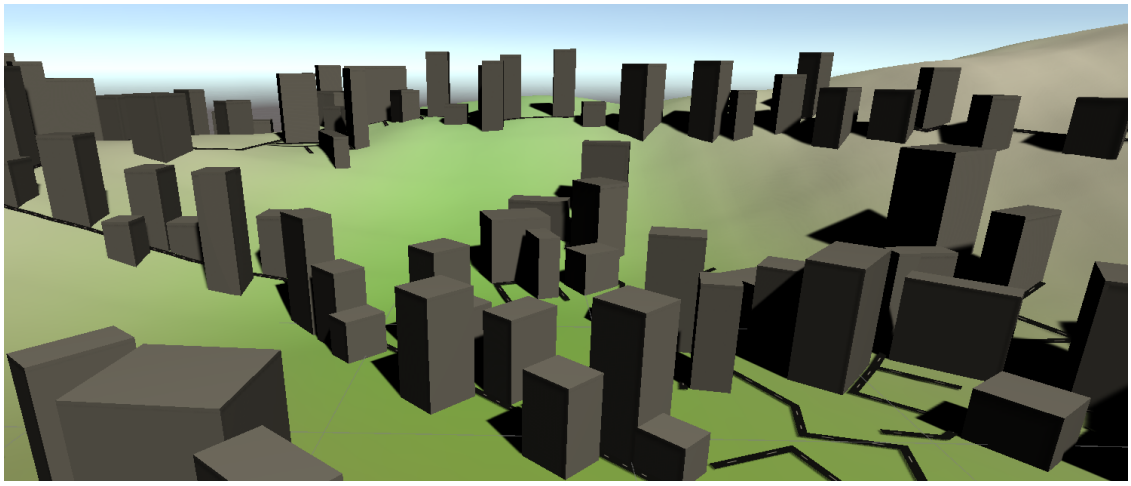


BIAS

A Unity Asset for Procedurally Generating Cities

Bachelor's of Science Thesis in Computer Science and Engineering

Alma Eriksson
David Hall
Ludwig Hultqvist
David Hultsten
William Johnsson
Ludvig Liljeqvist

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

Bachelor's Thesis 2020:85

BIAS

A Unity Asset for Procedurally Generating Cities

ALMA ERIKSSON
DAVID HALL
LUDWIG HULTQVIST
DAVID HULTSTEN
WILLIAM JOHNSSON
LUDVIG LILJEQVIST

UNIVERSITY OF
GOTHENBURG

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

BIAS
A Unity Asset for Procedurally Generating Cities

Alma Eriksson, David Hall, Ludwig Hultqvist, David Hultsten, William Johnsson, Ludvig Liljeqvist

Supervisor: Staffan Björk, Department of Computer Science and Engineering
Examiner: Olof Torgersson, Department of Computer Science and Engineering

Bachelor's Thesis 2020:85
Department of Computer Science and Engineering
Division of Interaction Design
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers' Department of Computer Science and Engineering
Gothenburg, Sweden 2020

Abstract

Procedural content generation, or PCG, is a method of algorithmically generating, for instance, video game content with a limited amount of user input. The purpose of this project is to explore how PCG may be used in game development to make the process of creating game assets less time consuming. To achieve this, a study of existing algorithms, such as rewriting systems and noise functions, is presented. These algorithms have been adapted to generate cities consisting of terrain, road networks, as well as buildings. The result is a tool called BIAS, *Built in a Second*, built on the game development platform Unity, which can be used to generate graphical representations of three-dimensional cities. Additionally, BIAS is designed to provide game developers with a degree of control of the generation. The tool consists of a user interface that can be used by developers to manipulate several aspects of the procedural generation. The resulting tool shows that it is possible to use PCG to generate cities, but it is far from perfect; a lot can be improved and added to make the tool better.

Keywords: procedural content generation, PCG, video game development, 3D cities, unity, terrain generation, city generation

Sammandrag

Processuell innehållsgenerering, eller PCG (“Procedural Content Generation”), är en metod för att algoritmiskt generera till exempel spelinnehåll med en begränsad mängd indata från användaren. Syftet med det här projektet är att utforska hur PCG kan användas inom spelutveckling för att göra processen mindre tidskrävande jämfört med manuell konstruering av grafiska komponenter. För att uppnå detta presenteras en studie om existerande algoritmer såsom omskrivningssystem och brusfunktioner. Dessa algoritmer har anpassats för att generera städer bestående av terräng, vägnätverk och byggnader. Resultatet är ett verktyg med namnet BIAS, *Built in a Second*, byggt på spelutvecklingsplattformen Unity som kan generera grafiska representationer av tredimensionella städer. Dessutom är BIAS designat för att ge spelutvecklare en nivå av kontroll av genereringen. Verktyget består av ett användargränssnitt som utvecklare kan använda för att påverka flera aspekter av den processuella genereringen. Det resulterande verktyget visar att det är möjligt att använda PCG för att generera städer, men det är långt från perfekt; mycket kan förbättras och läggas till för att göra verktyget bättre.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Purpose . . . 2
- 1.3 Goal . . . 2
- 1.4 Delimitations . . . 3
- 1.5 Ethics . . . 4

2 Theory **5**
- 2.1 Approaches to Procedural Content Generation . . . 5
- 2.2 Terrain Generation . . . 6
  - 2.2.1 Noise Maps . . . 6
  - 2.2.2 Whittaker Biomes . . . 7
  - 2.2.3 Agents . . . 8
- 2.3 Road Generation . . . 9
  - 2.3.1 Street Patterns . . . 10
  - 2.3.2 L-systems . . . 10
  - 2.3.3 A* Search . . . 11
- 2.4 City Generation . . . 12
  - 2.4.1 Urban Cities . . . 12
  - 2.4.2 Heat-Maps . . . 12
- 2.5 Unity Real-Time Development Platform . . . 13

3 Process **16**
- 3.1 Planning . . . 16
  - 3.1.1 Methodology . . . 16
  - 3.1.2 Development Tools . . . 17
  - 3.1.3 Problem Identification & Initial Feature Selection . . . 17
- 3.2 Implementation . . . 18
  - 3.2.1 Terrain Generation . . . 19
  - 3.2.2 Road Network Generation . . . 21

3.2.3 Unifying Road Networks With Terrain . . . 23
3.2.4 Generating Plots . . . 24
3.2.5 Generating Buildings . . . 25
3.2.6 The Editor . . . 26
3.2.7 Assembly Definitions . . . 27
3.2.8 Making Things More Responsive . . . 27

4 Results **29**
4.1 BIAS . . . 29
4.2 Utilization of Procedural Content Generation . . . 33
4.2.1 Noise . . . 33
4.2.2 Textures . . . 34
4.2.3 Road Networks . . . 35
4.2.4 Plots . . . 36
4.2.5 Buildings . . . 38
4.3 System Architecture . . . 39
4.3.1 The S.O.L.I.D. Principles . . . 39
4.3.2 High Cohesion, Low Coupling . . . 40
4.3.3 Model-View-ViewModel Pattern . . . 41
4.3.4 Publish-Subscribe Pattern . . . 43
4.3.5 Strategy Pattern . . . 45
4.3.6 Factory Method Pattern . . . 46
4.3.7 Dependency Injection . . . 47

5 Discussion **49**
5.1 Results . . . 49
5.2 Process . . . 50
5.3 Validity & Generalisation . . . 51
5.4 Ethics . . . 52
5.5 Future Work . . . 52
5.5.1 Generation Improvements . . . 53
5.5.2 Improving Performance . . . 53
5.5.3 Feature Ideas . . . 53
5.5.4 Publishing the Tool . . . 55

6 Conclusion **56**

Bibliography **57**

Appendices **62**

A Software and Games Utilizing Procedural Content Generation **63**

B Search-Based Approach to PCG **65**

C Creating Terrain Meshes In Unity **68**

1

Introduction

This chapter introduces the subject of procedural content generation and how it is currently utilized. Subsequently, it presents how the concept is utilized in the creation of the unity asset *BIAS*, which stands for *Built in a Second*.

1.1 Background

Many games are manually crafted by developers to create immense environments and experiences players can enjoy. Since creating everything manually is often a time-consuming task, many developers may benefit from more software tools to speed up the development process. This is where procedural content generation, or PCG, comes into the picture.

Procedural content generation can be defined as "the algorithmic creation of game content with limited or indirect user input" [1, p. 1]. This implies that by using PCG, content can be created with a low degree of human interaction. Togelius et al present several arguments for utilizing the concept [2, pp. 141-142]. Firstly, many PCG methods are capable of creating game worlds and vast amounts of game assets at faster rates than 3D-artists. As a result, development effort may be reduced by utilizing procedural content generation. PCG is also memory efficient since content can be created only when it is needed. With the capability of generating content in real-time, endless worlds can also be created for a player to explore. As a result, it has the potential of creating new kinds of games with a greater focus on replayability. This can subsequently provide a helping hand for thinking out of the box, by creating content that humans may not come up with on their own.

Procedural content generation may also be utilized as a complementary tool for manually created content [1, p. 57]. In this fashion, PCG is not used for creating new assets but rather hides, reveals or reshapes existing assets on demand. This is a

necessary utility in, for instance, games containing large worlds with great amounts of graphical assets. Rendering all assets in full resolution can be detrimental for performance and possibly redundant due to restraints on monitor resolution. In such contexts, the resolution of assets, or equally the level of detail, can be increased or decreased through procedural content generation. Because of the dynamic properties of such tasks, they cannot be done manually. Instead, they can be handed over to PCG algorithms.

Since January 2020, 58 games tagged with "Procedural Generation" have been released to the game distribution service Steam [3]. This is approximately 1.8% of the 26 games released daily on the service within that time-span [4]. The concept is commonly utilized in many games for creating content such as characters, levels, gameplay and worlds. PCG is, for instance, utilized in Diablo for the creation of new dungeons for each level [5]. It is also used in Minecraft, where vast landscapes with villages, temples, and dungeons are procedurally generated [6]. See Appendix A for a more detailed description.

1.2 Purpose

The purpose of this project is to explore how game development processes can be made less time consuming by utilizing procedural content generation in the creation of three-dimensional cities and surrounding terrain in a Unity asset.

1.3 Goal

A useful and interactive tool for generating cities and surrounding terrains in real time shall primarily be provided. To fully satisfy the purpose of making the development process of games more effortless, the generated result must be useful for the development of generic 3D-games.

The asset shall provide the user with a way of procedurally generating terrain elements such as mountains, valleys and watercourses in various climates from a set of parameters. The user shall then be able to place the location where a city is to be generated. The user shall subsequently be able to generate a city with roads, plots and buildings by using a variety of procedural generation methods.

To reduce the development time of games, the procedural generation of the resulting asset must be relatively time-efficient, using fast algorithms and an effective pipeline. To also not hinder the creativity of its users, resulting cities must also be open for customization. A user shall be able to add their own graphical and behavioral

assets, such as their own 3D models or simulated players. Similarly, the tool must also be expandable for the developers behind it, allowing it to be extended effortlessly with new PCG methods and algorithms.

Finally, the asset shall also be open for its users to explore a variety of PCG methods, providing game developers with a knowledge base for using procedural content generation in their own designs. This requires the tool to explore the advantages and disadvantages of various PCG algorithms.

1.4 Delimitations

To utilize procedural content generation more distinctly as a tool for creating custom game environments, the project is predominantly concentrated on developing a tool for game development and not on creating actual gameplay. Users shall only be able to utilize the tool for generating the layout and contents of the world, but will not be able to create the gameplay directly with it.

The tool shall be designed to allow developers to use it for creating 3D worlds for most game genres that make use of city environments with its surroundings. As a result, no specific genre is in focus, allowing the tool to provide the use of a wide range of use cases. This may, however, not be possible, due to the time constraints of the project and possible limitations of the PCG algorithms that are used.

Content shall predominantly be generated based on creating credible aesthetics, rather than realistic functionality. Generated cities will, for instance, have approximately realistic appearance but will lack utility such as subways and sewage systems. Cities will also lack historical accuracy of how realistic cities are formed and expanded. Focusing on credibility rather than realism was mainly a result of the project's time constraints, as representing realism accurately requires spending more time on, for instance, historical research. Being less restricting in terms of realism also ensures extra space for the creativity of the users.

The visual fidelity of the generated content is not a priority. The focus of the project lies on applying procedural content generation in the creation of major city components rather than drawing sophisticated art assets. This also ensures room for developers to apply their own textures on the generated content.

Finally, testing the usability of the final product with actual game developers is not included in the project's scope since it would require extensive studies with many different developers. As a result, the usability of the final product is only validated by its creators.

1.5 Ethics

The procedural generation tool is not likely to negatively affect society. The only topic worth mentioning is the possibility of some people losing their jobs if companies start using this tool. An example of people that may be affected are game designers, since the tool may automate their jobs. However, the tool is mainly meant to make game development more effortless for hobbyists and smaller development teams.

As only little regard is spent on the generated content realistically representing the real world, the tool will disregard some real ethical aspects. It will, for instance, not focus on the ethics of available transportation systems, such as trains and busses, that otherwise are considered in real cities. Instead, the tool shall only generate cities with road networks only for cars. As a result, only a sub-part of the real world is represented by the tool. However, as described in section 1.4, the focus of the tool lies on credibility rather than realism.

The resulting tool is also not intended for any form of actual city planning or other scientific applications, such as transportation simulations. Using the tool in such scenarios will likely not produce satisfactory results, which may lead to problems for the user. While this is a clear misuse of the tool, a small risk of misuse still exists, which may cause damage. For instance, conducting urban planning on a city layout generated by the tool would have no considerations for access to natural resources, as the tool does not account for such parameters in the generation process.

2

Theory

Behind the creation of a procedural content generation tool lies a variety of principles, methods and algorithms. This chapter presents the underlying theory behind the creation of a procedural generation asset. It covers everything from various approaches of procedural content generation to a set of methods and algorithms for generating city elements such as terrain, roads and buildings. The chapter subsequently presents an insight into understanding the tools used for implementing the BIAS asset.

2.1 Approaches to Procedural Content Generation

Procedural generation methods and algorithms may be distinguished by a variety of different approaches. Two distinguishable approaches are the Constructive & Generate-and-Test Approaches [2]. Constructive approaches involve only the actual generation of content. Once the content has been created, the process is over. The latter, however, also involves testing if the content satisfies a set of specified requirements. If it does not, the content is ignored, the process starts over and repeats until all criteria are fulfilled.

The Search-Based approach is a third distinguishable approach that is based on generate-and-test. The key difference is that in the search-based approach, the quality of generated content is graded, unlike in generate-and-test where it is instantly discarded if not optimal [2]. Grading is performed by fitness, or evaluation, functions that determine the suitability of the content within the context. The procedural generation of content subsequently depends on the prior fitness scores to increase the overall content quality [2]. The search-based approach predominantly consists of the three stages: content representations, evaluation functions and search algorithms, which are presented in detail in Appendix B.

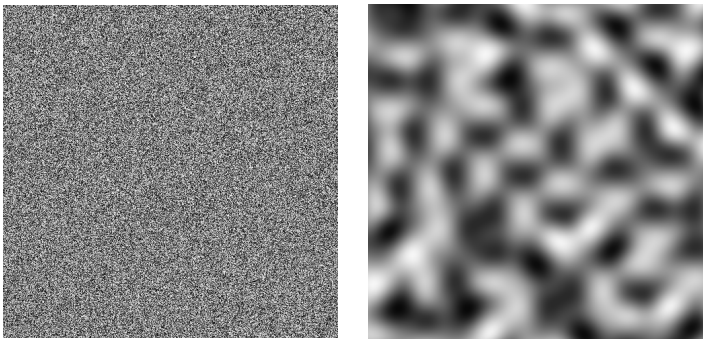
2.2 Terrain Generation

A key element of generating a city is the terrain that it is located on. This section covers the theory of various terrain generation methods, such as creating noise maps, Whittaker biomes and using agents.

2.2.1 Noise Maps

A common approach for generating terrain features such as heights and curvature is to create an intensity map, or noise map, represented by a two-dimensional matrix containing the brightness of each pixel [1, pp. 58-59]. The noise map can subsequently determine the terrain heights through displacement mapping.

One approach to create noise maps is to assign random values to each element of the matrix, as displayed in figure 2.1a. However, this results in terrain with random spikes, which is not suitable to adapt in a game environment. To combat such problems, a type of gradient noise can be used instead, which was formally presented by Ken Perlin in 1985 [7]. At the time, this resulted in more realistic looking terrains, compared to what was previously available. Named after its creator, such noise maps came to be known as Perlin noise [8]. A subversion of Perlin noise known as Simplex noise is displayed in figure 2.1b.



(a) Random noise. **(b)** Simplex noise.

Figure 2.1: A comparison of noise maps produced with random and Simplex noise.

