Procedural Generation of Modern 3D Cities

Bachelor's thesis in Computer Science and Engineering

Theodor Angergård
Marcus Ansamaa
Alexander Arvidsson
Jacob Eriksson
Anton Håkansson
Viktor Truvé

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

Bachelor of Science Thesis 2020

Procedural Generation of Modern 3D Cities

Theodor Angergård
Marcus Ansamaa
Alexander Arvidsson
Jacob Eriksson
Anton Håkansson
Viktor Truvé

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

Procedural Generation of Modern 3D Cities

Theodor Angergård
Marcus Ansamaa
Alexander Arvidsson
Jacob Eriksson
Anton Håkansson
Viktor Truvé

Supervisor: Staffan Björk, Department of Interaction Design and Technologies
Examiner: Michael Heron, Department of Computer Science and Engineering

Bachelor's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A 3D city model that was generated and exported with CityCraft. The rendering of the model was done in Blender.
Typeset in LaTeX
Gothenburg, Sweden 2020

Abstract

This paper investigates how techniques in Procedural Content Generation (PCG) can be combined with computer graphics theory to generate digital 3D cities suitable for use in media such as games, movies, and advertisements. This research was conducted to discover ways to alleviate the otherwise time-consuming process of manually modeling such cities. The investigation analyzes previous work and existing PCG techniques in order to propose a new system capable of generating modern cities from scratch. The validity of this system was then demonstrated through the implementation of a city generation software.

As a result of this research, we contribute with *CityCraft*; a free, open-source, MIT licensed desktop application that interactively generates modern cities in real-time. These generated cities can be exported as 3D models into the royalty-free file format glTF, making them compatible with a wide range of third-party tools for further refinements. Although of less quality than manually modeled ones, the generated cities are produced in a fraction of the time, and the application itself requires no modeling or programming expertise to be used efficiently.

CityCraft demonstrates the potential of combining PCG and computer graphics to automate the process of modeling cities. It also provides insight into how specific techniques, such as Agent-based road generation, L-system-based buildings, and Level of Detail (LOD), can be integrated to achieve performant generation. The ambition with this contribution is that CityCraft can act as a useful stepping stone for further city generation research within the open-source domain.

Sammandrag

Denna rapport undersöker hur metoder inom *Procedural Content Generation* (PCG) kan kombineras med datorgrafikteori för att procedurellt generera virtuella 3D städer, lämpade för användning inom datorspel, film, och reklam. Denna undersökning utfördes med syftet att underlätta den annars tidskrävande processen av att modellera sådana städer manuellt. Utredningen studerar tidigare arbeten och existerande PCG metoder för att föreslå ett nytt system kapabel till att generera moderna städer från grunden. Potentialen av detta system demonstrerades sedan via utvecklingen av en stadgenerationsprogramvara.

Som ett resultat av denna undersökning, så bidrar vi med *CityCraft* – ett kostnadsfritt, *open-source*, MIT-licensierat datorprogram som interaktivt genererar moderna städer i realtid. De genererade städerna kan exporteras som 3D modeller till det avgiftsfria filformatet glTF, vilket gör de kompatibla med flertalet tredje parts verktyg där de kan förfinas vidare. Kvaliteten på dessa städer åstadkommer inte den kvalité som manuellt tillverkade städer tenderar att ha. Däremot tar generationen endast en bråkdel av tiden som manuell modellering kräver. Dessutom kräver applikationen inga förkunskaper inom modellering eller programmering för att kunna användas effektivt.

Realiseringen av CityCraft påvisar potentialen av att kombinera PCG och datorgrafik för att automatisera processen av att modellera städer. Det bidrar även med insikt till huruvida specifika metoder, så som Agent-baserad väggenerering, L–system-baserade byggnader, och *Level of Detail* (LOD), kan integreras för att uppnå goda resultat. Ambitionen med detta bidrag är att CityCraft kan agera som en språngbräda till fortsatt forskning inom stadsgenerering inom *open-source* domänen.

Acknowledgements

First, we would like to give our thanks to the open-source community of the world, which has provided us with many useful libraries that have helped reduce the complexity of our implementation. As such, we hope that our work will also prove useful for others by making our implementation open-source as well.

We would also like to give a special thanks to our supervisor, Staffan Björk, who helped us with his prior knowledge and his great sense of humor throughout the project. Finally, we would like to thank Chalmers Writing Centre for helping us with questions regarding technical writing.

Contents

List of Figures

List of Tables

1
Introduction

In conjunction with increasing performance, memory and storage available on modern computers, the capacity for large amounts of varied content in games, movies, and other digital media grows respectively. However, the capability to manually produce interesting content that can fully utilize this potential remains a slow and expensive process. Consequently, many companies have decided to leverage the use of algorithms to partly, and sometimes completely, automate the production of such digital content. These algorithms are part of a broad research field within computer science called Procedural Content Generation (PCG) [1].

This thesis explores the usage of PCG algorithms in order to find a suitable approach for procedural generation of modern 3D cities. The investigation is conducted through the development of a user-friendly desktop application that can interactively generate and export models of 3D cities. These models can then be used in third-party modeling software and game engines. Thus, with this report we contribute with insight into various techniques applicable to city generation, as well as an open-source, MIT licensed application called *CityCraft* that demonstrates such generation.

This chapter introduces the project, its goals, and some related works that have been previously done in the field of city generation. Thereafter, follows a background chapter about the applications of PCG, and a theory chapter covering many of the fundamental techniques considered. Finally, the project's methods and results are presented, followed up by a discussion chapter and some notes for future work.

1.1 Purpose

Cities are complex and massive architectural infrastructures that appear all over the world, and a significant portion of the world population lives inside them. They are also a frequent scenery in digital media, such as advertisements, video games, and film. Unfortunately, the creation of digital cities can be a highly time-consuming effort if done by hand. Ideally, this task could be automated such that studios of any size could produce impressive cities by only specifying some rough design parameters. The purpose of this project is to contribute towards such an ideal.

Formally, the purpose of this project is to research and combine various PCG techniques in order to discover and demonstrate a suitable approach for generating modern 3D cities. The intention is to demonstrate this approach by implementing a standalone city generation software, whose generated cities could be refined by artists and then integrated into medias such as video games and film. This software should be free, open-source, and accessible to anyone in the industry of digital media.

The significance of such software compares to SpeedTree [2], which revolutionized the process of creating realistic forests. Of course, such an ambition is beyond realistic for the scope of this project, and that is why constraints have to be made.

1.2 Goal and Scope

The goal of this project is to develop a user-friendly application that can generate modern 3D cities in real-time and export them in a standard 3D file format such as *.fbx*. *User-friendly* denotes that users should not need any technical expertise or coding experience to fully utilize the application. Furthermore, *modern cities* will be defined as cities that resemble present industrialized cities such as New York, Paris, San Francisco, and Tokyo (see Figure 1.1). The intention is not to perfectly replicate these cities, but rather to draw inspiration from them.

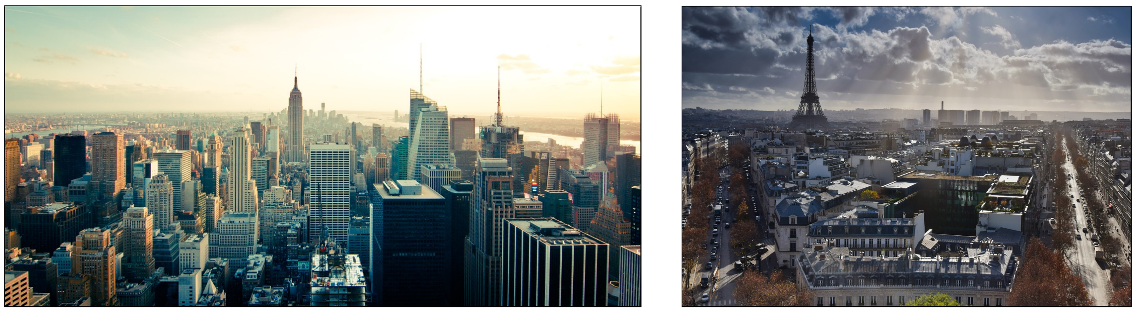

(a) Manhattan, New York [3].   (b) Paris [4].

Figure 1.1: Two examples of modern cities considered in this project.

To clarify, suburban and rural areas are also included as part of such modern cities.

The process of generating cities should be effortless both in the sense that minimal configuration should be required, and that generation of multiple cities should be feasible within a minute from application start when running on a modern laptop. With that said, the visual quality of models is not of immediate concern for this work. The intention is to demonstrate a proof of concept of how PCG algorithms can be used to leverage the effort required to build cities, not to produce production-ready software. Consequently, the use of free textures and models is encouraged in order to prioritize the development of algorithms. Additionally, any specific art style such as low poly [5] and voxel graphics [6] is not pursued.

The generation of roads and buildings will be of primary focus, as these are considered the core features of a city. Surrounding terrain also needs to be generated, mainly to form natural settings which the city infrastructures need to adjust after. Accordingly, the terrain itself does not need any significant level of detail. Other metropolis aspects such as sidewalks, parks and parking lots are also intended to be generated, albeit with less variation.

In this contribution, generated cities will be restricted to static models in order to support integration with a wider array of third-party software. Therefore, dynamic content such as simulation of road traffic, day-night cycles, and pedestrians are all considered out of scope. The interior of buildings is also considered out of scope since end-users are expected to desire more control over such content than what this project's time frame can support.

As a consequence of the random and visual aspects of PCG, it is difficult to express a precise definition that can measure whether this project's final results should be considered successful or not. Nevertheless, the following list has been constructed as a modest checklist to capture and discuss the quality of the resulting application.

Q1: Do models correctly integrate with third-party software such as Blender [7]?

Q2: How well is the codebase structured for replacement and expansion of features?

Q3: How much notable variety is there in the generated content?

Q4: To what degree do the generated cities resemble real-world cities?

Q5: How much control do the users have over the generation?

Q6: Are the cities suitable for use in digital media such as games and film?

Q7: Are users without technical expertise able to correctly use the application?

Q8: Are there any known bugs or crashes in the application, or any visual artifacts in the generated models?

1.3 Social and Ethical Aspects

When conducting research it is vital to recognize the social and ethical implications that it might have. This subchapter covers the major such aspects that were considered throughout this research.

The first aspect considers automation, which is a type of double-edged sword. On one hand, the development of a city generator could lead to decreased production time and cost for game and film studios alike. It might particularly help small production studios since they would be able to meet user expectations with less effort, and therefore better compete with the productions of larger studios. However, this level of automation also has the potential to cause unemployment, especially in major productions where the design of large scenery might involve hundreds of employees.

Unemployment due to automation is a report on its own, but in the case of PCG, it might lead to a shift from designers towards programmers rather than job loss. With that said, these two professions typically do not overlap in skillset. According to game developer Scott Beca, attempting to replace level designers with PCG algorithms is “a dangerous road to go down” [8]. He reckons that, although significant workload traditionally done by designers might transfer to programmers, designers will still be in demand to polish and configure the building blocks used by the PCG algorithms. Our belief is that such polish applies to the whole generation process, and therefore hope to empower designers by making the application user-friendly.

The second major aspect considers the representation of real-world cities. If the generated cities fail to capture the various facets that compose real cities, then the generated models might pose a misleading, falsified or biased distortion of reality. For instance, transportation facilities, history, culture, religion, poverty, politics, authorities and social injustice are all dimensions that can shape the appearance of a city. As an example, if the generated cities do not include public transportation, does that imply that the application encourages transportation by other means?

Ultimately, a scope will have to be made, and justifying the breadth or depth of real cities within this scope will not be feasible. Our intention is thus to only include a humble subset of the real-world dimensions, and instead encourage the contribution of further improvements by releasing the application as open-source.

To summarize, we conclude that there exist certain ethical aspects involved with the contribution of this report. However, measures have been taken to reduce the significance of these potential risks, and with that, the proceedings involved in this research have been considered ethical.

1.4 Related Works

Generation of cities has been an active area of research for almost 20 years by now, with the *Procedural Modeling of Cities* paper by Müller et al. being the pioneer [9]. This paper outlines the *CityEngine* system, which makes heavy use of L-systems (explained in Chapter 3.2) to generate Manhattan-style cities from various image maps as input. The system produces impressive results, and the paper itself proposes many interesting ideas that extend beyond L-systems. *CityEngine* later evolved into a commercial application [10], demonstrating the real-life potential of this type of software. Unfortunately, *CityEngine* is quite expensive and remains closed-source. Nonetheless, the original paper has been a significant inspiration for our work.

Kelly and McCabe have also made significant contributions to this field with the development of *Citygen* [11], and the conduction of a prestudy [12] summarizing common city generation techniques. *Citygen* is an interactive system where users model cities in real-time, being heavily assisted and accelerated with the use of PCG techniques. The paper emphasizes road generation and lot subdivision, detailing several intriguing concepts such as road graph hierarchy, snap algorithms for road segments, and strategies for building roads between significant elevation differences.

A tendency found amongst early work on road generation is that L-systems have frequently been used. Another common approach has been agent-based generation [13], especially in more recent work [14] [15]. Recently, there have also been some experimentation done using the WaveFunctionCollapse (WFC) algorithm [16] to generate roads [17]. However, it remains difficult to determine WFC's potential compared to previous approaches as more research needs to be done [17, p.50].

Outside of academia, there has also been promising work done related to city generation. A decade-old tech demo from *Introversion Software* shows a promising generation of the outline of a city, complete with a graphical interface [18]. The *SceneCity* plugin for Blender [7] is able to produce highly detailed buildings connected by grid-based roads [19]. Impressive PCG cities can also be found in modern games such as *Marvel's Spider-Man* [20] and *The Sinking City* [21].

Unfortunately, most progress towards city generation so far remains closed-source and behind paywalls. Exceptions exist but tend to be deprecated or incomplete solutions. Our contribution aims to deviate from this by being open-source, free, and able to generate cities complete with terrain, roads, and buildings. With this, we believe that the progress of city generation will be more accessible to industry and academia alike. Accordingly, ideas presented in previous work will be combined in various ways to produce a complete, standalone city generation software.

2

Background of PCG Applications

PCG is a family of algorithms that automatically (and often randomly) generates synthetic media such as 3D models, textures, items, quests, levels and music [1, p.1]. Typically, this generation is achieved by combining manually designed assets with stochastic, yet controllable algorithms. The algorithms are restricted by domain-specific rules to only produce desired content, and utilize randomness within these constraints to generate diverse results [8]. For instance, a correct PCG algorithm for car generation always produces functional cars, but the resulting cars will still vary by configurable properties such as wheels, seats, size, engine, price, and name.

In the early 1980s, PCG was primarily used in games as a workaround for the limited storage space in computers [1, p.4]. This workaround has lead to a whole new genre of games called *roguelike* that is still thriving today [22]. However, in recent years PCG has gained traction for other reasons than small file sizes.

Populating large games with content is a labor-intensive process if done by hand, and PCG has been used to tackle this problem by speeding up development times and by producing new experiences for each playthrough. For instance, the game *Minecraft* [23] uses PCG to generate all of its procedural worlds (see Figure 2.1).

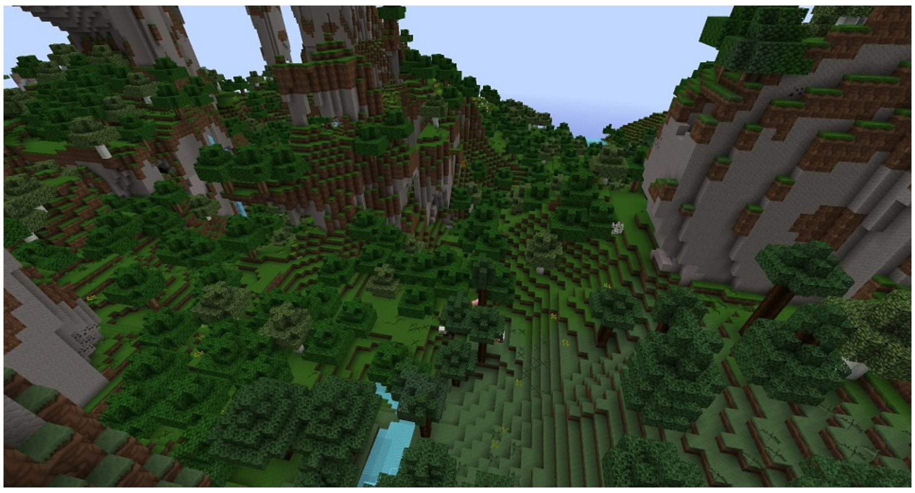


Figure 2.1: Example of a pseudo-infinite world generated in Minecraft [24].

PCG can be performed either before or during runtime, in which case the generation is referred to as offline or online respectively [1, p. 7-8]. Since online generation is performed while a user is interacting with the media, it is critical that any computations are light enough for the software to run smoothly and that all results produced are acceptable to the end-user. Meanwhile, offline generation has typically significantly looser requirements since it is performed during the development process. Online generation, however, has the crucial advantage of being able to dynamically adjust its generated content based on user interactions.

Offline generation is arguably the most common type of generation due to its broad definition and loose restrictions. It is commonly used to accelerate the work of artists by generating good candidates which artists can then manually refine. Examples of such usage include the creation of landscape in *Oblivion* [25]. This approach is also the typical way software like SpeedTree [2] is used, and is the intended way to use the city generation software presented in this report.

Online generation is also highly useful, but primarily in interactive media like video games. Some of its commercial applications include the

- pseudo-infinite worlds in *Minecraft* [23] and *No Man's Sky* [26].
- unpredictable level layouts in the *Civilization* [27] and *Diablo* [28] series.
- procedural weapons in *Dead Cells* [29], *Terraria* [30], and *Borderlands* [31].
- procedural creatures in *Spore* [32] [33] and *No Man's Sky* [26].
- dynamic difficulty found in *Left 4 Dead 2* [34].

There is of course also *Dwarf Fortress* [35] which generates everything imaginable, from world history to character personalities. All of these games benefit from improved replayability and unpredictability with online generation, albeit by different means and to various degrees. The prime examples being *Minecraft* and *Terraria* who both extensively use PCG throughout their game mechanics, and who both have received outstanding critical and commercial success [36] [37] [38] [39]. Although PCG itself is not a recipe for success, as seen with the launch of *No Man's Sky* [40], but even that game made a comeback [41].

Nowadays, the terms *PCG* and *procedural generation* are heavily tied to games, with some even describing PCG as “the algorithmic creation of game content” [1, p.1]. Although games have seen several benefits from PCG, such as improved variation and replayability, it is worth noting that the field has seen extensive use in other mediums as well. One such medium being film. For example, the software package MASSIVE was used throughout the *Lord of the Rings* trilogy to generate crowd-related visual effects [42], and generation of vegetation using SpeedTree has become an industry standard [43]. Nonetheless, most PCG research seems to target games.

3

Theory of PCG Techniques

This chapter introduces the PCG algorithms and concepts considered throughout the report and ends with an introduction to relevant fundamentals of 3D graphics. The PCG-related subchapters are roughly ordered by how frequently their concepts occur in PCG research, starting with the most common ones.

3.1 Noise

Noise is a controlled form of randomness and one of the most fundamental concepts of PCG. There exist several flavors of it, with the two major ones being *value noise* and *gradient noise*, which can both be expressed using the following function:

$$f(\vec{x}) = y \in [0,1] \text{ where } y \in \mathbb{R}, \vec{x} \in \mathbb{R}^n, n \in \mathbb{Z}^+ \tag{3.1}$$

This function is deterministic, but the output for any single input is perceived as random. This is achieved by combining a Random Number Generator (RNG) with an interpolation function. The way that these two functions are implemented and utilized is what differs value noise from gradient noise.

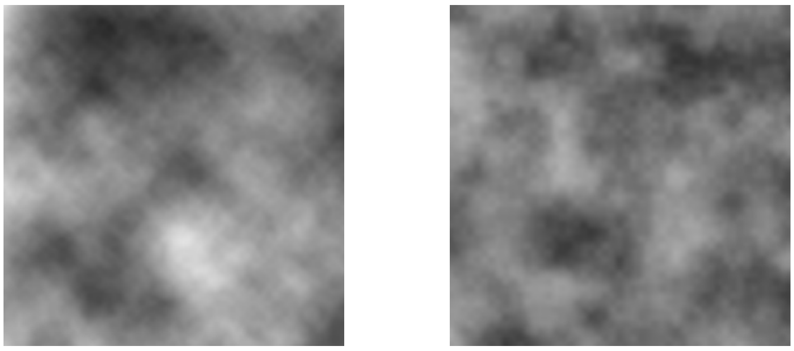
Value noise is constructed by first randomizing the values of all lattice points in an n-dimensional space using an RNG function. These points are then interpolated to define intermediate points, resulting in a continuous n-dimensional space of random values. The noise function then simply outputs the real number defined at coordinate $\vec{x}$ within this space. The interpolation function used is typically bilinear or bicubic in the setting of real-time computer graphics, as they are computationally cheap.

