Informatics

Data-driven Generation of Virtual City Layouts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Media and Human-Centered Computing

eingereicht von

Felix Hochgruber, BSc
Matrikelnummer 01226656

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.Doz. Mag. Dr. Hannes Kaufmann
Mitwirkung: Mag. Dr. Peter Kán

Wien, 28. Februar 2020

Felix Hochgruber Hannes Kaufmann

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Data-driven Generation of Virtual City Layouts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media and Human-Centered Computing

by

Felix Hochgruber, BSc
Registration Number 01226656

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.Doz. Mag. Dr. Hannes Kaufmann
Assistance: Mag. Dr. Peter Kán

Vienna, 28th February, 2020

Felix Hochgruber	Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Felix Hochgruber, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. Februar 2020

Felix Hochgruber

Acknowledgements

I would like to express my gratitude to Peter Kán, who provided invaluable assistance and support throughout the whole process that resulted in this work. I also wish to thank Hannes Kaufmann for his supervision of the thesis.

Furthermore, I am grateful to all my friends for their continued support throughout my studies. In particular, I would like to thank my girlfriend for the comfort and moral support she provided.

I would also like to thank all the participants of the user study for their time and the input they provided. Lastly, I wish to thank my parents for making it possible for me to pursue my studies. Their unconditional help and encouragement enabled me to follow my interests.

Kurzfassung

Der Bereich der prozeduralen Generierung ist ein wachsendes Fachgebiet in der Informatik, das zunehmend Aufmerksamkeit erlangt. Durch die automatische oder teilautomatische Erstellung von digitalen Inhalten kann die Kreativität gefördert, neue Erfahrungen angeboten, und Entwicklungskosten reduziert werden. Außerdem können dadurch Szenen und Landschaften von theoretisch unendlichen Ausmaßen erzeugt werden. Die Generierung von virtuellen Umgebungen, speziell Städten und urbanen Gegenden, kann für Designer und Entwickler erheblichen Nutzen bringen. Dieser Prozess hat in verschiedensten Anwendungsbereichen Einsatz gefunden, unter anderem in Videospielen, in der Filmproduktion, und in Simulationen. Die meisten bestehenden Herangehensweisen behandeln die künstliche Synthese von Stadtlayouts und setzen nur bedingt reale Daten ein.

In dieser Diplomarbeit untersuchen wir die direkte Integration von existierenden Stadtlayouts in den prozeduralen Generierungsprozess. Eine Filterfunktion wird bereitgestellt mit der die benötigten Daten im richtigen Format bezogen werden können. Die vorgeschlagene Methode für die Generierung von Stadtlayouts simuliert die Anziehung von zwei oder mehreren Straßennetzwerken zueinander. Dadurch wird ein kombiniertes Stadtlayout erzeugt, das interpolierte Eigenschaften der gewählten Eingabedaten hat. Für die Implementierung wurde Unity verwendet, da für bestimmte Operationen auf die integrierte Physik-Engine zurückgegriffen wird. Während des Generierungsprozesses ist die Interaktion durch die Benutzerin oder den Benutzer möglich, wodurch Strukturen nach Wunsch angepasst werden können. Die Performance der Applikation ist vergleichbar mit existierenden simulations- und agentenbasierten Herangehensweisen, wenn mittelgroße Gebiete verwendet werden. Eine Nutzerstudie hat bestätigt, dass die generierten Layouts erkennbare Charakteristika der Ausgangsdaten aufweisen.

Abstract

Procedural Content Generation (PCG) is a growing field in computer science that is progressively gaining attention. Through the automated or semi-automated creation of digital content, creativity can be increased, new experiences can be offered, and development costs can be reduced. Furthermore, it makes the fabrication of scenes and landscapes with theoretically infinite dimensions possible. The generation of virtual environments, particularly cities and urban areas, can provide numerous advantages to designers and developers. It has seen adoption in various domains, including, but not limited to, video games, movie production, and social simulations. Most existing city generation approaches investigate the artificial generation of new layouts, making only limited use of real-life data.

In this thesis, we explore the direct integration of existing city layout information into the generation process. A filter function is provided to obtain the required data in the correct format. The proposed method for city layout generation simulates the attraction and adaptation of two or more urban road networks to each other. As a result, a combined layout containing interpolated features of the selected input files is generated. The implementation was done in Unity, as we rely on the internal physics engine for certain operations. User interaction is possible during the generation process, allowing structures to be modified as desired. The performance is comparable to existing simulation- and agent-based approaches for medium-sized areas. A user study confirms that the generated layouts include recognizable characteristics of the input data.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The adoption of Procedural Content Generation (PCG) is becoming increasingly common in computer graphics. Thanks to more powerful processors and advances in 3D development, applications are now able to integrate 3D scenes of almost infinitely large sizes. Manual creation of content on this scale has thus become unfeasible, which is why automated and semi-automated methods are becoming increasingly important. Procedural systems can provide designers with starting points for content creation, which reduce the time spent on potentially repetitive tasks. In certain instances, they can even replace human creative professionals altogether. Consequently, the adoption of PCG is often motivated by possible cost savings [TYSB11]. Furthermore, procedurally generated content can supply creative assistance [Smi15]. Through the use of algorithmic methods, novel ideas may be sparked in designers. New approaches to creative tasks may arise, which are unexpected and fundamentally different from what a human would come up with.

Procedural generation methods can be applied both during the design phase, as well as during live operation. It is possible for applications to create new content as it is required. This approach can be utilized, for example, to generate seemingly infinite environments [GPSL03, HMVDVI13] or to adapt to a user's preferences and needs [HYWL18].

Procedural systems can be used to create a multitude of content types. Examples for data types that can be generated using PCG methods range from 2D visualizations and textures [Per85, KK96] to three-dimensional plant models [PL90, LYC+10] and landscapes [MKM89, DP10]. In the context of video games, quests and scenarios have been generated [Dor10], as well as complete environments [Kaz09]. Furthermore, building models have also been created procedurally [WWSR03, MWH+06, MZWVG07].

This work focuses on the procedural generation of city layouts. Cities are complex constructs of form and function, often resulting from long periods of growth and development. As such, they are frequently studied from several different perspectives. For instance, the location and size of urban areas have been investigated from a geographical point-of-view [Joh72], whereas social sciences have studied the behavior of the population [Ley83].

Procedural city modeling is an especially important field of research, due to the many application areas including game design, urban planning, movie production, and simulation applications. The entertainment industry, in particular, has a high demand for automated city generation. In many video games, cities are required as environments for players to explore. With consumers demanding increasingly large areas, developers have to spend more resources on level design. Similarly, the interest in virtual environments for movie production is rising as well. Due to advances in rendering and special effects, the use of 3D models has become a more and more feasible alternative to expensive on-site shooting. In simulation applications, procedurally generated cities can be used as virtual testbeds. Novel environments can provide challenges for emerging technologies such as self-driving cars [KKC18].

1.2 Problem Statement

Many different factors influence the development of cities, such as geology, vegetation, culture, and neighborhood [Joh72, Lyn60]. To generate a completely *realistic* city, all of these have to be considered. In reality, however, it is extremely difficult, if not impossible, to integrate all real-world factors into a procedural generation system. Therefore, most researchers focus on the creation of *plausible* cities, which means that the generated features should be within certain limits of the properties of real cities [KKC18]. Through the inclusion of existing city data in a procedural generation system, some of the influencing factors that led to a city's development can be considered and incorporated in the result.

Many procedural modeling techniques are based on grammars and parameters, that are either specified by hand or generated automatically [MWH$^+$06, WWSR03, PM01]. Grammar-based methods generally consist of a set of rules, according to which a city is generated. While these approaches can produce plausible results, they do not provide the possibility to easily integrate existing city layout information. Therefore, to generate a layout that has properties similar to a specific city, a manual adaptation of a rule- or parameter set is necessary. This often results in a significant amount of work for the user. When the goal is to generate a city layout that has properties of multiple existing cities, defining those rules will get increasingly difficult, and may not always be possible.

While some approaches exist that focus on the integration of existing data, these mostly employ parametric approaches [AVB08, NGDA16]. When city layout information is consolidated into a reduced set of parameters, loss of data is inevitable. Therefore, such approaches may be able to produce city layouts that apply the overall road network pattern (grid, radial, etc.) of the underlying source data, but don't contain any existing structures.

1.2.1 Key Research Questions

Based on the above problem definition, the following research questions can be defined:

How can existing city layouts be used in procedural city generation?

The aim of the thesis is to implement a novel method for generating city layouts based on existing information. This data-driven approach uses publicly available layout information of real cities and combines it to generate a derived city layout. Since methods for generating urban 3D geometry for a given street layout already exist [MWH+06, Mae17], this work focuses primarily on the generation of road networks. An application is developed as a proof-of-concept for the devised algorithm.

How does data-driven procedural city generation compare to other approaches, based on subjective perception?

The devised algorithm is evaluated by performing comparisons with more established techniques for procedural city generation. Furthermore, the quality of the result is judged by performing subjective assessments of the output data generated by the algorithm.

1.3 Approach

An exemplary algorithm has been devised to generate city layouts using existing layout information. The algorithm employs a non-parametric approach. No explicit consolidation of layout and pattern information is performed, so as to preserve as much data as possible. This information is provided in the form of publicly available road network data of real cities. Structures that exist in these cities are combined in a sensible manner to produce a resulting layout. Particular focus has been put on the recognizability of city features: The resulting layout contains identifiable properties from the input maps.

Two or more city layouts are used as input for the algorithm, along with a set of parameters that specify a mixing ratio for each input data set. Based on these ratios, information from the respective source maps is integrated into the result. A high value causes more of the specific city layout to be preserved, while a low value causes structures to be discarded more readily. This gives the user a certain level of control over the layout generation process. Furthermore, it allows different output layouts to be produced for the same input data sets.

The resulting road networks contain structures that are present in the input data sets, but they are adapted and interpolated to produce a new layout. Apart from the mixing ratios, this adaption process also incorporates the importance of road segments. Large roads, such as highways or streets with multiple lanes, are considered to be of higher priority. Thus, these structures are generally preserved in the resulting layout.

The presented algorithm is non-deterministic. At certain points, pseudo-random values are integrated that affect the behavior. As a result, different outputs are likely to be generated by consecutive runs, even if the input parameters remain unchanged. The algorithm can further be classified as simulation-based. Attractive forces are simulated in the different input layouts, causing the road networks to adapt to each other and eventually fuse into one. Ultimately, this process should produce a valid and plausible city layout that is a combination of the provided data sets. During the simulation, it is possible for users to interact with the individual components. A user can provide input to directly influence the positions of roads. These manipulations are incorporated into the behavior.

To present the proposed algorithm, a prototypical application was developed. It serves as a framework for controlling user input, visualization, and interaction. The output data produced by the application is evaluated through a user study with a focus on subjective preference. The plausibility of the result data is further compared with more established methods for procedural city generation.

The utilization of existing layouts is motivated by the complexity of city development. A multitude of circumstances has taken influence on each city's evolution, which caused the structures to form its present shape. By incorporating the complete layout of cities, these factors are considered implicitly, to a certain extent. The proposed algorithm should, therefore, present a viable method for creating a new city layout based on the structures that have formed in others.

CHAPTER 2

Related Work

2.1 Urban Layouts

In order to develop methods for procedurally generating city street maps, it is first necessary to understand how urban layouts are composed. The rules by which settlements form and expand are many, though one of the most important factors is the natural environment around them. It consists of the geology, the landform, water features, and plant and animal communities [Kro17]. These properties enable and restrict urban development in specific areas.

An especially important factor is the availability of water. It not only serves as necessary nourishment for the population but also enables trade through shipping. This encourages settlements to form directly at, or in close vicinity to rivers, large lakes, or oceans. Additionally, the climate also has a considerable impact on the types of structures that are built.

Technology is another important factor in the development of urban areas. The location of older cities from periods before the invention of the automobile and other long-distance transportation methods was heavily influenced by the location of other settlements in their vicinity [Joh72]. Furthermore, the maximum size of a city was also limited by the time it took individuals to travel from their homes to the center. Transport methods like streetcars and automobiles, but also rapid-transport systems like subways in more recent times, expedited inner-city travel. This allowed people to comfortably travel longer distances inside a city, thus making residences in outer regions more attractive.

2.1.1 Street Patterns

There are typically a number of different layout types that can be found in urban areas. Categorization has been attempted by several researchers [Lyn84, SO93, Mar04]. However, due to different classification systems and inaccuracy in descriptions, the terminology is often vague [Mar04]. For example, a radial-type street layout may be interpreted differently by different individuals, as there is no exact specification. This, in turn, can lead to divergent classifications. An example of an attempt to organize street layouts can be seen in Figure 2.1.

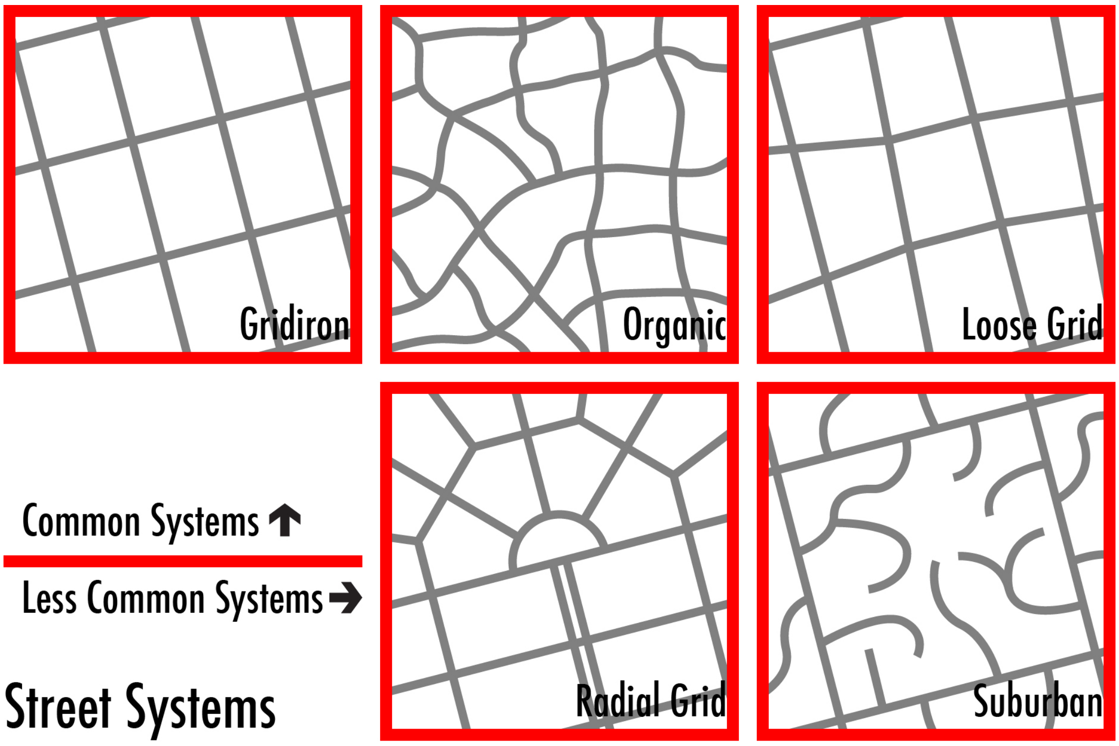


Figure 2.1: Different patterns found in urban road networks. [Mun13]

Some of the more commonly-found categories in research include grid, radial, and organic or irregular patterns [Mar04]. Nevertheless, further breakdowns of these are common, in attempts to differentiate more subtle distinctions. Grid-like patterns typically consist of two sets of parallel streets forming right angles between them. The roads follow a checkerboard pattern, forming four-way intersections at the crossing points. More relaxed versions of grids are also commonly found, featuring less-strict angles and possibly curved streets. Examples of grid patterns can be found in many American cities, most notably New York City.

In a radial layout, streets radiate outwards from points-of-interest, such as markets, churches, or palaces. These points are often encircled by multiple rings of roads acting as interconnections. The pattern is usually kept strictly geometric to further highlight the importance of its center. As such, appearances can be found in many European cities with structures resembling authority and power. An example of a city laid out according to a radial grid is Versailles, France, where the palace is a center point for multiple avenues spanning across the city. However, with increasing distance from the palace, the pattern quickly gives way to a loose grid layout.

In contrast to the planned nature of radial and grid patterns, organic layouts are very irregular, following no apparent rules. Streets can cross or branch at random positions, and curve sporadically. They are frequently encountered in historic centers of cities that have grown organically, but also in slums with fast-paced development [Mun13]. Instead of following a large-scale blueprint, organic layouts are typically planned on a per-building or per-parcel basis, giving a seemingly unplanned appearance [Kos91]. Planned suburban areas, as they are often found close to large cities in the United States of America, often try to mimic organic patterns by placing streets on tree-like structures with random curves.

In reality, cities do not only feature a single street pattern across the whole area they cover. Layouts frequently differ in areas based on factors such as age, zoning, population density, distance to the center, planning regulations, or cultural influences. In transition zones, patterns often mix, creating layouts that feature characteristics of both bordering sections. For example, a loose grid layout might be found between a historic city center laid out irregularly, and a planned commercial district with a gridiron structure.

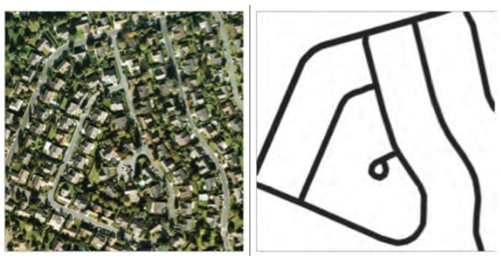
On a smaller level, 27 individual street patterns have been observed to make up many metropolitan areas worldwide [Whe15]. Wheeler argues that the presented landscape types can be used globally as a typology for urban regions. Nevertheless, many of the 27 landscape types can be split up into subtypes with slightly different features. The most frequently occurring patterns, in descending order, are “loops & lollipops”, “degenerate grids”, “rural sprawl”, and “workplace boxes”. Visual examples of these can be found in Figure 2.2.

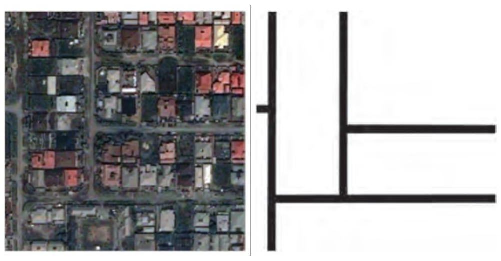
2.1.2 Data Representation

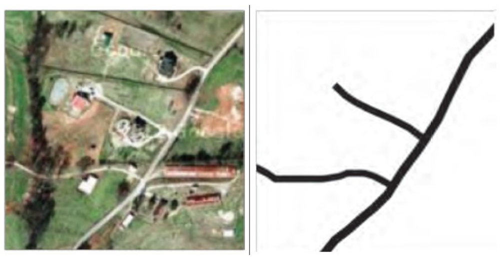
To be able to work with city layout information, the data needs to be available in a digital format. It allows the data to be stored, processed, and shared easily.

A simple representation of a road network can be achieved with a *link/node* approach [Goo00]. *Links* describe the streets themselves. They are made up of a start- and endpoint and possibly carry other attributes. *Nodes* are junctions between streets. Due to their interconnectivity, streets and roads form a *network* [LB14].

To store this information digitally, a data structure representing a planar graph can be used. This makes path-finding algorithms efficient and straightforward to implement. If required, the lengths of street segments can easily be implemented as weights on the edges to make distance calculations possible. However, this simple approach does not support curved roads explicitly. By splitting streets into multiple segments connected through nodes, curves can easily be approximated. Furthermore, Louf and Barthelemy argue that in addition to the adjacency matrix, important geometry information is encoded in the spatial distribution of nodes [LB14]. To accurately represent a city layout, this data, therefore, needs to be stored as well.


(a) Loops & lollipops


(b) Degenerate grid


(c) Rural sprawl

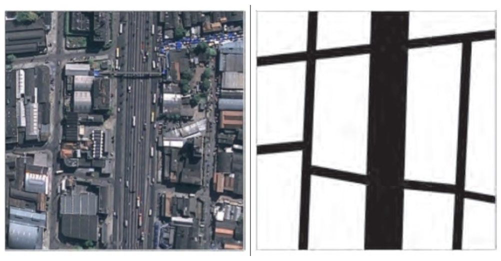

(d) Workplace boxes

Figure 2.2: Examples of the most commonly found landscape types as identified by Wheeler (in descending order). [Whe15]

In more complex scenarios, however, other solutions or extensions to a planar-graph data structure are required. For applications such as real-time navigation, information about lanes, possible transitions, and places like parking lots are necessary [Goo00]. Solutions for these scenarios have been presented [Wat99], but are out of the scope for this work.

2.2 Procedural Content Generation

The term *Procedural Content Generation (PCG)* describes the more or less automated creation of content using algorithms. As the exact boundaries of this concept are somewhat fuzzy, Togelius et al. have defined it as “the algorithmical creation of game content with limited or indirect user input” [TKSY11]. It makes it possible to create different kinds of data, that would otherwise be produced manually, in an automated or semi-automated manner. In general, at least a minimal amount of user input is necessary for PCG [TKSY11]. In some cases, the only required action may be triggering the start of the procedural generation system. In others, the system may depend on more elaborate user input. Therefore, PCG allows the process of creating content to be fully or partially automated, facilitating the task for humans and increasing the amount of data that can be produced. As procedural generation techniques enable the creation of lots of content quickly, less time and resources are often required. For this reason, it can potentially result in reduced development costs [TYSB11].

In most cases, a certain degree of randomness is included in the generation process. This allows procedural generation methods to produce variations in the results. However, deterministic techniques also exist that create the same content each time.

Even though the definition by Togelius et al. specifically mentions game content, PCG is not limited to this domain. Content that was produced by procedural generation methods can be used in a variety of different applications, such as computer games, simulations, visualizations, etc. Moreover, the type of content that can be created is also very diverse. In computer games, this may include maps, characters, or quests, while in visualization applications it could include patterns or style transfers.

2.2.1 Overview

The origins of procedural generation can be traced back to analog board games in the 18th and 19th centuries [BS20]. Movable terrain tiles could be used to configure war game boards differently for each new game. One of the first widespread appearances of digital PCG can be found in the video games *Rogue* and *Elite*. Both of these integrated procedurally generated worlds for players to explore: *Rogue* featured procedurally-generated dungeons, resulting in theoretically infinite unexplored environments. *Elite* relied on PCG to generate vast galaxies that would otherwise not have been possible due to the limited amount of memory available at the time.

Since then, many other games have integrated PCG as a more or less central gameplay element. It is used to create different types of game content, including terrain, maps, quests, and characters [TYSB11]. Games like *Minecraft* and *Civilization* rely on procedurally generated maps and territories, while *Diablo II* or *Left4Dead* utilize PCG to create random scenarios for the player [HMVDVI13].

A common type of content that is often generated procedurally is vegetation. Modeling individual plants and trees is a very time-intensive process, which is why many game studios and movie productions rely on procedural systems to create them. A widely-used commercial tool for this task is *SpeedTree*[1].

Strictly speaking, Artificial Intelligence (AI) in video games can also be seen as procedural content. The behavior of the agents follows a predefined procedure, depending on the current goals and environmental conditions. However, their actions and animations often contain some kind of randomness. For example, Non-Player Characters (NPCs) in the life simulation video game series *The Sims* show a randomly-selected reaction from a predefined set when a particular event occurs.

Video games are still the most common application of PCG to date. Nevertheless, procedural generation has seen adoption in many other fields of research as well [KKC18]. For instance, application areas include the movie industry, urban planning, social and environmental simulations, or machine learning. Another common use case for PCG is the generation of cities and urban environments for different domains, which will be covered in more detail in section 2.2.3.

[1]`https://store.speedtree.com` (Accessed: 17.02.2020)

There are many different reasons for utilizing procedural generation methods in games. Some of the more practical motivations include circumventing hardware limitations, as it was done for *Elite*, and saving time and resources on design- and development processes. Another motive is increased replayability of games with procedurally generated environments. These can ensure that the player is faced with different conditions, challenges, and opportunities for each playthrough. Similarly, PCG can produce adaptive content that is tailored to a player's current or overall status or skill. Procedural systems can also assist designers creatively in their work, by providing inspiration and alternative viewpoints in content creation. [TYSB11, STN16]

2.2.2 Methods

Many different forms of procedural generation have been presented over the last decades, and differentiating between their underlying methods can be difficult. Shaker et al. characterize them by drawing the following distinctions [STN16]:

- Online vs. offline
- Necessary vs. optional content
- Degree and dimensions of control
- Generic vs. adaptive
- Stochastic vs. deterministic
- Constructive vs. generate-and-test
- Automatic generation vs. mixed authorship

This taxonomy is not meant to categorize procedural generation methods by strictly placing them in one of the extreme points. Instead, for each dimension, they generally lie somewhere in between.

From a methodical perspective, a few approaches have been widely adopted. The most common of these will be presented in this section.

Fractals and Noise

Fractals are some of the most basic forms of procedural generation and are often used as a basis for terrain generation. Many physical processes found in nature show some form of self-similarity, that can be described using fractals [Pen84]. Mandelbrot has shown that when enlarging smaller natural phenomena, the same kinds of variations can be observed as on a larger scale [Man83]. This is especially visible in terrain: Mountain ranges and plains show variation in the elevation at the largest scale, but when zooming in, individual mountains and valleys show similar properties. Zooming in even more, there are often many smaller peaks on a single mountain. *Fractal terrain* intends to mimic this behavior to generate natural-looking landscapes. Figure 2.3 shows an example of how a landscape generated using fractal methods could look like.

Several different technical methods exist that make use of fractals in procedural generation [MKM89]. The *diamond-square* algorithm is one of the most widely used techniques, due to its computational efficiency and simple implementation [STN16].