Perlin noise is essentially generated by retrieving a smooth function through interpolation of a set of pseudo-random gradients. The results appear random but nonetheless artificial, which counteracts the goal of Perlin noise. Several layers of noise called *octaves* can also be combined to provide a noise map with varying levels of detail, as displayed in figure 2.2. For instance, in nature, mountainous terrains contain both large details, such as peaks and valleys, as well as smaller details,

such as boulders and rocks, which can be imitated with octaves. The contribution of each octave decreases exponentially according to a *persistence* value, while its frequency increases exponentially depending on a *lacunarity* value. Noise maps with a low persistence and high lacunarity will yield a smooth texture with some visible fine details [9].

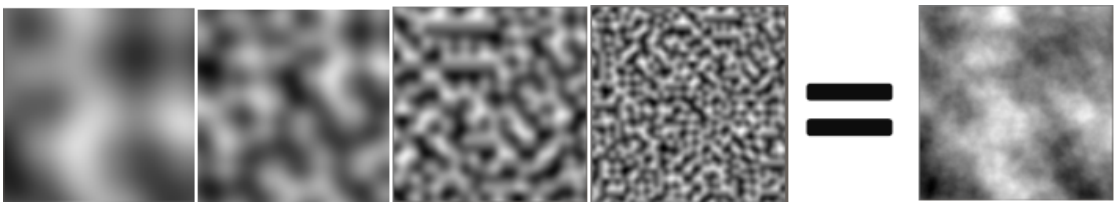


Figure 2.2: Combining a series of octaves with increasing frequency from left to right into a a single noise map.

Perlin noise was later improved by its creator in 2001, when he introduced the previously mentioned Simplex noise [8], as displayed in figure 2.1b. The improved noise algorithms scale to higher dimensions with less computational overhead and is easier to implement in hardware. To generate simplex noise in two dimensions, space is divided into triangles called simplexes, where pseudo-random gradients are located in the corner of each simplex. For every point in the space, the contribution of each simplex is calculated based on the "multiplication of the extrapolation of the gradient ramp and a radially symmetric attenuation function." [10].

A problem with noise functions is that all locations are independent of each other, which may appear as if there is not enough uniqueness for a game environment. However, multiple noise maps can be utilized together to create, for instance, overhangs and caves, resulting in more interesting terrain.

2.2.2 Whittaker Biomes

Credible terrain appearances may be generated by classifying the terrain with various ecosystems or biomes, such as deserts, rainforests and tundra. Biomes may subsequently be defined based on a variety of climatic features, such as vegetation, wind speeds and height [11]. A simple set of biome definitions that are commonly utilized in the context of procedural content generation is the scheme proposed by Robert Harding Whittaker [12]. Whittaker proposed that simple biomes can be defined by the two climatic features of the mean annual temperature and precipitation, without accounting for seasonality. An example of a Whittaker scheme is displayed in figure 2.3.

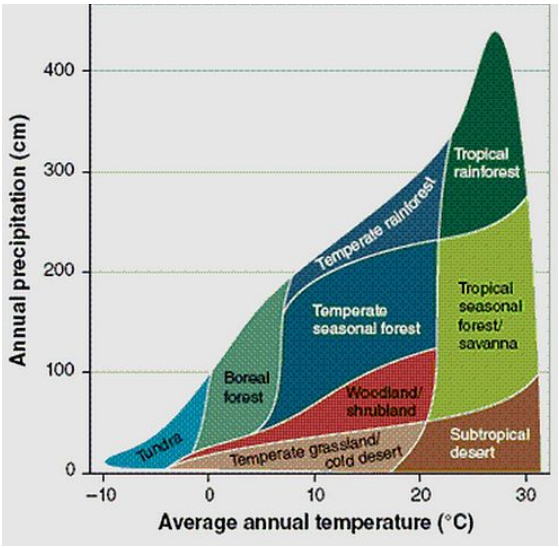


Figure 2.3: A Whittaker diagram with a variety biomes defined by respective area [13].

2.2.3 Agents

While the inherent randomness of PCG methods like Perlin noise is what makes them desirable, it can often be the detriment of customizability. Designers can tailor their terrain to some extent, but only on a global level through the adjustment of a set of parameters. Agent-based generation is an alternative which retains customizability while also providing expected PCG features, such as randomness, unpredictability and speed [1, p. 64].

Professors Jonathon Doran and Ian Parberry from The University of North Texas describe a way of procedurally generating terrain using software agents [14]. The method utilizes five different agent types working concurrently to simulate natural phenomena by sensing and altering their environment and at their "will". Designers may subsequently influence the generation of terrain by adjusting the number of agents of each type and restricting their respect by the number of actions they can perform. Once an agent has performed the maximum number of actions, it becomes inactive. By defining agents and their respective set of parameters, the generation of terrain can be customized to the requirements of the artist, yet remain unpredictable through the use of random seed numbers.

Agents modify the environments by performing three main tasks. The first phase consists of multiple agents creating the main shape of the terrain by generating coastlines. In the second phase, more agents are deployed to simultaneously generate more detailed terrain features, such as lowlands, beaches, and the details

of mountains. In the final phase, pieces of the terrain are chipped away to create rivers, where the number of rivers depends on the number of agents.

Each phase contains several types of agents, including (but not limited to) coastline agents, smoothing agents and beach agents. Coastline agents generate rough parts of the environment while smoothing agents eliminate any resulting rapid elevation changes. This is followed by beach agents traversing the shoreline, creating sandy areas near bodies of water. Examples of procedurally generated terrain with coastline agents and beach agents are displayed in figures 2.4 and 2.5.

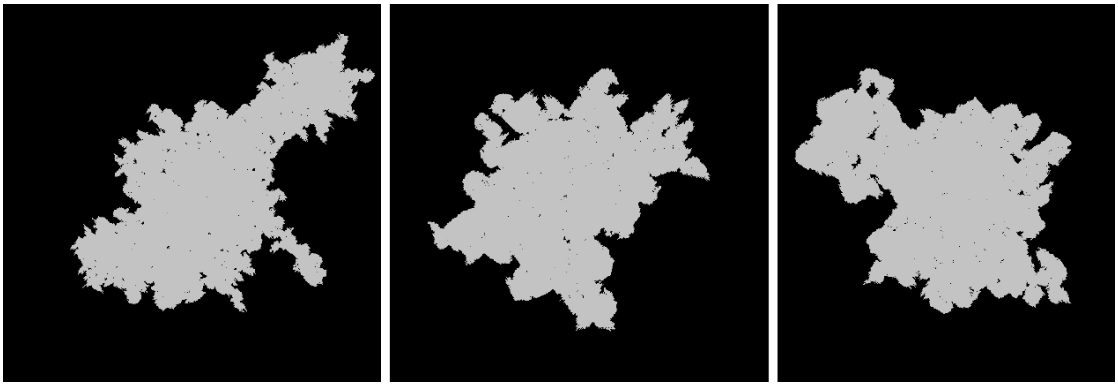


Figure 2.4: A coastline generated by coastline agents with low, medium and high number of actions [14].

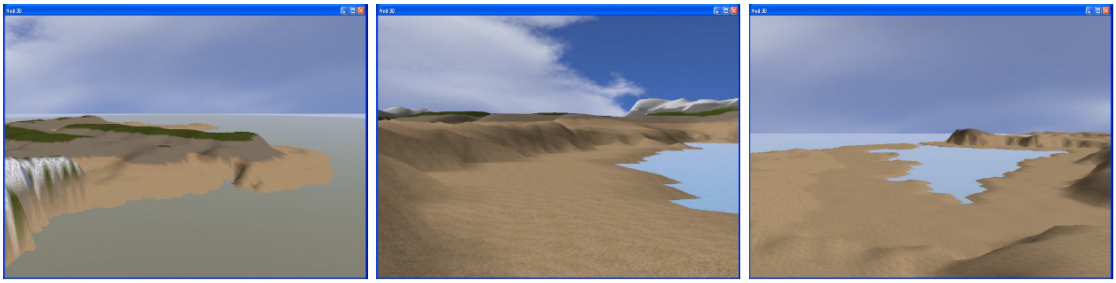


Figure 2.5: Beaches produced by beach agents with small, medium and large beach widths [14].

2.3 Road Generation

An important building block of a city is its underlying road network. This section presents the theory of various street patterns, as well as how road networks can be generated with L-systems and the A* search algorithm.

2.3.1 Street Patterns

Street patterns can have a large impact on what type of city is formed. The Gridiron grid system is an older street pattern, often used in larger cities and or in city centers, that "[...] came to America from various European sources, taking different forms in New Haven, Philadelphia, Savannah, New Orleans, and Canadian cities such as Halifax" [15]. The name came from its simple and economical layout of rectangular lots, as displayed in figure 2.6a. A newer take is the Cul-de-sac street pattern. Paul K. Asabere describes that "The cul-de-sac (or culs-de-sac in French) is essentially a dead-end street with a turn- around at the end for cars." [15]. Asabere subsequently explains the flexibility and creativity of the street pattern, and how it reduces the density of pedestrians and bicycles, making it popular in newly produced neighborhoods where roads do not have to be connected. Culs-de-sac is mainly used in suburbs. An example of the street pattern is displayed in 2.6b.

(a) Gridiron street pattern of San Francisco [16].

(b) Example of the Cul-de-sac street pattern [17].

Figure 2.6: Comparison of the Gridiron and Cul-de-sac street pattern.

2.3.2 L-systems

Road networks may be generated with the utilization of Lindenmayer Systems, or L-systems. L-systems are a special case of generative grammars and are commonly used to generate visualizations of expanding vegetation.

Grammars consist of a set of symbols with rules for rewriting a string, where each symbol in the string is replaced with the symbol dictated by the rules. The string of a grammar consists of terminals, non-terminals and an axiom, which is the initial string. A grammar may also associate parameters with rewriting rules, to modify certain properties of the result [18]. Rules in a non-deterministic grammar may either decide randomly which valid replacement should occur or use the parameters to determine which rule is the best fit, while deterministic ones only have a single

derivation [1, p. 75]. A defining feature of L-systems is that they in each iteration rewrite the symbols in a given string in parallel, which is displayed in figure 2.7.

$$
\begin{aligned}
&Rules: \\
&\quad A \rightarrow AF \\
&\quad F \rightarrow Ff \\
&Axiom: AF \\
&Iteration1: AFFf \\
&Iteration2: AFFfFff \\
&Iteration3: AFFfFffFfff
\end{aligned}
$$

Figure 2.7: A deterministic L-system rewriting its axiom in three iterations. Notice how all symbols of the string are rewritten in parallel for each iteration.

The procedural modelling of grammars, such as L-systems, generally consists of two parts; the rewriting algorithm described previously and an interpretation of the result [18]. A graphical representation of the generated data, such as vertices, may be produced by the interpreter. Note that L-systems are not limited to the strict representation of strings. Graphs and other objects may also be represented by L-systems [1, p. 75-79].

Though L-systems were initially introduced for modelling natural structures such as plants, various extensions to the base grammar have been developed for a variety of procedural generation algorithms [19]. L-systems may, for instance, be utilized in the generation of road networks, as presented by Yoav Parish and Pascal Müller [20]. Parish and Müller utilized an extended version of L-systems for creating a variety of road networks in the generation of a city. The extended version introduced customization and modularity by utilizing more parameters and constraints, such as population density, allowing specific street patterns to be produced [20].

2.3.3 A* Search

A variety of informed search methods can be utilized for creating paths between given points on a graph [21]. Many informed search methods may be generalised as a family of best-first search algorithms, defined as generic methods finding the optimal solution to a given search problem with a given evaluation function [21]. An evaluation function is, in this case, a function describing the desirability of expanding the search in some direction. The desirability can, for instance,

be defined as selecting the node with the cheapest accumulated cost from the start node, as in Uniform-cost search, or as selecting the node with the cheapest estimated cost from the goal node, as in Greedy best-first search [21].

The A* search algorithm can be defined as a mixture of the Uniform-cost and the Greedy best-first search algorithm [21]. The evaluation function of A* search combines the evaluation functions of the two algorithms, by expanding the search with the node that has the cheapest accumulated cost of reaching the node, summed with the estimated cost of reaching the goal. Ultimately, this combines the efficiency of Greedy best-first search and the completeness of Uniform-cost search, resulting in always finding the optimal solution [21]. In the context of procedural generation, the A* search algorithm can, for instance, be utilized for the generation of road networks between connecting cities by finding the optimal path between them.

2.4 City Generation

Several aspects, such as buildings, transportation-systems and population density need to be taken into consideration in the generation of realistic cities. This section covers how urban cities are historically built and how city elements may be visualised in heat maps.

2.4.1 Urban Cities

Real cities are built through many generations and have deep cultural roots forming its architecture and surroundings, such as parks, roads, public transportation, policies, hospitals and buildings [22]. Urbanization of larger cities, or Large-scale Urban Development (LUD), may be studied for insights into city development with increasing populations [23]. LUDS are a way of assuring that growing populations both have access to housing and that the housing is developed with the environment in mind. Today's buildings are generally taller than in the past, meaning that they are capable of housing more people in the same area than older buildings were [23].

2.4.2 Heat-Maps

Heat maps can be defined as graphical representations of collective data visualized with colours. Generally, they consist of pre-processed images representing pollution or heat concentration over a geographically bound region with a colour index scale, commonly blue to red [24]. Heat maps may be utilized for representing a variety of data, such as the population density of a city. In another example, heat maps were used for demonstrating positive and negative emotions in different

population areas from the collection of text information from digitized archives, such as Twitter or Google Books. Here, positive and negative words connected to emotions were scanned from individuals around the world. A heat map visualizing global happiness was subsequently created from the collective data [25], which is displayed in figure 2.8.

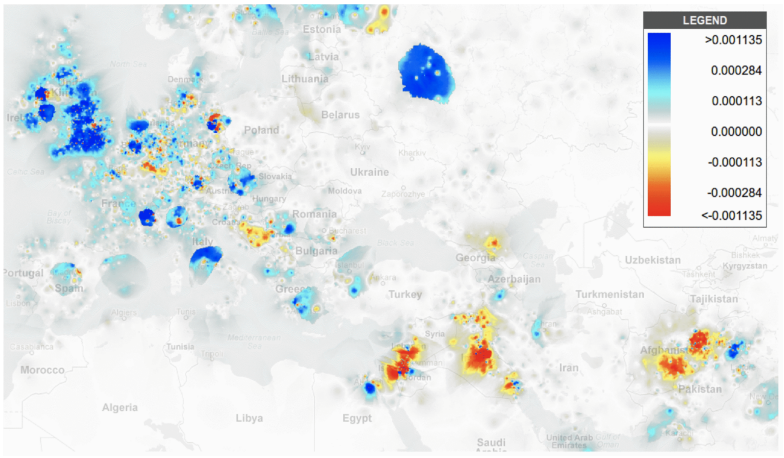


Figure 2.8: Emotional heat map of the globe zoomed over Europe [25]. Darker blue colors represent happiness, while darker red represent sadness.

2.5 Unity Real-Time Development Platform

Unity is a highly popular development platform owned by Unity Technologies and is often used in game development. As of 2020, 3.3 billion devices have been reached with Unity-made content [26], [27].

The Unity development platform can be extended to improve the workflow for developers. Extensions to the Unity editor can be created with editor scripting. Examples of components that may be added to the workflow are custom windows and various menu controls. New functionality may also be added to the scene view [28]. An official Unity API for editor scripting is well documented by Unity Technologies [29].

Many examples of Unity assets are available on the Unity Asset Store [30]. A few examples of editor extensions are *ProBuilder*, *Playmaker* and *Shader Forge*, all of which have custom graphical interfaces, adding new functionality to the Unity editor [31]–[33].

ProBuilder is a Unity tool used for designing levels and creating 3D models. The

extension has been an official, but optional, addition to the Unity editor since the 2018.1 version. Examples of games utilizing the tool are *SUPERHOT* and *Tunic* [31]. An example of *ProBuilder*'s graphical interface is displayed in figure 2.9

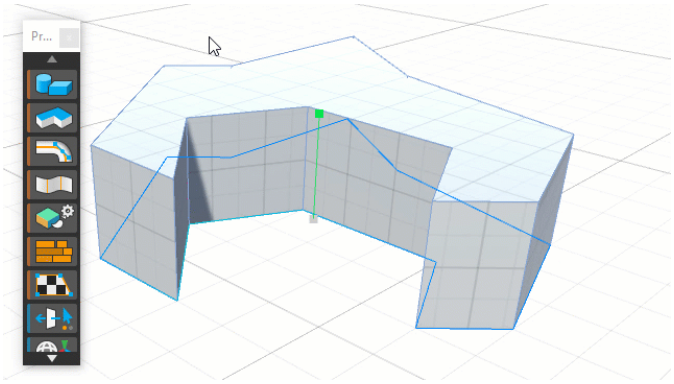


Figure 2.9: The graphical interface of *ProBuilder* [34].