Although simple to implement, value noise has several disadvantages. First off, the resulting noise has a visually grid-like structure. This is often undesired in graphics since it creates arbitrary patterns in the noise. Moreover, neighboring lattice points sometimes greatly differ in values which can result in sudden changes in intensity.

Gradient noise solves the major problems present in value noise by randomizing n-dimensional gradient vectors at each lattice point, instead of real numbers. This modification ensures that the interpolated values have smooth continuous gradients (i.e. partial derivatives) and not just smoothly interpolated values.

Some common implementations of gradient noise include *Perlin noise* [44] [45], *Simplex noise* [46], and *OpenSimplex noise* [47] [48]. Ken Perlin originally invented Perlin noise with the intention to produce more natural-looking textures [44], and later invented Simplex noise to address some of the limitations with Perlin noise [46]. Usage of Simplex noise is however restricted due to its patent [49]. Fortunately, Kurt Spencer invented a well-performing variation called *OpenSimplex noise* which was released into the public domain [47] [48]. The details of these algorithms are beyond the scope of this paper, but further reading of Perlin noise can be found in the original paper [44], and the ideas behind Simplex noise are described in great detail by Stefan Gustavson in his paper *Simplex noise demystified* [50].

In practice, it is common to use a Fractional Brownian Motion (fBm) implementation, which can be constructed by summing noise functions of different amplitudes and sampling frequencies. The variation in frequency and the variation in amplitude between these functions are sometimes referred to as lacunarity and gain respectively. The usage of fBm allows for greater control over the resulting noise function's visual properties, making it more suitable for many applications in computer graphics.



(a) Value noise [51]. (b) Gradient noise [52].

Figure 3.1: Examples of value noise and gradient noise represented using intensity maps. Notice how the value noise has cross-like patterns.

The output of noise functions can be visualized in many ways, intensity maps being one of them (see Figure 3.1). An intensity map is a bitmap image where each pixel represents the value of the underlying noise. Bright pixels represent high values, while dark pixels represent low values. These maps can be used to represent various things such as heights, in which case they are referred to as *heightmaps*. They can also be used to blend textures together, which is called *texture splatting*.

3.2 L-Systems

Another fundamental concept in PCG is L-systems. An L-system is a type of formal grammar where all symbols are evaluated in parallel. It is built up of strings which under a set of constraints recursively grow larger and more complex with each iteration. These strings can then be used to generate complex geometric structures, often with fractal properties [53]. The technique's inventor, Aristid Lindenmayer originally used it to model a wide array of plants [54]. L-systems are formally defined by three parameters which can be denoted as follows:

- V - The alphabet, consists of replaceable symbols (variables or non-terminals) and static symbols (constants or terminals).
- ω - The axiom, an initial variable from V which defines the initial state.
- P - The production rules, these are the constraints which determine how variables should be replaced.

For instance, $G(V, \omega, P) = (\{A, B\}, A, \{(A \rightarrow AB), (B \rightarrow A)\})$ would produce

$$\begin{aligned} n &= 0 : A \\ n &= 1 : AB \\ n &= 2 : ABA \\ n &= 3 : ABAAB \\ n &= 4 : ABAABABA \\ &\dots \end{aligned}$$

for the first four evaluation iterations. The symbols of the resulting string can then be interpreted as different actions. The above example could be used to build a corridor where A represents empty space and B represents a door.

There is a special case of L-systems which is called *Stochastic bracketed L-systems*. Stochastic means that all production rules have a probability of being used for an evaluation, which implies that multiple rules can target the same source string. Bracketed means that a stack structure is used where the '[' symbol pushes the current state, and ']' pops and restores the previously pushed state.

By being able to return to a previous state, one can produce content with non-continuous generation. This is well expressed in the book *Procedural Content Generation in Games* [1, p.77], where the authors describe the limitations of non-bracketed L-systems in the following way:

"While interpreting L-system-generated strings as turtle instructions allow us to draw complex fractal shapes, we are fundamentally limited by the constraint that the figures must be drawable in one continuous line—the whole shape must be drawn 'without lifting the pencil'".

The ideas from L-systems can also be extended beyond strings, with some examples being Shape grammars [55] and Graph grammars [56]. Graph grammars use graphs from discrete mathematics instead of letters. This difference enables graph grammars to produce non-sequential graph results, as opposed to 1-dimensional strings, making them suitable for generating quests and level layout. Unfortunately, this feature also makes them more difficult to implement and more expensive to process than L-systems. Shape grammars are similar, but operate on geometrical objects instead, making them suitable for Computer-aided Architectural Design (CAAD).

3.3 Search-based PCG

For some generation purposes it can be challenging to precisely describe desired results, but easier to describe desired properties and to compare alternative results. A suitable technique for handling these situations is Search-based PCG (SBPCG).

SBPCG is a collection of stochastic search algorithms that are combined with domain-specific evaluation functions to iteratively optimize some content [57] [58]. Traditionally such search has primarily been performed using Evolutionary Algorithms (EA) such as the $\mu + \lambda$ evolution strategy (ES) [1, p.18-20], and NSGA-II [59].

SBPCG via EAs is typically done in the following way.

1. Manually create or randomize a population of initial content candidates.
2. Evaluate all candidates with evaluation functions.
3. Maintain only the best performing candidates and discard the rest.
4. Search for new candidates similar to the current population. This step typically involves several mutations and crossover operations.
5. Repeat from step 2, unless current candidates are satisfactory.

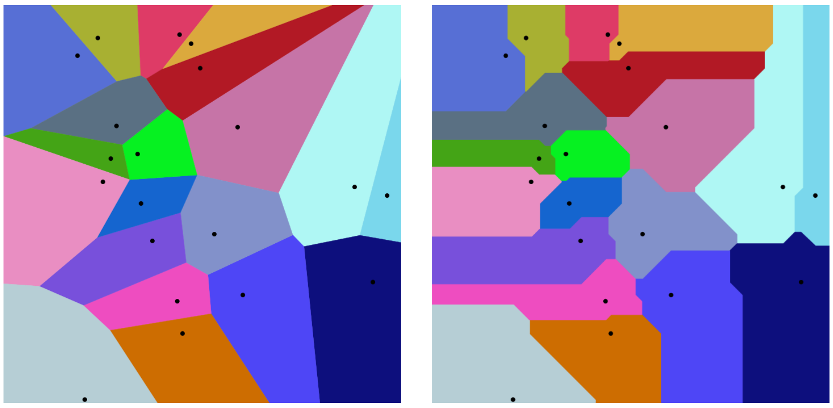
The core idea of this process is based on natural selection. There is a population of individuals (candidates) in each generation that are evaluated, and the fittest (highest evaluated) individuals get the chance to reproduce, while those least fit are removed. The evaluation of individuals may depend on multiple properties and in PCG it is often desired to maintain diversity in the population as well. Diversity is important to avoid local maximums, but also to ensure results have enough variation.

Machine Learning (ML) is another method that can be used instead of EAs, in which case the generation is often referred to as Procedural Content Generation via Machine Learning (PCGML) [60] [61] [62]. In PCGML the idea is to generate new content based on patterns found from analyzing existing content by using Artificial Neural Networks (ANN). This technique is especially effective when large amounts of quality content already exist. For instance, PCGML could be used to extract patterns from existing bridges in order to combine these patterns to form new bridges. The main drawback is that such well-structured data of quality content can be difficult to obtain.

There are three main categories to consider when deciding on what evaluation functions to use, and these categories are *direct*, *simulation-based*, and *interactive* [57, p.5-7]. In direct evaluation, content quality is determined directly by its properties such as size and weight. In simulation-based evaluation, the interaction between the content and some artificial agent is judged instead. Finally, in interactive evaluation a human is employed to interact with the generated content. The human then either explicitly answers what content they preferred, or such feedback is extracted implicitly from gameplay data.

3.4 Voronoi Diagrams

Voronoi diagrams are a set of techniques used to partition a plane into n regions (called Voronoi cells). This is done by first scattering n nodes, referred to as *seeds*, onto the plane. Subsequently, each seed is assigned its own region and all points on the plane will be assigned to the region of the closest seed. These regions can then be used in PCG-scenarios such as dividing land areas into regions with less obvious patterns than those formed by using a simple grid. Another use case is to reduce memory consumption through seamless chunk loading, not unlike *Minecraft* [23].


(a) Voronoi diagram using Euclidean distance [63].

(b) Voronoi diagram using Manhattan distance [64].

Figure 3.2: Voronoi diagrams using identical seeds but different distance metrics.

Voronoi diagrams can be visualized by marking seeds with black dots and by coloring each regions' points with a unique color (see examples in Figure 3.2). The construction of these diagrams can be implemented using either Lloyd's Algorithm [65], or Fortune's Algorithm [66].

3.5 Poisson Disc Sampling

Poisson Disc Sampling randomizes the placement of points in an N-dimensional space such that all points form a single cluster, while maintaining a minimum distance between points. This algorithm can be implemented in linear time [67], and can be used for randomly placing clusters of trees and shrubs.

The idea behind the algorithm is to specify a minimum distance R, a sample limit k, and then place an initial *active point* in a random position. Thereafter, the following procedure is repeated until there are no more active points remaining.

1. Sample an active point, P.
2. Place up to k new points inside the $[R, 2R]$ annulus centered at P.
3. For each new point,
   (a) Remove it, if it is within R radius of another point.
   (b) Otherwise, add it to the list of active points.
4. Mark P as an *inactive point.*

A depth-first variant of the above procedure has been visualized by Mike Bostock, and its JavaScript implementation has been made open-source [68].

Occasionally it is desired to place objects of various sizes together. For instance, one might want to place trees, bushes, and grass together in a forest. Blomqvist et al. showed that such placement could efficiently be done by separating objects with different R values into layers, and then generate each layer in decreasing order of R [69, p.32]. By applying this hierarchy, one avoids the problem of packing small objects so tightly that no room is left for larger objects. Instead, large objects such as trees get priority, and then smaller objects such as grass are generated in-between.

3.6 Basics of 3D Graphics

Everything in modern computer graphics is centered around polygon meshes. A polygon mesh is a geometric graph combined with a set of geometrical faces that together form a surface. Typically these faces are triangles, in which case the meshes are also referred to as triangle meshes, or simply meshes. An example of such a mesh is shown in Figure 3.3.

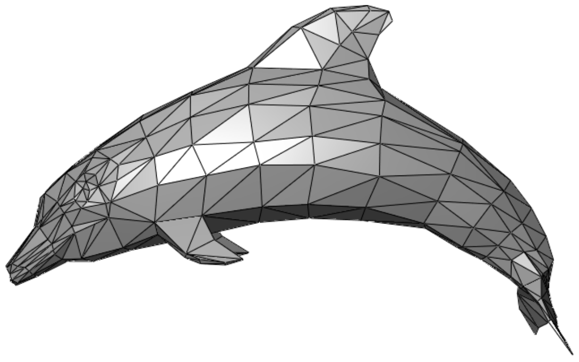
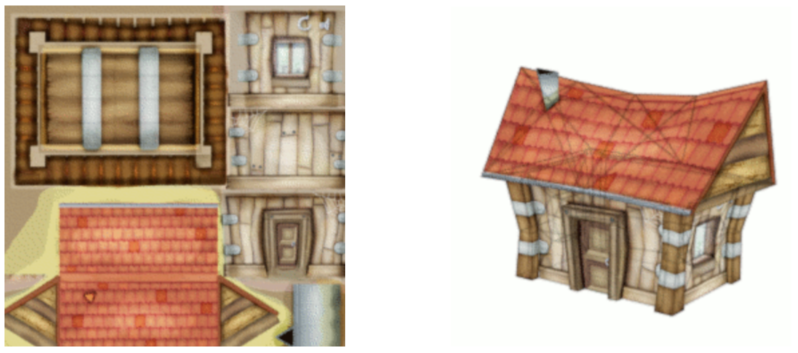


Figure 3.3: Example of a low poly triangle mesh representing a dolphin [70].

Each vertex can store arbitrary data alongside its own coordinates in space. Examples of such data include: an averaged normal of its neighboring faces, color, reflectance, bone weights and texture coordinates (a.k.a. UV coordinates). These stored properties can then be used to help compute realistic lighting, colors, animations, and more.

Nowadays it is common to apply high-quality images to meshes, rather than just colors and lighting shades. Applying these images can be done via a process called *texture mapping* or *UV mapping*, which entails designing 2D textures and mapping these onto 3D meshes using the UV coordinates of the vertices. An example of this process can be found in Figure 3.4.



(a) House texture map.

(b) Textured house mesh.

Figure 3.4: Example of UV mapping a texture map onto a model of a house [71]. Notice how UV coordinates are reused to apply the same texture to each pillar.

Models can be future enhanced by using materials and shaders. A material contains all the visual properties of a model except for the mesh itself. This may include multiple constants, textures, and shaders. Shaders are programs designed to run on GPUs and can be used to combine textures in various ways to create interesting visual effects. The downside is that shaders are typically strongly tied to specific software or hardware, making them difficult to port. Consequently, standard file formats for 3D models such as *.obj*, *.fbx*, and *.gltf* have limited support for shaders.

4 Methods

This chapter presents the methods, both in terms of algorithms and processes, that were used throughout the project to develop a city generation software. It also clarifies why these methods were chosen and what other options were considered. The first subchapter covers the graphical application and how user interaction was handled. The second subchapter details how an architecture was built to generate cities in the application using both PCG and 3D graphics techniques. The final subchapter presents how this work was executed in collaboration within the group.

4.1 Application

A user interface would be necessary for users to be able to control and see any generation. For this purpose, a GUI (Graphical User Interface) was chosen as user-friendliness was a key requirement for the application. But before the application was designed, it was deemed necessary to first formalize how the users would interact with it by writing down a user flow. This user flow went as follows.

1. The user starts the application and is greeted with an endless ocean and a few options for terrain generation.
2. The user clicks on a button that says “Generate Terrain”, which will generate a landmass in the ocean. The user can re-generate a new terrain as many times as they like. They may also modify any optional parameters that affect the generation, but good defaults are provided.
3. The user places markers on the terrain which represent cities and further specifies how much population each marker roughly represents. The user can also specify optional parameters for each marker e.g. Manhattan-style roads.
4. The user clicks on a button that says “Generate Roads”, which will generate a rough outline of the road network. This may be generated multiple times.

5. The user clicks on a button that says “Generate Streets”, which will complete the road network by creating all the streets that go between the main roads. These may also be generated multiple times.
6. The user clicks on a button that says “Generate Buildings” as many times as they like until they are satisfied with the generated buildings.
7. Finally, when the user is satisfied they can click on a button that says “Export” and the full model will be exported as a file on the local file system.

At any time during the flow, the user would be able to control the camera that visualizes the world in order to zoom in and see the model from different perspectives. It would also be possible to deviate from the flow by going backward and undoing previous generation steps.

The following subchapters present how such a GUI was implemented, and how the problem of exporting the produced cities as 3D model files was resolved.

4.1.1 Framework

Both GUI and 3D graphics would inevitably be an aspect of this research but they were not seen as the core parts. Therefore, a 3D library, framework, or engine was needed to leverage the workload on this front. LWJGL [72], JMonkeyEngine [73], Unity [74], Unreal Engine [75], and Godot [76] were all considered in this project. Unreal Engine was excluded because of its steep learning curve, and the way it favors visual programming using something called *blueprints*. Godot was excluded for its limitations in 3D and its young and small community. LWJGL and JMonkeyEngine were omitted for the amount of work that would have been required to implement simple features like visual debugging and live recompilation.

The choice ended up being the Unity 3D engine because of its large community, extensive range of tutorials, concise documentation, stability, cross-platform support, and wide array of first- and third-party tools. Moreover, a few group members already had previous experience with this engine, so this choice also included fewer uncertainties than the other options did.

At the time, Unity officially only supported C#, so the choice of programming language became straightforward. This aspect was taken into consideration when choosing the engine, and did not occur as a concern since all project members had previous experience with the Java programming language, which is similar to C#.

Thus, the application would be designed and debugged from the Unity editor, but it also had to be able to run outside of the editor as a standalone program. Consequently, the application could not depend on functionality provided in Unity’s editor to work. This restriction is normally not a problem, but it did complicate the serialization of runtime objects into standard 3D file formats, which was needed to export the city models.

4.1.2 Exportation of Cities as 3D Models

The cities had to be exported into some widely used format, and this had to be done through a library for the process to be stable and to save time. For this task, the UnityGLTF [77] library was used to export models into the glTF (*.gltf* and *.glb*) [78] format. This library was primarily chosen as it could run without the Unity editor, but it had several other attractive properties besides that.

Firstly, both UnityGLTF and glTF should be stable options as they are maintained by the Khronos Group [77] [78], which is an open industry consortium responsible for open-source 3D graphics standards such as Vulkan, OpenGL, and WebGL [79]. Secondly, many 3D file formats are available, but glTF seems to be moving towards industry-standard judging by its extensive ecosystem and industry support [78]. The exported cities should thus integrate well with a wide array of industry software if glTF is used. Lastly, the UnityGLTF library itself was easy to set up and configure.

Other 3D file formats were also considered but none of them had libraries that satisfied the requirements of this project. Unity's FBX Exporter [80] could be used to produce *.fbx* [81] files, which is a proprietary but powerful format. This library was especially attractive due to its ease of use and official support from Unity and Autodesk [80]. Unfortunately, the FBX Exporter heavily depended on the Unity editor. Wavefront's *.obj* [82] and *.mtl* [83] files were also considered, but the libraries found either depended on the Unity editor or were deprecated. Similar issues were also found with the COLLADA (*.dae*) [84] format. Hence, glTF using UnityGLTF was the only reasonable option found.

The correctness of the exported cities had to be verified somehow, and for this purpose the open-source 3D modeling software Blender [7] was used. The idea being that, if the models worked well in Blender, then in the worst-case scenario the cities could at least be re-exported from Blender to other tools. Ideally, one would have constructed an automated pipeline and verified against multiple third-party software, but this did not appear necessary for this project.

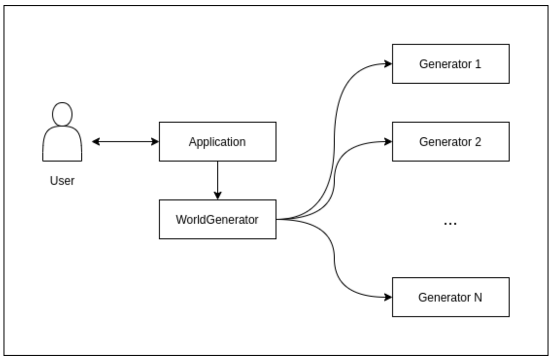
4.2 City Generation Architecture

To approach the problem of 3D city generation it was necessary to first break it down into manageable modules that could be worked on in parallel. One of these modules had to be responsible for the GUI presented in the previous subchapter.

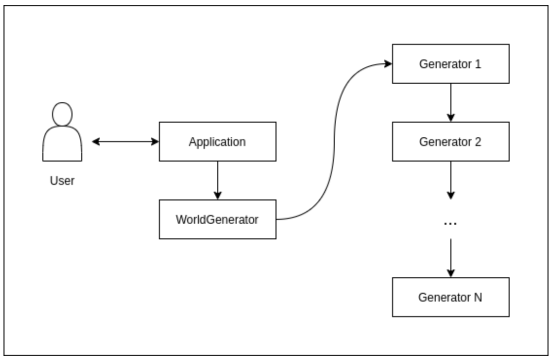
This module was named *Application* and could be kept relatively small as the Unity engine would handle most logic. There also needed to exist some module responsible for the PCG, since that is the core of this research. This module was named the *WorldGenerator*. WorldGenerator was then further split into eight submodules that would each be responsible for a distinct part of the model generation.

The implementation of the user interface would be rather simple and was thus treated as a single module, while the generation needed some architecture to handle the communication between submodules. The two main architectures considered for the generation were a function-based and a pipeline-based approach (see Figure 4.1).

