School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Designing 3D City Layouts Using Procedural Content Generation Techniques

Brayden Matthew Jalleh

Submitted in accordance with the requirements for the degree of BSc Computer Science

2023/24

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (29/04/24)
Link to online code repository	URL in Appendix C	Sent to supervisor and assessor (29/04/24)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Brayden Matthew Jalleh

Summary

Procedural content generation (PCG) utilises algorithms to automatically create diverse forms of digital content such as landscapes, levels, and objects, thereby eliminating the need for manual detail construction by programmers. Predominantly employed in simulations, virtual reality, and video games, PCG facilitates the efficient production of extensive content volumes. In this project, PCG is specifically applied to generate a 3D simulated city, ensuring that each instantiation of the city is unique and distinct. The primary objectives of this project were to demonstrate the systematic application of procedural techniques to simulate varied urban landscapes and to develop and refine algorithms that significantly impact city layouts and architectural diversity. Through this initiative, the project explored the adaptability, significance, and suitability of procedural algorithms for creating sophisticated and diverse urban environments, particularly emphasising their pivotal role in game development and urban simulation. This report outlines the methods used to automatically generate urban features within simulated cities, focusing on the procedural techniques employed. It discusses the practical applications of these techniques, providing an overview of their potential impact on creating realistic and dynamic urban simulations. The report highlights key observations about the effectiveness of these methods in enhancing the visual and operational aspects of the simulated environments. Furthermore, the report examines the scalability and performance efficiency of these procedural methods, shedding light on their feasibility and limitations within current technological frameworks. By exploring these areas, the report contributes to the broader discussion about integrating procedural content generation (PCG) into digital content creation.

Acknowledgements

I would like to extend my sincere appreciation to Dr. Mark Walkley, my project supervisor, for his invaluable advice and assistance throughout this project. Dr. Walkley's expert knowledge and insightful contributions have significantly influenced the direction and success of my work. Our regular project meetings throughout the academic year have been instrumental in completing this project. I am immensely grateful for his encouragement and the opportunities he has provided to help me improve.

I would also like to thank my assessor, Dr. YongXing Wang, for the valuable feedback he has given during the progress meeting. His insights have been instrumental in refining my project's direction and enhancing the quality of my work, and I am grateful for his guidance and support throughout this process.

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Procedural Content Generation (PCG) is an advanced technique that utilises algorithms to automatically generate digital content, such as landscapes levels, and objects, significantly reducing the need for manual input from programmers and designers. This efficiency is increasingly vital in the development of games and simulation of environments, as the demand for the creation of expansive and intricate virtual environments increases.

By making use of PCG, developers may avoid the steep costs and extensive labour involved in creating detailed environments manually. Furthermore, generating content dynamically not only speeds up the content production but also enhances user experience by providing them with diverse and unique environments. These enhancements are particularly crucial for projects aimed at delivering expansive and immersive experiences to users.

In addition, PCG's optimises computer resources and encourages design innovation in response to users inputs and environmental changes. It makes it much easier to create material on-the-fly, which reduces load times and saves memory. As a result, PCG is changing the way digital creation is done, making it much more efficient and flexible in response to the increasing demands in the industry.

1.2 Aims and Objectives

This project aims to investigate and apply foundational principles of procedural content generation techniques to the creation of 3D simulated cities, each uniquely generated to reflect varied urban environments.

The main objective is to demonstrate the potential of procedural techniques to effectively simulate urban landscapes. This involves the development and refinement of algorithms that affect the cities' architectural variety as well as their layout, which is essential for producing realistic and functionally diverse metropolitan areas. Additionally, the results of the implementations in creating diverse and dynamic 3D city models will be discussed, examining how these methods improve the simulated environment visual and practical elements.

1.3 Project Management

1.3.1 Planning and Methodology

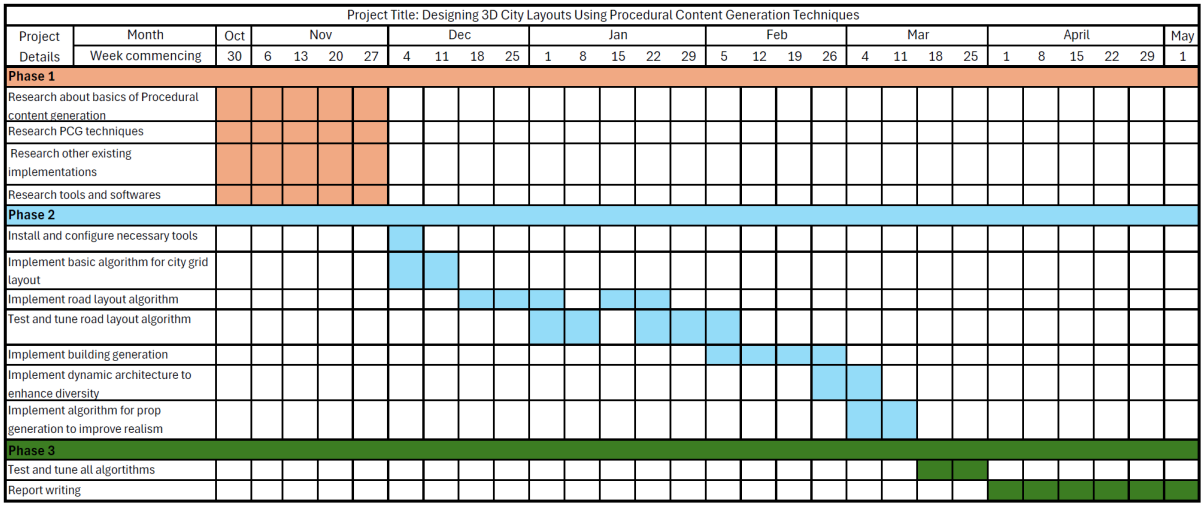


Figure 1.1: Gantt Chart used for project planning

The management of the project was done by implementing a Gantt chart in order to effectively visualize the project timeline and sequence the tasks strategically as shown in Figure 1.1. The project was separated into three phases, each with targeted goals and deadlines plotted on the Gantt chart to track progress. In Phase 1, extensive study was conducted to properly grasp all aspects of procedural content generation (PCG). This extensive study included not just the fundamental principles and theories behind PCG, but also a thorough examination of numerous existing implementations. The study developed a strong theoretical foundation by meticulously examining these methodologies and their implementations in a variety of scenarios. This foundation guaranteed that following phases of implementation were influenced by a thorough grasp of PCG, allowing for more informed decision-making and creative approaches in the project's later stages.

Phase 2 was the actual implementation phase, an agile methodology was used to divide the phase into a series of sprints, each with their own objectives and deliverables. Every sprint was built upon the accomplishments of the one before it, guaranteeing a coherent and incremental advancement of the project's technological and visual goals. Then initial sprint was dedicated to laying the foundation: setting up and configuring the required software tools and starting the creation of the city grid layout along with a basic road generation algorithm. This was followed by testing and optimisation ensuring that the basic infrastructure was reliable. The second sprint was used to enhance the road generation algorithm. Building upon the foundation in the first sprint, improvements were focused on making the algorithm more complex and effective. The third sprint covered building generation, dynamic architecture integration and prop generation methods. This sprint mainly dealt with the aesthetic aspects that would define the uniqueness of the city layouts. To ensure that the finished result worked well and matched the project's goal of realistic and diverse metropolitan landscapes, each of the components underwent testing and fine tuning in the final phase 3.

1.3.2 Risk Mitigation and Version Control

Key risks identified for the project include technical difficulties, integration challenges, and potential delays. Therefore, necessary strategies were employed with the structured project management approach previously outlined in section 1.3.1 to ensure that the development proceeds smoothly and that sufficient work was completed for the final submission. A flexible approach to project scope was critical, allowing for adjustments in methodology and objectives as needed without compromising the core goals of the project. This flexibility ensured that if a specific approach failed to yield the desired results or if a task proved to be unfeasible within the given constraints, alternative strategies could be employed swiftly. Furthermore, to mitigate the risk of work or data loss, GitHub repositories was used as a version control system to store all versions of the files since their creation. This not only provides a secure backup but also allowed for tracking changes and reverting to previous states if and when necessary. This practice is essential in software development as it ensures that every edit and version is documented.

1.4 Background research

The essence of procedural content generation (PCG) lies in the strategic use of randomness, managed primarily by pseudo-random number generators (PRNGs). PRNGs are algorithms that utilise mathematical formulas to generate sequences of seemingly random integers. These sequences play a crucial role in PCG by enhancing the unpredictability of the generated content, ensuring that each iteration produces distinct results [3][22]. PCG encompasses a variety of algorithms, each specifically tailored to different types of content creation. Common algorithms include Perlin noise [15], the Diamond-Square algorithm [9], L-systems, and cellular automata, chosen based on their ability to achieve the desired outcomes such as realistic textures, complex terrains, or dynamic systems within the environment [3][17].

1.4.1 Offline vs. Online Procedural Content Generation

Procedural Content Generation can be categorised into two distinct processes: offline and online generation, both of which are tailored to different phases of the content generation process and with distinct operational requirements. Offline PCG is implemented during the development phase or prior to initiation of a media session, where the demands on real-time performance is minimal, allowing for more complex and resource intensive content such as expansive environments and detailed maps to be generated. Furthermore, the offline method allows for creation of more foundational elements or set pieces that can be later fine-tuned or expanded upon during later stages of development.[17]
In contrast, online PCG enables real-time generation of varied content as the user engages with the media, which is crucial for maintaining user engagement, extending the life of the media as well as providing a personalised experience to the user. This would require the algorithms to be highly optimised to ensure smooth operation and uninterrupted user experience.[17]
Both online and offline PCG have significantly shaped the fields in which they are applied, particularly by enhancing interactive media by providing varied and dynamic content. While predominantly recognises in gaming, the principles of PCG are also applicable in other creative

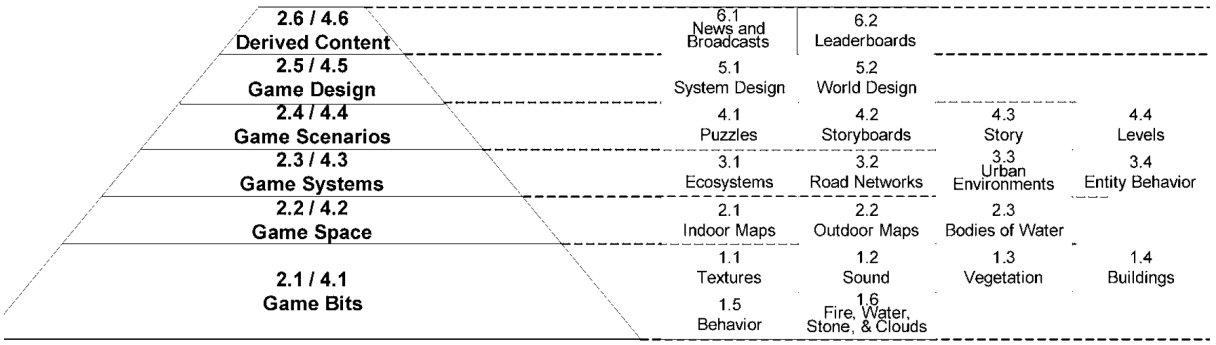


Figure 1.2: Hierarchical pyramid of game creation by Hendrikx et al.,2013 [11]

industries where dynamic and scalable content generation is beneficial. These technologies continue to evolve, pushing boundaries of what can be automatically generated and how such content can be integrated seamlessly into user experiences. [2]

1.4.2 Evolution and Impact of Procedural Content Generation

While procedural content generation finds applications in simulations and virtual reality, its most significant impact has undoubtedly been in the gaming industry. As we delve deeper, it becomes evident how integral PCG has been in shaping modern gaming landscapes, offering developers state-of-the-art methods to craft expansive, engaging game worlds that continuously adapt and respond to player interactions. Since its initial use in the early '80s through the groundbreaking dungeon crawler, Rogue, by Michael Toy and Glenn Wichman, PCG has significantly evolved [8]. Initially utilised to overcome the memory limitations of early computing hardware by generating diverse game levels on-the-fly, PCG quickly proved its broader value. It has significantly reduced the need for large design teams, as it allows for the automated creation of detailed game levels and elements, thus controlling production costs as project scales have expanded [8].

