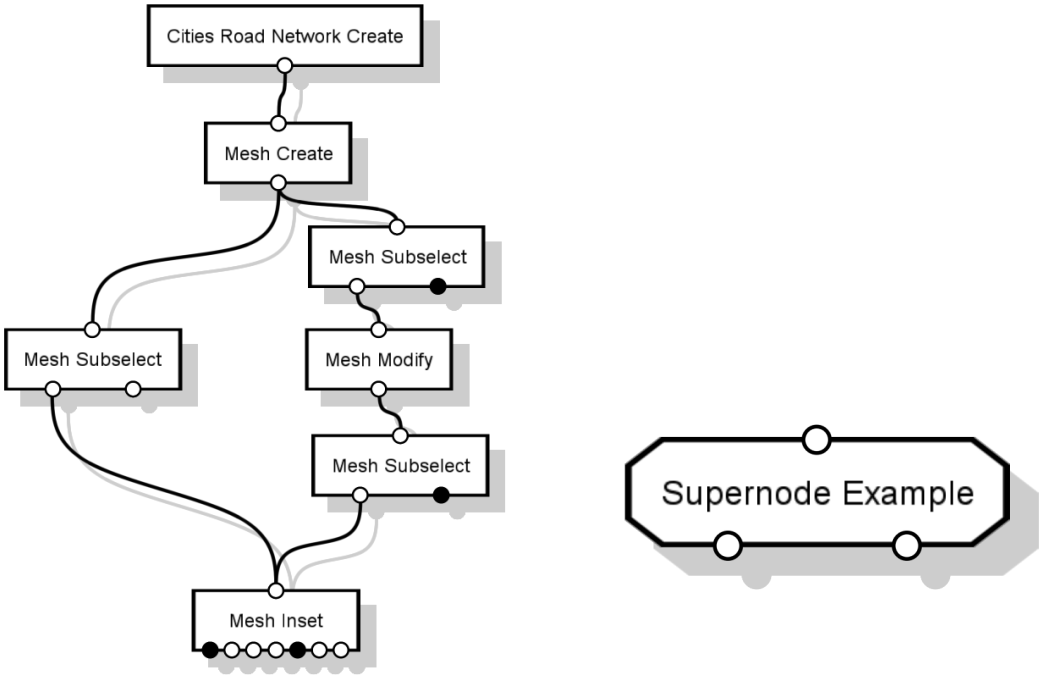


Figure 4.1: On the left, a Sceelix graph example. On the right, an example of a super-node.

In Sceelix, all data is represented in the form of **entities**. Although Sceelix possesses a wide number of entity types, the following are more relevant for this work: **surfaces**, **paths** and **meshes**. Surfaces are up-oriented grid of evenly spaced cells, optimized for the representation of terrains (containing a *heightmap*, a *colormap*, a *normalmap* and a given cell size). In this work, it was also used for the representation of the ocean surface. **Paths** represent networks, containing vertices and edges (similarly to a graph structure). This kind of entity is very useful for the representation of networks and simple paths. **Meshes** are widely used in 3D modelling applications. Each mesh represents a set of polygonal faces, connected to vertices. For this project, meshes are used to visually represent rivers, roads, highways and city lots.

Besides the mentioned entities, the following custom entities were created: **boolean maps**, **value maps** and **vertex lists**. A **boolean map** is an entity that contains a matrix of boolean values. A **value maps** is an entity that contains a set of useful variables that are used across the execution. A **vertex lists** represents a set of individual, unlinked vertices.

### 4.1.1 Note on Approaches

As mentioned in the section 3.2, approach A was designed and implemented prior to B. In so being, B shows a more mature design. Accordingly, the implementation was also different for both. Approach A uses more nodes. This was an attempt to simplify each node's operation (more nodes with simple operations versus less nodes with lots of operations). This revealed to be a cumbersome tactic, as data was required to flow from node to node, damaging graph readability, with many edges crossing themselves from node to node.

Approach B, on the other hand, does most processing and element creation internally, in a single **master node**, with only some final nodes that perform small transformation and mesh selection operations, resulting in a much cleaner visual graph.

## 4.2 Approach A

For approach A, seven primary nodes can be identified, as shown in figure 4.2. This nodes are separated by functionality:

- **Create Data**: this is the heaviest node. It is responsible for generating all the lookup data: terrain surfaces, river's control vertices, ocean boolean maps and road network (**Copy** is used simply to separate the several routes of the terrain surfaces, in order to keep the graph uncluttered). This node also creates the initial ocean-land connections for rivers (ocean vertices are then discarded);

- **City Networks**: this node creates the city networks from the road networks received from node *Create Data*. Each independent city network is separated and sent to node *Rivers*. Each city vertex is grouped by city in vertex list entities and send to node *Highways*;

- **Rivers**: this node's functionality is two-fold: first, it creates the rivers by connecting the land vertices received from node *Create Data*. Afterwards, it uses the information about the city networks received from node *City Networks* and "cuts" the paths that are intersected by the generated rivers, sending the resulting network to node *Streets*;

- **Highways**: this node creates the highway connections by using the cities vertices received from node *City Networks* as reference for possible anchor points to connect highways to;

- **Streets**: this node refines the city network received from node *Rivers*, placing the resulting network on the terrain and sending it to node *City Lots*;

- **City Lots**: this node creates the city street meshes from the refined networks received from node *Streets*. The mesh creation process of Sceelix allows for the automatic creation of the enclosed city lots in a two-for-one process. The lots are then sent to node *Land Usages*. The second input port for this node are the cities vertices. This vertices are received from node *City Networks* because they are simpler structures to deal with than networks and meshes, speeding up the evaluation of parameters distance to city centre;

- **Land Usages**: this node receives from node *City Lots* both the city lots and the cities vertices and does the attribution of land usages per lot, outputting the three separate lot types.