Playmaker has the highest rating in the Unity Asset Store and is a visual scripting tool, used for creating in-game behaviours without writing code. Behaviours are instead defined with state machines in a custom graphical interface in Unity. Two examples of games that utilize the tool are: *Hearthstone* and *Hollow Knight* [32]. The graphical interface of *Playmaker* is displayed in figure 2.10.

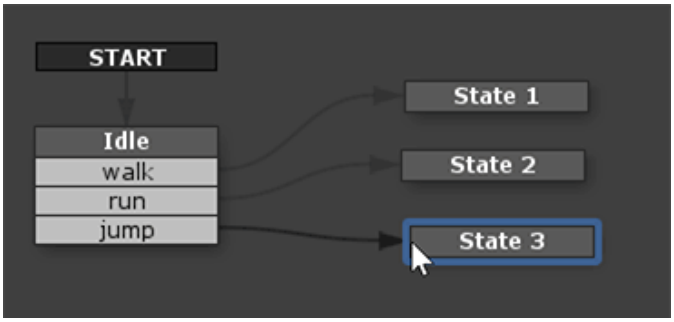


Figure 2.10: *Playmaker*'s graphical interface.

Shader Forge was released to the Unity Asset Store in 2011 but has since then been removed [35]. It is, however, still available on GitHub as an open-source repository [36]. It is a tool for creating shaders without writing any code and has its own custom graphical interface based on nodes. The tool has been promoted by game developers known for games such as *Dear Esther* and *Knytt Underground* [33]. The graphical interface of *Shader Forge* is displayed in figure 2.11.

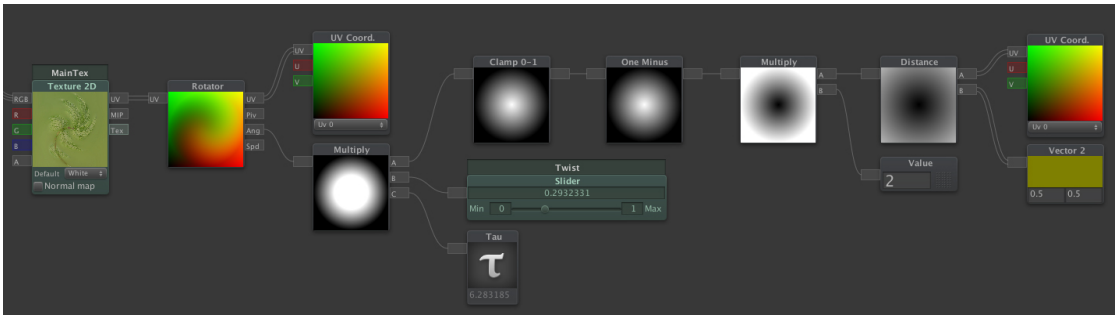


Figure 2.11: *Shader Forge*'s graphical interface [35].

3

Process

This chapter presents an insight into how the project progressed from the first ideas to the final results. It covers initial planning and design decisions, methods used for developments as well as the implementation's progression and design decisions made along the way.

3.1 Planning

After agreeing on a purpose, goal and scope of this project, as presented in section 1, the issue at hand was planning the implementation of the procedural generation asset. This section covers how the development methodology and tools were initially decided, as well as how initial features were selected and problems were identified.

3.1.1 Methodology

A variety of models and approaches may be used for developing and managing software projects. As the project had a relatively open nature without any final feature specifications defined initially, an iterative approach was deemed suitable. As a result, linear methodologies, such as the waterfall model [37] were to be avoided.

Developing an agile project iteratively requires objectives to be organized into shorter time-spans, with the implementation performed in parallel between all team members. The agile methodology Scrum [38] was deemed well suited for these requirements since it is tightly connected with iterative processes. Most team members also had prior experience developing projects with this methodology. However, following the Scrum methodology to the fullest extent would not be possible due to external factors, such as team members attending different courses, exam weeks and longer periods of self-studies. As a result, only the important

outlines of scrum were to be followed.

The implementation was initially divided in a set of sprints consisting of the agreed time-span of approximately two weeks, with the time-span adjusted to the current study situations of the team members. Sprints should begin with meeting for scoping objectives to achieve during the sprint, as well as dividing tasks between the team members. Objectives were to be selected based on their priority for the overall project. Sprints were to be subsequently finished with a review and reflection meeting, to reflect on the sprint's outcome. Team members were to frequently meet during sprints as well, to synchronize their progress.

3.1.2 Development Tools

Several tools were deemed both suitable and necessary for developing this project. Since the primary focus lied on developing a tool for procedural generation and not the creation of a game engine, the Unity Platform described in section 2.5 was selected as the primary development tool for this project [26].

Since a majority of the team members did not possess much game development experience before this project, it was deemed that the learning curve of Unity was reasonably steep to still be able to learn how to use it in the time-span of this project. The C# programming language was subsequently selected as the language to develop this project in, as a result of the close integration between Unity and C#. As several group members had prior experience with the language, this was another reason for developing the procedural generation asset in Unity.

Other development tools were also researched and deemed suitable for this project. However, they were not utilized for the final product, due to the time-span and scope of this project. An example is Blender [39], which could have been used for creating 3D assets that are not created directly by the tool. Since the focus lied on developing a tool for procedural content generation rather and not on creating 3D assets, Blender was never utilized in the final product.

3.1.3 Problem Identification & Initial Feature Selection

An early challenge was determining how a procedural city may be represented in Unity as well as identifying the required components for composing a procedural generation asset. The possibilities of representing a city with two-dimensional isometric layers were initially discussed. However, to better showcase the potential of utilizing procedural generation in the creation of cities, a three-dimensional approach was chosen instead. Identifying components of a city were performed with a logical

approach. Cities consist of a set of buildings with surrounding road networks. Buildings and road networks subsequently lie on top of the terrain. As a result, the three components of terrain, road networks and buildings were identified. It was subsequently decided to develop the generation of these components separately, in the previously mentioned logical order. A bigger challenge was also how the three components are to be combined into a functional and sustainable asset. However, this challenge was to be addressed further on.

Since a wide range of procedural generation methods and algorithms may be utilized in the creation of cities, a secondary objective was to research commonly used methods to see how they are currently used in game development. The methods and algorithms were subsequently evaluated on their suitability of generating the various components of city environments. The suitable candidates, as presented in section 2 were then selected for the implementation.

3.2 Implementation

The process of implementing the tool has been incremental in two senses. Firstly, the agile methodology similar to Scrum, described in section 3.1.1, and secondly, how features were selected. It was concluded that the most straight-forward approach of generating cities and surrounding terrain were from the ground up, which is similar to how cities are physically built in the real world and influenced the order of which the features were selected. As a result, the procedural generation were implemented in the order of terrain, road-networks, plots and buildings. The order was deemed the most logical and was not chosen according to what components have the highest priority.

The implementation heavily followed several object-oriented design principles and patterns, which are described in more detail in section 4.3. The high cohesion and low coupling principles [40], as well as the S.O.L.I.D principles, were followed to the best ability, to ensure a modular, maintainable and extendable tool. The latter set of principles consists of The Single Responsibility Principle , the Open-Closed Principle, the Liskov Substitution Principle, the Interface Segregation Principle and the Dependency Inversion Principle [41, pp. 51, 156] [42].

Early on, it was also decided to utilize well-known design patterns, such the Strategy and Model-View-ViewModel patterns, to solve common problems that were prone to happen during the tool's development [43, pp. 349-359][44]. Design patterns may be formally described as "[...] descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [43,

p. 4]. However, a specific set of patterns to utilize were not initially decided but grew during the implementation time to solve problems that came up along the way. The patterns that define the higher-level structure of the final implementation are described in section 4.3.

The selection order and implementation of features, as well as design decisions, are presented in detail in the following sections. The procedural generation features of the final implementation are described in section 4.2.

3.2.1 Terrain Generation

Initially, it was deemed necessary to start generating terrains, on which cities later could be generated. Thus, the first step of building the asset was to implement the procedural generation of 3D terrains. At this stage, two key components for the terrain generation feature where identified.

Firstly, generating the curvature of the terrain was required. It became apparent that the, perhaps, most common and most documented method was using two-dimensional noise maps as explained in section 2.2.1. However, since the noise maps are in two dimensions, a conversion of the 2D height-map into a 3D were required to be able to display it to the Unity scene, which was an obstacle to overcome. Secondly, without textures, 3D terrain meshes would look gray and dull, which does not represent plausible terrain. As a result, the second key component was to create plausible terrain textures.

With the key components in mind, the implementation of terrain generation could begin. Perlin noise was quickly identified as a good starting point for generating 2D height maps. Although Simplex noise is superior to Perlin noise in many ways, a decision was made to use the latter, since it is implemented natively in Unity. As mentioned in section 2.2.1, both are common approaches for creating artificial terrain with a continuous curvature and realistic looking shapes in both greater and smaller detail using a variety of parameters. Initially, the number of parameters were few, since the noise map only made use of one octave of noise as seen in figure 3.1. This was thought to be too artificial in appearance and several octaves were therefore combined to provide more detailed terrains. Nonetheless, it was quickly identified that storing the terrain as one large 2D height map is not a viable approach in the long run. Thus, generating and storing the terrain height-map in smaller, yet still continuous, parts were a potential obstacle to overcome later on.

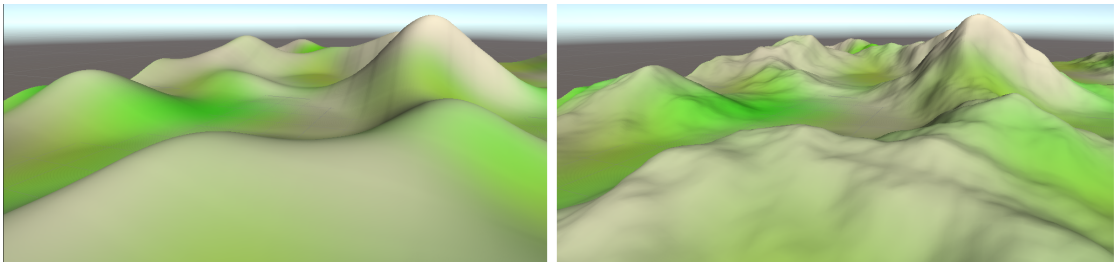


Figure 3.1: Comparison of terrain generation using only a single octave (left) versus using four octaves (right).

To convert the generated 2D height-map into a 3D mesh, a mesh generator with a height-map as input was created. This could be utilized as a common object for converting terrain from two dimensions into three independently of the algorithm generating the 2D map. ensuring that the mesh generator was referencing valid height-maps was, however, also an obstacle to overcome later on. For more details, see Appendix C.

For the creation of realistically looking terrain textures, it was decided that using Whittaker biomes, as described in section 2.2.2, were a reasonable starting point. To not spend too much time and effort on textures, however, only a simple version of Whittaker biomes was implemented. In parallel, a gray-scale texture generator was also created for visually debugging a height-map generator when it is displayed. Since both generators required a reference to a height-map, the obstacle of ensuring valid references arose here as well. Since this obstacle seemed recurrent, the necessity of finding a reasonably robust solution grew larger.

At this stage, a few important design decisions influencing further implementation were made. Firstly, it was decided to utilize a version of the Strategy design pattern to handle switching between different implementations of terrain and texture generation, as described in section 4.3.5 [43, pp. 349-359]. This decision ensured a low coupling in the system architecture and ensured that the tool is open for extension. The Strategy pattern also remained useful for handling similar scenarios in later implementation of road networks, plots and buildings. As a complement to the Strategy pattern, it was also decided to utilize a version of the Factory Method pattern. Section 4.3 covers these patterns in more detail. As a result, concrete strategy implementations could be kept hidden from other assemblies, yet still, be used internally. This ensured a lower coupling between the modules even further, as described in section 4.3.2.

To provide a robust solution for ensuring dependencies to valid object references,

as described earlier, it was deemed reasonable to use a version of dependency injection, as described in section 4.3.7. This ensured that all concerned objects can rely on an injector providing valid references, instead of ensuring the validity by themselves. This required a bit of refactoring, which may have slightly stalled the overall progression. It did, however, result in a sustainable solution further on.

Terrain generation was initially tied together in a public generator object utilizing both dependency injection for noise maps, as well as the strategy pattern for the generation of textures and 2D noise maps, before converting the noise map into a 3D mesh. The terrain generator was later refactored into a view-model, as described in section 4.3.3.

3.2.2 Road Network Generation

With a functioning generation of terrain, the next step was the generation of road networks within and surrounding cities. Firstly, a data structure was implemented for storing the data of a road network and handling the network's expansion. The road network was based on directed graphs and was designed to make it easy for procedural generation strategies to add road vertices without concerning themselves with issues such as finding intersections, which is handled by the road network class.

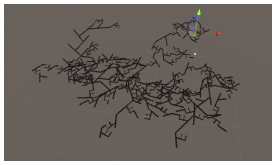
After adding the logic of road networks, a way of extracting the vertices of the network had to be implemented as well, for purposes such as visualizing them in a Unity scene. Firstly, only vertex pairs were retrievable from the road network as road parts, making it possible to visualize it, but resulted in an unnecessarily large amount of Unity game objects. A second functionality was subsequently added to the road network class that recursively extracts road vertex sequences as far as the inner adjacency list goes. As a result, fewer game objects were created, which improved performance.

Three aspects of creating a believable road network were quickly identified. Road networks of a city were first assumed to mainly consist of several expanding cores of road networks. Cities secondly consist of several connected cores. Thirdly, the location of city cores is generally determined by external factors such as religious buildings, economics or population density. As a result, methods for creating road networks, both expanding around a point and between given points, were required, as well as a method for simulating the location of cities.

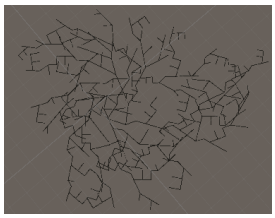
To simulate an expanding city core, a version of L-systems was utilized to generate a lush web of road networks, as described in section 2.3.2. L-systems have no preset

goal. Instead, they branch out in random directions with a set of rules determining their shape, which may be used for simulating, for instance, roads growing to make space for buildings in the real world.

The initial version of L-systems presents a promising road network but contains the unrealistic flaw of road vertices growing across each other, as displayed in figure 3.2a. As a result, the second version of L-systems was required to reduce the number of clustered intersections. As displayed in figure 3.2b, the second version of L-systems has a less cluttered look and largely avoids the issues of multiple road vertices converging in one location. This version was subsequently utilized and expanded to also generate grids similar to the gridiron street pattern, as described in section 2.3.1



(a) First version of L-systems generated in six iterations.



(b) Second version of L-systems generated in eight iterations.

Figure 3.2: A comparison of the first and second implementations of L-systems.

To connect several city cores and cities, the A* search algorithm was utilized, as described in section 2.3.3. A* search finds the most optimal path between two given points on a graph, which is suitable for generating a road network between the different city and city core locations. One requirement was to connect multiple cities and city cores with one road network. As a result, a version of A* search handling multiple starts and goal locations was implemented.

Several potential issues arose during the implementation of the A* search algorithm. Firstly, finding an optimal path sometimes resulted in road networks with non-plausible paths, such as crossing the peak of a steep mountain instead of circling it. Secondly, finding multiple paths between points located close to each other often resulted in several road networks instead of using an already existing one. However, since these were mainly edge cases, they would only be addressed if the problems were still apparent in the later application.

Simulating population density was deemed to be the most straightforward approach of determining the locations of city cores without having to depend on complex real-life factors. The simulation was to be performed by generating a heat map

of the population density similar to noise, as described in section 2.4.2. This would provide unpredictable locations of city cores which gradually transition into suburban and rural areas using the values of the heat map. Unfortunately, the generation of heat maps was never utilized in the final implementation as a result of time constraints towards the end of the project. An early version of the implemented heat map is displayed in figure 3.3.

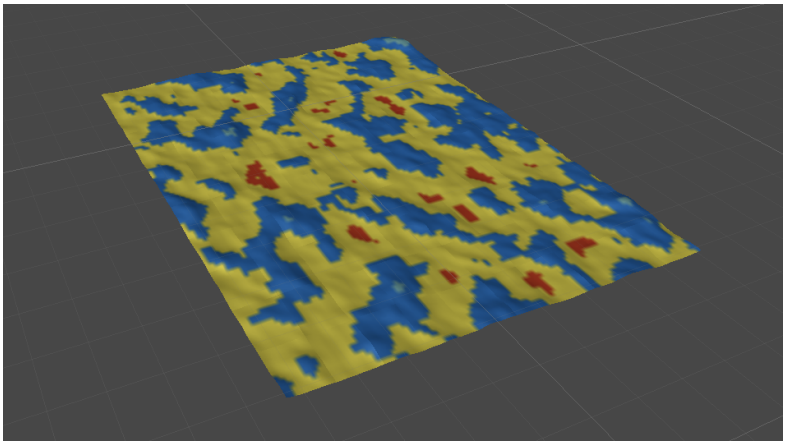


Figure 3.3: Early version of a heat map with density represent from high to low with the colors red and blue.