Expanding on this, procedural content generation's versatility is evident not only in environmental design but across all elements of gaming. While terrain and flora are common produced features, PCG has the capacity to create almost every facet of game design, this includes characters, artificial intelligence, stories and even music [3][12]. As procedural content generation has grown in the gaming industry, its applicability has spread across many levels of game development. This can be comprehensively understood through a hierarchical pyramid introduce by Hendrikx et al.,2013 that splits the application of PCG into levels, each representing a distinct layer of game creation as shown in Figure 1.2 [11] At the foundation lies 'Game Bits', encompassing the essential components such as textures, sound, and behavioural scripts. These fundamental elements are the atoms of game design, integral to the construction of a game's aesthetic and interactive qualities. Moving up the pyramid, 'Game Space' refers to the procedural creation of expansive and immersive environments. This layer includes the terrain and architectural elements that make up the physical world in which game narratives unfold. The next tier, 'Game Systems', encompasses the underlying mechanics that govern the operation of the game world. This includes simulated ecosystems, the behaviours of artificial intelligence entities, and the physical laws that regulate interaction within the environment.

Such systems ensure that the game responds and reacts in consistent, yet dynamic ways to the player's actions. At the top of the pyramid, 'Game Scenarios' shape the overarching stories and plot lines. This highest tier involves creating story arcs, quests, and events that propel the game's advancement and engage the user on a deeper level.[11]. Through this pyramid structure, we see how PCG is not a monolithic tool but rather a layered approach that imparts complexity and life into the virtual worlds it helps create. This structured application of PCG allows for a more directed and nuanced design process, enabling developers to breathe life into each layer of the game development process. Table 1.1 demonstrates how a number of well-known games have used PCG approaches at various levels [11].

Games (Year of release)	Game Bits	Game Space	Game Systems	Game Scenarios
Borderlands (2009)	x			
Diablo I (2000)		x		
Diablo II (2008)		x		
Elder Scrolls IV: Oblivion (2007)	x			
Elder Scrolls V: Skyrim (2011)				x
Gears of War 2 (2008)	x			
Left4Dead (2008)				x
Minecraft (2009)		x	x	
Rogue (1980)		x		

Table 1.1: Use of PCG techniques in games. Adapted from Hendrikx et al., 2013 [11]

1.4.3 Advantages and Disadvantages

Procedural content generation (PCG) is known to significantly reduce file sizes by saving disk space since the techniques utilised are significantly less than the large amounts of data they actually generate. The technique also makes it possible to create nearly and infinite amount of unique objects, which increases the diversity of content. This is especially useful where numerous similar yet distinct objects are required. Additionally, the flexibility of PCG permits quick iterative revisions to the generated content by simply changing parameters and rules [4][22]. Despite its numerous advantages, PCG can sometimes limit the level of detail and quality that can be achieved only through highly controlled manual processes. Such meticulous control is often crucial for creating immersive, cinematic moments that enhance narrative depth, but it may also limit creative freedom [23]. These limitations highlight the trade-offs involved in using PCG, where the need for dynamic content generation must be balanced against the desire for high-quality, engaging storytelling.

1.4.4 Existing Implementations of Procedural City Generation

Numerous projects have focused on the automatic generation of cities and have implemented it using a variety of methods. Here, we take a look at the other related implementations that demonstrate the methodologies and application for solving the complex problem of creating a realistic 3D city simulation using procedural generations methods:

Citygen [13]

Citygen is a project created in 2007 by G.Kelly, it offers interactive control over the generation process, allowing its users to manipulate geometric elements and PCG parameters. It is divided into three main stages:

1. Primary Road generation: This initial stage uses a graphical interface where users can create and adjust the primary road network by manipulating nodes. The system procedurally generates the roads connecting the nodes, adapting to the surrounding topography.[13]

2. Secondary Road generation: This stage automatically generates roads within enclosed areas. Users choose various patterns for the roads to suit different parts of the city in real time.[13]

3. Building generation: The system identifies the plot of land within the network of secondary roads. It divides the plots into smaller lots allowing buildings to be placed procedurally depending on the type of area the system is at.[13]

The key feature of G.Kelly's 'Citygen' would be the use of adaptive roads. The roads are not simply built on a flat surface but are intended to follow the natural forms of the landscape. This means that roads will curve around hills, adapt to valleys, and maintain realistic gradients contributing to a more authentic and functional city layout [13].

Procedural City Generator [18]

In this project by Sharma, the city generation process is implemented such that the design of the city is dependent on a user-provided grayscale population density map. The map is inputted as an image, directly influences the urban layout, where roads and building localities are aligned according to the grayscale values. White indicating high-density areas, black indicates low or no population. The generated city includes both primary highways and secondary roads, such as raster and rotary roads, with the road network's pattern directly influenced by the map. Road formation utilises the Catmull-Rom spline curve to ensure smooth transitions. Raster roads are further developed using an L-System to enhance structured road layout. Two techniques are used to identify city cycles, or the regions bounded by highways where estates and parkland are created: Minimum Cycle Basis and Matrix Ray Cast. A pseudo-random generator ensures uniqueness and randomness in the city's configuration. It processes a user-input seed string and translates it into a unique number that influences all procedural algorithms, from road lengths to branching angles to the selection of population nodes and matrix sizes. Each unique seed provided by a user results in a unique city layout, allowing for an unlimited variety in city design [18].

CityCraft [2]

CityCraft is a project released in 2020 that explores the usage of common PCG algorithms to devise suitable methods for the generation of modern 3D cities. This project offers valuable insights into algorithm selection and application, which are crucial to understanding the broader capabilities and challenges in procedural city generation. In this project, the techniques such as Gradient noise, L-system and Voronoi Diagrams were used. The city generation process is systematically divided into eight stages: Terrain, Population, Road, Blocks, Plots, Buildings, Parks and Parking. Each stage uses a specific method or algorithm optimised for its particular function. The algorithms differ, with some stages using fundamental PCG algorithms and other using unique methods created specifically for the project [2].

The terrain and population stages utilises simplex noise to generate a height map for terrain and a distribution map for population density respectively. The road generation uses a unique method that uses agents to traverse the generated terrain in order to construct roads according to the predefined building technique. These agents are directly influenced by the population density map generated during the Population stage, which employs gradient noise to dictate the density of streets and buildings within specific areas [2].

For the building structures, the project makes use of the L-system algorithm, which is responsible for generating the diverse architectural elements such as walls and floors. Meanwhile, the park stage makes use of a Poisson disc sampling technique that scatters points randomly across a plot to simulate naturalistic park layouts [2].

A Study on Creation and Usability of Real Time City Generator via Procedural Content Generation [14]

This project primarily aims to develop an efficient, cost-effective tool using procedural content generation (PCG) to create real-time, immersive city environments tailored to enhance the engagement and utility of virtual reality (VR) applications for seniors, thereby streamlining VR content production. The system leverages bespoke algorithms designed specifically for dynamic generation of urban environments as users navigate through the VR experience, such as during a simulated bus ride. These bespoke algorithms manage the creation of city elements: buildings, roads, and trees, using controlled randomisation to determine attributes like building height, width, and architectural features [14]. This dynamic generation process not only ensures variety but also tailors the environmental details to the user's perspective within the VR environment. For example, buildings viewed from the perspective of a bus ride display greater detail on the lower floors, while the upper floors are rendered with less detail. This strategic adjustment conserves processing resources and maintains optimal performance, which is essential for immersive VR applications. The PCG technique implemented utilises pseudo-random number generators to maintain both variability and consistency in the content generated. This contributes to user engagement by providing a realistic and ever-changing urban landscape. The algorithms are particularly efficient in how they prioritise detailing based on the user's eye level, focusing computational resources where they are most impactful [14]. By optimising the level of detail according to user perspective and interaction, the approach significantly reduces the time and costs associated with manually creating detailed 3D models

for each scenario. Overall, the technical execution of the city generator using the Unity engine and custom PCG algorithms offers a scalable, reusable approach to VR content creation. This adaptation to various user needs and simulation scenarios makes it an invaluable tool in the development of VR applications aimed at improving the quality of life for seniors. These innovations highlight a strategic departure from more traditional PCG methods that apply uniform detail across generated landscapes without considering the user's immediate visual field or interaction needs [14].

Chapter 2

Methods

2.1 Framework

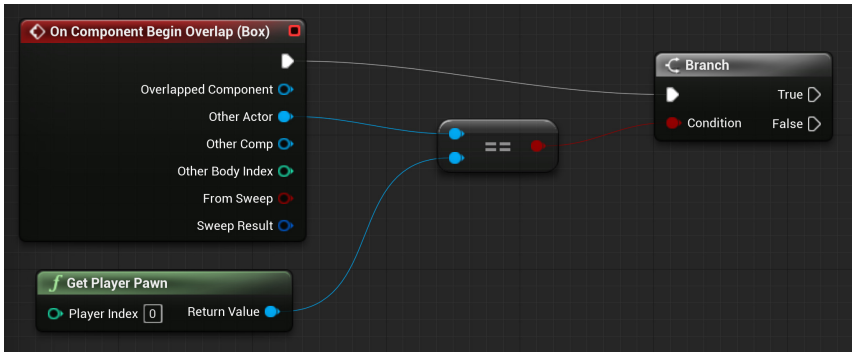


Figure 2.1: Example of an Unreal Engine 5 blueprint [6]

Given the vital role that 3D graphics plays in this project, choosing the right 3D framework or engine would be essential. The evaluation process involved a detailed comparison of potential solutions, including Godot [10], Unity 3D [20], Unreal Engine 5 [5], and the possibility of developing a bespoke engine. Each option was thoroughly assessed, considering a range of factors such as graphical capabilities, ease of use, community support, and suitability for integrating complex procedural content generation (PCG) logic.