The pipeline had the advantage of less overhead since each generator would directly (or through an interface) pass data onto the next. However, this lack of overhead could impose limitations and there still needed to exist some module responsible for receiving and exporting all the generated model data. Furthermore, when Blomqvist et al. developed a PCG engine, they experienced issues with their pipeline architecture, such as interconnected modules and bias of workload towards the early pipeline stages [69, p. 45]. Thus, with these concerns in mind, it seemed reasonable to settle for the function-based architecture instead.



(a) Function-based architecture. Each sub-generator is treated as an isolated function and WorldGenerator controls how and when they are invoked.



(b) Pipeline-based architecture. The first generator is invoked by WorldGenerator, and then each sub-generator passes on its output to the next sub-generator.

Figure 4.1: Two different architecture approaches considered for the generation logic.

The function-based architecture showed several promising advantages. For one, each generator would only receive input that it actually depended on, while with a pipeline each generator would have had to pass along all its data. This separation not only makes debugging easier, but it also improves performance slightly. Another recognized advantage was that generators could now be run asynchronously and in separate threads, further benefiting performance. Lastly, as generation steps would be invoked from GUI buttons, possibly in non-linear order and with *undo* operations, it was better suited if WorldGenerator could communicate with each sub-generator directly. Consequently, the function-based architecture was chosen.

Input		Function		Output
Size, Offset, SeaLevel	→	**TerrainGenerator**	→	*Terrain*
Terrain, PopulationAmplifier[]	→	**PopulationGenerator**	→	*PopulationMap*
Terrain, PopulationMap, Markers	→	**RoadGenerator**	→	*RoadNetwork*
RoadNetwork, PopulationMap	→	**BlockGenerator**	→	*Block[]*
Block, PopulationMap	→	**PlotGenerator**	→	*Plot[]*
Plot, Terrain, Population	→	**BuildingGenerator**	→	*Building*
Plot, Terrain	→	**ParkGenerator**	→	*Park*
Plot, Terrain	→	**ParkingGenerator**	→	*ParkingLot*

Table 4.1: The proposed generator functions needed to generate the 3D cities. The invocation of these functions are handled by WorldGenerator. '*[]*' denotes plural (e.g. list or array) of the preceding data structure.

An overview of the eight generators that were decided upon is shown in Table 4.1, while the implementations and responsibilities of these generators will be detailed in the following subchapters.

4.2.1 Terrain Generation

Generation of the underlying terrain is managed by the TerrainGenerator function (see Table 4.2). In this implementation, the terrain is represented as a colored and textured surface mesh with evenly spaced vertices, whose heights (y-position) have been altered to form hills and valleys. The vertices of the mesh are given different height values based on a layered Simplex noise, whose implementation was extracted into a separate module as it would come to use in Population Generation as well.

Input		Function		Output
Size, Offset, SeaLevel	→	**TerrainGenerator**	→	*Terrain*

Table 4.2: Definition of the TerrainGenerator function which is responsible for generating the terrain.

The input of this function is a set of scalars, specified by the user at runtime. *Size* determines the width (x-axis) and depth (z-axis) of the terrain, *Offset* affects the sampling location of the noise, and *SeaLevel* configures the y-position of the surrounding sea, which is represented as a flat, textured plane. The intention behind *Offset* was to give the user finer control over the noise (and thus the heights) of the terrain.

The noise function used for the terrain had four manually designed noise layers, which intended to form a balanced variation of lakes, plains, and mountains. The resulting mesh would then be textured and colored in order to make it look more convincing (see Figure 4.2). Colors were applied based on height levels, forming a blend between snowy mountain tops and verdant plains. To texture the mesh, a single texture was used with randomized UV coordinates, which was included to reduce Moiré patterns [85]. A texture atlas could have been used to support more textures, but the visuals from this approach were satisfactory enough.

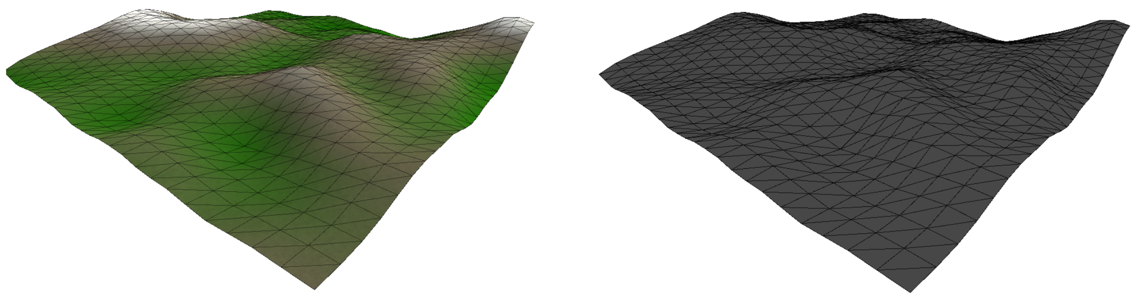


Figure 4.2: Example of a surface mesh with (left) and without (right) textures.

There were two options considered regarding the construction of the terrain. The first option was to make use of Unity's built-in terrain API, and the second was to construct a custom mesh, manipulating vertex data manually. The former option showed greater potential as it provided several advanced features such as texture splatting and dynamic resolution.

Unfortunately, these features were only applicable in runtime and could not be serialized into model files in any simple way. Furthermore, the produced terrains were custom objects specific to Unity, and would have to be converted into meshes before exportation. Thus, the second option was chosen as it would provide a simpler implementation.

4.2.2 Population Generation

The PopulationGenerator function produces an intensity map representing the population density across the terrain (see Table 4.3). The terrain parameter is required since certain areas of the terrain need to be masked off from population, such as oceans, rivers, and spiky mountain tops.

Input		Function		Output
Terrain, PopulationAmplifier[]	→	**PopulationGenerator**	→	*PopulationMap*

Table 4.3: Definition of the PopulationGenerator function which is responsible for generating an intensity map of population across the terrain.

The other input parameter is a set of amplifiers. Each amplifier is a 2D geometry visually represented with markers that give covered areas drastically increased population.

Through these amplifiers, the user may designate locations of city centers, and thus interactively modify the population across the terrain. An example of this usage can be found in Figure 4.3.

To generate the initial population distribution, the generator was implemented with a few layers of simplex noise. The population amplifiers from the function input are applied after this initial population map has been generated. The noise logic was eventually extracted into a separate module, which was then shared with the terrain generator, although with different manually designed layers.

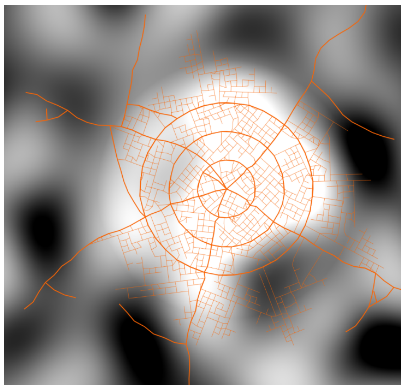


Figure 4.3: An example of a population map with a Paris city generated within it.

4.2.3 Road Generation

The fundamental infrastructure of a city is the road network and, as such, it was important that the road generation would be both flexible and offer realistic results. To generate such networks, the RoadGenerator function was designed (see Table 4.4).

Input		Function		Output
Terrain, PopulationMap, Markers	$\rightarrow$	**RoadGenerator**	$\rightarrow$	*RoadNetwork*

Table 4.4: Definition of the RoadGenerator which is responsible for generating road networks.

This function uses the terrain to place down roads, the population map to determine where roads are needed, and input markers to form city center patterns.

Three approaches were considered when designing the road generator. The first was based on a recursive approach where the world was divided into cells with roads placed between the cells, forming a similar structure to Voronoi Diagrams. However, this type of algorithm did not provide realistic-looking results. Regardless of being flexible, it was overly complicated for the purpose of creating realistic road networks. The second approach was search-based and would involve ML to produce structures based on data. However, proper data appeared difficult to gather and evaluation of candidate solutions would become highly challenging.

The final approach that was decided to be used in the project involved an Agent-based generation which simulated road workers traversing the terrain. The sole purpose of these Agents was to create roads based on different road building strategies. Agents could also decide to branch into multiple new Agents, forming intersections in the road network. Which strategy to use depends entirely on the goal of the generation, but the focus of this project has been to create a flexible base that could be extended to mimic any type of city generation. As such, this project only includes Paris and Manhattan strategies, with the generation of main roads depending on the city strategy in order to achieve the aesthetics of the specific city type. Paris-like cities have distinct rings within the city boundaries, while Manhattan-like cities follow a much more grid-like structure.

Furthermore, strategies may instruct the Agent to switch strategy mid-generation, be it randomly, or depending on some variables that the strategy has access to. For example, if a strategy detected a relatively low population density, it might instruct the Agent to switch to a village-type strategy that could produce smaller villages with a different layout than cities like Paris or Manhattan.

Strategies can also define the configuration of the Agent, such as step size, how many steps an Agent can take before terminating (step count), and how many times an Agent can branch. The strategy that the Agent uses is responsible for deciding when an Agent should terminate, except for step count which is handled automatically for all strategies.

The road generator uses the city markers as input to create the initial Agents that will start creating the road network. Each city type needs its own preset arguments for the Agents, and this was solved by assigning an Agent factory to each city type. The goal of the Agent factory is determining starting conditions for Agents depending on city type.

In Figure 4.4, the road generator started by creating a ParisAgentFactory, which takes the city position and radius as input, and spawns multiple Agents depending on random variables within the bounds of the input. The Agents are given a Paris-like strategy, which instructs the Agents to generate rings around the city center as well as main roads extending outwards. This particular strategy was configured to have a very aggressive branching, which in turn produces lots of intersections.

A similar, but a more basic approach, was applied to the Manhattan strategy, but instead of rings, multiple straight main roads were generated (see Figure 4.5).

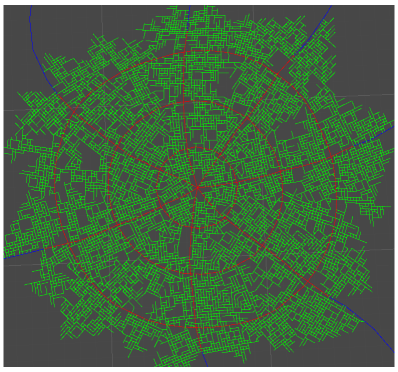


Figure 4.4: Example of the Paris road strategy.

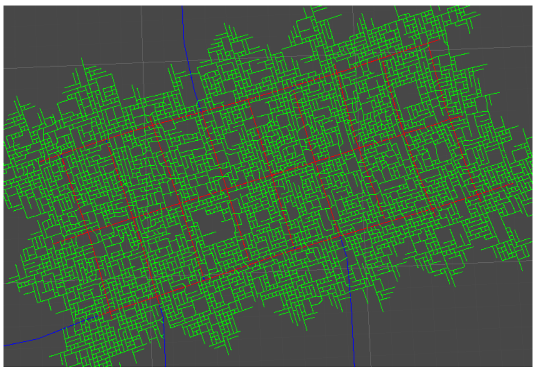


Figure 4.5: Example of the Manhattan road strategy.

While strategies handle how the Agent move, they do not decide how the roads should be placed. When an Agent decides to place a road, it will attempt to find nodes in the nearby area to snap to. If no such nodes are found, it will create a road between its previous position and its new position, and any roads that exist on the path in-between will be combined into an intersection. Figure 4.6 demonstrates the intersection created after an Agent attempts to cross another road, as well as an optimization technique making use of an R-Tree data structure [86].

The R-Tree is necessary to quickly search for intersecting roads by avoiding to iterate over all nodes in the road network, or else the performance would result in an unusable application. Each node is placed in the R-Tree with a bounding box that encapsulates each connecting node. When two nodes are connected, the road network will search for nearby nodes that intersect with the bounding box of the new connection, as well as a snap radius.

There are a few cases where it is preferable not to create new nodes for Agents, but rather snap to nearby nodes or network edges. The algorithm tests for four different cases, visualized in Figure 4.6.

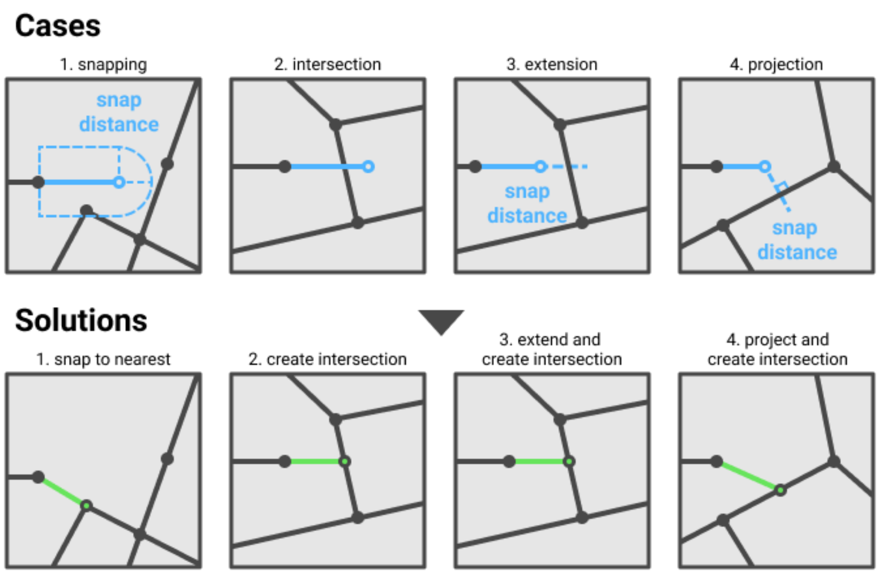


Figure 4.6: Four different cases that are handled in the road network connection logic.

The solutions to the first three cases are similar to those found in the description of the snapping algorithm in the CityGen paper [11, p. 6], but these were not enough as it often happened that nodes still appeared close to roads without creating an intersection. To combat this, the solution in the fourth case was added.

Case 1. The algorithm searches for nodes along the path to the new node, as well as within a radius around the node. The distance it searches for is defined in the Agent configuration, which any strategy can define. If a node is found, a connection between the previous node and that node will be made, making sure to re-run the algorithm in case paths are blocking.

Case 2. When there is a road between the origin node and the destination node, a new node will be created, splitting the road where the intersection point is. The destination node is then discarded, and the connection will be made from the origin node to the new node that was created. Agents will always stop at the intersection point instead of where it attempted to place the destination node.

Case 3. There are cases where an Agent lands just slightly before another road, but not near enough any other nodes. In these cases, the best approach is attempting to extend the node forward a certain distance (the snapping distance), and check if any other roads are intersecting. If any are found, split the road at the intersection point and connect the origin node to the intersection node.

Case 4. The final case occurs when the angle of approach relative to another road is small enough that the extension in the previous case will fail because the intersection point is further away than the snapping distance. To solve this, the network checks if there are any roads nearby by creating a perpendicular projection onto onto those roads. If the distance to that projection point is within the snapping distance, the same solution is applied as in **Case 3**.

First, the algorithm will search for nodes along the path to the new node, as well as a radius around the node. The second test attempts to *"extend"* the node further forward, and if it intersects with an edge, it will create an intersection there. The last test does another intersection test, but instead of extending the node forward, it will project the node to nearby edges and create an intersection there if it is within the snap radius.

While these cases solve most of the issues in the road network and make sure Agents are not creating unnecessary roads, it is still not enough to guarantee a good-looking road network. In cases 2, 3, and 4, one question that came up was *"what happens if the node that was created at the intersection is too close to one of the nodes at the ends of the connection?"*. The solution to this was that before the node is created and connected, it would take the point of the intersection and check for nearby nodes on the ends of the connection. If any of those are within the snapping distance, it will snap to the nearest one instead.

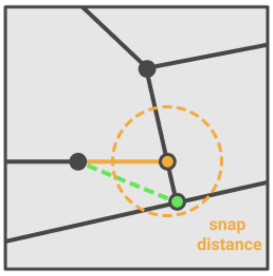


Figure 4.7: Example of the test step before intersection node is created.

Along with main roads within city bounds, RoadGenerator also produced highways that connected cities together. The goal of highway strategies was to instruct Agents to generally move towards higher population areas. When a suitable strategy is implemented that accomplishes this, other strategies can make use of the ability to switch Agent strategies mid-iteration. For instance, the Paris and Manhattan strategies that were used in the project would switch the Agent strategy to a highway one once the Agent stepped outside the boundaries of the city (see Figure 4.8).

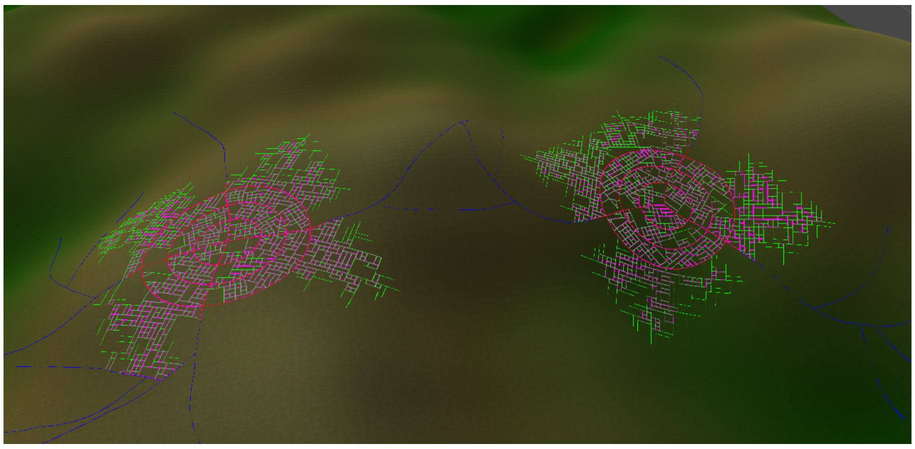


Figure 4.8: Highways created outside the bounds of the city (blue lines), as well as streets (green) with blocks (purple).

The highway strategies that were implemented into the project had a lower probability of branching than other strategies, which produced roads that were longer with fewer exits. The aim was to produce highways that mimic the structure of highways in the real world, which this logic usually did by connecting cities together with long highways.

When the road generator is finished with the general layout of the city, the final step of generating the city roads is streets. This was accomplished using the same road generator and Agents, but with a different strategy, which also shows the level of flexibility of the Agent-based approach. The only difference is that these Agents start on any of the nodes in the road network and uses a street strategy, instead of starting as a city type strategy.

Furthermore, the street strategy (like other strategies) can watch the population map and react accordingly, either by changing direction or terminating the Agent. Since the streets are a major part of the overall shape of the city, by letting the street strategy watch the population map, it could directly shape the city on a more detailed level than the city strategies could. If the population density is too low, it could terminate the Agent, making sure it does not create streets and houses in an area where no population exists.

4.2.4 City Block Generation

In this project, a city block is defined as a continuous area of land which is suitable in shape, size, and position for containing multiple buildings, parks, or parking lots. The generation of such blocks is managed by BlockGenerator (see Table 4.5).

Input		**Function**		**Output**
RoadNetwork, PopulationMap	$\rightarrow$	**BlockGenerator**	$\rightarrow$	*Block[]*

Table 4.5: Definition of the BlockGenerator function which is responsible for generating city blocks.

This function uses the road network from the road generation to find suitable areas of land to extract, and it uses the population map from the population generation to determine which type of city block it should label each area as. Examples of such labels include: *industrial*, *suburbs*, *downtown*, *skyscrapers*, *apartments* and *parks*.

The labeling of city blocks was implemented in a rather simple manner. Each label was restricted to only occur within a specific range of population density and was given a weighted probability of occurring compared to other valid labels. In practice, this was accomplished by assigning each label a probability distribution that could then be queried using the population density of the city block area. This approach makes it possible to distribute the frequency of labels based on population as well as proximity, which subsequently can be used to mimic patterns found in real-world cities such as San Francisco (see Figure 4.9).

Figure 4.9: San Francisco skyline has a cluster of skyscrapers in the city center and suburbs surrounding it [87]. This pattern should be possible to mimic using weighted labels.

The trickier part was to find suitable areas of land to extract into city blocks. The approach that was considered consisted of two iterations that both treated the road network as an undirected graph. The first iteration would find all the minimum cycles in the road network graph and extract the area enclosed by each cycle. These areas would then be treated as city blocks if their size and shape seemed suitable. A limitation of this iteration was that all blocks would be perfectly enclosed by roads.

The intention of the second iteration was to supplement the first by producing new cycles instead. The idea was to traverse the outskirts of the graph and attempt to extend imaginary nodes. These nodes would then help form new cycles as seen in Figure 4.10. The imaginary nodes would solely be used for defining the areas of the new city blocks, and would not contribute to the original road network. This iteration would help counter the limitation mentioned in the first iteration, but unfortunately, only the first iteration was implemented due to time constraints.