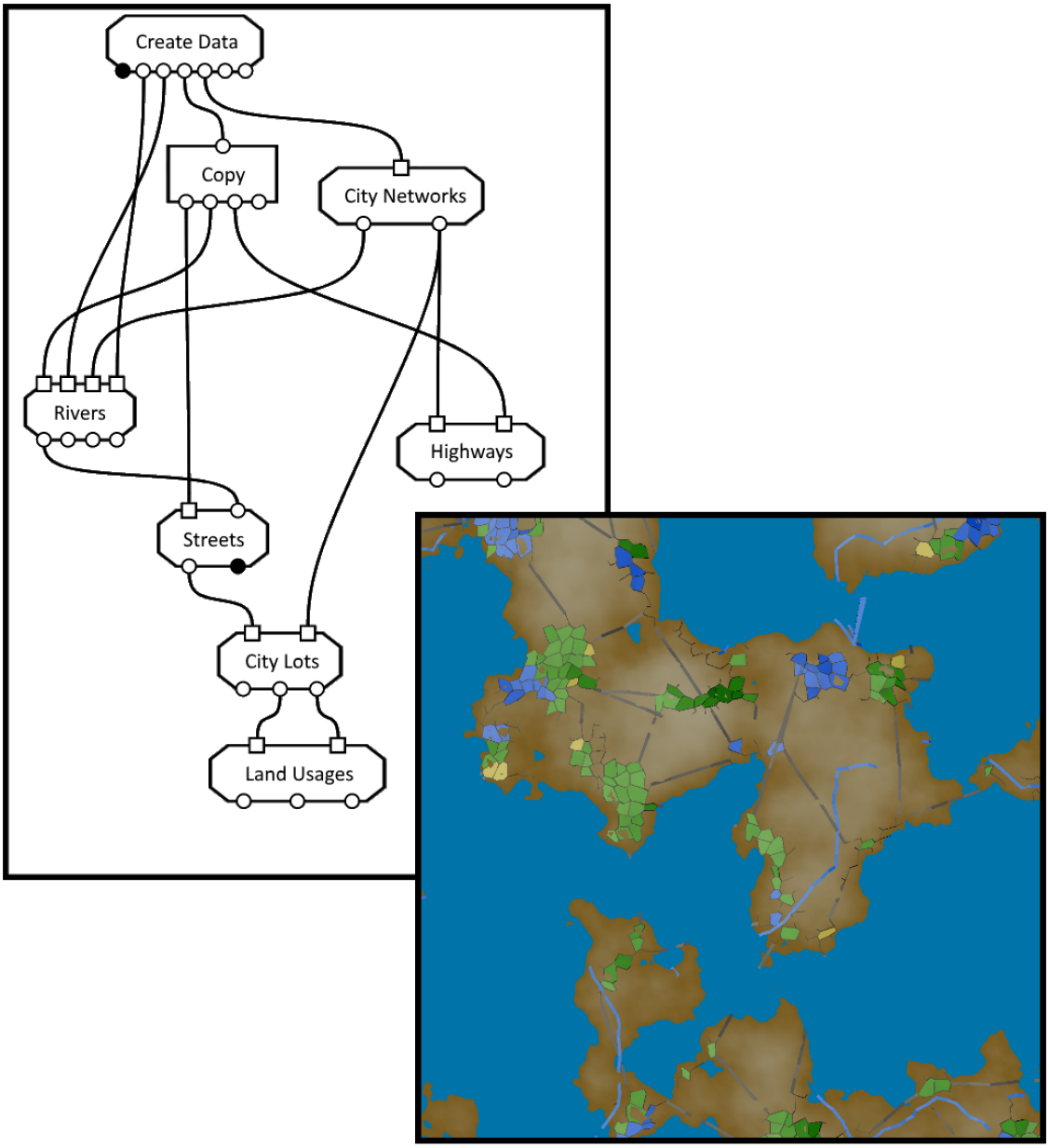


Figure 4.2: Approach A: graph and generation results

### 4.2.1 Observations

**Terrain & Ocean** As was mentioned in section 3.3.1, this approach uses a single Perlin value, resulting in a smooth topology. Since Perlin noise is tileable and continuous, it is not easy to distinguish where one chunk ends and another begins, which is a step towards ensuring coherence across chunks.

Regarding the generation of the remaining elements, this approach uses the Sceelix surface's own methods to query for height values across the generation, resembling the process of ray-casting. This introduced three main constraints: **a)** terrain must be generated before any other element which requires height information (this is valid for all elements except oceans); **b)** the

terrain entities must be sent to all needing graph nodes, which means that the graph can become visually cluttered, as it can be observed in figure 4.2, where four copies of the terrain are sent from the *Copy* node (these "copies" should not be confused with actual full data copies. Data is not duplicated, all surfaces sent from the *Copy* node are the same internal surface); **c)** although the Sceelix surface method is reliable for querying height, it uses bilinear interpolation. The problem with this interpolation is the fact that on the surface edges, height values are obtained by analysing less vertices than are usually required for a correct bilinear interpolation. As a result, two consecutive surfaces may obtain different height values for the same position, and these two values may even be different from the Perlin value of the terrain in that position.

**Cities** This approach does not generate border-intersecting city lots. As such, the issue described in section 3.5.6 regarding city lots on chunks border is not present here. On the other hand, by not generating these lots, a distinct orthogonal clearance is created in cities that cross chunk's borders, as it can be observed in figure 4.3.

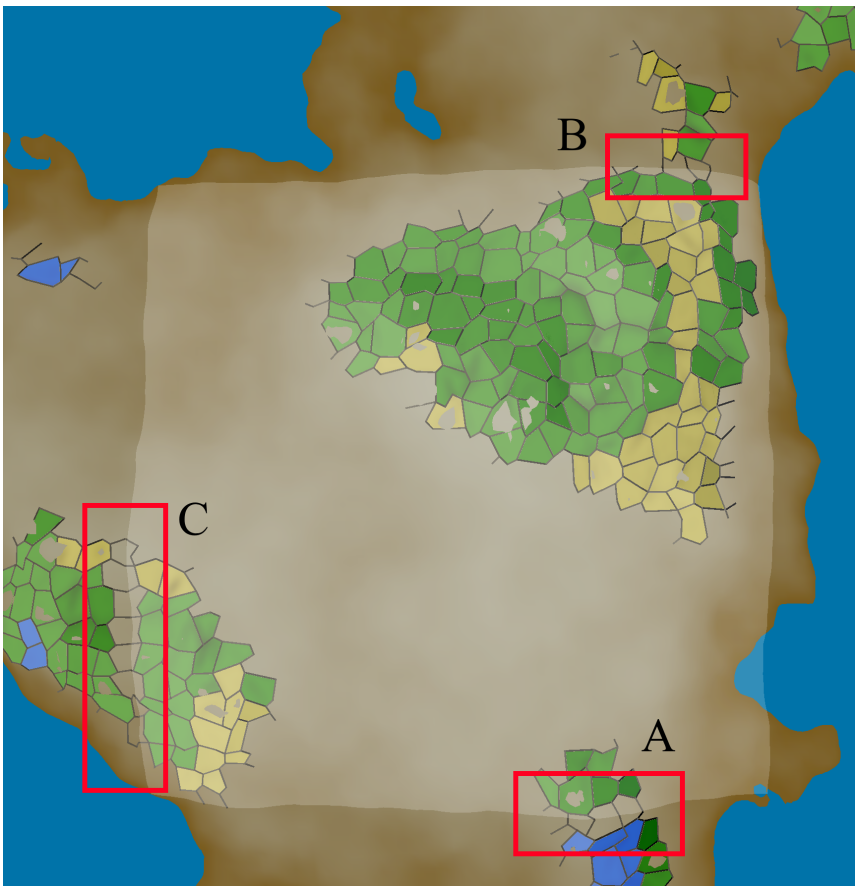


Figure 4.3: Approach A: no city lots in chunk's borders

**Highways** Approach A implements the city-based network for highways. That is, highways are created after cities. Since this process requires only information about the cities and the terrain, the graph flow is relatively simple and isolated, as seen in figure 4.2 (node "Highways"). As can be

observed, the highway network can become quite dense, resulting from the high density of small cities.