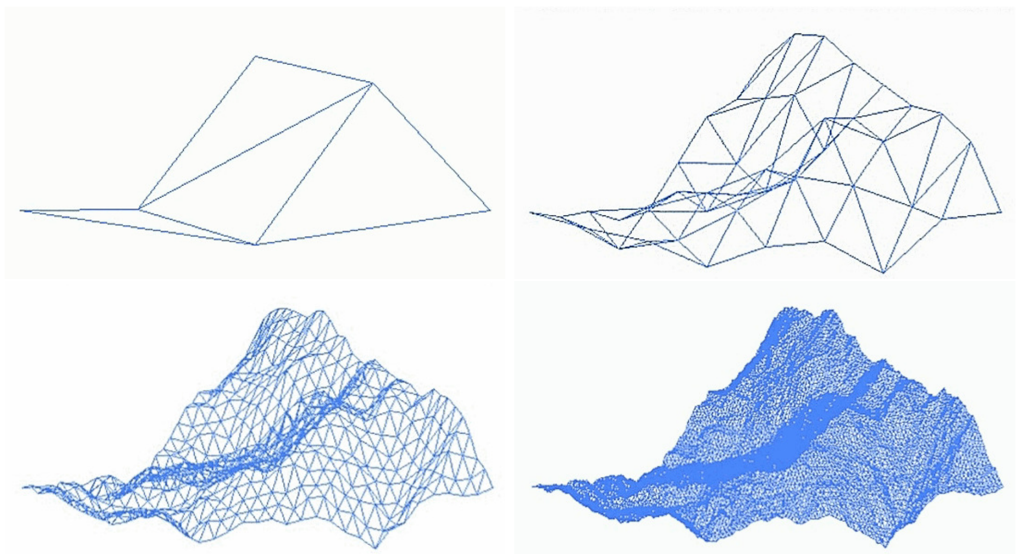


Figure 2.3: Generation of a procedural landscape using a fractal-based approach, in different detail levels. [dC11]

Noise is another common method for procedural generation and is frequently used to create synthesized textures and landscapes. It is often related to fractal-based methods, in that noise functions with different frequencies can be layered to produce a similar effect as fractals. One of the most commonly used techniques is *Perlin noise*, which is a type of gradient noise originally developed for image synthesis [Per85]. Instead of generating the height values for each point directly, only the slopes are generated. The height values are then inferred from those gradients, resulting in very smooth transitions. Another important noise function for PCG is *Simplex noise*. It is based on the same principles as Perlin noise, but has lower computational complexity and produces less directional artifacts [Per01].

Search-based Approaches

Search-based methods for PCG make use of evolutionary algorithms and other search/optimization techniques to generate content. Similar to genetic programming, the idea behind search-based content generation is that several possible results are randomly created in the beginning, and evaluated in each iteration. By dropping the worst candidates and adding random mutations to copies of the best candidates, a good enough solution should eventually be reached [TYSB11].

When creating a search-based content generation method, various challenges have to be overcome. One of these is the definition of a data format to represent the content that should be generated, as this defines and limits the content that can be produced [STN16]. Another is crafting an evaluation function that reliably and accurately maps generated content to a scalar that reflects its suitability. A common obstacle with that is that the designer first needs to formalize what is actually desired, which can be very difficult depending on the properties that are expected from the content [TYSB11].

Grammar-based Approaches

In formal language theory, a *grammar* is a set of rules for rewriting strings. These *production rules* specify exactly how strings are transformed, and which symbols get replaced. Beginning from a *start symbol*, the rules are applied repeatedly, until the resulting string contains no occurrence of the start symbol and no *nonterminal symbol*.

L-systems are a special class of grammars frequently used for procedural generation. They were introduced by the biologist Aristid Lindenmayer as a means to simulate the growth of different plants and other organisms [Lin68]. The distinctive feature of L-systems is *parallel rewriting*, meaning the parallel application of rules, as opposed to *sequential rewriting*, where rules are applied to the string from left to right. L-systems in their simplest form are deterministic, so the result is always the same for each generation process. However, variations also exist where multiple rules can match a given symbol, and one is chosen based on probability. The system is then referred to as a *stochastic L-system* [PL90].

Rules can easily be defined to create 2D and 3D visualizations by utilizing *turtle graphics*. Each symbol in the string is interpreted as a command (*draw* or *turn*) that is applied to a drawing area. This allows the generation of simple graphics, but also complex 3D models. A more elaborate example of a tree model generated using an L-system is shown in Figure 2.4.

Apart from procedurally generating vegetation, grammar-based approaches are also used for other types of content. Wonka et al. [WWSR03], as well as Müller et al. [MWH+06], have presented methods for generating complex 3-dimensional building models using novel types of grammars. Other procedural generation techniques use grammar-based approaches for animation [IFPW10], park layouts [Vas18], or quests in video games [Dor10]. A common difficulty when following a grammar-based approach for procedural generation is the definition of the rules. In general, these have to be created and adjusted manually by the developer. To achieve the desired outcome, a trial-and-error approach is often necessary, requiring a large amount of work.

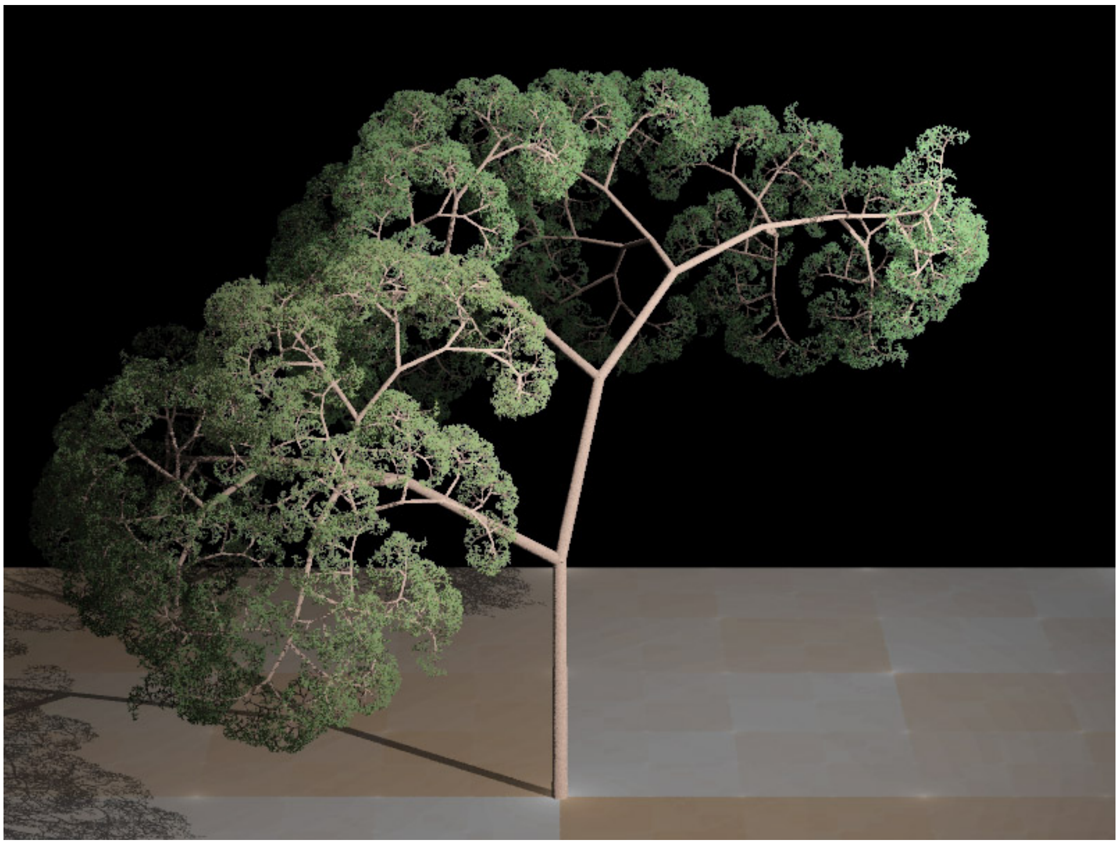


Figure 2.4: A tree model generated using an L-system. [Wik18]

Agent-based Approaches

Agent-based approaches to PCG make use of software agents to generate content. This technique is mostly used to generate environments. One or more agents move through a map and modify different properties according to their programmed behavior. In addition to this predefined heuristic, agents often include randomness in their behavior as well, making them useful for stochastic PCG. Doran and Parberry [DP10] have presented a method to procedurally generate terrain based on this approach. They make use of different types of agents, with many instances running concurrently on the same map. The agent's behaviors vary between types, but their only possible actions are reading height values and changing the heights. This enables the agents to define the coastline, form the land, and erode the terrain. The game *Spelunky* employs a similar technique to procedurally place obstacles and enemies in its levels [Kaz09, STN16]. The maps are based on a two-dimensional grid of rooms, with possible interconnections. After the basic layout is created, a software agent moves through the rooms and places different kinds of traps and monsters.

Other Approaches

Many other approaches to PCG exist, including variations of the methods described in this chapter. To organize the information, surveys have been published on this topic by Togelius et al. [TYSB11], Hendrikx et al. [HMVDVI13], and Smelik et al. [STBB14].

2.2.3 Procedural City Generation

The generation of cities, urban areas, and road networks is a common use case of PCG. Due to the complexity of these structures, the difference in resource requirements between models created by a human and those created by employing procedural generation techniques is especially high. This section will provide an overview of the current state-of-the-art in procedural city generation.

Many applications exist that integrate procedurally-generated cities. The entertainment industry, in particular, makes much use of them. In video games, generated cities can be used as environments for players to explore. In movie production, cities can be generated to match the desired style, and then used to fill backgrounds or define the setting in individual scenes. Apart from these, urban areas created procedurally are also used for simulations. They are valuable for research focusing on both social and urban-planning aspects. Emerging new technologies such as self-driving cars can make use of them as virtual testbeds, allowing the systems to be trained and evaluated in synthetic environments [KKC18].

Complete cities are typically generated in different stages [KM06]. The first stage usually consists of the road network generation. It provides a characteristic basis for the whole city and defines its general structure. Based on that, lots and parcels are created by filling in the space between roads. Buildings are then placed inside the parcels, which also includes the generation of their structures as well as their façades.

Because each of these stages is expected to produce a different kind of result, they are often viewed as individual problems. Consequently, the techniques that are employed are also different. Since the focus of this work lies in city layout generation, the creation of procedural buildings and façades will be kept short.

Road Networks

To generate road networks for cities, Parish and Müller have presented an approach based on L-systems [PM01]. The presented method includes two extensions to L-systems: *global goals* and *local constraints*. Global goals define large-scale targets for road connectivity, such as population density and conformity to the desired road pattern. Local constraints, in turn, verify that generated road sections are valid within the environment and try to adjust them otherwise. An example of such a constraint is a check for the underlying terrain, e.g., to test if the road does not end in water. With this extended system, the rise in complexity in production rules when generating detailed city layouts becomes more manageable compared to a strictly L-system-based method. Furthermore, new goals and constraints can easily be added without requiring a change in the rules.

Another grammar-based method has been presented by Sun et al. [SYBG04]. Their approach is template-based, allowing the user to select one of several templates as input for the system: a raster or radial template, a mixed template, or a population-based template visualized as a Voronoi diagram. Utilizing a set of rules and parameters from the selected template, an initial highway network is generated that roughly follows the same pattern. The generated roads are then validated and adjusted according to elevation- and population density maps. The regions between these larger roads are filled with smaller streets, which are created along a grid. While these approaches feature substantial improvements to standard L-systems, they still require the manual creation of complicated rules if specific outcomes are desired.

Kelly and McCabe have presented an interactive system to generate cities, called *Citygen* [KM07]. It allows the user to place and manipulate nodes directly on the terrain, which form the basis for the primary road network. These nodes are then connected via a sampling algorithm that adapts the roads to the environment and enforces various constraints. The cells formed by primary roads are filled with secondary roads that are generated by an algorithm based on L-systems. The production rules are predefined to generate one of three different road patterns: Raster, a relatively regular grid; Industrial, a less regular grid with dead ends; and Organic, a more random and natural-looking pattern.

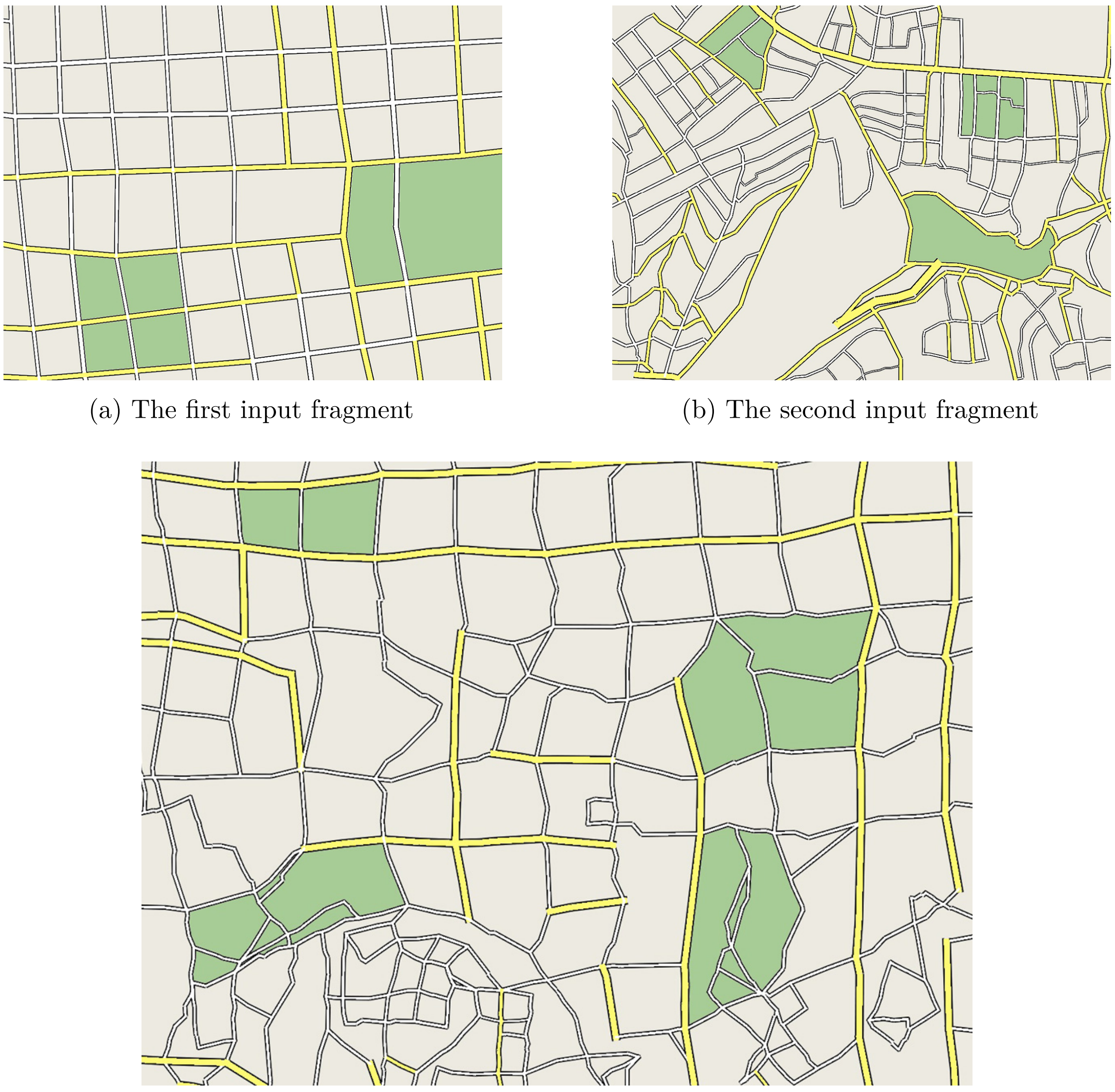
Lechner et al. have introduced an agent-based system to create artificial city layouts [LWWF03]. Two different types of agents are employed to generate the road network. Extender agents wander around the terrain and search for land that is not yet connected to the road network. If such a piece of land is found, the agent then tries to connect it by laying a road to the existing network. Connector agents operate on the already generated parts of the city, creating interconnections between existing roads. This system was later extended by a type of agent that is in charge of creating large primary roads and another that generates different types of parcels [LWW$^+$07]. In addition, the user can adjust different parameters as well as provide direct input by drawing different types of development zones on the city map. The results generated by the presented system are reasonably realistic, though the running time is quite long.

A simulation-based method has been presented by Vanegas et al. [VABW09]. The system requires user-drawn sketches of highways, population density, and employment areas as input, according to which urban areas are generated. The growth of smaller streets is governed by a clustering algorithm to generate seeds, combined with a breadth-first expansion method to place new road segments. Any changes in the input data are immediately taken into account and the corresponding structures are created, making it an interactive process. However, user control is limited to altering the sketches, as modifications to individual details are not possible. The approach presented by Weber et al. uses a similar technique to simulate the growth of cities over time [WMWG09]. Using an existing urban layout as input, the city expands into the surrounding land, taking environmental parameters such as terrain and land type into account. During the simulation, the user can still control different parameters to alter the city's growth. Another simulation-based approach has been presented by Beneš et al. [BWK14]. From individual settlements, new major roads grow to eventually form a combined city. A simple traffic simulation is applied that controls this growth by taking the neighborhoods into account.

While these approaches focus on generating more or less artificial road networks, research has also been conducted on how existing city layouts can be modified procedurally. With the goal to render an animation of a city's development over time, Krecklau et al. have proposed a method for temporal interpolation of city maps [KMK12]. The interpolation is based on a graph that defines interdependencies between events (e.g., starting the construction process of a building), which serve as preconditions.

A PCG system presented by Aliaga et al. employs an example-based approach to generate road networks [AVB08]. Utilizing existing data from public Geographic Information Systems (GISs), a parameterized model of a city is created. This model can then be interactively reconfigured by performing operations such as moving tiles, copy-pasting buildings, or stretching parts of the layout. Furthermore, it is possible to blend two datasets from different cities at defined intersection points. An example of this merging process can be found in Figure 2.5. These changes are also applied to a satellite image representation of the city model. However, the goals pursued by Aliaga et al. are very different from the results that we are looking to achieve with our system. Firstly, their method makes it possible to blend the styles of two cities, but not their complete street networks. Only a set of parameters extracted from the source datasets is used to create a combination. This means that information will inevitably get lost (see Section 2.2.4), and existing structures in either city will not be preserved. Secondly, the system implements blending mostly to generate transitions between two different styles of urban areas. It can create a road network with a spatially varying style, exhibiting properties of both source layouts mixed to form a smooth gradient. The blending operation is thus not performed over the whole city area, but only between user-defined intersection points. In contrast, our system employs a non-parametric method to include the source data directly in the generation process. Structures and patterns from the input datasets are therefore still included in the output. Furthermore, our system blends two different cities over their whole area, with little to no user interaction required.

Other works specifically focus on integrating user input into the generation process. Nishida et al. have presented an example-based method for creating road networks that uses a simple user-drawn sketch as input [NGDA16]. Along with that, a few sections are selected on a map, from which road patches are extracted. These serve as example shapes and are used to generate a road network using example-based and procedural growing algorithms.


(a) The first input fragment

(b) The second input fragment

(c) The resulting network, obtained by merging the two input fragments

Figure 2.5: An example produced by the system presented by Aliaga et al.: Two street networks with different styles (a, b) get merged into a combined network (c). [AVB08]

Buildings and other Geometry

A large number of methods to generate procedural buildings make use of grammar-based approaches, such as L-systems, split grammars, or shape grammars [STBB14]. While these techniques usually require much initial effort to create the rules, they can produce realistic-looking results with a relatively low computational cost. A common shortcoming of grammar-based methods, however, is the lack of support for the spatial constraints that buildings have to follow. The concept of *split grammars*, as presented by Wonka et al., tries to mitigate this issue by explicitly including support for such restrictions into the generation process [WWSR03]. Furthermore, it also supports the addition of geometric details to façades. An extension to this was presented by Müller et al., which allows the creation of more complex building shapes [MWH+06]. Figure 2.6 shows building models generated using this method. As noted by the authors, 190 CGA shape rules had to be manually created to achieve this result. To reduce the work required to define grammar rules, Wu et al. have presented an inverse procedural modeling system [WYD+14]. Taking an image of a façade as input, it derives a split grammar representing the given layout.

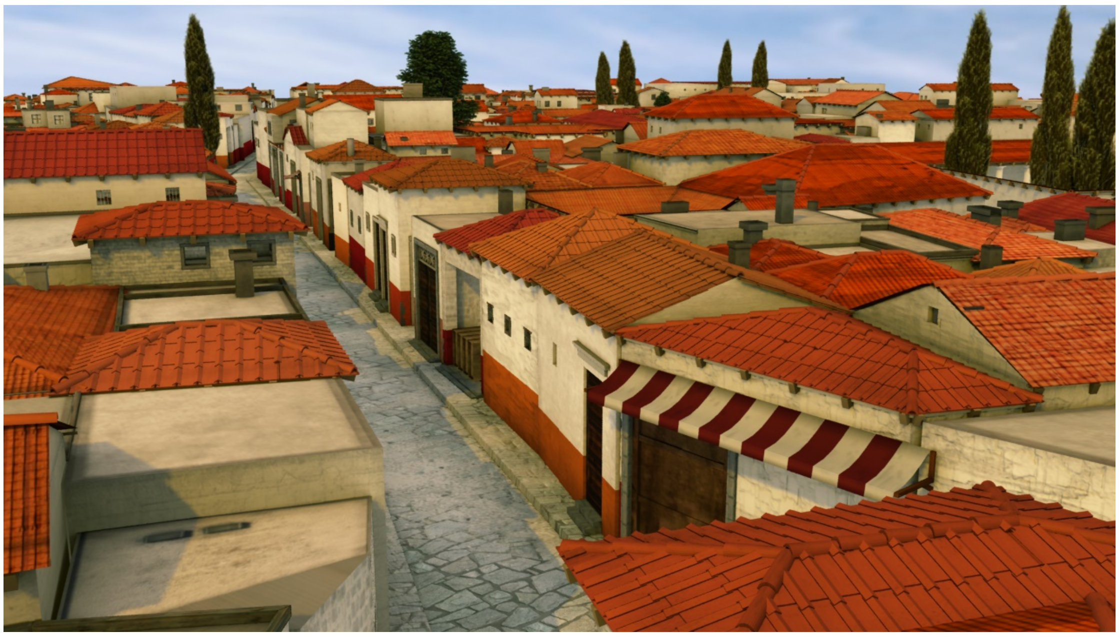


Figure 2.6: Procedurally generated buildings modeled after real footprints of Pompeii. [MWH+06]

Several of the systems mentioned in Section 2.2.3 also support the generation of buildings. Parish and Müller have adopted their layout-generation method to also generate buildings using L-systems [PM01]. Weber et al. include procedurally generated buildings as an integral part of their system, as they are used to visualize the change in population density over time [WMWG09]. Based on envelopes calculated from a lot's monetary value, shape grammars are used to generate building models. However, since their work does not focus on the creation of visually complex buildings, the generated architecture is relatively simple. Similarly, the *Citygen* system by Kelly and McCabe also includes basic building generation mainly for visualization purposes [KM07].

Another piece of research focuses on procedurally generating cities in real-time [GPSL03]. While the road layout consists of a simple 2D grid, the buildings feature more complexity. Floor plans are generated by iteratively placing and extruding regular polygons. After each extrusion step, a new polygon is added and merged with the existing geometry. This process is repeated until the building is finished. While this method provides a good trade-off between visual complexity and efficiency, it is limited to the creation of high-rise buildings. To achieve interactive frame rates, the researchers make use of view frustum filling and caching techniques for the generated geometry. New buildings are created as soon as the user encounters them, resulting in a pseudo-infinite city.

To model the interiors of buildings, several methods have been developed [HBW06, MSK10]. Research has also been presented to create other city features, such as traffic signs [TB16] or parks [Vas18].

2.2.4 Data-driven Procedural Generation

The main concept behind data-driven PCG is the utilization of preexisting data to create new content. The type of data used can vary widely, and consist of templates, patterns, or real examples [KKC18]. The technique is applicable to many different types of content, and data can be used in various ways. Loose classification can be made based on how the data is integrated into the generation process:

- Constraints: The data takes the form of affirmations and limitations, such as distribution maps or sets of rules that the generated object should follow.
- Example: The data is used as an example for different characteristics. It can take the form of real data, patterns, sketches, etc.
- Training Data: The data is used to train a generation system (e.g., a system making use of machine learning) to improve or adjust its output data.
- Sparse Data: Existing data that might be incomplete or in a different format. Examples for this include data points obtained from measurements that shall be used for object reconstruction or high-level requirements.

In reality, data-driven systems for PCG can make use of its input data in multiple ways, and can also include different types of data. These categories shall therefore not be viewed as strict, but rather as general guidance.

Furthermore, example-based methods can be divided into two categories: *parametric methods* and *non-parametric methods* [NGDA16]. In parametric methods, the data is reverse-engineered to obtain parameters and properties that characterize it. These extracted values are then used to generate new content. Especially highly-structured types of data can benefit from this approach, as a reduced parametric representation is often more easily possible. A side effect of this extraction process is the loss of information. This may sometimes be desired, so as to integrate implicit variation in the output data by constructing it from lower-dimensional inputs. However, it can also lead to an unwanted loss of interesting features.

In contrast, non-parametric methods incorporate the data directly. This approach mitigates the possible loss of information by explicitly including the complete example into the generation process. As a result, it is also applicable to more complex types of data.

Closely related to parametric example-based PCG is the concept of *inverse procedural generation*. It typically pursues the objective of generating a specific object from existing information. In its simplest form, this approach can be used to reconstruct objects from sparse data. However, it can also be used to create new objects that have a similar structure or follow the same rules as the input data. Due to the similarities between inverse procedural generation and parametric methods, and the lack of exact definitions, significant overlaps in terminology can be expected.

In city generation methods based on data-driven techniques, input data often takes the form of environmental information, such as natural or statistical distributions. For example, geographical circumstances largely influence the development of cities. To generate plausible output data, procedural methods have to respect these constraints. If existing data is available, population density maps can further control the generation process [SYBG04, VABW09].

