BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION [Informatics with English]

DIPLOMA THESIS

Automatic Procedural Generation for 3D Maps

Supervisor
[conf. univ. dr. Sergiu-Adrian DARABANT]

Author
[Cojocariu Vlad-Andrei]

2025

ABSTRACT

Procedural generation is an algorithmic technique for creating content dynamically. It involves using mathematical functions, randomization, and rules to generate diverse and unique elements such as landscapes, levels, characters, and quests. This approach enhances replayability, scalability, and creativity in various fields, particularly in the game industry. This thesis explores automatic procedural generation techniques for 3D maps. The research focuses on developing algorithms for terrain generation, object placement, and texture mapping. The study investigates the integration of procedural generation with user customization and interaction. The findings contribute to advancements in creating realistic and diverse 3D maps efficiently. The thesis provides insights, evaluations, and recommendations for future research in the field of automatic procedural generation for 3D maps.

Contents

Chapter 1

Introduction

Procedural Generation has quickly become an invaluable asset in game development to create unique and unpredictable gameplay experiences for players. Procedural Gen can also allow game designers to quickly produce expansive environments within game environments quickly. Consider procedural generation techniques when developing games; their automated processes not only save significant time but allow for greater creativity as well. They automate the generation process for game environments to streamline and speed up development but they can also offer greater creativity to designers. Game designers can focus their energy and attention on other important areas, like gameplay mechanics, narrative development and art design. Procedural generation techniques allow them to quickly generate highly replayable game environments. These procedural generated landscapes create more player engagement and satisfaction by offering new and challenging experiences with every playthrough of a game, keeping player interest alive throughout. Utilisation of procedural generation ensures maximum player involvement while procedural generation also provides procedural game playability with each new run through of playback. Generation techniques provide an added level of unpredictability and random-ness that helps create the sense of novelty and anticipation in games - leading to more immersive, rewarding player experiences. Landmass generation offers numerous advantages to 3D game environments. By simulating natural landscapes through procedural landmass generation, game designers can develop worlds which feel immersive and authentic. Procedural generation has concrete uses in the game industry, including generating expansive landscapes, creating dynamic levels and dungeons, populating game worlds with diverse characters, and generating personalized quests and storylines. These applications offer immersive and engaging gameplay experiences, enhancing replayability and player engagement. Exploring the techniques and challenges of procedural generation in this thesis aims to contribute to the advancement of this powerful tool, inspiring innovation in game development.

The introductory chapter provides an overview of procedural generation and its application in the game industry. It discusses the concept of generating content algorithmically and its potential to create dynamic and immersive experiences. The chapter highlights the benefits of procedural generation, such as cost-effectiveness and increased scalability, while acknowledging the challenges involved in algorithm design. Overall, the chapter sets the foundation for further exploration of procedural generation in the thesis.

Chapter 3 delves into different techniques for generating landmasses procedurally. It covers the generation of heightmaps using noise functions, fractal-based terrain generation, and mathematical methods for terrain generation. The chapter also discusses the applications of these techniques in games and the types of terrains that can be generated.

Chapter 4 introduces the Unity Engine, a popular game development platform. It provides an overview of Unity and its key features, including the Unity Editor, asset management, and physics simulation.

The final chapter presents an application of procedural generation, providing an explanation of the chosen approach. It discusses the playability and possibilities of the generated content. The algorithm used for the generation is explained, including the input parameters and the steps involved. It also explores the concept of noise and how it contributes to the generation process. Additionally, the chapter discusses the implementation of endless generation, allowing for a seamless and infinite world.

Chapter 2

Introduction to Procedural Generation

Procedural generation (PG) is an approach used in computer graphics, game development and other creative fields for the generation of content such as landscapes and buildings to characters and music. While PG's roots date back decades - from its debut in 1960 - its usage has only recently surged with technological developments and rising demands in game industries for unique and varied pieces of entertainment content.

2.1 Overview of Procedural Generation

Acording to [STN16a], Procedural Generation (PG) is a process which uses algorithms and mathematical formulas to produce content - from landscapes and environments, characters, music tracks, or stories - quickly and efficiently. Game developers utilize procedural generation techniques like these when producing vast amounts of game assets quickly. A set of rules or instructions used as templates allow PG's creators to craft an almost limitless variety of pieces of work through this approach.

Procedure-driven generation begins with a set of algorithms and parameters which outline rules for creating content. These may be simple or complex depending on desired output, such as randomly generating numbers; more complex algorithms might produce realistic landscapes instead.

Once algorithms and parameters have been defined, procedural generation begins. An algorithm uses rules defined in advance to generate content - anything from landscapes and environments, characters and music tracks through stories created dynamically by game developers can all be generated procedurally and modified real-time to provide dynamic game environments.

The book [STN16a] also talks about the indispensablility of procedural generation resources for game developers, enabling them to quickly generate fresh and var-

ied content quickly and efficiently. This approach can especially beneficial when developing open world or multilevel games; handcrafted content creation may prove challenging when creating this vast universe with limitless possibilities and variations - giving each gamer their own personal gaming experience!

Although procedural generation offers several benefits, it also presents some unique challenges. One such obstacle lies in creating content of high-quality and visual appeal using algorithms alone; this requires knowledge of art and design principles as well as being unpredictable enough for players navigating a game to enjoy it smoothly. Furthermore, procedural generation may sometimes create content too unpredictable or inconsistently which frustrate their gameplay experience.

Procedural generation is an efficient technique used by game developers to quickly generate diverse, unique, and varied content quickly and efficiently. Procedural Generation uses algorithms and mathematical formulas to produce output based on predetermined rules or parameters; although procedural generation offers numerous advantages it also poses unique challenges which must be surmounted in order to produce high-quality work.

2.2 Application of Procedural Generation in the Game Industry

The impact of Procedural generation is depcited in [TXS17] and it states that Procedural generation has gained increasing attention within the game industry for its ability to rapidly generate vast, complex game worlds quickly and effectively. Procedural Generation can be found in No Man's Sky, Minecraft and Spelunky among many others - this section will discuss its applications within game design as well as any benefits it brings.

Procedural generation plays a pivotal role in game industry through its use in creating open-world games. Open world games allow players to explore vast, open environments freely without restrictions from developers; creating these worlds can be time consuming when done by handcrafted content; procedural generation provides developers a quicker, simpler solution: using algorithms for landscape, environment and structure creation that offers players endless possibilities and keeps them interested.

Another application of procedural generation is in creating randomly-generated levels. Randomly generated levels have become an increasingly popular feature of popular videogames such as Spelunky and Dead Cells; every time players play they encounter fresh levels generated randomly - providing players with an entirely unique experience every time! Procedural generation for level creation can

especially come in handy when levels require exploration to progress forward.

Procedural generation has also become a powerful tool in game creation by way of Non-Playable Characters (NPCs) and enemies, providing players with challenges while furthering the storyline of many titles. Procedural Generation allows developers to rapidly generate unique, diverse NPCs and enemies quickly while setting parameters that govern how these non-playerables and enemies behave and interact within the world of gameplay for an engaging experience that adds extra layers to any given title.

Finally, procedural generation can also be utilized for creating music, sound effects and stories within games such as Minecraft. With AI Dungeon procedural generation is used to generate stories which offer players with new and thrilling adventures each time the game is played - providing players with a truly exciting gaming experience each time.

2.3 Benefits of Procedural Generation

Procedural generation offers numerous advantages to both game developers and their audiences alike, and this section will examine some key benefits associated with procedural generation.

1.Efficiency: Procedural generation allows developers to easily and quickly construct vast and intricate game worlds quickly, eliminating the time-consuming task of handcrafting every asset and level themselves by replacing this effort with algorithms which generate content automatically - freeing them up for other aspects of game development.

2.Diversity: Procedural generation creates game content that changes each time players access it, giving a refreshingly new experience each time the game is played - increasing replayability and engagement! For instance, No Man's Sky utilizes procedural generation to produce an infinite variety of planets each featuring distinct landscapes, flora, and fauna - increasing replayability and engagement significantly.