**Rivers** The Perlin vertex distribution, even being a random one, allows for the creation of reasonably long rivers, spanning for several chunks and demonstrating no coherence issues, as shown in figure 4.4 (the area is that of 9 chunks, $3 \times 3$ - black dots represent the generated Perlin vertices).

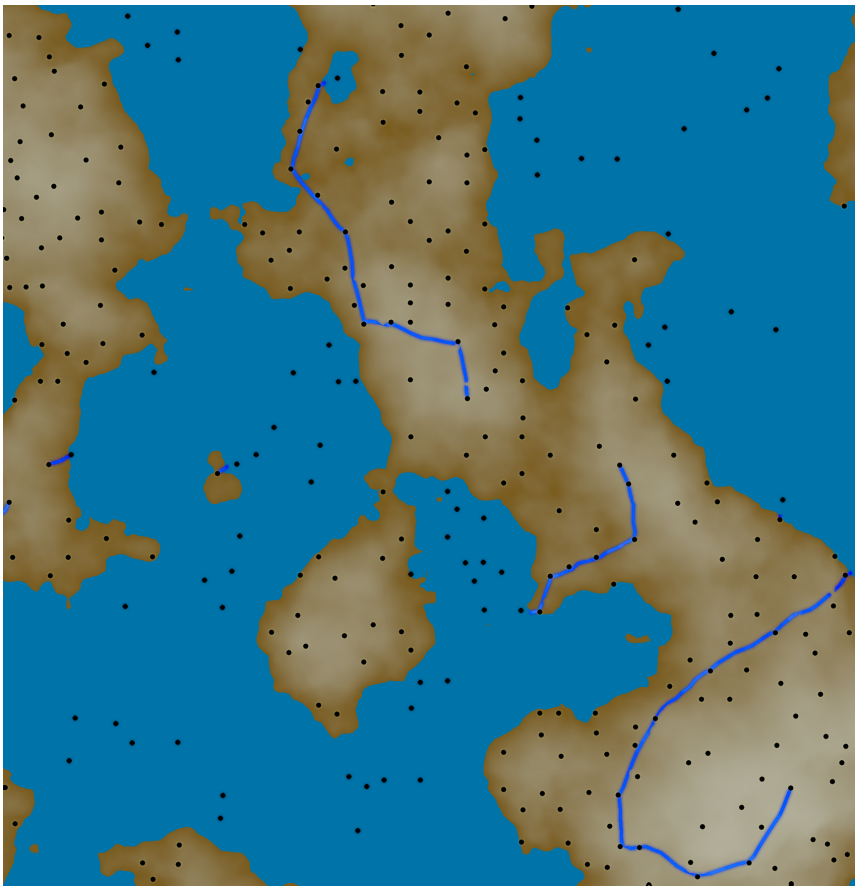

Figure 4.4: Approach A: generated rivers in a $3 \times 3$ chunk area

## 4.3 Approach B

As shown in figure 4.5, approach B can be summarised in eight nodes (two main nodes, on top, and six less important nodes, on the bottom):

- **Value Map**: this node creates the value map with all the input parameters and sends it to node B. This node is **not** repeated when several chunks are created. Instead, the same value map is sent to all B nodes (since the generation results depend on the same input values, it would be redundant to create several instances).

- **Create Chunk**: this node creates all the world elements. The process is either run in parallel or serialised. In serial order, the generation is executed as follows:

  1. The terrain surface is generated;
  2. The ocean surface is generated;
  3. The highway master network is generated;
     (a) The road network is created from the highway network vertices. The highway network is then adapted to connect the actual city vertices. This tight dependency prevents this two processes from being run in parallel (that is, they can still run parallel to the other processes);
     (b) The city meshes (streets and lots) are created and lots are attributed a land usage. Given their independence, each city - and even each lot - can be processed in parallel.
  4. The river network is generated, and any intersections are removed.

- **Roads**: this node adds additional information to the generated road meshes;

- **City Lots**: this node separates the three different lots types, applying a different colour material to each (commercial in blue, residential in green, industrial in yellow). Furthermore, it adds additional information to the resulting lots;

- **Highways**: this node creates the highway meshes, applies them a colour material and adds more information to the resulting meshes;

- **Rivers**: this node creates the river meshes, applies them a colour material and adds additional information to the resulting meshes;

- **Terrain**: this node applies a colour material to the generated terrain and adds additional information to the resulting surface;

- **Ocean**: this node applies a colour material to the generated ocean and adds additional information to the resulting surface.

For all element nodes, additional information is added, as mentioned. This relates to the chunk they belong to, data like length (in case of paths like rivers) and, for example, the city they belong to (in case of city lots and roads).

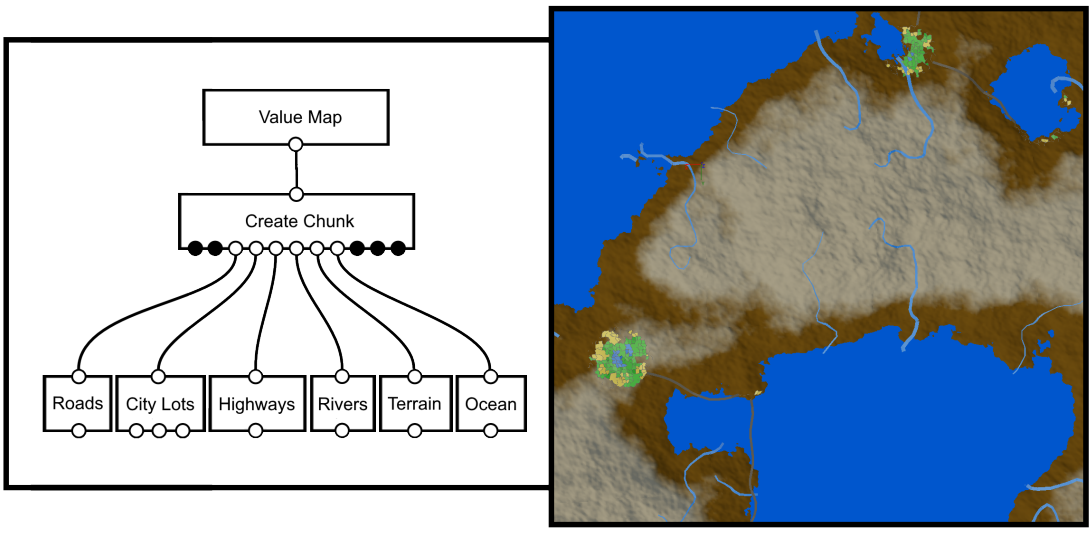


Figure 4.5: Approach B: graph and generation results

### 4.3.1 Observations

**Terrain & Ocean** As acknowledged in section 3.3.2, the Perlin values used for approach B allow for a more dynamic terrain, with unique features, such as isolated mountain peaks and a more uneven appearance, as shown in figure 4.6. Notice how the application of a second noise layer provides for a more detailed surface, with a more rough appearance than the one generated by the previous approach.