An innovative use case for a data-based procedural generation method has been presented by Hooshyar et al. [HYWL18]. The authors have developed an educational game that focuses on improving reading skills of children. By utilizing data collected during gameplay, content that is tailored to address the player's strengths and weaknesses is generated using a genetic algorithm. It is then directly integrated into the game. As a result, higher performance improvements could be achieved in children through the customization of game content.

Data-driven approaches to PCG have also been employed to generate terrain. In research presented by Zhou et al., existing elevation models can be combined with a user-drawn sketch to create synthesized landscapes [ZSTR07]. Features such as valleys and ridges are extracted from the input height field and stored as a tree. Following the user's sketch, a pattern matching algorithm then looks for suitable patches in the source model and places them in the output model. Figure 2.7 shows a synthesized terrain along with its respective input data. While the method can generate plausible terrain models, the input data that can be used is limited. As the feature extraction algorithm requires the presence of either ridges or valleys, an input map with low height variation will result in poor output data.

Nishida et al. make use of a similar technique to generate road networks [NGDA16]. The system extracts individual patches from user-selected source maps that act as examples. Additionally, a set of statistical parameters such as road length, orientation, and curvature, is computed from the input data. The road network is then generated using a growth-based algorithm, starting from a user-defined source point and iterating over all unconnected vertices. By combining the extracted patches based on a probability scale, a new road network featuring a similar style to the examples is created. If no

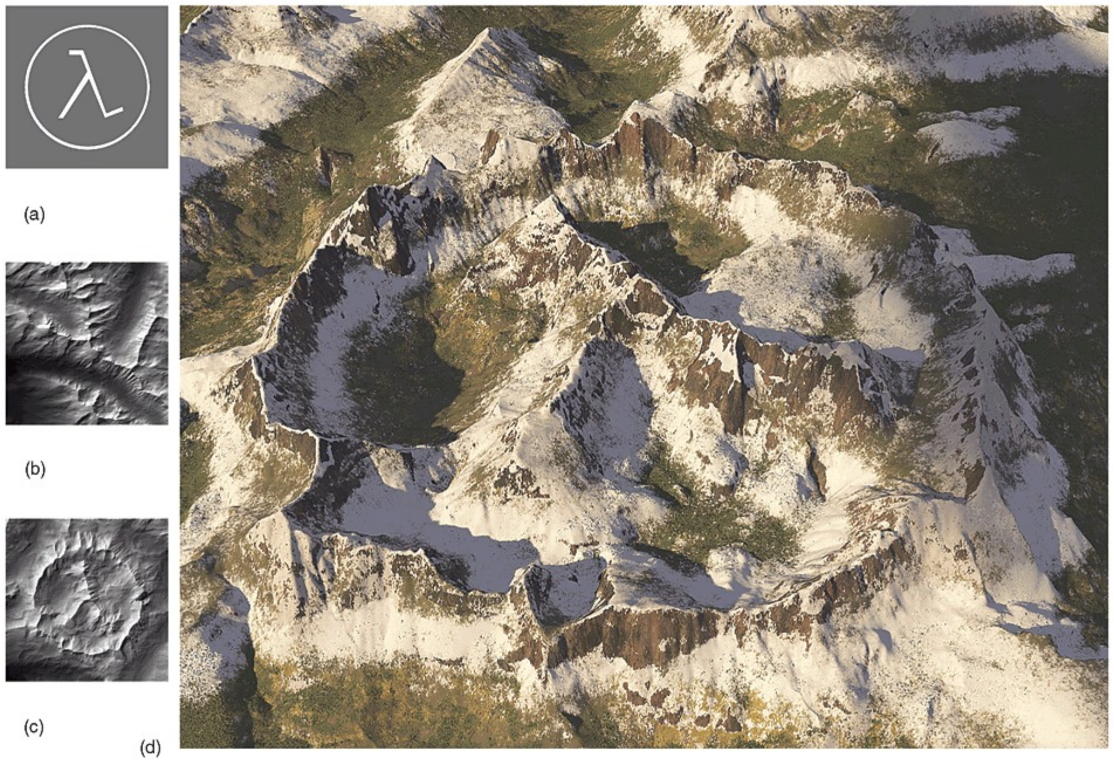


Figure 2.7: Synthesized terrain created using a data-driven method by Zhou et al. (a) Input user sketch. (b) Base elevation map. (c) Resulting synthesized elevation map. (d) Rendered result. [ZSTR07]

suitable patch can be found at a particular growing stage, the previously extracted features are used to generate a new patch procedurally. A shortcoming of this method is that for example-based growth, only patches that exist in the source data can be used. The fallback to procedural-based growth provides some mitigation, but many interesting features get lost using this technique, as noted by the authors.

Research has also been presented that employs procedural modeling techniques to reconstruct real-life objects from sparse data. Livny et al. have presented a method to generate tree models from point cloud data as it can be obtained from a Light Detection and Ranging (LIDAR) mapping system [LYC$^+$10]. Individual points from the input data are used to build a graph representing the tree skeleton. By integrating optimizations and refinements, such as assumptions on tree properties, a close approximation of the structural tree geometry can be constructed. Subsequently, fine branches and leaves are generated using L-systems since the source data resolution is typically too low to extract information about such small details with adequate precision.

A different approach to integrating data-driven PCG has been presented by Merrell et al. [MSK10]. Out of a list of high-level requirements, the presented system is able to generate a complete layout for a residential building. A Bayesian network trained on real-world data is the basis for the generation algorithm. It first generates a full architectural program for a building, consisting of a list of rooms including their desired area and general shape, as well as relationships between rooms. To obtain a complete and cohesive set of floor plans, further optimizations on possible layouts are performed. A 3D model of the building can then be created. The motivation to integrate data-driven techniques stems from the difficulty to formalize the factors influencing human comfort in architecture. Additionally, adjacency relationships between rooms are often part of a larger semantic structure that cannot be taken into account easily in simple rule-based systems.

An algorithm presented by Müller et al. uses inverse procedural generation to reconstruct building façades from single textures [MZWVG07]. Image processing techniques are applied to the source texture to detect floors and individual façade tiles. By matching the tiles to their respective 3D models from a library of common architectural elements, a complete façade model can be created. The system makes use of shape grammars to encode the resulting information as a set of rules. To generate new façades from this data, the rules would have to be extended by probabilistic parameters that allow for variation of the architectural properties.

A system that supports the generation of similar objects has been presented by Stava et al. [SPK$^+$14]. Taking an existing three-dimensional model of a tree as input, a tree graph is created from it to facilitate processing. A set of 24 parameters that represent the structure are then estimated using an iterative approach based on optimization and similarity measures. Monte Carlo Markov Chains are used to tune the parameters until a satisfactory level of similarity is achieved. The resulting attributes describe a tree that is stochastically similar to the one that was provided as an example. However, it is not possible to exactly recreate the input tree using this approach.

Similarly, the system presented by Aliaga et al. employs a parametric approach to the data-driven generation of urban layouts [AVB08]. From a road network that acts as an example for the synthesis of a new layout, a set of parameters is extracted. For each street, these include the hierarchy level, the mean and variance of the distances between points, and the mean and variance of the angles between segments. New streets are generated using a random walk that takes these parameters into account. A number of constraints control the creation of streets, such as limiting crossings with existing streets to intersection points. Additionally, a tortuosity attribute calculated from the source data is used to create a poly-line that more accurately represents the curvature of the streets. As noted above, this reduction of a complete road network to a set of parameters results in a loss of individual details. An example given by the authors shows that two cities with different road styles can still lead to very similar statistical properties.

A common difficulty for data-driven PCG techniques is the dependency on the source data. By design, such methods rely heavily on the input data to synthesize new content. However, depending on the origin and the type of data, it can be noisy, incomplete, or wrong. This can result in undesirable outputs from the procedural generation system, but may also completely prevent successful content creation. Consequently, it is vital for these data errors to be handled appropriately if they are likely to occur within a given context. This can either be done by cleaning up the data beforehand [KMK12] or by simply omitting undesirable input data from the generation procedure [LYC+10]. To improve the quality of the output data, post-processing techniques can also be used to remove potential artifacts or unwanted features [NGDA16].

CHAPTER 3

Data-driven Generation of City Layouts

3.1 Theoretical Approach

During our research, it became clear that a considerable amount of work has been done in the field of procedural city generation. However, not much work has been presented that focuses on data-driven city generation using layout information from multiple cities. The algorithm we present is focusing on exactly this problem. It uses a simulation-based approach to combine two or more city layouts in a sensible manner. A custom mixing ratio for the input maps can be defined. This influences the blending process and makes it possible to prioritize one city layout over others.

Data loaded from input files are preprocessed to remove artifacts and reduce the resolution if necessary. Afterward, the data is transformed into an internal node/link data structure. Links are represented as *edges*. Each edge corresponds to a single road segment, connected to exactly two nodes. Nodes are split into two types, based on the number of links that are connected. *Waypoint nodes* are always part of exactly two links, as they are used to split a long road into multiple parts (e.g., to approximate curves). *Junction nodes* either have one (in the case of a dead-end) or more than two links connected to them and usually form a junction at the intersection point.

The initial positioning of the input city layouts is based on the individual section boundaries. Each node is considered to find the largest extents of the respective input map. Using this information, the center point of the layout is calculated and the map is placed in the center of the virtual environment.

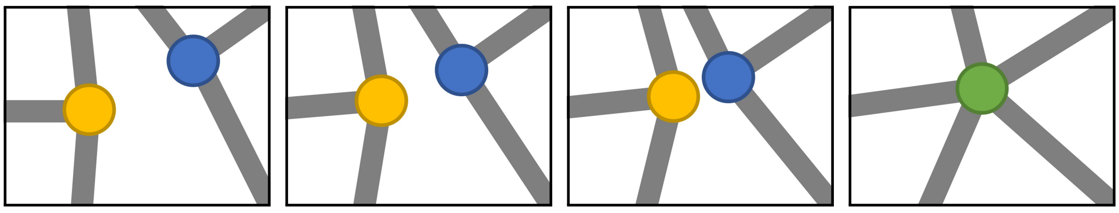
The presented algorithm uses a physics-based approach to merge road networks. Nodes and edges are both represented as objects in a physical environment. In this context, they behave according to properties assigned to them. The physical environment is used to combine street networks by simulating attraction between nodes of different input data sets. To achieve this, forces are applied to nodes that cause them to move closer together, and eventually merge into one.

Nodes originating from different city layouts are initially placed on different layers. Each junction node exerts an attraction force on nodes from other layers, pulling them towards it. The force is calculated using Newton's law of universal gravitation [OR13]:

$$F = G\frac{m_1 m_2}{r^2}$$

Here, F is the gravitational force, G is the gravitational constant ($G = 6.67430(15) \times 10^{-11}\ m^3{\cdot}kg^{-1}{\cdot}s^{-2}$), m_1 and m_2 are the masses of the respective objects, and r is the distance between the objects. The force is therefore directly proportional to the masses of the objects, and indirectly proportional to the square of the distance between them. For our use case, this means that the forces exerted on two nodes that are close together will be higher, and they are more likely to collide. On more distant objects, on the other hand, the impact will be much lower. Only junction nodes are affected by the attraction force. The mass of a node is interpolated between two predefined values, based on the mixing ratio of the respective layout and a dynamic priority parameter (explained further below). This range needs to be defined based on the implementation, as very low values may cause too erratic movements, while excessively high values will result in extremely slow simulations.

If two nodes collide, they are merged into one. The properties of both nodes are combined either by blending or by selecting one over the other. In the latter case, the respective properties with a higher priority are applied to the merged node. The edges connected to both nodes are combined in the merged node, though some limitations are applied to reduce the number of redundant edges in the output data. For example, if both nodes are connected to a third node, only one edge is kept. Similarly, if the angle between two of the edges is too low, one of them is removed. Figure 3.1 shows a step-by-step depiction of the attraction and merging process.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

Figure 3.1: A step-by-step depiction of node attraction and merging caused by gravitational forces. (a)-(c) The two nodes exert attraction forces on each other. (d) Nodes have collided and were merged into one. One of the two top edges has been removed.

To prevent a collapse of the road network due to the gravitational forces, a counter-force is necessary. This is implemented as a spring force. Each edge acts as an elastic spring that tries to keep its connected nodes in the same orientation and distance from each other. The springs are configured with certain stiffness and damping parameters that apply to both extension and compression. These allow roads to be stretched and compressed within certain limits. Due to the interconnectivity of road networks, the effects are relayed to other nodes and edges as well. The deformation caused by moving a single node is therefore distributed across multiple edges in the vicinity. This allows a layout to be transformed, while still keeping the overall shape intact. Figure 3.2 illustrates this behavior.

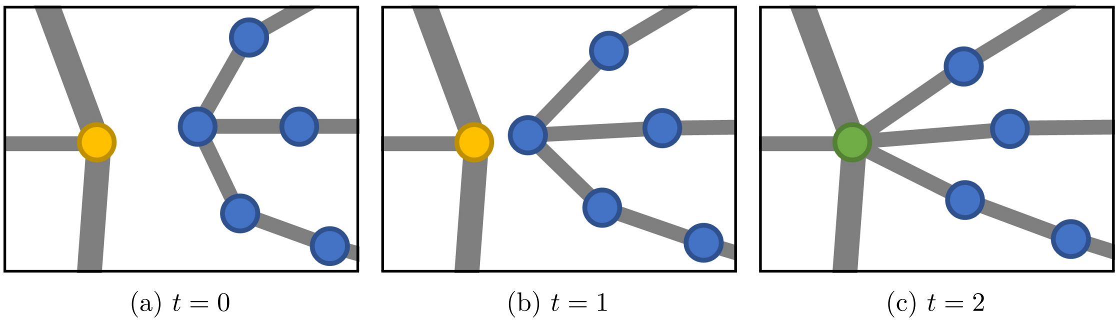

(a) $t = 0$ (b) $t = 1$ (c) $t = 2$

Figure 3.2: An illustration of the impact that the springs have on the rest of the road network. The gravitational force exerted on a single node is distributed across the local network.

The frequency of the springs, as well as the attractive forces exerted by the nodes, are scaled in direct proportion with the mixing ratio of the respective input map. Predefined value ranges serve as base for the interpolation:

$$f = f_{min} + mixRatio_l(f_{max} - f_{min})$$

Here, $mixRatio_l$ is the mixing ratio of the respective layer, and $[f_{min}, f_{max}]$ is the interval that specifies the frequency range. Overall, this results in a more stable network if the mixing value is high for a specific input data set. On the other hand, a lower value causes the layout to transform more easily. As the gravitational forces exerted by nodes on other layers have a greater impact, it will adjust to these more readily. Furthermore, the priority of each node is calculated based on the edges that connect to it. Larger roads are considered to be more important, as they represent the arterial connections in cities. To protect these structures from too much distortion, nodes with a higher priority exert greater attraction forces. Figure 3.3 shows a visualization of the forces present during the attraction process.

The above processes are executed in a loop with a configured number of iterations. It can be broken down into individual steps, the first of which performs the force calculations.

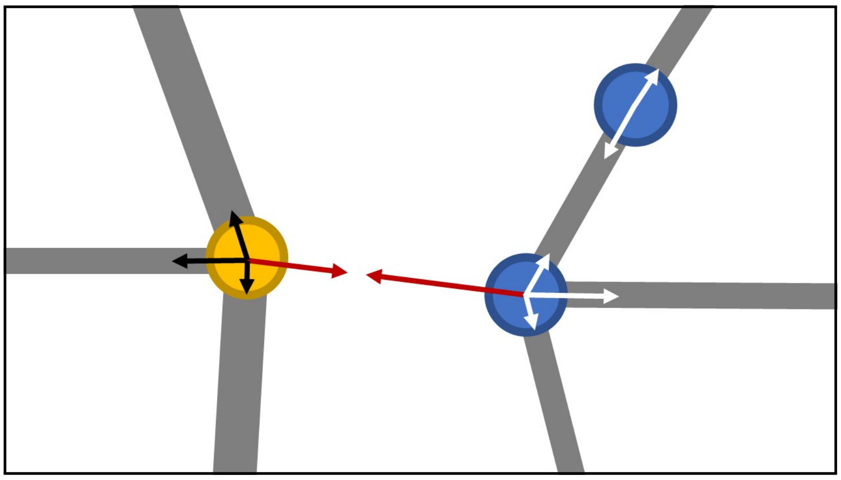


Figure 3.3: An illustration of the forces at play during attraction. The actual attraction forces are shown in red, the spring forces in black and white.

Attractive forces are calculated and temporarily stored for each node. Next, they are converted into actual movement through the physics simulation. This is performed with consideration of the physical properties assigned to nodes, as well as any spring forces that take influence. Through these movements, compression and extension of edges occur, which in turn cause spring forces to be applied to nodes. Nodes that have collided in the previous step are then merged if the conditions are met. After a certain number of iterations, the resulting layout is cleaned up in a post-processing phase. Invalid structures are corrected and the result is improved. For instance, roads might cross over each other without being connected via a node that acts as a junction. These situations can occur because road networks are simply placed on top of each other in the beginning. In such cases, a junction is created between the crossing roads.

The proposed algorithm employs a non-parametric approach to data-driven procedural generation. The complete road networks from the input data sets are included in the processing phase, resulting in no explicit loss of information.

3.2 Source Data

For applications relying on real-world geographical data, several sources are available. Previous works have frequently been seen to use GIS data [ABVA08, AVB08, MWH^{+}06, WMV^{+}08]. In fact, any kind of information bound to spatial coordinates can be regarded as a GIS. The inclusion of spatial information makes it possible to map individual data points to specific locations. It can take the form of latitude and longitude values, but also more complex positional information. As the definition of GIS is very imprecise, many different formats exist to digitally represent geographical data. Due to its complexity, special kinds of databases are often necessary to store the information. These factors can make it difficult to work with location-bound data. To combat this, institutions like the Open Geospatial Consortium have developed standards for GIS data to facilitate the integration in applications [OGC20].

One of the first modern, publicly available GISs was Google Maps [Law15]. Before its release in 2005, geospatial data was difficult to obtain and process. The web-based application allowed consumers to freely browse and search global GIS data. An open-source alternative is OpenStreetMap [HW08]. The community-driven project relies heavily on user contributions to create and maintain its map data. In turn, all of its data is available free of charge via APIs and web applications.

Many countries and cities nowadays also make their geographical information available online. Thanks to open-data initiatives, the public can generally access this data freely. Apart from easier data sharing, the drivers for governments to open their data include better transparency and accountability, the possibility for citizens to participate in governance, and increased innovation and economic growth [Jan12]. However, there are also considerable costs associated with open data for governmental bodies [JSS+17].

To obtain input data for our presented system, multiple sources have been evaluated. The data source for this work had to fulfill several criteria:

- Accessibility: The data had to be freely accessible.
- Correctness: The data had to be reasonably correct. Small issues do not cause any problems, though there should not be any major inaccuracies.
- Data format: The data had to be in a well-described format that is easy to parse algorithmically.
- License: The data had to be licensed as permissive as possible and permit the publishing of modified data.

Google Maps presents a relatively high data quality [CJMW10]. However, even though it is one of the most popular mapping services, the data cannot easily be exported and downloaded. API access is possible, but not in a way that would be suitable for our use case. Geospatial open data provided by cities themselves would most likely be the most correct data source, as they generally include all new developments as soon as they are constructed. Due to the fact that cities typically provide this data on their respective websites, the accessibility aspect is affected. Furthermore, the data formats that are used by different sources can vary, making it difficult to parse the data.

Data from OpenStreetMap, on the other hand, is easy to obtain and process. Public APIs exist that are free to access, and the resulting data follows a well-defined structure. The licensing model is very permissive, allowing the usage, modification, and redistribution of their provided data, as long as OpenStreetMap and its contributors are credited [Ope18]. Since the content is generally created by the community, the quality is not guaranteed.

In fact, research has found that the reliability of OpenStreetMap data is still lacking [BJA+16]. Especially the heterogeneity of the data, often caused by different production processes, could cause difficulties for applications [GT10]. Despite these drawbacks, it was decided to base our application on data provided by OpenStreetMap. The ease of access, open licensing model, and well-defined data structure have a high priority for this use case. As the city map generated by our application does not include any geospatial coordinates, slight inaccuracies in the input data don't cause any issues.

3.2.1 OpenStreetMap Data

OpenStreetMap data is organized based on a specified structure. The data model contains three basic elements [HW08, Ope14]:

- *Nodes*, which are points in space
- *Ways*, which define lines and boundaries
- *Relations*, which define relationships between other elements

Additionally, each element may have one or more *tags* associated with it. In general, the data structure follows an extended link/node model as it was described in Section 2.1.2. Links in the OpenStreetMap data format are represented by *way* elements. However, ways do not always just cover a single line segment but may cover many. An ordered array of up to 2000 nodes can be part of a way [Ope14]. Therefore, a way defines a polygonal chain. Direct connections between two nodes inside a way are always straight. To represent curves and more complex shapes, additional nodes can be created. By adding these at their respective positions in a way, irregular shapes can be sampled and approximated.

Among others, ways are used to represent roads. Other uses of ways include rivers, railway lines, or areas such as parks, but these are not relevant for the presented system. The start- and endpoints of a road are determined by the first and last nodes that are part of the way, respectively. A very simple, straight road in the OpenStreetMap data format thus consists of two nodes and one way element, which references these two nodes.

To identify a way as a road, OpenStreetMap uses the tag *highway*. Contrary to the meaning of the word in American and Australian English, it is used for "any road, route, way, or thoroughfare on land which connects one location to another and has been paved or otherwise improved to allow travel by some conveyance, including motorised vehicles, cyclists, pedestrians, horse riders, and others (...)" [Ope14]. Different kinds of roads can be distinguished by the value that is given to the tag. The most notable ones are *motorway*, *trunk*, *primary*, *secondary*, *tertiary*, *unclassified*, and *residential*. However, the tag is also used for other road features, such as crosswalks, elevators, or platforms at bus stops. To obtain a correct and cohesive data set of a city's road network, these features have to be filtered. This could generally be performed either as part of our application or at the API level.

3.2.2 Obtaining Data

Data can be obtained directly from the OpenStreetMap website in an XML format. A global `osm` element contains all elements representing nodes, ways, relations, and tags. XML tags are used to add more information to elements, such as a unique id, location data, or a timestamp of the last modification. An example of an OpenStreetMap data file can be seen in Listing 3.1.