3. Scalability: Procedural generation can produce content which adapts to match players of varying skill levels as they play through the game, automatically adapting its difficulty as players progress along their journey and guaranteeing they experience a more engaging gaming experience overall. Procedural Generation can produce dynamically tailored content which keeps gamers challenged while remaining engaging; making for an enriching gaming experience overall.

4.Consistency: Procedural generation ensures game content remains coherent and uniform over its entire run-time, mitigating any possibility for human error and making for an immersive, engaging player experience.

5.Accessibility: Procedural generation can make games more accessible to people with disabilities, like Minecraft for instance, by creating environments which make navigation simpler for players with mobility impairments. This ensures more players enjoy and have an enhanced gaming experience.

6. Cost-Efficient: Procedural generation can significantly cut game development costs by automating game content production. By eliminating handcrafted assets that add time and cost, procedural generation also cuts costs by decreasing team member counts like level designers who may otherwise add extra members into development teams - further decreasing game costs!

7.Innovation: Procedural generation provides game developers with opportunities for innovation in game design. By automating content production, developers are free to experiment with innovative game mechanics, environments and challenges which would otherwise be challenging or impossible to create manually - encouraging creativity while leading to more innovative and engaging game experiences overall.

2.4 Challenges of Procedural Generation

Procedural generation - or the process of algorithmically creating content - has gained more traction in game development over recent years, due to its apparent advantages: creating vast, dynamic worlds which challenge and engage players simultaneously. But like any emerging technology, procedural generation presents several obstacles which must be met head on for its full potential to be realized.

Procedural generation presents one of the greatest challenges: striking an equilibrium between randomness and predictability. While randomness helps create fresh content, too much randomness may result in disjointed game world environments or predictable experience for players; therefore achieving this balance between randomness and predictability is paramount to creating games which offer both challenging and engaging gaming experiences for all audiences.

Another challenge of procedural generation lies in its intricate design and implementation process. Procedural generation offers endless potentialities; however, its implementation must be thoughtful to ensure meaningful experiences are created for players. If improperly implemented procedural generation occurs it could result in game-breaking bugs, unbalanced gameplay or other issues which negatively affect player experience.

Also, procedural generation poses its own set of difficulties for game designers. While procedural games offer expansive and lively worlds to explore, creating meaningful narratives and character arcs through procedurally generated games may prove more challenging - this lack of emotional engagement may prevent player

enjoyment altogether!

Final challenge of accessibility. Procedural generation creates complex and engaging gameplay which may be hard for some players to navigate, while its heavy reliance on algorithms and programming makes understanding game mechanics and systems challenging and often results in frustration or disengagement.

Chapter 3

Other Procedural Landmass Generation Techniques

3.1 Generation of Heightmaps using Noise Functions

Procedural generation has grown increasingly popular in 3D gaming as it allows the creation of vast and immersive game worlds without being constrained by traditional handcrafted level design constraints. One key element of procedural generation is using noise functions to generate heightmaps.

Heightmaps are two-dimensional grayscale images that depict elevation changes of terrain. When used in 3D games, heightmaps often serve as the basis for creating terrain meshes which then can be filled out and decorated with objects like trees and rocks to produce realistic environments, this can be seen in Figure 3.1.

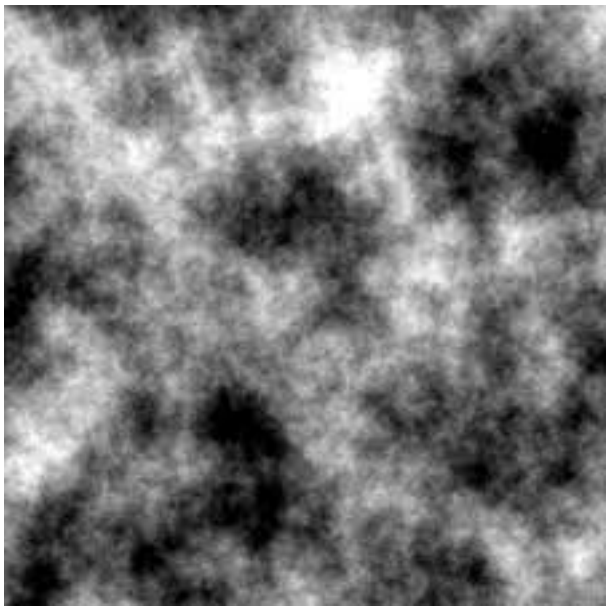

Figure 3.1: Noise Map

The book [TXS17] touches the subject of Noise functions. Mathematical algorithms that produce random values based on certain parameters, creating effects ranging from simple randomness to complex patterns. There are various kinds of noise functions commonly employed for procedural generation such as Perlin noise, Simplex noise and Worley noise functions.

Perlin Noise was developed by Ken Perlin in 1983 and remains one of the most frequently employed procedural generation noise functions today, often mimicking natural phenomena like clouds and water surfaces. Ken also created Simplex Noise which provides faster yet more effective versions of Perlin noise producing similar results; furthermore Worley noise (known as cellular noise) provides another type of function with similar patterns produced resembling cells.

Noise functions allow one to create heightmaps using grids of points that each receive a random value from their noise function of choice and interpolate these into smooth surfaces for interpolation purposes. Once complete, filters or adjustments to parameters can further customize and refine this heightmap for use as part of their search processes.

Noise functions offer an effective solution for creating terrains with organic forms that resemble those seen in nature. Traditional level design limits terrain creation to the imagination and skill of its designer; with noise functions however, terrain generation can occur randomly or with controlled variations for an engaging and intriguing landscape experience.

Noise functions provide developers with an effective tool to easily create diverse biomes and landscapes within one game world, quickly. By simply altering its parameters, they can produce different terrain features like mountains, hills and valleys in minutes - then fill these features with objects suitable for that biome such as snow-capped mountains or lush forests, we have an example in Figure 3.2.

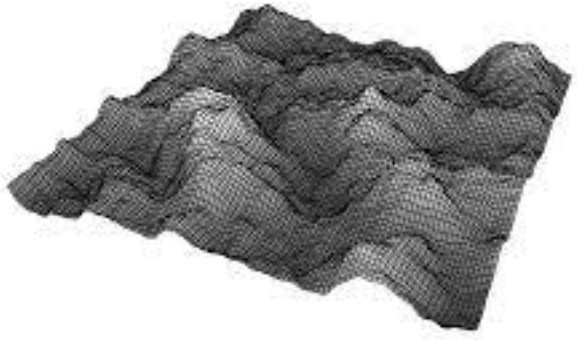

Figure 3.2: 3D Noise Map

Finally, using noise functions to easily vary terrain offers another practical benefit to game development. Developers can quickly create unique game environments without spending hours manually level designing levels themselves - saving both time and resources while freeing them up to focus on other aspects such as gameplay mechanics or story.

3.2 Fractal-based Terrain Generation

Fractal-based, as depicted in [KTY14], terrain generation is a technique employed in 3D game development that utilizes mathematical functions known as fractals to generate terrain. Fractals are patterns with repeating scale-dependent repeat patterns; their use for terrain creation enables realistic landscape designs, displayed in Figure 3.3.