Unreal Engine 5, known for its high-fidelity visuals and advanced rendering capabilities, was initially a strong contender. It is particularly favoured in projects that demand intricate visual details and robust PCG implementations. However, the decision to not use Unreal Engine was primarily due to its steep learning curve and heavy reliance on visual programming via "blueprints", as shown in Figure 2.1. While blueprints facilitate a more visual approach to programming and can accelerate development, it did not meet the specific scripting and customisation needs of this project. On the other hand, Godot offers a more accessible entry point with a gentler learning curve and supports multiple official programming languages such as C/C++, C#and its own GD Script. Nevertheless, Godot's limitations in handling more complex 3D content and its smaller developer community compared to its counterparts influenced the decision against its adoption for this project. The idea of developing a bespoke engine was deemed impractical for this project due to the extensive workload required, paired with personal limitations in experience and expertise. Ultimately, Unity 3D emerged as the preferred choice. This decision was influenced by several factors: Unity's extensive and active community support, the wealth of online tutorials and comprehensive documentation available, and the engine's overall stability and reliability. Unity's asset store offers an abundance of both first-party and third-party tools that enhance development flexibility, including user-friendly tools like 'Shader Graph' which simplify the creation and integration of procedural textures. The availability of the Unity Student plan, providing access to the latest software versions and features at no cost, further cemented Unity as the optimal choice. Additionally, Unity's

support for C#programming, paired with previous experience in C/C++, provided a strong foundation for efficiently managing the project's technical requirements.

2.2 City Generation Methodology

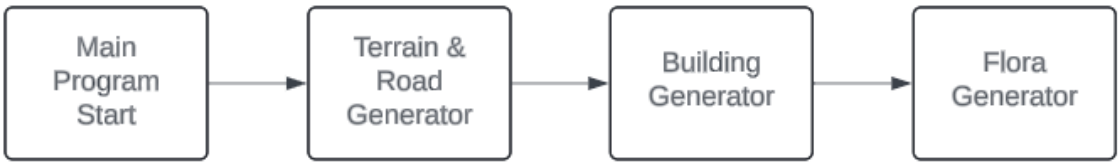


Figure 2.2: Sequential order of generation

For the project, we subdivided the work into separate stages: Terrain, Road, Building, and Flora, following a progressive implementation approach. This sequential order, from terrain to road, building and lastly flora as shown in Figure 2.2, was established by giving priority to the creation of essential aspects before addressing the less important ones, with flora being of lower importance. Each stage was supported by its dedicated function method, along with accompanying helper methods designed to facilitate its implementation. This methodology aligns with common practices in similar projects, which emphasises the importance of key components such as the road system, city block delineation, subdivision of blocks into building plots and side streets, and placement of city objects like buildings within these plots [4][7][18]. Additionally, carrying out the project in a sequential order also reflects the iterative and agile nature of the project's development process as discussed in Sub-section 1.3.1, ensuring a systematic and coherent progression towards the project's goals.

2.2.1 Terrain Generation

Figure 2.3: Top down (left) and Angled (right) view of generated 2D terrain

In the development of the procedural city algorithm, terrain generation uses a flat 2D model with a Cartesian coordinate system where the x-axis represents width and the z-axis depth. This is structured using a 2D array, which forms the foundational framework for the city

layout. The use of a 2D terrain without heightmaps, structured within a 100x100 grid, simplifies the design process and enhances computational efficiency. This setup focuses on optimising the urban layout and architectural diversity, facilitating a concentrated effort on the dynamic elements of the urban environment and streamlining the overall development process. The grid is populated using instances of the CityCells class, each characterised by a single Boolean attribute, 'isRoad'. This attribute is important as it dictates the designation of each cell within the grid, either as a part of road or a potential building plot. Through the initialisation, all cells in the array are set to isRoad = false, resulting in a uniform flat plane, which serves as the base for further city elements.

The DrawMesh method plays a crucial role in converting the 2D grid into a 3D mesh. This method constructs the terrain by iterating over each grid cell to define vertices that represent the corners of squares (or quads). These quads are laid flat on the ground to form the terrain's surface. Triangles are then established from these vertices to define the mesh's geometry, which Unity uses to render the terrain. UV coordinates are calculated for each vertex to facilitate correct texture mapping, ensuring that any textures applied will align properly within the grid. After iterating through all grid cells, the lists of vertices, triangles, and UVs are converted into arrays and assigned to the Mesh object, which is then set to recalculate its normals to ensure correct lighting and shading. The mesh is finally attached to a MeshFilter component and rendered using a MeshRenderer that is assigned the terrainMaterial. Within the Unity editor, a bespoke material, terrainMaterial, was crafted specifically for the terrain. This material is configured to exhibit a distinct green hue with a hexadecimal colour value of #029500, ensuring a vivid and uniform appearance for the base terrain. The material's properties were meticulously adjusted, setting both the smoothness and metallic sliders to 0, which results in a non-reflective, matte finish. This choice accentuates the flatness of the 2D terrain and avoids any visual distractions that might arise from shiny or glossy surfaces. The terrainMaterial is then applied to the terrain's mesh renderer, imbuing the 3D mesh with the intended colour and finish, which is especially crucial in the initial stages of development where the terrain is simply a flat green plane.

The DrawTexture method is crucial for applying a visual distinction to the terrain generated by the DrawMesh method. It starts by creating a Texture2D object that matches the dimensions of the grid, ensuring that each pixel in the texture corresponds directly to a cell in the grid. This method iterates over each cell to assign colours based on the cell's status. Initially, all cells are set to display a green colour, symbolising undeveloped terrain. Each colour is stored in an array which is then applied to the texture, transforming the abstract colour data into a visual format. The texture is set with a FilterMode.Point to keep the colours sharp and pixelated, enhancing the clarity of each grid cell when viewed up close. Once the colours are set, the texture is applied to the material of the terrain's mesh, providing a clear topographical view of the grid as shown in Figure 2.3.

2.2.2 Road Generation

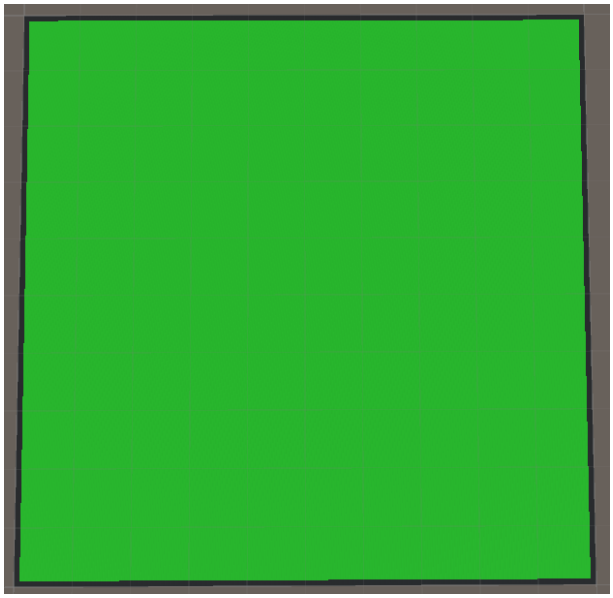


Figure 2.4: Terrain with outlining road border generated

The process of road generation integrates deterministic and stochastic methods to create a network of streets within the defined grid space, aiming to emulate the fundamental structure of modern urban designs that balances systematic order with elements of randomness. The visual goal for the city is to establish a grid system, a design pattern that is both common in modern city planning and functions as a straightforward yet efficient depiction of urban avenues [4]. As mentioned in the previous subsection on terrain generation, the grid is initially populated with CityCells marked as non-road areas. This process is followed by the outlining of the city's borders using roads, which serves to define the city limits and also lays the groundwork for the internal street layout, as illustrated in Figure 2.4. In the implementation, the outermost cells of the grid are designated as roads. Specifically, the code iterates over the grid's dimensions, setting the top and bottom rows, as well as the leftmost and rightmost columns of the grid to CityCells(true), which indicates these cells are roads. This forms a continuous boundary of roads around the perimeter. This methodical placement not only visually defines the city boundaries but also structurally prepares the framework for developing the internal street network and integrating other urban elements within these defined borders. The algorithm begins by methodically introducing horizontal streets at predetermined intervals. By making sure that these necessary roads cover the whole grid width, the programme establishes the framework for the road network (see 2.5). Two distinct algorithms for vertical road generation within the urban grid were developed and implemented across different sprint phases of the project.

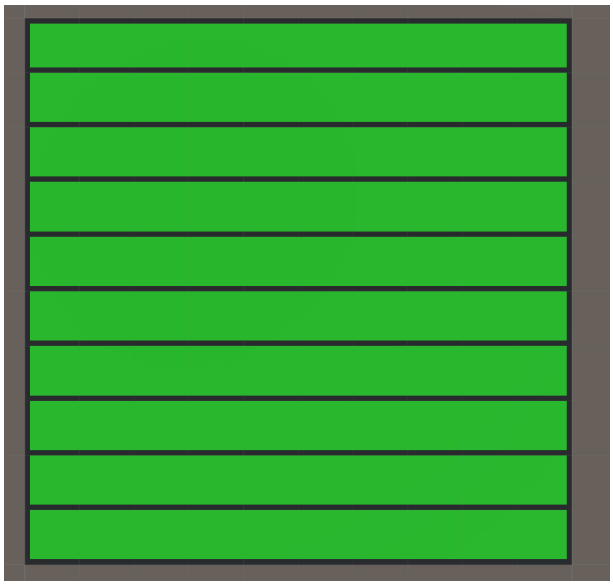


Figure 2.5: Screenshot of horizontal roads generated

Sprint 1 Implementation: Basic Road Generation

The first implementation is a simple algorithm that adheres to the regular grid pattern. This version ensures uniformity and straightforward navigation, ideal for the initial phase of development where simplicity is key. In the simple implementation, the algorithm generates vertical roads at intervals across the grid. It uses a random range within defined minimum and maximum street spacing to determine the placement of each new vertical street. As the algorithm iterates across the grid, it checks each cell along the potential street's path to ensure it does not overwrite any existing roads, maintaining the integrity of the intersections. Once a vertical road is placed, the grid is updated to reflect these new roads by setting the corresponding CityCells to isRoad =true. This method is repeated, incrementing the position by a randomly chosen value within the specified spacing range until the entire grid has been processed for vertical roads. This basic approach forms the first phase of the road layout, providing a fundamental grid structure as seen in many urban city plans.