Listing 3.1: An example of an OpenStreetMap data file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="Overpass␣API␣0.7.55.9␣ab41fea6">
<note>The data included in this document is from www.openstreetmap.org. The
    data is made available under ODbL.</note>
<meta osm_base="2020-02-02T15:04:02Z"/>

  <bounds minlat="48.1950352" minlon="16.3685828" maxlat="48.1954375" maxlon=
      "16.3690683"/>

  <node id="35479449" lat="48.1952970" lon="16.3693475"/>
  <node id="35479550" lat="48.1951977" lon="16.3684291"/>
  <way id="5132481">
    <nd ref="35479449"/>
    <nd ref="35479550"/>
    <tag k="highway" v="living_street"/>
    <tag k="lit" v="yes"/>
    <tag k="maxspeed" v="7"/>
    <tag k="name" v="Mozartgasse"/>
    <tag k="oneway" v="yes"/>
    <tag k="source:maxspeed" v="AT:walk"/>
    <tag k="surface" v="asphalt"/>
  </way>

</osm>
```

The export function on the OpenStreetMap website is limited to a maximum number of 50000 nodes. As there is no option for filtering the data, apart from a bounding box selection, this limit is quickly reached. This makes it unusable for obtaining city-scale datasets.

Other sources and mirrors exist that allow the download of larger amounts of data. For example, the project *Planet OSM*[1] provides regularly updated copies of the complete OpenStreetMap data. However, the compressed size of such a file is 85 GB at the time of writing, making it unsuitable for our use case. Another available data source is the *Overpass API*[2]. Contrary to the main OpenStreetMap API, it is optimized for reading and filtering data. Therefore, it also supports the use of search criteria to limit the number of results. To implement these filters, a special query language needs to be used. The *Overpass Query Language (Overpass QL)* allows the user to specify the desired

[1]`https://planet.openstreetmap.org` (Accessed: 11.02.2020)
[2]`http://overpass-api.de` (Accessed: 11.02.2020)

elements, perform filtering by bounding box, area, tag, or other property, find relations, and more. Besides, the *Overpass Turbo*[3] tool makes it possible to easily develop and test queries using a Graphical User Interface (GUI).

Thanks to the filtering capabilities of the Overpass API and Overpass QL, less work needs to be invested in preprocessing data to remove undesired road features. To identify which elements shall be included in the filter query, the OpenStreetMap data from several cities in different countries have been inspected. The goal was to find the criteria which would permit a general application of the filter, without being too restrictive (i.e., missing roads) or too permissive (i.e., undesirable data points). As all roads carry a *highway* tag, this was used as a starting point to define a white-list filter for possible values. Using Overpass QL, the filter function shown in Listing 3.2 has been developed to obtain the desired data.

Listing 3.2: A filter function for all roads using Overpass QL

```
[out:xml][timeout:25][bbox:{{bbox}}];

way["highway"~"motorway|trunk|primary|secondary|tertiary|unclassified|
    residential|motorway_link|trunk_link|primary_link|secondary_link|
    tertiary_link|living_street|service|pedestrian|road"]
  ["area"!="yes"]
  ({{bbox}});

// recurse down: add all nodes that are part of these ways
(._;>;);

out;
```

The filter function is intended to be used on Overpass Turbo, as this allows easy selection of the desired bounding box. First, the request settings are declared. Those specify the output data format and a timeout that should be long enough for city-scale bounding boxes. Secondly, a query statement for ways is performed inside the bounding box. Only ways that have a *highway* tag with one of the specified values are included. As ways can also be used to represent areas, those are removed from the result set. Next, recursion is performed to grab the remaining data. This step makes sure that all nodes that are part of the queried ways are also part of the output. Finally, the data is output.

3.2.3 Possible Issues

The heterogeneity of OpenStreetMap data, as described above, can lead to difficulties in the city generation process. This is often due to different sampling resolutions. For example, a long, straight road without any intersections can be represented in multiple ways. The simplest, as described above, only uses two nodes. However, it is also possible to subdivide it into smaller segments, by adding more nodes along the way. For a visualization application, this does not make a big difference, since the result would be

[3]`http://overpass-turbo.eu` (Accessed: 11.02.2020)

the same. An application that relies on these waypoint nodes to find possible intersection points, such as ours, will produce a different output.

Difficulties can further arise from parallel ways defined in OpenStreetMap. When lanes are divided by a barrier, such as a wall or vegetation, the road is modeled as two individual way elements. As the presented application uses a data format that only allows a single connection between two nodes, such situations have to be considered and handled appropriately. This is explained in more detail in Section 3.4.6.

Another possible issue is the inconsistent classification of roads. While a number of examples and best-practices for road classification exist [Ope14], these can hardly be enforced. As a result, additions or modifications from different sources may use other categories for the same road types. For example, alleys in the historic center of Rome (Italy) that are accessible to cars are mostly classified as *highway=service*. Similar streets in Vienna (Austria) are tagged as *highway=living_street* or *highway=residential*. In turn, the tag *highway=service* is mostly used for parking aisles. This difference can be seen in Figure 3.4. As a result of this varied classification, the tags cannot be used effectively as a measure for street priority for our application.

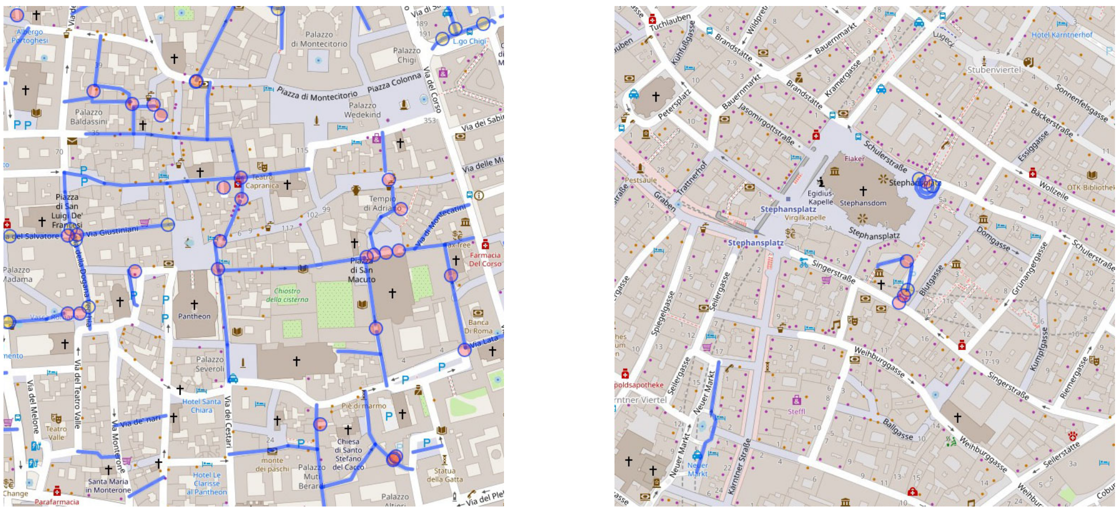


Figure 3.4: Difference in street labeling in OpenStreetMap. Streets of type *service* have been highlighted in the historic centers of Rome (left) and Vienna (right).

3.3 Implementation Concept

To showcase the devised algorithm, an application was developed that implements it. Several requirements have been composed, which the software should fulfill. These can be divided into scope constraints, functional requirements, and non-functional requirements.

3.3.1 Scope Constraints

The application is of prototypical nature. It serves mainly as a means to present and evaluate the algorithm described in Section 3.1. Therefore, the scope can be limited by the following constraints:

- The software does not need to run as a standalone application. It is sufficient to run it in the context of a game engine.
- The input data is provided by the user as files in the correct format. The bounding box is limited beforehand.
- The application operates on a two-dimensional plane. Height values from the input data sets (if present) are disregarded by the generation process and are not exported as part of the output data. In future work, it might be interesting to include height.
- The output does not contain any location information. Geographical coordinates are not processed by the algorithm, and no direct relations to geospatial locations are present in the output.
- The output is not represented in a GIS-compatible data format. As the prototype focuses on displaying the results, the preparation of the data for subsequent use is not part of the scope.

3.3.2 Functional Requirements

To accurately represent the devised algorithm, several requirements have to be met. The following enumeration lists the properties that are necessary for the system.

- The application allows the user to select multiple road layouts as input, along with their respective mixing ratios.
- The user-defined mixing ratios influence the procedural generation process. Higher values result in less modification of the source data. Lower values cause more distortion of the input.
- The user can influence the generation process indirectly. Default parameters can be modified before starting the algorithm to change the behavior of the system.
- The user can influence the generation process directly during runtime. By moving individual nodes and triggering merges, it is possible to modify the result.
- The application produces an output file that depicts the result of the generation process.

3.3.3 Non-functional Requirements

In addition to the functional requirements listed above, a number of non-functional requirements should be met by the software.

- The hardware requirements can be met by up-to-date consumer hardware. The demands for CPU and RAM for ideal operation are not higher than what can be provided with current gaming hardware.
- The running time of the software is acceptable. On up-to-date hardware, the time required for processing two large cities is no longer than 12 hours.
- Adequate frame rates of rendering are achieved for interactive use during iterations.
- The application is reusable for future projects. It is packaged in a way that allows other developers to make use of it.
- The source code is extensible. If new features are required in the future, they can easily be added to the application.

3.3.4 Software Design and Architecture

To support the efficient development of the prototype application, a game engine with support for physics simulations is used as a base. This reduces the amount of work considerably because many functionalities are already provided. Similarly, the availability of a physics engine removes the need to implement physical processes and interactions.

The application is intended to run on a single desktop PC. The actual algorithm is contained as an integral part of the application. As a result, no external interfaces are required.

Internally, the application logic is divided into several modules. From a high-level perspective, the following components can be identified in the system:

- The main controller which manages the program flow.
- The physics engine responsible for updating node and edge positions based on the exerted forces.
- Multiple Nodes, which operate independently in conjunction with the physics engine.
- Multiple Edges, for which the same applies.

A schematic view of these is shown in Figure 3.5. All of the components operate inside the game engine environment. The CityMerge controller is the main interaction point for the user, as it is in charge of handling the input data and setting up the environment.

As the devised algorithm follows an iterative approach, the main controller also counts the iterations and advances the physics simulation. Furthermore, it is responsible for switching between stages and triggering the output.

Nodes and edges are created based on the information extracted from the input data sets provided by the user. During the simulation phase, there is no global control over their behavior. All elements perform local operations on their own, acting as individual agents. They operate independently, based on preprogrammed rules and their physical properties. Modifications to shape and position are controlled by the physics engine. Implicit feedback is provided through extensions and compression of edges. Thus, each junction node is in charge of applying attraction forces to nodes in its vicinity. The actual translation of objects is performed by the physics engine, based on the applied forces.

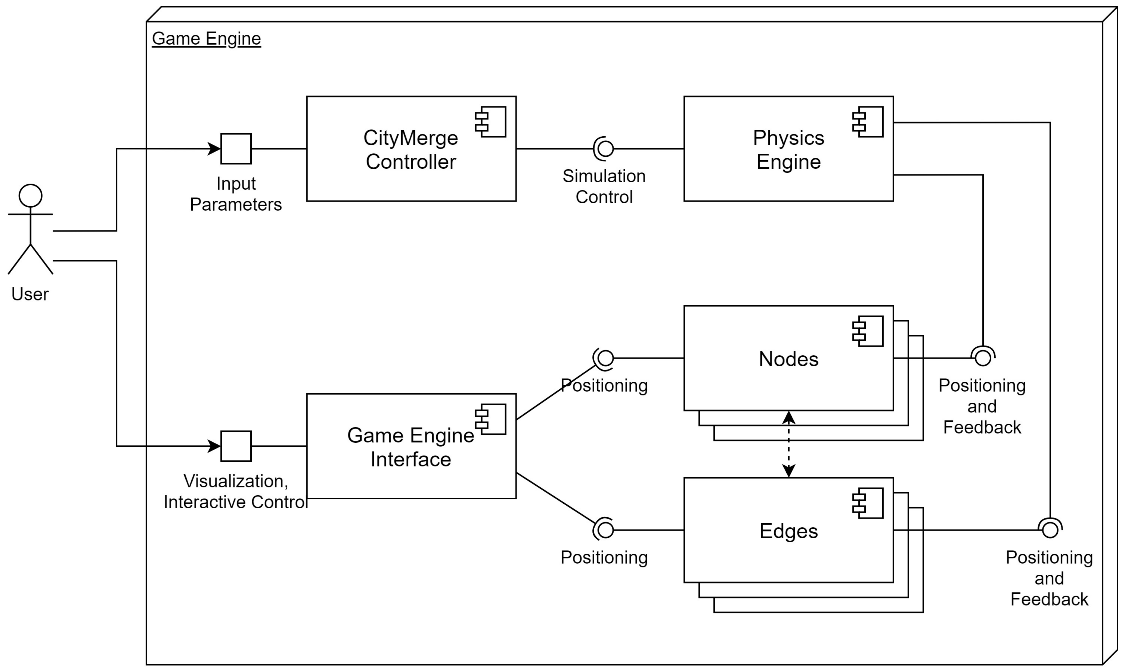


Figure 3.5: A schematic view of the individual components and their interconnectivity from a high-level perspective.

On a coarse level of detail, the program flow can be divided into different stages:

1. Initialization
2. Main content generation
3. Post-processing

This flow is managed by the main controller. In the initialization phase, the input data provided by the user is read and parsed. Nodes and edges are created to represent the road networks. They are initialized with appropriate parameters that control their behavior. The actual merging of road networks to a combined layout occurs in the main loop. This procedural generation process is run in a decentralized manner, distributed across all individual node and edge objects. The first step is the calculation of attractive forces in each junction. The forces are applied to the affected nodes, but no movement occurs yet. This is important so that the effect radius of other junctions is not influenced. Next, the physics simulation is performed. It uses the applied forces in combination with physical properties to execute the movement of the nodes. The physics engine also calculates and incorporates the springs at this point. Lastly, overlaps that have occurred due to node movements are resolved by merging the affected nodes. After the desired number of iterations have passed, this process is stopped. In the post-processing stage, the result is improved by fixing invalid data points and cleaning up the network, to achieve a more plausible result. Algorithm 3.1 shows the application flow in high-level pseudocode. A more detailed description of the implementation is described in Section 3.4.

Algorithm 3.1: The high-level application flow in pseudocode.

```
  Data: A list of input data sets $maps$
1 $nodes, edges \leftarrow$ Initialize($maps$);
2 while $i < i_{max}$ do
3     CalculateForces($nodes, edges$);
4     SimulatePhysics();
5     MergeCollidedNodes();
6     $i \leftarrow i + 1$;
7 end
8 Postprocess($nodes, edges$);
```

3.4 Implementation of the CityMerge System

In this section, a detailed description of the implementation will be given. Firstly, the selected development platform and the motivation for its selection will be presented. Details about extensions and libraries that are integrated into the application are also found below. Next, the different stages, as well as the individual components of the application, will be described. Lastly, several measures that were taken to improve the performance will be explained.

3.4.1 Unity Game Engine

The game engine selected for the implementation of this application is *Unity*[4]. A number of factors influenced this decision. Unity is a development platform available for Windows, macOS, and (in a preview version) Linux. This makes it usable on different platforms, though applications can be created to target a number of different devices. Several licensing models are available, though the basic version is completely free and royalty-free for personal use. For our use case, the functionality provided by this version is sufficient.

The Unity engine can be used to create applications in both 2D and 3D. Its adoption is not limited to video games. In fact, it is targeted towards many other industries as well, such as manufacturing, animation, or architecture [Unia]. It was also applied for simulation applications [Jar14]. In general, Unity is a very popular environment for 2D and 3D applications. A large number of resources in the form of documentation, online tutorials, and books are therefore available. For our application, this means that inexperienced users can get used to the environment quickly if they are not already familiar with it. Furthermore, it will not be difficult for developers to extend our system with new functionality. Another consequence of Unity's high popularity is the availability of libraries. In addition to packages distributed via the *Unity Asset Store*[5], many open-source projects are available for Unity. Further motivation to select Unity came from the fact that knowledge acquired in previous projects could be applied here.

Unity uses *GameObjects* to represent entities inside its environment. The behavior of these objects can be controlled by adding *Components*. These can be preprogrammed behaviors provided by Unity, such as colliders, or components developed by users. Custom scripts, called *MonoBehaviours*, can also be added to GameObjects. As a programming language, Unity uses C# based on .NET.

The main loop in Unity is controlled via event functions. These are called at specific points in time, depending on their purpose. For example, an *Update* function is available in all MonoBehaviours. It gets called once for each frame and is intended to contain game logic that requires continuous processing. These event functions are further used to manage the life cycle of objects. Figure 3.6 shows the game loop in Unity as a flow chart. The main loop in Unity thus consists of a physics update, processing inputs, running the game logic, and rendering. However, the physics update may be executed more than once per frame, or not at all [Unib]. This may happen because physics calculations are executed on a fixed time interval, as opposed to the constant dynamic calls used for the game logic. If this time interval is lower than the actual duration of the frame, another update is triggered. As the time elapsed between two *FixedUpdate* calls is constant, it allows video games and other applications to compute physics independent of the frame rate. For our use case, however, this behavior is not necessarily desired. Instead, control of the physics update is given to the main controller.

[4]`https://unity.com` (Accessed: 11.02.2020)
[5]`https://assetstore.unity.com` (Accessed: 11.02.2020)

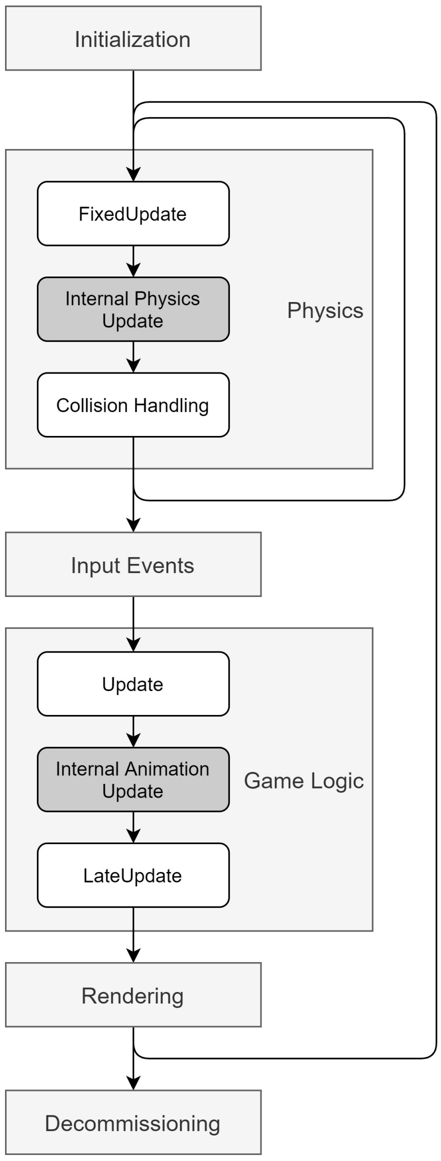


Figure 3.6: A visualization of the program flow in Unity [Unib]. For better visibility, some parts have been combined or omitted.

3.4.2 Frameworks and Libraries

To facilitate the development of the CityMerge application, external functionality has been integrated. This was done through the use of existing libraries at well-chosen points in our system, such as for physics simulation or data structures.

Unity comes prepackaged with a physics engine that enables the simulation of object interactions in real-time. For our presented system, this has a high priority as most of the recurring calculations are based on game physics. The engine simulates the motion of objects based on the laws of classical mechanics, which allows us to specify graspable parameters for nodes and edges. Furthermore, it supports joints and collision detection. Joints can be configured in various ways, such as to allow stretching and compression within certain constraints. Collision detection allows us to quickly check if two nodes have collided, and merge them into one as a result.

To work with the data from the input street layouts, the files need to be parsed. Data exported from OpenStreetMap via the Overpass API is typically stored in an XML format, making it relatively easy to process. The open-source project *real-world-map-data*[6] implements such a parser. The goal of this project is to visualize data from OpenStreetMap in Unity. As such, it provided a great starting point for our data parsing component. However, as we require much more complex handling of the data apart from visual representation, the code had to be heavily modified to suit our needs.

The data extracted from the input maps are stored in a graph data structure. For this purpose, an adaptation of the *QuickGraph* library[7] is used. It provides multiple types of graphs, though only the bidirectional graph is used in our application. Functions to perform different kinds of calculations on graphs are available in the library.

3.4.3 Project Setup

Development was done using version 2019.2.11f1 of Unity. This version was chosen over more stable Long Term Support (LTS) releases to maintain forward-compatibility as much as possible. To that end, care was also taken to not use any deprecated APIs that may be removed in future versions.

The project was created using the template aimed at 2D applications. As our system operates purely on a two-dimensional plane, this provided a useful default setup. Furthermore, the camera was set to use an orthographic projection method for rendering.

A few project settings were modified to better suit our needs. First of all, the API compatibility level was set to ".NET 4.x". By default, Unity uses ".NET Standard 2.0". However, external libraries used by our application require functionality that is not available in that profile. Secondly, the physics configuration was updated to reuse collision callbacks. When a collision occurs, Unity calls an event function implemented by the user, which can then act accordingly. This function receives an object containing information about the collision, such as the position, the two affected objects, or the velocity. To reduce memory allocation, Unity was instructed to reuse these objects, instead of destroying and re-creating them for each collision. Lastly, the fixed timestamp for physics calculation was raised from 0.02 to 0.05. Even though this results in slightly longer overall computation times, it makes the system more responsive to user input. Depending on the hardware, it may be advantageous to raise this value further or even

[6] `https://github.com/codehoose/real-world-map-data` (Accessed: 11.02.2020)
[7] `https://github.com/davidgutierrezpalma/quickgraph4unity` (Accessed: 11.02.2020)

reduce it. As a rule of thumb, a higher value will probably work better for less powerful systems, while lower values will result in a faster simulation if the hardware is strong enough.

3.4.4 Main Controller

The main controller is the principal interaction point for the user. It is implemented as a MonoBehaviour, which is assigned to a GameObject. Before starting the simulation, the user assigns the input maps and specifies the mixing ratios. Aside from that, the number of iterations to be performed is specified via this component, as well as further settings used mainly for debugging purposes. The same GameObject also has other components attached. Builder methods implemented as MonoBehaviours are assigned to it as well, which allows the selection of the respective prefabs. Furthermore, a *PostProcessor* script that handles the post-processing is attached.

The main controller is in charge of controlling the program flow. To simplify this, it is split into different phases which are run sequentially. Possible transitions are also handled by this component, which performs the necessary operations. Furthermore, the controller keeps track of the execution time for each phase. A timer is started before each phase is started, and stopped after it has finished. This allows simple statistics to be printed out for the user.

When the application is started, the provided input data is verified. The mixing ratios are further validated, as they are required to add up to a total of 1, or 100%. Afterward, the import of the data is started. This process is explained in detail in Section 3.4.5.

When the import is finished, the simulation loop is started. During this phase, the main controller is responsible for triggering the physics simulation. For each iteration, it calls the appropriate function to advance the generation process. As the automatic physics execution is disabled, the internal physics update needs to be called explicitly. This is done in the *FixedUpdate* function. As this event function is called on a fixed time interval, the behavior closely relates to the normal Unity event flow illustrated in Figure 3.6. The simulation loop is handled in Section 3.4.7. When the maximum number of iterations is reached, the controller switches to the post-processing phase.

The cleanup stage is the first of two post-processing stages. Here, the generated road network is validated by fixing any coherency issues. This is done through the use of an asynchronous function. After the cleanup is finished, a callback event is emitted that notifies the main controller. During the second post-processing stage, possible artifacts are removed from the resulting road network. This will be explained in more detail in Section 3.4.8.

When processing is finished, the main controller triggers the output. The road network is exported as an image file in the PNG format. The alpha channel is used for transparency, which facilitates further use. The output can also be manually triggered by the user. By pressing the *F9* key at any time during processing, the current node and edge positions are exported.

3.4.5 Map Import

The data import is triggered by the controller, with the source file paths provided by the user. Information about nodes and edges is extracted from each of the XML files. The XML library provided by .NET allows the use of *XPath* expressions [CD99]. This query language can be used to address nodes in an XML document using path-like expressions. It is assumed that the data has already been filtered using the recommendations provided in Section 3.2.2. Therefore, no more refinement is necessary at this stage. A list of nodes is obtained using the expression `/osm/node`, whereas edges are obtained with `/osm/way`. The data obtained this way is then transformed into an internal data structure.

In addition to that, the boundaries of the selection are obtained from the data. The bounding box is necessary to convert the location information from geographic coordinates into Cartesian coordinates. This is achieved using the Mercator projection [Sny87]. A static class implements functions to convert latitude and longitude values into x- and y-coordinates on a plane. Furthermore, the relative positions of each node are calculated by subtracting the center point of the bounding box. As a result, the imported map can be placed at the origin of Unity's coordinate system.

In the OpenStreetMap data format, nodes are only linked to edges by tags containing their respective identifiers. This allocation is made explicit by assigning references to node objects to edge instances. As an intermediate step, a bidirectional graph is created out of the nodes and edges extracted from the source data. The motivation to store the objects in this data structure is the facilitated preprocessing. The *QuickGraph* library provides useful functionality, such as the calculation of a node's degree.

The generated graph is then used to create the actual node and edge objects used for the simulation. On initialization, the type of a node needs to be determined. A project titled *OSMnx* has addressed a similar issue in the past [Boe17]. The authors have developed a method to automatically plot city maps based on data from OpenStreetMap. For this purpose, they intended to simplify the road network by removing all OpenStreetMap nodes which did not represent vertices from a graph-theoretical point-of-view. This method has been adapted to our use case. We determine the classification of a node as follows: A node is considered a junction node if any of these conditions apply:

1. The node is an endpoint, i.e., the end of a dead-end street
2. A change from a one-way to a two-way street occurs at the node
3. More than two streets intersect at the node

Otherwise, the node is considered a waypoint. Figure 3.7 illustrates the classification of nodes. It can easily be seen that the number of waypoint nodes is relatively high in comparison with the number of junctions. Especially at roundabouts, it is common that many nodes are created to approximate the curvature of the roads.

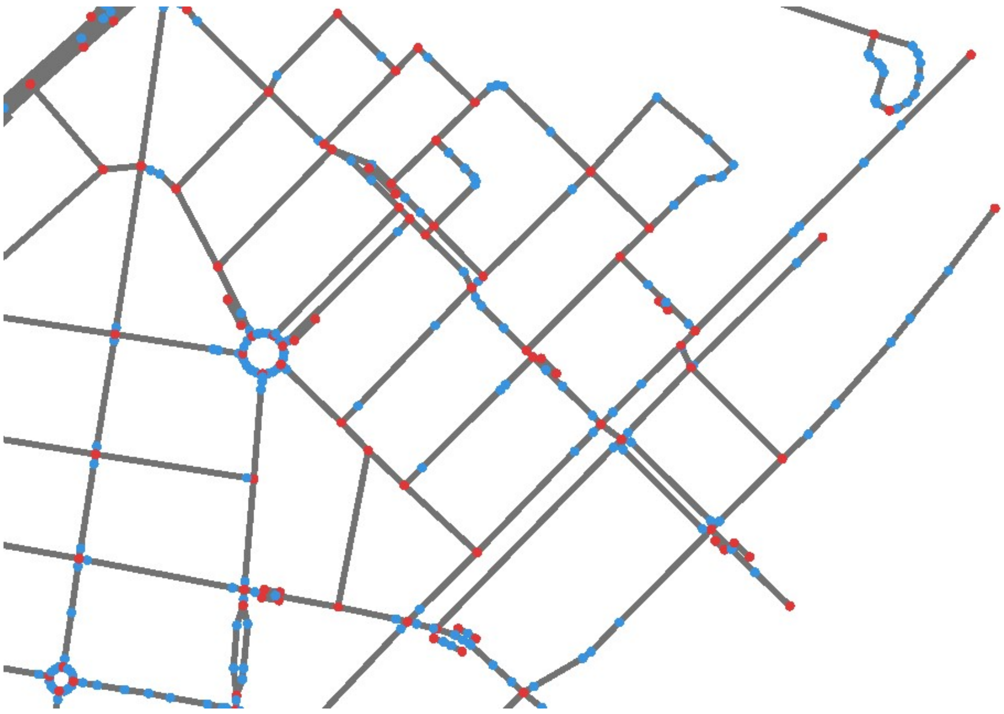


Figure 3.7: A visualization of the different node types. Junction nodes are highlighted in red, waypoint nodes are shown in blue.

3.4.6 Data Structure

The data structure used to store the road network information represents a directed graph. Node objects make up the vertices of the graph, while edge objects represent the links. Every edge connects exactly two nodes in a specified direction. For this reason, each edge carries references to its source and target nodes.

Similarly, nodes keep references to the edges they are a part of. The number of edges that start or end at a given node is not limited. However, duplicated edges between the same two nodes are not permitted, as this would make processing more complex. This also applies to road segments connecting the same nodes in opposite directions. If such a situation presents itself, only the edge that was first encountered is retained. The direction of roads is recorded, but only used for internal processing. To differentiate between incoming and outgoing edges, two distinct collections are maintained for each node. Furthermore, each node is of a specific type. In most cases where object-oriented programming methods are applied, such a classification can be modeled using inheritance. For our application, however, this would result in several difficulties. Waypoint nodes can be merged with other nodes during the simulation, which might result in a junction node. In such a case, the node type needs to change at runtime. If inheritance were to be used, this would require deletion and re-creation of the affected component. Likewise, the same would apply if a junction node were to change into a waypoint node when an edge gets removed. This would not be impossible, though it is generally advantageous to keep object initialization and destruction at runtime to a minimum, so as to reduce the performance impact caused by garbage collection. Therefore, a simple flag that specifies the node type is used. This is controlled by the node object itself, along with other properties that change together with the type.

In our application, nodes and edges are GameObjects with specific components attached. The attached components are in charge of different functionality and are decoupled from each other wherever possible. As such, they can easily be extended or replaced. The addition of new MonoBehaviours for added functionality can also be easily achieved. For nodes, the most notable component is *NodeBehaviour*. It holds the aforementioned references to the connected edges, as well as logic to change the node type. To determine the rank of a node during simulation, a *priority* field is used. This value is calculated internally, but it is made accessible via a public property. Similarly to nodes, edges hold an *EdgeBehaviour* component that contains references to the source and target nodes. It further holds the number of lanes of the represented road segment.

Figure 3.8 illustrates the relation between nodes and edges as a Unified Modeling Language (UML) class diagram. There is no complete collection of the whole network graph. Instead, the data structure is distributed across individual instances of nodes and edges. As a result, bidirectional references are necessary. The direction of edges does not represent the actual direction of travel in the real world. Therefore, the differentiation between incoming and outgoing edges would not strictly be necessary in most cases. However, this distinction is important for the presented system. This is caused by the use of spring joints, which will be detailed below.

Nodes and edges are generated by builder methods. In general, the builder design pattern is used to separate the construction of an object from its representation [GHJV95]. This separation has the goal of allowing the same instantiation process to be used for different representations. In our application, the motivation stems from the initialization process in Unity. Template objects are available for both nodes and edges, according to which new objects are created. These can be modified or replaced, if different properties are desired by the user. By providing builder components as a single point of contact, this replacement is very easy. The available prefabs are detailed in Section 3.4.10. Furthermore, the use of builders encapsulates the logic for instantiating new node and edge objects. This results in a looser coupling of the system, as changes can easily be made to the creation process by only changing a single script.

3.4.7 Simulation Loop

During this phase, the central algorithm is run. As the game logic is distributed across multiple components, several parts need to be considered. The *Update* and *FixedUpdate* methods provided by Unity are used as entry points. In general, the order in which these functions are executed in different scripts is arbitrary, as it solely depends on the order in which they are loaded. However, a strict order is necessary for our application.

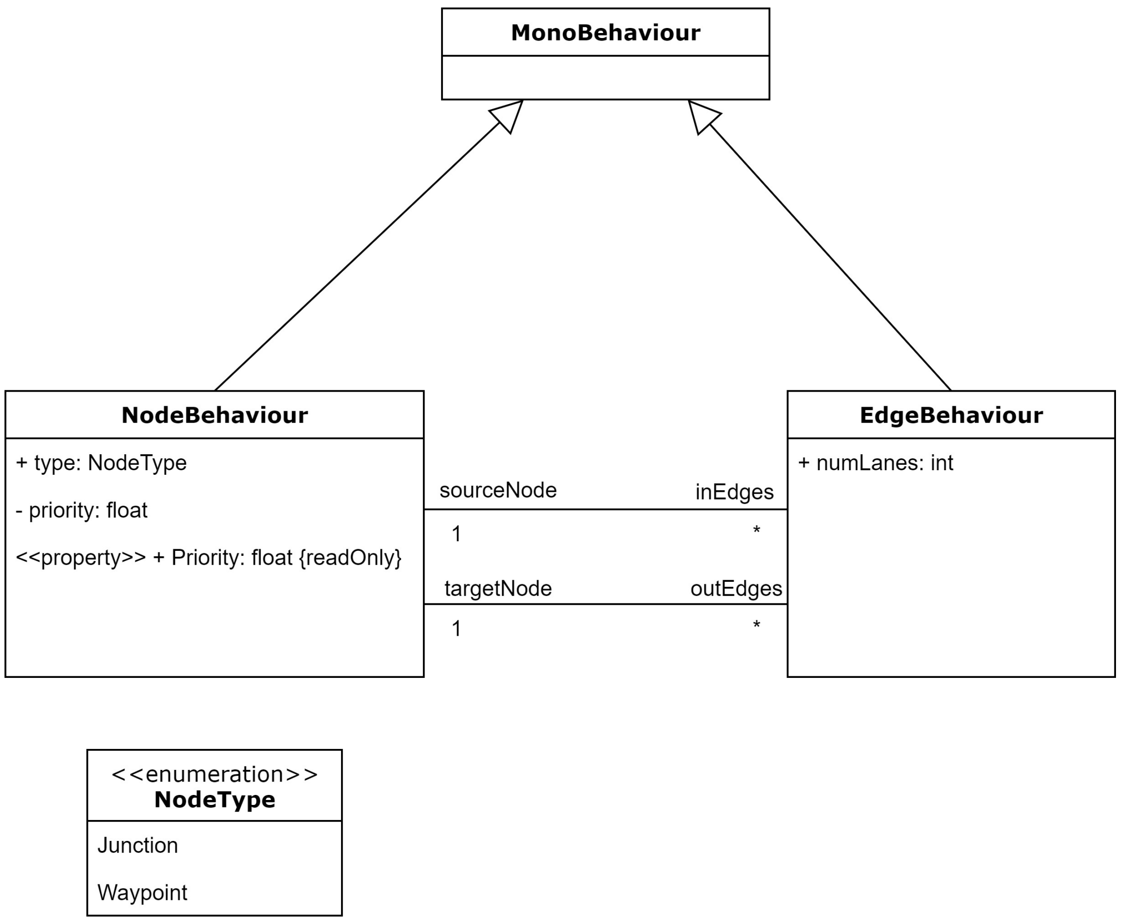


Figure 3.8: The relations between nodes and edges in UML notation.

Unity allows the specification of a custom execution order in the project settings. This is configured in the following way:

1. *Attractor*
2. Default
3. *NodeMerge*

First, the attraction calculation is run. The "default" entry includes all scripts that are not explicitly configured. This applies to the main controller, which triggers the physics simulation. Both attractive forces and spring forces are handled by the physics engine at this point. The script responsible for merging nodes if they overlap is executed last. This ensures that all other operations have finished at that point.

The *Attractor* script is responsible for calculating and applying the attraction forces to nodes. The component is applied to all nodes, but it is only enabled in junction nodes.

As the node type can change, it may get enabled or disabled for a given node after each iteration. The script uses a method provided by the *Physics2D* library to find nodes in its vicinity. This function returns all colliders within a given radius. Using a layer mask, the result can further be narrowed to only contain nodes on a different layer. For each node, an attraction force is then computed based on its physical properties. These are obtained from the *Rigidbody* component attached to each node. The most notable ones are mass and drag. The mass of both the attracting and the attracted nodes are directly integrated into the force calculation. The drag values, on the other hand, are solely used by the physics engine. These influence the amount of motion resulting from the application of a given force.

The calculation of the force is based on the equation for universal gravity. In theory, the gravitational constant is defined as $G = 6.67430(15) \times 10^{-11}\ m^3{\cdot}kg^{-1}{\cdot}s^{-2}$ [TMNT20]. Consequently, gravitational forces only have an impact when the masses are high and are negligible for small values. Unity generally uses a single-precision floating-point format to represent decimal values, which can lead to inaccuracies when very large or very small numbers are used [WFf11]. To work around this issue, the gravitational constant used in our application has been scaled up by a factor of 10^{13}. Furthermore, an additional *moveSpeed* factor has been introduced for more detailed adjustments of the force. This allows us to use smaller values for the masses of objects, while still keeping the same expected behavior. The function used to calculate the force is presented in Listing 3.3.

Listing 3.3: Implementation of the attraction force calculation