The generation of road networks was utilized as a strategy of a city generator class, before being refactored into a view-model. Since all prior design decisions were still relevant except for minor refinements, no major design changes were made at this stage. One refinement was the addition of an abstract strategy class to resolve code duplication. Though this may have created the possibility of future inheritance issues, it was proven to have been a suitable solution for more readable code.

3.2.3 Unifying Road Networks With Terrain

The generation of road networks and terrain were subsequently unified, resulting in quite realistic networks on a terrain mesh. The A* approach used to generate a road between two points had always followed the terrain's curves by only moving along it. However, the L-system strategy could expand in almost any direction. As a result, the latter strategy could generate roads that intersect the terrain mesh in a Unity scene. To combat similar intersections, measures had to be taken.

The first attempt of eliminating roads intersecting with terrain was to round the float values of each road vertex to determine the height location from the terrain's

height-map. The y-coordinate did, however, not correspond to the exact floating-point of the road vertex, resulting in jagged roads. They did not, however, intersect the terrain as much as before.

The quite successful and final attempt eliminating road intersections with terrain was to traverse all road vertices and downwards cast a ray from each one to find where it intersects with the terrain. Each y-coordinate of the ray/terrain intersection points could then replace the original y-coordinate of each corresponding road vertex. This solution looked better than the first attempt since it allowed the use of floating numbers, resulting in roads following the terrain more closely. An example of how this approach looks in a Unity scene is shown in figure 3.4.

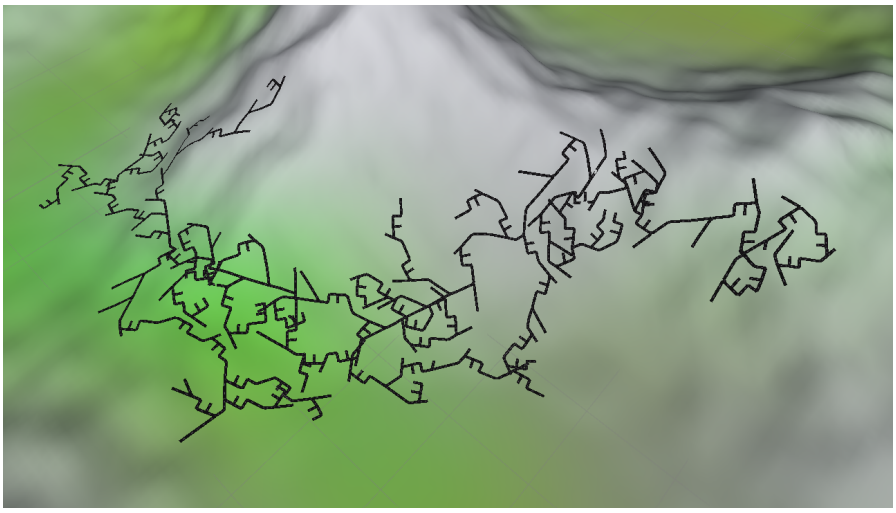


Figure 3.4: A road network based on an L-system, mapped to the heights of the terrain.

3.2.4 Generating Plots

After the generation of road networks were implemented, the next step of procedurally generating a city was determining the location of buildings. Placing buildings within plots along road networks was a logical starting point since plots define an area where a building can be placed. By only locating buildings within plots generated from road networks, they are also guaranteed not to intersect with any road vertices.

Initially, two types of plot generation were implemented. The first type located plots by finding minimal cycles within a road network. This version could be utilized for creating plots that, for instance, resemble areas within a city core. However, finding minimal cycles proved to be quite complicated. A first implementation

was functional but too time-consuming to be used since Unity froze for long time-periods when the generation was ongoing. This implementation recursively searched all vertices of a road network and saved the cycles it found. The cycles were subsequently sorted by size and iterated, were only non-overlapping cycles and the smallest of overlapping cycles were returned as plots.

The second implementation of minimal cycles was implemented as an attempt to speed up the algorithm. The second implementation provided similar results with a tremendous increase in speed. Is was, however, not perfect, since some plots were generated along protruding roads, resulting in buildings with strange spikes. Several unsuccessful attempts were made to fix this issue, but since strange-looking buildings were deemed better than a very time-consuming generation, this is the version of minimal cycles utilized in the final implementation.

However, as all plots of realistic cities are not located within cycles of roads, another type of plot generation was required as well. As a result, a second version plot generation was implemented to generate plots along single roads, resembling areas with less dense road networks. This was implemented with an algorithm that randomly generates plots adjacent to a road. A difficulty here was to ensure that plots neither collide with each-other nor with any road. Preventing plot overlaps was later solved with the separating axis theorem [45].

The various methods of plot generation were finally utilized as strategies in a view-model. An initial idea was also to combine the generation of cyclic and adjacent plots to generate a set of plots resembling a relatively believable city. The combination was later implemented as an additional plot generation strategy.

3.2.5 Generating Buildings

As mentioned previously, it was decided to let plot generation dictate where buildings are placed to facilitate sensible building placements. For this reason, all plots are assigned exactly one building. As an extension of this, it seemed natural to allow the plot to influence the building form as well. By using the polygonal shape of the plot with a set of rules applied for removing sharp angles and small protrusions, a building footprint can be constructed for a plot. In its simplest form, this strategy for building generation can extend the footprint upwards to a height determined by the population heat map. In a more advanced form, the footprint could act as a seed which would continue to be procedurally modified in segments that extend upwards, forming floors.

Implementing the basic footprint generation was mostly straightforward since three-

dimensional shapes from plots' polygonal outlines could be extruded. Since plots can be generated in any shape, it is not sufficient to simply connect vertices for roofs in the same manner as that done for the sides, and possibly concave polygons with variable numbers of vertices must be triangulated.

In the first attempt at building generation, meshes of buildings were displayed incorrectly. This was a result of Unity's representation of meshes, making use of duplicate vertices for hard edges such as the vertices between a wall and a roof. When using shared vertices in such scenarios, the single normal contained in the vertex data is averaged across the adjacent planes, resulting in incorrect lighting. This issue was solved by simply duplicating vertices to ensure that separate planes are formed over the mesh.

Some difficulties also arose with the triangulation of building footprints, due to edge cases with inconsistent plot generation and the triangulation algorithm itself. The triangulation was finally refactored and the basic form of building generation finalized.

One problem with building generation is the terrain placement, similarly as in the generation of road networks. Unlike the plots they are placed on, buildings are required to visually follow the curvature of the generated terrain. Unlike roads, however, the criteria for consistent building placement on the terrain is not as strict. It is enough to ensure that the building does not float above the terrain and that it extends above it. The generation algorithm accounts for this by checking the minimum height values of the underlying height map under the polygonal shape of the building in the XZ-plane and performs bi-linear interpolation to find correct height values.

Similarly to the generation of terrain, textures, road networks and plots, building generation was eventually utilized as a strategy in a view-model.

3.2.6 The Editor

During the development of the tool, a variety of demo editors were created to, for instance, test the various algorithms. This process seemed appropriate at the time, since the implementation and testing could be divided between the team members and performed separately. For the final implementation, a new editor was created to form a unified asset. This editor aggregated all finished functionality, based on knowledge from earlier editor prototypes and was incrementally improved in terms of appearance and functionality.

One example of improvements was the utilization of the Model-View-ViewModel

architectural design pattern, as described in section 4.3.3. Another improvement is the utilization of the Publish-Subscribe pattern [46], [47]. Initially, editor controls could not access the values of other controls. This was a problem for editor elements, such as the origin controls for the L-system algorithm for road networks. These controls initially did not depend on the size of the terrain even though they should have, since roads should not be able to be generated outside of the terrain. As a result, a version of the previously mentioned Publish-Subscribe pattern was implemented. This allows editors to be notified whenever values of other editors are updated, for instance when the terrain size is altered. This is just one example of where the event bus is useful, but it can be extended to create even more advanced editors. This is useful when more complicated behaviour may be needed in the user interface in future implementations.

3.2.7 Assembly Definitions

A problem with the initial Unity solution structure was that the architecture was not divided into different C# assemblies, as a result of difficulties with creating different internal projects within the Unity solution. The process was unfamiliar since it is not similar to the one of a "normal" C# solution. The issue resulted in the possibilities of internal classes of abstractly defined modules, or namespaces, to be externally accessible, even though they should only be accessible via polymorphism through factory classes, as described in section 4.3.6. As a result, it was possible to ignore the intended modular design and it was up to the developers to be disciplined and to evaluate each dependency to ensure a modular code-base.

The issue was later solved when assembly definition files were discovered when searching deeper into the Unity documentation [48]. Using assembly files led to what was desired in the first place; a truly modular code-base, as described in sections 4.3.1 and 4.3.2.

3.2.8 Making Things More Responsive

During the majority of the development process, the tool was slow and froze Unity whenever a new city generation was invoked. This made the tool feel sluggish and it was clear that parallelism (multithreading) or concurrency had to be introduced, for it to be more responsive. To accomplish this, the graphical user interface was separated as much as possible from any of the tool's computations using the Task Parallel Library of .NET [49], [50]. Only the creation of game objects and their visualization in the Unity scene still had to be dispatched to the main thread. Everything else, meaning the generation of terrain, roads, plots and buildings could be done in parallel, or concurrently depending on the number of CPU cores. A

few algorithms were even be split up into separate, parallel, tasks, thus probably reaching results faster. Time differences were, however, not studied extensively.

For other threads to invoke actions and functions on the main thread of Unity, some sort of dispatcher had to be implemented. The Singleton pattern [41, pp. 201-204] was used to implement the *Dispatcher* class, which is a thread-safe singleton type. Due to its singleton properties, it is statically accessible from anywhere. This enables any generator, that needs to do work on the main thread, to dispatch it there. The *Dispatcher* enqueues actions and functions and invokes them continuously on the main thread with the help of the *DispatchInvoker* class.

When the procedural generation ran on separate threads and Unity did no longer freeze, a new problem emerged. The user was able to press the generation button multiple times and thus start parallel generations. This resulted in a buggy behaviour and had to be prevented. The solution was to replace the generation button with a cancel button whenever it was pressed (see figure 3.5). The cancel button would then convert back to a generation button whenever the generation finished on its own or when it was pressed, thus cancelling the current generation.

After adding concurrency and a cancel button, one flaw remained. The user had no way of telling how far along the city generation was after invoking it. A progress indicator was therefore added, based on events sent on the event bus described in sections 3.2.6 and 4.3.4. The time of each generation is unknown, but the progress in percentage could be communicated to the user, as seen in figure 3.5.

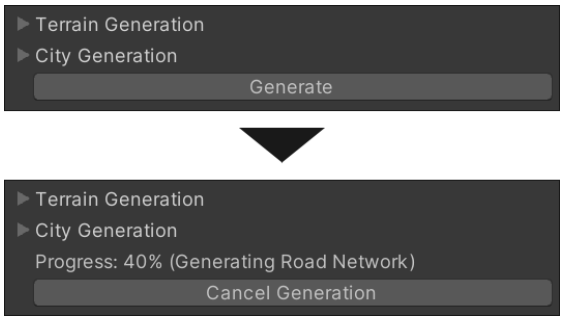


Figure 3.5: The editor's progress indicator and cancel button for city generation.

4

Results

This section presents the resulting Unity asset of this project, its underlying system architecture and how a variety of methods for procedural content generation is utilized in the generation of cities and surrounding terrain in the final implementation.

4.1 BIAS

Rome was not built in a day. With procedural generation, however, it may have been built even faster. BIAS, or *Built in a Second*, is the resulting product of this project. The full BIAS asset is displayed in figure 4.1. In a perfect world, BIAS would have been published on the Unity Asset Store as a downloadable asset for other developers to use. However, at the time of finishing this project, it is only accessible from within the source code.

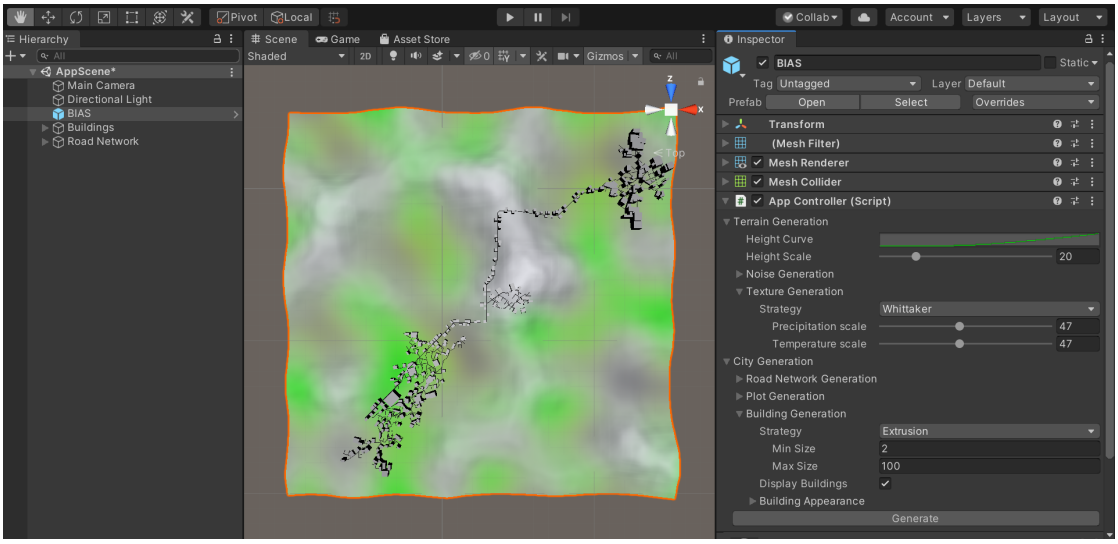


Figure 4.1: Full view off the BIAS asset in the Unity editor with the component of the asset highlighted.

The BIAS asset is defined by a prefabrication file, BIAS.prefab, acting as the template for creating a game object controlling the procedural generation of a city and surrounding terrain. It is accessible from the Unity Platform by creating a new empty scene and dragging an instance of the asset's prefabrication file into the scene. By highlighting the game object of the asset, an editor for controlling the procedural generation is displayed in the Unity inspector, as visualized in figure 4.2 and fully displayed in figure 4.1.

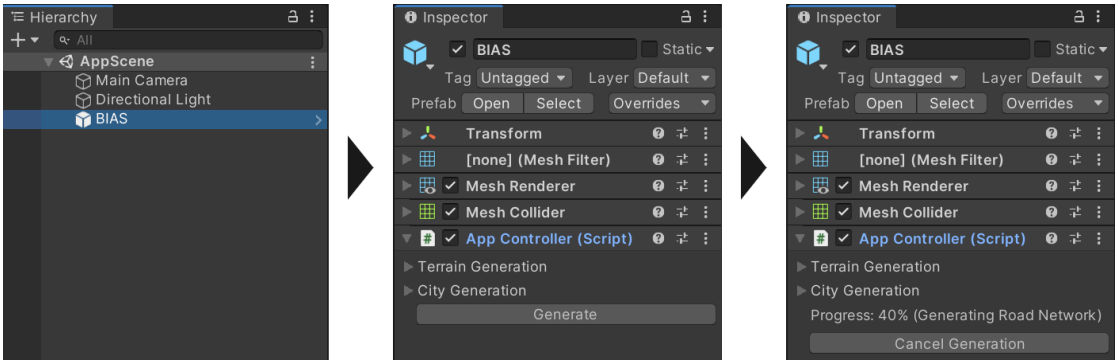


Figure 4.2: Accessing the editor of BIAS by selecting the game object of the prefabrication file in the Unity scene hierarchy. Procedural generation of a city and surrounding terrain is subsequently initialized by clicking the *Generate* button.

BIAS's editor consists of an accordion menu for controlling all input parameters

of the procedural generation, with sections each controlling the parameters of one aspect. The editor also contains a button for initializing and terminating the procedural generation as well as a progress indicator for the generation process. In figure 4.3, the section for controlling the parameters of terrain generation is displayed, with subsequent subsections for controlling the strategies of noise and texture generation. As displayed, the procedural generation is initialized, with a progress indicator and the possibility of cancellation visible. In the figure, the editor section for controlling the generation of cities and subsequently the generation strategies of road networks, plots and buildings are also displayed, but with the respective accordion closed. The full editor sections of city generation are displayed in figure 4.4.