Sprint 2 Implementation: Complex Road Generation

In the later stages of the project, a more complex implementation is introduced for road generation, the placement of vertical streets within the city grid is managed by a probabilistic algorithm that introduces variability and a more organic feel to the city layout. Each vertical road's direction, either upwards or downwards is determined through a randomized process at each column. Specifically, the algorithm generates a random value for each column and, based on this value, decides whether to create a vertical street. If the decision is affirmative, the direction of the street is then randomly chosen: a value of 0 indicates an upward direction, and a value of 1 indicates downward. This is executed using UnityEngine.Random.Range(0, 2), which equally weighs the chance for each direction. Upon determining the need and direction for a vertical street, the TryCreateVerticalStreet function is called to implement this decision. This function plays a crucial role in ensuring that these streets are added to the grid without

overlap and maintain minimum spacing from existing roads and is critical for actually drawing the vertical roads on the grid based on the parameters set by the earlier steps of the process. It takes parameters for the starting row, the column of the road, and the intended ending row of the street, along with the grid itself. Once invoked, it first adjusts the end row to remain within the grid boundaries, ensuring the road does not exceed the city limits. The function then determines the direction of street creation based on whether the end row is above or below the start row, iterating through each cell in the specified column between these two points. It checks each potential road cell against a minimum spacing requirement, ensuring that no new road is placed too close to an existing one within the given column range. If a road cell violates this spacing or if the road would stretch out of the grid's bounds, the function halts and returns false, indicating that the road could not be created under these conditions. Additionally, after attempting to place roads in each column, the algorithm verifies that at least one vertical road extends both upward and downward from each starting row, which is crucial for maintaining grid connectivity. If no road was successfully created in a desired direction, the function makes additional attempts in other columns of the same row until the connectivity is achieved. This systematic approach ensures that the generated city grid is not only visually diverse but also functionally interconnected, reflecting the dual objectives of spontaneity in appearance and structured connectivity in urban planning.

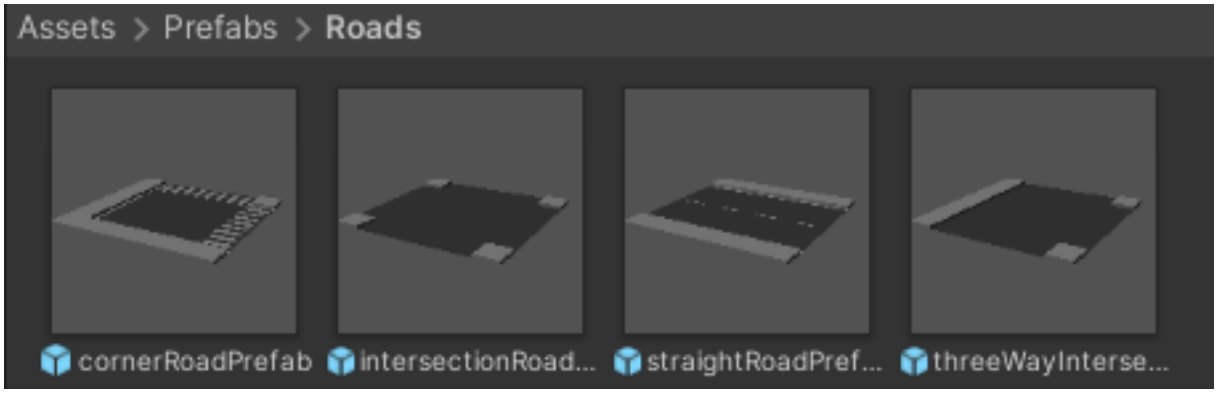


Figure 2.6: Road Prefabs imported from SimplePoly- City package

Once the underlying grid for the road network is established, the following phase involves visually enhancing the landscape of the city with 3D road models, known as 'prefabs'. These prefabs, sourced from the SimplePoly City package on the Unity Asset Store, include various road configurations like straight segments, intersections, three-way junctions, and corners, as illustrated in Figure 2.6. The placement of these prefabs is meticulously managed across the grid, ensuring each segment is appropriately represented and oriented to reflect the actual road connections. The placement process starts with iterating over each cell in the grid. For cells designated as roads (grid[x, y].isRoad), the DetermineRoadPrefab function is invoked to select the correct prefab based on the surrounding road connections. This function checks adjacent cells (north, south, east, and west) to determine the appropriate road type. For example, if a cell has roads on all four sides, an intersection prefab is chosen. If three sides are connected by roads, a three-way intersection is used, and so on. Special cases are handled for corners and edges of the grid, ensuring that each road piece fits seamlessly within the overall layout. Once the appropriate prefab is determined, the DetermineRoadRotation function is called to set the

correct orientation of the road segment. This function analyzes the same adjacent cells to decide how the prefab should be rotated to align correctly with other roads. Rotations are specified using Quaternion rotations in Unity, where, for instance, a road running east to west is rotated by 90 degrees to align horizontally. Different configurations like T-intersections or corners require specific rotations to ensure the visual continuity of roadways. Finally, the prefab is instantiated at the grid position with the determined rotation, using Unity's Instantiate function. This method places the prefab at the center of the grid cell and applies the calculated rotation. This meticulous approach ensures that each road segment not only fits logically within the grid but also aligns visually with adjacent segments, enhancing both the realism and functionality of the city simulation. By using this systematic method, the project achieves a dynamic and visually cohesive urban environment, reflecting the structured yet flexible nature of modern urban planning.

2.2.3 Building Generation

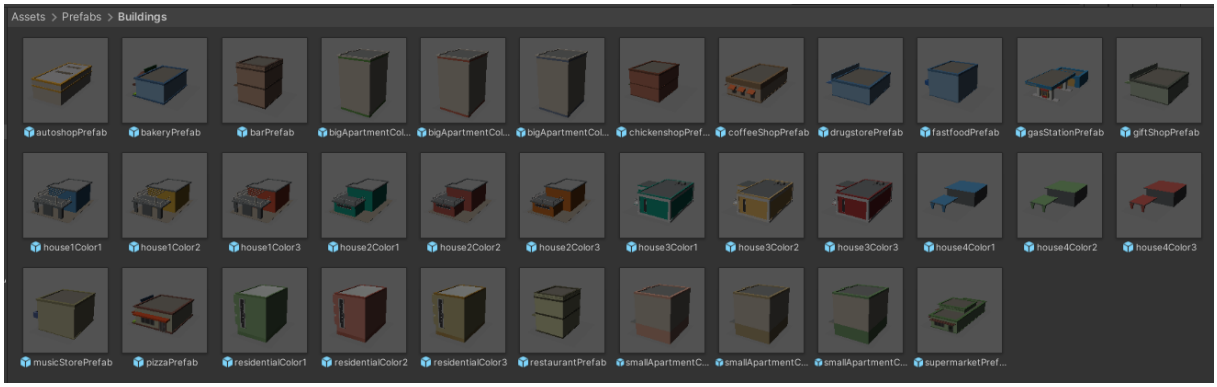


Figure 2.7: Building Prefabs imported from SimplePoly-City package

The building generation phase approach emphasises a systematic yet flexible placement of building prefabs, which are predesigned 3D models imported from the SimplePoly-City asset package [1] as illustrated in Figure 2.7. The use of prefabs not only speeds up the development process by reusing asset templates but also ensures consistency and quality across the generated urban environment. To standardise the footprint of various building models on the city grid, each building prefab is attached to a group of concrete tile prefabs too create a base. This methodological placement ensures that all buildings occupy a whole number of grid cells, avoiding fractional occupation which can complicate the generation logic. For instance, a building that might naturally cover 2.5x3 tiles in size is placed onto a concrete tile base that conforms to a whole number, such as 3x3 grid cells. This consistent sizing simplifies the city's layout and maintains uniformity in the allocation of space for each building, as shown in the accompanying visualisation Figure 2.8. By doing so, it guarantees that the placement logic remains coherent, and the city grid is utilised effectively, without irregular gaps or overlaps.

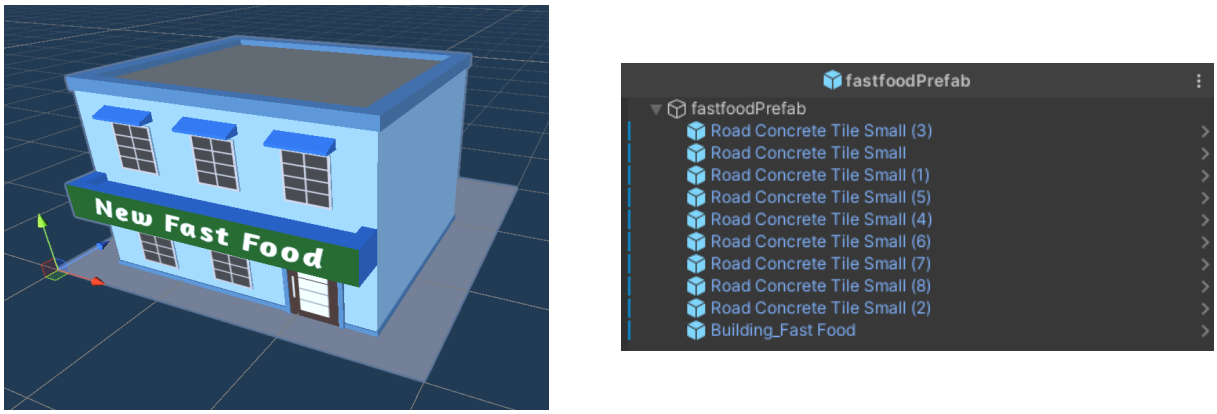


Figure 2.8: 'Building_Fast Food' prefab with its concrete tile base in Unity, showing 3D (left) and hierarchy (right) views

Building placement in the city simulation starts with the evaluation of suitable locations on the grid that are not occupied by roads or other structures. This step involves dividing the grid into two distinct zoning areas to facilitate structured development: the upper half for residential buildings, and the lower half for commercial buildings. This division is determined by the DetermineZone function, which categorises each grid cell's vertical position as either 'residential' or 'commercial'.

Once a zone is identified, the PlaceBuilding function is triggered for each unoccupied grid cell. This function coordinates the building placement process, starting with selecting an appropriate building for the space. The ChooseBuildingForSpace function incorporates the procedural element by randomly selecting a building prefab at random from a list of buildings that fits the zone's criteria. It checks the feasibility of placing each building in its default orientation and potential rotated positions using the CanPlaceBuilding function. It begins by accounting for the building's dimensions, which may need adjustment based on its rotation. For instance, if a building is rotated by 90 or 270 degrees, its width and depth are swapped to maintain accurate placement within the grid. The method then proceeds to assess whether the prospective building placement would exceed the bounds of the grid or overlap with existing structures or roads. This is done by iterating over each grid cell that the building would occupy and checking for any obstructions or infractions of the grid's limits. If any part of the building exceeds the grid or coincides with an already occupied cell, the method returns false, indicating that the building cannot be placed in that specific location. If a building passes the spatial requirements, the DetermineBuildingRotation method is initiated to define its orientation. Utilising quaternions to represent rotations, this method conducts a detailed analysis of the building's surroundings. It assesses the proximity to adjacent roads with the IsAdjacentToRoad function and allocates weights to each cardinal direction based on this adjacency. For instance, if there is a road directly to the north of the proposed building location, a greater weight is given to the north-facing orientation. Such weighted considerations ensure that the building's entrance is optimally positioned toward accessible roads, thereby enhancing the city's navigability and realism. The quaternion representing the rotation with the highest cumulative weight is then selected as the building's final orientation. In scenarios where road adjacency is equal in all directions or absent, the building is oriented to a default position using a standard quaternion rotation, preserving the consistency of the urban grid.