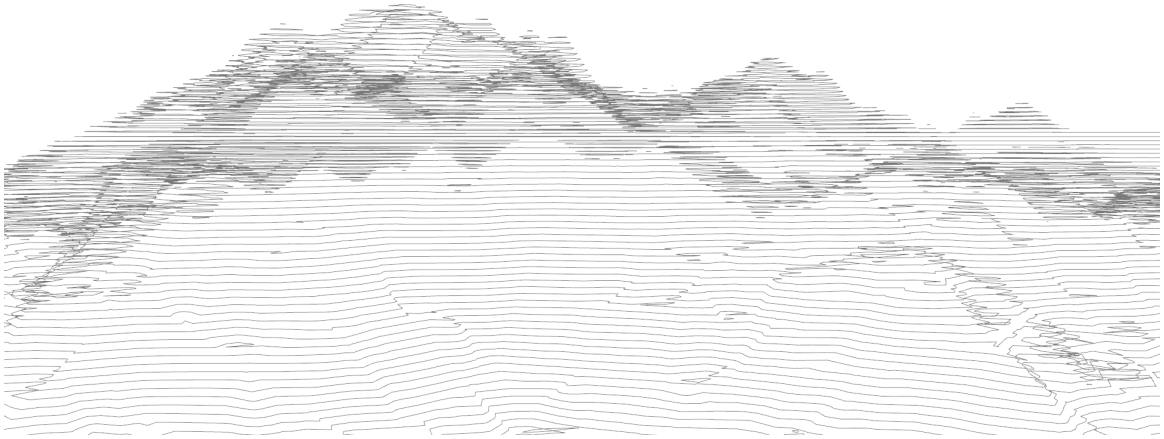

Figure 3.3: Fractal Generation

To create a fractal-based terrain, a fractal function must first be applied to a heightmap in order to generate elevation patterns that serve as building blocks of terrain meshes; these patterns may then be further refined via texturing and object placement.

The book [STN16b]states that ne of the most widely utilized fractal functions for terrain generation is Diamond-Square algorithm, commonly referred to as Midpoint Displacement algorithm. This function generates a fractal pattern by repeatedly subdividing one square into four smaller ones and computing their center point height by taking their average of four corners plus random offset. Once complete, this process repeats across each smaller square until creating something resembling natural terrain fractal patterns.

Perlin noise fractal function is discussed in [STN16b] as another popular terrain generation technique. Perlin noise can create all types of terrain features including

hills, mountains, and valleys, making complex landscapes that look realistically created using Perlin noise combined with other fractal functions such as polynomials or clustering algorithms possible.

Fractally generated terrain generation offers several distinct advantages when used for creating realistic and visually attractive landscapes without needing to spend hours manually level designing them. Fractal functions create realistic terrain that appears natural yet complex details add depth and immersion in any game world they inhabit.

Fractals enable developers to easily vary terrain by making it simple to change biomes and landscapes within one game world. By altering parameters of fractal functions, they can craft terrain that features deserts, forests and swamps with ease.

Fractal terrain generation also facilitates seamless blending between terrain features, making transitions between different biomes and landscapes seamless. This can enhance overall visual quality as well as player immersion in your world.

Fractal terrain generation offers another key benefit of game development efficiency. As terrain is generated procedurally, manual level design becomes unnecessary - freeing developers up to focus on other aspects such as gameplay mechanics or story.

3.3 Mathematical Methods for Terrain Generation

Mathematical methods are another method talked about in [KTY14] and have long been utilized in 3D game terrain generation. This chapter delves deeper into their use to produce visually attractive terrain in videogames using mathematical functions and algorithms to generate realistic terrain models.

One of the most often employed mathematical functions for terrain generation is sine function. By applying it to a heightmap, developers can generate undulating terrain that mimics real world landscapes with curves and slopes found naturally. Furthermore, its frequency and amplitude can be adjusted in order to create different kinds of landscapes, from gentle rolling hills to steep mountain ranges.

The article [KTY14] talks about Gaussian functions are another well-used mathematical function in terrain generating. These bell-shaped curves can help generate terrain features like hills and valleys by applying this mathematical function to heightmaps containing elevation values. Developers can utilize Gaussian functions for smooth elevation distribution that looks natural aesthetically.

As well as mathematical functions, algorithms have also been developed for terrain creation. One such algorithm is Voronoi diagram algorithm which creates terrain from point distribution in a grid by creating polygons around each point and their height determined by distance from nearest points - yielding hills and valleys

with distinct patterns created by point distribution.

Delaunay triangulation algorithm is another widely utilized terrain generating technique. This algorithm creates a triangulated mesh from points specified by an input noise function, with height determined for every point via noise functions. The mesh produced can then be used to craft complex, varied terrain with high levels of detail.

One advantage found in [STN16a] using mathematical methods for terrain generation is their ability to produce realistic yet visually appealing terrains. By applying mathematical functions and algorithms, developers can generate terrain that mimics real world landscapes while adding unique or appealing features that add life and interest.

Finally, mathematical methods enable easy terrain variation within one game world by altering parameters of mathematical functions and algorithms - creating biomes with diverse biomes like deserts, forests and swamps within its limits.

3.4 Applications in Games

Previous chapters explored various techniques for procedural generation of terrain in 3D games; here we'll look at their use within actual games.

Procedural terrain generation has proven particularly useful in open-world games like No Man's Sky and Minecraft where players are given free rein to roam freely across large environments. By employing procedural generation techniques developers are able to produce vibrant game worlds without manual level design being needed; such as No Man's Sky where game world is generated on-the-fly while player explores thus creating infinite varieties of terrains and landscapes for them to traverse.

An additional use for procedural terrain generation lies in creating randomly-generated levels in action and adventure games like Spelunky or Risk of Rain that feature such dynamic playability. Through procedural generation, developers are able to craft levels which change each time someone plays them - providing more replay value!

Procedural Generation can also help create terrain-based puzzles like Terraria and Starbound do. By randomly creating new layouts each playthrough, these games keep players guessing and add an exciting element of surprise and uncertainty for an unpredictable experience!

procedural generation also plays a pivotal role in creating lifelike landscapes for simulation and training games like Flight Simulator or Farming Simulator, providing safe environments where players can practice skills safely in controlled conditions.

Procedural generation can also be leveraged for use in creating unique enemies and creatures in videogames such as Spore or No Man's Sky by randomly generating enemy designs and behaviors to give players an unpredictability or challenge that engages their imaginations. By randomly producing various enemy designs and behaviors, procedural generation creates unpredictability while adding variety, expanding player discovery.

The book [STN16a] rocedural generation can also be utilized to generate weather and environmental effects in videogames such as Minecraft and Don't Starve, using procedural algorithms to produce dynamic weather patterns, seasons and natural disasters that allow players to explore realistic environments while staying alive in them. By simulating realistic settings with procedurally generated weather effects such as thunderstorms or natural disasters - procedural generation creates immersive and realistic worlds in which to play out adventures or find safe shelter from them all.

Periodic Generation can also be utilized in game development for creating novel loot as talked in [STN16a] systems using random loot drops and rewards generated using procedural generation, creating excitement and surprise amongst players as they interact with its content and explore its depths. Popular examples such as Borderlands and Diablo use this technique, adding replayability and longevity.

Overall, procedural generation's applications in games are vast. By employing algorithms and programming to dynamically generate content dynamically, developers are able to craft engaging game worlds that offer both unique and challenging gameplay experiences. From open world exploration to randomised levels and enemies - procedural generation provides game development teams with a versatile tool which continues to advance along with technological innovations.

3.5 Types of Terrains Generated

Procedural terrain generation can create a variety of terrains, ranging from realistic landscapes to fantastical and surreal environments. In this chapter, we will explore some of the most common types of terrains generated using procedural techniques.

Realistic Landscapes: One of the most common applications of procedural generation is the creation of realistic landscapes. These terrains are often based on real-world data, such as elevation maps, satellite imagery, and geological data. By using these inputs, developers can create game worlds that resemble real-world environments, complete with mountains, valleys, rivers, and other natural features. Games like Grand Theft Auto V and Red Dead Redemption 2 use this technique to create expansive game worlds that feel alive and dynamic.

As our subject is about realistic landscapes

Procedural terrain generation (PTG) is an indispensable technique in game development that enables developers to quickly generate complex landscapes without the need for manual level design. PTG employs algorithms and programming languages to generate terrain features like mountains, valleys, forests and rivers without manually designing levels from scratch. We will explore in this chapter all kinds of procedural generation terrains you can generate using procedural generation techniques.

Mountainous Terrain: For procedurally generated terrain generation to work effectively, mountains must include steep slopes, as seen in Figure 3.4, rugged peaks and deep valleys characterized by sharp corners. By employing noise functions to generate random heights and slopes for their mountain ranges in games like Skyrim and Far Cry, developers are able to produce realistic mountain ranges that create realistic gaming experience for users. This type of terrain often features prominently when created procedurally.