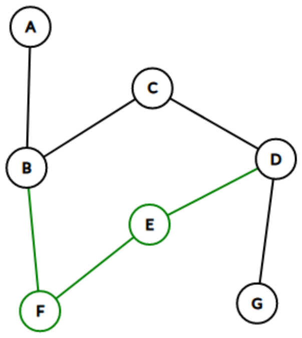


Figure 4.10: Conceptual example of what the second city block extraction iteration could have produced. Here, the new cycle $BCDEF$ has been formed. Intersections are illustrated as nodes, roads as edges, and the extended block is colored green.

The first iteration needed to find all minimum cycles in an undirected graph. This is a known problem called Minimum Cycle Bases (MCB) and can be solved in $\mathcal{O}(m^3 + mn^2 \log n)$ or $\mathcal{O}(m^2 n^2)$, for a graph with n vertices and m edges, using the efficient implementations proposed by Melhorn and Michail [88]. These asymptotic time complexities might suffice for small cities, but not for multiple large ones. Thus, either significant constraints had to be made, or another solution had to be found.

Fortunately, another solution was found. Unlike typical graphs found in discrete mathematics, these road networks are constrained by physical space, and are therefore a type of geometric graph. Essentially, each node has a position relative to others and each edge has an angle relative to others. With this information it actually becomes possible to solve the MCB problem, seemingly in only $\mathcal{O}(n + m)$.

The solution, first proposed by Petovan [89], can be described as follows:

1. For each node, dispatch a *turtle* along each connected edge. In this context, *turtle* represents a unit that is able to traverse the graph.
2. Let the turtle traverse the graph by always following the edge closest (in counter-clockwise direction) to the previously traversed edge. Essentially, let the turtle always follow the rightmost edge relative to its current heading.
   (a) If the turtle tries to traverse some edge in a direction that has already been explored (potentially by another turtle), then terminate this turtle.
   (b) If the turtle tries to traverse an edge it has already traversed, then terminate this turtle.
   (c) If the turtle found back to the start node, then
      i. If the turtle has turned an accumulated amount of 360 degrees counter-clockwise, discard it.
      ii. otherwise, add the nodes the turtle visited to a list of minimum cycles.
3. The resulting list of minimum cycles is the MCB of the graph.

Each turtle traverses the graph using the Pledge algorithm [90], which ensures that all non-terminated turtles will return to their start node. The step *2.a* ensures that all cycles have an area, and step *2.c.i* handles the edge case where one turtle follows the perimeter of the whole graph. The remaining cycles must be minimum since, otherwise, some edge would have to exist within the cycle but that edge would have been traversed by always following the rightmost edge, no matter the starting node (see Figure 4.11). Thus, the correctness of the algorithm is shown.

Each edge is traversed twice (once from each direction) and each node is considered once for dispatching the turtles. The rightmost edge can be queried in constant time by storing edges in counter-clockwise order in each node (and by storing array indices in the edges). Thus, the algorithm runs in $\mathcal{O}(n + m)$. With this efficiency, the generation of city blocks would no longer risk being a bottleneck.

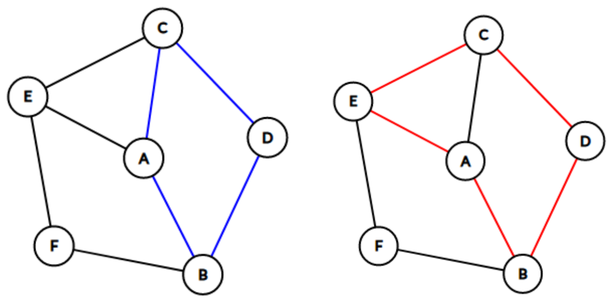


Figure 4.11: Example of a minimum cycle (blue) and a non-minimum cycle (red). The red cycle has the edge AC which splits the cycle into two minimum ones.

Before returning, the BlockGenerator also insets the polygons of the resulting city blocks. This step is needed to make sure the city blocks do not overlap with the road mesh of each edge. Any further refinements of city blocks are subsequently handled by the plot generation.

4.2.5 Plot Generation

In this project, a plot is defined as a continuous area of land within a block that is suitable for containing a single building, park, or parking lot. The generation of such plots is managed by PlotGenerator (see Table 4.6).

Input		**Function**		**Output**
Block, PopulationMap	$\rightarrow$	**PlotGenerator**	$\rightarrow$	*Plot[]*

Table 4.6: Definition of the PlotGenerator function, which is responsible to split a block into one or more plot.

Each plot also has a label associated with it that determines what type of content should later be generated inside it. The label of the block, along with the population for each plot, is the deciding factor when plot labels are selected. For instance, if the block label is *Park* or *Parking*, then the entire block is converted to one plot with the corresponding plot label. However, if the block label is *Downtown*, then the population density is used to sample a building from a pool of building types. As the population increases, so does the probability that the plot label will be set to *Skyscraper*, to accommodate for the population.

PlotGenerator was designed because blocks tended to be too large to suit a single structure, such as a building. Instead of directly placing multiple buildings on a single block, the block is divided into multiple plots. This separation of concern leads to greater modularity and makes it easier to mix a large variety of different plot labels within a single block. The core of this plot division problem was essentially about splitting an arbitrary polygon into sensible sub-polygons.

One of the proposed solutions was to draw a straight line through the block polygon and let the new sub-polygons be the result. This does not, however, result in a good split necessary. It is hard to control the size of the polygons, as some can become quite small. There is also the issue with concave polygons, where it can become difficult to create visually pleasing plots with consistency. One can not either reliably let the number of plots be a deciding factor to find the needed cut.

An article about splitting polygons was found that explained how to cut a polygon into N parts, where each part would have equal size [91]. In this algorithm, each iteration finds possible cuts that could be made. The cuts are found by finding different ways to draw lines through what is left of the polygon. Some cuts are omitted in cases where concave polygons exist since they can be outside of the polygon. The cut that is the shortest for each sub-polygon is the one that is applied and added to the output. This was ultimately the algorithm used in the final product.

4.2.6 Building Generation

The purpose of BuildingGenerator is to generate different kinds of visually pleasing buildings, everything from small houses to towering skyscrapers. BuildingGenerator takes three inputs: the plot in which to build, the terrain on which to build on top of, and the population density that helps determine the size of the building 4.7.

Input		**Function**		**Output**
Plot, Terrain, Population	$\rightarrow$	**BuildingGenerator**	$\rightarrow$	*Building*

Table 4.7: Definition of the BlockGenerator function, which is responsible for constructing a single building on top of a plot.

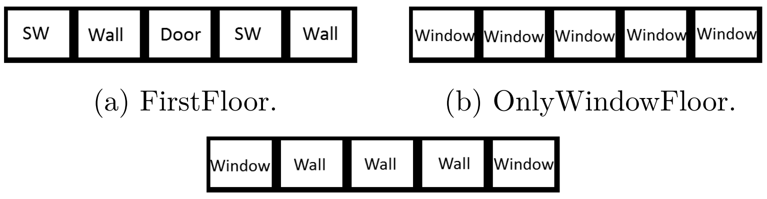
As buildings were considered a core part of a convincing modern city, it was essential to have a generator that could produce many different kinds of buildings. This problem was broken down into several sub-generators, each responsible for a particular type of building. The plot label largely determined the sub-generator chosen for a specific plot.

There two strategies used for generating buildings, the first of which used stochastic L-systems to generate diverse buildings. The L-systems were implemented with type parameters to be able to handle the generation of many objects e.g. walls and floors. The two types of parameters are the object type in the L-system and the data class that is used for the generation. An example of the code to generate wall segments can be found in Figure 4.13. A building generated via L-systems has some generated floors, where each floor has walls and is built with multiple smaller textured wall segments. These wall segments include windows, shop windows, and doors.

As an example, the steps for generating a Manhattan-style skyscraper are as follows:

1. An L-system is used to generate different kinds of floors for the building. These floor types indicate what kind of L-system should be used later for the generation of the wall segments. Example of floor types include:
   - FirstFloor - A floor that has shop windows and doors.
   - OnlyWindowFloor - A floor that will only have window segments.
   - MirrorFloor - A symmetric floor where one half of the floor segments are copied and flipped to the other half.

   These example types are visualized in Figure 4.12.
2. Each wall is then generated, floor-by-floor. Each floor type has its own L-system to generate different wall segments.
3. The walls are then put together inside the plot, with a flat roof on top.
4. The building is then placed on the highest point of the terrain within the plot. Basement walls are then generated downwards to connect the building with the ground.



(a) FirstFloor. (b) OnlyWindowFloor.

(c) MirrorFloor.

Figure 4.12: Three different floor types and an example of their wall segment generation.

The second strategy was only applied to certain skyscrapers and used a large texture atlas of windows from which subregions of windows were randomly sampled. These skyscrapers were also adjusted in size, depending on the population. This strategy, although quite simple, provided useful variation for windows, which tend to look a bit too similar otherwise.

```
lSystem = new LSystem<ManhattanWallSegmentType,
   ManhattanSegmentsGeneratorData>();

lSystem.ShouldContinue(value => value.widthLeft > 0);

lSystem.CreateRules(Corner)
    .Add(0.5f, Wall)
    .Add(0.5f, Window)
    .OnAccepted(value => new
       ManhattanSegmentsGeneratorData(value.widthLeft - cornerWidth));

lSystem.CreateRules(Window)
    .Add(0.5f, Wall)
    .Add(0.5f, Window)
    .ShouldAccept(value => value.widthLeft >= windowWidth)
    .OnAccepted(value => new
       ManhattanSegmentsGeneratorData(value.widthLeft - windowWidth));

lSystem.CreateRules(Wall)
    .Add(0.5f, Wall)
    .Add(0.5f, Window)
    .ShouldAccept(value => value.widthLeft >= wallWidth)
    .OnAccepted(value => new
       ManhattanSegmentsGeneratorData(value.widthLeft - wallWidth));
```

Figure 4.13: Detailed implementation of wall segment generation via the implemented L-system in CityCraft. This structure gives developers freedom to customize generation to their choosing. Notice how the *Wall* segment types have a 50% chance of the next segment being another *Wall*, or a *Window*.

4.2.7 Park Generation

The ParkGenerator, much like the BuildingGenerator, has the primary task of filling city plots with visually pleasing, interesting content. To accomplish this, the ParkGenerator was implemented as a function that takes two inputs, the plot to generate the park in and the underlying terrain. These parameters are then used to produce a park as output (see Table 4.8).

Real-world parks come in all sizes and shapes and can contain a large variety of different objects. Consequently, in order to not spend too much time looking for new objects to include within parks, the group decided to settle for bushes, trees, rocks, and paths.

Input		Function		Output
Plot, Terrain	$\rightarrow$	**ParkGenerator**	$\rightarrow$	*Park*

Table 4.8: Definition of the ParkGenerator function which is responsible for generating parks.

The development of the Park Generator consisted of two main phases:

- Filling the park with objects such as trees, bushes, and rocks.
- Creating natural paths for people to walk on.

To make the parks vary in appearance through procedural generation, a way to randomize the placement of objects inside any given plot had to be constructed. However, pure randomness of placement would produce some unrealistic results. Another problem to tackle was the frequency of objects i.e. determining how much of each object is reasonable to have within each park. For example, one tree, 43 stones, and two bushes would likely look like an odd park.

The algorithm for generating objects was implemented by making use of uniform randomness between 0 and 10. The range 0-10 was split into several separate, but not equally large sub-ranges, which would determine what object to generate. For example numbers in the range [6,10] could result in a tree, [2,5] could result in a bush, and [0,1] could result in a rock. As real-world objects are never identical, having one model for each object type would not be sufficient. Therefore, once the object type is decided, a model is sampled from an array of such model types, and then randomly rotated and scaled.

It was considered important to be able to distinguish the park-objects based on size and type as these parameters would determine how far from each other objects would be allowed to be generated. This is referred to as giving each object-type a radius relative to other object-types. A function was then constructed to search within the radius of each object and make sure no object could be created within another object's radius (see Figure 4.14). This implementation is based on Poisson Disc Sampling [67] in a manner that it scatters points randomly across the plot, while maintaining a distance from each already created point. Here the object-radius is used to determine how close objects may be placed. When the paths were implemented, a slight adjustment also had to be made to the algorithm for object placement, such that all objects took into consideration the coordinates of the path and were not placed on top of it.

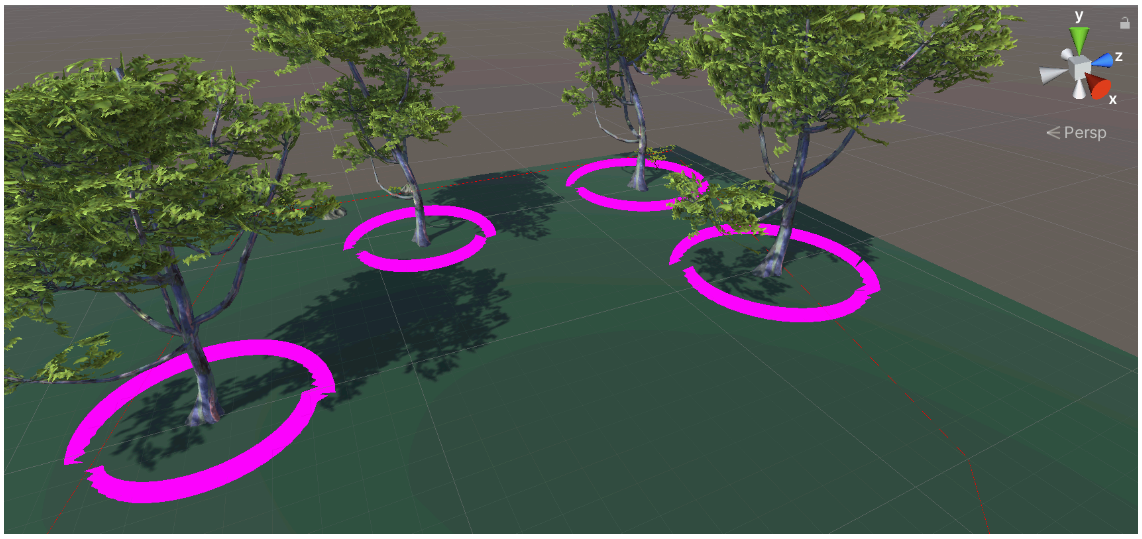


Figure 4.14: Radius of the tree object shown in pink. This radius indicates how closely trees may pack.

The algorithm for creating paths was inspired by Cyclic Dungeon Generation [92]. The main takeaway from that concept was that a start and goal node was generated for the algorithm to create a loop between. Having a loop would allow for people to take a stroll around the park and return to where they entered, and based on observation, this is also a frequently occurring pattern for paths in real-world parks.

The development of the path generation can be described as three iterations of implementation. The first implementation of the algorithm created two random points within the plot, one as start and another one as goal. The program then traveled the park at random until it found a path to the goal, at which point it would find a new way back to the starting point. It was however, quickly discovered that this approach had some problems.

One of these problems was that the paths would look unnatural when taking steps far away from the goal. A solution to this was to revert the path to its previous node if it took a step which was further away from the goal than it had previously been, this was an instant improvement in appearance. However, after observing many of the generated paths from this approach the paths still looked linear. This was believed to be because simply going from one point to another and not straying of too much from the goal could not produce much more interesting results than that (see Figure 4.15).

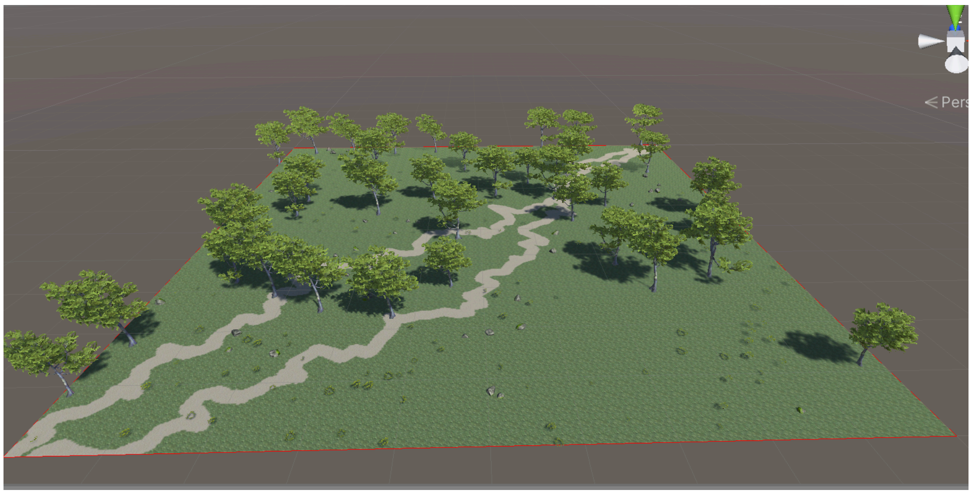


Figure 4.15: Example of a path generated by the first implementation of the algorithm.

The second implementation of the algorithm was inspired by the flaws which had been demonstrated by its predecessor, more specifically its linear shapes and its tendency to create steps with abrupt angle-changes. As a result of this, it was for the second implementation decided that rather than having a start and goal, the path would consist of an array of points, which all had to be visited before the path would be completed. This was changed in order to try to get rid off the linearity presented in the first implementation. For this implementation a control of how many degrees a step in the path could differ from the current point to its previous step was also added. This was done to avoid the zigzag-like patterns caused by the angle-changes from the first implementation. The second implementation would just like the first one look for one assigned point, but in this case start searching for the next point after finding the previous one and continue doing so until all points have been found (see Figure 4.16).

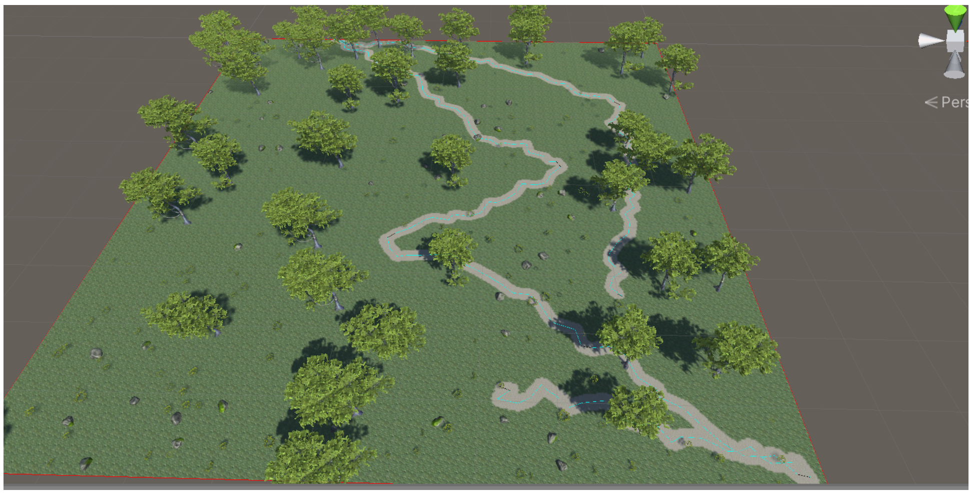
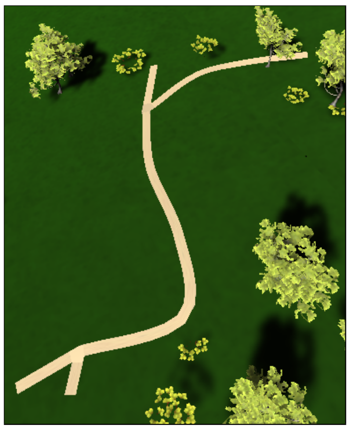


Figure 4.16: Example of a path generated by the second implementation of the algorithm.

The third and final implementation made use of a similar algorithm. Some fairly small changes were made in order to address the issues experienced with the second implementation. The second implementation would still create paths starting in the middle of the park which did not make a lot of sense, this was handled by always creating the start node on one of the edges of the plot. The algorithm then creates an additional number of randomly placed points, based on the size of the plot. Having this number be based on the size of the plot made for some more interesting and realistic results which was lacking in the second implementation.

Afterwards, the points are sorted by distance to enforce that the algorithm always traverses towards the point closest to itself. This change was inspired by seeing some paths traverse towards points far away from the starting point only to then traverse to a point close to the starting point. Once all goal points have been reached a path is either created back to the starting node, or a new path is created to a randomly selected edge on the plot. This change was made to stop enforcing loops and to add more variety to the generated paths. Finally, the algorithm creates entries to the park by projecting a line to closest edge of the plot from some of the points, and then creating a path between them.