```
private void Attract(Rigidbody2D rbToAttract) {

    Vector2 direction = thisRigidbody.position - rbToAttract.position;
    float sqrDistance = direction.sqrMagnitude;

    // Prevent division by 0
    if (Mathf.Approximately(sqrDistance, 0f))
        return;

    // Newton's gravity with scalable gravity constant
    float forceMagnitude = (G * moveSpeed) * (thisRigidbody.mass *
        rbToAttract.mass) / sqrDistance;

    // Clamp force magnitude
    if (forceMagnitude > MaxForce) {
        forceMagnitude = MaxForce;
    }

    Vector2 force = direction.normalized * forceMagnitude;

    rbToAttract.AddForce(force, ForceMode2D.Impulse);
}
```

Gravitational forces, in theory, are not limited by distance. If implemented according to this notion, it would mean that all junction nodes would impact all other nodes in our application. This would result in a computational complexity of $O(n^2)$ for the attraction algorithm. As the number of nodes is relatively high (around 25000 for medium-sized city portions), a considerable amount of calculations would have to be done for each step. In practice, gravitational forces become negligible when the distance between the objects is substantial. This characteristic is used by our application to limit the radius in which gravitational forces are applied to $r = 200$. As a result, the strain on the physics engine is reduced considerably.

In certain cases, the calculated force can reach extremely high values. This can occur when two nodes are very close to each other, but their colliders don't intersect. If such a large force were to be applied to a node, the next physics simulation would cause it to move over its target. As discrete collision detection is used in our application, the physics engine would not detect any contact in such a case. To prevent this, the force is limited to a maximum value of $F_{max} = 500$.

For 2D physics, Unity supports two types of forces[8]: *Force* and *Impulse*. The former adds a continuous force to the object, while the latter applies an instant force impulse. Both modes take mass and drag into account. As the forces are recalculated in each iteration, there is no need to apply a continuous force. Instead, an impulse is applied.

The *NodeMerge* script detects collisions between nodes and combines them into one. It is attached to each node. Each node also has a collider attached to it that acts as a trigger. Consequently, the *OnTriggerEnter2D* event function is used to detect overlaps. This method is called each time there is a collision between its collider and another object. As a parameter, it provides information about the collided object.

When a collision between two triggers occurs, this function is executed for both objects. This could cause issues for the presented application, as merging can not be performed twice. To overcome this, collisions between nodes are cached. Each node holds a list of objects with which it has collided. On collision, this list is checked in the other object to verify that merging has not already been done. For example, assume two nodes A and B collide. First, the event function in node A is executed. Node A checks if it is contained in the list of collided nodes in node B, but no occurrence is found. Thus, it adds node B to its list and processes the collision. Then, the event function is called in node B. Node B checks node A's list and finds that it is present. Therefore, no more processing is done, as the nodes can already be assumed to be merged.

When nodes are merged, one of them is used as a base. The edges of the other node are then attached to it. The selection of the base node depends on the priority. The node with the higher priority stays in place, while the edges of the lower-priority node are detached and reattached to the base node. The other node is then destroyed. During this procedure, the layer of the base node may change based on the selection of a random process. The probability of selection for all considered layers is directly proportional to their respective mixing ratios. Thus, the node will have a higher chance of being

[8]`https://docs.unity3d.com/ScriptReference/ForceMode2D.html` (Accessed: 09.02.2020)

placed on the layer with a higher mixing ratio. This inclusion of randomness makes it a non-deterministic process.

To retain plausibility of the road network as much as possible, some edges may get deleted during merging. Duplicated edges are always removed. Assume that node A and node B collide, and both have an edge to node C. The merging process then only retains one edge between the combined node AB and node C. Furthermore, the angle between edges is an influencing factor. In the real world, it is uncommon to encounter junctions with extremely steep angles between roads. Similar to other approaches [VGDA+12, WMWG09], the allowed angle between two road segments is limited. For each edge to be attached to the base node, the angles to all other edges $\phi_1 \ldots \phi_n$ are calculated. If the lowest value ϕ_{min} is below a certain threshold, the edge is deleted. By default, the limit is set to $\phi_{limit} = 10°$ as this was found to produce satisfactory results. As a further benefit, this implicitly reduces the number of edges connected to a single node, which lowers the overall complexity.

The procedure of reattaching edges is performed by replacing the source or target node. During this process, the physical link needs to be updated. Due to the way that joints are implemented in Unity, the respective component is actually attached to the source node, not the edge itself. Therefore, the joint first needs to be deleted from the edge's source node, regardless of which node is replaced. It is then recreated to point from the current source node to the current target node. This causes the joint's properties to be reset and adjusted to the actual distance between the two nodes.

During the internal physics update, node positions are changed based on the forces exerted on them, as well as their physical properties. To calculate the counter-forces produced by the edges, the presented application relies on components available in Unity. This is achieved using instances of the*SpringJoint2D* class. Joints are generally used to physically connect two Rigidbodies. A spring joint, in particular, does not enforce a tight connection but allows both objects to move independently for a certain amount. However, it tries to maintain a constant distance between them. Forces are applied to both objects if they move further apart or closer together than the configured range. As the spring force is adjustable, the joint can be used to simulate stretching and compressing. Furthermore, the joint behavior is implemented to oscillate, which is not desired for this application. Through the configuration of a damping ratio, this effect can be reduced.

In theory, the spring joint should be attached to the edge GameObject. In Unity, however, one of the connected Rigidbodies is always the one attached to the same GameObject as the joint. The other object is freely configurable. For this reason, the spring joint cannot be placed on the edge but has to be added as a component to the source node's GameObject. Control over the joint is still maintained by the edge. It carries a reference to the component and handles initial creation and deletion. Even though strict encapsulation at this point is not possible in practice, this still keeps the responsibilities consistent with the original concept.

The forces exerted by the *Attractor* component as well as the spring joints are processed by the physics engine. In combination with individual physical properties, this results in the movement of nodes and extension or compression of edges. The *mass* property of a node influences the impact of forces. A node with a high mass is less affected by attraction and spring forces. On the other hand, a node with a lower mass will experience more movement for the same forces. Similarly, the *drag* property also regulates the resulting position changes. Drag is the tendency of an object to slow its movement and is generally caused by a fluid surrounding it [Fal11]. A higher drag value causes a node to be more stable in the given environment. Thus, forces need to be exerted on it for a longer time before movement occurs. Both of these properties are linearly interpolated for each node. The values are calculated based on the priority of a node, as well as the mixing ratio of the respective layer. The following formula is used to determine the interpolation percentage t_i for a specific node:

$$nodePriority_i = \frac{numLanes_i}{maxLanes}$$

$$t_i = nodePriority_i \times mixRatio_l$$

Here, $numLanes_i$ is the highest number of lanes found in any connected edge, $maxLanes$ is the predefined maximum number of lanes to be considered, and $mixRatio_l$ is the mixing ratio of the layer which the node is on. The maximum number of lanes is defined as $maxLanes = 10$, as roads with more than 10 lanes are rarely encountered. The calculated percentage is then used to determine the interpolated values for mass and drag. The respective value ranges are $[1, 100]$ for mass and $[1, 50]$ for drag, as these intervals have been found to produce satisfactory results. As a result of this interpolation, the priority of an individual node, as well as the mixing ratio of its layer, heavily influences its stability. The same value is also used to control the attraction force exerted by a node. Similarly, it is interpolated on a linear scale between two values. As a result, higher-priority nodes are more likely to cause other nodes to move towards them.

The edges of a node may change during the simulation due to merging, which would possibly make the physical properties invalid. Furthermore, the type of a node may change between junction and waypoint. Therefore, these values need to be recalculated each time an edge is added to or removed from a node. This is achieved through the use of event functions. A function *OnEdgesModified* is available in nodes and is called every time such a change occurs. This method not only recalculates mass, drag, and attraction force but also verifies if the node type is still valid. If that is not the case, the appropriate measures are taken to change it. These include disabling of the *Attractor* component and changing the rendering.

3.4.8 Post-Processing

After the simulation loop is finished, the resulting road network might contain undesired structures, such as invalid crossings between roads. These issues are handled by a *PostProcessor* component attached to the main controller's GameObject.

Post-processing is done in two stages. In the first stage, the aforementioned crossing issues are resolved. This is done using logic based on ray casting, a technique commonly used for 3D rendering. In general, rays are cast into a data set from an origin position [RPSC99]. The intersection point with the data is recorded, and measurements are taken. This technique is applied by our application to find crossings between roads, as illustrated in Figure 3.9. For each edge, a ray gets cast from its source node to its target. If another road intersects it, a hit is registered. It returns details about the overlapping object, as well as the location. To fix this issue, two resolution options are available.

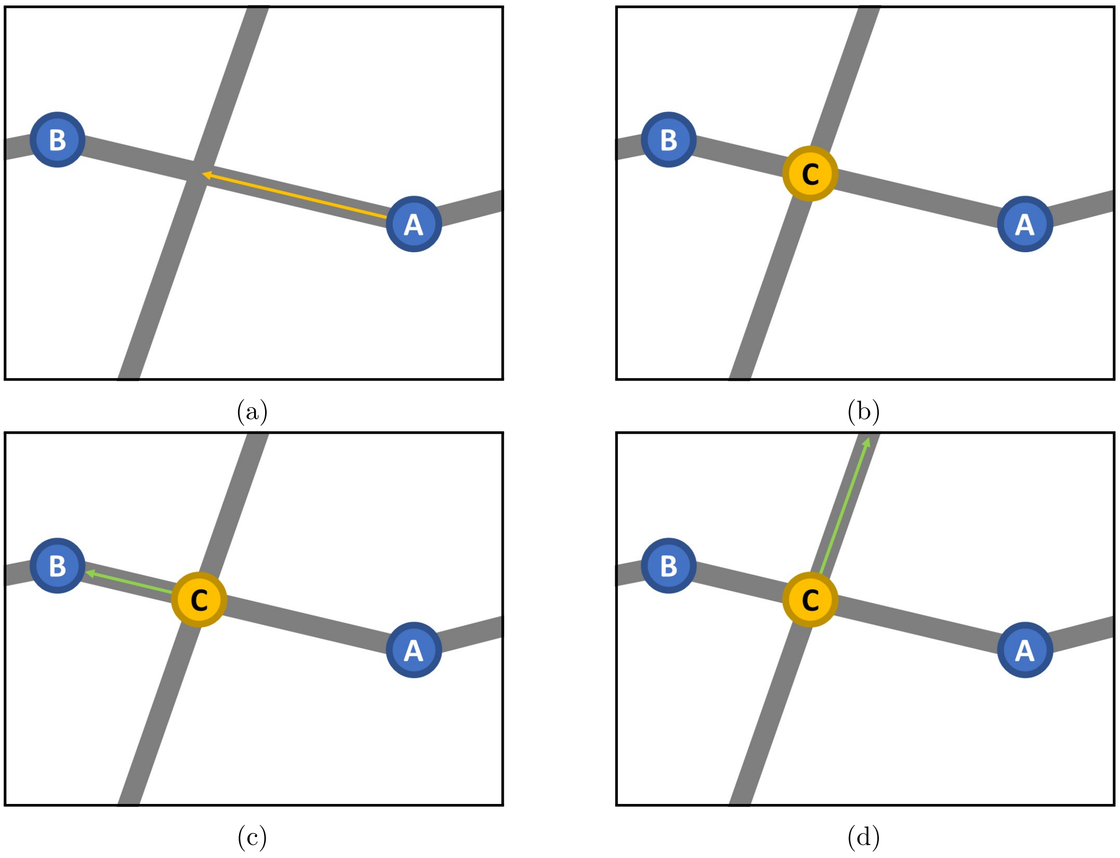


Figure 3.9: An illustration of the post-processing procedure. (a) A ray is cast from node A to node B. The ray hits another edge. (b) A new node C is added in between. New edges are created accordingly. (c)(d) Ray casts are performed for all added edges, sequentially.

In the first method, a new node is inserted at the intersection. Both edges are split into two new edges, respectively. As ray casts only return the first hit, multiple iterations may be necessary to resolve all issues. Therefore, all edges are first added to a stack. Validation is then done sequentially for all items on the stack. In each iteration, the first item is removed and ray casting is performed. If an issue is found and the edge is split, all new resulting edges get added to the stack. This procedure is repeated until the stack is empty.

The second mode simply deletes one of the two edges. This can be advantageous in areas with a high number of nodes and edges, so as to reduce the number of objects. This method has been found to produce more plausible results in such sections, and decrease the overall complexity of the road network. Our application supports the usage of both methods in parallel. The selection for how an intersection should be resolved is made based on the local density. To calculate this value, the number of objects within a given radius is checked. It is determined as follows:

$$density = \frac{numObjects(r)}{r^2\pi}$$

If the density is below a certain threshold, a new node is added at the intersection. If it is higher than the threshold, one of the two edges is deleted. This density limit is configurable and should generally fall within the range of 0.001 to 0.01. The selection of the edge to be removed is based on the number of lanes. Wider roads are considered more important, so the edge with more lanes is preserved, while the other one is removed. If both edges have the same width, the decision is made randomly based on the mixing ratios.

It is also possible for rays to hit nodes that lay on top of an edge but are not connected. In such a case, the edge is split in two at the node. The edge's source node connects to the intersecting node, and a new edge is created from there to the original edge's target. Because node colliders have a certain radius to it, this may result in a slight bend in the road if the node does not lay perfectly in the center of the edge.

The motivation to perform ray casts instead of other collision checks comes from the ease of use and the performance advantages. Collision callback methods require the physics update to run, as can be seen in Figure 3.6. As this would result in further node movement, it is not desired at this point. Another option would be to use static overlap checks using, for instance, *Physics2D.OverlapBox*. However, the performance is much worse than for ray casts. Individual overlap checks also exist, in the form of *Collider2D.OverlapCollider*. This function returns an array of all collider overlaps for the current object, but does not include the overlap positions. As those are required to create new nodes, this method is not suitable for the proposed application.

During the algorithm run, but especially during the first post-processing stage, artifacts may present themselves in the road network. Through the connection of crossing edges via a node, triangle-shaped structures can appear. These are made up of two junctions that are connected directly, but also via a waypoint node. The patterns represented by these do not frequently appear in real cities, as they provide somewhat redundant connectivity. Furthermore, the removal of individual nodes decreases the overall complexity of the result. In the second post-processing stage, these artifacts are removed. For each node, a check is performed to determine if it is connected to exactly two other nodes, both of these are edges, and they are also connected. If this is the case, the node gets removed. This process is illustrated in Figure 3.10.

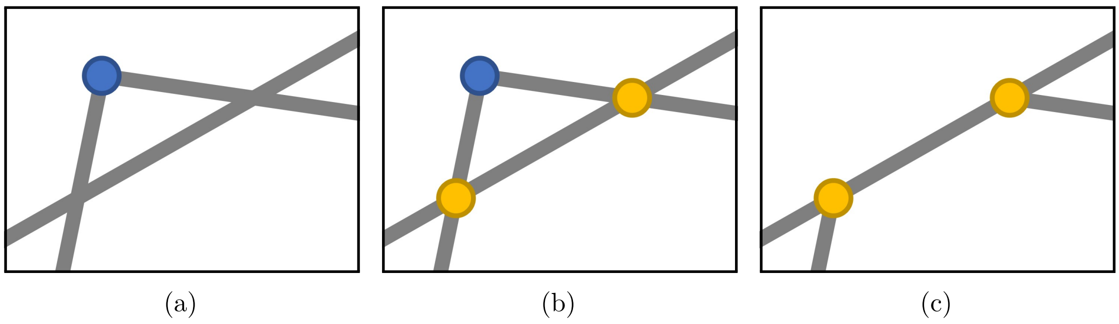


Figure 3.10: An example of the artifact removal process. (a) An edge crosses over other edges. (b) Nodes were added during post-processing to connect the respective edges. A triangle shape is formed. (c) The waypoint node was removed.

3.4.9 User Interaction

During the procedural generation process, the user can provide direct input to the system. As the application runs in the Unity Editor, he or she may manipulate objects using native tools. These changes are applied directly to the internal representation. Attraction and spring forces still have the expected effects. If the user moves a node, connected edges get stretched. As this results in forces being applied to affected nodes, the road network will adjust itself to the change.

By moving a node close to another node, the user can cause a collision event, which in turn will cause the nodes to be merged. Thus, the road network can be shaped extensively by the user. As edges are defined solely by their source- and target nodes, they cannot be modified directly.

3.4.10 Components and Prefabs