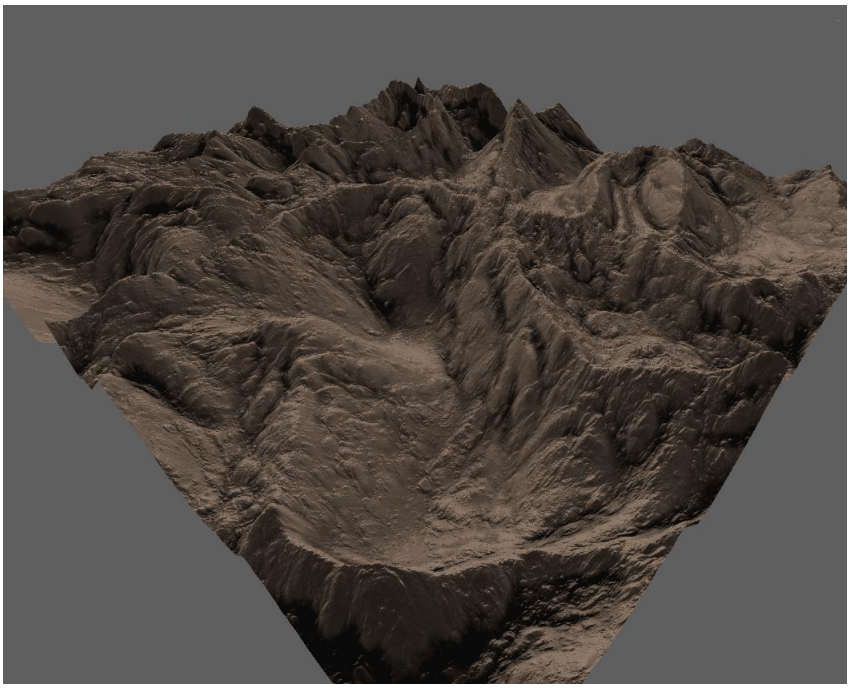

Figure 3.4: Moutainous Terrain Example

Forest Terrain: Procedural generation can also be utilized to produce forest terrains characterized by lush vegetation, trees and underbrush. By employing noise functions that generate random tree placement and density patterns, game developers can achieve realistic forests within their games such as The Elder Scrolls Online or The Forest. This form of terrain creation has proven popular within gaming applications like these two titles.

Tundra Terrain: Tundra terrain can also be generated using procedural generation techniques, with this particular terrain typically consisting of low temperatures,

snowfall and ice formations. By employing noise functions to randomly generate snowdrifts and rock outcroppings using noise functions to generate these features realistic tundra landscapes are created which have become predominantly featured in games like Frostpunk and The Long Dark.

Coastal Terrain: Procedural generation can also be used to generate coastal terrain, which typically features beaches, cliffs and rocky shores. By employing noise functions to produce random terrain features like wave patterns, rock formations or tide pools a developer can produce realistic coastal environments commonly found in games like Sea of Thieves and Subnautica.

Canyon Terrain: Procedural generation can also be utilized to generate canyon terrain, which features steep cliffs, narrow canyons and deep gorges. By employing noise functions to randomly generate rock formations and canyon walls using procedural generation techniques, developers are able to craft realistic canyon landscapes such as those seen in Horizon Zero Dawn or Tomb Raider games. This form of terrain generation has proven incredibly popular within game environments like these two titles.

Desert Terrain: An example of procedural generation terrain would be desert terrain, represented in Figure 3.5. Featuring sand dunes, rock formations and sparse vegetation reminiscent of desert environments found in games like Assassin's Creed Origins or Red Dead Redemption.

Figure 3.5: Desert Terrain Example

Developers use noise functions to generate realistic yet varied desert landscapes using hills, valleys, rock formations etc. to generate realistic yet diverse desert landscapes using procedural generation technology. This form of generation terrain generation has proven popular with developers using it extensively for use within procedural generation games like Red Dead Redemption as it creates realistic yet varied desert landscapes used throughout these games as it often feature realistic and varied desert environments found within games such as these titles like these titles or

Red Dead Redemption when creating realistic yet varied desert landscapes using procedural generation technologies as it uses noise functions randomized terrain features such as hills valleys etc generating random hill features/valleys/ rockformations randomly to produce realistic and varied desert landscapes used frequently throughout games like these titles such as these ones.

Procedural generation techniques offer developers the potential for creating dynamic terrain types in games ranging from mountains and desert landscapes, forests, tundra, coastal regions and canyons, plus many others - creating immersive and varied terrain features which enhance player immersion within an experience-rich gaming world. Utilizing noise functions or other algorithms developers can generate dynamic yet varied terrain features which enrich gameplay experiences while heightening immersion levels within their titles.

The limits are not only with real-life examples, we can go further to: Fantasy Environment, Post-apocalyptic Landscapes, Alien Worlds, Abstract Landscapes.

Chapter 4

Unity Engine

4.1 Overview of Unity Engine

Unity Engine is an open game development platform offering an assortment of tools and features designed to enable game creators quickly and efficiently create high-quality games quickly and cost effectively. Popular among indie developers as well as large studios alike.

Unity Engine was initially released by Unity Technologies of San Francisco, California, in 2005 as an affordable and accessible game development platform designed for small indie developers without extensive budget or programming knowledge. Since its initial release, however, Unity Engine has undergone significant development resulting in its transformation into an all-inclusive game development solution.

Unity Engine's key advantage lies in its ease-of-use: developers can use its intuitive user interface to easily create and modify game assets and elements quickly and effortlessly. Furthermore, this engine offers robust graphics/animation/physics/audio management/scripting language support tools which enable creators to make high quality games featuring realistic physics, high quality animation and immersive audio - perfect for high quality games with realistic physics models and immersive sound experiences.

Unity Engine's cross-platform development capabilities make it an attractive platform to developers, supporting desktop computers, mobile devices and game consoles across a wide variety of operating systems such as Windows Mac Linux iOS Android PlayStation allowing them to design games and experiences that appeal to a broad audience. This flexibility enables Unity developers to design immersive experiences targeted specifically towards each audience segment they create games for.

Unity Engine has quickly become one of the go-to game development platforms

today, yet there are dozens of competing game engines on the market with features, capabilities and popularity that rival its own. Here we compare Unity with some other leading game engines.

Unreal Engine, created and distributed by Epic Games, is an efficient and flexible game engine utilized extensively in game development. Popular among game designers for creating high-quality games with realistic graphics and physics, Unreal Engine provides tools and features for virtual reality development such as support for AI systems as well as multiplayer game support; it may be more challenging for indie developers than Unity when learning its use however.

CryEngine: Created by Crytek, CryEngine is another advanced game engine known for its impressive graphics capabilities. Offering various tools and features to facilitate game development such as support for physics-based rendering and advanced lighting techniques. Similar to Unreal Engine, however, CryEngine may prove more difficult and complex for smaller game development teams to master than Unity - thus rendering its use less accessible for them.

Godot Engine: Godot Engine has seen immense popularity over recent years due to its ease-of-use and flexibility, offering developers tools for 2D/3D graphics development as well as physics simulation. Furthermore, cross-platform development capability makes this an attractive option; however it lacks some of the advanced features and capabilities found within Unity Engine or commercial game engines such as Unreal.

GameMaker Studio: GameMaker Studio stands out as an accessible game development platform due to its simple and user-friendly design, offering developers an accessible yet simple user interface for quickly developing games without needing extensive coding knowledge. Furthermore, it supports multiple platforms as well as offering robust tools and features for game creation (support for 2D/3D graphics/physics/audio support among them), although its effectiveness tends to fall behind that of Unity Engine or similar commercial game engines.

Unity Engine provides developers with access to multiple platforms and operating systems - desktop computers, mobile devices and game consoles alike - giving them maximum flexibility when designing games or interactive experiences that target a broad audience. This flexibility also extends to Microsoft's Windows OS as well as Linux distributions like Ubuntu/Mint for creating games on these systems.

4.2 Key Features of Unity Engine

It is stated in [Wat19] that unity has grown increasingly popular as a game engine in recent years, particularly for mobile and independent game development. Here we explore its key features which contributed to its widespread usage and success.