Once the building's orientation is determined, the AdjustPositionBasedOnRotation function recalibrates the building's position on the grid to accommodate its rotation. This recalibration is crucial, as it takes into account the building's width and depth, adding offsets where necessary to ensure the building's placement is properly aligned with the grid layout. This methodical realignment is essential for the buildings to seamlessly integrate with the urban landscape and to preserve the visual and functional coherence of the city simulation. These adjustments, informed by the building's dimensions and the calculated orientation, contribute to a harmonious city design. The building is then instantiated at the adjusted position with the correct orientation using Unity's Instantiate method. Followed by a call to the MarkBuildingOccupied function, which updates the grid to reflect the space the new building occupies, marking these cells as occupied to prevent future overlaps.

Should the initially chosen building not fit the available space after attempting all orientations, the FindAlternativeBuilding function is employed to select another suitable building. It retains a list of candidates that exclude the previously attempted building and meet the zoning and placement criteria by reapplying the same checks for space, rotation, and road adjacency. Using a random number generator, the function then selects from these candidates, introducing variability and ensuring that the building selection remains diverse and unpredictable. This randomised selection is crucial for simulating the architectural variety inherent in urban landscapes. If no suitable alternatives are found after a comprehensive search, the placement attempt for that cell is halted, ensuring that every building placement is both practical and aesthetically pleasing. This systematic and iterative approach to building placement ensures efficient use of space within the grid while adhering to zoning laws and accessibility requirements, ultimately creating a realistic and functional urban environment in the simulation.

2.2.4 Flora generation

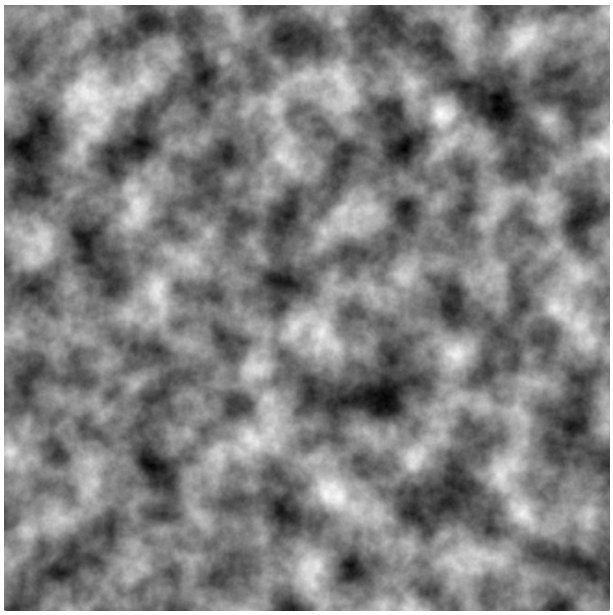


Figure 2.9: Example of a Perlin noise map [16]

Perlin noise, developed by Ken Perlin, is widely used in computer graphics to generate lifelike textures and landscapes. Unlike purely random noise, which can appear scattered and disjointed, Perlin noise creates a more coherent and smooth sequence of pseudo-random gradients. This is particularly useful in simulating the subtle variances found in natural environments, as depicted in the visual example of a Perlin noise map in Figure 2.9 [15][19].

In the city simulation, Perlin noise is employed to guide the generation of props and flora. Once again utilising asset packs like the SimplePoly-City [1] and SimpleNature [21], flora prefabs are strategically placed within the environment based on a noise map. This map is instrumental in influencing the distribution of natural elements, making the placement appear organic and random yet uniformly distributed across different scales. The propNoiseScale parameter adjusts the 'zoom' level of the noise, a smaller scale results in broader, more gentle variations, ideal for distributing larger objects such as trees. Conversely, a larger scale is used for clustering smaller objects like rocks or shrubs, providing intricate detail to the cityscape. Additionally, floraNoiseScale is employed to distribute smaller flora across different patterns, ensuring that these elements do not overshadow larger landscape features but instead enhance the overall visual complexity. Density parameters further refine this placement by setting thresholds based on noise values, dictating where flora and props will appear. A point on the map with a noise value below the propDensity or floraDensity becomes a potential placement location for a tree or flower, respectively. To enhance the non-repetitive nature of the environment, offsets are introduced during noise map generation. These offsets, random seed values, alter the noise pattern on the grid, ensuring that each simulation instance presents a unique arrangement of natural elements, thereby contributing to the organic feel of the virtual city's landscape. This methodical use of Perlin noise not only creates an illusion of randomness but also maintains spatial coherence, which is crucial for replicating realistic natural scenes in a digital format.

Chapter 3

Results

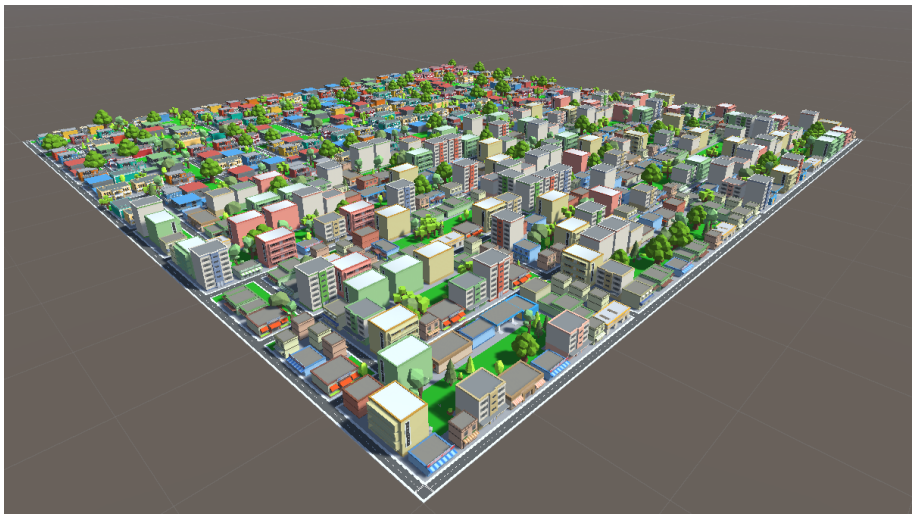


Figure 3.1: Screenshot taken of a city generated using the procedural algorithms

In this chapter, the outcomes of the procedural city generation system are systematically presented, revealing the system's ability to create complex and varied urban environments within a Unity-based simulation. A complete city, shown in Figure 3.1 highlights how well several algorithms are integrated, which together support the dynamic and finely detailed metropolitan landscapes that are produced. To visually underscore this point, close-up images are provided: the architectural diversity, precision of the road layout and the natural appearance of the flora is displayed in Figure 3.2. The images presented here, offer a snapshot of the intricate details and realism that the system can achieve, underlining the potential of a procedural approach to city generation. This successful integration of procedural techniques demonstrates that the project has effectively met its primary objectives, applying foundational principles to generate distinct, dynamic 3D cities and showcasing the substantial capabilities of procedural content generation in creating diverse and visually appealing urban landscapes.

Figure 3.2: Up close screenshots of buildings (left) and flora (right) generation

In the following sections, we will delve into a thorough assessment of the system's performance and evaluates the algorithm's efficiency and scalability. Following that, comparative assessments of basic and advanced road generation implementations provide more insight into the project's iterative development. The chapter concludes by comparing different iterations of the city generation process, showcasing the unique and varied outputs that affirm the algorithm's procedural nature. These sections collectively demonstrate the technical achievements of the generative approach and its potential for creating diverse and dynamic urban simulations.

3.1 Algorithm Performance Analysis

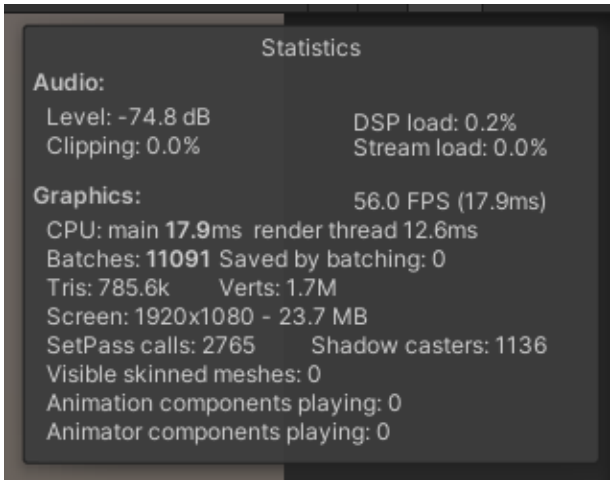


Figure 3.3: Unity's built in statistics feature

The simulations were conducted on a system equipped with an AMD Ryzen 7 5800H processor, NVIDIA GeForce RTX 3070 graphics card, and 16 GB DDR4 RAM, running Windows 11 Home. Critical metrics for assessing the performance of the city generation algorithm include memory usage, GPU usage and frames per second (FPS). Memory, CPU and GPU usage was carefully monitored using Windows Resource Monitor to ensure it stayed within the system's capabilities, avoiding potential slowdowns or system crashes. Meanwhile, FPS rates were tracked using the built-in stats feature in Unity (see Figure 3.3), providing a real-time measure of simulation smoothness. Higher FPS rates result in a more responsive experience, which is crucial for real-time applications. Maintaining high FPS rates is especially important in procedural city generation, where real-time user interaction and feedback are often necessary. Before initiating the city generation simulations, baseline performance metrics were established with the computer in an idle state to provide a reference point. Initial readings of memory, CPU and GPU usage were recorded to ascertain the system's base operational load.

Memory Usage	CPU Usage	GPU Usage
54% (8.6 GB / 16 GB)	5%	3%

Table 3.1: Baseline System Performance in Idle State

The primary variable in this analysis is the 'Size' parameter, which determines the dimensions of the grid representing the city. To comprehensively evaluate the impact of grid size on

performance, simulations were run at grid sizes of 50, 100, and 200. Each simulation was maintained for a duration of five minutes to gather ample data on memory usage, GPU usage, and frames per second (FPS). This duration allows for the calculation of average values, providing a more consistent and reliable measure of system performance under sustained load rather than peak conditions.

Size	Memory Usage	CPU Usage	GPU Usage	FPS
50	58% (9.3 GB / 16 GB)	25.7%	19.5%	185
100	63% (10 GB / 16 GB)	28.6%	18%	60
200	64% (10.2 GB/ 16 GB)	31.4%	16%	20

Table 3.2: Performance metrics of the system at runtime

The data in Table 3.2 indicates that the procedural city algorithm exhibits an expected increase in resource usage as the size of the city grows, this trend is normal in computational tasks of increasing complexity. The memory usage shows a moderate increase from 58% to 64% as the city expands from size 50 to 200. This increase is marginal in comparison to doubling and quadrupling the city grid, which suggests that the algorithm manages memory efficiently. The CPU usage increases steadily from 25.7% to 31.4% as the size parameter increases. This suggests that the algorithm requires more computational power for larger cities but does not indicate a significant strain on the CPU. The increase is proportional and suggest that CPU resources are being utilised effectively. An interesting thing to note would be the GPU usage as it decreases from 19.5% to 16% as the size increases. This could suggest that while the GPU is responsible for rendering the city, its load may not be directly proportional to the city size. Various factors could be affecting this such as the efficiency of rendering larger spaces or less frequent updates required for larger but dense cities. The FPS varies the greatest, peaking at 185 FPS for a city size of 50 and reducing dramatically to 20 FPS for a size of 200. This significant fall in FPS may indicate that the system is having problems displaying larger cities in real time, which might have an impact on the user experience in interactive apps. Overall, the system displays scalability in that it can handle larger city sizes, but there is a notable compromise in FPS, which is a critical factor for real-time applications. While memory and CPU usage show only slight increases, suggesting good scalability, the decline in FPS at larger city sizes points to limitations in rendering performance that may need to be addressed for optimal real-time interaction.