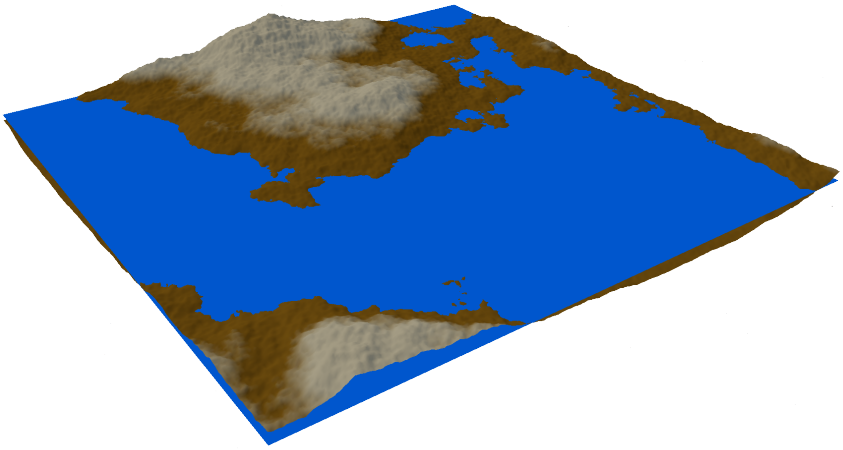

Figure 4.6: Approach B: generated terrain and ocean

Regarding the graph structure, unlike the previous approach, this one takes advantage of the fact that Perlin noise is a self-contained method, which can be queried at any time, provided that the input parameters are known (which is guaranteed by the value map sent to master node, as mentioned before). This does not require the terrain surfaces to be sent to all elements that require height values, allowing for a simpler execution process (although this is not evident by simply comparing the graphs, terrain generation can be executed in parallel, as mentioned in the beginning of section 4.3).

**Cities** The results for this approach can be seen in figure 4.5. Since the centres and radii are known values for each city, the calculations for parameters *historicWeight* and *centreWeight* (see section 3.5.3.3) are much simpler and direct than with the former method. This ease of calculation makes land usage attribution much more predictable, as can be observed in the same figure: cities are clearly dominated by commercial hubs, encircled by residential areas and industrial outskirts. However, it is also obvious that cities are much less dynamic, as their shape is clearly based in a circumference. And although this obviousness is diminished by decreasing the density with distance (as mentioned in the algorithm description in section 3.5.1.2), it is not entirely removed.

As mentioned in section 4.2.1, approach A does not generate city lots in chunks borders, so as to avoid inconsistencies between chunks. However, as mentioned in section 3.5.6, an assumption was made to prevent this inaccuracies: one can generate the entire city lot as long as the centroid belongs to the chunk. This approach applies that knowledge and results are consistent, as shown in figure 4.7. The black lots have their centroids inside the upper chunk. White blocks have their centroids inside the lower chunk. Despite the fact that these lots all cross the chunk's borders, they are fully generated if their centroids are inside the chunk. When both chunks are generated, all lots are covered, and the city is fully generated.

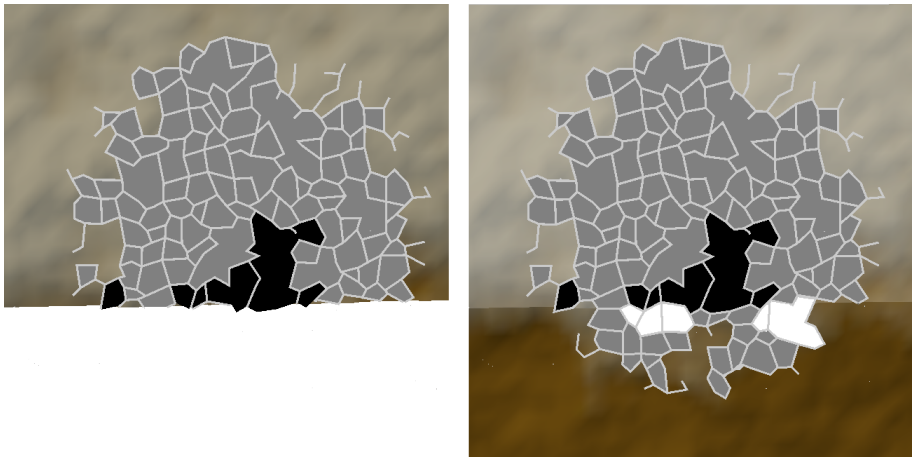

Figure 4.7: Approach B: city lots are consistent on chunks borders

**Highways** The generation based on a master network (see section 3.6.2) provides a more controlled density of the resulting network, unlike the previous approach. This can be observed in figure 4.8. Unfortunately, this process also carries some disadvantages: since the master network

generation algorithm is based on a regular grid, cities are almost always evenly spaced (saved for some exceptions, where vertices are underwater and therefore removed from the grid).

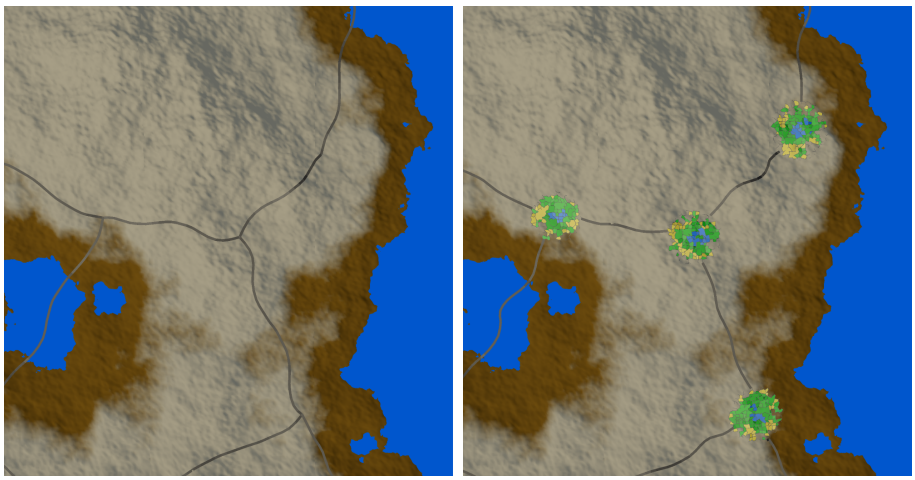

Figure 4.8: Approach B: generated highways

**Rivers** The use of directional vectors for this approach revealed advantages and disadvantages. Albeit being more crooked than on the previous approach, the generated rivers are generally smaller (see figure 4.9). This may be a direct consequence of the rougher terrain, "fooling" the algorithm by providing more spots for local minimums and prematurely stopping the generation algorithm).

Figure 4.9: Approach B: generated rivers

## 4.4 Summary

This chapter presented the implementation of all generated elements in the tool Sceelix, as well as the generated worlds and some observations regarding the obtained results.

For both approaches, of all the elements mentioned in chapter 3 following were implemented: **terrain**, **ocean**, **rivers**, **cities** and **highways**.