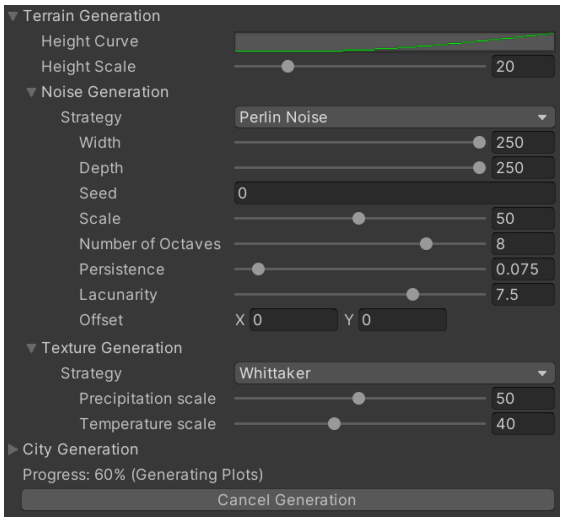


Figure 4.3: BIAS's editor sections for terrain generation parameters. In the figure, the Perlin noise and Whittaker texture strategies are selected with respective parameters displayed. Notice how the procedural generation is ongoing, with the progress indicator visible.

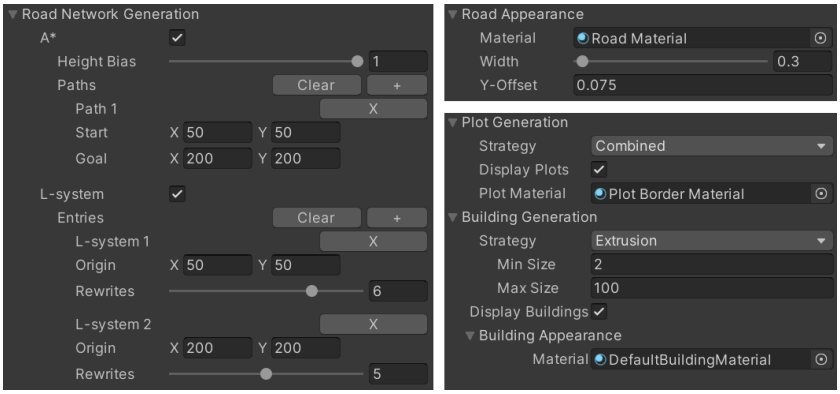


Figure 4.4: BIAS's editor sections for city generation parameters. The road network generation strategies of A* search and L-system can be toggled on and off, where a virtually infinite amount of paths and origins may be added to customize the road networks of a city. Similarly, the appearance of road networks, as well as the strategies of plot and building generation, can be customized with respective editor sections.

As the procedural generation is initialized, a city and surrounding terrain are generated by BIAS in the Unity scene with the selected parameters. An example of how a procedurally generated city may look is displayed in figure 4.5 with the input parameters of road network generation visible.

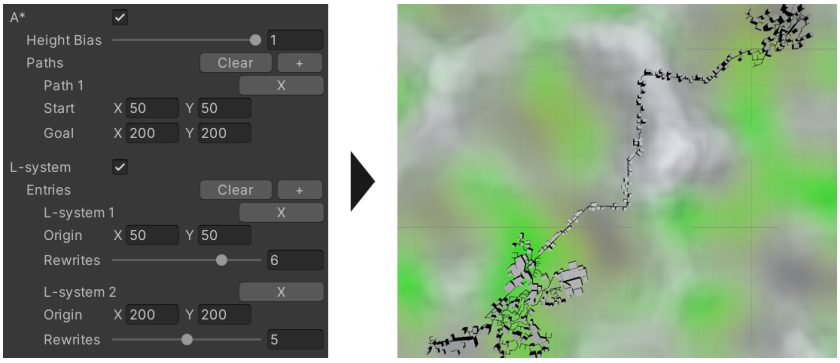


Figure 4.5: A procedural generated city and surrounding terrain with the BIAS asset. Notice how two connecting city cores are generated from the L-system origins with a A* search path created between them.

4.2 Utilization of Procedural Content Generation

Various constructive methods and algorithms of procedural content generation, as described in section 2.1, are utilized in multiple layers of the final implementation of BIAS. In the architecture, a key element is to also leave room for the possibility of adding more algorithms in future implementations.

Procedural generation of a city and the surrounding terrain is performed through several layers of delegation to several hidden generator objects. Generators on higher levels mainly delegate the generation to one or a few hidden objects and mend the results of the delegated calls. On lower levels, procedural generation is performed directly by the generators themselves.

On the highest level, the procedural generation is initialized from the user interface, where the generator method of the application controller is called. The controller itself, however, does not directly perform any procedural generation. Instead, it only updates a number of view objects with the results of delegating the generation to the city and terrain view models, as described in section 4.3.3. Similarly, the terrain and city view-models only act as another level of abstracts, and delegate the generation to, for instance, noise, texture and road network generators.

4.2.1 Noise

In the final implementation, noise is predominantly utilized for the creation of realistic terrain environments. It is, however, also used for creating textures for terrains. Procedural generation of noise is performed by creating a two-dimensional float matrix and populating it with generated values. In the final implementation, this is performed by applying a version of Perlin noise, described in section 2.2.1. Internally, this version uses a native method provided by Unity for creating Perlin noise. However, the native Perlin method does not provide any customization options. As a result, parameters such as a randomization seed, number of octaves, persistence and lacunarity were added for a more customizable solution. An example of terrain created from Perlin noise is displayed in figure 4.6.

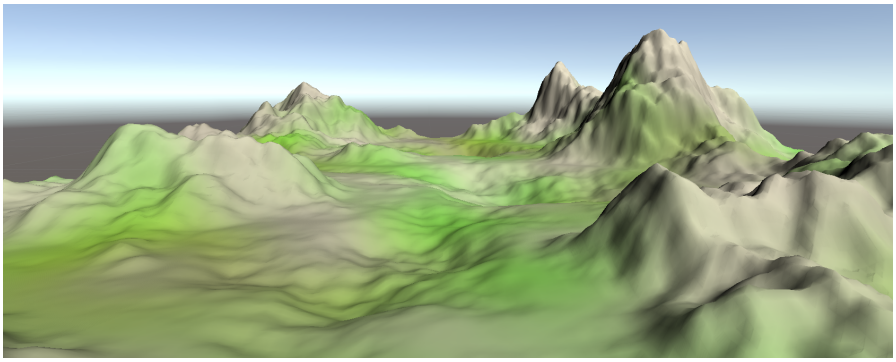


Figure 4.6: A terrain mesh generated with Perlin noise.

As noise maps are not directly displayable in a Unity scene, a mesh generator for converting noise maps into 3D Unity meshes was added as a compliment (see Appendix C). Parameters for the scale and curvature of terrain height were also added to the mesh generator to further customize the procedural generation of terrain. An example of how customizing the noise to height curvature affects terrain is displayed in figure 4.7.

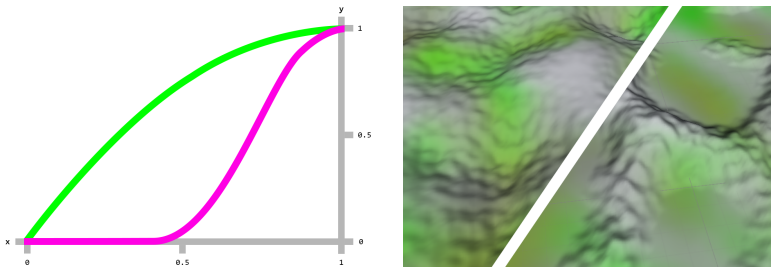


Figure 4.7: Terrain meshes based on different height curves. The left mesh is created with the green curvature while the right mesh is created with the magenta curvature. The x-axis represents noise map values while the y-axis represents mesh vertex height values.

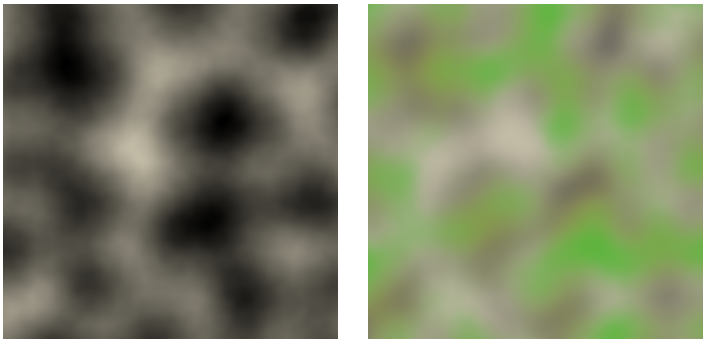
4.2.2 Textures

Procedural generation of textures is in the final implementation predominantly utilized in the creation of terrain appearance. Textures are created by initializing a two-dimensional texture object and setting each pixel to a generated color. The generation of texture colors is mainly performed by populating a two-dimensional float matrix, similar to the creation of noise maps. Colors are then generated by interpolating several colors with the values of the matrix.

The final implementation provides two versions of procedural texture generation.

One version generates a gray-scale texture representation of a given height map. The color of each pixel is calculated by interpolating between the colors black and white with the height of the element in the height-map. This version was mainly added for debugging purposes of the terrain generation. An example of a generated gray-scale texture is displayed in figure 4.8a.

The second version is utilizing a simple version of Whittaker biomes for creating textures with the appearance of the terrain, as described in 2.2.2. In addition to a height-map, the Whittaker generator utilizes noise maps to simulate precipitation and temperature in the generation of textures. The texture colors are for each pixel procedurally generated by performing tri-linear interpolation between the four colors green, yellow, white and gray. The interpolation is performed with the elements of the precipitation and temperature maps, using the height-map as a bias. This version is utilized as the main approach for generating textures. An example of a generated Whittaker texture is displayed in figure 4.8b.



(a) Gray-scale texture. **(b)** Whittaker texture.

Figure 4.8: Two dimensional textures generated with gray-scale and Whittaker biomes from the same height map.

4.2.3 Road Networks

Road networks are utilized as a basic building block of a procedurally generated city. In the final implementation, they are represented as directed graphs of vertices. A road network is created by procedurally generating roads between points and adding them to a common network. Road networks can also be expanded by merging several networks.

In the final implementation, there are two versions of procedural road network generation. L-system generation, as described in section 2.3.2, is utilized for simulating a road network in the core of a city. This is performed by iteratively expanding a road network with new roads in directions specified by a set of rules.

A few examples of rules used to add variation to L-system road networks are branching, splitting and creating gridiron patterns. As opposed to the other rules within the L-system, the one for gridiron patterns generates many road vertices per iteration, using randomly generated numbers to determine grids' size and shape. This improves the chance for the grids to grow large, and not be impeded by the rest of the road network.

A* search, as described in section 2.3.3, is utilized for connecting multiple locations, such as several city cores, with a plausible road. This is performed by finding the optimal road between the locations, given how much the road is affected by terrain heights. The algorithm will in some settings, for instance, rather place a road around a mountain than directly over it. An example of how road networks are procedurally generated with both L-systems and the A* search algorithm is displayed in figure 4.9.

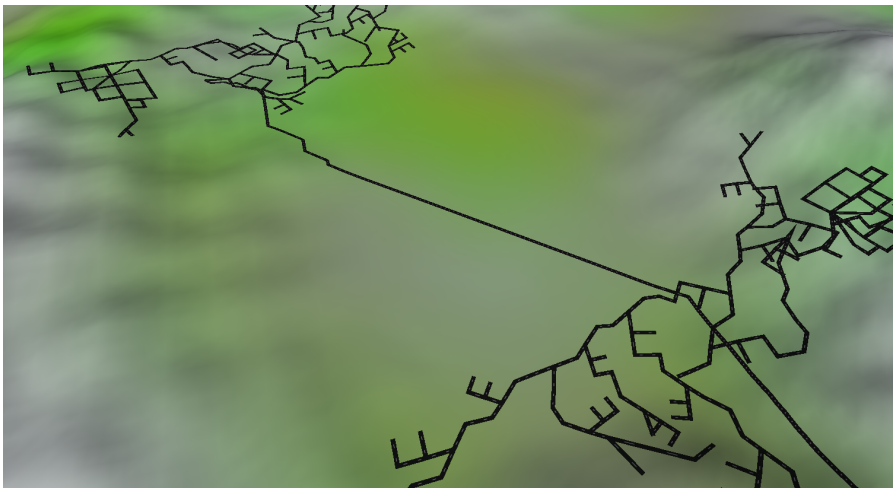


Figure 4.9: A procedurally generated road network utilizing L-systems and the A* search algorithms.

4.2.4 Plots

Plots are the first building block for generating buildings. A plot is defined by a collection of vertices encapsulating an area in which buildings can be generated. Since none of the procedural generation methods researched was directly applicable to generating plots, all versions of plot generation were implemented through trial and error. As mentioned in section 3.2.4, plots are procedurally generated from road networks and the generation is performed with three approaches.

The first approach is by finding minimum cycles within a road network. Only one of

the two implementation strategies of this approach was deemed functional enough for the final implementation. This strategy utilized an algorithm that recursively iterates adjacent road vertices by turning clockwise at each adjacent vertex. As a result, no detours are made and cycles are found relatively quickly. To subsequently extract only the minimal cycle plots generated by the algorithm, the strategy sorts the cycles by size and extracts the smallest ones. This approach is however not perfect since plot borders are prone to overlap depending on the start vertex of the search.

A second approach to the generation of plots is to place them adjacent to the vertices of a road network. The plots generated by this method are randomly sized rectangles, which are limited in size by the road vertices that it is generated along. For a plot to be generated it must first pass a series of collision checks. First, the collision with other plots is validated by using the separating axis theorem [45], which is essentially a fast algorithm for collision detection between two polygons. Secondly, the plot is tested if it is located within the terrain bounds and finally if it collides with any road vertices. If a collision with another plot is detected, multiple attempts are made to move the plot along the axis of least overlap. If they all fail, the plot is discarded.

The final approach is simply a combination of generating minimal cycle plots and adjacent plots. When the two approaches are combined, the cyclic strategy is simply executed before the adjacent strategy. An example of plot generation is displayed in figure 4.10.

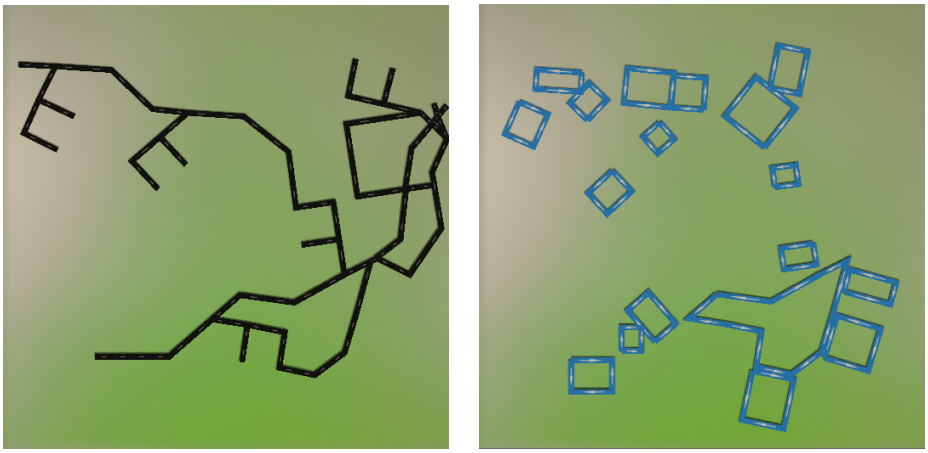


Figure 4.10: A generated set of plots surrounding a road network. Notice how adjacent plots are generated at the top of the the figure, while cyclic plots are generated at the bottom.

4.2.5 Buildings

In the final implementation, buildings are represented by meshes forming 3D boxes. The meshes are procedurally generated from the vertices of a given plot, using a type of polygon transformation called extrusion, to ensure the meshes are created within the plot. The generation of extrusion is performed by duplicating the polygon of a building base, offsetting it along an axis and constructing side faces (planes), which will connect the base to the top of the shape. This operation can be performed along any axis but is only used in the y-axis in the current iteration of the extrusion-strategy.

Ideally, a similar approach applied to the generation of buildings will permit additional transformations of the shape using slightly modified parts of the base building shape. A side face's vertices could, for instance, be offset and extruded a small distance inwards, to generate an inset of the building along a single side. Roof edges of a building could be constructed similarly. Transforming the building this way could follow some set of rules according to parameters such as plot/district type, population levels and possibly others. A grammar for the transformations would suit this requirement but is not yet implemented in BIAS.

When constructing buildings with plots in a procedural manner, it is hard to control the resulting shape. As a result, building generation must make some validation to ensure the shapes generated from the polygons are somewhat realistic. A building, therefore, has a minimum and maximum area limit, as well as minimum angles and small protrusions. The inward angle of a building corner should likely not be in the range of 1-20 degrees as that would serve little practical purpose, and such corners are removed. Examples of procedurally generated buildings are displayed in 4.11.