Figure 3.4: Close up screenshot of buildings from the city with grid Size 200

A visual assessment of the cities at different sizes reveals a consistent level of high-quality detail when viewed up close, as shown in Figure 3.4. This consistency indicates that the rendering process effectively scales with the increase in size without compromising on the finer aspects of the models. However, at the largest city size, 200, there is a noticeable pixelation occurring in areas further away from the camera's perspective. This effect can be attributed to standard rendering behaviour where distant objects become less clear as a consequence of screen space limitations. This loss of clarity at distance does not necessarily indicate a decrease in quality but rather a typical graphical response to viewing large-scale scenes on finite display resolutions. Despite this, the drop in FPS observed at larger city sizes suggests that the increased number of rendered objects might be placing a greater load on the system, particularly affecting real-time interaction capabilities. The accompanying Figures 3.5, 3.6, 3.7, provide a comparative visual context to these observations, showcasing the overall cityscape at different scales and demonstrating the simulation's graphical response to increased complexity.

Figure 3.5: City rendered at 'Size' 50

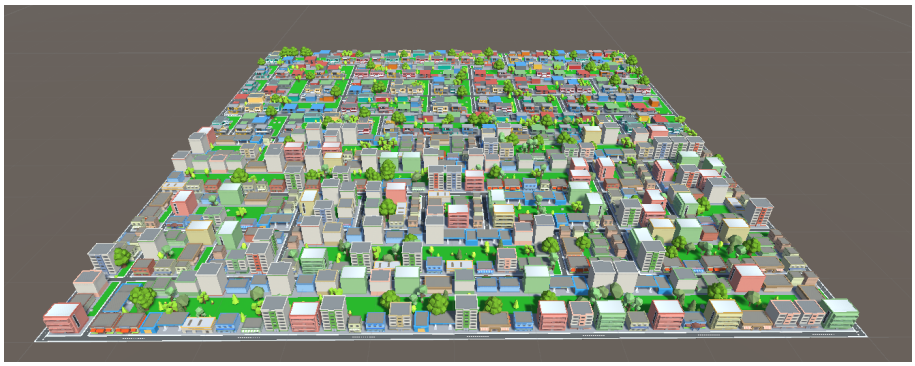


Figure 3.6: City rendered at 'Size' 100

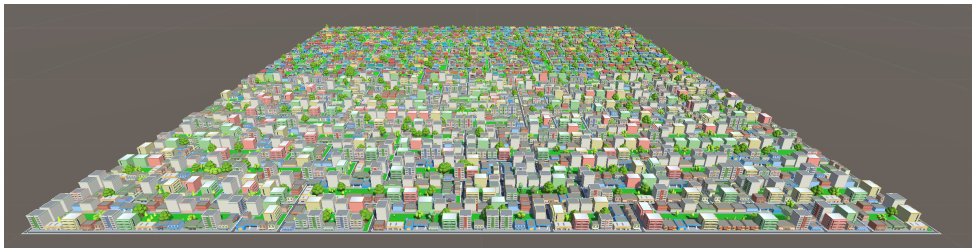


Figure 3.7: City rendered at 'Size' 200

3.2 Basic and Complex Implementation for road generation

The road generation within the city grid is a critical component of urban layout simulation, influencing the aesthetic appeal and overall realism of the environment. This section critically evaluates the two distinct road generation algorithms developed for the different sprint phases of the project. A detailed assessment will be provided for both the basic grid pattern and the advanced method, highlighting the resulting visual patterns and alignment with urban design standards. The investigation will address how each implementation impacts urban consistency, as well as how the simulation adheres to modern urban design concepts.

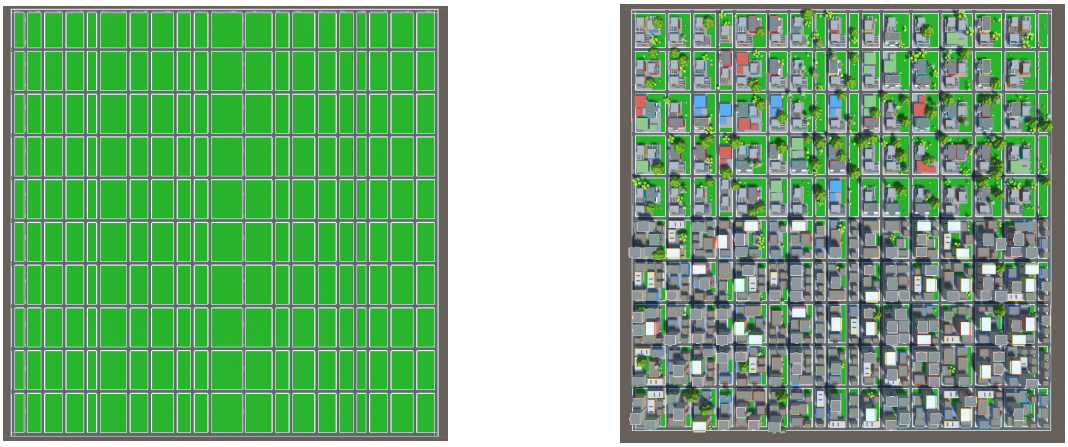


Figure 3.8: Results of the basic road generation algorithm. Left: Road network only. Right: Road network with buildings generated

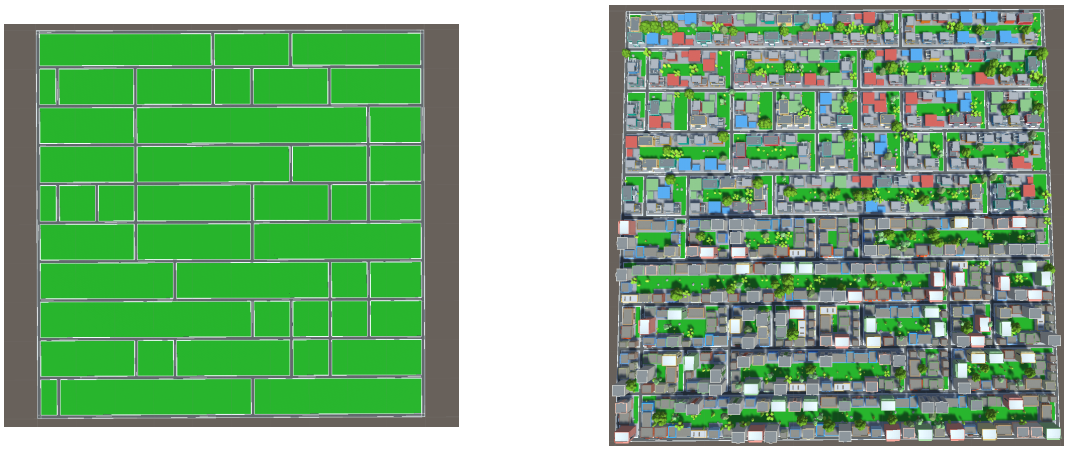


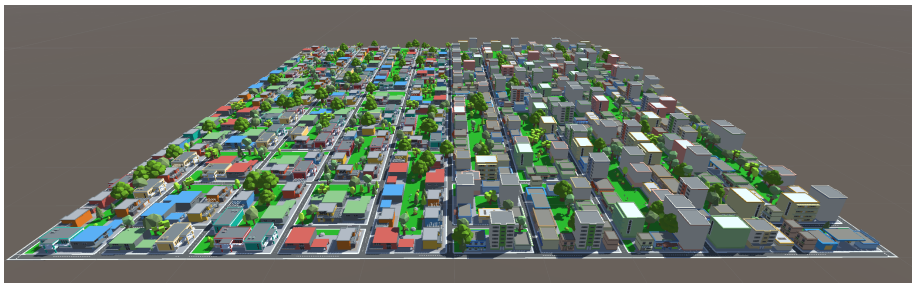
Figure 3.9: Results of the complex road generation algorithm. Left: Road network only. Right: Road network with buildings generate

The comparison between the basic and complex road generation implementations reveals distinct characteristics about the layout of the simulations. Figure 3.8 illustrates the basic algorithm, which randomises the intervals between vertical streets. Despite this stochastic element, the end result has a noticeable regularity, closely resembling a traditional grid system due to the continuous, unbroken extension of vertical streets from one end of the map to the other. This consistent layout results in a visually clustered arrangement of buildings and a uniform road spacing, which consequently diminishes the impression of procedural realism. The

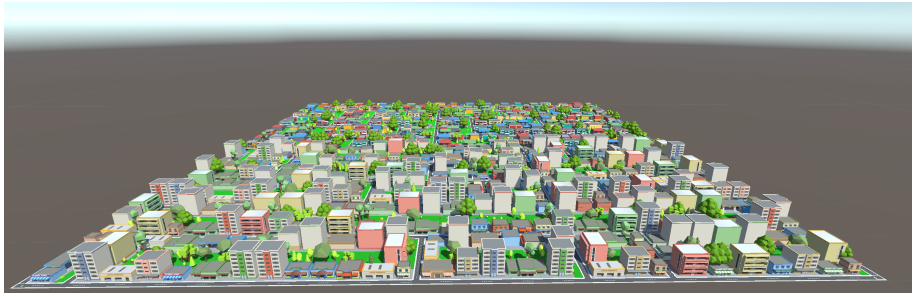
resultant cityscape, while successfully achieving a grid-like visual reminiscent of traditional urban planning, may inadvertently give the impression of being non-procedural due to its defined and fixed appearance. On the other hand, the complex algorithm employs a more probabilistic approach to road placement. This creates a more organic and irregular road network as shown in Figure 3.9. This algorithm simulates real-world urban irregularities by varying the shapes and sized of land parcels, utilising rectangles and squares of different dimensions. This variation is achieved by going through multiple random mechanisms within the algorithm, which influences the likelihood of road creation, thereby allowing larger plots of land to remain undeveloped for building. Although there is a degree of uniformity in the spacing of horizontal roads, it is largely overshadowed by the randomness introduced in the vertical road, contributing to a naturalistic urban appearance.

3.3 Comparison Between Different Instances of Generate Cities

This section delves into a comparative analysis of two distinct instances of city generation to demonstrate the variability and uniqueness in the procedural generation system. Despite employing the same algorithms, each generation results in a cityscape with its own unique characteristics and layout. This not only shows the flexibility and adaptability of the procedural approach but also showcases its ability to produce diverse urban environments that are not mere replicas of one another. The following comparison highlights the differences in spatial arrangement, providing clear evidence that each generated city is a unique product of the system's procedural nature.