**Terrain** The use of more than one layer of Perlin noise for approach B provided a rougher terrain with more unique features, such as lonely high peaks and deeper valleys, as well as providing the means to postpone pattern repetition, and in so re-enforcing the perception of an infinite world. For

both approaches, however, there is a lack of control of the generated features. Since generation is to be infinite, it is perhaps unrealistic to expect a great control on the generated terrain. However, an approach like the one presented by Hnaidi [HGA+10] could create a more realistic and natural topology.

**Ocean** Both approaches represent the ocean as a single height surface. Since the ocean is used to clip, for example, city road networks, this simplicity works for the system's benefit, providing immediate query results. Additionally, a single height surface is simple to represent and render. However, representing an ocean as a single height value prevents the formation of land masses beneath sea level.

**Rivers** Although both approaches generate rivers uphill, the first approach chooses the anchor points from a set of Perlin distributed points, whilst the second approach chooses the starting point from a set of ocean points, using a set of directional arrays to explore the surrounding terrain in search of valid points. Despite the different methods, the rendered results are quite comparable: rivers are generated with a reasonable length (few small rivers) and they meander, not always following a straight path. There are, however, limitations to these methods: the amount of rivers that are generated is based on probability, and not always subject to a tight control. Additionally, no branches are created, generating simple "one course" rivers.

**Cities** City generation for both approaches is fundamentally different. Approach A generates cities by filtering a Perlin noise map, whilst approach B generates radial concentric cities. The results are quite different: approach A generates more diverse city shapes, with a wide range of sizes and shapes, resulting in a mixed set of city layouts, which could work in favour of providing a more unique experience for an observer. Approach B creates circumference-like city layouts. Although these layout vary in their radii and outskirts density, they still appear quite similar to each other. However, since their generation is more manageable than with the previous approach, this one is able to more accurately classify city blocks with the intended land usage.

Regarding cities in chunk's borders, approach B does not clip border city lots, but instead generates them using a sort of boolean "difference" approach: all lots which cross borders and are not generated by one chunk, will be generated by the neighbour. Each city lot is generated by the chunk which contains its centroid.

**Highways** For both approaches, highway generation follows quite different approaches. In the first approach, highways connect generated cities. Since the cities generated by this approach are large in number, the generated highway count is also quite high, creating a dense highway network. For approach B, the highway network is generated first, and is therefore subject to a more controlled generation. However, since this network is generated based on a regular grid, this generates vertices that are more or less evenly spaced, resulting in a regular distribution of cities, losing the more "random looking" distribution of cities obtained by the previous approach.

# Chapter 5

# Validation

This chapter describes the evaluation and validation tests that were applied in order to corroborate the correctness of the solution, as well as their results. The tests focus on the consistency of elements required for an infinite, continuous and seamless generation of worlds, and should not be confused with the quality of the representation of those elements, nor the plausibility of the generated worlds.

## 5.1 Considerations

Chapter 3 presented some consistency regards for each element, which related to continuity and coherence issues. It also described the applied solutions to face those problems and explained, when necessary, why they were valid or infeasible. This chapter presents and explains the tests that were applied to validate the presented solutions, considering the knowledge applied in all the subsections of chapter 3.

## 5.2 Tests

This sections describes the two types of tests that were devised to validate the framework: **border consistency** and **overlay consistency** tests. As expected, chunks were generated independently, without any information on the neighbours (that is, the data that is generated for each chunk in the execution process is not used in any way by the next chunks).

### 5.2.1 Border Consistency

Border consistency tests were created to validate the seamless connections between elements that cross or interact in any way with a chunk's border. Two main types of consistencies were evaluated: **surface** and **network**.

**Surface** border consistency evaluates the terrains consistency with other terrains in neighbouring chunks, by comparing edge vertices. If an edge vertex is not in the same position as

the neighbour's chunk corresponding vertex[1], then the consistency test fails for that vertex. In figure 5.1, the vertices within the green box would be compared for edge consistency.

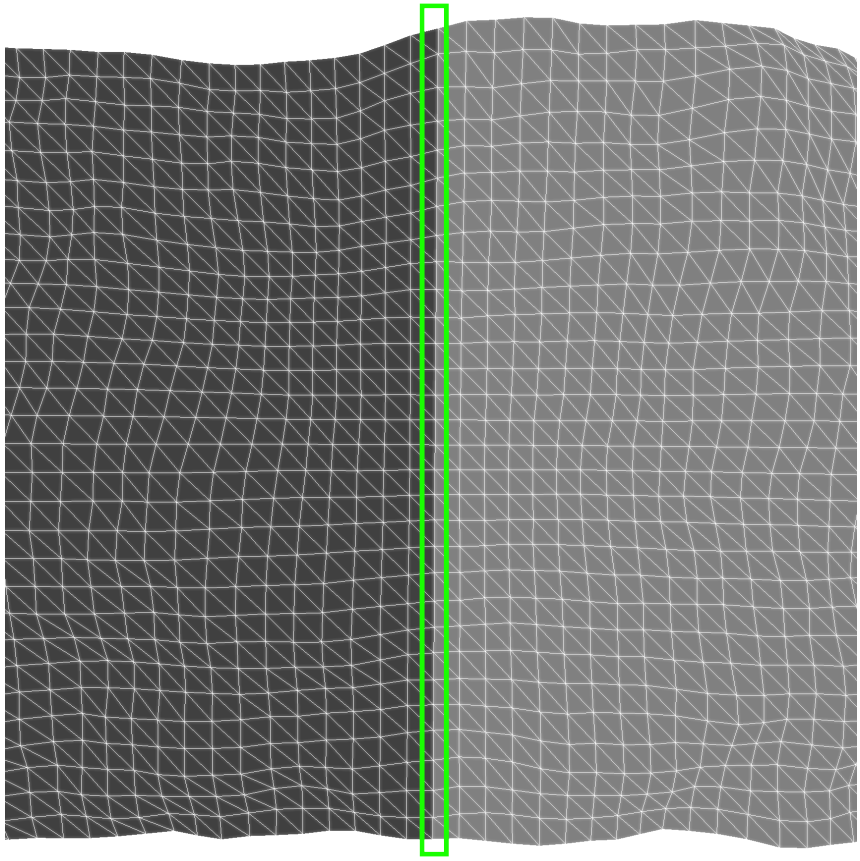

Figure 5.1: The vertices within the green box are compared for border consistency.

**Network** border consistency evaluates all clipped edges of a network. Clipped edges are edges which cross a chunk's border and must be "cut" when generation ends, so as not to extend outside the borders of the generated chunk. Since network generation should be consistent across chunks, the neighbouring chunk should mirror this behaviour, generating the remaining network and, consequently, the remaining parts for all clipped edges. This can be observed in figure 5.2, where the yellow vertex is a clipped vertex. Then, the partial networks (red and blue) should both have this vertex.

As mentioned in section 3.5.6, city lots are not clipped. As such, border consistency tests are not required. Nevertheless, the validity of cities that cross the borders of chunks depends on the validity of the road networks, since lots are generated from the road networks (the same networks generate the same lots, so the consistency of networks relates to the consistency of generated city lots).