Cross-Platform Development: Unity stands out as an outstanding platform to use when developing games that run across different devices and operating systems, from PC and Mac computers, Android mobile phones and iOS phones all the way down to other handhelds like smartphones or game consoles. This makes the program useful in reaching a wider audience of players by supporting multiple devices simultaneously for playing their games on different devices and reaching further into different communities with each release of your game.

Unity offers an intuitive user-interface designed to allow developers to quickly craft games without needing extensive programming knowledge. It features a drag-and-drop system for quickly adding assets and components, with access to an extensive library of prebuilt assets for creating game environments or objects quickly and effortlessly.

Unity provides developers with access to multiple scripting languages such as C as talked in our book [Wat19], JavaScript and Boo, giving them the flexibility of selecting their preferred approach for themselves and their teams. Unity also incorporates Mono, an open-source implementation of Microsoft.NET framework providing an abundance of libraries and tools with which developers can leverage to craft complex game systems.

Unity's scripting system has been carefully created for ease of use and learning. The engine provides extensive documentation and tutorials, while its code editor features such as code highlighting, error checking and autocompletion make scripting faster and less prone to mistakes than ever.

Unity's Powerful Graphics: Unity is well known for its impressive graphics capabilities, such as support for advanced rendering techniques such as HDR, bloom and post-processing effects that enable developers to craft visually striking games with realistic lighting, shadows and reflections. These techniques make Unity stand out among its competition when developing visually impressive games with realistic lighting, shadows and reflections.

Unity not only supports advanced rendering capabilities but also offers 2D and 3D graphics support as explained in [Wat19]. When creating 2D games using Unity's 2D tools such as sprite editors, animation tools, physics engines or even 2D maps. Furthermore, 3D engines offered by Unity allow developers to easily create immersive environments or characters in 3D space.

Unity supports multiple platforms, including mobile devices, consoles and PC. This enables developers to design games that run seamlessly on multiple hardware while optimizing graphics settings specifically tailored for each platform.

Unity Asset Store: With Unity's extensive asset store, developers gain access to an abundance of pre-built assets, more about this can be found in [Wat19], tools, and resources designed specifically for game creation. This makes finding components

quickly easier while decreasing development time and costs significantly.

Unity boasts an active community of developers and users that provides resources, tutorials and support services for solving problems quickly, sharing expertise among peers and staying current on industry developments. This support system assists Unity developers with solving any difficulties they might be having quickly as well as staying informed with latest industry trends and developments.

4.3 Unity Editor

The Unity Editor is an all-encompassing development environment designed to assist developers with creating , managing and overseeing projects from start to finish. Equipped with numerous tools and features designed specifically to facilitate building, testing and deploying games across various platforms quickly and effortlessly, Unity's editor makes projects possible from its inception being more detailed in [Wat19].

Interface: Unity editor's user-friendly interface was designed to offer developers a clean and intuitive workspace to manage their projects efficiently. Comprised of several windows - Scene View, Game View, Hierarchy Window, Project Window and Inspector Window among them -, these can all be tailored according to individual developer's preferences or rearranged and customized as necessary for effective collaboration on development tasks.

Scene View: Developers use Scene view as their main workspace for level design and environment creation, offering them a 3D perspective of game environments to edit. Users are able to manipulate game objects, add terrain features, position lights/cameras as needed - making the Scene view essential in level designing/environment creation processes.

Developers using Scene view can create and manipulate game objects by dragging and dropping them onto their workspace, using tools like Move, Rotate and Scale tools for positioning, rotation and size adjustments of objects. They may also utilize tools such as Move Rotate Scale tools in order to adjust these attributes of their objects. Furthermore, developers using Scene view have access to creating terrain - including adding or removing features like hills and valleys from terrain features such as hills.

Game View: The Game view allows developers to test their games in real-time. It displays what it would look like for players and allows developers to test game mechanics, camera angles, and other aspects of design in an immersive simulation format. Various aspects can be customized through its display options: camera angle controls, overlays or debug information can all be shown simultaneously on screen in this environment.

Developers can interact with their game in Game view just like playing it - moving their player character, activating game events and testing various game mechanics are just a few activities available here. Game view serves an essential function when testing game mechanics as part of level design process and works hand in hand with Scene view for level designing purposes.

As previously discussed, Unity's editor provides two essential tools - Scene view and Game view - which serve to facilitate creating game environments by giving developers access to an extensive workspace for creating game environments while real-time previewing provides feedback to aid testing and refinement processes as depicted in [Wat19]. Together these tools facilitate game production quickly and efficiently for developers.

Hierarchy Window: The Hierarchy window displays a list of all game objects present in a scene along with their parent-child relationships for easy organization into hierarchies and hierarchies. This window makes managing game objects simpler while organizing them more logically into hierarchies.

Project Window: The Project window presents an inventory of all assets present within a project, such as 3D models, textures, sounds and scripts. Easily import or export as necessary with just one glance!

Inspector Window: The Inspector window displays properties for game objects or assets selected within an XNA game. Using this window, developers are able to modify positions, sizes and behaviors of game assets as necessary.

Unity offers an advanced scripting system which enables developers to add customized behaviors for game objects in Unity's editor. Unity offers support for numerous scripting languages including C, JavaScript and Boo for creating complex gaming systems.

Unity's editor features an advanced asset importing system, making it simple for developers to import 3D models, textures, and other assets into their projects from various sources. The importer automatically recognizes files into Unity's native format for conversion allowing you to utilize assets from numerous sources simultaneously.

Collaboration: mes Unity's editor also features tools designed to enable teams of developers and artists to easily work together on projects. These features include version control and project management tools which make tracking changes and sharing assets simpler than ever.

4.4 Asset Management in Unity

Asset management is an indispensable aspect of game development, involving the organization, storage and manipulation of game assets. Unity's Asset Management

system was built with this purpose in mind - offering developers tools for overseeing 3D models, textures, audio files and scripts in their game worlds.

At its heart, Unity's Asset Management system relies on its Project window as its central workspace for managing game assets. Divided into various sections for organizing assets into folders or previewing assets files previewing files search functionality allows Unity users to locate specific assets with ease.

Unity's Asset Management system makes asset import and export an effortless process. Developers can import assets in various file formats like OBJ, FBX and WAV before organizing them within folders within Project window for quick and efficient management. This feature ensures all game assets can be quickly accessible when needed by developers.

Unity's Asset Management system not only imports and organizes assets, but it also features powerful tools for manipulating these assets. Developers can utilize Mesh and Terrain editors for editing 3D models or terrain features respectively; as well as its scripting engine which allows custom scripting of asset manipulation in various ways.

Unity Asset Management includes an easy version control system which makes tracking changes to game assets over time much simpler, making collaboration on game development projects simpler than ever - as developers can track updates easily while easily going backwards if required. This also makes for faster revision cycles!

Overall, Unity's Asset Management system provides developers with an effective and thorough solution for overseeing game assets. Developers can import, organize, and manipulate assets with ease as well as benefiting from robust scripting features and version control features - helping streamline development efforts while providing more focus towards crafting engaging gameplay experiences for gamers.

4.5 Unity Physics

Unity's physics system, that is talked about in [Wat19], is an efficient and versatile solution for creating immersive, engaging gaming worlds. Built upon Box2D physics engine for 2D games, which has also been extended with 3D support by Unity to accommodate dynamic game worlds.

Unity's physics system is built around rigidbodies - objects in games with mass that respond to gravity or other physical forces; developers can attach rigidbodies to characters, vehicles or environmental objects so that they interact with one another in-game.

Unity's physics system features and capabilities offer users a diverse set of options for creating realistic, interactive game environments. Such features as collision

detection, joint and constraint systems enable complex physics-based interactions among objects in your game world to occur seamlessly and realistically.

Unity's physics system stands out with its ability to simulate realistic physical interactions between objects in real time, giving developers a powerful way to create dynamic and interactive gaming environments where objects behave realistically while engaging each other complex ways. Furthermore, soft body physics allow users to simulate deformable materials like cloth or hair realistically in their simulations.