The application logic is distributed across multiple components, classes, and MonoBehaviours. These were created according to the component architecture shown in Figure 3.5, Section 3.3.4. The individual entities will be described below. Table 3.1 provides an overview of the created components. All of these inherit from Unity's MonoBehaviour class and are attached to GameObjects. An overview of all GameObjects with their attached components is shown in Figure 3.11.

Furthermore, a few classes were created to provide the required functionality. These are shown in Table 3.2. It includes classes originally defined by the *real-world-map-data*[9] project described in Section 3.4.2, which were modified for this project.

[9]`https://github.com/codehoose/real-world-map-data` (Accessed: 11.02.2020)

Component	Description
CityMergeController	Receive user input and configuration, control program flow
NodeBuilder	Create new nodes, apply properties from prefabs
EdgeBuilder	Create new edges
Attractor	Apply attraction forces to nodes in vicinity
NodeMerge	Combine two nodes on collision
PostProcessor	Perform post-processing and artifact removal
OutputController	Produce output file

Table 3.1: Components created for the CityMerge application.

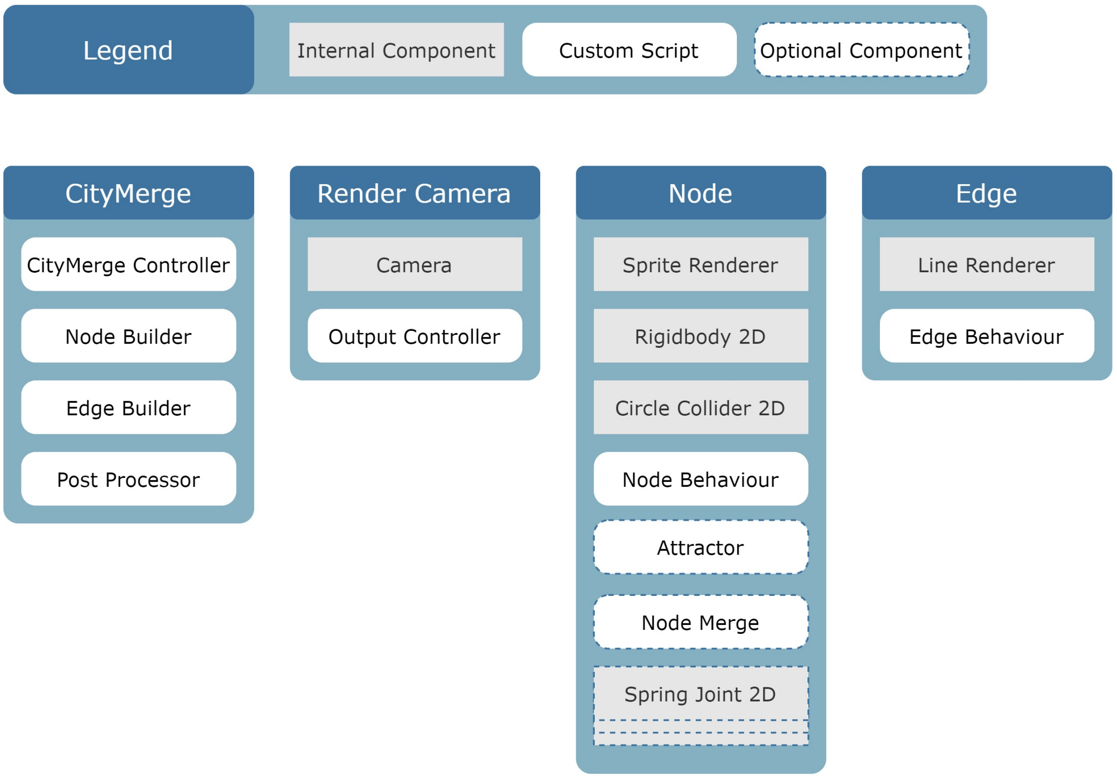


Figure 3.11: An overview of the types of GameObjects used in the application, with their respective components.

To facilitate the initialization of new GameObjects, Unity provides the *Prefab* system. It allows the creation and storage of GameObjects with all their attached components as reusable assets. Prefabs can be used to instantiate preconfigured objects at runtime, as well as to easily set up a scene in the editor. The supplied prefabs mostly correspond to the different object types shown in Figure 3.11.

The *CityMerge* prefab is provided for convenience. It includes all required components to

Class	Description
MapReader	Read input files, generate initial road networks
MercatorProjection	Transform geographical positions into Unity coordinates
GraphOperations	Provide static functions to modify the road network graph
Density	Provide functionality to calculate the area density
InterpolationValues	Hold value ranges and interpolate properties
LayerMaskUtils	Compute layer masks
Log	Provide extended logging functionality
ListExtensions	Extend the *List* class
StackExtensions	Extend the *Stack* class
DebugDraw	Visualize ray casts and hits
BaseOsm	Base for OpenStreetMap data, based on *real-world-map-data*
OsmBounds	Map boundaries, based on *real-world-map-data*
OsmNode	Node representation, based on *real-world-map-data*
OsmWay	Way representation, based on *real-world-map-data*

Table 3.2: Custom classes created for the CityMerge application.

control the application. This prefab is intended to be used to set up a new scene in which to run CityMerge. A *RenderCamera* child object with a *Camera* and an *OutputController* script is also part of this prefab.

The *NodePrefab* and *EdgePrefab* assets are used as templates by the application. They include all the necessary behavior components for nodes and edges, respectively. In the *NodeBuilder* component, the prefab used to generate new nodes can be changed. It is thus possible to create a custom template for nodes, as long as the required components are applied. For example, the desire for more elaborate rendering might motivate a switch to another prefab with a modified rendering component. The same applies to edges, for which the template can be changed in the *EdgeBuilder* component.

3.4.11 Performance Tuning

Several measures have been taken to reach and surpass the performance requirements defined in Section 3.3.3. One of these is the usage of a two-dimensional scene setup. This makes it possible to rely on a 2D physics engine instead of Unity's 3D engine, as it generally performs better. The computational complexity of collider checks in two dimensions is lower than in 3D.

Inside the main loop, the *Attractor* component runs for all junctions in each iteration. Therefore, it was optimized as much as possible. To find nodes in the vicinity, the function *Physics2D.OverlapCircleNonAlloc*[10] is used.

[10]`https://docs.unity3d.com/2019.2/Documentation/ScriptReference/Physics2D.OverlapCircleNonAlloc.html` (Accessed: 11.02.2020)

Compared to the default circle-overlap method, an array has to be provided as a parameter, which will then be filled with the results. Thus, no memory is allocated, which significantly improves garbage collection performance. Furthermore, the force calculation is optimized. Vector magnitude calculation performs a square-root calculation, which is computationally intensive. This is prevented by directly using the squared magnitude of the direction vector, as can be seen in Listing 3.3.

To improve rendering performance, custom material assets are used. These use an unlit shader, which does not include any light and reflection calculations. This results in faster drawing times, without any noticeable drawbacks. Since the focus of our application is not visualization, this does not impact the operation and may even be beneficial for users due to the reduced visual complexity.

As the application is based on Unity, the application design was limited by some characteristics in regards to performance. One of these factors is the single-threaded nature of Unity. While some functionality is executed on worker processes, most operations run on a single main thread. Additional CPU cores thus provide little performance advantage. To reduce this limitation, a new tech stack has been introduced for Unity, which explicitly supports multi-threading [Mei19]. In combination with the Unity Entity Component System (ECS)[11] and C# Job System [12], it promises to deliver higher performance through parallelization. At the time of implementation, these features were not yet finalized and therefore not recommended to be used for final products. Therefore, these concepts have not been used in the presented application.

[11]`https://docs.unity3d.com/Packages/com.unity.entities@0.5/manual/index.html` (Accessed: 11.02.2020)

[12]`https://docs.unity3d.com/2019.2/Documentation/Manual/JobSystem.html` (Accessed: 11.02.2020)

CHAPTER 4

Results

4.1 Generated Layouts

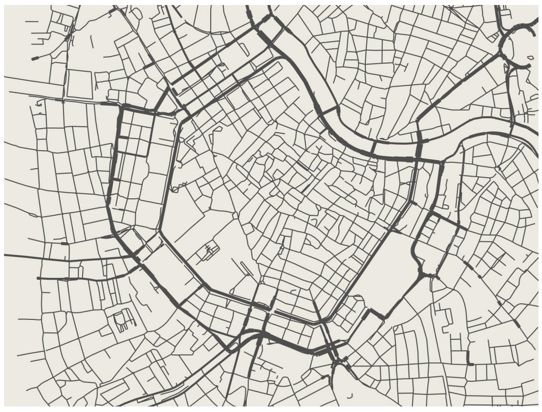
The output of the CityMerge application is a city layout that contains features and properties of both input maps. To assess the results, the program was run with different data sets, obtained according to the method described in Section 3.2.2. A few of these are highlighted below. Although the processing of large data sets is possible, it is very demanding in terms of resources. For this reason, and for improved visibility, only sections of cities are used for these visualizations.

4.1.1 Example 1: Vienna and Paris

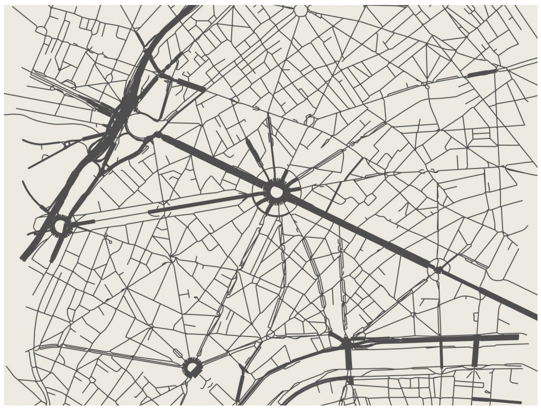
The application was run with a section of Vienna (Austria) that contains its recognizable *Ring* road and a section of Paris (France), centered around the *Arc de Triomphe.* Both cities contain easily identifiable features, as can be seen in Figure 4.1. The presented section in Paris shows a radial pattern, with roads stretching outwards from the center. Apart from that, smaller, star-shaped structures can be identified. In Vienna, on the other hand, the city center shows a relatively irregular pattern. However, a large road circling the inner city is present. Both cities feature a high road density in the selected areas.

Figure 4.2 shows the result of a procedural generation process using the aforementioned layouts as input. Many distinct features originating from both cities can be identified. For example, the large diagonal road is still present, as well as the general shape of the circular road around the center. However, some structures have been lost, such as certain long diagonal roads. The general road density is relatively high in the result.

The same input data was used to run the procedural generation process with a different number of iterations. As can be seen in Figure 4.3, this affects the result. By performing more iterations, the layout gets more distorted. For example, the roundabout in the



(a) Input map for Vienna



(b) Input map for Paris

Figure 4.1: Input data sets for Vienna and Paris, provided for reference.

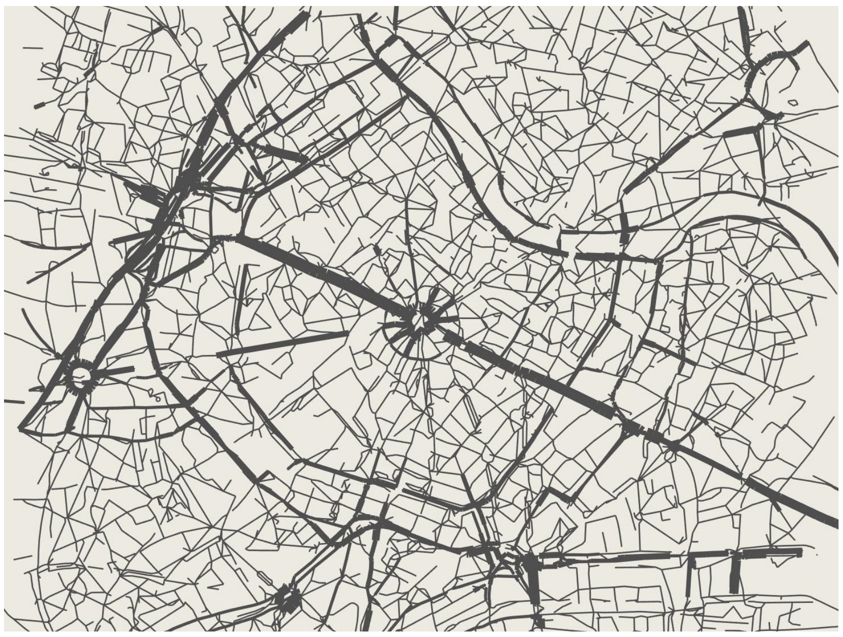


Figure 4.2: The result of a CityMerge application run using input maps from Vienna and Paris. Mixing ratios $mix = 0.5 : 0.5$, number of iterations $n = 200$

center is barely recognizable after 1000 iterations. As a result, fewer features from the input maps are present in the output.

4.1.2 Example 2: Barcelona and Lisbon

Further examples are presented that use the layouts of Barcelona (Spain) and Lisbon (Portugal) as a base. The extracted section of Barcelona is laid out according to a strict

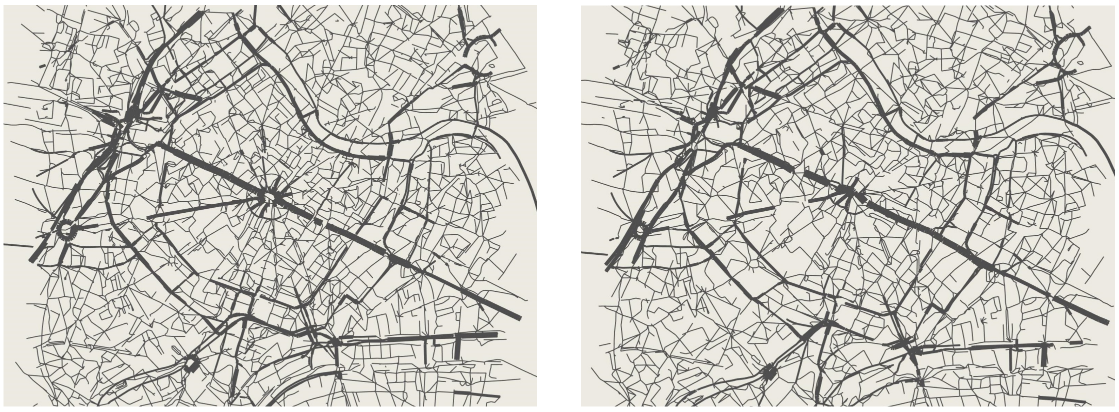

(a) Result for $n = 100$ (b) Result for $n = 1000$

Figure 4.3: Results for Vienna/Paris, using different iteration counts with $mix = 0.5 : 0.5$.

gridiron pattern, as can be seen in Figure 4.4. Several major roads can be identified, with one of them crossing diagonally. In turn, the layout of Lisbon is very irregular, but a large coastal road can be recognized.

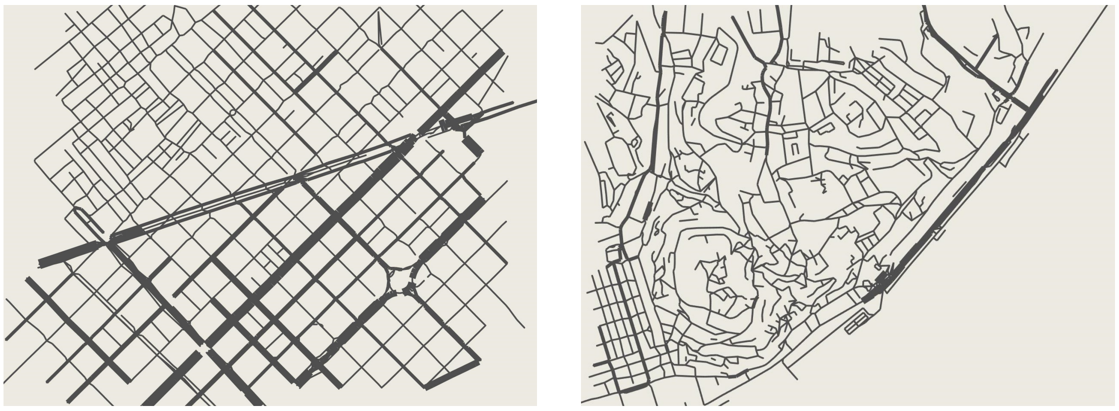

(a) Input map for Barcelona (b) Input map for Lisbon

Figure 4.4: Input data sets for Barcelona and Lisbon, provided for reference.

The result, shown in Figure 4.5, has retained most of the large roads present in the Barcelona data set. Similarly, the organic pattern found in Lisbon can also be seen. The diagonal coastal road has shifted further to the middle, which was caused by the relative positioning of the two layouts.

Figure 4.6 shows the result of procedural generation processes performed with different mixing ratios. A higher mixing value for Barcelona results in a layout with most of the grid structure still intact. On the other hand, a high ratio for Lisbon shows a more distorted road network.

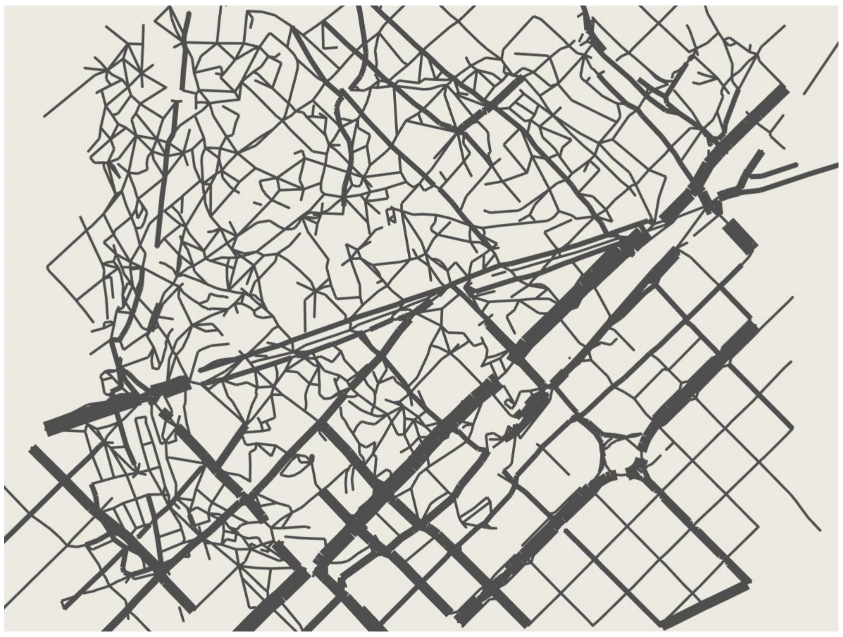
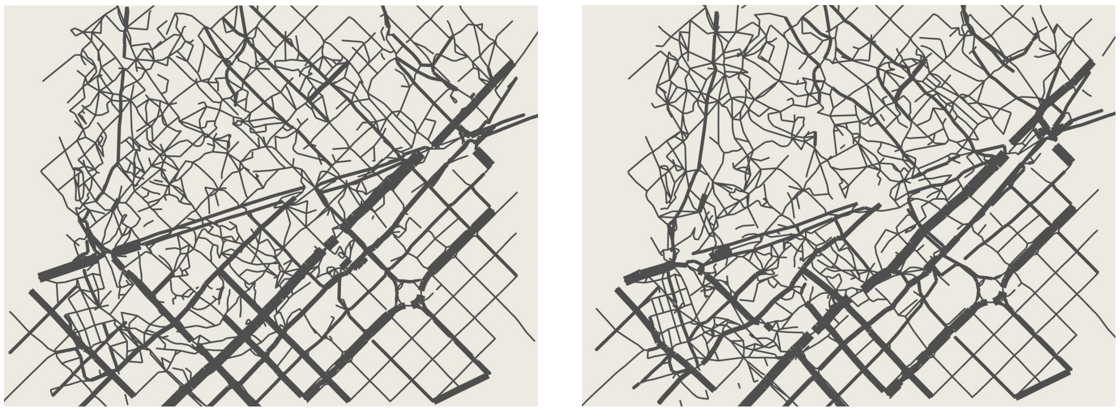


Figure 4.5: The result of a procedural generation process using input maps from Barcelona and Lisbon. $mix = 0.5 : 0.5$, $n = 300$



(a) Result for $mix = 0.8 : 0.2$

(b) Result for $mix = 0.2 : 0.8$

Figure 4.6: Results for Barcelona/Lisbon, using different mixing ratios with $n = 300$.

4.1.3 Pre-filtered Input Data

The above examples have shown results for procedural generation processes where complete layout portions of cities have been used as input. In this sense, "complete" refers to the inclusion of all relevant roads. However, it is also possible to explicitly perform filtering on the data, before using it as input for the CityMerge application. An example of this is shown in Figure 4.7. The layouts of Vienna and Paris have been used as source maps. The road network of Vienna is unchanged to the depiction in Figure 4.1a. However, only the major roads are utilized from the Paris layout. This has been achieved through a modified Overpass QL query. The filter function shown in Listing 3.2 has been adapted to only include roads classified with a priority of *secondary* or higher.

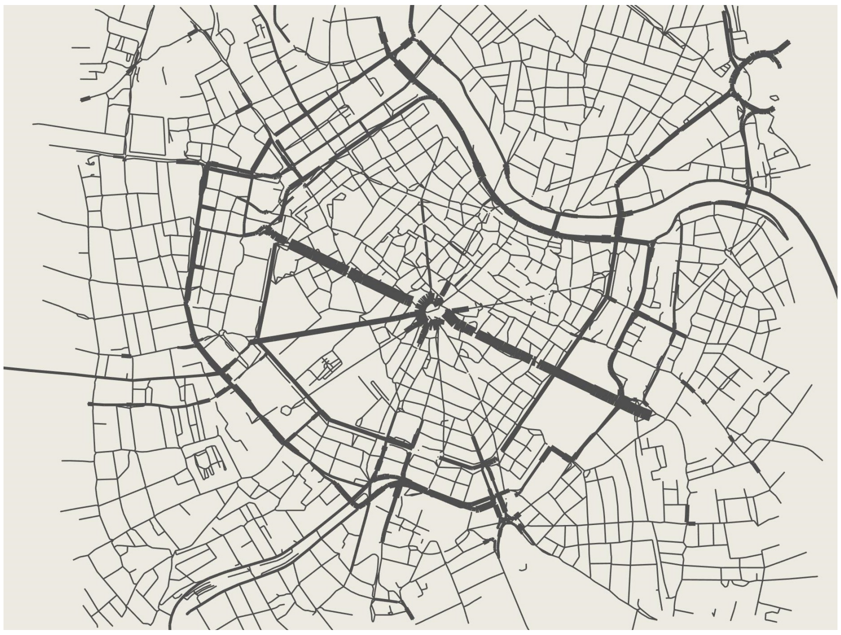


Figure 4.7: The result of a procedural generation process using the layout of Vienna and a pre-filtered map of Paris. $mix = 0.5 : 0.5$, $n = 100$

The result shows the road network of Vienna combined with the arterial roads found in Paris. Individual road segments originating from the different data sets have been connected by the algorithm. However, some details have been lost, such as parts of the roundabout in the center.

4.2 Performance

The performance of the proposed algorithm was evaluated by taking time measurements of generation processes. The duration of individual stages is timed and printed directly by the CityMerge application. As a test system, a PC with the specifications listed in Table 4.1 is used. The system is comparable with a current mid-level device.

Time measurements were taken for the example introduced in Section 4.1.1, which used Vienna and Paris as input cities. The presented maps each cover an area of about 10 km^2. In total, the layouts used as input contain 15 764 nodes and 17 717 edges. For $n = 200$ iterations, the process has been run 10 times. The average total processing time is $\mu = 119.6$ seconds, with a standard deviation of $s = 8.1s$. Most of this time is spent on the simulation, which was measured as $\mu = 91.4s$ with $s = 6.5$.

Operating System	Microsoft Windows 10 Home
Software Environment	Unity 2019.2.11f1 Personal
CPU	Intel Core i7-6700HQ @ 2.60 GHz
RAM	16 GB
Graphics Adapter	NVIDIA GeForce GTX 960M

Table 4.1: Components used for performance evaluation.

To determine the correlation between the number of iterations and the resulting processing time, more measurements have been taken. These are shown in Table 4.2. Figure 4.8 shows that the duration rises linearly with an increased number of iterations. However, the first few iterations have been found to take considerably longer. This is most likely owed to the high number of collisions occurring as a result of the city overlay.

Iterations	Processing Time [s]
10	57.8
50	70.8
100	86.8
200	119.6
400	167.8
600	228.7

Table 4.2: Simulation time measurements taken with different iteration counts.

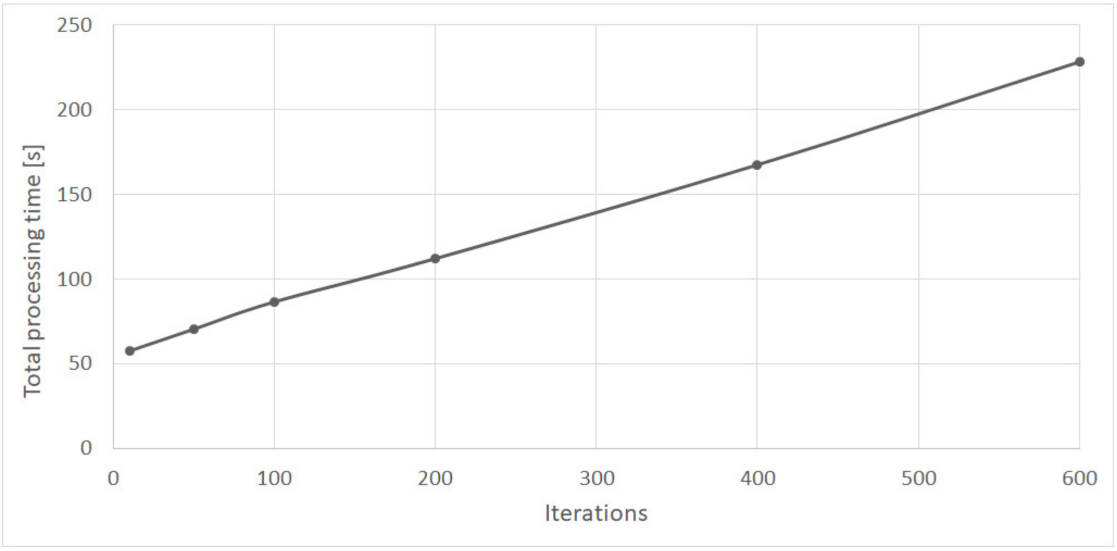


Figure 4.8: The correlation between the number of iterations and the resulting simulation time.

To discover the effect of the number of nodes on the simulation time, measurements have been taken for larger city portions. Two sections of approximately 38 km^2 have been used as input, which contain 55 508 nodes and 62 351 edges in total. The complete process, using 200 iterations, was finished after less than 9 minutes. The same has been done for very large city portions with a size of approximately 150 km^2. The combined node count was 184 166, and the layouts contained 204 656 edges. At just over an hour for 200 iterations, the process takes considerably longer if such a high number of objects is involved. Figure 4.9 shows a visual representation of this correlation.

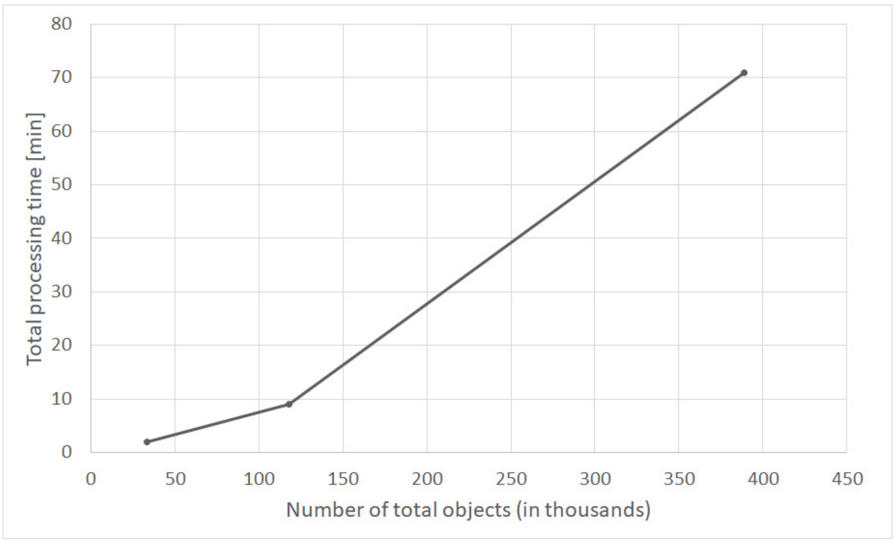


Figure 4.9: The correlation between the total number of objects (nodes and edges) and the resulting simulation time.

4.3 Subjective Evaluation

To assess the quality of the results produced by the presented algorithm, an evaluation is performed by conducting a user study. As no objective measures are available to our knowledge, we rely on subjective judgments of user preference. The goal is to determine if the generated layouts are plausible and to discover how the results compare to more established techniques. Furthermore, we aim to verify the influence of different patterns present in input maps on the result. The recognizability of features from the origin layouts is also evaluated.

4.3.1 Method

A survey was performed with 53 participants. It was specifically targeted towards nonprofessionals in urban planning or similar domains. A questionnaire was created on the online platform *Google Forms*[1], and the link was sent out to interested individuals. No explicit preselection of participants was performed, as no previous knowledge is required. At the start of the survey, a short description explains the process, the aim, and the estimated duration.

4.3.2 Structure

The survey was divided into three distinct parts, each of which was designed to answer a specific question:

1. *Comparison*: How do the results of this algorithm compare to other approaches?
2. *Plausibility*: How plausible are the generated results?
3. *Recognizability*: How well can existing structures be recognized?

The sections are sorted according to this specified order. No shuffling is performed, as the questions presented in the plausibility section might influence a participant's answer in the comparison section.

Comparison

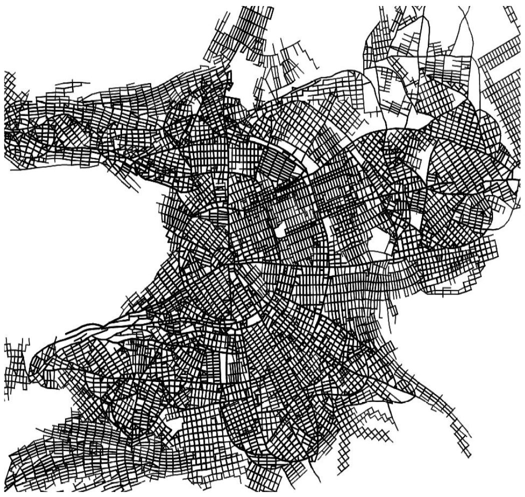
The participants are shown visual representations of city layouts that were produced by different algorithms. These were preselected to cover a wide spectrum of approaches. For each generation method, a representative algorithm was chosen. The image sources were hidden from the participants, to prevent any external factors from influencing the result. Instead, each image was assigned a letter ("Layout A" through "Layout D") to be able to link it to the answers. Furthermore, all images were converted to the same black-and-white color scheme. The following approaches were selected:

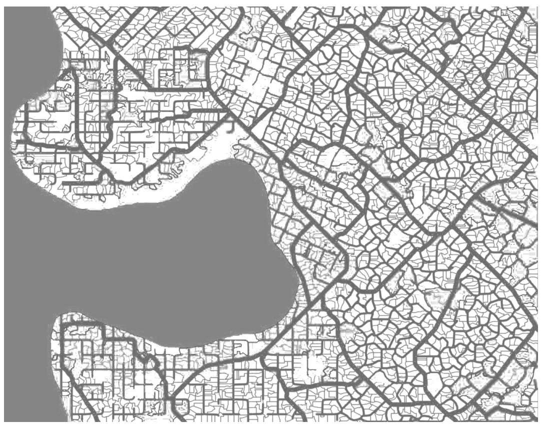
- Parish and Müller (2001) [PM01]: Grammar-based approach