The visual implementation of the paths was done by texture splatting on top of a Unity Terrain for the first two implementations (see Figure 4.15 and Figure 4.16). However, as the Unity Terrain needed to be replaced with a mesh generated one, this had to be re-implemented for the third implementation. Some different approaches for visualizing the paths were investigated, such as using quad and sphere meshes and then simply placing these along each path. The spheres would cause a shape that was still distinguishably circular and the quads would likewise create patterns that were clearly square. The final implementation would instead make use of the logic from the RoadGenerator to create its own internal road network, thereby inheriting the functionality used to generate the corresponding path mesh (see Figure 4.17).


(a) Visualization of a path using splines.

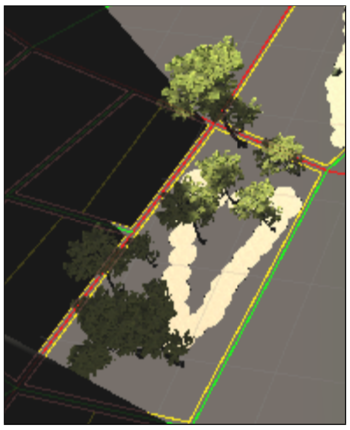

(b) Visualization of a path using spheres.

Figure 4.17: Two of the approaches for visualizing paths on top of the mesh generated terrain.

4.2.8 Parking Lot Generation

The ParkingGenerator function takes a plot and its underlying terrain as input, and produces a parking lot as output (see Table 4.9). This generator in particular is responsible for filling roughly ten percent of the generated cities with parking lots, a percentage based on research conducted in Phoenix, AZ [93].

Input		Function		Output
Plot, Terrain	$\rightarrow$	**ParkingGenerator**	$\rightarrow$	*Parking lot*

Table 4.9: Definition of the ParkingGenerator function which is responsible for generating parking lots.

One approach considered for the generation of parking lots was to cover areas with textures. This would have been a straightforward option to implement, but it was found that it offers little alternatives for modification. For example, the shape of the entire parking lot as well as the size of the individual parking spaces would be difficult to adjust afterward using this approach. Furthermore, using a mesh with one texture applied to it would remove the possibility of scaling the size of the parking spaces within the mesh without scaling the mesh itself. As every other generator provides fully scalable content, it would be inconvenient for the ParkingGenerator to be any different.

Another approach was to generate each line in the parking area individually. This was the initial approach that was used in the application, but was changed due to performance concerns when dealing with massive amounts of line objects.

The final approach was a combination of these two methods. The first step of this approach is to fill the plot with asphalt. This could be achieved by projecting an asphalt mesh onto the plot, however this mesh would need to be perfectly aligned with the terrain mesh. This was solved by writing an algorithm that essentially cut out a portion of the terrain mesh in the shape of the plot then slightly offsetting it upwards to avoid overlapping triangles with the terrain (see Figure 4.18). With this method, the mesh would never clip with the terrain.

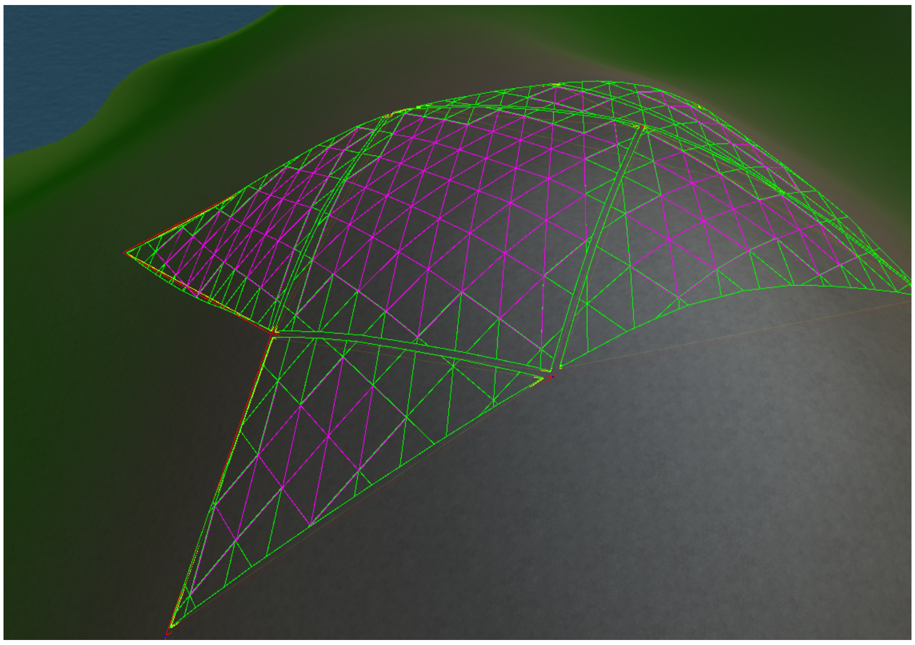


Figure 4.18: Mesh projected onto the terrain.

The second step is to place parking lots in suitable locations within the plot. This was done using a function to approximate the largest rectangle inside a polygon which finds a suitable rectangle to generate parking lots in (see Figure 4.19). Based on the size of the approximated rectangle, the algorithm generates as many rows of parking lots as it can fit. Each row contains a quad that is projected onto the terrain with a transparent texture where only the white parking lot lines are opaque. By modifying the UV coordinates of the quad, the texture can be repeated and scaled to any desired size. The plot is also given a margin around it to make sure there is enough space around the parking lots for cars to get in and out of the parking area.

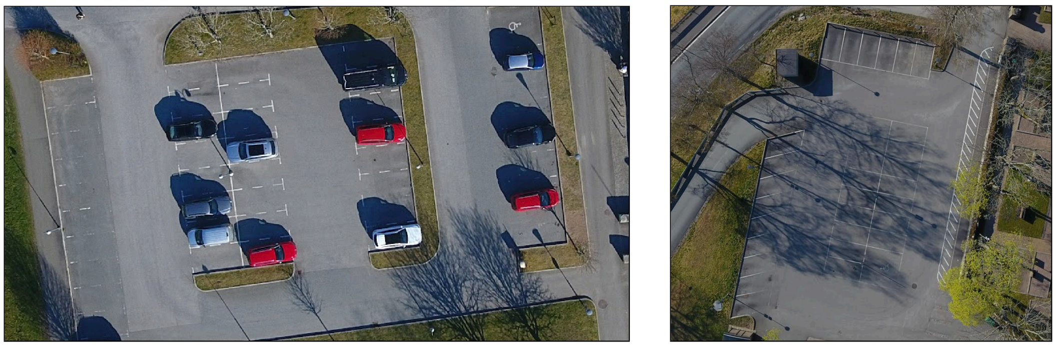


Figure 4.19: Two drone pictures of rectangular shaped parking lots taken by the project group. The left image most clearly displays the layout that was intended to mimic.

4.3 Process and Workflow

This chapter details how the group collaborated throughout the project. The first subchapter describes how the group approached the software development process and how the group distributed the workload. The second subchapter describes the tools and software used to aid the group to collaborate more effectively.

4.3.1 Software Development Process

The group decided that the development of the application would follow an agile development process. Agile software development generally involves self-organized teams working in iterations, where requirements and solutions actively evolve throughout the development process [94]. This approach felt suitable for the project's loose definition of when the application could be considered complete. For this reason, the group decided to work in intervals of two-week long iterations. At the end of each iteration, the previous iteration was evaluated and new goals were defined for the next iteration.

Group meetings were the main means of discussing the status of the project, general improvements, and other topics related to the project. All meetings were considered to be mandatory and therefore followed an opt-out regimen. This included weekly meetings with the project supervisor.

Initially, the group agreed to meet twice a week for meetings. The group also decided to meet twice a week for mutual work sessions. The work sessions were not mandatory, but the group still enforced an opt-out regimen to encourage frequent collaboration. The intended reason for these sessions was to help each other solve problems that often required communication between the different implementations. This approach was later discarded in favor of a fully remote workflow due to social distancing guidelines.

After transitioning to a remote workflow, virtual meetings were held three times a week instead. The remote meetings followed the same arrangement as before, but included a formal stand-up meeting structure. This addition attempted to further coordinate and track the progress of each group member. During each stand-up, every group member answered the following questions related to the iteration goal.

- What did I complete since the last meeting?
- What do I plan on finishing until the next meeting?
- Do I see any obstacles that could hinder me or the rest of the team?

Since all members worked on the same codebase, a certain level of correctness and readability had to be imposed. Primarily, the group addressed these concerns by requiring code reviews, enforcing CI pass status, and by writing a Definition of Done (DoD) document. The usage of the DoD involved verifying that any submitted code followed a certain checklist of requirements. For instance, it was required that submitted code would not throw any warnings or errors in the Unity Editor.

4.3.2 Collaboration Tools

Version control was managed using Git [95] and hosted on GitHub [96] for collaboration purposes. This approach was well suited for the application's non-linear workflow and the need for team members to work locally on different computers.

The correctness of requested code changes to version control was also alleviated with the help of Continuous Integration (CI). CI is the process of automating the build and testing of code every time a developer commits changes to version control. The project group made use of two CI tools for validating committed code. One such tool compiled the code and reported compilation errors on every requested change. This made it easy for other reviewers to quickly see whether the requested changes were syntactically legal. The second tool analyzed the code and reported stylistic deviations from what the group had defined in the tool's configuration file. This tool could also directly reformat the code according to the specified format which helped maintain readability since the codebase was consistently formatted.

Administrative documents were stored in a shared folder hosted on Google Drive [97]. This includes documents related to meetings, group contracts, DoD, and progress updates.

5
Results

This chapter presents CityCraft, the standalone city generation software that was implemented as a result of this research. CityCraft is a graphical desktop application that lets users interactively generate 3D cities, which can be exported as glTF files and used in third-party modeling software such as Blender. The application combines computer graphics theory and several PCG techniques to achieve its results. An example of CityCraft is shown in Figure 5.1, and a CityCraft generated model rendered in Blender is shown in Figure 5.2.

Source code: `https://github.com/DATX02-20-02/CityCraft.git`

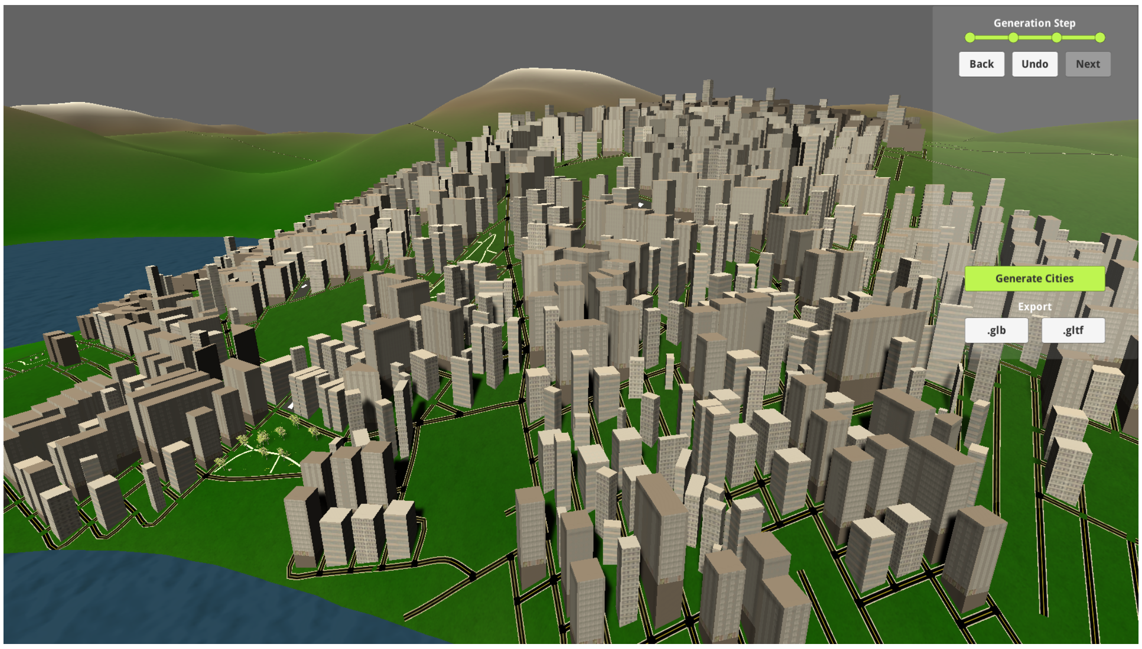


Figure 5.1: A screenshot taken from CityCraft as a user generates a city. In the top-right resides a menu panel which the user operates to perform the generation. The view of the world is controlled through a camera which is rotated with the mouse and moved with the keyboard.

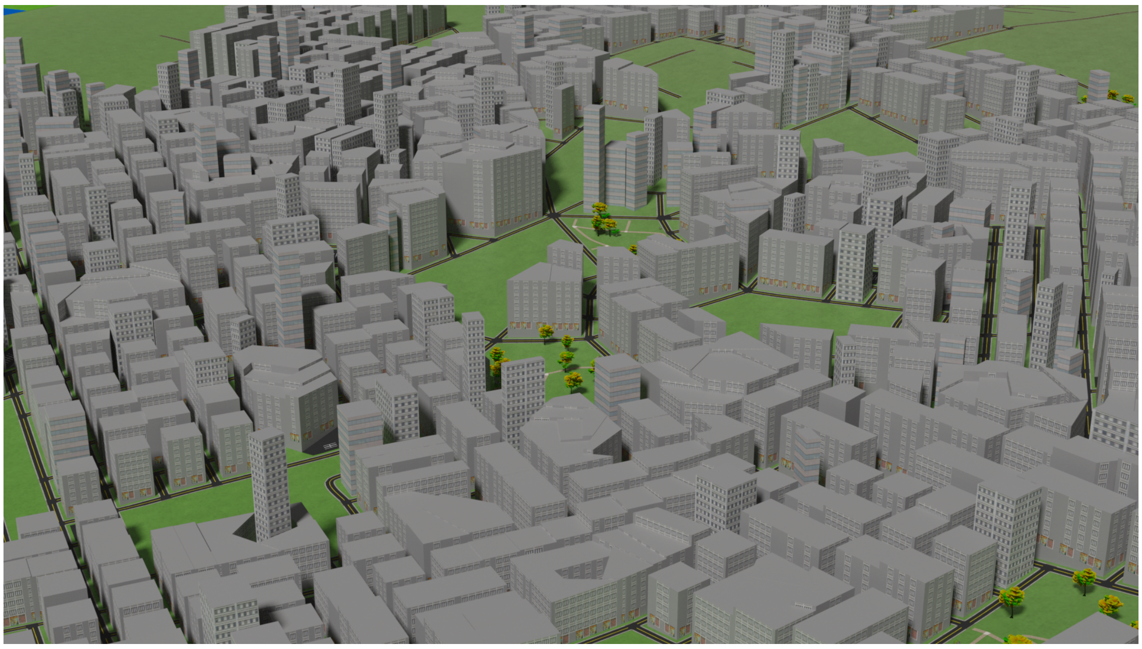
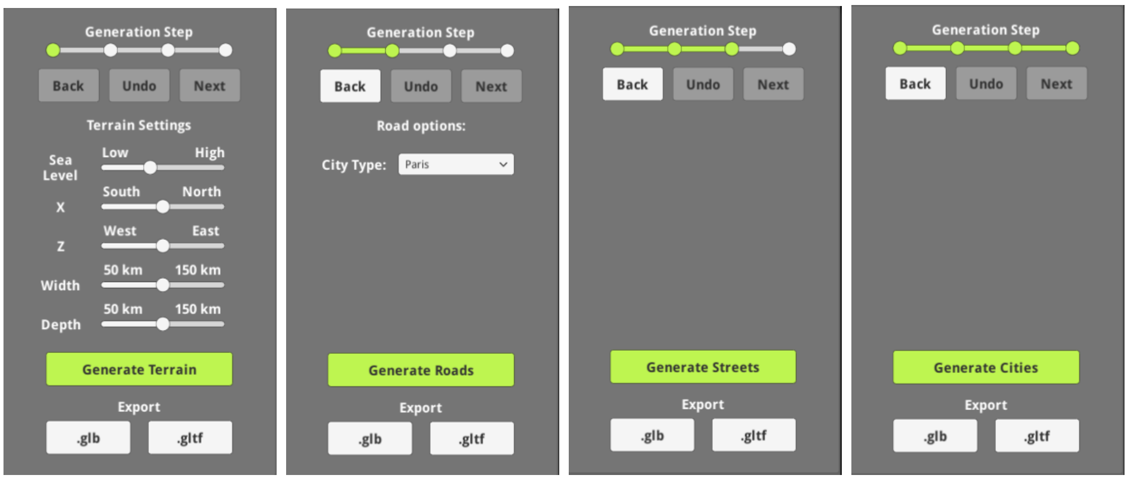


Figure 5.2: A city model exported from CityCraft and rendered inside Blender.

CityCraft's GUI follows the *Wizard* design pattern [98] to guide its users through the generation process, which is split into four steps (see Figure 5.3). The four generation steps are terrain, roads, streets, and cities. The last of which performs the generation of buildings, parks, and parking lots.



City Type: Paris

Figure 5.3: CityCraft's four steps of generation. Each step has a separate settings panel and each generation can be performed multiple times.

The four steps of the GUI are implemented as eight PCG-based generators, all of which are orchestrated by a single module called the *WorldGenerator*. The following subchapters will explain the details about the results of what each of these generators produces and how their results contribute to the final city models.

5.1 Terrain Generation

When users start CityCraft, they are first presented with an endless ocean and a menu panel, as shown in Figure 5.4. This is the start of the generation process, and the ocean represents the blank canvas on which the rest of the world will be built.

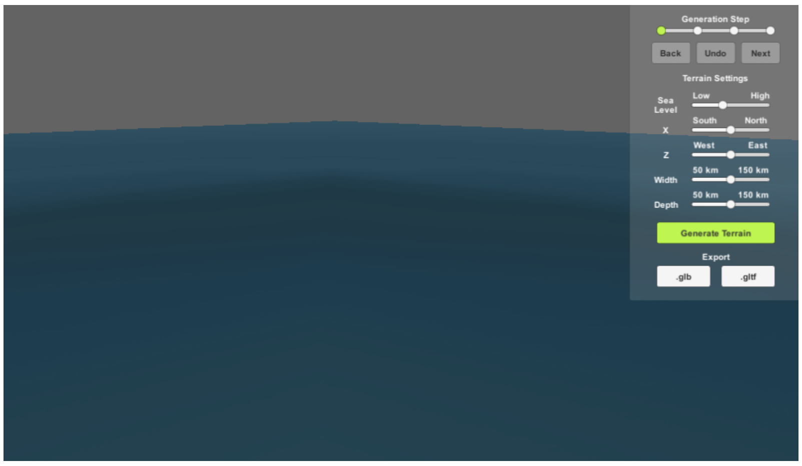


Figure 5.4: The application state before the *Generate Terrain* button has been pressed. The ocean is visible from the very beginning.

The very first step of the generation process is to generate the terrain, whose settings are adjusted in the top-right menu panel. The terrain settings that the user can adjust are:

- Sea level: modifies the water level.
- X/Z offset: offsets the sampling location along the respective axis of the noise function, effectively changing the height values of the terrain.
- Width/Depth: adjusts the size of the entire terrain.

The terrain is generated by creating a surface mesh whose vertices are given height values sampled from a 4-layered Simplex noise function. The terrain utilizes a single texture, which is repeated several times with randomized UV coordinates. The terrain is also colored based on height values, forming snowy mountain tops and green valleys. An example of a generated terrain is shown in Figure 5.5.

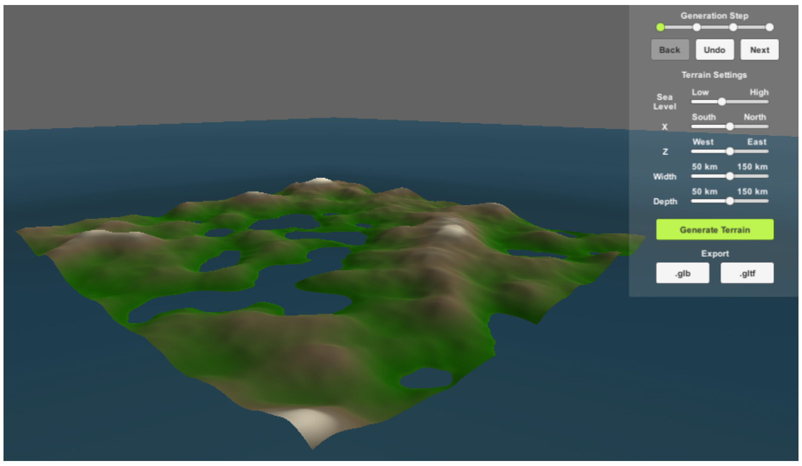


Figure 5.5: The application state after *Generate Terrain* has been pressed. The user may change settings and re-generate the terrain as many times as they like.