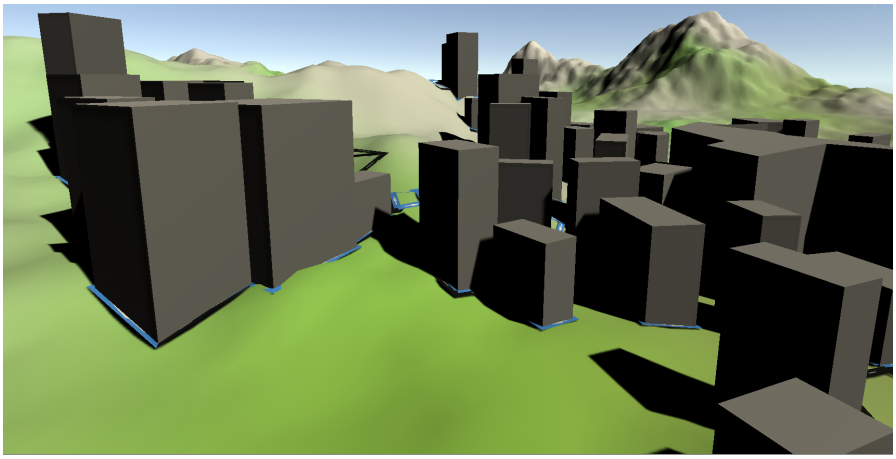


Figure 4.11: A procedurally generated set of buildings around a road network. Notice how all buildings populate a respective plot.

4.3 System Architecture

As mentioned in section 3.2, the system architecture of BIAS follows object-oriented principles and utilizes established design patterns to be as modular, maintainable and extendable as possible. This section covers in detail how the system architecture has followed the S.O.L.I.D., as well as the High Cohesion and Low Coupling principles. It also describes how design patterns were utilized throughout the architecture.

4.3.1 The S.O.L.I.D. Principles

All design decisions in the implementation of BIAS's system architecture have been taken with the S.O.L.I.D. principles in mind, to ensure a sustainable and extendable system, as mentioned in section 3.2. Most classes are designed with only one purpose, closely following the Single Responsibility principle [42]. For instance, *PerlinNoiseGenerator*'s sole purpose is to create Perlin noise-maps, while *NoiseMeshGenerator*'s only purpose is to transform such maps into three-dimensional meshes. Following this principle ensures that the components of BIAS's code-base are more reusable and ensures a lower coupling between them.

BIAS's classes and assembly modules are designed to be extendable without requiring internal modification, as stated by the Open-Closed Principle [41]. Each module and its dependencies are defined with an assembly definition file that Unity offers [48]. Since most classes follow the Single Responsibility principle, they are easily extendable internally within their respective module, without requiring

refactoring. An example of this, is how the plot generator *CombinedStrategy* effectively extends the behavior of *AdjacentStrategy* and *MinimalCycleStrategy*, by combining them without refactoring. Modules are easily extendable since their behaviors are accessible via polymorphism of abstractions such as interfaces. An example is how noise generation is accessible via the *IGenerator<float[,]>* interface, while the concrete implementation *PerlinNoiseStrategy* is hidden, internally of the assembly.

The Liskov Substitution principle is predominantly followed throughout BIAS's architecture since most classes only implement interfaces and do not inherit from concrete classes. Classes that do inherit from others follow the principle by merely extending the behaviour of their respective "parent", not altering any parental behaviour [41, pp. 51, 156]. Two examples are *ExtrusionStrategy* inheriting from *Strategy* and *CityViewModel* extending *ViewModel*. As a result, unexpected behavior from inheritance is avoided, making BIAS's code-base more maintainable.

BIAS's interfaces are as segregated as possible, according to the Interface Segregation principle [42]. This ensures that interfaces are easily reusable without creating unwanted dependencies. Examples of well-segregated interfaces are *IDisplayable*, *IGenerator* and *IInjector*. Each only has one purpose.

Finally, all higher-level modules in BIAS's architecture only depend on abstractions of lower-level modules, in agreement with the Dependency Inversion Principle [42]. This ensures a lower coupling with fewer concrete dependencies in BIAS's code-base. An example of how the principle is utilized is the Dependency Injection design pattern, as described in section 4.3.7. In short, the pattern allows objects in higher modules to depend on injectors rather than concrete classes from lower modules. Another example is the way assemblies depend on each other. For instance, the lower-level assembly *BIAS.PCG.Cities* does not depend on the higher-level assembly *BIAS.App*. The latter, however, depends on the former.

4.3.2 High Cohesion, Low Coupling

The high modularity of BIAS's system architecture is based on the High Cohesion and Low Coupling principles [40], as mentioned in section 3.2. In BIAS, coherent classes and namespaces are, as previously mentioned, grouped into assembly modules with minimal amounts of external dependencies. For instance, the texture generators *WhittakerStrategy* and *GrayScaleStrategy* are grouped into a texture assembly. As a result, modules and their internal classes are more maintainable and can easily be extended or replaced in future implementations, without affecting any types in external assemblies.

4.3.3 Model-View-ViewModel Pattern

The system architecture is, from a top-down perspective, implemented as a version of the Model-View-ViewModel, MVVM, structural design pattern [44]. Utilizing this pattern was made as a compromise to ensure that BIAS's models are sustainable and expandable, while still providing usable editors in the user interface. Standard editors provided by the Unity platform could not fulfil these requirements, since fields of abstract types cannot be visualized directly in the editors as they are not serializable. Since most models are externally exposed only by their respective abstract super-types, for instance *IGenerator<RoadNetwork>* exposing *LSystemStrategy* and *AStarStrategy*, they are not directly accessible from a standard editor. As a result, view-models are utilized to provide serializable fields for the editors in the user interface, while still being able to utilize abstract objects such as *IGenerators* internally. This ensures that the use of polymorphism is not restricted in future expansions of BIAS's models.

The template of this version of the Model-View-ViewModel design pattern separates the overall structure into four distinct objects, as visualized in figure 4.12. Only one model, view-model and view are visualised in the template. The application controller is, however, dependent on multiple view-models in the complete architecture. Similarly, a view-model may be dependent on multiple models and can create multiple views. View-models may also be dependent on sub-sequential view-models, as later described.

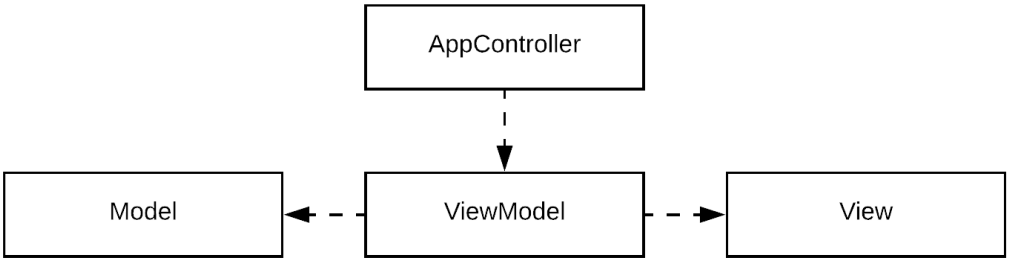


Figure 4.12: Structural relationship template in the utilized version of the Model-View-ViewModel design pattern. Multiples of models, views and view-models are abbreviated from the figure.

Models are provided, as a base, by abstract *IGenerator* objects exposing the behavior of hidden concrete implementations from various modules. The *IGenerator<float[,]>*, for instance, exposes the noise generation behavior of *PerlinNoiseS-*

trategy and *IGenerator<Texture2D>* exposes the texture generation behavior of *WhittakerStrategy* and *GrayScaleStrategy*. As a result, internal data and complex operations are stored and performed internally but are still accessible externally. This ensures that models provide their expected behaviours without exposing concrete dependencies. The models are exposed by utilizing factory objects, as presented in section 4.3.6

As mentioned before, abstract objects, such as *IGenerators*, cannot be serialized by Unity and, as a result, cannot be accessed directly from a Unity editor or stored after the application is terminated. Accessing and storing abstract objects can, however, be achieved by utilizing view-models. A view-model represents one or a few abstract objects by containing the serializable fields of the objects and implementing their respective interfaces. An example of a view-model is the *WhittakerStrategy* view-model. It contains the serialized fields *precipitationScale* and *temperatureScale* of the *WhittakerStrategy* model, and implements the same interface, *IGenerator<Texture2D>*.

With view-models, abstract objects are accessible from editors and storable after termination via the serialized fields of the view-model. The behaviors of the abstract objects are also accessible in run-time by the view-model creating the objects and delegating method calls to them. This structure allows abstract objects to be utilized internally at run-time, yet still be implicitly accessible from a Unity editor.

Many view-models are also broken down into several sub-view-models. As a result, view-models may be dependent on other view-models. The *TerrainViewModel*, for instance, is dependent on both the *TextureViewModel* and *NoiseViewModel*. This design ensures a greater separation of concern [51]. However, this also allows view-models to act as *Strategy* objects, as described in section 4.3.5.

All view-models are required to provide the same behavior of creating views, acting as a generator, as well as being able to access an event bus, as described in 4.3.4. Hence, they all inherit from the abstract class *ViewModel*. This may violate the Liskov Substitution principle, mentioned in section 4.3.1. However, unnecessary dependencies and code-duplication were avoided in BIAS's code as a result of this design decision.

Views are, in this structure, provided by the Unity platform as various menu controls, such as sliders, animation curves and number fields in a custom editor, as displayed in figure 4.13. Standard editors with standard view elements are provided by the Unity platform. Standard editors cannot, however, be used directly with non-serializable fields, as previously explained. As a result, a custom editor,

AppEditor, was utilized instead. The view elements of the custom editor are created explicitly by view-models themselves with their serialized fields. This may have violated the separation of concern [51]. It does, however, result in view-models with greater modularity and re-usability, as well as providing less restricting views with more customization options.

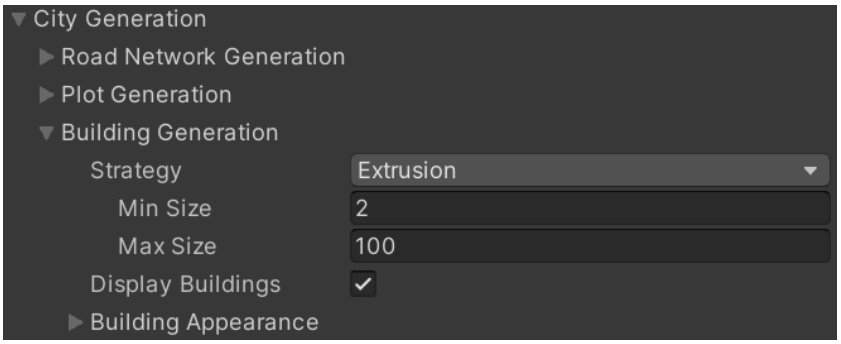


Figure 4.13: Examples of views in BIAS's editor. The figure displays fields for generating buildings, consisting of a drop-down list for selecting building strategies, number fields controlling the minimum and maximum size of buildings, as well as a switch for enabling and if buildings are displayed.

Views and view-models are, in run-time, utilized simultaneously by the application controller, *AppController*. The controller initializes the view-models and displays their respective editors via delegation. It also displays the complete results of the generated cities and surrounding terrain in the Unity scene. The generation itself is, however, concurrently performed via delegation to the view-models, as mentioned in section 3.2.8.

4.3.4 Publish-Subscribe Pattern

View-models may communicate with each other through the utilization of a variant of the Publish-Subscribe pattern, as mentioned in section 3.2.6. The pattern is similar to the Observer pattern, in the sense that both involve subscribers being notified of events. In the Observer pattern, however, publishers have dependencies to abstractions of their subscribers (also called listeners). The key difference between the two patterns is that the Publish-Subscribe pattern has an additional component, which acts as a broker. The broker, or event bus, is an object where events can be published and listened to. Publishers and subscribers therefore only have dependencies to this event bus, meaning that they do not depend on each other, as seen in figure 4.14 [46], [47].

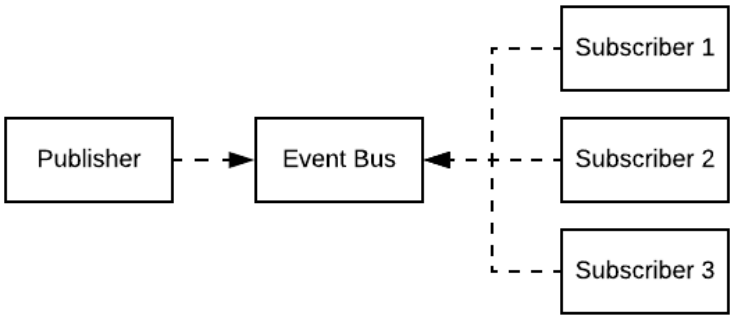


Figure 4.14: Structural relationship in this version of the Publish-Subscribe pattern. Notice how the publishers and subscribers do not have dependencies to one another.

An event bus is established by an instance of the *EventBus* class, which other objects, such as view-models, can subscribe and publish events on by implementing the *ISubscriber* interface. Hence, when events are published on the event bus, all subscribers are notified and react to the event in their predefined way. An example is the *PerlinNoiseStrategy* view-model, publishing the event of a user altering the terrain area. Subsequently, the *AStarStrategy* view-model reacts to the event, by updating its internal terrain boundaries. As a result, view-models may communicate via a common channel without being directly dependent on each other. This creates a low coupling between BIAS's view-models and ensures a more maintainable code-base.

Events are defined by a set of event identifiers in the *AppEvent* enumeration. The identifiers describe the type of events that may occur. Subscribers can thus react differently to different events. In the previous example, an event with the *UpdateNoiseMapSize* identifier is published by the *PerlinNoiseStrategy* view-model, which the *AStarStrategy* view-model reacts to.

The primary event bus is established by the *AppController* and is accessible by all view-models, as described in section 4.3.3. All view-models can act as subscribers, since the abstract view-model base class *ViewModel* implements the *ISubscriber* interface. The default subscriber behavior of view-models is to do nothing. Inheriting view-models may override the base behavior with behavior of their own. Doing nothing by default was an active design choice to ensure that all view-models have access to the event bus of the *AppController*, without having to duplicate code. This results in higher coupling of classes but is beneficial for the extendability of view-models.

4.3.5 Strategy Pattern

To easily switch between different procedural generation methods in run-time, a version of the Strategy design pattern was utilized throughout the underlying architecture of BIAS [43, pp. 349-359][52]. It is primarily utilized in the various view-models for switching the current strategy of generation. In, for instance, the *RoadViewModel*, road generation can be switched between using L-systems, A* search or both. Similarly, in the *TextureViewModel*, the generation of textures can be switched between gray-scale and Whittaker textures.

The pattern is in BIAS utilized by using an interface that defines the expected behavior of the strategy. A strategy object subsequently implements the interface and holds an internal reference to a settable object that also implements the interface. A strategy can then be switched by setting the internal interface reference, without creating any external dependencies. It is subsequently utilized by the surrounding object delegating each method call to the internal object that implements the interface, ensuring that the internal object remains hidden after it is set.

Strategies are, in the BIAS architecture, defined by the *IGenerator* interface and are created by implementing it. Texture generation are, for instance, defined by *IGenerator<Texture2D>* where two strategies implemented by the *WhittakerStrategy* and *GrayScaleStrategy*. View-models are then utilized as strategy objects, allowing strategies to be set from the user interface. Generation calls to the view-models are subsequently delegated to the currently selected, internal, strategy.

Selected strategies in the user interface are predominantly stored as enumerations, since abstract objects, such as *IGenerator* objects, cannot be stored after termination, as described in 4.3.3. Many strategies also require input from the user interface, which is not possible with abstract strategies. As a result, many strategies are hidden by a view-model delegating method calls to an internal abstract strategy. An example of this is the *LSystemStrategy* view-model, hiding an instance of its corresponding generation class as an abstract *IGenerator* object.

Many classes, such as *LSystemStrategy* and *WhittakerStrategy*, inherit from the abstract *Strategy* class. This resulted in less code-duplication but may violate the Liskov Substitution principle, mentioned in section 4.3.1. Any potential sub-classes of such strategies will, however, also require the same dependency.

The structural relationship of this version of the strategy pattern is displayed in figure 4.15. Notice that view-models are, in some cases, treated equally as strategy objects.

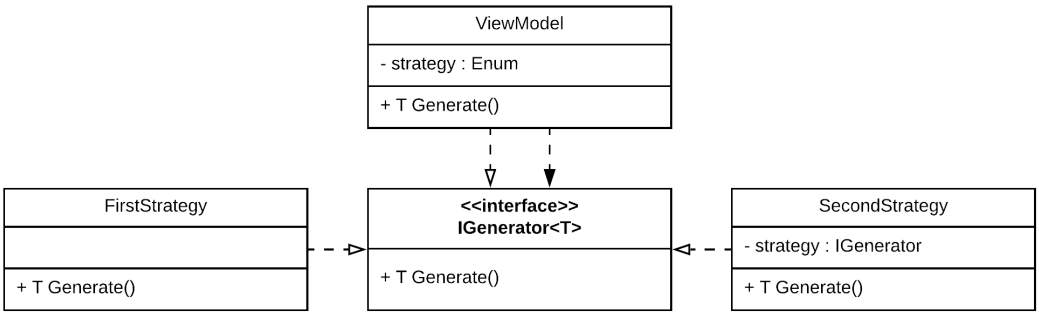


Figure 4.15: Structural relationship in this version of the Strategy pattern. Notice how the top view-model acts as the Strategy object by implementing the *IGenerator* interface and using an *IGenerator* object. The top-view model also holds a reference to the selected strategy enumeration.