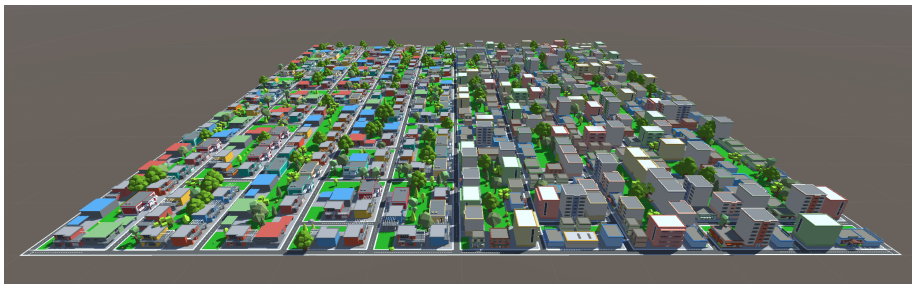
Side view of City A



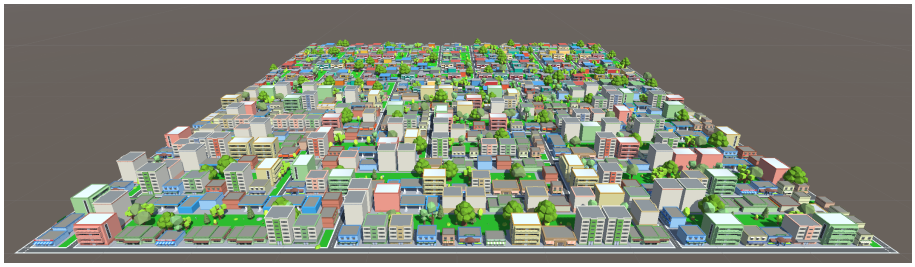
Front view of City A

Figure 3.10: Screenshots of City A from multiple angles

Despite the procedural nature of the city generation system, some underlying similarities between City A (Figure 3.10) and City B (Figure 3.11) can be observed, primarily due to the algorithmic constants in the system. Both cities exhibit similar patterns in the layout of



Side view of City B



Front view of City B

Figure 3.11: Screenshots of City B from multiple angles

horizontal roads, which are set at fixed intervals, and the zoning of residential and commercial areas, which is a predetermined split applied uniformly across generations. This fixed structuring lends a degree of similarity in the urban framework of the two cities. However, the placement and orientation of vertical roads show noticeable variability, with certain roads present in City A that are absent in City B and vice versa, as highlighted in the figures of both cities.

Additionally, while the building models are consistent between the two cities due to the use of predefined prefabs, the arrangement of these buildings varies significantly, contributing to the distinct character of each cityscape. This variation is evident in the provided figures, which showcase different configurations and placements of the same building types in each city. Examining the front views of City A and B in Figures 3.10 and 3.11 respectively, it becomes apparent that the front row of buildings differs significantly between the two instances. While there is some repetition of building models in certain areas, the overall arrangement and orientation vary considerably, showcasing the unique character of each generated cityscape. A similar observation applies when reviewing the side views. Figures 3.10 and 3.11 reveal that although some architectural elements recur, the composition and placement differ, further emphasising the individuality of each city.

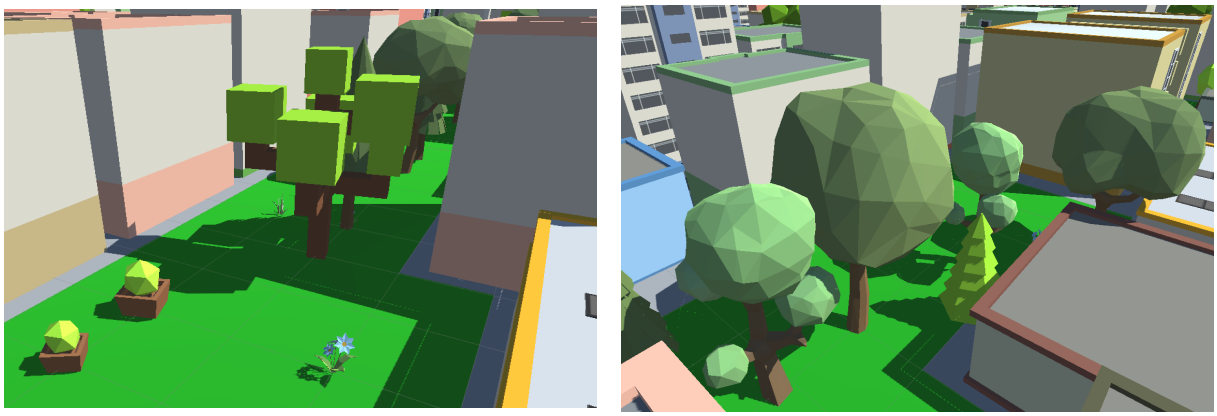


Figure 3.12: Close up screenshot of flora generated at X = 14,Z=5 in City A (Left) and City B (right)

Similarly, flora generation also demonstrates both consistency and uniqueness. Although the same models are used across both cities, their spatial arrangement and orientation differ due to random selection and rotation algorithms. This approach ensures that even with identical models, no two trees are exactly alike between the cities, further enhancing the distinctiveness of each generated environment. To delve deeper into this aspect, we compare the flora generation of the two cities at a specific point where x=14 and z=5. At this location, it is evident that each city has either generated a different type of tree or flora, or similar flora but with a distinct orientation. Additionally, the surroundings at this coordinate differ significantly, reinforcing the uniqueness of each urban landscape.

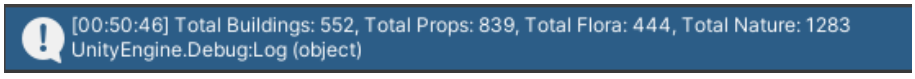


Figure 3.13: Total Count of objects generated in City 1

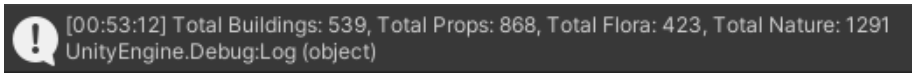


Figure 3.14: Total Count of objects generated in City 2

To further demonstrate the uniqueness between city instances, an analysis of the object counts reveals subtle yet significant differences, as documented in Figures 3.13 and 3.14. The first city instance (City A), detailed in Figure 3.13, comprises 552 buildings, 839 props, and 444 flora, amounting to a total nature count of 1,283. In contrast, the second city instance (City B), shown in Figure 3.14, presents 539 buildings, 868 props, and 423 flora, culminating in an overall nature count of 1,291. While the discrepancies in counts may appear marginal, they provide concrete evidence of the procedural generation system's ability to yield diverse urban configurations. These slight numerical distinctions, though indicative of the underlying algorithmic randomness, are significant as they could influence the predictability and reliability of the generation process, ensuring that each city remains a unique construct.

Chapter 4

Discussion

4.1 Conclusions

The aim of this project was to investigate and apply foundational principles of procedural content generation (PCG) techniques to create a 3D simulated city, demonstrating the potential of these techniques to produce urban landscapes with varied layouts and architecture that enhance realism. This project required extensive research into the history, evolution, and uses of PCG, as well as an examination of current implementations, all of which influenced the creation of a system capable of producing dynamic urban environments exclusively through algorithmic solutions.

The development process was methodically divided into multiple phases, each characterised by unique algorithms that built upon the capabilities of preceding stages, thereby adding layers of complexity and detail. The system employed a combination of Perlin noise and bespoke algorithms to create terrain, roads, buildings, and flora. This approach utilised randomness to ensure the procedural generation of urban landscapes, resulting in environments that were unique and unpredictable across different iterations. Despite some deterministic elements within the algorithms, the final outputs consistently demonstrated the procedural and unique nature of each generated city. .

Performance metrics evaluation revealed the high flexibility and scalability of the PCG techniques, with only marginal increases in memory and CPU usage as the size of the cities expanded. This supports the project's objective of demonstrating that procedural methods can effectively simulate urban environments without compromising system performance. Furthermore, the comparative analysis of road generation algorithms, alongside detailed reviews of two distinct city instances, provided solid evidence of the system's ability to enhance visual realism and create diverse, dynamic 3D city models. This analysis highlighted the differences between the cities, showing how each is uniquely affected by the procedural algorithms, thereby reinforcing the variability and uniqueness possible with PCG. These findings directly aligned with the project's goals to investigate how procedural techniques could improve both the visual and practical elements of simulated environments, affirming the project's success in meeting its stated objectives.

In conclusion, this report has highlighted the extensive capabilities of PCG in urban landscape simulations, showing that even basic implementations can produce diverse and visually appealing results. The PCG approach holds significant potential for resource efficiency, reducing the time and costs associated with content production while facilitating easy modifications for future applications. The outcomes of this project confirm that procedural content generation is a viable and transformative tool for creating dynamic, varied urban environments and paves the way for future advancements in urban planning and development projects.

4.2 Ideas for future work

While the current project establishes a solid foundation for demonstrating the potential of procedural content generation (PCG), the future path is both intriguing and required. Future efforts will try to improve the efficiency of the algorithms, lowering the computational overhead. This is especially important when creating large-scale cityscapes when the balance of detail and performance is vital. The Level of Detail (LoD) methodology is expected to help achieve this equilibrium. LoD may dramatically reduce processing power requirements and enhance frame rates by altering the complexity of object rendering according on the viewer's distance, which is an important component in user experience during real-time interactions. Further enhancements will look to diversify the urban ecosystem within the simulations. The introduction of additional elements such as parks, which serve as lungs for urban expanses, bridges that stitch disparate city sections, public transport systems that add dynamic flow, and the intricate detailing within building interiors, will contribute to a richer, more immersive urban experience. These enhancements not only improve the visual aesthetic but also encourage deeper user engagement with the simulated environment. The methodology for terrain generation also presents an opportunity for improvement. By integrating heightmaps, the terrain can exhibit a more varied and realistic topography, thus enabling more naturalistic landscape features. This would significantly augment the base upon which the urban structures are developed. In terms of road generation, there is a compelling need to move beyond the conventional grid patterns. The algorithms can be refined to simulate the more irregular, organically developed road networks often found in actual cities. This may include the implementation of non-cardinal directional roads that better mimic the natural evolution of a city's roadways. A shift from the reliance on prefab models to the use of algorithms that generate buildings procedurally, such as those based on L-systems, could introduce a novel degree of architectural diversity. Such procedural methodologies can potentially transform the architectural landscape, offering a multitude of styles and structures that resonate with the complexity and variety seen in real-world cities. Lastly, the project could greatly benefit from an interactive application that allows end-users to manipulate generation parameters directly. By enabling users to tweak settings and parameters on-the-fly, the system would not only increase its utility but also its accessibility to a broader audience. This interactivity could lead to user-driven content creation, opening up possibilities for personalised cityscapes and user-engaged design processes. In pursuing these enhancements, the project can continue to evolve, pushing the boundaries of PCG and cementing its role as a pivotal tool in the realm of digital simulation and beyond.

References

[1] U. 2024. Unity asset store: Simplepoly city - low poly assets, 2024. [Online]. [Accessed on 20/04/2024]. Available from: https://assetstore.unity.com/packages/3d/environments/simplepoly-city-low-poly-assets-58899.