The ocean was just implemented as a large textured plane that clips through the terrain, forming lakes at its intersections.

5.2 Population Generation

After the terrain has been generated, the user is provided with a tool that lets them mark areas on the terrain where they want cities to be generated. When one or more markers are placed, they will remain in their designated positions until interacted with by the user, or when the user presses the *Generate Roads* button. Each marker can be interacted with by hovering over it, which will turn it red. Clicking on it will make it yellow, which shows that it is selected (see Figure 5.6).

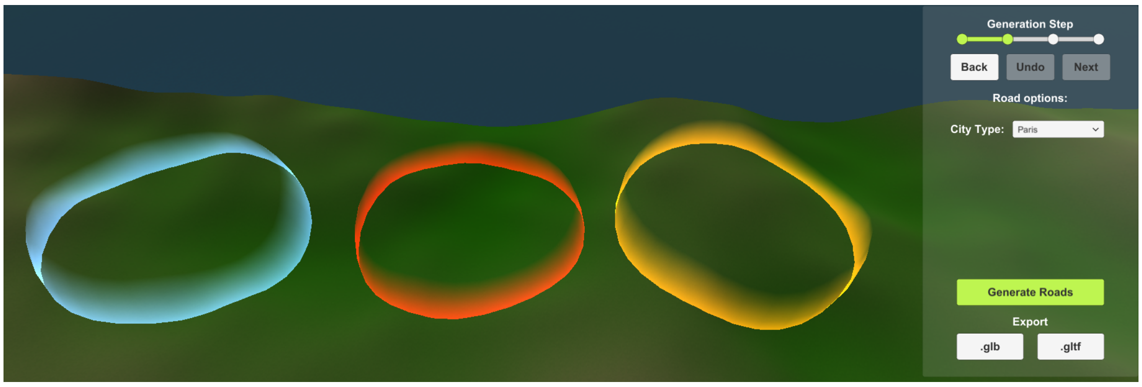


Figure 5.6: The city marker tool. Each ring represents a potential city, where the radius of the ring roughly corresponds to the size of the city that will be generated.

Selecting a city marker provides multiple options:

- Clicking again and dragging allows the user to freely reposition the marker.
- Scrolling up and down increases and decreases the radius of the marker.
- Pressing the delete key removes the marker.
- Through the menu panel, it is possible to change the type of the city that will be generated. Currently, there are Paris- and Manhattan-style cities available.

The population generation begins once the user presses the *Generate Roads* button. This generates a population density map that covers the entire terrain using a 2-layered Simplex noise function. Then, the city markers amplify the population of this map, at the spots where they were placed. Finally, once the population generation is finished, road generation begins.

5.3 Road Generation

After the city markers are specified and the population density map has been created, it is time to generate the road network. Using the markers, the road generator creates workers called Agents in the area specified by the markers and assigns them strategies depending on the corresponding city type. The purpose of the strategies is solely to give Agents instructions for movement and termination. Strategies take a few different variables into account when it decides these actions, for example the amount of steps an Agent has taken so far, the population density, and if the Agent has landed on an existing road. The application has Agent strategies for Paris and Manhattan cities, and an example of these are shown in Figure 5.10 and Figure 5.8.

How the roads are created and connected is entirely up to the road network. It handles placing down road nodes and connecting them together, making sure that intersections are created where necessary. The Agents instruct the road network where they want to place down roads, and the network handles the rest while also providing the Agent with information on how the road network was modified. Then, the strategies can use that information and react accordingly, for example to terminate the Agent.

From Figure 5.7 we can clearly observe the distinct grid-like structure of the road network, which in turn was the goal for our Manhattan generation strategy. The overall shape of the city does not necessarily mimic the shape of Manhattan in the real world because Manhattan is surrounded by water, something that we are not required to have in our application. If the city marker is placed on an isolated island, it would be much closer to the aspects of Manhattan in the real world.

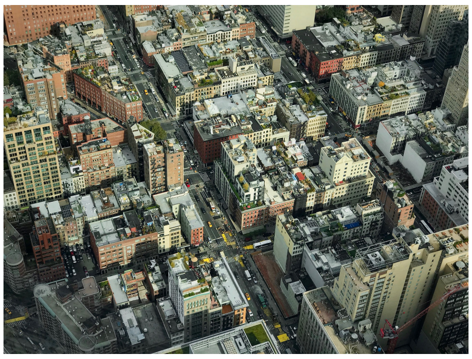


Figure 5.7: Picture of Manhattan grid-like road system [99].

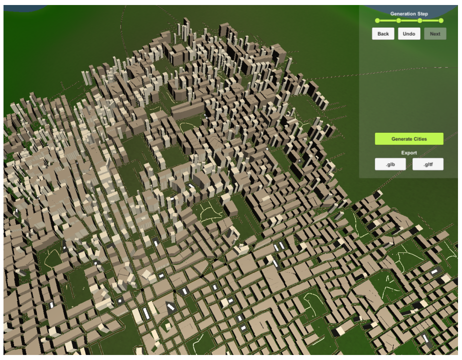


Figure 5.8: Example of a fully generated Manhattan-style city.

Meanwhile, Paris has a very different overall aesthetic than Manhattan. Around the Arc de Triomphe in Figure 5.9, there are distinct rings and roads extending from the center, which is something we aimed to mimic in this project. In the generated city in Figure 5.10, the structure does have distinct rings and roads similar to the pattern in Paris.

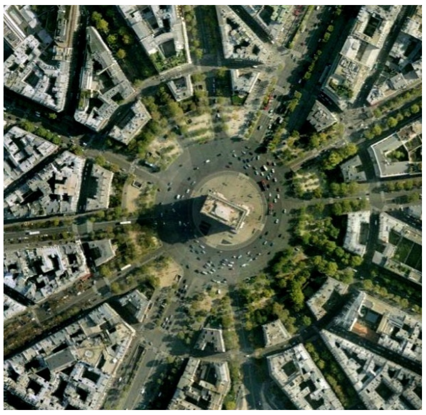


Figure 5.9: Picture of Paris ring-like road system [100].

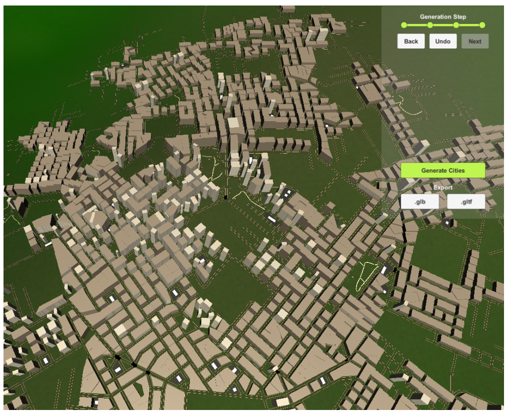


Figure 5.10: Example of a fully generated Paris-style city.

Both city types have streets branching out from their main roads, which in turn also branch out from themselves to form a grid-like street neighborhood. These streets are created using a street strategy that is shared between the strategies used for the two different city types, however it is possible to implement more complex street strategies for other types of cities.

The base of the road generator and its strategies allows any street pattern to be produced, given a suitable strategy is implemented. In order to limit ourselves in this project, the street strategy was implemented to produce grid-like streets for all city types. However, the configuration of the general street strategy can be configured per city type, which gives a small level of tweaking so the streets could match the city type better.

The generated cities are connected by the road network through what is called highways. Highways are created by the Agents through highway strategies when they have traveled far away from the designated city markers. The purpose of the highway strategy is to create longer roads with little change in direction. It also aims to follow the population map density to some degree, especially navigating towards higher populated areas. This results in highways which typically connect cities together.

5.4 City Block Generation

Once roads and streets have been generated, the city block generation starts. This generation step produces a list of polygons that mark which areas of the terrain are suitable for buildings, parks, and parking lots. City blocks may vary significantly in size, but a limit is set on how large they can become (see Figure 5.11).

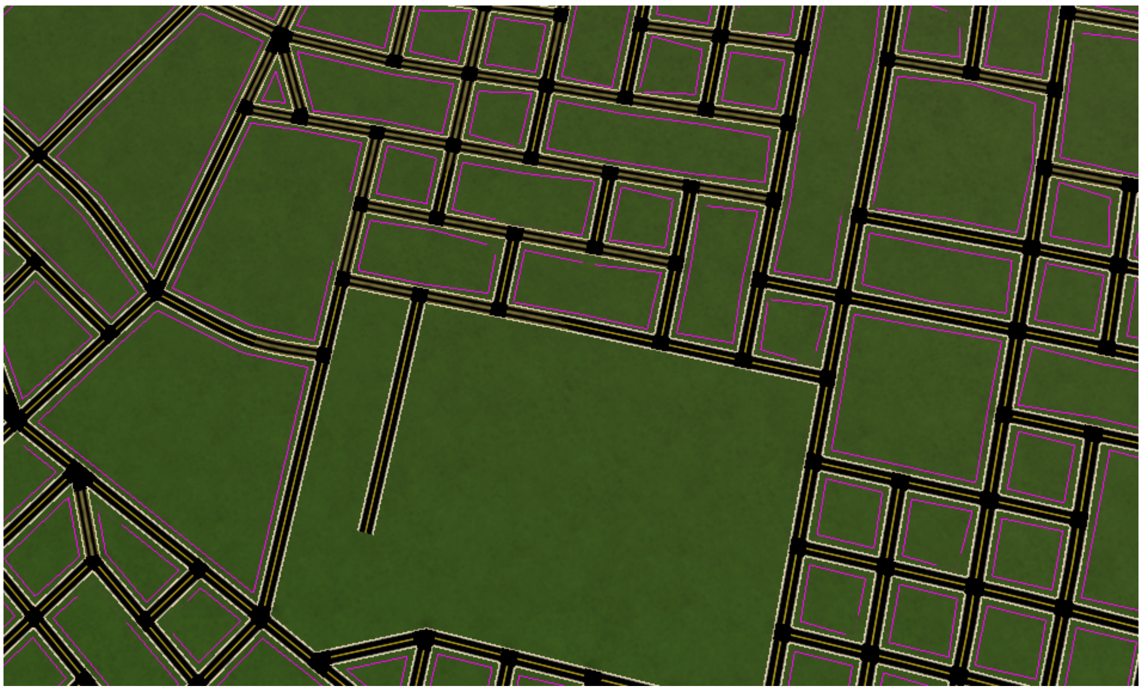


Figure 5.11: Generation of city blocks in a road network, where pink lines outline city blocks. Notice how the largest areas are not treated as blocks.

Each city block is also guaranteed to be connected to the road network, such that roads, streets, or both surround it. Consequently, each block has to be slightly inset in order to make room for the road meshes. This necessity becomes more apparent when all meshes are rendered (see Figure 5.12).

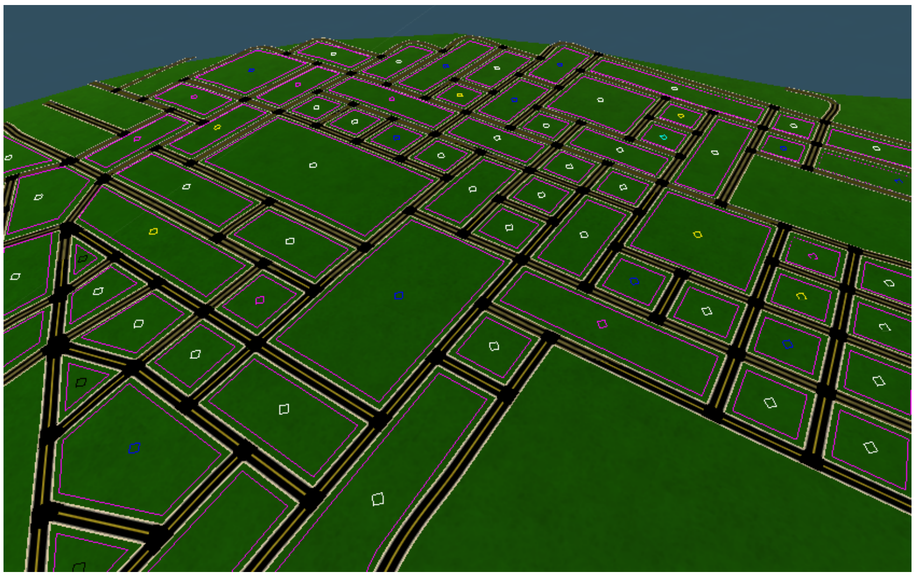


Figure 5.12: Blocks generated on a 3D terrain. Notice how each block is connected to the road network. The colored lines are only shown during development.

Each city block is assigned a label that helps the subsequent plot generation step determine what to generate inside each city block. In Figure 5.12 these labels are visualized as green triangles, meaning they have not been processed yet. The green triangles are also used to mark blocks that became too small after being inset, effectively excluding them from further processing. An example of this behavior is shown in Figure 5.13.

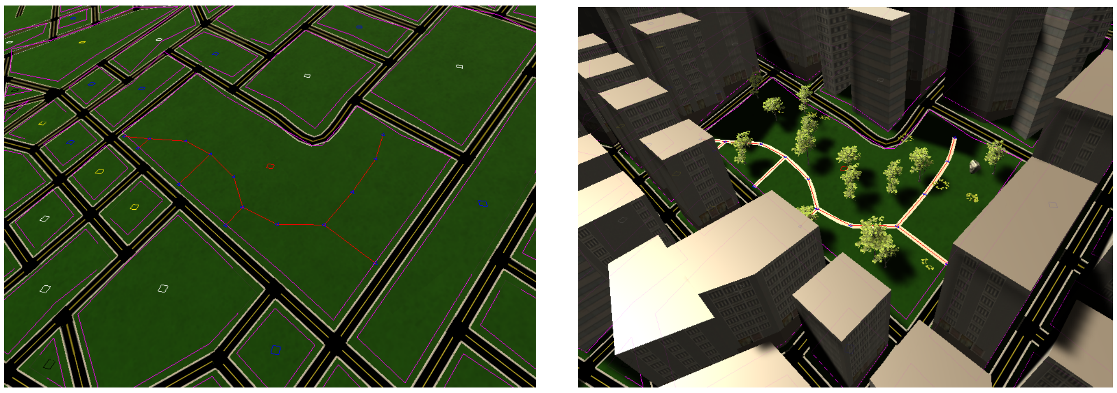


Figure 5.13: City blocks with different labels (shown as colored squares) shown without (left) and with (right) buildings rendered. Red labels indicate parks, while the other colors shown in this figure indicate various types of buildings.

The implemented labels are *industrial*, *suburbs*, *downtown*, *skyscrapers*, *apartments* and *parks*. Each label weights what type of content should be generated so that all plots within the same block share a certain theme. Finally, once the blocks are generated, they are passed on to the plot generation.

5.5 Plot Generation

Each block that is sent into the plot generator is treated differently depending on what the block label is. If the block label is either *Parks* or *Parking*, the entire block is turned into a plot and is sent to the park generator or park generator respectively. The rest of the block labels are split into multiple parts, where the amount of resulting plots depend on the area of the block as well as some random variables.

Figure 5.14 and 5.15 demonstrates some example outputs from the plot generator.

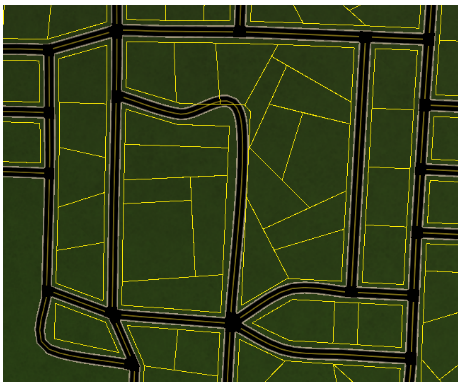


Figure 5.14: Close-up of the plot splitting algorithm.

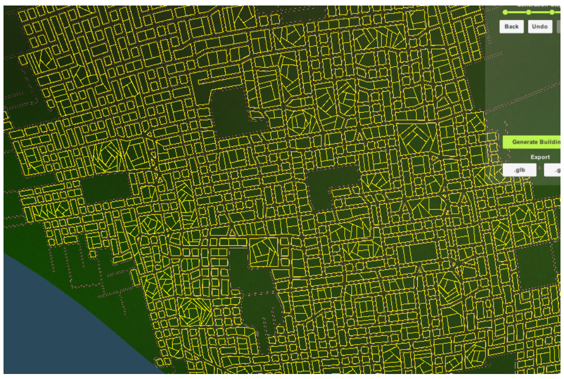


Figure 5.15: Generated plots within a larger city.

Each plot is assigned a plot label depending on the size of the plot and the population map. The plot labels are *Manhattan*, *Skyscraper*, *Park*, *Parking*, and *Empty*. *Manhattan* and *Skyscraper* are two different sub-generators for building generator. *Park* and *Parking* are used by the park generator and the parking generator respectively. If the plot label is *Empty*, then no content is generated upon the plot. Larger plots are always assigned with the *Park* label. The purpose of the plot labels is to let other generators know what should be generated within the plot.

5.6 Building Generation

Building generation produces two different types of buildings: one for plots labeled *Manhattan*, and one for those labeled *Skyscraper*. An example of a skyline shaped by these buildings is shown in Figure 5.16.

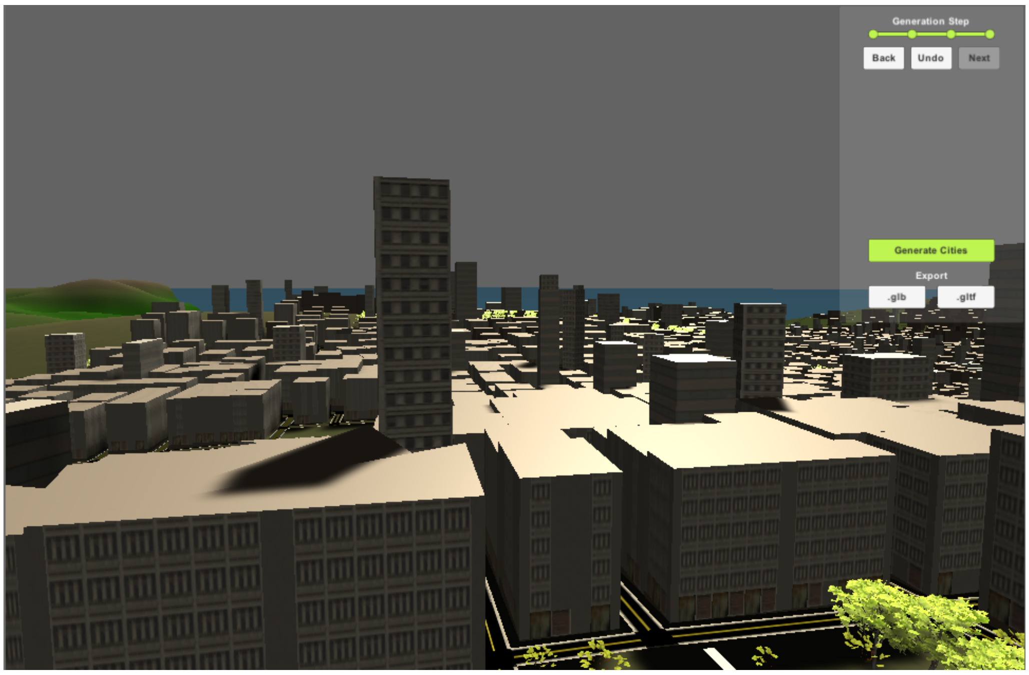


Figure 5.16: Skyline of *Manhattan* and *Skyscraper* buildings together, taller buildings being the skyscrapers

Manhattan buildings are generated with stochastic L-systems, providing versatility via modular floor- and wall-type L-systems. The final implementation had one floor-type and four wall-type generators, Figure 5.17 demonstrates some examples of this. *FirstFloor* is the first floor type for every building, its wall segment generator produces a combination of shop windows, walls, and doors. After the *FirstFloor*, the floor-type generator repeats one of the following floor-types until the desired building height has been reached. Note that each floor-type has its own wall segment generator that is run once per wall, and copied for each floor.