[1]This test considers a threshold of 0.001 to try and compensate for eventual floating point rounding errors. This threshold value was determined by the fact that it is small enough so that two distinct vertices should not be considered, yet large enough that it should cover any rounding errors.

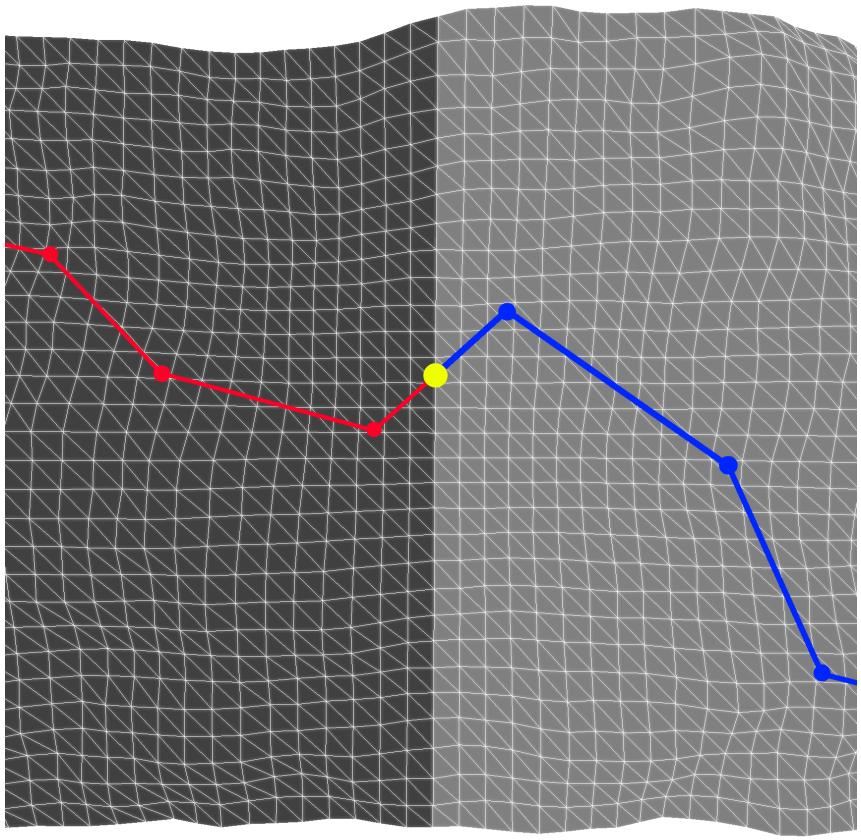

Figure 5.2: The yellow vertex is a clipped vertex. That is, it should be present in both the partial networks,in red and blue.

### 5.2.2 Overlay Consistency

Overlay tests evaluate the consistency across executions, by comparing several iterations of the same chunks. This tests are also performed for two types: **surface** and **network**.

**Surface** tests compares the chunk's terrain vertices against the same chunk's vertices for all iterations. If the vertices' positions are not equal[2], then this means that the system is not deterministic, and therefore not valid for the objectives of this work.

**Network** tests compare each network in a chunk against the corresponding networks in all iterations of the same chunk. For each edge, considering the two vertices that it connects, the validation system searches other executions for an edge that connects two vertices in the same positions[3]. This is valid because two edges that connect the same positions are, for practical purposes, the same edge.

### 5.2.3 Results

For each approach, three comprehensive tests runs were executed, as shown in table 5.1. The chunk size was set to 100 cells (101 vertices). Figure 5.3 shows an example of the chunk layout resulting from test run #1.

Table 5.1: Test Runs

| Test run | Iterations | Min. offset | Max. offset | Chunks (per iteration) |
|---|---|---|---|---|
| #1 | 3 | (15, 15) | (20, 20) | 36 |
| #2 | 3 | (495, 495) | (500, 500) | 36 |
| #3 | 3 | (-500, -500) | (-495, -495) | 36 |

#### Approach A

The results for approach A regarding border consistency are shown in tables 5.2, 5.3, 5.4 and 5.5 for terrains, roads, rivers and highways, respectively. Terrain consistency is absolute. Regarding networks, there are some inconsistencies. However, these represent a minority, creating a solid belief in the general correctness of approach A.

Table 5.2: Approach A: border consistency for terrains

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 36 | 12120 | 12120 | 1.00 |
| #2 | 36 | 12120 | 12120 | 1.00 |
| #3 | 36 | 12120 | 12120 | 1.00 |

[2]This test considers a small threshold, to compensate for eventual floating point rounding errors.
[3]See footnote 2.

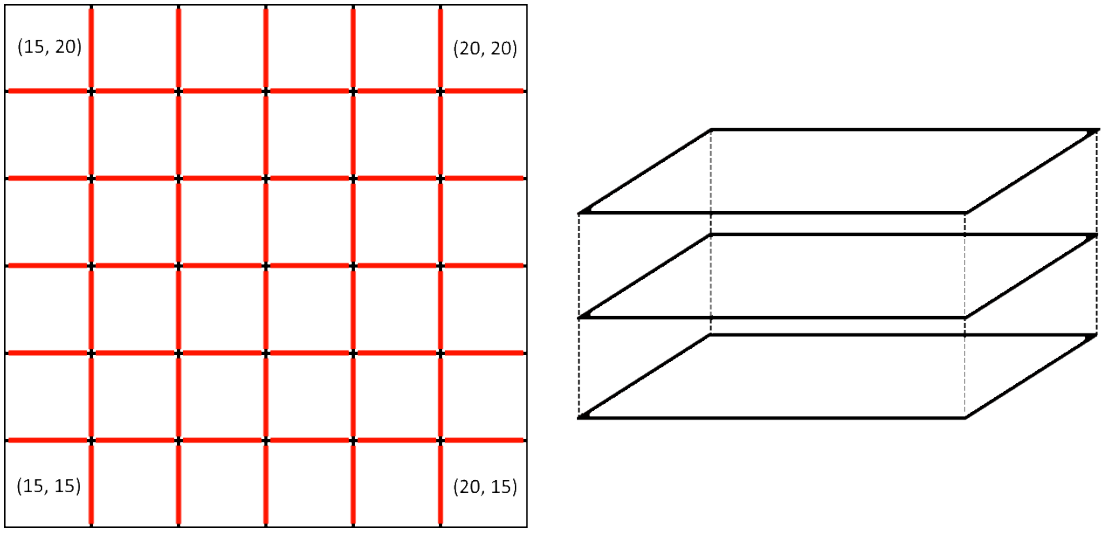


Figure 5.3: As observed, from offset (15, 15) to (20, 20), 36 chunks are generated. The marked red edges are evaluated for border consistency. The right image shows how overlay testing works: the results of the 3 iterations are compared between themselves for equality.