4.3.6 Factory Method Pattern

The Factory Method pattern was utilized in BIAS's architecture to ensure a lower coupling [43, pp. 107-117]. Factory classes are utilized as the only access points for creating instances of various procedural generation strategies, as described in section 4.3.5. Here, concrete instances of strategies are created but returned as their abstract supertype, *IGenerator*. The road network factory, for example, creates instances of *LSystemStrategy* and *AStarStrategy* but returns them as *IGenerator<RoadNetwork>* objects. As a result, each factory class ensures that the behavior of strategy instances are reachable by different assemblies via polymorphism. The concrete instances are only accessible within their respective assemblies. As discussed in section 4.3.2, this lowers the coupling in BIAS's system architecture. The structural relationship of factory objects is displayed in figure 4.16.

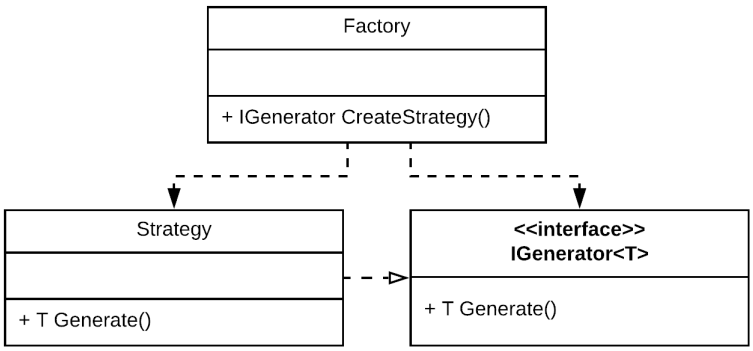


Figure 4.16: Structural relationship of the Factory pattern. The factory object is directly dependent on a concrete implementation of the *IGenerator* interface, since it creates the instance. However, the concrete instance is only exposed by the factory via the *IGenerator* interface.

4.3.7 Dependency Injection

Since most of BIAS's *IGenerator* objects create new object instances for each generation call, it may be difficult to ensure that object references are valid. *WhittakerStrategy* and *AStarStrategy* are, for instance, dependent on referencing a valid height-map of the terrain. To avoid similar unwanted direct dependencies to invalid instances, a version of dependency injection [53], [54] was utilized.

Dependency injection is utilized with the generic *IInjector* interface. An injector object implements the interface and holds valid references to generated objects. As a result, other objects can depend on an injector, providing valid references, instead of having to ensure the validity of the references themselves. Hence, the mentioned *WhittakerStrategy* and *AStarStrategy*, for instance, only require a dependency to an *IInjector<float[,]>*. This further ensures a less coupled architecture, since generators can simply depend on injector objects rather than on other generators. The general relationship of this version of dependency injection is displayed in figure 4.17.

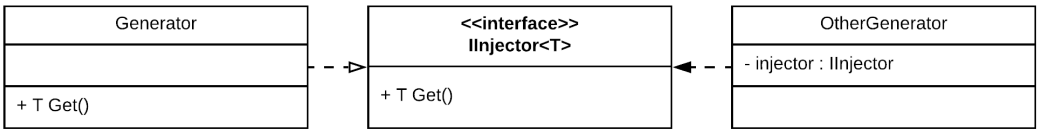


Figure 4.17: The general structural relationship of dependency injection used in BIAS. Notice how one generator implements the *IInjector* interface, with the other generator references an instance of the interface.

5

Discussion

This section discusses various aspects of the project, from the results and implementation process to its validity, ethical questions and future work.

5.1 Results

As described in section 1.2, the purpose of this project was to develop making the creation of three-dimensional cities more effortless through the utilization of procedural content generation. At the end of the project, the main purpose was fulfilled by the final implementation of BIAS. BIAS provides a useful, interactive and time-efficient generation of terrain and a variety of city elements by utilizing a number of procedural generation methods. The methods are subsequently open for users to explore and the resulting city scene can be extended with additional game objects. As a result, most of the goals described in section 1.3 were fulfilled as well. However, two goals were not completely fulfilled. Firstly, ensuring that the result of BIAS's generation is useful in the development of real 3D games requires the tool to validated by actual game developers. Since the results have only been validated by BIAS's creators, this goal may not have been fulfilled to its entirety. This topic is further discussed in section 5.3. Secondly, allowing users to place the location of where cities shall be generated was supposed to be performed with the implementation of a selection tool for marking areas where terrain and cities are to be generated. This functionality was unfortunately cut from the final implementation due to a lack of time. However, generation aspects, such as the area of terrain and the centre of road networks, can still be altered from BIAS's editor. As a result, the location of cities and a variety of rectangular terrain can still be generated, but in a less interactive manner that was initially intended. In one sense, the two goals are still fulfilled, but not to their fullest extent.

To fulfil the purpose and goals even further, several other ideas were also discussed

but ultimately not included in the final implementation. One idea was to allow users to create unique terrain shapes by adding vertexes directly in the Unity scene, while another was to generate different types of buildings, such as family homes, apartment complexes and schools. Such ideas were not included in BIAS due to a lack of time. This topic is further discussed in section 5.5.

5.2 Process

Due to the emergence of a global pandemic, the group worked together for the first half of the project on campus, with the second half taking place from individual homes. This made the development of the project different than what some were used to, having meetings via the Internet and sharing screens with other members. However, as the project did not require any special equipment, working from home was not that difficult. On the other hand, helping each other was harder to do over the Internet than when meeting in person.

As mentioned in section 3.1.1, it was initially decided to follow the Scrum methodology to make the implementation process as agile as possible. This remained the case for several weeks at the beginning of the implementation of BIAS, but the implementation process changed after a sprint lacked much progress. The sprint was extended and, after that, they were not as distinct as before. A Scrum board was still used to clarify what tasks were available or finished, and two meetings were held each week to update all group members on the current progress. The implementation process at the end of the project may not strictly be called Scrum. However, the process was still agile and incremental, and making progress with BIAS generally worked well.

Having six people working on a project has its positives and negatives. Making sure everyone stayed occupied with something productive was a challenge and weekly meetings were necessary to accommodate this. On the other hand, having several people to discuss ideas with helped to solve problems, and having people from different programs broadened the group's competences. The group consisted of computer and software engineering students from the Chalmers University of Technology, as well as a computer science student from Gothenburg University. One thing that all group members had in common, was that no one had any previous experience in using Unity or making PCG algorithms.

In the beginning, getting the project underway was a bit slow. This was partly due to the learning curve of Unity, but also due to the initial clarity of the project's goal. As a result, several team members worked on the same tasks during the first

week of the project. This was, however, probably beneficial to the project, since Unity at that point was unknown territory. Different approaches could then be tested, where the best approaches were subsequently chosen. After a slow start, things started to accelerate, with members finding their place in contributing to the development of the project.

The group never had any internal conflicts, but there were many discussions throughout the project. Everyone's ideas were considered before making decisions. An example of this was at the beginning of the project when all group members voted on making a tool that either could generate cities in three dimensions or two dimensions, isometrically. Three-dimensional cities thereafter won the vote.

5.3 Validity & Generalisation

The correctness of the final implementation of BIAS is in many ways subjective. Even though the asset was meant as a tool for game developers, the perspective of game developers was never taken into account. BIAS's correctness was solely validated in the eyes of its creators, which may result in a very biased opinion.

Many factors take place in the creation of cities in the real world. This may be factors such as various world regions, religions, cultures, economies and architectures. In the real world, cities are generally built through many generations of people continuously moving around areas such or churches, docks and industries. They are generally not created by placing a road network and populating the network with surrounding buildings. As a result, the tool's generated cities are not valid in terms of scientific or historical accuracy.

However, the goal of creating BIAS was to generate believability rather than realism. Generated cities are not realistically accurate, but they have an approximately realistic appearance. Many methods are also based on approximations of reality, which present non-accurate, yet still reasonable results. Perlin noise, for instance, is only based on visual approximations of terrain, yet still resulting in plausible appearance of terrains such as canyons and farmlands. Similarly, L-system road networks are based on approximations of random expansion around a centre point, yet still resulting in plausible road networks. In many contexts of 3D game development, credibility is also more than enough. Shadows, for instance, are often approximated with shadow maps and fire can be created through approximations of fluid dynamics.

Many of the procedural generation methods applied in the final implementation are only implemented for the sole purpose of creating various aspects of cities.

However, with minor refactoring, several methods can also be applied to other areas. Texture generation, for instance, can be utilized for creating the appearance of more than terrain, say building facades. Noise generation can be utilized for other things than terrain. It is usable whenever asymmetry is needed, such as in BIAS's generation of Whittaker textures. Most procedural generation algorithms do not have one sole purpose. What they are used for is up to the developer. The same applies for BIAS. Though many of the algorithms are built for the generation of cities, what they are utilized for is up the user.

5.4 Ethics

At the start of this project, it was discussed that procedural generation and this tool may harm the gaming industry since people might lose their jobs (see section 1.5). The extension can be used by companies to generate terrain and cities faster than designers and programmers could do. However, as the extension has parameters requiring input from a user, designers are most likely still needed. Even if the tool could affect peoples' jobs, it may be useful for people trying to establish a startup in, for instance, the gaming industry. As the main goal of startups is to establish a product on the market fast with minimum costs, this tool can be used to reach their goal without overspending.

As mentioned in section 1.5, cities are generated to be credible rather than realistic, as intended. The idea of adding transportation systems, such as buses and trains, to BIAS was discarded due to a lack of time. As was the idea of including religious buildings and service buildings, such as hospitals. Only generic buildings, consisting of differently sized blocks, are currently generated by the tool, resulting in credible cities at first glance. As cities are more than generic blocks, generated cities will therefore not be viewed as real. However, generic building blocks may also bring more users to the tool, since it allows users to decide if they want to keep the city generic or to add, for instance, religious building by themselves.

5.5 Future Work

Several aspects were not included in the final implementation of BIAS. This section covers how the asset may be improved in future implementations, both generally, in terms of performance as well as a few feature ideas. It also presents how the asset may be utilized in the future.

5.5.1 Generation Improvements

As a later part of the development process, building generation did not get as much attention as other generation steps. A priority for improvements in the tool is to extend the more basic forms of generation, such as buildings, to generate more defined shapes and add more customization. This would result in a more attractive tool for developers, which is ultimately the target of BIAS. There are also opportunities for improvement in some of the other generation steps, whether it be in the depth of customization or ease of use. Feature ideas are presented in more detail in section 5.5.3.

5.5.2 Improving Performance

As stated in section 3.2.8, multi-threading was introduced to BIAS at a late stage of its development, resulting in a more responsive user interface during the procedural generation. View models allowing several generation inputs additionally allows the generation to be performed in parallel. However, what may be improved in future implementations are the procedural generation algorithms themselves, since none of the final implementations utilize parallelism internally. For instance, the performance of the plot generation algorithm *AdjacentStrategy*, described in section 3.2.4 and 4.2.4, is a large problem, since it is currently a bottleneck of the generation algorithms in BIAS. There are also possible improvements in the currently used building strategy described in section 3.2.5 and 4.2.5. Introducing parallelism in such algorithms would probably shorten the waiting time of the city generation, resulting in a more responsive and usable tool for its end-users.

Another aspect to consider in future implementations is to refactor the current logic of task cancellation, if possible. As an example, there are currently many if-statements in various loops that respectively check, in each iteration, if the task in question has been requested to be cancelled. As a result, the code is a bit cluttered at places and it may be beneficial for its maintainability and comprehensibility if a more robust solution were found in the future. As parallelism was not an initial priority, a simple solution to this issue may have been missed.

5.5.3 Feature Ideas

A variety of feature ideas were discussed along the development of BIAS to improve both the usability of the asset and the realism of the generated content. However, as a result of the scope and time-constraints of this project, as well as the team-members abilities to develop applications such as BIAS, all ideas were ultimately not included in the final implementation. Nevertheless, a few feature ideas are still

very relevant to include in future iterations of BIAS.

As mentioned in section 5.1, a key-contribution of not fulfilling all goals of this project was BIAS's inability of providing interactive procedural generation. The generation is solely customized from the custom editor described in section 4.1, which results in predominantly static content. However, the initial idea was to implement a selection tool that would allow users to interact with the procedural generation as well as the created content directly in the scene. Users would then be able to, for instance, reshape specific terrain areas and mark locations where cities should be generated. An approach for this idea was initially tested but ultimately scrapped due to the prioritization of the features in the current implementation. In future iterations, however, a selection tool could be of high priority, since it would result in BIAS being more interactive.

Another feature discussed to improve BIAS's usability was to integrate an event handling system in the asset's architecture. In the procedural generation of the final implementation, it is not possible to revert to previous iterations of generation. This creates a high risk of users making accidental changes without the possibility of reverting them. However, the addition of an event handling system would ensure that users feel safe using the BIAS asset.

As of now, generated cities are also not that realistic. Terrain, as well as city elements, are quite bland and lack variety. To make BIAS's cities more interesting, additional elements are required. Road networks could, for instance, consist of different types of materials and various road widths. Two examples are highways and dirt roads. Roads made of asphalt could be more common closer to city centres, while dirt roads could be found outside of cities. Highways could also be more common between large cities, while not as common between small cities.

As a complementing feature for the procedural generation of realistic cities, heat maps, described in section 2.4.2, can be used for determining the population density of a city. This may subsequently be used for locating major components of a city. Areas with, for instance, more roads and taller buildings would then be procedurally generated where the population density is higher. Similarly, areas with a lower population density would result in the opposite. The implementation of a heat map was well underway at the end of this project. This was, however, never included in the final implementation, since there was a lack of time.

Another element that would improve the realism of generated cities is different types of buildings. Even minor improvements, such as procedurally generating building facades, would make the cities more believable. Buildings are currently

shaped based on the plot that they are built on. An enhancement would be to instead shape buildings according to building type blueprints and try to make them fit on their respective plots. Predefined rules for different building types would make it possible to procedurally generate buildings of similar types, but with slight differences. Cities would thus look more realistic since different neighbourhoods often consist of similar houses in reality. Tall office buildings, such as skyscrapers, could be more common in city centres, again depending on the population density.

It is also possible to improve the current form of building generation with transforms and shape grammars, as mentioned in 4.2.5. The plot-specific footprint of a building may be transformed according to shape grammar rules, such as L-systems described in section 2.3.2, that follow a general building pattern. This footprint would then become the basis of further transforms up to a height determined by population levels, which the choice of building shape rule would also be influenced by. This type of generation might be better suited to producing uniquely-shaped buildings by using the generated plot and could thus be combined with the previous approach in a larger strategy for even more realism.

Other than city elements, terrains would benefit from being more realistic. Terrains generated by the final product of this project only consist of land. Whittaker textures do make them appear somewhat alive, but there are many things that they lack. Generated worlds would benefit from having vegetation, such as forests, as well as water bodies, such as lakes and swamps. Adding such elements to the generated environments would make them more believable and interesting.

Adding more editor features and procedural generation details to the BIAS asset would probably make it more usable. Game developers presumably do not strive for bland game environments and it is important to have an intuitive interaction design. With this said, there are a lot of details that could be added, and improvements that could be made in the future. A future continuation of this project could include a lot of additional features, some mentioned earlier in this section. Since the project is quite open-ended, there is no limit for possibilities, but time.

5.5.4 Publishing the Tool

For developers to be able to use utilize the capabilities of BIAS, the asset has to be accessible from somewhere. Therefore, it could be released to, for instance, the Unity Asset store. Since the store is accessible directly from within the Unity editor, releasing BIAS here might be the most convenient solution for end-users.

6

Conclusion

The purpose of this project was to explore how procedural content generation could be utilized for creating three-dimensional cities to simplify game development. To achieve this, a study of procedural content generation techniques was performed and subsequently implemented as an asset in the game development platform that is Unity. The asset, called BIAS, provides the user with a simple interface for generating terrain and multiple cities on that terrain. The implementation uses several design patterns to divide generation into a set of steps where each step feeds information to the next.

Although the asset is to a degree able to generate a three-dimensional environment at a faster rate than a human, the generated content is in many aspects lacking what a game developer might want. For instance, the cities have an appearance of being generated, especially to someone familiar with procedural content generation methods. This does, however, open up the potential of future work where the focus may lie in adapting the asset to create more captivating game environments. Additionally, the project offers insight into the usage of different object-oriented design patterns as well as the usage of PCG in the context of cities. In conclusion, there is doubt as to whether the asset might be used by professional game developers. Nonetheless, the process through which the project underwent could aid others in avoiding the pitfalls involved with Unity and procedural content generation.

Bibliography

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation", *Applications of Evolutionary Computation*, pp. 141–150, 2010.