[2] T. Angergård, M. Ansamaa, A. Arvidsson, J. Eriksson, A. Håkansson, and V. Truvé. Procedural generation of modern 3d cities. 2020.

[3] M. Blatz and O. Korn. *A Very Short History of Dynamic and Procedural Content Generation*, pages 1–13. 04 2017. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1007/978-3-319-53088-8_1.

[4] A. Brol and I. Antoniuk. Procedural generation of virtual cities. In *2023 24th International Conference on Computational Problems of Electrical Engineering (CPEE)*, pages 1–4. IEEE, 2023. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1109/CPEE59623.2023.10285307.

[5] Epic Games, Inc. Unreal engine 5, 2024. [Online]. [Accessed on 16/04/2024]. Available from: https://www.unrealengine.com/en-US/unreal-engine-5.

[6] Epic Games, Inc. Unreal engine 5: Blueprints, 2024. [Online]. [Accessed on 18/04/2024]. Available from: https://docs.unrealengine.com/4.26/Images/ProgrammingAndScripting/Blueprints/QuickStart/BPQS_5_Step8.webp.

[7] B. Erdei and S. Szenasi. Procedural city generation. pages 000473–000476, 05 2023. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1109/SACI58269.2023.10158565.

[8] W. Forsyth. Globalized random procedural content for dungeon generation. *Journal of Computing Sciences in Colleges*, 32(2):192–201, 2016.

[9] A. Fournier, D. Fussell, and L. Carpenter. *Computer rendering of stochastic models*, page 189–202. Association for Computing Machinery, New York, NY, USA, 1998. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1145/280811.280993.

[10] Godot Engine. Godot engine documentation, 2024. [Online]. [Accessed on 16/04/2024]. Available from: https://godotengine.org.

[11] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1), feb 2013.

[12] C. Hosking. Opinion: Stop dwelling on graphics and embrace procedural generation, 2013. [Online]. [Accessed Date: 18/4/2024]. Available from: http://www.polygon.com/2013/12/10/5192058/opinion-stop-dwelling-on-graphics-and-embrace-procedural-generation.

[13] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. 11 2007. [Online]. [Accessed on 15/04/2024]. Available from: https://www.researchgate.net/publication/357658334_Citygen_An_Interactive_System_for_Procedural_City_Generation.

[14] J. W. Park and S. H. Oh. A study on creation and usability of real time city generator via procedural content generation: – focus on virtual reality contents for senior. In *2019 International Symposium on Multimedia and Communication Technology (ISMAC)*, pages 1–4, 2019. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1109/ISMAC.2019.8836162.

[15] K. Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296, New York, NY, USA, 1985. Association for Computing Machinery. [Online]. [Accessed on 15/04/2024]. Available from: https://dl.acm.org/doi/10.1145/325334.325247.

[16] Y. Scher. Playing with perlin noise: Generating realistic archipelagos, 2017. [Online]. [Accessed on 21/04/2024]. Available from: https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401.

[17] N. Shaker, J. Togelius, and M. Nelson. *Procedural Content Generation in Games*. 01 2016. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1007/978-3-319-42716-4.

[18] R. Sharma. Procedural city generator. In *2016 International Conference System Modeling & Advancement in Research Trends (SMART)*, pages 213–217, 2016. [Online]. [Accessed on 15/04/2024]. Available from: https://doi.org/10.1109/SYSMART.2016.7894522.

[19] V. Sjögren and W. Malteskog. Procedural worlds: A proposition for a tool to assist in creation of landscapes by procedural means in unreal engine 5. 2023.

[20] Unity Technologies. Unity 3d engine, 2024. [Online]. [Accessed on 16/04/2024]. Available from: https://unity.com/.

[21] unity2024. Unity asset store: Low-poly simple nature pack, 2024. [Online]. [Accessed on 20/04/2024]. Available from: https://assetstore.unity.com/packages/3d/environments/landscapes/low-poly-simple-nature-pack-162153.

[22] R. Watkins. *Procedural Content Generation for Unity Game Development*. Community experience distilled. Packt Publishing Ltd, Birmingham, 2016. Chapters 1 and 2, pp. 1-32.

[23] G. N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, 2011.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

The project successfully achieved its main goal of developing and refining algorithms that introduced architectural variety in the 3D city simulations, demonstrating the effective potential of procedural content generation (PCG) for realistic urban landscapes. Despite this success, several challenges were encountered, particularly in the algorithmic complexity and implementation. The building placement algorithm struggled with the orientation and scaling of buildings, with unresolved issues in specific cases of building orientation. Similarly, the terrain generation was simplistic, resulting in flat 2D terrains that did not contribute effectively to the simulation's depth. The road generation, while less sophisticated compared to existing projects, performed adequately but encountered technical issues that, although resolved, led to delays and impacted following phases by requiring the use of prefab models for buildings. Minor visual issues, such as clipping with props, were noted, primarily due to the lack of collision logic between tree leaves, branches and buildings. Initially, project management was strong, with the project ahead of schedule during the first sprint. However, challenges in later phases led to slight delays. The project provided a detailed and concise analysis of the PCG output, showcasing the method's advantages and a few disadvantages. This critical self-evaluation underlines the need for future improvement in combining computational efficiency with visual detail, as well as increasing algorithm sophistication, in order to better satisfy project requirements.

A.2 Personal reflection and lessons learned

Coming into this project, my background in graphics was limited primarily to an introductory module in the third year of my studies, which was considerably lighter in workload. With virtually no prior exposure to procedural content generation (PCG), I found the initial phase of background research both overwhelming and intriguing. The technical papers were dense and complex, leading to uncertainty about my readiness to implement my own system. Initially, I struggled to grasp the concepts and define clear aims and objectives, which stirred doubts about my ability to successfully manage such an ambitious project. However, after several productive meetings with my supervisor, I gained a clearer understanding of the necessary steps, which significantly bolstered my confidence and guided my efforts.
Despite this newfound clarity, I soon realised that the high expectations I had set for myself, such as planning to implement very complex algorithms and create multiple unique, interactive cities were beyond my current capabilities and the scope of the project. This realisation caused indecisiveness and further doubts. From this experience, I learned the importance of setting more realistic and achievable goals. I recognised the effectiveness of starting with a smaller, manageable project and incrementally adding features and complexity. This approach not only

refined my project management skills but also taught me to align my ambitions with practical timelines and resources.
As the project progressed, it deepened my understanding of PCG and its methodologies. I learned a great deal about the various techniques used in PCG and the fundamentals required to complete a PCG project independently. The project challenged my resilience and significantly expanded my capabilities. Throughout, I faced numerous technical challenges, particularly with the building placement and road generation algorithms, which initially overwhelmed me. However, I learned to manage my stress more effectively and focus on finding solutions. This journey from feeling overwhelmed to becoming proficient in PCG techniques has been immensely rewarding, illustrating significant personal and professional growth.
Additionally, managing the project's workload alongside my third-year coursework proved challenging. I attempted to balance my module load across both semesters but found myself overwhelmed by coursework and project demands. In the future, I plan to better manage my time by scheduling breaks and starting implementation earlier to accommodate unforeseen technical issues and provide a buffer period.
Seeing the cityscapes come to life at the end was particularly gratifying, each element a testament to both my technical skill and creative vision. The setbacks, though frustrating, taught me valuable lessons in patience and perseverance. I also discovered the importance of flexibility, often needing to pivot my approach and adjust my expectations in real-time. These experiences not only honed my technical abilities but also strengthened my personal resolve and confidence in my own skills.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

In the development of the PCG generation for 3D simulated cities, the project primarily utilises tools and libraries provided by Unity, a widely recognised platform for game development. Given that the simulations are not based on any real geographical data and are intended primarily for the creation of virtual environments, rather than real world applications, the legal implication are significantly limited and there is no concern about data privacy. The software libraries used are covered under Unity's licensing. Since Unity's assets were employed, adherence to their specific licensing terms is required. The project uses several 3D models and assets from the Unity Asset Store, which are available free of charge and can be used under Unity's standard terms of service. These terms typically allow for extensive use within the Unity environment, which aligns with the project scope of designing virtual cities. The assets used can be found in the Appendix B section of the report. There are no copyright concerns as the project doesn't involve redistribution of software outside of its intended use within Unity and since the work is not intended for publication, the risk of infringement on copyright through the use of these assets is minimal.

A.3.2 Social issues

PCG technology may have a limited but notable impact on social aspects, primarily transforming employment and enhancing accessibility within specific contexts. This technology does not necessarily increase or decrease job numbers but shifts job roles; employers may increasingly seek individuals skilled in PCG techniques. Although PCG facilitates the creation of diverse and expansive environments, it complements rather than replaces traditional content creation methods. Artists are still needed to design environmental models, and programmers must implement the placement logic. Furthermore, PCG simplifies content development by allowing independent developers to create realistic, large settings with lower resources, broadening creative possibilities across resource levels. To avoid a digital divide, it is crucial to ensure that the tools and knowledge for using PCG are widely accessible, promoting inclusivity and broadening the benefits of this technology in content creation.

A.3.3 Ethical issues

In the context of this project, significant ethical concerns are notably absent. This comes from several key aspects of the project's design and intent. Firstly, the generated cities are not based on real-world data nor intended for direct application in real urban planning scenarios, which minimises concerns about potential impacts on real communities. Additionally, the project does not involve any sensitive personal data, thus avoiding the issue of privacy or data protection. The transparency of the algorithm's functions within Unity also ensures that there are no hidden or unintended use of the output. Finally, since the project is designed for educational and experimental purposes without the intent for public deployment, the typical ethical issues such as bias or economic consequences are not relevant. Therefore, while ethical issues are always considered and acknowledged in the development of this project, it does not present any major concern or challenges.

A.3.4 Professional issues

The project requires adherence to professional practices in software development. These practices include testing, accurate documentation, and continuous updating of the technology to ensure reliability and functionality. Moreover, the project highlights the importance of ethical programming and responsible design, ensuring that the simulations do not unintentionally create unrealistic expectations about urban environments. Maintaining professional integrity also means being transparent about the capabilities and limitations of the simulations produced, ensuring that users have a clear understanding of what the technology can and cannot do.

Appendix B

External Material

There are multiple materials used in the solution of this project which are the works of others. This include the prefabs models for the buildings, roads and flora that was used in the project. Other than that, the lighting function as well as the camera function was already implemented by the Unity 3D framework upon creation of a new project.
The two asset imports that I used from Unity Asset Store include:
[21] unity2024. Unity asset store: Low-Poly Simple Nature Pack, 2024. [Online]. https://assetstore.unity.com/packages/3d/environments/landscapes/low-poly-simple-nature-pack-162153 [20/04/2024]
[1] unity2024. Unity asset store: SimplePoly city - low poly assets, 2024. [Online]. https://assetstore.unity.com/packages/3d/environments/simplepoly-city-low-poly-assets-58899 [20/04/2024].

Appendix C

Source Code

The source code for this project is available on GitHub:
https://github.com/uol-feps-soc-comp3931-2324-classroom/final-year-project-braydennj