Unity's physics system stands out in many ways, including its seamless integration with animation and scripting tools for game development. Developers can leverage it to create complex physical interactions in-game before using scripting to control them further. Furthermore, this system features support for physics-based animation for creating realistic character movements.

Chapter 5

Application

5.1 Algorithm Application Requirements

The application does not have a tutorial and the key features to our application might be easily overlooked, this chapter servers as a guide for what this research has come to and the future of it.

5.1.1 Playability

The application is not deployed, we are going to be using it inside the mentioned Unity Engine, in which it was made.By pressing on the play button, we can start the game, we will be dropped at the coordinates (0,0,0), which stand for (x,y,z), as seen in Figure 5.1.

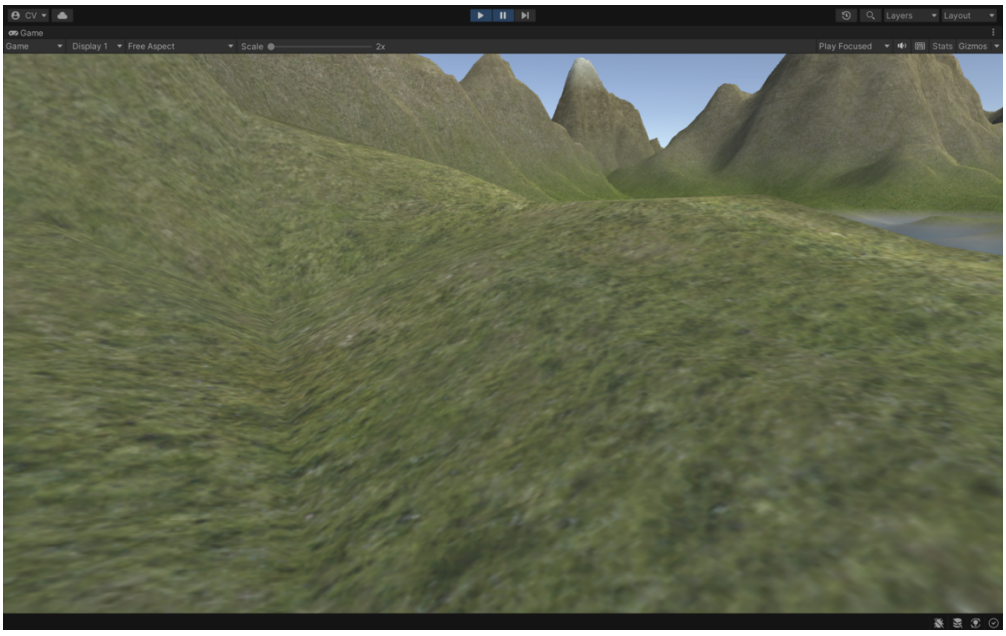

Figure 5.1: Game Screen

In Figure 5.2 the first thing that we see, is a big grass field, the game takes place in an endless generated world. We have different heights and with those heights we have different textures that have been assigned and made to blend together.The assigned areas are in height order:water,which is the lowest, grassland, gravel,stone at the start of the moutains and the peaks,these is the highest spot, of the moutains have been textured with white. Which we will be able to see in Figure 5.2.

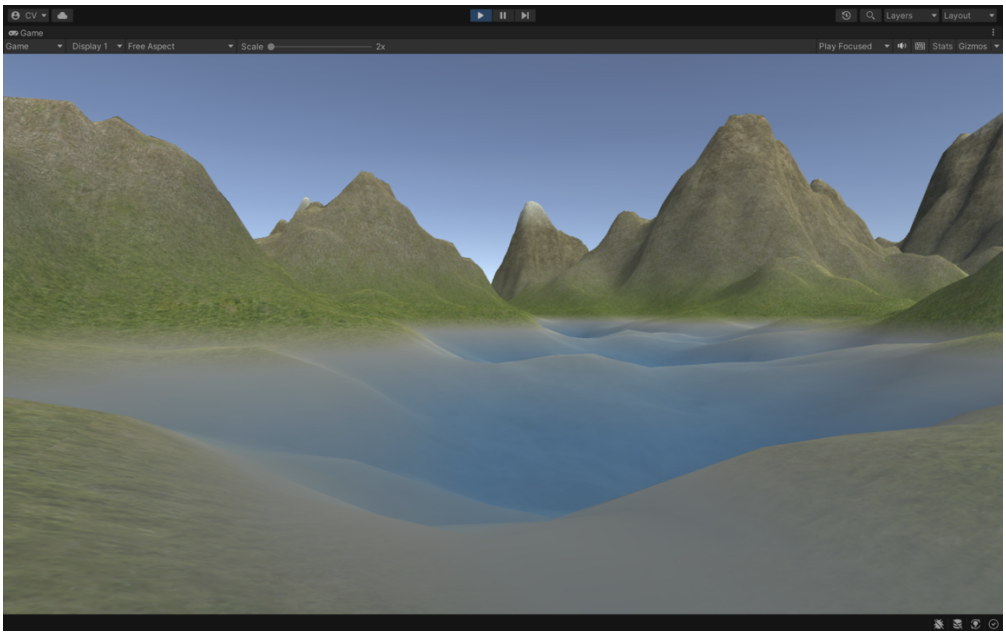

Figure 5.2: Game Screen

The second thing to take not of is how big the land is, this was achieved using individual chunks. By dividing the land into chunks I was able to create a big landmass by merging the chunks together.

The third most important feature is the generation itself and by this I mean that if I am at the coordinates mentioned above (0,0,0), I could go in any direction and have the game generate me a landmass, shown in Figure 5.3.

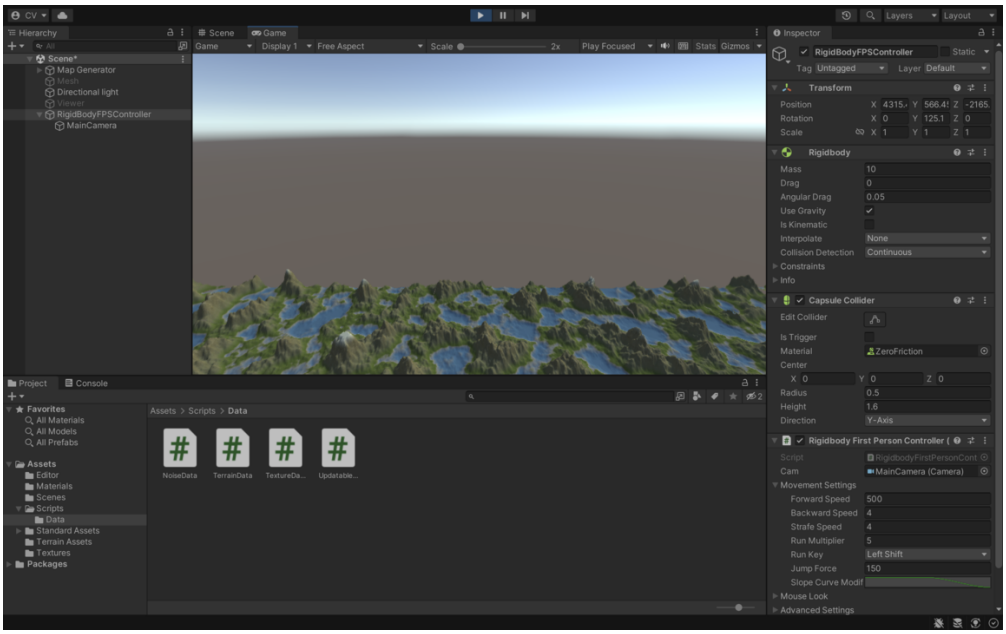

Figure 5.3: Game Screen

5.1.2 Posibilities

Right now the possibilities are not many as the bulk of the project was getting to the point in which you can create endless maps with different terrain generation using a simple seed.

What you can do in the project is the following:

Play with the inputs, that we are going to talk about in the next section. Doing this will change the terrain a lot and it might also create some inputs that are going to break the game.