- *NormalFloor* - generates wall and window segments, but it only generates half of the wall. For the other half, it copies the first half and reverses it.
- *EveryOtherFloor* - alternates between wall and window segments.
- *RepeatWindowFloor* - only generates window segments.

Each of these strategies also start and end with a corner segment.

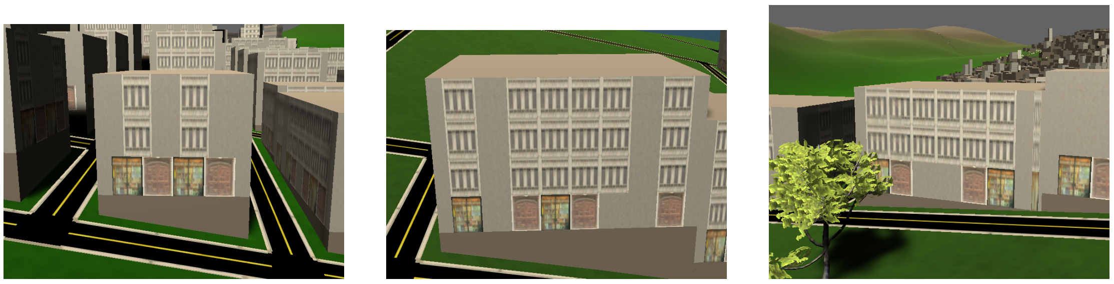

(a) *EveryOtherFloor*. (b) *NormalFloor*. (c) *RepeatWindowFloor*.

Figure 5.17: The three different wall segment generators. Notice that each building has a *FirstFloor* wall segment generator on the first floor.

The other building type is *Skyscraper*, of which examples are shown in Figure 5.18. These buildings use texture atlases of windows from which different sub-regions are sampled to produce a procedural pattern of random windows.

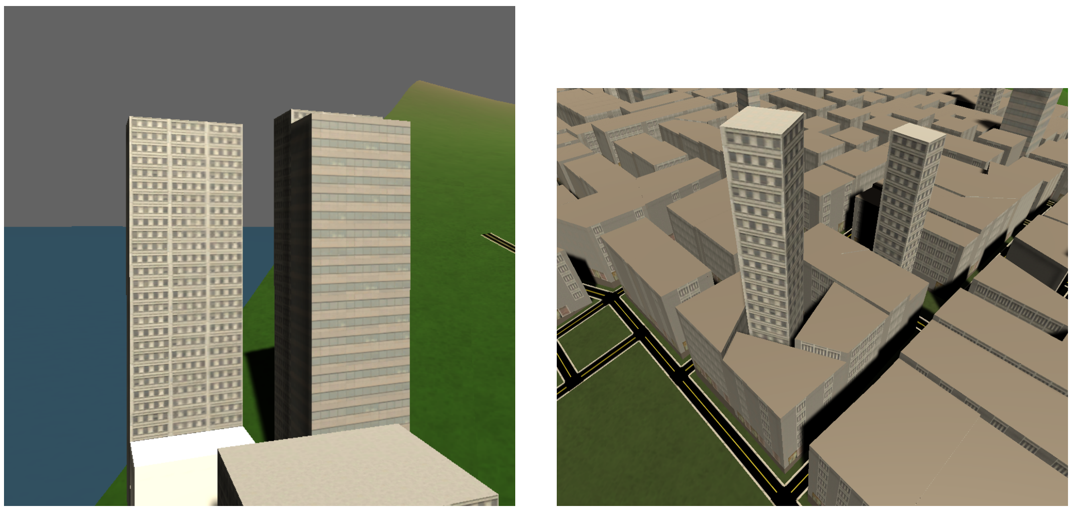


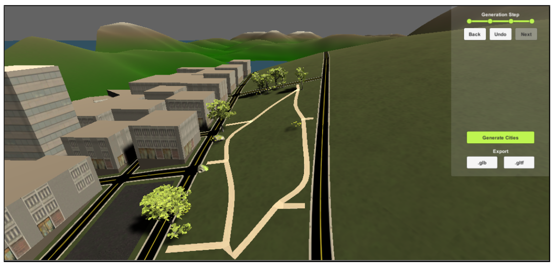
Figure 5.18: Two examples of skyscrapers generated in a city.

5.7 Park Generation

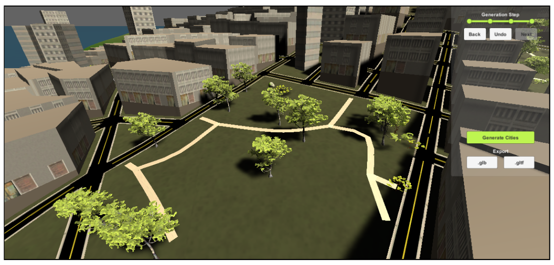
Once the plot generation is finished, the park generation starts filling some of the plots with parks. The parks are generated through two steps, the first step is to generate a natural-looking path, and the second step is to fill the park with objects such as trees, bushes, and rocks.

The algorithm for generating paths works by first generating a random point on one of the edges of the plot and then scattering points randomly inside the plot with a minimum distance from one another. The points are then sorted by distance, ensuring that the path always traverses to the closest possible point. Afterward, once all points are found, the algorithm either creates a path that loops back to the starting point, or a path towards another random point on the edge of the plot (see Figure 5.19). Finally, exits/entries are added to the park by creating more paths to the edges of the plot.

The algorithm for object placement runs after paths are created to ensure that objects do not intersect with the generated path. The first step of this algorithm is to assign each object type with an object radius, indicating the distance at which other objects are allowed to be created. Afterward, a probability of being generated is assigned to each object type. The algorithm then runs several times, and each time an object type is picked, a model of that object type is randomly chosen, scaled, and rotated before being generated and placed somewhere in the park.



(a) Park with looping path.



(b) Park with path that does not loop.

Figure 5.19: Two example of different generated parks. One with a path that loops back, and one that creates a path to another edge of the plot.

It is worth noting that even though the paths always end up looping, or connecting to a random edge, their shapes still vary a lot, leading to quite interesting results (see Figure 5.20). The fact that object placement takes the path into account also makes the park layout appear less random and more relative to its environment. This provides a more consistent feeling to the different parks.

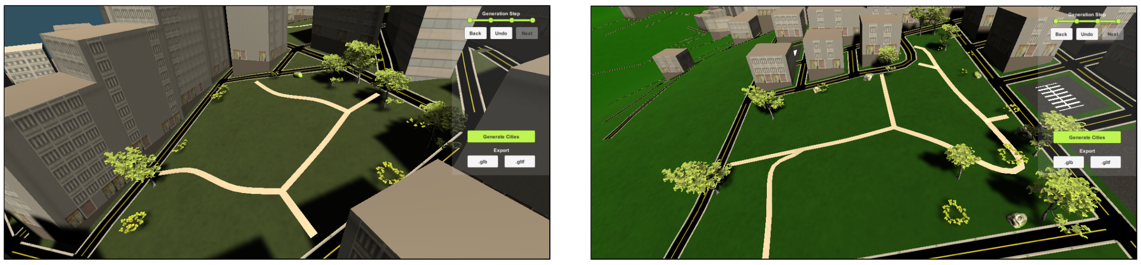
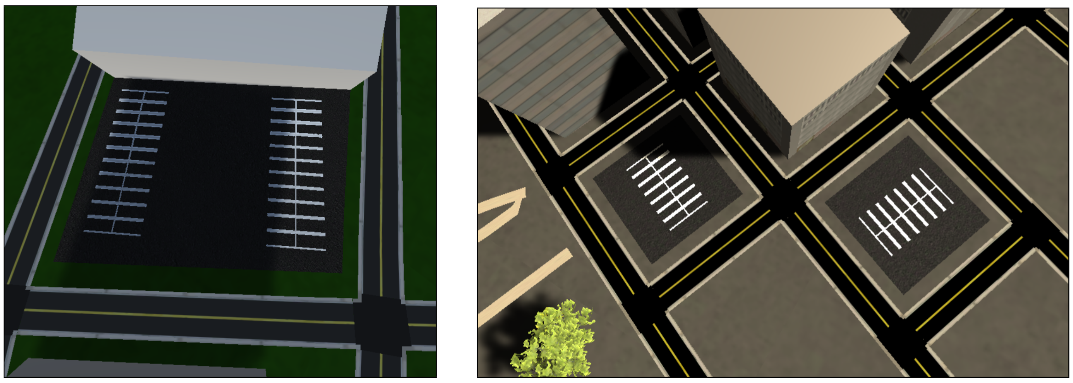


Figure 5.20: Two additional examples, showcasing different results.

5.8 Parking Lot Generation

Once the plot generation is finished the parking lot generation starts to fill some of the generated plots with parking lots. The generated parking lots consist of one or more rows of parking spaces (see Figure 5.21).



(a) Large parking lot consisting of two rows.

(b) Two small parking lots consisting of one row.

Figure 5.21: Two examples of different sized parking lots created by the generator.

These parking lots are generated by approximating the largest possible rectangle that fits inside a given plot, and then generating parking lots inside of it. The number of rows is based on the size of the computed rectangle, and the algorithm aims to fit as many rows of parking lots inside the rectangle such that there still exists space between the rows for cars to enter.

Having the parking lots consist of multiple rows depending on the size gives some more variety, but the decision to do this was also based on real-world parking lots (see Figure 4.19). In the left example in Figure 4.19, a road is used to separate parking spaces, making it impossible for parked cars to be surrounded and stuck by other parked cars. The right example, however, has a more interesting shape, the parking lot in the center of it is shaped in the two-row style, while the surrounding rectangle is large enough that cars can easily drive around and not be blocked by parked cars. This type of parking space is difficult to generate reliably in arbitrary plot shapes. Consequently, shapes like these were not considered for this project.

6

Discussion

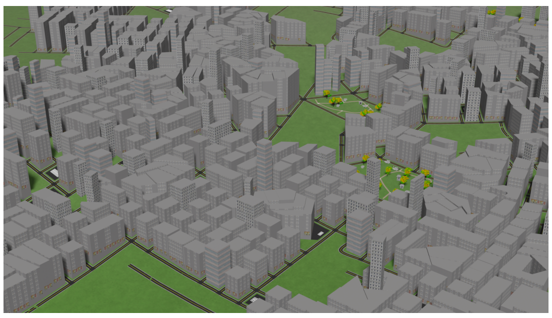
This chapter discusses the resulting city generation software, CityCraft, from three different perspectives. Firstly, the quality of the application and the city models it generates are estimated by addressing the checklist that was presented at the end of Chapter 1.2. This perspective also argues how the choice of different algorithms could have affected the results. Secondly, the validity of the processes used during development is debated. Finally, the last subchapter discusses future research.

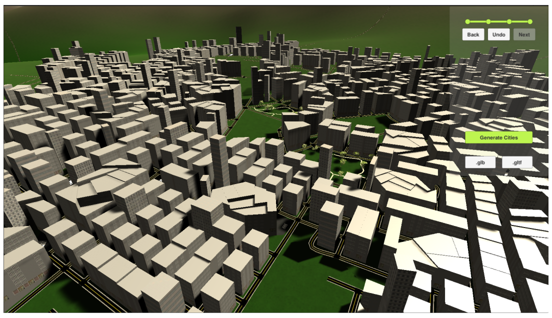
6.1 Quality of Results

This subchapter addresses the eight questions presented at the end of Chapter 1.2.

Q1: The generated city models import correctly into Blender. However, integration with other third-party tools needs to be tested more thoroughly as Blender was the main tool used in this project for verifying the correctness of the exported models.

Although these models are imported correctly in terms of data and structure, there still exist some undesired visual differences. These differences are, however, a consequence of the difference in rendering pipelines between Unity and any other third-party tool. Figure 6.1 showcases these visual differences with Blender.


(a) CityCraft scene rendered in Blender.



(b) CityCraft scene rendered in Unity.

Figure 6.1: Visual differences of cities when rendering in Blender and Unity.

Importing a generated city into Blender also takes a substantial amount of time in comparison to when exporting the files from within CityCraft. The reasons for this are unknown, and other tools might import faster. In conclusion, the integration with third-party software such as Blender works albeit with some frustrating elements.

Q2: Another aspect that was considered was how well the codebase structured for replacement and expansion of new features. As the generation is divided into several sub-generators it would be easy to add more generators on to the total generation. Furthermore, the generators themselves are structured in a way that adding more content or modifying them is a straightforward process.

Implementing more complicated features that have to alter previously generated content is however one limiting factor. For example, a feature that was of interest was to create road entrances that lead into parking lots. Accessing and creating a junction in the previously generated road network introduces new data that intermediate generation steps have not previously considered. Because of this, any modification to a previous generation step would require re-generation of all intermediate steps, which is not an optimal solution.

Q3: When discussing the variety of the generated cities the group distinguished between the overarching structural variety and the variety in textures and shapes of the content. In terms of the topological structure or layout of the generated cities, the group is content with the variety offered. Each city has a unique look and varies in size and layout in a way that the group considers realistic enough. An example of this variety is shown in Figure 6.2.

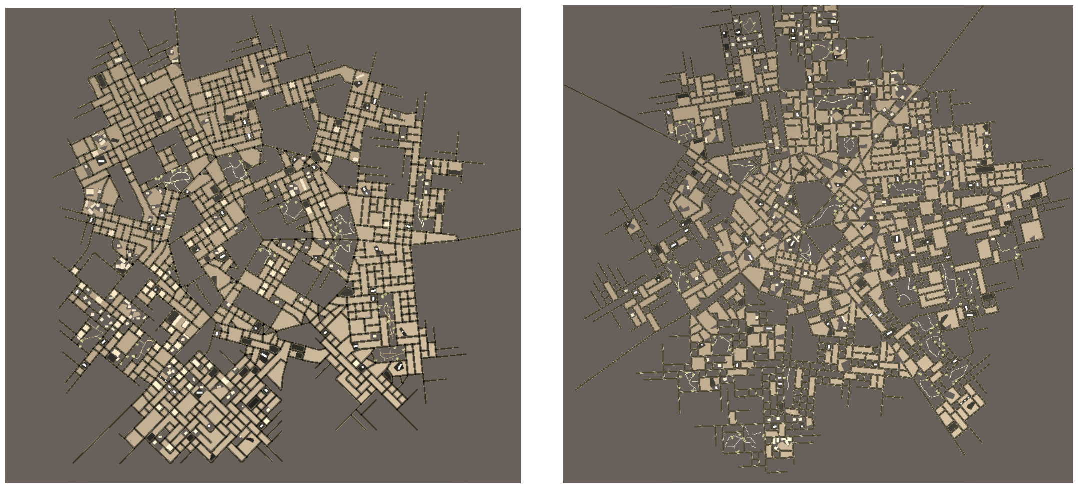


Figure 6.2: Two cities generated after consecutive runs under the same conditions highlighting the structural variety.

The variety in skyscrapers, buildings, and roads is limited though. For example, roads always have the same appearance in terms of their cross-section and texture. Buildings have some variety, but not close to those of reality. Fortunately, the generation of buildings is versatile enough for more variety to be easily added.

Q4: When discussing the real-world resemblance of the generated cities, mainly two aspects were considered: the structural layout of the cities and whether the generated cities are believable in a real-world setting. Concerning the first aspect, the project group considers the structural layout of the Manhattan- and Paris-style cities to resemble their real-world counterparts to an acceptable degree. This structural resemblance is demonstrated in Figure 5.8 and Figure 5.10.

Regarding the second aspect, the pathways between roads, parking lots, and parks were deemed impractical for human traversal due to their inaccessibility. For example, cars can not viably travel to and from the roads and parking lots without first crossing a patch of grass. Arguably such conditions are plausible in reality as well, but this is typically not the case. Several details like this one amount to a significant deviation between CityCraft's cities and real cities in terms of appearance. Thus, the project group does not consider the cities generated by CityCraft to be believable by real-world standards, and more work is needed to further close this gap.

Q5: The degree of control offered to the user is, with a few exceptions, at the level of supplying static settings to each sub-generator. The exception to this is that the user can dynamically place individual cities and re-generate the content for each sub-generator. This distinction highlights an important design choice of either creating a city editor with dynamic PCG, or a city generator capable of producing infinite worlds through static PCG. CityCraft, tries to be both in the sense that each sub-generator is essentially stateless, but still imposes some user choices in the placement of cities for example. Since the goal of CityCraft was to be a rough tool for generating cities that designers later could refine, the option for allowing the user to edit specific roads or alter the content of a selected block might not be deemed essential. However, it would significantly improve designers' level of control.

Q6: The cities generated by CityCraft are by no means mature enough for use in production quality digital media. Some limiting factors include the texture quality and the odd gaps between roads and their surrounding buildings, parks and parking lots. Although the generated cities might not be suitable as-is, CityCraft is still useful as a city prototyping tool as the cities can be generated in a short amount of time. The cities could also be used as backdrops, as templates for further refinement, and of course research.

Q7: When evaluating whether a user without any technical expertise would be able to correctly use the application, the project group concluded that such was the case. This was concluded with the reasoning that all implementation details are irrelevant to the user. Furthermore, the steps necessary to generate a city follow an easy step-by-step Wizard [98] that is familiar to most users. Admittedly, the application has not undergone any formal user testing to verify this. One area of usability improvement would be to improve the design and pliancy of the user interface to provide better feedback. For example, in the road generation step, the user might not notice that it is required to place a city marker to proceed.

Q8: There is one significant bug and one crash known that might occur when using CityCraft. The crash occurs under some unknown condition when placing city markers. This rarely happens but is considered the most critical issue as it completely obstructs the user from using the application. The bug is that the mesh generated for road intersections can sometimes become huge and cover a large part of the world. The cause for this is known, however, due to the time limit has not been addressed properly. In summary, the project group would consider CityCraft as stable given its development time.

6.2 Performance

One main takeaway from this project was that even though we believed our defined scope was reasonable, we did not take into account how much time one could spend improving parts that were already working. An area where this became particularly clear was performance, which was needed to perform the city generation in real-time. Throughout development, we would often find generators that produced visually sufficient content, but whose generation process was slow. The three main parts of the project where we observed that we could increase performance were the following:

- Compressing textures.
- Decreasing the triangle count of imported and generated models.
- Implementing Level of Detail (LOD).
- Combining meshes.

The first part was simple to tackle and did not require any additional code to be written, we simply altered the resolution at which our textures were being compressed. This issue itself was noticed when memory consumption suddenly started to increase at a rapid rate. Our solution, to drastically compress the textures, made runtime visuals degrade somewhat. However, full texture quality is still used when exporting the models, which was seen as more important.

The second part was identified by the fact that after generating buildings frame rate suddenly became a major concern. After inspecting this issue we found that the buildings were being composed of a lot more triangles than necessary and this was fixed by simply reducing the triangle count.

The third improvement of performance was using LOD. LOD works by decreasing the visual quality of an object based on some function of the camera distance to that object. Typically, different distance intervals are configured to correspond to meshes of varying visual quality. The further away the camera is, the lower the quality of the mesh becomes as to improve the efficiency of rendering without a noticeable difference to the user. In CityCraft, this technique was used extensively which significantly improved the performance.

The fourth and final major adjustment made for increasing performance was to combine some of our generated meshes. Combining meshes for the building generator, when using the L-system strategy, was crucial for lowering memory usage. The first version had a mesh for each wall segment, and each wall segment had a material with an accompanying texture. This lead to memory usage peaking over 10GB with a medium-sized city. Combining the wall segments into one building meant a drastic change in memory usage.

6.3 Process and Workflow

When dividing up the workload between group members, each generator described in Chapter 4.2 facilitated a natural division of work since an exchange of data between the generators had been established early on. This way of splitting up the workload between generators proved both beneficial and disadvantageous in some regard.

For one, each group member could initially regard the assigned generator as a simple, confined task that could be developed in isolation. This allowed group members to implement different parts of the software simultaneously without any major conflicts or bottlenecks. For example, the first iteration of the road generator only considered a flat terrain seeing that the terrain generator was not fully implemented yet.

The downside was that only one or two group members had a good understanding of how each particular sub-generator was implemented. Consequently, re-using already implemented code could have been more efficiently utilized had we chosen a more dynamic collaboration style. This became most obvious when working on the path for the parks. In one part of the project, code was being written for generating meshes for the park paths, while code for doing this had already been written and was being used in the road generation. Another problem that arose from this was that the parking lot generator was developed in isolation of roads, when they probably would have benefitted from sharing much more implementation.

6.4 Future Work

This subchapter highlights the possibilities of expanding CityCraft and the system it implements, mainly focusing on different ways to evolve the application beyond what we had time for. An obvious way to further develop CityCraft would be to simply add more aspects to the generators, such as additional road network strategies, buildings types, and objects inside parks. This would not only be a good way to stress the system, but would also make the generated cities further resemble reality.