- Lechner et al. (2007) [LWW$^+$07]: Agent-based approach
- Beneš et al. (2014) [BWK14]: Simulation-based approach
- Ours

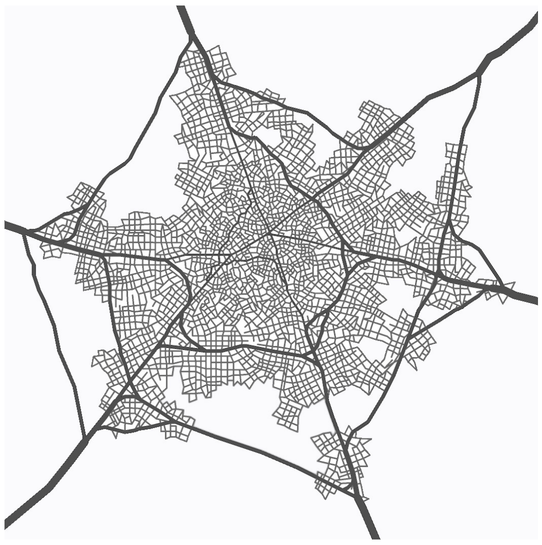
The selected layout created by Parish and Müller was produced specifically to represent an imaginary city [PM01]. Neither Lechner et al. nor Beneš et al. mention any specific real-life city that should be mimicked by their respective layouts, so the same can be assumed for these [LWW$^+$07, BWK14]. However, the road network generated by Lechner et al. was created with the intent to showcase a combination of grid- and organic patterns.

[1]`https://www.google.com/intl/en_us/forms/about/` (Accessed: 18.02.2020)

To represent our algorithm, a layout was generated using parts of Barcelona (Spain) and Prague (Czech Republic). Figure 4.10 shows the images used for comparison after the color changes were made.


(a) Layout A: Parish and Müller (2001) [PM01]


(b) Layout B: Lechner et al. (2007) [LWW$^+$07]


(c) Layout C: Beneš et al. (2014) [BWK14]

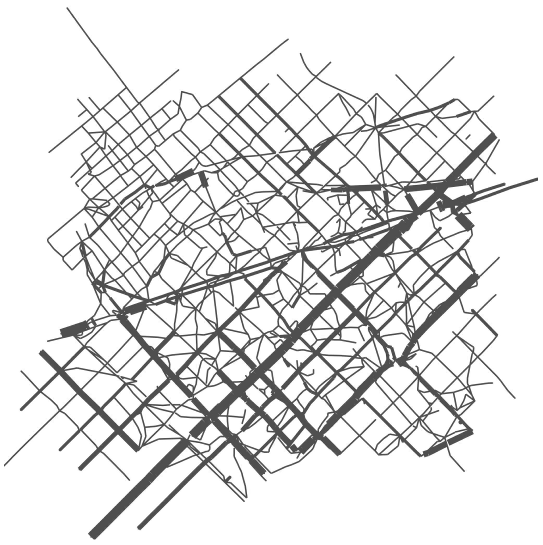
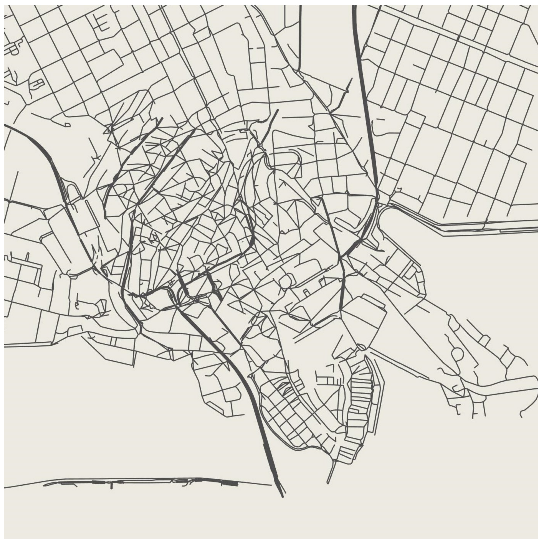

(d) Layout D: Ours

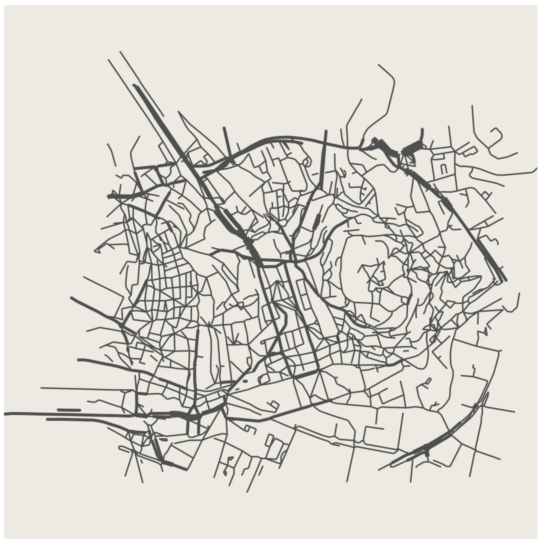
Figure 4.10: The images used for the comparison section of the user study. The images have been processed to achieve a uniform appearance.

The participants were asked to rate the layouts, based on how realistic they look to them. Answers were recorded through a five-point scale. Numbers ranging from 1 to 5 were presented as possible answers, with 1 labeled as "very unrealistic" and 5 as "very realistic".

Plausibility

In this section, the participants were shown three different images produced by the presented algorithm. The images were selected specifically to cover different combinations of road patterns. Thus, city layouts with the required patterns were used as input. Figure 4.11 shows the presented road networks.


(a) Layout 1: Stockholm/Amsterdam


(b) Layout 2: Budapest/Lisbon

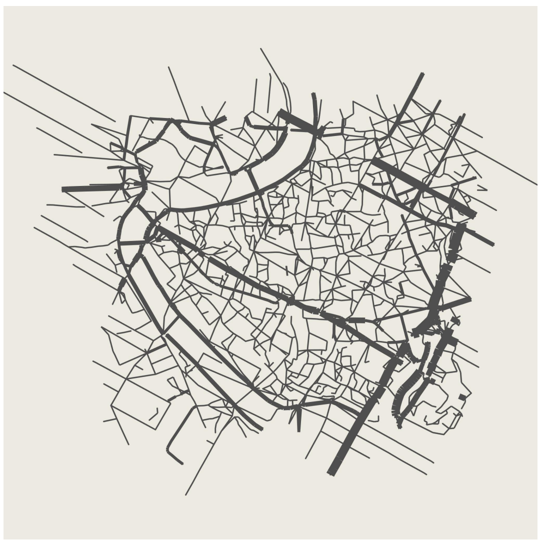

(c) Layout 3: Rome/New York City

Figure 4.11: The images used for the plausibility section of the user study.

Layout 1 was generated using parts of Stockholm (Sweden) and Amsterdam (Netherlands).

An irregular grid pattern can be found in Stockholm, while the selected section of Amsterdam contains a more organic layout. *Layout 2* was created from the road networks of Budapest (Hungary) and Lisbon (Portugal). Both of these cities feature an organic pattern. For *Layout 3*, sections of Rome (Italy) and New York City (USA), namely Manhattan, were used. The former is laid out in a very organic pattern, while the latter is built strictly along to a grid.

The names of these cities were not shown to the participants, as independent judgments were desired. To enable proper differentiation, layouts were named using numbers ("Layout 1" through "Layout 3"). Each layout was intended to be rated individually, based on the perceived realism. The same task definition and answer range as for the comparison section were applied.

Recognizability

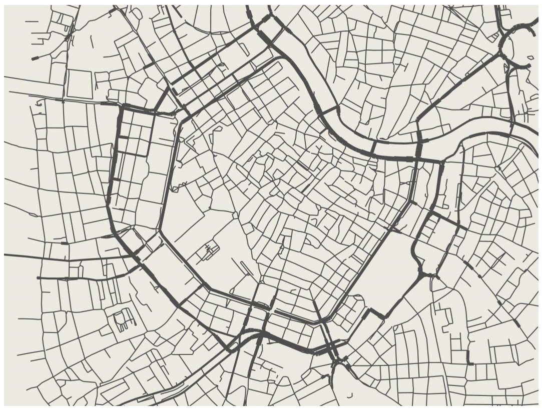
Participants were shown two sets of images, depicting the input and output of the presented application. Each set consisted of three individual city layouts: Two source layouts and a combined layout, which was the result of a generation process using the source maps as input.

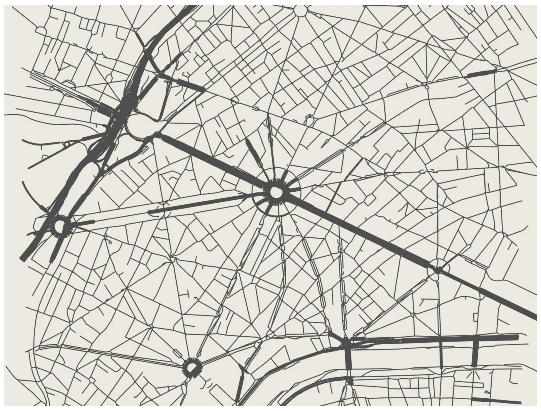
The first image set is shown in Figure 4.12. Layout portions from Vienna and Paris were used as input. These cities have been selected because they contain very distinct features. Vienna's circular road, as well as the radial roads in Paris, are both easily visible.

Figure 4.13 shows the images used for the second question. As input layouts for the procedural generation system, parts of Copenhagen (Denmark) and Barcelona (Spain) were used. Barcelona is made up of a grid structure with straight arterial roads. Copenhagen, on the other hand, features more curved roads.

For the second image set, custom mixing ratios have been selected. Copenhagen was assigned a value of 0.85, while 0.15 was used for Barcelona. This should allow the impact of mixing ratios on the recognizability aspect to be measurable.

The questions presented to the participants were structured as follows: *How well have the characteristics of <city> been preserved in the result?* For each of the four input city layouts, one such question was asked. As for the other sections, answers were given on a scale. However, a different configuration was used, as a five-point scale does not provide the required resolution to determine the impact of mixing ratios. Therefore, a ten-point scale was used, with the respective extremes defined as "very poorly" and "excellently".


(a) Input layout 1: Vienna


(b) Input layout 2: Paris

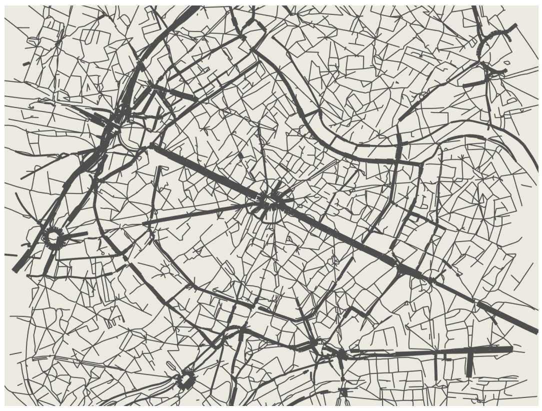
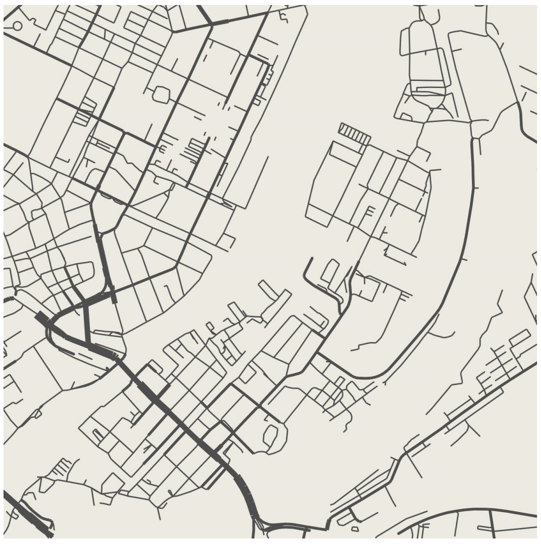
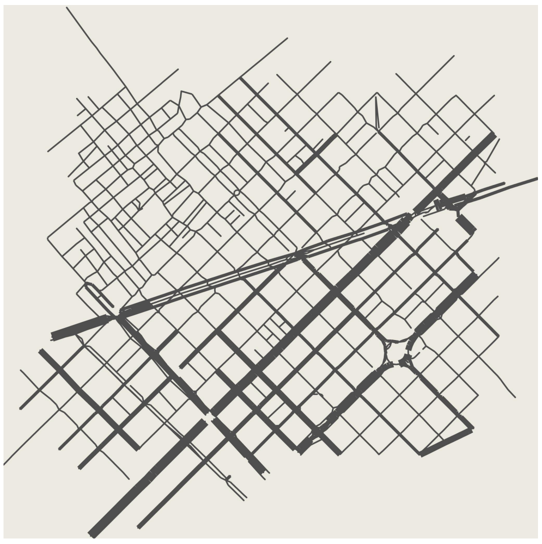

(c) Result layout produced using mixing ratios of 0.5 : 0.5

Figure 4.12: The images used for the recognizability section, part 1.


(a) Input layout 1: Copenhagen


(b) Input layout 2: Barcelona

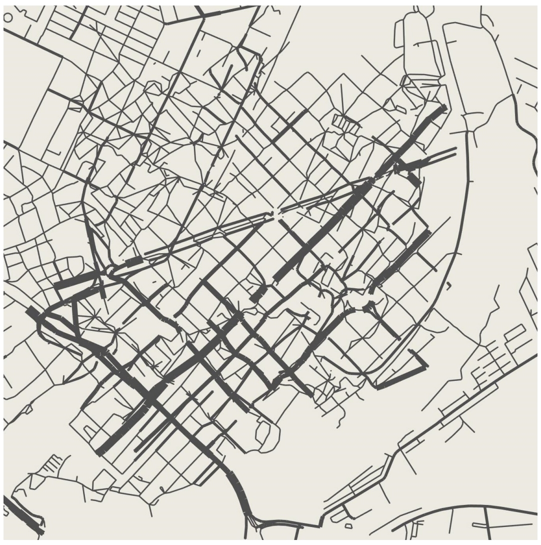

(c) Result layout produced using mixing ratios of 0.85 : 0.15

Figure 4.13: The images used for the recognizability section, part 2.

4.3.3 Outcome

Responses provided by participants have been recorded and analyzed using statistical methods. The resulting data is highlighted in this section, whereas the analysis of the results is presented in Section 5.1.

The first section of the survey aimed to compare the results of the presented algorithm to other procedural city generation systems. The results are outlined in Figure 4.14 in the form of a histogram. The highest number of "5" ratings was received by the layout generated by Lechner et al. However, perhaps somewhat surprisingly, this layout was also given a rating of "2" on many accounts. The layout generated by our application received the most ratings of "1", though it was also rated as very realistic by some participants.

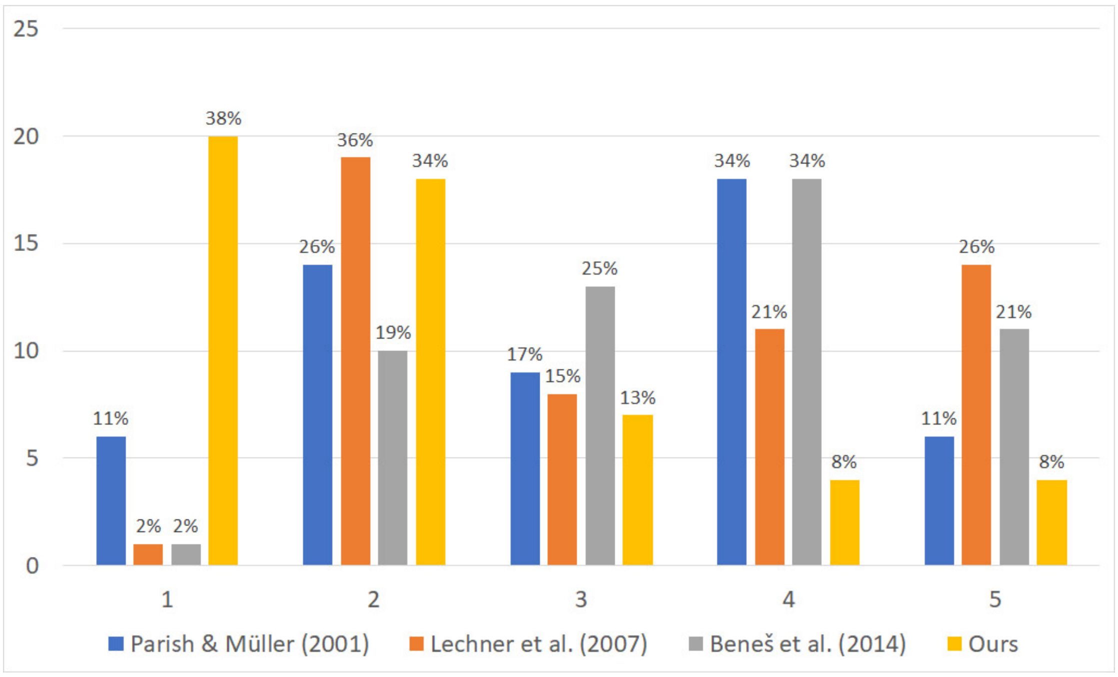


Figure 4.14: User ratings for the perceived realism of layouts generated by different systems. Participants were given the following task: "Please rate [the layouts] based on how realistic they look to you." The value "1" was defined as "very unrealistic" and "5" as "very realistic".

In an effort to make the values comparable, the mean and standard error for each question was calculated. These values are listed in Table 4.3. The bar chart shown in Figure 4.15 displays the average response for each layout within a 95% confidence interval. The highest average rating was given to the layout generated by Beneš et al., followed by Lechner et al., Parish and Müller, and our application.

In the plausibility section, participants were asked to rate the perceived realism of generated layouts. Individual measurements were obtained, as shown in Figure 4.16. The

System	Mean	Standard Deviation
Parish & Müller (2001)	3.08	1.24
Lechner et al. (2007)	3.34	1.27
Beneš et al. (2014)	3.53	1.08
Ours	2.13	1.23

Table 4.3: Mean and standard deviation of ratings for layouts produced by different systems.

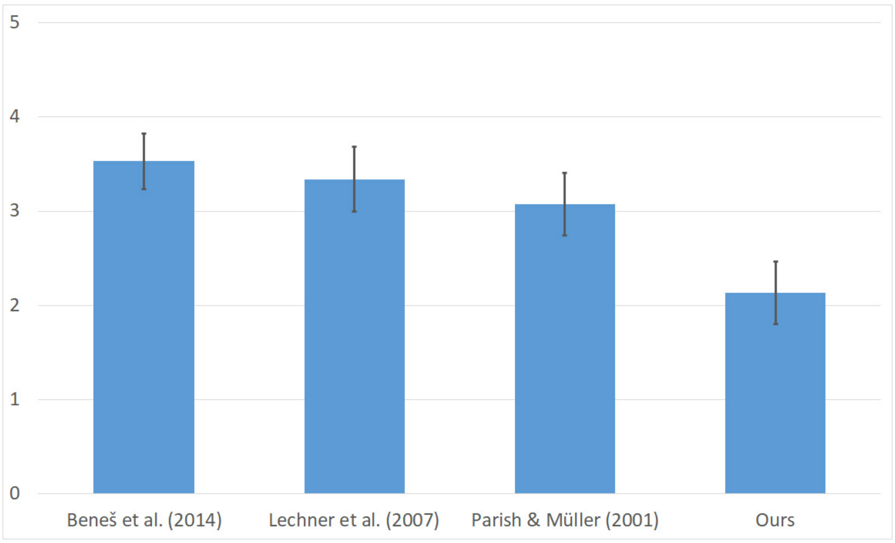


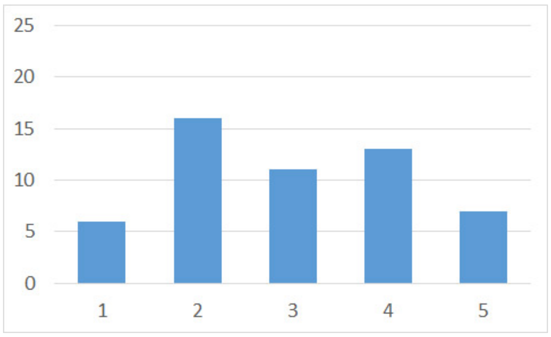
Figure 4.15: Average perceived realism, compared between different approaches (95% confidence)

data shows a more uniform distribution of ratings for Layout 1 and Layout 2. Layout 3, on the other hand, was mostly rated with values “1” and “2”. On average, the highest ratings were received for Layout 1, as shown in Table 4.4.

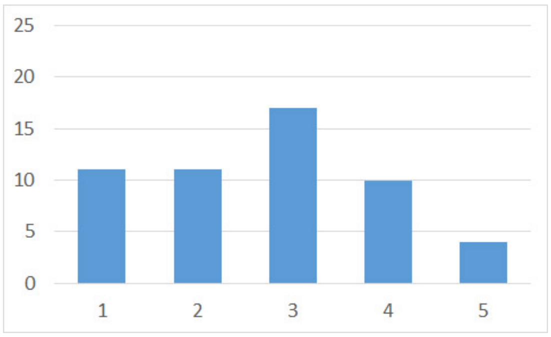
Layout	Mean	Standard Deviation
Layout 1: Stockholm/Amsterdam	2.94	1.27
Layout 2: Budapest/Lisbon	2.69	1.23
Layout 3: Rome/New York City	2.11	1.11

Table 4.4: Mean and standard deviation of ratings for different examples produced by the presented system.

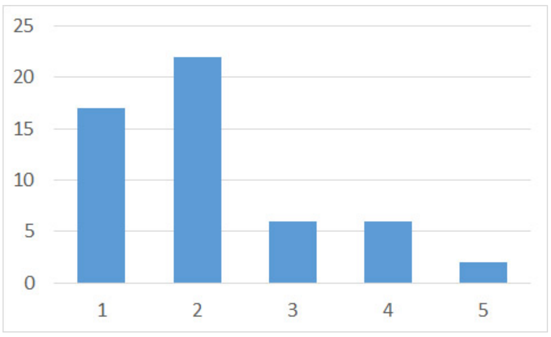
The measurement of subjective recognizability was performed through two generated layouts. Figure 4.17 shows the answer histogram for the first example. For both Vienna



(a) Layout 1: Stockholm/Amsterdam. $\mu_1 = 2.94$



(b) Layout 2: Budapest/Lisbon. $\mu_2 = 2.69$



(c) Layout 3: Rome/New York City. $\mu_3 = 2.11$

Figure 4.16: Subjective plausibility ratings provided by participants. A value of “1” corresponds to “very unrealistic” and “5” to “very realistic”.

and Paris, high ratings have been received, with the highest number of participants having selected the value “8”.

A different distribution of answers was received for the second example, which contained parts of Copenhagen and Barcelona. Figure 4.18 shows overall higher ratings for the recognizability of Barcelona, compared to Copenhagen. These results are in contrast to our initial expectations, as Copenhagen was assigned a higher mixing ratio. An overview of the average measurements for the recognizability section of the survey is shown in Table 4.5.

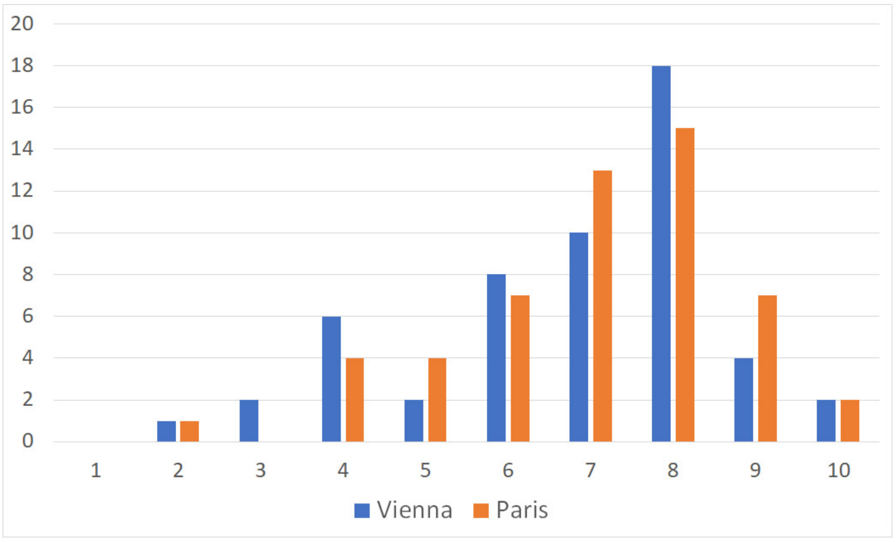


Figure 4.17: Recognizability of features from Vienna and Paris in the result. $\mu_{Vie} = 6.79$, $\mu_{Par} = 7.06$

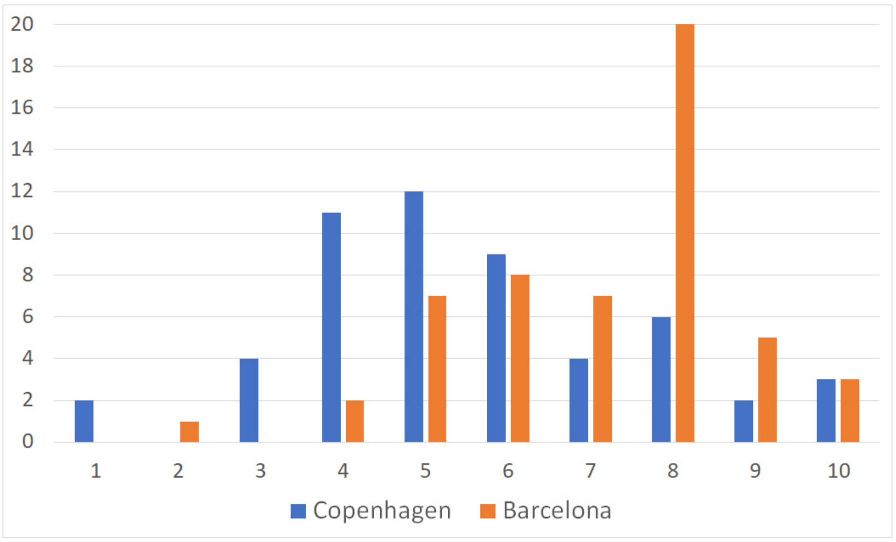


Figure 4.18: Recognizability of features from Copenhagen and Barcelona in the result. $\mu_{Cop} = 5.58$, $\mu_{Bar} = 7.11$

Input Layout	Mean	Standard Deviation
Vienna	6.79	1.84
Paris	7.06	1.67
Copenhagen	5.58	2.09
Barcelona	7.11	1.67

Table 4.5: Mean and standard deviation of ratings for the recognizability of individual cities' features.

CHAPTER 5
Discussion

5.1 User Study Analysis

The user study performed as part of this work provides helpful insight into the subjective perception of the results. A comparison with other research on procedural city generation shows lower overall values for the example created by our algorithm, as can be seen in Figure 4.15. Therefore, it can be assumed that the realism of the selected layout is not on par with the results of other algorithms. However, this does not necessarily apply to all results produced by the proposed system, as it is heavily dependent on the input data.

The general plausibility shows different result distributions for the three examples. Ratings for Layout 1 and Layout 2 exhibit a more even distribution across the range, whereas Layout 3 was perceived as rather unrealistic by the participants. All three examples have been generated using city layouts featuring different road patterns. Thus, a correlation between the shapes of the input networks and the plausibility of the result can be assumed. It is estimated that a combination of organic layouts delivers the most satisfactory outputs if realism is desired. On the other hand, the merging of an organic map section with a gridiron pattern results in a relatively unrealistic layout. This characteristic is likely caused by the irregularity produced by the CityMerge algorithm. The deformation of a road is much more noticeable if it is straight than if it is already curved and warped. Such discrepancies are more easily visible if the overall pattern is very strict.

In some cases, the results produced by our system were considered as realistic by the participants. We therefore conclude that the presented algorithm can achieve plausible results under suitable conditions. However, the realistic appearance of the output data is not always guaranteed.

The recognizability of characteristics from input cities was generally regarded as very high by the participants. In the first example, the preservation of features from the Vienna layout was rated approximately the same as the Paris layout. This coincides with the configured mixing ratio of 0.5 : 0.5. For the second image, a different mixing ratio was used, with a value of 0.85 for Copenhagen, and 0.15 for Barcelona. In theory, this should result in higher values given for Copenhagen. However, this is not reflected in the answers, as the recognizability was, overall, rated higher for Barcelona ($\mu_{Bar} = 7.11$, $\mu_{Cop} = 5.58$). A possible explanation for this phenomenon is the presence of larger roads in the selected layout of Barcelona. These are assigned a higher priority and are thus more likely to be retained.

Most participants have selected a value of 8, on a scale from 1 to 10, for the recognizability of features from Vienna, Paris, and Barcelona. In general, it can thus be concluded that the features and characteristics of input cities are integrated into the result of the procedural generation process.

The first two sections of the user study focus strictly on the perceived realism of the generated results. However, other characteristics may be of more interest in certain applications. For example, for a city layout to be used in a video game, realism may not be required. Depending on the genre and setting (such as on a different planet or in the future) it could even be undesirable. Therefore, additional measures of the overall quality produced by the proposed algorithm can be applied, based on the target use case.

Other characteristics of the result data have not been investigated as part of this study. For instance, manual adjustments could improve the plausibility. Individual undesirable structures could easily be removed by a user, to increase the realism of the result. Furthermore, the use of pre-filtered data was not tested. We estimate that a higher plausibility could be achieved if the secondary road network is only present in one input layout.

5.2 Performance Analysis

The performance of the application is within acceptable limits, and running times even surpass the goal set by the requirements. Existing approaches to data-driven city generation such as presented by Aliaga et al. [AVB08] and Nishida et al. [NGDA16] produce results within minutes, though no exact numbers are available. Similarly, the simulation-based system presented by Beneš et al. generates city layouts within running times of 2.5 to 4.5 minutes [BWK14]. To achieve such a performance, they rely heavily on parallel computation using CUDA GPUs. As expected, grammar-based approaches feature better performance. The system presented by Parish and Müller is able to generate a road network in less than ten seconds [PM01]. The agent-based approach presented by Lechner et al., on the other hand, requires about 15 minutes to half an hour to generate a city layout [LWWF03]. However, their system heavily focuses on land use, which is also included in the result.

Depending on the selected input data, our method can deliver similar performance. Even though the number of iterations heavily influences the running time, it is assumed that satisfactory results can be achieved with 200-300 simulation cycles. For small city portions, processing takes about two minutes, which rises to less than 9 minutes for medium-sized sections. Based on these numbers, it can be concluded that our application delivers comparable performance to similar approaches, given that no extremely large city sizes are desired.

5.3 Limitations and Future Work

The presented algorithm provides a viable approach to generate city layouts. However, some restrictions still apply. As the system is implemented with extendability in mind, future adjustments are possible.

5.3.1 Result Plausibility

The perceived realism of results generated by our algorithm, in its current configuration, does not reach the levels of the state-of-the-art. The approach is based on many individual components, most of which are configurable. Considerable time has been spent on the tuning of these parameters to produce satisfactory results. However, a better configuration may be possible. It might also be an option to expose certain parameters to the user, to provide more control over the generation process.

5.3.2 Mixing Ratios

The mixing ratios of input cities influence the generation process. The algorithm is designed in such a way that a higher mixing value causes a layout to have a higher priority, within certain limits. This affects how nodes are merged, and which roads are favored during post-processing. To preserve as much recognizable structure as possible, road priorities for attraction are mainly obtained from the number of lanes. This may cause unexpected behaviors. Tighter integration of the mixing ratios could increase the effect, but may also result in reduced preservation of features from input maps. Another option is the normalization of road priorities across all input data sets, which could prevent cities with many wide roads from predominating the result.

5.3.3 Performance and Scalability