[3] Steam search, [Online]. Available: `https://store.steampowered.com/search/?sort_by=Released_DESC&tags=5125&category1=998` (visited on 05/04/2020).

[4] Steam search, [Online]. Available: `https://store.steampowered.com/search/?sort_by=Released_DESC&category1=998` (visited on 05/04/2020).

[5] J. A. Brown, "Pitching diablo", *ICGA Journal*, Mar. 2019. DOI: 10.3233/ICG-180066. [Online]. Available: `https://content-iospress-com.proxy.lib.chalmers.se/articles/icga-journal/icg180066` (visited on 02/12/2020).

[6] C. Salge, M. C. Green, R. Canaan, and J. Togelius, "Generative design in minecraft (gdmc): Settlement generation competition", in *Proceedings of the 13th International Conference on the Foundations of Digital Games*, ser. FDG '18, Malmö, Sweden: Association for Computing Machinery, 2018, ISBN: 9781450365710. DOI: 10.1145/3235765.3235814. [Online]. Available: `https://doi.org/10.1145/3235765.3235814`.

[7] K. Perlin, "An image synthesizer", *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325247. [Online]. Available: `https://doi.org/10.1145/325165.325247`.

[8] ——, "Improving noise", *ACM Trans. Graph.*, vol. 21, no. 3, pp. 681–682, Jul. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566636. [Online]. Available: `https://doi.org/10.1145/566654.566636`.

[9] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation", *Institute of Technology Blanchardstown Journal*, vol. 14, Jan. 2006.

[10] S. Gustavson, *Simplex noise demystified*, 2002. [Online]. Available: http://staffwww.itn.liu.se/~stegu/simplexnoise/.

[11] Earth Science. (Jan. 8, 2018). Understanding the classification of the biomes of the world, [Online]. Available: https://www.geographyandyou.com/understanding-classification-biomes-world/ (visited on 03/16/2020).

[12] L. Mucina, "Biome: Evolution of a crucial ecological and biogeographical concept", *New Phytologist*, pp. 98–99, Nov. 2018. [Online]. Available: https://www.researchgate.net/publication/329227529_Biome_evolution_of_a_crucial_ecological_and_biogeographical_concept (visited on 03/16/2020).

[13] Jwratner1 at Wikipedia. (Jun. 22, 2016). Whittaker diagram, [Online]. Available: https://commons.wikimedia.org/wiki/File:PrecipitationTempBiomes.jpg (visited on 03/16/2020).

[14] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents", *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, pp. 111–119, Jul. 2010. DOI: 10.1109/TCIAIG.2010.2049020.

[15] Paul K. Asabere, "The value of a neighborhood street reference to a cul-de-sac", *The Journal of Real Estate Finance and Economics*, Feb. 1990. DOI: 10.1007/BF00216591. [Online]. Available: https://www.researchgate.net/publication/5151757_The_Value_of_a_Neighborhood_Street_Reference_to_a_Cul-de-Sac (visited on 02/11/2020).

[16] CNES. (2003). San francisco, usa, [Online]. Available: https://quest-eb-com.proxy.lib.chalmers.se/search/city-grid/1/132_1558872/San-Francisco-USA/more (visited on 03/27/2020).

[17] D. R. Suburban housing, [Online]. Available: https://quest-eb-com.proxy.lib.chalmers.se/search/subdivisions-housing/1/139_2013564/Suburban-housing/more (visited on 03/27/2020).

[18] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Springer, 1990, pp. 1–50.

[19] Interactive Data Visualization, Inc. SpeedTree, [Online]. Available: https://store.speedtree.com/ (visited on 02/04/2020).

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

[21] S. J. Russell and P. Norvig, "Artificial intelligence a modern approach", pp. 92–100, 1995. [Online]. Available: https://www.cin.ufpe.br/~tfl2/artificial-intelligence-modern-approach.9780131038059.25368.pdf (visited on 03/21/2020).

[22] Marina Mihaila, "City architecture as cultural ingredient", *Procedia - Social and Behavioral Sciences*, Oct. 2014. [Online]. Available: https://doi.org/10.1016/j.sbspro.2014.08.211 (visited on 02/05/2020).

[23] Efrat Eizenberg, "Large-scale urban developments and the future of cities: Possible checks and balances", *Urban Planning*, Nov. 2019. DOI: 10.17645/up.v4i4.2643. [Online]. Available: https://www.cogitatiopress.com/urbanplanning/article/view/2643/2643 (visited on 02/05/2020).

[24] Moumtzidou, V. Anastasia, C. Stefanos, K. Elisavet, and Ioannis, "Discovery of environmental resources based on heatmap recognition", 2013. DOI: 10.1109/ICIP.2013.6738305. [Online]. Available: https://ieeexplore.ieee.org/document/6738305 (visited on 02/13/2014).

[25] Shook, L. Eric, C. Kalev, P. Guofeng, W. Anand, and Shaowen, "Happy or not: Generating topic-based emotional heatmaps for culturomics using cybergis", 2012. DOI: 10.1109/eScience.2012.6404440. [Online]. Available: https://ieeexplore.ieee.org/document/6404440 (visited on 01/11/2013).

[26] Unity Technologies. Unity homepage, [Online]. Available: https://unity.com/ (visited on 02/11/2020).

[27] ——, This is why creators choose Unity, [Online]. Available: https://unity.com/our-company (visited on 03/18/2020).

[28] Unity. (Jan. 28, 2019). Extending the editor, [Online]. Available: https://docs.unity3d.com/Manual/ExtendingTheEditor.html (visited on 02/13/2020).

[29] U. Technologies. (Jan. 28, 2020). Editor, [Online]. Available: https://docs.unity3d.com/ScriptReference/Editor.html (visited on 02/14/2020).

[30] (2020). Unity asset store - the best assets for game making, [Online]. Available: https://assetstore.unity.com/ (visited on 05/13/2020).

[31] Unity Technologies. ProBuilder, [Online]. Available: https://unity3d.com/unity/features/worldbuilding/probuilder (visited on 02/13/2020).

[32] Hutong Games LLC. (Sep. 7, 2019). Playmaker, [Online]. Available: https://assetstore.unity.com/packages/tools/visual-scripting/playmaker-368 (visited on 02/13/2020).

[33] F. Holmér. Shader Forge, [Online]. Available: http://www.acegikmo.com/shaderforge/ (visited on 02/13/2020).

[34] Unity Technologies. ProBuilder, [Online]. Available: https://www.procore3d.com/probuilder/ (visited on 02/13/2020).

[35] F. Holmér. (Jul. 23, 2011). Shader Forge - A visual, node-based shader editor, [Online]. Available: https://forum.unity.com/threads/shader-forge-a-visual-node-based-shader-editor.222049/ (visited on 02/13/2020).

[36] ——, (Jan. 20, 2020). Shader Forge source code, [Online]. Available: https://github.com/FreyaHolmer/ShaderForge (visited on 02/13/2020).

[37] A. Powell-Morse. (Dec. 8, 2016). A description of the waterfall model, [Online]. Available: https://airbrake.io/blog/sdlc/waterfall-model (visited on 02/11/2020).

[38] K. Schwaber and J. Sutherland, *The Scrum Guide*. Nov. 2017.

[39] The Blender Foundation. Blender homepage, [Online]. Available: https://www.blender.org/ (visited on 02/11/2020).

[40] Z. M. Jiang, A. E. Hassan, and R. C. Holt, "Visualizing clone cohesion and coupling", in *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, 2006.

[41] D. J. Skrien, *Object-oriented design using Java*. McGraw-Hill, 2009.

[42] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education Limited, 2014.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[44] J. Kouraklis, *MVVM in Delphi*. Jan. 2016, pp. 1–11. DOI: 10.1007/978-1-4842-2214-0.

[45] S. Gottschalk, "Separating axis theorem", Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, Tech. Rep., 1996.

[46] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe", *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857078. [Online]. Available: https://doi.org/10.1145/857076.857078.

[47] (Oct. 2, 2018). Implementing event-based communication between microservices (integration events), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications (visited on 05/06/2020).

[48] (May 7, 2020). Assembly definitions, [Online]. Available: https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html (visited on 05/09/2020).

[49] (Sep. 12, 2018). Parallel programming in .net, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/ (visited on 05/07/2020).

[50] (Mar. 30, 2017). Task parallel library (tpl), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (visited on 05/07/2020).

[51] R. Mitchell, *Managing Complexity in Software Engineering*. Peter Peregrinus Ltd., 1990, ISBN: 0863411711.

[52] S. Jiang and H. Mu, "Design patterns in object oriented analysis and design", in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011, pp. 326–329.

[53] D. R. Prasanna, *Dependency injection*. Manning Publications, 2009.

[54] E. Razina and D. S. Janzen, "Effects of dependency injection on maintainability", in *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 2007, p. 7.

[55] Esri. Esri CityEngine, [Online]. Available: https://www.esri.com/en-us/arcgis/products/esri-cityengine/overview (visited on 02/04/2020).

[56] S. Persson. (May 17, 2019). Celebrating 10 years of minecraft, [Online]. Available: https://news.xbox.com/en-us/2019/05/17/minecraft-ten-years/ (visited on 02/04/2020).

[57] R. P. Wiegand and J. Sarma, "Spatial embedding and loss of gradient in cooperative coevolutionary algorithms", in *Parallel Problem Solving from Nature - PPSN VIII*, X. Yao, E. K. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. E. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 912–921. DOI: 10.1007/978-3-540-30217-9_92.

[58] (Mar. 17, 2020). Mesh, [Online]. Available: https://docs.unity3d.com/ScriptReference/Mesh.html (visited on 03/18/2020).

Appendices

A

Software and Games Utilizing Procedural Content Generation

Procedural content generation is utilized in both games and various software for a variety of reasons. This chapter presents a few examples of where the concepts are used.

A.1 Examples of Software Utilizing PCG

Two examples of software, utilizing PCG, are SpeedTree [19], a tool for generating virtual foliage that is used by architects and animators and Esri CityEngine [55], modeling software for creating immersive urban environments based on geographical data.

A.2 Examples of Games Utilizing PCG

A few examples of games where the utilization of procedural content generation is apparent is Minecraft and Diablo. Diablo is a fantasy game about a group of warriors on a mission to hunt down the demon lord Diablo, who is threatening the village of Tristram. To reach Diablo in Hell, the player must go through a series of quests that take them on a journey through the difficult dungeon. Diablo was likely a success because of its early on uses of PCG algorithms that randomly created dungeons for each new level. The developers used what they called a "Dynamic Random Level Generator" to generate a new dungeon for each level [5]. The generator provided a new gaming experience for the player every time the game was played.

Minecraft is a popular game with a three-dimensional world consisting of cubes

of earth, wood and other materials. The game is multi-functional, meaning that players can decide if they want to build structures and obtain objects from the world. There are environmental hazards that can kill players' avatars, but it is possible to avoid these in the game. *Minecraft* is based on algorithms like Perlin Noise to generate unique terrains with large voxels (see figure A.1). The game also uses Whittaker diagrams to determine what plants and other terrain features that should be added in different biomes [6]. As of May 2019, 10 years since its official release, *Minecraft* has sold over 176 million copies [56], proving its huge success.

(a) Minecraft world

(b) Amplified world in Minecraft

Figure A.1: Screenshots of Minecraft.

B

Search-Based Approach to PCG

The search-based approach to procedural content generation was briefly summarized in section 2.1. This chapter extensively describes all three parts of the approach; content representations, evaluation functions and search algorithms.

B.1 Content Representations

Content representations can be interpreted as blueprints, defining how different types of content are to be generated [1, p. 18]. A game level, for instance, probably has to contain certain elements to fit into the game that is being developed. The content representation can, therefore, contain data about what is required and can then be used to generate actual levels.

Togelius and Shaker describe content representations in the terms of "genotypes" and "phenotypes" in the context of evolutionary search algorithms. The former is what is used to generate content and the latter is the actual generated content, derived from the genotype [1, pp. 18, 20]. Referencing the earlier example about game levels; the genotype would be the data about what is required in the level and the phenotype would be the generated level itself.

B.2 Evaluation Functions

Evaluation functions determine how well-generated content fits into the context. An example of this is a game level that has been generated and then evaluated to verify that it is possible to beat and that it is neither too easy nor hard to play. There are at least three types of evaluation functions: direct, simulation-based and interactive [1, pp. 18-24].

B.2.1 Direct Evaluation Functions

Evaluation functions that directly calculate fitness values based on the generated content's attributes are called "direct". Such functions can either be driven by theory or by data. The former means that the evaluation is based on theories about how players will experience the content. The latter means that the evaluation is based on collected data and statistics about player experience [1, p. 23].

B.2.2 Simulation-Based Functions

As the name suggests, simulation-based evaluation functions involve simulating the use of generated content to evaluate their respective fitness values. The simulation is done by utilizing AI agents that for example play a generated game level to make estimations. Agents can either be dynamic or static, meaning that they either adapt during the simulation or not [1, pp. 23-24].

B.2.3 Interactive Functions

Evaluation functions can also be based on human interaction, meaning that human actions can determine the fitness of generated content. An example is giving different fitness values to generated weapons in games, depending on how much they are used by actual human players [1, p. 24].

B.3 Search Algorithms

Moving on to search algorithms, there are different types, such as: Evolutionary, exhaustive, random and solver-based. Togelius and Shaker describe an evolutionary search algorithm where the number of generation surviving "individuals" (content) is represented by μ and where the number of "offspring", derived from the previous generation, is represented by λ. The sum of these variables is the total size of the content population when generating. The algorithm consists of eight steps, where only the second one is optional [1, p. 18-20]:

1. A population with a size of $\mu + \lambda$ has to be initialized to generate anything. Since this is the initialization step and is only executed once at the very start of the algorithm, this can be done in any way that is deemed appropriate in the context. The initial population can, for instance, consist of individuals that are man-made, from earlier uses of the algorithm or generated at random.

2. Optionally shuffle the population to avoid loss-of-gradient situations [1], meaning that some subpopulations otherwise could start dominating others

[57].

3. The fitness for each content is calculated to a numeric value by using it as a parameter in an evaluation function.

4. The whole population is sorted by the fitness in ascending order.

5. Since μ is the number of surviving individuals after a generation, a total of λ individuals has to be discarded. The ones getting "killed" have the worst fitness scores of the population and are therefore not fit to generate any offspring for the next generation.

6. The discarded individuals are replaced by copies of the ones with higher fitness scores. Depending on the proportions between μ and λ, some elite individuals might not get copied at all or might be copied several times to reach the number of λ copies, i.e., the population size does not change.

7. Several λ individuals are randomly chosen to be mutated (modified) in arbitrary ways. An example of a mutation type is Gaussian mutation.

8. Depending on if an individual is deemed good enough, based on its fitness score, the algorithm is stopped. It is also stopped if the number of executed iterations is equal to a specified maximum. The algorithm otherwise repeats everything from the second step and thus starts a new generation of individuals.

C

Creating Terrain Meshes In Unity

The utilized terrain meshes are mainly based on noise maps consisting of two-dimensional float matrices. A mesh generator thus takes a noise map as input and maps each matrix element to a vertex in a 3D space. The indices of the matrix are mapped to x- and z-coordinates and the height of the vertex point (y-coordinate) is based on the value at (x,z) in the noise map. In figure C.1, such vertices are represented by red dots. Each created vertex is put in a one-dimensional array, explaining each dot's number.

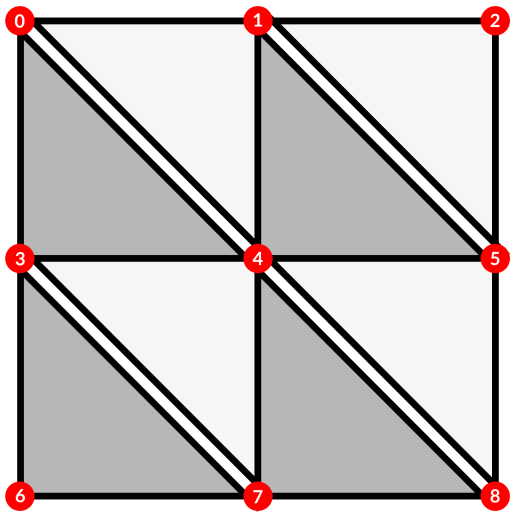


Figure C.1: Each red dot represents a vertex in the mesh and each black line represents a side in a triangle. The diagonal triangle edges overlap in the actual implementation, but are separated here for clarity.

In Unity, a mesh is partly based on collections of vertices and triangles [58]. As displayed in figure C.1, two triangles are added per vertex, excluding the vertices

to the far right and at the very bottom (vertices 2, 5, 6, 7 and 8 in the figure), since their edges otherwise would not have any closing vertices. Since each triangle consists of three points, this results in six points per vertex. The mesh represented by figure C.1 therefore consists of 9 vertices, 8 triangles and $8 \cdot 3 = 24$ triangle points.