As the terrain was never a core focus of CityCraft, it would be fitting to give users the option of providing their own terrain models, which the cities would adjust to. Users could also be given the ability to draw their own population map, as well as the ability to draw new roads during runtime. Furthermore, content such as road markings, road signs, and possibly bridges could also be added.

One way to further develop the building generation would be to simply include a larger set of textures, or to algorithmically generate textures in order to further increase visual variety. Another area of interest is to generate the interiors of buildings as well as adding public transport infrastructure. As for the parks, the generation of paths was implemented in a way that did not consider concave polygons. This could potentially be solved by either altering the algorithm for paths, or altering the way that plots are divided.

However, the primary goal of future work should be to validate the theoretical system that CityCraft implements. This could be done by comparing the algorithms and architecture proposed in this report with other approaches. For instance, Voronoi Diagrams, WaveFunctionCollapse, and Machine Learning are all potential options for such comparisons. By doing this, one could better identify the possible strengths and weaknesses of our proposed system. It will also be important to validate if the application GUI meets the user-friendly requirements by conducting user tests.

7

Conclusions

This project aimed to explore the usage of PCG algorithms to procedurally generate modern 3D cities. This work involved research on previous work and existing PCG techniques in order to design and propose a complete system capable of generating such cities. The system that we proposed was implemented and demonstrated in the form of a desktop application named CityCraft, which combined several techniques such as Agent-based generation, L-systems, and LOD to achieve performant city generation. With this implementation, we have demonstrated a suitable approach to city generation by combining PCG algorithms with theory from computer graphics.

Although far from commercial-ready production quality, the realization of CityCraft provides useful insight into various techniques applicable to city generation, how they can be combined, and also provides a step in the right direction for city generation research within the open-source domain. With glTF compatibility and no technical expertise required to operate, CityCraft can already find usage in the modeling industry. However, the software likely needs to mature before that is realistic, with the main limitations being texture quality and certain visual artifacts.

At its current state, CityCraft demonstrates the capability of generating all content included within the scope of this project, with no significant drawbacks or limitations identified. While further improvements could also include the generation of more content and fixing minor bugs, the primary concern of future research is to identify the limitations of the theoretical system that CityCraft implements. Our work demonstrates the potential of the proposed system, while future work needs to compare this system with other options to truly deem its quality. It would also be of interest to stress the system, by including more aspects to generate, such as bridges, interiors of buildings, and public transport infrastructure.

In summary, with this report we hope to bring insight into various methods within PCG and computer graphics suitable for the generation of modern 3D cities. We also propose a city generation system, utilizing a function-based architecture to orchestrate said methods, and release an implementation of it named CityCraft under the MIT license. With this contribution, we hope to assist the research of city generation within the open-source domain.

Bibliography

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Springer International Publishing, 2016, ISBN: 978-3-319-42716-4. DOI: 10.1007/978-3-319-42716-4.

[2] IDV, Inc. (2017). SpeedTree (homepage), [Online]. Available: https://store.speedtree.com (visited on 04/07/2020).

[3] Free-Photos. (Aug. 2014), [Online]. Available: https://pixabay.com/photos/manhattan-new-york-city-407703/ (visited on 04/08/2020).

[4] P. Henri. (Dec. 2018), [Online]. Available: https://pixabay.com/photos/eiffel-tower-paris-3862939/ (visited on 04/08/2020).

[5] Wikimedia Foundation, Inc. (Apr. 2020). Low poly, [Online]. Available: https://en.wikipedia.org/wiki/Low_poly (visited on 04/08/2020).

[6] Wikimedia Foundation, Inc. (Mar. 2020). Voxel, [Online]. Available: https://en.wikipedia.org/wiki/Voxel (visited on 04/08/2020).

[7] The Blender Foundation. (2020). Blender (homepage), [Online]. Available: https://www.blender.org/ (visited on 04/17/2020).

[8] S. Beca. (2017). Procedural Generation: a primer for game devs, [Online]. Available: https://www.gamasutra.com/blogs/ScottBeca/20170223/292255/Procedural_generation_a_primer_for_game_devs.php (visited on 04/07/2020).

[9] Y. Parish and P. Müller, "Procedural Modeling of Cities", vol. 2001, Aug. 2001, pp. 301–308. DOI: 10.1145/1185657.1185716.

[10] Esri. (2020). Esri CityEngine (homepage), [Online]. Available: https://www.esri.com/en-us/arcgis/products/esri-cityengine/overview (visited on 04/28/2020).

[11] G. Kelly and H. McCabe, "Citygen: An Interactive System for Procedural City Generation", 2007. [Online]. Available: http://www.citygen.net/files/citygen_gdtw07.pdf.

[12] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation", *Institute of Technology Blanchardstown Journal*, vol. 14, Jan. 2006. [Online]. Available: http://www.citygen.net/files/images/Procedural_City_Generation_Survey.pdf.

[13] Lechner et al., “Procedural City Modeling”, Jan. 2003. [Online]. Available: http://ccl.northwestern.edu/papers/ProceduralCityMod.pdf.

[14] Tobias Mansfield-Williams. (2015). Procedural City Generation, [Online]. Available: https://www.tmwhere.com/city_generation.html (visited on 04/28/2020).

[15] Robin. (Dec. 2015). Procedural modeling of cities, [Online]. Available: https://phiresky.github.io/procedural-cities/ (visited on 04/28/2020).

[16] M. Gumin. (Mar. 2020). WaveFunctionCollapse, [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse (visited on 04/29/2020).

[17] T. Møller and J. Billeskov, “Expanding Wave Function Collapse with Growing Grids for Procedural Content Generation.”, PhD thesis, May 2019. DOI: 10.13140/RG.2.2.23494.01607.

[18] Introversion Software. (Dec. 2007). Subversion Procedural Cities December 2007, [Online]. Available: https://www.youtube.com/watch?v=MG41yYBD-rE (visited on 04/28/2020).

[19] A. Couturier. (2009). SceneCity: City generator addon for Blender, [Online]. Available: https://www.cgchan.com/store/scenecity (visited on 04/29/2020).

[20] D. Santiago. (Apr. 2019). Procedurally Crafting Manhattan for ’Marvel’s Spider-Man’, [Online]. Available: https://www.gdcvault.com/play/1025765/Procedurally-Crafting-Manhattan-for-Marvel (visited on 04/29/2020).

[21] A. Yurkin and M. Rybalchenko. (Nov. 2017). Discover how Frogwares’ ’City Generator’ is saving valuable time during development of ’The Sinking City’, [Online]. Available: https://www.unrealengine.com/en-US/developer-interviews/discover-how-frogwares-city-generator-is-saving-valuable-time-during-development-of-the-sinking-city (visited on 04/29/2020).

[22] Valve. (2020). Browsing Roguelike, [Online]. Available: https://store.steampowered.com/tags/en/Roguelike/ (visited on 04/13/2020).

[23] Mojang. (2011). Minecraft (game), [Online]. Available: https://www.minecraft.net (visited on 02/11/2020).

[24] mowenmedia. (May 2015). Minecraft World Tree, [Online]. Available: https://pixabay.com/illustrations/minecraft-world-forest-773807/ (visited on 04/13/2020).

[25] RPGamer. (Feb. 2012). RPGamer Feature - The Elder Scrolls IV: Oblivion Interview with Gavin Carter, [Online]. Available: https://web.archive.org/web/20120207174858/http://www.rpgamer.com/games/elderscrolls/elder4/elder4interview.html (visited on 04/13/2020).

[26] Hello Games. (2016). No Man’s Sky (game), [Online]. Available: https://www.nomanssky.com/ (visited on 04/13/2020).

[27] 2K Games. (1991-2016). Sid Meier’s Civilization (game series), [Online]. Available: https://civilization.com/ (visited on 04/13/2020).

[28] Blizzard. (1996-2020). Diablo (game series), [Online]. Available: https://diablo4.blizzard.com/ (visited on 04/13/2020).

[29] Motion Twin. (2018). Dead Cells (game), [Online]. Available: https://dead-cells.com/ (visited on 04/13/2020).

[30] Re-Logic, *Terraria (game)*, 2011. [Online]. Available: https://terraria.org/ (visited on 04/13/2020).

[31] 2K Games. (2009-2019). Borderlands (game series), [Online]. Available: https://borderlands.com/en-US/ (visited on 04/13/2020).

[32] Maxis. (2008). Spore (game), [Online]. Available: https://www.ea.com/games/spore/spore (visited on 02/11/2020).

[33] GDC. (2017). Practical Procedural Generation For Everyone, [Online]. Available: https://www.youtube.com/watch?v=WumyfLEa6bU (visited on 04/29/2020).

[34] Valve. (2009). Left 4 Dead 2 (game), [Online]. Available: https://store.steampowered.com/app/550/Left_4_Dead_2/ (visited on 04/13/2020).

[35] Bay 12 Games. (2006). Dwarf Fortress (game), [Online]. Available: http://www.bay12games.com/dwarves/ (visited on 04/13/2020).

[36] Metacritic. (Feb. 2012). Minecraft, [Online]. Available: https://www.metacritic.com/game/pc/minecraft/critic-reviews?sort-by=date (visited on 04/13/2020).

[37] S. Persson. (May 2019). Celebrating 10 Years of Minecraft, [Online]. Available: https://news.xbox.com/en-us/2019/05/17/minecraft-ten-years/ (visited on 04/13/2020).

[38] Metacritic. (Mar. 2012). Terraria, [Online]. Available: https://www.metacritic.com/game/pc/terraria/critic-reviews?sort-by=date (visited on 04/13/2020).

[39] Loki. (Apr. 2020). Terraria Pushes Beyond 30 Million Copies Sold!, [Online]. Available: https://terraria.org/news/terraria-pushes-beyond-30-million-copies-sold (visited on 04/13/2020).

[40] K. MacDonald. (Jul. 2018). No Man's Sky developer Sean Murray: 'It was as bad as things can get', [Online]. Available: https://www.theguardian.com/games/2018/jul/20/no-mans-sky-next-hello-games-sean-murray-harassment-interview (visited on 04/13/2020).

[41] J. Porter. (Jun. 2019). No Man's Sky fans crowdfund a billboard message to thank developer for work, [Online]. Available: https://www.theverge.com/2019/6/18/18683405/no-mans-sky-billboard-hello-games-sean-murray-gofundme-thank-you-beyond (visited on 04/13/2020).

[42] Massive Software. (2017). About Massive, [Online]. Available: http://www.massivesoftware.com/about.html (visited on 04/07/2020).

[43] IDV, Inc. (2017). SpeedTree Cinema, [Online]. Available: https://store.speedtree.com/cinema/ (visited on 04/13/2020).

[44] K. Perlin, “An image synthesizer”, SIGGRAPH ’85, pp. 287–296, Jul. 1985. DOI: 10.1145/325334.325247.

[45] K. Perlin, “Improving Noise”, *ACM Trans. Graph.*, vol. 21, no. 3, pp. 681–682, Jul. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566636.

[46] K. Perlin. (2002). Noise Hardware, [Online]. Available: https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf (visited on 04/02/2020).

[47] K. Spencer. (Sep. 2014). Noise!, [Online]. Available: https://uniblock.tumblr.com/post/97868843242/noise (visited on 04/02/2020).

[48] K. Spencer. (Oct. 2014). OpenSimplex Noise in Java, [Online]. Available: https://gist.github.com/KdotJPG/b1270127455a94ac5d19 (visited on 04/02/2020).

[49] Kenneth Perlin, ”Standard for perlin noise”, U.S. patent US20020135590A1, issued March 15, 2005.

[50] S. Gustavson. (Mar. 2005). Simplex noise demystified, [Online]. Available: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf (visited on 04/02/2020).

[51] Matmuze. (Dec. 2011). File:Value noise 2D.png, [Online]. Available: https://en.wikipedia.org/wiki/File:Value_noise_2D.png (visited on 04/02/2020).

[52] Burgercat. (Apr. 2007). File:Perlin.png, [Online]. Available: https://en.wikipedia.org/wiki/File:Perlin.png (visited on 04/02/2020).

[53] Wikipedia, *Fractal*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Fractal (visited on 05/28/2020).

[54] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Berlin, Heidelberg: Springer-Verlag, 1990, ISBN: 0387972978.

[55] G. Stiny and J. Gips, “‘Shape Grammars and the Generative Specification of Painting and Sculpture’”, vol. 71, Jan. 1971, pp. 1460–1465.

[56] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997, ISBN: 9810228848.

[57] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, “Search-Based Procedural Content Generation”, in *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I*, ser. EvoApplicatons’10, Springer-Verlag, Apr. 2010, pp. 141–150, ISBN: 3642122388. DOI: 10.1007/978-3-642-12239-2_15.

[58] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, “Search-Based Procedural Content Generation: A Taxonomy and Survey”, *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, pp. 172–186, Oct. 2011. DOI: 10.1109/TCIAIG.2011.2148116.

[59] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II”, *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[60] Summerville et al, “Procedural Content Generation via Machine Learning (PCGML)”, Feb. 2017. [Online]. Available: https://arxiv.org/abs/1702.00539.

[61] Guzdial et al, “Explainable PCGML via Game Design Patterns”, Sep. 2018. [Online]. Available: https://arxiv.org/pdf/1809.09419.

[62] B. Rieder, “Using Procedural Content Generation via Machine Learning as a Game Mechanic”, Sep. 2018. [Online]. Available: https://static1.squarespace.com/static/559921a3e4b02c1d7480f8f4/t/5bc5afdf15fcc0ad8522a29c/1539682276053/Rieder+Bernhard_840.PDF.

[63] B. Ertl. (Feb. 2015). File:Euclidean Voronoi diagram.svg, [Online]. Available: https://en.wikipedia.org/wiki/File:Euclidean_Voronoi_diagram.svg.

[64] B. Ertl. (Feb. 2015). File:Manhattan Voronoi Diagram.svg, [Online]. Available: https://en.wikipedia.org/wiki/File:Manhattan_Voronoi_Diagram.svg.

[65] S. Lloyd, “Least squares quantization in PCM”, *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[66] “Fortune, S. A sweepline algorithm for Voronoi diagrams. Algorithmica 2, 153 (1987). https://doi.org/10.1007/BF01840357 ”,

[67] R. Bridson, “Fast Poisson Disk Sampling in Arbitrary Dimensions”, in *ACM SIGGRAPH 2007 Sketches*, ser. SIGGRAPH ’07, San Diego, California: Association for Computing Machinery, 2007, 22–es, ISBN: 9781450347266. DOI: 10.1145/1278780.1278807.

[68] M. Bostock. (Sep. 2019). Poisson-Disc II, [Online]. Available: https://bl.ocks.org/mbostock/dbb02448b0f93e4c82c3 (visited on 04/06/2020).

[69] Blomqvist et al., “Generating Compelling Procedural 3D Environments and Landscapes”, 2016. [Online]. Available: https://hdl.handle.net/20.500.12380/244588.

[70] Chrschn. (Mar. 2007). File:Dolphin triangle mesh.png, [Online]. Available: https://en.wikipedia.org/wiki/File:Dolphin_triangle_mesh.png (visited on 04/06/2020).

[71] Drummyfish. (Mar. 2019). File:Texture mapping demonstration animation.gif, [Online]. Available: https://en.wikipedia.org/wiki/File:Texture_mapping_demonstration_animation.gif (visited on 04/06/2020).

[72] Lightweight Java Game Library. (2020). Lightweight Java Game Library (LWJGL) (homepage), [Online]. Available: https://www.lwjgl.org/ (visited on 04/16/2020).

[73] jMonkeyEngine. (2020). JMonkeyEngine (JME) (homepage), [Online]. Available: https://jmonkeyengine.org/ (visited on 04/16/2020).

[74] Unity Technologies. (2020). Unity 3D Engine (homepage), [Online]. Available: https://unity.com/ (visited on 04/16/2020).

[75] Epic Games, Inc. (2020). Unreal Engine (homepage), [Online]. Available: `https://www.unrealengine.com/en-US/` (visited on 04/16/2020).

[76] Software Freedom Conservancy. (2020). Godot Engine (homepage), [Online]. Available: `https://godotengine.org/` (visited on 04/16/2020).

[77] The Khronos Group Inc. (Jan. 2020). UnityGLTF, [Online]. Available: `https://github.com/KhronosGroup/UnityGLTF` (visited on 04/16/2020).

[78] The Khronos Group Inc. (2020). glTF Overview, [Online]. Available: `https://www.khronos.org/gltf/` (visited on 04/16/2020).

[79] The Khronos Group Inc. (2020). About The Khronos Group, [Online]. Available: `https://www.khronos.org/about/` (visited on 04/17/2020).

[80] Unity Technologies. (2019). FBX Exporter, [Online]. Available: `https://docs.unity3d.com/Packages/com.unity.formats.fbx@2.0/manual/index.html` (visited on 04/16/2020).

[81] Autodesk Inc. (2020). FBX, [Online]. Available: `https://www.autodesk.com/products/fbx/overview` (visited on 04/16/2020).

[82] FileFormat.Info. (2006). Wavefront OBJ File Format Summary, [Online]. Available: `https://www.fileformat.info/format/wavefrontobj/egff.htm` (visited on 04/16/2020).

[83] FileFormat.Info. (2005). Alias/WaveFront Material (.mtl) File Format, [Online]. Available: `http://www.fileformat.info/format/material/` (visited on 04/16/2020).

[84] The Khronos Group Inc. (2020). COLLADA Overview, [Online]. Available: `https://www.khronos.org/collada/` (visited on 04/16/2020).

[85] E. Weisstein. (2005), [Online]. Available: `https://mathworld.wolfram.com/MoirePattern.html` (visited on 05/13/2020).

[86] Wikipedia, *R-tree*, 2020. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=R-tree&oldid=955552664` (visited on 05/13/2020).

[87] Free-Photos. (Sep. 2015), [Online]. Available: `https://pixabay.com/photos/san-francisco-cityscape-918903/` (visited on 05/13/2020).

[88] K. Mehlhorn and D. Michail, "Implementing Minimum Cycle Basis Algorithms", vol. 11, May 2005, pp. 32–43. DOI: `10.1007/11427186_5`.

[89] H. Petovan. (Apr. 2012). Algorithm 101 : Finding all polygons in an undirected graph, [Online]. Available: `https://blog.mrpetovan.com/web-development/algorithm-101-finding-all-polygons-in-an-undirected-graph/` (visited on 05/13/2020).

[90] H. Abelson and A. diSessa, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. The MIT Press, Jul. 1986, ISBN: 9780262362740. DOI: `10.7551/mitpress/6933.001.0001`.

[91] S. Khetarpal. (Dec. 2014). Dividing A Polygon In Any Given Number Of Equal Areas, [Online]. Available: `http://www.khetarpal.org/polygon-splitting/` (visited on 05/11/2020).

[92] J. Dormans. (Jul. 2016). A Handcrafted Feel: ‘Unexplored’ Explores Cyclic Dungeon Generation, [Online]. Available: https://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/ (visited on 04/21/2020).

[93] A. Schmit. (Feb. 2019). Parking? Lots! Car Spaces Would Comprise 10% of Phoenix, [Online]. Available: https://usa.streetsblog.org/2019/02/25/parking-lots-in-%09%09%20%09phoenix-10-percent-of-land-is-for-car-storage (visited on 05/11/2020).

[94] Agile Alliance. (2020). What is Agile Software Development, [Online]. Available: https://www.agilealliance.org/agile101/ (visited on 04/20/2020).

[95] Software Freedom Conservancy. (2020). Git (homepage), [Online]. Available: https://git-scm.com (visited on 04/20/2020).

[96] GitHub, inc. (2020). GitHub (homepage), [Online]. Available: https://github.com (visited on 04/20/2020).

[97] Google. (2020). Google Drive (homepage), [Online]. Available: https://www.google.com/drive/ (visited on 04/20/2020).

[98] J. Tidwell. (Jan. 2011). Wizard, [Online]. Available: https://designinginterfaces.com/patterns/wizard/ (visited on 05/14/2020).

[99] Learn From. Build More. (May 2011). Urban Intersection, [Online]. Available: https://www.flickr.com/photos/48321464@N05/5728771938 (visited on 05/13/2020).

[100] Unknown user (inactive account). (Oct. 2017). New York Manhattan USA semester, [Online]. Available: https://pixabay.com/sv/photos/new-york-manhattan-usa-semester-3301173/ (visited on 05/13/2020).