Table 5.3: Approach A: border consistency for roads

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 92 | 240 | 240 | 1.00 |
| #2 | 199 | 266 | 266 | 1.00 |
| #3 | 207 | 306 | 305 | 0.99 |

Table 5.4: Approach A: border consistency for rivers

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 52 | 34 | 34 | 1.00 |
| #2 | 117 | 64 | 64 | 1.00 |
| #3 | 109 | 66 | 61 | 0.93 |

Table 5.5: Approach A: border consistency for highways

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 124 | 80 | 79 | 0.99 |
| #2 | 239 | 127 | 124 | 0.98 |
| #3 | 261 | 133 | 133 | 1.00 |

Overlay results for approach A are shown in tables 5.6, 5.7, 5.8 and 5.9 for terrains, roads, rivers and highways, respectively. These results provide a high grade of assurance of the system's determinism regarding separate executions.

Table 5.6: Approach A: overlay consistency for terrains

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 108 | 1101708 | 1101708 | 1.00 |
| #2 | 108 | 1101708 | 1101708 | 1.00 |
| #3 | 108 | 1101708 | 1101708 | 1.00 |

Table 5.7: Approach A: overlay consistency for roads

| Test run | Element count | Edge count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 276 | 9044 | 9044 | 1.00 |
| #2 | 597 | 11062 | 11062 | 1.00 |
| #3 | 621 | 11122 | 11122 | 1.00 |

Table 5.8: Approach A: overlay consistency for rivers

| Test run | Element count | Edge count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 156 | 594 | 594 | 1.00 |
| #2 | 351 | 1056 | 1056 | 1.00 |
| #3 | 327 | 1011 | 1011 | 1.00 |

Table 5.9: Approach A: overlay consistency for highways

| Test run | Element count | Edge count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 372 | 531 | 531 | 1.00 |
| #2 | 717 | 847 | 847 | 1.00 |
| #3 | 783 | 924 | 924 | 1.00 |

### Approach B

The results for this approach regarding border consistency are shown in tables 5.10, 5.11, 5.12 and 5.13 for terrains, roads, rivers and highways, respectively. Consistency for the terrains is absolute. Regarding networks, the inconsistencies are almost non-existent, even more so than with the previous approach, corroborating the idea that this approach is more solid, as was to be expected, considering that it was designed and implemented later on the development of the framework.

Overlay results for approach A are shown in tables 5.14, 5.15, 5.16 and 5.17 for terrains, roads, rivers and highways, respectively. As for the previous approach, the results increase the confidence on the system's determinism across executions.

Table 5.10: Approach B: border consistency for terrains

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 108 | 12120 | 12120 | 1.00 |
| #2 | 108 | 12120 | 12120 | 1.00 |
| #3 | 108 | 12120 | 12120 | 1.00 |

Table 5.11: Approach B: border consistency for roads

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 52 | 244 | 244 | 1.00 |
| #2 | 42 | 183 | 183 | 1.00 |
| #3 | 46 | 270 | 270 | 1.00 |

Table 5.12: Approach B: border consistency for rivers

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 69 | 62 | 62 | 1.00 |
| #2 | 97 | 125 | 125 | 1.00 |
| #3 | 77 | 78 | 78 | 1.00 |

Table 5.13: Approach B: border consistency for highways

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 24 | 40 | 40 | 1.00 |
| #2 | 22 | 45 | 42 | 0.93 |
| #3 | 22 | 42 | 42 | 1.00 |

Table 5.14: Approach B: overlay consistency for terrains

| Test run | Element count | Vertex count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 108 | 1101708 | 1101708 | 1.00 |
| #2 | 108 | 1101708 | 1101708 | 1.00 |
| #3 | 108 | 1101708 | 1101708 | 1.00 |

Table 5.15: Approach B: overlay consistency for roads

| Test run | Element count | Edge count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 156 | 6950 | 6950 | 1.00 |
| #2 | 126 | 5899 | 5899 | 1.00 |
| #3 | 138 | 5949 | 5949 | 1.00 |

Table 5.16: Approach B: overlay consistency for rivers

| Test run | Element count | Edge count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 207 | 517 | 517 | 1.00 |
| #2 | 621 | 1008 | 1008 | 1.00 |
| #3 | 231 | 579 | 579 | 1.00 |

Table 5.17: Approach B: overlay consistency for highways

| Test run | Element count | Edge count | Validated | Consistency |
|---|---|---|---|---|
| #1 | 72 | 486 | 486 | 1.00 |
| #2 | 66 | 397 | 397 | 1.00 |
| #3 | 66 | 353 | 353 | 1.00 |

## 5.3 Discussion

Considering the shown test results, it is possible to conclude that, despite some minor errors, the solution is all-round adequate to represent a seamless and continuous world, whilst containing complex elements, such as highways, cities, roads and rivers.

The border consistency tests demonstrate that, for most cases, the element continuity across chunks is guaranteed by the generation system. This is particularly true for terrains and road networks, which present an almost perfect result. For highways and rivers, given that the number of non-validated vertices is marginal, it has been attributed to unresolved clipping issues and special edge cases resultant of incorrect input parameter values.

Regarding the overlay tests, the results are perfect, demonstrating no deviance or inconsistency whatsoever, for the evaluated characteristics. This is extremely important for it ensures the determinism of the system, since all tested executions were completely independent.

Although the number of test-runs was a small one, the final number of evaluated features is relatively high, increasing the confidence in the obtained results.

# Chapter 6

# Conclusions

As seen in the previous Chapter 2, there have been suggested many approaches on how to represent and procedurally generate complex world elements. Although some of these works render impressive results, these approaches have been applied to finite contexts. In fact, the research for this project found no relevant papers focusing on infinite procedural generation, despite the fact that many commercial products, namely computer games have been applying this kind of generation for years. The purpose of this work was to lay a foundation stone on the matter of infinite generation - especially considering a world composed of elements such as oceans, lakes, cities, rivers, highways and forests -, devising some solutions as to how these elements would be generated.

One of the most important notions that was introduced in section 1.2 and described afterwards in section 3.1 was the notions of chunks. That is, the world division in finite manageable sections. This division introduced some issues regarding border continuity for the aforementioned elements. In chapter 3 some solutions were described to overcome the continuity issues. For the terrain, these issues are automatically resolved by the nature of Perlin noise. However, for other elements, some techniques had to be implemented. For rivers and highways, for example, given that they are generally long-spanning paths, the solution was to simply clamp their maximum spanning length.

Still in chapter 3, two different approaches are discussed for terrains, cities, rivers and highways and compared for efficiency and robustness, as well as the main consistency issues that might arise for each approach.

Chapter 4 describes the implementation of the framework for both approaches in the procedural engine Sceelix, as well as the rendered results. This chapter also compares the implementation for both approaches in terms of performance and general complexity.

In chapter 5, tests and validations methods are presented and discussed. From the test results it is concluded that the presented approaches are, for most cases, correct, guaranteeing the continuity and coherence of the generated words with all their constituting elements.

## 6.1 Objective Completion

The main objectives of this work (see section 1.4) were:

- Creation of a framework that could infinitely generate a world composed of complex elements, such as terrains, oceans, lakes, forests roads, rivers and buildings;
- Development of an "exploration" module that would traverse the world, allowing it to be generated continuously by parcels;
- Creation of a validation module that would evaluate the consistency of the generated world.