The algorithm is implemented in such a way that each node acts as an individual agent. For large numbers of nodes, however, this can produce a bottleneck. Especially when the input maps are very large, reduced responsiveness can be experienced. The overhead for that many objects, in terms of memory as well as processing, is considerable.

The performance is further impacted by the single-threaded nature of Unity. By design, event functions in MonoBehaviour instances are called sequentially. Thus, the presented application is only able to make limited use of multi-core processors. In the future, the Unity ECS, as mentioned in Section 3.4.11, may provide a viable alternative to the current implementation as MonoBehaviours.

5.3.4 Input Data and Preprocessing

While the algorithm is designed to support more than two cities as input, it is not the main focus of this work. It is possible to generate layouts using three or more input maps, but the results feature a relatively high density of nodes and edges which may be perceived as unnatural. As a mitigation, the pre-filtering of one or two of the source layouts, as described in Section 4.1.3, may be an option.

In future work, such functionality could be integrated directly into the application. This could facilitate certain scenarios, such as the combination of a city's arterial roads with another city's secondary road network.

5.3.5 Buildings

Though it was not part of the scope for this work, the addition of building models to generated layouts could provide an interesting topic for future work. The combination of building types, as well as individual building models, are areas of research that have seen little exploration. Extensive adaptations to the current system would be necessary for such a purpose.

CHAPTER 6

Conclusion

In this work, we present a novel approach to the procedural generation of city layouts based on existing data. It allows the use of multiple real city maps to create new road networks. We provide a filter query for exporting data from OpenStreetMap, which is then directly usable as input for the proposed system. The presented algorithm employs a non-parametric data-driven approach to city generation. Through the direct inclusion of road networks in the generation process, no information is lost through consolidation or extraction operations, as they are employed by other works [AVB08, NGDA16]. The simulation-based nature of the algorithm allows direct user interaction throughout the generation process.

An application has been developed in the Unity game engine to showcase the devised algorithm. It is implemented with reliance on the internal physics engine, which is responsible for parts of the operations performed for road network adaptation. The required user input is minimal, though parameters can be changed if desired.

We have shown that the characteristics of input cities are successfully transformed and included in the result. Compared to the work presented by Nishida et al. [NGDA16], our approach produces existing and recognizable structures from the input data sets in the result, without the need for manual selection. Furthermore, road networks are combined across the whole area, as opposed to only in specific sections. In terms of subjective realism, the presented approach could not outperform the current state-of-the-art in city generation. The system functions as a proof-of-concept that can be extended and improved, such as through the addition of more elaborate post-processing heuristics. Further extensions are possible through the addition of building models to generated layouts.

List of Figures

Acronyms

AI Artificial Intelligence. 9

ECS Entity Component System. 55, 77

GIS Geographic Information System. 16, 28, 29, 34

GUI Graphical User Interface. 32

LIDAR Light Detection and Ranging. 21

LTS Long Term Support. 40

NPC Non-Player Character. 9

Overpass QL Overpass Query Language. 31, 32, 61

PCG Procedural Content Generation. xi, 1, 8–11, 13, 14, 16, 19–22

UML Unified Modeling Language. 44

Bibliography

[ABVA08] Daniel G. Aliaga, Bedřich Beněs, Carlos A. Vanegas, and Nathan Andrysco. Interactive reconfiguration of urban layouts. *IEEE Computer Graphics and Applications*, 28(3):38–47, 5 2008.

[AVB08] Daniel G. Aliaga, Carlos A. Vanegas, and Bedřich Benedš. Interactive example-based urban layout synthesis. In *ACM SIGGRAPH Asia 2008 Papers, SIGGRAPH Asia'08*, 2008.

[BJA+16] Anahid Basiri, Mike Jackson, Pouria Amirian, Amir Pourabdollah, Monika Sester, Adam Winstanley, Terry Moore, and Lijuan Zhang. Quality assessment of OpenStreetMap data using trajectory mining. *Geo-Spatial Information Science*, 19(1):56–68, 1 2016.

[Boe17] Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 9 2017.

[BS20] Joseph Alexander Brown and Marco Scirea. Procedural Generation for Tabletop Games: User Driven Approaches with Restrictions on Computational Resources. In *Advances in Intelligent Systems and Computing*, volume 925, pages 44–54. Springer Verlag, 2020.

[BWK14] Jan Beneš, Alexander Wilkie, and Jaroslav Křivánek. Procedural modelling of urban road networks. *Computer Graphics Forum*, 33(6):132–142, 9 2014.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath). *W3C Recommendation*, 1999.

[CJMW10] Błażej Ciepłuch, Ricky Jacob, Peter Mooney, and Adam C Winstanley. Comparison of the accuracy of OpenStreetMap for Ireland with Google Maps and Bing Maps. In *Proceedings of the Ninth International Symposium on Spatial Accuracy Assessment in Natural Resuorces and Enviromental Sciences 20-23rd July 2010*, page 337. University of Leicester, 2010.

[dC11] António Miguel de Campos. Animated fractal mountain - Wikimedia Commons. [Online; accessed 03.12.2019] https://commons.wikimedia.org/wiki/File:Animated_fractal_mountain.gif, 2011.

[Dor10] Joris Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Workshop on Procedural Content Generation in Games, PC Games 2010, Co-located with the 5th International Conference on the Foundations of Digital Games*, 2010.

[DP10] Jonathon Doran and Ian Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 6 2010.

[Fal11] Gregory Falkovich. *Fluid mechanics: A short course for physicists*. Cambridge University Press, 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[Goo00] M. F. Goodchild. GIS and transportation: Status and challenges. *GeoInformatica*, 4(2):127–139, 2000.

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

[GT10] Jean François Girres and Guillaume Touya. Quality Assessment of the French OpenStreetMap Dataset. *Transactions in GIS*, 14(4):435–459, 8 2010.

[HBW06] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Proceedings - Sandbox Symposium 2006: ACM SIGGRAPH Video Game Symposium, Sandbox '06*, pages 179–186, 2006.

[HMVDVI13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, 9(1):1–22, 2013.

[HW08] Mordechai Haklay and Patrick Weber. OpenStreet map: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 10 2008.

[HYWL18] D. Hooshyar, M. Yousefi, M. Wang, and H. Lim. A data-driven procedural-content-generation approach for educational games. *Journal of Computer Assisted Learning*, 34(6):731–739, 2018.

[IFPW10] Martin Ilčík, Stefan Fiedler, Werner Purgathofer, and Michael Wimmer. Procedural Skeletons: Kinematic Extensions to CGA-Shape Grammars. In *Proceedings of the Spring Conference on Computer Graphics 2010*, pages 177–184. Comenius University, Bratislava, 5 2010.

[Jan12] Katleen Janssen. Open government data and the right to information: Opportunities and obstacles. *The Journal of Community Informatics*, 8(2), 2012.

[Jar14] Michael Jaros. *Crowd Simulation for Virtual Environments in Unity*. Wien, Techn. Univ., Dipl.-Arb., 2015, Vienna, 2014.

[Joh72] James H. Johnson. *Urban Geography: An Introductory Analysis*. Pergamon, 2nd edition, 1972.

[JSS$^+$17] Peter A. Johnson, Renee Sieber, Teresa Scassa, Monica Stephens, and Pamela Robinson. The Cost(s) of Geospatial Open Data. *Transactions in GIS*, 21(3):434–445, 6 2017.

[Kaz09] Darius Kazemi. Spelunky's Procedural Space. [Online; accessed 05.12.2019] http://tinysubversions.com/2009/09/spelunkys-procedural-space/, 2009.

[KK96] Lance M. Kaplan and C. C.Jay Kuo. An improved method for 2-D self-similar image synthesis. *IEEE Transactions on Image Processing*, 5(5):754–761, 1996.

[KKC18] Joon-Seok Kim, Hamdi Kavak, and Andrew Crooks. Procedural city generation beyond game development. *SIGSPATIAL Special*, 10(2):34–41, 2018.

[KM06] George Kelly and Hugh McCabe. A survey of procedural techniques for city generation. *ITB Journal*, 7(2):87–130, 5 2006.

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

[KMK12] Lars Krecklau, Christopher Manthei, and Leif Kobbelt. Procedural Interpolation of Historical City Maps. *Computer Graphics Forum*, 31(2pt3):691–700, 2012.

[Kos91] Spiro Kostof. *The City Shaped: Urban Patterns and Meanings Through History*. Thames and Hudson, London, 1991.

[Kro17] Karl Kropf. *The Handbook Of Urban Morphology*. John Wiley & Sons, Ltd, Chichester, UK, 10 2017.

[Law15] Joel Lawhead. *Learning Geospatial Analysis with Python, 2nd Edition.* Packt Publishing Ltd, 2015.

[LB14] Rémi Louf and Marc Barthelemy. A typology of street patterns. *Journal of the Royal Society Interface*, 11(101), 12 2014.

[Ley83] David Ley. *A social geography of the city.* Harper & Row New York, New York, NY, USA, 1983.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[LWW$^+$07] Tom Lechner, Benjamin Watson, Uri Wilensky, Seth Tisue, Martin Felsen, Andy Moddrell, Pin Ren, and Craig Brozefsky. Procedural modeling of urban land use. Technical report, North Carolina State University. Dept. of Computer Science, 2007.

[LWWF03] Thomas Lechner, Ben Watson, Uri Wilensky, and Martin Felsen. Procedural city modeling. In *1st Midwestern Graphics Conference*, pages 1–6, 2003.

[LYC$^+$10] Yotam Livny, Feilong Yan, Baoquan Chen, Matt Olson, Hao Zhang, and Jihad El-Sana. Automatic Reconstruction of Tree Skeletal Structures from Point Clouds. *ACM Transactions on Graphics*, 29(6):1–8, 2010.

[Lyn60] Kevin Lynch. *The image of the city*, volume 11. The MIT Press, 1960.

[Lyn84] Kevin Lynch. *A Theory of Good City Form.* MIT Press, Cambridge, MA, 1984.

[Mae17] Luca Maestri. *Visualization of Computer-Generated 3D Cities using GIS Data.* Technische Universität Wien, Vienna, 2017.

[Man83] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*, volume 173. WH Freeman New York, New York, NY, 1983.

[Mar04] Stephen Marshall. *Streets and Patterns.* Spon Press, 2004.

[Mei19] Lucas Meijer. On DOTS: Entity Component System. [Online; accessed 11.02.2020] https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/, 3 2019.

[MKM89] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1989*, pages 41–50. Association for Computing Machinery, Inc, 7 1989.

[MSK10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-Generated Residential Building Layouts. *ACM Transactions on Graphics*, 29(6):1–12, 2010.

[Mun13] Dave Munson. Which street pattern represents your continent? | Munson's City. [Online; accessed 16.01.2020] https://munsonscity.wordpress.com/2013/10/09/which-street-pattern-represents-your-continent/, 2013.

[MWH$^+$06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06*, pages 614–623, 2006.

[MZWVG07] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics*, 26(3), 7 2007.

[NGDA16] G. Nishida, I. Garcia-Dorado, and D. G. Aliaga. Example-Driven Procedural Urban Roads. *Computer Graphics Forum*, 35(6):5–17, 2016.

[OGC20] OGC. Welcome to The Open Geospatial Consortium | OGC. [Online; accessed 30.01.2020] https://www.opengeospatial.org/, 2020.

[Ope14] OpenStreetMap Wiki contributors. OpenStreetMap Wiki. [Online; accessed 02.02.2020] https://wiki.openstreetmap.org/w/index.php?title=Main_Page &oldid=1060762, 2014.

[Ope18] Open Knowledge Foundation. Open Data Commons Open Database License (ODbL). [Online; accessed 31.01.2020] https://opendatacommons.org/licenses/odbl/1.0/, 2018.

[OR13] Hans C. Ohanian and Remo Ruffini. Newton's gravitational theory. In *Gravitation and Spacetime*, pages 1–46. Cambridge University Press, 3 edition, 4 2013.

[Pen84] Alex P. Pentland. Fractal-Based Description of Natural Scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):661–674, 1984.

[Per85] Ken Perlin. An Image Synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 7 1985.

[Per01] Ken Perlin. Noise hardware. *Real-Time Shading SIGGRAPH Course Notes*, 2001.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. The Virtual Laboratory. Springer New York, New York, NY, 1990.

[PM01] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, pages 301–308, 2001.

[RPSC99] Harvey Ray, Hanspeter Pfister, Deborah Silver, and Todd A. Cook. Ray casting architectures for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):210–223, 1999.

[Smi15] Gillian Smith. An Analog History of Procedural Content Generation. *Foundations of Digital Games*, pages 0–5, 2015.

[Sny87] John Parr Snyder. Map projections - a working manual. *US Geological Survey Professional Paper*, 1987.

[SO93] Michael Southworth and Peter M. Owens. The Evolving Metropolis: Studies of Community, Neighborhood, and Street Form at the Urban Edge. *Journal of the American Planning Association*, 59(3):271–287, 1993.

[SPK$^+$14] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, and B. Benes. Inverse procedural modelling of trees. *Computer Graphics Forum*, 33(6):118–131, 9 2014.

[STBB14] Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, 33(6):31–50, 9 2014.

[STN16] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Springer International Publishing, Cham, 2016.

[SYBG04] Jing Sun, Xiaobo Yu, George Baciu, and Mark Green. Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '02*, page 33, New York, New York, USA, 2004. ACM Press.

[TB16] Fieke C Taal and Rafael Bidarra. Procedural Generation of Traffic Signs. In Vincent Tourre and Filip Biljecki, editors, *Eurographics Workshop on Urban Data Modelling and Visualisation*. The Eurographics Association, 2016.

[TKSY11] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is procedural content generation? Mario on the borderline. In *ACM International Conference Proceeding Series*, 2011.

[TMNT20] Eite Tiesinga, Peter J Mohr, David B Newell, and Barry N Taylor. The 2018 CODATA Recommended Values of the Fundamental Physical Constants. *Web Version*, 8.1, 2020.

[TYSB11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. In *IEEE Transactions on Computational Intelligence and AI in Games*, volume 3, pages 172–186, 9 2011.

[Unia] Unity Technologies. Solutions | Unity. [Online; accessed 11.02.2020] https://unity.com/solutions.

[Unib] Unity Technologies. Unity - Manual: Order of Execution for Event Functions. [Online; accessed 06.02.2020] https://docs.unity3d.com/Manual/ExecutionOrder.html.

[VABW09] Carlos A. Vanegas, Daniel G. Aliaga, Bedřich Beneš, and Paul A. Waddell. Interactive Design of Urban Spaces using Geometrical and Behavioral Modeling. In *ACM Transactions on Graphics*, volume 28, pages 1–10. ACM, 2009.

[Vas18] Michael Vasiljevs. *Procedural modeling of park layouts.* Technische Universität Wien, Wien, 2018.

[VGDA+12] Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. Inverse design of urban procedural models. *ACM Transactions on Graphics*, 31(6):1, 2012.

[Wat99] Nigel Waters. Transportation GIS: GIS-T. *Geographical Information Systems*, pages 827–844, 1999.

[WFf11] Nathan Whitehead and Alex Fit-florea. Precision & Performance : Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *NVIDIA white paper*, 21(10):767–75, 2011.

[Whe15] Stephen M Wheeler. Built Landscapes of Metropolitan Regions: An International Typology. *Journal of the American Planning Association*, 81(3):167–190, 2015.

[Wik18] Wikimedia Commons. Fractal tree - Wikimedia Commons. [Online; accessed 05.12.2019] https://commons.wikimedia.org/w/index.php?title=File:Fractal_tree_(Plate_b_-_2).jpg&oldid=325669560, 2018.

[WMV+08] Benjamin Watson, Pascal Müller, Oleg Veryovka, Andy Fuller, Peter Wonka, and Chris Sexton. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, 28(3):18–26, 5 2008.

[WMWG09] Basil Weber, Pascal Müller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4D cities. *Computer Graphics Forum*, 28(2):481–492, 4 2009.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *ACM SIGGRAPH 2003 Papers, SIGGRAPH '03*, pages 669–677, New York, New York, USA, 2003. ACM Press.

[WYD$^+$14] Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. Inverse procedural modeling of facade layouts. *ACM Transactions on Graphics*, 33(4):1–10, 2014.

[ZSTR07] Howard Zhou, Jie Sun, Greg Turk, and James M. Rehg. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, 7 2007.