Play on different seeds, as each seed is different you can test or look at as many seeds as possible.

Walk across the map with an asset generated character, which is just an invisible asset that helps with navigation on the map.

5.2 Algorithm Design

In this chapter we will be going through the code of the application. I have divided it in chapters and we are going to take the most important parts for each chapter and explain line by line after seeing a code snippet, shown in Figure 5.4.

5.2.1 Input Parameters

The input parameters for our code are the classes: NoiseData, TextureData, TerrainData.

NoiseData that extends from UpdatableData and is used to store parameters for generating noise in Unity.

```
public class NoiseData : UpdatableData {

    IL code
    public Noise.NormalizeMode normalizeMode;

    IL code
    public float noiseScale;

    IL code
    public int octaves;
    [Range(0,1)]
    IL code
    public float persistance;
    IL code
    public float lacunarity;

    IL code
    public int seed;
    IL code
    public Vector2 offset;

    #if UNITY_EDITOR

    protected override void OnValidate() {
        if (lacunarity < 1) {
            lacunarity = 1;
        }
        if (octaves < 0) {
            octaves = 0;
        }

        base.OnValidate ();
    }
    #endif
```

Figure 5.4: Code Screen NoiseData

The script has several public variables that control different aspects of the generated noise. The normalizeMode variable determines how the noise values are nor-

malized. The noiseScale variable controls the overall scale or frequency of the noise pattern. The octaves variable specifies the number of layers or octaves of noise to generate. The persistance variable determines the contribution of each octave to the final noise pattern, with a range between 0 and 1. The lacunarity variable controls the change in frequency between octaves.

Other variables include seed, which is the seed value used for the random number generator to produce consistent noise patterns, and offset, which allows you to shift the position of the noise pattern in the x and y directions.

In the OnValidate method, certain validations are performed. If the lacunarity value is less than 1, it is set to 1 to prevent unexpected results. Similarly, if the octaves value is negative, it is set to 0 to avoid errors.

The TextureData script is a class in Unity used to store and manage texture-related data for materials, with the visualisation help in Figure 5.5.

```
const int textureSize = 512;
const TextureFormat textureFormat = TextureFormat.RGB565;

IL code
public Layer[] layers;

IL code
float savedMinHeight;
IL code
float savedMaxHeight;
```

Figure 5.5: Code Screen TextureData

At the beginning of the script, we have the using statements, including UnityEngine and System.Collections. These are necessary for accessing Unity's API and using collections in C.

The script is defined as a class called TextureData, which extends from the UpdatableData class. This suggests that it can be updated and modified during runtime.

Inside the TextureData class, we find a few important elements. First, there are two constants: textureSize and textureFormat. These constants determine the size and format of the textures that will be generated.

Next, we have a public array called layers, which holds multiple Layer objects. Each Layer represents a specific texture layer and contains properties such as the texture itself, tint color, tint strength, start height, blend strength, and texture scale. This allows for customization and layering of textures in materials.

The TerrainData script serves as a data structure specifically designed to store and manage terrain-related properties within the Unity engine, below in Figure 5.6.

```
public float uniformScale = 2.5f;
public bool useFlatShading;
public bool useFalloff;

public float meshHeightMultiplier;
public AnimationCurve meshHeightCurve;

public float minHeight {
    get {
        return uniformScale * meshHeightMultiplier * meshHeightCurve.Evaluate (0);
    }
}

public float maxHeight {
    get {
        return uniformScale * meshHeightMultiplier * meshHeightCurve.Evaluate (1);
    }
}
```

Figure 5.6: Code Screen TerrainData

One important property is uniformScale, which represents the overall scale of the terrain. It is a floating-point value that acts as a scaling factor applied to the generated terrain. By adjusting this value, the terrain's size can be uniformly increased or decreased. The default value for uniformScale is set to 2.5f.

Another property, useFlatShading, is a boolean that determines whether the terrain should be rendered using flat shading. When enabled, the terrain appears with a flat appearance, whereas disabling this option produces a smooth shading effect, resulting in a more realistic appearance.

The useFalloff property is also a boolean, indicating whether a falloff effect should be applied to the terrain. The falloff effect helps create a natural transition between different terrain features, enhancing the overall visual quality of the generated terrain.

Two numerical properties, meshHeightMultiplier and meshHeightCurve, further influence the terrain's appearance. meshHeightMultiplier is a scaling factor that multiplies the height of the terrain mesh, affecting the vertical exaggeration of the terrain. On the other hand, meshHeightCurve is an AnimationCurve that defines the height curve for the terrain. By customizing this curve, the distribution of height values across the terrain can be adjusted, allowing for more diverse and

interesting terrain formations.

Additionally, the TerrainData script provides two read-only properties: minHeight and maxHeight. These properties return the minimum and maximum heights of the terrain, respectively. They are calculated based on the uniformScale, meshHeightMultiplier, and the evaluated values of the meshHeightCurve at the minimum (0) and maximum (1) points. These properties are valuable for setting appropriate height thresholds and for rendering and interacting with the terrain.

5.2.2 Algorithm Steps

Procedural games are a fascinating genre that utilizes algorithms to generate and shape various aspects of gameplay, including levels, characters, and environments. These algorithms provide a dynamic and ever-changing experience for players, making each playthrough unique and exciting. In this chapter, we will explore the algorithmic steps involved in creating a procedural game, delving into the key components and processes that drive the generation of game elements. By understanding these algorithm steps, game developers can harness the power of procedural generation to craft immersive and replayable gaming experiences.

5.2.3 Noise

The first one of our algorithms is the noise, this was explained in a previous chapter as a mathematical algorithm that produces random values based on certain parameters. I am going to show pictures and explain it step by step, starting with Figure 5.7.

```
float maxPossibleHeight = 0;
float amplitude = 1;
float frequency = 1;

for (int i = 0; i < octaves; i++) {
    float offsetX = prng.Next (-100000, 100000) + offset.x;
    float offsetY = prng.Next (-100000, 100000) - offset.y;
    octaveOffsets [i] = new Vector2 (offsetX, offsetY);

    maxPossibleHeight += amplitude;
    amplitude *= persistance;
}

if (scale <= 0) {
    scale = 0.0001f;
}

float maxLocalNoiseHeight = float.MinValue;
float minLocalNoiseHeight = float.MaxValue;

float halfWidth = mapWidth / 2f;
float halfHeight = mapHeight / 2f;
```

Figure 5.7: Code Screen Noise

Figure 5.7 code calculates the maximum possible height of a procedurally generated terrain in a game. It uses a loop to iterate through multiple layers or octaves of noise. Each iteration generates random offsets and updates the maximum height based on the current amplitude. The amplitude is reduced in each iteration using persistence. The code aims to create diverse and dynamic terrain by combining multiple octaves of noise.

```
for (int y = 0; y < mapHeight; y++) {
    for (int x = 0; x < mapWidth; x++) {

        amplitude = 1;
        frequency = 1;
        float noiseHeight = 0;

        for (int i = 0; i < octaves; i++) {
            float sampleX = (x-halfWidth + octaveOffsets[i].x) / scale * frequency;
            float sampleY = (y-halfHeight + octaveOffsets[i].y) / scale * frequency;

            float perlinValue = Mathf.PerlinNoise (sampleX, sampleY) * 2 - 1;
            noiseHeight += perlinValue * amplitude;

            amplitude *= persistance;
            frequency *= lacunarity;
        }

        if (noiseHeight > maxLocalNoiseHeight) {
            maxLocalNoiseHeight = noiseHeight;
        } else if (noiseHeight < minLocalNoiseHeight) {
            minLocalNoiseHeight = noiseHeight;
        }
        noiseMap [x, y] = noiseHeight;
    }
}
```

local variable Single **noiseHeight**

Figure 5.8: Code Screen Noise

Figure 5.8 utilizes nested loops to iterate over each position on the map. Within the loops, multiple layers of noise are generated using Perlin noise. The resulting noise values are combined to calculate the final height for each map position. The code also tracks the maximum and minimum height values encountered during the generation process. The resulting noiseMap represents the height values for the entire terrain.

```
for (int y = 0; y < mapHeight; y++) {
    for (int x = 0; x < mapWidth; x++) {
        if (normalizeMode == NormalizeMode.Local) {
            noiseMap [x, y] = Mathf.InverseLerp (minLocalNoiseHeight, maxLocalNoiseHeight, noise
        } else {
            float normalizedHeight = (noiseMap [x, y] + 1) / (maxPossibleHeight/0.9f);
            noiseMap [x, y] = Mathf.Clamp(normalizedHeight,0, int.MaxValue);
        }
    }
}

return noiseMap;
```

Figure 5.9: Code Screen Noise

We normalise in Figure 5.9 the generated noise map by iterating through each position on the map. Depending on the specified normalization mode, it applies different normalization techniques to ensure the values fall within a desired range. The resulting normalized noise map is then returned.