The framework was successfully developed, with promising results, as can be seen in figure 4.5. The generation successfully creates oceans and terrains, a dense river network, cities and their connecting highways.

The "exploration" module was not developed. Its implementation would have automatised the process of continuously creating the world parcels, being particularly interesting for the purpose of games and virtual simulations. Nevertheless, it has no effect on the developed solutions, being more of an implementation object. As such, in developing a framework that successfully introduced some concepts to tackle the problem of infinite world generation, this project is considered to have completed its primary objective (the "backbone" of the objective section, so to speak).

Finally, and perhaps most importantly, the generated results and the validation tests by the dedicated validation module prove that it is possible to generate, in a parametrised and deterministic way, a virtual world, which is both continuous and consistency, whilst containing more intricate elements, such as highways, rivers and cities. In so being, the main research question (see section 1.3) is considered to have been positively answered.

## 6.2 Future Work

For future projects, it would be interesting to delve deeper in some of the generated elements, whilst implementing the non-generated ones. For oceans and terrains, it could be interesting to provide more control on the final results by introducing, for example, the notions of "continents" and "islands". For terrains, it would also be very interesting to provide more control on the existence of mountains, perhaps by using an approach similar to the one presented in [HGA+10], taking advantage of ridge lines.

Regarding rivers and lakes, it would be most interesting to explore branching and river deltas, perhaps using some of the knowledge presented in [GGG+13], so as to create more dense, dynamic and realistic results. The generation of lakes could perhaps decrease the "randomness" level of rivers, since they would start to flow in more realistic ways, as lakes are generally areas were rivers flood the terrain.

As for highways, their generation was overly simplified in this work. Notions introduced in [GPMG10] could immensely benefit the realism of the solution, as well as providing an extra layer of control to the world designer.

The cities implementation approaches in this work lack some of the notions mentioned in section 2.4.2.1, namely primary and secondary road networks, which could improve the precision of the method for attribution of land usages to the city lots. Furthermore, cities in the developed framework completely lacked buildings. Buildings are a well-discussed topic in several works (see [PM01, KM07, MWH+06], for example) and their generation was not a particular focus in this work. However, for urbanism analysis and realism, the development of buildings could be of particular interest. The implementation of biomes and classification of dominant cultures in the generated areas of the world would also enable the creation of culture and region-specific buildings with their own architectural styles.

As mentioned in section 2.6, this particular work provides a somewhat novel approach to world generation in the scientific context. As such, as mentioned in the previous paragraphs, there is large room for improvement. Nevertheless, it can provide a stepping stone for further scientific works in this challenging area of investigation.

# References


[BA05] Farès Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers. In *Proc. 3rd Int. Conf. Comput. Graph. Interact. Tech. Australas. South East Asia - Graph. '05*, volume 1, page 447, New York, New York, USA, 2005. ACM Press.

[CRU+09] Neil A. Campbell, Jane B. Reece, Lisa A. Urry, Michael L. Cain, Steven A. Wasserman, Peter V. Minorsky, and Robert B. Jackson. *Biology (8th Edition)*. Benjamin-Cummings Pub Co, 8th edition, 2009.

[For87] Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1-4):153–174, 1987.

[GGG+13] Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedřich Beneš. Terrain generation using procedural models based on hydrology. *ACM Trans. Graph.*, 32(4):1, 2013.

[GPMG10] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin. Procedural generation of roads. *Comput. Graph. Forum*, 29(2):429–438, 2010.

[GPSL03] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Undiscovered Worlds–Towards a Framework for Real-Time Procedural World Generation. In *Fifth Int. Digit. Arts Cult. Conf. Melbourne, Aust.*, 2003.

[Gro09] Saskia A Groenewegen. Procedural City Layout Generation Based on Urban Land Use Models. In *Proc. Eurographics 2009 - Short Pap.*, pages 45–48, Munich, Germany, 2009.

[HGA+10] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. *Comput. Graph. Forum*, 29(7):2179–2186, 2010.

[HMVI13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural Content Generation for Games: A Survey. *ACM Trans. Multimed. Comput. Commun. Appl.*, 9(1):1:1—-1:22, 2013.

[KM06] George Kelly and Hugh McCabe. A survey of procedural techniques for city generation. *ITB J.*, pages 87–130, 2006.

[KM07] George Kelly and Hugh McCabe. Citygen: An interactive system for procedural city generation. In *Proc. GDTW 2007 5th Annu. Int. Conf. Comput. Game Des. Technol.*, pages 8–16, 2007.

[LHD15] Xiaoming Lyu, Qi Han, and Bauke De Vries. Procedural urban modeling of population, road network and land use. *Transp. Res. Procedia*, 10(July):327–334, 2015.

## REFERENCES


[LWWF03] Thomas Lechner, Ben Watson, Uri Wilensky, and Martin Felsen. Procedural city modeling. *1st Midwest. Graph. Conf.*, pages 1–6, 2003.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614, 2006.

[Par14] Ian Parberry. Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data. *J. Comput. Graph. Tech.*, 3(1):74–85, 2014.

[Per85] Ken Perlin. An image synthesizer. *ACM SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[Per02] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):2–3, 2002.

[PGGM09] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou. Arches: A framework for modelling complex terrains. *Comput. Graph. Forum*, 28(2):457–467, 2009.

[PM01] Yoav I. H. Parish and Pascal Müller. Procedural Modeling of Cities. *28th Annu. Conf. Comput. Graph. Interact. Tech.*, (August):301–308, 2001.

[Sce] Sceelix Documentation Page. https://www.sceelix.com/documentation. Online; accessed 17-June-2017.

[SDG+09] Ruben M Smelik, Klaas Jan De Kraker, Saskia A Groenewegen, Tim Tutenel, and Rafael Bidarra. A survey of procedural methods for terrain modelling. *3AMIGAS - 3D Adv. Media Gaming Simul.*, (June 2015):25–34, 2009.

[STBB14] Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. *Comput. Graph. Forum*, 33(6):31–50, 2014.

[STDB10] Ruben M. Smelik, Tim Tutenel, Klaas Jan De Kraker, and Rafael Bidarra. Declarative terrain modeling for military training games. In *31st annual conference of the European Association for Computer Graphics*, volume 2010, 2010.

[STDB11] R. M. Smelik, T. Tutenel, K. J. De Kraker, and R. Bidarra. A declarative approach to procedural modeling of virtual worlds. *Comput. Graph.*, 35(2):352–363, 2011.

[SYBG02] Jing Sun, Xiaobo Yu, George Baciu, and Mark Green. Template-based generation of road networks for virtual city modeling. *Proc. ACM Symp. Virtual Real. Softw. Technol. - VRST '02*, page 33, 2002.

[Teo08] Soon Tee Teoh. River and coastal action in automatic terrain generation. In Leonidas Arabnia, Hamid R. and Deligiannidis, editor, *CGVR - Comput. Graph. Virtual Real.*, number 1, pages 3–9. CSREA Press, 2008.

[TYSB11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intell. AI Games*, 3(3):172–186, 2011.