5.2.4 EndlessGeneration

The concept of endless generation in game development refers to the ability to generate content dynamically and continuously without predetermined limitations. In this subsection, we explore the principles and techniques behind creating endless game worlds, levels, or environments that offer infinite or near-infinite possibilities for players to explore and engage with. By leveraging procedural algorithms, randomization, and clever design choices, developers can achieve seamless and ever-evolving game experiences that keep players engaged and immersed for extended periods. Let's delve into the fascinating world of endless generation and discover the key strategies used to create limitless gaming adventures. The explanation will be done the same as for the noise.

```
void Start() {
    mapGenerator = FindObjectOfType<MapGenerator> ();

    maxViewDst = detailLevels [detailLevels.Length - 1].visibleDstThreshold;
    chunkSize = mapGenerator.mapChunkSize - 1;
    chunksVisibleInViewDst = Mathf.RoundToInt(maxViewDst / chunkSize);

    UpdateVisibleChunks ();
}
```

Figure 5.10: Code Screen Start

The Start function, displayed in Figure 5.10, initializes variables and performs necessary calculations for the script. It finds the MapGenerator component, determines the maximum view distance, calculates the chunk size, and updates the visible chunks accordingly.

```
void Update() {
    viewerPosition = new Vector2 (viewer.position.x, viewer.position.z) / mapGenerator.terrainData

    if (viewerPosition != viewerPositionOld) {
        foreach (TerrainChunk chunk in visibleTerrainChunks) {
            chunk.UpdateCollisionMesh ();
        }
    }

    if ((viewerPositionOld - viewerPosition).sqrMagnitude > sqrViewerMoveThresholdForChunkUpdate)
        viewerPositionOld = viewerPosition;
        UpdateVisibleChunks ();
    }
}
```

Figure 5.11: Code Screen Update

In the Update function, Figure 5.11, the viewer's position is tracked and compared with the previous position. If there is a change in position, the collision mesh for each visible terrain chunk is updated. Additionally, if the viewer's movement exceeds a threshold, the visible chunks are updated.

```
d UpdateVisibleChunks() {
HashSet<Vector2> alreadyUpdatedChunkCoords = new HashSet<Vector2> ();
for (int i = visibleTerrainChunks.Count-1; i >= 0; i--) {
    alreadyUpdatedChunkCoords.Add (visibleTerrainChunks [i].coord);
    visibleTerrainChunks [i].UpdateTerrainChunk ();
}

int currentChunkCoordX = Mathf.RoundToInt (viewerPosition.x / chunkSize);
int currentChunkCoordY = Mathf.RoundToInt (viewerPosition.y / chunkSize);

for (int yOffset = -chunksVisibleInViewDst; yOffset <= chunksVisibleInViewDst; yOffset++) {
    for (int xOffset = -chunksVisibleInViewDst; xOffset <= chunksVisibleInViewDst; xOffset++) {
        Vector2 viewedChunkCoord = new Vector2 (currentChunkCoordX + xOffset, currentChunkCoordY
        if (!alreadyUpdatedChunkCoords.Contains (viewedChunkCoord)) {
            if (terrainChunkDictionary.ContainsKey (viewedChunkCoord)) {
                terrainChunkDictionary [viewedChunkCoord].UpdateTerrainChunk ();
            } else {
                terrainChunkDictionary.Add (viewedChunkCoord, new TerrainChunk (viewedChunkCoord,
            }
        }

    }
}
```

Figure 5.12: Code Screen UpdateVisibleChunks

The UpdateVisibleChunks, in Figure 5.13, function is responsible for dynamically managing the visibility and updates of terrain chunks based on the viewer's position. It iterates through the visible chunks, ensuring they are up-to-date by calling the UpdateTerrainChunk function. It also checks for additional chunks within the visible range and updates or creates them as needed. This ensures that only the necessary chunks are rendered and updated, optimizing the rendering process and providing an immersive and responsive terrain experience for the viewer.

```
public TerrainChunk(Vector2 coord, int size, LODInfo[] detailLevels, int colliderLODIndex, Transfo
    this.coord = coord;
    this.detailLevels = detailLevels;
    this.colliderLODIndex = colliderLODIndex;

    position = coord * size;
    bounds = new Bounds(position,Vector2.one * size);
    Vector3 positionV3 = new Vector3(position.x,0,position.y);

    meshObject = new GameObject("Terrain Chunk");
    meshRenderer = meshObject.AddComponent<MeshRenderer>();
    meshFilter = meshObject.AddComponent<MeshFilter>();
    meshCollider = meshObject.AddComponent<MeshCollider>();
    meshRenderer.material = material;

    meshObject.transform.position = positionV3 * mapGenerator.terrainData.uniformScale;
    meshObject.transform.parent = parent;
    meshObject.transform.localScale = Vector3.one * mapGenerator.terrainData.uniformScale;
    SetVisible(false);

    lodMeshes = new LODMesh[detailLevels.Length];
    for (int i = 0; i < detailLevels.Length; i++) {
        lodMeshes[i] = new LODMesh(detailLevels[i].lod);
        lodMeshes[i].updateCallback += UpdateTerrainChunk;
        if (i == colliderLODIndex) {
            lodMeshes[i].updateCallback += UpdateCollisionMesh;
        }
    }

    mapGenerator.RequestMapData(position,OnMapDataReceived);
}
```

Figure 5.13: Code Screen TerrainChunk

The TerrainChunk, shown above in Figure 5.13, constructor initializes a terrain chunk object with the given parameters. It sets the chunk's coordinates, detail levels, collider LOD index, position, bounds, and creates the necessary game objects and components such as the mesh renderer, mesh filter, mesh collider, and LOD meshes. Additionally, it sets the position, parent, and scale of the chunk's game object and requests map data from the mapGenerator.

For the most part these are the most important bits of code that create our endless maps.

Chapter 6

Conclusions

In conclusion, this thesis has explored the domain of automatic procedural generation for 3D maps, presenting a range of techniques and algorithms that enable the efficient and realistic generation of maps. Through extensive research and experimentation, various aspects of procedural generation, including terrain generation, object placement, and texture mapping, have been examined and evaluated. The results have demonstrated the potential of automatic procedural generation in creating visually appealing and interactive 3D maps. Furthermore, the integration of user customization and interaction has been investigated, providing a more immersive and personalized experience for map exploration. Despite the progress made, challenges such as balancing performance and realism, maintaining coherence and variety, and ensuring user satisfaction remain. Overall, this thesis contributes to the field by providing valuable insights, performance evaluations, and recommendations for future research, paving the way for further advancements in automatic procedural generation for 3D maps.

Bibliography

[KTY14] Ahmed Khalifa, Julian Togelius, and Georgios N. Yannakakis. A comparison of procedural level generation techniques in video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):381–395, 2014.

[STN16a] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[STN16b] Noor Shaker, Julian Togelius, and Mark J. Nelson. *The Procedural Generation of Meaningful Content for Games*. Springer, 2016.

[TXS17] Tarn Adams Tanya X. Short. *Procedural Generation in Game Design*. A. K. Peters, Ltd., 63 South Avenue Natick, MAUnited States, 2017.

[Wat19] Ryan Watkins. *Procedural Generation in Unity: Game Development with C*. Apress, 2019.